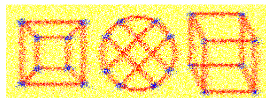


Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 21, no. 4, pp. 589–630 (2017)
DOI: 10.7155/jgaa.00431

Topological Decomposition of Directed Graphs

Ala Abuthawabeh Dirk Zeckzer[1]

[1]Leipzig University, Germany

Abstract

The analysis of directed graphs is important in application areas like software engineering, bioinformatics, or project management. Distinguishing between topological structures such as cyclic and hierarchical subgraphs provides the analyst with important information. However, until now, graph drawing algorithms draw the complete directed graph either hierarchically or cyclic. Therefore, we introduced new algorithms for decomposing the input graph into cyclic subgraphs, directed acyclic subgraphs, and tree subgraphs. For all of these subgraphs, optimized layout algorithms exist. We developed and presented a new algorithm for drawing the complete graph based on the decomposition using and combining these layouts. In this paper, we focus on the algorithms for the topological decomposition. We describe them on an intermediate level complementing the previous descriptions on the high and the low level. Besides the motivation, illustrative examples of all cases that need to be considered by the algorithm, standard as well as more complex ones, are given. We complement this description by a complexity analysis of all algorithms.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
December 2016	January 2017	February 2017	April 2017	April 2017
		Published: April 2017		
	Article type: Regular paper		Communicated by: G. Liotta	

E-mail addresses:
zeckzer@informatik.uni-leipzig.de (Dirk Zeckzer)

1 Introduction

Directed graphs are used for displaying relations between entities where the *direction* of the relation is important. An example from software engineering are call graphs. If method or function A calls method or function B, then the reverse relation—B calls A—does not necessarily hold. Besides software engineering there are several other application areas and applications where directed graphs are regularly used, like metabolic networks in biology and bioinformatics, or PERT charts for project management [8]. In the handbook of graph drawing [8], directed graphs are discussed in Chapter 13, "Hierarchical Drawing Algorithms".

Frequently, the Sugiyama algorithm [7] is used for drawing directed graphs, even though its main purpose is the drawing of directed *acyclic* graphs. Only recently, Bachmaier et al. [3, 4] proposed a cyclic layout to draw directed graphs containing cycles. These publications address the complete layout and the coordinate assignment phase and use the leveling phase described before [5]. The disadvantages of both layouts become obvious, when analyzing directed graphs. In a Sugiyama layout, cycles are difficult to detect, while in the cyclic layout, non-cyclic parts are not obvious. On the other hand, distinguishing between cyclic and non-cyclic sub-graphs of a directed graph are important in the applications areas. For example, cyclic dependencies of classes in an object-oriented model may reduce modularity.

Decomposing directed graphs into cyclic and non-cyclic sub-graphs as well as drawing these subgraphs such that the respective structure is easily recognizable enables a more efficient and effective analysis of the relations represented by the graphs. Hence, Abuthawabeh and Zeckzer [1, 2] presented their topological layout approach for directed graphs. First, the graph is decomposed into non-trivial cyclic subgraphs, trees, and DAGs. Then, each of the components is drawn using the most adequate layout: Bachmaier's algorithm for non-trivial cyclic subgraphs [3, 4] the Sugiyama layout for DAGs [7], and the tree layout proposed by Walker [9] and improved by Buchheim et al. [6].

Previously, the topological approach was described on a high level [2] and on a low level [1]. In this paper, it will be described on an intermediate level focusing on the topological decomposition only. For the topological decomposition much more details and examples are provided compared to the high level description [2], while the implementation details of the low level description [1] were omitted for clarity.

2 Motivation and Definitions

Let $G = (V, E)$ be a *directed graph* where V denotes a set of *vertices* and E denotes a set of *directed edges*. Until recently [8], the standard way of drawing such a graph was using the Sugiyama algorithm [7]. However, this algorithm was intended to be applied to *acyclic* directed graphs only [7]. Nevertheless, it was also used for cyclic directed graphs by first reversing enough edges to

make the graph acyclic, second layouting the acyclic graph, and finally, putting the original edges instead of the reversed ones thus obtaining the final drawing. While this keeps the layered part of the graphs, cycles are difficult to spot. We call this the *hierarchical approach*.

Recently, Bachmaier et al. [3, 4] proposed a cyclic layout for directed graphs. In this case, all cycles are clearly depicted and can readily be analyzed. However, also all layered, acyclic parts are drawn in a cyclic way, making it difficult to spot them. We call this the *cyclic approach*.

Therefore, Abuthawabeh and Zeckzer [1, 2] presented their topological layout approach for directed graph that decomposes the graph into non-trivial cyclic subgraphs (ntCS), directed acyclic graphs (DAGs), and trees. Then, each of the components is drawn using the best currently available algorithm: Bachmaier's algorithm for non-trivial cyclic subgraphs [3, 4], Sugiyama's algorithm for DAGs [7], and the improved Walker's algorithm for trees [6]. We call this the *topological approach*, because each subgraph type has a certain topology.

The goal of this paper is to give more details on the decomposition of directed graphs than in our previous publication [2]. One of the most important concepts of our approach is the distinction between *trivial cycle* and *non-trivial cycle*. In Figure 1(a), two strongly-connected components G1 and G2 are shown. In G1, there are two cycles between two nodes, each. G1 does not really need to be drawn in a cyclic way, as the two-node cycle will be clear in both approaches, hierarchical and cyclic. G2, however, contains as well two-node cycles as three-node cycles. In this case, the cycle can be detected best if the graph is drawn using the cyclic approach. This leads to the following definitions.

Definition 2.1 (Trivial Cycle) *A* trivial cycle *is a set of edges* $\{(a,b),(b,a)\}$*,* $a,b \in V$ *and* (a,b)*,* $(b,a) \in E$ *that form a cycle of two nodes. We say, the trivial cycle* contains *the nodes* a,b*.*

Definition 2.2 (Back Edge) *Let* $\{(a,b),(b,a)\}$ *be a trivial cycle. Then,* (b,a) *is the* back edge *of* (a,b) *and vice versa.*

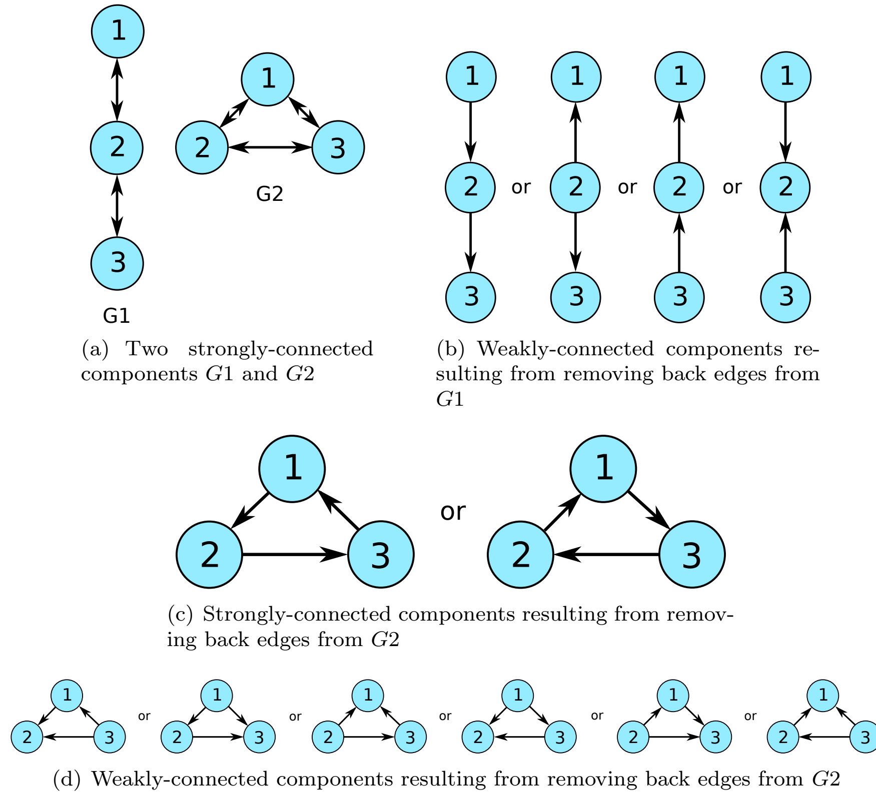
Definition 2.3 (Double Edge) *The set of edges* $\{(a,b),(b,a)\}$*,* $a,b \in V$ *and* $(a,b),(b,a) \in E$ *is called* double edge*.*

Corollary 2.4 *Each double edge gives rise to a trivial cycle.*

Definition 2.5 (Non-Trivial Cycle) *A* non-trivial cycle *is a cycle that is not trivial.*

Corollary 2.6 *A non-trivial cycle consists of at least three edges and contains at least three nodes.*

With these definitions, we can now define *non-trivial cyclic subgraphs (ntCS)*. Starting from G1 and G2, we can extract all subgraphs that do not contain trivial cycles. This can be achieved by removing one of the edges of each trivial cycle. In all these cases, we consider the removed edge to be the back edge.



(a) Two strongly-connected components $G1$ and $G2$

(b) Weakly-connected components resulting from removing back edges from $G1$

(c) Strongly-connected components resulting from removing back edges from $G2$

(d) Weakly-connected components resulting from removing back edges from $G2$

Figure 1: Two strongly connected components $G1$ and $G2$ and the resulting graphs when removing back edges from them.

Doing so, we find that for G1 this results in weakly-connected components only. Figure 1(b) shows all extractable subgraphs of G1.

Doing so for G2, however, results in either strongly-connected components (Figure 1(c)) or weakly-connected components (Figure 1(d)). By construction, none of the subgraphs of G2 contains trivial cycles. This motivates the following definition.

Definition 2.7 (Non-Trivial Cyclic Subgraph) *A* non-trivial cyclic subgraph *(ntCS) is a strongly-connected component $G = (V, E)$ that contains at least one strongly-connected component $G' = (V', E')$, $V' = V$, and $E' \subseteq E$ without trivial cycles.*

Remark 2.8 *In Definition 2.7, the strongly connected component G might contain several different strongly-connected components without trivial cycles.*

Remark 2.9 *In Definition 2.7, E' does not contain back edges.*

In the remainder of this paper, we will show how to decompose a directed graph into ntCSs, DAGs, and trees by removing back edges and thereby trivial cycles. For this, we extend the conventional definitions of tree and DAG.

Definition 2.10 (Trivial Down-Tree) *A* trivial down-tree *is a tree that does not contain trivial cycles and whose root node has in-degree zero.*

Definition 2.11 (Trivial Up-Tree) *A* trivial up-tree *is a tree that does not contain trivial cycles and whose root node has out-degree zero.*

Definition 2.12 (Trivial DAG) *A* trivial DAG *is a DAG that does not contain trivial cycles.*

Definition 2.13 (Non-Trivial Down-Tree) *A* non-trivial down-tree *is a wCC that contains trivial cycles and that can be transformed into a trivial down-tree by removing one edge of each double edge.*

Definition 2.14 (Non-Trivial Up-Tree) *A* non-trivial up-tree *is a wCC that contains trivial cycles and that can be transformed into a trivial up-tree by removing one edge of each double edge.*

Definition 2.15 (Non-Trivial DAG) *A* non-trivial DAG *is a wCC that contains trivial cycles and that can be transformed into a DAG by removing one edge of each double edge.*

Definition 2.16 (Down-Tree) *A* down-tree *is either a trivial or a non-trivial down-tree.*

Definition 2.17 (Up-Tree) *An* up-tree *is either a trivial or a non-trivial up-tree.*

Definition 2.18 (DAG) *A* DAG *is either a trivial or a non-trivial DAG.*

Remark 2.19 *Each (trivial, non-trivial) tree is a (trivial, non-trivial) DAG.*

Remark 2.20 *A wCC without ntCS that contains trivial cycles might be a non-trivial DAG, a non-trivial down-tree, and a non-trivial up-tree at the same time, depending on which back-edges are removed.*

3 Decomposing Directed Graphs

In our approach, directed graphs are decomposed as follows. Given a directed graph G, first all its weakly connected components (wCCs) are extracted. Each of these wCCs is then decomposed into ntCSs, DAGs, and trees. Figure 2 shows an overview of this decomposition [1]. It starts by detecting ntCSs in each wCC (Section 4). After removing the edges of all ntCSs found in the wCCs, it splits the remaining wCCs at the ntCSs into smaller wCCs (Section 5). Finally, the

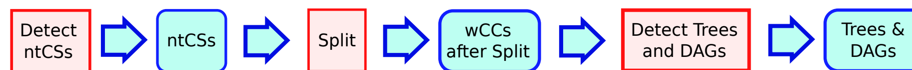


Figure 2: Decomposition Process of a wCC into ntCS, DAGs, and trees [1, 2].

resulting wCCs are classified as down-trees, up-trees, and DAGs (Section 6). Please notice, that these trees and DAGs might contain trivial cycles [1, 2].

In this paper, we focus on the special cases that need to be considered. Compared to our previous paper [2], much more detail is provided. On the other hand, the implementation details are described by Abuthawabeh in his PhD thesis [1].

4 Detecting Non-Trivial Cyclic Subgraphs

4.1 Overview

The first step of the topological decomposition process of a directed graph is detecting all non-trivial cyclic subgraphs (ntCSs). The detection of all ntCSs follows all edges. No edge will be revisited twice during this search.

This step can logically be decomposed into two sub-steps. First, a wCC component is randomly selected. The first sub-step recursively constructs paths through this wCC starting at a node and applying depth first search (Section 4.2). The second sub-step is responsible for checking if a part of this path is a (partial) ntCS (Section 4.3). If a ntCS is found, it is added to the set of ntCSs. If necessary, ntCSs are merged. After handling all nodes and edges of the current wCC, the next untreated wCC is randomly selected and the two sub-steps are repeated. After all wCCs have been checked for ntCSs, this step is finished.

Figures 3–9 show examples illustrating in detail all potential situations that might occur during ntCSs detection. They will be described in the respective parts of the ntCS detection algorithm. All relevant algorithms are listed in Appendix A, Algorithms 1–5.

4.2 Constructing Paths and Handling Back Edges

To find all ntCSs of a wCC ($Find_ntCS_in_wCC$, Algorithm 1), a *node* is randomly selected from the wCC and a path ($pathNodes$) through the wCC is created following outgoing edges only using depth first search ($Find_ntCS$, Algorithm 2 and $Find_ntCS_Rec$, Algorithm 3). While creating the path, it is checked if a part of the path forms a ntCS. All ntCS found during path creation are stored in a list ($ntCSs$). All nodes and edges found during path creation are marked. If there are unvisited nodes and edges of the wCC after the path creation for the current starting node (backtracking), a new path is created starting at a randomly selected unmarked node and the search is repeated.

The ntCS detection algorithm *Find_ntCS* (Algorithm 2) performs the following steps:

1. Increment the *incomingEdgeCounter* of the last node of the path stored in *pathNodes* (line 1)
2. If the last node of the path was already visited, check if this creates a ntCS (lines 2–3).
3. Otherwise, mark the last node of the path as visited (line 4–5).
4. Follow all outgoing edges of the last node of the path (line 7–30).

While following all outgoing edges, *Find_ntCS_Rec* is called. It takes care of the recursive call of *Find_ntCS* (line 4) storing the outgoing edge in the edges path (line 3) and the target node of the outgoing edge in the nodes path (line 2). Both will be deleted from their respective paths directly after invoking *Find_ntCS* (backtracking, lines 5–6). Moreover, the outgoing edge is marked as being visited (line 1). This flag will not be removed. As this code is needed two times by *Find_ntCS* (lines 16 and 26), calling *Find_ntCS_Rec* instead avoids duplicated code.

The first step of *Find_ntCS*, incrementing the counter of incoming edges, is needed by its fourth step and explained there. For the first node of the path, this step is not needed. However, this case is not checked as it has no influence on the algorithm.

Marking the last node of the path (*lastNode*, step 3) is needed for two reasons. First of all, an already marked node will be checked for finding ntCSs (step 2). Second, already marked nodes will not be considered as starting points for further paths by *Find_ntCS*.

In the following, steps 2 and 4 of the algorithm will be explained by describing two cases: (1) wCCs without trivial cycles (Figure 3) and (2) wCCs with trivial cycles (Figure 4). Only relevant steps will be described.

The example in Figure 3 shows the standard case of detecting a single ntCS in a wCC without trivial cycles (no back edges). Starting at node 0, a path of nodes containing node 0 and an empty path of edges are created. *Find_ntCS* marks node 0 as visited (line 5), traverses the outgoing edge $0 \rightarrow 1$ (lines 8, 9, 14–16), adds node 1 and edge $0 \rightarrow 1$ to the corresponding data structures (*Find_ntCS_Rec*), and marks the edge as visited (*Find_ntCS_Rec*). Then, it marks node 1 as being visited, follows edge $1 \rightarrow 2$, marks node 2 as being visited, and follows edge $2 \rightarrow 0$, updating the respective data structures. Now, the nodes path is $(0, 1, 2, 0)$, the edges path contains the edges $\{e1, e2, e3\}$, and $lastNode = 0$. Further, all nodes are marked as being visited. The next call to *Find_ntCS* finds that *lastNode* is marked visited (line 2) and therefore (line 3) calls *Check_ntCS* (Algorithm 4, Section 4.3). *Check_ntCS* detects the ntCS C1 having nodes $(0, 1, 2)$ and adds C1 to *ntCSs*, and *Find_ntCS* ends as no more unvisited outgoing edges are available.

If the wCC contains trivial cycles (back edges), the situation becomes more complex. Considering the wCC in Figure 4(a), there are two possibilities of

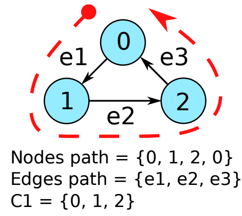
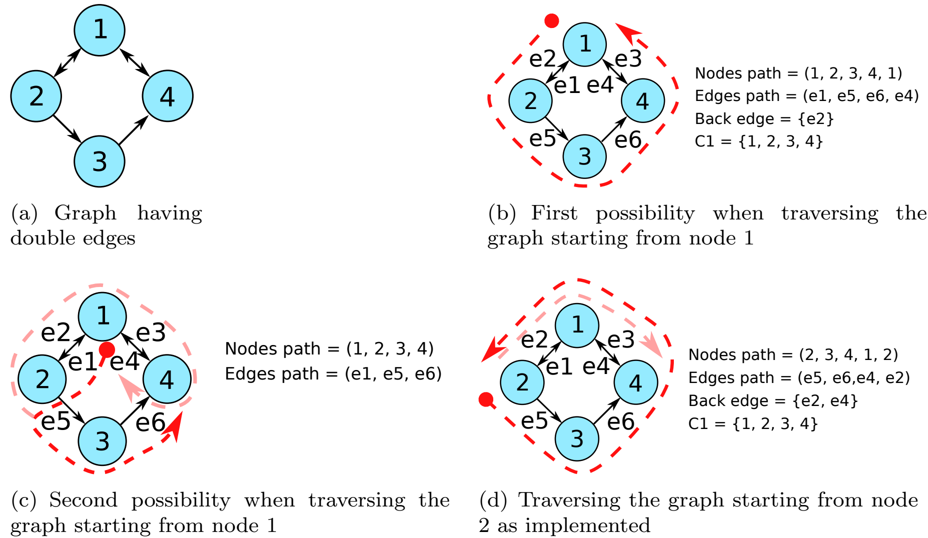


Figure 3: Single ntCS



(a) Graph having double edges

(b) First possibility when traversing the graph starting from node 1

(c) Second possibility when traversing the graph starting from node 1

(d) Traversing the graph starting from node 2 as implemented

Figure 4: ntCSs that are only reliably detected by handling back edges separately

traversing the wCC when starting from node 1 if back edges are not handled separately. The first possibility of traversing the wCC is shown in Figure 4(b). The algorithm starts by following the unvisited outgoing edge $1 \rightarrow 2$. Then, it continues by following the unvisited outgoing edges $2 \rightarrow 3$, $3 \rightarrow 4$, and $4 \rightarrow 1$ yielding the path $(1, 2, 3, 4, 1)$. Because node 1 was visited previously by the current path, the ntCS C1 is detected having nodes $\{1, 2, 3, 4\}$. The second possibility of traversing the wCC is shown in Figure 4(c). As before, the algorithm starts by following the unvisited outgoing edge $1 \rightarrow 2$. However, now, it continues by following the unvisited outgoing edges $2 \rightarrow 1$ (back edge of $1 \rightarrow 2$), $1 \rightarrow 4$, and $4 \rightarrow 1$ (back edge of $1 \rightarrow 4$) yielding the path $(1, 2, 1, 4, 1)$. The algorithm will stop when visiting node 1 again because there are no unvisited edges left. No ntCS was detected (the final path contains only cycles with back edges). After backtracking, the path is $(1, 2)$ and the unvisited outgoing edges $2 \rightarrow 3$ and $3 \rightarrow 4$ are followed resulting in the final path $(1, 2, 3, 4)$. Because edge $4 \rightarrow 1$ was already visited, the algorithm will not follow it and stops because there are no

unvisited outgoing edges. Again, no ntCS was found during path construction. Overall, the algorithm could not construct a ntCS without back edges in this case and the wCC was not classified as ntCS even though, a subgraph containing a ntCS without back edges can be constructed. To avoid this situation, back edges are not followed immediately by the algorithm. Instead, they are stored (lines 10–13) and all stored back edges are followed (lines 20—30), if there are no more unvisited incoming edges (line 20) and no other outgoing edges of the last node of the path (handled before, lines 8–19). Thus, after following edge $1 \rightarrow 2$, the implemented algorithm will always follow edge $2 \rightarrow 3$ and always store edge $2 \rightarrow 1$ when starting at node 1, regardless, which edge is handled first in the for loop. Only after all incoming edges of this node are marked, the stored back edges are followed. Instead of checking all incoming edges for being marked, a counter of incoming edges is used for efficiency reasons (lines 1, 20). All incoming edges being marked is equivalent to the counter of incoming edges being larger than or equal to the number of incoming edges as each incoming edge is exactly used once (all edges are visited by the algorithm).

The back edges need to be followed to find all ntCSs as the next example shows. Considering the same wCC as before, let the algorithm start at node 2 (Figure 4(d)). Further, let it follow the edges $2 \rightarrow 1$ and $1 \rightarrow 4$ and storing the edges $4 \rightarrow 1$ and $1 \rightarrow 2$ as back edges (both nodes 1 and 4 have unvisited incoming edges). After backtracking to node 2, it follows edges $2 \rightarrow 3$ and $3 \rightarrow 4$. As there are no more unvisited outgoing edges and no more unvisited incoming edges of node 4, it retrieves and follows the stored back edge $4 \rightarrow 1$ at node 4. The same situation occurs at node 1, and back edge $1 \rightarrow 2$ is retrieved and followed at node 1 yielding the path $(2, 3, 4, 1, 2)$. ntCS C1 is detected as node 2 was visited before by the current path.

All other possible cases for this wCC are similar to those explained above. Therefore, the wCC will always be classified as ntCS.

4.3 Checking a Path for being a (Partial) ntCS

If *Find_ntCS* finds a node that was already marked, it calls *Check_ntCS*. *Check_ntCS* (Algorithm 4) examines if a sequence of nodes in this path can form (a part of) a ntCS. First, some variables are initialized: the size of the path (line 1), the last node of the path (line 2), and the last and the second to last position of the last node in the path (lines 3–4). Second, it is checked if the last node and the second to last node belong to the same ntCS (lines 5–9). If so, no new ntCS has been found and the algorithms terminates returning "no ntCS found".

To identify the sequence of nodes potentially forming a new ntCS, the algorithm starts searching over all nodes in the path starting from the last node (the already traversed node) in reverse order and stopping when it finds the same node again (lines 10–15). Starting at the end and in reverse order is necessary, because due to the construction of the path, the last node could be multiple times in the path. This is a consequence of the handling of back edges and the potential existence of trivial cycles in the path.

Now, four cases might occur: (1) a trivial cycle was found (lines 16–18), (2) a ntCS was found (lines 19–27), (3) a partial ntCS was found (lines 28–29), or (4) no ntCS was found (line 31). A trivial cycle occurs when an edge is directly followed by its back edge, e.g., if a node is connected to the current node only by these two edges (Case 1). This case is handled by lines 16–18 of the algorithm and the algorithm terminates with "no ntCS found".

Next, it is checked if the last node occurs a second time in the path (line 19, Case 2). If yes, then the algorithm invokes *ComputeReducedPath* (Algorithm 5) to exclude double edges that are only parts of trivial cycles (line 20). If the reduced path is not empty (line 21), the algorithm constructs a new ntCS based on the nodes in the reduced path (lines 22–23, Section 4.3.1). If the ntCS found is not a part of a previously found ntCS, it is added to the list of all ntCSs (line 24, *SubCycle* [1]). Finally, all ntCSs found until now are combined, if they have shared nodes or edges (line 25, *MergeCycles* [1]) and the algorithm returns "ntCS found". Two examples illustrating this part of the algorithm will be given below (Section 4.3.2).

If the last node in the path occurs only once in the path, it is checked whether the last node already belongs to a ntCS (line 28, Case 3). If it does, the algorithm checks if the sequence of nodes in the path is part of a larger ntCS that can be constructed reusing the nodes of an existing ntCS (line 29, *PartialCycle* [1]). Two examples illustrating this situation are given below (Section 4.3.3).

Otherwise, no (part of a) ntCS could be found and the algorithm returns "no ntCS found" (line 31, Case 4).

4.3.1 Computing the Reduced Path

The algorithm *ComputeReducedPath* (Algorithm 5) determines a reduced path that could be part of a ntCS. Therefore, it removes connections between subgraphs that consist of double edges only from the input path. An example is shown in Figure 5 where two ntCSs are connected by the double edge $(e3, e8)$. First (lines 1–5), a copy of the input path is created. If the double edge is formed by the first and the last edge of the reduced path, then an empty path is returned (lines 6–7). In the case, that another part of the input path already forms a ntCS, this ntCS will be detected by the Algorithms 2–4. Otherwise, all edges of the reduced path starting from the first are used to construct test edges, which are potential back edges (lines 8–21). If a back edge is found, all nodes between these two edges are removed from the path (lines 15–17). Moreover, the second occurrence of the start node of the edge is removed from the path (lines 18–20).

In the example shown in Figure 5, the input path is (2, 1, 0, 4, 5, 6, 7, 4, 0, 3, 2). Edge $e3$ $(0 \to 4)$ leads to the creation of test edge $(4 \to 0)$. As this edge is a back edge, namely $e8$, nodes $4, 5, 6, 7$ are removed from the reduced path resulting in $(2, 1, 0, 0, 3, 2)$. Now, the second occurrence of node 0 is removed, too, and the resulting reduced path is $(2, 1, 0, 3, 2)$ yielding ntCS C2.

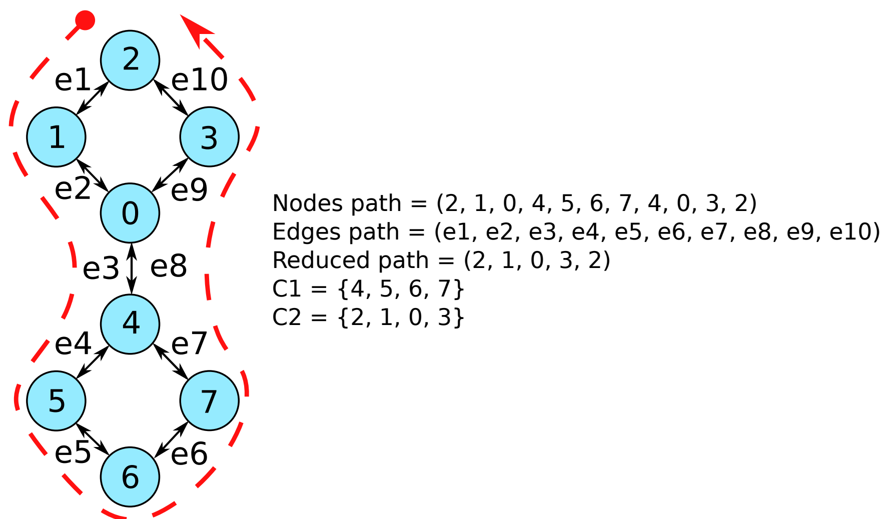


Figure 5: Two ntCSs that are connected by one double edge (trivial cycle) are considered to be two separated ntCSs.

4.3.2 Examples of Combining ntCSs

Two examples of combining ntCSs sharing one node and sharing one edge, respectively, will be presented next (Figures 6 and 7).

Shared Nodes An example for ntCSs sharing nodes is depicted in Figure 6. After constructing the path $(0, 1, 2, 3, 0)$, an already visited node is found (Figure 6(a)). After extracting the sub-path and reducing it (line 20), it is found that the $reducedPath$ is $(0, 1, 2, 3, 0)$ and thus not empty (line 21). Therefore, ntCS C1 is created (line 22–23). No sub ntCSs (line 24) are found and no merging (line 25) is necessary, as no other ntCSs exist until now. As no unvisited outgoing edges from node 0 exist, backtracking is performed until the current path reaches the state $(0, 1, 2)$. Now, the edge $2 \to 4$ is followed, and the path $(0, 1, 2, 4, 5, 6, 2)$ is constructed (Figure 6(b)). At this point, $Check_ntCS$ is called again. The extracted and reduced path is $(2, 4, 5, 6, 2)$. As it is not empty, a ntCS C2 is created. Again, no sub ntCSs are found. However, $MergeCycles$ will merge C2 into C1 due to the shared node 2.

If edge $2 \to 4$ is followed before edge $2 \to 3$, the situation depicted in Figure 6(c) is reached. As described before, a ntCS is constructed: C3 = {2, 4, 5, 6}. It contains no sub ntCSs and no other ntCSs exist. The algorithm continues, constructing the path $(0, 1, 2, 4, 5, 6, 2, 3, 0)$ (Figure 6(d)). Then, the extracted and reduced path is $(0, 1, 2, 4, 5, 6, 2, 3, 0)$ and ntCS C4 = {0, 1, 2, 3, 4, 5, 6} is created. ntCS C4 is merged into ntCS C3 yielding the same result as before when edge $2 \to 3$ is followed first at node 2.

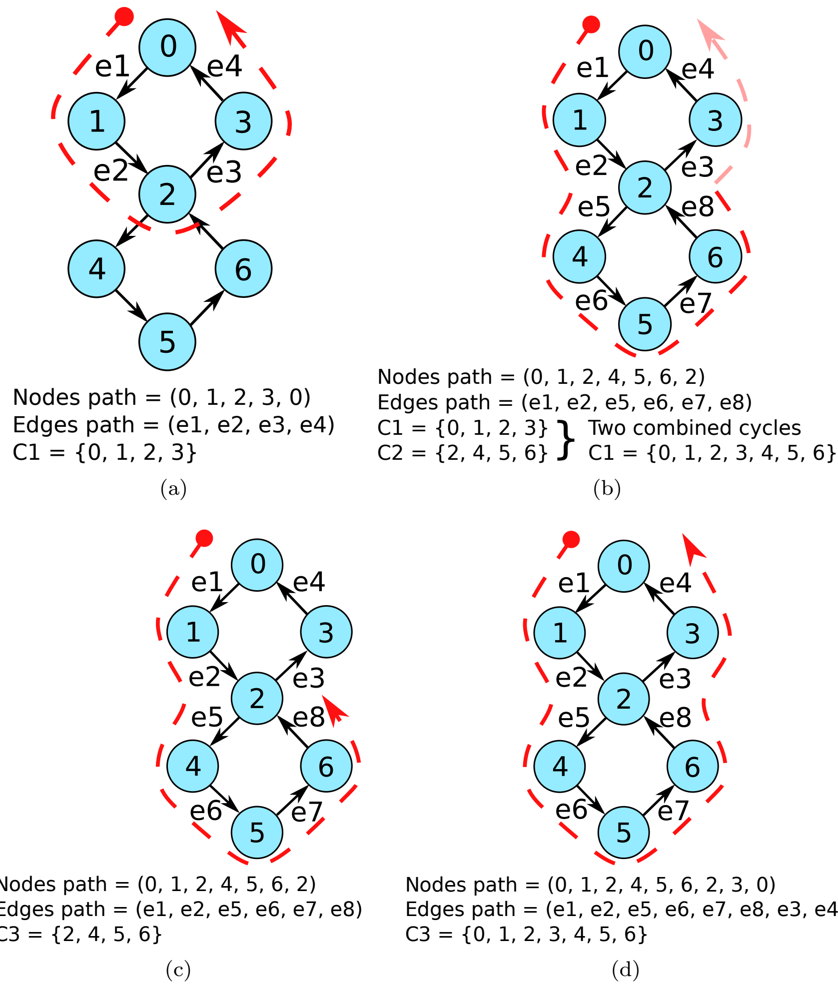


Figure 6: Two combined ntCSs sharing one node

Shared Edges An example for ntCSs sharing edges is shown in Figure 7. C1 is detected (Figure 7(a)) when visiting node 0 for the second time resulting in the current constructed path being $(0, 1, 2, 3, 0)$. The extracted and reduced path is $(0, 1, 2, 3, 0)$. Therefore, ntCS C1 is created. No sub ntCSs are found and no merging is necessary, as no other ntCSs exist. As no unvisited outgoing edges from node 0 exist, backtracking is performed until the current path reaches the state $(0, 1, 2, 3)$. Now, the edge $3 \to 4$ is followed, and the path $(0, 1, 2, 3, 4, 5, 2)$ is constructed (Figure 7(b)). Here, *Check_ntCS* is called again. The extracted and reduced path is $(2, 3, 4, 5, 2)$. As it is complete, a ntCS C2 is created. Again, no sub ntCSs are found. However, *MergeCycles* will merge C2 into C1 due to the shared edge $2 \to 3$.

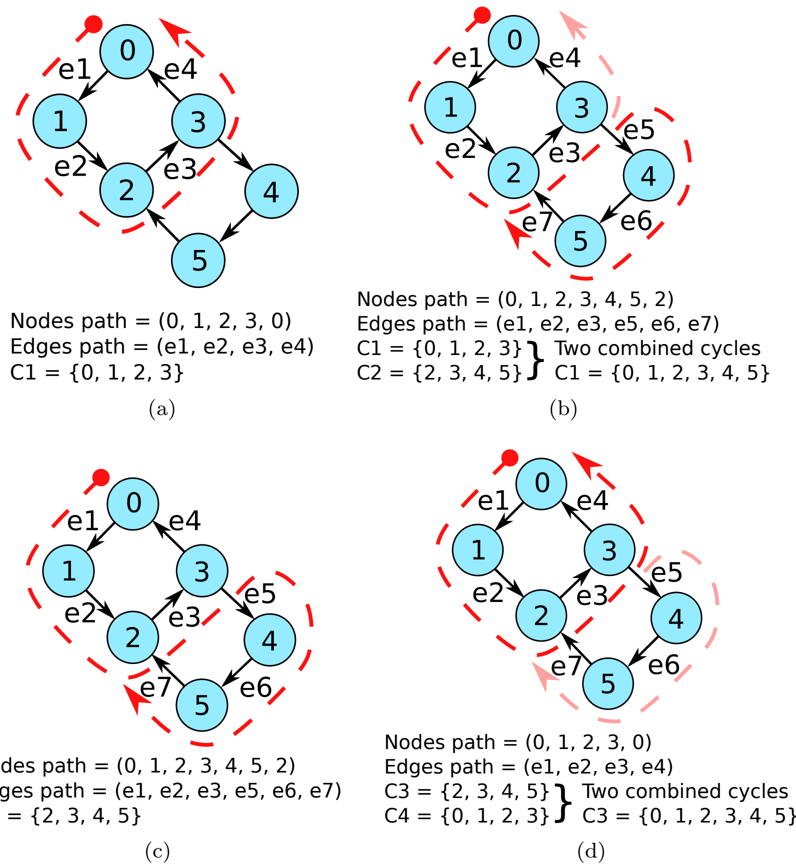


Figure 7: Two combined ntCSs sharing one edge

If edge $3 \to 4$ is followed before edge $3 \to 0$, the situation displayed in Figure 7(c) is obtained. As described before, a ntCS is constructed: C3 = {2, 3, 4, 5}. It contains no sub ntCSs and no other ntCSs exist. As no remaining unvisited outgoing edges from node 2 exist, backtracking is performed until the current path attains the state $(0, 1, 2, 3)$. Now, the edge $3 \to 0$ is followed, and the path $(0, 1, 2, 3, 0)$ is constructed (Figure 7(d)). Then, *Check_ntCS* is called again. The extracted and reduced path is $(0, 1, 2, 3, 0)$. As it is not empty, a ntCS C4 is created. Again, no sub ntCSs are found. However, *MergeCycles* will merge C3 into C4 due to the shared edge $2 \to 3$.

4.3.3 Examples of Handling Partial ntCSs

This example deals with handling partial ntCSs (*ParticalCycle* [1]). Let us assume that C1 is detected as shown in Figure 8(a) similar to the case in Figure 7(a). After, backtracking and following edge $2 \to 4$, the path $(0, 1, 2, 4, 5, 3)$

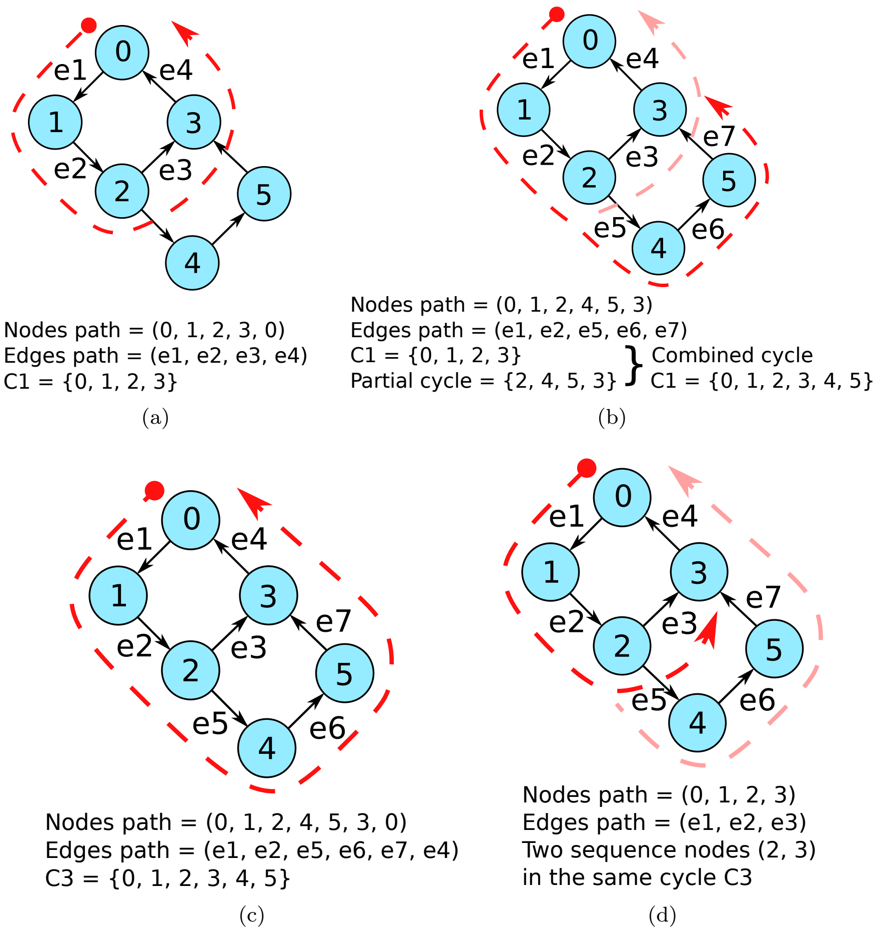
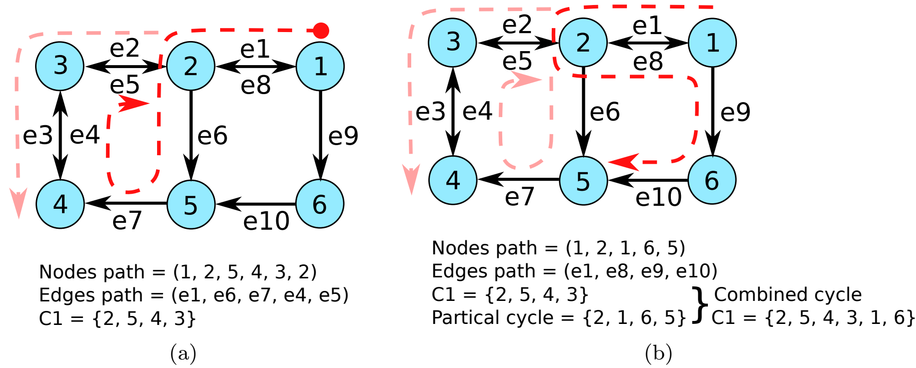


Figure 8: Partial ntCS

is constructed (Figure 8(b)). Now, node 3 belongs already to ntCS C1. Therefore, for each node in the current path—from the last to the first—it is checked, if the node belongs to C1, too. In the example, this holds for node 2. Thus, the partial ntCS C' = {2, 4, 5, 3} is created and merged into C1.

If edge $2 \rightarrow 4$ is followed before edge $2 \rightarrow 3$, then the ntCS C3 = {0, 1, 2, 3, 4, 5} is detected (Figure 8(c)). After, backtracking and following edge $2 \rightarrow 3$, the path $(0, 1, 2, 3)$ is constructed (Figure 8(d)). As nodes 2 and 3 are the last two nodes of the current path and as they belong to the same ntCS C3 (lines 16–18), no new ntCS is found and the algorithm returns "no ntCS found".



(a) (b)

Figure 9: Example for deriving ntCS.

4.3.4 Complex Example

Using a combination of all four algorithms allows to resolve the situation shown in Figure 9. In the detection sequence shown here, first (Figure 9(a)), ntCS C1 = {2, 5, 4, 3} is extracted from path $(1, 2, 5, 4, 3, 2)$. Then (Figure 9(b)), after backtracking, the partial ntCS C2 = {2, 1, 6, 5} is found and merged into C1.

All previously shown cases are the basic ones that can occur while decomposing a weakly connected component. All other weakly connected components contain these basic cases (or reduced versions thereof).

4.4 Complexity Analysis

Let $n = |V|$ denote the number of nodes, $e = |E|$ the number of edges, and c the number of ntCSs of a directed graph G. Tables 1–4 show the complexity of the ntCS detection algorithms providing additional information compared to [1].

The overall time and space complexity of ntCS detection is $O(c^3 \cdot n^2 \cdot (n+e))$. They are dominated by calling $Find_ntCS$ (Algorithm 2, line 6) for each not visited node, whose time and space complexity equals $O(c^3 \cdot n \cdot (n+e))$ (Table 1).

The time and space complexity of $Find_ntCS$ are dominated by the time and space complexity of $Check_ntCS$ (Algorithm 4, line 3).

Performing constant operations, $Find_ntCS_Rec$ (Algorithm 3) has time and space complexity equal $O(1)$ (Table 2). Please note, that the time and space complexity of calling $Find_ntCS_Rec$ from $Find_ntCS$ (Algorithm 1) in line 16 and line 26 are constant because $Find_ntCS_Rec$ can be considered being an inline block or part of $Find_ntCS$ which is introduced to avoid duplicated code.

$Check_ntCS$ (Algorithm 4) has time and space complexity being equal to $O(c^3 \cdot n \cdot (n+e))$ (Table 3). They are dominated by the time and space complexity of $PartialCycle$ (line 29, [1]).

$ComputeReducedPath$ has time and space complexity equal to $O(n)$ (Table 4). Both are dominated by list iteration at lines 3–5 and lines 11–21.

Table 1: Complexity analysis of *Find_ntCS* (Algorithm 2)

Line	Time complexity	Space complexity	Comments
3	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	Algorithm 4
7	$O(1)$		Hashtable get
8	$O(e)$	$O(e)$	Hashtable iterator
10	$O(1)$		
11	$O(1)$	$O(1)$	HashSet create
12	$O(1)$	$O(1)$	HashSet add
13	$O(1)$	$O(1)$	Hashtable add
16	$O(1)$		Algorithm 3
8–19	$\boldsymbol{O(e)}$	$\boldsymbol{O(e)}$	Hashtable iterator
22	$O(e)$		Hashtable iterator
26	$O(1)$		Algorithm 3
22–28	$\boldsymbol{O(e)}$		Hashtable iterator
Overall	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	

Table 2: Complexity analysis of *Find_ntCS_Rec* (Algorithm 3)

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	ArrayList add
3	$O(1)$	$O(1)$	Hashtable add
4	$O(1)$		Algorithm 1
5	$O(1)$		ArrayList remove
6	$O(1)$		Hashtable remove
Overall	$O(1)$	$O(1)$	

Table 3: Complexity analysis of *Check_ntCS* (Algorithm 4)

Line	Time complexity	Space complexity	Comments
5	$O(1)$		Hashmap get
6	$O(1)$		Hashmap get
10–15	$O(n)$		ArrayList iterate
20	$O(n)$	$O(n)$	Algorithm 5
23	$O(n+e)$	$O(n+e)$	HashSet add nodes and edges of one ntCS
24	$O(n)$	$O(1)$	SubCycle [1]
25	$O(c^3 \cdot (n+e))$	$O(c^3 \cdot (n+e))$	MergeCycles [1]
29	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	PartialCycle [1]
Overall	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	

Table 4: Complexity analysis of *ComputeReducedPath* (Algorithm 5)

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	ArrayList create
3	$O(n)$	$O(n)$	ArrayList iterate
4	$O(1)$	$O(1)$	ArrayList add
3–5	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	ArrayList iterate, add
9	$O(1)$	$O(1)$	LinkedHashSet create
11	$O(n)$	$O(n)$	ArrayList iterate
12	$O(1)$		ArrayList get using index
13	$O(1)$	$O(1)$	ArrayList add
14	$O(1)$		Hashtable get
11–21	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	ArrayList iterate, get, add
23	$O(n)$	$O(n)$	ArrayList create initialized by LinkedHashSet
Overall	$O(n)$	$O(n)$	

5 Splitting wCCs

5.1 Motivation

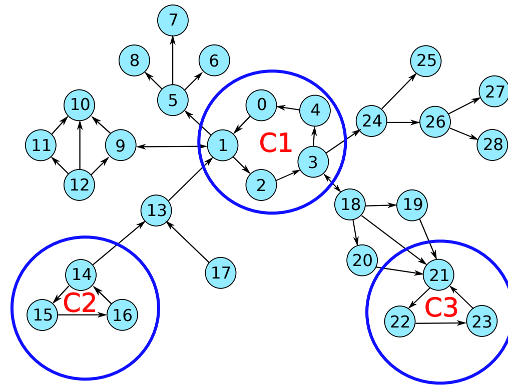
Figure 10(a) shows the situation of a wCC after detecting ntCSs. In this case, the ntCSs C1, C2, and C3 were detected. If all edges of these ntCSs are temporarily removed from this wCC, two types of wCCs remain (Figure 10(b)). The resulting isolated nodes forming one wCC each are ignored, as they belong to a single ntCS, only. The remaining two wCCs, however, contain edges and require further analysis before the categorization step (Section 6). A closer analysis of the different configurations that might occur shows that these wCCs could be further decomposed by splitting them at nodes of the ntCSs found (Figure 10(c)):

- The *green*, the *orange*, and the *red* wCCs are only attached to C1 at nodes 1, 3, and 1, respectively.
- The *brown* wCC connects C1 (node 1) and C2 (node 14).
- The *blue* wCC connects C1 (node 3) and C3 (node 21).

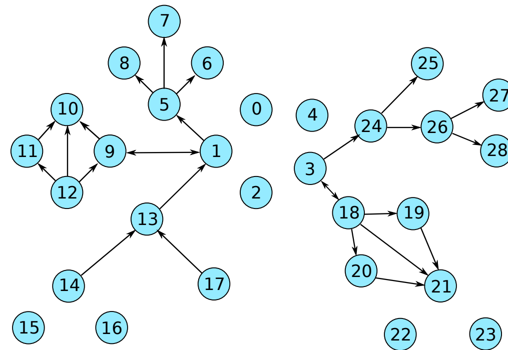
Thus, we can distinguish between those parts of the wCC that are only connected to one ntCS and those that connect two or more ntCSs. This is an important fact that can later be used for an improved layout of the graph [1, 2].

5.2 Algorithms

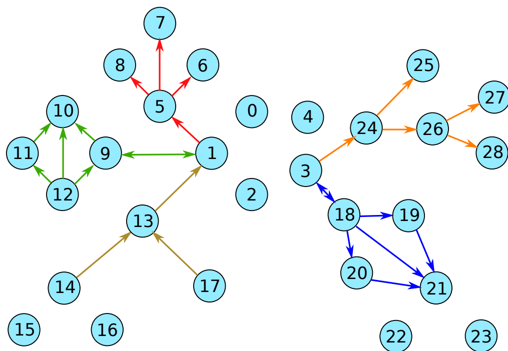
After removing the edges of the ntCSs, each node of each ntCS is taken as starting point of the split algorithm if it still has incoming or outgoing edges. These nodes are the potential split points examined by the algorithm; isolated



(a) One wCC after detecting ntCSs (marked using blue circles).



(b) The graph obtained after deleting the edges of the ntCSs found.



(c) Splitting the two wCCs with edges at the ntCSs nodes results in five wCCs in the final decomposition.

Figure 10: Removing the edges of the ntCSs detected and splitting the remaining wCCs with edges at the ntCSs nodes.

nodes are ignored. All edges of the wCC are treated as being undirected. For each node, all unvisited edges are followed. From the end node of each of the edges, a depth first search is started. A traversed edge will not be considered again. The depth first search backtracks, if the end node of an edge has no unchecked edges or if it belongs to a ntCS. Finally, each wCC found is added to the list of all wCCs.

The split is mainly performed by two functions: *Split_wCC_at_Node* (Algorithm 6) and *Split_wCC* (Algorithm 7). For each ntCS node, *Split_wCC_at_Node* follows all unvisited edges of the node (lines 5–6), creating a set to add all nodes in the sub-wCC (line 7), and adding the ntCS node to this set (line 8). Then, *Split_wCC_at_Node* calls *Split_wCC* to start the depth first search from the end node of the current edge considering the edges as being undirected (line 9). Finally, the wCC is formed from the set of nodes (line 10) and is added to the list of all wCCs (line 12), if it is not empty.

The *Split_wCC* algorithm (Algorithm 7) performs a depth first search. First, the current edge is marked as visited (line 1) as well as its reverse edge if it exists (lines 2–5) as edges are treated ignoring their direction and therefore edge and reverse edge are considered being the same. Then, the next node in the depth first search is determined (lines 6–10). If this node is a ntCS node of the initial wCC (line 11), no further edges are followed, and the node is added to the set of all ntCS nodes of this wCC (line 12) and to the set of all nodes (line 13). It might have been visited before as it is a split point being shared by more than one sub-component. Therefore, it is added to each of the sub-components. Moreover, the node might have unvisited edges that will be handled later by the split algorithm. Otherwise (lines 14–22), the node is not a ntCS node. If it was not visited before (line 14), it is marked as visited (line 15) and added to the set of all nodes (line 16). All unvisited edges of the node are then followed calling *Split_wCC* recursively (lines 17–22). If the node is marked and not a ntCS node, then either all edges were already followed or will be followed. Thus, this case is already handled implicitly.

Considering the example in Figure 10(c), *Split_wCC_at_Node* is called for the ntCS nodes 1 and 3. Starting at node 1, it will traverse the edges $1 \to 5$ (red), $1 \leftrightarrow 9$ (green), and $13 \to 1$ (brown), one after the other. Thereby, the order is not important. Starting with edge $1 \to 5$, the edge is marked as visited, an empty set is created, node 1 is added to the set, and *Split_wCC* is called to start the depth first search at node 5. After completing the search, the set of nodes contains the nodes $\{1, 5, 6, 7, 8\}$ forming a new wCC (red) which will be added to the list of all wCCs. The same holds for edges $1 \leftrightarrow 9$ and $13 \to 1$ resulting in the sets of nodes $\{1, 9, 10, 11, 12\}$ (green) and $\{1, 13, 14, 17\}$ (brown) forming two additional new wCCs. Handling edges $3 \to 24$ and $3 \leftrightarrow 18$ of node 3 will produce the sets $\{3, 24, 25, 26, 27, 28\}$ (orange) and $\{3, 18, 19, 20, 21\}$ (blue) forming two new ntCSs, respectively.

In the cases 1 (red), 2 (green), and 4 (orange), only the *else* statement (lines 14–22) is executed. In the cases 3 (brown) and 5 (blue), however, the code in lines 11–13 is executed. For the last case (number 5, blue), this means that reaching node 21 for the first time, no other incoming edge is followed. The

Table 5: Complexity analysis of *Split_wCC_at_Node* (Algorithm 6)

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	HashSet Create
2	$O(1)$	$O(1)$	HashSet Create
3	$O(1)$	$O(1)$	HashSet Add
4	$O(e_i)$	$O(e_i)$	HashSet Add multiple
5			HashSet Iterator
7	$O(1)$	$O(1)$	HashSet Create
8	$O(1)$	$O(1)$	HashSet Add
9			Algorithm 7
10	$O(e_i)$	$O(e_i)$	create_Graph [1]
12	$O(1)$	$O(1)$	ArrayList Add
Overall	$O(e_i)$	$O(e_i)$	

Table 6: Complexity analysis of *Split_wCC* (Algorithm 7)

Line	Time complexity	Space complexity	Comments
2	$O(1)$		Hashtable get
11	$O(1)$		HashSet containsKey (get)
12	$O(1)$	$O(1)$	HashSet Add
13	$O(1)$	$O(1)$	HashSet Add
16	$O(1)$	$O(1)$	HashSet Add
17	$O(e_i)$	$O(e_i)$	HashSet Add multiple
18			HashSet iterator
20			Algorithm 7
Overall	$O(e_i)$	$O(e_i)$	

node is added to both sets and backtracking is performed. The same holds the second time, node 21 is reached. As node 21 is already contained in both sets, it is not added. The same holds the third and last time node 21 is reached.

5.3 Complexity Analysis

Let $S = \{wCC_i\}$ the set of all wCCs remaining after the ntCS edges were removed from the wCC being the input graph. Let $wCC_i = (N_i, E_i)$ be a weakly connected component with N_i being the set of nodes and E_i being the set of edges. Let $n_i = |N_i|$ be the number of nodes and $e_i = |E_i|$ be the number of edges of wCC_i. Then, the total time and space complexity of the split algorithm is $O(e_i)$ for wCC_i.

This can be seen as follows. All HashSet operations have amortized time and space complexity $O(1)$ (Tables 5 and 6). The call to algorithm *Split_wCC* is only performed for each edge at most once (Algorithm 6, line 9 and Algorithm 7, line 20), as afterwards the edge is marked as visited (Algorithm 7, line 1). ntCS

Table 7: Conditions used for detecting down-trees, up-trees, and DAGs [1, 2]. Examples are shown in Figure 12.

Case	Condition	Result
I	$n_i \geq 2$ and $n_o \geq 2$	DAG
II	$n_i < 2$ and $n_o \geq 2$	down-tree or DAG
III	$n_i \geq 2$ and $n_o < 2$	up-tree or DAG
IV	$n_i < 2$ and $n_o < 2$	down-tree, up-tree, or DAG

nodes might be visited several times, but their edges are only followed once when calling Algorithm 6. Thus, all ntCS nodes belonging to wCC_i are reached by at most e_i edges. All non-ntCS nodes are handled only once and are then marked (Algorithm 7, lines 14 and 15). Therefore, also their edges are handled only once, i.e., at most e_i edges are followed (Algorithm 6, lines 4–5 and Algorithm 7, lines 17–18). Finally, the algorithm *create_Graph* [1] is called with e_i edges in sum over all split parts of wCC_i.

6 Detecting Trees and DAGs

6.1 Motivation

The wCCs that result from the previous splitting step will finally be classified as down-trees, up-trees, and DAGs. Distinguishing between trees and DAGs is important, as for trees optimal drawing algorithms exist [6, 9] while all algorithms for DAGs are based on heuristics [8]. The distinction between down- and up-trees is made for choosing the adapted drawing algorithms, as in the first case outgoing edges will be handled while in the second case incoming edges will be handled.

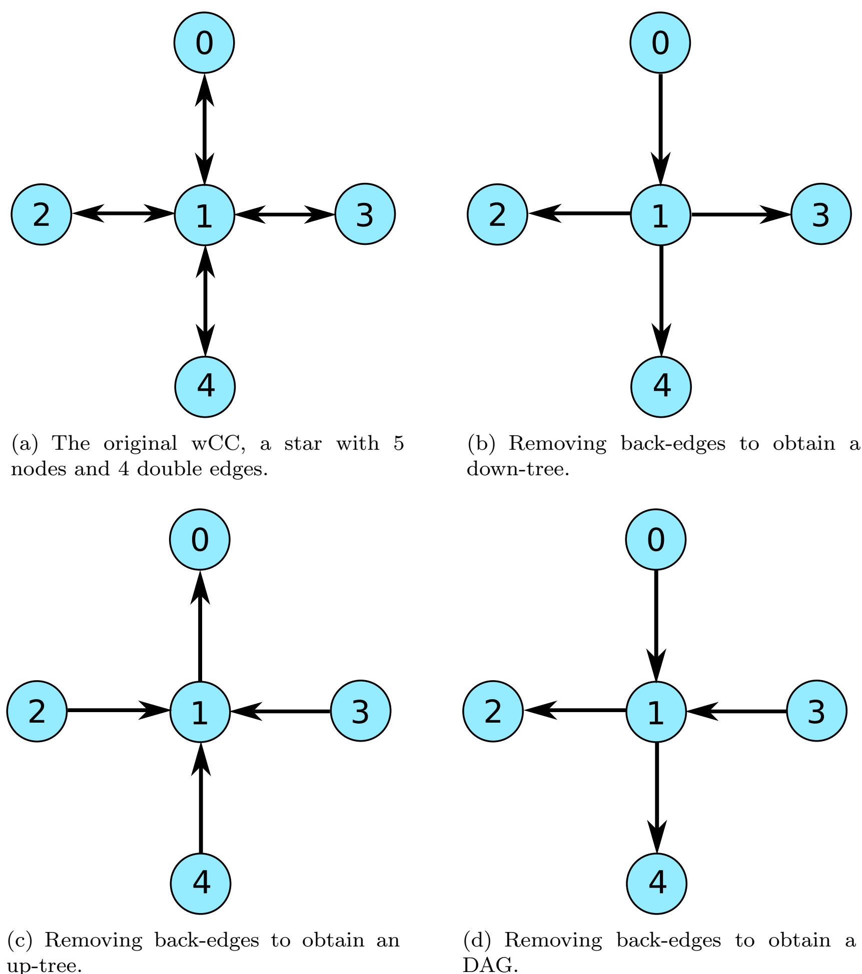
The wCC of G1 shown in Figure 1(a) (Section 2) can only be classified as down-tree or up-tree (Figure 1(b), Section 2). However, the larger wCC shown in Figure 11(a) is a minimal example of a wCC that could be classified as down-tree (Figure 11(b)), up-tree (Figure 11(c)), and DAG (Figure 11(d)), depending of which back-edges are removed.

Our algorithm was designed such that a wCC is classified as down-tree, if possible, as up-tree, if it can be classified as up-tree but not as down-tree, and as DAG, if no other classification is possible. The reason for preferring trees over DAGs was explained before: trees allow for improved algorithms. The choice of preferring down-trees over up-trees is arbitrary.

6.2 Algorithms

6.2.1 Detecting Trees and DAGs

To classify the wCCs, the four cases listed in Table 7 are distinguished. They take the number of source nodes n_i (in-degree zero) and sink nodes n_o (out-degree zero) of the respective wCC into account. In case I, $n_i \geq 2$ and $n_o \geq 2$.



(a) The original wCC, a star with 5 nodes and 4 double edges.

(b) Removing back-edges to obtain a down-tree.

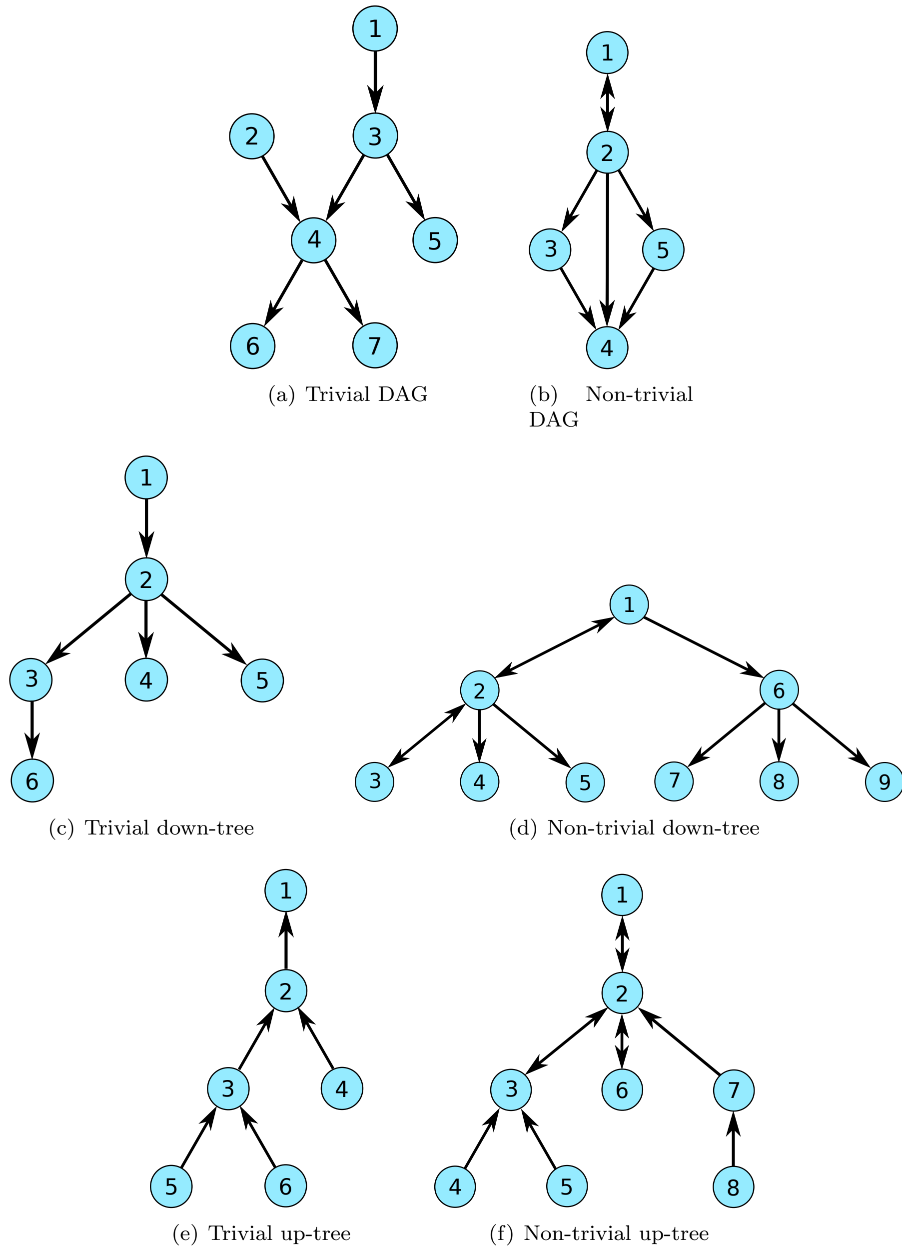
(c) Removing back-edges to obtain an up-tree.

(d) Removing back-edges to obtain a DAG.

Figure 11: A "star" wCC with five nodes is the minimal example of a wCC that can be transformed into a down-tree, an up-tree, and a DAG by removing the respective back-edges.

Thus, the wCC has to be a DAG and is classified as one. The other cases allow for tree or DAG classification. Cases II and III restrict the possible tree classification, while in the last case both tree classifications are possible. The cases II and III are handled using depth first search.

This decision table is implemented in algorithm $DetectDAGsTrees$ (Algorithm 8). First, additional information about the wCC is computed (lines 1–15). All nodes having in-degree zero and all nodes having out-degree zero are counted (lines 3 and 6, respectively). Further, they are added to the lists $inDegreeZeroNodes$ (line 4) and $outDegreeZeroNodes$ (line 7), respectively.



(a) Trivial DAG

(b) Non-trivial DAG

(c) Trivial down-tree

(d) Non-trivial down-tree

(e) Trivial up-tree

(f) Non-trivial up-tree

Figure 12: Examples of trees and DAGs.

Three more lists hold the nodes having only double edges (lines 8–9), only double and outgoing edges (lines 10–11), and only double and incoming edges (lines 12–13).

Then, the two counts are used for implementing the decision table presented before. In case I, the classification is DAG (lines 16–17) and a DAG is created (lines 25–27). The graph in Figure 12(a) shows an example of this case. The graph has two nodes with in-degree zero $\{1, 2\}$ and three nodes with out-degree zero $\{5, 6, 7\}$.

In case II (III), the results can be either a down-tree (an up-tree) or a DAG (lines 18–19, respectively 20–21, Section 6.2.2). If it is a tree, the tree is created during the decision process by $checkForDownTree$, Algorithm 9 ($checkForUpTree$, Algorithm 11). If the construction of the down-tree (up-tree) fails, the wCC is a DAG and a DAG is created. The graphs shown in Figures 12(c) and 12(d) are examples of case II. The graph in Figure 12(c) has one node with in-degree zero $\{1\}$, three nodes with out-degree zero $\{4, 5, 6\}$, and no double edges. Thus, it is a trivial down-tree. The wCC in Figure 12(d) has no nodes with in-degree zero and five nodes with out-degree zero $\{4, 5, 7, 8, 9\}$. In this case, a non-trivial down-tree can be constructed (see also Section 6.2.2). The graphs shown in Figures 12(e) and 12(f) are examples of case III. The wCC in Figure 12(e) has three nodes with in-degree zero $\{4, 5, 6\}$ one node with out-degree zero $\{1\}$, and no double edges. Thus, it is a trivial up-tree. The wCC in Figure 12(f) has three nodes with in-degree zero $\{4, 5, 8\}$ and no nodes with out-degree zero. In this case, a non-trivial up-tree can be constructed (see also Section 6.2.2).

In case IV, the classification is performed by calling $classify_wCC$ (lines 22–23, Algorithm 13). As before, if the wCC is classified as a down-tree or an up-tree, it is created during the classification process. Otherwise, a DAG is detected and will be created at the end (Algorithm 8, lines 25–27). In fact, this algorithm tries first to construct a down-tree (line 1). If this fails, it tries to construct an up-tree (lines 2–4). Finally, the result is returned (line 5). The wCC shown in Figure 12(b) is an example of case IV where the wCC has no nodes with in-degree zero, one node with out-degree zero $\{4\}$, and one node having only double edges $\{1\}$. Starting at node 4, the construction of an up-tree fails as node 2 has three outgoing edges. Starting at node 1, the construction of a down-tree fails as node 4 has three incoming edges. Thus, the final classification is DAG.

6.2.2 Down-Trees and Up-Trees

The algorithm $checkForDownTree$ (Algorithm 9) tries to construct a down-tree using the edges of the wCC. If the wCC does not contain double edges, then the unique trivial down-tree will be constructed. If the wCC contains double edges, it might be possible to construct one down-tree or several different down-trees. In the first case, the unique down-tree will be constructed. In the second case, one of the down-trees will be constructed. In all these cases, the wCC will be classified as down-tree. Otherwise, no down-tree can be constructed and the wCC will be classified as DAG.

The algorithm starts by incrementing the global path counter (line 1) which is used to check, if a node is reached twice ($checkForDownTreeDFS$: line 1). Further, a list of path nodes is created (line 2). Now, two cases are distinguished. If there is exactly one node with in-degree zero (line 3), only this node can be the root of the tree and is used as starting node for finding a down-tree (lines 4–16). Otherwise (lines 17–34), all potential root nodes will be used as starting nodes for finding a down-tree. In both cases, it is possible that no down-tree can be constructed. As both cases perform essentially the same steps with one and several starting nodes, respectively, the second case will be described. Pairs of line numbers are given in the form (second case, first case). If the line numbers for the first case are missing, the respective step is not needed for this case.

The detection starts by adding all starting nodes to a list in the order of nodes containing (1) only outgoing edges (line 18), (2) only outgoing and double edges (line 19), and (3) only double edges (line 20). Then, in case 2, all these nodes are considered one after the other (line 21). In case 1, the first (and only) node with in-degree zero is used as starting node (line 7). Now, all nodes are marked as unvisited (lines 22–24, 4–6) and the path nodes list is emptied (line 25). The current node is used as root node (line 26, 8), added to the current path (line 27, 9), and the depth first search of a down-tree is started calling $checkForDownTreeDFS$ (line 28, 10). If a down-tree could be constructed, it is created and the algorithm stops returning the classification "down-tree" (lines 29–32, 11–13). Otherwise, in case 2, the next potential starting node is considered. If no down-tree could be created using any of the potential starting nodes, the algorithm stops returning the classification "DAG" (line 34, 14–15).

The algorithm $checkForDownTreeDFS$ (Algorithm 10) first checks, if the current node was visited before (line 1). In this case, the wCC is a DAG and this classification is returned (line 2). Otherwise, if the path contains more than one node, it is checked whether the current node has incoming edges without back edges ignoring the edge from the parent node to the current node (line 4). If yes, then a DAG is detected and this classification is returned (line 5). Otherwise, the current node is marked (line 7) and the result is set to down-tree (line 8). Now, all outgoing edges of the current node are followed one after the other (line 9). First, the target node of the current outgoing edge is added to the path (lines 10–11). Then, it is checked, if the last edge was a back edge of the previous edge (lines 12–14). In this case, the last node of the path is removed and the next outgoing edge will be considered (lines 15–16). Otherwise, $checkForDownTreeDFS$ is called recursively (line 18). Afterwards, the path is restored and the result is checked (lines 19–22). If the result was DAG, then this result is returned. Only if none of the edges followed results in a DAG classification, the construction of the down-tree was successful and this classification is returned (line 24).

The wCC shown in Figure 12(c) is a trivial down-tree with root node 1. The wCC shown in Figure 12(d) has the potential root node 1. This node has one outgoing and one double edge. A non-trivial down-tree can be constructed in this case.

The attempt to construct up-trees is performed analogously to the down-

Table 8: Complexity analysis of *DetectDAGsTrees* (Algorithm 8)

Line	Time complexity	Space complexity	Comments
1	$O(n)$	$O(n)$	ArrayList Iterator
4	$O(1)$	$O(1)$	ArrayList add
7	$O(1)$	$O(1)$	ArrayList add
8	$O(e)$	$O(e)$	ArrayList Iterator
9	$O(1)$	$O(1)$	ArrayList add
10	$O(e)$	$O(e)$	ArrayList Iterator
11	$O(1)$	$O(1)$	ArrayList add
12	$O(e)$	$O(e)$	ArrayList Iterator
13	$O(1)$	$O(1)$	ArrayList add
1–15	$O(n + e)$	$O(n + e)$	
19	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 9
21	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 11
23	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 13
26	$O(n + e)$	$O(n + e)$	create DAG [1]
Overall	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	

tree construction (Algorithms 11 and 12). Instead of outgoing edges, incoming edges are followed.

The wCC shown in Figure 12(e) is a trivial up-tree with root node 1. The wCC shown in Figure 12(f) has the potential root nodes 1, 2, and 6. Nodes 1 and 6 have one double edge and node 2 has three double edges and one incoming edge. In all cases, a non-trivial up-tree can be constructed. The first non-trivial up-tree found is created as the result.

6.3 Complexity Analysis

Let n be the number of nodes, e be the number of edges, and md be the maximal degree of a node of the wCC to be classified. Then, the total time and space complexity of detecting trees and DAGs is $O(n^2 \cdot md + e)$ (Table 8). It is dominated by the tree detection algorithms *checkForDownTree* and *checkForUpTree* (Table 9). Both have time and space complexity $O(n^2 \cdot md + e)$. Therefore, also algorithm *classify_wCC* has the same time and space complexity (Table 11). Creating the lists of potential root nodes and computing the two counters can be done in $O(n + e)$ as each edge is checked at most twice: once for the source and once for the target node. Creating the DAG can also be done in $O(n + e)$ time and space.

Algorithm *checkForDownTree* (*checkForUpTree*) has time and space complexity $O(n^2 \cdot md + e)$. The e comes from the down-tree (up-tree) construction (lines 12 and 30); at most one down-tree (up-tree) is constructed. The factor $n \cdot md$ of the first summand comes from the recursive construction of the down-tree using *checkForDownTreeDFS* (up-tree, *checkForUpTreeDFS*). The factor n of the first summand comes from the worst case example.

Table 9: Complexity analysis of *checkForDownTree* (Algorithm 9). The complexity analysis for *checkForUpTree* (Algorithm 11) is identical (given in brackets).

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	ArrayList Create
4	$O(n)$		ArrayList Iterator
7	$O(1)$		ArrayList get
9	$O(1)$	$O(1)$	ArrayList add
10	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 10 (Algorithm 12)
12	$O(n+e)$	$O(n+e)$	create down-tree (up-tree) [1]
18	$O(n)$	$O(n)$	ArrayList addAll
19	$O(n)$	$O(n)$	ArrayList addAll
20	$O(n)$	$O(n)$	ArrayList addAll
21	$O(n)$	$O(n)$	ArrayList Iterator
22	$O(n)$		ArrayList Iterator
25	$O(n)$		ArrayList clear
27	$O(1)$	$O(1)$	ArrayList add
28	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 10 (Algorithm 12)
30	$O(n+e)$	$O(n+e)$	create down-tree (up-tree) [1]
21-33	$\boldsymbol{O(n^2 \cdot md + e)}$	$\boldsymbol{O(n^2 \cdot md + e)}$	$O(n \cdot (n+1+n \cdot md)$ $+(n+e))$
Overall	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	

Table 10: Complexity analysis of *checkForDownTreeDFS* (Algorithm 10). The complexity analysis for *checkForUpTreeDFS* (Algorithm 12) is identical (given in brackets).

Line	Time complexity	Space complexity	Comments
4	$O(md)$	$O(md)$	[1]
9	$O(md)$	$O(md)$	HashSet Iterator
11	$O(1)$	$O(1)$	ArrayList Add
12	$O(n)$		ArrayList indexOf
13	$O(n)$		ArrayList lastIndexOf
15	$O(1)$		ArrayList remove last
18	$O(n)$	$O(n)$	Algorithm 10
19	$O(1)$		ArrayList remove last
9-23	$\boldsymbol{O(n \cdot md)}$	$\boldsymbol{O(n \cdot md)}$	
Overall	$O(n \cdot md)$	$O(n \cdot md)$	

Table 11: Complexity analysis of $classify_wCC$ (Algorithm 13)

Line	Time complexity	Space complexity	Comments
1	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 9
3	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 11
Overall	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	

The time and space complexity of the down-tree (up-tree) construction $checkForDownTreeDFS$ ($checkForUpTreeDFS$) is $O(n \cdot md)$. For each outgoing edge of a node (limited by the maximal node degree md) two indices are computed and a recursive call is performed. The recursive call is limited by the number of nodes n, as at most one node is visited twice before the final classification.

Finally, algorithm $classify_wCC$ has time and space complexity $O(n \cdot md + e)$ as it essentially calls the algorithms $checkForDownTree$ and $checkForUpTree$ (Algorithms 9 and 11) whose time and space complexity are $O(n \cdot md + e)$.

7 Conclusion

To create a topological visualization of directed graphs, a new methodology was previously introduced [1, 2]. Here, an intermediate level description of the decomposition process introduced there is provided as complementary description to the previous high [2] and low [1] level descriptions. The focus in this paper is on the motivation of the process and of each of the steps as well as on illustrative examples of all cases that need to be considered by the algorithm, standard as well as more complex ones. All other situations include those described here.

All algorithms are complemented by their complexity analysis. It shows, that the detection of non-trivial cyclic subgraphs has $O(c^3 \cdot n^2 \cdot (n+e))$ as overall time and space complexity, where n designates the number of nodes, e the number of edges, and c the number of cycles found; splitting weakly connected components resulting from removing the edges of the non-trivial cyclic subgraphs can be performed in $O(e_i)$ as total time and space complexity where e_i is the number of edges of each of the weakly connected components, respectively; and classifying the resulting weakly connected components as trees and DAGs can be performed in $O(n^2 \cdot md + e)$ total time and space complexity, where n designates the number of nodes, e the number of edges, and md is the maximum degree of a node of the weakly connected component.

References

[1] A. Abuthawabeh. *Multi-Edge Graph Visualizations for Fostering Software Comprehension.* PhD thesis, Technische Universität Kaiserslautern, Kaiserslautern, Germany, 2016.

[2] A. Abuthawabeh and D. Zeckzer. An Improved Decomposition and Drawing Process for Optimal Topological Visualization of Directed Graphs. In *Proceedings of the 31th Spring Conference on Computer Graphics*, SCCG'15, pages 111–118. ACM, 2015. doi:10.1145/2788539.2788551.

[3] C. Bachmaier, F. Brandenburg, W. Brunner, and R. Fülöp. Coordinate assignment for cyclic level graphs. In *Computing and combinatorics*, pages 66–75. Springer, 2009. doi:10.1007/978-3-642-02882-3_8.

[4] C. Bachmaier, F. Brandenburg, W. Brunner, and R. Fülöp. Drawing Recurrent Hierarchies. *J. Graph Algorithms Appl.*, 16(2):151–198, 2012. doi:10.7155/jgaa.00254.

[5] C. Bachmaier, F. Brandenburg, W. Brunner, and G. Lovász. Cyclic Leveling of Directed Graphs. In *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 348–359. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-00219-9_34.

[6] C. Buchheim, M. Jünger, and S. Leipert. Improving Walker's Algorithm to Run in Linear Time. In M. T. Goodrich and S. G. Kobourov, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 344–353. Springer Berlin Heidelberg, 2002. doi:10.1007/3-540-36151-0_32.

[7] 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, Feb 1981. doi:10.1109/TSMC.1981.4308636.

[8] R. Tamassia, editor. *Handbook of Graph Drawing and Visualization.* Discrete Mathematics and Its Applications. Chapman & Hall/CRC, 2007.

[9] J. Q. Walker. A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20(7):685–705, 1990. doi:10.1002/spe.4380200705.

A Algorithms for Detecting Non-Trivial Cyclic Subgraphs

Algorithm 1 $Find_ntCS_in_wCC$

Description: Find all ntCSs of a wCC
Input: wCC
Output: $ntCSs$

1: **for all** $node \in wCC$ **do**
2: **if** $node$ is not marked **then**
3: Create a path of nodes: $pathNodes$
4: Add $node$ to $pathNodes$
5: Create a path of edges: $pathEdges$
6: Call $Find_ntCS$($ntCSs$, $pathNodes$, $pathEdges$, $node$) (Algorithm 2)
7: **end if**
8: **end for**

Algorithm 2 *Find_ntCS*

Description: Find all disjoint ntCSs in the graph using Depth First Search
Input: *ntCSs*, *pathNodes*, *pathEdges*, *lastNode*
Output: *ntCSs*

```
1: Increment lastNode.iEdgeCounter by one
2: if lastNode is marked with value = counterPath then
3:   Check_ntCS(ntCSs, pathNodes, pathEdges) (Algorithm 4)
4: else
5:   Mark lastNode with counterPath
6: end if
7: lastNodeBackEdges ← backEdges.get(lastNode)
8: for all outgoingEdge ∈ {outgoing edges of lastNode} do
9:   if outgoingEdge is marked as not VISITED then
10:    if isBackEdge(pathEdges, outgoingEdge) then
11:      If lastNodeBackEdges was not created before, create new one
12:      Add outgoingEdge into lastNodeBackEdges
13:      Put (lastNode, lastNodeBackEdges) in backEdges hashtable
14:    else
15:      targetNode ← target node of outgoingEdge
16:      Call Find_ntCS_Rec(ntCSs, pathNodes, pathEdges, targetNode,
   outgoingEdge, position(lastNode)) (Algorithm 3)
17:    end if
18:  end if
19: end for
20: if lastNode.iEdgeCounter >= lastNode number of incoming edges then
21:   if lastNodeBackEdges ≠ null then
22:     for all backEdge ∈ lastNodeBackEdges do
23:       if backEdge edge is marked as not VISITED then
24:         outgoingEdge ← backEdge
25:         targetNode ← target node of backEdge
26:         Call Find_ntCS_Rec(ntCSs, pathNodes, pathEdges, target-
   Node, outgoingEdge, position(lastNode)) (Algorithm 3)
27:       end if
28:     end for
29:   end if
30: end if
```

Algorithm 3 *Find_ntCS_Rec*

Description: Recursively call Find_Cycle
Input: *ntCSs*, *pathNodes*, *pathEdges*, *targetNode*, *outgoingEdge*, *nodePosition*
Output: *ntCSs*

1: Mark *outgoingEdge* as VISITED
2: Add *targetNode* to *pathNodes*
3: Put (*outgoingEdge*, *nodePosition*) as value in *pathEdges*
4: Call *Find_ntCS*(*ntCSs*, *pathNodes*, *pathEdges*, *targetNode*) (Algorithm 2)
5: Remove last occurrence of *targetNode* from *pathNodes*
6: Remove the *outgoingEdge* from *pathEdges*

Algorithm 4 *Check_ntCS*

Description: Check, if a part of the path is (a part of) a ntCS
Input: $ntCSs$, $pathNodes$, $pathEdges$
Output: true iff (part of) ntCS was found

1: $pathSize \leftarrow$ size of $pathNodes$
2: $lastNode \leftarrow$ last node in $pathNodes$
3: $lastPosition \leftarrow pathSize - 1$
4: $secondToLastPosition \leftarrow pathSize - 1$
5: $cycleLastNode \leftarrow$ the ntCS containing $lastNode$
6: $cycleSecondToLastNode \leftarrow$ ntCS containing node having index $pathSize - 2$
7: **if** $cycleSecondToLastNode \neq null \wedge cycleLastNode \neq null \wedge cycleSecondToLastNode = cycleLastNode$ **then**
8: Return false
9: **end if**
10: **for** $nodeIndex \leftarrow pathSize - 2$ **down to** 0 **do**
11: **if** $pathNodes[nodeIndex] = lastNode$ **then**
12: $secondToLastPosition \leftarrow nodeIndex$
13: Break
14: **end if**
15: **end for**
16: **if** $lastPosition - secondToLastPosition = 2$ **then**
17: Return false
18: **end if**
19: **if** $secondToLastPosition < pathSize - 1$ **then**
20: $reducedPath \leftarrow$ call $ComputeReducedPath(pathNodes, pathEdges, secondToLastPosition)$ (Algorithm 5)
21: **if** $reducedPath \neq null$ **then**
22: Create new ntCS $ntCS$
23: Add into $ntCS$ all nodes and edges of $reducedPath$
24: Call $SubCycle(ntCSs_of_wCC, ntCS)$ [1]
25: Call $MergeCycles(ntCSs_of_wCC)$ [1]
26: Return true
27: **end if**
28: **else if** $cycleLastNode \neq null$ **then**
29: Return $PartialCycle(ntCSs_of_wCC, pathNodes, lastNode)$ [1]
30: **end if**
31: Return false

Algorithm 5 *ComputeReducedPath*

Description: Remove a part of the potential ntCS path through searching over edges in the edges path
Input: *pathNodes*, *pathEdges*, *startPosition*
Output: *reducedPath*

1: $pathSize \leftarrow$ size of $pathNodes$
2: Create $workingPath$
3: **for** $nodePosition \leftarrow startPosition$ **to** $pathSize - 1$ **do**
4: Add the node $pathNodes[nodePosition]$ to $workingPath$
5: **end for**
6: **if** $workingPath[1] = pathNodes[pathSize - 2] \wedge workingPath[0] = pathNodes[pathSize - 1]$ **then**
{Start and end edge are reverse to each other.}
{No ntCS, reduced path is empty}
7: Return null
8: **else**
9: Create $reducedPath$
10: $workingPathSize \leftarrow$ size of $workingPath$
11: **for** $nodePosition \leftarrow 0$ **to** $workingPathSize - 1$ **do**
12: $testEdge \leftarrow$ linking ids of $workingPath[nodePosition + 1]$ and $workingPath[nodePosition]$, respectively
13: Add node $workingPath[nodePosition]$ to $reducedPath$
14: $endPosition \leftarrow$ the position of the target node of $testEdge$ from $pathEdges$
15: **if** $endPosition \neq null$ **then**
16: $endPosition \leftarrow endPosition - startPosition$
17: **end if**
18: **if** $endPosition \neq null \wedge endPosition > nodePosition$ **then**
19: $nodePosition \leftarrow endPosition$
20: **end if**
21: **end for**
22: **end if**
23: **Return** $reducedPath$

B Algorithms for Splitting wCCs

Algorithm 6 *Split_wCC_at_Node*

Description: Split a wCC at a ntCS node
Input: $ntCS_Node$
Output: $allWCCs$

1: Create $ntCS_Nodes$ set
2: Create $allEdges$ set
3: Add $ntCS_Node$ to $ntCS_Nodes$ set
4: Add all edges of $ntCS_Node$ to $allEdges$ set
5: **for all** $edge \in allEdges$ **do**
6:   **if** $edge$ is marked as not visited **then**
7:     Create $allNodes$ set
8:     Add $ntCS_Node$ to $allNodes$
9:     Call $Split_wCC(ntCS_Node, edge, allNodes, ntCS_Nodes)$ (Algorithm 7)
10:     $wcc \leftarrow create_Graph(ntCS_Nodes, allNodes)$ [1]
11:     **if** # nodes of $wcc > 0$ **then**
12:       Add wcc to $allWCCs$ list of all subgraphs
13:     **end if**
14:   **end if**
15: **end for**

Algorithm 7 *Split_wCC*

Description: Go over all nodes reachable by unvisited edges starting at $srcNode$

Input: $srcNode$, $edge$, $allNodes$, $ntCS_Nodes$

Output: $allNodes$, $ntCS_Nodes$

1: Mark $edge$ as visited
2: $reverseEdge \leftarrow$ get reverse edge of $edge$ from graph edges
3: **if** $reverseEdge \neq null$ **then**
4: Mark $reverseEdge$ as visited
5: **end if**
6: **if** end node of $edge \neq srcNode$ **then**
7: $node2 \leftarrow$ end node of $edge$
8: **else**
9: $node2 \leftarrow$ start node of $edge$
10: **end if**
11: **if** $node2 \in cyclesNodes$ **then**
12: Add $node2$ to $ntCS_Nodes$ set
13: Add $node2$ to $allNodes$ set
14: **else if** $node2$ is not marked **then**
15: Mark $node2$
16: Add $node2$ to $allNodes$ set
17: Add all edges of $node2$ to $allEdges$ set
18: **for all** $edge2 \in allEdges$ **do**
19: **if** $edge2$ is marked as not visited **then**
20: Call *Split_wCC*($node2$, $edge2$, $allNodes$, $ntCS_Nodes$) (Algorithm 7)
21: **end if**
22: **end for**
23: **end if**

C Algorithms for Detecting Trees and DAGs

Algorithm 8 *DetectDAGsTrees*

Description: Classify wCC (up-tree, down-tree, DAG)
Input: wCC
Output: return wCC classification (up-tree, down-tree, DAG)

1: **for all** $node \in$ {all nodes of wCC} **do**
2: **if** $node$ has in-degree 0 **then**
3: $countInDegreeZero \leftarrow countInDegreeZero + 1$
4: Add $node$ to $inDegreeZeroNodes$ ArrayList
5: **else if** $node$ has out-degree 0 **then**
6: $countOutDegreeZero \leftarrow countOutDegreeZero + 1$
7: Add $node$ to $outDegreeZeroNodes$ ArrayList
8: **else if** $node$ has only double edges **then**
9: Add $node$ to $potentialDoubleRoots$ ArrayList
10: **else if** $node$ has only double and outgoing edges **then**
11: Add $node$ to $potentialDownRoots$ ArrayList
12: **else if** $node$ has only double and incoming edges **then**
13: Add $node$ to $potentialUpRoots$ ArrayList
14: **end if**
15: **end for**
16: **if** $countInDegreeZero >= 2 \land countOutDegreeZero >= 2$ **then**
17: Mark $result$ as DAG
18: **else if** $countInDegreeZero < 2 \land countOutDegreeZero >= 2$ **then**
19: $result \leftarrow checkForDownTree(wCC, inDegreeZeroNodes, potentialDownRoots, potentialDoubleRoots)$ (Algorithm 9)
20: **else if** $countInDegreeZero >= 2 \land countOutDegreeZero < 2$ **then**
21: $result \leftarrow checkForUpTree(wCC, outDegreeZeroNodes, potentialUpRoots, potentialDoubleRoots)$ (Algorithm 11)
22: **else if** $countInDegreeZero < 2 \land countOutDegreeZero < 2$ **then**
23: $result \leftarrow classify_wCC(wCC, inDegreeZeroNodes, potentialDownRoots, outDegreeZeroNodes, potentialUpRoots, potentialDoubleRoots)$ (Algorithm 13)
24: **end if**
25: **if** $result = DAG$ **then**
26: Call $createDAG(wCC)$ [1]
27: **end if**

Algorithm 9 *checkForDownTree*

Description: Check, if a down-tree can be constructed
Input: *wCC*, *inDegreeZeroNodes*, *potentialDownRoots*, *potentialDoubleRoots*
Output: return wCC classification (up-tree, down-tree, DAG)

1: $counterPath \leftarrow counterPath + 1$
2: Create *pathNodes* list
3: **if** $countInDegreeZero == 1$ **then**
4: **for all** $node \in wCC$ **do**
5: Mark *node* as not visited with value -1
6: **end for**
7: $node \leftarrow$ first node of *inDegreeZeroNodes*
8: $root \leftarrow node$
9: Add *root* to *pathNodes*
10: $result \leftarrow checkForDownTreeDFS($*pathNodes*, *counterPath*, NULL, *root*$)$ (Algorithm 10)
11: **if** *result* **then**
12: Call $createDownTree(root, nodes)$ [1]
13: return *DOWN_TREE*
14: **else**
15: return *DAG*
16: **end if**
17: **else**
18: add *inDegreeZeroNodes* at end of *potentialOrderedRootsNodes*
19: add *potentialDownRoots* at end of *potentialOrderedRootsNodes*
20: add *potentialDoubleRoots* at end of *potentialOrderedRootsNodes*
21: **for all** $node \in potentialOrderedRootsNodes$ **do**
22: **for all** $node \in wCC$ **do**
23: Mark *node* as not visited with value -1
24: **end for**
25: Clear *pathNodes*
26: $root \leftarrow node$
27: Add *root* to *pathNodes*
28: $result \leftarrow checkForDownTreeDFS($*pathNodes*, *counterPath*, NULL, *root*$)$ (Algorithm 10)
29: **if** *result* **then**
30: Call $createDownTree(root, nodes)$ [1]
31: return *DOWN_TREE*
32: **end if**
33: **end for**
34: return *DAG*
35: **end if**

Algorithm 10 $checkForDownTreeDFS$

Description: Check, if a down-tree can be constructed
Input: $pathNodes$, $counterPath$, $parentNode$, $node$
Output: return wCC classification (up-tree, down-tree, DAG)

1: **if** $node$ is marked with $counterPath$ **then**
2: return DAG
3: **end if**
4: **if** size of $pathNodes > 1 \wedge countOneDirectionIncomingEdges(node, parentNode) > 0$ [1] **then**
5: return DAG
6: **end if**
7: Mark $node$ with $counterPath$
8: $result \leftarrow DOWN_TREE$
9: **for all** $outgoingEdge \in$ {outgoing edges of $node$} **do**
10: $targetNode \leftarrow$ target node of $outgoingEdge$
11: Add $targetNode$ to $pathNodes$
12: $first \leftarrow$ the first occurrence position (index) for last node in the path
13: $last \leftarrow$ the last occurrence position (index) for last node in the path
14: **if** $first > -1 \wedge last > -1 \wedge last - first = 2$ **then** {Ignore double edge}
15: Remove the last node from $pathNodes$
16: continue
17: **end if**
18: $result \leftarrow checkForDownTreeDFS(pathNodes, counterPath, node, targetNode)$ (Algorithm 10)
19: Remove the last node from $pathNodes$
20: **if** $result = DAG$ **then**
21: return DAG
22: **end if**
23: **end for**
24: return $DOWN_TREE$

Algorithm 11 *checkForUpTree*

Description: Check, if an up-tree can be constructed
Input: *wCC*, *outDegreeZeroNodes*, *potentialUpRoots*, *potentialDoubleRoots*
Output: return wCC classification (up-tree, down-tree, DAG)

```
1:  counterPath ← counterPath + 1
2:  Create pathNodes
3:  if countOutDegreeZero == 1 then
4:    for all node ∈ wCC do
5:      Mark node as not visited with value −1
6:    end for
7:    node ← first node of outDegreeZeroNodes
8:    root ← node
9:    Add root to pathNodes
10:   result ← checkForUpTreeDFS(pathNodes, counterPath, NULL, root)
      (Algorithm 12)
11:   if result then
12:     Call createUpTree(root, nodes) [1]
13:     return UP_TREE
14:   else
15:     return DAG
16:   end if
17: else
18:   add outDegreeZeroNodes at end of potentialOrderedRootsNodes
19:   add potentialUpRoots at end of potentialOrderedRootsNodes
20:   add potentialDoubleRoots at end of potentialOrderedRootsNodes
21:   for all node ∈ potentialOrderedRootsNodes do
22:     for all node ∈ wCC do
23:       Mark node as not visited with value −1
24:     end for
25:     Clear pathNodes
26:     root ← node
27:     Add root to pathNodes
28:     result ← checkForUpTreeDFS(pathNodes, counterPath, NULL,
        root) (Algorithm 12)
29:     if result then
30:       Call createUpTree(root, nodes) [1]
31:       return UP_TREE
32:     end if
33:   end for
34:   return DAG
35: end if
```

Algorithm 12 *checkForUpTreeDFS*

Description: Check, if an up-tree can be constructed
Input: *pathNodes*, *counterPath*, *parentNode*, *node*
Output: return wCC classification (up-tree, down-tree, DAG)

1: **if** *node* is marked with *counterPath* **then**
2: return DAG
3: **end if**
4: **if** size of $pathNodes > 1 \wedge countOneDirectionOutgoingEdges(node, parentNode) > 0$ [1] **then**
5: return DAG
6: **end if**
7: Mark *node* with *counterPath*
8: $result \leftarrow UP_TREE$
9: **for all** $incomingEdge \in \{$incoming edges of $node\}$ **do**
10: $sourceNode \leftarrow$ source node of $incomingEdge$
11: Add *sourceNode* to *pathNodes*
12: $first \leftarrow$ the first occurrence position (index) for last node in the path
13: $last \leftarrow$ the last occurrence position (index) for last node in the path
14: **if** $first > -1 \wedge last > -1 \wedge last - first = 2$ **then**
{Ignore double edge}
15: Remove the last node from *pathNodes*
16: continue
17: **end if**
18: $result \leftarrow checkForUpTreeDFS(pathNodes, counterPath, node, sourceNode)$ (Algorithm 12)
19: Remove the last node from *pathNodes*
20: **if** $result = DAG$ **then**
21: return DAG
22: **end if**
23: **end for**
24: return UP_TREE

Algorithm 13 $classify_wCC$

Description: Classify wCC (up-tree, down-tree, DAG)
Input: *wCC inDegreeZeroNodes*, *potentialDownRoots*, *outDegreeZeroNodes*, *potentialUpRoots*, *potentialDoubleRoots*
Output: return wCC classification (up-tree, down-tree, DAG)

1: *result* ← *checkForDownTree*(*wCC*, *inDegreeZeroNodes*, *potentialDownRoots*, *potentialDoubleRoots*) (Algorithm 9)
2: **if** *result* = *DAG* **then**
3:   *result* ← *checkForUpTree*(*wCC*, *outDegreeZeroNodes*, *potentialUpRoots*, *potentialDoubleRoots*) (Algorithm 11)
4: **end if**
5: return *result*