CARL VON OSSIETZKY
universität OLDENBURG


# BERICHTE

## AUS DEM DEPARTMENT FÜR INFORMATIK

der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften

Herausgeber: Die Professorinnen und Professoren
des Departments für Informatik


# General Graphs

Günther Stiege

Bericht

Nummer 02/07 - Juni 2007
ISSN 0946-2910

CARL VON OSSIETZKY universität OLDENBURG

# BERICHTE

**aus dem**

# DEPARTMENT FÜR INFORMATIK

**der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften**

Herausgeber: Die Professorinnen und Professoren
des Departments für Informatik

## General Graphs

**Günther Stiege**

Graphen und Netzwerke

**Bericht**

**BERICHT Nummer 02/07 - Juni 2007**

**ISSN 0946-2910**

*Author's address:*

Prof. em. Dr. Günther Stiege
Universität Oldenburg
Graphen und Netzwerke
D-26111 Oldenburg, Germany
*www-bvs.informatik.uni-oldenburg.de*

This report is available by www from `www-bvs.informatik.uni-oldenburg.de`

*Bibliographische Angaben:*
Stiege, Günther:
General Graphs / Stiege, Günther – Oldenburg: Universität Oldenburg, 2007
(Berichte aus dem Department für Informatik Nr. 02/07)

ISSN 0946-2910

© Günther Stiege, Oldenburg, 2007

# General Graphs

Günther Stiege
Universität Oldenburg
(June 2007)

**Abstract.** In general graphs undirected edges and directed arcs can be mixed freely. The report treats topics of algorithmic graph theory based on the concept of general graphs.

**Kurzfassung.** In allgemeinen Graphen können ungerichtete Kanten und gerichtete Bögen beliebig gemischt auftreten. Der Bericht behandelt Teile der algorithmischen Graphentheorie auf der Basis allgemeiner Graphen.

**Categories and Subject Descriptors:** E.1 [**Data Structures**]: Graphs and networks; G.2.2 [**Graph Theory**];

**General Terms:** Theory, algorithms

**Additional Keywords and Phrases:** General graph, graph decomposition, graph period.

## Contents

**1 Preface** **3**

**2 Definitions and Basic Facts** **4**
2.1 Definition of General Graphs . . . . . 4
2.2 Reorientations . . . . . 6

**3 Paths and Connected Components** **6**
3.1 Paths in General Graphs . . . . . 6
3.2 Reachability, Weak Components, Strong Components . . . . . 7
3.3 Acyclicity and Trees . . . . . 9
3.4 Partial Ordering and Level Numbering . . . . . 10
3.5 Component Classification . . . . . 13
3.6 Derived Graphs . . . . . 14

**4 Depth-First Search and Breadth-First Search 14**
4.1 Depth-First Search . . . 15
4.2 Depth-first search trees . . . 16
4.3 Finding weak components . . . 18
4.4 Finding strong components . . . 18
4.5 Breadth-first Search . . . 23

**5 The Biblock Decomposition 24**
5.1 Classes of closed a-paths and edge partitions . . . 24
5.2 The Biblock Decomposition of General Graphs . . . 25
5.3 Properties of the Biblock Decomposition . . . 28
5.3.1 Stopfree Kernel and Peripheral Trees . . . 28
5.3.2 Subcomponents and Internal Trees . . . 28
5.3.3 Biblocks . . . 29
5.4 Line and Vertex Classification . . . 29
5.5 The Biblock Graph . . . 30
5.6 Algorithms for Finding the Biblock Decomposition . . . 30
5.6.1 Determining the Peripheral Trees . . . 31
5.6.2 Determining the Structure of the Stopfree Kernel . . . 32
5.6.3 Efficiency . . . 35
5.7 Digraphs and Complete Orientations . . . 36
Remarks and Literature . . . 38

**6 Periodicity 39**
6.1 a-Period and f-Period . . . 39
6.2 Periodicity Classes . . . 41
6.3 An Algorithm for Finding the Period . . . 42
Remarks and Literature . . . 45

**7 Menger Structures 45**
7.1 Separating Sets and Disjoint Paths . . . 45
7.2 The Menger Theorems . . . 46
7.3 An Example . . . 50
7.4 Implementation and Efficiency of Menger Algorithms . . . 53
7.5 Extensions of the Menger Theorems . . . 55
7.6 The Structure of Menger Separating Sets . . . 57
7.7 An Algorithm for Finding the Menger Vertices . . . 59
Remarks and Literature . . . 62

**8 Higher Decompositions 63**
8.1 $k$-Reachability and $k$-Linereachability . . . 63
8.2 $k$-a-Lineconnectedness . . . 63
8.3 Simple Kernel . . . 67
8.4 $k$-a-Connectedness . . . 68
8.5 An Example . . . 72

8.6 Connectedness Defined by f-Paths . . . . . . . . . . . . . . . . . . . . . . 76
Remarks and Literature . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

**9 Finding Components of Higher Order 79**
9.1 Graph Decompositions . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
9.2 Decomposition into $k$-a-Components . . . . . . . . . . . . . . . . . . . . 80
9.3 Algorithm for Finding the $k$-a-Components . . . . . . . . . . . . . . . . . 81
9.4 Decomposition into $k$-a-Linecomponents . . . . . . . . . . . . . . . . . . 83
9.5 Algorithm for Finding the $k$-a-Linecomponents . . . . . . . . . . . . . . . 83
9.6 More Decompositions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
9.7 Complexity of Decomposition Finding . . . . . . . . . . . . . . . . . . . . 85
9.8 Dynamic Improvement of the RGB-Procedure . . . . . . . . . . . . . . . . 87
Remarks and Literature . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

**A Various Supplements 89**
A.1 Component Classification . . . . . . . . . . . . . . . . . . . . . . . . . 89
A.2 Derived Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
A.3 Biblock Decomposition . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

**List of Figures 101**

**List of Tables 103**

**References 104**

# 1 Preface

In the literature on graphs and graph algorithms a clear distinction between undirected graphs and directed graphs (digraphs) is made. In undirected graphs pairs of vertices are joined by undirected lines, called *edges*, whereas in digraphs directed lines, called *arcs*, join pairs of vertices. Mixed graphs are known but have not received much attention. Perhaps the best known and certainly surprising example is the Chinese postman problem which is solvable in polynomial time for undirected graphs and for digraphs but is NP-complete for mixed graphs [GareJ1979].

In undirected graphs as well as in digraphs the concept of "path" is a basic tool to define connectedness properties and prove results about these. It is an important concept also for treating other graph problems. Paths in undirected graphs are undirected and paths in digraphs are directed. However, it turns out that certain properties are best described, if undirected paths are also used with digraphs. A well known example are weakly connected components of digraphs. Sometimes undirected paths in digraphs are called semipaths [CharL1996].

Guided by this observation I propose the use of mixed graphs as the base of graph theory renaming them as *general graphs*. In general graph *lines* join pairs of vertices. Some lines are undirected (edges), the remaining are directed (arcs). Multiple lines and loops are allowed, too. For general graphs three kinds of paths are defined:

- a-paths (`any`).
  Arcs may be traversed in forward and backward direction.
- f-paths (`forward`).
  Arcs may be traversed in forward direction only.
- b-paths (`backward`).
  Arcs may be traversed in backward direction only.

Of course, in all paths edges may be traversed from one of its end points to the other.

When working on different problems in algorithmic graph theory the advantages of such an approach became gradually clear to me. Experiences in classroom lecturing at Universität Oldenburg enforced my conviction. With this motivation I wrote the textbook “Graphen und Graphalgorithmen” [Stie2006]. The present report is essentially a shortened version of the textbook, written to give a reference source to readers who do not speak German.

This report is organized as follows: Section 2 contains a formal definition of general graphs and same basic facts. In section 3 paths are introduced. Weak and strong components, trees and level numbers are treated. Section 4 is about depth-first search and bread-first search. Biblock decomposition, i.e. decomposition into biconnected components, is contained in section 5. Periods of weak and strong components are addressed in section 6. Theorems of Menger type and some consequences are examined in section 7. Finally, in section 8 higher decompositions of general graphs are introduced. A heuristics for finding these decompositions is presented in section 9.
Some examples have been put into the appendix.

Most of the results are well known. Some are new. Proof of all theorems, propositions, and lemmata can be found in [Stie2006], often without explicit reference in this report. A couple of proofs which I consider of interest have been repeated here.

# 2 Definitions and Basic Facts

## 2.1 Definition of General Graphs

### Graph Incidence Structure

Informally, in a general graph we have a set of *vertices* and a set *lines*. Both, vertices and lines bear unique identifiers. Every line *incides* with 2 vertices, its *incidence points*. The incidence points of a line need not be different. If they are equal the line is a *loop*. A vertex which is not incident with any line is called *isolated vertex*. Different lines may have the same incidence points. In this case we speak of *multiple lines*. Two vertices joined by line are called *neighbors* or *adjacent*.

We formalize these ideas in the following

**Definition 2.1** *Be $V$ and $L$ finite sets and $V \neq \emptyset$. A* graph incidence structure *over $V$* and *$L$ is a mapping*

$$\varphi : L \mapsto \mathcal{P}(V) \textit{ with } 1 \leq |\varphi(l)| \leq 2 \textit{ for all } l \in L$$

$\mathcal{P}(V)$ is the power set of $V$. $|\varphi(l)|$ is the cardinality of $\varphi(l)$. $\varphi$ is the incidence mapping.

### General Graph

Again informally: A line is either *undirected* and then called an *edge* or its is *directed* and then called an *arc*. The incidence points of a an edge are its *end points*. An arc starts at its *head* and ends at its *tail*. *Multiple edges* have identical end points. *Multiple arcs* point from the same head to the same tail. Figure 1 shows three multiple lines, none of which is multiple edge or multiple arc.

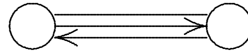

Figure 1: *Multiple lines*

Again we formalize

**Definition 2.2** *$G = (V, E, A, \varphi, \psi)$ is a* general graph *if*

1. *$\varphi$ is a graph incidence structure over $V$ and $L$.*
2. *$A \subseteq L$ and $E := L \setminus A$.*
3. *$\psi$ is a mapping*

$$\psi : A \mapsto V \times V \text{ with } \psi(a) = (v, w) \Rightarrow v, w \in \varphi(a) \text{ for all } a \in A$$

$E$ is the set of edges, $A$ the set of arcs. $\psi$ is the *orientation mapping*. If $\psi(a) = (v, w)$ then $v$ is the head and $w$ the tail of $a$. If $A = \emptyset$ then $G$ is called *undirected graph*. If $E = \emptyset$ then $G$ is called *directed graph* or *digraph*. As usual, a simple graph is an undirected graph without loops and multiple edges. $K_n$ is the complete simple graph with $n$ vertices.

### Subgraphs

We obtain a *subgraph* of a general graph by deleting some vertices together with all lines they are incident with. We may also delete additional lines. A formal definition can easily be derived from definitions 2.1 and 2.2 but is not of much use.

The terms *subgraph generated by a set of vertices* and *subgraph generated by a set of lines* have their usual meaning. The same is true for the term *generating subgraph* or *spanning subgraph*.

### Degree of a Vertex

The number of lines a vertex $v$ is incident with is called its *total degree*, denoted $\mathrm{td}(v)$. The *degree* $\mathrm{d}(v)$, the *indegree* $\mathrm{id}(v)$ and the *outdegree* $\mathrm{od}(v)$ count the number of edges, the number of incoming arcs and the number of outgoing arcs which are incident with $v$. Undirected loops are counted twice. A directed loop is an incoming arc as well as an outgoing arc. Therefore the following holds

$$2|E \cup A| = \sum_{v \in V} \mathrm{td}(v) \tag{1}$$

This equation implies the *handshaking lemma*: *In a general graph the number of vertices with odd total degree is even.*

### 2.2 Reorientations

All general graphs with identical graph incidence structure form an *orientation class*. If $G$ and $G'$ belong to the same orientation class $G'$ is called a *reorientation* of $G$. In a reorientation arcs change to edges, edges become arcs, and the direction of arcs is reversed. If $G'$ is a reorientation of $G$ and only edges have changed to arcs, then $G'$ is an *orientation* of $G$. If $G'$ is a digraph then it is called a *complete orientation* of $G$. If $G'$ results from $G$ only by changing arcs to edges, then $G'$ is called a *disorientation* of $G$. If $G'$ is an undirected graph then it is the *complete disorientation* of $G$. The complete disorientation of a general graph is identical for all graphs of the orientation class. The *inverse graph* of a general graph is its reorientation where all edges are unaltered and the directions of all arcs are inverted.

## 3 Paths and Connected Components

### 3.1 Paths in General Graphs

Paths are alternating sequences of vertices and lines:

$$v_0, l_1, v_1, \ldots, v_{n-1}, l_n, v_n \text{ with } n \geq 1 \tag{2}$$

where line $l_i$ is incident with vertices $v_{i-1}$ and $v_i$. Depending on which incidences are allowed we have 3 types of paths:

1. *f-path* (forward)
   Arcs may be traversed only from head to tail.

2. *b-path* (backward)
   Arcs may be traversed only from tail to head.

3. *a-path* (any)
   Arcs may be traversed in any direction.

   Of course, edges may always be traversed in any direction.

Let $v_0, l_1, v_1, \ldots, v_{n-1}, l_n, v_n$ be a path (of arbitrary type). $v_0$ is called the *starting vertex* and $v_n$ is called the *end vertex* of the path, all other vertices of the path are called *internal vertices*. $n$ is the *path length*. A path is *closed*, if $v_0 = v_n$, otherwise it is *open*. A path is called *line-simple*, if its lines are pairwise distinct. An open path is called *simple*, if all its vertices are pairwise distinct. A closed path is called simple, if only starting vertex and end vertex are identical.

A closed simple and line-simple a-path (f-path, b-path) is called *a-circuit (f-circuit, b-circuit)*. A general graph without a-circuits (f-circuits, b-circuits) is called *a-acyclic (f-acyclic, b-acyclic)*.

A *a-run (f-run, b-run)* is an infinite sequence $v_0, l_1, v_1, l_2, v_2 \ldots, v_{k-1}, l_k, v_k \ldots$ of vertices and lines, where for each $n$ $v_0, l_1, v_1, l_2, v_2 \ldots, v_{n-1}, l_n, v_n$ is an a-path (f-path, b-path).

The *support* of a path or a run is the subgraph of $G$ consisting of the traversed vertices and lines. In the sequel, we shall not always make a clear distinction between a path and its support. The intended meaning will be clear from the context.

The following proposition states well known and easy to prove facts on paths. With exception of the last two statements the paths may be of arbitrary type.

**Proposition 3.1**

1. *Every open path from $u$ to $v$ contains a line-simple path from $u$ to $v$.*
2. *Every line-simple open path from $u$ to $v$ contains a simple path from $u$ to $v$.*
3. *Every simple open path is line-simple.*
4. *Every closed path contains a closed simple path with the same starting vertex.*
5. *A closed simple path which is not line-simple must be of type $v, l, u, l, v$ with $v \neq u$.*
6. *Every simple closed path of length 1 or greater 2 is a circuit.*
7. *In an general graph without multiple lines every a-circuit is a loop or has length at least 3.*
8. *In a digraph, every simple closed f-path (b-path) is an f-circuit (b-circuit).*

## 3.2 Reachability, Weak Components, Strong Components

Let $G = (V, E, A, \varphi, \psi)$ be a general graph. Vertex $u$ is *a-reachable (f-reachable, b-reachable)* from vertex $u$ if there is an a-path (f-path, b-path) from $u$ to $v$. Since paths have positive length, no isolated vertex is reachable from any vertex. A vertex which is not f-reachable from itself is called a *vertex of no return*, otherwise it is a vertex of return.

a-reachability is an equivalence relation on the set of non-isolated vertices of $G$. Mutual f-reachability is an equivalence relation on the set of vertices of return. Mutual b-reachability is identical to mutual f-reachability. The subgraphs generated by the equivalence classes of a-reachability are called the *proper weakly connected components (proper weak components)* of $G$. Subgraphs consisting of one single isolated vertex are the *improper weak components* of $G$.

The subgraphs generated by the equivalence classes of mutual f-reachability are called the *proper strongly connected components (proper strong components)* of $G$. Subgraphs consisting of one single vertex of no return are the *improper strong components* of $G$. Every strong component is subgraph of a weak component.

**Note:** It would have been more consistent to call the weak components of a general graph *a-components* and the strong components *f-components*. However, weak component and strong component are well established names.

**Note:** For better intelligibility we shall use in the sequel the term 'weak component' for proper weak component and 'strong component' for proper strong component unless explicitly stated otherwise.

Bridges and cut points in general graphs are characterized by weak connectedness properties. A line of a general graph $G$ is a *bridge* if its deletion from the graph augments the number of (proper or improper) weak components. A vertex of $G$ is called *cut point* if its deletion together with all lines it is incident with augments the number of (proper or improper) weak components. The following proposition characterizes bridges by a-circuits and cut points by a-paths.

**Proposition 3.2** *1. A line $l$ of a general graph is a bridge if and only if there is no a-circuit which traverses it. A vertex $v$ of a general graph is a cut point if and only if there are vertices $u$ and $w$ distinct from $v$ and from each other such that every a-path from $u$ to $w$ passes trough $v$ and at least one such path exists*

Every non-isolated vertex and every line belongs to a unique weak component. Every edge and every loop belong to a unique strong component. What do proper weak components without strong components look like? The answer is given by the following theorem.

**Theorem 3.1** *A general graph has no strong components if and only if it is a f-acyclic digraph.*

Theorem 3.1 justifies the following definition: A general graph without strong components is called *dag (directed acyclic graph)*. In a weak component which contains strong components there may be arcs not belonging to any strong component. The subgraph generated by these arcs is called the *external dag* of the weak component. It is not necessarily weakly connected. It contains all vertices of no return. If the external dag is not empty, every strong component will contain vertices which also belong to the external dag. We call these vertices *weak attachment points* of the weak component. In the external dag, both head and tail of an arc may be weak attachment points. A strong component may have more than one weak attachment point but a weak attachment point belongs to only one strong component.

We call the decomposition of a general graph into weak components and isolated vertices its *weak decomposition*. The decomposition of a weakly connected graph into its strong components and its external dag is called its *strong decomposition*.

Whereas in the external dag all lines are arcs, in a strong component edges and arcs may be present in any mixture. Arcs of a strong component always lie on a f-circuit:

**Proposition 3.3** *Every arc of a strong component lies on a f-circuit.*

The following theorem is an immediate consequence of proposition 3.3.

**Theorem 3.2** *A strong component is f-acyclic if and only if it is undirected and a-acyclic.*

An edge always belongs to a strong component, but need not necessarily lie on a f-circuit. However:

**Proposition 3.4** *An edge is traversed by an f-circuit if and only if it lies on an a-circuit which is completely contained in the strong component.*

The proof uses proposition 3.8 of subsection 3.4.

## 3.3 Acyclicity and Trees

The following proposition gives well known estimates for general graphs which are weakly connected or which are a-acyclic.

**Proposition 3.5** *1. Every weakly connected general graph $G = (V, E, A, \varphi, \psi)$ has at least $|V| - 1$ lines.*
*2. Every a-acyclic general graph $G = (V, E, A, \varphi, \psi)$ has at most $|V| - 1$ lines.*

A weakly connected a-acyclic general graph is called *a-tree*. A general graph is called *a-forest* if all its weak components are a-trees. Undirected a-trees are also called *free trees*. The following theorem summarizes characterizing properties of a-trees.

**Theorem 3.3** *Let $G = (V, E, A\varphi, \psi)$ be a general graph. The following properties are equivalent.*

1. *$G$ is an a-tree.*
2. *$G$ is loopfree and any two distinct vertices are joined by a uniquely determined simple a-path.*
3. *$G$ is loopfree and weakly connected. The deletion of a line destroys weak connectedness.*
4. *$G$ is weakly connected and has $|V| - 1$ lines.*
5. *$G$ is a-acyclic and has $|V| - 1$ lines.*
6. *$G$ is a-acyclic. Adding a line destroys a-acyclicity.*

How should an f-tree be defined? It should be an a-tree. However, if we required also strong connectedness, then in general a-acyclicity would be lost. Therefore, we define an *f-tree* as an a-tree with a vertex from which all other vertices are f-reachable. Such a vertex is called a *root*. A root need not be unique. A *b-tree* is an a-tree with a vertex from which all other vertices are b-reachable. For f-trees (b-trees) which are digraphs holds the following proposition.

**Proposition 3.6** *1. In a digraph which is an f-tree there is exactly one vertex from which all other vertices are f-reachable. This vertex has no predecessor, all other vertices have exactly one predecessor.*

*2. In a digraph which is a b-tree there is exactly one vertex from which all other vertices are b-reachable. This vertex has no successor, all other vertices have exactly one successor.*

The following theorem characterizes digraphs as f-trees.

**Theorem 3.4** *A digraph $G$ is a f-tree if and only if the following three conditions hold:*

1. *$G$ is f-acyclic.*
2. *Every vertex has at most one predecessor.*
3. *There is a vertex from which all other vertices are f-reachable.*

## 3.4 Partial Ordering and Level Numbering

In a dag, if $v$ is f-reachable from $u$ then $u$ is not f-reachable from $v$, for there are no strong components. That means that in a dag f-reachability is a strict partial ordering on the set of vertices.

For the vertices of a strong component f-reachability is not a partial ordering but an equivalence relation. However, there is a partial ordering relating the vertices of a strong component and the vertices not belonging to this component. To that end, be $C(u)$ the proper or improper strong component to which vertex $u$ belongs. Then the following lemma holds:

**Lemma 3.1** *Be $C(u)$ and $C(v)$ two distinct proper or improper strong components of a general graph. Be there an f-path from $u$ to $v$. Then for every $u' \in C(u)$ and every $v' \in C(v)$ there is an f-path from $u'$ to $v'$ but none from $v'$ to $u'$.*

Lemma 3.1 allows us to define a strict partial ordering $\prec$ on the set of (proper or improper) strong components of a general graph. We use $\prec$ to assign recursively a *level number* $\mathrm{lv}(C(u))$ to each strong component.

1. If $C(u)$ is an initial element with respect to $\prec$ then $\mathrm{lv}(C(u)) := 0$.
2. Otherwise we take the maximum level number of the antecedents of $C(u)$ with respect to $\prec$ and add 1:

$$\mathrm{lv}(C(u)) := \left( \max_{C(v) \prec C(u)} \mathrm{lv}(C(v)) \right) + 1$$

We extend level numbering to the vertices assigning each vertex the level number of the strong component it belongs to.

When traversing an a-path (f-path, b-path) the level number of the vertices the path passes through may change. The is the case if and only if the line joining the two vertices is an arc of the external dag. If the arc is traversed in f-direction the level number grows, if the arc is traversed in b-direction the level number diminishes. We summarize this in the following proposition:

**Proposition 3.7** *On a path, the level number of two successive vertices changes if and only if the line traversed is an arc of the external dag. It grows if the arc is traversed in f-direction, it diminishes if the arc is traversed in b-direction. Especially, along an f-path level numbers cannot become smaller, along a b-path level numbers cannot become larger.*

Level numbers and partial ordering help to decide edge orientability. I. e. we want to find out whether an edge can be transformed into an arc without changing the correspondent strong component.

**Proposition 3.8** *An edge $e$ of a general graph is orientable if and only if it lies on a a-circuit, which is contained entirely in the strong corresponding component .*

**Proof:** Let $H$ be the strong component containing $e$. If $e$ is a bridge in $H$ no a-circuit contained in $H$ will pass through $e$ and no orientation of $e$ will maintain the the strong connectedness of $H$. a-circuits containing lines outside from $H$ may pass through $e$ but cannot restore the strong connectedness of $H$. If $e$ is part of an a-circuit in $H$ it is not a bridge and $H - e$ is weakly connected. We decompose $H - e$ into strong components and the external dag. If one single strong component results $e$ can be oriented arbitrarily. Otherwise, the decomposition must be such that adding $e$ a strongly connected graph results. This requires that there is exactly one minimal strong (proper or improper) component (with level number 0) and exactly one maximal (proper or improper) strong component. In addition, in $H$ $e$ must join a vertex $u$ in the minimal component to a vertex $v$ in the maximal component. Orienting $e$ from $v$ to $u$ leaves $H$ strongly connected. The reverse orientation destroys the strong connectedness. □

Level numbering also allows us to define vertices with root property. In subsection 3.3 we introduced the concept of a root in context with f-trees. We now extend the definition: A vertex $u$ is called a *root vertex* if there is an f-path to every other vertex of the graph.

**Proposition 3.9** *Be $G(V, E, A, \varphi, \psi)$ a general graph.*

1. *If there is a root vertex in $G$ then $G$ is weakly connected.*
2. *If $u$ is a root vertex of $G$ then all $u' \in C(u)$ are root vertices of $G$.*
3. *Vertex $u$ is a root vertex of $G$ if and only if $lv(C(u)) = 0$ and $lv(C(v)) > 0$ for all $C(v) \neq C(u)$.*

What do general graphs with root vertices look like? To avoid trivial cases we assume $L \neq \emptyset$. If a general graph with a root vertex is a-acyclic then it is an a-tree. All simple f-paths from one vertex to another are uniquely determined. Non-simple f-paths exist only if the graph contains edges. They only way to generate a non-simple f-path is by multiply traversing edges. The f-path is not line-simple.

If the general graph is a-cyclic, i.e. if it contains an a-circuit, then to each root vertex $u$ there exist vertices which are f-reachable from $u$ along (at least) two different line-simple paths:

**Proposition 3.10** *Let $G$ be a general graph and $u$ a root vertex.*

1. *If an a-circuit is contained completely in a strong component of $G$ then there is a vertex $u'$, which is f-reachable from $u$ along two different line-simple paths. One of these may have length 0.*

2. *If there is an a-circuit in $G$ traversing more than one (proper or improper) strong component then there is a vertex $u'$ distinct from $u$ which is f-reachable from $u$ along two different simple f-paths.*

**Proof:** 1. The strong component containing an a-circuit also contains an f-circuit (proposition 3.4). Let $x_0, l_1, x_1, l_2, \ldots, x_{n-1}, l_n, x_n = x_0$ be an f-circuit. If the circuit passes through $u$ we choose $u' = u$. Else a simple f-path from $u$ to the f-circuit is chosen. Let $u'$ be the first vertex on the path which belongs also to the f-circuit. The f-path from $u$ to $u'$ is the first path, the second is obtained adding the f-circuit to the first path. Both paths are line-simple.

2. We assume that there is an a-circuit traversing more than one (proper or improper) strong components. The circuit must have length at least 2 and not all its vertices can have the same level number. All its lines are pairwise distinct. Let $u'$ be a vertex on the a-circuit with highest level number. $\mathrm{lv}(u) = 0$ implies $u \neq u'$. Following the a-circuit from $u'$ in reverse direction, there is a first vertex $w$ with a smaller level number. Going from $u'$ in positive direction, there is also a first vertex with smaller level number. Be $w'$ that vertex. $w = w'$ is possible. The successor of $w$ on the a-circuit be $w_1$, the line joining $w$ and $w_1$ be $l_1$. In the same way $w_2$ is the predecessor of $w'$ on the a-circuit and $l_2$ the line joining $w'$ to $w_2$. The cases $u' = w_1$ and $u' = w_2$ are possible.
$u', w_1$ and $w_2$ have the same (highest) level number and belong to the same strong component which is improper if all three are equal. Now we choose simple f-paths (possibly of length 0) from $u$ to $w$ and from $u$ to $w'$. According to proposition 3.7 all vertices on these paths have level number less then $\mathrm{lv}(u')$ and hence are distinct from all vertices with this level number. We add $l_1$ to the first path and $l_2$ to the second. That is possible since both lines are arcs pointing from the vertex with lower level number to the vertex with higher level number. The result are simple f-paths from $u$ to $w_1$ and from $u$ to $w_2$. The paths differ at least in lines $l_1$ and $l_2$. If $u' \neq w_1$ we take in $C(u')$ a simple f-path from $w_1$ to $u'$ and add it to the first path. If necessary, the second path is extended in the same way. Finally we obtain two simple f-paths from $u$ to $u'$ which differ at least in one line. □

The following example shows that sometimes path length 0 or non-simple paths must be considered.

**Example 3.1** Figure 2 shows a general graph consisting of two weak components. Each of them contains a strong component with f-circuit. The only root vertex in the first weak component is $u$. The only vertex f-reachable from $u$ along two different line-simple paths is $u$ itself. One of these paths has length 0, the other is a loop. In the second weak component there are root vertices $u_1$ and $u_2$. The only vertex f-reachable from them along two different line-simple path is $v$. One of the paths is not simple. □

Of course, in a graph where all a-circuits are completely contained in strong component there may still exist distinct simple f-path from a root to a different vertex.

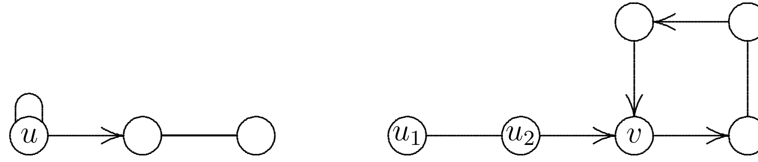


Figure 2: *f-paths from root vertices*

## 3.5 Component Classification

In subsection 3.2 we found the decomposition of a general graph as shown in table 1.

| Level | Component |
|---|---|
| 1 | general graph |
| 2 | improper weak component |
| 2 | proper weak component |
| 3 | strong component |
| 3 | external dag |

Table 1: *Decomposition into weak and strong components*

We now refine this classification of weak components using as criteria the existence of a-circuits and the existence of strong components.

**A. A-acyclic weak component:**
It is an a-tree, i. e there is a uniquely determined simple a-path joining any two distinct vertices.
*1. Without strong components*
It is a dag. If a root vertex exists it is unique and the weak component is an f-tree.
*2. With strong components*
The strong components consist of edges only. As subgraphs they are undirected free trees. If root vertices exist, each of them is root of an f-tree.

**B. A-cyclic weak component (an a-circuit exists):**
*1. Without strong components*
It is a dag but not an a-tree. If a root vertex exists it is uniquely determined. Then at least one other vertex exists which is f-reachable from the root along two different simple f-paths (proposition 3.10).
*2. With strong components*
Each of the strong components is either f-acyclic or contains an f-circuit. Each f-acyclic strong component is a-acyclic and undirected (theorem 3.1). We distinguish two cases:
2.a. All strong components are f-acyclic. Then every a-circuit contains vertices of different level numbers. If root vertices exist, from each of them exist two different simple f-paths to a suitable chosen vertex with positive level number (proposition 3.10).
2.b. In the general case there are strong components with f-circuits. These can be

decomposed further, see section 5. If there are root vertices from each of them a vertex can be reached along two different line-simple f-paths (proposition 3.10). There may be a-circuits traversing more than one strong component.

**Examples and Algorithms** The decomposition of a general graph into the various types of weak and strong components is illustrated in example A.1, page 89. Algorithms for finding weak and strong components together with the corresponding external dags, weak attachment points, and level numbers are presented in subsections 4.3 and 4.4.

### 3.6 Derived Graphs

It is often useful to emphasize certain structural properties of a given general graph, hiding at the same time unnecessary details. If this is done creating a new general graph we have a *derived graph*. Here we describe the condensed digraph and the component graph. Others will be introduced in the following sections.

**Condensed Digraph.** Be $G$ a general graph. We consider the proper and improper strong components of $G$ as vertices of a new digraph. An arc is drawn from $C(u)$ to $C(v)$ if $C(u) \neq C(v)$ and if there is an arc from a $u' \in C(U)$ to a $v' \in C(v)$. At most one such arc is drawn. The new digraph is called the *condensed digraph of* $G$, written $\text{cond}(G)$. It is also called the *reduced digraph* of $G$. The condensed digraph is a dag and there is a f-path (b-path, a-path) from $v_i \in C_i$ to $v_j \in C_j$ with $i \neq j$, if and only if there is a f-path (b-path, a-path) from $C_i$ to $C_j$ in $\text{cond}(G)$. $\text{cond}(G)$ has no multiple arcs. See example A.2, page 93.

**Component Graph.** The condensed graph reflects important properties of a general graph. However, if the embedding of the strong components into the external dag is needed with more detail, it is preferable to use the *component graph*. To construct it we start with the external dag with all its arcs and vertices, including the weak attachment points. An additional vertex is introduced for each (proper) strong component. Each new vertex is joined to each of its weak attachment points by an undirected line. This is *not an edge of the original graph*. It is therefore called a *metaedge*.

The component graph is f-acyclic, too. It may have multiple arcs, namely those of the external dag. In the original graph there is a f-path (b-path, a-path) from $u$ to $v$ ($C(u) \neq C(v)$), if and only if in the component graph there is such path from $C(u)$ to $C(v)$. See example A.3, page 93.

## 4 Depth-First Search and Breadth-First Search

Depth-first search and breadth-first search are the most important ways to traverse a general graph.

## 4.1 Depth-First Search

Be $G(V, E, A, \varphi, \psi)$ a general graph and $v$ one of its vertices. Table 2 shows procedure *f-DFS* defining *f-depth-first search with initial vertex v*. Replacing in line 4 the outgoing

***f-DFS(v)***

```
1   if (v marked) return;
2   mark v;
3   set v active;
4   for (all edges and all outgoing arcs l incident with v)
5      { if (l not marked)
6           { mark l;
7             f-DFS(otherend(l, v));
8      }    }
9   set v inactive;
10  return;
```

Table 2: *Procedure f-DFS for f-depth-first search in a general graph*

arcs by the incoming arcs we get *b-DFS*, the procedure defining *b-depth-first search with initial vertex v*. If both, outgoing and incoming arcs are considered we obtain procedure *a-DFS* defining *a-depth-first search with initial vertex v*.

Given a line $l$ and a vertex $v$ incident with $l$, $otherend(l, v)$ yields the other vertex incident with $l$. *f-DFS* uses marks for vertices and marks for lines. We say that a vertex is *visited* if in one of the recursive calls of *f-DFS* line 1 is executed with this vertex. A vertex may be not visited at all, visited exactly once or visited several times. Only at the first visit and only if at that time the vertex is not marked the vertex is *activated* and the processing of its edges and outgoing arcs (lines 4 - 8) takes place. When the line processing is completed the vertex is reset to *inactive*. Normally, after setting the vertex active and/or before resetting it to inactive some additional and problem depending processing is done.

**Remark 4.1** It is important to remember that in line 4 the order in which the lines are processed is not known. Any exhausting ordering is allowed. Which ordering really happens depends on the concrete graph data representation. Each concrete execution of *f-DFS* is a *f-depth-first search run* with initial vertex $v$. For b-depth-first search and a-depth-first search analog considerations are valid. □

To traverse a graph completely we embed *f-DFS* in an algorithm which acts as a frame. Table 3 shows this algorithm.
b-DFSframe and a-DFSframe result from pertinent changes in line 3. It is assumed that the graph to be traversed is known to f-DFSframe and that $V$ is the set of its vertices. Initially all vertices are unmarked and inactive. All lines are initially unmarked, too. At the end all vertices and all lines are marked.

The following theorem is of central importance.

**f-DFSframe**

```
1    for (for all vertices v in V)
2       { if (v not marked)
3            { f-DFS(v);
4       }    }
```

Table 3: *Frame for a complete f-depth-first search in a general graph*

**Theorem 4.1** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $v \in V$. If initially all vertices are unmarked f-DFS($v$) visits vertex $v$ and all vertices f-reachable from $v$. No other vertex is visited.*

What happens when at the moment *f-DFS*($v$) is called there are marked vertices? The following proposition answers that question.

**Proposition 4.1** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $v$ an unmarked vertex. Then with a call of f-DFS($v$) exactly those vertices $u$ will be visited, for which at calling time an f-path from $v$ to $u$ without marked vertices exists.*

Depth-first search is an efficient algorithm as the following proposition shows.

**Proposition 4.2** *f-Depth-first search in a general graph needs $O(|V| + |E| + |A|)$ time steps.*

If properly adjusted, theorem 4.1 and propositions 4.1 and 4.2 are valid also for b-depth-first search and a-depth-first search.

## 4.2 Depth-first search trees

During a run of a f-depth-first search (b-depth-first-search, a-depth-first search) characteristic properties of lines and certain trees result. The lines along which a f-depth-first search proceeds from a vertex to the next unmarked vertex are called *tree arcs*. We use this name independent of the real nature of the line (edge, arc) and independent of the direction the arc is traversed (in the case of b-depth-first search or a-depth-first search). With respect to the tail of a tree arc it is also called *entry arc* since via this line the vertex is visited for the first time and processing starts. We have the following lemma.

**Lemma 4.1** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $v \in V$. If initially all vertices are unmarked then the tree arcs of a run of f-DFS($v$) (b-DFS($v$), a-DFS($v$)) form a directed f-tree with root $v$.*

We now start with a completely unmarked general graph. A call to *f-DFS*($v$) with a first arbitrary vertex (for instance from a vertex list) yields a directed f-tree. If vertices no belonging to the tree remain, these generate a subgraph. We start *f-DFS*($v$) again with a vertex $v$ of the subgraph and obtain again a directed f-tree. Proceeding this way until no unmarked vertex remains we get a decomposition of the set of vertices into disjoint sets corresponding to the directed trees. The set of tree arcs is decomposed the same way. The directed f-trees and the directed f-forest thus obtained are called the *depth-first search trees* respectively the *depth-first search forest corresponding to the run of f-DFS*($v$). Obviously, the results are valid also for b-depth-first search and a-depth-first search.

A line with which a run of f-depth-first search (b-depth-first search, a-depth-first search) passes from a vertex to a marked and active vertex is call a *back arc.* Backward arcs are closely related to circuits.

**Lemma 4.2** *In a run of a f-depth-first search (b-depth-first search, a-depth-first search) every back arc closes a f-circuit (b-circuit, a-circuit) through head and tail of the arc.*

In a run of f-depth-first search an arc which leads form a vertex to a marked inactive vertex is called *forward arc* if the second vertex belongs to the same depth-first search tree. It is called *cross arc* if it belongs to a different depths-first search tree. The definition easily extends to b-depth-first search and a-depth-first search.

***Note:*** *Our definition for forward arc and cross arc differs from the usual.* See for instance Cormen/Leiserson/Rivest [CormLR1990]. □

We call the digraph consisting of tree arcs, back arcs, forward arcs, and cross arcs the *run digraph* of the general graph corresponding to the f-depth-first search (b-depth-first search, a-depth-first search) run.

**Lemma 4.3** *An edge is never a forward arc or a cross arc.*

**Corollary.** *A-depth-first search yields no forward arcs and no cross arcs.*

**Example 4.1** This example shows that it depends on the run of a f-depth-first search whether a f-circuit is found according to lemma 4.2 or not. We consider the graph of figure 3 and start a f-depth-first search in $v_0$. If in $v_2$ the arc traversed before the edge,

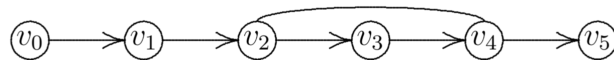


Figure 3: *F-depth-first search does not localize every f-circuit*

then in $v_4$ the edge is classified as back arc and the f-circuit is discovered. However, if in $v_2$ the edge is traversed before the arc then the depth-first search proceeds up to vertex $v_5$, then returns to $v_2$ , and finally processes the arc from $v_3$ to $v_4$ classifying it as forward arc. □

The case of example 4.1 cannot occur with a-depth-first search or with digraphs since the following theorem holds.

**Theorem 4.2** *1. A general graph contains a a-circuit if and only if every run of a a-depth-first search finds a back arc.*

*2. A digraph contains a f-circuit if and inly if every run of a f-depth-first search (b-depth-first search) finds a back arc.*

For strongly connected general graphs we have the following result.

**Theorem 4.3** *Let $G$ be a strongly connected general graph.*

1. *No run of a f-depth-first search (b-depth-first search) yields a cross arc.*
2. *There exists in $G$ a a-circuit if and only if there is a f-circuit in $G$. This is the case if and only if every run of a f-depth-first search (b-depth-first search) finds a marked vertex.*

## 4.3 Finding weak components

Important properties of depth-first search are run dependent. Other properties of at least the same importance are not. First examples are theorems 4.2 and 4.3. Finding the weak and strong decompositions of a general graph are other very important examples.

Weak components can be found easily by direct application of a-depth-first search. Table 4 shows algorithm WCOMP. It processes the set of vertices and creates a new (proper or improper) weak component $C$ for every unmarked vertex $v$. If the component is proper the recursive procedure $WCP(v, C)$ is called. Since a-reachability is the same relation as mutual a-reachability all vertices belonging to the same weak component as $v$ will be visited and only these. Thus $WCP(v, C)$ finds all vertices and all lines belonging to the component. Monitoring the existence of back arcs in line 2 determines whether a a-cycle exists. (Theorem 4.2).

What can we say if a weak component found is a-acyclic? From proposition 3.3, page 9, follows that is has no strong components at all or these consist only of edges. In the latter case strong components are a-acyclic and undirected, as subgraphs they are free trees. The general graph as a whole is a a-tree. In general, it is not a f-tree. In subsection 4.4 we will learn methods to classify the arcs of the general graph and to determine whether it is a f-tree. A a-acyclic proper weak component consisting of edges only is strongly connected. It is a free tree.

## 4.4 Finding strong components

In this subsection we consider only weakly connected general graphs. If necessary, we apply algorithm WCOMP first. Strong components are harder to find than weak components. Our approach uses the strict partial ordering $\prec$ of vertices introduced in subsection 3.4, page 10. In doing so, it may not come as a surprise that the problem of determining

```
WCOMP
          /* Initially all vertices and all lines are umarked */
1    for (all vertices v in V)
2        { if (v unmarked)
3             { create new proper and a-acyclic weak component C;
4               if (v isolated vertex)
5                   { tag C as improper weak component;
6                   }
7               else
8                   { tag C as a-acyclic;
9                     WCP(v, C);
10       }    }    }

WCP(v,C)
1    if (v marked)
2        { tag C as a-cyclic;
3          return;
4        }
5    add v to C;
6    for (alle lines l incident with v)
7        { if (l unmarked)
8             { mark l;
9               add l to C;
10              WCP(otherend(l, v), C);
11       }    }
12   return;
```

Table 4: *Algorithm WCOMP for finding weak component*

the strong components is intimately related to the problem of topologically sorting $\prec$, i.e. embedding consistently the partial order $\prec$ into a strict total order $<$.

The first step of the solution is algorithm PTOPSORT depicted in table 5. PTOPSORT uses f-depth-first search and a stack. A vertex is pushed onto the stack *after* it has been processed by depth-first search. The result of a run of PTOPSORT is that a set of essential vertices is ordered within the stack in topological order with respect to $\prec$. To this end, for each strong component the first vertex encountered by a f-depth-first search run is called entry point of the strong component corresponding to the run. We have the following proposition.

**Proposition 4.3** *In a run of PTOPSORT let $u$ be a vertex of no return or the entry point of a strong component. If there is a f-path from $u$ to a vertex $v$ but none from $v$ to $u$ then $u$ lies nearer to the top of the stack than $v$.*

```
PTOPSORT
          /* Initially all vertices are unmarked. */
          /* Initially the stack is empty. */

1   for (all vertices v in V)
2       { if (v unmarked)
3             PTPS(v);
4       };

PTPS(v)
1   if (v marked) return;
2   mark v;
3   for (all edges and all outgoing arcs l of v)    /* rekursive call */
4       { if (l unmarked)
5             { mark l;
6               PTPS(otherend(l, v));
7       }     };
8   push (stack, v);
```

Table 5: *Algorithm PTOPSORT for topological presorting a general graph*

**Proof:** 1. PTOPSORT visits $v$ before $u$ (first visit): Since $u$ is not reachable from $v$ the processing of $v$ will be finished, when the processing of $u$ begins. Hence $u$ will be pushed onto the stack at a later moment than $v$.
2. PTOPSORT visits $u$ after $v$ (first visit): While processing $u$, the case that $v$ is not reached and thus pushed onto the stack after $u$ occurs if and only if at the moment processing of $u$ begins all f-paths from $u$ to $v$ are blocked by at least one marked vertex (proposition 4.1). In this case, let a simple f-path from $u$ to $v$ be given and let $v_m$ be the last marked vertex on this path. Since $v$ is not marked $v_m$ must still be active. That means that there is a f-path from $v_m$ to $u$ and from $u$ to $v_m$. $v_m$ and $u$ belong to the same strong component. Therefore $u$ can neither be a vertex of no return nor the entry vertex of a strong component. □

**Remark 4.2** If PTOPSORT is applied to a dag one is almost done. There are no strong components and the linear order in the stack is a topological sorting of the vertices with respect to $\prec$. Solely the label numbering is still missing. □

The second and final step to find the strong components and to obtain a topological sorting of the vertices is algorithm STRONGCOMP. It is shown in table 7, page 22. It calls procedure *STRCP* shown in table 6. Applying STRONGCOMP to a weakly connected general graph needs a previous run of PTOPSORT with the graph as input. The results of applying STRONGCOMP to a general graph are formulated as the following theorem.

```
STRCP(v,SC)
1   if (v marked)
2      { classify SC as f-cyclic;
3        return;
4      }
5   mark v;
6   add v to SC;
7   add v to queue;
8   for (all edges and all incoming arcs l of v)
9      { if (l uncolored)
10           { color l blue;
11             add l to SC;
12             STRCP(otherend(l, v), SC);
13     }     }
```

Table 6: *Procedure STRCP for finding the strong components of a general graph*

**Theorem 4.4** *Let $G$ be a weakly connected general graph which has been processed by* PTOPSORT. *Then the application of* STRONGCOMP *to $G$ yields the following:*

1. *The algorithm determines all vertices of no return. It also determines all strong components with their vertices, edges, and arcs. For each strong component it is checked whether it is f-acyclic or f-cyclic. The result is recorded.*

2. *For each strong component all its weak attachment points are identified.*

3. *All arcs of the external dag are found.*

4. *The correct level number is assigned to all vertices of no return and to all strong components.*

5. *In a queue a topological sorting of the vertices of $G$ is constructed*

**Proof:** The complete proof is provided in [Stie2006].

**STRONGCOMP**

```
        /* Initially all vertices are unmarked and all lines are uncolored. */
        /* Initially all vertices and all strong components have level number 0 */
        /* The vertices are stored in stack in the order imposed by PTOPSORT. /*
1   v = pop(stack);
2   while (v ≠ NULL)    /* Process vertices from stack */
3     {if (v unmarked)
4        {if (v has no edges and no uncolored incoming arc)
5           {classify v as vertex of no return;
6            add v to queue;
7            for (all incoming arcs a of v)
8              { if (lv(otherend(a, v)) >= lv(v))  lv(v) = lv(otherend(a, v)) + 1 ;
9              }
10           for (all outgoing arcs a of v)
11             { classify a as external dag arc;
12               colour a yellow;
13          }    }
14       else
15          {create new strong component SC;
16           STRCP(v, SC);
17           for (all vertices v in SC)
18             {  if (v has yellow or uncolored arcs)
19                  {classify v as weak attachment point of SC;
20                   for (all yellow incoming arcs a of v)
21                     { if (lv(otherend(a, v)) >= lv(v))
22                           lv(v) = lv(otherend(a, v)) + 1 ;
23                     }
24                   if (lv(v) > lv(SC))  lv(SC) = lv(v);
25                   for (all uncolored outgoing arcs a of v)
26                     { classify a as external dag arc;
27                       color a yellow;
28             }     }    }
29           for (all vertices v in SC)  lv(v) = lv(SC);
30       }    }
31     v = pop(stack);
32    }
```

Table 7: *Algorithm STRONGCOMP for finding the strong components of a general graph*

## 4.5 Breadth-first Search

Table 8 shows the algorithm for f-breadth-first search in a general graph. With *b-breadth-*

```
f-BFS(v)
1   enqueue(v);
2   mark v;
3   while (queue not empty)
4      { u = dequeue;
5        for (all edges and outgoing arcs l of u)
6           { if (l marked) return;
7             mark l;
8             w = otherend(l, u);
9             if (not marked w)
10               { enqueue(w);
11                 mark w;
12     }    }    };
```

Table 8: *Procedure f-BFS for f-breadth-first search in a general graph*

*first search*, in line 5 we consider edges and incoming arcs. With *a-breadth-first search* we consider edges, incoming arcs, and outgoing arcs. Like depth-first search we imbed f-BFS in an algorithm which acts as a frame. Table 9 shows this algorithm. Again like

```
f-BFSframe
         /* Initially all vertices and all lines are unmarked */
1   for (all vertices v)
2      { if (v unmarked)
3           { f-BFS(v);
4      }    };
```

Table 9: *Frame for complete f-breadth-first search in a general graph*

depth-first search, breadth-first search uses markings and starts with initially unmarked vertices and lines. However, procedure *f-BFS* is not recursive but uses a queue. The first vertex it is called with is enqueued and then vertices are dequeued as long as the queue is not empty. For each vertex removed from the queue all its edges and outgoing arcs (edges and incoming arcs, respectively edges, outgoing arcs, and incoming arcs) are tested for being marked. If unmarked the other incidence vertex is also tested for being marked. If it is unmarked, it gets a mark and is enqueued. If *f-BFS* is called with initial vertex $v$ a vertex $u$ will be visited once, several times or not at all. Only at the first visit a vertex will be marked and put into the queue. Processing of a Vertex takes place after it has been dequeued.

In contrast to depth-first search, breadth-first search possesses an optimality property: It finds reachable vertices following shortest paths. To this end we introduce the *f-shell $i$ of a vertex $v$*. A vertex $w$ reachable from von $v (v \neq w)$ belongs to f-shell $i$ of $v$ if the length of a shortest f-path from $v$ to $w$ equals $i$. b-shell and a-shell are defined in an analogous manner. $\{v\}$ is shell 0. Starting with vertex $v$, breadth-first search visits (first visit) the reachable vertices in the order of ascending shell numbers.

**Theorem 4.5** *1. f-breadth-first search in a general graph uses $c_1 \cdot n + c_2 \cdot 2m$ time steps.*
*2. In an initially unmarked graph $BFS(v)$ visits all vertices reachable from $v$ and only those.*
*3. All vertices of shell $i$ are marked before a vertex of shell $i+1$ is marked.*

Theorem 4.5 also holds for b-breadth-first search and a-breadth-first search.

# 5 The Biblock Decomposition

The biblock decomposition extends the notion of a-connectedness and allows subdividing of weak components. For this reason, in this section we consider only a-paths. The only exception is subsection 5.7. The results are the same for all general graphs of an orientation class.

## 5.1 Classes of closed a-paths and edge partitions

The biblock decomposition of a general graph is determined by subgraphs in which there exist at least two *internally disjoint* or *line disjoint* a-path between every pair of distinct vertices. Internally disjoint means that the paths have only the end points in common. Line-disjoint means that the paths have no line in common. In both cases there must exist a-circuits in the subgraph. For this reason we start with a-cyclic weak components (see subsection 3.5, page 13). In these we consider a hierarchy of closed a-paths:

$$\text{a-}circuits \subseteq line\text{-}simple\ closed\ \text{a-}paths \subseteq stopfree\ paths$$

For line-simple paths and circuits see subsection 3.1, page 6. A *stopfree path* is a closed a-path, where immediately succeeding lines are different and, if the length is at least 2, the first line is different from the last[1]. A schematic picture is given in figure 4.

In Graph *Ugraph1*, figure 5, $d_2, d_1, d_0, c_0, c_3, c_2, c_1, c_0, d_0, d_2$ is a stopfree path[2], which is not line-simple. $d_2, d_1, d_0, c_0, c_3, c_2, c_1, c_0, d_0, d_1, d_2$ is not a stopfree path, since the first and the last line are equal.

**Definition 5.1** *We define relations between lines: Lines $e$ and $f$ are*

1. stopfree joined*, if there is a stopfree path through $e$ and $f$,*

[1] The name can be explained as follows: A car does not have to go into reverse to return to the starting point in the same direction it has left it.

[2] Since there are no multiple lines, paths can be specified by sequences of vertices.

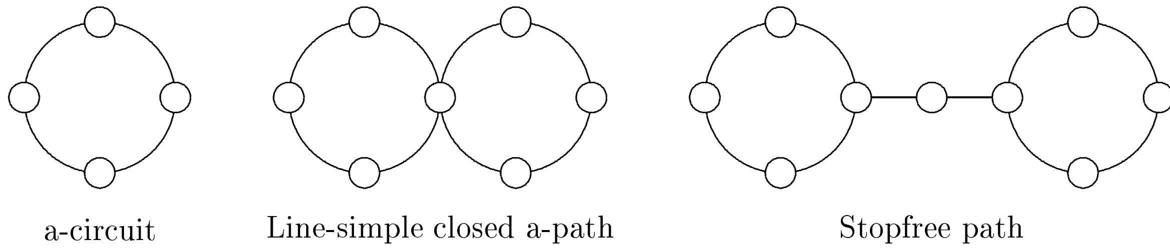


Figure 4: *Examples of closed a-paths*

2. line-simple joined, *if there is a line-simple closed a-path through $e$ and $f$,*
3. circuit joined, *if there is a a-circuit through $e$ and $f$.*

From definition 5.1 follow a hierarchy of partitions of a general graph. They are based on the following theorem.

**Theorem 5.1**

1. *Stopfree join is an equivalence relation on the set of lines which lie on a stopfree path.*
2. *Line-simple join is an equivalence relation on the set of lines which lie on a line-simple closed a-path.*
3. *Circuit join is an equivalence relation on the set of lines which lie on an a-circuit.*

## 5.2 The Biblock Decomposition of General Graphs

**Definition 5.2**

1. *An equivalence class of stopfree joined lines is called* stopfree kernel.
2. *An equivalence class of line-simple joined lines is called* subcomponent.
3. *An equivalence class of circuit joined lines is called* biblock.

*The subgraph generated from the equivalence classes have the same names.*

A-acyclic weak components are a-trees. In a a-tree every closed a-path must traverse a line twice in immediate sequence. A-trees cannot contain stopfree kernels and therefore they can neither contain subcomponents nor biblocks. A-cyclic weak components contain a-circuits and therefore at least one stopfree kernel. As stated in proposition 5.1 they contain exactly one stopfree kernel.

The set of lines of an a-cyclic weak component which do not belong to the stopfree kernel is either empty or generates an a-acyclic subgraph. The weak components of this subgraph are a-trees, called *peripheral trees.* Some vertices belong to both, a peripheral tree and the stopfree kernel. We call them *border points.*

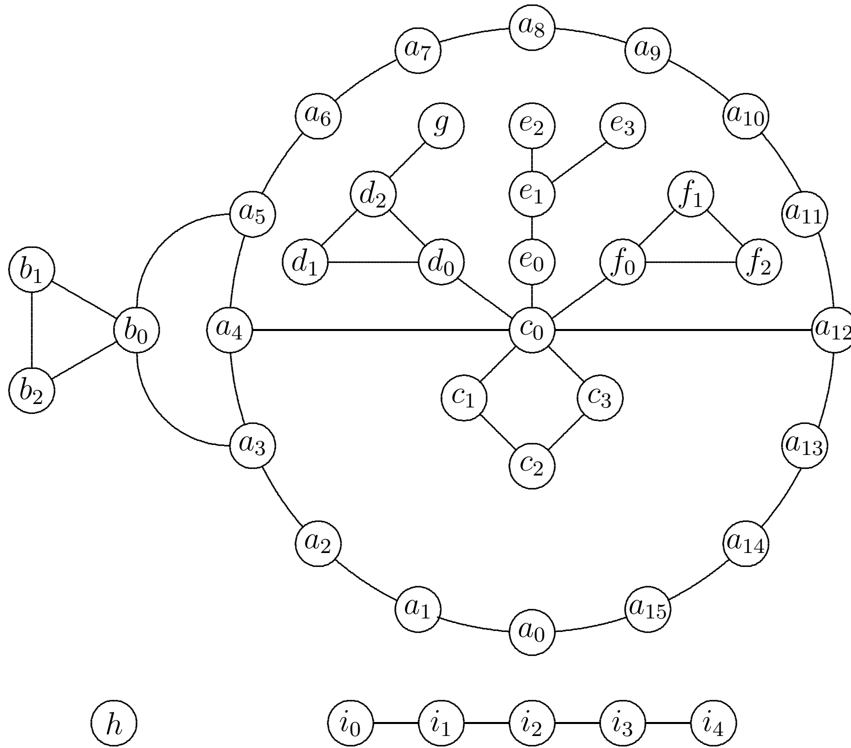


Figure 5: *Ugraph1*

The set of lines of a stopfree kernel not belonging to any subcomponent is either empty or generates an a-acyclic subgraph. The weak components of this subgraph are a-trees, called *internal trees*. Some vertices belong to both, an internal tree and a subcomponent. We call them *check points*.

A subcomponent consists of one or more biblocks. Each of its lines belongs to exactly one biblock. However, a vertex may belong to more than one biblock. Such a vertex is called *hinge point*.

Together, border points, check points, and hinge points are the *attachment points* of the a-cyclic weak component.
Figure 6 shows schematically the biblock decomposition of general graphs.

**Example 5.1** Graph *Ugraph1*, figure 5, page 26, is undirected. It consists of three (proper or improper) weak components[3]:

1. $\{h\}$
   Improper weak component.

2. $\{i_0, i_1, i_2, i_3, i_4\}$
   Proper a-acyclic weak component.

[3]Given by their vertices.

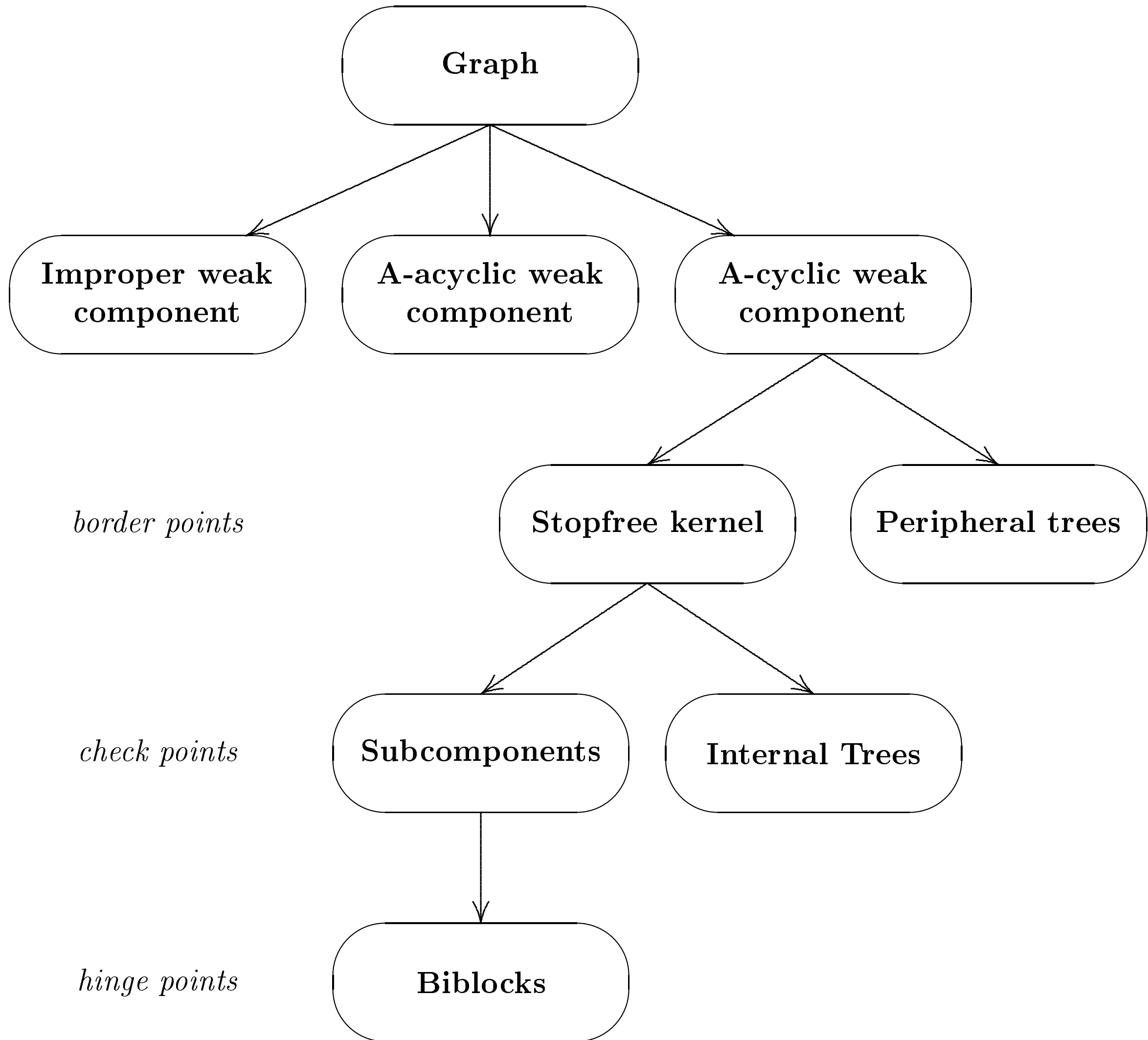


Figure 6: *Biblock decomposition of a general graph*

3. $\{a_0, \ldots, a_{15}, b_0, b_1, b_2, c_0, c_1, c_2, c_3, d_0, d_1, d_2, e_0, e_1, e_2, e_3, f_0, f_1, f_2\}$
   Proper a-cyclic weak component. The stopfree kernel consists of thee subcomponents. The first is multi-biblock. The biblocks are $\{a_0, \ldots, a_{15}, b_0, c_0\}$, and $\{b_0, b_1, b_2\}$, $\{c_0, c_1, c_2, c_3\}$. The remaining two subcomponents are mono-biblock: $\{d_0, d_1, d_2\}$ and $\{f_0, f_1, f_2\}$. The subcomponents are joined by the internal tree $\{c_0, d_0, f_0\}$.
   In addition, there exist the peripheral trees $\{d_2, g\}$ and $\{c_0, e_0, e_1, e_2, e_3\}$.
   $c_0$ is border point, check point, and hinge point. $d_2$ is border point. $d_0$ and $f_0$ are check points. $b_0$ is hinge point. □

## 5.3 Properties of the Biblock Decomposition

### 5.3.1 Stopfree Kernel and Peripheral Trees

The following proposition justifies the name stopfree kernel.

**Proposition 5.1** *In an a-cyclic weak component there exists exactly one stopfree kernel.*

According to proposition 3.2, page 8, the lines of biblocks, and therefore those of subcomponents, are non-bridges. All non-bridges belong to the stopfree kernel and every stopfree kernel contains a subcomponent. A stopfree kernel may contain bridges, too. The lines of its internal trees are bridges. Is it possible that a stopfree path consists of bridges only? The following theorem states that this is not possible and it specifies which a-circuits exist. To this end start with the incidence points $a$ and $b$ of a bridge. We remove the bridge from the graph and note $[a]$ the (proper o improper) weak component containing $a$. $[b]$ is defined in the same way.

**Theorem 5.2** *A bridge $l$ with incidence points $a$ and $b$ lies on a stopfree path if and only if there is an a-circuit in $[a]$ and an a-circuit in $[b]$.*

From theorem 5.2 follows that an a-acyclic weak components cannot have a stopfree kernel as already shown in figure 6.

**Proposition 5.2** *Every peripheral tree has exactly one border point.*

### 5.3.2 Subcomponents and Internal Trees

In contrast to biblocks, a vertex belongs to at most one subcomponent.

**Proposition 5.3** *All non-bridges incident with the same vertex are element of the same subcomponent.*

**Corollary:** *Every vertex of a general graph is member of at most one subcomponent.*

A path without bridges is called *bridgefree.* Two vertices of a weak component are called *bridgefree connected* if they are joined by a bridgefree a-path. They are named *line-simply cyclic connected* if there is a line-simple closed a-path passing through them, i.e. if they belong to the same subcomponent.

**Theorem 5.3**

1. *Two vertices $u$ and $w$ of a weak component are bridgefree connected if and only if they are line-simply cyclic connected.*
2. *Let the vertices be distinct and bridgefree connected. Then every bridgefree a-path from $u$ to $w$ can be extended by a line-simple a-path from $w$ to $u$ to become a closed line-simple a-path.*

The following proposition states simple properties of internal trees.

**Proposition 5.4**

1. *An internal tree and a subcomponent have at most one vertex in common.*
2. *An internal tree has at least two checkpoints. Each checkpoint is shared with exactly one subcomponent.*

### 5.3.3 Biblocks

Some simple properties of biblocks are stated in the following proposition.

**Proposition 5.5** *1. Each biblock of multi-biblock subcomponent contains at least one hinge point.*

*2. Two distinct biblocks of the same subcomponent have at most one hinge point in common.*

*3. Two hinge points of a subcomponent lie on an a-circuit if and only if they belong to the same biblock.*

## 5.4 Line and Vertex Classification

A line is either a bridge or a non-bridge. Bridges appear only in trees, i.e. internal trees, peripheral trees or a-acyclic weak components. The following proposition gives a type classification of bridges and is an extension of theorem 5.2.

**Proposition 5.6** *A bridge with incidence points a and b is*

1. *a line of an internal tree if and only if both,* [*a*] *and* [*b*] *contain a subcomponent.*
2. *a line of a peripheral tree if and only if either* [*a*] *or* [*b*] *contain a subcomponent.*
3. *a line of an a-acyclic weak component if and only if neither* [*a*] *nor* [*b*] *contain a subcomponent.*

The results we have obtained so far allow a complete and uniform classification of the a-properties of lines and vertices. The classification is shown in tables 10 and 11.

| *Name* | *Remarks* |
|---|---|
| Line of an a-acyclic weak component | Bridge. No a-circuit at any side. |
| Line of a peripheral tree | Bridge. a-circuit at only one side. |
| Line of n internal tree | Bridge. a-circuits at both sides. |
| Non-bridge | Belongs to exactly one biblock, one subcomponent, and one stopfree kernel. |

Table 10: *a-classification of lines*

| *Name* | *Incidences* | *Remarks* |
|---|---|---|
| Isolated vertex | 0 | The only vertex which is not part of a proper weak component. |
| End vertex | 1 | Occurs only in a-acyclic weak components and in peripheral trees. Not a cut point. |
| Internal vertex of a tree | $\geq 2$ | Occurs in a-acyclic weak components, in peripheral trees, and in internal trees. Cut point. |
| Internal vertex of a biblock | $\geq 2$ | Not a cut point. |
| *any combination of the following:* | *if more than one applies, the non-bridge lines coincide:* | |
| Border point | $\geq 1$ for the peripheral tree. $\geq 2$ lines of the stopfree kernel. | Belongs to exactly one stopfree kernel. Cut point. |
| Check point | $\geq 1$ for the internal tree. $\geq 2$ Lines of the subcomponent. | Belongs to exactly one subcomponent. Cut point. |
| Hinge point | $\geq 2$ for each biblock. | Belongs to at least two biblocks. Cut point. |

Table 11: *a-classification of vertices*

## 5.5 The Biblock Graph

The *biblock graph* of a general graph is a derived graph (see subsection 3.6, page 14). It represents the decomposition of the general graph into biblocks, peripheral trees, and internal trees. It is an undirected bipartite graph. The vertices of the biblock graph are the biblocks, peripheral trees, and internal trees on the one hand. On the other hand the attachment points (order points, checkpoints, hinge points) form the second class of vertices. An edge joins a biblock, peripheral tree or internal tree to an attachment point if it contains the latter. There are no other edges. These edges are not elements of the original graph. Therefore they are called *metaedges*. The biblock graph is a-acyclic. It is weakly connected if and only if the original general graph is weakly connected. In that case, the biblock graph is an a-tree, the *biblock tree*. An end vertex of the biblock tree is always a biblock or a peripheral tree. A biblock graph is shown as example A.5, page 96, in the appendix.

## 5.6 Algorithms for Finding the Biblock Decomposition

In this subsection we present two algorithms which together determine the biblock decomposition of a general graph. These are algorithm PERTREES, table 12, and algorithm

*STPFKERNEL*, table 14, page 33, and table 15, page 34. The general procedure is as follows:

1. As a first step, the improper weak components, the a-acyclic weak components, and the a-cyclic weak components are determined. This is done using algorithm WCOMP, table 4, page 19.
2. Then each a-cyclic weak component is processed.
   (a) Traversing its list of vertices we find the peripheral trees. To this end algorithm PERTREES is used.
   (b) Subsequently the recursive algorithm *STPFKERNEL* is applied to determine the structure of the stopfree kernel.

The algorithms use line colors brown, pink, magenta, orange, and green. In addition, *STPFKERNEL* uses a marking of the vertices. Initially, all vertices are unmarked an all lines are uncolored.

### 5.6.1 Determining the Peripheral Trees

Table 12 shows algorithm PERTREES for finding the peripheral trees of an a-cyclic weak

**PERTREES(*WC*)**

```
1    for (every vertex v in WC)
2       { if (v incides with exactly one line e && e is not a loop)
                /* v is an end vertex not yet processed */
3           { w = v;
4             while (w is incident with exactly one uncolored line e && e is not a loop)
5                { color e brown;
6                  label w as vertex of a peripheral tree (not a border point);
7                  w = otherend(e, w);
8                };
9             label w as border point;
10      }   };
```

Table 12: *Algorithm for finding peripheral trees*

component $WC$. The algorithm traverses the list of vertices of $WC$ and processes all vertices of total degree 1. From each of these there is a uniquely determined a-path to the border point of the tree. The algorithm starts from the end point and follows the path as long as there is only one uncolored line indicating the unique direction to continue (line 4). The lines thus found are colored brown (line 5). The vertices are identified as peripheral tree vertices, but not border points (line 6). Eventually, always a vertex is

reached which is incident with more than one uncolored lines. This is a candidate for a border point and labeled as such (line 9). However, this labeling is reset once the vertex is incident with brown lines only (lines 6, 7).

When a a-cyclic weak component has been completely processed by PERTREES then all its lines belonging to peripheral trees are colored brown. All other lines are still uncolored. All vertices still remain unmarked. However each vertex has been classified as border point, non-border point of a peripheral tree ore none of these.

### 5.6.2 Determining the Structure of the Stopfree Kernel

The stopfree kernel of an a-cyclic weak component is found with the recursive algorithm *STPFKERNEL*. In essence, this algorithm is an a-depth-first search. But this search has to be started in a vertex which is border point or not element of a peripheral tree. In addition, a global variable *counter* must have the starting value 0. Table 13 shows this Initialization. Tables 14 and 15 contain the recursive algorithm *STPFKERNEL*. Its

**INITIALIZATION (*WC*)**

*1* $v =$ first vertex in $WC$, which is border point or not vertex of a peripheral tree;
*2* $counter = 0$;
*3* $STPFKERNEL(WC, v, \text{NULL})$;

Table 13: *Starting vertex for STPFKERNEL*

calling parameters are the weak component $WC$, the starting vertex $v$, and the line along which it is entered, *entryedge*. At the first call *entryedge* equals NULL.

*STPFKERNEL* traverses in an a-depth-first search the complete stopfree kernel. The search starts with the vertex supplied with the first call. According to the corollary of lemma 4.3, page 17, there are only tree arcs and back arcs. With the exception of the first vertex, the first visit of every vertex is over an incoming tree arc. All other lines the vertex is incident with are either incoming back arcs or outgoing tree arcs or outgoing back arcs. There exist at least one outgoing arc and it is possible that all outgoing arcs are back arcs. Each outgoing tree arc uniquely determines a depth-first search subgraph. The actual vertex and the outgoing tree arc are not considered elements of this subgraph. Back arcs may point to vertices outside the subgraph. As easily can be seen the following proposition holds.

```
SUB    *STPFKERNEL(WCOMP *WC, VERTEX *v, EDGE *entryedge)
integer    incounter, outcounter;
SUB    *sub, *suba; /* variables for subcomponents */
1    mark v;
2    if (v border point) collect brown lines to build peripheral tree (depth-first search);
3    incounter = counter;
4    sub = NULL;
5    for (all uncolored lines e incident with v)
6        { if (otherend(e, v) marked) /* line is back arc */
7            { if (e loop)
8                { new biblock and, if necessary, new subcomponent; add e; }
9              else
10               { color e yellow;
11                 counter = counter + 1;    }
12           }
13         else
14           { color e pink; /* tree arc */
15             outcounter = counter;
16             suba = STPFKERNEL(WC, otherend(e, v), e);
17             add suba to sub;
18             if (counter > outcounter)
19               { for (all yellow lines ending as back arcs in v)
20                   { change color to magenta;
21                     counter = counter − 1;    }
22                 if (counter == outcounter) /* biblock complete */
23                   { new biblock and, if necessary, new subcomponent;
24                     starting with e collect all pink and magenta lines (depth-first
                       search), change color to orange, and add to biblock; }
25               }
26             else
27               {color e green;    }
28       }    } /* all lines incident with vertex are processed */
```

Table 14: *Recursive algorithm for determining the stopfree kernel (part I).*

```
29 if (counter > incounter) /* There are yellow back arcs from v or a successor
     of v ending in a predessor of v. */
30   { collect recursivley all green lines, if there are any, and create internal tree;
31     return sub;
32   }
33 else /* subcomponent complete or internal vertex of an internal tree */
34   { if (entryedge == NULL) /* starting vertex of depth-first search */
35     collect recursively all green lines, if there are any, and create internal tree;
36   };
37 return NULL;
```

Table 15: *Recursive algorithm for determining the stopfree kernel part (part II)*

**Proposition 5.7** 1. *An outgoing tree arc is a bridge if and only if all back arcs in the corresponding depth-first search subgraph end in a vertex of this subgraph.*

2. *A vertex is a cut point separating the depth-first search subgraph of an outgoing tree arc from the rest of the graph if and only if all back arcs of the subgraph end in the subgraph or the vertex.*

*STPFKERNEL* processes all unmarked vertices. At a border point all lines of the peripheral tree (brown) are collected. Then all uncolored lines the vertex is incident with are processed and colored (lines 5 to 28 of *STPFKERNEL*). Of essential importance is the global variable *counter*. Its initial value is 0. For each outgoing back arc different from a loop *counter* is augmented by 1. The back arc is colored yellow (lines 10 and 11). If the line is an outgoing tree arc it will be colored pink, the value of *counter* is saved in a local variable *outcounter*, and *STPFKERNEL* is called at the next recursion level (lines 14, 15 and 16).

After returning to the original recursion level (line 16) *counter* is compared to *outcounter* (line 18). If *counter* is greater than *outcounter*, then there may exist back arcs pointing from the corresponding depth-first search subgraph to the actual vertex. These arcs are recolored magenta and for each of them *counter* is diminished by 1 (lines 19, 20, and 21). Since *counter* is diminished only for lines which previously had been colored yellow, the value of *counter* cannot become negative.

If after this balancing *counter* equals *outcounter* then there is at least one back arc from the depth-first search subtree to the actual vertex but none to a previously visited vertex. The outgoing tree arc from which *STPFKERNEL* returned lies on an a-circuit through the actual vertex, i.e. it is element of a biblock. From proposition 5.7 follows that this vertex separates the biblock from all vertices previously visited by *STPFKERNEL*, if there are any. The biblock is completely determined with the outgoing tree arc. As a next step, a new biblock, and if necessary, a new subcomponent are created. Starting with the actual vertex all pink and all magenta lines are collected and put into the new biblock (lines 22, 23, and 25). Proposition 5.8 below proves this to be correct.

If after balancing *counter* is still greater than *outcounter* then there exist back arcs from the depth-first search subtree to a vertex visited earlier than the actual vertex. There is an a-circuit through the incoming tree arc. The corresponding biblock is not yet complete.

If after returning to the actual recursion level, *counter* equals *outcounter* (line 18), no balancing is necessary. Al back arcs in the depth-first search subgraph end in vertices of that subgraph. According to proposition 5.7 the outgoing tree arc is a bridge and belongs to an internal tree. It is colored green (line 27).

After having processed all lines incident with the actual vertex (line 28), $STPFKERNEL$ checks whether *counter* has a greater value than at entry time (line 29). In this case it is part of a not yet completed biblock. There may be green lines incident with the actual vertex. If so, they belong to an internal tree which is not extended beyond the incoming tree arc. This internal tree is completely determined and will be collected (line 30). The actual subcomponent is not yet complete (line 31).

If *counter* has the same value as it had at entry time then either the incoming tree arc is a bridge (proposition 5.7) or the recursion has returned to the first vertex of the depth-first search. In the former case either a subcomponent has been completed or the actual vertex is inner vertex of an internal tree. No subcomponent is returned (line 37). In the latter case a subcomponent (the last) may have been completed. This subcomponent may have been completed as well in a previous step. Whichever holds, it has to be tested whether an internal tree (the last) has to be collected (lines 35).

It remains to prove that collecting the pink and magenta lines (lines 23 and 24) is correct.

**Proposition 5.8** *Algorithm STPFKERNEL finds correctly the biblocks, internal trees, and subcomponents of the stopfree kernel of a a-cyclic weak component.*

**Sketch of the proof:** First it is proved that the algorithm is correct if the weak component consists of one singe biblock. Then the general case is proved by induction over the number of biblocks of the stopfree kernel. The details of the proof can be found in [Stie2006]. □

### 5.6.3 Efficiency

Algorithm PERTREES traverses the vertex list of the weak component and processes each line of a peripheral tree once when coloring brown. The number of steps needed is bounded by $O(n_s + m_s)$, where $n_s$ is the number of vertices and $m_s$ the number of lines of the weak component. For a-cyclic weak components holds always $n_s \leq m_s$, hence the complexity is bounded by $O(m_s)$.

Algorithm *STPFKERNEL* is a depth-first search. Each line is addressed several times for coloring and collecting. However, the number of times this is done does not depend of the size of the weak component. Without considering the complexity for creating the data structures, the complexity is also $O(m_s)$. It is possible to create the data structures in linear time, too. However, it is convenient to organize the various lists of vertices and lines as binary trees. So, the complexity is $O(m_s \cdot \ln(m_s))$.

*STPFKERNEL* collects in a subcomponent all biblocks which are part of it. This is done combining partial subcomponents (line 17). The number of steps needed for that depends on the number of biblocks and their relative position in the biblock tree. The union of partial subcomponents can be implemented using code which does not depend on the graph size and inserting each biblock twice into biblock lists. This yields $O(bn \cdot \ln(bn))$ for the complexity, where $bn$ is the number of biblocks in the stopfree kernel. Normally $bn$ will be small compared to the number of lines and the complexity of line 17 can be neglected. In any case, the total complexity is given by $O(m_s \cdot \ln(m_s))$.

## 5.7 Digraphs and Complete Orientations

The biblock decomposition of a general graph does not take into account the orientation of lines. However, it implies some orientation dependent properties, too. For instance, in a strongly connected graph all lines in peripheral or internal trees must edges. They are bridges. See example A.4, page 96, for an illustration. A strongly connected digraph is then a single subcomponent. It may consist of several biblocks separated by hinge points. Every arc lies on a f-circuit. Every two arcs lie an a-circuit. However, it is not correct, that every two arcs lie on a f-circuit.

In the following, we address some orientation dependent properties which are not directly related to the biblock decomposition. Orientation classes and complete orientations have been introduced in subsection 2.2, page 6. We now examine the problem of determining whether there are complete orientations of a general graph which have certain properties. We start with the question: Is there an f-acyclic complete orientation? For undirected graphs, the answer is simple:

**Proposition 5.9** *An undirected graph has an f-acyclic complete orientation if and only if it is loopfree.*

**Proof:** If there is a loop, no complete orientation can be f-acyclic.

If the graph is loopfree an f-acyclic complete orientation can be found applying an a-depth-first search. All edges which are tree arcs are oriented in the tree arc direction. All edges which are back arcs are oriented in the opposite direction. If the vertices are labeled with the number of the recursion level all new arcs will lead from a lower level to a higher level. Hence no f-circuits are possible. □

The question whether a general graph possesses an f-acyclic complete orientation is somewhat more difficult to answer. Of course, if the graph has a loop or there is an f-circuit consisting of arcs only then no complete orientation will be f-acyclic. It turns out that this necessary condition is sufficient, too.

**Proposition 5.10** *A general graph has an f-acyclic complete orientation if and only if it is loopfree and has no f-circuit consisting of arcs only.*

**Proof:** If in the graph there is a loop or an f-circuit consisting entirely of arcs then no complete orientation can be f-acyclic.

Assume there is a loopfree general graph with no f-circuit consisting of arcs only which cannot be completely oriented to an f-acyclic digraph. That means that in whatever order we orient the edges of the graph at some instant an f-circuit must result. We now start with a given sequence of the edges. We orient the edge and proceed if no f-circuit of arcs results. If that is not possible we orient the edge in the opposite direction. Our initial assumption implies that there must be a first edge which cannot be oriented in either direction without generating an f-circuit of arcs. The two arcs resulting from the orientations of the edge must be elements of the new f-circuits of arcs. Hence there are f-paths of arcs from each incidence point of the edge to the other which do not contain the edge. That implies that there must have been an f-circuit of arcs before the processing has reached the edge. This contradicts the assumption, that the edge is the first forcing the existence of an f-circuit of arcs. □

**Remark 5.1** From proposition 5.10 a simple algorithm for finding an f-acyclic complete orientation of a general graph fulfilling the prerequisites can easily be formulated: For every edge try an orientation and use the opposite direction if an f-circuit of arcs results. The worst case complexity of this algorithm is $O(m \cdot (m + n))$.

However, there is a more efficient algorithm which also provides another proof of proposition 5.10. Here is a sketch: Assume the set of arcs of a general graph not to be empty. Consider the subgraph generated from the set of arcs and add all still missing vertices of the original graph as isolated vertices. Decompose this new graph into proper and improper weak components. Number these in any order. The proper weak components are dags. Use algorithms PTOPSORT and STRONGCOMP (see subsection 4.4) to find within each weak component a linear numbering of the vertices topologically consistent with level numbers. Finally, find a numbering of all vertices of the graph topologically consistent with the numbering within each weak component as well as with component numbers. With this numbering an arc will always lead from the vertex with smaller number to the vertex with greater number. We now orient all edges from lower numbered incidence point to higher numbered incidence point to obtain a complete orientation. This orientation is f-acyclic. □

We now start to examine complete orientations which in a sense are the opposite of f-acyclic complete orientations, namely strongly connected complete orientations. For a general graph to have a strongly connected complete orientation it is necessary that the graph be strongly connected and have no bridges. It turns out that this condition is also sufficient.

**Theorem 5.4** *1. A strongly connected general graph has a strongly connected complete orientation if and only if it is bridgefree.*

*2. A strongly connected general graph has a strongly connected complete orientation if and only if every edge lies on an f-circuit.*

**Proof:** Let $G$ be a strongly connected general graph. Arcs of any orientation of $G$ lie on f-circuits (proposition 3.3, page 9). They cannot be bridges (proposition 3.2, page 8).

Propositions 3.2 and 3.4 imply that an edge is not a bridge if and only if it lies on an f-circuit. So, it suffices to prove the second statement of the theorem. This follows immediately from proposition 3.8, considering that an edge lies on an f-circuit in $G$ if and only if it lies on an a-circuit (proposition 3.4). □

**Algorithm** We sketch an algorithm for finding a strongly connected complete orientation if such an orientation exists. We use arguments from the proof of proposition 3.8.

1. Using the algorithms of subsection 5.6 we find the biblock decomposition. There are no bridges if and only if there are no peripheral trees and no internal trees.

2. If $G$ is bridgefree we construct $G-l$ for every edge $l$ and determine its decomposition into strong components and the external dag (subsection 4.4, page 18). If one single strong component results $l$ can be oriented arbitrarily. Otherwise the edge is oriented from the strong component with positive level number to the strong component with level number 0. □

**Remark 5.2** The worst case complexity of this procedure is $O(m \cdot (m+n))$, since for each edge $l$ the strong decomposition has to be determined. A heuristics which may result in improvements is the following: An edge incident with a vertex where all other incident lines are incoming arcs must be oriented to become an outgoing arc and similarly with outgoing arcs. Such a forced orientation may lead to additional orientations in the neighborhood of the vertex.

In the case of $G$ being an undirected bridgefree graph a simpler and more efficient algorithm exists: Use the tree arcs and the back arcs of any a-depth-first search.
Sketch of proof: If the graph is one single biblock there is an f-path from the starting point of the depth-first search to any other vertex (obvious) and an f-path back (possibly requiring some loop-like movements in the depth-first tree and the back arcs). If the graph consists of several biblocks each of them is traversed by depth-first search as if it were a single biblock. At the hinge points, interruptions leading to other biblocks are possible. Finally, observe that the union of two strongly oriented digraphs which have exactly one vertex in common is itself strongly connected.

□

## Remarks and Literature

In the graph theoretic literature *blocks* are considered. In addition to biblocks, isolated vertices and subgraphs generated by a bridge are called blocks, too. With these blocks a derived graph structure similar to the biblock graph and called *block-cutpoint graph* is defined. See for example Harary [Hara1969]. An efficient algorithm for finding biconnected blocks (i.e. biblocks) was found by Tarjan [Tarj1972] in the early seventies.

The biblock decomposition and the biblock graph presented in this section were introduced as an alternative by Stiege ([Stie1996b], [Stie1997], and [Stie1998]). They are clearer, more natural, and much better suited for applications. The algorithms for determining

the biblock decomposition have been found by Stiege [Stie1997a], too. They are of the same efficiency as Tarjan's algorithm but simpler and reveal more structural details.

# 6 Periodicity

## 6.1 a-Period and f-Period

Figure 7 shows a strongly connected digraph. Every closed f-path starting in vertex V06

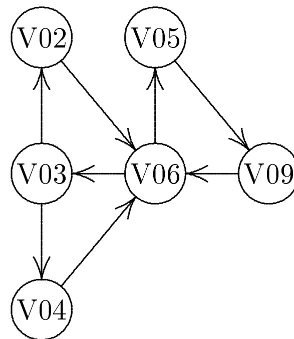


Figure 7: *Period of a strongly connected digraph*

has a length which is a multiple of 3. Motivated by the example we come up with the following definition.

**Definition 6.1** 1. *Let $v$ be a non-isolated vertex of a general graph $G$. The* a-period *of $v$ is the greatest common divisor of the lengths of all a-paths starting and ending in $v$.*

2. *Let $v$ be a vertex with return of a general graph $G$. The* f-period of $v$ *is the greatest common divisor of the length of all f-paths starting an ending in $v$.*

For isolated vertices no a-periods and for vertices of no return no f-periods are defined. In figure 7 we observe, that not only V06 but all vertices of the graph have f-period 3. The following theorem states that to be a general property.

**Theorem 6.1** *All vertices of a weak component have the same a-period. All vertices of a strong component have the same f-period.*

**Proof:** We show that any two vertices of the same weak (strong) component have the same a-period (f-period). Let $s$ be the a-period (f-period) of $u$ and $t$ the a-period (f-period) of $v$. We choose a closed a-path (f-path) from $v$ to $u$ and from $u$ back to $v$. Its length be $l_1$. Furthermore, let be given an arbitrary closed a-path (f-path) starting in $u$. Its length be $l_2$. See figure 8. We compose both paths going from $v$ to $u$ on the first path, then from $u$ to $u$ on the second and finally back from $u$ to $v$ on the first. The composed path is an a-path (f-path) from $v$ to $v$ and has length $l_1 + l_2$. I. e. there is $k'$ such that $l_1 + l_2 = k't$.

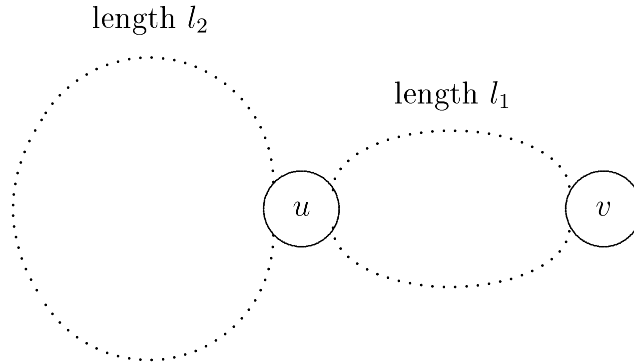


Figure 8: *Mutually reachable vertices have the same period*

In addition, there is $k$ , such that $l_1 = kt$. That implies $l_2 = l_1 + l_2 - l_1 = (k' - k)t$. Hence, the lengths of all a-paths (f-paths) from $u$ to $u$ are multiples of $t$. Therefore $t \leq s$. In the same way $s \leq t$ is proved yielding finally $s = t$. □

This result allows us to define the a-period aper($WC$) of a weak component $WC$ as well as the f-period fper($SC$) of a strong component $SC$. The following proposition follows obviously from definition 6.1 and theorem 6.1.

**Proposition 6.1** *The lengths of all closed a-paths (f-paths) of a weak (strong) component is a multiple of the component's a-period (f-period).*

The a-path from a non-isolated vertex to a neighbor and back has length 2. Accordingly, the f-path from one incidence point of an edge to the other and back has also length 2. This imposes clear restrictions on a-periods and f-periods.

**Proposition 6.2** *The a-period of a weak component can at most be 2.*
*The f-period of a strong component containing an edge can at most be 2.*

f-periods greater 2 can only exist in digraphs. A weak (strong) component with a-period (f-period) 1 is called *aperiodic*. Components containing a loop are aperiodic. The same is true if a weak (strong) component has closed a-paths (f-paths) with even and with odd lengths. All closed paths in an a-tree have even lengths and therfore a-period 2.

If we want to determine the a-period (f-period) according to the definition we have to take into account all closed a-paths (f-paths) with a fixed starting point. There are criteria which relate the period to smaller sets of paths. Of special importance in that respect are *paths of first return*. We call a closed a-path (f-path) starting in $v$ an a-path (f-path) of first return to $v$ if $v$ is no inner vertex of the path.

**Proposition 6.3** *Let $v$ be a vertex of the weak component $WC$ (strong component $SC$). The a-period of $WC$ (f-period of $SC$) equals the greatest common divisor of the lengths of all a-path (f-paths) of first return to $v$.*

## 6.2 Periodicity Classes

It is to be expected that weak components with period 2 and strong components with period greater 1 will have special structural property. A first hint is given by the digraph in figure 7. All closed f-paths starting in $V06$ pass through the vertex sets $\{V06\}, \{V03, V05\}, \{V02, V04, V09\}$ in that cyclic order. The following theorem states that this holds in general.

**Theorem 6.2** *Let $G$ be a weakly connected (strongly connected) general graph. Let $p \geq 2$ be its a-period (f-period). Then the following hold*

1. *If there is an a-path (f-path) of length $n_0p$ from $u$ to $v$ then the length of every a-paths (f-path) from $u$ to $v$ is a multiple of $p$.*
2. *Mutual a-reachability (f-reachability) in $np$ $(n \geq 1)$ steps is an equivalence relation in $V$.*
3. *$V$ is partitioned into $p$ equivalence classes. Each step of an a-path (f-path) leads from one equivalence class to another. In $p$ steps all $p$ classes a traversed in a fixed order and the last step leads to the first class of that order.*

**Proof:** 1. The given a-path (f-path) of length $n_0p$ can be extended to a closed a-path (f-path) beginning in $u$. This path has length $n_1p$ and therefore the length of the path back from $v$ to $u$ is $(n_1 - n_0)p$. If there were an a-path (f-path) from $u$ to $v$ whose length is not a multiple of $p$ then this path could be extended adding the back path to yield a closed path beginning in $u$ with length which is not a multiple of $p$.

2. Reflexivity follows from the definition of an a-period (f-period). Transitivity is obvious. The back path found under 1. shows symmetry.

3. We consider an a-path (f-path) $v_0, v_1, \cdots, v_{p-1}$ of length $p$. The vertices must be elements of pairwise distinct equivalence classes. Otherwise there would be a closed path of length smaller than $p$. So, there are at least $p$ equivalence classes. Neither can there be more than $p$: Assuming $|V| > p$, let $v$ be a vertex different from $v_0, \ldots, v_{p-1}$. We extend the path $v_0, \ldots, v_{p-1}$ by an a-path (f-path) from $v_{p-1}$ to $v$. Let $l = np + r$ with $0 \leq r < p$ be the length of the path $v_{p-1}, \ldots, v$. If $r = 0$ $v_{p-1}$ and $v$ belong to the same equivalence class. If $0 < r < p$ then $p - r$ steps are missing to reach a length which is a multiple of $p$. If these steps are added and the path is started in $v_{r-1}$ we see that $v_{r-1}$ and $v$ belong to the same equivalence class. There are exactly $p$ equivalence classes.
Now, let $v$ be reachable from $u$ in a-direction (f-direction) within 1 step. If $u$ belongs to the same equivalence class as $v_i$ with $0 \leq i < p - 1$ then we can get from $v$ to $v_{i+1}$ in the following way: From $v$ to $u$, from $u$ to $v_i$ and from $v_i$ to $v_{i+1}$. For the lengths of these paths hold: $l_1 = n_1p - 1$, $l_2 = n_2p$ and $l_3 = 1$. Therefore $v$ and $v_{i+1}$ belong to the same equivalence class. If $u$ belongs to the same equivalence class as $v_{p-1}$ then $v$ must belong to the same equivalence class a $v_0$. Otherwise $v$ would be element of an equivalence class together with $v_i$ with $1 \leq i < p-1$. If this were the case one could get from $u$ to $u$ in the following way: From $u$ to $v$, from $v$ to $v_i$, from $v_i$ to $v_{p-1}$, and from $v_{p-1}$ to $u$. For the length of these paths hold: $l_1 = 1$, $l_2 = n_1p$ , $l_3 = p - 1 - i$, and $l_4 = n_2p$. Since $1 \leq i$ the

length of the total path would not be a multiple of $p$ as it ought to be. Therefore, in $p$ steps all $p$ classes are traversed in fixed cyclic order. □

The vertex classes given by theorem 6.2 are called *periodicity classes*.

**Remark 6.1** We shall call weak components with a-period 2 *bipartite* although in the commonly used sense, bipartite graphs are undirected and need not be connected. Normally, bipartite graphs are defined by a partition of the graph's vertices into 2 classes such that the incidence points of every edge belong to different classes. It is not difficult to show that all weak components of such graph have a-period 2.
For f-periods of strongly connected digraphs the situation is somewhat different. If in such a digraph there exists a partition of the vertex set into $q$ classes such that in $q$ steps the classes are traversed in a fixed cyclic order, $q$ need not be the f-period. However, the f-period is a multiple of $q$. □

It is not to be expected that aperiodic graphs have similar vertex classes as given by theorem 6.2. In a certain sense the opposite holds.

**Theorem 6.3** *If $v$ is a vertex of an aperiodic weak (strong) component then there exists $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$ there is a closed a-path (f-path) of length $n$ through $v$.*

## 6.3 An Algorithm for Finding the Period

Why should we care to find the period of a component? On the one hand, sometimes it is important to know that a component is aperiodic. An example are transition digraphs of Markov chains with finite state space. Often but not always, aperiodicity can be inferred from the existence of loops. In the absence of loops some algorithm to determine the period is needed.

On the other hand, if the period is at least 2 we would like to know the structure imposed by the periodicity classes. Given a weak (strong) component with a-period (f-period) $p \geq 2$ t is easy to find those. We start an a-depth-first search (f-depth-first search) and collect in a class all vertices which are $p$ steps apart.

The problem is to find $p$. Table 16 shows a simple algorithm which determines the f-period of a strong component. The algorithm uses f-depth-first search and does not follow arcs of the external dag. The complexity is $O(n + m) = O(m)$. The algorithm starts with $PERIOD(root, 0)$ where *root* is an arbitrary vertex of the strong component Initially all vertices are unmarked. Vertices are represented by records of data type VERTEX. The algorithm uses an integer valued field *v-level*. $p$ is a global variable whose initial value is 0. As shown below, the final value of $p$ is the f-period. *gcd* yields the greatest common divisor of two non-negative integers not both 0.

To get an algorithm for finding the a-period of a weak component minor changes to $PERIOD$ suffice. To this end we admit in line 5 all lines incident with $v$ and do not exclude arcs of the external dag in line 6. The algorithm is started with $PERIOD(root, 0)$, too.

The algorithm $PERIOD$ is very simple. However, it is not all obvious that it does what it is intended to do.

```
void PERIOD(VERTEX v, int level)

1  { if (p == 1) return; /* aperiodic */
2    if (v is not marked)
3       { mark v;
4         v-level = level;
5         for (all edges and outgoing arcs l incident with v)
6            { if (l is not an arc of the external dag) PERIOD(otherend(l, v), level + 1);
7            }
8       }
9    else
10      { p = gcd(p, |level − v-level|);
11      };
12   return;
13 }
```

Table 16: *Recursive procedure for finding the f-period of a strong component*

**Proposition 6.4** *Algorithm PERIOD correctly finds the f-period of a strong component.*

**Proof:** Let $v_r$ be the vertex the algorithm starts with. Let $q$ be the the f-period we seek. $p_f$ is the value the algorithm finally yields. We show: I. Each execution of line 10, including the last, assigns to $p$ a value which is a multiple of $q$. II. The final value $p_f$ divides the lengths of *all* closed f-paths through $v_r$. Together this results in $p_f = q$.

*I. Line 10 yields a multiple of the f-period:* The f-depth-first search reaches a vertex $vi \neq v_r$ for the first time following a tree arc. Additional visits, if there are any, follow a back arc or a forward arc. $v_r$ is visited exclusively via back arcs. We combine back arcs and forward arcs under the name secondary arc of the depth-first search. Line 10 is executed if and only if a vertex is visited via a secondary arc. It is not obvious that in line 10 *gcd* can always be calculated. It is left as an exercise that in fact it can.

Let us assume now that the f-depth-first search enters vertex $v$ via a secondary arc. Then there is a uniquely determined f-path consisting of tree arcs from $v_r$ to $v$. Be $\lambda_1$ its length (see figure 9). The case $\lambda_1 = 0$ is possible. In addition, there is a uniquely determined f-path of tree arcs from $v_r$ to the head of the considered secondary arc. Together with this secondary arc we have a second f-path from $v_r$ to $v$. Its length be $\lambda_2$. It holds $\lambda_2 \geq 1$. Both paths can be extended adding an arbitrary but fixed f-path from $v$ to $v_r$. Two closed f-paths through $v_r$ result. Let $\lambda_r$ be the length of the complementary path. Then $\lambda_1 + \lambda_r = n_1 q$ and $\lambda_2 + \lambda_r = n_2 q$. Hence $|\lambda_1 - \lambda_2| = |n_1 - n_2|q$. If $p = 0$ or if $p$ is a multiple of $q$ then $p = \gcd(p, |\lambda_1 - \lambda_2|)$ is also a multiple of $q$. From the algorithm we see that $\lambda_1 = v\text{-}level$ and $\lambda_2 = level$. Hence all values of $p$ resulting from line 10 of the

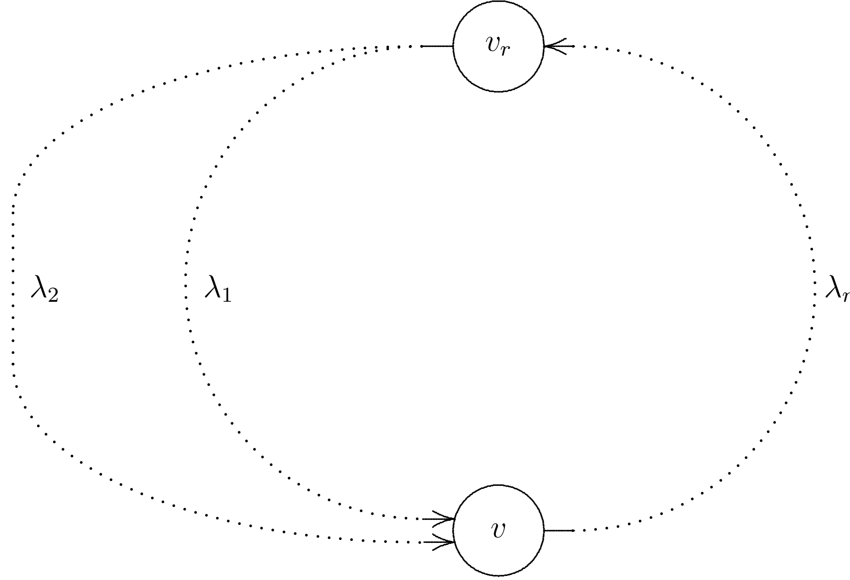


Figure 9: *Calculating the f-period*

algorithm are multiples of the f-period $q$.

*II. The final value $p_f$ divides the lengths of all closed f-paths through $v_r$:* Let

$$v_r = v_0, l_1, v_1, \ldots, l_{n-1}, v_{n-1}, l_n, v_n = v_r$$

be a closed f-path. We consider an arbitrary but fixed line $l_i$ $(1 \leq i \leq n)$ of the path. If the line is an arc, then f-depth-first search will traverse it from $v_{i-1}$ to $v_i$. Since PERIOD does not use line markings every edge will be processed twice: In the direction from $v_{i-1}$ to $v_i$ and in the direction from $v_i$ to $v_{i-1}$. In the direction from $v_{i-1}$ to $v_i$ the following system of equations result:

$$\begin{array}{lcl} v\text{-}level(v_0) + 1 - v\text{-}level(v_1) & = & k_0 \cdot p_f \\ v\text{-}level(v_1) + 1 - v\text{-}level(v_2) & = & k_1 \cdot p_f \\ \ldots & & \ldots \\ v\text{-}level(v_{n-2}) + 1 - v\text{-}level(v_{n-1}) & = & k_{n-1} \cdot p_f \\ v\text{-}level(v_{n-1}) + 1 - v\text{-}level(v_0) & = & k_n \cdot p_f \end{array}$$

This follows from line 10 of the algorithm if the line is processed as a secondary arc. If it is processed as a tree arc then $v\text{-}level(v_{i-1}) + 1 = v\text{-}level(v_i)$ and the corresponding equation holds with $k_i = 0$. Note that the equation holds even if the path is not line-simple. Adding the equations yields $n = k \cdot p_f$. I.e. $p_f$ divides the length of the path.

If there are edges $p_f$ must be a divisor of 2, too. To see that, let $l_i$ be an edge. From the two directions in which the edge is processed we obtain

$$\begin{array}{lcl} v\text{-}level(v_{i-1}) + 1 - v\text{-}level(v_i) & = & k_i \cdot p_f \\ v\text{-}level(v_i) + 1 - v\text{-}level(v_{i-1}) & = & k'_i \cdot p_f \end{array}$$

Addition yields $2 = (k_i + k_i') \cdot p_f$. □

The proof is valid also for the correctness of the algorithms for finding the a-period of a weak component. In this case every line of the path will be processed twice by the depth-first search. The line is either an edge or it is outgoing arc for its head and incoming for its tail

**Remark 6.2** For an edge, calculating the f-period requires calculating the greatest common divisor for each direction processed as secondary arc. Calculating the a-period, this is necessary for every line. Using line markings and suitable chosen initial parameters for $PERIOD$ it is possible to reduce gcd calculating to at most once per line. Of course, for determining the a-period there is a simple algorithm without any greatest common divisor calculating.

### Remarks and Literature

Bipartite graphs are treated in all textbooks on graph theory. Normally, they are introduced only as undirected graphs. In general, f-periods are are neither mentioned in textbooks on graph theory nor in textbooks on algorithms and data structures. They seem to be unknown in these areas. In the theory of Markov chains periodic and aperiodic chains are well known and important concepts. See for instance Isaacson/Madsen [IsaaM1976]. Starting from Markov chains Stiege [Stie1995d] introduced periodicity for digraphs.

# 7 Menger Structures

In the year 1927 Karl Menger [Meng1927] proved a theorem which had to become one of the fundamental theorems in graph theory. This theorem, its variants, and the corresponding algorithms are addressed in this section. The results are the basis for higher graph decompositions presented in section 8.

## 7.1 Separating Sets and Disjoint Paths

We start with a general graph $G(V, E, A, \varphi, \psi)$. Let $u$ and $v$ be two non-neighbored vertices in $G$. We now consider vertex sets $S_V$ which separate $u$ and $v$ on the one hand, and sets of paths $W$ from $u$ to $v$ on the other hand. We assume $u$ and $v$ not being elements of the separating sets and define: $S_V$ is an *a-separating vertex set (f-separating vertex set)* for $u$ and $v$, i.e. every a-path (f-path) from $u$ to $v$ passes through a vertex of $S_V$. Note that an f-separating set for $u$ and $v$ need not be an f-separating set for $v$ and $u$. For a-separating (f-separating) vertex sets we examine sets of a-paths (f-paths) from $u$ to $v$. We shall require these paths to be "different" in a reasonable manner: The paths have pairwise no internal vertices in common and neither $u$ nor $v$ appear as internal vertices. The paths are *internally disjoint.*

Instead of sets of vertices we may separate $u$ and $v$ by sets of lines. We define a-separating (f-separating) line set for $u$ and $v$ in a way analogous to separating vertex sets. With

respect to separating line sets we examine sets of a-paths (f-paths) from $u$ to $v$ which have pairwise no lines in common. The paths are *line-disjoint.*

**Remark 7.1** If vertices $u$ and $v$ of a general graph are distinct but adjacent then they cannot be separated by a set of vertices. They can be separated by a set of lines, though. In this case we therefore shall allow $u$ and $v$ to be adjacent, but not to be equal. □

## 7.2 The Menger Theorems

Now, what is the relationship between separating sets and sets of paths? If $u$ and $v$ are distinct and not adjacent then there always exist separating vertex sets. Just take $S_V := V \setminus \{u, v\}$. Then there are also a-separating (f-separating) vertex sets with minimal cardinality. Likewise, there are a-separating (f-separating) line sets with minimal cardinality.

Internally disjoint a-paths from $u$ to $v$ have no internal vertices in common. Hence, there are at most as many paths of this kind as there are vertices in an a-separating vertex set of minimal cardinality. Likewise, the maximum numbers of internally disjoint f-paths, line-disjoint a-paths, and line-disjoint f-paths are bounded by the minimum cardinalities of the corresponding separating sets.

The Menger theorems state that these bounds are achieved in all four cases.

**Theorem 7.1 (Vertices and a-paths)** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $u$ and $v$ two different non-adjacent vertices. Then the maximum number of internally disjoint a-paths from $u$ to $v$ equals the minimum number of vertices of an a-separating vertex set for $u$ and $v$.*

**Theorem 7.2 (Vertices and f-paths)** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $u$ and $v$ two different non-adjacent vertices. Then the maximum number of internally disjoint f-paths from $u$ to $v$ equals the minimum number of vertices of an f-separating vertex set for $u$ and $v$.*

**Theorem 7.3 (Lines and a-paths)** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $u$ and $v$ two different vertices. Then the maximum number of line-disjoint a-paths from $u$ to $v$ equals the minimum number of lines of an a-separating line set for $u$ and $v$.*

**Theorem 7.4 (Lines and f-paths)** *Let $G(V, E, A, \varphi, \psi)$ be a general graph and $u$ and $v$ two different vertices. The the maximum number of line-disjoint f-paths from $u$ to $v$ equals the minimum number of lines of an f-separating line set for $u$ and $v$.*

We prove theorems 7.1 and 7.2 by specifying an algorithm which finds a separating vertex set of minimal cardinality and a set of internally disjoint paths from $u$ to $v$ of the same size. The algorithm is proved to be correct.

Proof and algorithm for theorems 7.3 and 7.4 are quite similar and are left as an exercise.

**Algorithm and Proof for Theorems 7.1 and 7.2:** The theorems hold if there is no a-path (f-path) from $u$ to $v$. Otherwise we determine such a path, for instance using breadth-first search. Then we proceed with the following iterative procedure: We start with $m$ $(1 \leq m)$ internally paths and either find $m+1$ internally disjoint paths of the same type from $u$ to $v$ or an a-separating (f-separating) set of $m$ vertices. Obviously, the procedure ends with an $m_0$ such that there are $m_0$ internally disjoint paths from $u$ to $v$ and a vertex set of $m_0$ separating vertices. We call *main loop* the transition from $m$ internally disjoint paths to $m+1$ internally disjoint paths.

So, let $F_1, F_2, \ldots F_m$ be internally disjoint a-paths (f-paths) from $u$ to $v$. Without loss of generality, we may assume the $F_i$ to be simple paths. Let $W$ be the set of their vertices. For two different vertices $v_1, v_2 \in W$ we say that $v_1 \prec v_2$ if $v_1$ and $v_2$ lie on the same path and $v_1$ is nearer to $u$ than $v_2$. We also consider the extended partial order $v_1 \preceq v_2$. Let $\widetilde{F}$ the set of simple a-paths (f-paths) starting and ending in $W$ but with no internal vertex in $W$. Neither are they allowed to contain a line of the given paths. Finally, let $z_i$ $(i = 1, \ldots, m)$ be the neighbor of $u$ on $F_i$.

For the paths $F_1, F_2, \ldots, F_m$ we construct in a *step loop* vertices which are maximal *u-reachable* at step $r$.

1. $z_i$ is maximal $u$-reachable at step 0. We set $x_i^{(0)} := z_i$ for $i = 1, \ldots, m$.

2. We examine all a-paths (f-paths) in $\widetilde{F}$ which start in $u$ and distinguish the following cases:

   - None of these paths ends in a vertex $x \in W$ with $x_i^{(0)} \prec x$ where $x$ lies on $F_i$. Then the vertices $x_1^{(0)}, x_2^{(0)}, \ldots, x_m^{(0)}$ separate vertices $u$ and $v$ and the main loop ends.
   - One of these paths ends in $v$. Then we have found directly another path from $u$ to $v$ internally disjoint to the existing paths. The step loop ends. A new pass of the main loop starts with $m+1$ internally disjoint paths.
   - There exist paths ending in vertices $x$ with $x_i^{(0)} \prec x$ but none of these ends in $v$. On each path $F_i$ we choose as $x_i^{(1)}$ the vertex $x$ lieing nearest to $v$. If there is no such $x$ on $F_i$ we set $x_i^{(1)} := x_i^{(0)}$. In this manner we determine $x_1^{(1)}, x_2^{(1)}, \ldots, x_m^{(1)}$, the maximal $u$-reachable vertices at step 1. We proceed with step 2 of the step loop.

3. Assume we have found for $1 \leq r$ the maximal $u$-reachable vertices $x_1^{(r)}, \ldots, x_m^{(r)}$ at step $r$. I.e. we know that within $r$ steps we reach these vertices but we cannot get nearer to $v$. We examine paths in $\widetilde{F}$ which start in vertices $y$ with $x_i^{(r-1)} \preceq y \prec x_i^{(r)}$ and consider again the following cases:

   - We have $x \preceq x_j^{(r)}$ for all end points $x$ of these paths. Then we cannot circumvent vertices $x_1^{(r)}, \ldots, x_m^{(r)}$ and reach a vertex $x$ which equals $v$ or lies nearer to $v$ than $x_k^{(r)}$ on a path $F_k$. Neither can we circumvent the $x_1^{(r)}, \ldots, x_m^{(r)}$ using paths from $\widetilde{F}$ which start in a vertex $y \prec x_i^{(r-1)}$ for these have been examined already

when constructing steps 1 to $r$. The vertices $x_1^{(r)}, \dots, x_m^{(r)}$ separate $u$ and $v$ and the main loop ends.

- One of these paths ends in vertex $v$. Then $v$ is $u$-reachable at step $n := r + 1$. We use the unravel routine explained beneath to construct $m + 1$ internally disjoint paths from $u$ to $v$. We continue the main loop with these paths.
- $v$ is not $u$-reachable at step $r+1$ but there exist paths which end in a vertex $x$ with $x_j^{(r)} \prec x$. If such an $x$ exists on path $F_j$ we set $x_j^{(r+1)}$ to the maximum of the $x$. That is, for $j = 1, \dots, m$ we again choose the $x$ nearest to $v$. If there is no such $x$ we set $x_j^{(r+1)} := x_j^{(r)}$. $x_1^{(r+1)}, \dots, x_m^{(r+1)}$ are the maximal $u$-reachable vertices at step r+1 and we start step $r + 2$ of the step loop. Since each step shortens the remaining length of at least one of the paths $F_i$ by at least 1 the step loop must end after finitely many steps with either $m$ separating vertices or $m + 1$ internally disjoint paths. In the latter case we proceed with the next pass of the main loop.

***Unravel Routine:*** If $x_i^{(r-1)} \prec x_i^{(r)}$ $(r = 1, \dots, n)$ we call $y_i^{(r)}$ the starting point of the path in $\widetilde{F}$ with which $x_i^{(r)}$ was constructed. Note that $y_i^{(r)}$ may lie on path $F_j$ with $i \neq j$. $y^{(n)}$ is the vertex from which the path ending in $v$ was constructed. We assume the path from $y_i^{(r)}$ to $x_i^{(r)}$ has been stored and is uniquely determined. From these paths and the given paths $F_1, \dots, F_m$ we construct $m + 1$ internally disjoint paths $G_1, \dots, G_m, G_{m+1}$. To this end, we start in $v$, construct the paths in backward direction, and follow the step numbers. Firstly every path $G_i$ $(i = 1, 2, \dots, m)$ is mapped to $F_i$. The construction of $G_{m+1}$ starts with step $n$ from $y^{(n)}$ to $v$. Assume $y^{(n)}$ lieing on $F_j$. Then we have the following situation: On $F_j$ holds $y^{(n)} \prec x_j^{(n-1)} \prec v$. As yet, on all other paths no vertex besides $v$ has been used in the construction of the new paths $G_i$.

Next, we construct $G_j$ following $F_j$ from $v$ to $x_j^{(n-1)}$ and from there following the path of step $(n - 1)$ to $y_j^{(n-1)}$. There are three cases:

1. We have $y_j^{(n-1)} = u$. See figure 10, part A.
   Then $n = 2$ and path $G_j$ is complete. Path $G_{m+1}$ will be extended following $F_j$ from $y^{(n)}$ to $u$ and is also complete. Al other paths $G_i$ (not shown in the figure) are constructed following the respective $F_i$ from $v$ to $u$. We obtain $m + 1$ paths. These are internally disjoint. The original paths $F_i$ were internally disjoint and the paths used from $\widetilde{F}$ have neither internal vertices from $W$ nor common vertices.

2. It is $y_j^{(n-1)} \neq u$ and $y_j^{(n-1)}$ lies also on $F_j$. See figure 10, part B.
   Then on $F_j$ no $w$ with $w \prec y_j^{(n-1)}$ has been used for the construction of new paths, yet. In addition we have $y_j^{(n-1)} \prec x_j^{(n-2)} \preceq y^{(n)}$.

3. We have $y_j^{(n-1)} \neq u$ and $y_j^{(n-1)}$ lies on $F_k$ with $j \neq k$. See figure 10, part C.
   Then on $F_j$ no $w$ with $w \prec y^{(n)}$ and on $F_k$ no $w$ with $w \prec y_j^{(n-1)}$ has been used for constructing new paths, yet. In addition, $y_j^{(n-1)} \prec x_k^{(n-2)} \preceq v$.

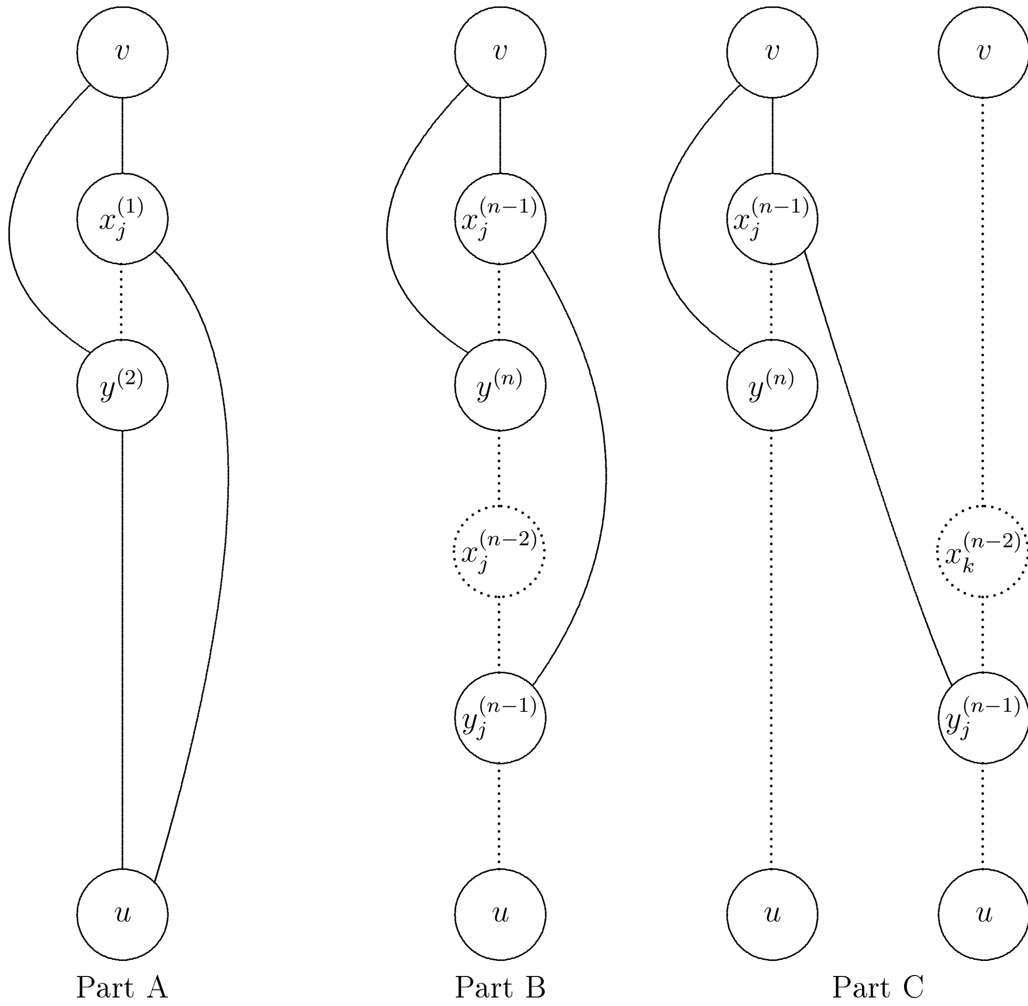


Figure 10: *Unravel Routine*

Thus we have the start of an induction. We now assume that the unravel routine after processing step $r$ ($2 \le r \le n$) leaves the following state: On each path $F_i$ there is an actual construction point. That is either $v$ or the last vertex $y$ used by the unravel routine as starting point for a path from $\widetilde{F}$. The following holds

a. All vertices $w \prec y$ of path $F_i$ have not yet been touched by the construction of new paths.

b. $y \prec x_j^{(r-1)} \preceq y'$ with $y$ being the actual construction point at step $r$, $F_j$ is the path on which $y$ lies and $y'$ is the actual construction point on $F_j$ after processing step $r+1$. This vertex can have been reached by the unravel routine with step $\rho > r+1$.

The unravel routine now processes step $r-1$. The path ending in $y'$ is extended following $F_j$ to $x_j^{(r-1)}$ and from there to $y_j^{(r-1)}$ following the path from $\widetilde{F}$ corresponding to step $r-1$. It is not difficult to see that again we have three cases as discussed above for $r = n-1$. If $y_j^{(r-1)} = u$, i.e. $r = 2$, a new path has been completed. The $m$ paths which remain are

obtained extending $F_i$ from the actual construction point to $u$. If $r > 2$ the extension in step $r - 1$ leaves again a state as describe above.

To conclude the proof it remains to show that the paths $G_1, G_2, \ldots, G_m, G_{m+1}$ we have found are internally disjoint. In fact, each of the new paths is composed by pieces from the $F_i$ and paths from $\widetilde{F}$. With each step only on of these is used and paths from $\widetilde{F}$ constructed by the step loop in different steps are internally disjoint. This together with properties a. and b. implies that the new paths are internally disjoint. □

## 7.3 An Example

The aim of this example is to illustrate the interplay between step loop and unravel routine. Figure 11 shows four internally disjoint a-paths from vertex $a$ to vertex $b$:

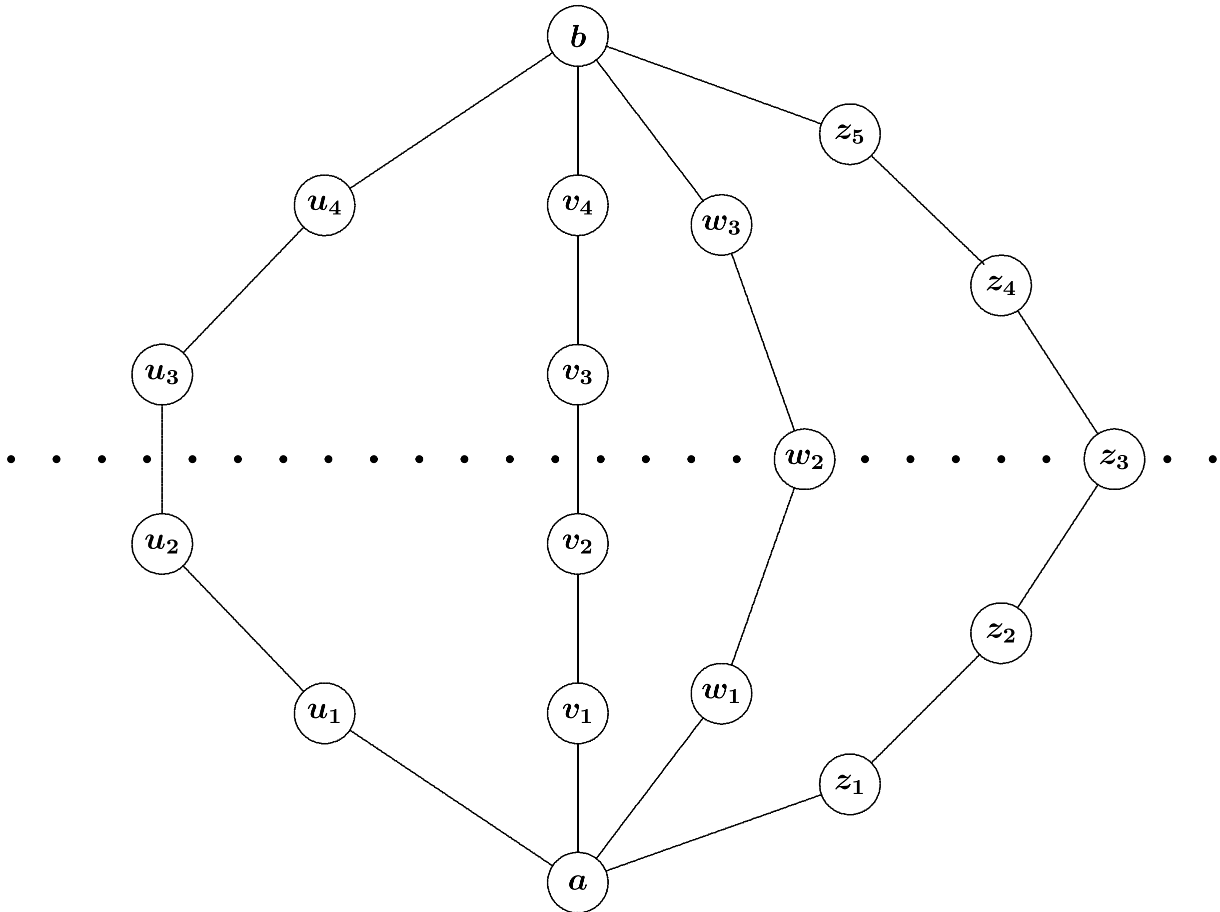


Figure 11: *Example illustrating the Menger algorithm: Initial configuration*

$F_1 : (a, u_1, u_2, u_3, u_4, b)$, $F_2 : (a, v_1, v_2, v_3, v_4, b)$, $F_3 : (a, w_1, w_2, w_3, b)$ and,
$F_4 : (a, z_1, z_2, z_3, z_4, z_5, b)$. In the figure, the boldface dots mean that the remaining vertices and lines are not shown. The state after the step loop has run is shown in figure 12. There are steps 0 to 5. In step 1 a path from $a$ to $u_4$ is found, in step 2 a path from

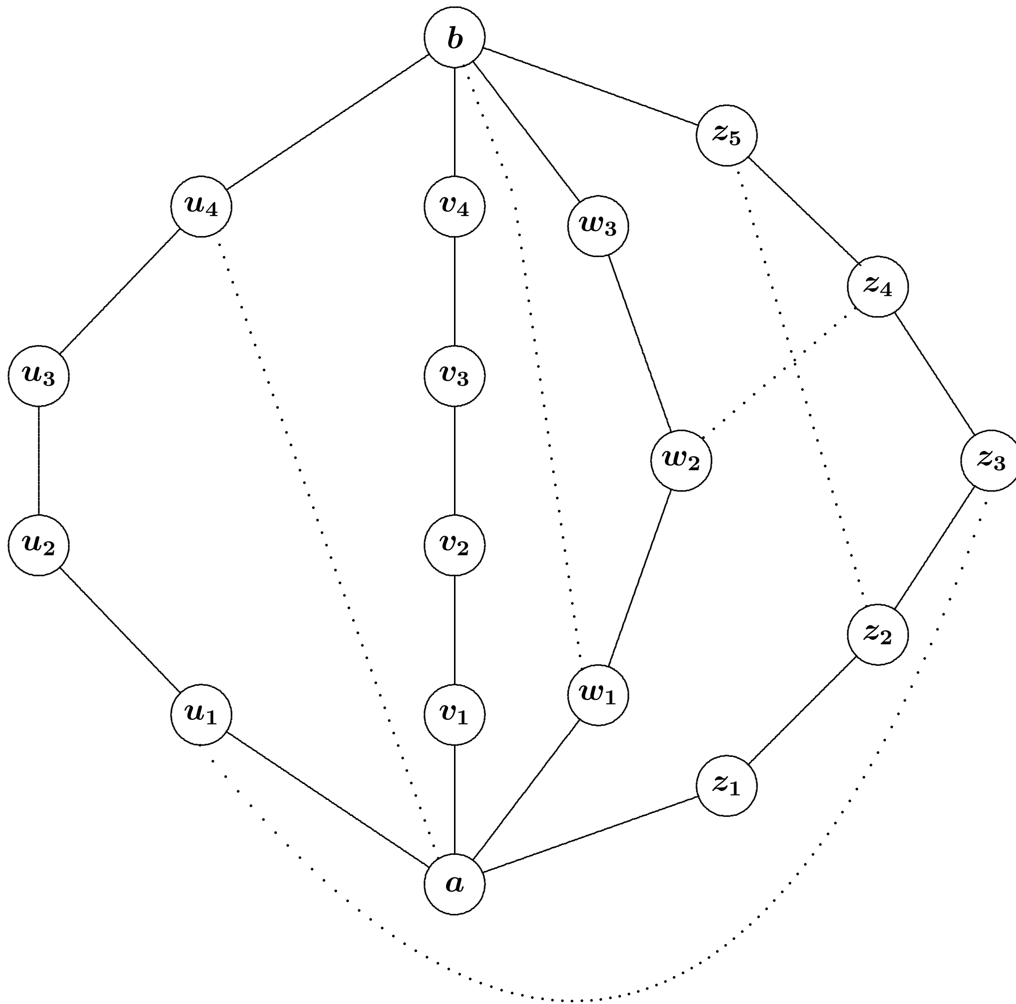


Figure 12: *Example illustrating the Menger algorithm: After finishing the step loop*

$u_1$ to $z_3$ and so on. If we call $x_i^{(r)}$ the highest vertex reached in step $r$ on path $F_i$ the following table results:

| | | | |
|---|---|---|---|
| $x_1^{(0)} = u_1$ | $x_2^{(0)} = v_1$ | $x_3^{(0)} = w_1$ | $x_4^{(0)} = z_1$ |
| $x_1^{(1)} = \mathbf{u_4}$ | $x_2^{(1)} = v_1$ | $x_3^{(1)} = w_1$ | $x_4^{(1)} = z_1$ |
| $x_1^{(2)} = u_4$ | $x_2^{(2)} = v_1$ | $x_3^{(2)} = w_1$ | $x_4^{(2)} = \mathbf{z_3}$ |
| $x_1^{(3)} = u_4$ | $x_2^{(3)} = v_1$ | $x_3^{(3)} = w_1$ | $x_4^{(3)} = \mathbf{z_5}$ |
| $x_1^{(4)} = u_4$ | $x_2^{(4)} = v_1$ | $x_3^{(4)} = \mathbf{w_3}$ | $x_4^{(4)} = z_5$ |
| $x^{(5)} = \mathbf{b}$ | | | |

The changes within a step on each path are highlighted in boldface. For simplicity the example has been chosen such that there is only one change in each step. In general, several changes per step may occur.

The result of the unravel routine is shown in figure 13. Parts of the old paths $F_1, F_2, F_3, F_4$

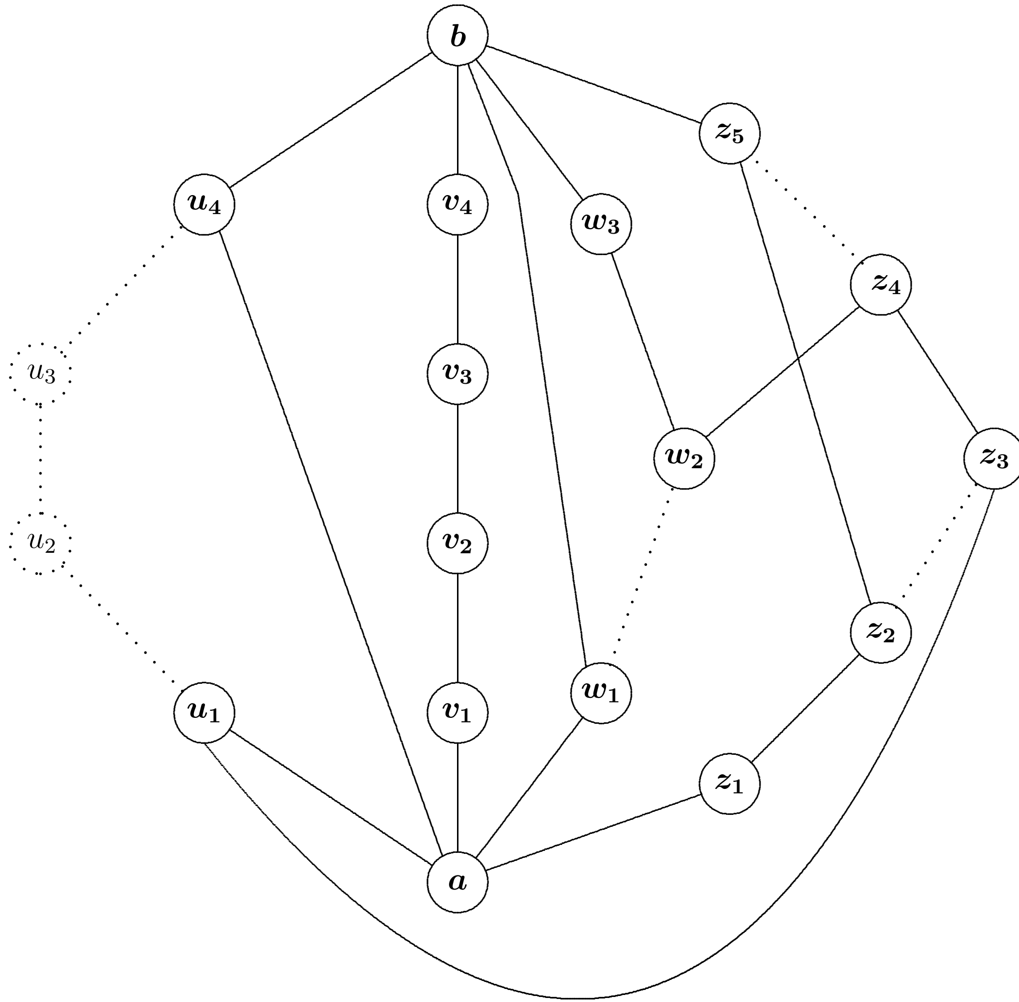


Figure 13: *Example illustrating the Menger Algorithm: After finishing the unravel routine*

not used in the new paths $G_1, G_2, G_3, G_4, G_5$ are drawn in dotted mode. The following table shows the individual steps of the unravel routine.

| *step* | $G_1$ | $G_2$ | $G_3$ | $G_4$ | $G_5$ |
|---|---|---|---|---|---|
| 5 | $(b)$ | $(b)$ | $(b)$ | $(b)$ | $(\mathbf{b}, \mathbf{w_1})$ |
| 4 | $(b)$ | $(b)$ | $(\mathbf{b}, \mathbf{w_3}, \mathbf{w_2}, \mathbf{z_4})$ | $(b)$ | $(b, w_1, )$ |
| 3 | $(b)$ | $(b)$ | $(b, w_3, w_2, z_4)$ | $(\mathbf{b}, \mathbf{z_5}, \mathbf{z_2})$ | $(b, w_1)$ |
| 2 | $(b)$ | $(b)$ | $(\mathbf{b}, \mathbf{w_3}, \mathbf{w_2}, \mathbf{z_4}, \mathbf{z_3}, \mathbf{u_1})$ | $(b, z_5, z_2)$ | $(b, w_1)$ |
| 1 | $(\mathbf{b}, \mathbf{u_4}, \mathbf{a})$ | $(b)$ | $(b, w_3, w_2, z_4, z_3, u_1)$ | $(b, z_5, z_2)$ | $(b, w_1)$ |
| 0 | $(b, u_4, a)$ | $(\mathbf{b}, \mathbf{v_4}, \mathbf{v_3}, \mathbf{v_2}, \mathbf{v_1}, \mathbf{a})$ | $(\mathbf{b}, \mathbf{w_3}, \mathbf{w_2}, \mathbf{z_4}, \mathbf{z_3}, \mathbf{u_1}, \mathbf{a})$ | $(\mathbf{b}, \mathbf{z_5}, \mathbf{z_2}, \mathbf{z_1}, \mathbf{a})$ | $(\mathbf{b}, \mathbf{w_1}, \mathbf{a})$ |

The new paths are constructed piecewise in backward direction. The construction starts with step 5, processes the steps in reverse order, and extends each time one of the paths by the corresponding path from $\widetilde{F}$. In each step the the extended path is emphasized by boldface. The two last vertices indicate in reverse order the path from $\widetilde{F}$ used as extension. Eventually all new paths are extended up to $a$ in step 0.

## 7.4 Implementation and Efficiency of Menger Algorithms

The Menger algorithms yield $M$ internally disjoint (line-disjoint) a-paths respectively f-paths. In addition, to every path they yield a separating vertex (separating line). For the implementation of the algorithms advantageously data structures for the representation of paths are introduced. The step loop is that part of the algorithms that determine their efficiency. Using breadth-first search (subsection 4.5), an efficient implementation is possible. A detailed description for internally disjoint a-paths follows. Tables 17 and 18

**STEPLOOP**

```
1    r = 0;
2    for (j = 1; j ≤ m; j++)  x_j^(0) = z_j;
3    for (j = 1; j ≤ m; j++)  x_j^(1) = x_j^(0);
4    i = BFSMENG(u);
5    while (i == 1)
6       { r = r + 1;
7         for (j = 1; j ≤ m; j++)  x_j^(r+1) = x_j^(r);
8         i = 0;
9         for (j = 1; j ≤ m; j++)
10          { for (all y with x_j^(r-1) ⪯ y ≺ x_j^(r))
11              { i = max(i, BFSMENG(y));
12                if (i == 2) break;
13              }
14            if (i == 2) break;
15      }   }
16   if (i == 0)
17      { mengerend;  /* x_1^(r), x_2^(r), ..., x_m^(r) are separating vertices */
18      }
19   if (i == 2)
20      { if (r == 0)  newpath;  /* add directly additional path */
21        if (r > 0)  unravel;  /* construct system of m + 1
22                          internally disjoint paths. */
23      };
```

Table 17: *Algorithm STEPLOOP*

```
int BFSMENG(VERTEX xstart)
1   empty queue;
2   back = 0;
3   x = xstart;
4   while (x ≠ NULL ∧ back == 0)
5       { for (all lines l incident with x)
6           { if (l uncolored)
7               { color l red;
8                 w = otherend(l, x);
9                 if (w uncolored)
10                    { color w red;
11                      enqueue(w);
12                      set l in w as back pointer;
13                    }
14                if (w brown)
15                    { color w green;
16                      set l in w as back pointer;
17                      if (w == v)
18                          { back = 2;
19                            break;
20                          }
21                      else
22                          { if (x_i^(r+1) ≺ w)
23                              { x_i^(r+1) = w;
24                                back = 1;
25                    }     }     }
26          if (back == 2) break;
27          x = dequeue;
28      }
29  return back;
```

Table 18: *Breadth-first search in the step loop of the Menger algorithm*

show an implementation of algorithm STEPLOOP and the program BFSMENG it calls on. STEPLOOP starts with a system of $m$ internally disjoint a-paths from $u$ to $v$ and ends either with $m$ a-separating vertices or yields a system of $m+1$ internally disjoint a-paths from $u$ to $v$.

BFSMENG uses colors for vertices and lines. Initially the vertices and lines of the given paths are colored brown. All other vertices and lines are uncolored. BFSMENG is called with a vertex which is used as starting vertex for the breadth-first search. This vertex is put into a queue. In the following, vertices are removed from the queue until the queue is empty. For each vertex removed from the queue, all lines it is incident with and which have not yet been visited (i.e. which are uncolored) are processed (lines 5 and 6 of BFSMENG). Note that lines lines which are colored red remain red in successive calls of BFSMENG. If the line being processed leads to a new (i.e. uncolored) vertex this vertex is (permanently) colored red and the line is stored as its back pointer.

If the line leads to a vertex on the given paths which as yet has not been visited (color brown) this vertex is colored green and again the line is stored as back pointer. In addition, it is checked whether $v$ has been reached (line 17) or a new reachable vertex has to be recorded with the path (line 22). In the first case the breadth-first search is terminated (lines 19 and 26) and a corresponding value is returned to the calling program. In the second case the maximal $u$-reachable vertex on the path is updated and a a corresponding return value is set (line 23 and 24).

**Efficency:** In general, a run of STEPLOOP will call several times the procedure BFSMENG with different starting vertices. However, line coloring ensures that altogether each line will be processed only once. The processing of an uncolored vertex also takes place only once and consists in checking and, if necessary, processing all lines it is incident with. The same is true for all vertices on the given paths if they are start vertex for the breadth-first search. If a vertex on the given paths occurs for the first time as end point in the breadth-first search the comparison in line 22 takes place. That can be implemented efficiently taking into account that the internally disjoint paths are always simple and enumerating their vertices in ascending order, i. e. from $u$ to $v$. It is easy to see that the same considerations are valid for the algorithms of theorems 7.2, 7.3, and 7.4 and we have the following proposition.

**Proposition 7.1** *The algorithms of theorems 7.1, 7.2, 7.3, and 7.4 have complexity $k \cdot O(m+n)$ where $k$ is the minimal number of separating vertices, respectively lines, $m$ is the number of lines and $n$ the number of vertices of the graph.*

## 7.5 Extensions of the Menger Theorems

### Adjacent vertices

Theorems 7.1 and 7.2 require vertices $u$ and $v$ to be non-adjacent. We are now going to examine these conditions in more detail. a-adjacent vertices $u$ and $v$ cannot be a-separated by vertices whatever the kind of neighborhood is: edge, arc from $u$ to $v$, arc from $v$ to $u$. Be there $r$ such lines. If these lines are removed from the graph theorem 7.1 can be

applied. One gets $k$ internally disjoint a-paths from $u$ to $v$ and an a-separating vertex set of $k$ vertices. Resuming: There are $k+r$ internally disjoint a-paths from $u$ to $v$ and an a-separating set of $k+r$ elements, $r$ of which are lines.

With f-paths we do not have to consider arcs from $v$ to $u$. They are not part of any system of internally disjoint simple f-paths from $u$ to $v$. If there are $r$ direct f-connections from $u$ to $v$ then we will have $k+r$ internally disjoint f-path from $u$ to $v$ and a set of $k+r$ elements which f-separates $u$ and $v$ and and $r$ of which are lines.

### Menger Theorems for Sets of Vertices

Another kind of extension of the Menger theorems results if instead of vertices $u$ and $v$ we consider non-empty vertex sets $U$ and $W$. We want to find all internally disjoint or line-disjoint paths between $U$ and $W$ and sets of elements which separate the two sets.

***Making the sets disjoint:*** All vertices in $U \cup W$ stand for a-paths/f-paths (of length 0) from $U$ to $W$. So, any set of elements a-separating (f-separating) $U$ and $W$ must contain $U \cup W$. We then proceed assuming $U \setminus W \neq \emptyset$ and $W \setminus U \neq \emptyset$. We then determine joining paths and separating elements for disjoint vertex sets and add $|U \cup W|$ to the numbers found.

***Linking disjoint vertex sets by internally disjoint (line-disjoint) a-paths (f-paths):*** To this end we construct a new graph

- The vertices of $U$ become a (new) vertex $\tilde{u}$, the vertices of $W$ become a (new) vertex $\tilde{w}$. All other vertices remain unchanged.
- Lines which join pairs of vertices in $U$ or pairs of vertices in $W$ are deleted.
- Lines which neither incide with a vertex from $U$ nor with a vertex from $W$ remain as the are.
- Lines which incide with a vertex from $U$ and with a vertex from $W$ become lines joining $\tilde{u}$ and $\tilde{w}$. Directions of arcs are retained.
- Lines which join a vertex $x \in V \setminus (U \cup W)$ with a vertex from $U$ or a vertex from $W$ become lines which join $x$ with $\tilde{u}$, respectively $\tilde{w}$. Directions of arcs are retained.

Now, theorems 7.1, 7.2, 7.3 and 7.4 including the extensions for adjacent vertices carry over easily to the new graph. The results can be interpreted in an obvious way as "Menger theorems for disjoint vertex sets."

*Algorithms:* To obtain the corresponding algorithms in is not necessary to construct the new graph explicitely. The algorithm from page 7.2 can easily be modified such that it holds for vertex sets $U$ and $W$ in the original graph. In essence, changes have to be made in the step loop. At step 1 we have to consider all vertices in $U$ as starting vertex of paths from $\widetilde{F}$. The step loop terminates and the unravel routine starts when for the first time a vertex $w \in W$ is reached. For the other algorithms similar changes have to be made.

***Linking vertex sets by $UW$-paths:*** There is a different but equivalent way to formulate Menger theorems with vertex disjoint paths. On the one hand we examine vertex sets

$U$ and $W$, not necessarily disjoint, and consider *$UW$-paths.* These are a-paths (f-paths) starting in $U$ and ending in $W$. Paths of length 0 are allowed. We also redefine the concept of separating set. A vertex set $X$ is an a-separating set (f-separating set) for $U$ and $W$ if every a-$UW$ path (f-$UW$ path) contains a vertex from $X$. Then $U \cap V \subseteq X$ must hold. In addition, such a separating set also contains vertices from $U \cup W$ which are not elements in $U \cap W$. If $u \in U$ and $w \in W$ are joined by a line then an a-separating set for $U$ and $W$ must contain either $u$ or $v$ or both.

On the other hand, a new definition of "distinctness" of paths has to be used. We consider $UW$-paths which are totally disjoint, i.e. which have no common vertices at all. Theorems 7.1 and 7.2 now become:

**Theorem 7.5** *Let $U$ and $V$ be nonempty vertex sets of a general graph. The smallest cardinality of an a-separating vertex set for $U$ and $W$ equals the greatest cardinality of a system of disjoint a-$UW$ paths in the graph.*

**Theorem 7.6** *Let $U$ and $V$ be nonempty vertex sets of a general graph. The smallest cardinality of an f-separating vertex set for $U$ and $W$ equals the greatest cardinality of system of disjoint f-$UW$ paths in the graph.*

**Proposition 7.2** *Theorems 7.1 and 7.5 are equivalent. Theorems 7.2 and 7.6 are equivalent.*

**Proofs:** The proof uses additional vertices $\tilde{u}$ and $\tilde{v}$. For more details see [Stie2006]. □

*Algorithms:* The graph extended by vertices $\tilde{u}$ and $\tilde{v}$ is explicitely constructed and then the Menger algorithms are applied.

**Remark 7.2** 1. Of course, in general finding internally disjoint paths from $U$ to $W$ yields different results than finding disjoint $UW$-paths.

2. Since line-disjoint paths may have vertices including end vertices in common, it does not make sense to consider different kinds of line-disjoint $UW$-paths. If $U \cap W \neq \emptyset$ the separating set of lines must be complemented by the vertices of the intersection.

□

## 7.6 The Structure of Menger Separating Sets

According to the theorems in subsection 7.2 the maximal number of internally disjoint (line-disjoint) paths between two vertices equals the minimal number of vertices (lines) necessary to separate the two vertices. Neither the paths nor the separating vertex (line) sets are uniquely determined. We call such paths a *system of Menger paths* and the vertex (line) sets *Menger separating sets.* We shall also speak of a family of Menger paths. For the sake of simplicity we restrict the following considerations to a-paths and separating sets of vertices. The generalization to f-paths and separating sets of lines is easy. As in subsection 7.2 we consider all paths to be simple. In this subsection we examine the structure of Menger separating sets. In subsection 7.7 we present an algorithm to determine the Menger separating sets.

A single path from $u$ to $v$ is called a *Menger path* if it can be complemented by other paths such that a system of Menger paths results. A single vertex is called a *Menger vertex* for $u$ and $v$ if it can be complemented by other vertices such that a Menger separating set for $u$ and $v$ results. A first result concerning the structure of Menger separating sets follows directly from the proof of theorem 7.1.

**Proposition 7.3** *Let $P_1, P_2, \ldots, P_k$ be the Menger paths and $v_1, v_1, \ldots, v_k$ the Menger vertices found by the algorithm from page 47. Then there is no Menger vertex on any of the $P_i$ ( $i = 1, \ldots, k$) nearer to $u$ than $x_i$.*

The following two lemmata are useful. For the proofs see again [Stie2006].

**Lemma 7.1** *Let $u$ and $v$ be non-adjacent vertices in a weakly connected general graph. Let $P_i$ ($i = 1, \ldots, k$) and $Q_i$ ($i = 1, \ldots, k$) be families of Menger paths from $u$ to $v$ and let $v_i$ ($i = 1, \ldots, k$) a Menger separating set for $u$ and $v$. We construct a new family of paths from $u$ to $v$ starting with the $P_i$ from $u$ to $v_i$ and continuing from $v_i$ to $v$ along $Q_i$. Then the so composed paths are a family of Menger paths.*

**Lemma 7.2** *Let $u$ and $v$ be non-adjacent vertices of a weakly connected general graph. Let $k$ be the number of vertices of a Menger separating set.*

1. *If $P$ is a Menger path from $u$ to $v$ and if $x$ and $y$ are Menger vertices on $P$ then $y \neq v_i$ for all Menger separating sets $x, v_1, \ldots, v_{k-1}$.*

2. *Let $P$ be a Menger path from $u$ to $v$ and $x$ and $y$ Menger vertices on $P$. Then if on $P$ vertex $x$ is nearer to $u$ than vertex $y$ then $x$ is nearer to $u$ than $y$ on all Menger paths passing through both vertices.*

3. *If $x$ and $y$ are Menger vertices on the Menger path $P$ and if there is on $P$ no Menger vertex between $x$ and $y$ then there is no Menger vertex between $x$ and $y$ on any Menger path containing both vertices.*

**Remark 7.3** Lemma 7.2, item 3, remains valid if either $u = x$ or $v = y$. □

**Remark 7.4** From lemma 7.2, item 3, follows that a chain of successive Menger vertices of a Menger path is repeated in the same order on any other Menger path through all vertices of the chain. It is not true, however, that a chain consisting of more than two vertices of a Menger path is repeated on all other Menger paths containing the first and the last vertex of the chain. □

Lemmata 7.1 and 7.2 suggest the following definition of a partial order on the set of Menger vertices fur $u$ and $v$.

**Definition 7.1** *Let $u$ and $v$ two non-adjacent vertices of a general graph and $x$ and $y$ two Menger vertices for $u$ and $v$. We define $x \preceq_{u,v} y$ ("$x$ lies before $y$ in direction from $u$ to $v$"),[4] if $x = y$ or if there is a Menger path on which $x$ is nearer to $u$ than $y$.*

[4] The partial order defined here is not be confounded with the partial order $\preceq$ defined on page 47.

In the following we always assume the direction from $u$ to $v$ and omit the subscript $_{u,v}$. In fact, definition 7.1 generates a partial order as stated by the following proposition.

**Proposition 7.4** *Let $u$ and $v$ be two non-adjacent vertices o a weakly connected general graph. The relation $x \preceq y$ defines a partial ordering on the Menger vertices corresponding to $u$ and $v$.*

**Remark 7.5** We complete the partial order $\preceq$ by adding $u$ as smallest and $v$ as greatest element. The symbols $\prec$, $\succeq$, $\succ$ have their usual meaning. □

## 7.7 An Algorithm for Finding the Menger Vertices

We present an algorithm for finding the Menger separating sets corresponding to two non-adjacent vertices of a weakly connected general graph. Again we restrict ourselves to a-paths and to separating sets of vertices. The algorithm is described informally using an example. The exact formulation of the algorithm as well as proofs are left to the reader. The example is depicted in figure 14. It shows parts of an undirected graph with

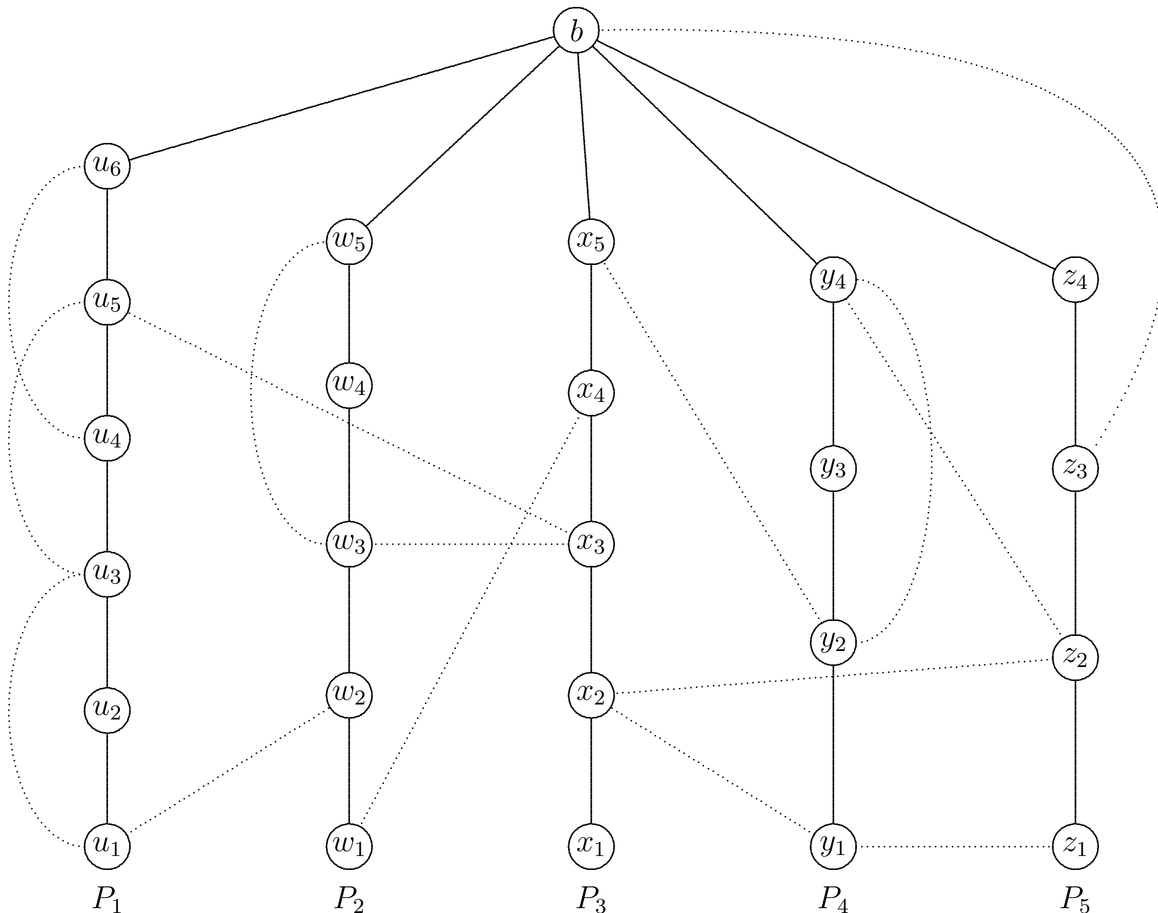


Figure 14: *Example of Menger vertices*

5 internally disjoint Menger paths $P_1, P_2, P_3, P_4.P_5$. These lead from the starting vertex $a$ (not shown) to the the end vertex $b$. The algorithm runs in 4 steps.

**Step 1:** Using the algorithm from subsection 7.2 we find the Menger paths $P_1, P_2, P_3$, $P_4, P_5$ and the Menger separating vertices $u_1, w_1, x_1, y_1, z_1$. According to proposition 7.3 there are no Menger vertices between $a$ and $u_1, w_1, x_1, y_1, z_1$. Therefore, the initial parts of the Menger paths are not shown in the figure. According to proposition 7.4 these Menger vertices are nearest to $a$ on any other family of Menger paths $Q_1, Q_2, Q_3, Q_4, Q_5$. All Menger vertices must lie on one of the $P_i$. Those which lie on the same path are ordered by $\preceq$.

**Step 2:** We consider the Menger paths

$$\begin{aligned} P_1 &= a, \ldots, u_1, u_2, u_3, u_4, u_5, u_6, b \\ P_2 &= a, \ldots, w_1, w_2, w_3, w_4, w_5, b \\ P_3 &= a, \ldots, x_1, x_2, x_3, x_4, x_5, b \\ P_4 &= a, \ldots, y_1, y_2, y_3, y_4, b \\ P_5 &= a, \ldots, z_1, z_2, z_3, z_4, b \end{aligned}$$

and proceed as we have done with the step loop in subsection 7.2. Again we define $W$ to be the set of vertices lieing on one of the paths $P_i$. The vertices between $a$ and $u_1$ ($w_1$, $x_1$, $y_1$, $z_1$) are not included. $\widetilde{F}$ is defined again as the set of simple open a-paths beginning and ending in vertices from $W$, with no internal vertex in $W$ and no line in common with the $P_i$. Some paths from $\widetilde{F}$ are shown in figure 14 as dotted lines. We find three classes of paths:

1. Paths which join two vertices of the starting Menger set, as for instance the path from $y_1$ to $z_1$. These paths can be discarded since $u_1, w_1, x_1, y_1, y_1$ has already been established as Menger separating set.

2. Paths like the path from $u_3$ to $u_5$ having both end points on the same $P_i$. We use these paths in step 3 to find the “candidates”.

3. The remaining paths – e.g. the path from $u_5$ to $x_3$ – will be used in step 4 to find the final list of Menger vertices.

**Step 3:** On every path $P_i$ we find all its *candidates*. A vertex $u$ on $P_i$ is a candidate if

1. $v_i \prec u \prec b$ where $v_i$ is the first separating vertex on $P_i$, for instance $v_3 = x_1$.

2. There is no path in $\widetilde{F}$ with both, starting point $x$ and end point $y$ on $P_i$ and $x \prec u \prec y$.

It is easy to see that only candidates can be Menger vertices. If we assume that there are no other paths in $\widetilde{F}$ than the ones shown in figure 14 we find the following candidates:

On $P_1$: $u_3, u_6$. On $P_2$: $w_2, w_3, w_5$. On $P_3$: $x_2, x_3, x_4, x_5$
On $P_4$: $y_2, y_4$ On $P_5$: $z_2, z_3$.

**Step 4:** This is the central part of the algorithm. On each path $P_i$ we find all candidates which are Menger vertices. In addition, we find all Menger separating sets $v_1, v_2, \ldots, v_k$ ($v_i \in P_i$) which are a *minimal Menger separating set* for a Menger vertex. These allow the construction of a complete list of all Menger separating sets. They also can be used to test whether a given set of Menger vertices is a Menger separating set. We proceed in the same way as with the step loop in subsection 7.2 and determine *escapes*. We explain this using figure 14 as an example.

We use a token moving technique. Initially a token is put on each vertex of the Menger separating set $(u_1, w_1, x_1, y_1, z_1)$. The the tokens are moved along $P_i$ in direction to $b$. Let $u_3$ be the first candidate to be tested, that is we move the token from $u_1$ to $u_3$. This opens an escape from $u_1$ to $w_2$. To block this escape we must move the token on $P_2$ from $w_1$ to $w_2$. This in turn opens an escape from $w_1$ to $x_4$. $P_3$'s token must be moved from $x_1$ to $x_4$. After that three new escapes are open namely from $x_3$ to $u_5$, from $x_3$ to $w_3$, and from $x_2$ to $z_2$. The path from $x_2$ to $y_1$ is also open. However, it is not an escape since no new vertex is is reached on $P_4$. To block the new escapes the corresponding tokens have to be put on $u_6$, $w_3$, and $z_2$. Finally there remain no new escapes and we have found the Menger separating set $(u_6, w_3, x_4, y_1, z_2)$. It has the following properties:

1. $u_6$ is the first (and in this only) candidate on $P_1$ which is a Menger vertex.

2. On no path $P_i$ $(i = 2, 3, 4, 5)$ any vertex which is nearer to $a$ than $w_3$, respectively $x_4$, $y_1$, $z_2$ can constitute a Menger separating set together with $u_6$. We call $u_6, w_3, x_4, y_1, z_2$ the minimal Menger separating set for $u_6$.

In the same manner we find all minimal Menger separating set. The result is shown in table 19. Each 5-tuple is a minimal Menger separating set. The vertex printed in boldface

| | | | | |
|---|---|---|---|---|
| $(\mathbf{u_6}, w_3, x_4, y_1, z_2)$ | $(u_6, \mathbf{w_5}, x_4, y_1, z_2)$ | $(u_6, w_3, \mathbf{x_5}, y_1, z_2)$ | $(u_6, w_3, x_5, \mathbf{y_4}, z_2)$ | $(u_6, w_3, x_5, y_4, \mathbf{z_3})$ |
| | $(u_6, \mathbf{w_3}, x_4, y_1, z_2)$ | $(u_6, w_3, \mathbf{x_4}, y_1, z_2)$ | $(u_1, w_1, x_2, \mathbf{y_2}, z_1)$ | $(u_1, w_1, x_1, y_1, \mathbf{z_2})$ |
| | | $(u_1, w_1, \mathbf{x_3}, y_1, z_2)$ | | |
| | | $(u_1, w_1, \mathbf{x_2}, y_1, z_1)$ | | |

Table 19: *Minimal Menger separating sets*

is the vertex to which it is minimal. □

In this way, for two non-adjacent vertices $a$ and $b$ we find all Menger vertices. However, we do not find all Menger separating sets but only the minimal ones. For instance, if the graph of figure 14 consists of the paths $P_1, P_2, P_3, P_4, P_5$, edges joining $u_1, w_1, x_1, y_1, z_1$ directly to $a$, and no other lines and vertices then every 5-tuple with one vertex from each path is a Menger separating set. However, the minimal Menger separating set of a vertex consists of this vertex and first vertices on all other paths.

In general, the number of Menger separating sets for $a$ and $b$ will grow very fast with the number of Menger vertices. Assume that in spite of that fact we want a list of all Menger separating sets. As a first try we check all k-tuples containing one vertex from each path if they separate $a$ and $b$. This could be done starting a depth-first search in $a$

which avoids the vertices to be tested. If $r_i$ is the number of Menger vertices on path $P_i$ the complexity is

$$r_1 \cdot r_2 \cdot \ldots \cdot r_k \cdot O(m+n) \leq \left(\frac{n}{k}\right)^k \cdot O(m+n)$$

The above example shows that in fact each k-tuple could be a Menger separating set. Therefore, the complexity can only be reduced if it is possible to make the check faster than a depth-first search. That is indeed possible and is a consequence from the following proposition.

**Proposition 7.5** *Let $P_1, P_2, \ldots, P_k$ be a system of Menger paths from $a$ to $b$. Let $x_1, x_2, \ldots, x_k$ be a k-tuple of Menger vertices with one vertex on each path. Then $x_1, x_2, \ldots, x_k$ is a Menger separating set if and only if for all $i$ and all $j$ with $i, j = 1, 2, \ldots, k$ holds*

$$x_{ij} \preceq x_j \tag{3}$$

*where $x_{ij}$ is the vertex on $P_j$ in the minimal Menger separating set corresponding to $x_i$.*

**Proof:** If condition 3 does not hold then the construction above shows that $x_1, x_2, \ldots, x_k$ cannot be a separating set.
Let condition 3 be true. Then the assumption that $x_1, x_2, \ldots, x_k$ is not a Menger separating set leads to a contradiction. Let $P$ be a path from $a$ to $b$ which avoids $x_1, x_2, \ldots, x_k$. $a$ is a vertex which lies before all $x_i$ on $P_i$. $b$ lies after all $x_i$. On $P$ there must be a last vertex which lies on a path $P_i$ *before* $x_i$. Let $x$ be the next vertex which P and one of the $P_j$ have in common. $x$ lies on $P_j$ *after* $x_j$. It is not possible that $x = b$, for then $x_i$ were not a Menger vertex. Now, from the construction of the minimal Menger separating set for $x_i$ follows $x \preceq x_{ij}$. That implies $x_j \prec x_{ij}$ and this is a contradiction to the initial assumption. □

With proposition 7.5 we can implement a fast check to decide whether a given $m$-tuple is a Menger separating set. We store with each Menger path all its Menger vertices using hash tables or search trees. Together with each vertex we store its minimal Menger separating set and a numerical key for fast checking of $x \prec y$. With this data structure we need only $O(1)$ or at most $O(\ln n)$ steps to check whether $x_1, x_2, \ldots x_m$ is a Menger separating set.

**Remark 7.6** For a given system of Menger paths, two Menger vertices $x$ and $y$ for which $x \prec y$ holds may well lie on different Menger path. From the Menger paths resulting from algorithm 7.1 the relation $\prec$ cannot be determined completely. It seems to be difficult to find an efficient algorithm which decides $x \prec y$ in general. Perhaps it is a hard problem. □

## Remarks and Literature

We have entitled this section "Menger structures" because the first proof of theorem 7.1 was given by Karl Menger [Meng1927]. Menger's proof uses terminology and arguments of general curve theory and is of topological nature. Its connection to graph-theoretic and

combinatorial problems as formulated in theorem 7.1 is not easily discerned. In the sequel numerous and quite different proofs appeared. For examples see Dirac [Dira1966], Harary [Hara1969], Sachs [Sach1970], Tutte [Tutt1984], Thulasiraman/Swamy [ThulS1992]. Diestel [Dies2000a] alone presents three proofs. A very nice and clear proof was given by Grünwald [Grun1938], see also Wagner [Wagn1970]. This proof has a definitely algorithmic structure. It served as basis for the proof and algorithm of theorem 7.1. Grünwald's proof served also as basis for a first version of the algorithm in Stiege [Stie1998].
In spite of the formal similarity with separating vertex sets, separating line sets have been investigated only several decades later and theorem 7.3 was not formulated and proved until the fifties of the last century: Ford/Fulkerson [FordF1956], Elias/Feinstein/Shannon [EliaFS1956]. The transfer of Menger theorems from a-paths to f-paths is very simple and is barely discussed in the literature. For extensions of the Menger theorems as examined in subsection 7.5 see also Harary [Hara1969]. The structure of Menger separating sets (subsection 7.6) seem not have found interest and is not dealt with in the textbook literature.

# 8 Higher Decompositions

## 8.1 $k$-Reachability and $k$-Linereachability

In section 3 we introduced weak and strong connectedness. We started with (mutual) reachability along an a-path, respectively mutual reachability along an f-path. We generalize now these concepts to $k$ "different" paths. As in section 7, this means that they are internally disjoint, respectively line-disjoint in each direction. However, paths from $u$ to $v$ and paths from $v$ to $u$ may overlap arbitrarily.

**Definition 8.1** *1. In a general graph, vertex $v$ is* $k$-a-reachable *from vertex $u$ if there are $k$ internally disjoint a-paths from $u$ to $v$.*
*In a general graph, vertex $v$ is* $k$-a-linereachable *from vertex $u$ if there are $k$ line-disjoint a-paths from $u$ to $v$.*

*2. In a general graph, vertex $v$ is* $k$-f-reachable *from vertex $u$ if there are $k$ internally disjoint f-paths from $u$ to $v$.*
*In a general graph, vertex $v$ is* $k$-f-linereachable *from vertex $u$ if there are $k$ line-disjoint f-paths from $u$ to $v$.*

The aim of this section is to determine maximal vertex sets respectively maximal subgraphs all whose vertices are mutually $k$-reachable. We first look at a-paths. Since Linereachability is simpler than reachability we start with the former. Finally, the results are carried over to f-paths.

## 8.2 $k$-a-Lineconnectedness

**Definition 8.2** *Let $G$ be a general graph and $U$ a set of vertices in $G$ with at least 2 vertices. $U$ is termed* $k$-a-lineconnected in $G$, *if for all vertices $u, v \in U$ with $u \neq v$ vertex*

*$v$ is $k$-a-linereachable from vertex $u$ in $G$.*
*A subgraph $H$ of $G$ is termed $k$-a-lineconnected if its vertex set $V(H)$ is $k$-a-lineconnected in $H$.*

Note that in a $k$-a-lineconnected subgraph the line-disjoint paths must run within the subgraph.

**Remark 8.1**

1. Every set of vertices consisting of at least two vertices can be made a $k$-a-lineconnected general graph. It may be necessary to include multiple lines.
2. Let $G$ be a general graph having at least two vertices.
   a. $G$ is 0-a-lineconnected.
   b. $G$ is 1-a-lineconnected if and only $G$ is weakly connected.
   c. $G$ is 2-a-lineconnected if and only if $G$ is a subcomponent. See section 5.
3. If we admitted graphs with only one vertex in definition 8.2 several difficulties would arise.
   a. A graph with one single vertex is $k$-a-lineconnected for *every* $k$. That does not make sense.
   b. We could omit the condition $u \neq v$ in definition 8.2. On the one hand, a graph with one vertex and a loop would then be 1-a-lineconnected. On the other hand, according to the definitions in section 5 it would be a subcomponent, i.e. twofold a-lineconnected. More important, theorem 7.3, which is essential for determining $k$-a-lineconnected graphs, only holds for two different vertices.
4. If $U$ is a $k$-a-lineconnected vertex set in $G$ then $U$ need not be $k$-a-lineconnected in $[U]$, the subgraph generated by $U$. If, for instance, the vertices in $U$ are pairwise non-adjacent then $G[U]$ consists only of isolated vertices. On the other hand, $U$ may be $k$-a-lineconnected in a subgraph $H$ which is a proper subgraph of $G[U]$.
5. If $I$ is $k$-a-lineconnected subgraph of $G$ then the vertex set $V(I)$ is $k$-a-lineconnected in every subgraph $H$ with $I \subseteq H \subseteq G$. □

The most important tool for studying $k$-a-lineconnected vertex sets is theorem 7.3. Some useful lemmata and corollaries derived from this theorem follow.

**Lemma 8.1** *Let $U$ be a $k$-a-lineconnected vertex set in the general graph $G$. If a new vertex $\tilde{v}$ and at least $k$ lines joining $\tilde{v}$ with vertices in $U$ are added to $G$ then $U \cup \{\tilde{v}\}$ is $k$-a-lineconnected in the extended graph $\widetilde{G}$.*

Note that the new lines may well be multiple lines.

**Corollary 1** *Let $U$ be a $k$-a-lineconnected vertex set in a general graph $G$. Let $W \subset U$ be a vertex set with $1 \leq |W| \leq k$ and let $v \in U \setminus W$. The vertices $w \in W$ are mapped to positive natural number $b(w)$ such that $\sum_{w \in W} b(w) = k$. Then there are $k$ line-disjoint a-path in $G$ which start in in $v$ and end in $W$, namely $b(w)$ in $w$.*

**Corollary 2** *Let $U$ be a $k$-a-lineconnected vertex set in a general graph $G$. Let $W_1, W_2 \subset U$ be disjoint vertex sets with $1 \leq |W_1|, |W_2| \leq k$. To each $w \in W_1$ we assign a positive natural number $b_1(w)$ such that $\sum_{w \in W_1} b(w) = k$. An analogous assignment $b_2(w)$ is given for $W_2$. Then there are $k$ line-disjoint a-paths from $W_1$ to $W_2$ in $G$. $b_1(w)$ of these start in $w \in W_1$ and $b_2(w)$ end in $w \in W_2$.*

Frequently we do not want to add a new vertex to a $k$-a-lineconnected vertex set but a vertex which already exists in the graph. We may use the following lemma for that purpose.

**Lemma 8.2** *Let $U$ be a $k$-a-lineconnected vertex set in the general graph $G$. If $v \notin U$ is a vertex of $G$ then $U \cup \{v\}$ is $k$-a-lineconnected if and only if $v$ is joined to vertices in $U$ via $k$ line-disjoint a-paths.*

**Lemma 8.3** *Let $U$ be a $k$-a-lineconnected vertex set in a general graph $G$. We extend $U$ (and $G$) by $k$ new vertices. In addition we add new edges in such a way that each new vertex is linked to every other new vertex by exactly one new edge and that it is also linked to a vertex in $U$. Then the extended vertex set $\widetilde{U}$ is $k$-a-lineconnected in the extended graph $\widetilde{G}$.*

**Proposition 8.1** *Let $U$ and $W$ be $k$-a-lineconnected ($k \geq 1$) vertex sets of a general graph $G$. If $U \cap W \neq \emptyset$ then $U \cup W$ is $k$-a-lineconnected, too.*

**Proof:** We show that in $G$ there are $k$ line-disjoint a-paths from every $u \in U \setminus W$ to every $w \in W \setminus U$. Let $z \in U \cap W$. In $G$, $k'$ lines with $k' < k$ can neither separate $u$ nor $w$ from $z$. Hence, they cannot separate $u$ from $w$. According to theorem 7.3 then there are $k$ line-disjoint a-paths from $u$ to $w$. □

We call a $k$-a-lineconnected vertex set maximal if no proper superset is $k$-a-lineconnected. The following theorem is an immediate consequence of proposition 8.1

**Theorem 8.1** *Maximal $k$-a-lineconnected vertex sets in a general graph are uniquely determined.*

The union of two $k$-a-lineconnected vertex sets of a general graph may be $k$-a-lineconnected even if the sets are disjoint as the following proposition states.

**Proposition 8.2** *Let $U$ and $W$ be disjoint $k$-a-lineconnected vertex sets in a general graph $G$. If and only if $U$ and $W$ are linked via at least $k$ line-disjoint a-paths $U \cup W$ is $k$-a-lineconnected in $G$, too.*

**Proof:** If $k$ line-disjoint a-paths do not exist then $U \cup W$ is not $k$-a-lineconnected in $G$.

We assume that $k$ line-disjoint a-paths joining $U$ and $W$ exist. Then the assumption that $u \in U$ is separated from $w \in W$ by $k' < k$ lines lead to a contradiction. In fact: At least one of the $k$ joining paths does not contain a separating line. Let $u' \in U$ be its starting point and $w' \in W$ its end point. $u$ cannot be a-separated from $u'$ by $k'$ lines. The same holds for $w$ and $w'$. Hence, here is always an a-path from $u$ to $w$. □

We now examine $k$-a-lineconnected subgraphs $H$ and $I$ of a general graph $G$. In the subgraph $H+I$ [5] the vertex set $V(H)$ as well as the vertex set $V(I)$ are $k$-a-lineconnected. If $H \cap I \neq \emptyset$ holds then according to proposition 8.1 also $V(H) \cup V(I) = V(H+I)$ is $k$-a-lineconnected in $H+I$. Therefore, the following proposition analogous to proposition 8.1 holds.

**Proposition 8.3** *Let $H$ and $I$ be $k$-a-lineconnected ($k \geq 1$) subgraphs of a general graph $G$. If $H \cap I \neq \emptyset$ holds then also $H+I$ is a $k$-a-lineconnected subgraph of $G$.*

We call a $k$-a-lineconnected subgraph of $G$ maximal if every extension leads to a subgraph which is not $k$-a-lineconnected anymore. We have the following theorem.

**Theorem 8.2** *Maximal $k$-a-lineconnected subgraphs of a general graph are uniquely determined.*

For maximal $k$-a-lineconnected subgraphs we introduce a new name.

**Definition 8.3** *A maximal $k$-a-lineconnected subgraph of a general graph is called* $k$-a-linecomponent *of the graph.*

The subgraph $H+I$ may be $k$-a-lineconnected even if $H$ and $I$ are disjoint. The following proposition holds.

**Proposition 8.4** *Let $H$ and $I$ be disjoint $k$-a-lineconnected subgraphs of a general graph. If $H$ and $I$ are directly linked via at least $k$ lines then $H+I$ is $k$-a-lineconnected, too.*

**Example 8.1** Proposition 8.4 is in some sense a restriction of proposition 8.2. Two disjoint $k$-a-lineconnected subgraphs $H$ and $I$ may well be linked via $k$ or more one-disjoint a-paths without having a common $k$-a-lineconnected supergraph in $G$. That is true even if all vertices have degree at least $k$. Figure 15 shows an example. The graph is undirected, 2-a-lineconnected and has minimal degree 3. The subgraphs $H_1$ and $H_2$ are 3-a-linecomponents and are linked together via three line-disjoint a-paths. However, the graph is not 3-a-lineconnected, for the subgraphs lieing between $H_1$ and $H_2$ are separated form the remaining graph by 2 edges. □

### Decomposition into a-linecomponents

Every $k$-a-lineconnected subgraph of a general graph is also $(k-1)$-a-lineconnected. That is, each $k$-a-linecomponent is contained in a $(k-1)$-a-linecomponent. A $(k-1)$-a-linecomponent $EC$ contains 0 or more $k$-a-linecomponents. If there is no $k$-a-linecomponent then the decomposition of $EC$ into a-linecomponents ends. Alternatively there are in $EC$ $k$-a-linecomponents and possibly also lines which do not belong to any $k$-a-linecomponent. The subgraph generated by these lines is called the *$k$-a-colinecomponent* of $EC$. If there is in $EC$ exactly one $k$-a-linecomponent then the $k$-a-colinecomponent may be empty

[5] $H+I$ is the subgraph of $G$ which is generated by the vertex set $V(H) \cup V(I)$.

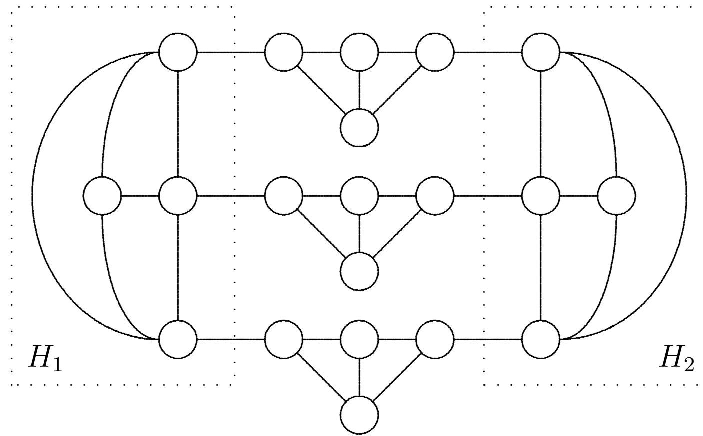


Figure 15: *A graph with two 3-a-linecomponents linked by three line-disjoint a-paths*

and $EC$ is a $k$-a-linecomponent, too. However, there may exist also a non-empty $k$-a-colinecomponent, for instance, if $EC$ contains vertices of degree smaller than $k$.

If there are at least two $k$-a-linecomponents in $EC$, then these are disjoint (proposition 8.3). However,they are linked in $EC$ by a-paths. Thus in this case there must exist lines in $EC$ which do not belong to any $k$-a-linecomponent. The $k$-a-colinecomponent of $EC$ is not empty in this case. As shown in example 8.1 the $k$-a-linecomponents may be joined by $k$ line-disjoint a-paths which proceed in the $k$-a-colinecomponent. However, not all joining paths are allowed to have length 1 (proposition 8.4).

## 8.3 Simple Kernel

We now come to connectedness by internally disjoint paths. In a manner analogous to definition 8.2, in a general graph $G$ we could call a vertex set $U$ consisting of at least two elements $k$-a-connected if for all $u, v \in U$ with $u \neq v$ vertex $v$ is $k$-a-reachable from vertex $u$. A subgraph $H$ of $G$ would be called $k$-a-connected if $V(H)$ were $k$-a-connected in $H$. However, this raises difficulties. If in $G$ each vertex is joined $k$-fold to every other vertex by multiple lines, then according to this definition we would have $k$-a-connectedness. The graph could be called $k$-a-complete. If this completeness condition is not fulfilled, how many vertices must have $G$ as a minimum to be $k$-a-connected according to the proposed definition?

It is easy to answer this question with the help of Menger's theorem 7.1. Let $\text{na}(u, v)$ be the number of direct a-links between $u$ and $v$ $(u \neq v)$. As a minimum we need $\max(k - \text{na}(u, v), 0)$ vertices to a-separate $u$ and $v$ after removing all lines joining $u$ and $v$ directly. Hence, the number $n$ of vertices of $G$ is at least $\max_{u \neq v}(\max(k - \text{na}(u, v), 0)) + 2$. For $k$-a-complete $G$ this results in $n \geq 2$. If there are two vertices which are not a-neighbors the minimum number is $n \geq k + 2$. To avoid these difficulties and special cases caused by multiple lines we examine $k$-a-connectedness only for simple graphs and then extend the results to arbitrary general graphs using the concept of *simple kernel*.

**Definition 8.4** *The* simple kernel *of a general graph $G$ is a simple graph with the same vertex set. Two of its vertices are joined by an edge if and only if they are joined in $G$ by a line.*

In a strict sense, the simple kernel of $G$ is not uniquely defined. The set of its edges is not exactly defined. We will not define it, but require the lines sets of $G$ and the edge set $E$ of the simple kernel to be disjoint. The simple kernel of $G$ neither is a subgraph of $G$ nor does it belong to the orientation class of $G$. Especially, it is not the complete disorientation of $G$. If in the simple kernel of $G$ we have $k$ internally disjoint a-paths then there are always $k$ corresponding internally disjoint a-paths in $G$. These traverse the same vertices in the same order but the traversed lines are in general not uniquely determined. The $k$-a-components of the simple kernel which we will examine in the next subsection yield in $G$ uniquely defined maximal subgraphs. We just take the subgraphs generated from the corresponding vertex sets. These have the same connectedness properties and will also be called $k$-a-components.

## 8.4 $k$-a-Connectedness

If in a simple graph there are $k$ internally disjoint a-paths starting in vertex $v$ then $v$ has at least $k$ neighbors and the graph at least $k+1$ vertices. In such a graph we could call a vertex set $k$-a-connected if every pair of distinct vertices is joined by at least $k$ internally disjoint a-paths. In this case, however, it could happen that $U$, $W$, and $U \cap W$ are $k$-a-connected but $U \cup W$ is not. To guarantee unique extensibility the vertex sets are required to be sufficiently large. So, we arrive at the following definition.

**Definition 8.5** *Let $G$ be a simple graph and $U$ a set of its vertices with at least $k+1$ elements. $U$ is called* $k$-a-connected *in $G$, if for all $u, v \in U$ with $u \neq v$ vertex $v$ is $k$-a-reachable from $u$ in $G$.*
*A subgraph $H$ of $G$ is called $k$-a-connected if its vertex set $V(H)$ is $k$-a-connected in $H$.*

Note that in a $k$-a-connected subgraph the internally disjoint paths must run in the subgraph. In addition, it has at least $+1$ vertices. Remark 8.1 caries over almost literally to $k$-a-connectedness with subcomponents substituted by biblocks.

As with $k$-a-lineconnectedness, the most important tool with $k$-a-connectedness is the corresponding Menger theorem, namely theorem 7.1. A first useful consequence of this theorem is the following well known proposition.

**Proposition 8.5** *In a simple graph $G$ let $U$ be a vertex set of at least $k+1$ vertices. $U$ is $k$-a-connected in $G$ if and only if every pair of non-adjacent vertices in $U$ is joined by at least $k$ internally disjoint a-paths.*

**Proof:** If $U$ is $k$-a-connected in $G$ then the condition is satisfied.
Let the condition be true. It remains to show that all pairs of adjacent vertices $u, v \in U$ are joined by $k$ internally disjoint a-paths. Assume $u, v \in U$ being adjacent vertices and linked in $G$ by not more than $k' < k$ internally disjoint a-paths. $k' - 1$ of these have a

length greater 1. Let $l$ be the edge joining $u$ and $v$. From theorem 7.1 follows that there are $k' - 1$ vertices separating $u$ and $v$ in $G - l$. Since $|U| \geq k + 1$ there exists at least one vertex $v'$ in $U$ which is different from $u$, $v$, and the separating vertices. In $G - l$, $v'$ is separated form $u$ or from $v$ by the separating vertices. For otherwise these would not separate $u$ and $v$. Assume the separating vertices separate $v'$ from $u$. Then $v'$ and $u$ are not adjacent neither in $G - l$ nor in $G$. By hypothesis there are in $G$ $k$ internally disjoint a-paths from $v'$ to $u$. However, in $G - l$ there are at most $k' - 1$ such paths, hence in $G$ at most $k' - 1 + 1 < k$ and that is a contradiction. □

The following lemmata are quite similar to the lemmata 8.1, 8.2, and 8.3.

**Lemma 8.4** *Let $U$ be a k-a-connected vertex set in a simple graph $G$. If a new vertex $\tilde{v}$ and at least $k$ new lines joining $\tilde{v}$ with pairwise distinct vertices in $U$ are added to $G$ then $U \cup \{\tilde{v}\}$ is k-a-connected in the extended graph $\widetilde{G}$.*

**Corollary 1** *Let $U$ be a k-a-connected vertex set in the simple graph $G$. Let $W \subset U$ be a set of $k$ vertices and $v \in U \setminus W$. Then there are $k$ internally disjoint a-paths from $v$ to pairwise distinct vertices in $W$.*

**Corollary 2** *Let $U$ be a k-a-connected vertex set in the simple graph $G$. Let $W_1$ and $W_2$ be disjoint subsets of $U$ containing $k$ vertices each. Then there are $k$ vertex-disjoint a-paths starting in $W_1$ and ending in $W_2$.*

**Lemma 8.5** *Let $U$ be a k-a-connected vertex set of the simple graph $G$. Let $v \notin U$ be a vertex of $G$. If there are $k$ edges joining $v$ with vertices of $U$ then $U \cup \{v\}$ is k-a-connected in $G$.*

Note that the condition is sufficient but not necessary.

**Lemma 8.6** *Let $U$ be a k-a-connected vertex set in the simple graph $G$. We extend $U$ (and $G$) by $k$ new vertices. In addition we add new edges in such a way that each new vertex is linked to every other new vertex by exactly one new edge. Finally we add $k$ new edges which link each new vertex to a vertex in $U$ such that their end vertices in $U$ are pairwise distinct. Then the extended vertex set $\widetilde{U}$ is k-a-connected in the extended graph $\widetilde{G}$.*

We will now carry over propositions 8.1 and 8.2 to $k$-a-connectedness using a somewhat different approach. Let $U$ and $W$ be a vertex sets of a simple graph. We consider a-$UW$-paths and use theorem 7.5 (see page 57). Remember: The vertices in $U \cap W$ are considered to be $UW$-paths of length 0. For $k$-a-connectedness the following proposition holds.

**Proposition 8.6** *Let $U$ and $W$ be k-a-connected vertex sets in the simple graph $G$. $U \cup W$ is k-a-connected in $G$ if and only if there exist $k$ disjoint a-$UW$-paths.*

**Proof:** If there are less than $k$ disjoint a-UW-paths then $U$ is separated from $W$ by less than $k$ vertices and $U \cup W$ cannot be $k$-a-connected.
If there are (at least) $k$ disjoint a-$UW$-paths then the assumption that $w \in W \setminus U$ is separated from $u \in U \setminus W$ by less than $k$ vertices leads to a contradiction. There is an a-$UW$-path which passes through none of the separating vertices. Since they are separated, not both, $u$ and $w$, can lie on this path. If only one, say $u$, were lieing on this path, then the path's end point in $W$ were separated from $w$ by less than $k$ vertices. Finally, if neither $u$ nor $v$ lie on the $UW$-paths then $u$ must be separated from the path's end point in $U$ or $w$ must be separated from the path's end point in $W$ by less than $k$ vertices. □

Proposition 8.6 has important consequences. Let $U$ be a $k$-a-connected vertex set in $G$. We call $U$ *maximal* if no proper superset of $U$ is $k$-a-connected in $G$. If $U$ and $W$ are maximal $k$-a-connected vertex sets in in $G$ and if $|U \cap W| \geq k$ then there are $k$ disjoint a-$UW$-paths (of length 0) and according to proposition 8.6 $U \cup W$ is $k$-a-connected. From the maximality follows $U = W$. That can be interpreted in the following manner: If we extend in $G$ a $k$-a-connected vertex set step by step until we finally reach a maximal $k$-a-connected vertex set then we obtain the same maximal vertex set independently from the order of the steps taken. The $k$-a-connected vertex set we start from uniquely determines the maximal $k$-a-connected vertex set. If "uniquely determined" is meant in this sense we have the following theorem.

**Theorem 8.3** *In a simple graph maximal $k$-a-connected vertex sets are uniquely determined.*

*Notabene:* Theorem 8.3 does not mean maximal $k$-connected vertex sets to be disjoint. In fact, they can have as many as $k-1$ vertices in common.

We shall now examine $k$-a-connected subgraphs of a simple graph. Proposition 8.5 immediately implies the following proposition.

**Proposition 8.7** *Let $G$ be a simple graph and $H$ a subgraph of $G$. Then $H$ is $k$-a-connected if and only if*

1. *all vertices of $H$ have degree at least $k$ and*
2. *in $H$, every pair of non-adjacent vertices is joined by at least $k$ internally disjoint a-paths.*

Let $H$ and $I$ be $k$-a-connected subgraphs of a simple graph. Then $V(H)$ and $V(I)$ are $k$-a-connected vertex sets in $H + I$. If $|V(H) \cup V(I)| \geq k$ then by proposition 8.6 $V(H) \cup V(I) = V(H+I)$ is $k$-a-connected in $H+I$. I. e. $H+I$ is a $k$-a-connected subgraph. If the $k$-a-connected subgraphs $H$ and $I$ are maximal and if they have a common $k$-a-connected subgraph then $H = I$. Using "uniquely determined" in the sense defined above we have the following theorem.

**Theorem 8.4** *In a simple graph maximal $k$-a-connected subgraphs are uniquely determined.*

As we did with $k$-a-lineconnected subgraphs, we introduce a special name for maximal $k$-a-connected subgraphs of simple graphs.

**Definition 8.6** *A maximal $k$-a-connected subgraph of a simple graph is called a $k$-a-component.*

If $H$ and $I$ are $k$-a-connected subgraphs then $H + I$ may be $k$-a-connected even if $H$ and $I$ have less than $k$ vertices in common. To apply proposition 8.6 at least $k$ disjoint a-$V(H)V(I)$-paths are required to exist in $H + I$. To examine this closer we introduce two concepts. We define $ap(H, I)$ to be the intersection of the vertex sets of subgraphs $H$ and $I$. The set of edges linking directly $H$ and $I$ but not inciding with a vertex from $ap(H, I)$ is called $aed(H, I)$. Finally, $\beta_1(aed(H, I))$ is the maximum number of pairwise non-adjacent edges in $aed(H, I)$. With this we can formulate the following proposition.

**Proposition 8.8** *Let $G$ be a simple graph and $H$ and $I$ $k$-a-connected subgraphs. Then $H + I$ is $k$-a-connected if and only if $|ap(H, I)| + \beta_1(aed(H, I)) \geq k$.*

**Proof:** If $H + I$ is $k$-a-connected then there are $k$ disjoint a-$V(H)V(I)$-paths in $H + I$ (proposition 8.6). Each of these paths not containing a vertex from $ap(H, I)$ must pass through at least one edge in $aed(H, I)$ and edges from different paths are non-adjacent. Hence, $|ap(H, I)| + \beta_1(aed(H, I)) \geq k$.
If the inequality holds then there are obviously at least $k$ disjoint a-$V(H)V(I)$-paths in $H + I$ and according to proposition 8.6 $H + I$ is $k$-a-connected. □

### Decomposition into a-components

Every $k$-a-connected ($k \geq 2$) subgraph is also $(k-1)$-a-connected. Every $k$-a-component is contained in a $(k-1)$-a-component. A $(k-1)$-a-component $C$ contains 0 or more $k$-a-components. If there is no $k$-a-component the decomposition into a-components ends with $C$. Otherwise there exist in $C$ $k$-a-components and possibly additional edges which do not belong to any $k$-a-component. The latter generate a subgraph called *$k$-a-cocomponent* of $C$.

In contrast to colinecomponents, the $k$-a-cocomponent of $C$ may be empty even in $C$ exist more than one $k$-a-component. $k$-a-components are uniquely determined but no necessarily disjoint. They may have up to $k-1$ vertices in common and, due to maximality, share also all edges between these vertices. Figure 16 shows a 2-a-connected graph with three 3-a-components all having vertices $a$ and $b$ and the edge joining these in common. The 3-a-cocomponent of the graph is empty.

A non-empty $k$-cocomponent may consist of one single edge and its incidence vertices. An example is shown in figure 17. The graph is 3-a-connected and has two 4-a-components which have the edge linking $c$ and $d$ in common. The 4-a-cocomponents consists of the edge linking $a$ and $b$ together with these vertices. Since the two 4-a-components are not vertex-disjoint the whole graph is 4-a-lineconnected, i.e. it is 4-a-linecomponent.

According to proposition 8.8 two disjoint $k$-a-components cannot be joint directly by $k$ or more pairwise non-adjacent vertices. They can, however, be joined by $k$ disjoint a-paths even if all vertices have degree at least $k$. The graph in figure 15 is an example.

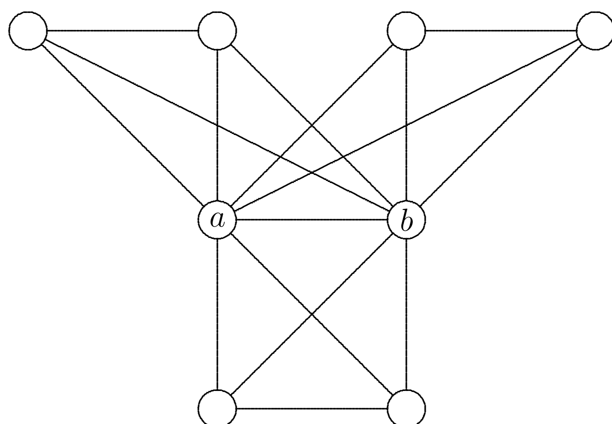


Figure 16: *Three 3-a-components with a common edge*

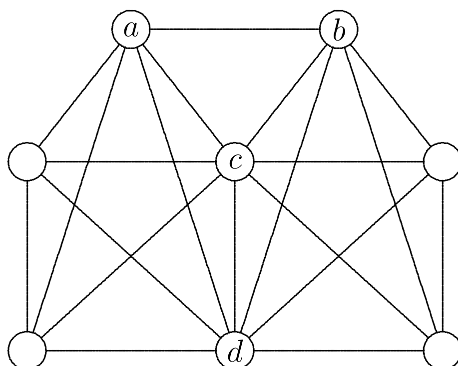


Figure 17: *A 4-a-cocomponent consisting of one single edge*

A vertex of a $k$-a-component may be linked to a vertex in another component by $k$ or more internally disjoint w-paths without the components being identical. An example is shown in figure 18. The graph as a whole is not 3-a-connected. There are two 3-a-components given by the the vertex sets $\{a, b, c, d\}$ and $\{s, t, u, v\}$. However, there are 4 internally disjoint a-paths from $a$ vertex to vertex $t$.

In contrast to $k$-a-components, a $k$-a-cocomponent can have more than $k - 1$ vertices in common with a $k$-a-component even if the graph has minimal degree $k$. Figure 19 shows a 2-a-connected graph with a 3-a-component and a 3-a-cocomponent. From the 3-a-component only the vertices where the 3-a-component is attached are shown (double circles).

## 8.5 An Example

Figure 20 shows schematically the decomposition of a simple graph into $k$-a-linecomponents and and $k$-a-components. Only a cutout is depicted. We start with a $(k-1)$-a-linecomponent $EC$, drawn as oval. In $EC$, there are the subgraphs $EC_1$, $EC_2$, and $EC_3$ which are $k$-a-linecomponents. In addition, there must exist in $EC$ a non-empty

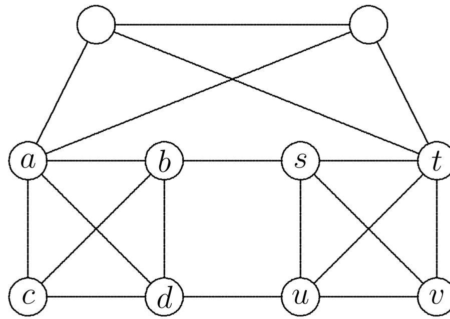


Figure 18: *More than $k-1$ internally disjoint a-paths between two different k-a-components*

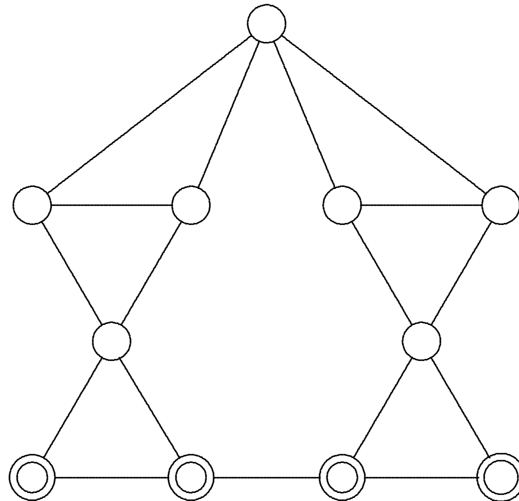

Figure 19: *A k-a-component and a k-a-cocomponent may share mor than $k-1$ vertices.*

$k$-a-colinecomponent. It is drawn as a rectangle.

$EC$ also contains four $(k-1)$-a-components as subgraphs, namely $C_1$, $C_2$, $C_3$, and $C_4$. These are uniquely determined but not necessarily disjoint. It is assumed that $C_2$ and $C_3$ have a non-empty intersection. $C_1$ contains the $k$-a-components $C_{11}$, $C_{12}$, and $C_{13}$. $C_2$ contains $C_{21}$ and $C_{22}$ – in different $k$-a-linecomponents – and $C_3$ contains $C_{31}$ and $C_{32}$. $C_4$ does not contain a $k$-a-component. In general, the $(k-1)$-a-component $C_1$ will also contain edges which are neither element of $C_{11}$ nor $C_{12}$ nor $C_{13}$. I.e. its $k$-a-cocomponent is not empty. The same holds for $C_2$ and $C_3$. Each of these cocomponents may be scattered in a rather unclear manner among $CEC$, $EC_1$, $EC_2$, and $EC_3$ and hence have not been included in figure 20.

The decomposition in figure 20 is schematic. The following example shows that there is a graph which satisfies the requirements set up in the scheme. We only construct the critical parts of the graph and choose $k = 4$. Figure 21 shows the 3-a-component $C_1$. It contains the subgraphs $C_{11}, C_{12}, C_{13}$ which we assume to be $K_5$ and therefore 4-

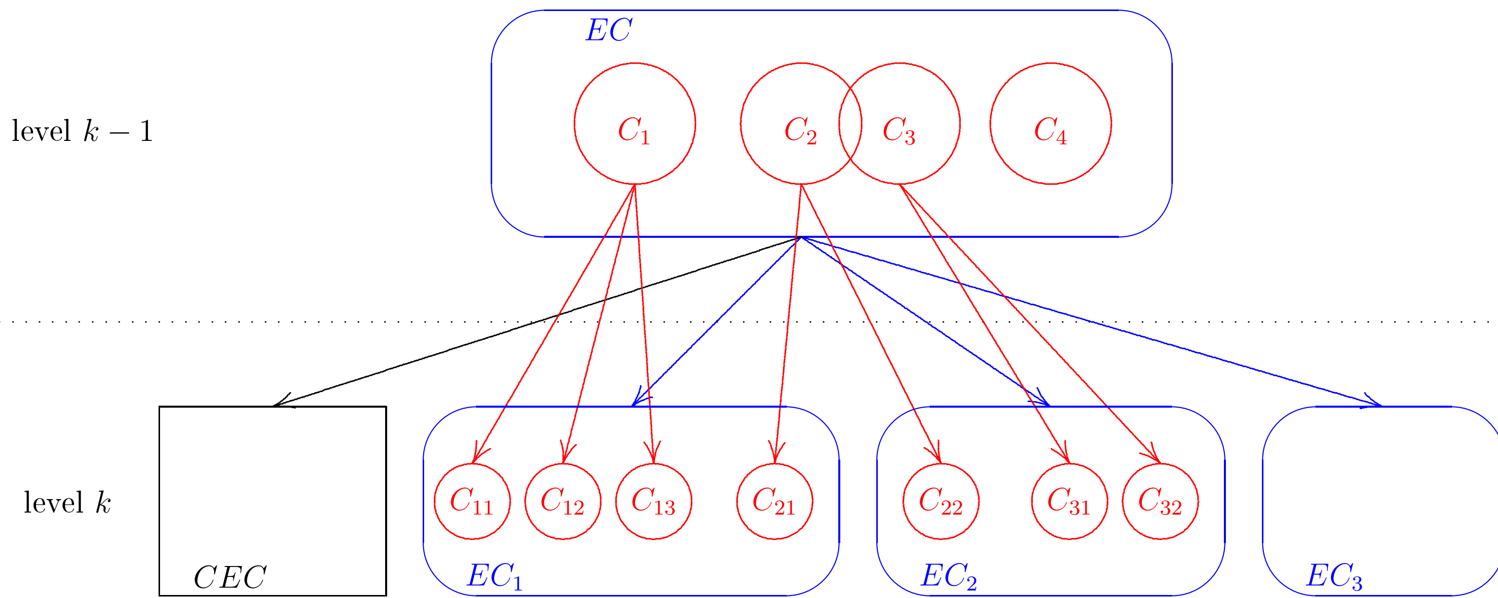


Figure 20: *Hierarchical decomposition into k-a-linecomponents and k-a-components*

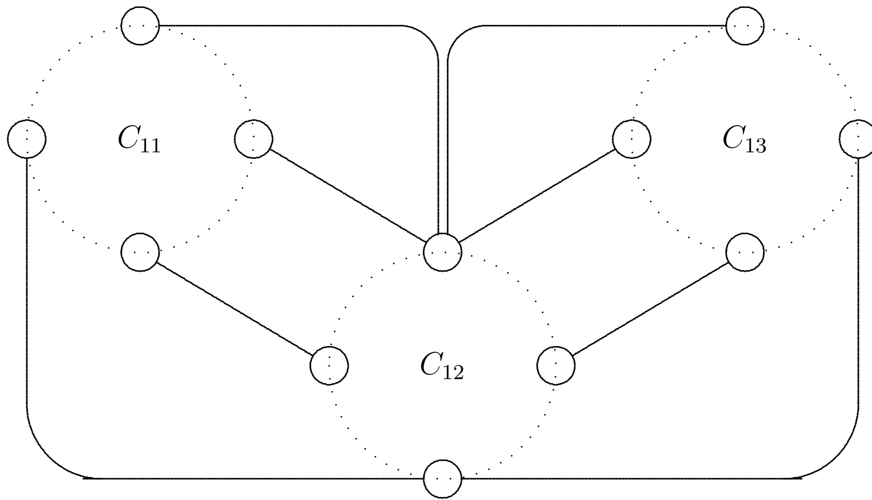


Figure 21: *3-a-component $C_1$ contains 4-a-components $C_{11}$, $C_{12}$, and $C_{13}$*

a-connected. For more clarity the internal structure of these subgraphs is not depicted. Only their attachment points are shown. $C_{11}$, $C_{12}$, and $C_{13}$ are 4-a-components since they are mutually separated from each other by 3 vertices. As a whole, $C_1$ is 4-a-lineconnected and therefore completely contained in one 4-a-linecomponent, for instance in $EC_1$.

Next we examine the 3-a-component $C_4$. It must not contain any 4-a-components but we want it to be 4-a-lineconnected. Figure 22 shows a subgraph with this property. Hence it

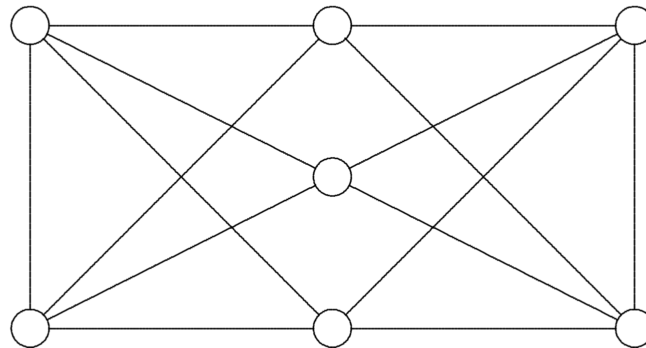

Figure 22: *3-a-component $C_4$ does not contain a 4-a-component*

is contained in a 4-a-linecomponent. We assume it to have exactly two vertices in common with $C_{13}$ and hence belonging to $EC_1$. It is not shown as a red circle on level $k$.

Figure 23 shows the 3-a-component $C_2$. It contains the 4-a-components $C_{21}$ and $C_{22}$. These are assumed to be $K_5$ and are depicted by dotted lines. They are joined via three explicitly drawn vertices and external edges. Hence, they belong to the same 3-a-component but to different 4-a-linecomponents. To force $C_{21}$ and $C_{13}$ to be disjoint but both being parts of the same 4-a-linecomponent we also attach $C_{21}$ to $C_4$. To this end, we again use two vertices common to both, $C_{21}$ and $C_4$, taking care that these are pairwise

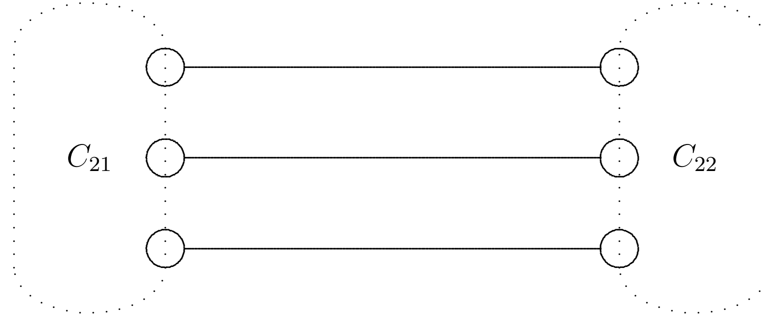


Figure 23: *3-a-component $C_2$ contains 4-a-components $C_{21}$ and $C_{22}$*

distinct from the vertices used to attach $C_{13}$ to $C_4$. If so, $C_{13}$ and $C_{21}$ are not only disjoint but belong to different 3-a-components.

The 3-a-component $C_3$ need not be illustrated by a special figure. We construct $C_3$ the same way as $C_1$ with two 4-a-components $C_{31}$ and $C_{32}$. To make their intersection non-empty let them have one single vertex in common.

## 8.6 Connectedness Defined by f-Paths

The results of the preceding subsections carry over to f-paths very easily. For this reason, we just mention the essential definitions and theorems, referring for the rest and a more detailed treatment to [Stie2006].

#### $k$-f-Lineconnectedness

**Definition 8.7** *Let $G$ be a general graph and $U$ a set of vertices in $G$ with at least 2 vertices. $U$ is termed* $k$-f-lineconnected *in $G$, if for all pairs of vertices $(u, v) \in U \times U$ with $u \neq v$ vertex $v$ is k-f-linereachable from vertex $u$ in $G$.*
*A subgraph $H$ of $G$ is termed k-f-lineconnected if its vertex set $V(H)$ is k-f-lineconnected in $H$.*

**Proposition 8.9** *Let $U$ and $W$ k-f-lineconnected ($k \geq 1$) vertex sets of a general graph $G$. If $U \cap W \neq \emptyset$ then $U \cup W$ is k-f-lineconnected, too .*

The following theorem is an immediate consequence of proposition 8.9.

**Theorem 8.5** *Maximal k-f-lineconnected vertex sets in a general graph are uniquely determined.*

The following proposition is analogous to proposition 8.2.

**Proposition 8.10** *Let $U$ and $W$ be disjoint k-f-lineconnected vertex sets in a general graph $G$. If and only if $U$ is linked to $W$ and $W$ is linked to $U$ by at least k line-disjoint f-paths then $U \cup W$ is also k-f-lineconnected in $G$.*

We now examine $k$-f-lineconnected subgraphs $H$ and $I$ of a general graph $G$.

**Proposition 8.11** *Let $H$ and $I$ be $k$-f-lineconnected ($k \geq 1$) subgraphs of a general graph $G$. If $H \cap I \neq \emptyset$ then also $H + I$ is a $k$-f-lineconnected subgraph of $G$.*

**Theorem 8.6** *Maximal $k$-f-lineconnected subgraphs of a general graph are uniquely determined.*

**Definition 8.8** *A maximal $k$-f-lineconnected subgraph of a general graph is called* $k$-f-linecomponent *of the graph.*

**Proposition 8.12** *Let $H$ and $I$ be disjoint, $k$-f-lineconnected subgraphs of a general graph. If there are at least $k$ direct f-links from $H$ to $I$ and at least $k$ direct f-links from $I$ to $H$ then $H + I$ is also $k$-f-lineconnected.*

***Decomposition into f-linecomponents***

The decomposition of a general graph into $k$-f-linecomponents and $k$-f-linecocomponents is completely analogous to the decomposition using a-paths.

### $k$-f-Connectedness

***Directed Kernel***

As with a-paths, defining $k$-f-connectedness using f-paths causes difficulties if multiple edges or multiple arcs were allowed.

**Definition 8.9** *The* directed kernel *of a general graph $G$ is a graph with the same vertex set, without loops, without multiple edges, and without multiple arcs. Two of its vertices are joined by an edge if and only if they are joined by an edge in the original graph. If two vertices in $G$ are joined by arcs but not by an edge then in the directed kernel they are joined by exactly one arc in each direction in which they are joined in $G$.*

The remarks in subsection 8.3 corresponding to the simple kernel apply in an analogous manner to the directed kernel. To avoid a new name we call a general graph a directed kernel if there is a general graph for which it is the directed kernel.

***Definitions and properties***

**Definition 8.10** *Let $G$ be a directed kernel and $U$ a set of its vertices with at least $k+1$ elements. $U$ is called* $k$-f-connected *in $G$, if for all vertices $u, v \in U$ with $u \neq v$ vertex $v$ is $k$-f-reachable from vertex $u$ and vertex $u$ is $k$-f-reachable from vertex $v$.*
*A subgraph $H$ of $G$ is called $k$-f-connected if its vertex set $V(H)$ is $k$-f-connected in $H$.*

**Proposition 8.13** *Let $U$ be a vertex set with at least $k+1$ elements of the directed kernel $G$. $U$ is $k$-f-connected in $G$ if and only if for every pair $u, v \in U$ with $u \neq v$ and no direct f-link from $u$ to $v$ there are (at least) $k$ internally disjoint f-paths from $u$ to $v$.*

We now consider the equivalent of proposition 8.6 and use again $UW$-paths (see page 55).

**Proposition 8.14** *Let $U$ and $W$ be k-f-connected vertex sets in the directed Kernel $G$. $U \cup W$ is k-f-connected in $G$ if and only if there are in $G$ $k$ disjoint f-$UW$-paths in each direction.*

With the interpretation on "uniquely determined" as unique maximal extensibility we have the following theorem.

**Theorem 8.7** *In a directed kernel maximal k-f-connected vertex sets are uniquely determined.*

We now start examining $k$-f-connected subgraphs of a directed kernel. From proposition 8.13 we obtain the following proposition.

**Proposition 8.15** *Let $G$ be a directed kernel and $H$ a subgraph of $G$. Then $H$ is k-f-connected if and only if*

1. *all vertices of $H$ have minimal degree $k$ and*
2. *in $H$, for every pair $u, v \in H$ with $u \neq v$ and no direct f-link from $u$ to $v$ there are (at least) $k$ internally disjoint f-paths from $u$ to $v$.*

Again we have uniqueness.

**Theorem 8.8** *In a directed kernel k-f-connected subgraphs are uniquely determined.*

We introduce a special name.

**Definition 8.11** *A maximal k-f-connected subgraph of a directed kernel is called* $k$-f-component.

To formulate the equivalent of proposition 8.8 we define $ap(H, I)$ to be the set of vertices common to subgraphs $H$ and $I$. Furthermore, $aed(H, I)$ is the set of lines from $H$ to $I$ without inciding with a vertex from $ap(H, I)$. Finally, $\beta_1(K)$ is again the maximal number of pairwise non-adjacent lines in a line set $K$.

**Proposition 8.16** *Let $G$ be a directed kernel and $H$ and $I$ k-f-connected subgraphs of $G$. Then $H + I$ is k-f-connected if and only if $|ap(H, I)| + \beta_1(aed(H, I)) \geq k$ and $|ap(H, I)| + \beta_1(aed(I, H)) \geq k$.*

***Decomposition into f-components***

The considerations from page 71 hold accordingly for f-connectedness. A $(k-1)$-f-component $C$ contains 0 or more $k$-f-components. If there is no $k$-f-component then the decomposition of $C$ ends. Otherwise there are $C$ $k$-f-components and perhaps a non-empty *k-f-cocomponent.*

### Remarks and Literature

The basis of this section are the reports [Stie2001c] and [StieS2003]. The results have been revised and extended. $k$-a-connectedness and $k$-a-lineconnectedness is treated in Harary [Hara1969] and especially in Jungnickel [Jung1994]. Neither $k$-a-connected and $k$-a-lineconnected vertex set nor f-connectedness in general seem to have found interest. They are not included in the textbook literature.

# 9 Finding Components of Higher Order

## 9.1 Graph Decompositions

Loosely speaking, by a *graph decomposition* we will understand a family of subgraphs whose union is the given graph and which have as few as possible vertices and edges in common. We call a decomposition of a graph *hierarchical* if some of the decomposing subgraphs are decomposed themselves and this procedure is reiterated.

From the definitions in section 8 we obtain the four-dimensional decomposition scheme:

$$\text{level } k \Rightarrow \text{level } (k-1) \quad (4)$$

$$\text{internally disjoint paths} \Rightarrow \text{line-disjoint paths} \quad (5)$$

$$\text{f-connectedness} \Rightarrow \text{a-connectedness} \quad (6)$$

$$\text{subgraphs} \Rightarrow \text{vertex sets} \quad (7)$$

The first three implications mean that every vertex set, respectively every subgraph for which the left side holds also satisfies the right side. The last implication says that the vertex set of a subgraph has (at least) the connectedness properties of the subgraph. Together with the uniqueness theorems 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, and 8.8 these implications yield a variety of graph decompositions. The decomposition elements may be maximal subgraphs, i.e "components", as well as maximal vertex sets with certain connectedness properties. A cutout of a decomposition which takes into account only a-paths and the implications 4 and 5 has been examined in detail in subsection 8.5.

For a general graph, how can a decomposition effectively be found? For $k = 1$, i.e. for weak and strong connected components, linear algorithms have been presented in subsections 4.3 and 4.4. For $k = 2$ and a-connectedness there is the biblock decomposition (see section 5) for which also a linear algorithm exists. For $k = 3$ and a-connectedness Hopcroft and Tarjan [HopcT1973a] presented a linear algorithm.

But how can a graph decomposition be found for general $k$? As a first step we answer the question how to determine whether for given $k$ a vertex set $U$ or a subgraph $H$ have certain connectedness property. For instance, we want to know whether $U$ is $k$-f-connected. To this end we apply the algorithm of Menger theorem 7.2 to all pairs of vertices $(u, v)$ in $U$ with $u \neq v$. $U$ is $k$-f-connected if and only if there are in each case (at least) $k$ internally disjoint f-paths from $u$ to $v$. If the algorithm is modified such as to stop once $k$ internally disjoint f-paths has been found then according to proposition 7.1, page 55, the complexity

is $n' \cdot (n'-1) \cdot k \cdot O(m+n)$ where $n' = |U|$, $n$ the number of vertices in $G$ and $m$ the number of lines in $G$. If we have to determine the $k$-f-connectedness of a subgraph $H$ then we restrict the Menger algorithm to $H$ and the complexity is $n \cdot (n-1) \cdot k \cdot O(m+n) = O(n^2 \cdot m + n^3)$ where $n$ is the number of vertices and $m$ is the number of lines of the subgraph. A easily can be seen, we have analogous results for $k$-f-lineconnectedness, $k$-a-connectedness and $k$-a-lineconnectedness.

If we are not only interested in the $k$-connectedness of a single subgraph but want to know the complete hierarchical decomposition, how should we proceed? In the following we describe a procedure which may be used. *We restrict ourselves to one singe decomposition dimension, namely the connectedness number $k$.* As an example, we want to find for all $k$ the $k$-a-components or the maximal $k$-f-lineconnected vertex sets. The decompositions in other dimensions are then simple consequences. If for instance the $k$-a-components as well as the $k$-f-components of a general graph are known then it is easy to map every $k$-f-component to the uniquely determined $k$-a-component it is part of.

The basis of the procedure are the Menger algorithms as described above. It is enhanced by a heuristics which hopefully will improve the efficiency. The heuristics characterizes vertices and lines using colors red, green, and blue and is therefore called *RGB-procedure*. We shall describe in detail the RGB-procedure for $k$-a-components and $k$-a-linecomponents in the next subsections. For the remaining decompositions the procedure is sketched in subsection 9.6.

## 9.2 Decomposition into $k$-a-Components

In this subsection we only consider simple graphs. To decompose such a graph into its $k$-a-components we could start with $k = 2$, i.e. with its biblocks (section 5), and then in turn decompose each of the graph's $k$-a-components into its $(k+1)$-a-components and the $(k+1)$-a-cocomponent. The minimal degree of a $(k+1)$-a-component is not less than $k+1$. Hence the first step in decomposing a $k$-a-component will be to recursively delete all vertices with degree smaller than $k+1$. This may result in a clear reduction of size. This observation suggests another approach to decompose the graph, namely bottom up.

To this end we introduce a preliminary decomposition of the graph into *$k$-a-candidates*. A $k$-a-candidate is a 2-a-connected subgraph whose minimal degree is at least $k$. Biblocks are the maximal 2-a-candidates of the graph. We find the maximal $(k+1)$-candidates of a given $k$-a-candidate by recursively deleting all vertices of degree less than $k+1$. It may happen that the resulting subgraph is not 2-a-connected. In this case we must find the biblocks of this subgraph. These in turn may contain vertices with degree less than $k+1$ and must be reduced recursively to minimal degree $k+1$. We have to alternatively apply degree reduction and biblock decomposition until we have found 0 or more maximal $(k+1)$-a-candidates in the $k$-a-candidate. For finding the biblocks we use the algorithm of section 5. Finding an efficient algorithm to recursively delete all vertices with degree less than $k$ is an easy exercise left to the reader.

We start now with the maximal $k$-a-candidates with the greatest $k$ and proceed bottom up to lower values of $k$. That means that we start the processing of a candidate only if all maximal candidates of higher order which it contains are already processed. Processing a

candidate means: *Find the $k$-a-components of the candidate using algorithm RGBv from subsection 9.3.* At this moment all $(k+1)$-a-components contained in the $k$-a-candidate are already known. Possibly additional $k$-a-connected subgraphs of the $k$-a-candidate, for instance the $k$-f-components, are known, too. All these subgraphs can be used to determine the $k$-a-components.

Once a complete decomposition of the graph into $k$-a-components have been found it is easy to determine the $(k+1)$-a-cocomponent of each $k$-a-component.

## 9.3 Algorithm for Finding the $k$-a-Components

This subsection is dedicated to the algorithm RGBv for finding the $k$-a-components of a simple graph. The main tool is the algorithm described with the proof of theorem 7.1, page 46. See also subsection 7.4. For two non-adjacent vertices $u$ and $v$ it yields $n$ internally disjoint (simple) a-paths from $u$ to $v$ as well as $n$ vertices $v_1, v_2, \dots v_n$ which a-separate $u$ and $v$. To test whether a $k$-a-candidate is $k$-a-connected using proposition 8.5 we must, in principle, apply the Menger algorithm to all unordered pairs of non-adjacent vertices. As a result either all pairs of vertices are joined by $k$ internally disjoint a-paths or we find a first pair $\{u, v\}$ being a-separated by less the $k$ vertices $v_1, v_2, \dots, v_n$.

In the first case we have found out that the $k$-a-candidate is a $k$-a-component.

In the second case we stop testing and use vertices $v_1, v_2, \dots, v_n$ to decompose $G$ into two or more subgraphs with the separating vertices as attachment points. This is done in the following way:

1. We use the separating points $v_1, v_2, \dots, v_n$ to build a partition of the edges of the $k$-a-candidate. This can be done, for instance, with a depth-first search which for which $v_1, v_2, \dots, v_n$ are limiting points. These are the attachment points of the partition.

2. Edges joining two separating vertices form classes of a single element. These classes are removed. The corresponding edge is joined to all classes which have both incidence points of the edge as attachment points. The class containing $u$ and the class containing $v$ are such classes. After this not all classes are pairwise disjoint anymore.

3. In the next step we generate subgraphs from the edge classes and discard all subgraphs with less than $k+1$ vertices. The remaining subgraphs are not necessarily $k$-a-candidates. It may happen that they are not 2-a-connected and that their attachment points are of degree less than $k$. As explained in subsection 9.2 we have to reduce the subgraph to obtain the minimal degree and 2-a-connectedness. Finally 0 or more $k$-a-candidates remain. The Menger algorithm is applied again to the unordered non-adjacent pairs of these new $k$-a-candidates to test whether they are $k$-a-components or an additional decomposition into smaller $k$-a-candidates becomes necessary.

To improve the efficiency of testing the unordered pairs with the Menger algorithm we introduce the heuristics RGB. If in a $k$-a-candidate a $k$-a-connected subgraph $H$ is known

then we need not test pairs of vertices $u, v \in H$. Pairs of vertices $u \notin H$ and $v \in H$ need not to be tested if there is a $v' \in H$ which is joined to $u$ via $k$ internally disjoint a-paths. In the following we examine this in detail.

Let $H_1, H_2, \ldots, H_r$ be $k$-a-connected subgraphs of a $k$-a-candidate. They need not be disjoint. In a *preprocessing phase* we apply lemma 8.4 and proposition 8.8 as long as possible to enlarge or merge the $H$. After that we know that a vertex which is joined to vertices of $H$ by at least $k$ edges belongs to $H$. We also know that $H$ and $H'$ have at most $k-1$ vertices in common.

**Remark 9.1** The condition in proposition 8.8 is $|ap(H, H')| + \beta_1(aed(H, H')) \geq k$. Whereas $|ap(H, H')|$, the number of common vertices of $H$ and $H'$, can easily be determined, calculating $\beta_1(aed(H, H'))$ means finding a maximal bipartite matching. Though that is not difficult (see [ThulS1992] or [McHu1990]) we shall not require it for the algorithm. For us merging the $H$ up to a point where no two of them have more than $k-1$ common vertices suffices. □

We now come to the algorithm RGBv. Applying RGBv to a $k$-a-candidate $G$ with preprocessed $k$-a-connected subgraphs $H$ yields a modified simple graph $\widetilde{G}$ in which some elements are colored in the following way:

i. All edges of $G$ remain uncoloreds (i.e. black).

ii. The attachment points of all $H$ are colored green. A vertex of $H$ is an attachment point if it incides with an edge not in $H$.

iii. All vertices in $H$ which are not attachment points remain uncolored.

iv. All vertices which are not element of any $H$ are colored blue.

v. For each $H$ we add a new red vertex which is linked to all green attachment points by a new red edge.

Red edges and red vertices do not exist in $G$. There may exist red vertices with a degree smaller than $k$. In this case the corresponding $H$ is separated from the rest of the candidate by less then $k$ vertices. It is a $k$-a-connected subgraph which cannot be augmented, i.e. it is a $k$-a-component. We use its attachment points as a-separating vertices and build the corresponding edges classes as described above. One of the so generated subgraphs is $H$. Without further processing $H$ is classified and recorded as $k$-a-component. To the other subgraphs degree reduction and biblock decomposition has to be applied until $k$-a-candidates remain. Eventually a graph $\widetilde{G}$ results all whose red vertices have degree $k$ or higher.

We now use $\widetilde{G}$ to accelerate the test whether $G$ is $k$-a-connected. This is done be reducing the number of vertex pairs to be tested. We apply the algorithm of Menger theorem 7.1 to $\widetilde{G}$ and find the maximal number of internally disjoint a-paths joining $\tilde{u}$ and $\tilde{v}$ where

1. $\tilde{u}$ and $\tilde{v}$ are distinct and non-adjacent.

2. $\tilde{u}$ and $\tilde{v}$ are blue or red and

3. red vertices are not admitted as inner vertices of the paths found.

As a result we either find a first unordered vertex pair $\tilde{u}$ and $\tilde{v}$ and $r$ $(r < k)$ vertices which a-separate and $\tilde{u}$ and $\tilde{v}$ or all tested pairs are joined by at least $k$ internally disjoint a-paths. In the second case we call $\widetilde{G}$ *RGB-k-a-connected.* The following proposition states that, in fact, we are allowed to confine ourselves to testing the pairs in $\widetilde{G}$.

**Proposition 9.1** *1. $G$ is $k$-a-connected if and only if $\widetilde{G}$ is RGB-$k$-a-connected.*

*2. If in $\widetilde{G}$ vertices $\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_r$ with $r < k$ are found which a-separate vertices $\tilde{v}$ and $\tilde{u}$ then these vertices are not red and a-separate in $G$ suitable chosen vertices $v$ and $u$.*

**Proof:** The proof is not difficult but somewhat tedious and lengthy. For details see [Stie2006]. □

## 9.4 Decomposition into $k$-a-Linecomponents

In this subsection we consider arbitrary general graphs. To find its $k$-a-linecomponents we use again a bottom up approach. We determine a preliminary decomposition of the graph into *k-a-linecandidates.* A $k$-a-linecandidate is a 2-a-lineconnected subgraph with minimal degree $k$. The subcomponents (see subsection 5.2) are the maximal 2-a-linecandidates of the graph. As with $k$-a-candidates we find the maximal $(k+1)$-a-linecandidates of a $k$-a-linecandidate by alternating reduction to degree $k+1$ and decomposition into subcomponents.

We start now with the maximal $k$-a-linecandidates with the greatest $k$ and proceed bottom up to lower levels of $k$. That means that we start the processing of a candidate only if all maximal candidates of higher order which it contains are already processed. Processing a candidate means: *Find the k-a-linecomponents using algorithm RGBe from subsection 9.5.* At this moment all $(k+1)$-a-linecomponents in the $k$-a-linecandidate are already known. Possibly additional $k$-a-lineconnected subgraphs of the $k$-a-linecandidate, for instance the $k$-a-components, are known, too. All these subgraphs can be used to determine the $k$-a-linecomponent.
Once a complete decomposition of the graph into $k$-a-linecomponents has been found it is easy to determine the $(k+1)$-a-colinecomponent of each $k$-a-linecomponent.

## 9.5 Algorithm for Finding the $k$-a-Linecomponents

This subsection is dedicated to the algorithm RGBe for finding the $k$-a-linecomponents of a general graph. The main tool is the algorithm described with the proof of theorem 7.3, page 46. For two distinct vertices $u$ and $v$ it yields $n$ line-disjoint (simple) a-paths from $u$ to $v$ as well as $n$ lines $l_1, l_2, \ldots, l_n$ a-separating $u$ and $v$. To test whether a $k$-a-linecandidate is $k$-a-lineconnected we must, in principle, apply the Menger algorithm to all unordered pairs of distinct vertices. As a result either all pairs of vertices are joined

by $k$ line-disjoint a-paths or we find a first pair $\{u, v\}$ being a-separated by less than $k$ lines $l_1, l_2, \ldots, l_n$.

In the first case we have found out that the $k$-a-linecandidate is a $k$-a-linecomponent.

In the second case we stop testing and use lines $l_1, l_2, \ldots, l_n$ to decompose $G$ into two or more subgraphs With the separating lines as joining lines. This is done in the following way:

1. We use the separating lines $l_1, l_2, \ldots, l_n$ to build a partition of the vertices of the $k$-a-linecandidate. This can be done, for instance, with a depth-first search for which the $l_1, l_2, \ldots, l_n$ are limiting lines.
2. Then we generate subgraphs from the vertex classes. These contain at least two vertices indeed. However, they are not necessarily 2-a-lineconnected and of minimal degree at least $k$. We have to reduce each subgraph until finally 0 or more $k$-a-linecandidates remain.

Again we improve the efficiency of testing using the heuristics RGB. If in a $k$-a-linecandidate a $k$-a-lineconnected subgraph is known then we need not test pairs of vertices $u, v \in H$. Pairs of vertices $u \notin H$ and $v \in H$ need not to be tested if there is a $v' \in H$ which is joined to $u$ via $k$ line-disjoint a-paths. In the following we examine this in detail.

Let $H_1, H_2, \ldots, H_r$ be $k$-a-lineconnected subgraphs of a $k$-a-linecandidate. In a *preprocessing phase* we apply lemma 8.1 and proposition 8.3 as long as possible to enlarge or merge the $H$. After that we know that a vertex which is a-joined to vertices of an $H$ by at least $k$ lines belongs to $H$. We also know that different $H$ and $H'$ have no common vertices and are not a-linked directly via $k$ lines. This allows us to introduce colors for vertices and lines and make use of red vertices in a more efficient way than with algorithm RGBv.

We start with a general graph $G$ and preprocessed $k$-a-lineconnected subgraphs $H_1, H_2, \ldots,$ $H_r$. Applying RGBe to $G$ yields a graph $\widetilde{G}$ in the following way:

i. All attachment points are colored green.

ii. In all subgraphs $H$ we delete all lines and all not green vertices.

iii. All remaining vertices and lines are colored blue.

iv. We add a new red vertex for each $H$.

v. Every red vertex is joined to each of its attachment points by $k$ red edges.

Preprocessing assures that a green vertex is linked to exactly one red vertex via $k$ red edges. All other lines the green vertex is incident with are blue. It is possible that the green attachment points of a subgraph $H$ incide with less than $k$ blue lines altogether. In this case these lines a-separate $H$ from the rest of $G$ and $H$ is a $k$-a-linecomponent. We use the blue separating lines to generate a vertex partition of $G$. One of the subgraphs so generated is $H$. It is classified and recorded as $k$-a-linecomponent. To the other vertex

classes degree reduction and subcomponent decomposition has to be applied until $k$-a-linecandidates remain. Eventually a graph $\widetilde{G}$ results in which the green attachment points corresponding to a given red vertex incide with at least $k$ blue lines altogether.

To test $k$-a-lineconnectedness in $\widetilde{G}$ we apply RGBe in the following way:
*Test all unordered pairs of distinct, non-green vertices $\tilde{v}$ and $\tilde{u}$.*
Note that red vertices may occur as inner vertices of the paths. We call $\widetilde{G}$ *RGB-k-a-lineconnected* if all tested pairs are joined via at least $k$ line-disjoint a-paths. The following proposition states that, in fact, we are allowed to confine ourselves to testing pairs in $\widetilde{G}$.

**Proposition 9.2** 
1. *$G$ is $k$-a-lineconnected if and only if $\widetilde{G}$ is RGB-$k$-a-lineconnected.*
2. *If in $\widetilde{G}$ lines $\tilde{l}_1, \tilde{l}_2, \ldots, \tilde{l}_r$ with $0 < r < k$ are found that a-separate vertex $\tilde{v}$ from vertex $\tilde{u}$ then these lines are blue and a-separate in $G$ suitable chosen vertices $u$ and $v$.*

**Proof:** Again, the proof is not difficult but tedious. For details see [Stie2006]. □

## 9.6 More Decompositions

**Decomposition into $k$-f-components and $k$-f-linecomponents:** The procedure for finding these f-components is almost identical to that for finding a-components. Of course, the Menger algorithms of theorems 7.2, respectively theorem 7.4 have to be applied in both directions. If in either direction less then $k$ f-separating vertices, respectively lines, are found we must use them to generate the corresponding line, respectively vertex partitions and decompose the f-candidate into smaller f-candidates.

**Decomposition into maximal $k$-a-connected vertex sets or maximal $k$-a-lineconnected vertex sets:** On the whole, the procedure is the same as with $k$-a-components, respectively $k$-a-linecomponents. Some changes are necessary to take into account the fact that now vertices not belonging to the vertex set may be traversed by the paths. Again, the RGB-heuristics is applicable. For details see [Stie2006].

**Decomposition into maximal $k$-f-connected vertex sets or maximal $k$-f-lineconnected vertex sets:** Analogous to maximal $k$-a-connected vertex sets, respectively maximal $k$-a-lineconnected vertex sets.

## 9.7 Complexity of Decomposition Finding

### Decomposition into a-components

We start with a simple graph which is 2-a-connected and of minimal degree $k$. The goal is to find all its $k$-a-components. We ignore RGB-techniques and use only the algorithm of theorem 7.1 as described in subsection 9.3, steps 1 to 3. The run of the algorithm can be depicted as a tree. The root is the given graph. Applying the Menger algorithm either

yields a $k$-a-connected graph and the decomposition is done. Or it yields two or or more subgraphs generated from edge partitions. Eventually each of these subgraphs is either completely reduced or gives rise to one or more $k$-a-candidates. These are the sons of the original graph in the tree. Reiterating this procedure we finally get the complete tree of the decomposition run, decomposition tree for short.

In the decomposition tree a path from the root to a leaf is at most of length $n$, the number of vertices of the root. For each son has always fewer vertices as its father. Each time a set of $l$ ($l \leq k$) separating vertices is found the total number of vertices to be considered in the following steps is increased by a positive multiple of $l$. Since no node in the tree can have more than $n$ sons, $l(n)$, the tree's total number of leaves, is bounded by $n^n$. The critical question is whether $l(n)$ is polynomially bounded in $n$ or not.

1. *$l(n)$ is polynomially bounded in $n$.*
   The Menger algorithm for a-paths is of polynomial complexity (proposition 7.1, page 55) and so is the test phase of the unordered pairs of a $k$-a-candidate. Of course, partitioning, degree reduction, and biblock decomposition are of polynomial complexity, too. That means, finding the sons of a node in the decomposition tree is polynomially bounded. The total number of nodes is also polynomially bounded, and therfore for a fixed $k$, finding all $k$-a-components of a graph is of polynomial complexity. Since $k < n$ we have the final result: *Finding the a-decomposition of a graph is of polynomial complexity.*

2. *$l(n)$ is not polynomially bounded in $n$.*
   Then for each $i \in \mathbb{N}$ there are graphs such that $l(n)$ exceeds $n^i$. For these graphs, finding all a-components is not feasible in polynomial time, since at least one operation has to be executed for each a-component, say recording in the data structure. We have the result: *Finding the a-decomposition of a graph is of higher than polynomial complexity.*

I do not know which of 1. or 2. is true but I suspect $l(n)$ to be polynomially bounded.
*Note: The arguments given in [Stie2006], section 15.7, that a-decomposition is of polynomial complexity, are incorrect.*
Even if a-decomposition of a graph were of polynomial complexity it is difficult to estimate it and it will be high. The critical point are the test phases using the Menger algorithm. The efficiency of this algorithm very likely will not be improved but in minor details. The most effective reduction in running time will probably be achieved by reducing the number of test phases to be executed. The heuristics RGB aims at that. Another heuristics may take into account that a pair of vertices which is to be tested using the Menger algorithm has already been tested in a previous phase. In general, testing anew is necessary. Perhaps simple conditions can be found which allow skipping the additional test run. Another idea is to use the structure of the Menger separating set (see subsection 7.1). Still another heuristics is presented in the next subsection.

**Decomposition into a-linecomponents**

$k$-a-linecomponents ($k$ fixed) have no vertices in common. Hence there are at most $n$ such components for each $k$. That means that their number is polynomially bounded in $n$ and

the algorithm for finding them has polynomial complexity. However, heuristics like RGB which improve the efficiency of the decomposition algorithm are still very desirable.

#### f-Decompositions

Decomposition into f-components have the same complexity as for a-components. Tests always have to be executed in both directions. The same heuristics can be used.

#### Finding Maximal Sets of $k$-Connected Vertex Sets

Finding maximal sets of $k$-a-connected ($k$-f-connected) vertex sets is more involved than finding components but of the same complexity. Again, the same heuristics can be used.

## 9.8 Dynamic Improvement of the RGB-Procedure

We only examine a-components and RGBv. This algorithm uses all $k$-a-connected subgraphs which are known at the beginning of a test phase. It may be useful to consider also $k$-a-connected subgraphs which become known only during the test phase. The following procedure aims at recognize a $k$-a-connected subgraph as early as possible.

We start with two vertices $v_1$ and $v_2$ of a candidate. The distance between these should be short, preferable 2. If there exist at least $k$ internally disjoint a-paths from $v_1$ to $v_2$, then at least $k$ vertices different from $v_1$ and $v_2$ lie on these paths. We introduce queues $Q_1$ and $Q_2$. $v_1$ and $v_2$ are added to $Q_1$, the remaining vertices on the paths are added to $Q_2$. Subsequently we execute algorithm DYNTEST shown in table 20. If the algorithm ends

**DYNTEST**

```
1   while (Q2 is not empty)
2     { u = dequeue(Q2);
3       forall (v ∈ Q1)
4         { apply Menger algorithm to u and v an;
5           if (u and v are separated by less than k vertices) break;
6           forall (internal vertices w of the paths from u to v)
7             { if (w ∉ Q2) enqueue(w, Q2);
8             }
9         }
10      enqueue (u, Q1);
11    }
```

Table 20: *Algorithm for dynamic recognition of k-a-connected subgraphs*

without `break` $Q_2$ is empty and the vertices in $Q_1$ generate a $k$-a-connected subgraph. Applying DYNTEST to a RGB-colored candidate yields one of the following three cases:

1. Less than $k$ separating vertices are found and the candidate is decomposed further.

2. No separating vertices are found and $Q_1$ contains *all* vertices of the candidate. Then the candidate is a $k$-a-component.

3. No separating vertices are found and $Q_1$ does not contain all vertices of the candidate. Then a proper $k$-a-connected subgraph has been found. It is included in a new preprocessing phase of the candidate. The new $k$-a-connected subgraph may contain red vertices. If so, it also contains their original vertices. Subsequently RGBv is applied again.

## Remarks and Literature

Graph decomposition into components of different kinds seem to be new. The results of this section are based on the report [StieS2003].

# A Various Supplements

## A.1 Component Classification

**Example A.1** Consider graph *wcompgraph* shown in figure 24. It consists of 13 weak

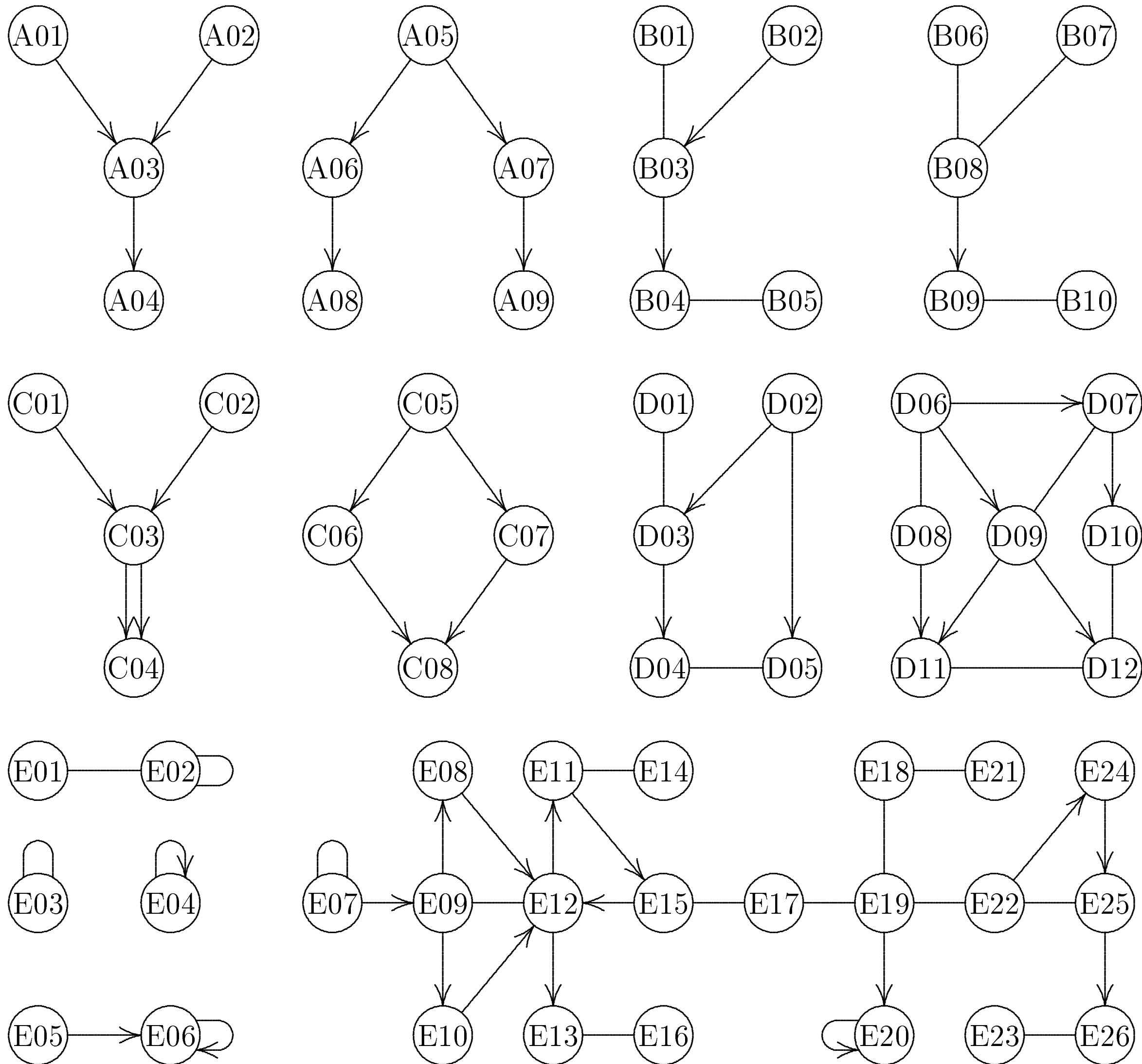


Figure 24: *wcompgraph*

components and provides examples for all 5 types of weak components described in subsection 3.5. Tables 21 and 22 show the classification and decomposition of the weak components. The decomposition list was produced with the *graph handling system GHS*.[6]The listing starts with a header line, indicating that that only essential structural information

[6]Information about GHS including a manual can be found under
`http://www-bvs.informatik.uni-oldenburg.de/projects/Stiege/ghs.html`

is printed ("REDUCED STRUCTURE"). The header line is followed by some general statistics about the graph.

The next line indicates that graph wcompgraph consists of 13 weak components. GHS uses a hierarchical naming scheme. `wcompgraph.WCOMP3.SCOMP1` means that within graph wcompgraph we consider the weak component numbered 3 and within this the strong component numbered 1.
*A-acyclic weak components without strong components:* `WCOMP0` and `WCOMP1`. The a-period `aper` equals 2 in both cases, i.e. the weak components are bipartite. `WCOMP1` is an f-tree with vertex `A05` as root.
*A-acyclic weak components with strong components:* `WCOMP2` and `WCOMP3`. The weak components being a-acyclic all its strong components must be f-acyclic. `WCOMP2` is an f-tree with vertex `B02` as root. There are two strong components consisting of one edge each. The first – `SCOMP0` – has level number 1, the second – `SCOMP1` – level number 2. The f-period `fper` equals 2 for both strong components. The external dag consists of two arcs and three vertices. Altogether there are 2 weak attachment points. Weak component `WCOMP3` also contains two strong components. The vertices of the first – `SCOMP0` – are both roots of the f-tree structure of `WCOMP3`. The second strong component – `SCOMP1` – has level number 1. The external dag consists of one arc, whose incidence points are weak attachment points.
*A-cyclic weak components without strong components:* `WCOMP4` and `WCOMP5` contain a-circuits but no strong components. `WCOMP5` has root vertex `C05` but is not an a-tree.
*A-cyclic weak components with strong components (all strong components are f-acyclic):* `WCOMP6` and `WCOMP7` are also a-cyclic. They contain strong components but no f-circuits. Root vertex of `WCOMP6` is vertex `D02` and there are two strong components. Weak component `WCOMP7` consists of three strong components at levels 0, 1 , and 2, respectively. The vertices of the first strong component – `D06` and `D08` – are root vertices.
*A-cyclic weak components with strong components (f-cyclic strong components exist):* `WCOMP8, WCOMP9, WCOMP10, WCOMP11` and `WCOMP12` all of which are not bipartite (`aper = 1`). The first 4 illustrate simple special cases. The last – `WCOMP12` – contains two f-acyclic strong components and three strong components with f-circuits. Strong components `SCOMP0` and `SCOMP1` are f-acyclic and consist of one single edge each. Strong components `SCOMP2` and `SCOMP4` are made up of a single loop. The first has level number 0 and its vertex `E07` is root vertex of the weak component. The second contains vertex `E20` and has level number 2.

Strong component `SCOMP3` contains 14 vertices (namely `E08, E09, E10, E11, E12, E14, E15, E17, E18, E19, E21, E22, E24`, and `E25`), 8 edges, and 9 arcs. It is f-aperiodic (`fper = 1`). It obviously has a more complex structure than the other strong components considered so far. It makes sense to decompose it further. Example A.4, page 96, shows the biblock decomposition of this strong component.

The arcs from `E07` to `E09`, from `E12` to `E13`, from `E19` to `E20`, and from `E25` to `E26` generate the external dag.

□

```
BEGIN WEAK AND STRONG COMPONENTS - REDUCED STRUCTURE
$GRAPH wcompgraph
$TYPE  GG
$No_VERTICES                65
$No_EDGES                   25
$No_ARCS                    44
$No_ISOLATED_VERTICES        0
$No_WEAK_COMPONENTS                                             13
$No_A-ACYCLIC_WEAK_COMPONENTS_WITHOUT_STRONG_COMPONENTS          2     (9V, 0E, 7A)
   $WEAK_COMPONENT wcompgraph.WCOMP0  (4V, 0E, 3A)  aper = 2
   $WEAK_COMPONENT wcompgraph.WCOMP1  (5V, 0E, 4A)  aper = 2
      f-TREE   root: A05 (vertex)
$No_A-ACYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS             2     (10V, 5E, 3A)
   $WEAK_COMPONENT wcompgraph.WCOMP2  (5V, 2E, 2A)  aper = 2
      f-TREE   root: B02 (vertex)
   $No_STRONG_COMPONENTS                                         2
   $No_f-ACYCLIC_STRONG_COMPONENTS                               2     (4V, 2E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP2.SCOMP0  (2V, 1E, 0A)  (1WAP) lv = 1  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP2.SCOMP1  (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP2.EXD  (3V, 0E, 2A)  (2WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP3  (5V, 3E, 1A)  aper = 2
      f-TREE   root: wcompgraph.WCOMP3.SCOMP0 (strong component)
   $No_STRONG_COMPONENTS                                         2
   $No_f-ACYCLIC_STRONG_COMPONENTS                               2     (5V, 3E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP3.SCOMP0  (3V, 2E, 0A)  (1WAP) lv = 0  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP3.SCOMP1  (2V, 1E, 0A)  (1WAP) lv = 1  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP3.EXD  (2V, 0E, 1A)  (2WAP)
$No_A-CYCLIC_WEAK_COMPONENTS_WITHOUT_STRONG_COMPONENTS           2     (8V, 0E, 8A)
   $WEAK_COMPONENT wcompgraph.WCOMP4  (4V, 0E, 4A)  aper = 2
   $WEAK_COMPONENT wcompgraph.WCOMP5  (4V, 0E, 4A)  aper = 2
      rooted   root: C05 (vertex)
$No_A-CYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS              2     (12V, 6E, 9A)
    $(ALL_STRONG_COMPONENTS_F-ACYCLIC)
   $WEAK_COMPONENT wcompgraph.WCOMP6  (5V, 2E, 3A)  aper = 2
      rooted   root: D02 (vertex)
   $No_STRONG_COMPONENTS                                         2
   $No_f-ACYCLIC_STRONG_COMPONENTS                               2     (4V, 2E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP6.SCOMP0  (2V, 1E, 0A)  (1WAP) lv = 1  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP6.SCOMP1  (2V, 1E, 0A)  (2WAP) lv = 2  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP6.EXD  (4V, 0E, 3A)  (3WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP7  (7V, 4E, 6A)  aper = 1
      rooted   root: wcompgraph.WCOMP7.SCOMP0 (strong component)
   $No_STRONG_COMPONENTS                                         3
   $No_f-ACYCLIC_STRONG_COMPONENTS                               3     (7V, 4E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP7.SCOMP0  (2V, 1E, 0A)  (2WAP) lv = 0  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP7.SCOMP1  (2V, 1E, 0A)  (2WAP) lv = 1  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP7.SCOMP2  (3V, 2E, 0A)  (3WAP) lv = 2  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP7.EXD  (7V, 0E, 6A)  (7WAP)
```

Table 21: *Decomposition of graph wcompgraph into weak and strong components (Part1)*

```
$No_A-CYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS         5    (26V, 14E, 17A)
    $(F-CYCLIC_STRONG_COMPONENTS_EXIST)
   $WEAK_COMPONENT wcompgraph.WCOMP8  (2V, 2E, 0A)  aper = 1
      rooted   root: wcompgraph.WCOMP8.SCOMP0 (strong component)
   $No_STRONG_COMPONENTS                                    1
   $No_f-ACYCLIC_STRONG_COMPONENTS                          0
   $No_f-CYCLIC_STRONG_COMPONENTS                           1
      $STRONG_COMPONENT wcompgraph.WCOMP8.SCOMP0  (2V, 2E, 0A)  (0WAP) lv = 0  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP8.EXD  (0V, 0E, 0A)  (0WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP9  (1V, 0E, 1A)  aper = 1
      rooted   root: wcompgraph.WCOMP9.SCOMP0 (strong component)
   $No_STRONG_COMPONENTS                                    1
   $No_f-ACYCLIC_STRONG_COMPONENTS                          0
   $No_f-CYCLIC_STRONG_COMPONENTS                           1
      $STRONG_COMPONENT wcompgraph.WCOMP9.SCOMP0  (1V, 0E, 1A)  (0WAP) lv = 0  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP9.EXD  (0V, 0E, 0A)  (0WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP10  (1V, 1E, 0A)  aper = 1
      rooted   root: wcompgraph.WCOMP10.SCOMP0 (strong component)
   $No_STRONG_COMPONENTS                                    1
   $No_f-ACYCLIC_STRONG_COMPONENTS                          0
   $No_f-CYCLIC_STRONG_COMPONENTS                           1
      $STRONG_COMPONENT wcompgraph.WCOMP10.SCOMP0  (1V, 1E, 0A)  (0WAP) lv = 0  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP10.EXD  (0V, 0E, 0A)  (0WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP11  (2V, 0E, 2A)  aper = 1
      rooted   root: E05 (vertex)
   $No_STRONG_COMPONENTS                                    1
   $No_f-ACYCLIC_STRONG_COMPONENTS                          0
   $No_f-CYCLIC_STRONG_COMPONENTS                           1
      $STRONG_COMPONENT wcompgraph.WCOMP11.SCOMP0  (1V, 0E, 1A)  (1WAP) lv = 1  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP11.EXD  (2V, 0E, 1A)  (1WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP12  (20V, 11E, 14A)  aper = 1
      rooted   root: wcompgraph.WCOMP12.SCOMP2 (strong component)
   $No_STRONG_COMPONENTS                                    5
   $No_f-ACYCLIC_STRONG_COMPONENTS                          2
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP0  (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP1  (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
   $No_f-CYCLIC_STRONG_COMPONENTS                           3
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP2  (1V, 1E, 0A)  (1WAP) lv = 0  fper = 1
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP3  (14V, 8E, 9A)  (4WAP) lv = 1  fper = 1
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP4  (1V, 0E, 1A)  (1WAP) lv = 2  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP12.EXD  (8V, 0E, 4A)  (8WAP)
END REDUCED STRUCTURE
```

Table 22: *Decomposition of graph wcompgraph into weak and strong components (Part2)*

## A.2 Derived Graphs

**Example A.2 (Condensed Graph)** In figure 25 `Graph2` is shown. Table 23 shows its

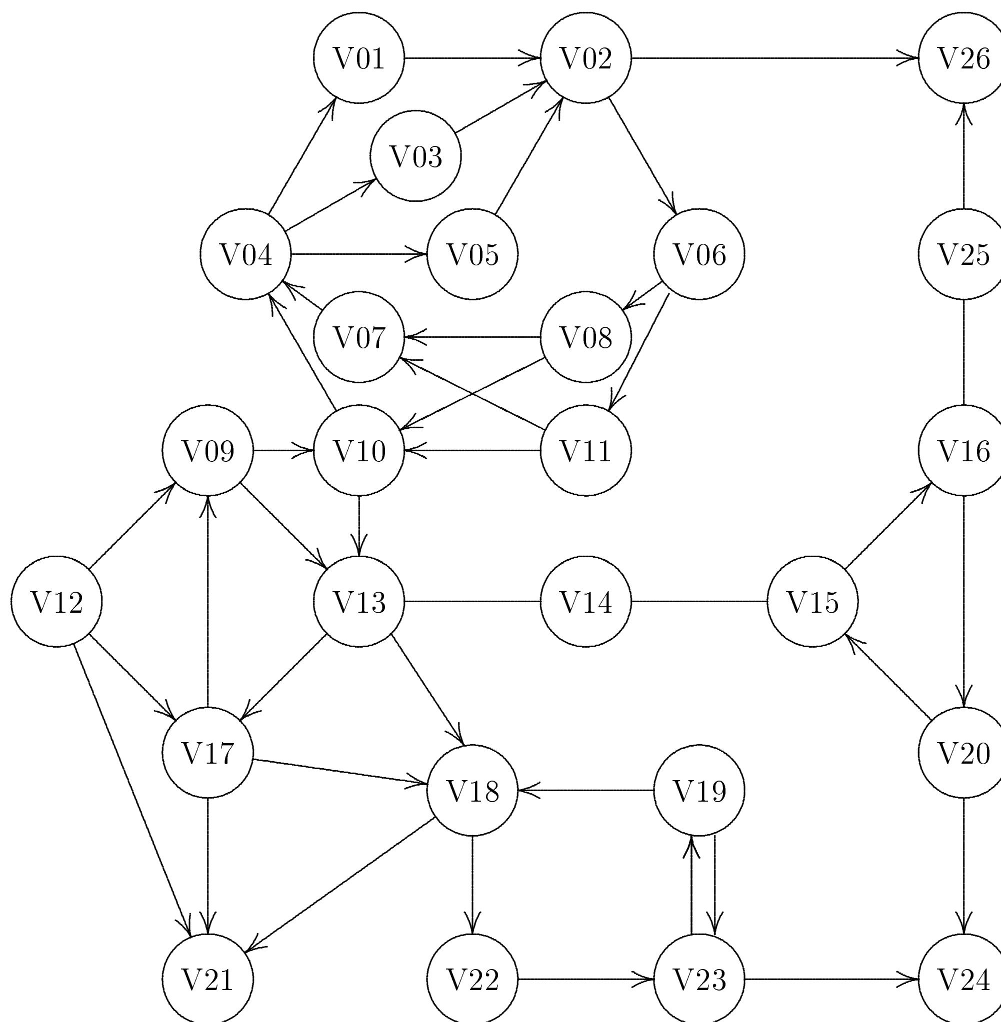


Figure 25: *Graph2*

decomposition into strong components and the external dag. The corresponding condensed graph can be seen in figure 26. Dotted lines separate different levels. $\{v_{12}\}$ has level number 0.

**Example A.3 (Component graph)** The component graph of Graph2 is shown in figure 27. □

| | | |
|---|---|---|
| $SC_1$ | $=$ | $\{V01, V02, V03, V04, V05, V06, V07, V08, V09, V10, V11, V13, V14, V15, V16, V17, V20, V25\}$<br>Weak attachment points: $\{V02, V09, V13, V17, V20, V25\}$ |
| $SC_2$ | $=$ | $\{V18, V19, V22, V23\}$<br>Weak attachment points: $\{V18, V23\}$ |
| $ED$ | $=$ | $\{dV12V09,\ dV12V17,\ dV12V21,\ dV13V18,\ dV17V18,\ dV17V21,\ dV18V21,\ dV23V24,\ dV20V24,\ dV02V26,\ dV25V26\}$ |

Table 23: *Strong components and external dag of Graph2*

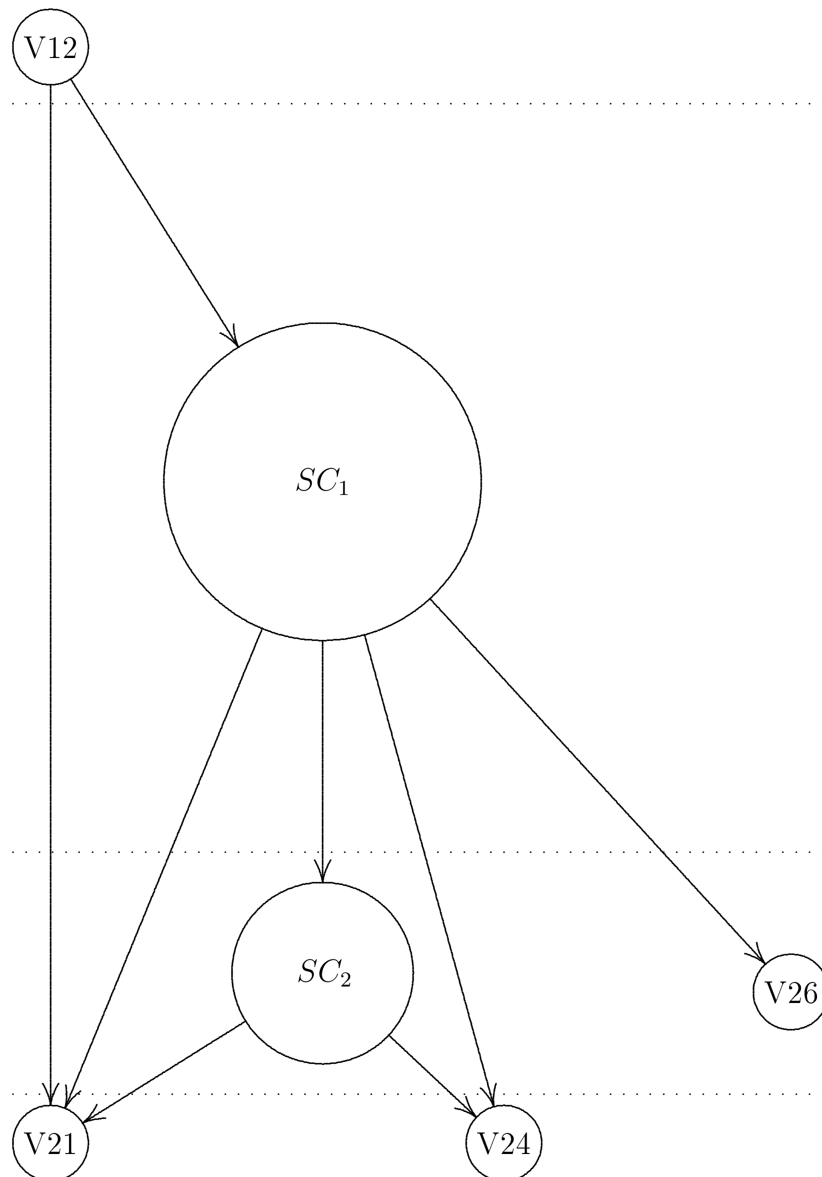


Figure 26: *Condensed graph of Graph2*

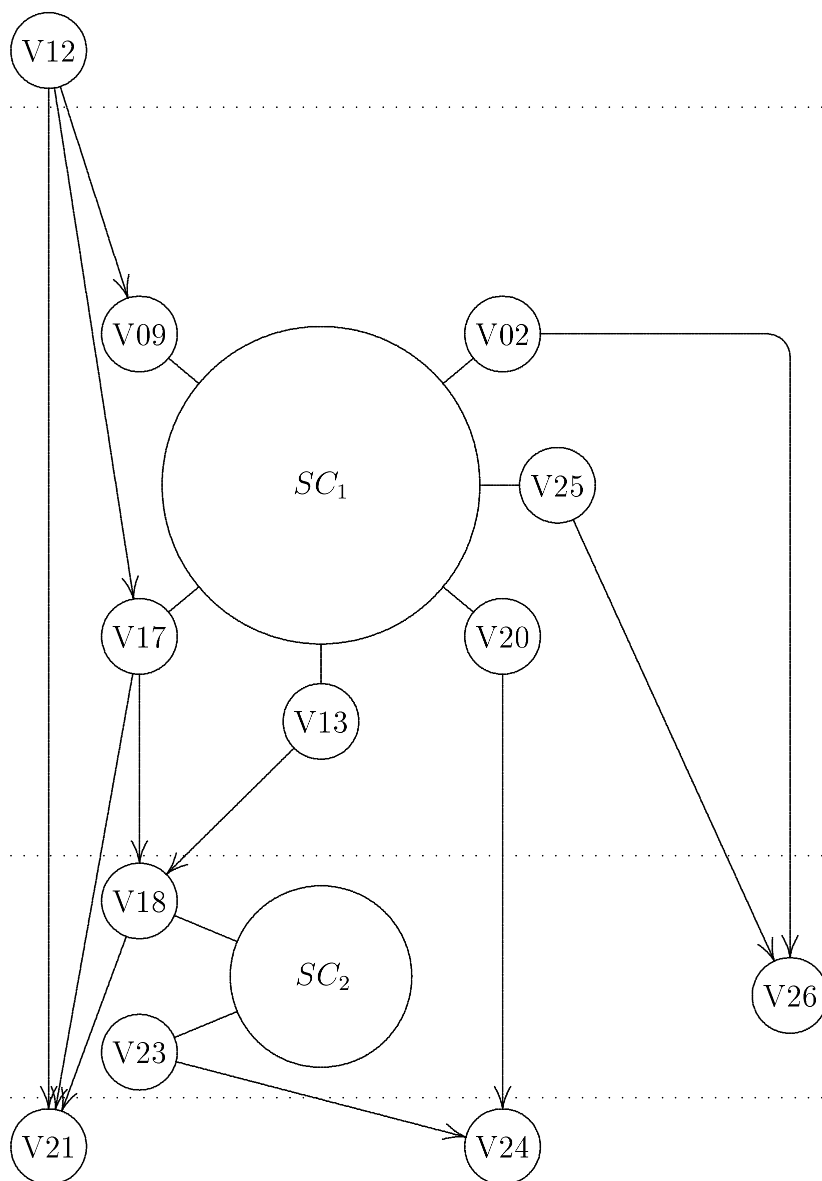


Figure 27: *Component graph corresponding to Graph2*

## A.3 Biblock Decomposition

**Example A.4 (Biblock decomposition of a strong component)** Figure 28 shows the strong component `SCOMP3` of the last weak component of graph `wcompgraph` on page 89. As an independent graph it has been renamed to `scomp3graph`. `SCOMP3` is the non-

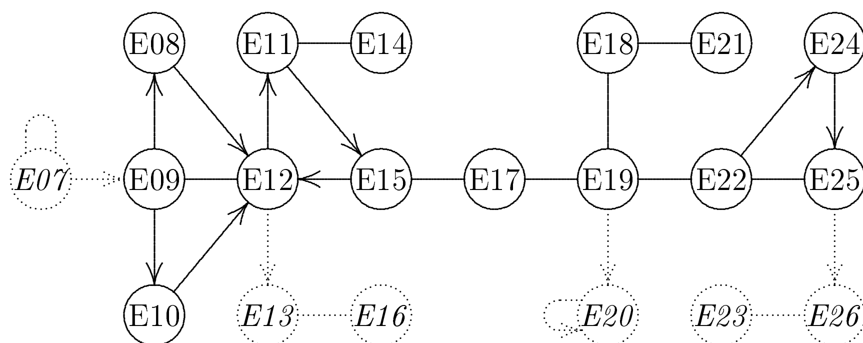


Figure 28: *Strong component* `wcompgraph.WCOMP12.SCOMP3`

dotted part of the figure. Table 24 shows the biblock decomposition of `SCOMP3`. There are 2 peripheral trees. The stopfree kernel consists of 2 subcomponents and 1 internal tree. One subcomponent is built up from 1 single biblock, the other contains 2 biblocks. Altogether there are 2 border points (`E11` and `E19`), 2 check points (`E15` and `E22`), and 1 hinge point (`E12`). □

**Example A.5 (Biblock decomposition and biblock graph)** Example 5.1, page 26, shows the biblock decomposition of graph *Ugraph1*, page 26. For a listing of this decomposition as produced by GHS see table 25.

Figure 29 shows the biblock graph of graph *Ugraph1*. The elementary building blocks of the biblock decomposition, namely biblocks, peripheral trees and internal trees, are depicted by boxes of different shapes. Small circles represent the attachment points. The subcomponents are identified by dashed lines, the stopfree kernel is delimited by a dotted line. □

```
BEGIN BIBLOCK DECOMPOSITION
REDUCED STRUCTURE
$GRAPH scomp3graph
$TYPE  GG
$No_VERTICES                14
$No_EDGES                    8
$No_ARCS                     9
$No_ISOLATED_VERTICES        0
$No_A-ACYCLIC_WEAK_COMPONENTS       0     (0V, 0E, 0A)
$No_A-CYCLIC_WEAK_COMPONENTS        1     (14V, 8E, 9A)
   $A-CYCLIC_WEAK_COMPONENT scomp3graph.WCOMP0  (14V, 8E, 9A, 5AP, 2BP, 2CP, 1HP)
   $No_PERIPHERAL_TREES 2
        $PERIPHERAL_TREE scomp3graph.WCOMP0.PT0  (3V, 2E, 0A)
        $PERIPHERAL_TREE scomp3graph.WCOMP0.PT1  (2V, 1E, 0A)
        $STOPFREE_KERNEL scomp3graph.WCOMP0.STP  (11V, 5E, 9A, 5AP, 2BP, 2CP, 1HP)
        $No_SUBCOMPONENTS 2
            $SUBCOMPONENT scomp3graph.WCOMP0.SUB0     (3V, 1E, 2A, 1AP, 0BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK scomp3graph.WCOMP0.SUB0.BLB0          (3V, 1E, 2A, 1AP, 0BP, 1CP, 0HP)
            $SUBCOMPONENT scomp3graph.WCOMP0.SUB1     (6V, 1E, 7A, 3AP, 1BP, 1CP, 1HP)
            $No_BIBLOCKS 2
                $BIBLOCK scomp3graph.WCOMP0.SUB1.BLB0          (3V, 0E, 3A, 3AP, 1BP, 1CP, 1HP)
                $BIBLOCK scomp3graph.WCOMP0.SUB1.BLB1          (4V, 1E, 4A, 1AP, 0BP, 0CP, 1HP)
        $No_INTERNAL_TREES 1
            $INTERNAL_TREE scomp3graph.WCOMP0.IT0    (4V, 3E, 0A, 3AP, 1BP, 2CP, 0HP)
END REDUCED STRUCTURE
```

Table 24: *Biblock decomposition of* `scomp3graph`

```
BEGIN BIBLOCK DECOMPOSITION
REDUCED STRUCTURE
$GRAPH Ugraph1
$TYPE  UGSLF
$No_VERTICES              40
$No_EDGES                 44
$No_ARCS                   0
$No_ISOLATED_VERTICES      1
$No_A-ACYCLIC_WEAK_COMPONENTS       1     (5V, 4E, 0A)
$No_A-CYCLIC_WEAK_COMPONENTS        1     (34V, 40E, 0A)
   $A-CYCLIC_WEAK_COMPONENT Ugraph1.WCOMP1   (34V, 40E, 0A, 5AP, 2BP, 3CP, 2HP)
   $No_PERIPHERAL_TREES 2
        $PERIPHERAL_TREE Ugraph1.WCOMP1.PT0  (5V, 4E, 0A)
        $PERIPHERAL_TREE Ugraph1.WCOMP1.PT1  (2V, 1E, 0A)
        $STOPFREE_KERNEL Ugraph1.WCOMP1.STP  (29V, 35E, 0A, 5AP, 2BP, 3CP, 2HP)
        $No_SUBCOMPONENTS 3
            $SUBCOMPONENT Ugraph1.WCOMP1.SUB0     (23V, 27E, 0A, 2AP, 1BP, 1CP, 2HP)
            $No_BIBLOCKS 3
                $BIBLOCK Ugraph1.WCOMP1.SUB0.BLB0          (3V, 3E, 0A, 1AP, 0BP, 0CP, 1HP)
                $BIBLOCK Ugraph1.WCOMP1.SUB0.BLB1          (4V, 4E, 0A, 1AP, 1BP, 1CP, 1HP)
                $BIBLOCK Ugraph1.WCOMP1.SUB0.BLB2          (18V, 20E, 0A, 2AP, 1BP, 1CP, 2HP)
            $SUBCOMPONENT Ugraph1.WCOMP1.SUB1     (3V, 3E, 0A, 2AP, 1BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK Ugraph1.WCOMP1.SUB1.BLB0          (3V, 3E, 0A, 2AP, 1BP, 1CP, 0HP)
            $SUBCOMPONENT Ugraph1.WCOMP1.SUB2     (3V, 3E, 0A, 1AP, 0BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK Ugraph1.WCOMP1.SUB2.BLB0          (3V, 3E, 0A, 1AP, 0BP, 1CP, 0HP)
        $No_INTERNAL_TREES 1
            $INTERNAL_TREE Ugraph1.WCOMP1.IT0    (3V, 2E, 0A, 3AP, 1BP, 3CP, 1HP)
END REDUCED STRUCTURE
```

Table 25: *Biblock decomposition of graph Ugraph1*

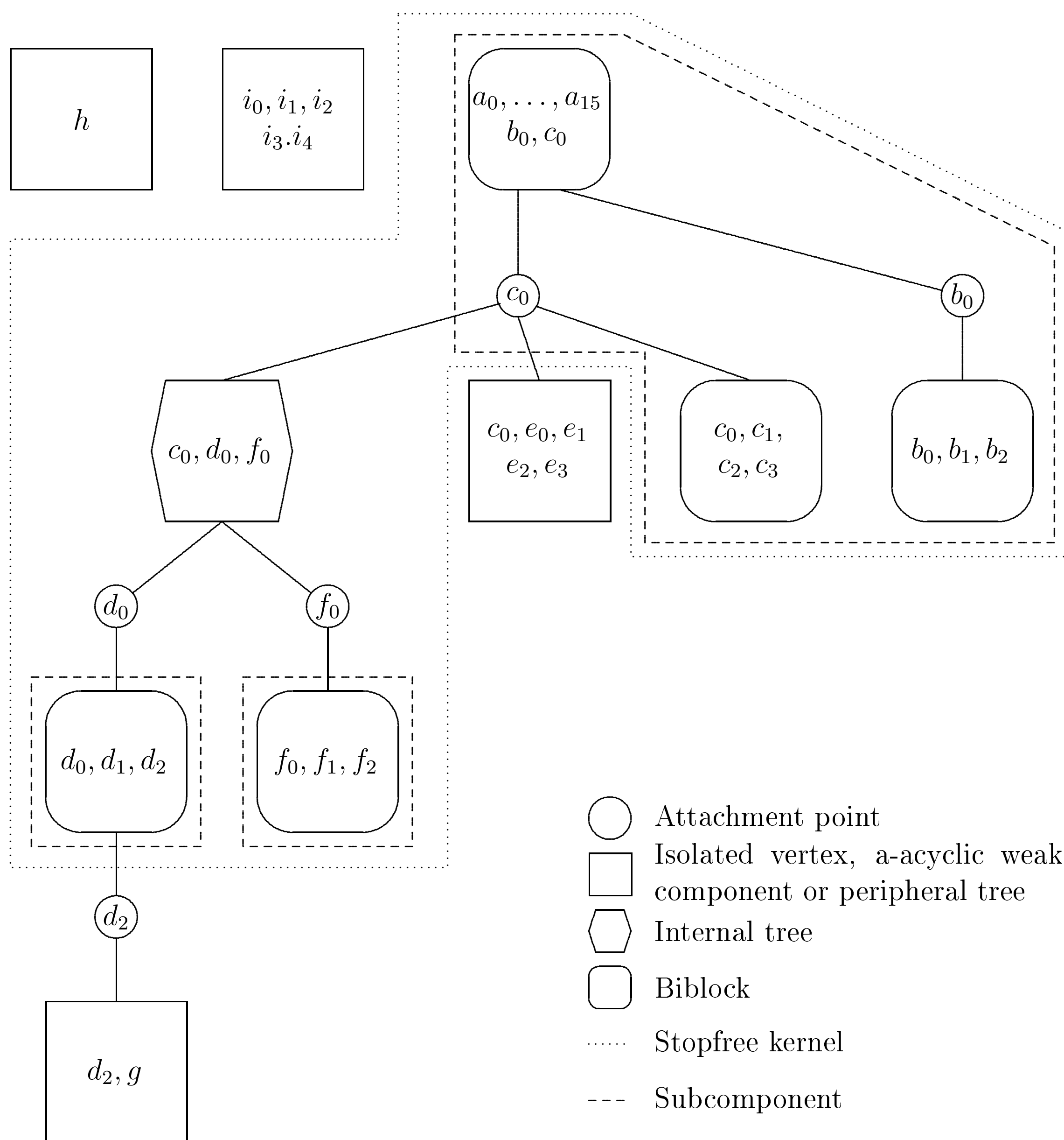


Figure 29: *The biblock graph of graph Ugraph1*

# List of Figures


1 *Multiple lines* . . . 5
2 *f-paths from root vertices* . . . 13
3 *F-depth-first search does not localize every f-circuit* . . . 17
4 *Examples of closed a-paths* . . . 25
5 *Ugraph1* . . . 26
6 *Biblock decomposition of a general graph* . . . 27
7 *Period of a strongly connected digraph* . . . 39
8 *Mutually reachable vertices have the same period* . . . 40
9 *Calculating the f-period* . . . 44
10 *Unravel Routine* . . . 49
11 *Example illustrating the Menger algorithm: Initial configuration* . . . 50
12 *Example illustrating the Menger algorithm: After finishing the step loop* . . . 51
13 *Example illustrating the Menger Algorithm: After finishing the unravel routine* . . . 52
14 *Example of Menger vertices* . . . 59
15 *A graph with two 3-a-linecomponents linked by three line-disjoint a-paths* . 67
16 *Three 3-a-components with a common edge* . . . 72
17 *A 4-a-cocomponent consisting of one single edge* . . . 72
18 *More than $k-1$ internally disjoint a-paths between two different k-a-components* . . . 73
19 *A k-a-component and a k-a-cocomponent may share mor than $k-1$ vertices.* 73
20 *Hierarchical decomposition into k-a-linecomponents and k-a-components* . . 74
21 *3-a-component $C_1$ contains 4-a-components $C_{11}$, $C_{12}$, and $C_{13}$* . . . 75
22 *3-a-component $C_4$ does not contain a 4-a-component* . . . 75
23 *3-a-component $C_2$ contains 4-a-components $C_{21}$ and $C_{22}$* . . . 76
24 *wcompgraph* . . . 89
25 *Graph2* . . . 93
26 *Condensed graph of Graph2* . . . 94
27 *Component graph corresponding to Graph2* . . . 95
28 *Strong component* `wcompgraph.WCOMP12.SCOMP3` . . . 96
29 *The biblock graph of graph Ugraph1* . . . 99

# List of Tables

1 *Decomposition into weak and strong components* . . . 13
2 *Procedure f-DFS for f-depth-first search in a general graph* . . . 15
3 *Frame for a complete f-depth-first search in a general graph* . . . 16
4 *Algorithm WCOMP for finding weak component* . . . 19
5 *Algorithm PTOPSORT for topological presorting a general graph* . . . 20
6 *Procedure STRCP for finding the strong components of a general graph* . . . 21
7 *Algorithm STRONGCOMP for finding the strong components of a general graph* . . . 22
8 *Procedure f-BFS for f-breadth-first search in a general graph* . . . 23
9 *Frame for complete f-breadth-first search in a general graph* . . . 23
10 *a-classification of lines* . . . 29
11 *a-classification of vertices* . . . 30
12 *Algorithm for finding peripheral trees* . . . 31
13 *Starting vertex for STPFKERNEL* . . . 32
14 *Recursive algorithm for determining the stopfree kernel (part I).* . . . 33
15 *Recursive algorithm for determining the stopfree kernel part (part II)* . . . 34
16 *Recursive procedure for finding the f-period of a strong component* . . . 43
17 *Algorithm STEPLOOP* . . . 53
18 *Breadth-first search in the step loop of the Menger algorithm* . . . 54
19 *Minimal Menger separating sets* . . . 61
20 *Algorithm for dynamic recognition of k-a-connected subgraphs* . . . 87
21 *Decomposition of graph wcompgraph into weak and strong components (Part1)* 91
22 *Decomposition of graph wcompgraph into weak and strong components (Part2)* 92
23 *Strong components and external dag of Graph2* . . . 94
24 *Biblock decomposition of* `scomp3graph` . . . 97
25 *Biblock decomposition of graph Ugraph1* . . . 98

# References

*Pages where cited are in parentheses.*

[CharL1996] Chartrand, Gary · Lesniak, Linda. *Graphs & Digraphs.* Chapman & Hall, 3 edition, 1996. *(3)*

[CormLR1990] Cormen, Thomas. H · Leiserson, Charles E. · Rivest, Ronald L. *Introduction to Algorithms.* The MIT Electrical and Computer Science Series. The MIT Press / McGraw-Hill Book Company, 1990. *(18)*

[Dies2000a] Diestel, Reinhard. *Graph Theory.* Number 173 in Graduate Texts in Mathematics. Springer, 2 edition, 2000. *(64)*

[Dira1966] Dirac, G. A. Short proof of Menger's graph theorem. *Mathematica*, 13:42–44, 1966. *(64)*

[EliaFS1956] Elias, P. · Feinstein, A · Shannon, C.E. A note on the maximum flow through a network. *IRE Trans. Inform. Theory*, IT-2:117–119, 1956. *(64)*

[FordF1956] Ford, L.R. · Fulkerson, D.R. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956. *(64)*

[GareJ1979] Garey, Michael R. · Johnson, David S. *Computers and Intractability.* A Series of Books in Mathematical Sciences. W. H. Freeman and Company, 1979. Tweny-first printing 1999. *(3)*

[Grun1938] Grünwald, Tibor. Ein neuer Beweis eines Mengerschen Satzes. *Journal of the London Mathematical Society*, 13:188–192, 1938. *(64)*

[Hara1969] Harary, Frank. *Graph Theory.* Addison-Wesley Series in Mathematics. Addison-Wesley Publishing Company, 1969. *(39, 64, 80)*

[HopcT1973a] Hopcroft, John E. · Tarjan, Robert Endre. Dividing a Graph into Triconnected Components. *SIAM Journal of Computing*, 2(3):135–158, 1973. *(80)*

[IsaaM1976] Isaacson, Dean L. · Madsen, Richard W. *Markov Chains Theory and Applications.* Robert E. Krieger Publishing Company, Inc., 1976. *(46)*

[Jung1994] Jungnickel, Dieter. *Graphen, Netzwerke und Algorithmen.* BI Wissenschaftsverlag, 3 edition, 1994. Vollständig überarbeitete und erweiterte Auflage. *(80)*

[McHu1990] McHugh, James A. *Algorithmic Graph Theory.* Prentice Hall, 1990. *(83)*

[Meng1927] Menger, Karl. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927. *(46, 63)*

[Sach1970] Sachs, Horst. *Einführung in die Theorie der endlichen Graphen I.* Mathematisch-Naturwissenschaftliche Bibliothek. BSB B. G. Teubner, 1970. *(64)*

[Stie1995d] Stiege, Günther. Periodicity: An Application of Digraphs to Markov Chains. Hildesheimer Informatik-Berichte 24/95, Universität Hildesheim, 1995. Available from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/hib95-24.html.* *(46)*

[Stie1996b] Stiege, Günther. Connectivity and Periodicity in Undirected Graphs. Hildesheimer Informatik-Berichte 31/96, Universität Hildesheim, 1996. Available by www from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/hib96-31.html.* *(39)*

[Stie1997] Stiege, Günther. Remarks on Periods and Colorings in Graphs. Hildesheimer Informatik-Berichte 03/97, Universität Hildesheim, 1997. Available by www from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/hib97-03.html.* *(39)*

[Stie1997a] Stiege, Günther. An Algorithm for Finding the Connectivity Structure of Undirected Graphs. Technical Report 13/97, Universität Hildesheim, 1997. Available from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/hib97-13.html.* *(40)*

[Stie1998] Stiege, Günther. Edge Partitions in Undirected Graphs. Berichte aus dem Fachbereich Informatik 5/98, Universität Oldenburg, 1998. Available from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/oib98-05.html.* *(39, 64)*

[Stie2001c] Stiege, Günther. Standard Decomposition and Periodicity of Digraphs. Berichte aus dem Fachbereich Informatik 5/01, Universität Oldenburg, 2001. Available from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/oib01-05.html.* *(80)*

[Stie2006] Stiege, Günther. *Graphen und Graphalgorithmen.* Reihe Informatik. Shaker Verlag, 2006. *(4, 21, 36, 58, 59, 77, 84, 86, 87)*

[StieS2003] Stiege, Günther · Stierand, Ingo. Connectedness Based Hierarchical Decomposition of Undirected Graphs. Bericht 3/03, Department für Informatik der Universität Oldenburg, 2003. *(80, 89)*

[Tarj1972] Tarjan, Robert Endre. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):215–225, 1972. *(39)*

[ThulS1992] Thulasiraman, K · Swamy, N. M. S. *Graphs: Theory and Algorithms.* John Wiley & Sons, Inc., 1992. *(64, 83)*

[Tutt1984] Tutte, W. T. *Graph Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1984. *(64)*

[Wagn1970] Wagner, Klaus. *Graphentheorie.* BI Hochschultaschenbücher. Bibliographisches Institut, 1970. *(64)*