Institute for Software Technology

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

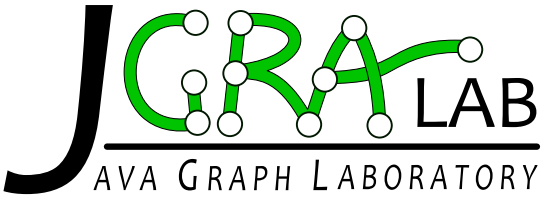
Mid-study thesis

"*Algolib*, a generic algorithm library for JGraLab"

Sascha Strauß

(206110097)

strauss@uni-koblenz.de



Advisors: Prof. Dr. Jürgen Ebert, Dr. Volker Riediger

{ebert | riediger}@uni-koblenz.de

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den 2. Mai 2011 . . . . . .

Abstract

Graphs are known to be a good representation of structured data. TGraphs, which are typed, attributed, ordered, and directed graphs, are a very general kind of graphs that can be used for many domains. The Java Graph Laboratory (JGraLab) provides an efficient implementation of TGraphs with all their properties. JGraLab ships with many features, including a query language (GReQL2) for extracting data from a graph. However, it lacks a generic library for important common graph algorithms.

This mid-study thesis extends JGraLab with a generic algorithm library called *Algolib*, which provides a generic and extensible implementation of several important common graph algorithms. The major aspects of this work are the generic nature of *Algolib*, its extensibility, and the methods of software engineering that were used for achieving both. *Algolib* is designed to be extensible in two ways. Existing algorithms can be extended for solving specialized problems and further algorithms can be easily added to the library.

Abstract

Graphen sind eine gute Wahl um strukturierte Daten zu repräsentieren. TGraphen (typisierte, attributierte, geordnete und gerichtete Graphen) sind eine sehr generische Graphenart, die in vielen Bereichen verwendet werden können. Das Java Graphenlabor (JGraLab) bietet eine effiziente Implementierung von TGraphen mit all ihren Eigenschaften. Zusätzlich stellt es, unter anderem, die Anfragesprache GReQL2 zur Verfügung, die dazu verwendet werden kann, Daten aus einem Graphen zu extrahieren. Es verfügt jedoch nicht über eine generische Bibliothek von gängigen Graphalgorithmen.

Diese Studienarbeit ergänzt JGraLab durch eine generische Algorithmenbibliothek namens *Algolib*, die eine generische und erweiterbare Implementierung einiger wichtiger gängiger Graphalgorithmen enthält. Das Hauptaugenmerk dieser Arbeit liegt auf der Generizität von *Algolib*, ihrer Erweiterbarkeit und der Methoden der Softwaretechnik die benutzt wurden um beides zu erreichen. *Algolib* ist auf zwei Weisen erweiterbar. Bereits enthaltene Algorithmen können erweitert werden um speziellere Probleme zu lösen und weitere Algorithmen können auf einfache Weise der Bibliothek hinzugefügt werden.

Contents

1 Introduction

This chapter gives a short introduction into JGraLab and TGraphs. It also contains the motivation for developing a graph algorithm library and shows the goals of this mid-study thesis.

1.1 JGraLab and TGraphs

Graphs are a suitable representation for structured data. They are used for representing arbitrary entities and the relationships between them.

For providing a generic graph representation, TGraphs have been developed. *TGraphs* are typed, attributed, ordered, and directed graphs. A mathematical definition for TGraphs can be found in section 2.4 on page 28. The Java Graph Laboratory (*JGraLab*), provides an efficient implementation of TGraphs with the properties mentioned above.

JGraLab is based upon an older implementation, written in C++, called *GraLab*. The base implementation of JGraLab was performed by Steffen Kahle in his diploma thesis [Kah06]. The development was continued by the working group of Jürgen Ebert at the Institute for Software Technology at the University of Koblenz.

The first official version of JGralab (Version *Anatotitan*[1]) was released November 2nd 2007. JGraLab's current version is *Dimetrodon*.

1.1.1 Graph schemas

A TGraph always corresponds to a certain so-called *graph schema*. Such a graph schema defines the types and attributes of graph elements. It also defines which edge types are possible between which vertex types. Graph schemas can be defined in *grUML*, a subset of UML [BHR+10].

A *grUML* diagram is a UML class diagram following certain rules. It declares one so-called *graph class*, which is the type of the graph. It further declares the vertex and edge classes, where vertex classes are visualized as UML classes and edge classes are visualized as either UML associations or UML association classes. Graph classes, vertex classes, and edge classes

[1] The versioning schema uses dinosaur names but does not use numbers. The position of the version name's first letter in the alphabet corresponds to the version number.

can have attributes. Furthermore, *grUML* allows generalization among vertex classes and edge classes. Also multiple inheritance is possible.

Attributes may be of several types, which are called *domains*. The simplest domains are the primitive domains (`Integer`, `Long`, `Double`, and `Boolean`). There are also domains that correspond to Java collections (`List`, `Set`, and `Map`). The last group of domains are self-defined domains. As the name suggests, they have to be explicitly defined in the *grUML* diagram. The first one is the enum domain, which corresponds to a Java enum type. The second one is the record domain, which corresponds to a tuple type, comparable to a `struct` in C.

grUML also supports packaging, meaning that vertex classes, edge classes, and the self-defined domains may be placed in different packages.

1.1.2 Object oriented access layer

The exported XMI model of a *grUML*diagram can be converted into JGraLab's TG file format[2]. Currently, this is only possible for models that have been exported with the tool *Rational Software Architect*[3] from IBM using the tool `rsa2tg`, which ships with JGraLab.

For working with graph instances belonging to a schema, JGraLab can provide an object oriented access layer. This layer has to be generated in order to work with instances. JGraLab contains a Java code generator that generates the code for this access layer from the TG file containing the desired schema. Afterwards the schema has to be compiled before it can be used. For schemas that are frequently used, the generated code and the compiled class files can be stored persistently. JGraLab also provides a way of doing this on-the-fly, meaning the classes are directly compiled for instant use.

The object oriented access layer of a schema provides several interfaces, corresponding to the classes that have been defined in the source TG file. It also reflects the schema's package structure. The schema itself is represented by the schema class. This class is a singleton and can be used, among other things, for creating new graphs, loading graphs from or storing graphs to a hard disk. When creating or loading a graph, an instance of the graph class, belonging to the schema, is returned, which represents the instance of the graph. All graph operations are performed on the graph class instance.

Persistent serialized copies of a graph are stored in TG files. The header of a TG file is always the schema of the graph. The remainder, which is mostly the major part of the file, contains the serialized, but still human readable representation of the TGraph.

[2] It is also possible to define a schema directly in TG.

[3] `http://www-01.ibm.com/software/rational/products/swarchitect/`

1.1.3 *GReQL2*

Since JGraLab's first official version, the second version of the Graph Repository Query Language (*GReQL2*) is shipped with JGraLab. It allows querying graphs for data which is stored in the graph.

The *GReQL2* language is based on the original *GReQL* language that was used in the old GraLab. The *GReQL2* language itself was specified by Katrin Marchewka in her diploma thesis [Mar06]. She also implemented a *GReQL2*parser in this work. The *GReQL2* evaluator was implemented by Daniel Bildhauer in his diploma thesis [Bil08].

A *GReQL2* query is parsed into a TGraph representing this query. For reducing the evaluation time, an optimizer was created by Tassilo Horn in his diploma thesis [Hor09], that optimizes the query graph for achieving a better time efficiency.

An introduction to *GReQL2*, how it works, and how it can be used, can be found in [EB10].

1.2 Motivation

As indicated above, JGraLab is a very powerful implementation for TGraphs. It provides an efficient general purpose graph data structure. With *GReQL2*, it also provides a sophisticated querying mechanism, which makes handling large amounts of data stored in a TGraph fairly easy.

However, in some domains the capabilities of *GReQL2* are insufficient, because many graph problems cannot be solved with a bare *GReQL2* query. For solving such graph problems, there exist approved algorithms. The design of JGraLab's object oriented access layer allows for implementing these algorithms easily. One domain, where this has been done, is described later in this work (see chapter 6 for details).

If those algorithms are required in many different domains, multiple specialized variants would be implemented, solving the same problems in different contexts. For avoiding such an uncontrolled growth of specialized algorithms, it is feasible for JGraLab to provide a generic algorithm library, containing many important common graph algorithms. Such an algorithm library is developed in this work.

1.3 Goals

The goal of this mid-study thesis is the development of a generic, extensible algorithm library for JGraLab, which will be called *Algolib*in the following. First, the problems that are initially covered by *Algolib* have to be specified. Here, the algorithms solving those problems are also introduced conceptually. Afterwards, the mathematical structures, that arise from the problem

definitions, have to be mapped to Java data structures for being used by algorithm implementations. Then, concepts for making the algorithm implementations generic and extensible have to be introduced. With these concepts, the algorithms are implemented. Finally the implementation rules for algorithms have to be collected in order to add further algorithms to *Algolib* in the future.

Algolib is required to be generic, extensible, and fast. Generic means, that *Algolib*'s algorithm implementations have to be adjustable to arbitrary domains. Extensible means, that the algorithms can be used for solving problems, they are normally not capable of. Some algorithms, that are currently included in *Algolib*, demonstrate how this is done. Fast means, that these generic algorithms have to perform similarly well in comparison to domain specific implementations of the same algorithms. Naturally a generic approach is expected to be slower than a specialized one. A concrete goal, for a generic algorithm implementation that has been adjusted to a specific domain, is consuming at most twice the time a corresponding domain specific algorithm implementation does.

1.4 Overview

In chapter 2, some mathematical definitions are made about graphs and relations. It contains definitions about relations for two reasons. Firstly, relations are important for understanding the mathematical nature of graphs. And secondly, the solution of some problems solved by *Algolib* have relations as result.

Chapter 3 introduces some basic concepts and introduces some problems and their algorithmic solutions. The drafts of the algorithms in this chapter serve as base for *Algolib*'s generic implementation.

Chapter 4 shows how the mathematical structures, that arose from the problem definitions in chapter 3, are mapped to Java data structures. Those structures are used in *Algolib*'s implementation.

Chapter 5 shows how graph algorithms are implemented in *Algolib* while introducing the concepts for its generic nature and extensibility. It also gives examples on how graph algorithms can be implemented by extending other graph algorithms.

Chapter 6 shows how graph algorithms can be adjusted to a specific domain. The chosen domain also allows a comparison between a generic algorithm implementation from *Algolib* and a specialized algorithm implementation that already exists. This comparison is also shown in this chapter. Finally the overhead, that arises from extending graph algorithms, is measured, for judging the speed efficiency of *Algolib* and its mechanism for extending algorithms.

Chapter 7 summarizes rules for extending *Algolib* with new algorithms. Some of them arise from chapter 5, others are less obvious. Every developer, planning on adding new algorithms to

Algolib, is required to obey these rules for creating algorithms that are compatible with *Algolib*'s concepts.

Finally chapter 8 compares the resulting implementation of *Algolib* with the goals from section 1.3. It evaluates the results and gives a proposal on *Algolib*'s future.

2 Definitions

In this chapter all relevant graph related definitions and some theorems are given. The proofs for the theorems are not included in this thesis. However, for some theorems there is a reference to a proof.

Section 2.1 is an excursus to relations. It gives some definitions concerning relations. They are necessary for some definitions concerning graphs. Sections 2.2, 2.3, and 2.4 contain definitions concerning directed graphs, undirected graphs, and *TGraphs* respectively.

2.1 Definitions on relations

This section is an excursus to relations and contains several definitions concerning them. These definitions are important for the understanding of some graph related definitions.

Definition: Relation
Given two sets X and Y,
$R \subseteq X \times Y$ is called a *relation* between X and Y.

Relations are sets of tuples. For relations the infix notation can be used. Given a relation R, then $(x, y) \in R \Leftrightarrow xRy$.

Given a relation $R \subseteq X \times Y$, two sets can be derived:

- $xR := \{y \in Y \mid (x, y) \in R\}$, $xR \subseteq Y$
- $Ry := \{x \in X \mid (x, y) \in R\}$, $Ry \subseteq X$

Definition: Function
Given two sets X and Y,
a *(partial) function* $f : X \nrightarrow Y$ is a right-unique relation between X and Y, meaning $\forall (x_1, y_1), (x_2, y_2) \in f : x_1 = x_2 \Rightarrow y_1 = y_2$.
The set $dom(f) := \{x \in X \mid \exists y \in Y : (x, y) \in f\}$ is called the *domain* of f. A function can have the following properties:

- *finite*, iff $dom(f)$ is finite; Notation: $f : X \nrightarrow Y$ and
- *total*, iff $dom(f) = X$; Notation $f : X \rightarrow Y$.

Definition: Relation on one set
Given a set X,
a relation $R \subseteq X \times X$ is called *relation on* X.

Relations on one set allow several operations on them.

Definition: Operations on relations
Given a set X and a relation R on X,
the relation $R^t := \{(y,x) \mid (x,y) \in R\}$ is called the *inverse relation*.
Given a set X, a relation R on X and a relation S on X,
the relation $RS := \{(x,z) \mid \exists y \in X : (x,y) \in R \wedge (y,z) \in S\}$ is called the *product* of R and S.
Given a set X and a relation R on X. The n-th *power of* R is inductively defined by:

- $R^0 := Id_X$ where $Id_x = \{(x,x) \mid x \in X\}$.
- $R^{i+1} = R^i R$

With these operations, several properties of relations can be defined.

Definition: Properties of relations
Given a set X and a relation R on X,
R is called:

1. reflexive, iff $R^0 \subseteq R$
2. irreflexive, iff $R^0 \cap R = \varnothing$
3. symmetric, iff $R = R^t$
4. asymmetric, iff $R \cap R^t = \varnothing$
5. antisymmetric, iff $R \cap R^t \subseteq R^0$
6. transitive, iff $R^2 \subseteq R$

Based on these properties, several new relations can be derived from a given relation. These derived relations are called *closure*s.

Definition: Closures
Given a set X and a relation R on X,
the following relations can be derived from R:

- $\widehat{R} := R \cup R^0$ is called the *reflexive closure*,
- $\overline{R} := R \cup R^t$ is called the *symmetric closure*,
- $R^+ := \bigcup_{k=1}^{\infty} R^k$ is called the *transitive closure* and
- $R^* := R^0 \cup R^+ = \bigcup_{k=0}^{\infty} R^k$ is called the *reflexive-transitive closure*

of the relation R.

Relations can be classified by their properties. One very important class of relations is the class of equivalence relations.

Definition: Equivalence relation
Given a relation $\equiv$,
$\equiv$ is called *equivalence relation*, iff it is reflexive, symmetric, and transitive.

If an equivalence relation R is a relation on a set X, this relation can be used to create a *partition* of X.

Definition: Partition
Given a set X,
the set $\mathcal{X} \subseteq \mathbb{P}(X) \setminus \{\varnothing\}$ is called *partition*, iff $\forall M, N \in \mathcal{X} : M \cap N = \varnothing \wedge \bigcup_{M \in \mathcal{X}} M = X$.

Theorem:
Given an equivalence relation $\equiv$ on a set X,
$\equiv$ creates a partition $\mathcal{X}$ with $\mathcal{X} = \{\{y \in X \mid y \equiv x\} \mid x \in X\}$.
The notation of such a partition is $X/_{\equiv}$.
The elements of $X/_{\equiv}$ are also called *equivalence classes*.

A partition can be described by a *representative function* defined below.

Definition: Representative function
Given a set X and a partition $\mathcal{X}$.
An equivalence class $X_i \in \mathcal{X}$ can be identified by an arbitrarily chosen $x_i \in X_i$. x_i is called the *representative* of the equivalence class X_i. A function $rep : X \to X$ that assigns to every element $x \in X$ the representative x_i of the equivalence class X_i with $x \in X_i$, is called *representative function*.

2.2 Definitions on directed graphs

After giving several definitions about relations, this section gives important definitions about directed graphs. The first definition is the graph itself.

Definition: Directed graph
A directed graph $G = (V, E, \varphi)$ consists of

1. a finite set V of vertices with $V \neq \varnothing$,
2. a finite set E of edges with $V \cap E = \varnothing$,
3. an incidence mapping $\varphi : E \to V \times V$.

Given a directed graph $G = (V, E, \varphi)$ and an edge $e \in E$ with $\varphi(e) = (v, w)$, v is called the *start vertex* and w is called the *end vertex* of e.
It is possible to define these vertices by functions:

- $\alpha : E \to V$ gives the start vertex of a given edge ($\alpha(e) = v$).
- $\omega : E \to V$ gives the end vertex of a given edge ($\omega(e) = w$).

If $\alpha(e) = \omega(e)$, e is called a *loop*.
Given a directed graph $G = (V, E, \varphi)$ and two vertices $v \in V$, $w \in V$,
The relation $\to \subseteq (V \times V)$ with $v \to w \Leftrightarrow \exists e \in E : \varphi(e) = (v, w)$ is called *successor relation*.
Given a directed graph $G = (V, E, \varphi)$, for a vertex $v \in V$

- $\Gamma^+(v) = \{w \in V \mid v \to w\}$ is called the set of direct successors of v,
- $\Gamma^-(v) = \{w \in V \mid w \to v\}$ is called the set of direct predecessors of v,
- $\Lambda^+(v) = \{e \in E \mid \alpha(e) = v\}$ is called the set of outgoing edges of v,
- $\Lambda^-(v) = \{e \in E \mid \omega(e) = v\}$ is called the set of incoming edges of v,
- $\delta^+(v) = |\Lambda^+(v)|$ is called the out-degree of v,
- $\delta^-(v) = |\Lambda^-(v)|$ is called the in-degree of v, and
- $\delta(v) = \delta^+(v) + \delta^-(v)$ is called the degree of v.

The following theorem correlates the edge number to the degree functions.

Theorem:
Given a directed graph $G = (V, E, \varphi)$,
$|E| = \sum_{v \in V} \delta^+(v) = \sum_{v \in V} \delta^-(v)$

The next definition uses the terminology that was just introduced for defining directed paths and terminology concerning directed paths.

Definition: Directed path
Given a directed graph $G = (V, E, \varphi)$,
an alternating sequence $C =< v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k >$, $k \geq 0$ is called directed path from v_0 to v_k, iff $\forall\, i, 1 \leq i \leq k : \alpha(e_i) = v_{i-1} \wedge \omega(e_i) = v_i$. The set of all paths in G is called $\mathcal{PATH}_G$.
$|C| := k$ is the *length* of C.
$\alpha(C) := v_0$ is the start vertex of C.
$\omega(C) := v_k$ is the end vertex of C.
$V(C) := \{v_0, \ldots, v_k\}$ is the *vertex set* of C.
$E(C) := \{e_1, \ldots, e_k\}$ is the *edge set* of C.
A directed path C with $|C| = 0$, is called *empty path*.
A directed path C with $|V(C)| = |C| + 1$ is called a *directed simple path* (there are no multiple occurrences of vertices in C).
A directed path C is called *closed* iff $\alpha(C) = \omega(C)$. A closed directed path is called *cycle* iff $|V(C)| = |C|$. G is called *acyclic*, iff G contains no cycles.

Theorem:
Based on the successor relation defined above, several other relevant relations can be derived: $\rightarrow^*$, $\rightarrow^k$, $\rightarrow^+$.
These relations have the following properties:

- $\rightarrow^*$: $(v, w) \in \rightarrow^* \Leftrightarrow \exists$ a directed path $C : \alpha(C) = v \wedge \omega(C) = w$,
- $\rightarrow^k$: $(v, w) \in \rightarrow^k \Leftrightarrow \exists$ a directed path $C : \alpha(C) = v \wedge \omega(C) = w \wedge |(|\, C) = k$, and
- $\rightarrow^+$: $(v, w) \in \rightarrow^+ \Leftrightarrow \exists$ a directed path $C : \alpha(C) = v \wedge \omega(C) = w \wedge |(|\, C) \geq 1$.

Definition: Relational graph
Given a directed graph $G = (V, E, \varphi)$,
G is called *relational*, iff φ is injective, meaning, there are no multiple edges.

For a relational graph $G = (V, E, \varphi)$, this definition yields $|E| \leq (|V|)^2$.

The following is the definition of a directed subgraph.

Definition: Subgraph
Given a directed graph $G = (V, E, \varphi)$,
a graph $G_s = (V_s, E_s, \varphi_{|_{E_s}})$ is called a *subgraph* of G, iff $V_s \subseteq V \wedge E_s \subseteq E$.

The following is a definition of a special graph, which is very important in various fields, including graph algorithms.

Definition: Directed rooted tree

A directed rooted tree $T = (V, E, \varphi)$ is a directed graph with the following properties:

- $\exists! r \in V$ with $\delta^-(r) = 0$. The vertex r is called the *root* of the tree.
- $\forall v \in V, v \neq r : \delta^-(v) = 1$. Every non-root vertex in the tree has exactly one predecessor.
- T is acyclic.

A vertex $v \in V$ with $\delta^+(v) = 0$ is called a *leaf*.
A vertex $w \in V$ with $\delta^+(w) > 0$ is called an *inner node*.
Given an edge $e \in E$ with $\varphi(e) = (v, w)$. v is called the *parent node* of w and w is called the *child node* of v.

Theorem:

Given a directed rooted tree $T = (V, E, \varphi)$,
$\forall v \in V \exists!$ a directed path C with $\alpha(C) = r \wedge \omega(C) = v$, meaning there is exactly one path for each vertex v leading from the root to v.

2.3 Definitions on undirected graphs

The definitions on undirected graphs, covered in this section, are similar to the corresponding definitions of directed graphs.

Definition: Undirected graph
An undirected graph $G = (V, E, \chi)$ consists of

1. a finite set V of vertices with $V \neq \varnothing$,
2. a finite set E of edges with $V \cap E = \varnothing$,
3. an incidence mapping $\chi : E \to \{W \subseteq V \mid 1 \leq |W| \leq 2\}$.

Given an undirected graph $G = (V, E, \chi)$ and two vertices $v \in V$, $w \in V$, the relation $\mapsto \subseteq (V \times V)$ is called *adjacency* relation with $v \mapsto w \Leftrightarrow \exists e \in E : \chi(e) = \{v, w\}$. $\mapsto$ is a symmetric relation.
Given an undirected graph $G = (V, E, \chi)$, for $v \in V$

- $\Gamma(v) = \{w \in V \mid v \mapsto w\}$ is called the set of adjacent vertices of v,
- $\Lambda(v) = \{e \in E \mid v \in \chi(e)\}$ is called the set of incident edges of v, and
- $\delta(v) = |\Lambda(v)| + |\{e \in E \mid \chi(e) = \{v\}\}|$ is called the degree of v.

Note that loops are counted twice by $\delta(v)$. This is done so the following theorem is compatible with the corresponding theorem on directed graphs.

Theorem:
Given an undirected graph $G = (V, E, \chi)$,
$\sum_{v \in V} \delta(v) = 2 * |E|$

Undirected paths can be defined, in analogy to the definition of directed paths.

Definition: Undirected path
Given an undirected graph $G = (V, E, \chi)$,
an alternating sequence $C =< v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k >$, $k \geq 0$ is called *undirected path* from v_0 to v_k, iff $\forall\, i, 1 \leq i \leq k : \chi(e_i) = \{v_{i-1}, v_i\}$. The set of all paths in G is called $\mathcal{PATH}_G$.
$|C| := k$ is the *length* of C.
$\alpha(C) := v_0$ is the start vertex of C.
$\omega(C) := v_k$ is the end vertex of C.
$V(C) = \{v_0, \ldots, v_k\}$ is the *vertex set* of C.
$E(C) = \{e_1, \ldots, e_k\}$ is the *edge set* of C.
An undirected path C with $|C| = 0$, is called *empty path*.
An undirected path C with $|V(C)| = |C| + 1$ is called a *directed simple path* meaning, there are no multiple occurrences of vertices in C.
A directed path C is called *closed* iff $\alpha(C) = \omega(C)$. A closed directed path is called *cycle* iff $|V(C)| = |C| \wedge |E(C)| = |(|\, C)$.
Given an undirected graph $G = (V, E, \chi)$,
G is called *cycle-free*, iff G contains no cycles.

Theorem:
Based on the adjacency relation defined above, several other relevant relations can be derived: $\overset{*}{\mapsto}$, $\mapsto^k$, $\mapsto^+$. The following list describes these derived relations:

- $\overset{*}{\mapsto}$: $(v, w) \in \overset{*}{\mapsto} \Leftrightarrow \exists$ an undirected path $C : (\alpha(C) = v \wedge \omega(C) = w) \vee (\alpha(C) = w \wedge \omega(C) = v)$,
- $\mapsto^k$: $(v, w) \in \mapsto^k \Leftrightarrow \exists$ a directed path $C : \alpha(C) = v \wedge \omega(C) = w \wedge |(|\, C) = k$, and
- $\mapsto^+$: $(v, w) \in \mapsto^+ \Leftrightarrow \exists$ a directed path $C : \alpha(C) = v \wedge \omega(C) = w \wedge |(|\, C) \geq 1$.

Undirected graphs are generalizations of directed graphs.

Definition: Underlying graph
Given a directed graph $G = (V, E, \varphi)$,
The graph $U(G) = (V, E, \chi)$ is called the *underlying graph* of G, iff $\forall\, e \in E : \chi(e) = \{\alpha(e), \omega(e)\}$.

The definitions for Γ, Λ, δ and $\mapsto$ of $U(G)$ are compatible with the corresponding definitions on G.

Theorem:
Given a vertex $v \in V$:

- $\Gamma(v) = \Gamma^-(v) \cup \Gamma^+(v)$,
- $\Lambda(v) = \Lambda^-(v) \cup \Lambda^+(v)$,
- $\delta(v) = \delta^-(v) \cup \delta^+(v)$, and
- $\mapsto = \rightarrow \cup \leftarrow$

A directed path in G is also an undirected path in $U(G)$.

So every directed graph can also be interpreted as an undirected graph. This implies, that all concepts, definitions, and algorithms on undirected graphs can also be applied to directed graphs.

As a convention, when just speaking of a graph $G = (V, E, \varphi)$, both a directed or an undirected graph is possible. This is important for several problem definitions in chapter 3, which are defined for both, directed and undirected graphs.

2.4 Definitions on TGraphs

The following definition on TGraphs is taken from [EB10].

TGraphs are typed, attributed, and ordered directed graphs, i.e. all graph elements are typed and may carry type-dependent attribute values. Furthermore, there are orderings of the vertex and the edge sets of the graph and of the incidences at all vertices. Lastly, all edges are assumed to be directed.

Definition: TGraph

Let

- *Vertex* be the universe of *vertices*,
- *Edge* be the universe of *edges*,
- *TypeId* be the universe of *type identifiers*,
- *AttrId* be the universe of *attribute identifiers*, and
- *Value* be the universe of *attribute values*.

Assuming two finite sets,

- a *vertex set* $V \subseteq Vertex$ and
- an *edge set* $E \subseteq Edge$,

be given. $G = (Vseq, Eseq, \Lambda seq, type, value)$ is a *TGraph* iff

- $Vseq \in \text{iseq}\, V$ is a permutation of V,
- $Eseq \in \text{iseq}\, E$ is a permutation of E,
- $\Lambda seq : V \to \text{iseq}(E \times \{in, out\})$ is an *incidence function* where
  $\forall e \in E : \exists! v, w \in V : (e, out) \in \text{ran}\, \Lambda seq(v) \wedge (e, in) \in \text{ran}\, \Lambda seq(w)$,
- $type : V \cup E \to TypeId$ is a *type function*, and
- $value : V \cup E \to (AttrId \twoheadrightarrow Value)$ is an *attribute function* where
  $\forall x, y \in V \cup E : type(x) = type(y) \Rightarrow \text{dom}(value(x)) = \text{dom}(value(y))$.

Thus, a TGraph consists of an ordered vertex set V and an ordered edge set E. They are connected by the incidence function Λseq which assigns the sequence of its incoming and outgoing edges to each vertex. For a given edge e, $\alpha(e)$ and $\omega(e)$ denote its *start vertex* and *target vertex*, respectively. Furthermore, all elements (i.e. vertices and edges) have a type and carry a type dependent set of attribute-value pairs.

TGraphs can also be seen as regular directed graphs. The vertex set and the edge set can remain. The incidence function φ can be derived from Λ_{seq} as follows:
$\forall e \in E, u, v \in V : \ \varphi(e) = (u, v) \Leftrightarrow ((e, out) \in \Lambda_{seq}(v) \wedge (e, in) \in \Lambda_{seq}(w))$
This also allows interpreting TGraphs as undirected graphs.

Although *Algolib* works only on TGraphs, the problem definitions in the next chapter are done for regular graphs.

3 Problems, algorithms, and interfaces

This chapter describes the graph problems that are covered in this mid-study thesis. For each definition there is at least one rough idea how the problem can be solved meaning which algorithm can be used for solving it. Some problems can be solved by multiple algorithms.

Each problem definition is placed inside a box with normal corners. A problem definition consists of a name, the input, the output, and the type of output. The problem definitions are in mathematical notation. Solutions are placed inside a box with round corners. A solution consists of a description which problem is solved and which algorithm can be used for the solution. The description only refers to one algorithm. In addition to this, the time- and space complexity is given in $\mathcal{O}$-notation, where n is the number of vertices and m is the number of edges. The space complexity only considers the additional memory required by the algorithm and not the memory that is already used by the graph. In JGraLab, the space complexity for storing a graph is $\mathcal{O}(max\,(m,n))$. When speaking of the space complexity of an algorithm, the space complexity for storing the graph is omitted.

For each algorithm there is a draft in a pseudocode with a syntax similar to Java. The core syntax is identical to Java. Some more complex operations may be simplified by using comments. Unlike in Java, generic type variables can be instantiated with primitive types. This is done for simplifying the interfaces that are used in the algorithms. Some variables used in the algorithms are of interface types. All variables that are not explicitly instantiated are assumed to be instantiated. For explaining the algorithms it is not significant, where the data comes from. The implementation of the algorithms is subject of chapter 5.

In section 3.1, the interfaces used by the algorithms are introduced. This includes the interfaces used for the graph and the interfaces used for in- and output parameters. In section 3.3, the problems and algorithms are introduced.The problems are divided into so-called problem groups and the subsections reflect this.

3.1 The interfaces for graph element access

This section introduces three interfaces allowing the interaction with a graph and its elements. The interfaces are `Graph`, `Vertex`, and `Edge`. The method signatures are identical to those in JGraLab's corresponding interfaces. Only a fraction of JGraLab's methods are needed for

drafting graph algorithms. So the following interfaces only provide a subset of methods that can be found in JGraLab.

In JGraLab all graphs are stored as directed graphs. So this problem and algorithm overview will also assume this.

3.1.1 The interface `Graph`

Listing 3.1 shows the interface `Graph`.

Listing 3.1: The interface `Graph`

```
1 public interface Graph{
2     public Iterable<Vertex> vertices();
3     public Iterable<Edge> edges();
4     public int getVCount();
5     public int getECount();
6 }
```

This interface contains methods for retrieving graph elements and gaining information about the graph's size. Any implementation of `Graph` is assumed to realize a directed graph. However, directed graphs can also be treated as undirected graphs. The differences only concern edges and the details are described in section 3.1.2.

The method `vertices()` returns an `Iterable` for traversing all vertices of the graph. The method `edges()` does the same for edges. The method `getVCount()` returns the number of vertices in the graph. The method `getECount()` returns the number of edges in the graph.

3.1.2 The interface `Edge`

In JGraLab, an `Edge` can either be *normal* or *reversed*. Both, normal and reversed edges, implement the interface `Edge`. In a graph, every edge has a normal and a reversed representation. Incoming incident edges of a `Vertex` are reversed edges, outgoing incident edges of a `Vertex` are normal edges. The alpha- and omega vertices of a normal edge are identical to the alpha- and omega vertices of a reversed edge respectively.

In addition to the alpha- and omega vertices, JGraLab also provides the so-called *this* and *that* vertices. If an `Edge` is a normal edge, *this* is the alpha vertex and *that* is the omega vertex. If an `Edge` is a reversed edge, *this* is the omega vertex and *that* is the alpha vertex. This mechanism is useful when working on an undirected graphs.

Listing 3.2 shows the interface `Edge`. This interface contains methods for retrieving incident vertices.

Listing 3.2: The interface `Edge`

```
1 public interface Edge{
2     public Vertex getAlpha();
3     public Vertex getOmega();
4     public Vertex getThis();
5     public Vertex getThat();
6     public boolean isNormal();
7 }
```

The method `getAlpha()` returns the start vertex of this `Edge`. The method `getOmega()` returns the end vertex of this `Edge`. The method `getThis()` returns the start vertex of this `Edge` if treated as an undirected edge. The method `getThat()` returns the end vertex of this `Edge` if treaded as an undirected edge. The method `isNormal()` returns `true`, if this `Edge` is a normal edge, `false` otherwise.

3.1.3 The interface `Vertex`

Listing 3.3 shows the interface `Vertex`. This interface contains methods for retrieving and

Listing 3.3: The interface `Vertex`

```
1 public interface Vertex{
2     public Iterable<Edge> incidences(EdgeDirection dir);
3     public int getDegree(EdgeDirection dir);
4 }
```

counting incident edges. The method `incidences(...)` returns an `Iterable` for traversing incident edges. The method `getDegree(...)` returns the number of incident edges.

The results can be filtered by the desired edge direction which can be specified by the `enum EdgeDirection`. Its literals are `IN`, `OUT`, and `INOUT`. The edge direction `IN` refers to incoming edges and can be used for following and counting incident edges in reversed direction. The edge direction `OUT` refers to outgoing edges and can be used for following and counting incident edges in normal direction. The edge direction `INOUT` refers to all incident edges and can be used for following and counting incident edges without paying attention to their direction. The pseudocode will omit the prefix `EdgeDirection.` for these literals (e.g. `v.incidences(OUT)`).

3.2 Other interfaces and concepts used by graph algorithms

This section introduces some interfaces and concepts that are used in most graph algorithms described in this chapter. It also shows how they are noted in the pseudocode used to draft these algorithms.

3.2.1 The concept of work lists

Many algorithms require so-called *work lists*. A work list is data structure that contains elements that are about to be processed. Such data structures have to provide at least operations to

- add a new element to the work list,
- retrieve and remove the next element from the work list and
- detect if the work list is empty.

In the pseudocode, the usage of and all operations on explicit work lists are noted using angle brackets (e.g.: « `add v to the queue` »).

A work list does not necessarily be a class of its own. It can also be simulated by data structures that are not explicitly designed to be a work list (e.g. an array) or be implicitly present (e.g. the call stack). If this is the case, it is mentioned in the algorithm description and can also be commented in the pseudocode.

The differences between different kinds of work lists primarily concern the order elements are removed and the complexity of the operations. The latter has an influence on the complexity of the algorithm using the work list. By default a time complexity of $\mathcal{O}(1)$ is assumed for all operations. Some algorithms are defined by the kind of work list that is used. If an algorithm uses a work list, the corresponding description also addresses these issues.

3.2.2 The concept of working points

All algorithms that are introduced in this chapter provide so-called working points. These working points are positions in the algorithm, where additional operations can be added in the context of graph elements and the current algorithm state.

A working point has a name and affects one graph element. In the pseudocode the working points are described using comments (e.g. `/* working point "visitVertex(v)" */`).

Working points can be used for extending an algorithm with new functionality and for solving other problems. For every working point in a graph algorithm, every graph element is at most processed once.

3.2.3 Input parameters, runtime variables, and results

Algorithms are used to solve problems. The problems that are specified in this chapter have *input parameters* and they have at least one result. An algorithm that solves a problem has the same input parameters as the problem that is solved. In the pseudocode, the input parameters are the parameters of the method that represents the algorithm.

An algorithm can solve multiple problems. Every problem has its own specified *result* set. So an algorithm has at least as many results as specified by the problems it solves. In addition to this, many algorithms can also compute results that are not specified by any problem that is solved. These results are called *additional results*. In the pseudocode all additional results are computed as well. All results are defined as global variables.

Most of the algorithms require additional variables for their computation. Such variables are called *runtime variables*. In the pseudocode the runtime variables are also declared as global variables.

3.2.4 The interface `Function`

Some results and input parameters can be interpreted as partial functions. For this purpose, an interface is defined, that represents such functions. The interface `Function` from listing 3.4 is used for this. An element of the functions' domain is called *parameter*. An element of the

Listing 3.4: The interface `Function`

```
1 public interface Function<DOMAIN,RANGE>{
2     public RANGE get(DOMAIN parameter);
3     public boolean isDefined(DOMAIN parameter);
4     public void set(DOMAIN parameter, RANGE value);
5 }
```

functions' range is called *value*.

The method `get(...)` returns the value for the given parameter. The method `isDefined(...)` checks if the function is defined for the given parameter. The method `set(...)` adds the given value for the given parameter to the function.

All these operations are assumed to have a time complexity of $\mathcal{O}(1)$. An implementation is assumed to have a space complexity of $\mathcal{O}(n)$, where n is the number of defined parameter value pairs.

Binary functions

Some functions that occur in this chapter are binary functions. For such functions an analogous interface can be specified. Listing 3.5 shows the interface `BinaryFunction`.

Listing 3.5: The interface `BinaryFunction`

```
1 public interface BinaryFunction<DOMAIN1,DOMAIN2,RANGE>{
2     public RANGE get(DOMAIN1 parameter1, DOMAIN2 parameter2);
3     public boolean isDefined(DOMAIN1 parameter1, DOMAIN2 parameter2);
4     public void set(DOMAIN parameter1, DOMAIN2, parameter2, RANGE value);
5 }
```

The methods work in analogy to the interface `Function` and are assumed to have the same time and space complexity. For binary functions whose domain is $V \times V$ and whose range is E, the corresponding interface is `BinaryFunction<Vertex,Vertex,Edge>`.

3.3 Problems and algorithms

This section introduces all problems and algorithms that are covered by this mid-study thesis and that are implemented in *Algolib*. Most problems are defined for directed and undirected graphs. If this is the case, the algorithm drafts only describe the solution for directed graphs. An additional description will contain the information how to transform the draft for working on undirected graphs.

The most basic problems on graphs are traversal problems. So the first subsection is about these traversal problems and how to solve them.

3.3.1 Problem group: Traversal

Definition: Traversal
Given a graph $G = (V, E, \varphi)$,
a *traversal* of G is an activity that visits each vertex $v \in V$ and each edge $e \in E$ exactly once.

3.3.1.1 Traversal from r

This section describes a traversal from a given vertex $r \in V$. This traversal only considers the *reachable subgraph* from r.

Definition: Reachable subgraph
Given a graph $G = (V, E, \varphi)$ and a vertex $r \in V$,
let $V_r := \{v \in V \mid r \rightarrow^* v\}$ and $E_r := \{e \in E \mid \alpha(e) \in V_r \wedge \omega(e) \in V_r\}$.
The subgraph $G_r := (V_r, E_r, \varphi_{|_{E_r}})$ is called the *reachable subgraph* from r.

Problem: Traversal from r
Input: a graph $G = (V, E, \varphi)$ and a vertex $r \in V$
Output: a traversal of the reachable subgraph G_r
Output type: a permutation of V_r and a permutation of E_r

This problem can be solved using a *search algorithm* described below.

A search algorithm starts at a given vertex r and traverses the reachable subgraph from r. During a search, a *reachability tree* is created.

Definition: Reachability tree
Given a graph $G = (V, E, \varphi)$, a vertex $r \in V$ and the reachable subgraph $G_r = (V_r, E_r, \varphi_{|_{E_r}})$, a *reachability tree* of r is a rooted tree $T_r := (V_r, E_t, \varphi_{|_{E_t}})$ with r being the root of T_r and $E_t \subseteq E_r$.
The reachability tree, that is created during a search, is also called *search tree*.
The edges in this tree are called *tree edge*s.
All other edges in the reachable subgraph are called *fronds*.
A path from the root of the search tree to a leaf is called *search path*.

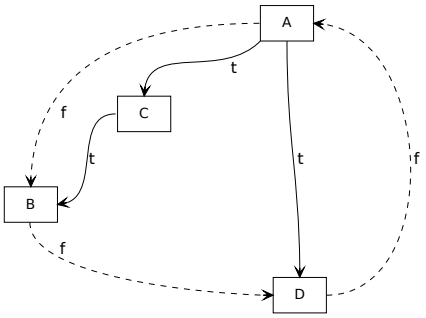


Figure 3.1: Sample graph with tree edges and fronds

Figure 3.1 shows a small sample graph with tree edges and fronds. The tree's root is vertex A. The tree edges are indicated by $\overset{t}{\rightarrow}$. The fronds are indicated by $\overset{f}{\dashrightarrow}$.

3.3.1.2 Search algorithms

A search algorithm starts at a given vertex r and traverses its incident edges. It uses a work list for storing the vertices whose incident edges have not been traversed yet. Whenever an incident edge is encountered, the vertex on the other end (*that* for directed graphs this corresponds to *omega*) is added to the work list.

It is also possible to implement a search algorithm that uses a work list for storing edges instead. However, in the following a work list for vertices is used.

3.3.1.3 The results of a search algorithm

Besides the specified results from *traversal from r*, a search can compute several additional results. All of them can be interpreted as functions and are defined in the following.

Definition: Functions that can be computed during a directed search

- The function *vertexOrder* : $[1..n] \twoheadrightarrow V$ defines the order, in which the vertices are traversed.
  vertexOrder(i) := the ith vertex that has been traversed.
  vertexOrder can also be interpreted as a sequence of vertices.
- The function *eorder* : $[1..m] \twoheadrightarrow E$ defines the order, in which the edges are traversed.
  edgeOrder(i) := the ith edge that has been traversed.
  edgeOrder can also be interpreted as a sequence of edges.
- The function *number* : $V \twoheadrightarrow [1..n]$ defines the position of a vertex in the *vertexOrder* sequence.
- The function *enumber* : $E \twoheadrightarrow [1..m]$ defines the position of an edge in the *edgeOrder* sequence.
- The function *level* : $V \twoheadrightarrow [1..n]$ defines the level of a vertex v in the search tree T.
  level$(v) := \overrightarrow{d_T}(r, v)$, where r is the root of T.
- The function *parent* : $V \setminus \twoheadrightarrow E$ defines the structure of the search tree T with root r. For every non-root vertex, the incoming edge in T from the parent node is returned.

3.3.1.4 The generic search algorithm

A search algorithm needs to know which vertices have already been traversed (or visited). For this purpose the function *number* can be used. If the *number* value of a vertex has been computed (is defined), it has been traversed.

Listing 3.6 shows the generic search algorithm with computation of all functions mentioned above. The algorithm is called generic, because the actual work list is not specified (line 12). The runtime variables `num` and `eNum` (lines 10 and 11) are required for the computation of `vertexOrder`, `number`, `enumber`, and `edgeOrder`. In line 18 a working point is specified that visits the root of the search tree. The work list is initialized with the root (line 20).

The main loop (lines 21 - 40) runs until the work list is empty. The first step is taking the next vertex from the work list (line 22). Its outgoing edges are traversed in the inner loop (lines 23 - 39).

In line 26 a working point is specified that visits the current incident edge. In line 27 the *that* vertex of the current edge is taken into consideration. It is visited if it has not been visited yet (lines 28 - 35). Line 33 (and line 19) define a working point that visits a vertex before it is added to the work list. Line 35 adds the current vertex to the work list right before the inner loop's current iteration ends.

Listing 3.6: Generic search algorithm from *r*

```
1  // results
2  Function<int,Vertex> vertexOrder;
3  Function<int,Edge> edgeOrder;
4  Function<Vertex,int> number;
5  Function<Edge,int> enumber;
6  Function<Vertex,int> level;
7  Function<Vertex,Edge> parent;
8
9  // runtime variables
10 int num = 0;
11 int eNum = 0;
12 /* << a work list for storing vertices >> */
13
14 public void search(Graph g, Vertex r) {
15     number.set(r,++num);
16     vertexOrder.set(num,r);
17     level.set(r,0);
18     // working point: visitRoot(r)
19     // working point: visitVertex(r)
20     << add r to work list >>
21     while(<< work list is not empty >>){
22         << get next vertex v from work list >>
23         for(Edge e: v.incidences(OUT)){
24             enumber.set(e,++eNum);
25             edgeOrder.set(eNum,e);
26             // working point: visitEdge(e)
27             Vertex w = e.getThat();
28             if(!number.isDefined(w)){
29                 number.set(w,++num);
30                 vertexOrder.set(num,w);
31                 level.set(w,(level.get(v) + 1));
32                 parent.set(w,e);
33                 // working point: visitVertex(w)
34                 // working point: visitTreeEdge(e)
35                 << add w to work list >>
36             } else {
37                 // working point: visitFrond(e)
38             }
39         }
40     }
41 }
```

The generic search also defines working points for distinguishing between tree edges and fronds. If the next vertex (`w`) has not been visited yet (line 28), the current edge (`e`) it is a tree edge (line 34). Otherwise it is a frond (line 37).

Solution: Traversal from r
Perform a *generic directed search algorithm from r* as shown in listing 3.6.
Space complexity: $\mathcal{O}(max(m,n))$
Time complexity: $\mathcal{O}(max(m,n))$

Undirected graphs

For undirected graphs, the algorithm would have to operate on all incident edges instead of only the outgoing ones. This can be achieved in line 23 by replacing the parameter `OUT` by `INOUT`. However, in doing this, every reachable edge is considered twice. For avoiding visiting edges twice, the algorithm has to keep track of the edges that have already been visited.

3.3.1.5 Specific search algorithms

In order to create an actual search algorithm, the work list has to be specified. The kind of work list directly influences the order the vertices and edges are traversed.

Using a queue creates a *breadth first search* (BFS). In this search, all successor vertices are put into a queue. This ensures a horizontal traversal of the graph, meaning each level of the search tree is traversed completely before the next level is considered.

Using a stack creates a *depth first search* (DFS). In this search, all successor vertices are pushed on a stack. This ensures a vertical traversal of the graph.

In the following, both of them are described in detail.

3.3.1.6 Breadth first search

A breadth first search (BFS) can be realized by using a queue that serves as work list. Here the function *vertexOrder* is used as queue. This is possible, because the function value are computed in successor order, controlled by the runtime variable `num`. The removal of elements is done using a second runtime variable `firstV` that points to the next vertex. If the next vertex is requested, this variable is incremented and the element is virtually removed from the queue.

Listing 3.7 shows the BFS, which is simply a modification of the generic search algorithm. The only difference to the generic search from listing 3.6 is the usage of `vertexOrder` as a queue buffer. In line 17 and 31, the current vertex is implicitly added to the queue. The counter `firstV` is used to decide if the queue is empty (line 21). Whenever the queue contains more elements, the counter `num` is greater than `firstV`. If they are equal, they refer to an undefined function value, which means the queue is empty.

Figure 3.2 shows the same sample graph as Figure 3.1, except that the search tree is different. The search tree, defined by the tree edges, is a possible search tree when performing a BFS

Listing 3.7: breadth first search from *r*

```
// results
Function<int,Vertex> vertexOrder;
Function<int,Edge> edgeOrder;
Function<Vertex,int> number;
Function<Edge,int> enumber;
Function<Vertex,int> level;
Function<Vertex,Edge> parent;

// runtime variables
int num = 0;
int eNum = 0;
int firstV = 1;

public void search(Graph g, Vertex r) {
    number.set(r,++num);
    // implicit add to queue
    vertexOrder.set(num,r);
    level.set(r,0);
    // working point: visitRoot(r)
    // working point: visitVertex(r)
    while(firstV < num){ // queue not empty
        // implicit get from queue
        Vertex v = vertexOrder.get(firstV++);
        for(Edge e: v.incidences(OUT)){
            enumber.set(e,++eNum);
            edgeOrder.set(eNum,e);
            // working point: visitEdge(e)
            Vertex w = e.getThat();
            if(!number.isDefined(w)){
                number.set(w,++num);
                // implicit add to queue
                vertexOrder.set(num,w);
                level.set(w,(level.get(v) + 1));
                parent.set(w,e);
                // working point: visitVertex(w)
                // working point: visitTreeEdge(e)
            } else {
                // working point: visitFrond(e)
            }
        }
    }
}
```

on this sample graph. It also shows the values of *number*, *parent*, and *level* for the search that created this specific search tree as vertex attributes.

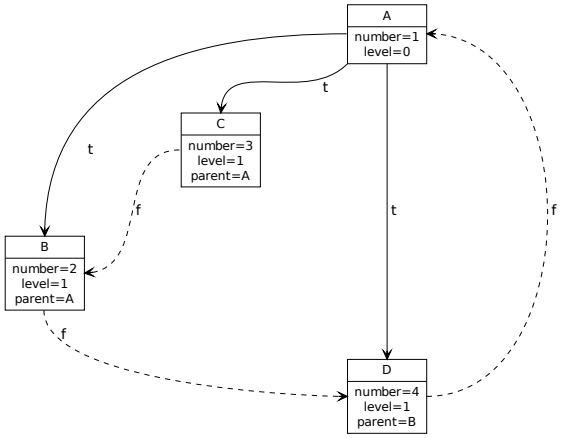


Figure 3.2: Sample graph with tree edges created by a BFS and fronds

3.3.1.7 Depth first search

The depth first search (DFS) is capable of distinguishing between three different kinds of fronds. The easiest way to implement it is using recursion. However, this implementation does not contain an explicit work list as seen in BFS. The call stack serves as an implicit work list.

The recursive DFS allows two working points for vertices and edges instead of one. The first working point for either of them is before the recursive call and the second one is after the recursive call. This also allows the definition of two additional functions that can be computed by a recursive DFS. For defining these functions, the term *completely traversed* has to be introduced.

Definition: Completely traversed during a DFS
During the execution of a DFS,

- a vertex is *completely traversed*, if it has been visited and all its relevant incident edges have been completely traversed as well,
- a tree edge is *completely traversed* if its *that* vertex has been completely traversed, and
- a frond is *completely traversed* if it has been visited.

Definition: Functions that can be computed during a recursive DFS

- The function $rorder : [1..n] \twoheadrightarrow V$ defines the order in which the vertices are completely traversed.
  $rorder(i) :=$ the ith vertex that has been completely traversed.
  rorder can also be interpreted as a sequence of vertices.
- The function $rnumber : V \twoheadrightarrow [1..n]$ defines the position of a vertex in the *rorder* sequence.

As mentioned above, the recursive DFS allows a distinction between three different kinds of fronds, *forward arc*s, *backward arc*s, and *cross link*s.

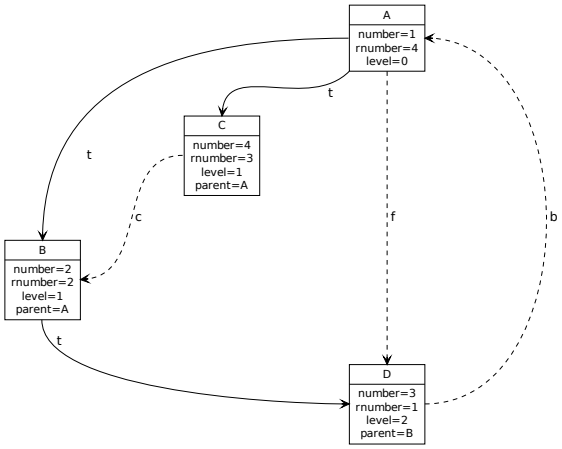


Figure 3.3: Sample graph with tree edges created by a DFS and different kinds of fronds

Figure 3.3 shows the same sample graph as figure 3.1 but distinguishes between the different kinds of fronds. Tree edges are indicated by $\xrightarrow{t}$. Forward arcs are indicated by $\overset{f}{\dashrightarrow}$. Backward arcs are indicated by $\overset{b}{\dashrightarrow}$. Cross links are indicated by $\overset{c}{\dashrightarrow}$. The search tree, defined by the tree edges, is a possible search tree if performing a DFS on this sample graph. The figure also shows the values of *number*, *rnumber*, *parent*, and *level* for the search that created this specific search tree as vertex attributes.

The functions *number* and *rnumber* can be used for defining the fronds. Table 3.1 considers a frond $v \to w$ and defines which kind of frond it is, using these functions.

	$number(v) < number(w)$	$number(v) \geq number(w)$
$rnumber(v) \leq rnumber(w)$	$\times$	backward arc
$rnumber(v) > rnumber(w)$	forward arc	cross link

Table 3.1: Fronds overview

The entry $\times$ indicates that this case cannot occur. Assuming an edge $e = v \to w$. If $number(w)$ has not been computed yet, e is a tree edge. If it has already been computed, it is a forward arc. In both cases vertex w is completely traversed before vertex v, so $rnumber(v)$ cannot be greater than $rnumber(w)$.

For every kind of frond, a different working point can be specified.

Listing 3.8 shows the directed recursive version of DFS with computation of all functions from the directed generic search and the newly defined functions *rorder* and *rnumber*.

The actual search is started in line 18 and defined by the method `dfs`. The functions `vertexOrder`, `edgeOrder`, `number`, `enumber`, `level` and `parent` are computed in analogy to the BFS. The DFS defines similar working points as the BFS. In addition to those, lines 35 and 49 define working points for visiting edges and vertices respectively when being com-

Listing 3.8: Recursive depth first search algorithm from *r*

```
// results
Function<int,Vertex> vertexOrder;
Function<int,Edge> edgeOrder;
Function<Vertex,int> number;
Function<Edge,int> enumber;
Function<Vertex,int> level;
Function<Vertex,Edge> parent;
Function<int,Vertex> rorder;
Function<Vertex,int> rnumber;

// runtime variables
int num = 0;
int rnum = 0;

public void search(Graph g, Vertex r) {
    level.set(r,0);
    // working point: visitRoot(r)
    dfs(r);
}

private void dfs(Vertex v){
    number.set(v,++num);
    vertexOrder.set(num,v);
    // working point: visitVertex(v)
    for(Edge e: v.incidences(OUT)){
        enumber.set(e,++eNum);
        edgeOrder.set(eNum,e);
        // working point: visitEdge(e)
        Vertex w = e.getThat();
        if(!number.isDefined(w)){
            level.set(w,(level.get(v) + 1));
            parent.set(w,e);
            // working point: visitTreeEdge(e)
            dfs(w); // recursive call
            // working point: leaveTreeEdge(e)
        } else {
            // working point: visitFrond(e)
            if(!rnumber.isDefined(w)){
                // working point: visitBackwardArc(e)
            } else if(number.get(w) > number.get(v)){
                // working point: visitForwardArc(e)
            } else{
                // working point: visitCrossLink(e)
            }
        }
    }
    rnumber.set(v,++rnum);
    rorder.set(rnum,v);
    // working point: leaveVertex(v)
}
```

pletely traversed. Lines 38 - 44 distinguish between the different kinds of fronds using the functions `number` and `rnumber` and define working points for each of them.

3.3.1.8 Complete traversal

The traversals described so far in this section only traverse the reachable subgraph from r. A more generic traversal problem can be defined where the whole graph is traversed.

Problem: Directed complete traversal
Input: a graph $G = (V, E, \varphi)$
Output: a traversal of G.
Output type: a permutation of V and a permutation of E

A trivial solution for the complete traversal is the traversal without concerning the structure of the graph. For this purpose, the methods `vertices()` and `edges()` from the interface `Graph` are used.

In this approach no search tree is created and no distinction between fronds and tree edges can be made. This approach can compute the functions *vertexOrder*, *edgeOrder*, and *number*. For some problems this is a sufficient traversal. Listing 3.9 shows this approach.

Listing 3.9: Complete traversal ignoring the structure of the graph

```
1  // results
2  Function<int,Vertex> vertexOrder;
3  Function<int,Edge> edgeOrder;
4  Function<Vertex,int> number;
5  Function<Edge,int> enumber;
6
7  // runtime variables
8  int num = 0;
9  int eNum = 0;
10
11 public void completeSearch(Graph g){
12     for(Vertex v: g.vertices()){
13         number.set(v,++num);
14         vertexOrder.set(num,v);
15         // working point: visitVertex(v)
16     }
17     for(Edge e: g.edges()){
18         edgeOrder.set(e,eNum++);
19         enumber.set(eNum,e);
20         // working point: visitEdge(e)
21     }
22 }
```

Solution: Complete traversal

Traverse all vertices and edges from the graph, without considering the structure of the graph.

Space complexity: $\mathcal{O}(max(m, n))$

Time complexity: $\mathcal{O}(max(m, n))$

If the structure of the graph is important, the search algorithms described above, can be extended to a *complete search algorithm*. The actual search algorithm is called multiple times but the runtime variables remain. This is realized using an outer loop. Listing 3.10 shows this extension.

Listing 3.10: Extension for the complete search algorithm

```
1 public void completeSearch(Graph g){
2     for(Vertex v:g.vertices()){
3         if(!number.isDefined(v)){
4             search(g, v);
5         }
6     }
7 }
```

This code above can be added to any search algorithm shown in this section. The loop ensures that every vertex is considered. If `number` has already been set for a given vertex, it is ignored. If not, a new search is started from this vertex.

Solution: Complete traversal

Extend any search algorithm from r by calling it from within a loop iterating over V.

Space complexity: $\mathcal{O}(max(m, n))$

Time complexity: $\mathcal{O}(max(m, n))$

3.3.2 Problem group: Acyclicity

The problem group *Acyclicity* covers the problems acyclicity and topological order. The problems belong to the same group, because the existence of a topological order and acyclicity are equivalent.

Problem: Acyclicity

Input: a directed graph $G = (V, E, \varphi)$

Output: the decision if G is acyclic

Output type: a boolean value

Definition: Topological order
Given a directed acyclic graph $G = (V, E, \varphi)$,
a bijective function $tnumber : V \to [1..n]$ with $\forall\, v \to w \in E : tnumber(v) < tnumber(w)$ is called *topological numbering*.
A topological order (*torder*) is the inverse function of *tnumber*.

With this, the following problem can be defined.

Problem: Topological order
Input: an acyclic directed graph $G = (V, E, \varphi)$
Output: a topological order of V
Output type: a permutation of V

Please note that these two problems are only defined for directed graphs.

3.3.2.1 Solution with the DFS algorithm

These two problems can be solved using a complete DFS. If a backward arc is detected, the graph contains at least one cycle and the vertices do not have a topological order. If the complete search terminates without a detected backward arc, a topological order can be derived from the function *rorder*.

For achieving this, the DFS has to be extended with the code from listing 3.11. The working

Listing 3.11: Code for extending the DFS to detect cycles

```
1  // add to results
2  boolean acyclic = true;
3
4  // add to working point: visitBackwardArc(e)
5  acyclic = false;
6  // terminate
7
8  public void acyclic(Graph g){
9      completeSearch(g);
10 }
```

point `visitBackwardArc` has to be extended with the code from line 5. If a backward arc is encountered, the graph is cyclic and the algorithm may terminate.

If the graph is acyclic, the function *torder*, that represents the topological order, can be defined using the function *rorder*:

$torder(i) := rorder(n - i + 1)$.

Solution: Acyclicity and topological order
Perform a complete DFS and compute *rorder*.
Space complexity: $\mathcal{O}(max(m,n))$
Time complexity: $\mathcal{O}(max(m,n))$

3.3.2.2 The Kahn-Knuth algorithm

With the algorithm above, a topological order has to be computed offline. If online actions on all vertices in a topological order are desired, the *Kahn-Knuth* algorithm with working points can be used.

Listing 3.12 shows the Kahn-Knuth algorithm. It uses a temporary function $inDegree : V \to \mathbb{N}$ (line 9). This function returns the temporary in-degree for all vertices. Initially the temporary in-degree is set to the real in-degree (line 13) using the method `getDegree(...)` from the interface `Vertex`.

During the algorithm, after a vertex has been traversed, the vertex is logically removed from the graph and the temporary in-degree of all successor vertices is reduced by 1 (line 25).

In analogy to the BFS, the Kahn-Knuth algorithm computes *torder* as a function and uses this function as a queue. This queue holds vertices with a temporary in-degree of 0. It is initialized in lines 14-17. The variable `tnum` refers to the last element that has been added to the queue and works in analogy to the variable `num` from BFS. As in BFS, the variable `firstV` refers to the head of the queue. In line 22 a working point is defined for visiting vertices in topological order.

The inner loop reduces the temporary in-degree of all successor vertices by 1. In doing this, the current vertex `v` is logically removed from the graph. All successor vertices with a temporary in-degree of 0 are added to the queue (line 26-29).

The variable `acyclic` (line 2) is set to `false` after the execution of the main loop, if the graph is cyclic (lines 32-33). `tnum` is also an implicit counter for the number of vertices that have been added to `torder`. It can be used for checking if all vertices have been added to the queue. Only if all vertices of the graph have been added to the queue, the graph is acyclic. If this is the case, the function `torder` represents a topological order of V.

Solution: Acyclicity and topological order
Perform the *Kahn-Knuth* algorithm.
Space complexity: $\mathcal{O}(n)$
Time complexity: $\mathcal{O}(max(m,n))$

The disadvantage of Kahn-Knuth is the additional time that is needed for the computation of the temporary in-degree. It is possible to avoid this without losing the ability to perform online actions by performing a reversed DFS.

Listing 3.12: The Kahn-Knuth algorithm

```
1  // results
2  boolean acyclic = true;
3  Function<int, Vertex> torder;
4  Function<Vertex,int> tnumber;
5
6  // runtime variables
7  int tnum = 0;
8  int firstV = 1;
9  Function<Vertex,int> inDegree;
10
11 public void kahnKnuth(Graph g){
12     for(Vertex v: g.vertices()){
13         inDegree.insert(v,v.getDegree(IN));
14         if(inDegree.get(v) == 0){
15             tnumber.set(v, ++tnum);
16             torder.set(tnum, v);
17         }
18     }
19
20     while(firstV < tnum) {
21         Vertex v = torder.get(firstV++);
22         // working point: visitVertexInTopologicalOrder(v)
23         for(Edge e: v.incidences(OUT)){
24             Vertex w = e.getThat();
25             inDegree.set(w, inDegree.get(w) - 1);
26             if(inDegree.get(w) == 0){
27                 tnumber.set(w, ++tnum);
28                 torder.set(tnum, w);
29             }
30         }
31     }
32     if(tnum < g.getVCount()){
33         acyclic = false;
34     }
35 }
```

3.3.2.3 Solution with the reversed DFS

Search algorithms can be modified easily for searching the graph in reversed orientation. In the generic search algorithm from listing 3.6 on page 37 this can be done in line 22 by changing the parameter of `v.incidences(...)` from `OUT` to `IN`. This makes the algorithm follow incoming incident edges instead of outgoing ones. For BFS and DFS this can be done analogously.

For solving the problems *topological order* and *acyclicity* a reversed DFS can be used. The only difference is the definition of the function *torder*. Here it is $torder(i) := rorder(i)$.

The function *rorder* is computed right before the working point `leaveVertex(v)` (see listing 3.8 on page 42 in lines 47-49). By adding the working point

`visitVertexInTopologicalOrder(v)` to the working point `leaveVertex(v)`, this algorithm gains the ability to perform online actions in topological order.

3.3.3 Problem group: shortest paths

From a vertex $v \in V$ every reachable vertex has a *distance* from v. This is defined in the following definition.

Definition: Distance
Given a graph $G = (V, E, \varphi)$ and two vertices $v \in V, w \in V$,
the *distance* $\overrightarrow{d}(v, w)$ is the length of a shortest path from v to w.
If $w \notin v \rightarrow^*$, then $\overrightarrow{d}(v, w) := \infty$.

Now a problem can be specified for the computation of the distance from a vertex r to all vertices.

Problem: Distance from r
Input: a graph $G = (V, E, \varphi)$ and a vertex $r \in V$
Output: the distance $\overrightarrow{d}(r, v)$ for every vertex $v \in V$
Output type: a function $d_r : V \rightarrow \mathbb{N} \cup \{\infty\}$

3.3.3.1 Solution with the BFS

The distance from r can be computed by performing a directed BFS with the computation of *level*. The level of a vertex in the search tree created by a BFS is also the distance from the root r of this tree, i.e., for reachable vertices $v \in V_r : d_r(v) = level(v)$. For unreachable vertices $v \in V \setminus V_r : d_r := \infty$

Solution: Distance from r
Perform a BFS from r with the computation of *level*. The function d_r is identical to the function *level*.
Space complexity: $\mathcal{O}(max(m, n))$
Time complexity: $\mathcal{O}(max(m, n))$

Problem: Shortest paths from r
Input: a graph $G = (V, E, \varphi)$ and a vertex $r \in V$
Output: for every vertex v with $r \rightarrow^* v$ a simple path from r to v with minimal length
Output type: a function $sp_r : V \rightarrow \mathcal{PATH}_G$

For all reachable vertices v from r a set of simple paths can be computed by performing a directed BFS with the computation of *parent*. The actual paths from r to v can be computed offline.

Solution: Shortest paths from r

Perform a BFS from r with the computation of *parent*. With *parent*, the paths can be computed offline.

Space complexity: $\mathcal{O}(max(m,n))$

Time complexity: $\mathcal{O}(max(m,n))$

3.3.4 Problem group: strong components

Definition: Strongly connected

Given a directed graph $G = (V, E, \varphi)$,
Two vertices $v \in V$ and $w \in V$ are *strongly connected*, iff
$v \rightarrow^* w \wedge w \rightarrow^* v$.
Notation: $v \overset{*}{\leftrightarrow} w$

$\overset{*}{\leftrightarrow}$ is an equivalence relation and the partition $V/_{\overset{*}{\leftrightarrow}}$ is a partition of V into *strong components*. The edges connecting different strong components of a graph G are called *reduced edges*.

Definition: Reduced edge

Given a directed graph $G = (V, E, \varphi)$ and an edge $e \in E$.
e is called *reduced edge* iff $!(\alpha(e) \overset{*}{\leftrightarrow} \omega(e))$

Problem: Strong components

Input: a directed graph $G = (V, E, \varphi)$

Output: the strong components of G, i.e. $V/_{\overset{*}{\leftrightarrow}}$

Output type: a partition of V

Please note that this problem is only defined for directed graphs.

3.3.4.1 Solution with the DFS and *lowlink*

This problem can be solved with the DFS. In order to solve the problem with it, the function *lowlink* for DFS trees is needed.

Definition: Lowlink

$lowlink : V \twoheadrightarrow [1..n]$ with $lowlink(v) := min(\{number(v)\} \cup \{number(w) \mid v \rightarrow^* \dashrightarrow w \wedge v \overset{*}{\leftrightarrow} w\})$.
(*The symbol* $\dashrightarrow$ *means* frond.)

The lowlink value of a vertex is equal to the number value of a vertex of the same strong component. If $number(v) = lowlink(v)$, then v is the root of a strong component, also called *strong root*. This function and the solution of this problem using DFS goes back to Tarjan [Tar72].

For solving the problem *strong components* with the DFS, it has been extended at some of its working points. Listing 3.13 shows how this is done.

Listing 3.13: Code for extending DFS for computing lowlink and strong components

```
// add to results
Function<Vertex,Vertex> rep;
Function<Vertex,int> lowlink;

// add to runtime variables
/* << a stack for vertices >> */

// add to working point: visitVertex(v)
<< push v on the stack >>
lowlink.set(v,number.get(v));

// add to working point: leaveTreeEdge(e@(v->w))
lowlink.set(v,min(lowlink.get(v),lowlink.get(w)));
detectReducedEdge(e);

// add to working point: visitForwardArc(e)
detectReducedEdge(e);

// add to working point: visitBackwardArc(e@(v->w))
lowlink.set(v,min(lowlink.get(v),number.get(w)));

// add to working point: visitCrosslink(e@(v->w))
if(<< w is on the stack >>){
    lowlink.set(v,min(lowlink.get(v),number.get(w)));
}
detectReducedEdge(e);

// add to working point: leaveVertex(v){
if(lowlink.get(v) == number.get(v)){
    do {
        << pop vertex x from the stack >>
        rep.set(x,v);
    } while(x != v);
    // working point: visitRepresentativeVertex(v)
}

private void detectReducedEdge(e@(v->w)){
    if (rep.isDefined(w)) {
        // working point: visitReducedEdge(e)
    }
}

public void strongComponents(Graph g){
    completeSearch(g);
}
```

For simplicity reasons, an edge `e` is written `e@(v->w)`[1] for accessing the alpha and omega vertex directly. The working points `leaveTreeEdge` (line 12), `visitBackwardArc` (line 15) and `visitCrossLink` (line 18) realize the computation of the function *lowlink*.

The detection of strong components is done using the working points `visitVertex` (line 8) and `leaveVertex` (line 28). The vertices that are visited are pushed on a stack (line 9). When vertices are left, the strong roots are detected using the function `lowlink` (line 29). If a strong root is encountered, vertices are popped from the stack until the current strong root is reached (lines 30-33). All those vertices belong to the same strong component. The current strong root is assigned to them as representative vertex (line 32). For strong roots, which are used as representative vertices, a new working point is added in line 34.

The detection of reduced edges is also possible. This is done by the method `detectReducedEdge(...)` (line 37). It defines a working point for reduced edges (line 39). If the strong component of the omega vertex is already known, it cannot belong to the strong component of the alpha vertex and the current edge is a reduced edge. If the strong component of the omega vertex is unknown (not defined yet), it belongs to the same strong component as the alpha vertex and the current edge is not a reduced edge. For tree edges this is only true when they are left, so this method is called at the working point `leaveTreeEdge(e)` (line 14). For forward and backward arcs this is true when they are encountered during the DFS, so this method is called at their working points (lines 17 and 26). Backward arcs can never be reduced edges, they only occur inside of strong components.

Solution: Strong components

Perform a directed complete depth first search with the computation of *lowlink* and an additional vertex stack.

Space complexity: $\mathcal{O}(n)$

Time complexity: $\mathcal{O}(max(m, n))$

3.3.5 Problem group: reachability

Definition: Reachable

Given a graph $G = (V, E, \varphi)$, and two vertices $v, w \in V$,
w is *reachable* from v, iff $v \rightarrow^* w$.

Problem: Reachable

Input: a graph $G = (V, E, \varphi)$ and two vertices $v, w \in V$.

Output: the decision if w is reachable from v

Output type: a boolean value

[1] similar to Haskell

3.3.5.1 Solution with a generic search algorithm

The above problem can be solved by any search algorithm. Listing 3.14 shows how a search algorithm can be extended for solving this problem.

Listing 3.14: Code for extending a search algorithm for solving the problem *reachable*

```
1  // add to results
2  boolean reachable = false;
3
4  // add to runtime variables
5  Vertex target;
6
7  // add to working point: visitVertex(v)
8  if(v == target) {
9      reachable = true;
10     // terminate
11 }
12
13 public void reachable(Vertex r, Vertex t){
14     target = t,
15     search(r);
16 }
```

In line 8 it is checked if the target vertex has been reached. In this case the algorithm may terminate (line 10). If the target vertex has not been reached, the result `reachable` remains `false` (line 2).

3.3.5.2 The Warshall algorithm

It is also possible to compute the whole reachability relation $\rightarrow^*$.

Problem: Reachability
Input: a graph $G = (V, E, \varphi)$
Output: The reachability relation $\rightarrow^*$
Output type: a relation on V

Such a relation can be interpreted as binary function. that assigns to each vertex tuple a boolean value if this tuple is part of the relation or not. If a vertex v is reachable from another vertex w, the corresponding function value for (v, w) is set to *true*.

It is also possible to compute simple paths leading from any vertex v to all vertices that are reachable from v.

Problem: Simple paths
Input: a graph $G = (V, E, \varphi)$
Output: for each vertex tuple $(v, w) \in V \times V$ a simple path from v to w
Output type: a function $(V \times V) \rightarrow \mathcal{PATH}_G$

It is sufficient if the algorithm only computes the function *successor*: $(V \times V) \rightarrow E$ with $successor((v, w)) :=$ the successor vertex on a simple path from v to w iff $v \rightarrow^* w$. If the target vertex is not reachable or the source- and target vertex are the same, the function value is undefined. Actual simple paths can be computed offline using the function *successor*.

An algorithm for computing the reachability relation and the corresponding simple paths is the *Warshall* algorithm. This algorithm can be found in listing 3.15.

Listing 3.15: The Warshall algorithm

```
1  // results
2  BinaryFunction<Vertex,Vertex,boolean> reachable;
3  BinaryFunction<Vertex,Vertex,Edge> successor;
4
5  public void warshall(Graph g){
6      for (Vertex v: g.vertices()) {
7          for (Vertex w: g.vertices()) {
8              reachable.set(v,w,false);
9          }
10         reachable.set(v,v,true);
11     }
12
13     for (Edge e@(v->w): g.getAllEdges()) {
14         if (v != w) {
15             reachable.set(v,w,true);
16             successor.set(v,w,e);
17         }
18     }
19
20     for (Vertex v: g.vertices()) {
21         for (Vertex u: g.vertices()) {
22               for (Vertex w: g.vertices()) {
23                 if (reachable.get(u,v) && reachable.get(v,w) &&
                       !reachable.get(u,w)){
24                     reachable.set(u,w,true);
25                     successor.set(u,w,successor.get(u,v));
26                 }
27             }
28         }
29     }
30 }
```

In line 8, all vertex tuples are marked as unreachable. In lines 10 all vertices are marked reachable to themselves. In lines 13-18 the information from all edges in the graph are stored in the

functions. For each edge, its omega vertex (`w`) is marked reachable from its alpha vertex (`v`) and the current edge is set as successive edge in the function `successor`.

The main loop (lines 20-29) iterates over all possible vertex triples (u, v, w) and creates the transitive closure. In line 23 it is checked, if `v` is reachable from `u` and if `w` is reachable from `v`. Only if the reachability from `u` to `w` has not been detected before, this gap is closed (line 24) and the successive edge for reaching w from u is set to the edge for reaching `v` from `u` (line 25).

Solution: Reachability and simple paths
Perform the Warshall algorithm
Space complexity: $\mathcal{O}(n^2)$
Time complexity: $\mathcal{O}(n^3)$

Because of the quadratic space complexity and the cubic time complexity, the Warshall algorithm is not applicable for graphs with a high vertex count.

3.3.6 Problem group: weighted shortest paths

In section 3.3.3, the problems *distance from* r and *shortest paths from* r are defined. There, all edges are assumed to have a weight of 1. This section defines analogous problems for edges that can have arbitrary weights, including negative ones.

First the weighted length of a path has to be defined, followed by the weighted distance of two vertices.

Definition: Weighted length
Given a graph $G = (V, E, \varphi)$, an edge weight function $g : E \to \mathbb{R}$ and a directed path C,
The *weighted length* $l_g(C) := \sum_{i=1}^{|C|} g(e_i)$, where $e_i \in E(C)$.

Definition: Weighted distance
Given a graph $G = (V, E, \varphi)$, an edge weight function $g : E \to \mathbb{R}$ and two vertices $v \in V$, $w \in V$,
the *distance* $\overrightarrow{d_g}(v, w)$ is the weighted length of a shortest path from v to w.
If $w \notin v \to^*$, then $\overrightarrow{d_g}(v, w) := \infty$.

Now several problems can be defined using the definitions from above.

Problem: Negative cycles
Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$
Output: the decision if G has a cycle with negative length with respect to g
Output type: a boolean value

Problem: Weighted distances

Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$; G may not have negative cycles with respect to g

Output: for every pair of vertices $(v, w) \in V \times V$ the distance $\overrightarrow{d_g}(v, w)$

Output type: a function $\overrightarrow{d_g} : V \times V \to \mathbb{R} \cup \{\infty\}$

Problem: Weighted shortest paths

Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$; G may not have negative cycles with respect to g

Output: for every pair of vertices $(v, w) \in V \times V$ a simple path from v to w with minimal weighted length.

Output type: a function $sp_g : (V \times V) \to \mathcal{PATH}_G$

For solving both of these problems, the Floyd algorithm can be used.

3.3.6.1 The Floyd algorithm

The Floyd algorithm is similar to the Warshall algorithm from listing 3.15. The binary function *reachable* is replaced by a binary function *distance* that stores the distances between all vertex pairs. The binary function *successor* is the same as in the Warshall algorithm, but here it represents all shortest weighted paths from all vertices. The *Floyd algorithm* can be found in listing 3.16

The initialization is similar to the Warshall algorithm. The distance is initially set to ∞ for all vertex pairs (line 9) except the pairs of same vertices. They are set to `0` (line 11). For each edge, the distance from its alpha vertex (`v`) to its omega vertex (w) is set to its weight (line 18). If multiple edges with the same alpha and omega vertex exist, the edge with the lowest weight is taken. This is ensured by the condition in line 17. The initialization of the successor array is the same as in Warshall (line 19).

The main loop, that iterates over all possible vertex triples, is also the same. The important difference is in lines 27 - 31. Here, the current shortest distance from vertex u to vertex w is compared to the distance of following the path from u over v to w. If following this path is shorter, the edge $u \to v$ is stored in the `successor` array and the new shorter distance is stored in the `distance` array.

If any entry in the `distance` function contains a negative value for any vertex to itself (line 32), the graph contains at least one negative cycle (line 33) and the algorithm can terminate (line 34).

If the graph does not contain negative cycles, the function `distance` provides the distance for every vertex pair. The function `successor` contains the information for creating a shortest weighted path for every vertex to every vertex. The actual paths can be computed offline.

Listing 3.16: The Floyd algorithm

```
// results
BinaryFunction<Vertex,Vertex,double> distance;
BinaryFunction<Vertex,Vertex,Edge> successor;
boolean negativeCycleDetected = false;

public void floyd(Graph g, Function<Edge,double> weight){
    for (Vertex v: g.vertices()) {
        for (Vertex w: g.vertices()) {
            distance.set(v,w,infinity);
        }
        distance.set(v,v,0);
    }

    for (Edge e@(v->w): g.getAllEdges()) {
        if (v != w) {
            double newDistance = weight.get(e);
            if(newDistance < distance.get(v,w)){
                distance.set(v,w,newDistance);
                successor.set(v,w,e);
            }
        }
    }

    for (Vertex v: g.vertices()) {
        for (Vertex u: g.vertices()) {
              for (Vertex w: g.vertices()) {
                double newDistance = distance.get(u,v) + distance.get(v,w);
                if (distance.get(u,w) > newDistance){
                    distance.set(u,w,newDistance);
                    successor.set(u,w,successor.get(u,v));
                }
                if (u == w && distance.get(u,w) < 0){
                    negativeCycleDetected = true;
                    // terminate
                }
            }
        }
    }
}
```

Solution: Negative cycles, weighted distances and weighted shortest paths

Apply the Floyd algorithm

Space complexity: $\mathcal{O}(n^2)$

Time complexity: $\mathcal{O}(n^3)$

Since the Floyd algorithm relies on the same iteration over all vertex triples and computes similar binary functions, it succumbs the same efficiency problem as Warshall.

3.3.6.2 The Ford-Moore algorithm

If only some weighted shortest paths are relevant, using the Floyd algorithm would be unreasonable. In the following, more restricted problems are defined, whose result contains only one weighted shortest path instead of all.

Problem: Weighted distances from r
Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$; G may not have negative cycles with respect to g
Output: the distance $\overrightarrow{d_g}(r, v)$ for every vertex $v \in V_r$
Output type: a function $d_r : V \to \mathbb{R}$

Problem: Weighted shortest paths from r
Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$; G may not have negative cycles with respect to g
Output: for every vertex v with $r \to^* v$ a path from r to v with minimal weighted length
Output type: a function $wsp_r : V \to \mathcal{PATH}_G$

If only the weighted shortest path from one vertex to one target vertex is relevant, the following problems can be solved.

Problem: Weighted distance from r **to** t
Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$; G may not have negative cycles with respect to g
Output: the distance $\overrightarrow{d_g}(r, t)$
Output type: a value $d_{r,t} \in \mathbb{R}$ with $d_{r,t} = d_r(v)$

Problem: Weighted shortest path from r **to** t
Input: a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$
Output: a path from s to t
Output type: a path

All these problems can be solved using the *Ford-Moore* algorithm. It actually only solves the problems *weighted shortest distances from r* and *weighted shortest paths from r*. But all algorithms solving these problems also solve the problems *weighted shortest distance from r to t* and *weighted shortest path from r to t* implicitly.

The Ford-Moore algorithm can be found in listing 3.17.

The algorithm searches for the shortest paths in the reachable subgraph from r. A queue is used for all vertices whose incident edges might decrease the length of the shortest path to any vertex found so far.

Listing 3.17: The Ford-Moore algorithm

```
1  // results
2  Function<Vertex,Edge> parent;
3  Function<Vertex,double> distance;
4
5  // runtime variables
6  Function<Vertex,int> pushCount;
7  /* << a queue for vertices >> */
8  boolean negativeCycleDetected;
9
10 public void fordMoore(Graph g, Function<Edge,double> weight, Vertex r){
11     int maxPushCount = g.getVCount() - 1;
12     for(Vertex v: g.vertices(){
13         distance.set(v,infinity);
14     }
15     distance.set(r,0);
16     << add r to queue >>
17     while (<< queue is not empty >>) {
18         << get next vertex v from queue >>
19         for (Edge e: v.incidences(OUT)) {
20             Vertex w = e.getThat();
21             double newDistance = distance.get(v) + weight.get(e);
22             if (distance.get(w) > newDistance){
23                 parent.set(w,e);
24                 distance.set(w,newDistance);
25                 int newCount = pushCount.get(w) + 1;
26                 if(newCount > maxPushCount){
27                     negativeCycleDetected = true;
28                     // terminate
29                 }
30                 pushCount.set(w,newCount);
31                 << add w to queue >>
32             }
33         }
34     }
35 }
```

The distance to all vertices is initialized with ∞ (lines 12-14). The distance to the start vertex `r` is initially set to 0 (line 15).

In the main loop of the algorithm (lines 17-34), firstly a vertex is taken from the queue (line 18). The inner loop (lines 19-33) iterates over all relevant incident edges from this vertex. A new potential distance from r to w, the *that* vertex of the current edge, is computed (line 21). Only if the potential new distance is shorter, it is actually set as distance from r to w (lines 22-32) and w is added to the queue (line 31). The path information is stored in the function `parent` in analogy to the parent function used for directed search algorithms (see section 3.3.1).

If the reachable subgraph from r contains negative cycles, the Ford-Moore algorithm provides a mechanism to detect that. The nature of the algorithm requires that vertices can be added to the queue multiple times. The algorithm keeps track of this in the function `pushCount`.

The variable `maxPushCount` limits the number of pushes per vertex to $\mid V \mid -1$. If this limit is exceeded, the algorithm detects a negative cycle and terminates (lines 26-29). Without this mechanism, the algorithm would never terminate in case of negative cycles. The vertices on negative cycles would be added endlessly to the queue.

If no negative cycle was detected, the required weighted shortest paths can be computed offline using the function `parent`.

Solution: Weighted distances from r, weighted shortest distance from r to t, weighted shortest paths from r and weighted shortest path from r to t
Apply the Ford-Moore algorithm from listing 3.17
Space complexity: $\mathcal{O}(n)$
Time complexity: $\mathcal{O}(n \cdot m)$

This algorithm has a better time- and space complexity than Floyd and can also be applied on bigger graphs.

3.3.6.3 The Dijkstra algorithm

If the limitation of non-negative cycles is tightened to a limitation to non-negative weights, the problems from above can be solved using the *Dijkstra* algorithm. Depending on the vertex and edge count, this algorithm can have a better time complexity than the Ford-Moore algorithm.

The Dijkstra algorithm is very similar to the Ford-Moore algorithm from listing 3.17. Instead of a queue, a priority queue is used that is sorted by the distance computed so far. The priority queue is assumed to have a time complexity of $\mathcal{O}(\log(n))$ for adding and removing items.

The Dijkstra algorithm can be found in listing 3.18.

The algorithm uses the function `number` for deciding whether a vertex has been visited or not. The function `number` also defines the order in which the vertices are removed from the priority queue for the first time. In line 17, the vertex with the lowest cost so far is removed from the priority queue. Because of the limitation on positive weights, at this point, a shortest path from `r` to `v` has been computed.

The nature of this implementation allows vertices to be added to the priority queue multiple times with different priorities. So the inner loop(21-30) is only executed, if the current vertex has not already been removed from the queue before with a higher priority. This loop is almost identical to the inner loop in the Ford-Moore algorithm. The difference is the usage of the priority queue.

Depending on the ratio between the vertex and edge count, this algorithm can provide a more efficient solution for the problem of weighted shortest paths from r, compared to the Ford-Moore algorithm. Its disadvantage is the limitation on positive weights.

Listing 3.18: The Dijkstra algorithm

```
// results
Function<Vertex,int> number;
Function<Vertex,Edge> parent;
Function<Vertex,double> distance;

// runtime variables
/* << a priority queue for vertices sorted by distance >> */
int num = 0;

public void dijkstra(Graph g, Function<Edge,double> weight, Vertex r){
    for(Vertex v: g.vertices(){
        distance.set(v,infinity);
    }
    distance.set(r,0);
    << add r to priority queue with value 0 >>
    while (<< priority queue is not empty >>) {
        << get next vertex v from priority queue >>
        if (!number.isSet(v)) {
            number.set(v,++num);
            // working point: visitVertex(v)
            for (Edge e: v.incidences(OUT)) {
                Vertex w = e.getThat();
                double newDistance = distance.get(v) + weight.get(e);
                // working point: visitEdge(e)
                if (distance.get(w) > newDistance){
                    parent.set(w,e);
                    distance.set(w,newDistance);
                    << add w to priority queue with value newDistance >>
                 }
            }
        }
    }
}
```

As in Ford-Moore, the actual relevant shortest paths can be computed offline using the function parent.

Solution: Weighted distances from r, weighted shortest distance from r to t, weighted shortest paths from r, and weighted shortest path from r to t with positive weights
Apply the Dijkstra algorithm from listing 3.18
Space complexity: $\mathcal{O}(m)$
Time complexity: $\mathcal{O}(m \cdot \log(m))$

It is also possible to use a different priority queue implementation which uses an unsorted list instead of a heap. In such an implementation, adding to the queue requires constant time. However, removing elements requires linear time, which would change the time complexity of Dijkstra to $\mathcal{O}(max(m, n^2))$.

3.3.6.4 The Dijkstra algorithm with early termination

For the solution of the problems *weighted shortest distance from r to t* and *weighted shortest path from r to t*, the Dijkstra algorithm can be improved, by terminating the algorithm when the target vertex t is removed from the priority queue. This can be achieved by extending the algorithm as shown in listing 3.19.

Listing 3.19: Extension of the Dijkstra algortihm for early termination

```
1 // change method signature to
2 public void dijkstra(Graph g, Function<Edge,double> weight, Vertex r,
     Vertex t)
3
4 // add to working point: visitVertex(v)
5 if (v == t){
6     // terminate
7 }
```

Due to the early termination of the algorithm, only the computed path from r to t, is guaranteed to be a shortest path.

3.3.6.5 The A*-Search

A further improvement can be achieved by adding additional information to the Dijkstra algorithm.

Definition: Heuristic
Given a graph $G = (V, E, \varphi)$ and an edge weight function $g : E \to \mathbb{R}$,
a *heuristic* is a function $h : V \times V \to \mathbb{R}$ with the following properties:

- h is optimistic: $\forall v \in V, w \in V : h(v, w) \leq d(v, t)$
- h is monotonic: $\forall (u \to v) \in E, w \in V : h(u, w) \leq g(u \to v) + h(v, w)$

A heuristic h assigns each vertex $v \in V$ an estimated distance to any vertex $w \in V$.

In addition to the target vertex t, a heuristic h is passed to the algorithm. This heuristic is used for directing the search towards the target vertex t and for avoiding paths that do not lead to t. The resulting algorithm is called *A*-Search*. Listing 3.20 shows the A*-Search.

The structure is very similar to the Dijkstra algorithm. The algorithm is terminated when the target vertex is removed from the priority queue (line 22). The priority queue is sorted by *distance* + *heuristic* (line 31), meaning, the search is directed to the vertices that are assumed to be good candidates for the shortest path.

The actual path can be computed offline. If the target vertex t is not reachable from s, the algorithm searches the whole reachable subgraph from r and $parent(t)$ is undefined. If the

Listing 3.20: The A*-Search algorithm

```
// results
Function<Vertex,int> number;
Function<Vertex,Edge> parent;
Function<Vertex,double> distance;

// runtime variables
/* << a priority queue for vertices sorted by distance >> */
int num = 0;

public void aStar(Graph g, Function<Edge,double> weight,
    BinaryFunction<Vertex,Vertex,double> heuristic, Vertex r, Vertex t){
    for(Vertex v: g.vertices(){
        distance.set(v,infinity);
    }
    distance.set(r,0);
    /* add r to priority queue with value 0 */
    while (/* priority queue is not empty */) {
        /* get next vertex v from priority queue */
        if (!number.isSet(v)) {
            number.set(v,++num);
            // working point: visitVertex(v)
            if (v == t){
                // terminate
            }
            for (Edge e: v.incidences(OUT)) {
                Vertex w = e.getThat();
                double newDistance = distance.get(v) + weight.get(e);
                // working point: visitEdge(e)
                if (distance.get(w) > newDistance){
                    parent.set(w,e);
                    distance.set(w,newDistance);
                    /* add w to priority queue with value (newDistance +
                       heuristic.get(w,t))*/
                }
            }
        }
    }
}
```

heuristic is good, the A*-Search algorithm may find the shortest path earlier than the Dijkstra algorithm with early termination. If $\forall\, v \in V : h(v,t) = 0$, the A*-Search algorithm is equivalent to a Dijkstra algorithm with early termination.

Solution: Weighted shortest path from r to t with positive weights

Apply the A*-Search algorithm from listing 3.20

Space complexity: $\mathcal{O}(m)$

Time complexity: $\mathcal{O}(m \cdot \log(m))$

As in Dijkstra, the time complexity also depends on the the implementation of the priority queue. So if using an unsorted list instead of a heap, the time complexity is changed to $\mathcal{O}(max(m, n^2))$.

4 Java mapping

This chapter describes how the mathematical structures, used in chapter 3, are realized in Java. It also describes the role of JGraLab and how it is used for and will be extended with graph algorithms.

4.1 JGraLab

The implementation of all graph algorithms in *Algolib* is done using JGraLab. *Algolib* extends JGraLab with these algorithms and becomes a part of JGraLab. For graph access, JGraLab provides the interfaces described in section 3.1. The actual interfaces in JGralab provide more methods, but they are not relevant for *Algolib* and therefore are not explained here.

4.1.1 Ordered graphs

As described in section 2.4 the graphs implemented by JGraLab have several properties. The property of edges, vertices, and incidences being ordered is of high importance for most graph algorithms. The method `edges(...)` from the interface `Graph` provides iteration over the edges in edge order. The method `vertices(...)` from the interface `Graph` analogously provides iteration over all vertices in vertex order. E.g., the algorithm *complete search* uses the method `vertices(...)` for deciding the start vertex of the next search. So the vertex order in the graph dictates the order the vertices are used as start vertex for the search. The method `incidences(...)` from the interface `Vertex` provides iteration over all incident edges in incidence order. This fact has a high impact on the order incident edges are followed. E.g., for search algorithm, the incidence order also dictates the layout of the search tree.

4.1.2 Subgraphs

JGraLab provides a mechanism for declaring subgraphs. However, subgraphs are not manifested as actual objects. A subgraph is declared by using a function that is defined for both, edges and vertices and assigns for each graph element whether it is part of the subgraph or not. This function is called `subgraph function` in the following. The combination of a graph and a subgraph function defines a subgraph. *Algolib* can exploit this for applying graph algorithms on subgraphs.

Using subgraph functions also makes *Algolib* compatible with GReQL, which is able to compute them. Every algorithm in *Algolib* supports subgraph functions.

The next section shows how functions are implemented in *Algolib*.

4.2 Functions

This section describes the realization of functions in *Algolib*. This also includes other mathematical structures that can be interpreted as functions (e.g., relations).

In this context functions are generally partial functions. This is especially true for functions that are computed as results. However, some input functions may also be total.

4.2.1 Unary functions

For unary functions *Algolib* provides the interface `Function` as described in section 3.2.4 with additional methods. The whole interface can be found in listing 4.1.

Listing 4.1: The interface `Function`

```
1 public interface Function<DOMAIN,RANGE> extends
      Iterable<FunctionEntry<DOMAIN, RANGE>>{
2     public RANGE get(DOMAIN parameter);
3     public boolean isDefined(DOMAIN parameter);
4     public boolean set(DOMAIN parameter, RANGE value);
5     public Iterable<DOMAIN> getDomainElements();
6 }
```

The method `get(...)` retrieves or computes the function value for the given `parameter`. The method `isDefined(...)` returns `true` if the function is defined for the given `parameter`, `false` otherwise. This method is useful for deciding, if a function value exists for the given `parameter`. The method `set(...)` is used to modify the function. It defines a value for a given parameter. If the function already contains a value for the parameter, it is overwritten. This is an optional operation that throws an `UnsupportedOperationException` if the function is immutable. The method `getDomainElements()` returns an iterator that enumerates all domain elements of a function. This is an optional operation that throws an `UnsupportedOperationException` if the domain elements are not enumerable. The inherited method `iterator()` returns an iterator that enumerates all function entries of a function. The class `FunctionEntry<DOMAIN,RANGE>` is a self-defined tuple type.

4.2.1.1 Functions with primitive types as domain or range

As seen in chapter 3, some of the functions have a domain or range that corresponds to a primitive type. It is possible to use the interface `Function` with wrapper classes of primitive types

and use Java's mechanisms of autoboxing and unboxing. This technique, however, at least doubles the amount of memory used for a value. When computing functions for large graphs, this can be a handicap. *Algolib* tries to avoid autoboxing and unboxing wherever possible.

For functions with the primitive ranges `int`, `long`, `double`, and `boolean`, *Algolib* provides specialized interfaces that only allow a type variable for the domain of the function. The names of these interfaces are derived from the primitive type names by using the primitive type name as prefix. E.g., the interface representing all functions with range `double` is called `DoubleFunction`. The definition of this interface is analogous to the interface `Function` in listing 4.1. It only contains one generic type, which is `DOMAIN`.

The primitive function interfaces are also capable of providing an iterator for enumerating all function entries. However, the generic tuple type `FunctionEntry<DOMAIN,RANGE>` is not used here, because this would require autoboxing and unboxing. *Algolib* provides special tuple types for functions with primitive range. E.g., the tuple type for `DoubleFunction` is called `DoubleFunctionEntry`. The other entry types are called analogously.

There are also functions with primitive types as domain. However, in *Algolib* $\mathbb{N}$ is so-far the only domain that corresponds to a primitive type. All functions in *Algolib* that have this domain can be interpreted as permutations and they are treated differently (see section 4.3).

4.2.1.2 The implementation of unary functions

Most of the algorithms in *Algolib* compute functions during their run. The computation of a function can be done with an arbitrary internal data structure. The result however has to implement one of *Algolib*'s function interfaces. This can be achieved by wrapping the data structure inside an object that implements a function interface.Alternatively the class representing the type of the data structure can directly implement one of the function interfaces.

Functions whose domain is either V or E are realized using JGraLab's mechanism for treating so-called *temporary attributes*. Temporary attributes are attributes for graph elements that are not stored in the graph and only valid during runtime. Temporary attributes can be of arbitrary types. They are realized using so-called *graph markers*.

Graph markers

A graph marker is a data structure that allows assigning each graph element an additional attribute. Most graph markers are limited to either assign attributes to vertices only or edges only. An arbitrary number of graph markers may exist. They are only valid at runtime and their data cannot be stored with the graph.

JGraLab provides several graph markers using different implementations.

One type of graph marker is implemented using hash maps (map graph markers). Map graph markers are very useful if only a fraction of graph elements should receive temporary attributes. Since Java's implementation of hash maps is used, using primitive types as attribute type would result in autoboxing and unboxing. Another type of graph markers is implemented using arrays (array graph markers). The implementation of array graph markers has been motivated by the creation of *Algolib*. Compared to map graph markers, they provide a more efficient memory consumption if most vertices or edges are expected to receive temporary attributes. They also allow the usage of primitive types without autoboxing and unboxing. JGraLab provides array graph markers for the primitive types `int`, `long`, `double`, and `boolean`[1]). In general, *Algolib* uses array graph markers, especially for avoiding autoboxing und unboxing.

With the introduction of *Algolib*, all graph markers implement one of *Algolib*'s function interfaces. In *Algolib* they are operated using the function interfaces.

Table 4.1 shows a mapping of some function interfaces and the corresponding graph marker classes that are used in *Algolib*. All graph markers can be found in the package

Function interface	Corresponding graph marker class
`Function<Vertex,RANGE>`	`ArrayVertexMarker<RANGE>`
`Function<Edge,RANGE>`	`ArrayEdgeMarker<RANGE>`
`IntFunction<Vertex>`	`IntegerVertexMarker`
`IntFunction<Edge>`	`IntegerEdgeMarker`
`DoubleFunction<Vertex>`	`DoubleVertexMarker`
`DoubleFunction<Edge>`	`DoubleEdgeMarker`
`LongFunction<Vertex>`	`LongVertexMarker`
`LongFunction<Edge>`	`LongEdgeMarker`
`BooleanFunction<Vertex>`	`BitSetVertexMarker`
`BooleanFunction<Edge>`	`BitSetEdgeMarker`
`BooleanFunction<GraphElement>`	`SubGraphMarker`

Table 4.1: Overview of graph marker classes

`de.uni_koblenz.jgralab.graphmarker`. A graph marker has to be initialized with the graph whose graph elements should be extended with a temporary attribute. If graph elements should receive multiple temporary attributes, a graph marker with a tuple type as `RANGE` can be used. The alternative is using multiple graph markers. *Algolib* uses the latter for flexibility reasons[2].

4.2.2 Binary Functions

As seen in chapter 3, also binary functions occur in *Algolib*. For this purpose, *Algolib* contains the interface `BinaryFunction`, that was already introduced in chapter 3.2.4 on page 33.

[1] The graph markers with range `boolean` are implemented using a `BitSet` instead of an array, but conceptually they are equivalent to array graph markers.

[2] Chapter 6 will show that the chosen alternative also provides a higher speed efficiency.

The methods work in analogy to the methods from the interface `Function`. The interface `BinaryFunction` does not provide methods for creating iterators for enumerating domain elements or function entries. For binary functions such methods would be inefficient and they are not required.

For allowing primitive ranges for binary functions, *Algolib* provides special interfaces. They are named similarly to the corresponding interfaces for unary functions. E.g., the interface for the primitive type `double` is called `BinaryDoubleFunction<DOMAIN1,DOMAIN2>`.

4.2.2.1 The implementation of binary functions with domain $V \times V$

In *Algolib* the most relevant domain for binary functions is $V \times V$. The internal data structure for such functions in *Algolib* is a two dimensional quadratic array of the range type. E.g., in the case of *successor* from the Warshall- and Floyd algorithm, it is `Edge[][]`. For runtime efficiency reasons, algorithms that compute binary functions with domain $V \times V$ operate directly on the array.

An issue when using a two dimensional array for this type of function is the index numbering. In JGraLab, each vertex has a unique id, which is always a positive `int` value. So the easiest index numbering would be using the vertex id. However, this numbering is not necessarily contiguous, because there may be unused ids in the graph. Since the space complexity of a two dimensional array is $O(n^2)$, this is not feasible for graphs with many unused ids.

To overcome this problem, an alternative numbering has to be used instead. Any complete search algorithms provided by *Algolib* can be used for computing the function *number* that creates a continuous numbering of all vertices. So all algorithms that compute binary functions with domain $V \times V$ should run an arbitrary search algorithm first for computing the function *number* and using it as index numbering. Please note that the numbering, provided by *number*, by convention starts with 1. So the array indexing also starts with the index 1 causing the arrays at the position 0 to be `null` in both dimensions.

When the result is requested, the two dimensional array representation of the function is wrapped in an implementation of the interface `BinaryFunction<Vertex,Vertex,RANGE>`. For this purpose, *Algolib* already provides the class `ArrayBinaryFunction<DOMAIN,RANGE>`. It only has one type variable for the domain type, because it assumes both parts of the domain being of the same type. For the domain $V \times V$ this type variable is set to `Vertex`. This class turns the two dimensional array into an immutable instance of `BinaryFunction<Vertex,Vertex,RANGE>`, meaning the method `add(...)` throws an exception.

If `RANGE` is a primitive type, the corresponding interface for primitive types is used. *Algolib* also provides corresponding classes for wrapping two dimensional arrays representing binary functions with primitive range types.

4.2.3 Method calls

It is possible to wrap simple method calls in function objects. In doing this, it is very easy to specify functions whose values are not obtained by a lookup in a data structure. Such functions are immutable and can only serve as input functions.

Algolib provides adapter classes for method calls that implement the corresponding function interfaces. In these adapters, the methods `set(...)`, `getDomainElements(...)` and `iterator(...)` throw exceptions.

For example the heuristic of the A*-search can be realized using the adapter class for `BinaryDoubleFunction`.

4.3 Permutations

Permutations are sequences of all elements of a set. As done in chapter 3, permutations can be interpreted as functions with domain $\mathbb{N}$. In *Algolib* permutations have an interface of their own that is defined similarly to the interface `Function`. This interface can be found in listing 4.2.

Listing 4.2: The interface `Permutation`

```
1 public interface Permutation<RANGE> extends
     Iterable<PermutationEntry<RANGE>>{
2     public RANGE get(int index);
3     public void add(RANGE value);
4     public boolean isDefined(int index);
5     public int length();
6     public Iterable<RANGE> getRangeElements();
7 }
```

The method `get(...)` retrieves or computes the function value for the given `index`. The method `add(...)` adds the given `value` to the end of the permutation. This is an optional operation that throws an `UnsupportedOperationException` if the permutation is immutable. A permutation is built successively, starting at index 1. The method `isDefined(...)` returns `true` if the permutation contains a value at the given `index`, `false` otherwise. The method `length()` retrieves or computes the length of the permutation, which is equivalent to the index of the last element that was added due to the index numbering convention. The method `getRangeElements()` returns an iterator that enumerates all entries of the permutation in the same order as they have been added to the permutation (from 1 to `length`). The inherited method `iterator()` returns an iterator that enumerates all permutation entries in analogy to the interface `Function`. The class `PermutationEntry<RANGE>` is a self-defined tuple type whose first element is of the primitive type `int`.

An example for a permutation is the function *vertexOrder* that is computed by all search algorithms.

4.3.1 The implementation of permutations

In *Algolib*, the underlying data structure of a permutation is an array of the range type. E.g., in case of `vertexOrder` this is `Vertex[]`. For runtime efficiency reasons, algorithms that compute permutations operate directly on the array. Here the indexing also stars at 1 causing the first index of the array (0) to be undefined. For arrays this means, the first field at position 0 is always empty.

When the result is requested, the array representation of the permutation is wrapped in an implementation of the interface `Permutation<RANGE>`. For this purpose, *Algolib* already provides the class `ArrayPermutation<RANGE>`. This class turns the array into an immutable instance of `Permutation<RANGE>`, meaning the method `add(...)` throws an exception. This class also provides appropriate iterators for the methods `iterator()` and `getRangeElements()`.

4.4 Partitions

Some algorithms compute partitions on either the vertex- or edge set. Partitions can be interpreted as representative functions (see page 21 for definition).

In *Algolib*, representative functions, defining partitions, are realized using graph markers (see section 4.2.1.2).

If a different representation of partitions is required (e.g., `Set<Vertex>[]`), the representative function can be used for computing such a representation offline.

An example for a partition on V are the strong components of a graph.

4.5 Relations

Relations on either the vertex set V or the edge set E are realized using their *characteristic function*. This is a function whose domain is $V \times V$ or $E \times E$ and whose range is *boolean*. The function value indicates, whether the given pair is an element of the relation or not.

In *Algolib*, characteristic functions are realized using the interface `Relation`, which is actually a special interface for binary functions whose domain is `boolean`. All its methods work in analogy to the interface `BinaryFunction`.

An example for a relation is the reachability relation computed by the Warshall algorithm.

The implementation of relations on V follows the same rules as implementations of binary functions with domain $V \times V$ as described in section 4.2.2.1.

The corresponding interface is `Relation<DOMAIN1,DOMAIN2>` and the class for wrapping a two dimensional arrays representing relations is `ArrayRelation<DOMAIN>`.

5 Implementation

This chapter describes how some of the graph algorithms from chapter 3 are implemented in *Algolib*. It introduces the strategy pattern and the visitor pattern and shows how these are used for algorithm implementation.

Section 5.1 describes *Algolib*'s package structure and how it is integrated into JGraLab's. Section 5.2 introduces the strategy pattern and shows how the problems from chapter 3 are specified in *Algolib*. Section 5.3.4 is the major part of this chapter. It introduces the visitor pattern, shows common classes, needed for algorithm creation and gives some examples on how concrete algorithms are implemented.

5.1 Package structure

This section describes the package structure of *Algolib*. It shows where the packages of the current implementation are located. Figure 5.1 shows the package structure of *Algolib*.

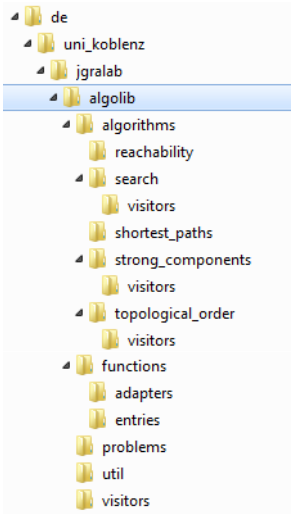


Figure 5.1: Overview of the package structure

Since *Algolib* is an extension of JGraLab, the base package of *Algolib* is a subpackage of JGraLab's base package. So the base package of *Algolib* is `de.uni_koblenz.jgralab.algolib`. All further package references are relative to

this base package (e.g., `problems`). If other subpackages of JGraLab are referenced, the prefix `jgralab` is used (e.g., `jgralab.graphmarker`).

5.1.1 Package `functions`

The package `functions` contains all function interfaces described in section 4.2. It also contains the interfaces for permutations and relations.

The tuple types implementing function entries for unary partial functions, which are used for the iterators created by functions, can be found in the package `functions.entries` (see section 4.2.1). The adapter classes for easily wrapping method calls in function interfaces (see section 4.2.3) can be found in the package `functions.adapters`.

5.1.2 Package `problems`

The package `problems` contains all problem interfaces. Details about these interfaces can be found in section 5.2.

5.1.3 Package `algorithms`

The package `algorithms` contains all algorithm implementation in *Algolib*. Details about the algorithm implementations can be found in section 5.3 on page 78.

The subpackages are mostly named after the problem groups from section 3.3. Table 5.1 shows, which problem group corresponds to which subpackage. It also gives an overview of the subpackages that currently exist in *Algolib*. The order of the entries corresponds to the order the problem groups have been introduced in section 3.3.

problem group	package
traversal	`algorithms.search`
acyclicity	`algorithms.topological_order`
shortest paths	`algorithms.shortest_paths`
strong components	`algorithms.strong_components`
reachability	`algorithms.reachability`
weighted shortest paths	`algorithms.shortest_paths`

Table 5.1: overview of algorithm packages

Abstract classes, that give several common attributes and behavior to algorithms from multiple subpackages, can be found directly in the package `algolib.algorithms`.

5.1.4 Package `visitors`

The package `visitors` contains all base interfaces and base implementations of visitors. Details about visitors can be found in section 5.3.2.

If a problem group has their own visitors, they will be placed in a subpackage named `visitors` of the problem group. E.g., the visitors for search algorithms can be found in the package `algorithms.search.visitors`.

This package does not contain any subpackages.

5.1.5 Package `util`

The package `util` contains everything that does not belong to the other packages. This includes data structures, such as custom worklists, that are only used by *Algolib* and convenience classes for post-processing several results.

5.2 Problems

In *Algolib*, problems and their solution are strictly separated. This section shows how problems are described in *Algolib* and it gives a first hint on how solutions can be realized. The core idea of the problems and their solution is the usage of the strategy pattern which is introduced in the following.

5.2.1 The strategy pattern

The strategy pattern is one of the behavioral patterns introduced by Gamma, Helm, Johnson and Vlissides [GHJV94]. The pattern allows defining a set of algorithms that solve the same problem, encapsulating each implementation and making them interchangeable. That means, algorithms are made first class citizens. When solving a problem, the decision which implementation should be used can be made at runtime.

For achieving this, an interface is created, which describes the problem, specifies the input parameters and results and declares a so-called *execute method* that is used to solve the described problem. Classes implementing this interface provide a *strategy* (or algorithm) to solve the described problem. This is mainly done by implementing the execute method that was declared in the interface. This allows a decision made at runtime about which solution for a problem to use.

There are multiple possibilities how to handle input parameters and results. The parameters can be passed as arguments to the execute method or by setter methods before the execute method is invoked. The result can be the return type of the execute method or obtained by

a getter method after the execute method has terminated. It is also possible to mix these approaches or provide multiple alternatives.

All algorithms in *Algolib* are implemented using the strategy pattern. The following section describes (among other things), how the strategy pattern is actually implemented in *Algolib*.

5.2.2 The problem interfaces

The description of a problem and the specification of an interface for algorithms solving this problem is done using a Java interface. Such interfaces are called *problem interfaces* in the following. As mentioned in section 5.1.2, all problem interfaces can be found in the package `algolib.problems`. Descriptions of the problems are phrased similarly to the descriptions of problems in chapter 3.

Algolib has a naming convention for these problem interfaces. The name consists of the problem name, followed by the suffix `Solver`. E.g., the interface for the problem *simple paths* is called `SimplePathsSolver`. The suffix `Solver` was chosen, because all algorithms that solve a problem have to implement the problem interfaces of all the problems they solve. This makes algorithms *problem solvers* and the problem interfaces should reflect this.

A difference arises from the handling of input parameters and results compared to chapter 3. There, input parameters and results have been described very vaguely. Here, they have to be specified more concretely. But first two different types of input parameters have to be distinguished.

5.2.2.1 Input parameters

In chapter 3, all input parameters were passed directly to the algorithms. This was done to simplify their description.

Algolib distinguishes between two different types of input parameters. The first type is passed directly to the execute method. It is only valid until the execute method terminates. This type of parameter is called *transient parameter* in the following. The second type has to be set before the execute method is invoked. For setting the latter type of parameters, setter methods are used. The name of such a setter method is `set«ParameterName»`. These types of parameters are called *durable parameters* because they remain valid even after the execute method terminated.

Both, durable and transient parameters, are described in the Javadoc text of the interface itself. Section 5.2.3 gives an overview of the problem interfaces that currently exist in *Algolib*. Some of these interfaces only specify durable parameters and do not have an execute method.

The Javadoc text of the setter methods and of the execute method can provide further information on the input parameters. This can be implementation related information, which is irrelevant for the problem description.

5.2.2.2 Results

In *Algolib*, The handling of results is done using getter methods. The name of the getter methods for results is `get«ResultName»`. This method has to be called after the execute method terminated for retrieving the result of the algorithm. The description of the results is done in the Javadoc text of the interface itself.

The Javadoc text of the getter methods can provide further information on the results. This can be implementation related information, which is irrelevant for the problem description.

Only mandatory results require getter methods in the problem interfaces. All optional results are specified in the implementations.

5.2.2.3 The execute method

For all problem specifications, the execute method has the name `execute`. The return type is always the problem interface in which the execute method is declared in. This allows an easy pattern for accessing one result after the execute method has been terminated:

```
ResultType r = solver.execute().getResult();
```

The following code snippet shows a real example on how the BFS can be called for computing the order function.

```
Permutation<Vertex> order = bfs.execute().getOrder();
```

All execute methods are declared to throw the exception `AlgorithmTerminatedException` for allowing early termination. Details about this can be found in section 5.3.4.3 on page 91.

The parameter list of an execute method only contains the transient parameters of the problem.

5.2.3 Overview of problem interfaces

Figure 5.2 shows all problem interfaces that currently exist in *Algolib*. The super interface of all problem interfaces is called `ProblemSolver`. It has no execute method. It defines the common durable parameters for all problems, which are the graph and the function defining a subgraph. All other problem interfaces are derived from it.

The interface `TraversalSolver` has no execute method. It defines the common durable parameter `navigable`, which is a function that defines for every edge of the graph whether it is navigable or not. The navigability of an edge can depend on the start and end vertex. So the function `subgraph` is insufficient for deciding it. This is highly domain specific, so `navigable` provides a generic way of defining the navigability independently from

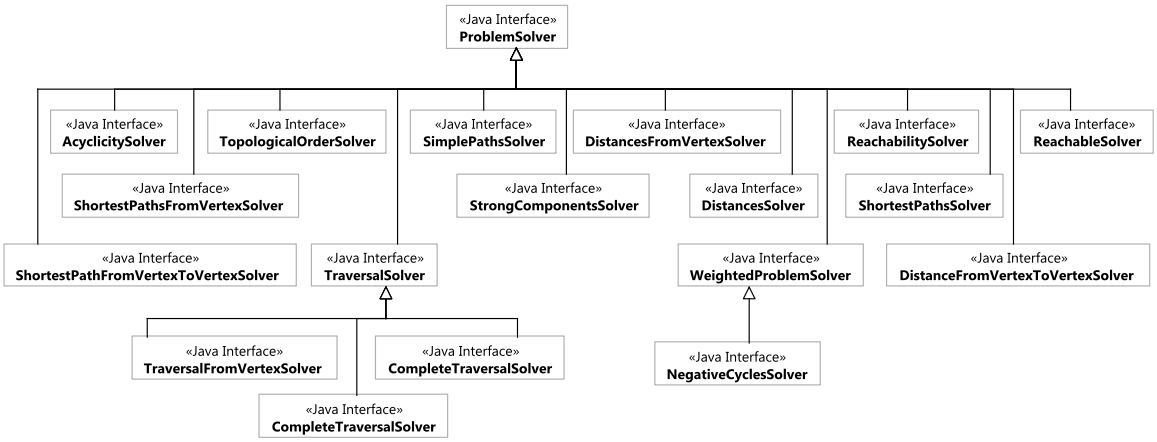


Figure 5.2: An overview of the problem interfaces

the function `subgraph`. `TraversalSolver` is the super interface of the problem interfaces `TraversalFromVertexSolver` and `CompleteTraversalSolver`.

The interface `WeightedProblemSolver` has no execute method either. It defines the common parameter `edgeWeight`, which is a function that assigns a weight to every edge of the graph. Algorithms operating on edge weighted graphs implement this interface. E.g., `FordMooreAlgorithm` implements (among others) the interfaces `ShortestPathsFromVertexSolver` and `WeightedProblemSolver`, because it solves the problem *weighted shortest paths from vertex*. It is the super interface of `NegativeCyclesSolver`, because the corresponding problem is only defined for edge weighted graphs.

5.3 Algorithms

This section describes the implementation of graph algorithms in *Algolib*.

Solutions of problems are called *algorithms* in the following. Classes implementing problem interfaces are called *algorithm classes*. There is one algorithm class per algorithm implementation. Instances of algorithm classes are called *algorithm objects*.

Algolib also contains some abstract algorithm classes that unite common features of algorithm classes. The next section gives an overview of the algorithm class hierarchy.

5.3.1 Overview of algorithm classes

Figure 5.3 shows all algorithm classes that currently exist in *Algolib*. The abstract super class of all algorithm classes is called `GraphAlgorithm`. It contains all common attributes and

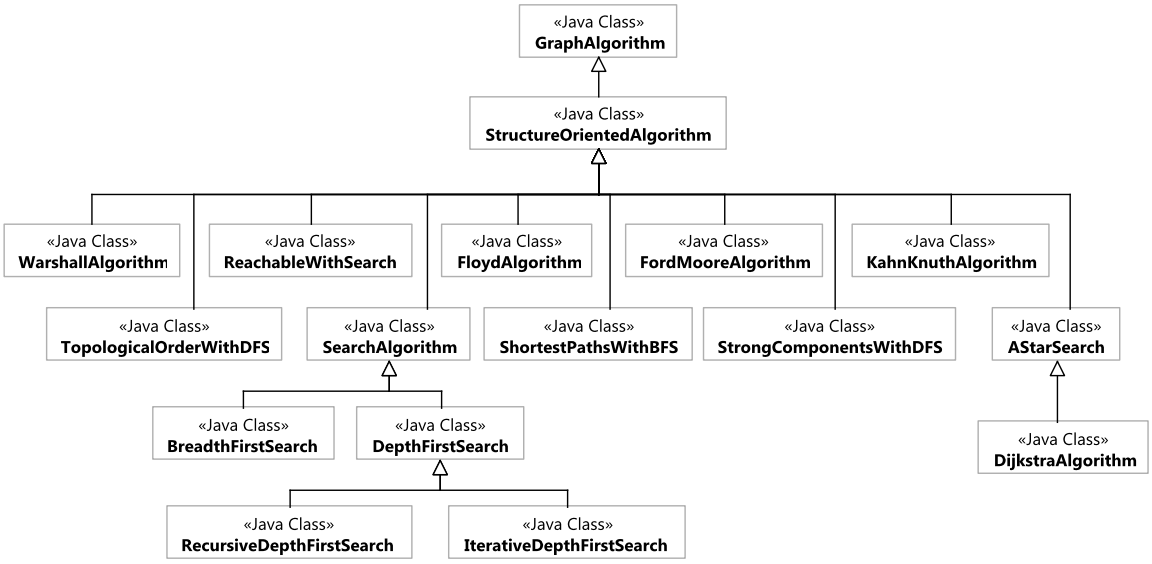


Figure 5.3: An overview of the algorithm classes

methods for all graph algorithms in *Algolib*. Details about its implementation can be found in section 5.3.4.3 on page 85.

The abstract super class of all algorithm classes, that implement algorithms with respect to the graph structure, is called `StructureOrientedAlgorithm`. This class allows a traversal of edges in normal, reversed, and undirected orientation. Currently, all algorithm classes are derived from this class. Details about its implementation can be found in section 5.3.4.4 on page 92.

5.3.2 The visitors pattern

Before actual implementations of algorithm classes can be shown, the implementation of working points has to be described. The concept of working points was introduced in section 3.2.2 on page 32. Here the implementation of working points in *Algolib* is described.

For extending algorithms with further functionality, a variant of the visitor pattern is used. The visitor pattern is one of the behavioral patterns introduced by Gamma, Helm, Johnson and Vlissides [GHJV94]. It is used for representing operations that are performed on elements of an object structure. The pattern allows the definition of new operations on these elements, without the need for changing the elements' classes.

In its original form, the visitor pattern is used for performing independent operations on data structures, that can be implemented without the need of changing the classes implementing the data structure. For this purpose, all elements of the data structure have to provide methods for *accepting* visitors. The visitors provide so-called *visit methods*, that are called in the *accept methods* of the elements. There is exactly one visit method per element class. The algorithm for

traversing the data structure is implemented in the visitors. This algorithm has to ensure that each element in the data structure is visited at most once (depending on the domain this can also be exactly once).

5.3.3 The visitor pattern in *Algolib*

JGraLab does not provide a visitor interface for vertices and edges. Even if it would do so, this would be inappropriate for the working points in graph algorithms. For *Algolib* a variant of the visitor pattern is used.

Instead of accepting visitors for each call of a visit method, they are registered in the algorithm objects, before the algorithm is executed. Algorithm objects can register an arbitrary number of visitors. Details about this can be found in section 5.3.3.2.

The traversal of elements is implemented in the algorithm classes instead of the visitor classes. In addition, there can be multiple visit methods per element type. E.g., in a search algorithm every edge can be visited as edge and additionally as either tree edge or as frond. Details about the different visitors and the visit methods they provide can be found in section 5.3.3.1.

It is also possible to define visit methods for element tuples. E.g., this is used in the Warshall algorithm for visiting vertex triples.

One rule has to be obeyed by all graph algorithms: *Every visit method may only be called at most once for every graph element or every graph element tuple respectively*.

In *Algolib* visitors can be used for computing additional results. In order to do this, they might need to have their own local runtime variables and they might need access to the algorithm object's runtime variables. Details about how this is achieved can be found in section 5.3.4.

Visitors can also be used for causing an algorithm to terminate earlier, e.g., in case the result has been computed before the algorithm would normally terminate. *Algolib* uses the exception `AlgorithmTerminatedException` for this purpose. All visit methods are declared to throw this exception. Details about early termination can be found in section 5.3.4.3 on page 91.

5.3.3.1 Overview of visitors

Figure 5.4 shows all visitor interfaces that currently exist in *Algolib*. The definition of interfaces is done in Java interfaces. These interfaces are called *visitor interfaces* in the following. Every visit method, that is provided by a visitor interface, corresponds to one of the working points introduced in chapter 3.

The most generic visitor is simply called `Visitor`. The method `reset()` is used for reinitializing the visitor's runtime variables. The method `setAlgorithm(...)` is used for binding the visitor to a specific algorithm object in order to access its runtime variables.

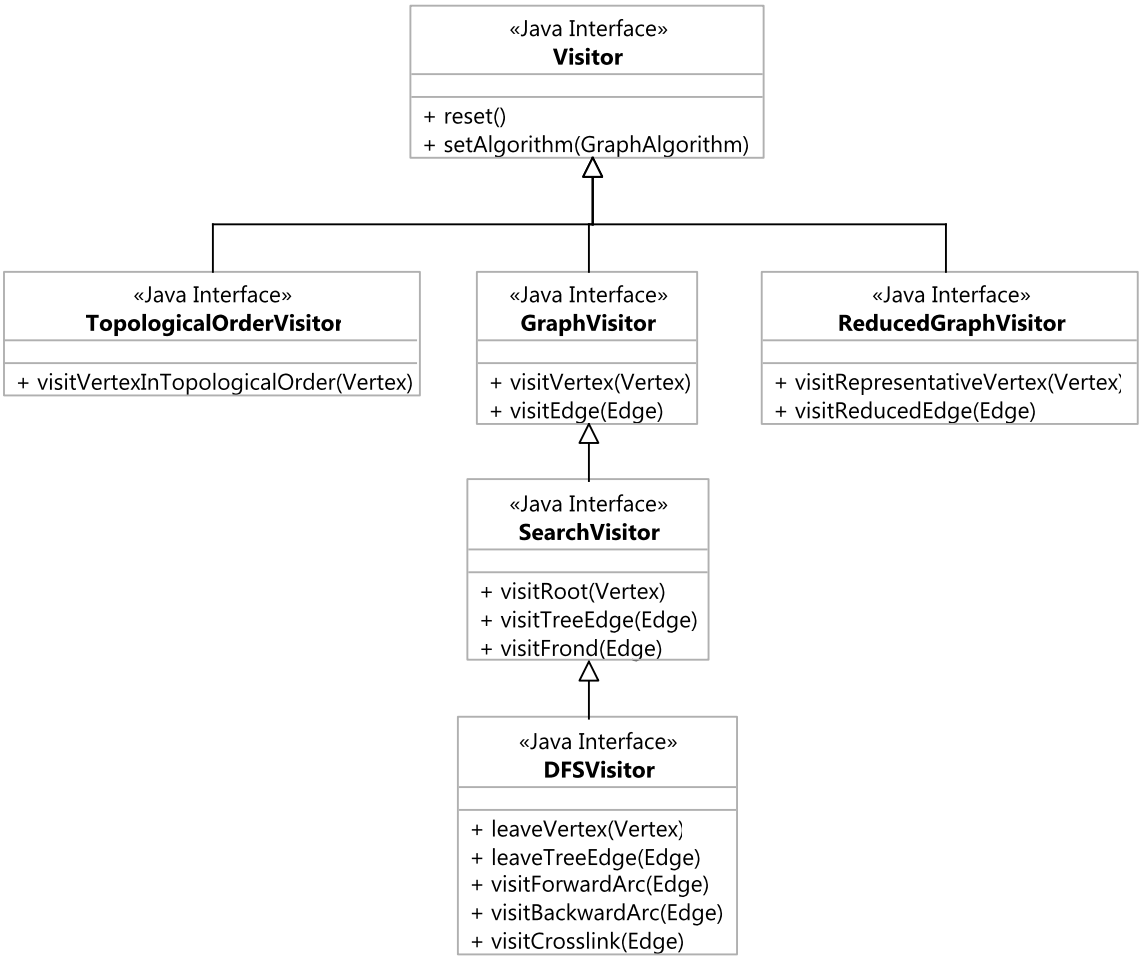


Figure 5.4: An overview of the visitor interfaces

The visitor `GraphVisitor` is a basic visitor for visiting graph elements without them being a special context in the algorithm. It provides the visit methods `visitVertex(...)` and `visitEdge(...)`.

The visitor `SearchVisitor` is a visitor that is used in search algorithms. It extends `GraphVisitor` with the visit methods `visitRoot(...)`, `visitTreeEdge(...)` and `visitFrond(...)`. `visitRoot(...)` visits vertices which are roots of search trees. `visitTreeEdge(...)` visits edges in search trees. `visitFrond(...)` visits edges that are in search trees.

The visitor `DFSVisitor` is a visitor that is used in the depth first search algorithm. It extends `SearchVisitor` with the visit methods `leaveVertex(...)`, `leaveTreeEdge(...)`, `visitForwardArc(...)`, `visitBackwardArc` and `visitCrosslink(...)`. `leaveVertex(...)` and `leaveTreeEdge(...)` visit vertices and edges after they have been completely traversed in the search (after the recursive call). The other visit methods distinguish between the different frond types.

The visitor `TopologicalOrderVisitor` is only used in algorithms that can compute a topological order (e.g., Kahn Knuth). It provides only one visit method (`visitVertexInTopologicalOrder(...)`), that visits all vertices of the graph in the topological order that is computed by the algorithm.

The visitor `ReducedGraphVisitor` is used by algorithms that can compute strong components. It provides the visit methods `visitRepresentativeVertex(...)` and `visitReducedEdge(...)`. `visitRepresentativeVertex(...)` visits the representative vertices of the strong components. For the algorithm described in section 3.3.4 on page 49 these are the strong roots. `visitReducedEdge(...)` visits the reduced edges that connect the strong components.

All visitors provide a visitor adapter for simplification. These adapters implement all methods mentioned above with empty stubs. This is in analogy to the adapter classes for listeners in Swing. The visitor adapters have an analogous inheritance tree in comparison to the the visitor interfaces.

5.3.3.2 Visitor list

In order to support multiple visitors, each visitor interface requires a so-called *visitor list*. In a nutshell, a visitor list is a class that implements a visitor interface containing a list of visitors. It delegates all visit method calls to the visitors contained in the list.

All algorithm classes internally use a visitor list for storing the visitor objects. When registering a visitor object, it is added to the visitor list of the algorithm object.

As mentioned above, visitors can be used for computing additional results. Furthermore it is possible that one visitor depends on the results of another visitor. Such visitors are called *dependent visitors* in the following. This demands several further requirements for visitors and visitor lists. Visitors computing further results have to provide access to their runtime variables in the same way as algorithm classes. Dependent visitors need a reference to their required visitors.

Dependent visitors generally need the intermediate results of their required visitor (stored in the runtime variables) *at the same working point* in the algorithm. This is possible, if the visit method of the required visitor is called before the visit method of the dependent visitor. The visitor lists are designed to invoke the visit methods of the stored visitors in the order the visitors have been added to the visitor list. So the developer using dependent visitors has to ensure that the visitors are registered with the algorithm object in correct order. The documentation of a dependent visitor has to explicitly indicate this.

The implementation of visitor lists

Algorithm classes generally only contain exactly one reference to a visitor list. However, visitor lists are designed to support not only the visitors of the own visitor interface, but also all visitors of super interfaces. In order to achieve this, visitor lists have an analogous inheritance tree in comparison to the visitor interfaces. Every visitor list contains a `List` of visitors. This `List` only contains the visitors implementing the same visitor interface (including derived visitors). E.g., in case of `SearchVisitor`, this also includes `DFSVisitor` but not `GraphVisitor`. When adding visitors to a visitor list, the type of the visitor is checked. If it implements the same interface, it is added to the own `List`. It is always added to the `List` of the visitor list's super class. The visit methods only consider the `List` of visitors that is present on the same level as the visit method is declared. E.g., The method `visitTreeEdge(...)` from `SearchVisitor` iterates over the `List` of `SearchVisitor`s in the class `DFSVisitorList`, which can also include `DFSVisitor`s. The method `leaveTreeEdge(...)` in `DFSVisitor` iterates over the `List` of `DFSVisitor`s in the class `DFSVisitorList`. These multiple `List`s always ensure the relative order of the visitors at each working point.

The consequence of doing it that way is having multiple references to the same visitor. The alternative would be having only one `List` of visitors at the top level of the inheritance tree. This would reflect the absolute order of the visitors and only require to have one reference per visitor. The disadvantage of this approach is the necessity of performing the type checks (`instanceof`) at every call of the visit method.

Both variants have been tried and experiments have shown, that the variant with multiple `List`s have performed significantly better because the type checks are only necessary when adding the visitors to the `List`.

5.3.4 The implementation of algorithm classes

This section shows how algorithm classes are implemented in *Algolib*. First a general concept is introduced, accompanied by the description of two generic classes. The first example for actual implementations, including the handling of visitors, are the search algorithms. Finally two examples are presented that implement graph algorithms in using other algorithms.

5.3.4.1 Algorithm states

At runtime, an algorithm object can be in several states. An overview of these states can be found in figure 5.5.

The method names shown in this diagram refer to methods that are present in every algorithm class. They are explained in section 5.3.4.3.

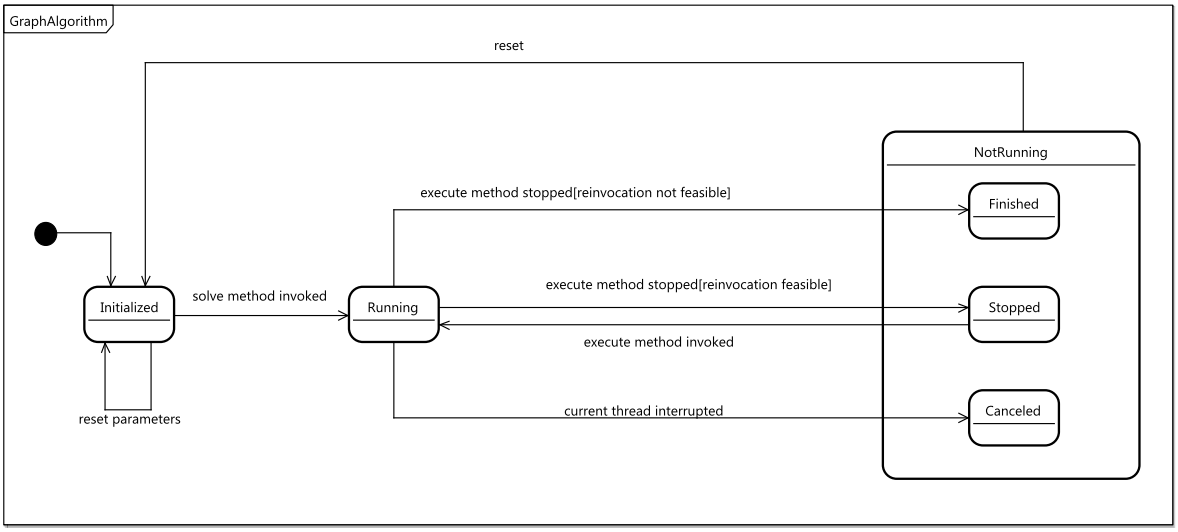


Figure 5.5: The possible states of algorithm objects

The possible states are realized using a Java `enum`. Listing 5.1 shows this Java `enum`. The description below refers to the names in the `enum`.

Listing 5.1: The enum `AlgorithmState`

```
1 public enum AlgorithmState {
2     INITIALIZED, RUNNING, STOPPED, FINISHED,CANCELED;
3 }
```

After an algorithm object is created, it is in the state `INITIALIZED`. In this state a reference to the graph has been created and all runtime variables and durable parameters have been initialized. After changing durable parameters, the object remains in this state. This is the only state that allows changes to durable parameters.

When invoking the execute method, the algorithm object changes into the state `RUNNING`.

After the termination of the execute method, the object either changes into the state `STOPPED` or `FINISHED`. The behavior depends on the type of algorithm. If a re-invocation of the execute method is feasible with the current runtime variables and their values, the algorithm should change into the state `STOPPED`. If a re-invocation is not feasible, meaning no possible re-invocation would contribute to the result, the algorithm should change into the state `FINISHED`. An example for such an algorithm is any search algorithm that searches in the reachable subgraph of a given start vertex. If, however, the algorithm is not re-usable, it should directly change into the state `FINISHED`. While being in one of these states, the results can be obtained from the algorithm object.

If the algorithm was terminated *from outside* (the current thread was interrupted), changes into the state `CANCELED`. In this state no results can be obtained. Details about how to terminate algorithms from outside can be found in section 5.3.4.3 on page 91.

5.3.4.2 Additional results

As already mentioned in section 3.2.3 on page 33, an algorithm can have multiple results. Results that are not specified in the problem interfaces are called *additional results*. Most additional results are not required for successfully executing an algorithm. Those results are called *optional results*.

In *Algolib* optional results are not computed by default. They have to be explicitly activated and they can be deactivated. All additional results can be retrieved in the same way and in the same state as regular results. If an optional result, that has not been computed, is requested after the algorithm is done, `null` is returned by convention.

5.3.4.3 The class `GraphAlgorithm`

The class `GraphAlgorithm` unites all common features of all graph algorithms in *Algolib*. It implements the interface `ProblemSolver`. Listings 5.2-5.10 show the sourcecode of this class.

Listing 5.2 shows the member variables of `GraphAlgorithm`.

Listing 5.2: Member variables

```
1 public abstract class GraphAlgorithm implements ProblemSolver {
2     protected Graph graph;
3     protected BooleanFunction<GraphElement> subgraph;
4     private int vertexCount;
5     private int edgeCount;
6     protected AlgorithmState state;
```

`graph` and `subgraph` describe the graph the algorithm works on. If `subgraph` is set to `null`, the algorithm will work on the whole graph. `vertexCount` and `edgeCount` contain the graph element count with respect to the function subgraph. They are lazily computed and their default values is `-1` for indicating that the values have not been computed yet.

Listing 5.3 shows the constructors of `GraphAlgorithm`.

Listing 5.3: Constructors

```
7     public GraphAlgorithm(Graph graph) {
8         this.graph = graph;
9         this.state = AlgorithmState.INITIALIZED;
10        resetParameters();
11        reset();
12    }
13
14    public GraphAlgorithm(Graph graph, BooleanFunction<GraphElement>
          subgraph) {
15        this(graph);
16        this.subgraph = subgraph;
17    }
```

Please note that `resetParameters()` and `reset()` are called in the constructor. These methods are described below.

Listing 5.4 shows the getter and setter methods for the graph and the subgraph.

Listing 5.4: Changing the graph

```
18    @Override
19    public void setGraph(Graph graph) {
20        checkStateForSettingParameters();
21        this.graph = graph;
22        vertexCount = -1;
23        edgeCount = -1;
24    }
25
26    @Override
27    public void setSubgraph(BooleanFunction<GraphElement> subgraph) {
28        checkStateForSettingParameters();
29        this.subgraph = subgraph;
30        vertexCount = -1;
31        edgeCount = -1;
32    }
33
34    public Graph getGraph() {
35        return graph;
36    }
37
38    public BooleanFunction<GraphElement> getSubgraph() {
39        return subgraph;
40    }
```

The methods `setGraph` and `setSubgraph` are inherited from the interface `ProblemSolver`. The method `checkStateForSettingParameters()` ensures that the graph can only be changed if the algorithm object is in the state `INITIALIZED` (see below). If the graph changes, the graph element count is set back to `-1` because the old element count becomes invalid.

Listing 5.5 shows the methods for obtaining derived graph properties in the context of the graph algorithm.

Listing 5.5: Graph properties

```
41    public int getVertexCount() {
42        if (vertexCount < 0) {
43            if (subgraph == null) {
44                vertexCount = graph.getVCount();
45            } else {
46                vertexCount = 0;
47                for (Vertex currentVertex : graph.vertices()) {
48                    if (subgraph.get(currentVertex)) {
49                        vertexCount++;
50                    }
51                }
52            }
53        }
54        return vertexCount;
55    }
56
57    public int getEdgeCount() {
58        if (edgeCount < 0) {
59            if (subgraph == null) {
60                edgeCount = graph.getECount();
61            } else {
62                edgeCount = 0;
63                for (Edge currentEdge : graph.edges()) {
64                    if (subgraph.get(currentEdge)) {
65                        edgeCount++;
66                    }
67                }
68            }
69        }
70        return edgeCount;
71    }
72
73    public abstract boolean isDirected();
74
75    public abstract boolean isHybrid();
```

The methods `getEdgeCount()` and `getVertexCount()` lazily compute the graph element number with respect to the function `subgraph`. The methods `isDirected()` and `isHybrid()` handle the interpretation of the graph. In JGraLab all graphs are directed. However, they can be interpreted as undirected graphs. The method `isDirected()` tells if the algorithm object currently interprets the graph as directed graph. This highly depends on the algorithm. Some algorithms even allow both. The method `isHybrid()` tells if the algorithm object's interpretation of the graph can be changed. If an algorithm supports this, its algorithm class defines how the interpretation can be changed.

Listing 5.6 shows the most generic reset methods of graph algorithms.

Listing 5.6: Reset methods

```
76    public synchronized AlgorithmState getState() {
77        return state;
78    }
79
80    public void reset() {
81        if (getState() != AlgorithmState.RUNNING) {
82            this.state = AlgorithmState.INITIALIZED;
83        } else {
84            throw new IllegalStateException(
85                    "The algorithm may not be reseted while it is
                        running.");
86        }
87    }
88
89    public void resetParameters() {
90        checkStateForSettingParameters();
91        this.subgraph = null;
92        vertexCount = -1;
93        edgeCount = -1;
94        disableOptionalResults();
95    }
96
97    public abstract void disableOptionalResults();
```

The method `getState()` simply returns the algorithm state. The method `reset()` reinitializes all runtime variables and changes the algorithm object's state to `INITIALIZED`. This may only be done if the algorithm object is not in the state `RUNNING`. The method `resetParameters()` assigns each durable parameter its default value. The method `disableOptionalResults()` deactivates the computation of optional results if the algorithm class provides some. Optional results are activated through special setter classes. Examples can be found in section 5.3.4.5.

Listing 5.7 shows methods that check the algorithm state and throw an exception if the algorithm is in the wrong state.

Listing 5.7: State checks

```
98  public void checkStateForResult() {
99      if (state != AlgorithmState.FINISHED
100             && state != AlgorithmState.STOPPED) {
101         throw new IllegalStateException(
102                 "The result cannot be obtained while in this state: "
103                         + state);
104     }
105 }
106
107 public void checkStateForSettingParameters() {
108     if (getState() != AlgorithmState.INITIALIZED) {
109         throw new IllegalStateException(
110                 "Parameters may not be changed while in state " +
                        state);
111     }
112 }
113
114 public void checkStateForSettingVisitors() {
115     if (getState() == AlgorithmState.RUNNING
116             || getState() == AlgorithmState.CANCELED) {
117         throw new IllegalStateException(
118                 "Parameters may not be changed while in state " +
                        state);
119     }
120 }
```

The method `checkStateForResults()` is called whenever a result is obtained. Results may only be obtained when the algorithm is terminated or interrupted (`FINISHED` or `STOPPED`). The method `checkStateForSettingParameters()` is called whenever a parameter is changed. Parameters may only be changed if the algorithm object is in the state `INITIALIZED`. The method `checkStateForSettingVisitors()` is called whenever a visitor is added or removed. Visitors may only be altered if the algorithm object is not in the state `RUNNING` or `CANCELED`.

Listing 5.8 shows methods called by the execute method for controlling the algorithm object's state.

Listing 5.8: State control methods

```
121     protected void startRunning() {
122         if (state == AlgorithmState.INITIALIZED
123                 || state == AlgorithmState.STOPPED) {
124             state = AlgorithmState.RUNNING;
125         } else {
126             throw new IllegalStateException(
127                     "The␣algorithm␣cannot␣be␣started,␣when␣in␣state␣" +
                        state);
128         }
129     }
130
131     protected abstract void done();
```

The method `startRunning()` is called when the execute method is invoked. It ensures that it is only invoked if the algorithm is either in the state `INITIALIZED` or `STOPPED` and sets it to `RUNNING`. The method `done()` is called when the execute method is finished. This method sets the state of the algorithm object either to `STOPPED` or `FINISHED`. It depends on the algorithm which state is the correct one.

Listing 5.9 shows methods for registering and unregistering visitors. The method

Listing 5.9: Methods for handling visitors

```
132     public abstract void addVisitor(Visitor visitor);
133
134     public abstract void removeVisitor(Visitor visitor);
```

`addVisitor(...)` registers a visitor with the algorithm object and adds it to its visitor list. This method throws an exception if the wrong type of visitor is passed. The method `removeVisitor(...)` removes a registered visitor from the algorithm object's visitor list.

Early termination for algorithms

Listing 5.10 shows methods for handling early algorithm termination.

Listing 5.10: Terminating algorithms

```
135     public void terminate() throws AlgorithmTerminatedException {
136         if (getState() == AlgorithmState.RUNNING) {
137             done();
138             throw new AlgorithmTerminatedException("Terminated␣by␣
                    algorithm.");
139         } else {
140             throw new IllegalStateException(
141                     "The␣algorithm␣may␣only␣be␣terminated,␣when␣in␣state␣"
142                             + AlgorithmState.RUNNING);
143         }
144     }
145
146     protected synchronized void cancelIfInterrupted() throws
            AlgorithmTerminatedException {
147         if (Thread.interrupted()) {
148             state = AlgorithmState.CANCELED;
149             Thread.currentThread().interrupt();
150             throw new AlgorithmTerminatedException("Thread␣interrupted.");
151         }
152     }
153 }
```

The method `terminate()` terminates the algorithm from inside. It is generally called by visitors and can also be called by the algorithm itself for indicating an early termination. The `AlgorithmTerminatedException` is used for aborting the execution of the algorithm. The method `cancelIfInterrupted()` indirectly terminates the algorithm from outside. It is called multiple times by each algorithm that supports external termination. This mechanism allows users to terminate an algorithm by interrupting the Thread the algorithm runs in. A supporting algorithm has to ensure that this method is called periodically. Currently all algorithms in *Algolib* support this feature.

Both termination mechanisms use the same type of exception if the algorithm terminates. Only the state of the algorithm object allows users to distinguish between the two termination variants. The external termination always sets the algorithm object in the state `CANCELED`. The internal termination invokes the method `done()` (see below) for setting the correct state, which is either `STOPPED` or `FINISHED`.

5.3.4.4 The class `StructureOrientedAlgorithm`

The class `StructureOrientedAlgorithm` unites all common features of algorithms relying on the graph structure (they follow the edges during their run). Listings 5.11-5.15 show the source code of this class. Currently all algorithms in *Algolib* rely on the graph structure and are derived from this class. However, there are currently unimplemented graph algorithms that do not rely on it. For allowing such algorithms to be added to *Algolib* in the future, the class `StructureOrientedAlgorithm` is separated from the class `GraphAlgorithm`.

Listing 5.11 shows the member variables of `StructureOrientedAlgorithm`.

Listing 5.11: Member variables

```
1  public abstract class StructureOrientedAlgorithm extends GraphAlgorithm
       implements
2          TraversalSolver {
3      public static final EdgeDirection DEFAULT_TRAVERSAL_DIRECTION =
           EdgeDirection.OUT;
4      protected BooleanFunction<Edge> navigable;
5      protected EdgeDirection traversalDirection;
```

`navigable` is a function that decides if an edge is navigable independently from the function `subgraph`. `traversalDirection` indicates the direction edges are traversed. The default value can be found in the constant `DEFAULT_TRAVERSAL_DIRECTION` and is set to `OUT`. The Java enum `EdgeDirection` has been described in section 3.1.3 on page 31.

Listing 5.12 shows the constructors of `StructureOrientedAlgorithm`.

Listing 5.12: Constructors

```
6      public StructureOrientedAlgorithm(Graph graph) {
7          this(graph, null, null);
8      }
9
10     public StructureOrientedAlgorithm(Graph graph,
11             BooleanFunction<GraphElement> subgraph,
12             BooleanFunction<Edge> navigable) {
13         super(graph, subgraph);
14         this.navigable = navigable;
15     }
```

In *Algolib* all graph algorithms provide two constructors. The first one only provides the graph as durable parameter. The second one provides all durable parameters.

Listing 5.13 shows the getter and setter methods for the function `navigable` and for `traversalDirection`.

Listing 5.13: Getters and setters

```
16      @Override
17      public void setNavigable(BooleanFunction<Edge> navigable) {
18          checkStateForSettingParameters();
19          this.navigable = navigable;
20      }
21
22      public void setTraversalDirection(EdgeDirection traversalDirection) {
23          checkStateForSettingParameters();
24          if (!isHybrid()) {
25              if (isDirected() && traversalDirection == EdgeDirection.INOUT)
                    {
26                  throw new UnsupportedOperationException(
27                          "This␣algorithm␣does␣not␣support␣undirected␣
                               graphs.");
28              } else if (!isDirected() && traversalDirection !=
                    EdgeDirection.INOUT) {
29                  throw new UnsupportedOperationException(
30                          "This␣algorithm␣does␣not␣support␣directed␣
                               graphs.");
31              }
32          }
33          this.traversalDirection = traversalDirection;
34      }
35
36      public EdgeDirection getTraversalDirection() {
37          return traversalDirection;
38      }
39
40      public BooleanFunction<Edge> getNavigable() {
41          return navigable;
42      }
```

`setNavigable` changes the function `navigable`. It uses the method `checkStateForSettingParameters()` for ensuring that the algorithm object is in the correct state. `setTraversalDirection` changes the direction the edges are traversed. In the case of algorithms that rely on the graph structure, this value also decides if the current graph is interpreted as directed or undirected graph. The meaning of the possible values was already introduced in section 3.1.3 on page 31. If a graph algorithm is only capable to work on either directed or undirected graphs (`!isHybrid()`), the method throws an exception if the graph type would be changed. The remaining two methods are simply getter methods.

Listing 5.14 shows the reset methods for `StructureOrientedAlgorithm`.

Listing 5.14: Reset methods

```
43      @Override
44      public void resetParameters() {
45          super.resetParameters();
46          this.navigable = null;
47          this.searchDirection = DEFAULT_SEARCH_DIRECTION;
48      }
```

Here only `resetParameters()` is overridden, because `StructureOrientedAlgorithm` does not introduce new runtime variables. After calling the method `resetParameters()` from `GraphAlgorithm`, it sets the two durable parameters to their default values.

Listing 5.15 shows some convenience methods for setting the parameter `traversalDirection`.

Listing 5.15: Convenience methods

```
49      public StructureOrientedAlgorithm normal() {
50          setTraversalDirection(EdgeDirection.OUT);
51          return this;
52      }
53
54      public StructureOrientedAlgorithm reversed() {
55          setTraversalDirection(EdgeDirection.IN);
56          return this;
57      }
58
59      public StructureOrientedAlgorithm undirected() {
60          setTraversalDirection(EdgeDirection.INOUT);
61          return this;
62      }
63
64      @Override
65      public boolean isDirected() {
66          return searchDirection != EdgeDirection.INOUT;
67      }
68
69  }
```

`normal()` sets `traversalDirection` to `OUT`. `reversed()` sets `traversalDirection` to `IN`. `undirected()` sets `traversalDirection` to `INOUT`. All three methods return `this` for convenience. E.g., the algorithm class `BreadthFirstSearch` can be instantiated with:

```
BreadthFirstSearch bfs = new BreadthFirstSearch(graph).undirected();
```

This creates a new algorithm object that operates on undirected graphs in one line.

5.3.4.5 Example: Search algorithms

This section gives an example on how concrete algorithms are implemented in *Algolib*. For this purpose the implementation of search algorithms is shown. As seen in chapter 3, search algorithms share some parameters and results. They also share some of the runtime variables. In *Algolib* this is exploited for creating an abstract class defining all common features of search algorithms. This class is called `SearchAlgorithm`. Listings 5.16-5.21 show the source code of this class.

Listing 5.16 shows the member variables of `SearchAlgorithm`.

Listing 5.16: Member variables

```
1  public abstract class SearchAlgorithm extends StructureOrientedAlgorithm
       implements
2          TraversalFromVertexSolver, CompleteTraversalSolver {
3      protected Vertex[] vertexOrder;
4      protected Edge[] edgeOrder;
5
6      protected BooleanFunction<Vertex> visitedVertices;
7      protected BooleanFunction<Edge> visitedEdges;
8      protected int num;
9      protected int eNum;
10
11     protected IntFunction<Vertex> level;
12     protected IntFunction<Vertex> number;
13     protected IntFunction<Edge> enumber;
14     protected Function<Vertex, Edge> parent;
```

`vertexOrder` and `edgeOrder` are internal representations of the results. `visitedVertices`, `visitedEdges`, `num` and `eNum` are runtime variables, where the first two are functions for marking which graph elements have already been visited. `num` and `eNum` are counters that are required for the computation of the results. `level`, `number`, `enumber` and `parent` are optional results.

The constructors are left out here, because they have the same signatures as the constructors of `StructureOrientedAlgorithm` and only delegate to them.

Listing 5.17 shows the methods for handling the optional results.

For each optional parameter there is a method with prefix `with` for activating its computation and a method with prefix `without` for deactivating its computation. They follow the same convenience schema as the methods `normal()`, `reversed()` and `undirected()`. Optional results are only computed if their reference is not `null`. Additionally there is the method

Listing 5.17: Methods for optional results

```
15    public SearchAlgorithm withLevel() {
16        checkStateForSettingParameters();
17        level = new IntegerVertexMarker(graph);
18        return this;
19    }
20
21    public SearchAlgorithm withoutLevel() {
22        checkStateForSettingParameters();
23        level = null;
24        return this;
25    }
26    // analogously for number, enumber, and parent
27
28    @Override
29    public void disableOptionalResults() {
30        checkStateForSettingParameters();
31        level = null;
32        number = null;
33        enumber = null;
34        parent = null;
35    }
```

`disableOptionalResults()` which can be used for deactivating the computation of all optional results. This method is required to enable the deactivation of optional results in the method `resetParameters()` in the class `GraphAlgorithm` (see listing 5.6 on page 88).

Listing 5.18 shows the reset methods for `SearchAlgorithm`.

Listing 5.18: Reset methods

```
36    @Override
37    public void reset() {
38        super.reset();
39        vertexOrder = new Vertex[getVertexCount() + 1];
40        edgeOrder = new Edge[getEdgeCount() + 1];
41
42        visitedVertices = new BitSetVertexMarker(graph);
43        visitedEdges = new BitSetEdgeMarker(graph);
44        num = 1;
45        eNum = 1;
46
47        level = level == null ? null : new IntegerVertexMarker(graph);
48        number = number == null ? null : new IntegerVertexMarker(graph);
49        enumber = enumber == null ? null : new IntegerEdgeMarker(graph);
50        parent = parent == null ? null : new
              ArrayVertexMarker<Edge>(graph);
51    }
```

Here only the method `reset()` is overridden, because there are no new parameters in this class. The arrays `vertexOrder` and `edgeOrder` are initialized with the correct length, with respect to the subgraph. In *Algolib*, by convention, the index *0* is not used, so the arrays have to be one element longer than the graph element count. This happens in analogy to the indexing in JGraLab. The runtime variables are re-instantiated or set to their start value respectively. The variables for optional results are only re-instantiated if they should be computed.

Listing 5.19 contains methods for retrieving intermediate results. These are mainly required by visitors. By convention they provide the internal representation of the result without performing any state checks. This means, intermediate results can always be obtained.

Listing 5.19: Methods for retrieving intermediate results

```
52  public Vertex[] getInternalVertexOrder() {
53      return vertexOrder;
54  }
55  // analogously for edgeOrder, level, number, enumber, and parent
56
57  public BooleanFunction<Vertex> getVisitedVertices() {
58      return visitedVertices;
59  }
60  // analogously for visitedEdges, num and eNum
```

For (optional) results, these getter methods contain the infix `Internal` for distinguishing them from the getter methods for the results (see listing 5.21). These methods have to be public, because visitors are not necessarily in the same package as the algorithm class.

Listing 5.20 contains several methods concerning the algorithm itself.

Listing 5.20: Algorithm related methods

```
61  @Override
62  public boolean isHybrid() {
63      return true;
64  }
65
66  @Override
67  public abstract SearchAlgorithm execute(Vertex root) throws
        AlgorithmTerminatedException;
68
69  @Override
70  public SearchAlgorithm execute() throws AlgorithmTerminatedException {
71      for (Vertex currentRoot : graph.vertices()) {
72          execute(currentRoot);
73          if (state == AlgorithmState.FINISHED) {
74              break;
75          }
```

```
76          }
77          assert (state == AlgorithmState.FINISHED);
78          return this;
79      }
80
81      @Override
82      protected void done() {
83          if (state != AlgorithmState.CANCELED) {
84              state = num < getVertexCount() + 1 ? AlgorithmState.STOPPED
85                      : AlgorithmState.FINISHED;
86          }
87      }
```

`isHybrid()` always returns `true`, because all search algorithms in *Algolib* are implemented to work on directed and undirected graphs. `execute(Vertex root)` is inherited from the interface `TraversalFromVertexSolver` and implements the actual search algorithm that searches the reachable subgraph from the given vertex `root`. It is implemented in the concrete classes for search algorithms. `execute()` implements the execute method of the problem interface `CompleteTraversal`. It iterates over all vertices in the order the vertices are stored in the graph. For each vertex it invokes the execute method of the actual search algorithm implementation. `done()` sets the algorithm state after one run of the actual search algorithm implementation is done according to the number of vertices that have been visited. If all vertices in the subgraph have been visited, the state is set to `FINISHED`

Listing 5.21 shows the methods for retrieving the results of the search algorithm.

Listing 5.21: Results

```
89      @Override
90      public Permutation<Vertex> getVertexOrder() {
91          checkStateForResult();
92          return new ArrayPermutation<Vertex>(vertexOrder);
93      }
94      // analogously for edgeOrder
95
96      public IntFunction<Vertex> getLevel() {
97          checkStateForResult();
98          return level;
99      }
100     // analogously for number, enumber, and parent
101 }
```

These methods return the results in the desired output format. In the case of search algorithms, the two results `vertexOrder` and `edgeOrder` are each wrapped in an instance of `ArrayPermutation`, which is a convenience class that implements the interface

`Permutation` for arrays. If for some reason the actual array is needed, the methods for the internal representation can still be used. If the internal representation of a result is identical to the desired output format, which is true for `level`, `number`, and `parent`, the getter methods for the results are identical to the getter methods for retrieving their internal representation, except for the state check. This is done for achieving a high level of analogy. In doing this, all internal representations are treated in the same way and all results are treated in the same way.

The class `BreadthFirstSearch`

This section shows the implementation of breadth first search using the class `SearchAlgorithm` described above. Listings 5.22-5.26 show extracts from the source code of the class `BreadthFirstSearch`.

Listing 5.22 shows the member variables.

Listing 5.22: Member variables

```
1  public class BreadthFirstSearch extends SearchAlgorithm implements
2          TraversalFromVertexSolver {
3
4      private SearchVisitorList visitors;
5      private int firstV;
```

`visitors` is the visitor list that stores all visitors for this BFS algorithm. `firstV` is the runtime variable that is required to turn the array `vertexOrder` into a queue (see section 3.3.1.6 on page 38 for details). Since `firstV` is a runtime variable, it also has a getter (not shown here), so visitors can access its value.

The constructors are left out here, because they have the same signatures as the constructors of `SearchAlgorithm` and only delegate to them.

The convenience methods for deciding the graph interpretation and the handling of optional results have to get the correct signature in this class. As an example the method `withLevel()` is shown in listing 5.23.

Listing 5.23: Convenience methods

```
7      @Override
8      public BreadthFirstSearch withLevel() {
9          super.withLevel();
10         return this;
11     }
12     // analogously for the other convenience methods
```

The return type is changed to `BreadthFirstSearch`, so users can still operate on an instance of `BreadthFirstSearch` instead of `SearchAlgorithm`. After the method `withLevel()`

from `SearchAlgorithm` is called, the correct `this` reference is returned. The other convenience methods have to be redefined analogously.

Listing 5.24 shows the reset methods.

Listing 5.24: Reset methods

```
13    @Override
14    public void reset() {
15        super.reset();
16        firstV = 1;
17        visitors.reset();
18    }
19
20    @Override
21    public void resetParameters() {
22        super.resetParameters();
23        visitors = new SearchVisitorList();
24    }
```

`reset()` sets the runtime variable `firstV` to its initial value and calls the method `reset()` for all visitors. `resetParameters()` re-initializes the visitor list.

Listing 5.25 shows how visitors can be registered with and removed from a BFS algorithm.

Listing 5.25: Registering and removing visitors

```
25    @Override
26    public void addVisitor(Visitor visitor) {
27        checkStateForSettingVisitors();
28        visitor.setAlgorithm(this);
29        visitors.addVisitor(visitor);
30    }
31
32    @Override
33    public void removeVisitor(Visitor visitor) {
34        checkStateForSettingVisitors();
35        visitors.removeVisitor(visitor);
36    }
```

`addVisitor(...)` registers the given visitor with this algorithm. First, this algorithm object is set as algorithm for the visitor. Then the visitor is added to the visitor list. Please note that a type check is performed in the method `setAlgorithm(...)` in the given visitor. Only the visitor can decide if this algorithm object is compatible. `removeVisitor(...)` simply removes the given visitor from the visitor list.

Listing 5.26 finally shows the actual implementation of the breadth first search. In addition to the algorithm draft from section 3.3.1.6 on page 3.3.1.6, several implementation specific modifications can be found.

Listing 5.26: The execute method

```
37  @Override
38  public BreadthFirstSearch execute(Vertex root) throws
        AlgorithmTerminatedException {
39      if (subgraph != null && !subgraph.get(root)
40              || visitedVertices.get(root)) {
41          return this;
42      }
43      startRunning();
44      vertexOrder[num] = root;
45
46      if (level != null) {
47          level.set(root, 0);
48      }
49      visitors.visitRoot(root);
50
51      if (number != null) {
52          number.set(root, num);
53      }
54      visitors.visitVertex(root);
55
56      visitedVertices.set(root, true);
57      num++;
58      while (firstV < num && vertexOrder[firstV] != null) {
59          Vertex currentVertex = vertexOrder[firstV++]; // pop
60          for (Edge currentEdge :
                currentVertex.incidences(traversalDirection)) {
61              cancelIfInterrupted();
62              if (visitedEdges.get(currentEdge) || subgraph != null
63                      && !subgraph.get(currentEdge) || navigable != null
64                      && !navigable.get(currentEdge)) {
65                  continue;
66              }
67              Vertex nextVertex = currentEdge.getThat();
68              assert (subgraph == null || subgraph.get(nextVertex));
69
70              edgeOrder[eNum] = currentEdge;
71              if(enumber != null){
72                  enumber.set(currentEdge, eNum);
73              }
74              visitors.visitEdge(currentEdge);
```

```
75                  visitedEdges.set(currentEdge, true);
76                  eNum++;
77
78                  if (visitedVertices.get(nextVertex)) {
79                      visitors.visitFrond(currentEdge);
80                  } else {
81                      visitors.visitTreeEdge(currentEdge);
82                      vertexOrder[num] = nextVertex;
83                      if (level != null) {
84                          level.set(nextVertex, level.get(currentVertex) +
                                1);
85                      }
86                      if (parent != null) {
87                          parent.set(currentEdge.getThat(), currentEdge);
88                      }
89                      if (number != null) {
90                          number.set(nextVertex, num);
91                      }
92                      visitors.visitVertex(nextVertex);
93                      visitedVertices.set(nextVertex, true);
94                      num++;
95                  }
96              }
97          }
98          done();
99          return this;
100     }
101 }
```

Lines 39-42 ensure that the algorithm is only executed, if the given vertex is part of the subgraph and has not been visited yet. This is especially useful for the method `execute()` from `SearchAlgorithm` that implements the complete traversal. Line 43 sets the algorithm object's state to `RUNNING`. The computation of optional results is controlled using *null checks* (E.g., line 46). They look ugly and inflate the code tremendously, but experiments have shown that they are very efficient because they practically have no significant overhead. The calls to the visit methods (e.g., line 49) are done at the working points. Please note that the computation of (optional) results always occur at the working points and that visitor calls always happen *after* the computation of (optional) results. This is done, because most visitors rely on intermediate results at a specific working point.

The `for` loop, starting at line 60, iterates over all incident edges filtered by `traversalDirection`. In line 61 the method `cancelIfInterrupted` is called for aborting the algorithm if the current Thread has been interrupted. Lines 62-66 skip the current edge,

if it has already been visited (this can only occur with undirected graphs), if it is not part of the subgraph or if it is not navigable.

The rest of the algorithm is in analogy to the draft from chapter 3. The execute method for complete traversal is redefined in the same way as the convenience methods are.

The implementation of depth first search

Algolib provides two implementations of the depth first search. The common variables and methods can be found in the class `DepthFirstSearch`, which is derived from `SearchAlgorithm`. Instead of a `SearchVisitorList`, a `DFSVisitorList` is used. It declares the runtime variables `rNum` and `rnumber` for computing the additional result `rnumber`. The functions `rnumber` and `number` are mandatory for DFS, because they are needed for distinguishing between the different frond types. Since `SearchAlgorithm` declares `number` as optional result, `DepthFirstSearch` has to override this. For doing this, the method `withoutNumber()` throw an exception. `DepthFirstSearch` introduces a new optional result `rorder`, which is the inverse function of `rnumber`.

The only method that is not implemented in `DepthFirstSearch`, is `execute(Vertex root)`. The first actual implementation of DFS can be found in the class `RecursiveDFS`. It implements the DFS in analogy to the draft from section 3.3.1.7 on page 40. The implementation specific modifications for subgraph handling, navigability, and optional results are in analogy to the implementation of BFS.

For large graphs the recursive DFS can cause a `StackOverflowError` due to an overflow of the JVM's implicit call stack. For avoiding this problem, *Algolib* provides an alternative implementation for DFS that behaves exactly the same way as the recursive implementation, but is implemented iteratively. Its class is called `IterativeDepthFirstSearch`. Instead of using the implicit call stack, it uses an explicit stack of vertices and a graph marker that assigns each vertex the information about which incident edges have not been traversed yet[1]. For implementing the working point `leaveTreeEdge`, the information from the function `parent` is used, which makes this additional result mandatory for the iterative implementation. The latter is the only disadvantage of this implementation, because more memory is required. A user of *Algolib* can decide which implementation to use.

It is also possible to try running the algorithm with the recursive implementation first and catch the `StackOverflowError`. If it occurs, the search can be restarted with the iterative implementation.

[1] the explicit iterators for incident edges

5.3.5 The Implementation of algorithms using other algorithms

In *Algolib* it is possible to implement algorithms that depend on executing other algorithms. They behave like normal algorithms but delegate the actual work to the algorithm object of the algorithm they use. Before they execute the algorithm, they configure it according to their requirements.

5.3.5.1 Example: `ShortestPathsWithBFS`

The easiest example in *Algolib* is the algorithm class `ShortestPathsWithBFS`. It solves the problems *shortest paths from vertex* and *distance from vertex*. It uses breadth first search with computing the optional results `parent` and `level` (see section 3.3.3 on page 48). Listings 5.27-5.29 show the source code of `ShortestPathsWithBFS`.

Listing 5.27 illustrates how the reference to the algorithm object of `BreadthFirstSearch` is handled.

Listing 5.27: Constructors

```
1  public class ShortestPathsWithBFS extends StructureOrientedAlgorithm
       implements
2          DistanceFromVertexSolver, ShortestPathsFromVertexSolver {
3
4      private BreadthFirstSearch bfs;
5
6      public ShortestPathsWithBFS(Graph graph, BreadthFirstSearch bfs) {
7          this(graph, null, bfs, null);
8      }
9
10     public ShortestPathsWithBFS(Graph graph,
11             BooleanFunction<GraphElement> subgraph, BreadthFirstSearch bfs,
12             BooleanFunction<Edge> navigable) {
13         super(graph, subgraph, navigable);
14         this.bfs = bfs;
15     }
```

`bfs` is a member variable of `ShortestPathsWithBFS` and has to be provided at initialization. This requires the algorithm object to be instantiated by the user. In doing it that way, the user has the option to configure the algorithm object before it is used in `ShortestPathsWithBFS` for solving another probably custom problem simultaneously (e.g., computing `number`). The remaining parameters are delegated to the super class as usual.

Listing 5.28 shows methods that are inherited from the abstract super classes.

Listing 5.28: Inherited methods

```
16    @Override
17    public void addVisitor(Visitor visitor) {
18        checkStateForSettingVisitors();
19        bfs.addVisitor(visitor);
20    }
21    // analogously for removeVisitor
22
23    @Override
24    public ShortestPathsWithBFS normal() {
25        super.normal();
26        return this;
27    }
28    // analogously for the other convenience methods
29
30    @Override
31    public boolean isHybrid() {
32        return true;
33    }
```

The handling of visitors is completely delegated to `bfs`. The other methods are implemented as usual.

Listing 5.29 shows the execute method.

Listing 5.29: The execute method

```
34    @Override
35    public ShortestPathsWithBFS execute(Vertex start) throws
          AlgorithmTerminatedException {
36        bfs.reset();
37        bfs.setGraph(graph);
38        bfs.setSubgraph(subgraph);
39        bfs.setNavigable(navigable);
40        bfs.setTraversalDirection(traversalDirection);
41        startRunning();
42        try {
43            bfs.withLevel().withParent().execute(start);
44        } catch (AlgorithmTerminatedException e) {
45        }
46        done();
47        return this;
48    }
49
50    @Override
```

```
51      protected void done() {
52          state = bfs.getState() == AlgorithmState.STOPPED ?
                AlgorithmState.FINISHED
53                  : bfs.getState();
54      }
```

Before `bfs` is executed, its `reset` method is called (line 36) and the parameters are copied (lines 37-40). When executing the algorithm, the optional results are activated using the convenience methods (line 43). The invocation of the algorithm happens inside of a try-catch block, for reacting correctly to early algorithm termination. The method `done()` sets the algorithm state according to the state of `bfs`. The only difference is, `ShortestPathsWithBFS` may enter `FINISHED`, even if `bfs` is only in the state `STOPPED`.

Listing 5.30 shows the methods for retrieving the results.

Listing 5.30: Results

```
55      @Override
56      public IntFunction<Vertex> getDistance() {
57          checkStateForResult();
58          return bfs.getLevel();
59      }
60      // analogously for parent
61  }
```

They are simply delegated to the corresponding results computed by `bfs`.

5.3.5.2 Adapting Visitors

In *Algolib* it is possible to create an algorithm using another algorithm and also use a different type of visitor. The algorithm class `TopologicalOrderWithDFS` computes a topological order using an instance of `DepthFirstSearch`. *Algolib* provides a visitor for visiting vertices in topological order called `TopologicalOrderVisitor`. This visitor however, is incompatible with `DepthFirstSearch`. In order to use `TopologicalOrderVisitor`s, a `DFSVisitor` has to be adapted. Listings 5.31-5.35 show the source code of `TopologicalOrderWithDFS`.

Listing 5.31 shows the member variables.

Listing 5.31: Member variables

```
1  public class TopologicalOrderWithDFS extends StructureOrientedAlgorithm
       implements
2          AcyclicitySolver, TopologicalOrderSolver {
3
4      private DepthFirstSearch dfs;
5      private DFSVisitorAdapter torderVisitorAdapter;
6      private boolean acyclic;
7      private TopologicalOrderVisitorList visitors;
```

`dfs` is the reference to the depth first search algorithm that is used, in analogy to `bfs` in section 5.3.5.1. `torderVisitorAdapter` is a reference to the visitor adapter that is declared and passed to `dfs` below. `acyclic` is a result of the algorithm. `visitors` is a visitor list for `TopologicalOrderVisitor` and works in analogy to the other visitor lists shown in section 5.3.3.2 on page 82. The constructors work in analogy to the constructors from `ShortestPathsWithBFS` from section 5.3.5.1 and are not shown here.

Listing 5.32 shows the handling of visitors.

Listing 5.32: Visitors

```
9      @Override
10     public void addVisitor(Visitor visitor) {
11         checkStateForSettingVisitors();
12         if (visitor instanceof TopologicalOrderVisitor) {
13             visitor.setAlgorithm(this);
14             visitors.addVisitor(visitor);
15         } else {
16             dfs.addVisitor(visitor);
17         }
18     }
19     // removeVisitor is defined analogously
```

Here the type of visitor is determined for deciding which algorithm object will handle it. Only if the given visitor is a `TopologicalOrderVisitor`, it is added to or removed from the visitor list of this algorithm object, otherwise it is delegated to `dfs`.

The methods `normal()` and `reversed()` are defined in analogy to the methods from `ShortestPathsWithBFS` (see listing 5.28 on page 105). The method `isHybrid()` returns `false`, because the problem is only defined for directed graphs.

Listing 5.33 shows the reset methods.

Listing 5.33: Reset methods

```
20      @Override
21      public void resetParameters() {
22          super.resetParameters();
23          visitors = new TopologicalOrderVisitorList();
24          torderVisitorAdapter = new DFSVisitorAdapter() {
25              @Override
26              public void visitBackwardArc(Edge e) {
27                  acyclic = false;
28                  dfs.terminate();
29              }
30
31              @Override
32              public void leaveVertex(Vertex v) {
33                  visitors.visitVertexInTopologicalOrder(v);
34              }
35          };
36          this.normal();
37      }
38
39      @Override
40      public void reset() {
41          super.reset();
42          acyclic = true;
43      }
```

The method `reset()` sets the result `acyclic` to `true`.

The method `resetParameters()` creates the visitor `torderVisitorAdapter` as anonymous class. It only implements two of the visit methods. In `visitBackwardArc(...)`, the existence of a cycle is detected, which causes the result `acyclic` to be set to `false` and terminates `dfs`. In `leaveVertex(...)`, the adaptation to `TopologicalOrderVisitor` happens. When searching in reversed orientation, the vertices are left in topological order, so the method `visitVertexInTopologicalOrder(...)` is called here.

After the declaration of the visitor, the orientation is set to *normal*, where the orientation of `dfs` is set to *reversed*.

Listing 5.34 shows the execute method.

Listing 5.34: The execute method

```
44    @Override
45    public TopologicalOrderWithDFS execute() throws
          AlgorithmTerminatedException {
46        dfs.reset();
47        dfs.setGraph(graph);
48        dfs.setSubgraph(subgraph);
49        dfs.setNavigable(navigable);
50        if(traversalDirection == EdgeDirection.OUT){ //normal
51            dfs.reversed();
52        } else { // reversed
53            dfs.normal();
54        }
55        dfs.addVisitor(torderVisitorAdapter);
56        startRunning();
57        try {
58            dfs.withRorder().execute();
59        } catch (AlgorithmTerminatedException e) {
60        }
61        done();
62        dfs.removeVisitor(torderVisitorAdapter);
63        return this;
64    }
65
66    @Override
67    protected void done() {
68        state = dfs.getState();
69    }
```

The execute method works in analogy to the execute method of `ShortestPathsWithBFS`. However, there are two differences. The algorithm operates the depth first search in reverse orientation. Before the execution of the algorithm, the orientation is adapted accordingly to the desired orientation. For providing the working point `visitVertexInTopologicalOrder`, the visitor `torderVisitorAdapter` is added to `dfs` before its execution. After the execution it is removed from `dfs`. This is one reason, why visitors can be altered in the states `STOPPED` and `FINISHED`.

After the execution, the method `done()` copies the state from `dfs`. This is correct, because `dfs` searches in the whole graph.

Listing 5.35 shows the getter methods for the results.

Listing 5.35: The getter methods for the results

```
70      @Override
71      public Permutation<Vertex> getTopologicalOrder() {
72          checkStateForResult();
73          return dfs.getRorder();
74      }
75
76      @Override
77      public boolean isAcyclic() {
78          checkStateForResult();
79          return acyclic;
80      }
81  }
```

6 Using *Algolib*

Before the implementation of *Algolib* is described, this chapter gives some examples on how graph algorithms, provided by *Algolib*, are used. The purpose is to motivate JGraLab users to use *Algolib*'s algorithms instead of implementing a domain specific variant of an algorithm.

6.1 Routing in JGStreetMap

Before *Algolib* was introduced, graph algorithms had to be implemented for specific domains. One of these domains is the project *JGStreetMap* ([Zie]). This project uses the data provided by the open street map project[1](OSM) for routing. OSM is a software for demonstrating a practical use of JGraLab.

6.1.1 The OSM project

The open street map project (OSM project) is a project who's goal is generating a free map of the world for everyone. Anyone, owning a GPS device, can contribute to the project by adding new data to the map. The map itself is stored in a database. It is possible to export parts from this database in several formats, including xml. The exported data can be used for several purposes, including rendering a map. E.g., the OSM project itself uses Mapnik[2] for rendering the maps shown on their websites.

6.1.1.1 OSM graphs in JGraLab

JGStreetMap can import the OSM xml files and transform them to TGraphs. The imported graph contains the nodes and relations that are declared in the xml file. Figure 6.1 shows a part of the OSM schema. It only contains the classes that are needed for importing the graph from the xml file.

The data in the OSM database mainly consists of `Node`s. The attributes of the graph element classes are taken directly from the OSM data model. The most important part of OSM's data

[1] `http://www.openstreetmap.org` (December 2010)
[2] `http://mapnik.org/` (December 2010)

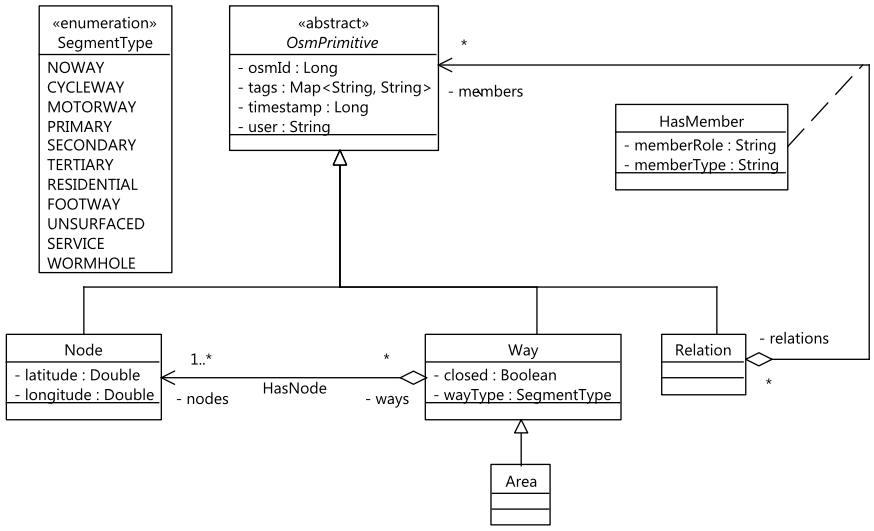


Figure 6.1: Part of the OSM schema for the imported data

model is the map of `tags` in `OsmPrimitive`. Here, all relevant data for classifying the elements of the map can be found.

Streets are modeled by vertices of the type `Way`. Since the streets are modeled by vertices instead of edges, a computation of shortest paths with standard algorithms is not possible using this model. So the OSM schema has been extended with a class `Segment`, which allows edges between Nodes that can be used for routing with standard algorithms. Figure 6.2 shows this extension.

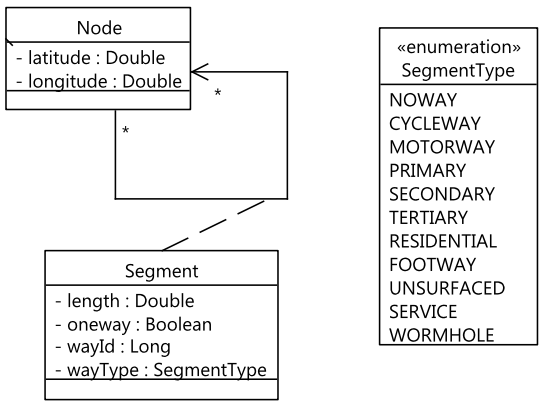


Figure 6.2: Part of the OSM schema for routing

The types `Node` and `Segment` are the only relevant types for routing. The enumeration `SegmentType` classifies the type of way that is represented by a segment. This information is taken from the map of tags in the represented `Way`. The other vertex and edge classes from figure 6.1 are only relevant for rendering the map.

6.1.1.2 Routing in OSM graphs

JGStreetMap ships with an implementation of the Dijkstra algorithm, which is used for computing the shortest paths, representing shortest routes, in OSM graphs. An OSM graph is generally interpreted as undirected graph. This is done for minimizing the number of generated segments. However, some edges are interpreted as directed edges. It is especially true for segments representing a one-way road. For segments representing one-way roads, the class `Segment` has the attribute `oneway`. Only if this attribute is `true`, the edge is interpreted as directed edge. Furthermore many segments are not important for routing. E.g., when computing a route for driving by car, the segments representing footways are ignored. JGStreetMap computes routes for driving by car by default. Alternatively it is possible to compute routes for riding a bike and for walking. In the following, routes for driving by car are assumed.

The information, which segments are relevant, the edge direction interpretation and the edge weight function are directly embedded into the implementation. As a consequence, this implementation of Dijkstra is hardly reusable. So it is feasible to have a reusable implementation of Dijkstra, that can also be used in the domain of JGStreetMap. The following section shows how *Algolib*'s implementation of Dijkstra can be used in JGStreetMap.

6.1.2 Adjusting *Algolib* for JGStreetMap

Algolib also ships with an implementation of the Dijkstra algorithm. However, this implementation is very generic and has to be configured properly for using it with JGStreetMap.

6.1.2.1 Defining the subgraph

The schema introduced above suggests that an instance of an OSM graph contains more graph elements than needed for routing. For filtering relevant graph elements, *Algolib* provides the concept of subgraphs (see chapter 4). Subgraphs can either be defined by a graph marker or by a wrapped method call. Since creating a graph marker takes some time ($\mathcal{O}(max(m,n))$) and the definition of a proper subgraph for this domain can be defined by a simple predicate, a wrapped method call is used. Listing 6.1 shows how this can be defined in *Algolib*.

Listing 6.1: The subgraph definition for OSM graphs

```
1 private static BooleanFunction<GraphElement> subgraph = new
     MethodCallToBooleanFunctionAdapter<GraphElement>() {
2         @Override
3         public boolean get(GraphElement parameter) {
4             return parameter.getM1Class() instanceof Segment
5                     || parameter.getM1Class() instanceof Node;
6         }
7     };
```

When using this subgraph, only the vertices of type `Node` and the edges of type `Segment` are considered.

6.1.2.2 Defining the navigability

As mentioned above, not all segments are relevant for routing and some edges are treated differently (e.g., edges representing one-way roads). For deciding whether an edge is navigable or not, *Algolib* allows defining a function `navigable`. For the OSM domain, this function has to ensure that only specific segments are navigable and that some segments are treated differently. Listing 6.2 shows the definition of the function `navigable` for OSM graphs when computing routes for driving by car.

Listing 6.2: The navigable definition for OSM graphs

```
8  private static BooleanFunction<Edge> navigable = new
       MethodCallToBooleanFunctionAdapter<Edge>() {
9
10         private Set<SegmentType> relevantTypes;
11         {
12             relevantTypes = new HashSet<SegmentType>();
13             // fill with relevant types ...
14         }
15
16         @Override
17         public boolean get(Edge parameter) {
18             if(parameter instanceof Segment)
19                 Segment currentSegment = (Segment) parameter;
20                 return relevantTypes.contains(currentSegment.get_wayType())
21                         && (currentSegment.isNormal() || !currentSegment
22                                 .is_oneway());
23             } else {
24                 return false;
25             }
26         }
27
28     };
```

For filtering out irrelevant segment types, simply a set of relevant segment types is used. The more interesting part is the way segments are treated differently. Lines 20 and 21 ensure that relevant segments with normal edge orientation are always considered, whereas reversed oriented segments are only considered, if the segment does not model a one-way road (line 22).

6.1.2.3 Defining the edge weight

Algolib's implementation of the Dijkstra algorithm needs to know the weight of each relevant edge. JGStreetmap provides several possible weights, but for simplicity reasons, only the length of a segment is taken as example. The type `Segment` has an attribute `length` that contains the length of the segment in meters. The weight function, for this example, is simply a wrapped method call that delegates to the getter of this attribute. Listing 6.3 shows this wrapper.

Listing 6.3: The edge weight function

```
29 private static DoubleFunction<Edge> edgeWeight = new
       MethodCallToDoubleFunctionAdapter<Edge>() {
30
31         @Override
32         public double get(Edge parameter) {
33             return ((Segment) parameter).get_length();
34         }
35
36     };
```

6.1.2.4 Executing the algorithm

Now finally *Algolib*'s implementation of Dijkstra can be run with the information provided above. Listing 6.4 shows how this is done.

Listing 6.4: Running Dijkstra

```
37 DijkstraAlgorithm dijkstra = new DijkstraAlgorithm(graph, subgraph,
38             navigable, edgeWeight).undirected();
39     try {
40         dijkstra.execute(start);
41     } catch (AlgorithmTerminatedException e) {
42     }
43     Function<Vertex, Edge> parent = dijkstra.getParent();
```

`graph` is a reference to an OSM graph instance. `start` is a reference to the start node. During the instantiation of the algorithm object, all required functions are passed as parameters. After the instantiation, the algorithm object is configured for interpreting the graph as undirected graph. The desired result is the function `parent`. It describes a tree containing all shortest paths from the start node to any reachable node. It can be used for deriving an actual path to a reachable destination.

6.1.3 Speed comparison

In order to show the efficiency of *Algolib*'s generic approach, a speed comparison between JGStreetMap's built-in implementation of Dijkstra and *Algolib*'s generic implementation of Dijkstra was performed. The results of this comparison are shown in this section. Furthermore, the overhead of adding visitors to *Algolib*'s implementation of Dijkstra was analyzed.

6.1.3.1 Setup

For measuring the runtime of a variant implementation of Dijkstra, multiple runs have been performed, and the final result was determined by the mean of the durations for each run. More specifically, 150 runs have been made per variant. The final result was calculated based on the mean of all results ignoring the 8 lowest (roughly 5%) and 30 (roughly 20%) highest results. The latter was necessary because some runs of JGStreetMap's internal implementation of Dijkstra have been disturbed by the garbage collector, resulting in extremely high duration values (e.g., 6 seconds instead of 0.7 seconds).

The input graph was an older map of Rhineland-Palatinate that has been imported using JGStreetMap's importer for OSM-XML files. The number of graph elements for this graph is 3301351, with 900870 vertices and 2400481 edges. The number of graph elements relevant for computing shortest routes for driving by car is 826770 with 402920 nodes and 423850 segments. All measurements have been made for the same start node, which is located at the University of Koblenz.

A program was written which executed the different implementations one after another, recorded the results and computed the final result. The tested implementations included the implementation of Dijkstra, that ships with JGStreetmap, a variant of this implementation using a different set of graph markers (see section 6.1.3.2 for details) and finally the generic implementation of Dijkstra, that is included in *Algolib*. The latter was measured with different numbers of empty visitors (0 up to 15) for judging the overhead created by visitors.

The system executing the measuring program was an AMD Phenom II X4 with 3GHz and 4GB of RAM running a 64 bit version of Ubuntu 10.04. It was executed directly from within Eclipse with a maximum heap size of 3GB.

6.1.3.2 Results

Table 6.1 contains the results of the comparison between the implementation of JGStreetMap's and *Algolib*'s implementation of the Dijkstra algorithm.

JGStreetMap's original implementation of Dijkstra runs slower than *Algolib*'s generic implementation. It also required the usage of the garbage collector more frequently than *Algolib*'s implementation. The reason for both is the handling of temporary attributes. Both approaches

use graph markers, but *Algolib*'s implementation uses one graph marker per temporary attribute and JGStreetMap's uses one graph marker for all temporary attributes. This one graph marker marks the graph elements with tuples containing the values for the attributes. An instance of such a tuple requires the overhead of creating objects and destroying them. The former causes the additional runtime and the latter causes the garbage collector to be called more frequently.

For a fairer comparison, a variant of JGStreetMap's implementation has been created, that uses multiple graph markers for avoiding the tuples. The runtime result for this variant can be found in the rightmost column of table 6.1.

The runtime measurement shows that a domain specific implementation can be faster than *Algolib*'s generic implementation, if the former is implemented properly. However, the generic implementation is sufficiently fast for competing with the domain specific implementation.

The overhead of visitors

One of *Algolib*'s biggest advantages over domain specific implementations is the extendability with visitors. Naturally, adding visitors slows down the runtime of algorithms supporting them.

The Dijkstra algorithm supports the `GraphVisitor` and thus is extendible at two working points. Figure 6.3 illustrates the runtime increase when adding empty visitors to the algorithm. The runtime increases linearly according to the number of visitors.

Please note that this measurement only considered the overhead caused by visitors. When using actual visitors, the actual runtime increase will be higher, depending on the task that is performed by the visitor. Also note that the slope of the runtime increase will be steeper for algorithms with more than two working points. If a visitor increases the asymptotic complexity of an algorithm, the runtime increase will not remain linear.

This measurement shows that *Algolib*'s visitor concept keeps the overhead caused by visitors fairly low. Even after adding 15 visitors to an algorithm, which is expected to be a rare case, the runtime is less than twice the runtime of the algorithm without visitors.

JGStreetMap original	*Algolib* without visitors	JGStreeMap variant
756.21 ms	597.45 ms	469.63 ms

Table 6.1: Mean runtime of different implementations of the Dijkstra algorithm (in ms)

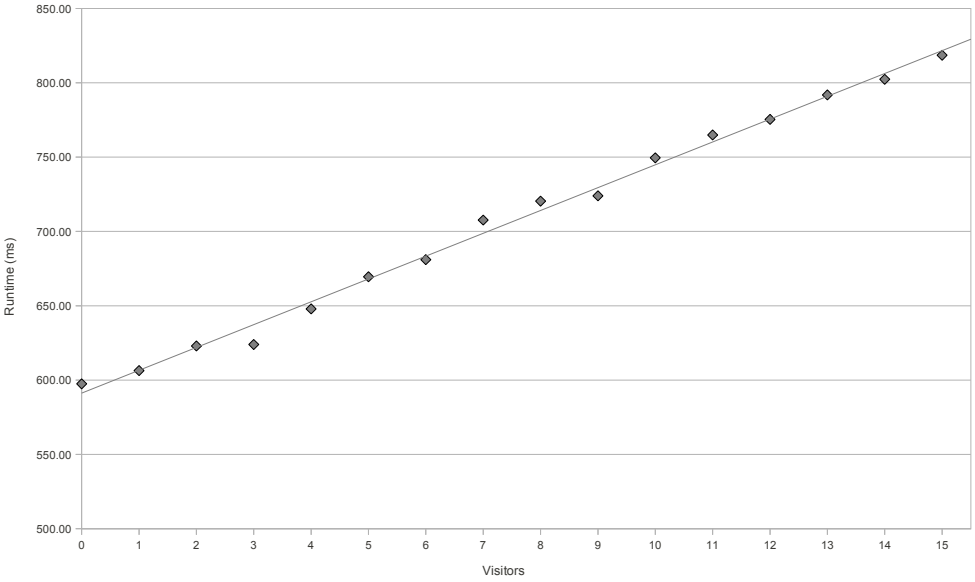


Figure 6.3: Runtime increase when adding visitors

7 Extending *Algolib*

Chapter 5 gave an overview on how algorithms are implemented in *Algolib*. Even though some rules for algorithm creation are denoted, no concrete rules have been phrased. So this chapter uses the information from chapter 5 for creating rules that have to be followed by developers who want to extend *Algolib* with further problems and algorithms.

Section 7.1 lists all rules for creating new problem interfaces. Section 7.2 lists all rules for creating new algorithm classes. Section 7.3 lists all rules for using visitors. Section 7.4 lists all rules for creating new visitors.

7.1 Rules for creating problem interfaces

This section introduces the rules for creating new problem interfaces.

- Problem interfaces have to be placed in the package `problems`.
- The name of a problem interface has to include the problem name and has to contain the suffix `Solver`.
- All problem interfaces have to be derived from `ProblemSolver` (directly or indirectly).
- The Javadoc of a problem interface has to contain:
  - information whether the problem is defined for both directed and undirected graphs or only one of them,
  - the description of all further input parameters that are not specified by `ProblemSolver` or the information that there are no further parameters, and
  - the description of all results.
- The Javadoc of a durable parameter's setter method may contain implementation specific information (e.g., what happens if the parameter is set to `null`).
- The Javadoc of a results's getter method may also contain implementation specific information (e.g., what happens if the method is called in the wrong state).
- The Javadoc of the execute method may contain further information about the transient parameters and should describe the functionality of the `AlgorithmTerminatedException`.

Listing 7.1 shows the problem interface `TraversalFromVertexSolver` as an example for the rules above.

Listing 7.1: Example of a problem interface

```
/**
 * The problem <b>traversal from vertex</b> is defined for directed and
 * undirected graphs. The only further input parameter is the <i>start
 * vertex<\i>. The results are a <i>permutation of vertices<\i> and a
 * <i>permutation of edges</i> of the reachable subgraph from the start
 * vertex.
 */
public interface TraversalFromVertexSolver extends TraversalSolver {
    /**
     * Solves the problem <b>traversal from vertex</b>.
     * @param root
     *            the vertex to start the traversal at
     * @return this algorithm object
     * @throws AlgorithmTerminatedException
     *             if this algorithm terminates before the actual
     *             execution is completed. This can happen from
     *             inside (early termination) or from outside
     *             (Thread interruption). The algorithm state
     *             changes accordingly.
     */
    public TraversalFromVertexSolver execute(Vertex root) throws
        AlgorithmTerminatedException;

    /**
     * Retrieves the result <code>vertexOrder</code> as permutation of
     * vertices.
     *
     * @return the result <code>vertexOrder</code>.
     * @throws IllegalStateException
     *             if the result is requested without being available
     */
    public Permutation<Vertex> getVertexOrder();
    // getEdgeOrder() analogously
}
```

Please note that the parameters `graph`, `subgraph`, and `navigable` are inherited from the super interfaces `ProblemSolver` and `TraversalSolver` and therefore not described in this interface.

7.2 Rules for creating algorithm classes

This section introduces the rules for creating new algorithm classes.

General rules

- Algorithm classes have to be placed in the package `algorithms`.

- An algorithm class has to be placed in the subpackage of the problem group it belongs to. If such a subpackage does not exist yet, a new subpackage should be created for it.
- The name of an algorithm class has to contain the name of the algorithm it implements with the suffix `Algorithm` (e.g., `WarshallAlgorithm`). Search algorithms may have the suffix `Search` instead. Algorithms using other algorithms are named after one problem they solve[1] and the algorithm they use (e.g., StrongComponentsWithDFS).
- Algorithm classes have to be derived from `GraphAlgorithm` (directly or indirectly).
- Algorithm classes have to implement all corresponding problem interfaces of problems they solve. The execute methods of the problem interfaces should be compatible, meaning they should have the same parameter list. E.g., `KahnKnuthAlgorithm` implements `AcyclicitySolver` and `TopologicalOrderSolver`. Exceptions may occur for similar problems with slightly different execute methods (e.g., `DijkstraAlgorithm`).
- Each algorithm class has to provide at least two constructors. The first constructor may only takes durable parameters that cannot be set to a default value. In all cases this includes a reference to the graph. If an algorithm class depends on another algorithm class, this constructor also takes an instance of the required algorithm. The second constructor takes all durable parameters. If the class does not introduce new parameters, the constructors may simply delegate their calls to the superclass. If new parameters are introduced, the first constructor should delegate to the second setting all additional parameters to `null`. In this case only the second constructor delegates to the superclass' constructor.
- An algorithm class should have a reference to one visitor list of the matching visitor type.
- Each algorithm class should override the methods `reset()` and `resetParameters()`. In these methods, the first statement has to be a call to the corresponding method of the superclass, because these methods handle the state checks in the class `GraphAlgorithm`. The method `resetParameters()` sets all durable parameters to their default values. So for newly introduced durable parameters this behavior has to be implemented. The method `reset()` has to be implemented so it initializes all runtime variables. Please note that both methods are called implicitly by the constructor of `GraphAlgorithm`, which is called before the constructors of actual algorithm classes. This means, accessing member variables, that are not initialized by these methods, would cause a `NullpointerException` and has to be avoided.
- Every execute method always returns `this`. The execute method's return type is always the concrete algorithm class. If an algorithm class inherits an execute methods (e.g., concrete search algorithms inherit the execute method for complete traversal), this method has to be overridden with the correct return type, delegate the call to the superclass and finally return `this`.

[1] This should be the most important problem, e.g., `TopologicalOrderWithDFS` also solves the problem *acyclicity*, but the problem *topological order* is considered more important. The choice however is made by the developer of the algorithm class.

- The method `isHybrid()` has to be implemented for telling whether the algorithm is able to work on directed and undirected graphs or only on one of them.
- The method `isDirected()` has to be implemented for telling whether the algorithm is (currently) working on a directed or undirected graph.

Rules for parameters, runtime variables and results

- Setter methods for durable parameters have to check the state first. The method `checkStateForSettingParameters()` should be used for this purpose.
- All runtime variables have to be made accessible through getters. If a runtime variable is an internal representation of a result, the getter method must have the infix `Internal`. If the internal representation of a result is identical to the result type (e.g., the function `number` in search algorithms), nevertheless there has to be a getter method for the runtime variable. There is no state check required for runtime variables.
- All results (mandatory or optional) have to be made accessible through getter methods. These getter methods have to provide a state check. The method `checkStateForResult()` should be used for this purpose.
- Every optional result requires two convenience methods for enabling and disabling its computation. These methods have the prefixes `with` or `without` respectively. Since the decision, if a specific optional result should be computed or not, is considered a durable parameter, the convenience methods for optional results have to perform the corresponding state check using the method `checkStateForSettingParameters()`. These convenience methods always return `this` and use the concrete type of the algorithm class as return type (in analogy to the execute method). This also means inherited convenience methods have to be overridden and adapted to the specialized class. By convention, the computation of an optional result is controlled by the instantiation of its internal representation. If it not instantiated (`null`), the corresponding optional result is not computed. If optional results are possible, the method `disableOptionalResults()` has to be overridden and implemented so it sets all internal representations of optional results to `null`. This method is also called by the method `resetParameters()` from `GraphAlgorithm` for disabling all optional parameters by default.

Rules for the actual algorithm implementation

- The methods `startRunning()` has to be called at the beginning and `done()` has to be called in the end of the execute method.
- The method `done()` must be overridden to set the algorithm state correctly after the algorithm terminated (either `STOPPED` or `FINISHED`)

- Whenever a graph element is accessed, the function `subgraph` has to be consulted first, if it is set. This means, first a `null`-check has to be performed on this function, followed by querying it in case it is not `null`, before actually accessing the graph element.
- Every algorithm that supports visitors, has to provide working points for each visit method of the visitor they are compatible with. If these working points include steps for the computation of (optional) results, these have to be performed *before* the corresponding visit method is called. This is important for visit methods in order to benefit from intermediate results.
- Every working point should be placed, so it is at most called once per graph element.
- Whenever an optional result is computed, a `null`-check has to ensure that its computation is actually activated.

The implementation of the breadth first search (listing 5.26 on page 101) is a good example for these rules.

Rules for algorithms relying on the graph structure

- Algorithms relying on the graph structure have to be derived from `StructureOrientedAlgorithm` for retrieving the durable parameter `navigable`.
- The function `navigable` has to be used in addition to `subgraph` for deciding whether an edge is navigable or not. This function may be `null` and has to be handled analogously to `subgraph`.
- The inherited convenience methods `normal()`, `reversed()` and `undirected()` have to be overridden and adapted in analogy to the convenience methods for optional results. If the algorithm only works on directed graphs, the method `undirected()` does not need to be overridden, because the implementation in `StructureOrientedAlgorithm` handles this. In analogy, if the algorithm only works on undirected graphs, the methods `normal()` and `reversed()` do not need to be overridden.
- For algorithms that work for both, directed and undirected graphs, the member variable `traversalDirection` should be used for deciding which incident edges are considered. While doing this, it has to be ensured that each edge is handled only once per working point. This can be done by a graph marker that stores which edges have already been visited (e.g., the search algorithms use such a graph marker).

Rules for creating algorithms using other algorithms

An example for algorithms that use other algorithms has been shown in section 5.3.5.1 on page 104. Here the rules for creating such algorithms are summarized. When an algorithm *a* is implemented using an algorithm *b*, *a* is called the *dependent algorithm* and *b* is called the *required algorithm* in the following.

- The name of a dependent algorithm class has to include the name of the required algorithm (e.g., `StrongComponentsWithDFS`).
- An instance of the required algorithm class has to be passed to the dependent algorithm class' constructor.
- The dependent algorithm can have a compatible visitor type which does not match the visitor type that is expected by the required algorithm class (e.g., `TopologicalOrderVisitor` in `TopologicalOrderWithDFS`). So the dependent algorithm class only handles its own visitor type and delegates other visitors to the required algorithm object. This happens in the method `addVisitor(...)` and requires a type check. If the algorithm class does not have such a visitor type, the visitors are simply passed to the algorithm object of the required algorithm class. Section 7.3 contains more information about visitor types of dependent visitors and how they are used in the algorithm.
- Dependent algorithm classes are not allowed to declare methods for delegating the setting of parameters to the required algorithm object. This also includes the delegation of convenience methods. Doing this would cause too much confusion when using these algorithms. It would also require to define methods for delegating results that are not computed by the dependent algorithm. If calling these methods is required, this can happen before the algorithm object of the dependent algorithm is created. E.g., if the algorithm `ShortestPathsWithBFS` is executed and the required BFS should also compute the optional result `level`, this can be achieved by calling the method `withLevel()` before the instance of `ShortestPathsWithBFS` is created. The result `level` can only be returned by the instance of `BreadthFirstSearch` because the getter methods for results of the required algorithm objects are generally not delegated.
- The methods `reset()` and `resetParameters()` of the dependent algorithm class are not allowed to delegate to the corresponding methods from the required algorithm class. The required algorithm object is a member variable. Accessing it in `resetParameters()` would cause a `NullPointerException`, because it is called in the constructor of `GraphAlgorithm`, which is called before the constructor of the dependent algorithm class is executed.
- The execute method of a dependent algorithm class sets up the algorithm object of the required algorithm so it can solve the problem. This includes
  - resetting,
  - copying (or adapting[2]) the parameters from the dependent algorithm (e.g., the graph),
  - enabling optional results that are required for the computation,
  - registering a visitor adapter (see section 7.3),
  - call `startRunning()` and `execute()` for the required algorithm object,
  - call the method `done()`, and finally

[2]E.g., in `TopologicalOrderWithDFS` the traversal direction is inverted.

  - remove the visitor adapter that was registered earlier.
- The method `done()` has to set the final state of the dependent algorithm object using the final state of the required algorithm object. In most of the cases it can be just copied, but it depends on the algorithm (e.g., in `ReachabilityWithSearch` it is set to `FINISHED` even if the search algorithm is in the state `STOPPED`).
- The getter methods for results can delegate to getter methods of the required algorithm object if the result of the required algorithm corresponds to a result of the dependent algorithm. If the result was computed in runtime variables of the dependent algorithm object, the getter methods are implemented in the same way as for normal algorithm classes. In both cases a call to `checkStateForResult()` is required.

7.3 Rules for using visitors

This section describes how visitors are actually used in *Algolib*. Using a visitor means implementing a visitor interface (e.g., by creating a subclass of a visitor adapter class) and registering an instance with an algorithm object.

- The method `setAlgorithm(...)` has to check if the passed algorithm is compatible with the visitor, even if it does not need to access its runtime variables. The method `addVisitor(...)` from `GraphAlgorithm` relies on this check. If a reference to the algorithm is required, it has to be set in this method. If the algorithm is compatible, the method `reset()` has to be called for (re-)initializing the visitor's runtime variables. When using `SearchVisitor` or `DFSVisitor`, this method is already implemented in the corresponding visitor adapter classes. The attribute `algorithm` is used for accessing the algorithm object of `SearchAlgorithm` or `DepthFirstSearch` respectively.
- The method `reset()` works in analogy to the corresponding method from `GraphAlgorithm`. It initializes all runtime variables used by the visitor.
- Only the required visit methods have to be overridden.
- If the visitor uses runtime variables, it needs to provide getter methods for them. This happens in analogy to the getter methods of runtime variables in algorithm classes and allows dependent visitors to access them.
- If the visitor depends on another visitor, its constructor has to take the required visitor and set a reference to it. Also the documentation of such a visitor has to clarify this dependency relation.
- If the visitor computes a result, it has to provide a getter method for this result. It has to call the method `checkStateForResult()` from the algorithm object before returning the result.
- When registering visitors with an algorithm that include dependency constraints, the user is responsible for registering them in correct order. Required visitors have to be registered before their dependent visitors are registered.

Visitors adapting other visitors

As mentioned above, algorithms classes depending on other algorithm classes generally have a visitor type of their own matching the problem they solve. In general this visitor type is incompatible with the required algorithm class. For such algorithms, a special visitor, which is compatible with the required algorithm class, has to adapt the calls to the visitor list containing the visitors of the dependent algorithm class. Section 5.3.5.2 on page 106 shows an example for it.

- An adapting visitor should be declared as anonymous nested classes for accessing the runtime variables of the dependent algorithm. It implements the interface of visitors compatible with the required algorithm. In doing it that way, the adapting visitor can also access the runtime variables of both, the dependent and the required algorithm class.
- The implemented visit methods can call the visit methods of the visitor list declared in the dependent algorithm class.
- The adapting visitor object has to be registered with the algorithm object of the required algorithm class before the execute method of the required algorithm object is called.
- The adapting visitor has to be removed from the required algorithm object after the method `done()` was called.

In *Algolib* currently the algorithm classes

- `ShortestPathsWithBFS`,
- `ReachabilityWithSearch`,
- `TopologicalOrderWithDFS` and
- `StrongComponentsWithDFS`.

use adapting visitors

7.4 Rules for creating new visitor interfaces

This section describes the rules for creating a new visitor interface.

- The new visitor interface has to be derived from the interface `Visitor` (directly or indirectly).
- New visitors require the definition of a visitor adapter that implements all inherited methods as empty stubs. If the new visitor interface is indirectly derived from `Visitor`, it has to be derived from the visitor adapter belonging to the interface it is directly derived from (e.g., `SearchVisitorAdapter` is directly derived from `GraphVisitorAdapter`, because `SearchVisitor` is derived from `GraphVisitor`.
- They also require the definition of a visitor list in analogy to the other visitor lists that can be found in *Algolib*. This includes:
  - deriving it from the corresponding visitor list in analogy to the visitor adapter,

- defining a list of visitors that does not allow duplicate entries,
- initialize the list as array list in the constructor,
- checking the type of a visitor before it is added,
- passing the added visitor to the superclass of the visitor list and
- implementing the visit methods by iterating through all visitors in the `ArrayList` using their index instead of an iterator.

The implementations of the visitor lists that are already defined can serve as orientation for implementing a custom visitor list.

Listing 7.2 shows the visitor list for `SearchVisitor`.

Listing 7.2: The visitor list of `SearchVisitor`

```
1  public class SearchVisitorList extends GraphVisitorList implements
2          SearchVisitor {
3
4      private List<SearchVisitor> visitors;
5
6      public SearchVisitorList(){
7          visitors = new ArrayList<SearchVisitor>();
8      }
9
10     @Override
11     public void addVisitor(Visitor visitor) {
12         super.addVisitor(visitor);
13         if (visitor instanceof SearchVisitor) {
14             if (!visitors.contains(visitor)) {
15                 visitors.add((SearchVisitor) visitor);
16             }
17         }
18     }
19
20     @Override
21     public void removeVisitor(Visitor visitor) {
22         super.removeVisitor(visitor);
23         if (visitor instanceof SearchVisitor) {
24             visitors.remove(visitor);
25         }
26     }
27
28     @Override
29     public void clearVisitors() {
30         super.clearVisitors();
31         visitors.clear();
32     }
33
```

```
34      @Override
35      public void visitFrond(Edge e) throws AlgorithmTerminatedException {
36          int n = visitors.size();
37          for (int i = 0; i < n; i++) {
38              visitors.get(i).visitFrond(e);
39          }
40      }
41
42      // the other visit methods are declared analogously
43  }
```

The method `addVisitor(...)` shows how to ensure that no duplicate entries are added to the list. Using an array list also preserves the relative order of the added visitors.

The remaining methods show how the other rules from above should be applied. By not using iterator objects the runtime efficiency is significantly improved, because otherwise for each visit method a creation of a visitor object would be required.

8 Summary

This final chapter concludes this mid-study thesis. In section 8.1, it is evaluated if the goals from section 1.3 on page 15 have been reached with the implementation of *Algolib*. Finally section 8.2 predicts some future work based on *Algolib*.

8.1 Evaluating the goals

8.1.1 Generic nature

The first goal was the generic nature of *Algolib*. This included the capability of adjusting an algorithm to arbitrary domains. This goal was reached by introducing several optional parameters to all graph algorithms.

The most important one is the parameter *subgraph*, which allows all algorithms to run on subgraphs. Chapter 6 showed that a subgraph can be declared as a simple method call.

The other important parameters are *navigable* for all structure oriented algorithms and *edge weight* for all algorithms working on edge weighted graphs. Furthermore many algorithms allow the user decide if the graph is interpreted as directed or undirected graph. So-far, all structure oriented algorithms, operating on directed graphs, allow for operating in reverse edge direction, which allows interesting applications (e.g. the online computation of a topological order using DFS).

Taking all this into consideration, *Algolib*'s graph algorithms are applicable to many domains where these graph algorithms can be used for solving domain specific problems. So they are sufficiently generic for reaching this goal.

8.1.2 Extensibility

The second goal was the extensibility of *Algolib*. This means the ability of the algorithms for solving problems they are normally not designed to solve.

Algolib addresses this by using optional visitor objects for computing additional results during an algorithm run. Several examples have been shown, how the visitor objects are used for this purpose (e.g. the computation of strong components using DFS). At least those algorithms supporting visitor objects, which is the majority, can be called extensible.

Algolib itself is also extensible, because one result of this thesis is a rule-set for creating further graph algorithms and adding them to *Algolib*.

So this goal has also been reached.

8.1.3 Efficiency

The final goal of *Algolib* was being fast (meaning efficient).

This goal addresses two aspects. Firstly, *Algolib*'s algorithms, that have been adjusted to specific domains, are required to be similarly fast in comparison to a domain specific implementation of corresponding algorithms. Secondly, *Algolib*'s overhead for adding visitors has to be kept low in order to remain this efficiency even when extending an algorithm.

Both aspects have been analyzed in chapter 6. The first result was, specialized graph algorithms are sometimes implemented less efficient then expected. The second result was, if implementing specialized algorithms properly, *Algolib* still performs well in comparison. It is far from taking twice as much time as a specialized implementation. The third result was, the implementation of visitors has been done sufficiently efficient. The overhead, when running an algorithm with several visitors, is fairly low. When adding more visitors, the growth is linear and the slope is low.

Taking the example form chapter 6 into consideration, it is possible to say that the goal of efficiency has been reached.

8.2 Future work

of Professor Ebert *Algolib* so-far does not include all important graph algorithms. So the obvious future work is including further algorithm implementations into the library.

Currently only a few domains with domain-specific algorithm implementations exist. Those could be replaced by their *Algolib* counterparts. Especially the implementation of JGStreetMap's Dijkstra should be replaced by *Algolib*'s implementation.

The working group for Software Technology at the university of Koblenz, especially Daniel Bildhauer, is currently doing some research on so-called Distributed Hieracical Hyper-TGraphs (DHHT graphs) [BE11]. *Algolib* could be adjusted to those graphs, e.g. for supporting parallel search on distributed graphs.

8.3 Conclusion

Algolib is a complex, but useful enrichment for JGraLab. The work on it was fun, but it took longer than initially expected. A very hard part was dissolving friction between the conception and the implementation of the graph algorithms.

I learned much about graphs and graph algorithms, which I consider valuable knowledge. Even though this work took so much time, I am satisfied with the final result.

List of Figures

Bibliography

[BE11] BILDHAUER, DANIEL and JÜRGEN EBERT: *DHHTGraphs - Modeling Beyond Plain Graphs*. in *Proceedings of the 2nd International Workshop on Graph Data Management: Techniques and Applications (GDM 2011)*. IEEE, 2011.

[BHR+10] BILDHAUER, DANIEL, TASSILO HORN, VOLKER RIEDIGER, HANNES SCHWARZ and SASCHA STRAUSS: *grUML - A UML based modeling language for TGraphs*. Technical report, University Koblenz-Landau, Institute for Software Technology, 2010. unpublished.

[Bil08] BILDHAUER, D.: *Entwurf und Implementation eines Auswerters für die TGraphanfragesprache GReQL 2*. VDM Verlag, 2008.

[EB10] EBERT, JÜRGEN and DANIEL BILDHAUER: *Reverse Engineering Using Graph Queries*. in ENGELS, GREGOR, CLAUS LEWERENTZ, WILHELM SCHÄFER, ANDY SCHÜRR and BERNHARD WESTFECHTEL (editors): *Graph Transformations and Model-Driven Engineering*, volume 5765 of series *Lecture Notes in Computer Science*, pages 335–362. Springer Berlin / Heidelberg, 2010.

[GHJV94] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON and JOHN M. VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Hor09] HORN, T.: *Ein Optimierer für GReQL2*. GRIN Verlag GmbH, 2009.

[Kah06] KAHLE, STEFFEN: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[Mar06] MARCHEWKA, KATRIN: *GReQL 2*. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[Tar72] TARJAN, ROBERT ENDRE: *Depth-First Search and Linear Graph Algorithms*. SIAM J. Comput., 1(2):146–160, 1972.

[Zie] ZIEGLER, ELISA: *JGStreetMap - Ein Navigationssystem als Anwendungsbeispiel von Graphentechnologie*.