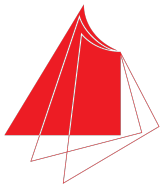


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Starke Zusammenhangskomponenten

Projektarbeit

von

Nicolas Ernst

Studiengang Informatik (M.Sc.)

Fakultät für Informatik und Wirtschaftsinformatik

Hochschule Karlsruhe - Technik und Wirtschaft

Betreuer:	Prof. Dr. Heiko Körner
Matrikelnummer:	74977
Abgabedatum & -ort:	24. Februar 2021, Karlsruhe

Zusammenfassung

Die vorliegende Arbeit behandelt die Realisierung eines Programms, das für die Darstellung von Graphen sowie die Visualisierung von Algorithmen auf diesen verwendet werden kann. Der Fokus liegt dabei insbesondere auf der Visualisierung des Algorithmus Starke Zusammenhangskomponenten.

Dieser wird zu Beginn zusammen mit anderen im Rahmen der Arbeit relevanten Grundlagen erarbeitet. Ausgehend von Stacks als grundlegende Datenstruktur in der Informatik, werden in diesem Kapitel auch die Grundlagen von Graphen als Datenstrukturen betrachtet. Dies umfasst wichtige Eigenschaften dieser, wie zum Beispiel den Zusammenhang, sowie deren Repräsentation in Computern. Auf Basis davon wird dann der Algorithmus Depth First Search aufgezeigt, welcher mehrfach in dem anschließend beleuchteten Algorithmus Starke Zusammenhangskomponenten verwendet wird.

Im Anschluss an die Grundlagen folgt eine Anleitung für die Bedienung des im Rahmen der Arbeit realisierten Programms. Diese geht neben notwendigen Schritten für die Installation hauptsächlich auf die Verwendung und Erweiterung des Programms ein.

Letztere wird dann in dem Kapitel „Implementierung“erneut aufgegriffen. Dabei wird anhand eines Komponenten- und Klassendiagramms aufgezeigt, wie die Erweiterung des Programms sowie der dafür notwendige, funktionale Rahmen realisiert wurde. Dieser Rahmen umfasst den Import und Export von Dateien, die Darstellung von Graphen sowie die Steuerung von Algorithmen, die über die realisierte Schnittstelle in das Programm eingebunden werden können. Zudem wird aufgezeigt, wie die in den Grundlagen erläuterten Algorithmen konkret realisiert wurden.

Abschließend wird die Arbeit durch ein Fazit abgerundet. Hierbei wird die in der Einleitung definierte Zielsetzung erneut aufgegriffen und analysiert, welche der darin erläuterten Aspekte durch das Programm und die vorliegende Arbeit erfüllt werden. Mit einem Ausblick werden zudem Möglichkeiten für die künftige Verbesserung, Erweiterung und Weiterentwicklung des Programms gegeben.

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Graphen stellen in der Informatik eine bedeutende Datenstruktur dar, welche die Grundlage vieler Problemlösungen bildet. Mit Hilfe dieser werden zum Beispiel Graph-Datenbanken oder Routenberechnungen in Navigationssystemen realisiert. Dementsprechend gibt es zahlreiche Algorithmen, die auf Graph-Strukturen operieren. So wird die Routenberechnung auf Basis des Dijkstra- oder A*-Algorithmus realisiert (Beck 2019).

Darüber hinaus werden Graphen oftmals für die Abbildung von Netzwerktopologien verwendet und dienen so als Grundlage für verschiedene Routingprotokolle. Darunter befinden sich zum Beispiel der topologiebasierte Ad-hoc On-demand Distance Vector-Routingalgorithmus oder das Optimized Link State Routing (Karin Anna Hummel 2010).

Dabei lassen sich mit Graphen nicht nur die Abhängigkeiten zwischen einzelnen Netzwerkknoten abbilden, sondern zum Beispiel auch Abhängigkeiten zwischen verschiedenen Internetseiten. Letztere ergeben sich, in dem eine Internetseite auf eine andere Internetseite verweist. Dies ist besonders für Suchmaschinen von Bedeutung, da diese so thematisch verwandte Internetseiten identifizieren und häufig verlinkte Internetseiten in den Ergebnislisten höher platzieren können. Auf die daraus resultierenden Graphen können dann wiederum verschiedene Algorithmen angewandt werden, wie beispielsweise der Algorithmus Depth First Search zum Durchmustern oder der Algorithmus Starke Zusammenhangskomponenten für das Identifizieren von zusammenhängenden Knoten (Körner 2009). Aufgrund deren heutigen Relevanz sollen die beiden zuletzt genannten Algorithmen im Rahmen der Arbeit näher betrachtet werden.

1.2 Problemstellung

Für die detaillierte Betrachtung von Algorithmen, die auf Graphen operieren, sowie für das Nachvollziehen derer Funktionsweisen existieren nur wenige und oftmals nicht mehr zeitgemäße oder kostenpflichtige Tools.

Zudem werden Graph-Algorithmen oft nur als fertige Programm-Bibliotheken angeboten. Diese gleichen dabei einer Art „Blackbox“, welche nur das Ergebnis des Algorithmus zu einem zuvor spezifizierten Graphen berechnen und bereitstellen. Dadurch kann die

Funktionsweise des implementierten Algorithmus nicht durch den Benutzer nachvollzogen werden.

Darüber hinaus arbeiten viele Tools befehlszeilenbasiert, indem sie die Eingabe des Benutzers intern in Graph-Strukturen überführen und das Ergebnis des angewandten Algorithmus wieder in der Befehlszeile ausgeben. Somit kann nicht interaktiv mit den Graph-Strukturen interagiert werden, was zu einer geringen Benutzerfreundlichkeit führt.

1.3 Zielsetzung

Ziel dieser Arbeit ist es, ein Programm zu entwickeln, welches vom Benutzer spezifizierte und beschriebene Graph-Strukturen visualisiert. Dabei soll der Benutzer mit diesen interaktiv interagieren und durch die Visualisierung einzelne Schritte der Algorithmen nachvollziehen können.

Neben dem Pausieren der Algorithmen-Ausführung sollte diese auch durch den Benutzer verlangsamt werden können. Darüber hinaus sollte die Ausführung auch schrittweise ermöglicht werden, so dass der Benutzer einfach zwischen den verschiedenen Algorithmusschritten navigieren kann.

Darüber hinaus sollten Graphen gespeichert und mit dem Programm wieder geöffnet werden können, so dass gegebenenfalls ein Austausch unter mehreren Benutzern möglich ist. Mit Hilfe von vorgefertigten Templates soll zudem die Erstellung von neuen Graphen für den Benutzer erleichtert werden.

Damit das Programm nicht auf ausgewählte Algorithmen beschränkt ist, sollte dieses zudem eine entsprechende Schnittstelle bereitstellen, über die vom Benutzer weitere Algorithmen zum Programm hinzugefügt werden können.

Ziel der Arbeit ist es auch, mit den Algorithmen Depth First Search sowie Starke Zusammenhangskomponenten zwei Algorithmen beispielhaft umzusetzen, so dass diese als Orientierung für spätere Erweiterungen des Programms dienen können. Aus dem gleichen Grund soll mindestens ein Template für die vereinfachte Erstellung eines Graphens anfertigt werden.

2. Grundlagen und Theorie

Dieses Kapitel behandelt die im Rahmen der Arbeit relevanten Grundlagen. Hierbei werden zunächst bekannte Datenstrukturen eingeführt, welche den Ausgangspunkt für die am Schluss des Kapitels erläuterten Algorithmen bilden.

2.1 Datenstrukturen

In der Informatik gibt es zahlreiche etablierte Datenstrukturen, die in der Programmierung eingesetzt werden können. Besonders relevant sind hierbei Stacks und Graphen, die von den beiden folgenden Unterkapiteln näher beleuchtet werden.

2.1.1 Stacks

Stacks, auch Stapel oder Keller genannt, stellen eine dynamische Datenstruktur dar, die mehrere Objekte aufnehmen kann. Dabei arbeiten Stacks nach dem Last-In-First-Out (LIFO)-Prinzip, wodurch man nur auf das zuletzt zum Stack hinzugefügte Objekt zugreifen und dieses gegebenenfalls wieder entfernen kann. Um auf das nächsttiefere Element zugreifen zu können, muss also das sich darüber befindliche Objekt zuerst entfernt werden (Ullenboom 2020).

Aus dieser Beschreibung ergeben sich drei charakteristische Methoden, die mit einem Stack einhergehen (Ullenboom 2020):

- `push`: Die `push`-Methode nimmt ein Objekt entgegen und legt dieses auf den Stack. Dadurch kann nur auf dieses neu hinzugefügte Objekt zugegriffen werden.
- `peek`: Mit Hilfe der `peek`-Methode kann auf das zuletzt zum Stack hinzugefügte Element zugegriffen werden. Dabei wird dieses nicht vom Stack entfernt.
- `pop`: Die `pop`-Methode entspricht im Wesentlichen der `peek`-Methode, nur dass in diesem Fall das oberste Element des Stacks auch entfernt wird. Dadurch wird das nächsttiefere Element zugreifbar.

2.1.2 Graphen

Graphen bilden heute als Datenstruktur die Grundlage vieler Problemlösungen. Neben Graph-Datenbanken werden Graphen beispielsweise auch für die Routenberechnung in Navigationssystemen verwendet (Beck 2019).

Dadurch sind im Laufe der Zeit viele Algorithmen entstanden, die auf Graph-Strukturen operieren. Mit dem Algorithmus Depth First Search und Starke Zusammenhangskomponenten werden im weiteren Verlauf der Arbeit zwei dieser Algorithmen betrachtet.

Graphen bestehen im Wesentlichen aus einer Menge von Knoten sowie einer Menge von Kanten, die die Knoten miteinander verbinden. Dabei unterscheidet man allgemein gerichtete sowie ungerichtete Graphen, wobei die Kanten letzterer immer bidirektional zwischen zwei Knoten definiert sind. Kanten gerichteter Graphen sind hingegen unidirektional und explizit in eine Richtung mit zugehörigem Start- und Zielknoten definiert. Eine Kante in die umgekehrte Richtung muss ebenfalls explizit definiert werden, was bei ungerichteten Kanten nicht der Fall ist (Körner 2009).

Die Repräsentation von Graphen als Datenstrukturen in Computern kann auf zwei unterschiedliche Arten erfolgen. Zum einen können sogenannte Adjazenzlisten in Knoten gespeichert werden, die die von dem jeweiligen Knoten aus erreichbaren Knoten beinhalten. Alternativ dazu kann auch eine Adjazenzmatrix geführt werden, welche die Knoten als Zeilen und Spalten umfasst. Eine Kante zwischen zwei Knoten kann dann einfach im zugehörigen Feld der Matrix eingetragen werden, das sich aus den beiden an der Kante beteiligten Knoten ergibt (Beck 2019).

2.2 Zusammenhang in der Graphentheorie

Einen ungerichteten Graphen bezeichnet man als zusammenhängend, wenn es zwischen zwei beliebigen Knoten `v` und `w` des Graphen mindestens einen Weg gibt. Ein Weg kann dabei aus mehreren Kanten und somit auch aus mehreren durchlaufenen Knoten bestehen, wobei die Start- und Zielknoten `v` und `w` entsprechen.

Darüber hinaus kann man in ungerichteten Graphen auch sogenannte Zusammenhangskomponenten definieren, bei denen es sich um Teilmengen von Knoten handelt. Eine solche Teilmenge ist so definiert, dass...

- zwei Knoten aus dieser immer über einen Weg verbunden sein müssen.
- kein Weg von einem Knoten aus der Teilmenge zu einem Knoten, der nicht in der Teilmenge enthalten ist, existieren darf.

- die Teilmenge knotenmaximal gewählt ist. So hat jeder weitere, nicht an der Zusammenhangskomponente beteiligte Knoten keine Verbindung zu den Knoten der Zusammenhangskomponente.

Dadurch sind die Knoten innerhalb einer Zusammenhangskomponente paarweise über Wege miteinander verbunden (Körner 2009).

Abbildung 2.1: Darstellung zweier nicht zusammenhängender Graphen (Körner 2009)

Abbildung 2.1 zeigt zwei Graphen, die beide nicht die Eigenschaft des Zusammenhangs erfüllen. Gemäß der erläuterten Definition besitzt der linke Graph jedoch drei, der rechte Graph zwei Zusammenhangskomponenten. Dies ist in der Abbildung auch daran zu erkennen, dass zwischen jeweils zusammenhängenden Knoten einer Zusammenhangskomponente kein Weg zu anderen, nicht an dieser Zusammenhangskomponente beteiligten Knoten existiert. Wichtig dabei ist, dass Zusammenhangskomponenten auch für Knoten-Teilmengen der Mächtigkeit 1, jedoch nicht für Knoten-Teilmengen der Mächtigkeit 0 definiert sind.

Darüber hinaus können die Eigenschaft des Zusammenhangs sowie die Zusammenhangskomponenten auch für gerichtete Graphen definiert werden. Hierbei spricht man von starkem Zusammenhang sowie starken Zusammenhangskomponenten, die in Kapitel 2.4 erläutert werden (Körner 2009).

2.3 Algorithmus Depth First Search

Der Algorithmus Depth First Search (DFS) stellt eine von zwei möglichen Durchmusterungsmethoden dar, mit denen Graphen systematisch durchsucht werden können. Der wesentliche Unterschied zum Algorithmus Breadth First Search besteht in der Verwendung eines Stacks statt einer Queue als zugrundeliegende Datenstruktur. Dadurch werden Pfade im Graphen zunächst vollständig in die Tiefe beschritten, bevor die Durchsuchung abzweigender Pfade erfolgt.

Abbildung 2.2 zeigt den Ablauf des DFS-Algorithmus in Form von Pseudocode. Dieser nutzt die Uhr `u`, mit der die Zeitpunkte der `push`- und `pop`-Operationen eines Knotens in den Zeitmarken `d[v]` und `f[v]` gespeichert werden.

```
ALGORITHMUS DFS

1    Push(Q, s);
2    Markiere s;
3    N[s] := Menge ℓ_s aller Nachfolgerknoten von s;
4    u := 1;
5    d[s] := u;
6    while not Stack-empty(Q) do {
7        v := top(Q);
8        if N[v] ≠ ∅ then {
9            Wähle v' ∈ N[v];
10           N[v] := N[v] \ {v'};
11           if (v' ist noch nicht markiert) then {
12               Push(Q, v');
13               Markiere v';
14               N[v'] := Menge ℓ_v' aller Nachfolgerknoten von v';
15               u := u + 1;
16               d[v'] := u;
17           }
18       } else {
19           Pop(Q);
20           u := u + 1;
21           f[v] := u;
22       }
23   }
```

Abbildung 2.2: Pseudocode des Algorithmus DEPTH FIRST SEARCH (Körner 2009)

Vor Beginn des Algorithmus muss zunächst noch ein Startknoten `s` aus der Knotenmenge des Graphens spezifiziert werden, auf den der Algorithmus angewendet werden soll.

Der gewählte Startknoten wird in Zeile 1 dann mit der in Kapitel 2.1.1 erläuterten `push`-Methode zum Stack `Q` hinzugefügt. Anschließend wird `s` markiert und die Menge der von `s` aus erreichbaren Knoten in `N[s]` bestimmt. Durch die Markierung ist der Algorithmus in der Lage, bereits bearbeitete Knoten von noch nicht bearbeiteten Knoten zu unterscheiden.

Darüber hinaus wird die Uhr `u` mit dem Wert `1` initialisiert und der zugehörige `d[s]`-Wert auf den Wert der Uhr gesetzt. Damit beträgt die Push-Zeit von `s` ebenfalls `1`.

Nach der Initialisierung des DFS-Algorithmus beginnt in Zeile 6 eine `while`-Schleife, welche erst verlassen wird, wenn in `Q` keine Objekte mehr enthalten sind.

Mit Hilfe der `top`-Methode wird dabei zunächst das zuletzt zu `Q` hinzugefügte Objekt in der Variablen `v` gespeichert, jedoch ohne das Objekt von `Q` zu entfernen. Die `top`-Methode entspricht damit der in Kapitel 2.1.1 erläuterten `peek`-Methode.

Anschließend wird geprüft, ob die Menge der von `v` aus erreichbaren Knoten leer ist. Sollte das nicht der Fall sein, wird ein Knoten `v'` aus dieser Menge gewählt und entfernt. Anschließend wird geprüft, ob dieser bereits markiert wurde. Ist dies nicht der Fall, bedeutet dies, dass `v'` noch nicht durch den Algorithmus bearbeitet wurde. Darauf hin wird `v'` zu `Q` hinzugefügt, `v'` markiert und die Menge der von `v'` aus erreichbaren Knoten `N[v']` bestimmt. Zudem wird die Uhr `u` inkrementiert und `d[v']` auf deren Wert gesetzt.

Die eben erläuterte `if`-Klausel führt dazu, dass durch die Wahl des Knoten `v` der Stack `Q` gegebenenfalls um weitere Elemente ergänzt wird, während hingegen die Menge der von `v` aus erreichbaren Knoten schrittweise reduziert wird.

Sollte die Menge `N[v]` der von `v` aus erreichbaren Knoten leer sein, so werden die Zeilen 18 bis 21 des Algorithmus ausgeführt. Diese entfernen `v` vom Stack `Q` durch die Verwendung der `pop`-Methode. Darüber hinaus wird die Uhr inkrementiert und die Pop-Zeit von `v`, also `f[v]`, auf den Wert der Uhr gesetzt. Somit kommt es zu einer Reduzierung der in `Q` enthaltenen Objekte.

Sind keine Elemente mehr in `Q` enthalten, wird die in Zeile 6 beginnende `while`-Schleife verlassen und der Algorithmus terminiert. Für alle vom Startknoten `s` aus erreichbaren Knoten wurden nun die Push- und Pop-Zeiten ermittelt, so dass die Durchmusterung des Graphen nachvollzogen werden kann. Knoten, deren Push- und Pop-Zeiten nicht gesetzt wurden, sind entsprechend nicht von `s` aus erreichbar. (Körner 2009).

2.4 Starke Zusammenhangskomponenten

2.4.1 Definition

Analog zu Zusammenhangskomponenten in ungerichteten Graphen, kann man starke Zusammenhangskomponenten für gerichtete Graphen definieren. Hierbei handelt es sich ebenfalls um eine nicht-leere Teilmenge von Knoten des Graphens, für die gilt, dass...

- zwischen zwei Knoten der Teilmenge immer ein Pfad existiert.
- zwischen einem in der Teilmenge enthaltenen Knoten `v` und einem nicht in der Teilmenge enthaltenen Knoten `w` kein Pfad von `v` nach `w`, von `w` nach `v` oder beides existiert. Somit dürfte zwischen `v` und `w` also maximal eine gerichtete Kante in eine Richtung existieren.

- diese knotenmaximal gewählt wurde.

Somit sind alle Knoten einer Zusammenhangskomponente paarweise über Pfade verbunden. Sind alle Knoten eines gerichteten Graphen an nur einer starken Zusammenhangskomponente beteiligt, so spricht man auch von starkem Zusammenhang (Körner 2009).

2.4.2 Gegenseitige Erreichbarkeit

Zur Ermittlung starker Zusammenhangskomponenten in einem gerichteten Graphen ergibt sich zunächst ein intuitiver Algorithmus. Hierbei ermittelt man unter Hinzuziehung von DFS für jeden Knoten `v` des Graphens die Menge aller von diesem aus erreichbaren Knoten `Reach[v]`.

Existieren zwischen zwei Knoten `v` und `w` Pfade in beide Richtungen, so sind die Knoten Teil der gleichen Zusammenhangskomponente. Diese Bedingung ist erfüllt, wenn `v` in der Menge `Reach[w]` und `w` in der Menge `Reach[v]` enthalten ist. Dies lässt sich leicht überprüfen und ist auch als Algorithmus Gegenseitige Erreichbarkeit bekannt (Körner 2009).

Der sich aus den Erläuterungen ergebende Pseudocode wird aufgrund dessen geringer Komplexität nicht im Rahmen der Arbeit betrachtet.

2.4.3 Algorithmus Starke Zusammenhangskomponenten

Der Algorithmus Gegenseitige Erreichbarkeit weißt nur für die Vorbereitungen bereits eine quadratische Komplexität auf und ist daher entsprechend ineffizient. Der Algorithmus Starke Zusammenhangskomponenten setzt an dieser Stelle an und ist in Abbildung 2.3 als Pseudocode dargestellt.

Wie aus Abbildung 2.3 hervorgeht, führt der Algorithmus Starke Zusammenhangskomponenten von jedem Knoten des Graphen eine DFS-Durchmusterung durch. Hierbei wird die in Kapitel 2.2 erläuterte Uhr `u` jedoch nicht bei dem Beginn einer jeden Durchmusterung zurückgesetzt, sondern weiter inkrementiert. Dadurch wird nach Durchführung aller DFS-Durchmusterungen deutlich, in welcher Reihenfolge die Knoten des Graphen durchlaufen wurden. Darüber hinaus wird bei jeder DFS-Durchmusterung der jeweils von `Q` entfernte Knoten auf einen zusätzlichen Stack `L` gelegt.

Nach Durchführung aller Durchmusterungen, werden die Knotenmarkierungen, die im Rahmen dieser entstanden sind, von allen Knoten entfernt und die Richtungen aller Kanten umgekehrt.

```
ALGORITHMUS DFS

1    Push(Q, s);
2    Markiere s;
3    N[s] := Menge ℓ_s aller Nachfolgerknoten von s;
4    u := 1;
5    d[s] := u;
6    while not Stack-empty(Q) do {
7        v := top(Q);
8        if N[v] ≠ ∅ then {
9            Wähle v' ∈ N[v];
10           N[v] := N[v] \ {v'};
11           if (v' ist noch nicht markiert) then {
12               Push(Q, v');
13               Markiere v';
14               N[v'] := Menge ℓ_v' aller Nachfolgerknoten von v';
15               u := u + 1;
16               d[v'] := u;
17           }
18       } else {
19           Pop(Q);
20           u := u + 1;
21           f[v] := u;
22       }
23   }
```

Abbildung 2.3: Pseudocode des Algorithmus STARKE ZUSAMMENHANGSKOMPONENTEN (Körner 2009)

Anschließend wird eine `while`-Schleife ausgeführt, die verlassen wird, wenn der aufgebaute Stack `L` keine Objekte mehr enthält. In dieser Schleife wird zunächst das zuletzt zu `L` hinzugefügte Objekt in der Variable `s` gespeichert und anschließend von `L` entfernt. Sollte `s` noch nicht markiert sein, wird von `s` aus eine weitere DFS-Durchmusterung gestartet. Diese unterscheidet sich jedoch marginal von der ersten DFS-Durchmusterung, da hier jeweils alle von `s` aus erreichbaren und neu markierten Knoten als starke Zusammenhangskomponente ausgegeben werden. Beispielsweise kann durch das Hinzuziehen eines Zählers, der bei jedem Start einer von `s` ausgehenden DFS-Durchmusterung inkrementiert wird, eine eindeutige Kennung der starken Zusammenhangskomponente generiert und gleichzeitig die Anzahl dieser im Graphen bestimmt werden. Sobald die erläuterte `while`-Schleife verlassen wird, terminiert der Algorithmus (Körner 2009).

3. Bedienungsanleitung

Zur Anwendung von Algorithmen auf Graphen wurde im Rahmen der Arbeit das Algorithm Visualization Tool (AVT) konzipiert und realisiert. Dieses ermöglicht neben der Ausführung des Algorithmus Starke Zusammenhangskomponenten auch die Ausführung anderer Algorithmen durch einen Plugin-Mechanismus. Das folgende Kapitel zeigt zunächst das Vorgehen für die Installation und Verwendung des AVTs auf. In Kapitel 4 folgen dann ausgewählte Details der Implementierung.

3.1 Installation

Voraussetzung für die Ausführung des AVTs ist die Installation des .NET Frameworks in der Version 4.7.2 oder höher. Anschließend kann das AVT mit einem Doppelklick auf die ausführbare Datei direkt gestartet werden.

Optional kann der Ordner, in dem sich die ausführbare Datei befindet, an eine beliebige Stelle kopiert und gegebenenfalls eine Verknüpfung zur ausführbaren Datei erstellt werden.

3.2 Verwendung

Nachdem die Ausführung des AVTs mit einem Doppelklick gestartet wurde, öffnet sich das in Abbildung 3.1 gezeigte Start-Fenster. Dieses teilt sich in zwei Hälften, die einerseits die Erstellung neuer und andererseits das Öffnen bestehender Graphen ermöglichen.

Letzteres wird dabei durch die rechte Hälfte abgebildet, die im Wesentlichen eine Übersicht über die vom Benutzer zuletzt geöffneten Graphen umfasst. Die in der Übersicht dargestellten Einträge können mit Hilfe einer Suchleiste gefiltert werden und liefern wichtige Details zu einem bereits geöffneten Graphen, wie dessen Namen und den Speicherort der zugehörigen `graph`-Datei. Letztere kann mit einem Doppelklick auf den Eintrag geöffnet werden, was zur Anzeige des Graph-Fensters führt.

Sollte ein zu öffnender Graph nicht in der Übersicht aufgeführt sein, kann dieser durch einen Klick auf den `Open graph...`-Button und der Auswahl der entsprechenden `graph`-Datei geöffnet werden. Neben der Anzeige des Graph-Fensters führt dies automatisch auch zu einem neuen Eintrag in der Übersicht. Die Einträge werden dabei in den Anwendungseinstellungen des AVTs persistiert, welche so auch bei einem Neustart des Systems

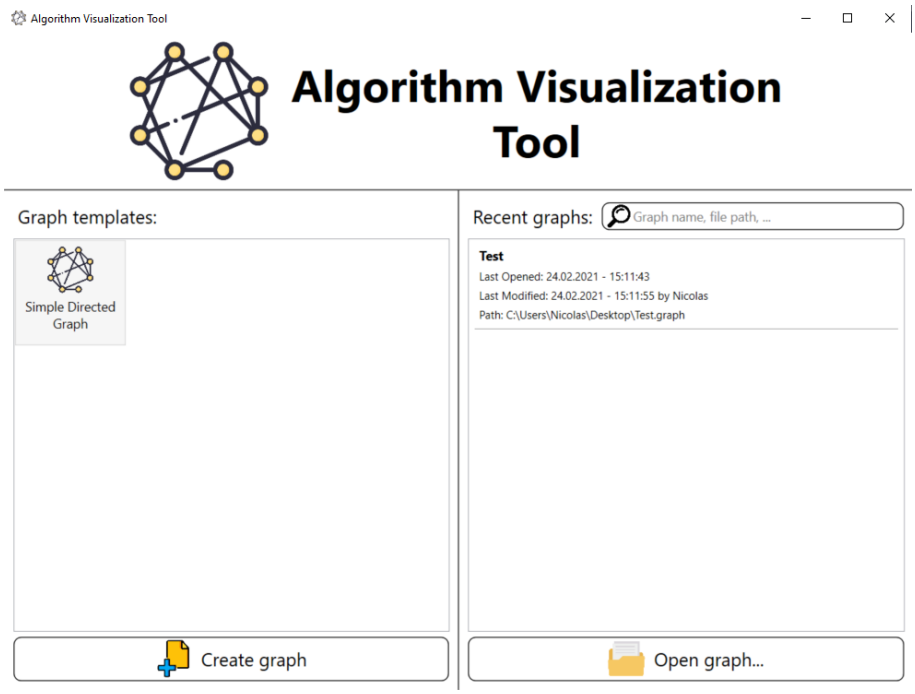


Abbildung 3.1: Start-Fenster des Algorithm Visualization Tool

erhalten bleiben. Darüber hinaus werden alle Einträge beim Start des AVTs auf Ungültigkeit geprüft und gegebenenfalls entfernt.

Neben `graph`-Dateien können mit dem `Open graph...`-Button auch Graph-Projekte importiert werden, die in Form von `gpr`-Dateien vorliegen. Diese beinhalten neben dem Graphen selbst auch Algorithmen, die auf dem Graphen angewendet werden können, sowie gegebenenfalls den Zustand eines aktuell angewandten Algorithmus. Zu beachten ist allerdings, dass ein modifizierter Graph, der aus einem Graph-Projekt stammt, nicht direkt aktualisiert werden kann. Hierzu muss entweder ein erneuter Export als Graph-Projekt oder die Speicherung des Graphen als separate `graph`-Datei vorgenommen werden.

Alternativ zum Öffnen bestehender Graphen mit der rechten Hälfte des in Abbildung 3.1 gezeigten Start-Fensters, können mit der linken Hälfte neue Graphen erstellt werden. Dazu kann der `Create graph`-Button oder alternativ eines der vorgefertigten Templates verwendet werden. Letztere sind in der zugehörigen Übersicht jeweils mit einem Bild und einer kurzen Beschreibung aufgeführt und können mit einem Doppelklick ausgewählt werden. In beiden Fällen öffnet sich zunächst ein Dialog-Fenster zur Eingabe des Graph-Namens, bevor sich anschließend das Graph-Fenster öffnet.

Das Graph-Fenster ist in Abbildung 3.2 dargestellt und teilt sich, wie das Start-Fenster auch, in zwei Teile. Die rechte Seite dient der Darstellung des Graphens, mit dem durch drei Tabs auf der linken Seite interagiert werden kann.

Abbildung 3.2 zeigt den `Overview`-Tab, der dem Benutzer verschiedene Informationen

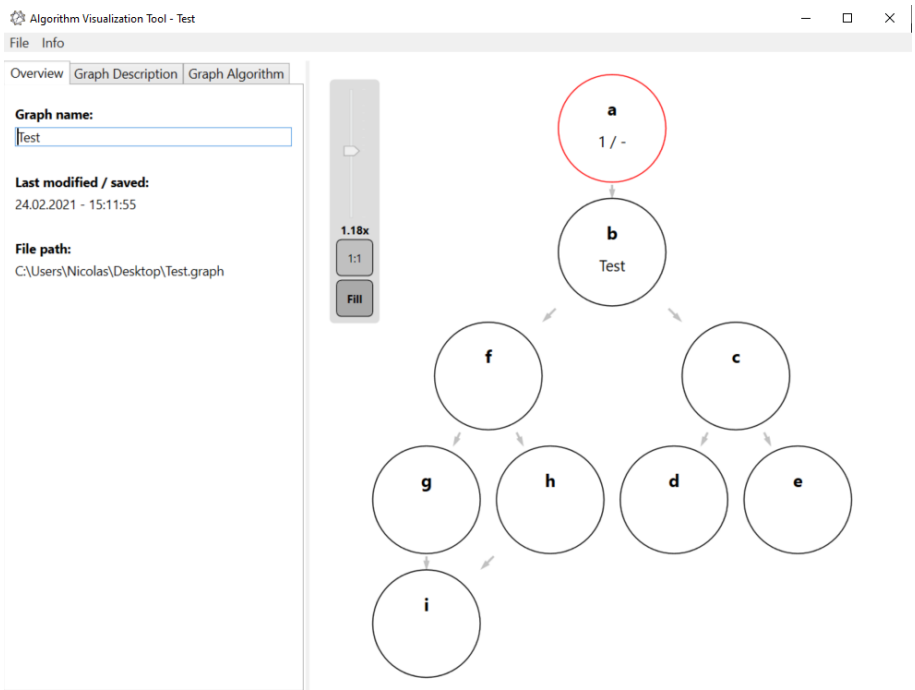


Abbildung 3.2: Overview-Tab im Graph-Fenster des ALGORITHM VISUALIZATION TOOL

über den aktuell geöffneten Graphen liefert. Diese umfassen den letzten Änderungszeitpunkt des Graphen, den Benutzer, der die letzte Änderung vornahm, den letzten Zeitpunkt der Öffnung, den Speicherort der zugehörigen `graph`-Datei sowie den aktuell gewählten Graph-Namen. Letzterer kann jederzeit geändert werden, was auch zur Anpassung des Fenstertitels führt, welcher den gwählten Namen beinhaltet.

Der in Abbildung 3.3 gezeigte `Graph Description`-Tab dient zur Beschreibung des Graphen in einer DOT-ähnlichen Graphbeschreibungssprache. Grundlage für diese bildet die in Form der Erweiterten Backus-Naur-Form (EBNF) dargestellten Grammatik in Listing 3.2.

Sollte im Start-Fenster ein neuer Graph erstellt worden sein, ist die im Tab angezeigte Beschreibung entsprechend leer. Andernfalls werden hier die im gewählten Template hinterlegte Beschreibung oder die Beschreibung des geöffneten Graphen angezeigt.

```
GRAPHDESCRIPTION ::= ( STATEMENT )*;
       STATEMENT ::= ( VERTEXDEFINITION | EDGEDEFINITION )
                     ( PROPERTYLIST )? ";";

VERTEXDEFINITION ::= ( VERTEXNAME | VERTEXGROUP );
     VERTEXGROUP ::= "{" VERTEXNAME ( "," VERTEXNAME )* "}";
     VERTEXNAME  ::= [A-Za-z0-9_][A-Za-z0-9_ ];
```

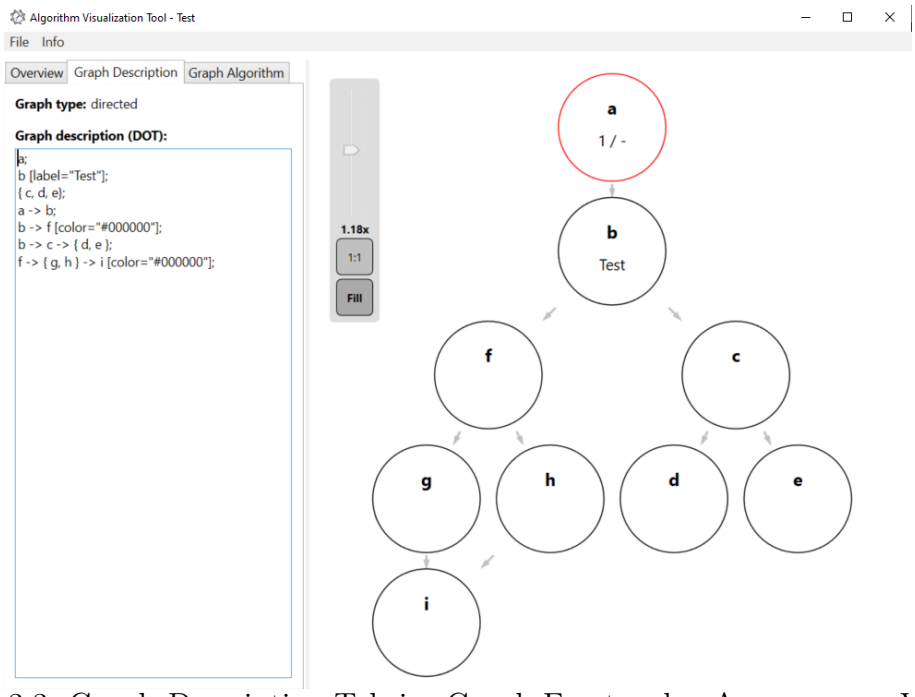


Abbildung 3.3: Graph Description-Tab im Graph-Fenster des ALGORITHM VISUALIZATION TOOL

```
EDGEDEFINITION ::= ( DIRECTEDEDGEDEFINITION |
                     UNDIRECTEDEDGEDEFINITION )
DIRECTEDEDGE ::= VERTEXDEFINITION "->" VERTEXDEFINITION
                 ( "->" VERTEXDEFINITION )*;
UNDIRECTEDEDGE ::= VERTEXDEFINITION "-----" VERTEXDEFINITION
                   ( "-----" VERTEXDEFINITION )*;

PROPERTYLIST ::= "[" ( PROPERTYLISTENTRIES )* "]";
PROPERTYLISTENTRIES ::= ( PROPERTYLISTENTRY
                        ( "," PROPERTYLISTENTRY )* )?;
PROPERTYLISTENTRY ::= ( [A-Za-z0-9_] )+ "="
                      """ ( [A-Za-z0-9_] )* """;
```

Listing 3.1: EBNF der Grammatik zur Beschreibung eines Graphen

Wie aus der Grammatik in Listing 3.2 hervorgeht, besteht eine Graph-Beschreibung aus beliebig vielen Statements, deren jeweiliges Ende mit einem Semikolon markiert wird. Ein Statement selbst kann dann entweder eine Knoten- oder Kanten-Definition beinhalten, welche optional um eine Liste von Eigenschaften ergänzt werden kann.

Eine Knoten-Definition besteht dabei entweder aus einem Knoten-Namen oder einer

Art des Statements	**Beispiel**
Einfache Knoten-Definition	`a;`
Einfache Knoten-Definition mit optionaler Liste von Knoten-Eigenschaften	`b [label="Test"];`
Definition einer Knoten-Gruppe	`{ c, d, e };`
Definition einer Knoten-Gruppe mit optionaler Liste von Knoten-Eigenschaften	`{ f, g, h, i } [label="Test 2"];`
Definition einer einfachen, gerichteten Kante	`a -> b;`
Definition einer einfachen, gerichteten Kante mit optionaler Liste von Kanten-Eigenschaften	`b -> f [color="#000000"];`
Definition mehrfacher, gerichteter Kanten mit Knoten-Gruppe	`b -> c -> { d, e };`
Definition mehrfacher, gerichteter Kanten mit Knoten-Gruppe und optionaler Liste von Kanten-Eigenschaften	`f -> { g, h } -> i [color="#000000"];`
Definition einer einfachen, ungerichteten Kante	`a -- b;`
Definition einer einfachen, ungerichteten Kante mit optionaler Liste von Kanten-Eigenschaften	`b -- f [color="#000000"];`
Definition mehrfacher, ungerichteter Kanten mit Knoten-Gruppe	`b -- c -- { d, e };`
Definition mehrfacher, ungerichteter Kanten mit Knoten-Gruppe und optionaler Liste von Kanten-Eigenschaften	`f -- { g, h } -- i [color="#000000"];`

Tabelle 3.1: Arten von Statements zur Beschreibung von Graphen

Knoten-Gruppe. Bei einer Knoten-Gruppe handelt es sich um mehrere, mit Komma separierte Knoten-Namen innerhalb einer geschweiften Klammer. Ein Knoten-Name muss mit einem Großbuchstaben, Kleinbuchstaben oder Unterstrich beginnen und kann anschließend auch Leerzeichen enthalten.

Kanten-Definitionen können entweder gerichtet oder ungerichtet sein, wobei sich auf Ebene der Grammatik lediglich das Trennzeichen zwischen `->` und `----` unterschiedet. Die an einer Kante beteiligten Knoten werden in Form einer Knoten-Definition angegeben und führen zur Erstellung der Knoten, sofern diese nicht bereits im Rahmen voriger Definitionen erstellt wurden. Andernfalls werden die in der Knoten-Definition spezifizierten und an der Kante beteiligten Knoten lediglich referenziert.

Aus der in Listing 3.2 beschriebenen Grammatik ergeben sich verschiedene Arten, wie ein Statement aufgebaut werden kann. Diese sind beispielhaft in Tabelle **??** aufgezeigt, wobei alle Knoten-Definitionen sowie alle Definitionen gerichteter Kanten der Graph-Beschreibung in Abbildung 3.3 gleichen.

Ist der Graph beschrieben, können mit dem in Abbildung 3.4 gezeigten `Graph Algorithm`-Tab verschiedene Algorithmen auf den erzeugten Graph-Strukturen ausge-

führt werden.

Dabei ist zu beachten, dass die Graph-Algorithmen die Graph-Strukturen, wie Knoten und Kanten, definieren. Dies führt dazu, dass das AVT möglichst generisch erweitert werden kann und Entwickler von Graph-Algorithmen die vom AVT mitgelieferten Standard-Graph-Strukturen gegebenenfalls anpassen können. Ein konkretes Beispiel wäre die Erweiterung der Hervorhebung von Knoten. Statt wie in Abbildung 3.4 gezeigt, nur rote Umrandungen für die Hervorhebung eines Knoten zu nutzen, könnte ein Graph-Entwickler zusätzlich auch den Hintergrund des jeweiligen Knotens einfärben.

Gleichzeitig bedeutet dies jedoch auch, dass der beschriebene Graph bis zur Selektion eines Algorithmus durch den Benutzer nicht in Graph-Strukturen überführt und damit auch nicht angezeigt werden kann. Darüber hinaus obliegt es dem Graph-Algorithmus, die aus der Graph-Beschreibung extrahierten Eigenschaften für Knoten und Kanten zu interpretieren und die Darstellung entsprechend anzupassen.

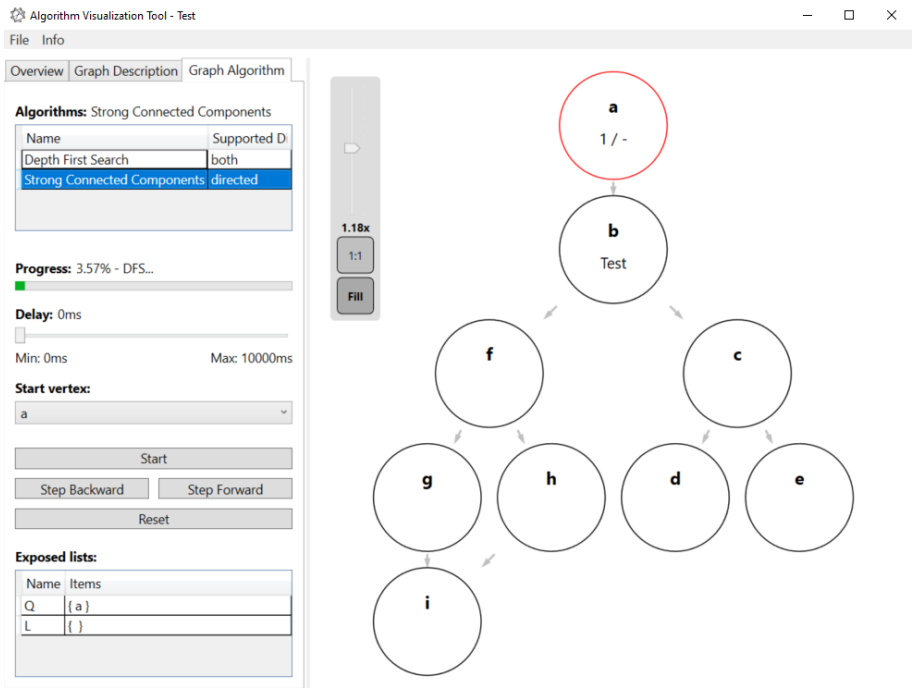


Abbildung 3.4: Graph Algorithm-Tab im Graph-Fenster des ALGORITHM VISUALIZATION TOOL

Neben der Auswahl eines Graph-Algorithmus, bietet der in Abbildung 3.4 gezeigte `Graph Algorithm`-Tab auch entsprechende Möglichkeiten zur Steuerung dieser. So kann der selektierte Algorithmus gestartet und anschließend über den gleichen Button wieder pausiert werden. Mit Hilfe des `Delay`-Schiebereglers kann zudem die abzuwartende Zeitspanne zwischen den Algorithmusschritten angepasst werden. Über einen Doppelklick auf das rechte Label mit der Aufschrift `10.000ms` kann das einstellbare Intervall entsprechend angepasst werden.

Alternativ zu dem `Start`- und `Stop`-Button können auch die `Step Forward` und `Step Backward`-Buttons verwendet werden, um den Algorithmus schrittweise auszuführen. In beiden Fällen muss für die Ausführung der Start-Knoten über das entsprechende Dropdown-Menü ausgewählt werden.

Zudem bietet der `Graph Algorithm`-Tab die Möglichkeit, vom Algorithmus gekennzeichnete Listen zusammen mit deren Inhalt darzustellen. Dadurch kann der Benutzer die Ausführung des Algorithmus zusammen mit der entsprechenden Anpassung der Graph-Darstellung besser nachvollziehen.

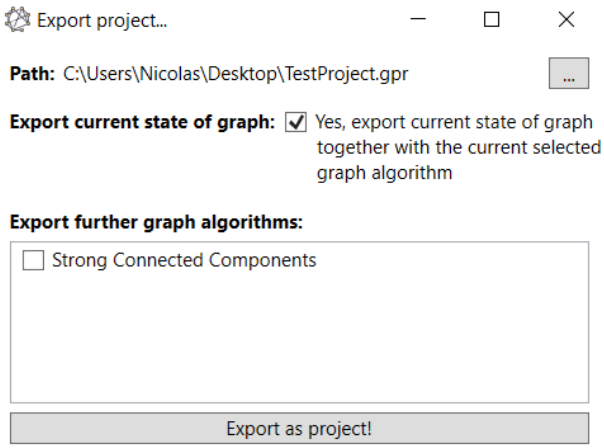


Abbildung 3.5: Export-Fenster des Algorithm Visualization Tool

Wie bereits zuvor erwähnt, kann der aktuelle Graph in Form eines Graph-Projekts zusammen mit dessen aktuellem Zustand exportiert werden. Das in Abbildung 3.5 gezeigte Export-Fenster kann über die Menü-Leiste im Graph-Fenster erreicht werden, indem auf `File` und anschließend `Export` geklickt wird.

Wie Abbildung 3.5 zeigt, muss für den Export zunächst ein Dateipfad angeben werden, unter dem die `gpr`-Datei gespeichert werden soll. Durch den Haken bei `Export current state of graph` wird der Graph zusammen mit dem aktuellen Zustand des angewandten Algorithmus exportiert. Dies führt dazu, dass beim Import der `gpr`-Datei der angewandte Algorithmus direkt ausgeführt wird, bis der exportierte Zustand wiederhergestellt wurde. Dazu wird der angewandte Algorithmus mit exportiert beziehungsweise importiert und kann durch die Algorithmen im Bereich `Export further graph algorithms` im Graph-Projekt gegebenenfalls ergänzt werden.

3.3 Erweiterung

Das AVT kann um Templates zur Erstellung von Graphen sowie um Algorithmen zur Anwendung auf Graphen erweitert werden. Die beiden folgenden Kapitel beschreiben das

dazu jeweilig notwendige Vorgehen.

3.3.1 Templates

Templates, die für die Erstellung eines Graphen verwendet werden können, befinden sich im gleichnamigen `Templates`-Ordner und liegen als `template`-Dateien vor. Hierbei handelt es sich um textbasierte Dateien, die Templates in Form der JSON definieren.

```
{
    "ImagePath": "SimpleDirectedGraph.png",
    "ImageDescription":"Simple Directed Graph",
    "DOTDescription":"a -> b;"
}
```

Listing 3.2: Beispiel für ein Graph-Template in der JSON

Listing 3.3.1 zeigt ein Beispiel für den Inhalt einer solchen `template`-Datei. Wie daraus hervor geht, besteht ein Template aus drei Eigenschaften:

- `ImagePath`: Diese Eigenschaft gibt einen von der `template`-Datei ausgehenden, relativen Pfad zu einem Bild an, welches vom AVT dann zur Darstellung in der Benutzeroberfläche verwendet wird. Ist diese Eigenschaft leer oder kann das referenzierte Bild nicht gefunden werden, wird vom AVT das Standard-Bild für Templates verwendet.
- `ImageDescription`: Bei dieser Eigenschaft handelt es sich um die kurze Beschreibung des Templates, die vom AVT in der Benutzeroberfläche unterhalb des Bilds angezeigt wird.
- `DOTDescription`: Die `DOTDescription` legt die Graph-Beschreibung fest, die bei der Verwendung des Templates für den neu erstellten Graphen übernommen wird. Dabei sollte auf das Escapen von Anführungsstrichen geachtet werden, so dass die Gültigkeit der JSON bestehen bleibt.

Um das AVT nun um weitere Templates zu erweitern, kann die in Listing 3.3.1 gezeigte JSON-Struktur herangezogen und nach entsprechender Anpassung als neue `template`-Datei im `Templates`-Ordner gespeichert werden. Für Templates, die auf Bild-Dateien zurückgreifen, wird empfohlen, die `template`-Datei zusammen mit dem Bild in einem weiteren, für das Template neu angelegten Unterordner zu speichern. So genügt es, in der `ImagePath`-Eigenschaft den Dateinamen des Bildes anzugeben. Das AVT ist entsprechend darauf ausgelegt, alle Unterordner im `Templates`-Ordner zu durchsuchen.

3.3.2 Algorithmen

Algorithmen, die im AVT auf Graphen angewandt werden können, liegen in Form von `dll`-Dateien im `Algorithms`-Ordner vor. Für die Erweiterung des AVTs um weitere Algorithmen genügt es, die `dll`-Datei eines Algorithmus in den `Algorithms`-Order zu verschieben. Hierbei ist darauf zu achten, dass unterschiedliche Algorithmen auch unterschiedliche Dateinamen tragen. Andernfalls können bereits vorhandene Algorithmen überschrieben werden.

Die Algorithmen werden bei jedem Start des AVTs eingelesen. Alternativ dazu können Algorithmen auch über Graph-Projekte exportiert und importiert werden. Das damit verbundene Vorgehen wurde bereits in Kapitel erläutert.

Kapitel 4.2 zeigt auf, wie Graph-Algorithmen entwickelt und als `dll`-Dateien bereitgestellt werden können.

4. Implementierung

4.1 Architektur

Die Projektmappe des AVTs besteht aus insgesamt vier Projekten, die jeweils einer Komponente in der Architektur des AVTs entsprechen. Deren Zusammenhang wird mit dem in Abbildung 4.1 gezeigten Komponentendiagramm deutlich.

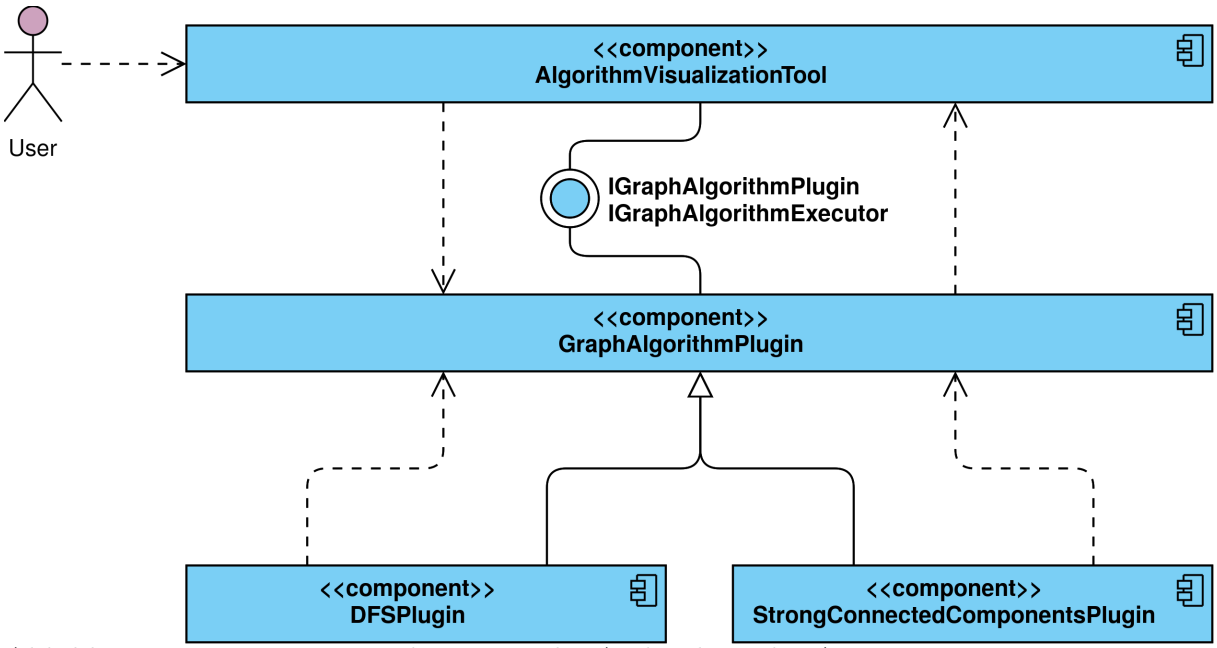


Abbildung 4.1: Komponentendiagramm der Architektur des Algorithm Visualization Tool

Wie aus dem Komponentendiagramm hervorgeht, handelt es sich beim AVT selbst um eine Komponente, die den funktionalen Rahmen für die Ausführung von Graph-Algorithmen bildet. Dies umfasst neben dem User Interface als Schnittstelle zum Benutzer und dem Management von `graph`- sowie `gpr`-Dateien auch die Anbindung des dazu notwendigen Plugin-Mechanismus.

Darüber hinaus zeigt das Diagramm in Abbildung 4.1, dass mit dem `DFSPlugin` sowie dem `StrongConnectedComponentsPlugin` zwei konkrete Komponenten realisiert wurden, die den in Kapitel 2.3 erläuterten Algorithmus Depth First Search und den in Kapitel 2.4 erläuterten Algorithmus Starke Zusammenhangskomponenten implementieren.

Durch die Einbindung dieser über den Plugin-Mechanismus können die beiden Algorithmen mit dem AVT genutzt werden.

Der Plugin-Mechanismus wird wiederum durch die `GraphAlgorithmPlugin`-Komponente realisiert, welche eine zusätzliche Abstraktionsebene zwischen dem AVT und den ausführbaren Graph-Algorithmen einführt. So stellt die Komponente Schnittstellen und Klassen bereit, die vom AVT und den Algorithmen gleichermaßen benötigt werden. Hierzu zählen beispielsweise die Standard-Graph-Strukturen, welche in Kapitel 4.1.2 näher erläutert werden und gegebenenfalls durch Graph-Algorithmen erweitert werden können.

Darüber hinaus abstrahiert die Komponente auch die Überführung der vom AVT entgegengenommenen Graph-Beschreibung in die vom Graph-Algorithmus spezifizierten Graph-Strukturen. Dadurch wird gewährleistet, dass die Graph-Beschreibung immer auf die gleiche Art und Weise in Graph-Strukturen überführt wird, dabei aber die vom Graph-Algorithmus spezifizierten Graph-Strukturen verwendet werden. Dabei kann es sich entweder um die Standard-Graph-Strukturen oder um eine Spezialisierung dieser im Graph-Algorithmus handeln. Dies wird in Kapitel 4.1.1 detaillierter beschrieben.

Die erzeugten Graph-Strukturen werden von der `GraphAlgorithmPlugin`-Komponente dann wiederum an das AVT übergeben, welches diese anschließend visualisiert und so dem Benutzer eine Rückmeldung liefert. Sollte ein Graph-Algorithmus ausgeführt werden, übergibt die Komponente zusätzlich auch den nächsten auszuführenden Algorithmus-Schritt sowie den aktuellen Zustand des Algorithmus an das AVT. Die konkreten Algorithmus-Schritte werden dabei von dem aktuell ausgewählten Graph-Algorithmus definiert, welcher somit eine Spezialisierung der `GraphAlgorithmPlugin`-Komponente darstellt. Dies kann ebenfalls dem in Abbildung 4.1 gezeigten Komponentendiagramm entnommen werden.

4.1.1 Schnittstelle für Erweiterungen

Das in Abbildung 4.2 gezeigte Klassendiagramm verdeutlicht den bereits angesprochenen Plugin-Mechanismus für Graph-Algorithmen durch die grün gefärbten Klassen.

Wie bereits in Abbildung 4.1 angedeutet, wird der Plugin-Mechanismen durch die beiden Schnittstellen `IGraphAlgorithmPlugin` und `IGraphAlgorithmExecutor` realisiert, die beide in der `GraphAlgorithmPlugin`-Komponente enthalten sind.

In der `AlgorithmVisualizationTool`-Komponente leitet sich die Klasse `GraphAlgorithmExecutor` dann von der `IGraphAlgorithmExecutor`-Schnittstelle ab und implementiert die darin spezifizierten Methoden und Properties. Neben den Properties `Progress` und `ProgressText` sowie der Methode `FinishedAlgorithm` ist dabei die Methode `MakeAlgorithmStep` besonders relevant, auf die später noch einmal

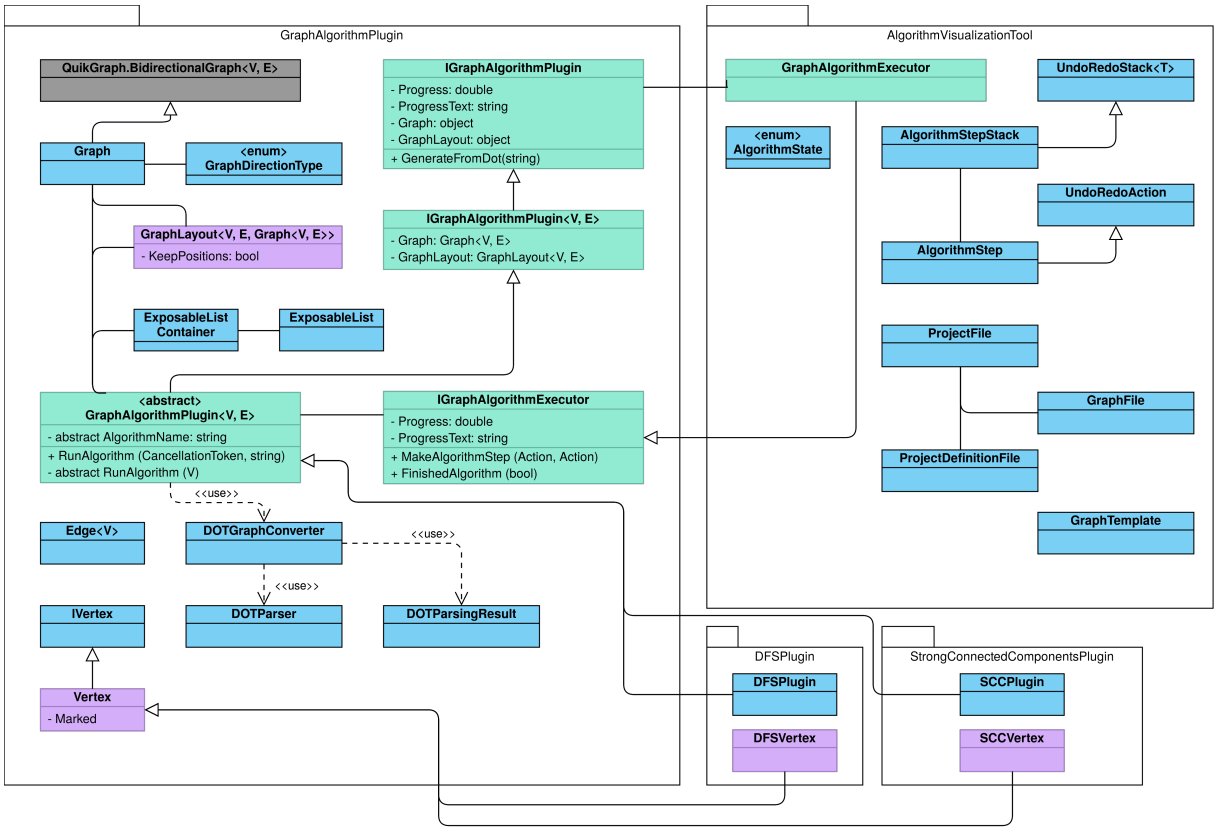


Abbildung 4.2: Vereinfachtes und verkleinert dargestelltes Klassendiagramm des Algorithm Visualization Tool. Anhang A5 zeigt das Klassendiagramm in voller Größe.

eingegangen wird.

Des Weiteren referenziert die `GraphAlgorithmExecutor`-Klasse eine beliebige Anzahl an Graph-Algorithmen, die durch die nicht-generische Variante der `IGraphAlgorithmPlugin`-Schnittstelle repräsentiert werden. Die hierbei verwendete `ObservableCollection` wird gemäß Kapitel 3.3.2 beim Start des AVTs befüllt. Aufgrund der Verwendung des Model-View-ViewModel (MVVM)-Entwurfsmusters führt dies durch entsprechende Bindings an die `ObservableCollection` automatisch auch zu einer aktualisierten Darstellung im User Interface.

Neben der durch die `GraphAlgorithmExecutor`-Klasse referenzierte, nicht-generische `IGraphAlgorithmPlugin`-Schnittstelle, liegt diese auch in einer generischen Variante vor. Letztere leitet sich von der nicht-generischen Variante ab und blendet die in dieser spezifizierten Properties durch die Verwendung des `new`-Schlüsselwortes aus. Dadurch können die ausgeblendeten, verallgemeinerten Properties des Typs `object` nun mit den vom Graph-Algorithmus spezifizierten, generischen Typen definiert werden. In der generischen `GraphAlgorithmPlugin`-Klasse werden die generischen Properties dann auf die nicht-generischen Properties gemappt, so dass letztere vom AVT unabhängig von den im Graph-

Algorithmus spezifizierten Typen verwendet werden können. Dadurch kommt es zu einer Entkopplung des AVTs und der Graph-Algorithmen.

Entsprechend leitet sich die `GraphAlgorithmPlugin`-Klasse von der generischen `IGraphAlgorithmPlugin`-Schnittstelle ab. Darüber hinaus hält die Klasse eine Referenz auf eine Instanz des Typs `IGraphAlgorithmExecutor`, wodurch die im Komponentendiagramm aufgezeigten Aspekte, wie der nächste auszuführende Algorithmusschritt, an das AVT übergeben werden können. Hierbei wird die bereits angesprochene Methode `MakeAlgorithmStep` verwendet, die in der `IGraphAlgorithmExecutor`-Schnittstelle spezifiziert ist. Die konkrete Instanz der `GraphAlgorithmExecutor`-Klasse, die sich von der `IGraphAlgorithmExecutor`-Schnittstelle ableitet, wird durch diese bei der Selektion eines Graph-Algorithmus über das gleichnamige Property gesetzt.

Wie dem Klassendiagramm in Abbildung 4.2 weiter entnommen werden kann, ist die `GraphAlgorithmPlugin`-Klasse als abstrakt gekennzeichnet. Konkrete Graph-Algorithmen können dann als Spezialisierung dieser Klasse realisiert werden, wobei das abstrakte Property `AlgorithmName`, das für die Darstellung des Algorithmus im User Interface verwendet wird, sowie die abstrakte Methode `RunAlgorithm`, die die einzelnen Algorithmusschritte definiert, überschrieben werden müssen.

Bei der Spezialisierung der abstrakten `GraphAlgorithmPlugin`-Klasse durch einen konkreten Graph-Algorithmus muss dieser zudem die Typen der Knoten und Kanten spezifizieren, die für die Erstellung der Graph-Strukturen verwendet werden sollen. Grundsätzlich können hierzu die Standard-Graph-Strukturen verwendet werden, die den Klassen `Vertex` und `Edge` in der `GraphAlgorithmPlugin`-Komponente entsprechen. Alternativ können auch spezialisierte Typen verwendet werden, die sich von den Standard-Graph-Strukturen ableiten. So nutzt die `DFSPlugin`-Komponente Knoten vom Typ `DFSVertex`, der sich von `Vertex` ableitet. Ähnlich verhält es sich mit Knoten des Typs `SCCVertex`, die von der `StrongConnectedComponentsPlugin`-Komponente verwendet werden.

Spezialisierte Knoten-Typen erlauben es, ...

- das im User Interface dargestellte Erscheinungsbild eines Knotens anzupassen.
- die virtuellen Methoden `Emphasize` und `Deemphasize` zu überschreiben und so das Erscheinungsbild bei der Hervorhebung des Knotens anzupassen.
- die virtuelle Methode `OnInitialize` zu überschreiben und so auf Eigenschaften zu reagieren, die aus der Graph-Beschreibung extrahiert wurden.
- zusätzliche Properties zu spezifizieren, welche gegebenenfalls die `Emphasize`- und `Deemphasize`-Methoden nutzen oder den Inhalt des Knotens manipulieren können. Ein Beispiel hierfür ist das boolesche `Marked`-Property der `Vertex`-Klasse, das bei

dem Setzen eines `true`-Werts die `Emphasize`-Methode und bei dem Setzen eines `false`-Werts die `Deemphasize`-Methode aufruft.

Ähnlich hierzu können auch Kanten-Typen spezialisiert werden, so dass auf extrahierte Eigenschaften reagiert und das Erscheinungsbild dieser gegebenenfalls angepasst werden kann. Im Rahmen der Arbeit wurde jedoch kein spezifizierter Kanten-Typ erstellt, weshalb auch nicht auf kantenbezogene Properties in der Graph-Beschreibung reagiert werden kann.

Damit abgeleitete Knoten- und Kanten-Typen im User Interface angezeigt werden können, muss es sich bei diesen logischerweise um User Controls handeln. Daher leiten sich diese auch von der im .NET-Framework enthaltenen Klasse `UserControl` ab und sind, wie alle Klassen, auf die diese Eigenschaft zutrifft, im Klassendiagramm lila gefärbt.

4.1.2 Domain Model

Das Domain Model des AVTs erstreckt sich über alle in Abbildung 4.1 dargestellten Komponenten. Besonders relevant ist dabei die `GraphAlgorithmPlugin`-Komponente, die mit den Klassen `Vertex` und `Edge` die Standard-Graph-Strukturen enthält. Darüber hinaus umfasst diese auch die Klassen `ExposableListContainer` und `ExposableList`, wobei erstere mehrere Instanzen der `ExposableList`-Klasse referenziert. Die `GraphAlgorithmPlugin`-Klasse hält wiederum eine Instanz der `ExposableListContainer`-Klasse, über die der konkrete Graph-Algorithmus Listen spezifizieren kann, die im von Abbildung 3.4 dargestellten `Graph Algorithm`-Tab zusammen mit deren Inhalt angezeigt werden sollen.

Darüber hinaus ist auch die generische `Graph`-Klasse Bestandteil der `GraphAlgorithmPlugin`-Komponente, die eine Graph-Struktur mit den vom Graph-Algorithmus spezifizierten Typen repräsentiert. Dazu greift die Klasse auf die QuikGraph-Bibliothek zurück und erbt von der darin enthaltenen, generischen Klasse `BidirectionalGraph`. So wird letztere durch die `Graph`-Klasse im Domain Model gekapselt.

Bei QuikGraph handelt es sich um einen Fork der QuickGraph-Bibliothek, welcher Überarbeitungen und Anpassungen für die Bereitstellung als NuGet-Pakete beinhaltet. Dabei stellt die Bibliothek generische Datenstrukturen für gerichtete und ungerichtete Graphen zur Verfügung und ergänzt diese um verschiedene Algorithmen (Rabérin 2021b). Letztere sind aufgrund der fehlenden Möglichkeit zur Visualisierung und der nicht nachvollziehbaren Algorithmenschritte nicht relevant für die Arbeit und können entsprechend nicht in Verbindung mit dem AVT genutzt werden.

Auf Seite des AVTs besteht das Domain Model hauptsächlich aus Klassen, um die Ausführung der vom Algorithmus spezifizierten Schritte sowie die Navigation zwischen diesen zu gewährleisten. Die `GraphAlgorithmExecutor`-Klasse hält hierfür eine Instanz der `AlgorithmStepStack`-Klasse, welche eine Spezialisierung des generischen `UndoRedoStack` darstellt. Letzterer umfasst einen Undo- sowie einen Redo-Stack, der mit Instanzen der Klasse `UndoRedoAction` befüllt werden kann. Durch die beiden Stacks können Aktionen rückgängig gemacht beziehungsweise wiederholt werden. Im Falle des spezifischeren `AlgorithmStepStack` werden die Undo- und Redo-Stacks mit Instanzen der `AlgorithmStep`-Klasse befüllt, die sich von der `UndoRedoAction`-Klasse ableiten. Die Instanzen werden dabei jeweils mit den `Action`s erstellt, die dem `GraphAlgorithmExecutor` in der `MakeAlgorithmStep`-Methode übergeben werden. So wird die Navigation zwischen den einzelnen Algorithmusschritten ermöglicht.

Darüber hinaus umfasst die `AlgorithmusVisualizationTool`-Komponente verschiedene Klassen, die zur Serialisierung und Deserialisierung der mit dem AVT verbundenen Dateien genutzt werden. Dabei wird die Klasse `GraphFile` für `graph`-Dateien, die Klasse `GraphTemplate` für `template`-Dateien sowie die Klassen `ProjectFile` und `ProjectDefinitionFile` für `gpr`-Dateien genutzt. Letztere dient dabei zum Serialisieren und Deserialisieren von Projekteigenschaften einer durch die `ProjectFile`-Klasse repräsentierten `gpr`-Datei.

Zudem wird das Domain Model je nach Graph-Algorithmen um weitere Klassen erweitert. Im Rahmen der Arbeit handelt es sich dabei um die `DFSPlugin`- und `DFSVertex`-Klassen der `DFSPlugin`-Komponente sowie um die `SCCPlugin`- und `SCCVertex`-Klassen der `StrongConnectedComponentsPlugin`-Komponente.

4.1.3 Erfassung und Darstellung von Graphen

Für die Überführung der vom AVT übergebenen Graph-Beschreibung in eine Graph-Struktur mit den vom selektierten Graph-Algorithmus spezifizierten Knoten- und Kanten-Typen, nutzt die `GraphAlgorithmPlugin`-Klasse die beiden statischen Klassen `DOTParser` und `DOTGraphConverter`. Somit wird die Graph-Struktur immer auf die gleiche Art und Weise erzeugt, jedoch unter der Verwendung jeweils anderer, vom Algorithmus abhängiger Typen.

Dementsprechend handelt es sich bei dem `DOTGraphConverter` um eine generische Klasse. Diese nutzt für das Parsen und Interpretieren der Graph-Beschreibung die Funktionen des `DOTParser`, die den in Kapitel 3.3.2 erläuterten Grammatik-Regeln entsprechen. Je nach identifiziertem Statement werden dann Knoten oder Kanten des jeweiligen Typs instantiiert und dem Graphen hinzugefügt. Dabei wird auch die jeweilige, in Kapitel

4.1.1 angesprochene `OnInitialize`-Methode aufgerufen, so dass bestimmte Eigenschaften aus der Graph-Beschreibung durch den Graph-Algorithmus berücksichtigt werden können.

Damit der erzeugte Graph visualisiert werden kann, wird bei Instantiierung des Graph-Algorithmus ein `GraphLayout` erzeugt. Wie aus dem Klassendiagramm in Abbildung 4.2 hervorgeht, handelt es sich aufgrund der lila Färbung um ein User Control, welches von der `GraphAlgorithmPlugin`-Klasse über das `GraphLayout`-Property bereitgestellt wird. Über die nicht-generische Schnittstelle kann dann vom AVT aus auf das `GraphLayout` zugegriffen werden. Da sich dieses von der `UserControl`-Klasse ableitet, kann das `GraphLayout` durch ein einfaches `ContentControl` mit entsprechendem Binding auf das `GraphLayout`-Property dargestellt werden. Im Graph-Fenster wird das `ContentControl` zusätzlich von einem in der Bibliothek `WPFExtensions` enthaltenen `ZoomControl` umgeben, so dass der Benutzer Scrollen und Zoomen kann.

Damit das `GraphLayout` als solches überhaupt visualisiert werden kann, leitet sich dieses von der gleichnamigen Klasse aus der GraphShape-Bibliothek ab. GraphShape ist mit QuikGraph kompatibel und ermöglicht so eine einfache Visualisierung der mit QuikGraph erzeugten Graph-Strukturen. Dabei werden auch die von Algorithmus spezifizierten Typen berücksichtigt, bei denen es sich gegebenenfalls auch um `UserControls` handelt. Darüber hinaus basiert GraphShape auf einer überarbeiteten und ergänzten Version von GraphSharp und stellt verschiedene Algorithmen für die Anordnung der Knoten sowie Kanten zur Verfügung (Rabérin 2021a).

4.2 Algorithmen

Für die Implementierung eines Graph-Algorithmus muss zunächst eine WPF-Steuerelementbilbiothek (.NET Framework) erstellt werden, welche die `GraphAlgorithmPlugin`-Komponente sowie Bibliotheken QuikGraph und GraphSharp als Abhängigkeiten referenziert.

Anschließend basiert die Realisierung auf der Spezialisierung der abstrakten `GraphAlgorithmPlugin`-Klasse sowie dem Überschreiben der abstrakten Properties und Methoden. Wie bereits in Kapitel 4.1.1 erläutert wurde, handelt es sich dabei um das `AlgorithmName`-Property, welches für die Darstellung des Graph-Algorithmus im User Interface verwendet wird, sowie um die `RunAlgorithm`-Methode, in der die einzelnen Algorithmusschritte definiert werden.

Letzteres erfolgt dabei unter Hinzuziehung der in der `GraphAlgorithmPlugin`-Klasse definierten `MakeAlgorithmStep`-Methode, welche zwei `Actions` als Parameter entgegen nimmt. Der zugehörige Methodenaufruf wird dann über die `GraphAlgorithmPlugin`-

Klasse an die darin referenzierte `GraphAlgorithmExecutor`-Instanz weitergeleitet, welche den Algorithmusschritt unter Verwendung des `AlgorithmStepStack` ausführt.

Die beiden `Actions`, die die `MakeAlgorithmStep`-Methode entgegen nimmt, entsprechen dabei den Befehlen, die bei der Ausführung des Algorithmusschritts beziehungsweise bei dessen Rückgängigmachen ausgeführt werden sollen. Je nach dem, ob der Benutzer den in Abbildung 3.4 gezeigten `Step Forward`- oder `Start`-Button verwendet hat, wartet das AVT auf das erneute Klicken des `Step Forward`-Buttons oder auf das Ablaufen der vom Benutzer definierten Zeitspanne, bevor der nächste definierte Algorithmusschritt ausgeführt wird. Dabei kommen asynchrone Funktionen zum Einsatz, so dass das User Interface des AVTs währenddessen nicht einfriert. Dementsprechend muss das `await`-Schlüsselwort in Verbindung mit der `MakeAlgorithmMethode` verwendet werden.

Darüber hinaus kann der Fortschritt des Algorithmus über die `Progress`- und `ProgressText`-Properties an das AVT übergeben werden, so dass diese durch die in Abbildung 3.4 gezeigte Fortschrittsanzeige visualisiert werden.

Der Zugriff auf die Graph-Struktur durch den Algorithmus ist über das `Graph`-Property möglich. Sollten Elemente hinzugefügt oder entfernt werden, so kommt es durch den Layout-Algorithmus des `GraphLayouts` automatisch zu einer Neuanordnung dieser. Dies kann verhindert werden, in dem der Wert des im `GraphLayout` spezifizierten `KeepPositions`-Property auf `false` gesetzt wird. Im Anschluss an die Modifikationen in der Graph-Struktur sollte das Property allerdings wieder auf `true` gesetzt werden, da andernfalls Knoten und Kanten im User Interface nicht mehr vom Benutzer verschoben werden können. Jedoch ist die korrekte Funktionsweise des Properties nicht gewährleistet.

Weiterhin sollte die `RunAlgorithms`-Methode mit einem Try-Catch-Block umgeben werden, der Exceptions vom Typ `TaskCanceledException` sowie `OperationCanceledException` abfängt. Diese können durch den Abbruch eines ausgeführten Algorithmus durch den Benutzer entstehen, haben aber keine direkten Auswirkungen auf das AVT. Aus diesem Grund sollte die Unterbrechung der Ausführung des AVTs durch das Abfangen der Exceptions vermieden werden.

Nach der Implementierung des Algorithmus kann dieser durch Visual Studio kompiliert werden, wobei eine DLL erzeugt wird. Diese DLL kann nun in den `Algorithms`-Ordner des AVTs kopiert werden, so dass der Algorithmus nutzbar wird.

4.2.1 Depth First Search

Die `DFSPlugin`-Komponente realisiert den in Kapitel 2.3 erläuterten Algorithmus Depth First Search durch die zwei Klassen `DFSVertex` und `DFSPlugin`, deren Code in Anhang A1 beziehungsweise A2 aufgeführt ist.

Wie aus Anhang A1 hervorgeht, erweitert die `DFSVertex`-Klasse die `Vertex`-Klasse aus der `GraphAlgorithmPlugin`-Komponente um die beiden Properties `PushTime` und `PopTime`. Wie aus Kapitel 2.3 hervorgeht, handelt es sich dabei um zwei relevante Knoteneigenschaften, die bei der Ausführung des Algorithmus Depth First Search verwendet werden. Ändert sich der Wert eines dieser Properties, so führt dies automatisch zu einer Aktualisierung des im User Interface dargestellten Knoteninhalts, der sich aus den beiden Properties zusammensetzt. Dies erfolgt unter Verwendung der `UpdateContent`-Methode.

Anhang A2 zeigt nun die Definition der `DFSPlugin`-Klasse. Dabei wird die Verwendung der `DFSVertex`-Klasse als Knotentyp durch deren Angabe bei der Spezialisierung der generischen `GraphAlgorithmPlugin`-Klasse deutlich.

Die in der `RunAlgorithm`-Methode definierten Algorithmenschritte bilden exakt den in Kapitel 2.3 aufgezeigten Algorithmus Depth First Search ab und werden daher nicht näher erläutert.

Von besonderer Relevanz im `DFSPlugin` ist lediglich die Verwendung des `ExposedLists`-Properties, bei dem es sich um eine Instanz der `ExposableListContainer`-Klasse handelt. Diesem wird im weiteren Verlauf der `RunAlgorithm`-Methode eine Liste mit dem Namen `Q` hinzugefügt, welche dem gleichnamigen Stack aus dem Algorithmus in Kapitel 2.3 entspricht. Da es sich hierbei um eine Liste, und nicht um einen Stack handelt, muss diese also parallel zu der eigentlich verwendeten Datenstruktur geführt werden.

4.2.2 Starke Zusammenhangskomponenten

Die `StrongConnectedComponentsPlugin`-Komponente realisiert mit den `SCCPlugin`- und `SCCVertex`-Klassen den aus Kapitel 2.4 bekannten Algorithmus Starke Zusammenhangskomponenten. Der Code der beiden Klassen ist in Anhang A3 und A4 aufgezeigt.

Wie Anhang A3 verdeutlicht, gleicht die `SCCVertex`-Klasse mit Ausnahme des `SccID`-Properties der Klasse `DFSVertex`. In diesem kann die Nummer der zugehörigen und durch den Algorithmus ermittelten Zusammenhangskomponente gespeichert werden, wobei die Änderung des Properties zu einer automatischen Änderung im User Interface führt.

Der in der Klasse `SCCPlugin` implementierte Algorithmus Starke Zusammenhangskomponenten setzt sich aus drei Teilen zusammen, wie bereits aus Kapitel 2.4 bekannt ist. Zu Beginn wird zunächst der Algorithmus Depth First Search ausgeführt, bis alle Knoten markiert sind. Gegebenenfalls muss dieser hierzu mehrfach mit verschiedenen Start-Knoten ausgeführt werden, wobei Uhr `u` für die Ermittlung der `PushTime` und `PopTime` nicht zurückgesetzt wird.

Anschließend werden alle Kanten im Graph umgeordnet. Damit es dabei nicht zu einer Neuanordnung der Graph-Strukturen im User Interface kommt, wird zuvor der Befehl `GraphLayout.KeepPositions = true;` ausgeführt. Im Anschluss wird dieser Wert wieder auf `false` gesetzt und erneut der Algorithmus Depth First Search ausgeführt. Dieser muss dabei ein Mal pro identifizierter starker Zusammenhangskomponente ausgeführt werden. Somit entspricht jeder erneute Start des Algorithmus einer weiteren gefundenen Zusammenhangskomponente, was zum einem Inkrement der `sccID` führt. Der Wert dieser Variable wird dann an das `SccID`-Property gefundener Knoten übergeben. Die Verwendung der `sccID` stellt hierbei den wesentlichen Unterschied zu dem zuerst verwendeten, auf der Uhr `u` basierenden Algorithmus Depth First Search dar.

5. Fazit

5.1 Ergebnisse der Arbeit

Mit dem AVT wurde im Rahmen der Arbeit ein interaktives und benutzerfreundliches Tool auf Basis von WPF und MVVM realisiert. Hierbei wurde viel Wert auf die Unterstützung des Benutzers gelegt, was beispielsweise durch das Start-Fenster zum Ausdruck kommt. Dieses ermöglicht dem Benutzer die Erstellung neuer Graphen auf Basis vorgefertigter Templates und bietet eine Übersicht über zuletzt geöffnete Graphen.

Darüber hinaus geht mit dem AVT eine umfassende Import- und Export-Funktionalität einher. Neben einfachen Graphen können so auch Graph-Projekte im- beziehungsweise exportiert werden, welche neben dem eigentlichen Graph auch Graph-Algorithmen sowie den Zustand eines angewandten Algorithmus umfassen. Dadurch kann auch ein entsprechend hoher und einfacher Austausch zwischen mehreren Benutzern des AVT erreicht werden.

Begünstigt wird dies auch durch die Erweiterbarkeit von Graph-Templates sowie -Algorithmen, welche beide über eine entsprechende Schnittstelle in das AVT eingebunden werden können. Das AVT selbst stellt also den funktionalen Rahmen für die Ausführung von Graph-Algorithmen bereit.

Für die Ausführung können Benutzer im AVT Graphen mit einer einfachen und intuitiven Graphbeschreibungssprache spezifizieren. Die daraus erzeugten Strukturen werden dann an den jeweiligen Graph-Algorithmus übergeben, welcher die im Graph zu verwendenden Knoten und Kanten spezifizieren kann. Dadurch wird eine vereinfachte Entwicklung von Graph-Algorithmen für das AVT sowie eine möglichst generische Erweiterung dessen erreicht. Darüber hinaus werden Änderungen an der Graph-Beschreibung bei selektiertem Graph-Algorithmus automatisch übernommen und im User Interface angezeigt. Durch die Manipulation der Graph-Strukturen während der Ausführung kann die Vorgehensweise des Algorithmus zudem optimal visualisiert werden. Dies wird durch die Darstellung von Listen im User Interface, die bei der Ausführung vom Algorithmen verwendet werden, ergänzt.

Die Ausführung der Algorithmen wurde zudem asynchron realisiert, so dass die Bedienfähigkeit des AVTs erhalten bleibt. Dies erlaubt auch das Pausieren sowie Verzögern von ausgeführten Graph-Algorithmen. Alternativ hierzu können Benutzer durch den realisierten Undo-Redo-Mechanismus auch schrittweise durch die einzelnen Algorithmusschritte

navigieren.

Darüber hinaus wurden mit dem Algorithmus Depth First Search und Starke Zusammenhangskomponenten zwei relevante und in den Graphentheorie bedeutsame Algorithmen realisiert, welche zusammen mit dem AVT visualisiert und in ihrer Funktionsweise nachvollzogen werden können.

Alles in allem stellt das AVT ein solides Tool zur Visualisierung von Graphen und Graph-Algorithmen dar, welches nur wenig Verbesserungspotential aufweist. Somit handelt es sich bei dem AVT um eine optimale Grundlage für die künftige Weiterentwicklung. Die zu Beginn der Arbeit formulierte Zielsetzung konnte somit übertroffen werden.

5.2 Ausblick

Auf kurzfristige Sicht könnte das AVT um formale Tests ergänzt werden, so dass eine testorientierte Entwicklung möglich und die Stabilität des AVTs wird. Dies konnte aufgrund der nur begrenzt zur Verfügung stehenden Zeit nicht im Rahmen der Arbeit realisiert werden. Zudem wäre die Unterstützung von User Controls für die Darstellung von Kanten wünschenswert.

Darüber hinaus könnte das AVT um den Export von Knoten-Positionen erweitert werden, so dass diese auch nach dem Öffnen eines Graphen oder dem Import eine Graph-Projekts erhalten bleiben. Dabei bietet das Tool durch das Zwischenspeichern der aktuellen Knotenpositionen bereits eine entsprechende Grundlage.

Auch die vom Algorithmus bereitgestellten Listen für die Darstellung im User Interface könnten überarbeitet werden, so dass diese nicht mehr parallel zu den eigentlichen Datenstrukturen geführt werden müssen. Eine Alternative wäre beispielsweise das Einführen von Annotationen, die dann direkt auf im Algorithmus spezifizierte Datenstrukturen angewandt werden können. Dies würde die Entwicklung von Graph-Algorithmen zusätzlich vereinfachen und entsprechend begünstigen.

Auf langfristige Sicht könnte das Tool als Grundlage für eine Entwicklungsumgebung herangezogen werden, die die Realisierung und das Testen von auf Graphen operierenden Algorithmen ermöglicht. Somit könnte ein enormer Mehrwert und gegebenenfalls auch eine kostenlose Alternative zu kommerziellen und gewerblich eingesetzten Programmen geschaffen werden.

A. Anhang

A.1 Code der in der Komponente DFSPlugin enthaltenen Klasse DFSVertex

```
public class DFSVertex : Vertex
{
    #region PushTime

    private int pushTime = 0;

    /// <summary>
    ///
    /// </summary>
    public int PushTime
    {
        get
        {
            return pushTime;
        }
        set
        {
            if (pushTime == value)
            {
                return;
            }

            pushTime = value;

            UpdateContent();
        }
    }
```

```
#endregion

#region PopTime

private int popTime = 0;

/// <summary>
///
/// </summary>
public int PopTime
{
    get
    {
        return popTime;
    }
    set
    {
        if (popTime == value)
        {
            return;
        }

        popTime = value;

        UpdateContent();
    }
}

#endregion


public DFSVertex() : base()
{
    // Nothing to do here
}
```

```
    public DFSVertex(string vertexName) : base(vertexName)
    {
        // Nothing to do here
    }


    private void UpdateContent()
    {
        string content = "-";

        if (PushTime > 0)
        {
            if (PopTime > 0)
            {
                content = PushTime + " / " + PopTime;
            }
            else
            {
                content = PushTime + " / -";
            }
        }

        VertexContent = content;
    }
}
```

A.2 Code der in der Komponente DFSPlugin enthaltenen Klasse DFSPlugin

```
public class DFSPlugin : GraphAlgorithmPlugin<DFSVertex, Edge<DFSVertex>>
{
    public override string AlgorithmName => "Depth First Search";

    public override GraphDirectionType CompatibleGraphDirections =>
    GraphDirectionType.Both;

    protected override async Task RunAlgorithm(DFSVertex startVertex)
    {
        try
        {
            Stack<DFSVertex> q = new Stack<DFSVertex>();
            Dictionary<DFSVertex, HashSet<DFSVertex>> descendantVertices
            = new Dictionary<DFSVertex, HashSet<DFSVertex>>();
            int u = 1;
            ExposableList exposedQ = new ExposableList("Q");
            ExposedLists.Add(exposedQ);

            await MakeAlgorithmStep(() =>
            {
                q.Push(startVertex);
                exposedQ.Insert(0, startVertex);
                startVertex.Marked = true;
                GetDescendants(startVertex, descendantVertices);
                startVertex.PushTime = u;
                Progress = (u / (Graph.VertexCount * 2.0)) * 100;
                ProgressText = "Executing...";
            }, () =>
            {
                q.Pop();
                exposedQ.Remove(startVertex);
                startVertex.Marked = false;
                descendantVertices.Remove(startVertex);
```

```
        startVertex.PushTime = 0;
        Progress = 0;
        ProgressText = "Initializing...";
    });

    while(q.Count > 0)
    {
        DFSVertex v = q.Peek();
        if (descendantVertices.ContainsKey(v) &&
        descendantVertices[v].Count > 0)
        {
            DFSVertex w = descendantVertices[v].First();
            descendantVertices[v].Remove(w);
            if (!w.Marked)
            {
                await MakeAlgorithmStep(() =>
                {
                    q.Push(w);
                    exposedQ.Insert(0, w);
                    w.Marked = true;
                    GetDescendants(w, descendantVertices);
                    u += 1;
                    w.PushTime = u;
                    Progress = (u / (Graph.VertexCount * 2.0)) *
                    100;
                }, () =>
                {
                    q.Pop();
                    exposedQ.Remove(w);
                    w.Marked = false;
                    descendantVertices.Remove(w);
                    u -= 1;
                    w.PushTime = 0;
                    Progress = (u / (Graph.VertexCount * 2.0)) *
                    100;
                });
            }
```

```
                }
                else
                {
                    await MakeAlgorithmStep(() =>
                    {
                        q.Pop();
                        exposedQ.Remove(v);
                        u += 1;
                        v.PopTime = u;
                        Progress = (u / (Graph.VertexCount * 2.0)) * 100;
                    }, () =>
                    {
                        q.Push(v);
                        exposedQ.Insert(0, v);
                        u -= 1;
                        v.PopTime = 0;
                        Progress = (u / (Graph.VertexCount * 2.0)) * 100;
                    });
                }
            }
        }
        catch (TaskCanceledException) { }
        catch (OperationCanceledException) { }
    }

    private void GetDescendants(DFSVertex vertex, Dictionary<DFSVertex,
    HashSet<DFSVertex>> descendants)
    {
        if (!descendants.ContainsKey(vertex))
        {
            descendants.Add(vertex, new HashSet<DFSVertex>());
        }
        foreach(DFSVertex descendantVertex in Graph.OutEdges(vertex).
        Select(x => x.Target))
        {
            descendants[vertex].Add(descendantVertex);
        }
```

```
    }
}
```

A.3 Code der in der Komponente StrongConnectedComponentsPlugin enthaltenen Klasse SCCVertex

```
public class SCCVertex : Vertex
{
    #region SccID

    private int sccID = 0;

    /// <summary>
    ///
    /// </summary>
    public int SccID
    {
        get
        {
            return sccID;
        }
        set
        {
            if (sccID == value)
            {
                return;
            }

            sccID = value;

            UpdateContent();
        }
    }

    #endregion

    #region PushTime
```

```
    private int pushTime = 0;

    /// <summary>
    ///
    /// </summary>
    public int PushTime
    {
        get
        {
            return pushTime;
        }
        set
        {
            if (pushTime == value)
            {
                return;
            }

            pushTime = value;

            UpdateContent();
        }
    }

    #endregion

    #region PopTime

    private int popTime = 0;

    /// <summary>
    ///
    /// </summary>
    public int PopTime
    {
        get
        {
```

```
            return popTime;
        }
        set
        {
            if (popTime == value)
            {
                return;
            }

            popTime = value;

            UpdateContent();
        }
    }

    #endregion


    public SCCVertex()
    {
        UpdateContent();
    }


    private void UpdateContent()
    {
        string content = "";
        if (SccID > 0)
        {
            content = SccID.ToString();
        }
        else
        {
            if (PushTime > 0)
            {
                if (PopTime > 0)
                {
```

```
                    content = PushTime + " / " + PopTime;
                }
                else
                {
                    content = PushTime + " / -";
                }
            }
        }
        VertexContent = content;
    }
}
```

A.4 Code der in der Komponente StrongConnectedComponentsPlugin enthaltenen Klasse SCCPlugin

```
public class SCCPlugin : GraphAlgorithmPlugin<SCCVertex, Edge<SCCVertex>>
{
    public override string AlgorithmName => "Strong Connected
    Components";

    public override GraphDirectionType CompatibleGraphDirections =>
    GraphDirectionType.Directed;

    private readonly ExposableList publicL = new ExposableList("L");
    private readonly ExposableList publicQ = new ExposableList("Q");

    protected override async Task RunAlgorithm(SCCVertex startVertex)
    {
        ExposedLists.Clear();
        publicQ.Clear();
        publicL.Clear();
        ExposedLists.Add(publicQ);
        ExposedLists.Add(publicL);

        try
        {
            Stack<SCCVertex> l = new Stack<SCCVertex>();
            int u = 0;
            Progress = 0;
            ProgressText = "Initializing...";

            u = await RunDFS(startVertex, u, l);
            foreach (SCCVertex s in Graph.Vertices)
            {
                if (!s.Marked)
                {
                    u = await RunDFS(s, u, l);
```

```
        }
    }

    List<Edge<SCCVertex>> edges = new
    List<Edge<SCCVertex>>(Graph.Edges);
    GraphLayout.KeepPositions = true;
    await MakeAlgorithmStep(() =>
    {
        foreach (SCCVertex s in Graph.Vertices)
        {
            s.VertexContent = "";
            s.Marked = false;
            Graph.ClearEdges(s);
        }
        Graph.AddEdgeRange(edges.Select(x =>
        new Edge<SCCVertex>(x.Target, x.Source)));
        ProgressText = "Invert edges...";
        Progress = ((2 * Graph.Vertices.Count() + 1) / (3.0 *
        Graph.Vertices.Count() + 1.0)) * 100;
    }, () =>
    {
        GraphLayout.KeepPositions = true;
        foreach (SCCVertex s in Graph.Vertices)
        {
            s.UpdateLayout();
            s.Marked = true;
            s.UpdateContent();
            Graph.ClearEdges(s);
        }
        Graph.AddEdgeRange(edges);
        ProgressText = "DFS...";
        Progress = (2 * Graph.Vertices.Count() / (3.0 *
        Graph.Vertices.Count() + 1.0)) * 100;
    });
    GraphLayout.KeepPositions = false;

    int sccID = 1;
```

```
            u = 0;
            while (l.Count > 0)
            {
                SCCVertex s = l.Pop();

                if (!s.Marked)
                {
                    u = await IdentifySCCs(s, sccID, u);
                    sccID += 1;
                } else
                {
                    await MakeAlgorithmStep(() =>
                    {
                        publicL.Remove(s);
                    }, () =>
                    {
                        publicL.Insert(0, s);
                    });
                }
            }
        }
        catch (TaskCanceledException) { }
        catch (OperationCanceledException) { }
    }

    private void GetDescendants(SCCVertex vertex, Dictionary<SCCVertex,
HashSet<SCCVertex>> descendants)
    {
        if (!descendants.ContainsKey(vertex))
        {
            descendants.Add(vertex, new HashSet<SCCVertex>());
        }
        foreach (SCCVertex descendantVertex in Graph.OutEdges(vertex).
        Select(x => x.Target))
        {
            descendants[vertex].Add(descendantVertex);
        }
```

```
}

private async Task<int> RunDFS(SCCVertex startVertex, int u,
Stack<SCCVertex> l)
{
    Stack<SCCVertex> q = new Stack<SCCVertex>();
    Dictionary<SCCVertex, HashSet<SCCVertex>> descendantVertices =
    new Dictionary<SCCVertex, HashSet<SCCVertex>>();
    u += 1;

    await MakeAlgorithmStep(() =>
    {
        q.Push(startVertex);
        publicQ.Insert(0, startVertex);
        startVertex.Marked = true;
        GetDescendants(startVertex, descendantVertices);
        startVertex.PushTime = u;
        Progress = (u / (3.0 * Graph.Vertices.Count() + 1.0)) * 100;
        ProgressText = "DFS...";
    }, () =>
    {
        q.Pop();
        publicQ.Remove(startVertex);
        startVertex.Marked = false;
        descendantVertices.Remove(startVertex);
        startVertex.PushTime = 0;
        Progress = 0;
        ProgressText = "Initializing...";
    });

    while (q.Count > 0)
    {
        SCCVertex v = q.Peek();
        if (descendantVertices.ContainsKey(v) &&
        descendantVertices[v].Count > 0)
        {
            SCCVertex w = descendantVertices[v].First();
```

```
            descendantVertices[v].Remove(w);
            if (!w.Marked)
            {
                await MakeAlgorithmStep(() =>
                {
                    q.Push(w);
                    publicQ.Insert(0, w);
                    w.Marked = true;
                    GetDescendants(w, descendantVertices);
                    u += 1;
                    w.PushTime = u;
                    Progress = (u / (3.0 * Graph.Vertices.Count() +
                    1.0)) * 100;
                }, () =>
                {
                    q.Pop();
                    publicQ.Remove(w);
                    w.Marked = false;
                    descendantVertices.Remove(w);
                    u -= 1;
                    w.PushTime = 0;
                    Progress = (u / (3.0 * Graph.Vertices.Count() +
                    1.0)) * 100;
                });
            }
        }
        else
        {
            await MakeAlgorithmStep(() =>
            {
                q.Pop();
                publicQ.Remove(v);
                l.Push(v);
                publicL.Insert(0, v);
                u += 1;
                v.PopTime = u;
                Progress = (u / (3.0 * Graph.Vertices.Count() +
```

```
                1.0)) * 100;
            }, () =>
            {
                q.Push(v);
                publicQ.Insert(0, v);
                if (l.Count > 0)
                    l.Pop();
                publicL.Remove(v);
                u -= 1;
                v.PopTime = 0;
                Progress = (u / (3.0 * Graph.Vertices.Count() +
                1.0)) * 100;
            });
        }
    }

    return u;
}

private async Task<int> IdentifySCCs(SCCVertex startVertex,
int sccID, int u)
{
    Stack<SCCVertex> q = new Stack<SCCVertex>();
    Dictionary<SCCVertex, HashSet<SCCVertex>> descendantVertices =
    new Dictionary<SCCVertex, HashSet<SCCVertex>>();
    u += 1;

    await MakeAlgorithmStep(() =>
    {
        q.Push(startVertex);
        publicQ.Insert(0, startVertex);
        publicL.Remove(startVertex);
        startVertex.Marked = true;
        GetDescendants(startVertex, descendantVertices);
        startVertex.SccID = sccID;
        ProgressText = "Identify SCCs...";
        Progress = ((2 * Graph.Vertices.Count() + 1 + u) / (3.0 *
```

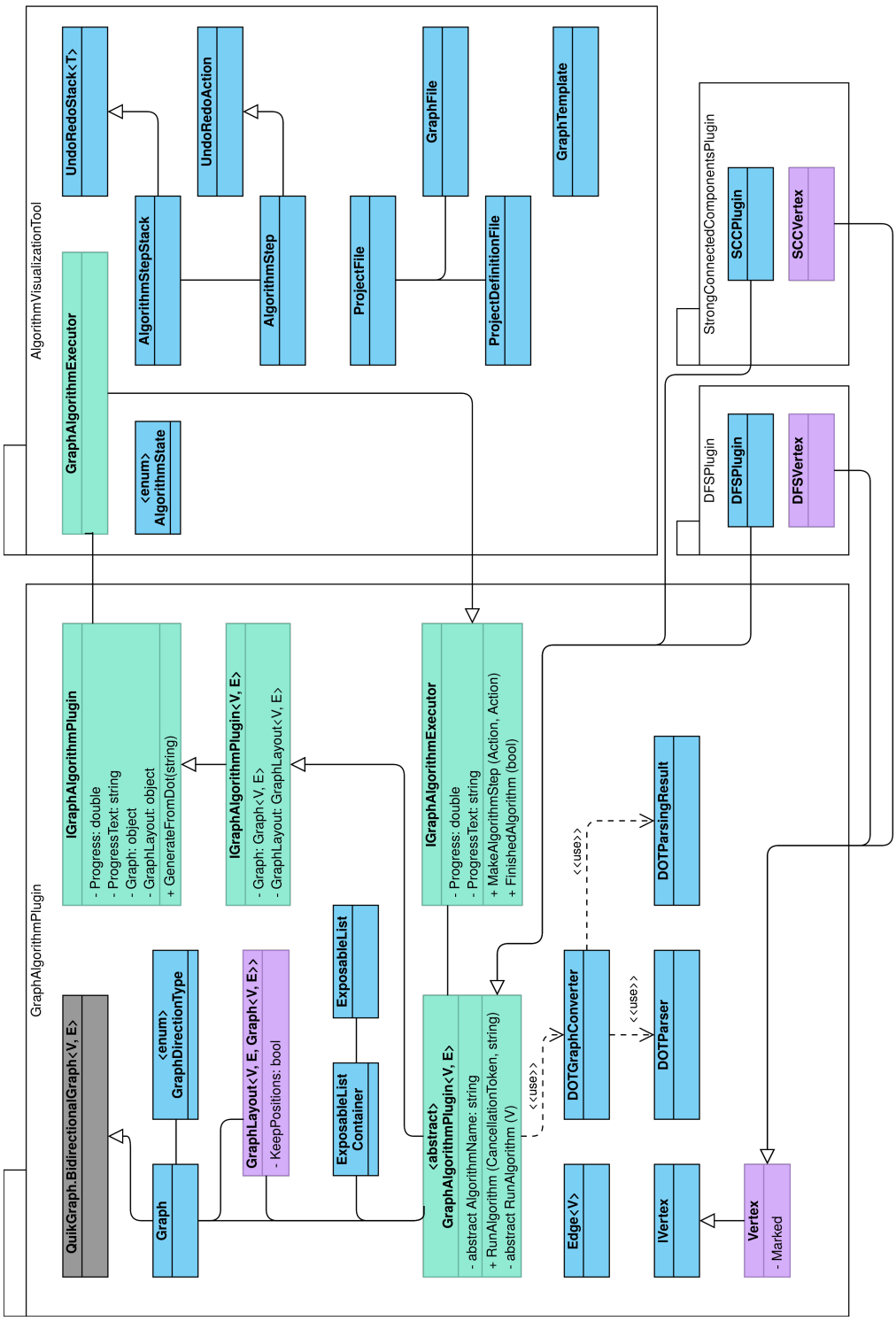
```
        Graph.Vertices.Count() + 1.0)) * 100;
    }, () =>
    {
        if (q.Count > 0)
            q.Pop();
        publicQ.Remove(startVertex);
        publicL.Insert(0, startVertex);
        startVertex.Marked = false;
        descendantVertices.Remove(startVertex);
        startVertex.SccID = 0;
        startVertex.VertexContent = "";
        ProgressText = "Invert edges...";
        Progress = ((2 * Graph.Vertices.Count() + u) / (3.0 *
        Graph.Vertices.Count() + 1.0)) * 100;
    });

    while (q.Count > 0)
    {
        SCCVertex v = q.Peek();
        if (descendantVertices.ContainsKey(v) &&
        descendantVertices[v].Count > 0)
        {
            SCCVertex w = descendantVertices[v].First();
            descendantVertices[v].Remove(w);
            if (!w.Marked)
            {
                await MakeAlgorithmStep(() =>
                {
                    q.Push(w);
                    publicQ.Insert(0, w);
                    w.Marked = true;
                    GetDescendants(w, descendantVertices);
                    w.SccID = sccID;
                    u += 1;
                    Progress = ((2 * Graph.Vertices.Count() + 1 +
                    u) / (3.0 * Graph.Vertices.Count() + 1.0)) * 100;
                }, () =>
```

```
                    {
                        q.Pop();
                        publicQ.Remove(w);
                        w.Marked = false;
                        descendantVertices.Remove(w);
                        w.SccID = 0;
                        w.VertexContent = "";
                        u -= 1;
                        Progress = ((2 * Graph.Vertices.Count() + 1 +
                        u) / (3.0 * Graph.Vertices.Count() + 1.0)) * 100;
                    });
                }
            }
            else
            {
                await MakeAlgorithmStep(() =>
                {
                    q.Pop();
                    publicQ.Remove(v);
                }, () =>
                {
                    q.Push(v);
                    publicQ.Insert(0, v);
                });
            }
        }

        return u;
    }
}
```

A.5 Vereinfachtes Klassendiagramm des Algorithm Visualization Tool



Literaturverzeichnis

Beck, Hannah (2019). *Grundlagen der Graphdatenverarbeitung*. URL: http://www.imn.htwk-leipzig.de/~kudrass/Lehrmaterial/Oberseminar/2019/05-Abstract-GraphProcessing.pdf.

Karin Anna Hummel, Andrea Hess und Harald Meyer (2010). *Mobilität im „Future Internet". Geräte- und Ressourcenmobilität: Herausforderungen und Techniken im Überblick*.

Körner, Heiko (2009). *Algorithmen auf Graphen. Durchmusterungen und kürzeste Wege*.

Rabérin, Alexandre (2021a). *GraphShape*. URL: https://github.com/KeRNeLith/GraphShape.

— (2021b). *QuikGraph*. URL: https://github.com/KeRNeLith/QuikGraph.

Ullenboom, Christian (2020). *Java ist auch eine Insel. Einführung, Ausbildung, Praxis*. Rheinwerk Verlag. ISBN: 978-3-8362-7737-2.