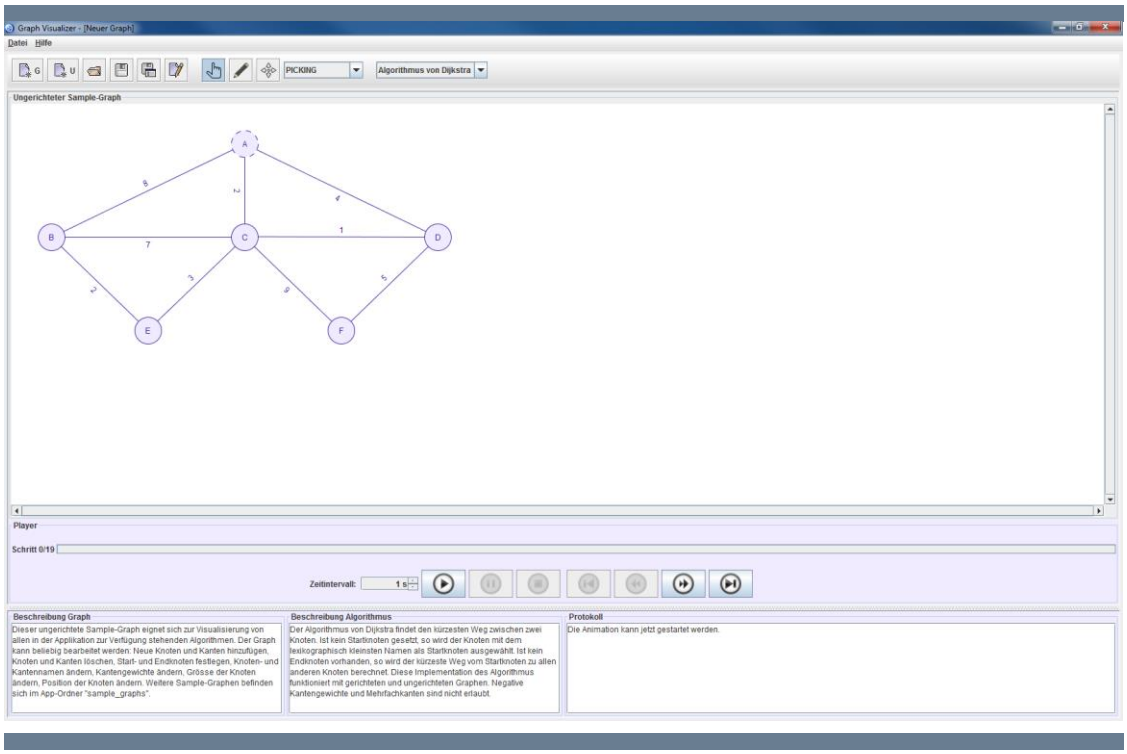


# Graph Visualizer

## Eine Applikation zur Visualisierung von Graphen und Algorithmen

**Projektdokumentation**


| | |
|---|---|
| Studiengang: | Informatik, Modul BTI7301 (Projekt 1), HS 2013/14 |
| Autor: | Patrick Kofmel (kofmp1@bfh.ch) |
| Betreuer: | Jürgen Eckerle (juergen.eckerle@bfh.ch) |
| Datum: | 13. Juni 2014 |

# Inhaltsverzeichnis

# 1 Einleitung

Dieses Dokument enthält die Projekt-Dokumentation zur Applikation *Graph Visualizer*. Zu diesem Projekt existiert ein *GitHub-Repository* (Link vgl. Anhang). Dort sind neben dieser Dokumentation auch alle anderen für das Projekt relevanten Artefakte abgelegt (Quellcode-Dateien, API-Dokumentation, UML-Diagramme).

| Verwendete Technologien und Tools: | |
|---|---|
| Programmiersprache: | Java (Java SE 7) |
| Frameworks: | Java Universal Network/Graph Framework (JUNG) |
| Entwicklungsumgebung: | Eclipse IDE (Kepler Service Release 2) |
| Build-Management-Tool: | Apache Maven (Maven-Tools Eclipse) |
| Version-Control-System: | GIT (GIT-Client Eclipse) |
| Sonstige Tools: | MS Word 2010, MS Visio 2010 |

Tabelle 1: Technologien und Tools

# 2 Requirements und Use Cases

In diesem Kapitel werden zuerst in kurzer Form die *User Requirements* formuliert (basierend auf der Projektbeschreibung des Dozenten). Dann folgt eine Erläuterung der *System Requirements*. Diese beschreiben in detaillierter Form die Systemfunktionen und Systemeigenschaften. Basierend auf den Requirements werden dann die wichtigsten *Use Cases* in einem Diagramm dargestellt.

### 2.1 User Requirements

Es soll eine Software erstellt werden, welche gerichtete und ungerichtete Graphen erstellen und darstellen kann. Die Software soll es ermöglichen, Graphen aus einer Datei zu laden, zu bearbeiten und in einer Datei zu speichern.
Gleichzeitig soll die Software zur Visualisierung der Traversierung von Graphen dienen. Verschiedene Algorithmen wie z. B. Tiefensuche, Breitensuche, Dijkstra oder Kruskal sollen mit diesem Werkzeug auf einfache Weise visualisierbar werden. Die Visualisierung der Algorithmen soll dabei entweder in einer Animation oder Schritt für Schritt möglich sein. Die Applikation soll sich als didaktisches Hilfsmittel für beliebige Graphen-Algorithmen eignen.

### 2.2 System Requirements

Die System Requirements bestehen zum grössten Teil aus *funktionalen Requirements*. Diese beschreiben die konkrete Funktionalität der Applikation. Es gibt aber auch einige *nicht-funktionale Requirements*, welche allgemeine Systemeigenschaften beschreiben.

### 2.2.1 Funktionale Requirements

| Neuen Graphen erstellen: |
|---|
| Gerichtet oder ungerichtet |
| Gewichtet oder ungewichtet (wenn ungewichtet, dann ist das Gewicht bei allen Kanten = 1) |
| Schlingen sind bei gerichteten und ungerichteten Graphen zulässig |
| Mehrfachkanten und Hyperkanten sind *nicht* zulässig |

Tabelle 2: Funktionale Requirements - neuen Graphen erstellen

| Graph-Elemente (Knoten und Kanten) bearbeiten: | |
|---|---|
| Eindeutigen Name (ID) festlegen und anzeigen | |
| Knoten: | Neuen Knoten einfügen |
| | Knoten löschen (inzidente Kanten werden automatisch gelöscht) |
| | Höhe und Breite ändern:<br>Darstellung als Kreis oder Ellipse |
| | Maximal einen Startknoten festlegen (optional):<br>Darstellung als gestrichelter Kreis oder Ellipse |
| | Maximal einen Endknoten festlegen (optional):<br>Darstellung als gepunkteter Kreis oder Ellipse |
| | Position ändern durch *Drag and Drop* |
| Kanten: | Neue Kante zwischen zwei Knoten einfügen |
| | Kante löschen |
| | Darstellung als Pfeil (gerichtet) oder Linie (ungerichtet) |
| | Gewicht festlegen und anzeigen |

Tabelle 3: Funktionale Requirements - Graph-Elemente bearbeiten

| Graphen bearbeiten: |
|---|
| Name und Beschreibung ändern und anzeigen |
| Verschiebung, Drehung, Scherung |

Tabelle 4: Funktionale Requirements - Graphen bearbeiten

| Algorithmus auswählen (Vorberechnung der Animations-Schritte): | |
|---|---|
| Allgemein: | Name und Beschreibung des gewählten Algorithmus anzeigen. |
| | Falls kein Startknoten festgelegt wurde, wähle den Knoten mit dem lexikografisch kleinsten Namen als Startknoten. |
| Tiefensuche (DFS): | Falls ein Endknoten festgelegt wurde und dieser vom Startknoten aus erreichbar ist:<br>Besuche durch *Tiefensuche* alle Knoten bis der Endknoten erreicht wurde und speichere die einzelnen Schritte in einer Datenstruktur. |
| | Falls *kein* Endknoten festgelegt wurde oder der Endknoten vom Startknoten aus *nicht* erreichbar ist:<br>Besuche durch *Tiefensuche* alle vom Startknoten aus erreichbaren Knoten und speichere die einzelnen Schritte in einer Datenstruktur. |

| | |
|---|---|
| | auf gerichtete und ungerichtete Graphen anwendbar |
| Breitensuche (BFS): | Falls ein Endknoten festgelegt wurde und dieser vom Startknoten aus erreichbar ist:<br>Besuche durch *Breitensuche* alle Knoten bis der Endknoten erreicht wurde und speichere die einzelnen Schritte in einer Datenstruktur. |
| | Falls *kein* Endknoten festgelegt wurde oder der Endknoten vom Startknoten aus *nicht* erreichbar ist:<br>Besuche durch *Breitensuche* alle vom Startknoten aus erreichbaren Knoten und speichere die einzelnen Schritte in einer Datenstruktur. |
| | auf gerichtete und ungerichtete Graphen anwendbar |
| Dijkstra: | Falls ein Endknoten festgelegt wurde und dieser vom Startknoten aus erreichbar ist:<br>Suche mit dem *Dijkstra-Algorithmus* den kürzesten Weg vom Start- zum Endknoten und speichere die einzelnen Schritte in einer Datenstruktur. |
| | Falls *kein* Endknoten festgelegt wurde oder der Endknoten vom Startknoten aus *nicht* erreichbar ist:<br>Suche mit dem *Dijkstra-Algorithmus* den kürzesten Weg vom Startknoten zu allen anderen Knoten und speichere die einzelnen Schritte in einer Datenstruktur. |
| | auf gerichtete und ungerichtete Graphen anwendbar |
| Kruskal: | Berechne mit dem *Kruskal-Algorithmus* einen minimal aufspannenden Wald und speichere die einzelnen Schritte in einer Datenstruktur. |
| | nur auf ungerichtete Graphen anwendbar |

Tabelle 5: Funktionale Requirements - Algorithmus auswählen

| Schrittweise Traversierung (mit Hilfe der vorberechneten Datenstruktur): | |
|---|---|
| Visualisierung des Graphen bei jedem Schritt anpassen durch Farbänderung | |
| Statusmeldung und Zwischenresultate für jeden Schritt anzeigen | |
| Fortschritt der Traversierung grafisch darstellen (Progress Bar) | |
| Navigationsmöglichkeiten: | Zum Anfang |
| | Zum Ende |
| | Einen Schritt vorwärts |
| | Einen Schritt zurück |
| Nach dem letzten Schritt: | Falls ein Endknoten festgelegt wurde, zeige den Weg vom Start- zum Endknoten (DFS, BFS, Dijkstra) |
| | Zeige den minimal aufspannenden Wald bei Kruskal |

Tabelle 6: Funktionale Requirements - schrittweise Traversierung

| Animation anzeigen: | |
|---|---|
| Abspielgeschwindigkeit: | Zeitintervall festlegen (in Sekunden) |
| Abspielen: | Animation schrittweise abspielen und GUI sperren |
| Anhalten: | Animation an aktueller Position anhalten |
| Abspielung fortsetzen: | Fortsetzung der Animation ab aktueller Position |
| Abbrechen: | Zum Anfang der Traversierung zurückkehren und GUI aktivieren |
| Nach dem letzten Schritt: | An aktueller Position anhalten und GUI aktivieren |

Tabelle 7: Funktionale Requirements - Animation anzeigen

| IO-Operationen: |
|---|
| Graphen aus Datei laden |
| Graphen in Datei speichern |
| Spezielles XML-Format (GraphML) wird zur Speicherung verwendet |

Tabelle 8: Funktionale Requirements - IO-Operationen

### 2.2.2 Nicht-funktionale Requirements

| Einige nicht-funktionale Requirements: | |
|---|---|
| Erweiterbarkeit: | zusätzliche Algorithmen hinzufügen mit minimalen Änderungen am bestehenden Code |
| Usability: | Shortcuts für alle wichtigen Funktionalitäten |
| | Tooltips mit erklärendem Text zu allen wichtigen GUI-Elementen |
| Input-Validierung: | Validierung der XML-Datei beim Laden eines Graphen |
| | Validierung von Benutzereingaben |
| Warnhinweise: | Beim Überschreiben einer existierenden Datei |
| | Beim Verwerfen nicht gespeicherter Änderungen |

Tabelle 9: Nicht-funktionale Requirements

### 2.3 Use Cases

Im folgenden Diagramm werden die wichtigsten *Use Cases* dargestellt. Sie zeigen die Systeminteraktionen mit dem User und dem Betriebssystem sowie die Beziehungen untereinander.

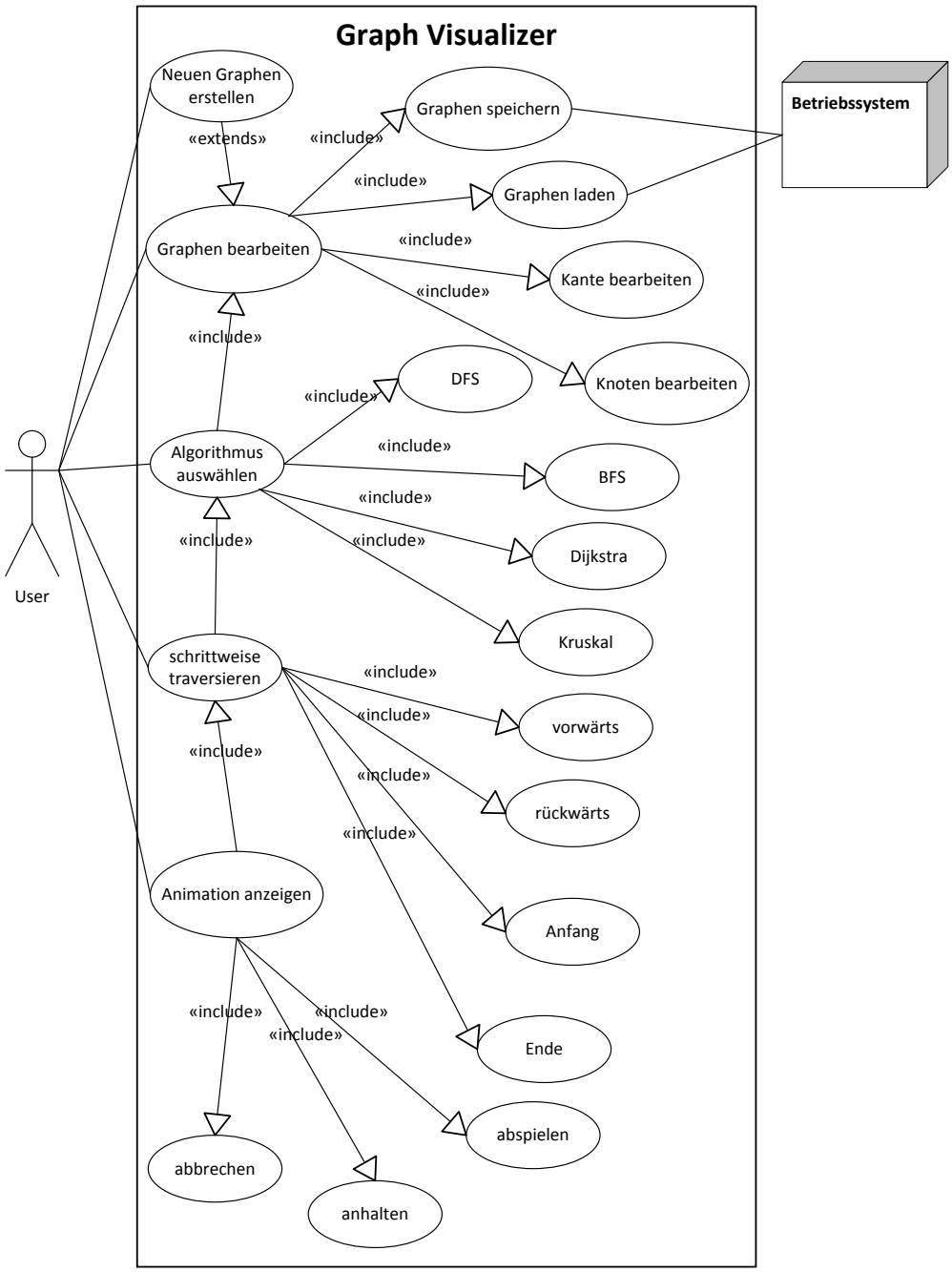


Abbildung 1: Use Case Diagramm

# 3 Systemarchitektur und Design

In diesem Kapitel folgen einige Erläuterungen zur Systemarchitektur und den verwendeten *Design Patterns*. Für die Systemarchitektur wurden die folgenden *Architectural Patterns* verwendet:

- Schichtenarchitektur
- Model-View-Controller (MVC) Pattern

Es folgt eine Liste mit den verwendeten *Design Patterns*:

- (Static) Factory Method
- Composite
- Decorator
- Façade
- Observer
- Iterator
- Strategy
- Command

## 3.1 Schichtenarchitektur

Die Applikation weist eine Schichtenarchitektur auf. Jede Schicht bietet Dienste an, die von den darüber liegenden Schichten verwendet werden können. Der Aufbau ist in der folgenden Abbildung dargestellt:

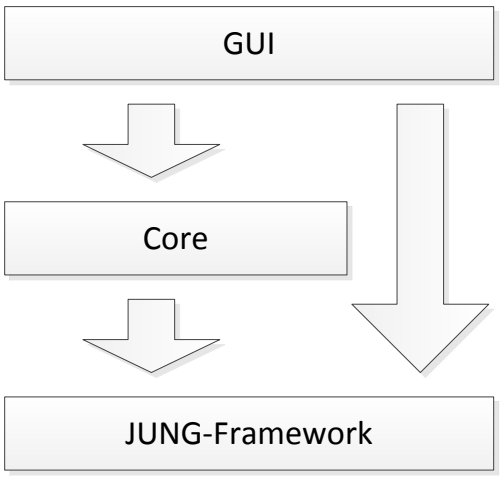


Abbildung 2: Schichtenarchitektur

Das JUNG-Framework stellt viele nützliche Interfaces und Klassen zur Manipulation und Visualisierung von Graphen zur Verfügung. Die Core- und die GUI-Schicht nutzen diese Klassen und Interfaces an sehr vielen Stellen.

Die Core-Schicht bietet der GUI-Schicht grundlegende Dienste an:

- Erzeugung von Instanzen der Datenstruktur *Graph* (Klasse `GraphFactory`)
- Laden eines Graphs aus einer Datei (Interface `ICore`)
- Speichern eines Graphs in einer Datei (Interface `ICore`)
- Ausführen eines Algorithmus auf einem Graphen und Rückgabe eines Iterators über alle Animations-Schritte (Interface `ICore`)
- Abrufen von Informationen zu bestimmten Algorithmen (Name und Beschreibung) (Interface `ICore`)

Die GUI-Schicht ist für die Darstellung des User Interfaces und des Graphen zuständig. Für die Darstellung des Graphen nutzt die GUI-Schicht zum Teil direkt Klassen aus dem JUNG-Framework. Auch alle Event-Handler, die auf User-Interaktionen reagieren, befinden sich in der GUI-Schicht.

### 3.2 Model-View-Controller (MVC) Pattern

Für die Interaktion des User Interfaces mit den anderen Systemkomponenten wurde das klassische Model-View-Controller Pattern verwendet. Die relevanten Klassen für das MVC befinden sich in der GUI-Schicht. Die folgende Abbildung zeigt wichtige Schnittstellen und Klassen von den einzelnen Komponenten des MVC. Es werden zur Vereinfachung nur Assoziations-, Realisierungs- und Vererbungsbeziehungen dargestellt. Die View-Klassen befinden sich auf der rechten Seite, die Controller-Klassen und Interfaces ganz oben und die Model-Klassen und Interfaces unten links.

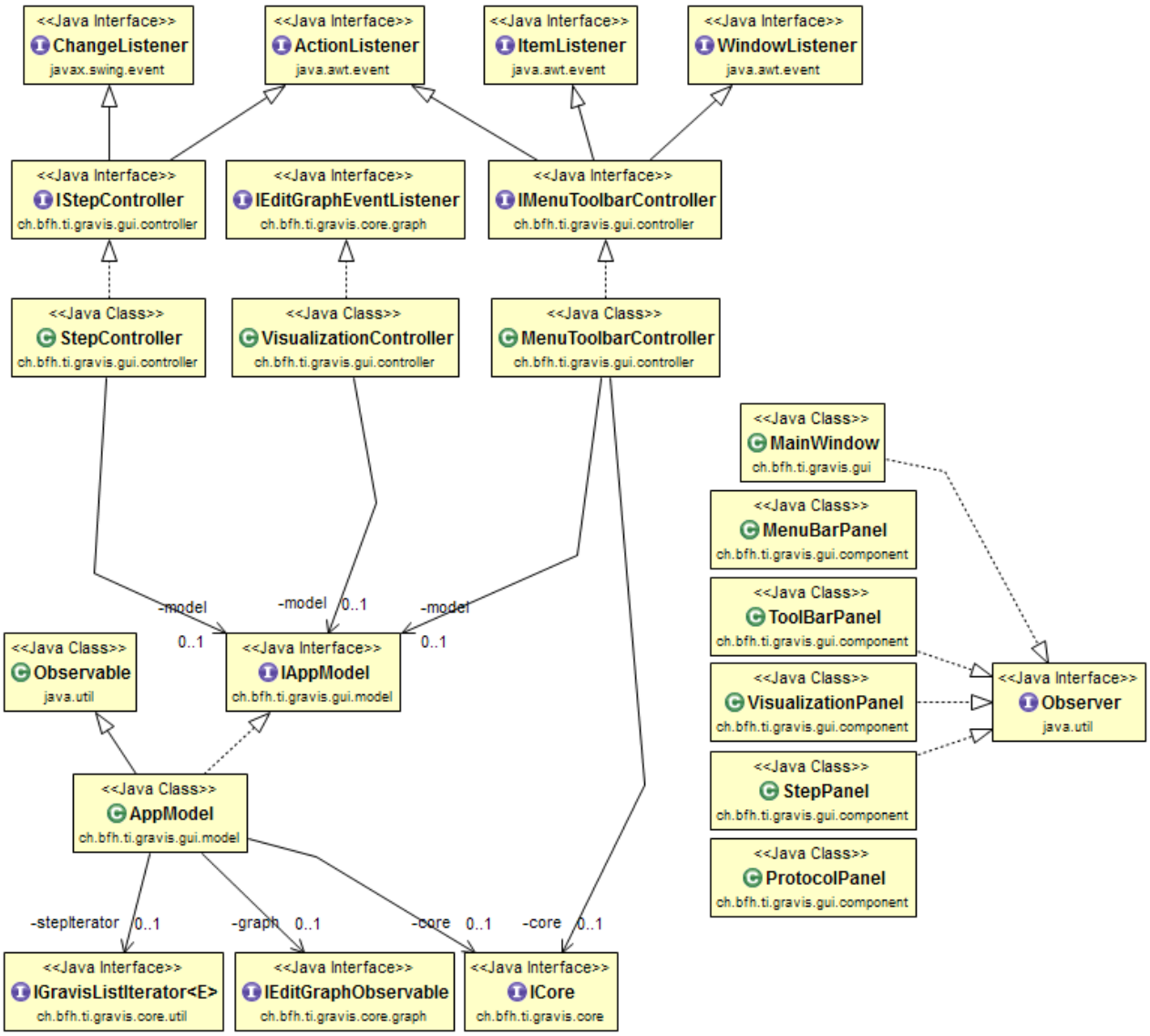


Abbildung 3: MVC-Pattern

Die *View-Klassen* halten selbst keine Daten und sind nur für die grafische Darstellung der GUI-Komponenten zuständig. Einige implementieren das `Oberserver`-Interface von Java und können so über Zustandsänderungen im Model benachrichtigt werden. Zudem haben die View-Klassen keine Kenntnisse über die vorhandenen konkreten Controller- und Model-Klassen.
Die konkrete *Model-Klasse* `AppModel` implementiert das Model-Interface `IAppModel`. Sie verwaltet die darzustellenden Daten und den aktuellen Zustand der Applikation. Die Klasse `AppModel` delegiert Daten- und Zustandsänderungen weiter an Instanzen von anderen Model-Klassen. So hat einerseits jede GUI-Komponente ein Model, das ihren aktuellen Zustand verwaltet. Andererseits hat `AppModel` aber auch Zugriff auf Interfaces der Core-Schicht. `AppModel` besitzt Felder für den aktuellen Graphen (Interface `IEditGraphObservable`), für den aktuellen Step-Iterator (Interface `IGravisListIterator`) und

für das Interface ICore. Zudem erbt AppModel von der Java-Klasse Observable und kann so alle registrierten Observer über Änderungen benachrichtigen (*Observer-Pattern*).
Die *Controller-Klassen* verarbeiten verschiedene Events. Dazu implementieren sie entsprechende Event-Listeners. Die Controller-Klassen können nun über eine Instanz vom Typ IAppModel Änderungen am Model vornehmen. Auch haben sie bei Bedarf Zugriff auf das Interface ICore. Einige Events werden direkt durch User-Interaktionen ausgelöst (z. B. Klick auf einen Button). Ein Event kann aber auch bei Änderungen in den Core-Klassen ausgelöst werden. Zum Beispiel werden beim Hinzufügen eines Knotens zu einem Graphen, der die Schnittstelle IEditGraphObservable implementiert, alle registrierten Listener vom Typ IEditGraphEventListener benachrichtigt.
Um das Design zu vereinfachen, wurde an einigen Stellen auf die strikte Trennung zwischen View und Controller verzichtet. So wurden einige Event-Listener, die keine Interaktion mit dem Model erfordern, direkt im View implementiert (z. B. in der Klasse MenuBarPanel). Einige Events, die das Bearbeiten des Graphen betreffen (z. B. neuen Knoten hinzufügen), werden direkt vom JUNG-Framework abgefangen und bearbeitet. Alle Dialog- und Popup-Komponenten werden bei Bedarf temporär konstruiert und implementieren ihre meist sehr kurzen Event-Listener selbst. Die Validierung von Benutzereingaben erfolgt ebenfalls direkt in den Dialog-Klassen.

### 3.3 Package-Struktur

| Alle vorhandenen Packages im Überblick: | |
|---|---|
| gravis | Enthält die Start-Klasse mit der main-Methode |
| gravis.core | Alle Klassen der Core-Schicht |
| gravis.core.algorithm | Algorithmen-Implementationen, Klasse AlgorithmFactory |
| gravis.core.graph | Implementationen der Datenstruktur Graph, Klasse GraphIOManager und GraphFactory |
| gravis.core.graph.comparator | Implementationen des Comparator-Interfaces zum Vergleichen von Graph-Elementen |
| gravis.core.graph.item | Abstrakte Basisklassen und Interfaces für Knoten und Kanten |
| gravis.core.graph.item.edge | Implementationen von Kanten, Klasse EdgeFactory |
| gravis.core.graph.item.vertex | Implementationen von Knoten, Klasse VertexFactory |
| gravis.core.graph.transformer | Instanzen von Transformer-Klassen transformieren Objekte eines bestimmten Input-Datentyps in Objekte eines bestimmten Output-Datentyps (entspricht Funktoren in C++, Delegates in C#) |
| gravis.core.step | Klassen zur Erzeugung von Animations-Schritten aus Sequenzen von Graph-Elementen |
| gravis.core.util | Wichtige Konstanten, Interface IGravisListIterator, Klasse ValueTransformer |
| gravis.gui | Alle Klassen der GUI-Schicht, insbesondere View-Klasse MainWindow |
| gravis.gui.component | View-Klassen für verschiedene GUI-Komponenten |
| gravis.gui.controller | Controller-Klassen und Interfaces |
| gravis.gui.dialog | View-Klassen für Dialoge |
| gravis.gui.model | Model-Klassen und Interfaces |
| gravis.gui.popup | View-Klassen für Popup-Menüs |
| gravis.gui.verifier | Klassen für Input-Validierung |
| gravis.gui.visualization | Klassen für die Graph-Visualisierung (abgeleitet von Klassen des JUNG-Frameworks) |

Tabelle 10: Package-Struktur

### 3.4 Core-Interface

Das Interface `ICore` ermöglicht den Zugriff auf Methoden der Core-Schicht. Es ist ein einheitliches und vereinfachtes Interface zu einer Menge von anderen Core-Klassen und Interfaces (*Façade Pattern*). Wichtig ist in diesem Zusammenhang auch die Klasse `GraphFactory`, mit welcher von aussen neue Graph-Instanzen konstruiert werden können. Die folgende Abbildung zeigt Das Interface `ICore` in Beziehung zu anderen wichtigen Klassen und Interfaces der Core-Schicht. Die gestrichelten Linien mit spitzem Pfeil stellen Abhängigkeitsbeziehungen dar.

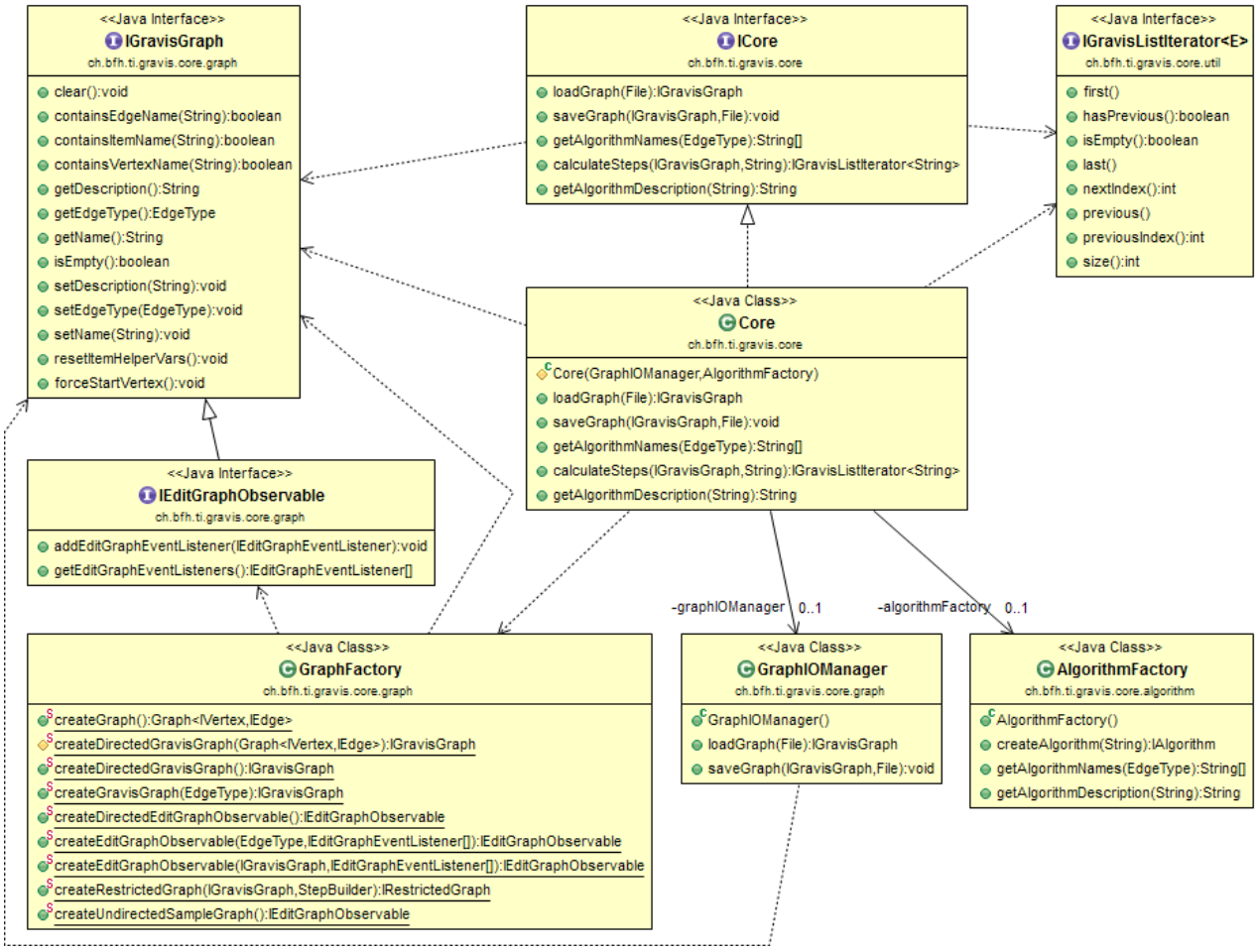


Abbildung 4: Core-Interface

### 3.5 Graph-Klassen

Das Interface `IGravisGraph` erweitert das Interface `Graph<V,E>` aus dem JUNG-Framework und fügt zusätzliche Methoden hinzu. Um einem Graphen Listener vom Typ `IEditGraphEventListener` hinzuzufügen, muss das Interface `IEditGraphObservable` verwendet werden, welches `IGravisGraph` erweitert.
Die konkrete Klasse `GravisGraph` dekoriert einen gegebenen Graphen vom Typ `Graph<IVertex,IEdge>` (*Decorator Pattern*). Die Graph-Instanz wird im Konstruktor von `GravisGraph` übergeben. Weiter dekoriert die Klasse `EditGraphDecorator` einen Graphen vom Typ `IGravisGraph`.
Bei Graphen vom Typ `IRestrictedGraph` können keine neuen Knoten und Kanten hinzugefügt werden. Diese Graphen werden in den Algorithmen-Klassen verwendet. Auch in der Klasse `RestrictedGraph` wird ein `IGravisGraph` dekoriert.

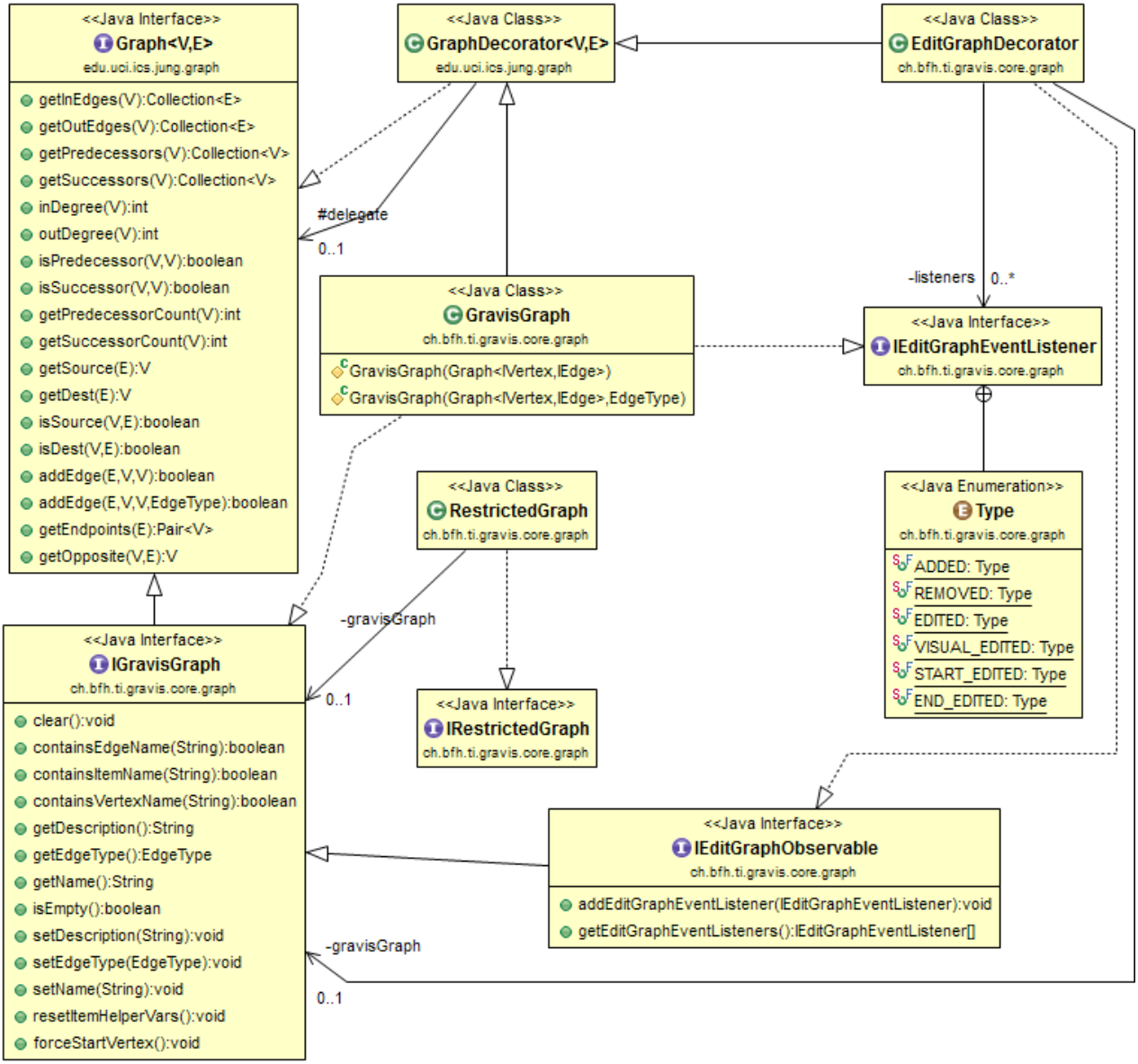


Abbildung 5: Graph-Klassen

### 3.6 Graph-Elemente

Die Knoten- und Kanten-Klassen eines Graphen implementieren die Interfaces `IGraphItem`, `IEditItemObservable` und `IRestrictedGraphItem`. Das Interface `IRestrictedGraphItem` bietet nur eingeschränkten Zugriff auf ein Graph-Element, während in `IGraphItem` alle Methoden zur Verfügung stehen. *Restricted*-Elemente werden bei Graphen vom Typ `IRestrictedGraph` verwendet.
Durch das Interface `IEditItemObservable` können einem Graph-Element Listener vom Typ `IEditGraphEventListener` hinzugefügt werden. Diese werden dann beim Eintreten bestimmter Events benachrichtigt (z. B. wenn der Name geändert wird).
Über das Interface `IGraphItem` kann auch der Zustand `ItemState` eines Graph-Elements gesetzt oder abgerufen werden. Dieser Zustand bestimmt beim Ausführen der Animation die Farbe und den Kommentar des Graph-Elementes.

#### 3.6.1 Knoten

Die Knoten-Klassen des Graphen implementieren zusätzlich die Interfaces `IVertex` und `IRestrictedVertex`. Diese geben Zugriff auf Methoden, welche nur für Knoten relevant sind (z.B. Start- oder Endknoten setzten). Das Interface `IRestrictedVertex` beschränkt den Zugriff auf getter-Methoden. Mit den Methoden der Klasse `VertexFactory` können Vertex-Instanzen konstruiert werden (*Factory Method Pattern*).

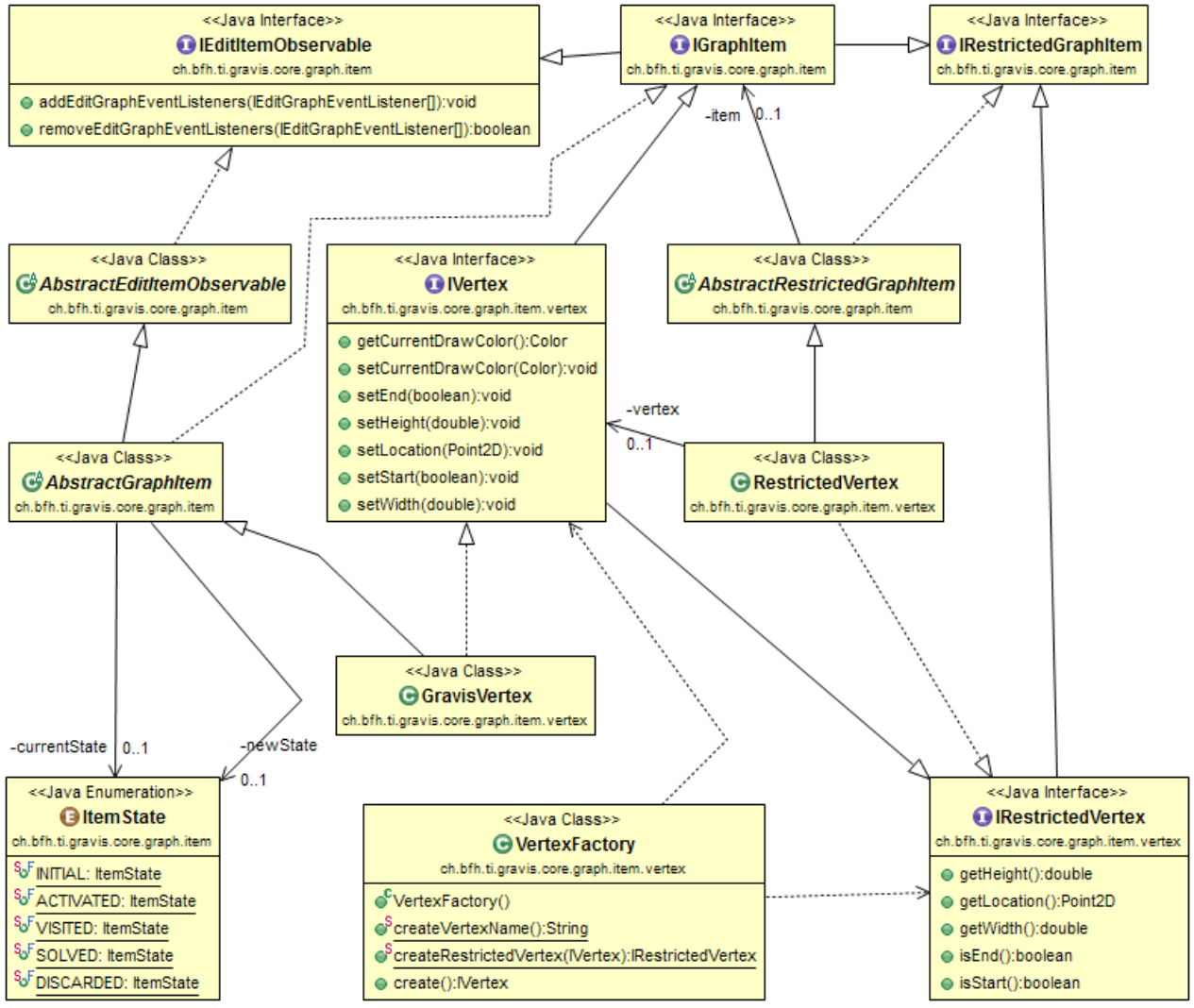


Abbildung 6: Vertex-Klassen

### 3.6.2 Kanten

Auch die Kanten-Klassen des Graphen implementieren zusätzlich Interfaces: `IEdge` und `IRestrictedEdge`. So kann das Gewicht einer Kante gesetzt oder abgefragt werden. Das Interface `IRestrictedEdge` beschränkt den Zugriff auf die getter-Methode. Mit den Methoden der Klasse `EdgeFactory` können Edge-Instanzen konstruiert werden (*Factory Method Pattern*).

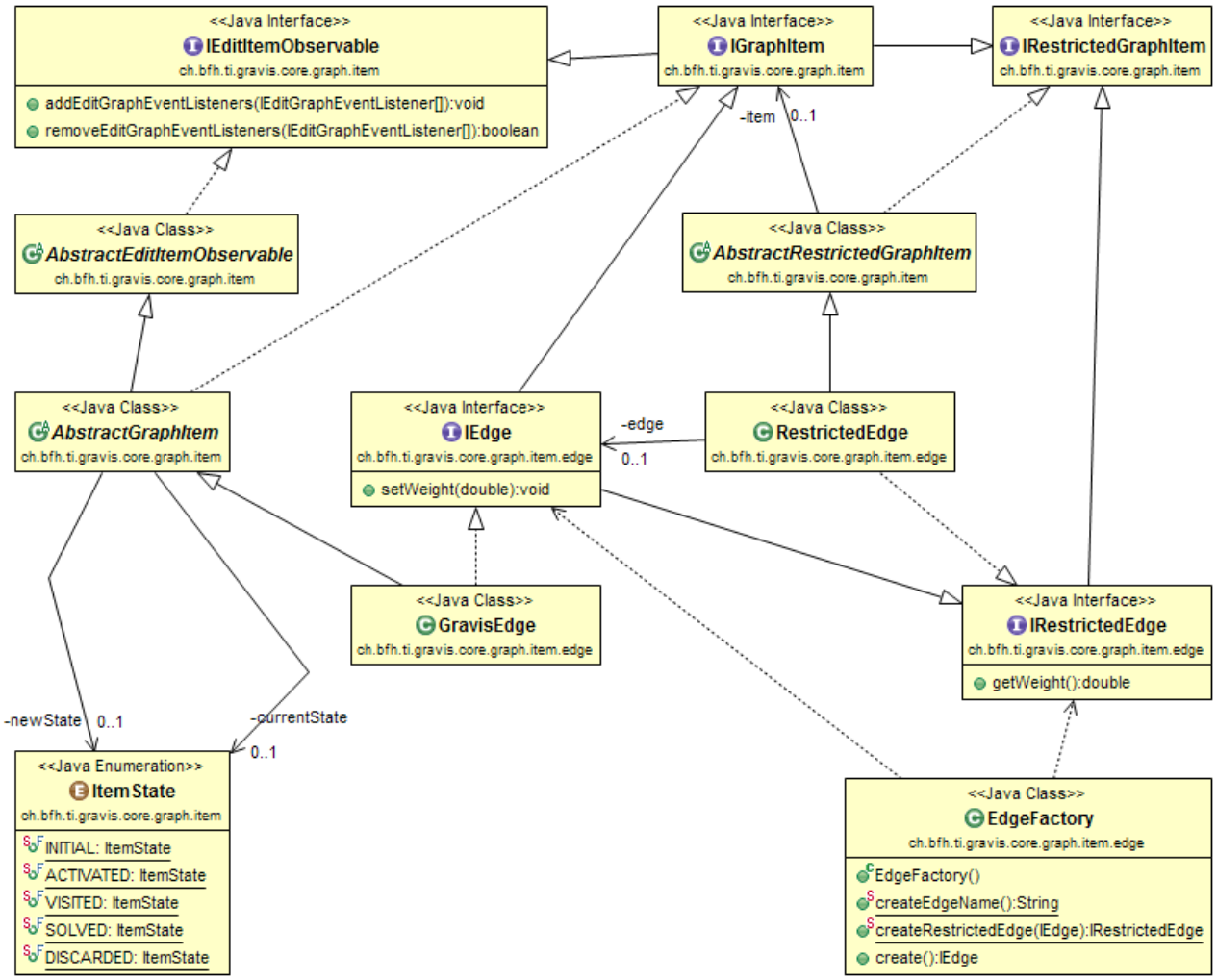


Abbildung 7: Edge-Klassen

## 3.7 Step-Klassen

Die Klassen und Interfaces im `step`-Package sind für die Konstruktion der Animationsschritte verantwortlich (vgl. Abbildung weiter unten).
Ein Schritt (im folgenden Step genannt) repräsentiert eine Instanz vom Typ `IStep`. Dieses Interface schreibt zwei Methoden vor: execute() und `unExecute()`. Mit execute() wird eine Operation ausgeführt und mit `unExecute()` wird die Operation wieder rückgängig gemacht (DO- und UNDO-Operationen). Beide Methoden liefern eine Instanz vom Typ `IStepResult` zurück. Damit kann dann der Kommentar des Steps abgefragt werden.
Alle Steps erben von der Klasse `EmptyStep`, welche das Interface `IStep` implementiert und standardmässig nichts tut. Ein `ComplexStep` kann eine beliebige Anzahl von Steps enthalten. Einzelne Steps können so beliebig verschachtelt werden (*Composite Pattern*).
Atomare Step-Klassen haben das Suffix *Command* in ihrem Klassennamen. Instanzen dieser Klassen halten eine Referenz auf ein Graph-Element `IGraphItem` und führen auf diesem Objekt ihre Operationen aus (*Command Pattern*). Es sind die folgenden Operationen möglich:

- Zustand `ItemState` ändern
- Resultat ändern
- Sichtbarkeit ändern

- Fette Markierung ändern
- Gestrichelte Markierung ändern

Die Klasse `StepBuilder` koordiniert den Konstruktionsprozess der Steps. Die Methode `addStep()` konstruiert aus einem Array von Graph-Elementen einen neuen Step vom Typ `ComplexStep` und fügt diesen der Liste mit den bestehenden Steps hinzu. Die Konstruktion wird dabei für jedes Graph-Element an ein Objekt vom Typ `StepTransformer` delegiert. Ein `StepTransformer` konstruiert aus einem einzelnen Graph-Element einen Step. Der `StepBuilder` fügt diese Steps dann zu einem `ComplexStep` zusammen.
Mit der Methode `createStepIterator()` wird schliesslich ein Iterator vom Typ `IGravisListIterator<String>` konstruiert. Dieser Iterator wird zum durchlaufen der einzelnen Steps verwendet (*Iterator Pattern*). Er kann die Liste der Steps vorwärts oder rückwärts durchlaufen und bei jedem Step automatisch eine DO- bzw. UNDO-Operation ausführen.
Die Klasse `StepRecorder` ist eine Hilfsklasse, die in den Algorithmen-Klassen zur Konstruktion eines Steps verwendet werden kann. Damit werden verkettete Aufrufe von Graph-Element Methoden ermöglicht. Der Code wird so übersichtlicher und kompakter.

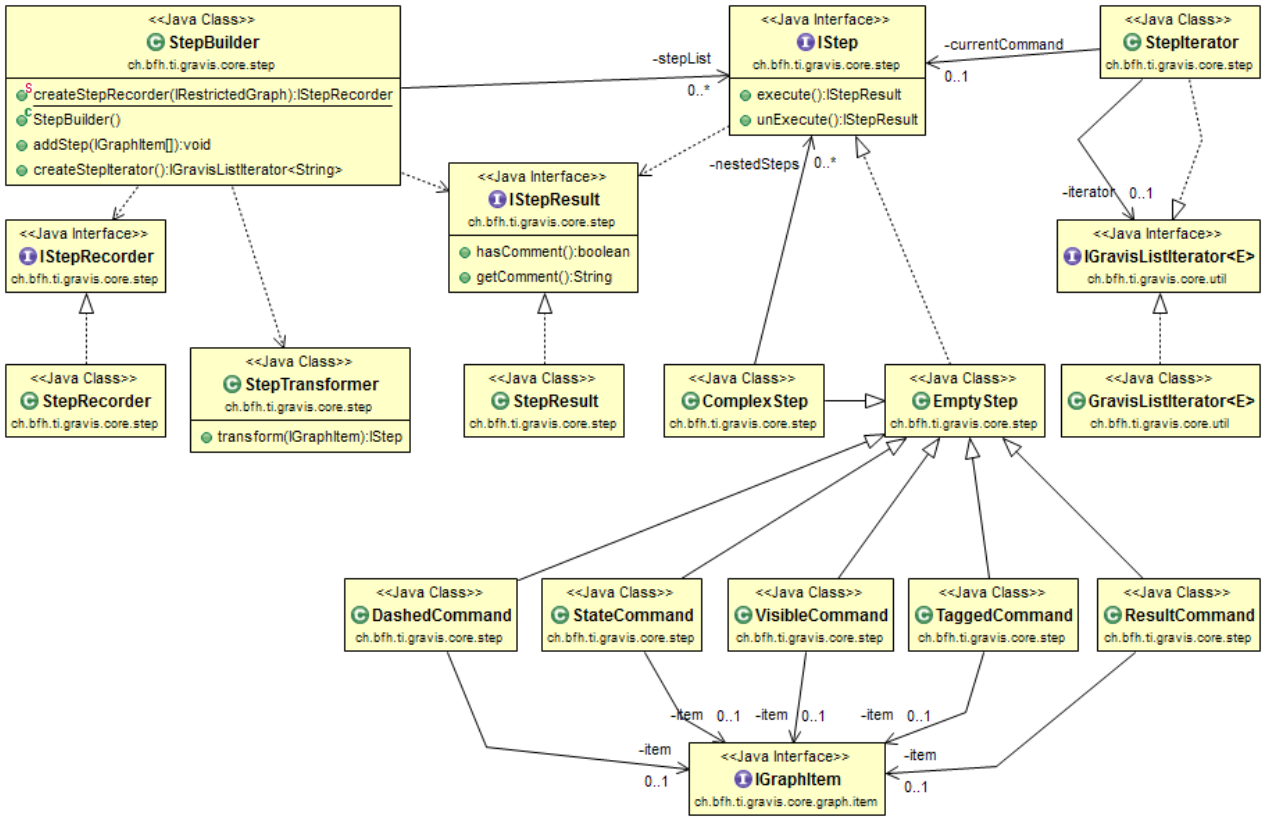


Abbildung 8: Step-Klassen

### 3.8 Algorithmen-Klassen

Alle Algorithmen müssen das Interface `IAlgorithm` implementieren. Die Klasse `AlgorithmFactory` konstruiert Instanzen von Algorithmen basierend auf einem gegebenen Namen. Zur Laufzeit kann dann ein Algorithmus ausgewählt und ausgeführt werden, indem der Methode `calculateSteps()` in `Core` eine Graph-Instanz und ein passender Algorithmen-Name übergeben wird (*Strategy Pattern*). Das Resultat dieser Berechnung ist dann ein Iterator vom Typ `IGravisListIterator<String>`.
Um inkonsistente Zustände zu vermeiden, operiert ein Algorithmus auf einem `IRestrictedGraph` mit Elementen vom Typ `IRestrictedGraphItem`. Der Algorithmus darf also keine neuen Knoten oder Kanten hinzufügen und auch nichts an den Graph-Elementen ändern, was das Resultat des Algorithmus verfälschen könnte (z. B. Kantengewicht ändern). Die Konstruktion von Steps im Algorithmus wird zudem durch eine Instanz vom Typ `IStepRecorder` vereinfacht.

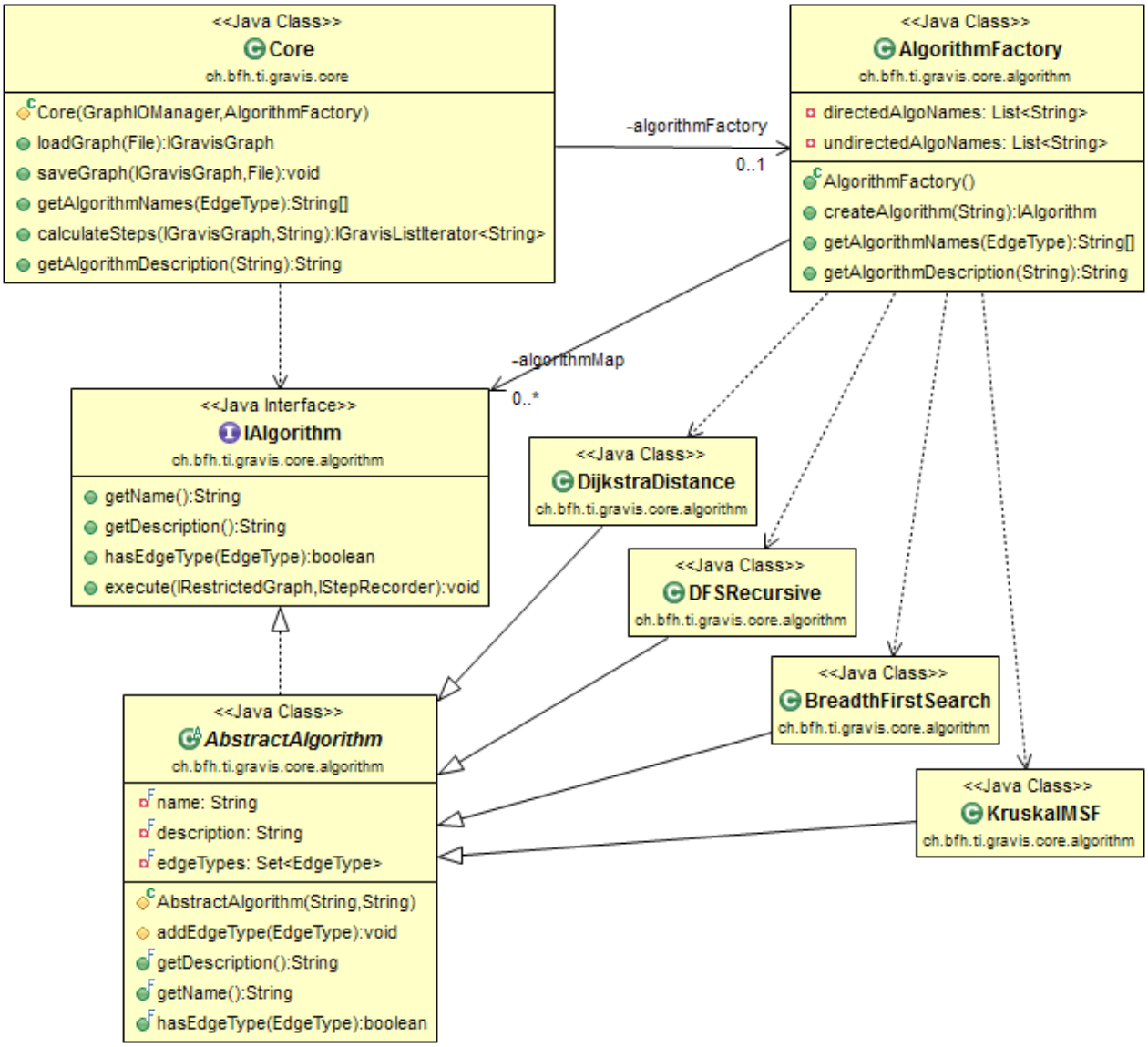


Abbildung 9: Algorithmen-Klassen

# 4 Implementation und Tests

In allen Klassen und Interfaces des Projektes ist der Name des Entwicklers mit der Java-Annotation `@author` gekennzeichnet. Bei einigen aus dem JUNG-Framework abgeleiteten Klassen wurde teilweise bestehender Code aus der Basisklasse übernommen und angepasst. Solche Klassen sind zusätzlich mit dem Namen des Basisklassen-Entwicklers annotiert.

## 4.1 Coding Conventions

Neben den üblichen Coding Conventions von Java und Eclipse wurden in diesem Projekt einige zusätzliche Coding Conventions verwendet:

- Interfaces beginnen mit dem Präfix `I` und abstrakte Klassen mit dem Präfix `Abstract`.
- Für Literale werden am Anfang der Klassendefinition Konstanten definiert.
- Variablen, deren Werte nie ändern, werden mit `final` deklariert.

## 4.2 Algorithmen-Implementation

Es folgen einige Bemerkungen zur Implementation der Algorithmen. Detaillierte Kommentare sind im Quellcode der Algorithmen zu finden.

### 4.2.1 Tiefensuche (DFS)

Die Tiefensuche ist rekursiv implementiert. Die Knoten des Graphen werden dabei in Preorder traversiert und als besucht markiert sowie aufsteigend nummeriert. Für die Speicherung des Weges wird ein Stack verwendet. Am Anfang der rekursiven Methode wird der aktuelle Knoten auf den Stack gelegt und am Ende wieder entfernt. Wird der Endknoten gefunden, so steht am Ende der Weg vom Start- zum Endknoten in umgekehrter Reihenfolge auf dem Stack. Falls kein Endknoten gefunden wird, ist der Stack am Schluss leer.

### 4.2.2 Breitensuche (BFS)

Die Breitensuche ist mit Hilfe einer Queue implementiert. Vor dem `enqueue()` eines Knotens wird dieser als besucht markiert, aufsteigend nummeriert und sein Vorgänger gespeichert (nur der Startknoten hat keinen Vorgänger). Nach dem `dequeue()` eines Knotens wird dieser expandiert, das heisst, es werden alle seine noch nicht besuchten Nachfolger besucht. Wird nach dem `dequeue()` der Endknoten erreicht, so kann der Weg vom Start- zum Endknoten über den jeweils gespeicherten Vorgänger rekonstruiert werden.

### 4.2.3 Dijkstra-Algorithmus

Bei der Implementation des Dijkstra-Algorithmus wird zur Bestimmung der minimalen Distanz eine *Priority Queue* verwendet. Die Klasse `MapBinaryHeap<IRestrictedVertex>` aus dem JUNG-Framework ist hier sehr hilfreich. Diese Klasse bietet auch eine *update()* Methode an, falls der Key-Value (aktuelle Distanz) sich ändern sollte.
Jeder Knoten, der aus der *Priority Queue* entnommen wird und eine endliche Distanz hat, wird zur Lösung hinzugefügt. Dieser Knoten wird dann expandiert, das heisst, es werden alle Nachfolger besucht und die Distanzen angepasst. Für jeden besuchten Knoten wird auch der Vorgänger mit dem kürzesten Weg gespeichert. Wie bei BFS kann so der Weg vom Start- zum Endknoten rekonstruiert werden.

### 4.2.4 Kruskal-Algorithmus

Beim Kruskal-Algorithmus wird eine *Disjoint-Set* Datenstruktur verwendet (vgl. Link im Anhang). Zu diesem Zweck steht die nützliche Klasse `Partition` zur Verfügung. Zum Bestimmen der minimalen Kantengewichte wird wieder eine *Priority Queue* benutzt. Sehr hilfreich zum Anzeigen von Zyklen ist die rekursive Tiefensuche. Ein Stack speichert dabei alle zum Zyklus gehörenden Knoten und Kanten.

### 4.3 Erweiterbarkeit

Zusätzliche Algorithmen können ohne grossen Aufwand zur Applikation hinzugefügt werden. Als erstes braucht es eine neue Algorithmen-Klasse, die von AbstractAlgorithm abgeleitet ist und sich im Package `ch.bfh.ti.gravis.core.algorithm` befindet. Es müssen dabei der Algorithmus-Name, die Beschreibung und der zulässige `EdgeType` im Konstruktor angegeben werden. Dann muss man noch eine Instanz der neuen Algorithmen-Klasse im Konstruktor von `AlgorithmFactory` zur `algorithmMap` hinzufügen (vgl. Abbildung).

```
IAlgorithm algorithm = new DFSRecursive();
this.algorithmMap.put(algorithm.getName(), algorithm);
algorithm = new BreadthFirstSearch();
this.algorithmMap.put(algorithm.getName(), algorithm);
algorithm = new DijkstraDistance();
this.algorithmMap.put(algorithm.getName(), algorithm);
algorithm = new KruskalMSF();
this.algorithmMap.put(algorithm.getName(), algorithm);
```

Abbildung 10: Neuen Algorithmus hinzufügen

Nach dem Kompilieren und Starten der Applikation wird der neue Algorithmus automatisch eingebunden.

### 4.4 Tests

Für die wichtigsten Core-Klassen wurden JUnit-Tests implementiert. Der Quellcode zu diesen Tests befindet sich ebenfalls im *GitHub-Repository* (Ordner *src/test*). Bei jedem Build mit Maven werden alle Tests automatisch ausgeführt. Zum testen der Algorithmen und des GUIs wurden auch einige Beispielgraphen erstellt. Diese befinden sich unter *sample_graphs* im Applikations-Ordner.

## 5 User-Dokumentation

Dieses Kapitel zeigt mit Hilfe von Screenshots die Funktionalitäten der Applikation. Zudem wird auch noch kurz das Dateiformat *GraphML* vorgestellt.
Hinweis: Im Anhang ist ein Link zur API-Dokumentation zu finden.

### 5.1 Applikation

Systemvoraussetzungen:

- Java (JRE) 7 oder höher
- Bildschirmauflösung von 1280 x 1024 oder höher empfohlen

Bearbeitungsmodi:

- PICKING: Einen oder mehrere Knoten verschieben
- EDITING: Knoten oder Kanten hinzufügen oder löschen, Knoten oder Kanten bearbeiten
- TRANSFORMING: Verschiebung, Drehung oder Scherung des ganzen Graphen

Allgemeine Hinweise:

- Die Popup-Menüs im Graph-Fenster sind nur im EDITING-Modus aktiv.
- Beim Speichern muss die Dateierweiterung **.graphml* angegeben werden.
- Bei Dijkstra werden in den Knoten-Labels die Zwischenresultate angezeigt.
- Eine Bearbeitung im EDITING-Modus erfordert eine Neuberechnung des Algorithmus.
- Für die Buttons *Anfang*, *Ende*, *vorwärts*, *rückwärts* gibt es die Shortcuts *Ctrl+Left*, *Ctrl+Right*, *Alt+Left*, *Alt+Right*.
- In einem Dialog kann der Button mit dem aktuellen Fokus mit der Space-Taste geklickt werden.

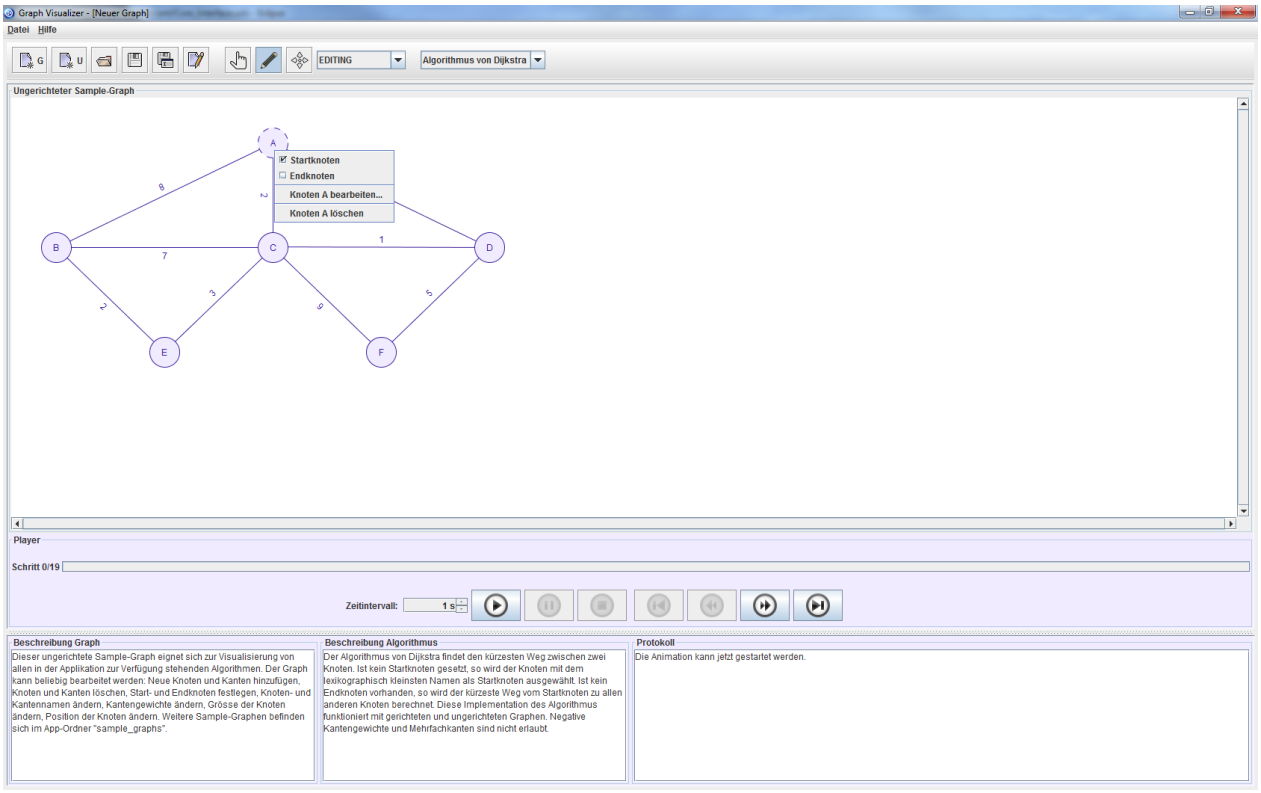


Abbildung 11: Popup-Menü von Knoten A im EDITING-Modus

### 5.1.1 Dijkstra

Farben der *Knoten* bei Dijkstra:

- *Blau*: Dieser Knoten wurde noch nicht besucht.
- *Gelb*: Dieser Knoten wurde bereits besucht, der kürzeste Weg wurde aber noch nicht gefunden (der Knoten gehört noch nicht zur Lösung).
- *Grün*: Für diesen Knoten wurde der kürzeste Weg bereits gefunden (der Knoten gehört bereits zur Lösung).
- *Rot*: Dieser Knoten ist vom Startknoten aus unerreichbar.

Farben der *Kanten* bei Dijkstra:

- *Blau*: Diese Kante wurde noch nicht besucht.
- *Gelb*: Diese Kante wurde bereits besucht, sie ist aber noch nicht Teil eines kürzesten Weges (die Kante gehört noch nicht zur Lösung).
- *Grün*: Diese Kante ist bereits Teil eines kürzesten Weges (die Kante gehört bereits zur Lösung).
- *Rot*: Diese Kante kann nicht Teil eins kürzesten Weges sein und wurde aus der Lösung ausgeschlossen.

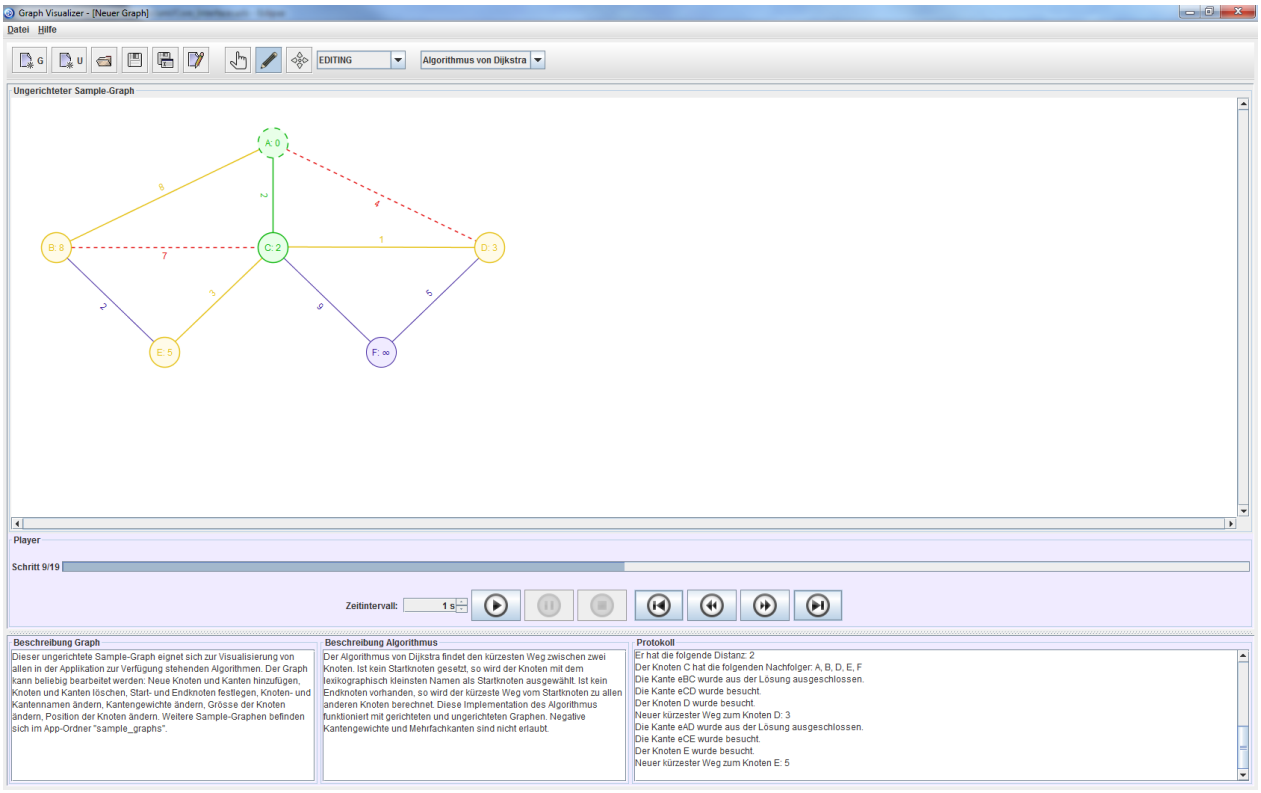


Abbildung 12: Dijkstra in Aktion

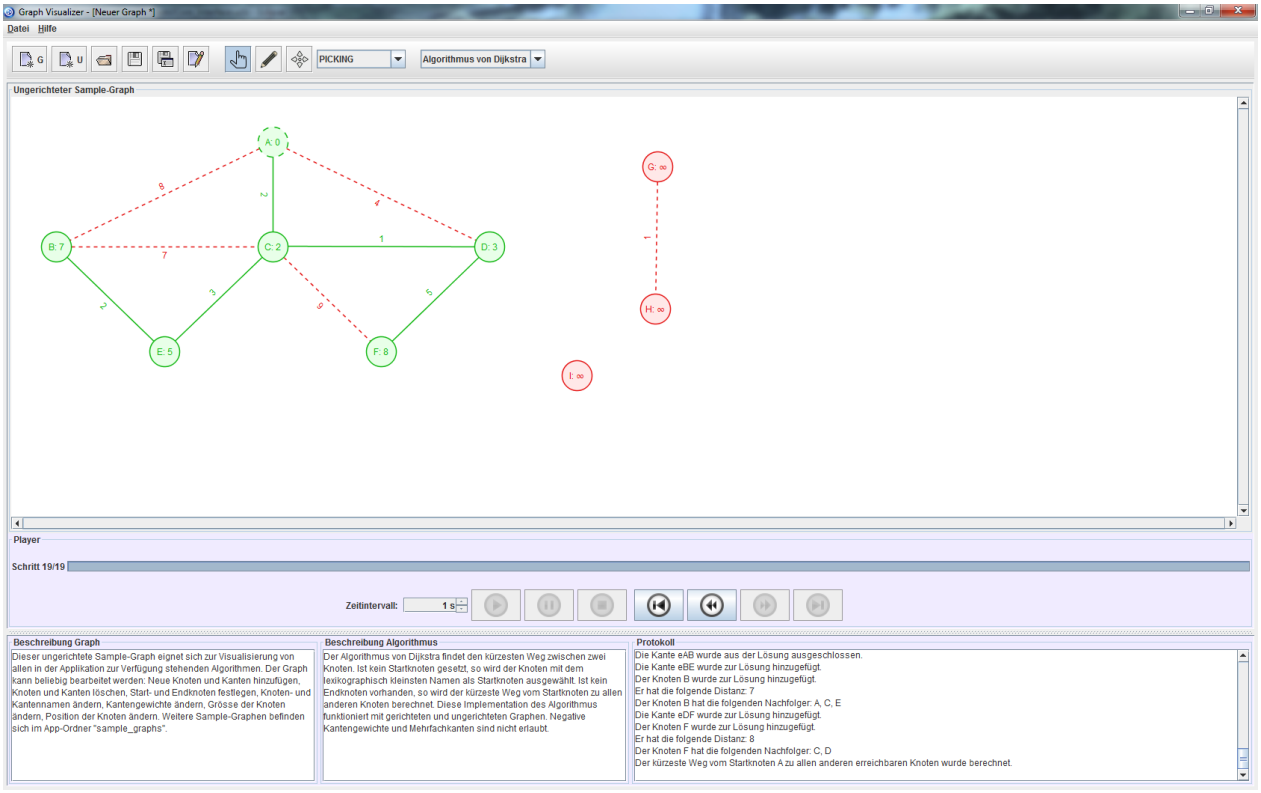


Abbildung 13: Dijkstra ohne Endknoten

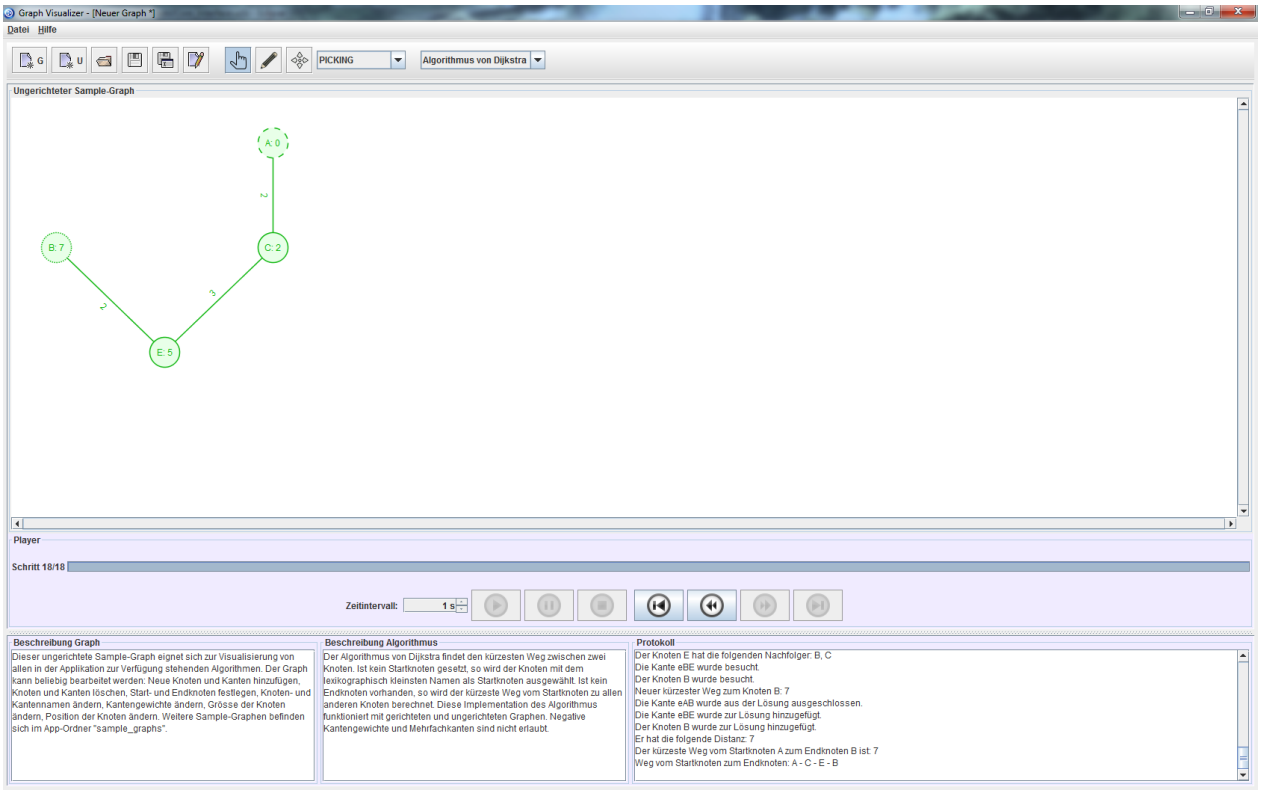


Abbildung 14: Dijkstra mit Endknoten

### 5.1.2 DFS

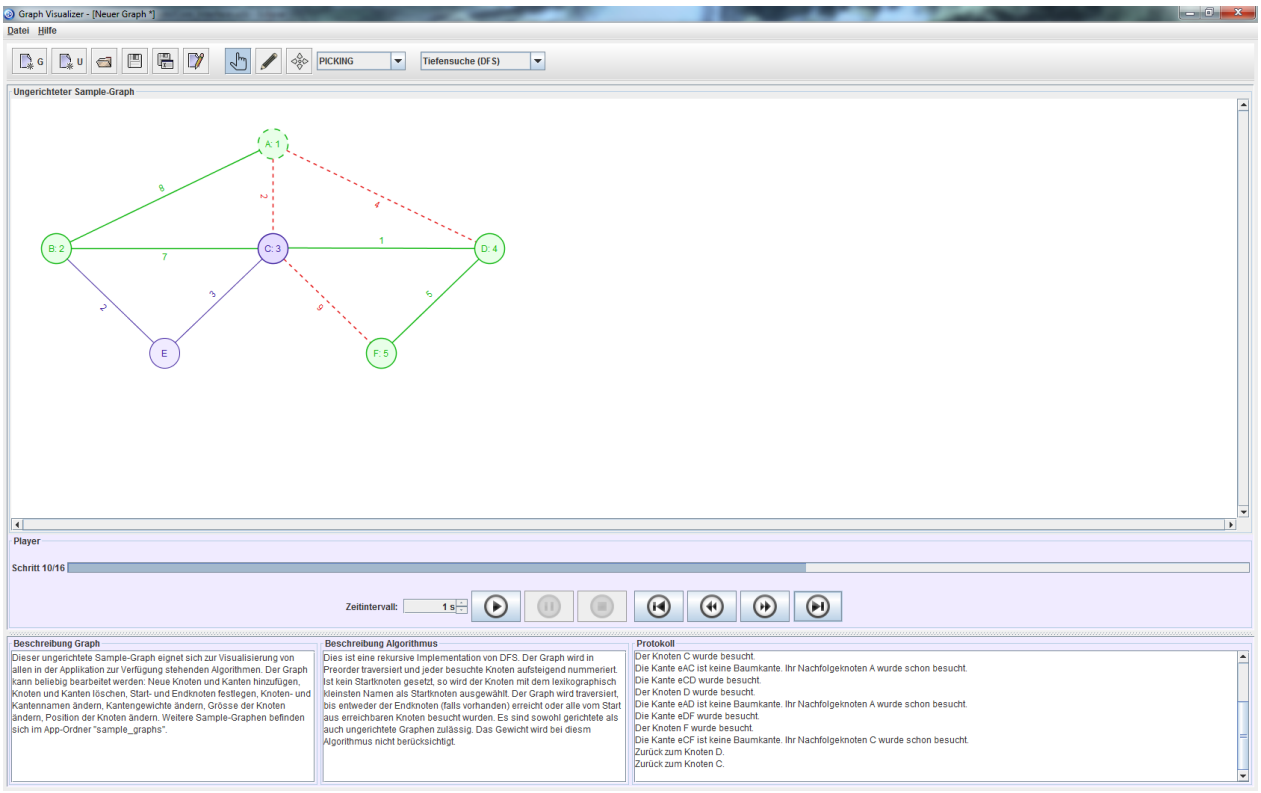


Abbildung 15: DFS in Aktion

Farben der *Knoten* bei DFS:

- *Blau*: Dieser Knoten wurde noch nicht besucht.
- *Grün*: Dieser Knoten wurde bereits besucht.
- *Rot*: Dieser Knoten ist vom Startknoten aus unerreichbar.
- *Dunkles blau*: Dieser Knoten wurde bereits besucht und wird im Moment gerade wieder besucht (Rücksprung in der Rekursion).

Farben der *Kanten* bei DFS:

- *Blau*: Diese Kante wurde noch nicht besucht.
- *Grün*: Diese Kante wurde bereits besucht.
- *Rot*: Diese Kante ist keine Baumkante. Ihr Nachfolgeknoten wurde bereits besucht.

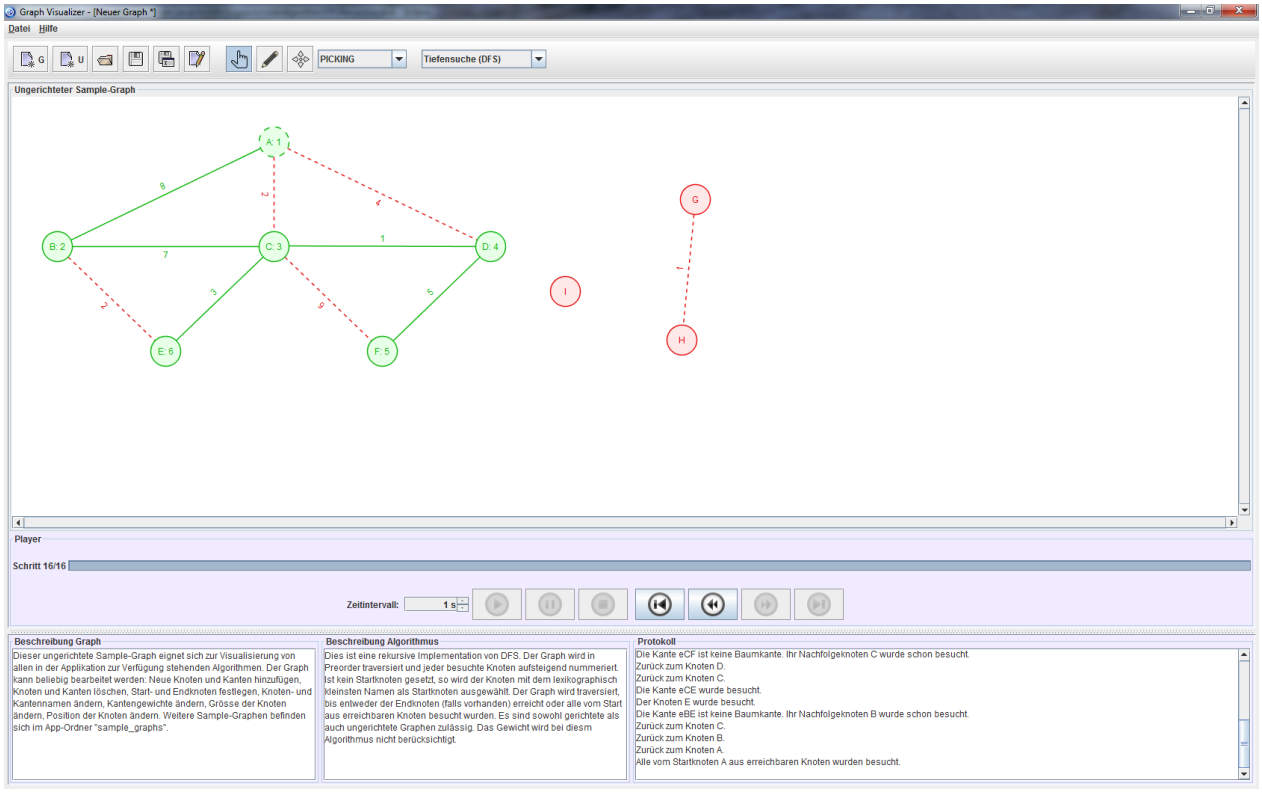


Abbildung 16: DFS ohne Endknoten

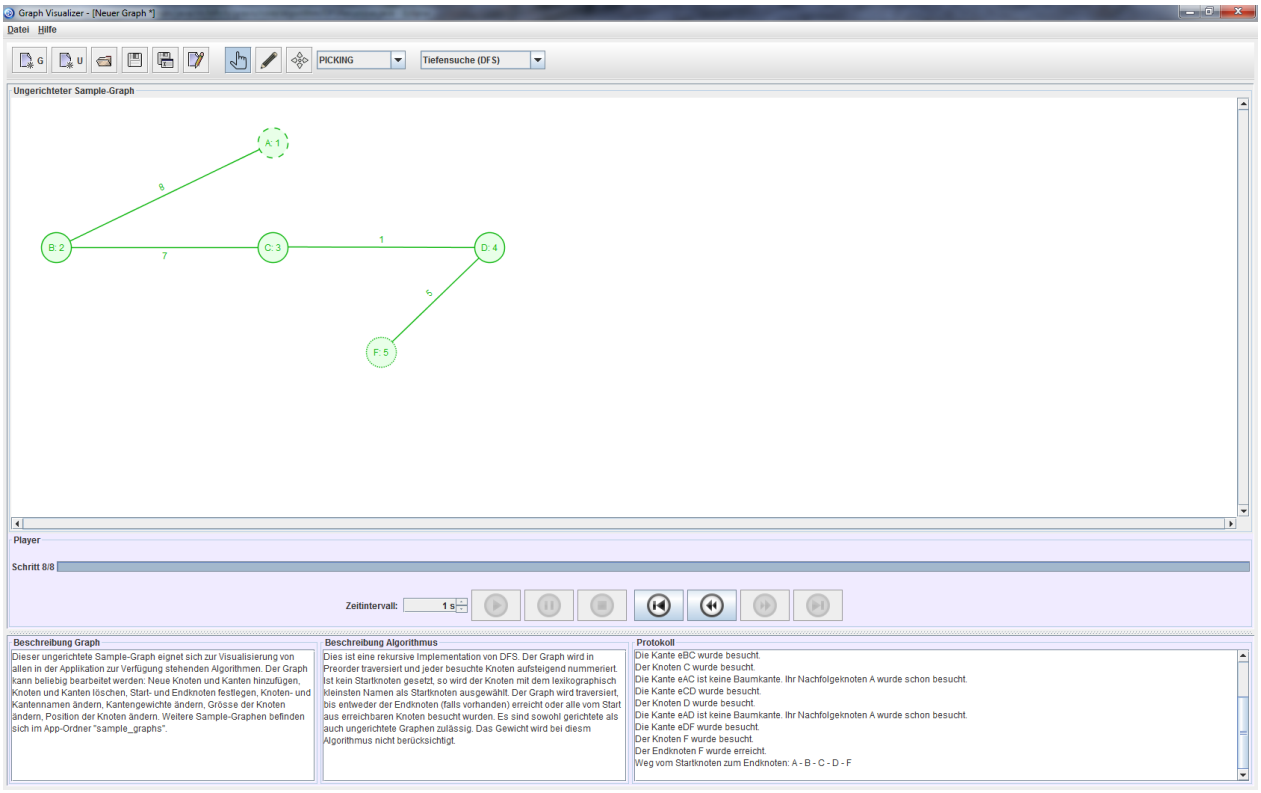


Abbildung 17: DFS mit Endknoten

### 5.1.3 BFS

Farben der *Knoten* bei BFS:

- *Blau*: Dieser Knoten wurde noch nicht besucht.
- *Gelb*: Dieser Knoten wurde bereits besucht.
- *Grün*: Dieser Knoten wurde bereits besucht und wird gerade expandiert oder ist bereits vollständig expandiert. Expandieren heisst in diesem Zusammenhang, dass die Nachfolgeknoten des Knotens gerade besucht werden.
- *Rot*: Dieser Knoten ist vom Startknoten aus unerreichbar.

Farben der *Kanten* bei BFS:

- *Blau*: Diese Kante wurde noch nicht besucht.
- *Gelb*: Diese Kante wurde bereits besucht.
- *Grün*: Der Nachfolgeknoten dieser Kante wurde bereits besucht und wird gerade expandiert oder ist bereits vollständig expandiert.
- *Rot*: Diese Kante ist keine Baumkante. Ihr Nachfolgeknoten wurde bereits besucht.

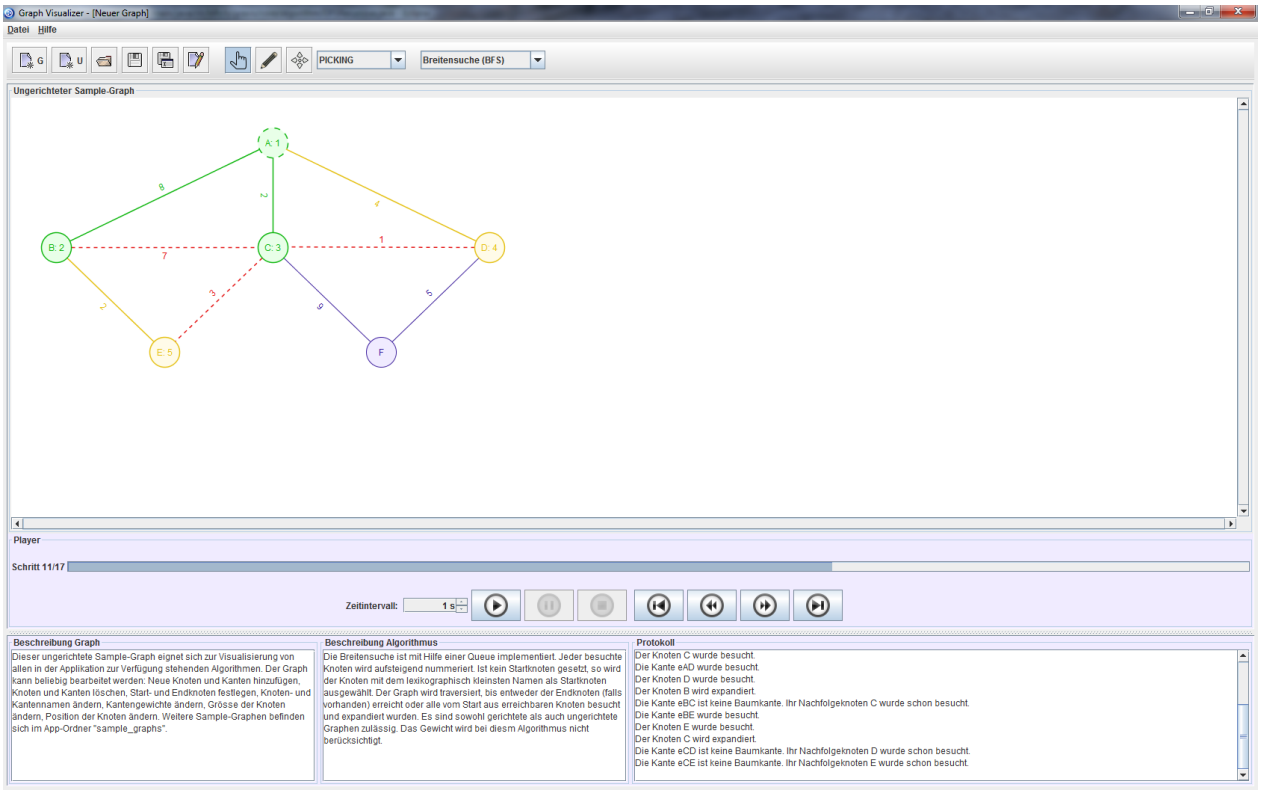


Abbildung 18: BFS in Aktion

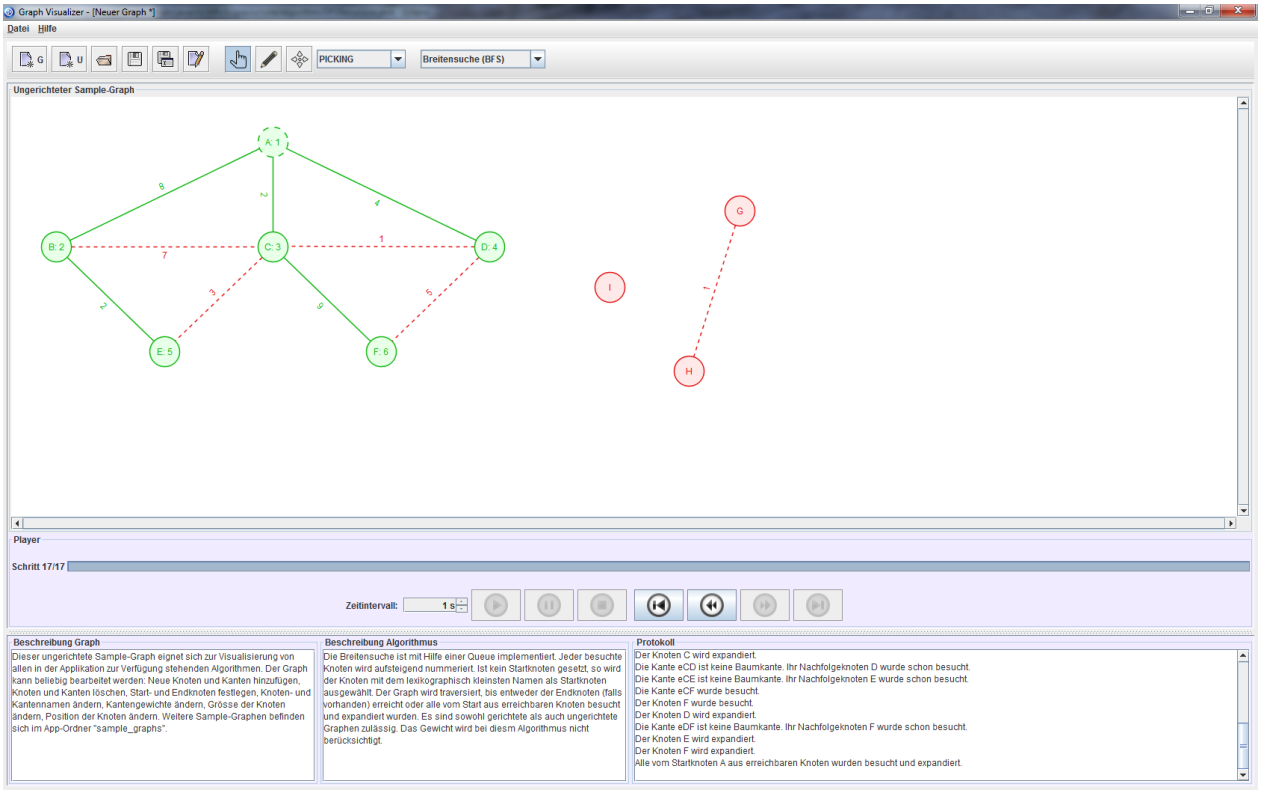


Abbildung 19: BFS ohne Endknoten

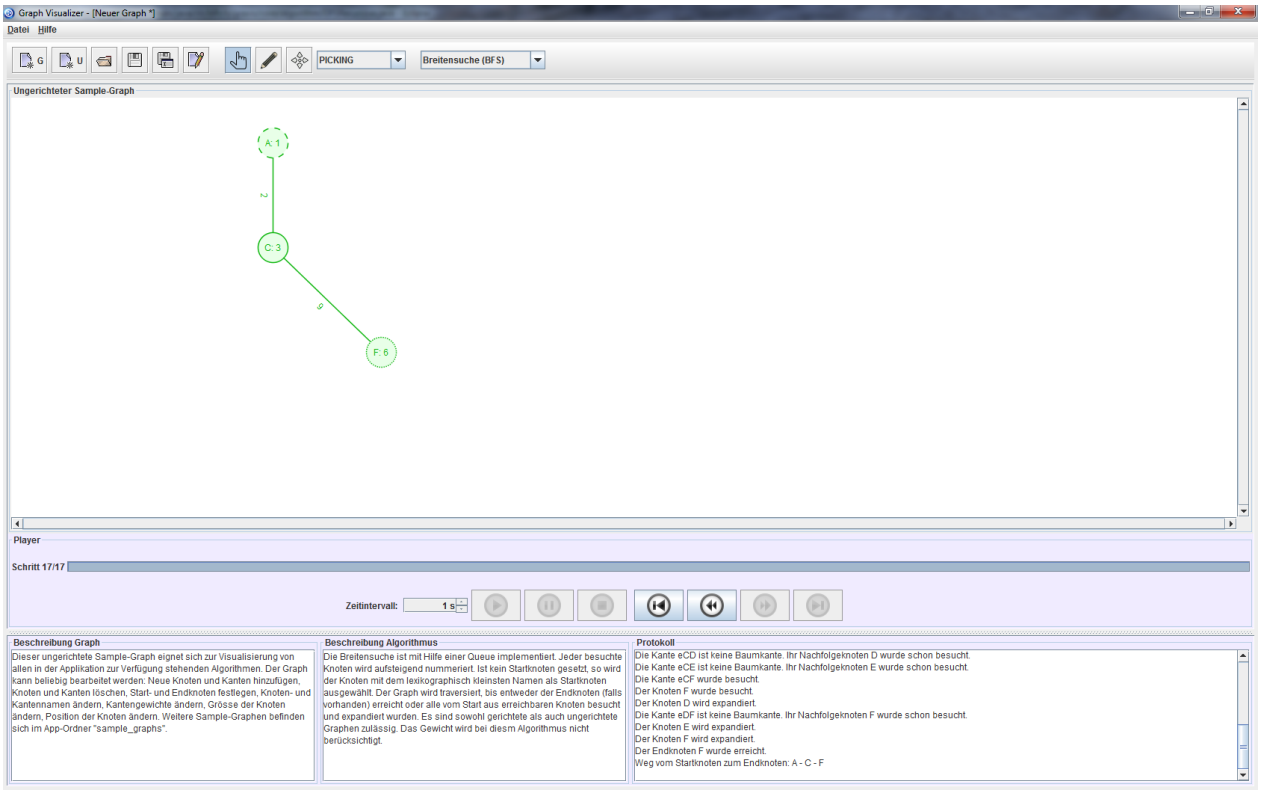


Abbildung 20: BFS mit Endknoten

### 5.1.4 Kruskal

Farben der *Knoten* bei Kruskal:

- *Blau*: Dieser Knoten ist noch nicht mit einer Lösungskante verbunden.
- *Gelb*: Es wird gerade überprüft, ob die Kante zwischen den beiden gelben Knoten zur Lösung hinzugefügt werden kann.
- *Grün*: Dieser Knoten ist bereits mit einer Lösungskante verbunden.
- *Rot*: Werden die rot markierten Knoten durch Kanten verbunden, so entsteht ein Zyklus.

Farben der *Kanten* bei Kruskal:

- *Blau*: Diese Kante wurde noch nicht überprüft.
- *Gelb*: Diese Kante wird gerade überprüft.
- *Grün*: Diese Kante wurde bereits zur Lösung hinzugefügt.
- *Rot gestrichelt*: Diese Kante wurde aus der Lösung ausgeschlossen, da sonst ein Kreis entsteht.
- *Rot nicht gestrichelt*: Die roten, nicht gestrichelten Kanten bilden einen Zyklus.

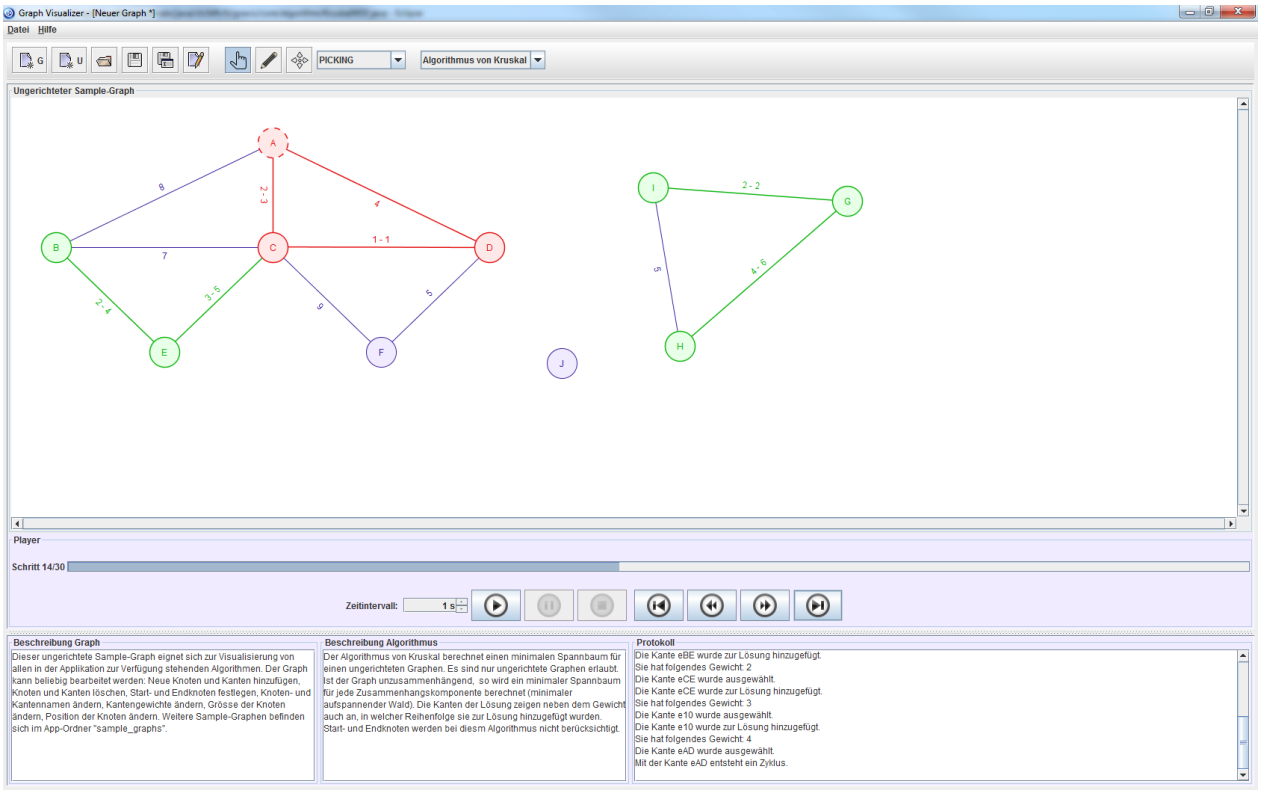


Abbildung 21: Kruskal in Aktion

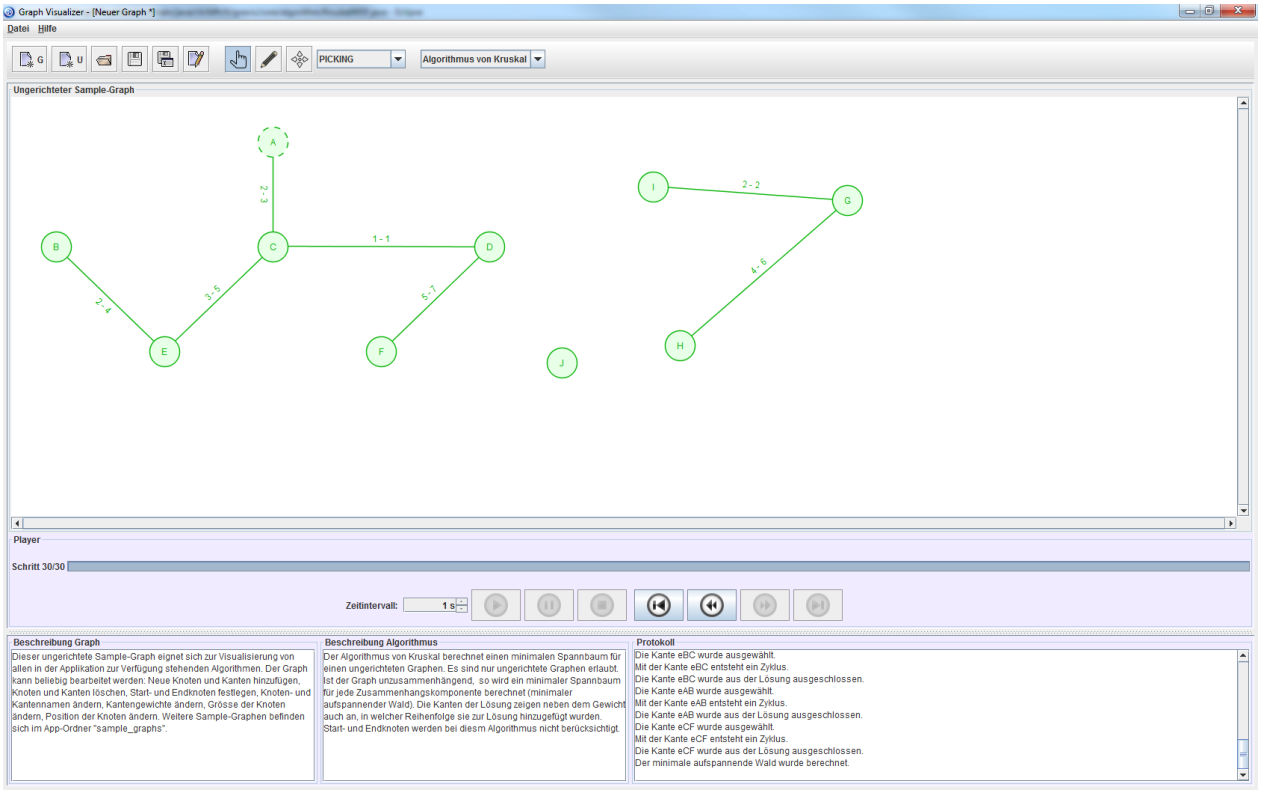


Abbildung 22: minimal aufspannender Wald nach Kruskal

### 5.2 GraphML-Format

Ein Graph wird im Dateiformat *GraphML* gespeichert. Der folgende Screenshot zeigt einen Ausschnitt aus einer *GraphML*-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

  <key id="description" for="graph">
    <default>Kruskal Graph 1</default>
  </key>
  <key id="vertexLocation.y" for="node">
    <default>50</default>
  </key>
  <key id="vertexWidth" for="node">
    <default>40.0</default>
  </key>
  <key id="endVertex" for="node">
    <default>false</default>
  </key>
  <key id="vertexColor" for="node">
    <default>lightBlue</default>
  </key>
  <key id="vertexLocation.x" for="node">
    <default>50</default>
  </key>
  <key id="startVertex" for="node">
    <default>false</default>
  </key>
  <key id="vertexHeight" for="node">
    <default>40.0</default>
  </key>
  <key id="weight" for="edge">
    <default>1.0</default>
  </key>
  <key id="edgeColor" for="edge">
    <default>blue</default>
  </key>

  <graph id="Kruskal Graph 1" edgedefault="undirected">
    <data key="description">Ein ungerichteter Graph zum Testen des Kruskal-Algorithmus.</data>
    <node id="D">
      <data key="vertexLocation.y">249.0</data>
      <data key="vertexWidth">40.0</data>
      <data key="endVertex">false</data>
      <data key="vertexColor">lightBlue</data>
      <data key="vertexLocation.x">89.0</data>
      <data key="startVertex">false</data>
      <data key="vertexHeight">40.0</data>
    </node>
    <node id="C">
      <data key="vertexLocation.y">77.0</data>
      <data key="vertexWidth">40.0</data>
      <data key="endVertex">false</data>
```

Abbildung 23: *GraphML*-Datei

# 6 Abbildungsverzeichnis



# 7 Tabellenverzeichnis

# 8 Anhang

| Einige Links zu diesem Projekt: | |
|---|---|
| GitHub-Repository: | https://github.com/kofmp1/GraphVisualizer |
| Projektdokumentation: | https://github.com/kofmp1/GraphVisualizer/tree/master/ProjectGraphVisualizer/doc |
| API-Dokumentation: | http://htmlpreview.github.io/?https://github.com/kofmp1/GraphVisualizer/blob/master/ProjectGraphVisualizer/doc/apidocs/index.html |
| JUNG-Framework: | http://jung.sourceforge.net/ |
| Dijkstra Pseudocode: | http://www-m9.ma.tum.de/Allgemeines/DijkstraCode |
| DFS Pseudocode: | http://en.wikipedia.org/wiki/Depth-first_search |
| BFS Pseudocode: | http://de.wikipedia.org/wiki/Breitensuche |
| Kruskal Pseudocode: | http://en.wikipedia.org/wiki/Kruskal%27s_algorithm |
| Disjoint-set data structure: | http://en.wikipedia.org/wiki/Disjoint-set_data_structure |