

Visualisieren von Algorithmen in Compilern

Tom Schreiner

Bachelorarbeit

Beginn der Arbeit:	05. November 2024
Abgabe der Arbeit:	05. Februar 2025
Gutachter:	John Witulski Fabian Ruhland

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 05. Februar 2025

Tom Schreiner

Zusammenfassung

Die Ziele dieser Bachelorarbeit waren einerseits einige Algorithmen und Konzepte aus dem Compilerbau zu visualisieren, aber auch ein Framework zu entwickeln welches diese und beliebige weitere Algorithmen verständlich und Schritt für Schritt visualisieren kann.

Folgende Algorithmen wurden implementiert:

1. Analyse von erreichenden Definitionen für Grundblöcke
2. Liveness Analyse für Grundblöcke
3. Liveness Analyse für einzelne 3-Address-Code Instruktionen
4. Erstellen von Grundblöcken für ein Programm geschrieben in 3-Address-Code
5. Erstellen eines Kontrollflussgraphen für ein 3-Address-Code Programm

Um diese Algorithmen simpel und gut verständlich zu visualisieren brauchte das Framework zwei Arten von Darstellungen:

1. Graphen: um Kontrollflussgraphen darzustellen
2. Tabellen: um Datenflusswerte darzustellen und 3-Address-Code in einer angenehmen Art und Weise zu visualisieren

Desweiteren brauchte es die Möglichkeit 3-Address-Code und Grundblöcke einfach zu verarbeiten.

Außerdem ist es durch Implementierung eines Interfaces einfach weitere Algorithmen hinzuzufügen. Es wurden weitere Interfaces implementiert mit denen häufig genutzte Funktionalitäten wie zum Beispiel einen Button, um Code aus einer Datei zu laden, einfach in neuen Plugins genutzt werden können. Somit können weitere Plugins mit sehr viel weniger Aufwand hinzugefügt werden.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Theoretische Grundlagen	2
1.2.1	Drei Address Code	2
1.2.2	Grundblöcke	3
1.2.3	Kontrollflussgraphen	5
1.2.4	Erreichende Definitionen	6
1.2.5	Lebendige Variablen	6
1.2.6	Konstantenfaltung	6
2	Framework	7
2.1	Gui	10
2.2	Darstellen von Daten	11
2.2.1	Graphen	11
2.2.2	Tabellen	16
2.3	Drei Address Code	17
2.3.1	Die ThreeAddressCodeInstruction Klasse	17
2.3.2	Die BasicBlock Record-Klasse	20
2.3.3	Die ThreeAddressCode Klasse	21
2.4	Implementierung von Algorithmen	22
2.4.1	Das Plugin Interface	23
3	Implementierte Algorithmen	24
3.1	Erstellung von Grundblöcken aus 3-Address-Code	25
3.2	Erstellung eines Kontrollflussgraphen aus 3-Address-Code	26
3.3	Analyse von erreichenden Definitionen	27

3.4	Analyse von lebendigen Variablen bezüglich Grundblöcken	28
3.5	Analyse von lebendigen Variablen bezüglich einzelnen Instruktionen	29
3.6	Optimierung von Grundblöcken mit Konstantenfaltung	30
4	Related Work	31
5	Evaluation	31
6	Future Work	31
7	Fazit	31
	Abbildungsverzeichnis	32
	Tabellenverzeichnis	32
	Literatur	33

1 Einleitung

1.1 Motivation

Als ich im Wintersemester das Modul "Compilerbau" belegt habe, habe ich mich das erste mal mit einigen Algorithmen beschäftigt die eben in diesem Themengebiet angewendet werden. Dabei fiel es mir bei einigen schwer mir diese ohne weiteres vorzustellen. Ein Tool, mit dem man Schritt für Schritt durch diese Algorithmen gehen kann hätte mir das Verstehen der Algorithmen und vor allem das Entwickeln einer Intuition warum diese Algorithmen überhaupt so funktionieren wie sie es tun starkt erleichtert.

In dieser Bachelorarbeit wird ein Framework in Java entwickelt, mit dem es einfach sein soll Algorithmen zu visualisieren. Zudem werden einige der Algorithmen, die im Compilerbau verwendet werden, implementiert. Das Framework basiert darauf, dass durch Implementierung von Interfaces einfach neue Algorithmen als Plugins hinzugefügt werden können. Dadurch kann gewährleistet werden, dass wenn etwas im Framework nicht mehr funktioniert, zum Beispiel durch neue Versionen von Java oder einem Update einer Dependency, dieses Modul ausgetauscht werden kann ohne die implementierten Plugins aktualisieren zu müssen.

1.2 Theoretische Grundlagen

Im folgenden Abschnitt werden die für diese Arbeit notwendigen grundlegenden Konzepte und Algorithmen erklärt da im weiteren Verlauf der Arbeit nur auf die Implementierung dieser eingegangen wird.

1.2.1 Drei Address Code

Drei Address Code ist eine Art von Zwischencode. Charakterisierend für Drei Address Code (folgend auch 3AC genannt) ist, dass einzelne Instruktionen auf maximal drei Adressen (beziehungsweise Variablen oder Konstanten) zugreifen, also die Variable in die der resultierende Wert gespeichert wird und eine oder zwei Variablen oder Konstanten aus denen sich der Resultierende Wert bildet. Dazu ist noch die Operation die ausgeführt wird angegeben.

Für diese Bachelorarbeit wurde eine Teilmenge des im Drachenbuch [Aho08] beschriebenen 3AC verwendet.

Folgende Operationen gibt es:

1. Binäre Operationen $X = Y \text{ op } Z$
In denen das Resultat aus einer der folgenden binären Operation in der Adresse X gespeichert wird. Implementiert wurden Addition, Subtraktion, Multiplikation und Division.
2. Die unäre Operation $X = - Y$
In der der invertierte Wert von Y in X gespeichert wird.
3. Der Kopierbefehl $X = Y$
In der der Wert Y in X kopiert wird
4. Der unbedingte Sprung goto X
Hier wird kein Wert gespeichert, sondern zu der Adresse die in X gespeichert ist gesprungen
5. Die bedingten Sprünge if Y goto X und ifFalse Y goto X
In denen wenn der Wert Y entweder true oder false repräsentiert zur Adresse X gesprungen wird, oder die nächste Instruktion ausgeführt wird wenn dies nicht der Fall ist.
6. Die bedingten Sprünge if Y relOp Z goto X
In denen auch zu X gesprungen wird, wenn die Relation $Y \text{ relOp } Z$ wahr ist, sonst wird auch hier die nächste Instruktion ausgeführt.
Die Implementierte Relationen sind: $Y < Z$, $Y \leq Z$, $Y > Z$, $Y \geq Z$, $Y = Z$ und $Y \neq Z$.

ode erklä-
n sprechen
aupt über zwi-
?

Hierbei können die Adressen X, Y und Z beliebige Zeichenfolgen sein, Y und Z können ausserdem Konstanten sein. Die einzelnen Elemente jeder Instruktion sind durch ein Leerzeichen von einander getrennt. Verschiedene Instruktionen werden durch einen Zeilenumbruch getrennt. Da sich die Algorithmen in dieser Arbeit nicht mit komplexeren Aufgaben wie Speichermanagement beschäftigen, wurde sich dagegen entschieden den 3AC umfangreichen zu Modellieren.

Ein Beispiel für gültigen 3-Address-Code wäre also:

Listing 1: 3-Address-Code der die n-te Fibonacci Zahl ausrechnet und in x speichert

```
0: fib = 1
1: lst = 1
2: n = n - 2
3: if n > 0 goto 9
4: hlp = lst
5: lst = fib
6: fib = lst + hlp
7: n = n - 1
8: goto 3
9: x = fib
```

1.2.2 Grundblöcke

Grundblöcke sind Partitionen eines Zwischencodeprogrammes in denen Instruktionen immer zusammen ausgeführt werden.[Aho08] Dies ermöglicht es, einen Block an Instruktionen anzuschauen und bestimmte Optimierungsalgorithmen auf sie anzuwenden, ohne Gefahr zu laufen die Semantik des Programmes zu verändern.

Um ein Programm in Grundblöcke aufzuteilen kann man wie folgt vorgehen[Aho08]:

1. Markiere die erste Instruktion des Programmes als Leader, da diese immer ausgeführt wird.
2. Markiere alle Instruktionen als Leader, die Ziel eines Sprungs sind oder auf einen Sprung folgen.
3. Alle Instruktionen die auf eine markierte Instruktion folgen, bis zu einer neuen markierten Instruktion gelten nun als ein Grundblock.

Grundblöcke be-
klären, dass jen-
nix mit cobra am
das auch verste

Folglich kann man für jeden Grundblock bestimmen, welche Grundblöcke auf ihn folgen, daraus lässt sich ein Flussgraph bestimmen den wir, da er den Kontrollfluss beschreibt, folglich Kontrollflussgraphen nennen werden. In diesem stellt jeder Grundblock einen Knoten dar und jede Kante einen Sprung von einem Grundblock zum anderen.

Aus unserem Beispielcode aus dem Letzten Kapitel können wir also folgende Instruktionen markieren:

Listing 2: Fibonacci 3-Address-Code mit markierten Leadern

```
0: fib = 1 //Leader, da erste Instruktion
1: lst = 1
2: n = n - 2
3: if n > 0 goto 9 //Leader, da 8 hierhin springt
4: hlp = lst //Leader, da 3 ein bedingter Sprung ist
5: lst = fib
6: fib = lst + hlp
7: n = n - 1
8: goto 3
9: x = fib //Leader, da 8 ein Sprung ist und 3 hierhin springen kann
```

Daraus folgt, dass wir folgende Grundblöcke haben:

1. Ein Grundblock B_0 der die Adressen 0, 1 und 2 besitzt.
2. Der Block B_1 der nur die Adresse 3 hat.
3. Grundblock B_2 der die Adressen 4 bis 8 besitzt.
4. und Block B_3 der nur die Adresse 9 besitzt.

1.2.3 Kontrollflussgraphen

Kontrollflussgraphen sind gerichtete Graphen, deren Knoten aus Grundblöcken und deren Kanten aus den jeweilig folgenden Grundblöcken besteht.

Auf ihnen lassen sich bestimmte Mengen für alle Grundblöcke definieren, welche für die in dieser Arbeit implementierten Datenflussanalysen benötigt werden:

1. Die $gen[B]$ Menge:
Beschreibt alle Instruktionen in einem Grundblock B welche einer Adresse einen Wert zuweisen.
2. Die $kill[B]$ Menge:
Beschreibt alle Instruktionen ausserhalb des Grundblocks B welche einer Adresse einen Wert zuweisen, der durch eine Instruktion in der $gen[b]$ Menge überschrieben wird.
3. Die $def[B]$ Menge:
In der $def[B]$ Menge sind alle Variablen enthalten, denen im Grundblock B ein Wert zugewiesen wird, bevor diese "verwendet" wird.
4. die $use[B]$ Menge:
Die $use[B]$ Menge definiert alle Variablen, deren Werte vor ihrer Definition verwendet werden.

Um unser Beispiel weiterzuführen, bildet sich folgender Kontrollflussgraph:

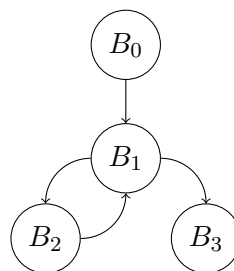


Abbildung 1: Resultierender Kontrollflussgraph

1.2.4 Erreichende Definitionen

Dies ist eine der gebräuchlichsten und nützlichsten Datenflussanalysen[Aho08, S.734]. Über sie können wir herausfinden welchen Variablen zu einem bestimmten Zeitpunkt ein Wert zugewiesen ist. Eine Anwendung wäre zum Beispiel zu kontrollieren ob eine Variable zu einem bestimmten Zeitpunkt überhaupt einen Wert hat, sofern die ursprüngliche Programmiersprache dies als notwendig erachtet. Man kann aber auch schauen, ob die Variable eine Konstante ist und somit Instruktionen gespart werden könnten.

is erklären

1.2.5 Lebendige Variablen

Bei der Analyse lebendiger Variablen(folgend auch liveness Analyse genannt) bringen wir in Erfahrung ob ein bestimmter Wert zu einem bestimmten Zeitpunkt lebendig ist. Lebendig bedeutet in diesem Kontext, das dieser Wert definiert wurde und zu einem späteren Zeitpunkt im Programm auch noch genutzt wird.

Die analyse lebendiger Variablen hat viele Anwendungsgebiete, zum Beispiel hat ein realer Computer nur eine begrenzte Anzahl an Registern, somit können nicht unendlich viele Variablen gleichzeitig zur Benutzung zur Verfügung stehen. Die liveness Analyse kann hier berechnen wie viele Register wir maximal benötigen, da eventuell nicht alle Variablen gleichzeitig lebendig sind, also gebraucht werden. (Frei nach dem Drachenbuch[Aho08] zitiert.)

is erklären

1.2.6 Konstantenfaltung

Die Konstantenfaltung ist ein Optimierungsalgorithmus durch den Rechenoperationen gespart werden können, indem alle Instruktionen eines Grundblocks darauf geprüft werden ob sie nur Konstanten enthalten. Sollte dies der Fall sein wird diese Instruktion zur Compilezeit berechnet, wodurch sie zur Laufzeit nicht mehr berechnet werden muss.[Aho08]

is erklären

2 Framework

In diesem Abschnitt soll es um das Implementierte Framework gehen. Aus den theoretischen Grundlagen lässt sich schließen, dass folgende Daten(-Strukturen) unbedingt im Framework implementiert sein sollten:

1. Drei Address Code Instruktionen:
Um einzelne Instruktionen zu Modellieren zu können brauchen diese einen eigenen Datentyp um Eigenschaften wie Sprünge, konstante Werte oder gelesene und geschriebene Variablen darstellen zu können.
2. Drei Address Code Operationen:
Da es 16 verschiedene Operationen gibt, von denen sich auch noch einige gruppieren lassen, ist es sinnvoll ein *enum* zu schreiben um mit switch-Statements arbeiten zu können.
3. Drei Address Code:
Diese Klasse soll alle Instruktionen sammeln um für sie Grundblöcke und Datenflussmengen zu berechnen.
4. Grundblöcke:
Die Grundblockklasse soll speichern wo im Drei Address Code Programm einzelne Grundblöcke anfangen, aufhören und zu welchen Adressen sie springen.

Da der Usecase dieses Frameworkes das Visualisieren von Algorithmen ist braucht, es folglich eine Möglichkeit den 3-Address-Code, den die Algorithmen benutzen, zu visualisieren. Hier wurde sich für eine Tabelle entschieden, da eine 3AC Instruktion in acht Zellen ¹ aufgeteilt werden kann. Da es sich bei einem Programm immer um eine Liste an Instruktionen handelt, ergibt sich so ein 2-Dimensionales Feld an Daten. Ausserdem möchten wir ja auch die Daten die wir aus unseren Analysen erheben angezeigt bekommen. Auch hier ist eine Tabelle sinnvoll, zum Beispiel um Datenflussmengen unserer Grundblöcke anzuzeigen.

Ein weiterer Algorithmus der implementiert werden soll ist die Erstellung eines Kontrollflussgraphen. Hier ist es natürlich auch wichtig dass dieser dargestellt werden kann, daher sollen auch Graphen dargestellt werden können.

¹if *Y relOp X goto L* ist die längste legale 3AC Instruktion, dazu noch eine Zelle für die aktuelle Instruktion und eine für Kommentare

Aus diesen Anforderungen ergibt sich folgendes Klassendiagramm:

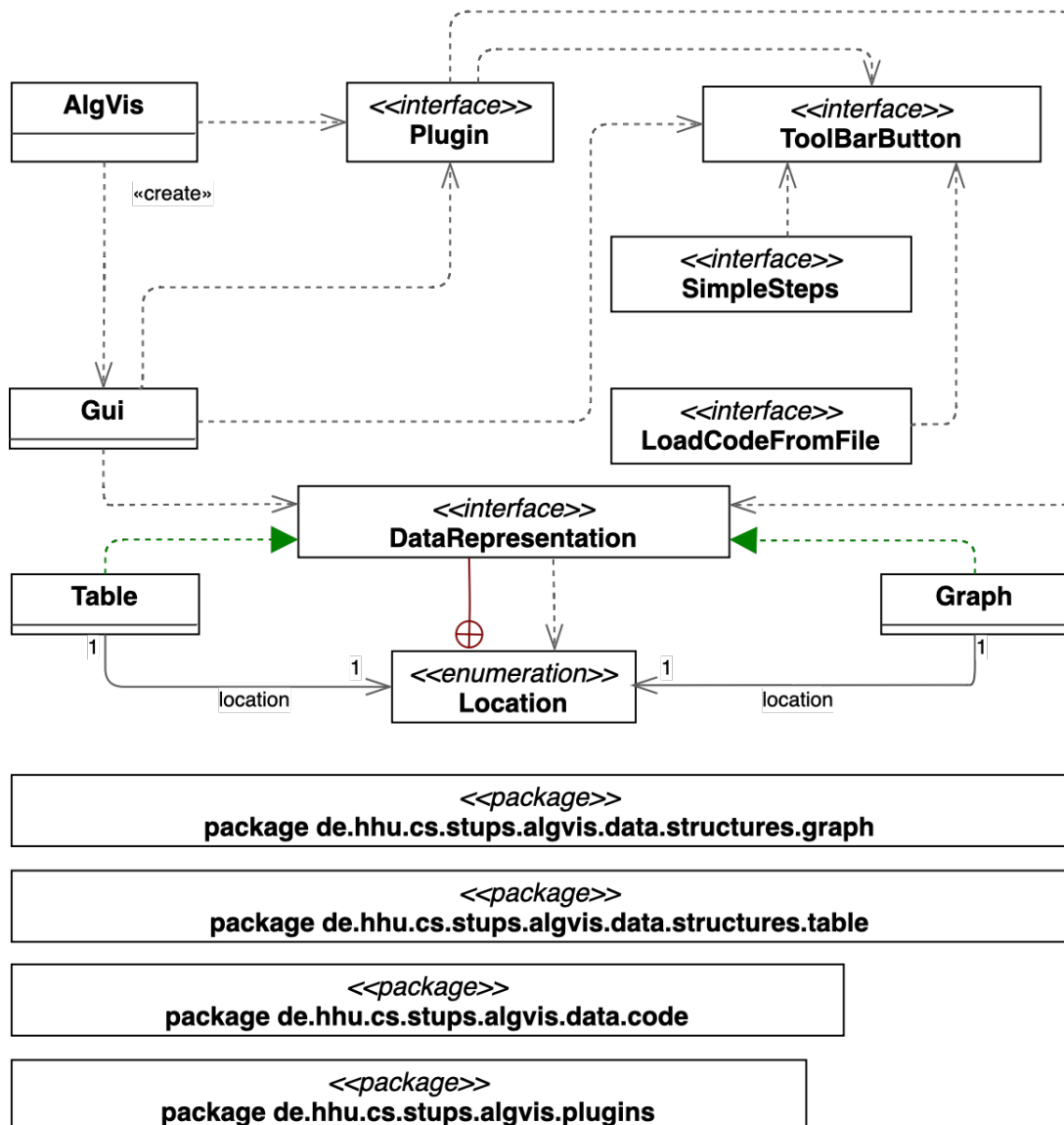


Abbildung 2: Klassenstruktur der GUI-Klassen

- Die Klasse *AlgVis* gilt hier als Entry point. Sie lädt alle Plugins, erstellt ein *Gui* Objekt und übergibt diesem die Plugins.
- Das Interface *Plugin* definiert welche Methoden neue Plugins beziehungsweise Algorithmen benötigen um hinzugefügt zu werden, diese werden in *de.hhu.cs.stups.algvis.plugins* implementiert.
- Zudem definiert das Interface *ToolBarButton* wie Buttons der ToolBar, welche im nächsten Kapitel im Detail erklärt wird, implementiert werden können. Beispiele dafür liefern das *SimpleSteps* und *loadCodeFromFile* Plugin, welche vordefinierte Buttons anbieten.
- Die Packages endend auf *graph* und *table* enthalten hierbei Helferklassen für die jeweiligen Komponenten.
- Im Package *de.hhu.cs.stups.algvis.data.code* sind die vorhin genannten Datenstrukturen für 3-Address-Code, Grundblöcke, 3-Address-Code-Instruktionen und -Operationen implementiert.

2.1 Gui

Die *Gui* Klasse ist das Herzstück der Visualisierung. Hier wird ein *JFrame*, also ein GUI Fenster der Java Standardlibrary *Swing* geladen. In ihr befinden sich drei GUI-Elemente auch aus der *Swing*-Library:

1. Eine Menüleiste in der über ein Dropdown alle Plugins aufgerufen werden können.
2. Ein *JPanel* welches *ContentPanel* genannt wurde, in ihm befinden werden alle grafischen Elemente des aktuell genutzten Plugins geladen. Die verfügbaren grafischen Elemente sind Implementationen des Interfaces *DataRepresentation*, also entweder *Table* oder *Graph* auf die an einem späteren Zeitpunkt genauer eingegangen wird. Sollte später ein Plugin eine andere Art der Darstellung benötigen, kann dieses das Interface mit einer anderen *awt*-Komponente implementieren. Zum Start des Frameworks zeigt es jedoch einen *SplashScreen* mit dem Schriftzug "welcomeän".
3. Und eine *JToolBar*.

Da die meisten Plugins ähnliche Funktionalitäten haben, wie zum Beispiel das schrittweise Durchlaufen eines Algorithmuses, wurde ein GUI-Element hinzugefügt in dem Buttons zur Kontrolle des Plugins hinzugefügt werden können. Diese ist auf der Abbildung Abb. 3 nicht zu sehen, da aktuell kein Plugin geladen ist und somit auch kein Buttons angezeigt werden

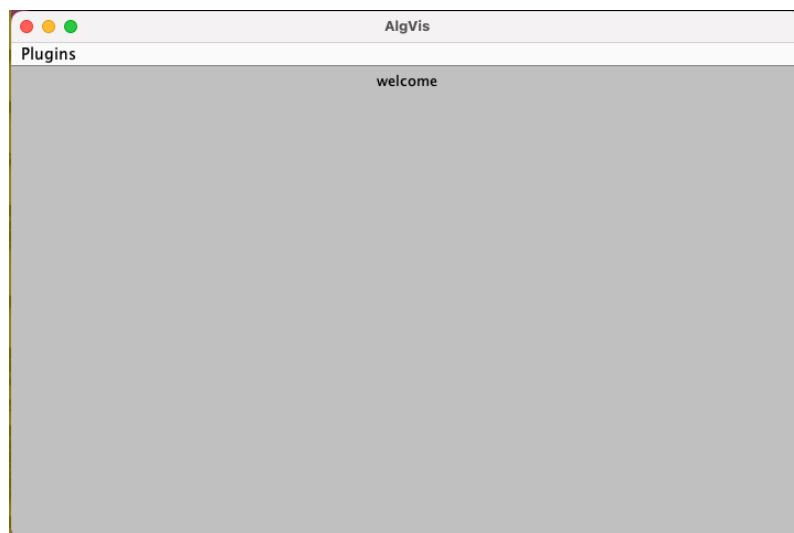


Abbildung 3: Ansicht des Programmes direkt nach Start

2.2 Darstellen von Daten

Um grafische Elemente im *ContentPanel* darzustellen wird das Interface *DataRepresentation* von Gui-Klassen implementiert. Dadurch kann auf zwei Methoden zugegriffen werden:

- *getSwingComponent()* gibt die *awt*-Komponente zurück, welche dem *ContentPanel* hinzugefügt wird.
- Durch die Methode *getComponentLocation()* wird bestimmt an welcher Position im *ContentPanel* diese angezeigt wird. Dies wird

Das Enum *Location* entscheidet hierbei wo im *ContentPanel* das Gui-Objekt angezeigt wird.

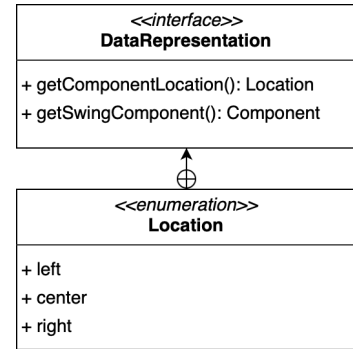


Abbildung 4: Das DataRepresentation Interface und sein Location enum

Für diese Arbeit wurden folgende Gui-Klassen implementiert:

2.2.1 Graphen

Graphen werden durch die Klasse *Graph* und zwei Subklassen *Edge* und *Node* realisiert. Da das Schreiben einer eigenen Engine für die Darstellung von Graphen diese Bachelorarbeit übertreffen würde wurde entschieden eine externe Library zu verwenden. Hier wurde sich für GraphStream[GTc] entschieden.

Graphstream ist eine Library zum modellieren, analysieren und visualisieren von Graphen, im Framework wird sie jedoch nur benutzt um Graphen zu visualisieren.

Um einen Graphen im *ContentPanel* anzeigen zu lassen gibt die Methode *getSwingComponent()* ein *JPanel* wieder, in diesem *JPanel* befindet sich ein *GraphStream View* Objekt welches von einem *SwingViewer* erstellt wird, welcher in einem eigenen Thread(siehe Listing 4) einen *MultiGraph* rendert(siehe Abb. 5). Dies stellt die default Renderingstrategie dar[GTb].

Graphstream bietet einige verschiedene Implementierungen von Graphen an, die je nach Usecase unterschiedliche Vorteile haben. Hier wurde sich dafür entschieden nur die *MultiGraph* implementierung zu verwenden, da es möglich sein muss mehrere Kanten zwischen denselben zwei Knoten

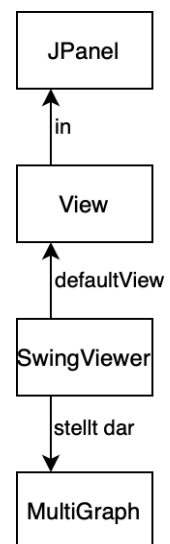


Abbildung 5: Rendering Pipeline eines Graphen

zu haben.² Diese Implementierung wird laut Dokumentation nur von einem *MultiGraph* implementiert.[GTa]

Nun wird im GUI ein leerer Graph angezeigt.

Um dem *MultiGraph* nun einen Knoten hinzuzufügen benutzen wir die Methode *addNode(String)*. Diese erwartet einen String als ID, dementsprechen müssen alle Knoten die wir hinzufügen einen eindeutigen String haben.

Um eine Kante hinzuzufügen wird die Methode *addEdge(String, String, String, boolean)* hierbei soll der erste String die eindeutige ID der Kante sein und die beiden folgenden Strings die IDs der jeweiligen Aus- und Eingangsknoten. Der Boolean-Wert gibt hierbei an ob die Kante eine gerichtete Kante ist, oder nicht. Um Knoten oder Kanten zu entfernen gibt es die Methoden *removeNode(String)* und *removeEdge(String)* die nach demselben Prinzip agieren.

Um das Aussehen eines Graphen anzupassen werden Attribute benutzt[GTd]. Attribute können Graphen, Knoten und Kanten mit der Methode *setAttribute(String, Object)* hinzugefügt werden, hierbei ist der erste String das Attribut und das folgende Objekt der Wert des Attributes. Im Falle dieser Arbeit sind zwei Attribute relevant:

- Das generelle „look and feel“ des Graphen wird durch das „ui.stylesheet“ Attribut bestimmt. Dieses setzen wir für das Graph-Objekt mit *graph.setAttribute(„ui.stylesheet“, „[...]“)*
- Um die einzelnen Knoten voneinander unterscheiden zu können, gibt es die Möglichkeit diese mit einem Text zu versehen. Das Attribut hierfür ist „ui.label“. Um einem Knoten ein Label zu geben ist unser Methodenaufruf folglich *node.setAttribute(„ui.label“, „[...]“)*. Da wir diese Methode auf einem node-Objekt aufrufen, dies aber nicht selber erstellen, benötigen wir ausserdem die Methode *graph.getNode(String)*, bei der der übergebene String wieder die ID des Knotens ist.

²Es wird die Möglichkeit benötigt eine Kante $a \rightarrow b$ und eine Kante $b \rightarrow a$ gleichzeitig darzustellen

Um unabhängig von *GraphStream* mit Graphen umzugehen zu können wurde mit folgender (Abb. 6) Abstraktion gearbeitet:

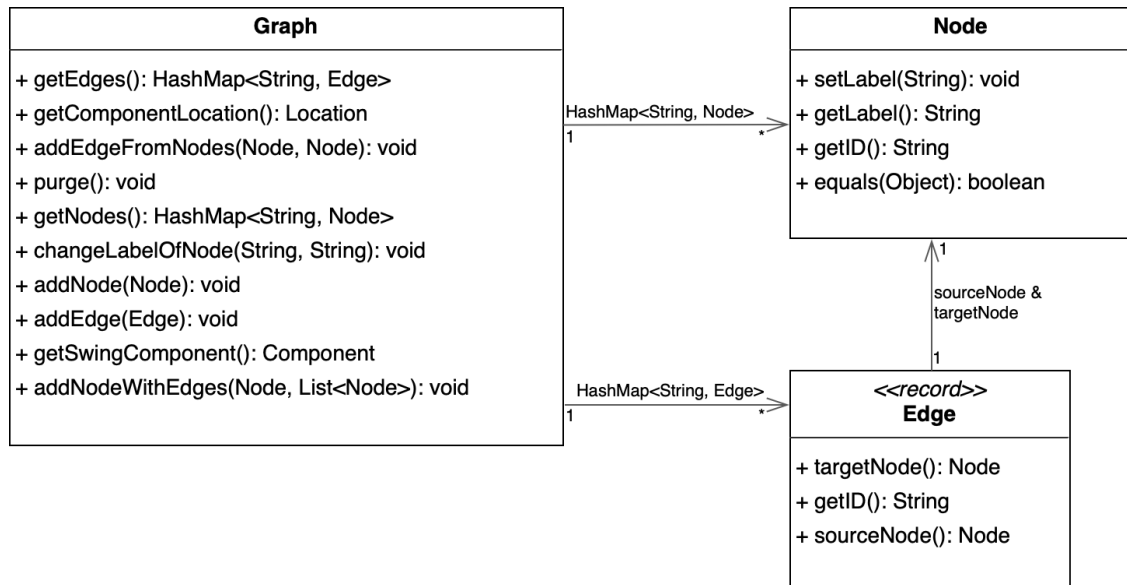


Abbildung 6: Klassenstruktur der Graph-Klassen

Hierbei sind die Klassen *Node* und *Edge* Helferklassen, um anstelle von Strings mit Node- beziehungsweise Edge-Objekten zu arbeiten.

Einem *Node*-Objekt wird zu ihrer Erstellung eine eindeutige ID zugewiesen um das Objekt später mit in einem *GraphStream*-Graphen verwenden zu können.

Ein *Edge*-Objekt ist ein Record welcher zwei Node-Objekte *sourceNode* und *targetNode* beinhaltet. Die für *GraphStream* eindeutige ID bestimmt sich dann durch folgende Methode:

Listing 3: Generieren der ID für eine Kante

```

public String getID(){
    return "(" + sourceNode.getID() + ")_->_(" + targetNode.getID() + ")";
}

```

Die *Graph* Klasse ist die zentrale Klasse für die Visualisierung von Graphen. Sie implementiert das *DataRepresentation* Interface und alle nötige Kommunikation mit der *GraphStream* Library. Wir erstellen also einen neuen Graphen wie folgt:

Listing 4: Konstruktor der Klasse *ThreeAddressCodeInstruction*

```
public Graph(Location location){
    this.location = location;
    exportedPanel = new JPanel(new BorderLayout());
    switch(location){...} // Groesse des exportierten JPanels festlegen

    nodes = new HashSet<>();
    edges = new HashSet<>();

    graph = new MultiGraph("Graph");
    graph.setAttribute("ui.stylesheet", ...); // stylesheet festlegen
    viewer = new SwingViewer(graph, ...); // threading festlegen
    View view = viewer.addView(false);

    layout = new LinLog();
    viewer.enableAutoLayout(layout);

    exportedPanel.add((Component) view, BorderLayout.CENTER);
}
```

Die im Graph enthaltenen Knoten und Kanten werden in einem *Set* gespeichert, wenn ein Plugin dem Graphen einen Knoten hinzufügen möchte, ruft er die Methode *addNode(Node)*(Listing 5) auf. Diese fügt den Knoten dem Set der im Graphen enthaltenen Knoten hinzu und fügt den Graphen einen neuen Knoten mit der ID des übergebenen Knotens hinzu. Anschließend wird die Methode *layout.shake()* aufgerufen. diese Methode sorgt dafür dass die Anordnung der Knoten neu berechnet wird, sodass diese immer einen angemessenen Abstand zu einander haben und sich nicht überlappen.

Listing 5: Implementierung der *addNode(Node)* Methode

```
public void addNode(Node newNode){
    if (nodes.contains(newNode))
        return;
    nodes.add(newNode);
    graph.addNode(newNode.getId());
    layout.shake();
}
```

Analog gibt es auch die Methode *addEdge(Edge)* um Kanten hinzuzufügen und die Methoden *removeEdge(Edge)* und *removeNode(Node)* um diese wieder zu entfernen.

Um den Graphen komplett zu leeren wurde die Methode *purge()* hinzugefügt. diese entfernt alle Knoten und Kanten vom Graphen.

Um einem Knoten ein Label zu geben, wurde die Methode *setLabelOfNode(Node, String)*(Listing 6) hinzugefügt.

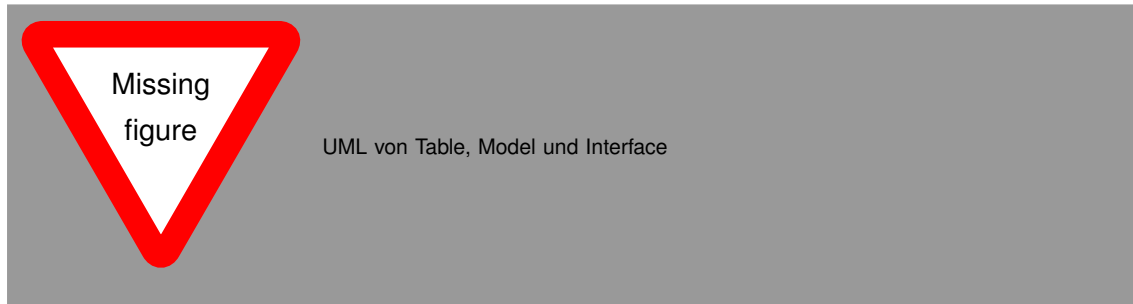
Listing 6: Implementierung der *setLabelOfNode(Node, String)* Methode

```
public void setLabelOfNode(Node Node, String label){  
    if (nodes.contains(newNode))  
        graph.getNode(node).setAttribute("ui.label", label);  
}
```

2.2.2 Tabellen

Um 3AC und ermittelte Datenflusswerte darzustellen wurde ausserdem die Möglichkeit implementiert, Daten als Tabellen darzustellen. Diese ist in der Klasse *Table* welche die *swing* Komponente *JTable* erweitert.

erklären,
sse erklären



2.3 Drei Address Code

Im folgenden Kapitel erläutere ich die Implementierung des drei Address Codes. Wie zu Beginn von Kapitel 2 angeführt werden diese Klassen benötigt:

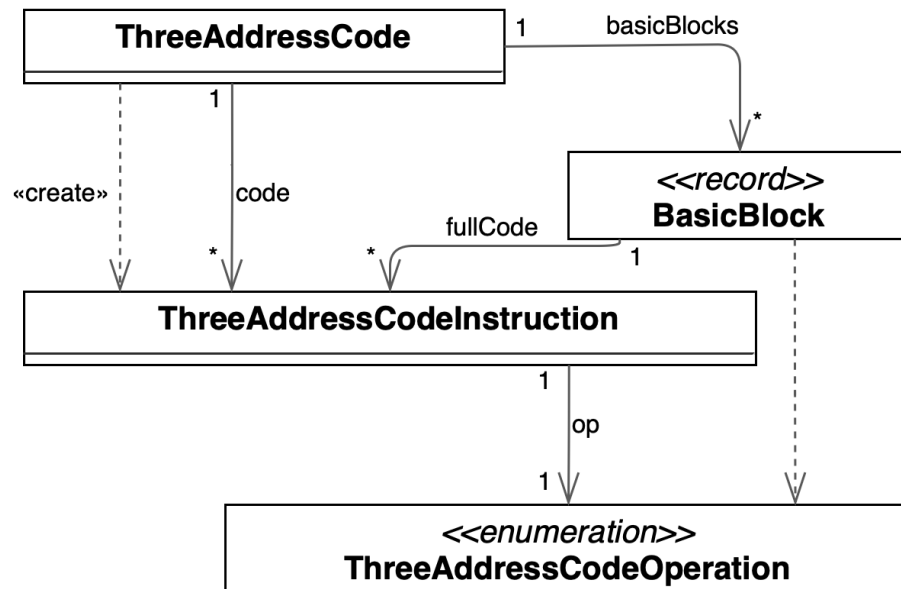


Abbildung 7: Implementation der drei Address Code Klassen

2.3.1 Die ThreeAddressCodeInstruction Klasse

Um einzelne Instruktionen zu modellieren wird die Klasse *ThreeAddressCodeInstruction* verwendet. Generiert werden diese aus einem *String* und einer Zahl, nämlich der Adresse an der Stelle ihres Programms. Dies ist zwar streng gesehen eine Ungenauigkeit der Modellierung, da jede Instruktion zwar eine Adresse hat, diese jedoch nicht in Bezug direkt auf die einzelne Instruktion steht, sondern nur im Kontext des Gesamtprogrammes gesehen werden kann. Da es jedoch die Implementierung einiger folgender Methoden stark vereinfacht und um aktuellen Usecase des Frameworks keine Instruktion ohne eindeutige Adresse existiert wurde sich dazu entschieden für diese ein Attribut anzulegen.

ThreeAddressCodeInstruction
+ canJump(): boolean + getComment(): String + writesValue(): boolean + getUsedIdentifiers(): Collection<String> + setComment(String): void + compareTo(ThreeAddressCodeInstruction): int + nextPossibleInstructionAddresses(): Set<Integer> + getDestination(): String + getAddress(): int + getRepresentationAsStringArray(): String[] + getOperation(): ThreeAddressCodeOperation

Abbildung 8: Die ThreeAddressCodeInstruction Klasse

Um die in Kapitel 1.2.1 definierten Instruktionen zu modellieren werden folgende weitere Attribute benötigt:

1. Eine *ThreeAddressCodeOperation* dies ist ein enum, welches definiert welche Operation die angegebene Instruktion ausführt.
2. Ein *String* namens *destination*. Dieser gibt entweder das Ziel eines Sprunges an, oder den Ort an dem eine Berechnung gespeichert werden soll.
3. Den *String* *source*. Da alle Instruktionen ausser dem unbedingten Sprung mindestens einen Wert verarbeiten.
4. und einen *String* *modifier*, für Instruktionen die zwei Werte entweder vergleichen oder arithmetisch verrechnen.

Desweiteren wurde ein weiteres *String*-Attribut zum hinzufügen von Kommentaren hinzugefügt.

Um aus dem Eingabestring ein passendes *ThreeAddressCodeInstruction*-Objekt zu generieren wird der Eingabestring im Konstruktor der Klasse nach Leerzeichen aufgeteilt und wie folgt die richtige Operation ausgewählt:

Listing 7: Konstruktor der Klasse *ThreeAddressCodeInstruction*

```
public ThreeAddressCodeInstruction(String rawInput, int address){
    ...
    String[] pieces = rawInput.split(" ");
    switch (pieces.length) {
        case 2 -> ... //Unbedingter Sprung goto X
        case 3 -> ... //Kopierbefehl X = Y
        case 4 -> { //entweder unaere Operation oder ein bedingter Sprung
            switch(pieces[2]){
                case "-" -> ... //Unaere Operation X = - Y
                case "goto" -> { //ein bedingter Sprung
                    switch(pieces[0]){
                        case "if" -> ... //bedingter Sprung if Y goto X
                        case "ifFalse" -> ... //bedingter Sprung ifFalse Y goto X
                    }
                }
            }
        }
        case 5 -> ... //Binaere Operation X = Y op Z
        case 6 -> { //bedingter Sprung
            switch(pieces[2]){
                case "<" -> ... //if Y < Z goto X
                case ">" -> ... //if Y > Z goto X
                case "<=" -> ... //if Y <= Z goto X
                case ">=" -> ... //if Y >= Z goto X
                case "==" -> ... //if Y == Z goto X
                case "!=" -> ... //if Y != Z goto X
            }
        }
        ...
    }
}
```

An den durch "..."-markierten Stellen die zu gehörigen Attribute initialisiert.

Ausserdem wurden folgende Helfermethoden, welche im späteren Verlauf dieser Arbeit benötigt werden implementiert:

1. Die Methode *canJump()* gibt den Wahrheitswert *wahr* zurück, wenn die Instruktion springen kann oder immer springt, ansonsten gibt sie *falsch* zurück.
2. Die Methode *writesValue()* gibt den Wahrheitswert *wahr* zurück, wenn die Instruk-

tion den Wert der in *destination* gespeichert ist überschreibt. Ansonsten gibt sie *falsch* zurück.

3. *getUsedIdentifiers()* gibt die Menge der Werte in *source* und *modifier* zurück, wenn diese Variablen referenzieren und keine Konstanten sind.
4. Die Methode *compareTo(ThreeAddressCodeInstruction)* implementiert das Interface *Comparable<>*. Wir sehen später dass dies sinnvoll ist wenn in Erfahrung gebracht werden soll sich ob zwei Instruktionen im selben Grundblock befinden.
5. *nextPossibleInstructionAddresses()* gibt die Menge der nächsten möglichen Instruktionsadressen zurück. Dies ist in den meisten Fällen einfach die nächste Adresse, wenn die aktuelle Instruktion jedoch ein unbedingter Sprung ist, ist es der Wert in *destination*, wenn die Instruktion ein bedingter Sprung ist, wird sowohl die nächste Adresse als auch der Wert in *destination* zurückgegeben.
6. Die Methode *getRepresentationAsStringArray()* gibt eine Repräsentation der Instruktion als *String[]* in der Form zurück, dass sie nacheinander die eingelesene Instruktion bilden zum Beispiel:

Listing 8: "Rückgabewert von *getRepresentationAsStringArray(true)*"

```
new String []{ address , destination , "=", source , op.getRepresentation ()
```

Die Methode *toString()* vereinigt das Array in einem String.

Die restlichen in *ThreeAddressCodeInstruction* enthaltenen Klassen sind getter und setter Methoden.

2.3.2 Die BasicBlock Record-Klasse

Um die Grundblöcke eines Programmes zu modellieren reicht uns eine Record-Klasse, da Grundblöcke immer ein gesamtes Programm partitionieren reicht es die erste und letzte der enthaltenen und die Menge der folgenden Adressen zu kennen.

<<record>> BasicBlock
+ lastAddress(): int + firstAddressesOfSuccessors(): List<Integer> + firstAddress(): int

Abbildung 9: Die *ThreeAddressCodeInstruction* Klasse

2.3.3 Die ThreeAddressCode Klasse

Alles

2.4 Implementierung von Algorithmen

Dieser Abschnitt befasst sich damit wie Algorithmen als Plugins implementiert werden, sodass diese vom Framework richtig visualisiert werden.

Hierbei gibt es zwei Interfaces die implementiert werden, das *Plugin* Interface und das *ToolBarButton* Interface. Letzteres spezifiziert alle Buttons welche in der Toolbar angezeigt werden wenn das Plugin geladen ist. Es ist sehr einfach aufgebaut und verfügt nur über zwei Methoden. Eine Methode *getText()* gibt einen String zurück, welcher im Button angezeigt wird. Die andere Methode *action()* wird ausgeführt wenn der Nutzer auf den zugehörigen Button drückt.

Da alle in dieser Arbeit implementierten Algorithmen sowohl Schrittweise durchlaufen werden sollen als auch Code laden müssen wurden ausserdem zwei Interfaces implementiert welche *ToolBarButtons* für Plugins bereitstellen.

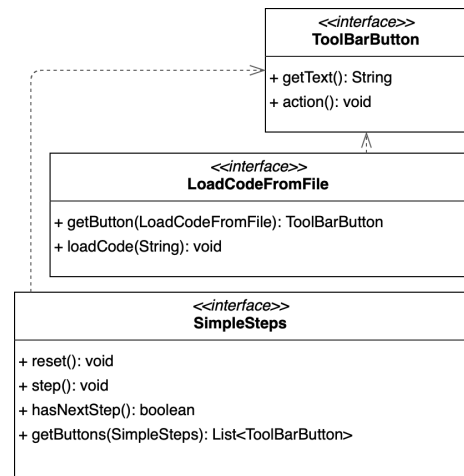


Abbildung 10: Das Plugin Interface

Das Interface *LoadCodeFromFile* stellt sicher dass es eine Funktion *loadCode(String)* gibt, in die Code als ein *String* in das Plugin geladen werden kann.

Die Funktion *getButton()* ist hier eine statische Funktion, welche eine Implementation von *ToolBarButton* zurückgibt welche zuerst einen *JFileChooser*³ öffnet und dann die Datei als *String* lädt. Hierbei wird auch der Unterschied zwischen Windows und *nix basierenden Systemem berücksichtigt und die einzelnen Zeilen nur mit einem Zeilenumbruch(\n) konkateniert.

Das Interface *SimpleSteps* implementiert sogar drei Buttons und implementiert drei Methoden. Ein Reset Button, der die Methode *reset()* aufruft, ein Step Button, der die Methode *step()* aufruft und einen Run Button, der die Methode *step()* so lange aufruft wie die Methode *hasNextStep()* *wahr* zurückgibt. Diese drei Buttons werden wieder in einer statischen Funktion *getButtons()* generiert.

³ *JFileChooser* ist eine Klasse aus der *swing*-Library welche eine Bedienfläche zum öffnen von Dateien bietet

2.4.1 Das Plugin Interface

Das *Plugin* definiert die Hauptklasse des hinzugefügten Plugins.

1. *onPluginLoad()* wird ausgeführt wenn das Plugin geladen wird, also zum Beispiel wenn der Nutzer im Plugin Dropdown Menü auf den Button mit dem Namen des Plugins drückt.
2. *getToolBarButtons()* gibt eine Collection der implementierten *ToolBarButtons* zurück, sodass diese wenn das Plugin geladen wird in die *ToolBar* eingefügt werden.
3. *getName()* gibt den Namen des Plugins zurück, so dass er im Plugin Dropdown Menü angezeigt wird.
4. *getGuiElements()* gibt eine Collection der *DataRepresentation* Elemente zurück die das Plugin anzeigen soll.

<<interface>> Plugin
+ onPluginLoad(): void + getToolBarButtons(): Collection<ToolBarButton> + getName(): String + getGuiElements(): Collection<DataRepresentation>

Abbildung 11: Das Plugin Interface

Mehr Abstraktionen braucht das Framework nicht um alle Algorithmen die in dieser Arbeit implementiert werden sollten zu Visualisieren. Im nächsten Kapitel wird nun auf die spezifische Implementationen der Algorithmen eingegangen.

3 Implementierte Algorithmen



3.1 Erstellung von Grundblöcken aus 3-Address-Code

3.2 Erstellung eines Kontrollflussgraphen aus 3-Address-Code

3.3 Analyse von erreichenden Definitionen

3.4 Analyse von lebendigen Variablen bezüglich Grundblöcken

3.5 Analyse von lebendigen Variablen bezüglich einzelnen Instruktionen

Alles

3.6 Optimierung von Grundblöcken mit Konstantenfaltung

konnte leider nicht implementiert werden.

4 Related Work

Alles

5 Evaluation

Alles

6 Future Work

Alles

7 Fazit

Alles

Abbildungsverzeichnis

1	Resultierender Kontrollflussgraph	5
2	Klassenstruktur der GUI-Klassen	8
3	Ansicht des Programmes direkt nach Start	10
4	Das DataRepresentation Interface und sein Location enum	11
5	Rendering Pipeline eines Graphen	11
6	Klassenstruktur der Graph-Klassen	13
7	Implementation der drei Address Code Klassen	17
8	Die ThreeAddressCodeInstruction Klasse	17
9	Die ThreeAddressCodeInstruction Klasse	20
10	Das Plugin Interface	22
11	Das Plugin Interface	23

Tabellenverzeichnis

Literatur

- [Aho08] AHO, Alfred V. ; LEUSCHEL, Michael (Hrsg.): *Compiler Prinzipien, Techniken und Werkzeuge*. 2., aktualisierte Aufl., German language ed. München u.a. : Pearson Studium, 2008 (It, Informatik). https://digitale-objekte.hbz-nrw.de/storage/2008/03/03/file_131/2343468.pdf
- [GTa] GRAPHSTREAM-TEAM: *Graph Implementations*. <https://graphstream-project.org/doc/FAQ/The-Graph-Class/What-are-the-Graph-implementations/>. – zuletzt besucht am 30.01.2025
- [GTb] GRAPHSTREAM-TEAM: *Graph Visualisation*. <https://graphstream-project.org/doc/Tutorials/Graph-Visualisation/>. – zuletzt besucht am 30.01.2025
- [GTc] GRAPHSTREAM-TEAM: *GraphStream*. <https://graphstream-project.org/>. – zuletzt besucht am 30.01.2025
- [GTd] GRAPHSTREAM-TEAM: *Storing, retrieving and displaying data in graphs*. <https://graphstream-project.org/doc/Tutorials/Storing-retrieving-and-displaying-data-in-graphs/>. – zuletzt besucht am 31.01.2025