Hochschule Merseburg

Fachbereich Ingenieur- und Naturwissenschaften

Studiengang Angewandte Informatik

Bachelorarbeit

Automatischer Abgleich und Bewertung von Programmablaufplänen

vorgelegt bei
Prof. Dr. Sven Karol

Zweitprüfer: Prof. Dr. Andreas Spillner

Eingereicht von: Sören Taube
Matrikel: 27675

Abgabetermin: 10.12.2024

Hochschule Merseburg
Prof. Dr. Sven Karol
Eberhard-Leibnitz-Straße 2
06217 Merseburg
Telefon + 49 3461 46 2253
E-Mail: sven.karol@hs-merseburg.de

HoMe HOCHSCHULE MERSEBURG University of Applied Sciences

Aufgabenstellung für die Bachelorarbeit

von

Sören Taube

Thema: Automatischer Abgleich und Bewertung von Programmablaufplänen

Erstbetreuer: Prof. Dr. Sven Karol
Zweitbetreuer: Prof. Dr. Andreas Spillner

Aufgabenstellung

Programmablaufpläne sind ein einfaches Mittel, den Ablauf eines imperativen Programms oder eines Algorithmus grafisch darzustellen. In den einführenden Lehrveranstaltungen zur Programmierung werden sie daher häufig eingesetzt, um den Ablauf von Programmen zu visualisieren oder das Programmverständnis von Studierenden zu überprüfen. Üblicherweise werden Programmablaufpläne als verbundene, gerichtete Graphen dargestellt, die aus Anfangs- und Endknoten, Anweisungsknoten, Verzweigungsknoten sowie Kanten zwischen den Knoten bestehen.

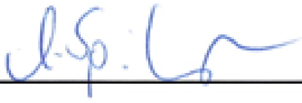
Während sich die syntaktische Integrität eines Programmablaufplanes sehr leicht automatisiert überprüfen lässt, ist die automatische Bestimmung der Korrektheit bezüglich eines Programms schwieriger zu realisieren. Zwar lässt sich für ein imperatives Programm leicht ein Kontrollflussgraph erzeugen, der genau dem Ablauf des Programms entspricht. Da Programmablaufpläne jedoch manuell erstellt werden, weichen sie häufig von automatisch erzeugten Kontrollflussgraphen ab, auch wenn sie objektiv als richtig zu bewerten sind. Das kann beispielsweise durch die Beschriftung von Knoten, das Zusammenfassen von Anweisungen oder einem höheren Abstraktionsgrad verursacht werden. Darüber hinaus müssten auch Programmablaufpläne bewertet werden, die nur teilweise richtig sind, weil zum Beispiel eine Schleife vergessen oder eine Anweisung falsch verbunden wurde.

Das Ziel dieser Arbeit besteht daher darin, einen Ansatz zu entwickeln, der die Korrektheit eines gegebenen Programmablaufplanes bezüglich eines Programms ermittelt und auch teilkorrekte Lösungen bewerten kann. Auf Grundlage des entwickelten Ansatzes soll ein passendes Programm erstellt werden, welches sich als Bibliothek leicht integrieren und von der Kommandozeile aufrufen lässt. Nach Möglichkeit soll auch eine grafische Benutzeroberfläche erstellt und in ein bestehendes Softwarewerkzeug integriert werden.

Schwerpunkte

1. Einarbeitung in gerichtete Graphen, Kontrollflussgraphen und Programmablaufpläne
2. Identifizierung geeigneter Algorithmen zum Vergleich gerichteter Graphen bzw. Kontrollflussgraphen

3. Entwicklung eines Ansatzes zum Vergleich einfacher C-Programme mit korrekten und teilkorrekten Programmablaufplänen
4. Implementierung des Ansatzes in einem prototypischen Programm
5. Gegebenenfalls Einbindung des Programms in eine bereits bestehende .NET-Anwendung



Prof. Dr. Andreas Spillner	Prof. Dr. Sven Karol
Vorsitzender des Prüfungsausschusses	Themenstellender Hochschullehrer

Kurzfassung

Programmablaufpläne dienen bei der Vermittlung von Kenntnissen zur Programmierung als Hilfsmittel, um das Verständnis für Kontroll- und Datenflüsse beim Ablauf von Programmen zu überprüfen. Dies erfolgt häufig durch eine Umsetzung des Plans in ein lauffähiges Programm und anschließendem Abgleich auf Übereinstimmung. Um diesen Abgleich und eine Bewertung der abgegebenen Leistung möglichst effizient und zeitnah vornehmen zu können, bedarf es einer automatisierten Lösung.

Das Ziel dieser Arbeit ist es, einen Lösungsansatz auf Basis der Graphentheorie zu entwickeln und prototypisch umzusetzen. Dabei sollen die Repräsentation von Programm und Ablaufplan als Kontrollflussgraphen auf Ähnlichkeit untersucht und Bewertungen auch für teilkorrekte Umsetzungen vorgenommen werden können. Eine auftretende Schwierigkeit dabei ist es, den Unterschied im Abstraktionsgrad der Anweisungen in beiden Graphen zu überwinden und diese auf geeignete Weise vergleichbar zu machen.

Im Verlauf dieser Arbeit werden Ansätze geliefert, wie Programm und Programmablaufplan als Kontrollflussgraphen in ein geeignetes Vergleichsformat gebracht werden können, wie auf Basis dieses Formats eine strukturelle Analyse durch Ermittlung gemeinsamer Untergraphen erfolgt und wie durch Berechnung der Graph-Edit-Distanz eine Grundlage für die Bewertung geschaffen wird. Darüber hinaus wird eine Metrik erarbeitet, welche die Schwierigkeit der Vergleichbarkeit von Anweisungen behandelt.

Als Ergebnis steht eine prototypische Umsetzung dieser Ansätze für den Abgleich und die Bewertung für einfache C-Programme und zugehörige Programmablaufpläne, welche im Umfeld der Hochschularbeit zum Einsatz kommen kann, um studentisches Verständnis im Rahmen der Einführungsveranstaltungen für die Programmierung effektiv zu überprüfen.

Abstract

Program flowcharts are used as an aid when teaching programming skills to check the understanding of control and data flows in running programs. This is often done by converting the plan into an executable program and then comparing it for compliance. An automated solution is required to carry out this comparison and make an evaluation of the output as efficiently and promptly as possible.

The aim of this thesis is to develop and prototype a solution based on graph theory. The representation of program and flow chart as control flow graphs should be examined for similarity and evaluations should be made possible for partially correct implementations. One difficulty that arises is to overcome the difference in the degree of abstraction of the instructions in both graphs and to make them comparable in a suitable way.

In the course of this work, approaches are provided as to how program and program flow chart can be brought into a suitable comparison format as control flow graphs, how a structural analysis is carried out based on this format by determining common subgraphs and how a basis for the evaluation is created by calculating the graph edit distance. In addition, a metric is developed that deals with the difficulty of comparing statements.

The result is a prototypical implementation of these approaches for the comparison and evaluation of simple C programs and associated program flowcharts, which can be used in the context of university work to effectively test student understanding in the context of introductory programming courses.

Inhaltsverzeichnis

Abbildungsverzeichnis ... **V**

Tabellenverzeichnis ... **VI**

Listings ... **VII**

Abkürzungsverzeichnis ... **VIII**

1 Einleitung ... **1**

1.1 Gegenstand und Motivation ... 1

1.2 Anforderungen und Zielsetzung ... 2

1.3 Gliederung der Arbeit ... 3

2 Theoretische Grundlagen ... **4**

2.1 Graphentheorie ... 4

2.1.1 Allgemeine Einführung in Graphen ... 4

2.1.2 Graphenisomorphie ... 6

2.1.3 Traversierung ... 8

2.1.4 Maximum Common Subgraph ... 10

2.2 Metriken zur Ähnlichkeitsbestimmung ... 12

2.2.1 Levenshtein-Distanz ... 12

2.2.2 Graph Edit Distance ... 14

2.2.3 Struktur abstrakter Syntaxbäume ... 16

2.2.4 Jaccard-Index ... 17

2.3 Kontrollflussgraphen ... 18

2.4 Programmablaufplan ... 20

3 Konzeption ... **23**

3.1 Rahmenbedingungen und Gesamtkonzeption ... 23

3.1.1 Spezifikation der Zielgraphen ... 23

3.1.2 Vergleichsbasis der Graphen ... 24

3.1.3 Technische Rahmenbedingungen ... 25

3.1.4 Gesamtkonzept der Planung ... 25

3.2 Extraktion von Kontrollflussgraphen aus C-Programmen ... 28

3.2.1 Compiler-Auswahl ... 28

3.2.2 Erstellung der internen CFG-Repräsentation ... 31

3.3 Import manuell erzeugter Programmablaufpläne ... 33

3.3.1 Auswahl des Werkzeugs zur PAP-Erstellung ... 33

3.3.2 Erstellung der internen FC-Repräsentation 35

3.4 Ermittlung struktureller Gemeinsamkeiten von Graphen 37

3.5 Entwurf einer Bewertungsgrundlage für teilkorrekte Ablaufpläne 39

3.6 Entwicklung einer Vergleichsmöglichkeit von Knoteninhalten 42

4 Umsetzung 45

4.1 Aufbau des Lösungsansatzes und Grundfunktionalitäten 45

4.2 Erstellung interner Graphenrepräsentation aus Compiler-Daten 49

4.3 Nutzung von XML basiertem Importformat zur Graph-Generierung 54

4.4 Bestimmung des Maximum Common Connected Subgraphs 57

4.5 Implementierung eines Algorithmus für die Graph Edit Distance 62

4.6 Integration einer zeichenketten- und strukturbasierten Metrik 65

4.7 Konsolenanwendung und Gesamtbewertung 69

5 Fazit 71

5.1 Zusammenfassung 71

5.2 Ausblick 72

Literaturverzeichnis 73

Anhang i

A1 Vollständiger Kontrollflussgraph im LLVM (Clang)-Compiler i

A2 Vollständiger Kontrollflussgraph im GCC-Compiler ii

A3 Testreihen zur Laufzeitermittlung von Positionsvergleichen iii

Eidesstattliche Erklärung iv

Abbildungsverzeichnis

Abbildung 1: Reihenfolge der Traversierung bei Tiefensuche 8
Abbildung 2: Reihenfolge der Traversierung bei Breitensuche 9
Abbildung 3: Pseudocode eines Zustandsraum-Suchalgorithmus für die MCS Suche 10
Abbildung 4: Beispiel Aufbau Programmablaufplan 21
Abbildung 5: Grundstruktur der betrachteten Graphen 23
Abbildung 6: Entwicklung des Gesamtkonzepts 26
Abbildung 7: Exemplarisches C-Programm mit Graphenrepräsentation 27
Abbildung 8: Visuelle Repräsentation in Clang 29
Abbildung 9: Visuelle Repräsentation in GCC 30
Abbildung 10: Parsen der internen Repräsentation 31
Abbildung 11: Grafische Erzeugung des PAP in draw.io 35
Abbildung 12: vollständiges Matching 38
Abbildung 13: unvollständiges Matching 38
Abbildung 14: GED Ermittlung anhand der Rest-Knoten 39
Abbildung 15: Bestandteile der Metrik zum Labelvergleich 42
Abbildung 16: Laufzeit Positionsvergleich 60
Abbildung 17: Häufigkeit von Abweichungen bei MCCS Bestimmung isomorpher Graphen 61
Abbildung 18: Durchführung exemplarischer Ähnlichkeitsmessung 68
Abbildung 19: Windows-Ausführung der Konsolenanwendung 69
Abbildung 20: Ausgabe der Konsolenanwendung 70

Tabellenverzeichnis

Tabelle 1: Vergleich verschiedener Werkzeuge zur PAP-Erstellung 34
Tabelle 2: Übersicht einer Auswahl an Metazeichen für reguläre Ausdrücke in C#..................50

Listings

Listing 1: Beispiel draw.io-Graph in XML....36
Listing 2: Knoten-Klasse und dazugehörige Attribute....46
Listing 3: Interne Graphen-Klasse mit Attributen....46
Listing 4: Klasse zur Konfiguration der Bibliothek....47
Listing 5: Konfigurationsdatei der Bibliothek im JSON-Format....47
Listing 6: Erweiterung von Graphen auf eine Anweisung pro Block....48
Listing 7: Prozess mit Compiler-Ausführung zur Erzeugung des CFG....49
Listing 8: Erstellung von Knoten und Kanten....51
Listing 9: Auslesen der temporären Return-Variable....51
Listing 10: Beispiel für temporäre Variablen und Static Single Assignment....52
Listing 11: Rückführung von Static Single Assignments und temporären Variablen....52
Listing 12: Extraktion der XML-Datei....54
Listing 13: Erstellung des internen FC....55
Listing 14: Erstellung einer eigenen Graph XML....55
Listing 15: Erstellung eigener XML-Schema-Definition....55
Listing 16: draw.io spezifische Anpassung der Labelinhalte....56
Listing 17: Rekursive Ermittlung des MCCS mittels Tiefensuche....57
Listing 18: Berechnung der kürzesten Wege mittels Breitensuche....59
Listing 19: Positionsvergleich von zwei Knoten innerhalb ihrer Graphen....59
Listing 20: MCCS Ermittlung mit Positionsvergleich bei Over-Matching....60
Listing 21: Rückgabe bei vollständigem Matching....62
Listing 22: Behandlung bei Match von Knoten mit gleichem Label....62
Listing 23: Einfügen eines Knotens bei kompletter Abweichung....63
Listing 24: Relabeling eines Knoten bei gleicher Nachbarschaft....63
Listing 25: Berechnung der String-Distanz....65
Listing 26: Berechnung der String-Ähnlichkeit....65
Listing 27: Erstellung des Syntaxbaums eines Labels....66
Listing 28: Berechnung der syntaktischen Ähnlichkeit....67
Listing 29: Berechnung der semantischen Ähnlichkeit....67
Listing 30: Berechnung der strukturellen Ähnlichkeit und Gesamtähnlichkeit....67
Listing 31: Labelvergleich mit Schwellenwert....68
Listing 32: Grundlogik der Konsolenanwendung....69

Abkürzungsverzeichnis

AST	- Abstract Syntax Tree
BFS	- Breadth First Search
CFG	- Control Flow Graph
CLI	- Command-Line Interface
DFS	- Depth First Search
FC	- Flow Chart
FIFO	- First-In-First-Out
GCC	- GNU Compiler Collection
GED	- Graph Edit Distance
LIFO	- Last-In-First-Out
MCS	- Maximum Common Subgraph
MCCS	- Maximum Common Connected Subgraph
PAP	- Programmablaufplan
XML	- Extensible Markup Language
XSD	- XML Schema Definition

1 Einleitung

1.1 Gegenstand und Motivation

Bei der Erlangung von Kenntnissen über die Programmierung ist neben der reinen Aneignung von Wissen über die syntaktischen Gestaltungselemente auch der Aufbau eines Verständnisses für Programmabläufe und Datenflüsse von enormer Bedeutung.

Programmabläufe können mit wachsendem Umfang und mit zunehmend tiefgreifender Verschachtelung der Anweisungen das Programm immer schlechter nachvollziehbar werden lassen. Dies erschwert das Auffinden von Fehlern und inkorrekten Datenzuständen nicht nur für den Programmieranfänger, sondern bei ausgeweiteten Softwareprodukten auch für den erfahrenen Softwareentwickler.

Um Abhilfe für diesen Umstand zu schaffen, wurde bereits zu Zeiten fast ausschließlich imperativer Programmierung ein Hilfsmittel geschaffen, welches durch visuelle Repräsentation des Programmablaufs, Klarheit über Anweisungsfolgen und Datenflüsse schaffen soll: der Programmablaufplan (engl. flow chart) [1]. Programmablaufpläne sind mit ihren standar–disierten und von Programmiersprachen unabhängig anwendbaren Stilelementen geeignet, in verschiedenen Kontexten und Abstraktionsgraden in der Softwarenentwicklung eingesetzt zu werden. Mit ihrer Hilfe können damit sowohl präzise, detailliert beschriebene Programme, als auch grobe, unspezifische Abläufe modelliert werden.

Diese vielseitige Einsatzmöglichkeit wird in Einstiegsveranstaltungen zur Programmierung gerne genutzt, um Studierenden einfache Konzepte von Programmabläufen und Datenflüssen mit dafür geeigneten syntaktischen Anweisungen zu vermitteln. Darüber hinaus kann mit Hilfe von Programmablaufplänen aber auch Programmverständnis und korrekte Syntaxanwendung überprüft werden. Diese Überprüfung erfolgt entweder dadurch, dass bestehende Programmablaufpläne in konkrete, lauffähige Programme in einer Programmiersprache nach Wahl zu übersetzen, oder aus vorliegendem Programmcode Programmablaufpläne in einem bestimmten Abstraktionsgrad zu erstellen sind.

Die Konsequenz der Nutzung dieser Form der Wissensüberprüfung ist, dass erstellter Programmablaufplan und das zugehörige, lauffähige Programm auf Korrektheit zueinander abgeglichen werden müssen. Dieser Abgleich kann in komplexen Programmen bei manueller Durchführung zeitaufwendig und fehleranfällig sein. Zusätzlich muss auf dessen Basis eine Bewertung gefunden werden, welche möglichst gerecht den nachzuweisenden Wissensstand des Studierenden reflektiert. Auch dies kann sich aufgrund des unterschiedlichen Abstraktions-grads und der Verschiedenheit konkreter Programmsyntax zu menschlicher Formulierung als schwierig herausstellen. Setzt man überdies noch den Umstand, dass die Wissensüberprüfung für eine große Zahl an Studierenden parallel stattfinden muss und somit Abgleich und Bewertung in vielfacher Form möglichst zeitnah durchzuführen sind, wird der Wunsch nach Automatisierung dieses Prozesses förmlich laut, welcher mit dieser Arbeit erfüllt werden soll.

1.2 Anforderungen und Zielsetzung

Die Ausarbeitung eines Hilfsmittels, welches einen automatisierten Abgleich von Programm–ablaufplänen und zugehörigen Programmen vornimmt und auf dessen Basis eine Bewertung erstellt, ist Zielsetzung dieser Arbeit. Dem Nutzer soll es mit diesem Hilfsmittel ermöglicht werden, den Aufwand bei der Bearbeitung solcher Aufgaben zu reduzieren und eine schnelle Abarbeitung zu realisieren.

Da sich Programmablaufpläne und Programme mittels geeigneter Transformation in Form von Kontrollflussgraphen auf den mathematischen Bereich der Graphentheorie abbilden lassen, kann die Zielsetzung dahingehend abstrahiert werden, dass eine Möglichkeit gefunden werden soll diese Kontrollflussgraphen zu vergleichen, auf ihre Ähnlichkeit hin zu untersuchen und zu bewerten. Aus diesem Grund wird im Verlauf der Arbeit der Fokus vor allem auf die Lösung der Aufgabe und Zielerreichung innerhalb dieser Abstraktion gelegt.

Bei der Bearbeitung dieses Themas gibt es Anforderungen an das zu entwickelnde Hilfsmittel. So soll im Rahmen des Abgleichs von Programm und Programmablaufplan nicht nur die vollständige Korrektheit geprüft werden, sondern auch eine quantifizierbare Teilkorrektheit ermittelt werden. Auf Grundlage dieser quantifizierbaren Übereinstimmung soll ein Bewertungsmaß gefunden werden, welches die Leistung, die bei der Erstellung des Programmablaufplans erbracht wurde, angemessen reflektiert. Um eine universelle Integration in eine bestehende Anwendung zu ermöglichen, muss das Hilfsmittel zudem in Form einer Bibliothek im .NET Umfeld erstellt werden, wobei zur initialen Nutzung bereits die Einbindung in einer prototypischen Konsolenanwendung realisiert werden soll. Eine Integration der Bibliothek in die bestehende Anwendung wird zwar angestrebt, aber nicht als Teil dieser Arbeit umgesetzt.

Darüber hinaus müssen auch Einschränkungen bei der Zielsetzung getroffen werden, um einen angemessenen Rahmen für die Bearbeitung zu finden. Das zu erstellende Hilfsmittel begnügt sich von seiner Leistungsfähigkeit her zunächst mit dem Abgleich und der Bewertung von „einfachen“ C-Programmen. Das bedeutet, dass der Umfang der C-Programme auf eine einzelne, imperativ abzuarbeitende Funktion (die Main-Funktion oder eine daraus aufgerufene Funktion) begrenzt wird. Zudem beschränkt sich der Inhalt der Programme auf grundlegende Sprachelemente der Programmiersprache C (ISO/IEC 9899:2024). Dazu gehört die Verwendung von primitiven Datentypen sowie die Nutzung von Anweisungen, insbesondere mit binären Operationen (beispielsweise arithmetischen und logischen) und Kontrollstrukturen. Explizit ausgenommen aus den Betrachtungen sind daher komplexe Datentypen (wie beispielsweise Strukturen und Unions), Präprozessordirektiven und Anweisungen zur dynamischen Speicher–allokation.

1.3 Gliederung der Arbeit

Diese Arbeit gliedert sich im Folgenden in die vier Kapitel: Theoretische Grundlagen, Konzeption, Umsetzung und Fazit. Das grundsätzliche Vorgehen beruht darauf, zunächst eine Wissensbasis für relevante Konzepte und Begrifflichkeiten der Thematik zu schaffen, bevor diese dann in einem Entwurf für die Lösung der Aufgabenstellung Anwendung finden. Anhand dieses Entwurfs erfolgt anschließend die Implementierung der Lösung und die Erfassung der Umsetzungsergebnisse. Im Folgenden eine kurze Zusammenfassung spezifischer Kapitelinhalte.

Kapitel 2 beschäftigt sich mit den Grundlagen der Graphentheorie in dem Umfang, wie es die Bearbeitung der Aufgabenstellung erforderlich macht. Es wird eine allgemeine Definition von Graphen getroffen und wesentliche Eigenschaften vorgestellt. Darüber hinaus wird auf die für Vergleiche von Graphen bedeutende Graphenisomorphie eingegangen und mit der Traversierung ein Konzept vorgestellt, welches bei der Bearbeitung der Aufgabe im Folgenden von Bedeutung sein wird. Einen wesentlichen Bestandteil beim Vergleich von Graphen stellen zudem Metriken für die Ähnlichkeitsbestimmung dar, welche auch innerhalb dieses Kapitels beschränkt auf die zur Anwendung kommenden vorgestellt werden. Abschließend werden die Begrifflich–keiten Kontrollflussgraph und Programmablaufplan genauer erläutert.

In Kapitel 3 wird zunächst das Grundkonzept bei der Bearbeitung der Aufgabenstellung und die Rahmenbedingungen der Umsetzung vorgestellt. Im Anschluss werden die aus dem Grundkonzept abgeleiteten fünf zu erstellenden Komponenten jeweils in einem eigenen Abschnitt analysiert und spezifische Konzepte zur Umsetzung aufgezeigt. Diese fünf Komponenten umfassen: die Erstellung einer analysierbaren Graphenrepräsentation aus dem zugrunde liegenden C-Programm, die Überführung des manuell erzeugten Programmablaufplans ebenfalls in eine solche Repräsentation, den strukturellen Vergleich beider Graphen, die Bewertung struktureller Ähnlichkeit sowie abschließend die Schaffung einer Vergleichsmöglichkeit auf Basis der Knotenlabels, welche die Anweisungen des Programms bzw. des Programmablaufplans beinhalten.

Kapitel 4 befasst sich als logische Folge mit der Umsetzung der entwickelten Konzeptionen. Zunächst wird die generelle Umsetzung in der Entwicklungsumgebung behandelt, gefolgt von einer spezifischen Betrachtung der bereits erwähnten fünf Komponenten. Dabei werden Implementierungsdetails vorgestellt, auftretende Komplikationen und der Umgang mit diesen aufgezeigt und eine Evaluation der jeweiligen Umsetzung vorgenommen. Am Ende des Kapitels wird das Gesamtergebnis der Umsetzung anhand einer Konsolenanwendung präsentiert und eine Bewertung der Lösung gegeben.

Im abschließenden Kapitel 5 erfolgt die Zusammenfassung der relevanten Ergebnisse. Darüber hinaus wird in einem Ausblick auf Möglichkeiten der Verbesserung und Ansatzpunkte einer Erweiterung aufmerksam gemacht.

2 Theoretische Grundlagen

2.1 Graphentheorie

Die Grundlage für die Bearbeitung der Aufgabenstellung bilden das Wissen und die Erkenntnisse der Graphentheorie, die im Bereich der diskreten Mathematik angesiedelt ist [2]. Ihr Ursprung liegt in der Lösung des Königsberger Brückenproblems durch Leonhard Euler im 18. Jahrhundert, der mit Hilfe von Graphen nachweisen konnte, dass bei einem Rundgang durch Königsberg, nicht alle Brücken genau einmal überquert werden können [2–4]. Seitdem hat die Graphentheorie eine zentrale Rolle bei der Lösung verschiedenster Probleme eingenommen. So lassen sich beispielsweise kürzeste Wege in Navigationssystemen, Engpässe in Produktionsabläufen oder der Beziehungsgrad von Personen in sozialen Netzwerken mit ihrer Hilfe ermitteln. In der Informatik hat sie einen weiteren, wichtigen Anwendungsfall im Rahmen der Gestaltung und Bearbeitung von Datenstrukturen. Dabei besteht ein großer Vorteil der Graphentheorie und ihrer Konzepte darin, dass sie sich durch Visualisierung einfach und verständlich darstellen lassen.

2.1.1 Allgemeine Einführung in Graphen

Ausgangspunkt aller Betrachtungen zu graphenbasierten Algorithmen und Konzepten ist eine grundlegende Definition des Graphen und der für die Aufgabenstellung relevanten Eigen–schaften, um mit ihrer Hilfe genaue Spezifikationen für den zu bearbeitenden Typ von Graphen, ableiten zu können.

Definitionsgemäß besteht ein Graph G aus einer endlichen, nicht leeren Menge von Knoten V (engl. vertices, nodes) und einer Menge von Kanten E (engl. edges) und wird zusammenfassend mit $G = (V, E)$ beschrieben [4]. Dabei stellt jede Kante (u, v) eine Relation zwischen den Knoten $u, v \in V$ dar, die sie verbindet [3]. Die Kante kann je nach Anwendungsfall eine spezifische Bedeutung erhalten, wie beispielsweise ein Netzwerkkabel, das Rechnersysteme verknüpft, oder eine Straße, die Städte miteinander verbindet. Sie kann aber auch zusätzlich eine Bewertung bzw. Gewichtung erhalten [3, 5], wie zum Beispiel Entfernungswerte oder Kosten, welche bei der Bewegung entlang dieser Kante berücksichtigt werden müssen. Auch Knoten können je nach Kontext zusätzliche Informationen wie Bezeichnungen von Orten oder, wie im Rahmen dieser Arbeit, Anweisungen eines Computerprogramms tragen. Die typische Darstellungsform von Graphen ist durch Kreise oder Punkte, welche die Knoten repräsentieren, und Linie zwischen den Kreisen, welche die Kanten darstellen, gekennzeichnet.

Eine wesentliche Eigenschaft von Graphen ist die Richtung der Kanten. Diese ist ausschlag-gebend für die Möglichkeiten von Bewegungen innerhalb des Graphen, beispielsweise im Rahmen der strukturellen Erkundung. Sollten die Relationen symmetrisch gestaltet sein, also (1) gelten, so ist der Graph ungerichtet.

(mod. nach [6])
$$(u, v) \in E \Rightarrow (v, u) \in E \tag{1}$$

Das bedeutet, dass eine Bewegung in beide Richtungen entlang der Kante zwischen zwei Knoten erfolgen kann. Im Umkehrschluss bedeutet ein Fehlen dieser Symmetrie, dass der Graph

gerichtet ist. In diesem Fall ist eine Bewegung nur in der Richtung entlang der Kante möglich, die durch das geordnete Knotenpaar (u, v) mit Anfangsknoten u und Endknoten v vorgegeben wird. Die visuelle Darstellung eines gerichteten Graphen unterscheidet sich von der eines ungerichteten dadurch, dass die Kanten durch richtungsangebende Pfeile statt Linien repräsentiert werden. Die Richtung führt zudem noch zu zwei weiteren Ausprägungen von Kanten: den Mehrfachkanten (auch als parallele Kanten bezeichnet), bei denen mehrere Kanten die gleichen Knoten verbinden, und den Schlingen, bei denen Anfangs- und Endknoten gleich sind [2, 3]. Graphen, bei denen diese zwei zusätzlichen Ausprägungen nicht vorkommen, werden als schlichte Graphen bezeichnet [3, 7].

Die Nachbarschaft der Knoten, als weitere Eigenschaft, gibt Aufschluss über die Struktur des Graphen. Knoten, die durch eine Kante miteinander verbunden sind, werden als benachbart bzw. adjazent bezeichnet [3, 7]. Bei ungerichteten Graphen kann die Menge an Nachbarn eines Knoten u durch $N(u)$ angegeben werden. Bei gerichteten Graphen muss dabei N noch weiter in Nachfolger gemäß (2) und Vorgänger gemäß (3) unterteilt werden, da aufgrund der Richtung der Kanten die Relation zu den Nachbarn eindeutig zu bestimmen ist.

$$N^{+}(u) = \{u' \mid (u, u') \in E\} \quad (2)$$

$$N^{-}(u) = \{u' \mid (u', u) \in E\} \quad (3)$$

(mod. nach [7])

Aus der Nachbarschaft heraus wird auch der Grad g bestimmt, welcher für gerichtete Graphen mit $g(u) = |N(u)|$ angegeben und analog für ungerichtete Graphen aufgeteilt in Ausgangsgrad $g^{+}(u)$ und Eingangsgrad $g^{-}(u)$ ermittelt wird [7]. Mit Hilfe des Grades, lässt sich ein einfacher, numerischer Vergleich über die strukturelle Ähnlichkeit von Knoten durchführen.

Eine weitere Eigenschaft, die es zu betrachten gilt, ist die Zyklizität. Um den Begriff des Zyklus genau zu beschreiben, bedarf es einer zusätzlichen Erläuterung von relevanten Begriffen. In einem Graphen wird eine Folge von Knoten $u_0, u_1, \dots, u_n$, für die gilt, dass alle aufeinander folgenden Knoten adjazent sind, als Kantenfolge bezeichnet [3]. Eine Kantenfolge, welche aus paarweisen verschiedenen (disjunkten) Kanten besteht, wird als Weg bezeichnet. Sollten die Knoten zusätzlich noch disjunkt sein, wird von einem Pfad gesprochen. Sind in einem Pfad mit mindestens der Länge 3 zusätzlich sowohl alle Kanten disjunkt als auch Anfangs- und Endknoten identisch, wird dafür schließlich der Begriff Zyklus benutzt [3]. Zur Abgrenzung dazu sei zudem der Begriff Kreis, als ein Weg mit gleichem Anfangs- und Endknoten, definiert. Die Erkenntnis darüber, ob ein Graph Zyklen aufweist oder nicht, ist essenziell für die Auswahl geeigneter Algorithmen für die Graphenanalyse, da nicht jeder Algorithmus Zyklen behandeln kann und somit Gefahr läuft nicht terminieren zu können, weil er in eine Endlosschleife gerät.

Als letzte Eigenschaft sei an dieser Stelle der Zusammenhang genannt. Für ungerichtete Graphen gilt, dass bei Vorliegen einer Kantenfolge zwischen jedem verschiedenen Paar von Knoten, der Graph zusammenhängend ist [3]. Bei gerichteten Graphen muss, aufgrund der Einschränkungen durch die Richtung der Kanten, eine Unterteilung in schwachen und starken Zusammenhang vorgenommen werden. Es liegt schwacher Zusammenhang vor, wenn die Richtung der Kanten ignoriert wird und der Zusammenhang gemäß der Definition für ungerichtete Graphen festgestellt wird. Starker Zusammenhang liegt vor, wenn unter Beachtung der Richtung jedes beliebige Paar von Knoten durch eine Kantenfolge miteinander verbunden werden kann.

2.1.2 Graphenisomorphie

Um einen Vergleich von Graphen durchführen zu können, muss zunächst geklärt werden, unter welchen Bedingungen überhaupt Graphen gleich sind. Strukturelle Gleichheit von Graphen wird mit dem Begriff Isomorphie zum Ausdruck gebracht. Die Graphenisomorphie ist aufgrund ihrer großen Relevanz für viele Anwendungsbereiche, wie in der Biologie und Chemie, bei der Untersuchung molekularer Strukturen, oder in der Bild- und Textverarbeitung, beim Vergleich von auftretenden Mustern, ein viel untersuchtes und diskutiertes Thema der Mathematik und theoretischen Informatik.

Dabei werden zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ als isomorph bezeichnet, wenn es eine bijektive Abbildung f zwischen den Knotenmengen V_1 und V_2 gibt (Knotenbedingung) und zudem für alle Knoten $u, v \in V_1$ die Kantenbedingung (4) gilt [8, 9].

$$(u, v) \in E_1 \Leftrightarrow (\, f(u), f(v)\,) \in E_2 \tag{4}$$

(mod. nach [9])

Bei der Kantenbedingung wird zudem gefordert, dass auch nicht-existierende Kanten zwischen den Knoten in der Abbildung erhalten bleiben müssen. Sind zwei Graphen G_1, G_2 isomorph, so wird dies mit $G_1 \cong G_2$ zum Ausdruck gebracht und die Abbildung f als Isomorphismus zwischen G_1 und G_2 bezeichnet [8, 9]. Bei isomorphen Graphen sind zudem alle kennzeichnenden Eigenschaften identisch, wie beispielsweise Zyklizität, Richtung und Zusammenhang, da diese auf Beziehungen und Eigenschaften von Knoten und Kanten zurückgeführt werden können [9].

Die grundlegenden drei Eigenschaften der Isomorphie lassen sich gemäß (5) anhand der Graphen G_1, G_2, G_3 formulieren.

$$\begin{aligned}
&\textit{Reflexivität} && G_1 \cong G_1 \\
&\textit{Symmetrie} && G_1 \cong G_2 \longrightarrow G_2 \cong G_1 \\
&\textit{Transitivität} && G_1 \cong G_2 \; und \; G_2 \cong G_3 \longrightarrow G_1 \cong G_3
\end{aligned} \tag{5}$$

(nach [9])

Diese Eigenschaften können gut zur Prüfung genutzt werden, ob ein Algorithmus bestimmte Eigenschaften korrekt in Vergleichen von Graphen erkennt, indem beispielsweise ein Graph, welcher diese Eigenschaften trägt, zweimal als Eingabeparameter übergeben wird.

Des Weiteren muss betrachtet werden, wie aufwändig der Vergleich in Form einer Isomorphie-Prüfung durchzuführen ist. Für eine Abschätzung darüber wird der Begriff der Zeitkomplexität verwendet, welcher angibt, wie stark das Wachstum der Laufzeit eines Algorithmus mit steigender Größe der Eingabe, in unserem Fall der Größe der zu vergleichenden Graphen, zunimmt [3]. Dabei wird zur Beschreibung des Aufwands die O-Notation (auch Landau Notation) als Ordnung der Komplexität genutzt, die das asymptotische Verhalten der Laufzeit für diese Eingabemengen beschreibt [3].

Das Erkennen von Isomorphie stellt ein Problem dar, für welches es im allgemeinen Fall, bis heute noch keine Lösung gibt, die in polynomieller Zeit $O(n^k)$ mit $k \in \mathbb{N}$ berechenbar ist [8–10].

Die Brute-Force-Methode, welche die Ermittlungen von Isomorphie durch Ausprobieren aller Permutationen von bijektiven Abbildungen zwischen zwei Graphen beschreibt, weist faktorielle Zeitkomplexität $O(n!)$ auf und auch naive Algorithmen lösen das Problem nur in exponentieller Zeit $O(2^n)$. Die bisher besten, gefundenen Algorithmen zur Lösung des Problems liegen im Bereich zwischen polynomieller und exponentieller Zeitkomplexität, was als quasipolynomiell bezeichnet wird, angeführt von dem 2016 von László Babai veröffentlichten Algorithmus aus dem Bereich der Gruppentheorie, welcher bei $O(2^{\text{poly}(\log n)})$ liegt [10].

Im speziellen Fall gibt es jedoch Algorithmen, welche das Graphenisomorphie-Problem für bestimmte Arten von Graphen effizienter, sogar in polynomieller Zeit, lösen können. Beispiele für solche Algorithmen finden sich für planare Graphen, Bäume (zusammenhängende Graphen ohne Kreis) oder Graphen mit beschränkten Eigenschaften (wie Grad, Baumweite) [11].

In der Praxis haben sich mit „Nauty“ (No AUTomorphism, Yes?), „Bliss“ (Bipartite Labeled Induced Subgraph Symmetry) und „Traces“ Programme etabliert, welche auf Basis der Symme–trieanalyse von Automorphismen (Abbildungen eines Graphen auf sich selbst) und zusätzlicher Heuristiken effizient Isomorphismen erkennen können [11]. Im Worst-Case-Fall ist ihre Laufzeit aber immer noch im exponentiellen Bereich.

Aufgrund der Tatsache, dass die Ermittlung eines Isomorphismus zweier Graphen im Allge–meinen für große Graphen praktisch nicht umsetzbar ist und im Spezialfall Einschränkungen hinsichtlich der Art und Größe der verwendeten Graphen mit sich bringt, kann auch in Erwägung gezogen werden das Fehlen von Isomorphie nachzuweisen, anstatt einen Isomorphismus zu suchen, indem nach Eigenschaften gesucht wird, welche in den zu vergleichenden Graphen unterschiedlich sind [9]. Das erfordert allerdings eine umfangreiche Analyse und Bearbeitung der Graphen und hat damit auch einen großen Einfluss auf die Laufzeit des Algorithmus. Außerdem müssen die betrachteten Graphen ausreichend bekannt sein, um abschätzen zu können, bei welchen Eigenschaften Unterschiede zu erwarten sind, um diese in die Prüfung durch den Algorithmus mit aufzunehmen.

Abschließend muss bei Betrachtungen über die Eigenschaften auch darauf geachtet werden, dass bei Vorliegen von Gleichheit der Eigenschaften, nicht automatisch auf das Vorliegen von Isomorphie geschlossen werden kann. Es handelt sich in diesem Fall zwar um eine notwendige, nicht aber um eine hinreichende Bedingung. Diese wird nur erfüllt, wenn eine vollständige Untersuchung der Graphenstruktur durchgeführt wird.

2.1.3 Traversierung

Im Rahmen der Untersuchung von Graphen wird das schrittweise Erkunden adjazenter Knoten entlang ihrer Kanten als Traversierung bezeichnet. Mit ihr verbunden sind Zielsetzungen, die Gesamtstruktur des Graphen zu erfassen, ihn auf das Vorhandensein bestimmter Eigenschaften, wie Zyklizität oder Zusammenhang, zu prüfen oder Informationen zu suchen, welche in den Knoten gespeichert sind. Meist bilden daher Konzepte zur Traversierung die Basis für kom–plexere Algorithmen der Graphenanalyse.

Die beiden grundlegendsten Algorithmen zur Traversierung von Graphen sind die Tiefensuche (engl. Depth-First-Search, DFS) und die Breitensuche (engl. Breadth-First Search, BFS). Beide dienen der Erkundung der Graphenstruktur, haben dabei aber unterschiedliche Vorgehens–weisen, was sie für unterschiedliche Anwendungsbereiche prädestiniert.

Die Tiefensuche erhält ihren Namen daher, dass sie ausgehend von einem Startknoten auf einem Pfad so weit wie möglich in tiefere Ebenen eines Graphen vordringt, indem in jedem Schritt ein Nachfolger des aktuell besuchten Knotens besucht wird, bis der Pfad bei einem Knoten endet, der keinen Nachfolger hat. Dann erfolgt ein Rückschritt in die höheren Ebenen zu Knoten, welche noch unbesuchte Nachfolger haben und die Erkundung wird von dort aus analog fortgesetzt. Dieser Rückschritt wird als Backtracking bezeichnet [5]. Abbildung 1 verdeutlicht diesen Ablauf durch die Traversierung entlang der gleich gefärbten Knoten.

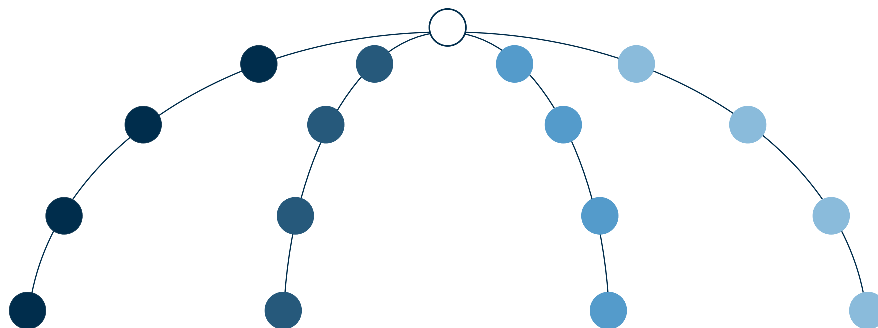


Abbildung 1: Reihenfolge der Traversierung bei Tiefensuche

Der Algorithmus findet sein Ende, sobald der letzte Knoten exploriert worden ist und kein weiterer Nachfolger unbesucht im Graphen verbleibt. Die Besuchsreihenfolge wird dabei von einer Datenstruktur, die als Stack (dt. Stapel) bezeichnet wird, gesteuert. Auf diesen werden die unbesuchten Knoten gelegt und nach dem Last-In-First-Out-Prinzip (LIFO) für die Bestimmung des nächsten zu besuchende Knoten entnommen [3, 4]. Der Algorithmus kann auch rekursiv gestaltet werden. In diesem Fall wird kein expliziter Stack benötigt, weil dieser implizit durch den Aufrufstack der Rekursion bereitgestellt wird [4]. Die Zeitkomplexität der Tiefensuche ist dabei linear in Bezug auf Anzahl an Knoten V und Kanten E und beträgt $O(|V| + |E|)$ [9]. Typische Anwendungsbereiche der Tiefensuche sind Wegfindungsprobleme, wie die Suche in Labyrinthen oder der Ermittlung von Routen in Navigationssystemen.

Bei der Breitensuche wird im Gegensatz dazu die Erkundung des Graphen von Ebene zu Ebene in der Breite durchgeführt. Dabei werden ausgehend von einem Startknoten zuerst dessen direkte Nachfolger besucht, bevor im Folgenden deren Nachfolger besucht werden und so weiter. In Abbildung 2 wird der Ablauf wiederum durch gleiche Färbung verdeutlicht.

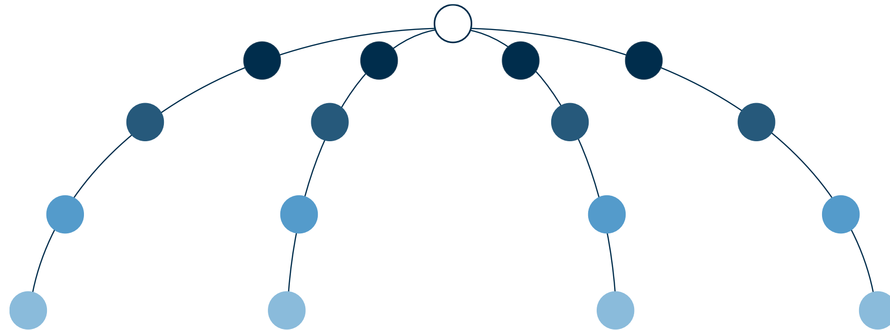


Abbildung 2: Reihenfolge der Traversierung bei Breitensuche

Die Breitensuche terminiert, sobald auf unterster Ebene der letzte Nachfolger besucht wurde. Im Vergleich zur Tiefensuche liegt der Breitensuche eine Datenstruktur, die als Queue (dt. Warteschlange) bezeichnet wird, für die Verwaltung der Besuchsreihenfolge zugrunde. Die Queue wird wiederum mit den unbesuchten Knoten befüllt, aber nach dem First-In-First-Out-Prinzip (FIFO) für die Wahl des nächsten Knotens abgearbeitet [3]. Auch in der Breitensuche liegt dabei eine Zeitkomplexität von $O(|V| + |E|)$ vor [9]. Breitensuche wird typischerweise bei der Abstandsberechnung von Knoten oder beim Test auf Zyklen im Graph eingesetzt [4].

Zu beiden Algorithmen muss angemerkt werden, dass sie in der beschriebenen Form nur für azyklischen Graphen durchführbar sind. Sollten zyklische Graphen traversiert werden, muss zusätzlich noch für jeden Knoten oder jede Kante eine Markierung erfolgen, wenn ein Besuch stattgefunden hat, und bei der Auswahl des nächsten Knotens darauf geprüft werden. Im Fall, dass eine Markierung vorliegt, darf der Knoten oder die Kante nicht erneut besucht werden, um Endlosschleifen zu vermeiden.

Zusätzlich ist zu beachten, dass die Wahl darüber, welcher der Nachfolger zuerst auf den Stack bei Tiefensuche oder in die Queue bei Breitensuche gespeichert wird, von der Reihenfolge, in der die Nachfolger im Graphen gespeichert werden, bestimmt wird [5]. Das kann Auswirkungen auf die Ergebnisse einer Graphenanalyse haben, wie beispielsweise der Analyse gefundener Pfade, indem ein Pfad zu einem Knoten a von zwei Vorgängern b, c gebildet werden kann und die Reihenfolge der Traversierung von b und c darüber bestimmt, ob Kante (b, a) oder (c, a) in den gefundenen Pfad aufgenommen wird.

2.1.4 Maximum Common Subgraph

Neben der Untersuchung der gesamten Graphen, besteht auch die Möglichkeit, Strukturen innerhalb der Graphen zu vergleichen. Dies erfolgt auf Basis von Teilgraphen (engl. subgraphs) oder induzierten Teilgraphen (engl. induced subgraphs). Ein Graph $G' = (V', E')$ wird dabei als Teilgraph eines Graphen $G = (V, E)$ bezeichnet, wenn $E' \subseteq E$ und $V' \subseteq V$ gilt [9]. Wird diese Definition noch um die Bedingung (6) erweitert, so spricht man von einem induzierten Teilgraphen oder auch Untergraphen.

$$E' = \{(u, v) \in E | u, v \in V'\} \tag{6}$$

(mod. nach [9])

Mit Hilfe dieser lässt sich nun ein Vergleich zwischen zwei Graphen G_1, G_2 aufbauen. Ist ein Graph T isomorph zu den Teilgraphen von G_1 und G_2, so wird T gemeinsamer Teilgraph von G_1 und G_2 genannt. Gibt es keinen weiteren Teilgraphen, der mehr Knoten als T beinhaltet, so wird dieser zusätzlich als Maximum Common Subgraph (dt. maximaler gemeinsamer Teilgraph, kurz MCS) bezeichnet [12].

Ein Ansatz zur rekursiven Erstellung des MCS wird abstrahiert in Abbildung 3 dargestellt. Dabei beschreibt s einen Teilgraphen, welcher als Zwischenstand entsteht und bei Abschuss des Algorithmus, welcher als Zustandsraum-Suchalgorithmus bezeichnet wird, den gefundenen MCS bildet [12]. Es gilt zudem: $n1 \in V(G_1)$ und $n2 \in V(G_2)$.

```
Procedure MCS(s,n1,n2)
Begin
   if (NextPair(n1,n2)) then
      begin
         if (IsFeasiblePair(n1,n2)) then
           AddPair(n1,n2);
         CloneState(s,s');
         while(s' is not a leaf of the search tree)
         begin
            MCS(s',n1,n2);
            BackTrack(s');
         End
         Delete(s');
      End
End procedure
```

Abbildung 3: Pseudocode eines Zustandsraum-Suchalgorithmus für die MCS Suche (nach [12])

Der rekursive Algorithmus startet mit der Auswahl des nächsten zu untersuchenden Paares mit Aufruf von *NextPair.* Im Anschluss wird mit *IsFeasiblePair* geprüft, ob das Paar gemäß zu definierender Eigenschaften bzw. Kriterien in den MCS aufgenommen werden kann. Bei positivem Ergebnis wird das Paar in s durch *AddPair* aufgenommen und der Zwischenstand durch *CloneState* kopiert, um diesen unabhängig vom weiteren Verlauf zu halten und beim Backtracking den Zustand wiederherstellen zu können. Im Anschluss erfolgt der Rekursions–aufruf *MCS* mit dem Klon, vorausgesetzt s ist kein Endzustand auf dem derzeitigen Suchpfad. Nach erfolgter Rekursion wird mittels *BackTrack* zum vorhergehenden Zustand zurückgekehrt

und nach alternativen Pfaden gesucht. Abschließend wird durch *Delete* der Klon wieder gelöscht. Der Algorithmus endet, wenn er alle möglichen Pfade durchlaufen hat.

Der MCS wird nach seiner erfolgreicher Erstellung auf gewünschte Eigenschaften und Muster hinsichtlich des Vergleichs der Ausgangsgraphen untersucht. Anwendungsbereiche des MCS sind interdisziplinär und finden sich beispielsweise in der Chemoinformatik, bei der Ausrichtung von Molekülen, bei der Malware-Erkennung und der Mustererkennung [13].

Darüber hinaus wird er in Forschung und Praxis anhand spezifischer Eigenschaften, die für ihn gelten sollen, in weitere Varianten unterteilt. Dabei wird bei der Erstellung des Maximum Common Connected Subgraphs (MCCS) die Eigenschaft des Zusammenhangs von Knoten innerhalb des Teilgraphs, beim Maximum Common Induced Subgraph (MCIS) das Vorliegen eines induzierten Teilgraphs und beim Maximum Common Connected Induced Subgraph (MCCIS) eine Kombination beider Eigenschaften gefordert [14]. Diese Varianten haben gemeinsam, dass ein starker Fokus auf die Eigenschaften der Knoten, wie etwa Typ, Färbung oder Beschriftung, gelegt wird, um den MCS zu prägen. Darüber hinaus gibt es aber noch Varianten, bei denen die Struktur des MCS vorwiegend durch die Eigenschaften von Kanten bestimmt wird. Zu diesen Eigenschaften gehören beispielsweise die Gewichtung oder Topologie der Kanten. Jede dieser Varianten hat für sie spezifische Anwendungsbereiche und eine Vielzahl an spezialisierten Algorithmen.

2.2 Metriken zur Ähnlichkeitsbestimmung

In diesem Kapitel wird eine begrenzte Auswahl an Ansätzen zur Bestimmung einer quanti–fizierbaren Ähnlichkeit von Graphen vorgestellt, die unter den Begriff der Metriken in allge–meiner Bedeutung fallen. Während das Entscheidungsproblem, ob zwei Graphen isomorph zueinander sind, nur mit Ja oder Nein beantwortet werden kann, wird im verneinten Fall von der Aufgabenstellung gefordert eine differenzierte Aussage darüber zu treffen, wie unähnlich die beiden Graphen sich sind, um daraus eine Möglichkeit der Bewertung ableiten zu können. Die im Folgenden vorgestellten Ansätze entspringen verschiedenen Disziplinen der Mathematik und Informatik, werden aber interdisziplinär in zahlreichen, graphentheoretischen Algorithmen verwendet.

2.2.1 Levenshtein-Distanz

Die Levenshtein-Distanz (auch Edit-Distanz), benannt nach dem russischen Mathematiker Vladimir Levenshtein, stammt aus dem Bereich der theoretischen Informatik und basiert auf Berechnungen der Ähnlichkeit von Zeichenketten (engl. strings) [4]. Der zugrundeliegende Algorithmus berechnet dabei den minimalen Aufwand an Operationen, welcher notwendig ist, um eine Zeichenkette X in eine Zeichenkette Y zu transformieren. Zu den Operationen zählen dabei das Einfügen (E), das Löschen (L) und das Ersetzen (R) von Zeichen [4]. Diesen kann bei Bedarf jeweils noch ein Gewichtung w zugeordnet werden, um die spezifischen Kosten der Durchführung widerzuspiegeln [3]. Wird die Levenshtein-Distanz $dist$ berechnet, so ergibt sich (7) als minimaler Aufwand, unter Berücksichtigung der Anzahl a der durchgeführten Operationen.

$$dist(X, Y) = \min\{w_E * a_E, w_L * a_L, w_R * a_R\} \tag{7}$$

(mod. nach [3])

Die Berechnung der Levenshtein-Distanz basiert häufig auf dem Prinzip der dynamischen Programmierung, bei dem die schrittweise Lösung von komplexeren Problemen durch die Kombination von Lösungen von Teilproblemen herbeigeführt wird [4]. Dabei wird zur Speicherung von Zwischenergebnissen eine Distanzmatrix $D(n+1, m+1)$ mit $n = |X|$ und $m = |Y|$ verwendet, welche vor Beginn des eigentlichen Algorithmus mit den Basisfällen der Levenshtein-Distanz gemäß (8) in der ersten Spalte und Zeile initialisiert wird.

$D(0,0) = 0$ → $dist$ zwischen zwei leeren Zeichenketten ist 0

$D(i, 0) = i * w_L$ → $dist$ für die Löschung der ersten i Zeichen von X zur leeren Zeichenkette für $1 \leq i \leq n$

$D(0, j) = j * w_E$ → $dist$ für das Einfügen der ersten j Zeichen von Y in die leere Zeichenkette X für $1 \leq j \leq m$ (8)

(mod. Bezug zu *D* statt *dist,* nach [3])

Nach erfolgter Initialisierung der Distanzmatrix kann ausgehend von $D(1,1)$ mit dem Algorith–mus zeilenweise von links nach rechts fortgesetzt werden. Es erfolgt dabei eintragsweise eine Ermittlung der minimalen Distanz zwischen den Zeichenketten $X(i..n)$ und $Y(j..m)$, indem

ausgehend von der aktuellen Position $D(i,j)$ der Minimalwert gemäß (9) an dieser Position gespeichert wird. Dabei bezeichnen x_i und y_j die Zeichen an der i-ten Stelle von X und j-ten Stelle von Y.

$$D(i,j) = min\begin{cases} D(i-1,j) + w_L \\ D(i,j-1) + w_E \\ D(i-1,j-1) + w_R(i,j) \end{cases} \qquad w_R(i,j) = \begin{cases} 0 & f\ddot{u}r\ x_i = y_j \\ w_R & f\ddot{u}r\ x_i \neq y_j \end{cases} \tag{9}$$

(mod. Bezug zu *D* statt *dist,* nach [3])

Nach Befüllung der Distanzmatrix terminiert der Algorithmus und die Levenshtein-Distanz $dist(X,Y)$ kann dem Eintrag in $D(n+1,m+1)$ entnommen werden.

Sollte es erforderlich sein, die Operationen, die zum Ergebnis geführt haben, nachvollziehbar machen zu müssen, so besteht eine Möglichkeit darin per Rekursion rückwärts durch die Distanzmatrix zu gehen. Dabei wird ausgehend von Eintrag $D(n+1,m+1)$ mittels Abgleich der benachbarten Einträge $D(i-1,j)$, $D(i,j-1)$ und $D(i-1,j-1)$ anhand der minimalen Werte ermittelt, welche Operation für den Einzelschritt durchgeführt wurde. Dieser Abgleich wird wiederholt, bis der Anfang der Matrix erreicht wird. Eine andere Möglichkeit besteht darin, parallel zur Befüllung der Distanzmatrix eine zusätzliche Matrix mit der in jedem Schritt ausgewählten Operation zu führen. Diese wird anschließend direkt rückwärts durchlaufen, um die durchgeführten Operationen zu ermitteln, wie ausführlich in [4] beschrieben.

Die Levenshtein-Distanz wird im Normalfall durch eine rekursive Funktion mit exponentieller Zeitkomplexität berechnet [3]. Die vorgestellte Methode der Berechnung der Levenshtein-Distanz mittels dynamischer Programmierung macht es möglich, durch die Reduzierung des Problems auf eine Matrixbefüllung, die Zeitkomplexität auf $O(n*m)$ zu reduzieren [4].

Anwendung findet die Levenshtein-Distanz in der Textverarbeitung, beispielsweise bei der Mustererkennung oder Fehlerkorrektur, sowie in der Biologie, im Rahmen von durchgeführten DNA-Sequenzvergleichen [4].

2.2.2 Graph Edit Distance

Die Graph Edit Distance (dt. Graph-Editier-Distanz) GED ist eine Verallgemeinerung der bereits besprochenen Levenshtein-Distanz. Sie misst die Ähnlichkeit zweier Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ durch die Berechnung der minimalen Kosten an Editieroperationen, welche notwendig sind, um G_1 in G_2 zu transformieren [15, 16]. Mögliche Editieroperationen op können gemäß (10) auftreten.

$$\begin{aligned} (v \to \varepsilon) \text{ oder } (e \to \varepsilon) \quad &\to \quad \text{Löschen eines Knotens } v \in V \text{ oder einer Kante } e \in E \\ (\varepsilon \to v) \text{ oder } (\varepsilon \to e) \quad &\to \quad \text{Einfügen eines Knotens oder einer Kante} \\ (u \to v) \quad &\to \quad \text{Ersetzen eines Knotens } u \in V \end{aligned} \tag{10}$$

(mod. nach [16])

Die Reihenfolge, in der die Operationen zur Umwandlung durchgeführt werden, wird als Trans–formationsfolge (engl. edit path) mit $D = (op_1, \ldots, op_k)$ bezeichnet und ergibt nach Abschluss den Resultatgraph $D(G_1)$, für den gilt: $D(G_1) \cong G_2$ [17]. Jede angewandte Operation ver–ursacht Kosten, deren Summe die Transformationskosten $c(D)$ ergibt. Da in der Regel mehrere Transformationsfolgen mit unterschiedlichen Kosten vorhanden sind, ist die Bestimmung der GED mit der Suche nach der Folge mit den minimalen Kosten verbunden, sodass (11) gilt.

$$GED(G_1, G_2) = min\,\{c(D) | D \in \mathcal{D}(G_1, G_2)\} \tag{11}$$

(mod. nach [17])

Die Algorithmen, welche für die Berechnung der Graph Edit Distance eingesetzt werden, lassen sich unterschiedlich klassifizieren. Es wird beispielsweise zwischen Algorithmen zur exakten Distanzberechnung, die häufig aufgrund von Matching-Methoden exponentielle Zeitkomplexität aufweisen, und Algorithmen zur approximierten Distanzberechnung, die durch Restriktionen und Heuristiken teilweise in polynomieller Zeit berechnet werden, aber nur Näherungslösungen liefern, unterschieden [15]. Ein häufig eingesetzter Algorithmus zur Bestimmung der exakten Distanz ist beispielsweise der A*GED- Algorithmus, welcher iterativ die kostenminimale Trans–formationsfolge auf Basis der Gesamtkosten, bestehend aus den aktuellen Tranformations–kosten und einer Heuristik, welche die noch anfallenden Kosten für die optimale Lösung nie unterschätzt, ermittelt [16]. Ein Beispiel für approximative Algorithmen ist die Berechnung der Graph Edit Distance für bipartite Graphen [18].

Des Weiteren kann eine Kategorisierung in fehlertolerante und fehlerintolerante Algorithmen vorgenommen werden. Diese Unterscheidung ist wichtig, weil es in Anwendungen, wie beispielsweise bei der Erkennung von Fingerabdrücken, zu Inkonsistenzen und Rauschen in den Daten kommen kann [19]. Ein fehlertoleranter Algorithmus ist unter diesen Umständen in der Lage, trotzdem einen Abgleich durchzuführen und eine sinnvolle Graph Edit Distance zu berechnen.

Abschließend erfolgt zudem oftmals eine Unterscheidung in Algorithmen, die für Graphen mit attribuierten Knoten oder Kanten konzipiert sind und die GED aus dem Vergleich der Attribute

ableiten, und solche, welche für Graphen ohne Attribute gelten und basierend auf reinem Strukturabgleich die Graph Edit Distance berechnen [20]. Für Graphen mit attribuierten Knoten und Kanten kommen beispielsweise Ansätze zum Einsatz, die auf Wahrscheinlichkeiten oder der Analyse von Subgraphen und Supergraphen basieren [20]. Graphen mit Knoten und Kanten ohne Attribute werden hingegen häufig in eine Repräsentation als Zeichenkette überführt und anschließend mit Algorithmen zur Berechnung der Edit-Distanz, zum Beispiel auf Basis des Dijkstra Algorithmus, anstelle der Graph Edit Distance bearbeitet [20].

Es gibt zahlreiche Anwendungsbereiche der Graph Edit Distance. So wird sie in verschiedenen Bereichen der Text- und Schriftverarbeitung eingesetzt, beispielsweise bei der Erkennung von Schriftzeichen (engl. Optical Character Recognition) [17]. Auch in der Bild- und Mustererkennung zur Identifikation von Personen und Objekten oder bei der Validierung und Konsistenzprüfung innerhalb von Graphendatenbanken kommt die GED zum Einsatz [15]. Darüber hinaus wird sie in Geoinformationssystemen zur Speicherung von Daten und bei der Analyse von Straßennetz–werken eingesetzt [15].

Trotz der verbreiteten Anwendung der GED aufgrund ihrer flexiblen Einsetzbarkeit und tief–greifenden Analysemöglichkeiten bei der Optimierung der zugrunde liegenden Algorithmen, gibt es noch verschiedene Aspekte, die in der Forschung weiter untersucht werden müssen. Dazu gehören beispielsweise die richtige Wahl der zu vergleichenden Attribute von Knoten und Kanten, die Entwicklung von allgemeinen Kostenfunktionen für die Editieroperationen und eine bessere Anpassung der Suchstrategien an die Vergleichsmethoden [20].

2.2.3 Struktur abstrakter Syntaxbäume

Abstrakte Syntaxbäume (engl. abstract syntax trees, AST) sind nicht nur ein wichtiger Bestandteil des Compilerbaus und Kompilierungsprozesses, sondern spielen auch bei der Analyse und Optimierung von Programmen in der Softwareentwicklung eine bedeutende Rolle. Im Rahmen dieses Ansatzes wird mittels abstrakter Syntaxbäume ein struktureller Vergleich zweier Anwei–sungen angestrebt. Dabei stellen AST die syntaktischen Informationen eines Programms in einer hierarchischen, übersichtlichen und leicht zugänglichen Baumstruktur, einem zusammen–hängenden, kreisfreien Graphen mit ausgewiesenem Startknoten (Wurzel), dar.

Die Kompilierung eines Programms beginnt mit der lexikalischen Analyse, bei der aus dem Quellcodetext mit Hilfe von Scannern einzelne Wörter, sogenannte Token, erstellt werden, die die kleinsten Informationseinheiten eines Programms darstellen [21, 22]. Zu diesen gehören beispielsweise Schlüsselwörter, Literale, Bezeichner und Operatoren. In der sich anschließenden syntaktischen Analyse, erfolgt dann mittels eines Parsers die Überprüfung, ob diese Token ein korrektes Programm im Sinne der formalen Grammatik der zugrunde liegenden Programmier–sprache darstellen [21, 22].

Die Syntax von Programmiersprachen wird in der Regel mithilfe kontextfreier Grammatiken definiert [23]. Eine kontextfreie Grammatik ist dabei ein Quadrupel (N, T, P, S), bestehend aus den endlichen Mengen von Nichtterminalen N, die als Platzhalter dienen, von Terminalen T, die die Token darstellen, wobei gilt: $N \cap T = \emptyset$, von Produktionsregeln P, die die Form $X \rightarrow a$ haben mit $X \in N, a \in (N \cup T)^*$ und dem Startsymbol $S \in N$ [23].

Bei der Überprüfung wird mit Hilfe der Produktionsregeln das Programm ausgehend von S in endlich vielen Schritten von Nichtterminalen zu Terminalen abgeleitet. Diese Ableitung erfolgt in Form einer Baumkonstruktion, bei der S den Wurzelknoten bildet und die einzelnen, verwendeten Produktionsregeln als neue Knoten hinzugefügt werden, wobei N den Vater–knoten (linke Seite des Pfeils) darstellt und a die Nachfolger (rechte Seite des Pfeils) sind. Am Ende der Konstruktion bilden die Terminale die Blätter (Knoten ohne Nachfolger) des entstandenen Baums, der als Ableitungsbaum (engl. parse tree, syntax tree) bezeichnet wird [21, 23]. Nach erfolgter Ableitung wird durch rückwärtiges Durchlaufen des Ableitungsbaums, von jedem Blatt zum Startknoten, vom Parser geprüft, ob dieser erreichbar ist. Im positiven Fall wird festgestellt, dass das Programm korrekt gemäß der Grammatik erstellt wurde.

Schließlich ergibt sich der abstrakte Syntaxbaum aus dem Ableitungsbaum, indem dieser um die Darstellung der Nichtterminale bereinigt und damit kompakter gemacht wird [21]. Dies stellt einen Vorbereitungsschritt für die folgende semantische Analyse dar.

Neben ihrer großen praktischen Relevanz im Compilerbau, besteht auch in der aktuellen Forschung ein Interesse an abstrakten Syntaxbäumen. So gibt es beispielsweise Artikel über die Nutzung von abstrakten Syntaxbäumen zur Klassifizierung von Programmen, unabhängig ihrer zugrunde liegenden Programmiersprache [24], zur Erkennung von Programmcode-Plagiaten [25], sowie zur Bewertung der Korrektheit zwischen zwei Programmen im Rahmen der Lehre, ähnlich wie in dieser Arbeit [22].

2.2.4 Jaccard-Index

Der Begriff des Jaccard-Index, auch als Jaccard-Ähnlichkeit oder Jaccard-Koeffizient bezeichnet, beschreibt in der Mengentheorie ein Maß für die Ähnlichkeit zweier Mengen X und Y [26]. Diese wird dadurch ermittelt, dass die Anzahl der gemeinsamen Elemente (Schnittmenge) durch die Anzahl aller Elemente in beiden Mengen (Vereinigung) geteilt wird, wie in Formel (12) gezeigt [26, 27].

Jaccard-Index
$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \qquad (12)$$

(mod. nach [27])

Das Ergebnis dieser Berechnung liegt dabei im Bereich $0 \leq J(X,Y) \leq 1$, wobei mit $J(X,Y) = 1$ die vollkommene Gleichheit und $J(X,Y) = 0$ die vollkommene Verschiedenheit zum Ausdruck gebracht wird [27]. Dieser Bereich kann sowohl nach oben, wenn durch große Unterschiede in der Anzahl der Elemente in beiden Mengen, zum Beispiel für $|X| < |Y|$, nur noch maximal gilt: $X \subseteq Y$, als auch nach unten, wenn beispielsweise der gewichtete Jaccard-Index genutzt wird, noch weiter beschränkt sein [27].

Der gewichtete Jaccard-Index wird als Erweiterung des einfachen Jaccard-Index angesehen, bei der jedem Element $x_k \in X$ und $y_k \in Y$ eine Gewichtung k zugewiesen wird und sich der Index gemäß (13) berechnet [28].

Gewichteter Jaccard-Index
$$J(X,Y) = \frac{\sum_k min\,(x_k, y_k)}{\sum_k max\,(x_k, y_k)} \qquad (13)$$

(mod. nach [28])

Der gewichtete Index respektiert dabei nicht nur das Auftreten der Elemente in beiden Mengen, sondern ermöglicht es auch differenzierter die Ähnlichkeit anhand der Bedeutung oder Häufigkeit des Vorkommens der Elementen zu bestimmen.

Häufig wird bei Untersuchungen zum Nachweis von Verschiedenheit zweier Mengen auch die Jaccard-Distanz verwendet, das Komplement des Jaccard-Index, dessen Berechnung gemäß (14) erfolgt.

Jaccard-Distanz
$$d_j(X,Y) = 1 - J(X,Y) \qquad (14)$$

(mod. nach [27])

Anwendung finden der Jaccard-Index und die Jaccard-Distanz beispielsweise im Bereich der Cybersecurity, bei der Klassifizierung von Schadsoftware [29], im Rahmen der Umgebungs–überwachung bei der Wiedererkennung von Personen [30] oder bei der Texterkennung im Rahmen der Recherche wissenschaftlicher Artikel [31].

2.3 Kontrollflussgraphen

Kontrollflussgraphen (engl. control flow graphs, CFG) sind ein wichtiges Werkzeug bei der Analyse und Optimierung von Programmen. Sie sind gerichtete Graphen, die zur Visualisierung der Programmsteuerung dienen und sich formal als 6-Tupel mit $G = (V, E, s, e, L, l)$ beschreiben lassen [32]. Dabei bezeichnet:

- V die Menge der als Basisblöcke bezeichneten Knoten,
- E die Menge der Kanten, welche die möglichen Kontrollflüsse von einem Block zum nächsten darstellt,
- s und e den Startknoten (Quelle) und Endknoten (Senke) des Programms,
- L die Menge an Labeln, welche die Bedingungen für das Traversieren einer Kante bei Verzweigungen beinhaltet
- und der Funktion $l: L \to E$, welche den Kanten die Bedingungen zuordnet [32].

Bei Basisblöcken handelt es sich in diesem Zusammenhang um eine oder mehrere Anweisungen des Programms, welche strikt sequenziell und ohne Unterbrechung ausgeführt werden [23]. Die Knoten s und e stellen dabei eigenständige Blöcke dar, auch wenn sie keine Anweisungen enthalten [23]. Die Anweisungen innerhalb eines Blocks werden häufig in eine standardisierte Form gemäß (15) gebracht, um Optimierungen im Rahmen von Kompilierungen zu vereinfachen. Die Überführung wird dabei durch Aufteilung komplexerer Anweisungen und Speicherung von Zwischenergebnissen in Hilfsvariablen realisiert [23]. Anhand von Bezeichnern, die Anweisungen zugewiesen werden, wird bei Sprüngen das Ziel angegeben.

$$
\begin{aligned}
&\text{Zuweisung} && Var \coloneqq Operand \mid Operand1\ Operator\ Operand2 \\
&\text{bedingter Sprung} && if(Operand1\ Operator\ Operand2)\ GOTO\ Bezeichner \\
&\text{unbedingter Sprung} && GOTO\ Bezeichner
\end{aligned}
\tag{15}
$$

(mod. nach [23])

Die Zuordnungsfunktion l bestimmt das Verhalten beim Traversieren durch den Kontrollfluss–graphen, indem sie sicherstellt, dass bei auftretenden Verzweigungen der korrekte weitere Kontrollfluss, anhand der Auswertung und Zuweisung der Bedingungen an die entspreche Kante, genommen wird.

Kontrollflussgraphen lassen sich in zwei Typen klassifizieren, intraprozedurale Kontrollfluss–graphen und interprozedurale Kontrollflussgraphen, wobei die Unterscheidung anhand der Grenzen des Kontrollflussgraphen stattfindet [32]. Ein intraprozeduraler Kontrollflussgraph beschreibt den Kontrollfluss innerhalb einer einzelnen Funktion oder Prozedur. Dabei bildet der Einstiegspunkt den Funktionsaufruf und der Ausstiegspunkt das Return oder den Funktions–abschluss, sodass der Graph nur den Kontrollfluss innerhalb der Funktion selbst abbildet. Ein interprozeduraler Kontrollflussgraph dagegen umfasst mehr als nur eine Funktion oder Prozedur, die meist als separate intraprozedurale Kontrollflussgraphen aufgefasst werden, aber bei denen es in der Literatur unterschiedliche Definitionen gibt, ob diese Kontrollflussgraphen miteinander verbunden sind und wie gegenseitige Funktionsaufrufe zu modellieren sind [32].

Die Erstellung eines Kontrollflussgraphen lässt sich auf verschiedenen Wegen durchführen. Der gängigste Weg ist im Zuge der Kompilierung durch die im Compiler integrierten Parser. Des Weiteren besteht die Möglichkeit Kontrollflussgraphen über spezialisierte Software-Tools, wie CodeSonar, einem Tool zur Erkennung von Sicherheitslücken, oder Ghidra, einem Reverse-Engineering-Framework, zu erhalten [33].

Kontrollflussgraphen besitzen aufgrund ihrer universellen Einsetzbarkeit bei der Visualisierung und Analyse von Programmcode ein großes Spektrum an Einsatzmöglichkeiten. So finden sie Anwendung im Compilerbau zur Optimierung von Code, bei der statischen Programmanalyse zur Fehlererkennung und bei der Datenflussanalyse im Rahmen der Abhängigkeitsuntersuchung von Variablen.

In der Forschung lassen sich zahlreiche Studien im Zusammenhang mit Kontrollflussgraphen finden. Beispiele hierfür sind eine Studie zur Erkennung von Schadsoftware mithilfe von Machine-Learning-Algorithmen, die auf Kontrollflussgraphen basieren [34], sowie die Entwick–lung eines Parsers für veraltete Programmcode-Kontrollflussgraphen, die für den Korrektheits–nachweis bei der Code-Modernisierung dienen [33].

2.4 Programmablaufplan

Die Anfänge der Programmablaufpläne (PAP) reichen in die 1960er und 70er Jahre zurück, als die zunehmende Komplexität entwickelter Programme dadurch zum Problem wurde, dass aufgrund mangelnder Überschaubarkeit des Programmcodes eine effiziente Durchführung von Wartung und Weiterentwicklung erschwert wurde [1]. Zur Lösung des Problems entwickelten führende Informatiker und Mathematiker, unter anderem Edsger Dijkstra, das Konzept der strukturierten Programmierung, in welcher Programme grundsätzlich in den Kontrollstrukturen Sequenz, Selektion und Iteration organisiert werden müssen [1]. Zur Unterstützung bei der Entwicklung und Dokumentation strukturierter Programme etablierte sich dabei der Programmablaufplan, ein grafisches Hilfsmittel, das mit seinen Gestaltungselementen die Kontrollstrukturen visuell aufbereitet und so Abläufe in der Programmlogik nachvollziehbar werden lässt.

Die Standardisierung des Programmablaufplans, in welcher die Struktur und die verwendeten Symbole definierten sind, erfolgte offiziell mit der DIN 66001 im Jahr 1977 und der ISO 5807 zwei Jahre später [35]. Gemäß der DIN 66001 sind in einem PAP keine Symbole für Daten vorgesehen, da die Norm eine strikte Trennung von diesen und den mit einem PAP darzu–stellenden „Verarbeitungsfolgen in einem Programm“ vorsieht [35]. In der Praxis wird aus Gründen der Verständlichkeit und Vollständigkeit trotzdem häufig ein Symbol für die Dateneingabe und -ausgabe verwendet, weshalb im Folgenden nicht darauf verzichtet wird.

Die grundsätzliche Struktur eines PAP wird in Abbildung 4 mit den wesentlichen Elementen beispielhaft an einem abstrakt gehaltenen Ablaufplan dargestellt. Dabei ist allen PAPs gemein, dass sie mit einem Startsymbol begonnen und mit einem oder, bei Verzweigungen im Programmfluss, mehreren Endsymbolen beendet werden. Diese werden als Ovale visualisiert und in der Regel mit den Bezeichnungen „Start“ und „Ende“ beschriftet. Des Weiteren werden diese und alle weiteren Kontrollstrukturen durch Pfeile in Richtung des Programmflusses verbunden. Zwischen Start- und Endsymbol wird der Programmablauf in Verarbeitungen (Rechtecke), die die Anweisungen des Programms enthalten, in Verzweigungen (Rauten), die den Ablauf anhand einer booleschen Bedingung verzweigen, und in Schleifen, die wiederholte Programmausführungen durch Rückführung von Pfeilen zu einer Verzweigung oder einem vorherigen Prozessschritt darstellen, unterteilt. Für die Eingabe und Ausgabe von Daten wird zudem das Symbol eines Parallelogramms verwendet, wobei der Inhalt des Symbols angibt, um welche Art von Datenfluss es sich handelt.

Der Inhalt sowie die formale Beschreibung des Inhalts der einzelnen Symbole werden nicht durch die Normen vorgegeben [35]. Dies ermöglicht den universellen Einsatz von PAPs in verschiedenen Anwendungsbereichen, unabhängig von formalen Vorgaben wie der Syntax spezifischer Programmiersprachen, und die freie Wahl eines geeigneten Abstraktionsgrads für die Darstellung des Programms. Darüber hinaus erlaubt die visuell nachvollziehbare Struktur des PAPs in Verbindung mit der zielgruppengerechten Gestaltungsmöglichkeit eine Überprüfung des Programms auf die zu erfüllenden Funktionalitäten und korrekte Ausführung entlang der Kontrollstrukturen.

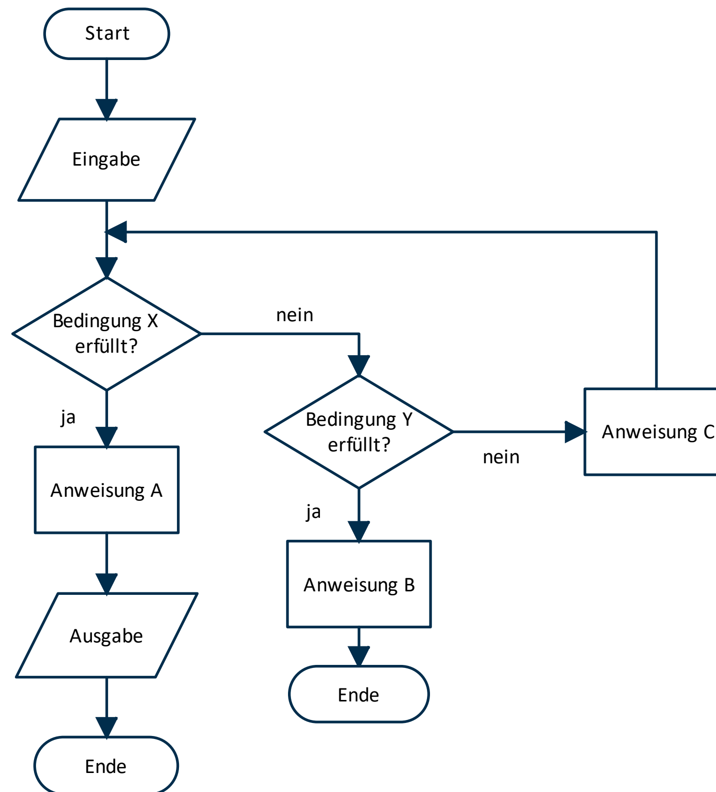


Abbildung 4: Beispiel Aufbau Programmablaufplan

Die Nachteile, die sich aus der Nutzung von PAPs ergeben, bestehen darin, dass die Modellierung umfangreicher und komplexer Programme mit vielen Iterationen und Selektionen unübersichtlich wird, was Korrekturen und Erweiterungen erschwert. Das Finden eines geeigneten Abstraktionsgrads, welcher alle für eine ausreichende Kommunikation mit den Adressaten des PAP notwendigen Informationen enthält, stellt überdies eine Herausforderung dar. Schließlich ist auch die strikte Trennung von Daten und Programmablauf problematisch, da eine integrierte Betrachtung übergeordneter Geschäftsprozesse, die als Grundlage für den Programmentwurf dienen und sowohl Abläufe als auch Daten zusammenhängend umfassen, behindert wird [1].

Das Bestreben, Geschäftsprozesse in die Programmentwicklung mit einzubeziehen und eine bessere Anpassung der Programme an die Anforderungen zu erreichen, war auch ein wesentlicher Grund, neben verbesserter Wartbarkeit und Wiederverwertbarkeit, warum die strukturierte Programmierung nach und nach von der objektorientierten Programmierung abgelöst wurde. In der objektorientierten Programmierung werden Daten und ihre Verarbeitung in Objekten zusammengefasst, was eine engere Verknüpfung von Programmlogik und realen Geschäftsprozessen ermöglicht [1].

Diese Art der Programmierung brachte mit den Methoden und Verfahren der Unified Modeling Language (UML), die aktuell im Standard 2.5.1 festgelegt ist [36], neue Darstellungsformen für Programmabläufe mit sich, welche die Integration von Daten und Geschäftsprozessen zum Hauptbestandteil haben und stark auf den Prinzipien der Objektorientierung aufbauen [1].

Auch wenn angesichts der geschichtlichen Entwicklung angenommen werden könnte, dass das Konzept des Programmablaufplans im Laufe der Zeit mit den Veränderungen der Software–entwicklung und der zugrunde liegenden Technologien veraltet und kaum noch praktikabel ist, zeigen neben dieser Arbeit auch neuere Forschungsbeiträge, dass das Thema weiterhin in der Forschung, Lehre und in vielen Anwendungsbereichen von Interesse ist. Die Vorteil, welche die Darstellung als PAP hat, bleiben weiterhin bestehen und werden mit neuen Technologien und Konzepten genutzt. So gibt es beispielsweise Ansätze mit Hilfe von künstlicher Intelligenz im Bereich Natural Language Processing im Rahmen der Interpretation textueller Programm–beschreibungen (Pseudocode) automatisch PAPs zu erzeugen [37, 38] oder mit Generative AI, wie einem Generative Pre-trained Transformer (GPT), aus Prompt-Anforderungen Erklärungen mit Hilfe von PAPs zu generieren [39]. Auch die Einführung von Programmablaufplänen in der Lehre, beispielsweise in der Elektrotechnik zur visuellen Unterstützung bei der Netzwerkanalyse, werden untersucht [40].

3 Konzeption

3.1 Rahmenbedingungen und Gesamtkonzeption

Im ersten Schritt der Konzeption werden die inhaltlichen und technischen Rahmenbedingungen festgelegt, um die Grundlagen für die Erstellung eines Gesamtkonzepts zu schaffen. Auf dieser Basis werden im Rahmen des Gesamtkonzepts die spezifischen Teilaufgaben zur Bearbeitung der Aufgabenstellung ausgearbeitet, wobei das Teile-und-Herrsche-Prinzip der Informatik Anwendung findet.

3.1.1 Spezifikation der Zielgraphen

Um ein Programm und einen Programmablaufplan (PAP) hinsichtlich ihrer Übereinstimmung miteinander vergleichen zu können, müssen beide zunächst in eine vergleichbare Form überführt werden. Dabei bietet sich die Repräsentation als Graph, genauer gesagt als Kontroll–flussgraph (CFG), an. Dies liegt daran, dass sowohl Programme als auch PAPs bereits die grundlegenden Eigenschaften eines CFGs teilen, wie etwa die Darstellung von Ablaufstrukturen, Verzweigungen und Schleifen. Für die Bearbeitung der Aufgabenstellung müssen die zu vergleichenden CFGs hinsichtlich ihrer Eigenschaften aber genauer spezifiziert und teilweise eingeschränkt werden. Die beispielhafte Darstellung eines Graphen, der die wesentlichen Festlegungen für die CFGs beinhaltet, ist in Abbildung 5 zu sehen.

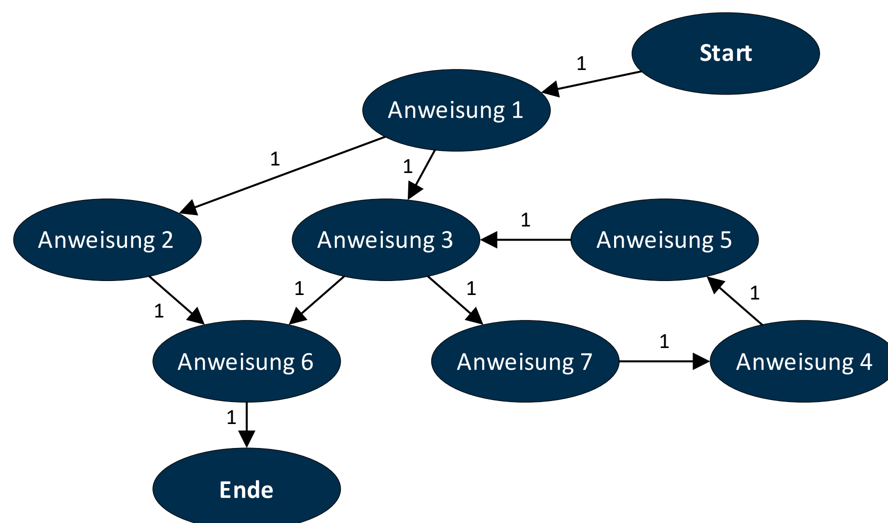


Abbildung 5: Grundstruktur der betrachteten Graphen

In den hier durchgeführten Betrachtungen, die vor allem strukturelle Eigenschaften (siehe Kapitel 2.1.1) beinhalten, wie das Vorhandensein von Pfaden und die Nachbarschaft der Knoten, wird auf Größen zur Programmoptimierung, wie beispielsweise Ausführungswahrscheinlich–keiten oder Laufzeitkosten, die sich durch Gewichtung einbeziehen lassen, verzichtet und den Kanten eine standardmäßige Gewichtung von 1 zugewiesen. Aufgrund der Gleichwertigkeit wird dabei zukünftig auf die Ausweisung der Gewichtung an den Kanten verzichtet.

Darüber hinaus werden die Basisblöcke der CFGs auf eine Anweisung pro Block beschränkt, sodass jeder Knoten nur mit einer Anweisung gekennzeichnet wird. Dadurch lassen sich Abweichungen zwischen den zu vergleichenden Graphen direkter erkennen und berechnen. Da PAPs keine Einschränkungen bezüglich der Inhalte der Kontrollstrukturen haben und aus Programmen automatisch erzeugte CFGs in der Regel mit Blockreduzierung entstehen, muss ein zu entwickelnder Algorithmus als Vorverarbeitung eine Aufsplittung der Anweisungen auf einzelne Knoten vornehmen.

Des Weiteren wird als Ergänzung zu den in Abschnitt 1.2 gemachten Ausführungen zu einfachen C-Programmen präzisiert, dass der Untersuchungsrahmen auf intraprozedurale, schlichte CFGs beschränkt ist. Das schließt auch PAPs mit mehreren Endpunkten im Programmablauf aus, weil gefordert wird, dass nur ein Austrittsknoten am Ende eines CFG vorliegt. Durch die Beschränkung auf einfache C-Programme und damit einfache Selektion und Iteration, wird der maximale Ausgangsgrad der Knoten mit 2 festgesetzt. Es sind Zyklen möglich, weshalb ein zu entwickelnder Algorithmus in der Lage sein muss, diese zu verarbeiten.

Abschließend muss zu den Eigenschaften noch einschränkend erwähnt werden, dass die Knotenlabels von CFGs aus manuell erzeugten PAPs durchaus auch keinem Muster einer programmatischen Anweisung entsprechen können. Dies ist dadurch begründet, dass PAPs keine Einschränkungen bezüglich der Inhalte der Kontrollstrukturen haben und somit ein hoher Grad an Abstraktion bei der Formulierung des Inhalts gewählt werden kann. Ein Kernaspekt dieser Arbeit ist ein möglicher Umgang mit diesem Umstand. In Bezug auf die Kontrollstrukturen des PAP ist zu sagen, dass die visuellen Gestaltungselemente (Rechteck, Raute, Oval) in Programmen keine direkte Entsprechung haben. Es müssten umständlich Relationen über die Anweisungen hergestellt werden, um eine Vergleichbarkeit zu ermöglichen. Aus diesem Grund spielen diese Elemente in den folgenden Betrachtungen keine Rolle, was bei einer Bewertung des PAP in Bezug auf die Korrektheit der verwendeten Gestaltungselemente berücksichtigt werden muss. Zudem können bei der Erstellung manueller PAPs Fehler auftreten, beispielsweise durch fehlende Knoten oder Kanten, die den Zusammenhang im Graphen zerstören. Auch diese muss ein Algorithmus in angemessener Weise berücksichtigen.

3.1.2 Vergleichsbasis der Graphen

Nachdem eine geeignete Repräsentation mit den zugrundeliegenden Eigenschaften gefunden wurde, muss im Rahmen des Vergleichs der beiden Kontrollflussgraphen, im Folgenden als CFG für den automatisch erstellten Programm-Kontrollflussgraphen sowie als FC (Flow Chart) für den des Programmablaufplans bezeichnet, festgelegt werden, auf welchen Eigenschaften der Vergleich zwischen CFG und FC erfolgen soll. Dies ist erforderlich, weil ein zu entwickelnder Algorithmus dahingehende Eigenschaften erfassen und auswerten muss.

In dieser Arbeit erfolgt der Vergleich von CFG und FC in Form paarweiser Knotenvergleiche. Diese basieren auf den Knotenlabels, die als Anweisungen eines C-Programms angenommen werden, sowie auf der lokalen Nachbarschaft der Knoten, die durch ihre Vorgänger und Nachfolger bestimmt wird. Grund hierfür ist zum einen die Überprüfung studentischen Wissens auf die korrekte syntaktische Umsetzung von Anweisungen anhand der Labels und zum anderen die Verbesserung der Zuordnung der passenden Knoten zueinander im Sinne der Isomorphie.

Durch eine Einbeziehung der Nachbarschaft in den Vergleich, soll sichergestellt werden, dass eine bijektive Abbildung zwischen Knoten gewährleistet wird, da es Knoten geben könnte, welche gleiche Labels tragen und Bijektivität durch einen reinen Labelvergleich damit nicht sichergestellt werden kann.

3.1.3 Technische Rahmenbedingungen

Die technischen Rahmenbedingungen für das Konzept sowie die spätere Umsetzung werden maßgeblich durch die Anforderungen bestimmt. Da die entwickelte Lösung in einer .NET-Umgebung nutzbar sein soll, liegt es nahe, die Entwicklung ebenfalls in diesem Umfeld durchzuführen.

Für die technische Umsetzung bietet sich C# als Programmiersprache an, da es vollständig mit den Features von .NET kompatibel ist und somit die Notwendigkeit entfällt, Schnittstellen zu anderen Umgebungen zu entwickeln. Die Entwicklungsumgebung Visual Studio stellt durch ihre umfassende Integration von Werkzeugen zur Codeerstellung, Projektverwaltung, Debugging und Testdurchführung eine ideale Plattform für die Realisierung bereit. Die zu entwickelnde Lösung lässt sich klar und übersichtlich in verschiedene Projekte unterteilen, die die geforderte Bibliothek, eine prototypische Konsolenanwendung sowie eine Unit-Testsuite zur Überprüfung der Korrektheit der implementierten Lösung umfassen. Die Testsuite wird mit xUnit erstellt, einem modernen Test-Framework, das sich durch eine einfache Handhabung und eine gute Unterstützung für aktuelle .NET-Versionen auszeichnet.

3.1.4 Gesamtkonzept der Planung

Bei der Gestaltung eines Gesamtkonzeptes zur Bearbeitung der Aufgabenstellung wird das EVA-Prinzip zur Anwendung gebracht. Dabei handelt es sich um ein grundlegendes Konzept in der Informatik, nach welchem Softwareentwicklung betrieben wird, in dem die Strukturierung eines Programms in die Teile Eingabe, Verarbeitung und Ausgabe vorgenommen wird. Die Struktu–rierung der Konzeptplanung kann in Abbildung 6 nachverfolgt werden.

Ausgangspunkt der Überlegungen ist die Dateneingabe. Im konkreten Fall muss zunächst geklärt werden, wie C-Programm und PAP in eine .NET-Anwendung, in Form einer Bibliothek, überführt werden können. Für das C-Programm stellt sich dabei die Frage, wie zunächst der Kontroll–flussgraph gebildet und anschließend die interne Repräsentation (IR) erzeugt werden kann, um diese für die Integration in die .NET-Anwendung nutzbar zu machen. Erste Überlegungen zur Beantwortung dieser Frage umfassen den Einsatz von Compilern wie GCC oder des LLVM-Frameworks mit Clang als C-Frontend, die später noch genauer erklärt werden. Diese Überlegungen werden im Rahmen der Konzeption der Komponente (Abb.6, Nr. I) in Bezug auf die Integration des CFG mittels eines Parsers fortgeführt und entschieden. Im Zuge der Erstellung des PAP müssen Überlegungen angestellt werden, wie Studenten einen PAP anfertigen sollen, entweder als Abgabe in Form eines speziellen Datenformats oder mit Hilfe eines grafischen Tools. Dabei müssen zusätzliche Überlegungen angestellt werden, ob im Fall einer grafischen PAP-Erstellung ein eigenes Tool erstellt werden muss oder ein Tool eines Drittanbieters verwendet werden kann. Diesen Fragen wird in der Konzeption der Komponente

(Abb.6, Nr. II) zur Erstellung eines Parsers für den PAP-Import weiter untersucht und eine Entscheidung getroffen.

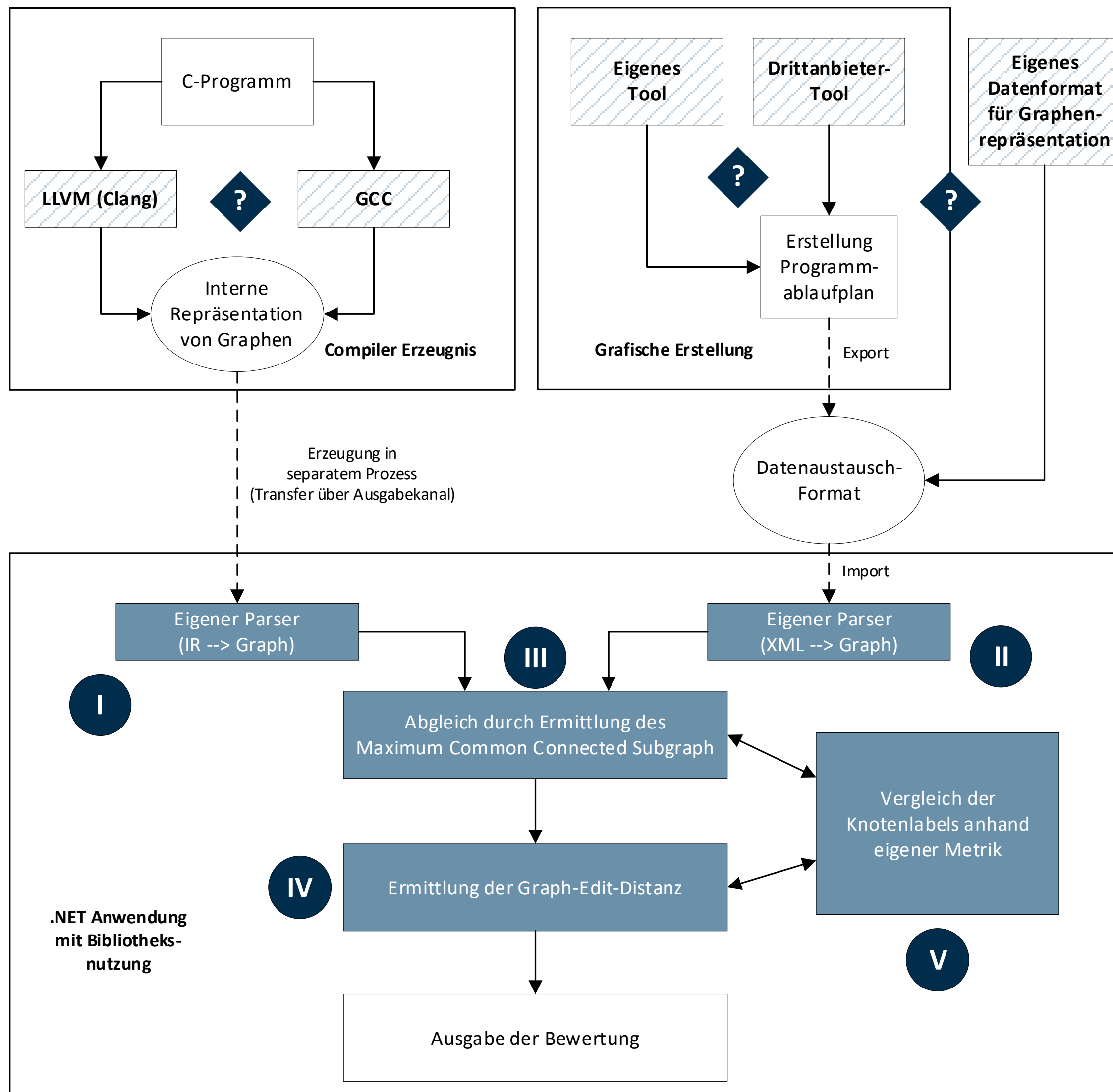


Abbildung 6: Entwicklung des Gesamtkonzepts

Im Anschluss an die Dateneingabe erfolgen Überlegungen zur Datenverarbeitung. In der Komponente (Abb. 6, Nr. III) werden zunächst die Eingaben von CFG und FC entgegengenommen, und auf dieser Basis wird ein Konzept für den strukturellen Vergleich mittels Subgraphen entwickelt. Anschließend erfolgt in Komponente (Abb. 6, Nr. IV) die Ausarbeitung einer Bewertung des Vergleichs, basierend auf den Anpassungskosten beider Graphen bei Abweichungen in der Struktur. In Komponente (Abb. 6, Nr. V) wird ein Vergleichskonzept für die Knotenlabels erstellt, das in den verbundenen Komponenten zum Einsatz kommt. Abschließend erfolgt im Rahmen der Datenausgabe des EVA-Prinzips die Ausgabe der Bewertungsergebnisse. Aufgrund ihres einfachen, prototypischen Charakters ist hierfür jedoch keine eigene Konzeption erforderlich.

Abschließend wird das exemplarische C-Programm vorgestellt, das im Rahmen der Konzeption und Umsetzung als Entwicklungsbeispiel dient und zugleich als Grundlage für die erstellten

Testfälle fungiert. Dabei handelt es sich um eine einfache C-Funktion zur iterativen Ermittlung des größten gemeinsamen Teilers (ggt-Funktion), deren Aufbau sowohl im Code als auch in der Graphenrepräsentation in Abbildung 7 veranschaulicht wird.

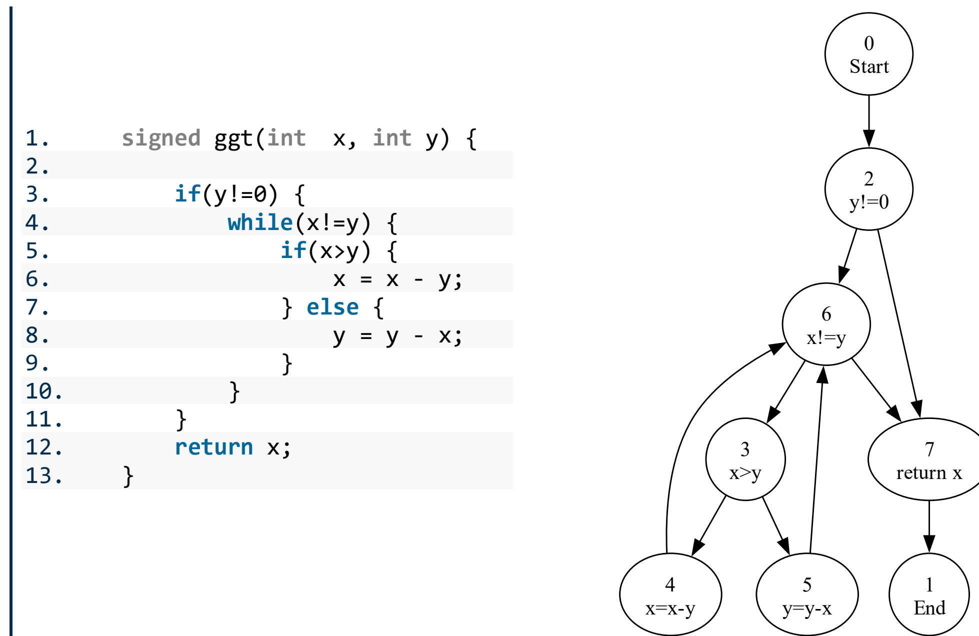


Abbildung 7: Exemplarisches C-Programm mit Graphenrepräsentation

Diese Funktion enthält mit Sequenzen, einer Schleife und zwei Verzweigungen alle wesentlichen Kontrollstrukturen, die repräsentativ für einfache C-Programme sind und ist somit ideal als Referenz einsetzbar. Die Schleife bietet die Möglichkeit, den entwickelten Algorithmus auf Robustheit gegenüber Zyklen zu prüfen und im Rahmen des strukturellen Vergleichs der Subgraphen Vorwärts- und Rückwärtskanten auf ihre korrekte Integration in den Subgraphen zu untersuchen. Vorwärtskanten sind dabei Kanten zu späteren Knoten in der Traversierungs–reihenfolge, während Rückwärtskanten Kanten zu vorherigen Knoten in dieser Reihenfolge sind.

Zur Graphenrepräsentation sei erwähnt, dass diese mithilfe der Software Graphviz [41] erstellt wurde. Graphviz erzeugt aus Dateien im DOT-Format, einem Textformat zur Beschreibung und Gestaltung von Graphen, visuelle Darstellungen. Diese werden im Rahmen der Sichtkontrolle verwendet, um die korrekte Erstellung der Graphen zu überprüfen.

3.2 Extraktion von Kontrollflussgraphen aus C-Programmen

Die Erstellung des CFG aus einem bestehenden C-Programm kann entweder durch spezialisierte Software-Tools für die Programmanalyse oder als Teil des Kompilierungsprozesses einer C-Datei zur ausführbaren Datei durch Compiler erfolgen. Da im Rahmen dieser Aufgabenstellung die Anforderungen an den zu erstellenden CFG darauf begrenzt sind, lediglich die grundlegende Struktur des Kontrollflusses und seine Anweisungen darzustellen und keine tiefgreifenden Analysen erforderlich sind, wurde die Entscheidung zugunsten der Erstellung des CFG durch einen Compiler getroffen. Spezialsoftware bietet aufgrund ihres Einsatzzwecks, beispielsweise als Analysewerkzeug zur Fehlererkennung oder Sicherheitsprüfung, umfangreichere, rechen–intensive Prüfungen des Programms, was einen höheren Ressourcenverbrauch und zusätzlichen Konfigurationsaufwand mit sich bringt, der in diesem Fall nicht erforderlich ist. Darüber hinaus sind viele spezialisierte Software-Tools kostenpflichtig oder ihre Lizenzierung ist schwierig, was den Aufwand im Fall einer Nutzung weiter erhöhen würde.

3.2.1 Compiler-Auswahl

Nachdem die Entscheidung getroffen wurde, den CFG durch einen Compiler zu erstellen, muss im nächsten Schritt eine geeignete Eingrenzung der zu betrachtenden Compiler für die Programmiersprache C getroffen werden, da es aufgrund der Bedeutung und Verbreitung von C in der Systemprogrammierung eine Vielzahl an C-Compilern gibt. Die beiden Compiler-Frameworks GNU Compiler Collection (GCC) und Low Level Virtual Machine (LLVM) wurden aufgrund ihrer Popularität in der Praxis und Lehre, der aktiven Community mit regelmäßigen Updates im Rahmen der Open-Source-Lizenzierung sowie der umfassenden Dokumentation ihrer Funktionalitäten und Konfigurationen als geeignete Optionen ausgewählt. GCC war ursprünglich ein reiner C-Compiler, hat sich jedoch im Laufe der Zeit zu einer Sammlung von Compilern für verschiedene Programmiersprachen wie C++, Objective-C, Fortran und Go weiterentwickelt [42]. Bei LLVM hingegen handelt es sich um eine modulare Sammlung von Werkzeugen zur Compiler- und Toolchain-Erstellung, die ursprünglich aus einem Forschungsprojekt zur Untersuchung der Kompilierung von beliebigen Programmiersprachen hervorgegangen ist, wobei Clang das C-Frontend von LLVM bildet [43].

Im Anschluss an die Auswahl muss geprüft werden, ob sowohl GCC als auch Clang den erforderlichen Funktionsumfang für die Erstellung eines leicht auswertbaren CFGs abdecken, der alle notwendigen Informationen enthält. Darüber hinaus ist es wichtig zu klären, ob eine nahtlose Integration beider Compiler in die zu entwickelnde .NET-Bibliothek sowie der Export des CFGs möglich sind. Ziel dabei ist es, dem Nutzer zusätzlichen Aufwand, wie etwa die Installation eines Compilers oder die manuelle Generierung des CFGs innerhalb einer Anwendung, zu ersparen und damit die Benutzerfreundlichkeit zu erhöhen. Basierend auf diesen Kriterien wird im letzten Schritt der geeignetere von den beiden Compiler-Frameworks ausgewählt und ein Konzept zur Implementierung entwickelt.

Beide Compiler werden über ein Command-Line Interface (CLI), die Kommandozeile, ausgeführt, was die Integration in die .NET-Bibliothek erleichtert, da zur Nutzung des Compilers innerhalb der Bibliothek lediglich ein Startbefehl mit den entsprechenden Parametern ausgeführt und

keine umfangreiche Schnittstelle entwickelt werden muss. Das CLI bieten über Parameter gemäß (16) für Clang und (17) für GCC bei der Ausführung die Möglichkeit CFGs zu erstellen.

Clang CLI-Befehl → *clang -cc1 -analyze -analyzer-checker=debug.DumpCFG ggt.c (=debug.ViewCFG ggt.c)* (16)

GCC CLI-Befehl → *gcc -fdump-tree-cfg-graph ggt.c* (17)

Bei der Verwendung des Clang-Befehls wird mit *-cc1 -analyze* direkt auf das C-Frontend zugegriffen und die statische Analyse gestartet, ohne den vollständigen Kompilierungsprozess zu durchlaufen. Mittels *-analyzer-checker* werden die spezifischen Prüfungen bzw. Werkzeuge angegeben, die während der Analyse ausgeführt werden sollen. In diesem Fall wird für die Erstellung des CFG entweder *debug.DumpCFG* verwendet, um die Ausgabe im CLI darzustellen, oder *debug.ViewCFG*, um eine DOT-Datei zu erzeugen. Der abschließende Parameter *ggt.c* gibt mit dem exemplarischen C-Programm die zu untersuchende Datei an. Die Ausgabe als DOT-Datei ist in Abbildung 8 auszugsweise dargestellt und kann im Anhang A1 vollständig eingesehen werden. Die CLI-Ausgabe unterscheidet sich lediglich dadurch, dass sie die visuelle Darstellung als Graph nicht enthält und deswegen Vorgänger und Nachfolger mittels Blocknummern explizit angegeben sind.

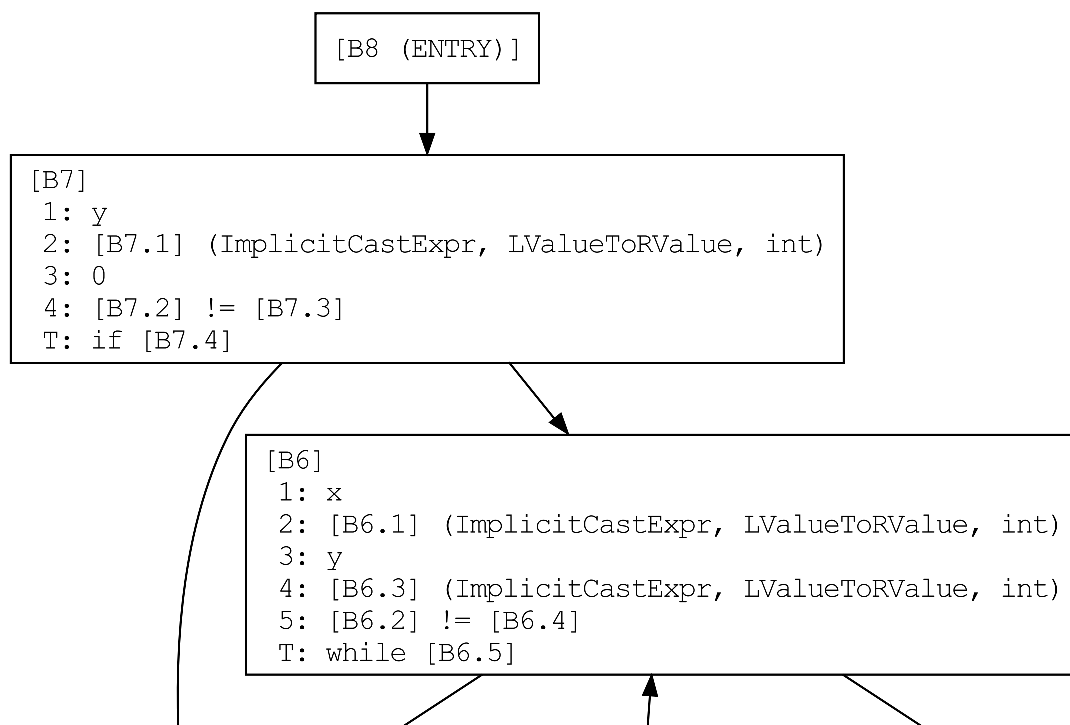


Abbildung 8: Visuelle Repräsentation in Clang (Auszug)

In der Abbildung werden drei Knoten bzw. Basisblöcke des CFG mit den Bezeichnern [B6-B8] visualisiert, wobei [B8] den Startknoten ohne Inhalt, [B7] die erste Bedingung der ggt-Funktion und [B8] den Kopf der while-Schleife darstellen. Erkennbar ist, dass die Anweisung für Analyse- und Optimierungszwecke in jedem Block in ihre Bestandteile zerlegt und temporären Hilfs–variablen zugeordnet wird. Die Anweisung kann durch Rückwärtsverfolgung der Zeilen wieder zusammengesetzt werden.

Die Ausführung des GCC-Befehls gemäß (17) erfolgt durch die Verkettung von Optionen als erstem Parameter. Mit der Option *fdump-tree* wird die interne Repräsentation des Programms festgelegt und mit *cfg-graph* wird diese sowohl als CFG in einer reinen Textdatei (*.cfg) als auch als Datei im DOT-Format ausgegeben. Der zweite Parameter *ggt.c* gibt die zu analysierende Datei an. In Abbildung 9 findet sich ein Ausschnitt der Darstellung in Graphenform. Für den vollständigen CFG wird auf Anhang A2 verwiesen.

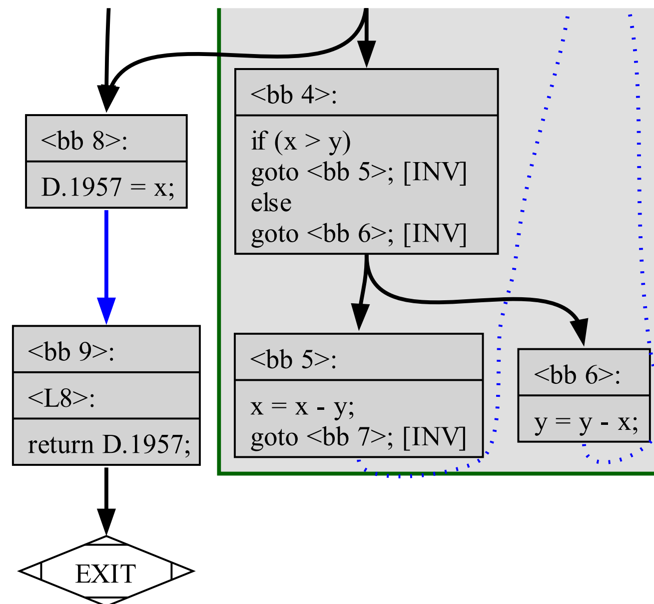


Abbildung 9: Visuelle Repräsentation in GCC (Auszug)

Im Gegensatz zum mittels Clang erzeugten CFG werden in dieser Repräsentation die Anweisungen in den Basisblöcken, welche mit *bb*-Markierungen gekennzeichnet sind, unverändert beibehalten. Eine Ausnahme davon bildet der Rückgabewert, welcher für die Weiterverwendung in einem funktionsübergreifenden Kontext in einer temporären Variable *(D.1957)* abgelegt wird. Die Beziehungen zwischen den Basisblöcken werden sowohl in der visuellen als auch internen Repräsentation durch *goto*-Markierungen mit Angabe des Zielblocks angezeigt.

Die Analyse beider Compiler zeigt, dass sowohl GCC als auch Clang (LLVM) durch ihre CLI-basierte Steuerung eine einfache Integration in die .NET-Bibliothek ermöglichen. Darüber hinaus sind beide Tools in der Lage extrahierbare CFGs zu erzeugen, entweder über die CLI-Ausgabe bei Clang oder über die cfg-Datei bei GCC. Beide Compiler enthalten alle Informationen die für die Aufgabenbearbeitung notwendig sind, bei der Darstellung der Anweisungen innerhalb der Basisblöcke gibt es aber Unterschiede. Die Aufteilung der Anweisungen in Hilfsvariablen im Clang-Compiler sorgt dafür, dass bei der Erstellung einer Graphendarstellung in der .NET-Bibliothek entweder eine Rückabwicklung zu den vollständigen Anweisungen oder eine vergleichbare Aufteilung der Anweisungen im Kontrollflussgraphen des PAP vorgenommen werden muss, um die Vergleichbarkeit sicherzustellen. Dieser zusätzliche Aufwand entfällt bei der Nutzung des GCC-Compilers und macht ihn damit zur bevorzugten Wahl.

3.2.2 Erstellung der internen CFG-Repräsentation

Abschließend wird das Konzept zur Überführung des Kontrollflussgraphen, welcher durch die Kompilierung mit GCC entsteht, in den Zielgraphen des internen CFG der .NET-Bibliothek vorgestellt. Die Überführung erfolgt im Wesentlichen in zwei Schritten.

Im ersten Schritt muss der GCC-Compiler innerhalb der .NET-Bibliothek nutzbar gemacht werden, ohne dass ein aktives Eingreifen des Anwenders erforderlich ist. Der Ansatzpunkt hierfür ist die Ausführung von GCC in einem separaten Systemprozess, der ausgehend von einer Anwendung, wie der prototypischen Konsolenanwendung, beim Aufruf einer Bibliotheks–funktion gestartet wird. Der Parameter mit den Optionen *-fdump-tree-cfg* zur Erstellung des CFG kann übernommen und fest in den GCC-Aufruf des Prozesses integriert werden, wobei auf die Option *-graph* verzichtet wird, um ausschließlich die cfg-Datei mit der internen Repräsentation zu erhalten. Der Parameter zur Dateispezifizierung muss von der Anwendung bereitgestellt werden, indem bei Programmstart die C-Datei als Eingabeparameter angegeben wird, die als Grundlage für die Erstellung des CFG dient. Die erstellte cfg-Datei muss vom Prozess im Dateisystem an einer per Parameter festgelegten Stelle abgelegt werden, um sie für den nächsten Schritt verfügbar zu machen.

Der zweite Schritt umfasst das Einlesen der erstellten cfg-Datei und das Parsen dieser in die interne Graphenrepräsentation, wie in Abbildung 10 visualisiert.

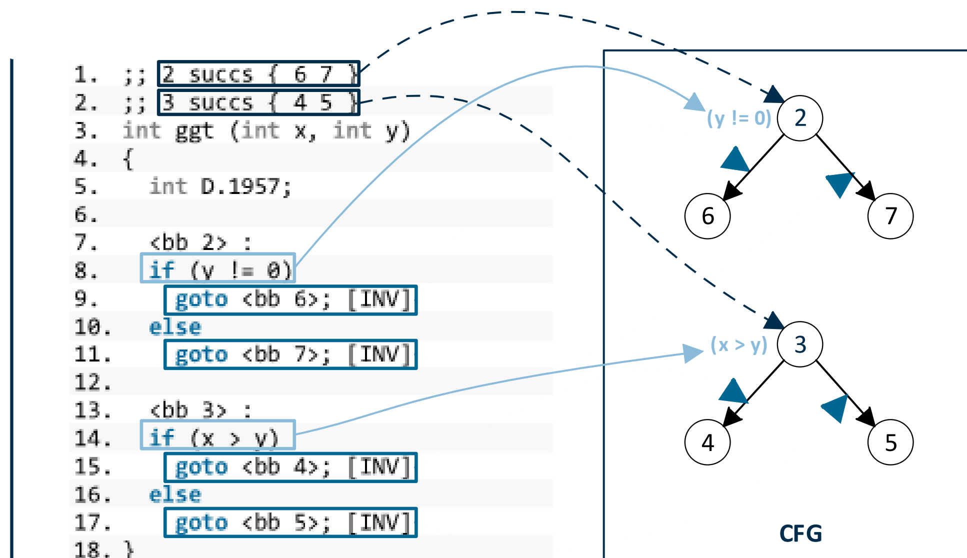


Abbildung 10: Parsen der internen Repräsentation (Auszug)

Dabei werden im Zuge zeilenweiser Auswertung des Eingangsdatenstroms der cfg-Datei zunächst alle Knoten aus den Compiler-Kommentaren zur CFG-Erstellung (Zeile 1 + 2) erstellt. Anschließend werden im Funktionskörper, nach Erkennen der Basisblocknummer, die ent–sprechenden Anweisungen den Knoten als Label zugeordnet (Zeile 8 + 14). Anhand der *goto*-Markierungen (Zeile 9, 11, 15, 17) werden abschließend die Kanten eingefügt, sodass am Ende der vollständige Zielgraph in der Anwendung zur Verfügung steht. Da Start- und Endblock in der cfg-Datei nicht angegeben werden, müssen diese separat erstellt und Kanten zum ersten

Basisblock der Funktion für den Startknoten und vom letzten Basisblock zum Endknoten generiert werden. Für die interne Identifikation der Knoten werden die ID-Nummern aus den Basisblocknummern übernommen, wobei für den Startknoten immer ID = 0 und den Endknoten ID = 1 vergeben werden, entsprechend der von GCC generell festgelegten Basisblocknummern für Start- und Endblock [44].

Die Datenstruktur, in welcher der Zielgraph in der Anwendung nach Abschluss des Parsens vorliegt, ist eine Adjazenzliste. Dabei handelt es sich um eine Liste aller Knoten des Graphen, wobei jedem Knoten zusätzlich eine Referenzliste mit den zu ihm adjazenten Knoten zugeordnet ist. Diese Referenzliste stellt in einem gerichteten Graphen somit die Kanten zu den Nachfolgern des Knotens dar und erlaubt das Traversieren ausgehend von einen Startknoten entlang der Nachfolger. Die Adjazenzliste wird im Rahmen der Aufgabenstellung jedoch erweitert. Da im Zuge des Vergleichs von Knoten neben dem Label auch die Nachbarschaft, welche Nachfolger und Vorgänger einschließt, betrachtet werden soll, wird zusätzlich zur Referenzliste der Nachfolger eine weitere Liste für die Vorgänger angelegt.

Die Entscheidung als grundlegende Datenstruktur des Zielgraphen eine Adjazenzliste zu ver– wenden, basiert auf der Tatsache, dass der maximale Ausgangsgrad eines Knotens zwei beträgt und damit nur wenige Kanten im Zielgraphen existieren. Adjazenzlisten speichern nur vorhandene Kanten was sie im Vergleich zu anderen Datenstrukturen, wie Matrizen, welche Einträge zu allen möglichen Kanten speichern, effizienter im Speicherverbrauch werden lässt.

3.3 Import manuell erzeugter Programmablaufpläne

Bei der Erstellung des Kontrollflussgraphen aus einem Programmablaufplan muss zunächst entschieden werden, in welcher Form der PAP für den Abgleich eingereicht werden soll. Dabei gibt es grundsätzlich zwei Möglichkeiten: die klassische grafische Darstellung eines PAPs, die jedoch in digitale Form überführt wird, oder die textuelle Beschreibung des PAPs mithilfe strukturierter Datenformate wie XML oder JSON. Da neben der Wissensvermittlung zu Programmabläufen auch die Nutzung des Programmablaufplans als Werkzeug der grafischen Modellierung ein Ziel der Lehre ist, würde eine rein textuelle Beschreibung durch den Medienbruch diesem Ziel entgegenwirken. Daher ist eine grafische Darstellung zu bevorzugen. Folglich muss entschieden werden, ob das Werkzeug zur Erstellung selbst entwickelt oder über Drittanbietertools bereitgestellt werden soll. Diese Entscheidung lässt sich jedoch schnell treffen, da eine zusätzliche Eigenentwicklung den inhaltlichen und zeitlichen Rahmen dieser Aufgabe überschreiten würde. Als Konsequenz daraus wird im folgenden Schritt die Auswahl eines geeigneten Drittanbietertools getroffen.

3.3.1 Auswahl des Werkzeugs zur PAP-Erstellung

Die bereits im Zusammenhang mit Programmablaufplänen erwähnte zunehmende Orientierung auf Geschäftsprozesse und deren Modellierung im Unternehmensumfeld, sorgt für eine Vielzahl an verfügbaren Tools mit teilweise sehr spezifischen Anwendungsgebieten. Daher muss eine weitreichende Eingrenzung vorgenommen werden, um die Auswahl angemessen durchführen zu können. Ausgehend von der Zielstellung, ein Tool für die Erstellung von PAPs bereitzustellen, das für Studierende leichte Handhabung, hohe Verfügbarkeit, unkomplizierten Zugang und vor allem geringe Kosten bietet, haben sich verschiedene Tools als geeignet herausgestellt. Die folgenden drei Tools wurden aufgrund jeweils zusätzlicher, besonderer Features ausgewählt.

Das Tool draw.io (auch als diagrams.net bekannt) ist ein beliebtes, kostenloses Open-Source-Tool zur Erstellung verschiedenster Arten von Diagrammen, das sowohl in der Software–entwicklung als auch im Geschäftsumfeld eingesetzt und von der Firma JGraph Ltd. und der draw.io AG bereitgestellt wird [45]. Aufgrund seiner flexiblen Einsetzbarkeit als Webanwendung ohne Registrierung und seiner großen Nutzerzahl zählt es zu den populärsten Diagramm-Tools und wird daher in die Auswahl aufgenommen.

MS Visio hingegen ist ein kostenpflichtiges Produkt der Microsoft Corporation zur Erstellung von Diagrammen und technischen Zeichnungen [46]. Es zeichnet sich besonders durch eine enge Integration in die Microsoft 365-Technologien aus und ist zudem für Studierende der Hoch–schule im Rahmen einer Hochschullizenz kostenfrei verfügbar, was ebenfalls einen Grund für die Aufnahme in die Auswahl darstellt.

Der PAPDesigner als dritte Auswahl ist ein kostenloses Tool, das speziell zur Erstellung von PAPs nach DIN 66001 im Rahmen der Programmierausbildung des Georg-Simon-Ohm-Berufskollegs Köln entwickelt wurde [47]. Durch seinen klaren Fokus auf die PAP-Erstellung stellt es eine wertvolle Ergänzung zu den anderen beiden Tools dar.

Um eine abschließende Auswahl eines der Tools durchführen zu können, muss ein Vergleich bezüglich der in Tabelle 1 aufgeführten Kriterien durchgeführt werden.

Tabelle 1: Vergleich verschiedener Werkzeuge zur PAP-Erstellung

	Draw.io	**PAPDesigner**	**MS Visio**
Diagrammtyp für PAP?	**ja** - große Auswahl verschiedener Diagrammtypen	**ja** - ausschließliche PAP nach DIN 66001	**ja** - große Auswahl verschiedener Diagrammtypen
Handhabung	- generell einfach - schwer für korrekte PAP	- einfache Handhabung - Fokus auf korrekte PAP Elemente	- generell einfach - schwer für korrekte PAP
Kosten	- kostenfrei	- kostenfrei	- über die Hochschule kostenfrei beziehbar
Registrierung	- nicht erforderlich	- nicht erforderlich	- erforderlich
Installation	- Lokale Installation - Webanwendung	- Lokale Installation	- Lokale Installation - SharePoint Cloud
Exportformate	- PDF, JPEG, SVG, HTML, XML	- PNG, TIF, JPG, BMP	- PDF, JPEG, PNG, MS Dokumentformate

Wie der Tabelle zu entnehmen ist, erfüllen alle drei Tools grundsätzlich die Anforderungen an ein PAP-Tool sowie die meisten Kriterien für eine studentische Nutzung. Der Unterschied liegt jedoch darin, dass bei draw.io und Visio zwar eine einfache Bedienung vorliegt, die große Auswahl an Diagrammtypen und Gestaltungselementen jedoch die korrekte Auswahl und Anwendung von PAP-Kontrollstrukturen erschwert. Der PAPDesigner hingegen enthält aus–schließlich spezifische PAP-Gestaltungselemente, wodurch die Gefahr, falsche Elemente zu verwenden, verringert wird, erfordert allerdings eine lokale Installation und ist nicht so flexibel einsetzbar wie die beiden anderen Tools. Ein entscheidendes Kriterium stellt allerdings das Exportformat der PAPs dar. Einzig draw.io bietet mit dem XML-Format ein strukturiertes, leicht übertragbares Datenformat, während Visio und der PAPDesigner vorwiegend grafische Exportformate verwenden, die sich weniger gut für das Parsen in den internen FC eignen. Aus diesem Grund wird draw.io bevorzugt.

3.3.2 Erstellung der internen FC-Repräsentation

Die Erstellung eines PAPs mit der Webanwendungs-Version von draw.io ist in Abbildung 11 dargestellt. Der Nutzer wählt die erforderlichen Kontrollstrukturen für den PAP aus den Bereichen „Allgemein“ oder „Ablaufdiagramm“ aus und platziert diese auf der Zeichenfläche. Anschließend werden die Elemente beschriftet und mit Pfeilen verbunden. Nach Fertigstellung des PAPs erfolgt der Export in Form einer XML-Datei, die als Download bereitgestellt wird.

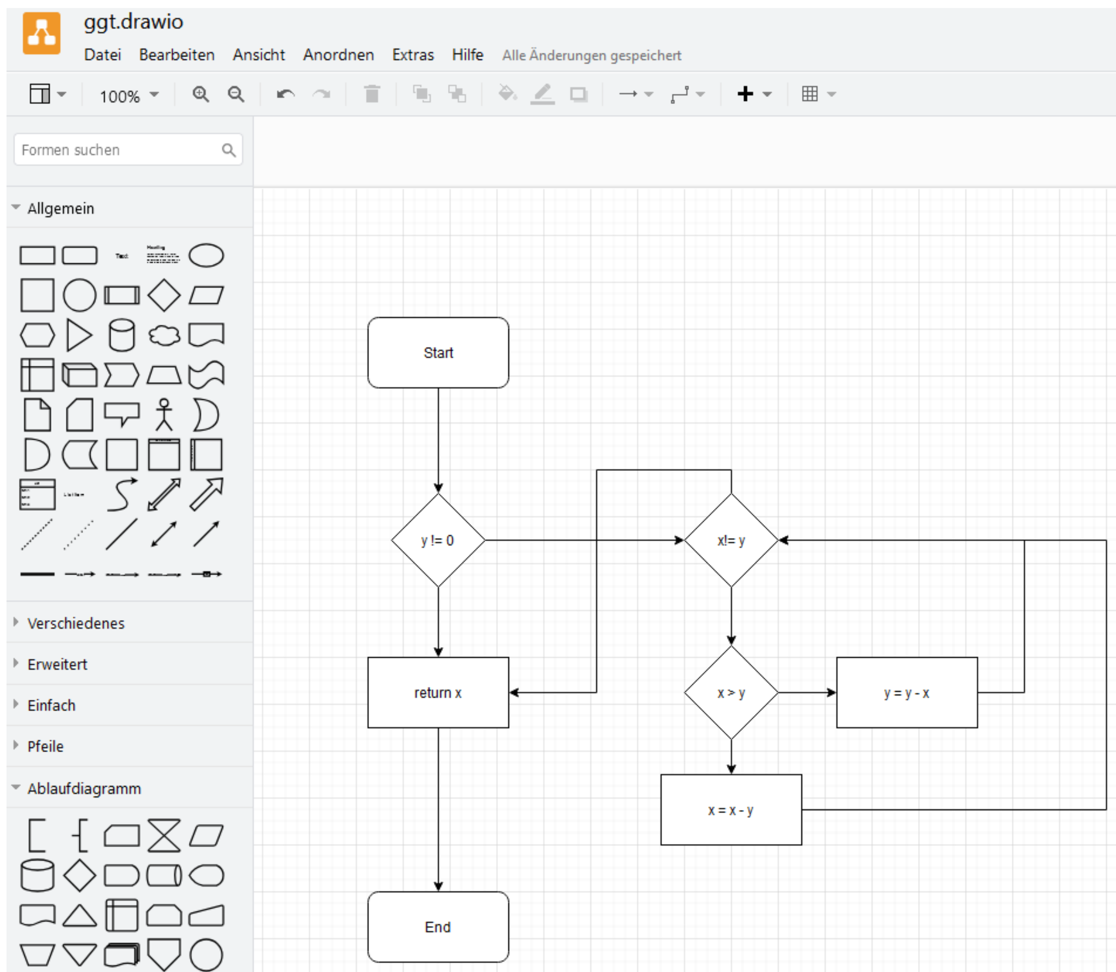


Abbildung 11: Grafische Erzeugung des PAP in draw.io

Um nun im Folgeschritt die XML-Datei in die .NET-Anwendung importieren zu können, muss zunächst der Aufbau der XML-Datei analysiert werden.

Bei XML (Extensible Markup Language) handelt es sich um eine Meta-Sprache zur Beschreibung von hierarchisch strukturierten Daten in einem Textformat [3]. In ihr werden XML-Elemente, die beliebig tief verschachtelt werden können, durch selbstgewählte Markup-Tags, wie beispiels–weise *<tag>…</tag>*, dafür verwendet, den Inhalt und die Bedeutung von strukturierten Daten zu repräsentieren [3]. Diese Elemente können dafür mit Werten zwischen den Tags oder mit Attributen innerhalb der Tags, die zusätzliche Informationen zu dem XML-Element beinhalten, beschrieben werden. Mit XML-Schema-Definitionen (XSD) kann festgelegt werden, welche Elemente in einer XML-Datei enthalten sein dürfen und welche Datentypen sie haben [3]. Eine

XSD kann verwendet werden, um die strukturelle und inhaltliche Validierung einer XML-Datei durchzuführen und so deren Korrektheit und Konsistenz zu gewährleisten.

Für die XML-Datei, die aus dem Export der draw.io-Zeichnung resultiert, ergibt sich in gekürzter Form folgender Inhalt, wie in Listing 1 beschrieben.

```
1.  <root>
2.    <mxcell id="yA2Fn3WxIahBYsPciDm3-9"
3.            value=""
4.            style="edgeStyle=orthogonalEdgeStyle;html=1;"
5.            parent="WIyWlLk6GJQsqaUBKTNV-1"
6.            source="yA2Fn3WxIahBYsPciDm3-2"
7.            target="yA2Fn3WxIahBYsPciDm3-8"
8.            edge="1">
9.      <mxgeometry relative="1" as="geometry"></mxgeometry>
10.   </mxcell>
11.   <mxcell id="yA2Fn3WxIahBYsPciDm3-2"
12.           value="y != 0"
13.           style="rhombus;whiteSpace=wrap;html=1;"
14.           parent="WIyWlLk6GJQsqaUBKTNV-1"
15.           vertex="1">
16.     <mxgeometry x="190" y="260" width="80" height="80"></mxgeometry>
17.   </mxcell>
18. </root>
```

Listing 1: Beispiel draw.io-Graph in XML (gekürzt)

Dargestellt werden zwei XML-Elemente des Typs *<mxcell>*, von denen das erste Element aufgrund seiner Attribute *source*, *target* und *edge* als Kante und das zweite Element mit den Attributen *value* und *vertex* als Knoten identifiziert werden kann. Somit lässt sich innerhalb der XML-Datei bereits die Struktur des PAP als Graph erkennen.

Folglich muss für die .NET-Bibliothek ein XML-Parser entwickelt werden, der die draw.io-Datei einliest und anhand dieser Attribute Knoten und Kanten erstellt, um die Struktur des PAP korrekt in den internen FC zu überführen. Zudem ist zur Verbesserung der Lesbarkeit und Zuordnung der Knoten eine Neuzuweisung der IDs sinnvoll, da diese in der draw.io-Zeichnung schwer nachvollziehbar sind, wie beispielsweise in Zeile 11 ersichtlich.

Ein wichtiger Aspekt, der darüber hinaus beachtet werden muss, ist die Schaffung einer Mög–lichkeit, die .NET-Bibliothek auch ohne draw.io als grafisches Tool nutzbar zu machen, um Abhängigkeiten zu vermeiden. Dafür muss die XML-Schnittstelle der .NET-Bibliothek so gestaltet werden, dass auch ein eigenes, unabhängiges XML-Format für den FC-Import genutzt werden kann. Damit nur wohlgeformte XML-Dateien, die den XML-Spezifikationen entsprechen und für die .NET-Bibliothek valide sind, anerkannt werden, muss zusätzlich eine eigene XSD integriert werden.

3.4 Ermittlung struktureller Gemeinsamkeiten von Graphen

Nachdem sowohl CFG als auch FC in der internen Repräsentation in vergleichbarer Form vorliegen, folgt im Anschluss die Konzeption des strukturellen Vergleichs beider Graphen.

Die Aufgabenstellung umfasst den Abgleich von einfachen C-Programmen und Programm–ablaufplänen. Daher ist davon auszugehen, dass die resultierenden internen Graphen im Normalfall eine begrenzte Größe von etwa 20 bis 30 Knoten aufweisen und aufgrund ihrer Struktur tendenziell wenige Kanten beinhalten. Diese Annahme erscheint realistisch, da eine größere Anzahl an Knoten den manuellen Aufwand für die Erstellung des PAP ohne zusätzlichen Mehrwert unnötig erhöhen würde. Daher wird bei der Konzeption des Vergleichsalgorithmus weniger Augenmerk auf das Isomorphie-Problem bei großen Graphen und die Optimierung der Zeitkomplexität der Algorithmen gelegt. Stattdessen wird das KISS-Prinzip (Keep It Simple, Stupid) der Softwareentwicklung angewendet, das eine einfache, benutzerfreundliche, nachvollziehbare und wartbare Lösung bevorzugt, im Gegensatz zu einer komplexen Lösung, die genau die gegenteiligen Eigenschaften aufweist. Daher werden fertige Bibliotheken oder Tools zur Berechnung von Graphenisomorphie, die vor allem für große Graphen konzipiert wurden, wie NetworkX, graph-tool, Nauty, Bliss oder Traces, nicht berücksichtigt, sondern eine eigene Lösung angestrebt.

Das grundlegende Vorgehen besteht darin, die Traversierung der Graphen und den Abgleich der Knoten gleichzeitig durchzuführen. Dabei werden die Labels und Nachbarschaften der Knoten beider Graphen während der Traversierung direkt verglichen. Auf diese Weise können alle strukturellen Informationen zu den besuchten Knoten unmittelbar verarbeitet werden, wodurch unnötige, redundante Suchen in anderen Datenstrukturen nach diesen Informationen vermieden werden. Dies führt zu einer effizienteren und gleichzeitigen Nutzung aller relevanten Daten, was den Abgleich der beiden Graphen optimiert.

Ein anfänglicher, intuitiver Ansatz des Vergleichs ergibt sich aus der Aufgabe eines Kontrollfluss–graphen alle möglichen Programmabläufe abzubilden, indem beide Graphen mittels rekursiver Tiefensuche über alle Pfade hinweg verglichen werden, die vom Startknoten zum Endknoten verlaufen. Dabei geben die Übereinstimmung von Anzahl und Länge der Pfade in der Gesamtstruktur der Graphen und das direkte Matching während der Traversierung innerhalb der gefundenen Pfade Auskunft darüber, wie ähnlich die Graphen sind. Nachteile dieses Ansatzes sind, dass Redundanz entsteht, weil gemeinsame Teilpfade mehrfach erfasst werden, und dass bei Anomalien in diesen Teilpfaden, wie etwa fehlenden Knoten oder Kanten, ein Fehler mehrfach auftreten kann, was eine präzise Bewertung der Ähnlichkeit erschwert. Aufgrund dessen wurde der Ansatz verworfen und ein alternativer Ansatz gesucht.

Bei der Recherche in Studien nach Möglichkeiten die Struktur der Graphen beim Traversieren möglichst redundanzfrei zu erfassen, wurde der Ansatz über die Erstellung des Maximum Common Subgraphs mittels eines Zustandsraum-Suchalgorithmus in [12] als vielversprechend identifiziert und ein Ansatz basierend darauf entwickelt.

In diesem Ansatz wird für alle Knotenpaare aus CFG und FC eine parallele Traversierung beider Graphen durchgeführt, wobei die Tiefensuche und lokale Knotenvergleiche entlang der Pfade verwendet werden. Die Betrachtung aller Knotenpaare ist notwendig, da aufgrund von Fehlern

im PAP auch nicht zusammenhängende FC auftreten können, die bei einer einmaligen Traversierung nicht vollständig erfasst würden. Im Gegensatz zum anfänglichen Ansatz wird jedes gefundene Matching einem Zwischenzustand hinzugefügt, der für jedes Knotenpaar einen Common Connected Subgraph (CCS) erzeugt. Durch die Speicherung des größten CCS aus jedem Durchlauf wird entweder bei vollständigem Matching beider Graphen, wie in Abbildung 12 dargestellt, der Maximum Common Connected Subgraph vorzeitig oder spätestens nach Abschluss der letzten Traversierung ermittelt.

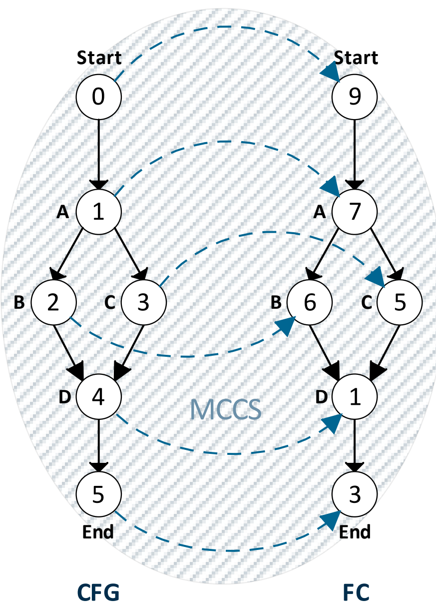


Abbildung 12: vollständiges Matching

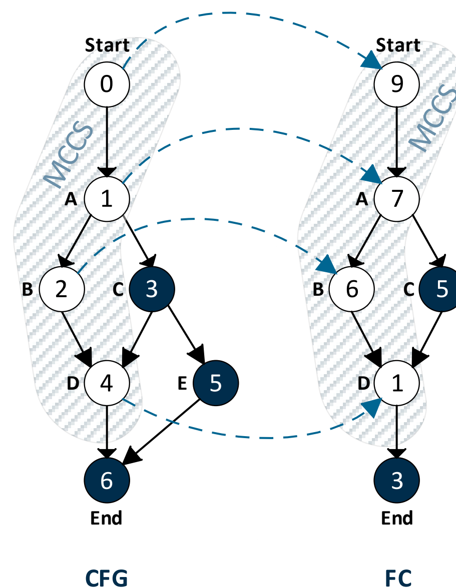


Abbildung 13: unvollständiges Matching

Im Fall eines vollständigen Matchings liegt Isomorphie vor und eine Übereinstimmung zwischen CFG und FC wird ausgegeben. Für den konkreten Anwendungsfall bedeutet dies, dass der PAP und das Programm vollständig übereinstimmen. Sollte der Algorithmus seinen Abschluss ohne vollständiges Matching finden, wie in Abbildung 13 dargestellt, muss für die außerhalb des MCCS liegenden Knoten, in dunkelblau dargestellt, eine Möglichkeit zur Bewertung der Unterschied–lichkeit entwickelt werden.

Zur Präzisierung der Aufnahme von Knoten in den MCCS sei gesagt, dass diese nur bei vollständiger Übereinstimmung der lokalen Nachbarschaft erfolgt. So wird gemäß Abbildung 13 das Matching der blauen Knoten *(3,5)*, da Knoten 5 eine ausgehende Kante fehlt, und das Matching *(6,3)*, da Knoten 3 eine eingehende Kante fehlt, nicht aufgenommen. Diese strikte Vollständigkeitsbedingung ergibt sich aus der Tatsache, dass sonst die Gefahr besteht, Matchings aufzunehmen, bei denen Vorwärts- und Rückwärtskanten nicht korrekt erfasst werden, wodurch ein unvollständiger MCCS entstehen kann.

3.5 Entwurf einer Bewertungsgrundlage für teilkorrekte Ablaufpläne

Die Ermittlung des MCCS führt im Idealfall zur Feststellung der vollständigen Übereinstimmung von CFG und FC, was die Beantwortung der Frage, ob PAP und Programm übereinstimmen, mit „ja“ und einer festzulegenden maximalen Bewertung beendet. Anschließend muss nun jedoch geklärt werden, wie eine unvollständige Übereinstimmung zu behandeln ist, um eine Bewertung der Größenordnung der Abweichung zu ermöglichen.

Ein naiver Ansatz wäre es, den Jaccard-Index (siehe Kapitel 2.2.4) beider Graphen zu ermitteln, indem die Knoten und Kanten des MCCS ins Verhältnis zur Gesamtanzahl der Knoten und Kanten in beiden Graphen gesetzt werden. Die Möglichkeit, einen solchen Jaccard-Index zu berechnen, wird in (18) beschrieben.

$$J_V = \frac{|V_{MCCS}|}{|V_{CFG} \cup V_{FC}|} \qquad J_E = \frac{|E_{MCCS}|}{|E_{CFG} \cup E_{FC}|} \qquad J_{Gesamt} = \frac{J_V + J_E}{2} \tag{18}$$

Dabei ergibt sich ein gemeinsamer Jaccard-Index auf Basis von Knoten und Kanten, der je nach Bedarf in verschiedene Bewertungsskalen (z.B. Prozentsatz, Universitätsnoten usw.) für die Bestimmung der Ähnlichkeit von PAP und Programm umgerechnet werden kann. Dieser Ansatz berücksichtigt nicht die Struktur der Knoten und Kanten außerhalb des MCCS, was eine verzerrte Einschätzung zur Folge hat. Da für die Aufnahme von Knoten in den MCCS eine vollständige Übereinstimmung der Nachbarschaft erforderlich ist, würden partielle Übereinstimmungen bei der Bewertung ignoriert, was zu einer Unterbewertung der tatsächlichen Ähnlichkeit führt.

Um die Struktur der Knoten und Kanten außerhalb des MCCS angemessen in die Bewertung einzubeziehen, wird ein alternativer Ansatz verfolgt. In diesem Ansatz werden sowohl CFG als auch FC um die Knoten und Kanten im MCCS bereinigt, diese werden im Rahmen der Bewer–tung als korrekte Übereinstimmungen betrachtet, und eine Ähnlichkeitsberechnung der Rest-Knoten und ihrer existierenden Nachbarschaft, bestehend aus den direkten Vorgängern und Nachfolgern, anhand der Graph Edit Distance durchgeführt. Das Vorgehen wird in Abbildung 14 gezeigt.

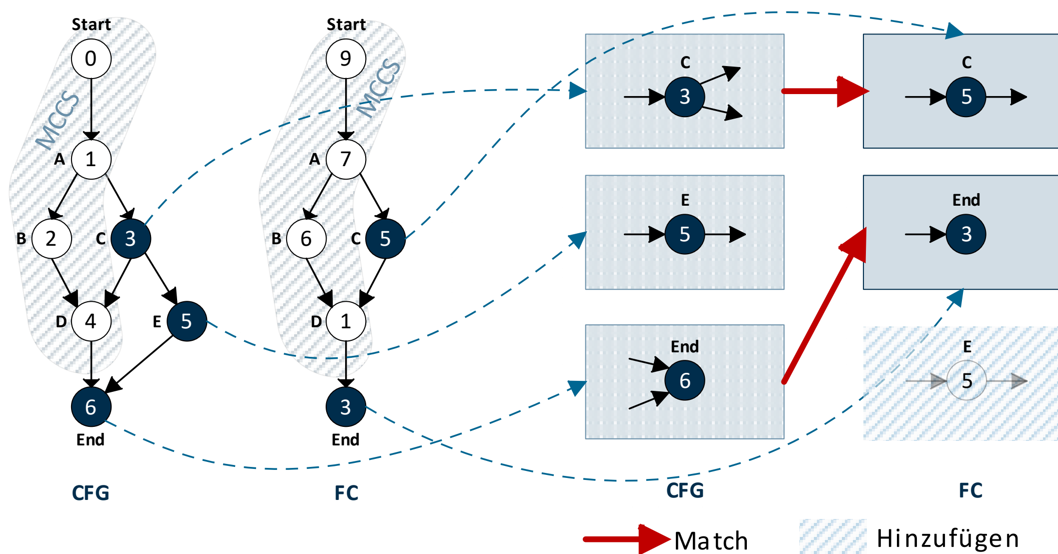


Abbildung 14: GED Ermittlung anhand der Rest-Knoten

Beim Abgleich der Knoten des CFG mit denen des FC wird der FC so angepasst, dass ein isomorpher Graph zum CFG als Resultatgraph entsteht und dabei die Transformationskosten erfasst werden, welche im Anschluss in eine Bewertung der Ähnlichkeit der Graphen einfließen. Das grundlegende Vorgehen in diesem Ansatz umfasst eine Prüfung der Ähnlichkeit, die im Vergleich zur Bildung des MCCS in eine Prüfung des Knotenlabels und eine Prüfung der Nachbarschaft unterteilt ist.

Zunächst wird geprüft, ob ein Matching anhand des Knotenlabels erzeugt werden kann. Im positiven Fall wird die Nachbarschaft des Knotens im FC auf fehlende Kanten untersucht, und für jede fehlende Kante werden Editkosten für das Hinzufügen dieser Kante zu den Transformationskosten addiert.

Sollte kein Matching auf Basis des Labels gefunden werden, wird nach einem Knoten mit der gleichen Nachbarschaft gesucht. Wird ein solcher Knoten gefunden und ist kein anderer Knoten des CFG darauf abbildbar, erfolgt ein Relabeling, bei dem die Editkosten für das Ersetzen erfasst werden.

Falls weder ein Knoten mit passendem Knotenlabel noch ein Knoten mit einer identischen Nachbarschaft gefunden wird oder ein möglicher Knotenkandidat von einem anderen Knoten im CFG besser abgebildet werden kann, wie in Abbildung 14 anhand des Matchings (3→5) statt (5→5) zu erkennen, wird der Knoten des CFG zusammen mit seiner Nachbarschaft im FC eingefügt, wobei die Editkosten für das Hinzufügen des Knotens und seiner Kanten berücksichtigt werden.

Anschließend erfolgt eine Gegenprüfung der Knoten des FC gegen die Knoten des CFG, wobei das gleiche Vorgehen angewendet wird. Der Unterschied besteht darin, dass bei fehlendem Matching die Knoten und Kanten im FC gelöscht und die Editkosten dafür zu den Transfor–mationskosten addiert werden.

Die im Rahmen dieses Ansatzes verwendeten Editkosten für das Hinzufügen und Löschen von Knoten und Kanten sowie für das Ersetzen von Knoten müssen dem Algorithmus als Parameter übergeben werden, um eine flexible Anwendung unterschiedlicher Bewertungsgrundlagen und -maßstäbe bei der Ermittlung der Ähnlichkeit von PAP und Programm zu ermöglichen.

Die im Zuge der Ausführung ermittelten Transformationskosten müssen im Kontext der Bewertung einer studentischen Leistung als Abweichung bzw. Fehler im PAP im Vergleich zum Programm interpretiert werden.

Da die Aufgabenstellung keine Anforderungen an ein Bewertungssystem zur Beurteilung der studentischen Leistung stellt, erfolgt in diesem Ansatz keine Entwicklung einer Berechnungs–vorschrift für eine universitäre Note. Stattdessen wird lediglich eine rudimentäre Berechnung auf der einfachsten Grundlage gemäß (19) mit maximaler Punktzahl P_{max}, erhaltener Punktzahl P_{erh} und Transformationskosten c innerhalb der prototypischen Konsolenanwendung durchgeführt.

$$P_{max} = 2 * |V_{CFG}| + |E_{CFG}| \qquad\qquad P_{erh} = P_{max} - c \tag{19}$$

Der CFG bildet dabei die Referenz für die Bewertung, da er direkt aus dem Programm abgeleitet wird und ein geringes Fehlerpotenzial in Bezug auf die Kontrollflüsse aufweist. Es wird angenommen, dass ein korrekter Knoten bzw. eine korrekt ausgefüllte Kontrollstruktur aufgrund ihres Vorkommens im PAP und der enthaltenen Anweisung mit zwei Punkten bewertet wird, während eine korrekte Kante mit einem Punkt in die Berechnung der maximalen Punktzahl einfließt. Bei einem vollständigen Matching ergibt sich $c = 0$, was zu einer Bewertung mit der maximalen Punktzahl führt.

3.6 Entwicklung einer Vergleichsmöglichkeit von Knoteninhalten

Ein wesentlicher Bestandteil bei der Ermittlung der Ähnlichkeit ist der Vergleich der in den Programmablaufplänen enthaltenen mit den automatisch aus dem Kontrollflussgraphen des Programms den Knoten zugewiesenen Anweisungen. Dieser Vergleich ist jedoch nicht trivial. Die Knotenlabels im CFG werden automatisch aus dem Programmcode erzeugt und enthalten die exakten Anweisungen. Im Gegensatz dazu können die Knotenlabels im FC, bedingt durch ihre manuelle Erstellung, sowohl kleinere Abweichungen wie Rechtschreibfehler oder Sonderzeichen als auch größere Abweichungen aufweisen, die durch unterschiedliche Abstraktionsgrade im Vergleich zum Programmcode entstehen. Ein strikter Vergleich basierend auf exakten Zeichenketten wäre daher zu restriktiv und würde die Ermittlung der Ähnlichkeit erheblich einschränken. Daher muss ein Konzept zum Vergleich der Knoteninhalte eine unscharfe Übereinstimmung ermöglichen.

In der Literatur gibt es zahlreiche Metriken zur Berechnung von Ähnlichkeiten, die verschiedene Analysebereiche abdecken. So sind beispielsweise die Levenshtein-Distanz und die Hamming-Distanz, welche die Anzahl der Positionen misst, an denen sich Zeichen zweier gleichlanger Strings unterscheiden, für den Vergleich von Zeichenketten relevant. Im Gegensatz dazu ist beispielsweise die Tree Edit Distanz zur Berechnung der Distanz zweier abstrakter Syntaxbäume für die Ermittlung der strukturellen Ähnlichkeit besonders geeignet. Ein an dieser Stelle entwickelter Ansatz, geht über die Grenzen einer spezifischen Analyse hinaus und integriert in einer multidimensionalen Metrik verschiedene Aspekte des Vergleichs. Diese Metrik soll den Anforderungen der Aufgabenstellung gerecht werden und berücksichtigen, dass die Knoten–inhalte des FC sowohl zwischen reinen Zeichenketten bei hohem Abstraktionsgrad als auch Anweisungen eines Programms bei geringem Abstraktionsgrad variieren. Abbildung 15 veran–schaulicht den Aufbau dieser Metrik, der im Folgenden erläutert wird.

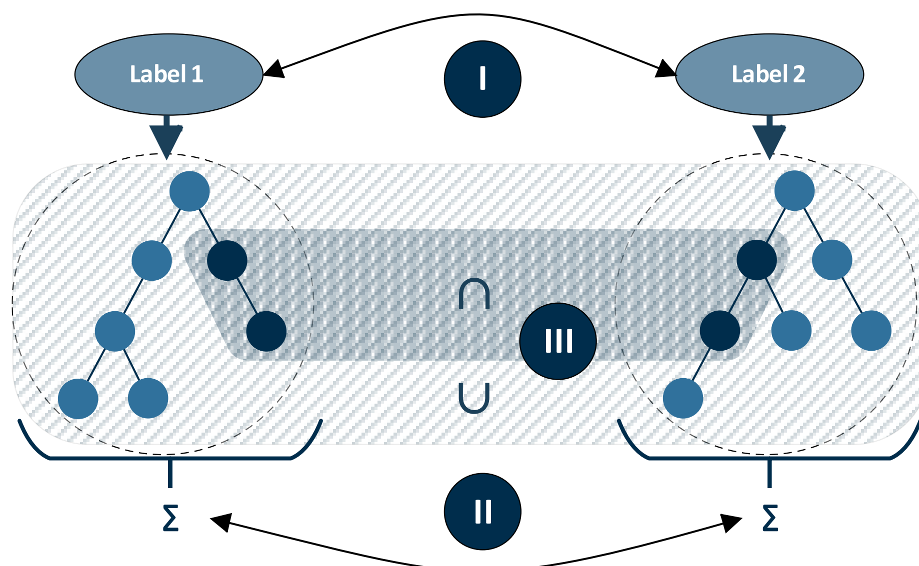


Abbildung 15: Bestandteile der Metrik zum Labelvergleich

Die Metrik setzt sich aus den Bestandteilen (I) bis (III) zusammen, wobei (I) die Ähnlichkeit der Knotenlabels als Zeichenketten und (II) in Verbindung mit (III) die strukturelle Ähnlichkeit der abstrakten Syntaxbäume der Knotenlabels als Anweisungen erfasst.

In der Analyse der Knotenlabels als Zeichenketten L_{CFG} und L_{FC} wird in Bestandteil (I) eine normalisierte String-Ähnlichkeit $S_{literal}$ basierend auf der Levenshtein-Distanz $dist$ im Verhältnis zum längsten der beteiligten Strings als oberer Schranke, wie in (20) dargestellt, ermittelt.

Abb. 15 (I)
$$S_{literal} = \left(1 - \frac{dist(L_{CFG}, L_{FC})}{\max(|L_{CFG}|, |L_{FC}|)}\right) * 100 \quad (20)$$

Ziel dieser Berechnung ist es, die Übereinstimmung der textuellen Beschreibung der Knotenlabels zu quantifizieren, wobei auch kleine Abweichungen, wie beispielsweise Recht–schreibfehler, berücksichtigt werden. Diese Berücksichtigung ermöglicht eine realistische Bewertung der Ähnlichkeit und verhindert, dass solche Abweichungen die gesamte Ähnlichkeitsberechnung verzerren. Die String-Ähnlichkeit ist besonders dann relevant, wenn eine direkte Behandlung von Knotenlabels des FC als Anweisung aufgrund eines hohen Abstraktionsgrads erschwert ist.

Die strukturelle Ähnlichkeit der abstrakten Syntaxbäume umfasst sowohl syntaktische als auch semantische Aspekte. In Bestandteil (II) wird die syntaktische Ähnlichkeit S_{syn} durch den Vergleich der Anzahl der Token T_{CFG} und T_{FC} in beiden abstrakten Syntaxbäumen ermittelt. Dies erfolgt gemäß (21) durch eine normierte Berechnung des Durchschnitts der Tokenanzahl beider Bäume.

Abb. 15 (II)
$$S_{syn} = \left(\frac{|T_{CFG}| + |T_{FC}|}{2 * \max(|T_{CFG}, T_{FC}|)}\right) * 100 \quad (21)$$

Die syntaktische Ähnlichkeit gibt Aufschluss darüber, ob eine grundsätzliche Übereinstimmung in der Struktur der abstrakten Syntaxbäume vorliegt und lässt darauf schließen, ob die Anwei–sungen aufgrund gleicher Anordnung und Organisation in beiden Bäumen ähnlich sind.

Als Ergänzung dazu erfolgt in Bestandteil (III) die Bestimmung der semantischen Ähnlichkeit S_{sem}. Diese wird mittels Jaccard-Index über die Anzahl der gemeinsamen Token in beiden Bäumen im Verhältnis zur Gesamtzahl aller einmaligen Token beider Bäume gemäß (22) ermittelt.

Abb. 15 (III)
$$S_{sem} = \left(\frac{|T_{CFG} \cap T_{FC}|}{|T_{CFG} \cup T_{FC}|}\right) * 100 \quad (22)$$

Sie unterstützt die in der syntaktischen Analyse getroffenen Aussagen zur Ähnlichkeit dadurch, dass die Token beider abstrakter Syntaxbäume auch inhaltlich auf Übereinstimmung geprüft werden. Die syntaktische und die semantische Ähnlichkeit in Verbindung mit der Gewichtung w_{qq} werden in diesem Ansatz zu einer gemeinsamen strukturellen Ähnlichkeit S_{struct} gemäß (23) zusammengeführt. Die Gewichtung ermöglicht eine flexible Anpassung der Ähnlichkeits–bestimmung, sodass bei der Nutzung der .NET-Bibliothek frei gewählt werden kann, ob die Syntax oder die Semantik stärker in die Berechnung der Gesamtähnlichkeit einfließen soll.

$$S_{struct} = S_{sem} * w_{qq} + S_{syn} * (1 - w_{qq}) \tag{23}$$

Im abschließenden Schritt werden die String-Ähnlichkeit sowie die strukturelle Ähnlichkeit zur Gesamtähnlichkeit S_{total} zusammengeführt, wie in (24) dargestellt. Auch hier besteht die Möglichkeit, über eine Gewichtung $w_{literal}$ Einfluss auf die Gesamtbewertung zu nehmen.

$$S_{total} = S_{literal} * w_{literal} + S_{struct} * (1 - w_{literal}) \tag{24}$$

Die Möglichkeit, die Gewichtung der String-Ähnlichkeit und der strukturellen Ähnlichkeit zu modifizieren, erlaubt es, die Gesamtbewertung flexibel an die Inhalte der Knotenlabels des FC anzupassen. Bei sehr abstrakten Beschreibungen innerhalb der Kontrollstrukturen des PAP kann die String-Ähnlichkeit stärker gewichtet werden, während bei detaillierteren Programm–anweisungen die strukturelle Ähnlichkeit stärker berücksichtigt wird.

In diesem Ansatz wird der Anforderung einer unscharfen Übereinstimmung dahingehend nachgegangen, als dass die Gesamtbewertung in Folge jedes Vergleichs von Knotenlabeln berechnet und gegen einen parametrisierten, frei wählbaren Schwellenwert EQT geprüft wird, wie in Abbildung 25 ersichtlich.

$$S_{total} \geq EQT * 100 \tag{25}$$

Dadurch wird erneut eine Flexibilisierung in der Schärfe des Vergleichs und damit einer Anpassung an verschiedene Beschreibungen von PAP Kontrollstrukturen erzeugt.

Die Berechnung der Metrik für die Gesamtähnlichkeit wird während der Vergleiche der Knotenlabels in den Komponenten zur Bestimmung des MCCS sowie bei der Berechnung der Graph Edit Distanz häufig durchgeführt. Da diese Berechnungen für die Gesamtähnlichkeit von großer Bedeutung sind, ist es entscheidend, den Vergleich der Knotenlabels auf eine möglichst effiziente Weise durchzuführen. Aus diesem Grund verwendet der Ansatz bewusst einfache Eigenschaften zur Bestimmung der Ähnlichkeit von Knotenlabels, um die Komplexität der Berechnungen zu reduzieren und die Effizienz zu steigern.

4 Umsetzung

4.1 Aufbau des Lösungsansatzes und Grundfunktionalitäten

Nach der erfolgreichen Konzeption der relevanten Bestandteile des Lösungsansatzes folgt nun die praktische Umsetzung. Diese wird durch die Erstellung von drei Visual Studio Projekten realisiert, die jeweils einen Aufgabenteil bei der Umsetzung der Lösung übernehmen.

Das erste Projekt beinhaltet die für die Implementierung der Graphenvergleichsfunktionen zuständige NET-Bibliothek. Sie stellt die grundlegenden Datenstrukturen für die interne Darstellung der Kontrollflussgraphen zur Verfügung und umfasst in statischen Klassen und Methoden alle Algorithmen, die für den Lösungsansatz benötigt werden. Die Verwendung eines statischen Lösungsansatzes ist damit begründet, durch direkte und schnelle Zugriffe die erforderlichen Funktionen verfügbar zu machen, ohne den zusätzlichen Aufwand für die Instanziierung und den damit verbundenen Speicherverbrauch zu verursachen. Obwohl die Implementierung in statischen Klassen und Methoden erfolgt, ist die Bibliothek flexibel und kann problemlos in anderen Anwendungen eingesetzt werden.

Das zweite Projekt ist eine Konsolenanwendung, die zu prototypischen Zwecken entwickelt wird und die in der .NET-Bibliothek implementierten Funktionen in einer praktischen Anwendung bereitstellt. Über diese Anwendung können Nutzer ein C-Programm sowie den zugehörigen Programmablaufplan in Form einer XML-Datei einlesen. Mithilfe der Vergleichsfunktionen wird dann der Abgleich zwischen C-Programm und PAP durchgeführt, und auf dessen Grundlage eine einfache Bewertung abgegeben. Die Anwendung dient als Schnittstelle zur Bibliothek und ermöglicht es, die Funktionalität der Algorithmen über eine sichtbare Ausgabe in der Konsole zu überprüfen.

Das dritte Projekt ist eine Testsuite, die mit xUnit als Test-Framework arbeitet. Mit ihrer Hilfe werden Unit-Tests geschrieben, um die Korrektheit der implementierten Graphenvergleichs–funktionen zu überprüfen. Damit soll sichergestellt werden, dass die Anwendung wie gewünscht funktioniert. Die Testsuite soll dazu beitragen auf einfache und schnelle Weise, die Qualität der Implementierung zu verbessern und mögliche Fehler zu identifizieren.

Alle drei Projekte werden in der Entwicklungsumgebung Visual Studio 2022 erstellt und mit .NET 8.0 (der aktuellen LTS-Version) sowie C# 12 umgesetzt. Zur Erweiterung der Funktionalität werden die Bibliotheken *Microsoft.Extensions*, für die Implementierung einer JSON-Konfi–gurationsmöglichkeit, sowohl für die Bibliothek als auch die prototypische Anwendung, und *Microsoft.CodeAnalysis*, für die Entwicklung des Abgleichs von Knotenlabels mit Roslyn, der Compiler-API im .NET-Umfeld für die Erstellung der abstrakten Syntaxbäume, eingebunden. Außerdem wird das xUnit-Paket in der Version 2.5.3 eingebunden, um die Erstellung und Ausführung von Unit-Tests im Rahmen des dritten Projekts, möglich zu machen.

Im Folgenden wird eine kurze Vorstellung der beiden Klassen und ihrer zugrunde liegenden Datenstrukturen gegeben, welche das Fundament für die Entwicklung dieses Lösungsansatzes bilden und in allen Komponenten zur Anwendung kommen.

Die Klasse *Node* und ihre Instanzen bilden die grundlegende Repräsentation der Knoten im internen Kontrollflussgraphen. Die zugehörigen Attribute und Datenstrukturen sind in Listing 2 dargestellt.

```
1. public class Node : IComparable<Node> {
2.     public int Id { get; set; }
3.     public int InDegree { get; private set; }
4.     public int OutDegree { get; private set; }
5.     private readonly List<string> Label = [];
6.     private List<Node> Predecessors = [];
7.     private List<Node> Successors = [];
```

Listing 2: Knoten-Klasse und dazugehörige Attribute

Die *Id* dient ausschließlich der Identifikation innerhalb des zugeordneten Graphen und hat in den Vergleichen mit anderen Graphen keine Bedeutung. Für den Vergleich werden stattdessen die Knotenlabels sowie die Vorgänger- und Nachfolgerknoten verwendet, die als Attribute in den Listen-Datenstrukturen *Label*, *Predecessors* und *Successors* gespeichert sind. Die Attribute *InDegree* und *OutDegree*, welche bei Veränderungen der Nachbarschaft automatisch angepasst werden, bieten eine schnell zugängliche, oberflächliche Bewertung der Nachbarschaft, die bei der Suche nach Quellen, Senken oder sequenziellen Folgen im Graphen hilfreich ist.

Die Klasse *Graph*, wie in Listing 3 gezeigt, verwendet eine *AdjacencyList* in Form einer Key-Value-basierten Liste, die in C# als Dictionary bezeichnet wird. In dieser Struktur repräsentiert der Key die interne Knoten-Id, während der Value das zugehörige Knotenobjekt selbst enthält.

```
1. public class Graph {
2.     public string Description { get; set; } = "";
3.     public int NodeCount { get; private set; }
4.     public int EdgeCount { get; private set; }
5.     private Dictionary<int, Node> AdjacencyList = [];
```

Listing 3: Interne Graphen-Klasse mit Attributen

Die *AdjacencyList* stellt eine Erweiterung der klassischen Adjazenzliste dar. Durch direkten Zugriff auf die Knoten können sowohl Vorgänger als auch Nachfolger eines Knotens effizient abgefragt werden, was die Analyse der Nachbarschaft innerhalb des Graphen erleichtert. Anstelle einer separaten Klasse für Kanten, werden diese durch die Beziehungen zwischen den Knoten abgebildet. Dieser Ansatz reduziert den Speicherbedarf und den Aufwand bei der Verwaltung der Datenstrukturen. Die Attribute *NodeCount* und *EdgeCount* geben die Anzahl der Knoten und Kanten im Graphen an und ermöglichen eine einfache Analyse der Graphenstruktur. Die *Description* beinhaltet eine optionale Beschreibung des Graphen, um eine Identifikation im Vergleich zu anderen Graphen zu ermöglichen.

Abschließend werden Funktionalitäten vorgestellt, die für die Nutzung der Bibliothek und der .NET-Anwendung von Bedeutung sind. Aufgrund ihres Einflusses auf alle Entwicklungskompo–nenten können sie nicht direkt einer einzelnen Komponente zugeordnet werden und werden daher an dieser Stelle gesondert behandelt.

Die Komponenten zur Erstellung des CFG, zur Berechnung der Graph Edit Distance sowie die Funktion für den Knotenlabel-Vergleich benötigen eine Reihe von Parametern, die nicht manuell

eingegeben werden müssen. Diese werden stattdessen über die Klasse *Configuration* der Biblio– thek bereitgestellt. Eine Beschreibung der Klasse ist in Listing 4 zu finden.

```
1. public static class Configuration {
2.
3.     internal static IConfiguration config = new
4.                         ConfigurationBuilder()
5.                         .AddJsonFile("CfgCompLibConfig.json").Build();
6.
7.     public static double NodeInsertCost { get; set; } =
8.             config.GetRequiredSection("GraphEditCosts")
9.                 .GetValue<double>("NodeInsert");
```

Listing 4: Klasse zur Konfiguration der Bibliothek (gekürzt)

In dieser Klasse werden mithilfe der *Microsoft.Extensions* Bibliothek Konfigurationsdaten aus einer JSON-Datei geladen und für die interne Verwendung bereitgestellt, wie in Zeile 5 gezeigt. Über das *IConfiguration* Interface und den *ConfigurationBuilder* kann auf eine hierarchische Struktur von Sektionen zugegriffen werden (siehe Zeile 8), um Einzelparameter in der JSON-Datei abzurufen. Zudem wird eine Typumwandlung in den erforderlichen Datentyp durch– geführt, bevor der Parameter intern genutzt werden kann (siehe Zeile 9).

Für die Bibliothek steht eine entsprechende Konfigurationsdatei zur Verfügung, deren Inhalt in Listing 5 gezeigt wird. In dieser Datei werden in der Sektion *Settings* die Parameter für die Erstellung des CFG und FC, in *GraphEditCosts* die Editkosten für die GED-Berechnung und in *LabelComparison* die Gewichtungen der Metrik definiert. Die Verwendung der Konfigurations– datei erleichtert die Nutzung der Bibliothek erheblich, da die Parameter dauerhaft gespeichert werden und ein erneutes Anpassen bei jedem Programmstart entfällt. Zudem können verschiedene Parameter-Sets für unterschiedliche Anwendungsfälle erstellt werden, was eine hohe Flexibilität bietet.

```
1. {  "Settings": {
2.      "CompilerOptimization": "-O1",
3.      "CompilerDir": "C:\\Program Files\\MinGW\\bin",
4.      "OutputDir": "C:\\Folder\\Output",
5.      "XSDPath": "C:\\Folder\\FlowChart.xsd"
6.    },
7.    "GraphEditCosts": {
8.      "NodeInsert":  2.0,
9.      "NodeDelete":  1.0,
10.     "NodeRelabel": 1.0,
11.     "EdgeInsert":  1.0,
12.     "EdgeDelete":  1.0
13.   },
14.   "LabelComparison": {
15.     "QualtoQuantWeight":    0.5,
16.     "LiteralWeight":        0.5,
17.     "EqualityThreshold":    0.8  }}
```

Listing 5: Konfigurationsdatei der Bibliothek im JSON-Format

Eine einzige Konfigurationsdatei für sowohl die Bibliothek als auch die Anwendung würde dem Ziel widersprechen, diese beiden Komponenten unabhängig voneinander nutzbar zu machen. Deswegen wird analog dazu in der Konsolenanwendung die Klasse *Settings* implementiert, die die Konfiguration der Standard-Eingabepfade für die auszuwertenden C-Programme und XML-

Dateien des PAP sowie den Ausgabepfad für erstellte visuelle Repräsentationen im DOT-Format ermöglicht. Dies verbessert die Usability, da beim Programmstart nur der Programmname und die XML-Datei als Parameter eingegeben werden müssen.

Eine zentrale Funktionalität im Zusammenhang mit einer einheitlichen Graphenstruktur ist die Methode *ExpandToMaxGraph*, deren Arbeitsweise in Listing 6 dargestellt wird. Gemäß der Konzeption, die in den Basisblöcken von CFG und FC nur einzelne Anweisungen vorsieht, ermöglicht diese Methode die Aufteilung von Knoten, die mehrere Anweisungen enthalten.

```
1.  public static Graph ExpandToMaxGraph(Graph graph) {
2.      Graph expGraph = DeepCopy(graph);
3.      int maxId = expGraph.GetHighestId();
4.      foreach (var node in expGraph.GetNodes().Values) {
5.          Node expandedNode = node;
6.          while (node.GetLabel().Count > 1) {
7.              expandedNode = new(++maxId, [node.GetExpressionAtRow(1)],
8.                                 [expandedNode], expandedNode.GetSuccessors());
9.              node.RemoveExpression(1);
10.             expGraph.AddSequenceNode(expandedNode);}}
11.     return expGraph; }
```

Listing 6: Erweiterung von Graphen auf eine Anweisung pro Block

In dieser Methode wird die Anzahl der Labelinhalte gezählt und innerhalb der Schleife in Zeile 6 werden für Knoten mit mehreren Anweisungen im Label neue Knoten durch Abspaltung von einzelnen Anweisungen erzeugt. Die neu entstandenen Knoten werden nach ihrer Erstellung sequenziell aneinandergereiht in den Graphen eingefügt. Die Methode *AddSequenceNode* in Zeile 10 stellt dabei sicher, dass die Beziehungen der Vorgänger und Nachfolger nach dem Hinzufügen von Knoten wieder richtig hergestellt werden. Die Methode ist somit wichtig in der Vorbereitung der Graphen auf die Anwendung der Vergleichsfunktionen. Die Implementierung dieser erfolgt in den nächsten Abschnitten anhand der ausgearbeiteten Konzeptionen.

4.2 Erstellung interner Graphenrepräsentation aus Compiler-Daten

Die Implementierung der ersten Komponente erfolgt gemäß Konzeption in zwei Schritten. Zuerst wird ein neuer Systemprozess, der die Ausführung der CFG-Erzeugung mittels GCC-Compiler durchführt, angelegt. Im Anschluss wird ein Parser implementiert, welcher den CFG in die interne Graphenrepräsentation der Bibliothek überführt.

Im ersten Schritt wird innerhalb der Bibliothek eine Methode erstellt, die die Prozessausführung vorbereitet und startet. Zunächst erfolgt die Konfiguration des Prozesses als *ProcessStartInfo*, wobei die in Listing 7 angegebenen Parameter verwendet werden.

```
1.  ProcessStartInfo psiGcc = new("gcc.exe") {
2.      WorkingDirectory = compilerPath,
3.      UseShellExecute = false,
4.      CreateNoWindow = true,
5.      RedirectStandardOutput = true,          //Output -> Anwendung
6.      RedirectStandardError = true,
7.      ArgumentList = {
8.          $"{compilerOpt}",                   //Optimierungslevel
9.          "-w",
10.         "-fno-builtin",                     //Unterdrückung Builtins
11.         "-fdump-tree-cfg=stdout",           //cfg -> Output
12.         $"{cFilePath}",
13.         "-o",
14.         $"{outputPath}\\compileDump.bin" }}; //Pfad für ausführbare Datei
```

Listing 7: Prozess mit Compiler-Ausführung zur Erzeugung des CFG

Der Prozess wird gemäß dieser Konfiguration im Hintergrund ausgeführt, wobei seine Standard-Ausgabeströme aufgrund des Fehlens einer Shell-Ausführung in Verbindung mit den *Redirect*-Parametern in einen StreamReader umgeleitet werden, der innerhalb der Bibliothek abrufbar ist. In Kombination mit dem in der *ArgumentList*, der Parameterliste für den GCC-Compiler, angegebenen, modifizierten Parameter für die CFG-Erstellung (Zeile 11) [48] ermöglicht dies eine direkte Weiterleitung des CFG in die Bibliothek zur Weiterverarbeitung durch den Stream–Reader. Das Erstellen einer cfg-Datei im Dateisystem sowie deren Einlesen, wie im Konzept vorgesehen, entfällt damit, wodurch der Aufwand im zweiten Schritt verringert wird.

Darüber hinaus werden in der Konfiguration innerhalb der *ArgumentList* weitere Parameter für den GCC-Compiler eingebunden, darunter der Parameter *compilerOpt* für das Optimierungs–level der Kompilierung, *cFilePath*, der sich aus dem in der JSON-Datei angegebenen Ordnerpfad und dem von der Konsolenanwendung stammenden Dateinamen zusammensetzt, sowie *outputPath* für den Ordnerpfad der erstellten, ausführbaren Datei des C-Programms. Diese Parameter werden durch die JSON-Konfigurationsdatei von der Bibliothek bereitgestellt.

Das Optimierungslevel, das hauptsächlich die Stufen -O0 (keine Optimierung) bis -O3 (maximale Optimierung) umfasst, beschreibt den Aufwand, den GCC in Bezug auf die Verringerung der Laufzeit oder des Speicherplatzes des kompilierten Programms leisten soll. Dieser Ansatz sieht in der Konzeption keine Optimierung des CFG vor. Tests der erstellten internen CFG ergaben jedoch, dass ohne Optimierung Basisblöcke enthalten sind, die keine relevanten Informationen für die Auswertung der Kontrollflüsse liefern. Diese Basisblöcke sorgen dafür, dass Knoten im

CFG entstehen, die keinem Knoten im FC entsprechen können und so zu einer Verzerrung des Abgleichs führen. Daher wurde im Rahmen der Implementierung entschieden, zumindest die einfachste Optimierungsstufe *-O1* zu verwenden, wobei diese Einstellung über die JSON-Konfiguration jederzeit angepasst werden kann.

Der Parameter *-fno-builtin* für den GCC-Compiler stellt eine weitere Anpassung bei der Erstel–lung des CFG dar, indem die Verwendung interner GCC-Funktionen (Builtins) ausgeschlossen wird [49]. Builtins sind Funktionen, die maschinenoptimierte Varianten existierender Biblio–theksfunktionen darstellen und diese beim Kompilieren zur Optimierung ersetzen [50]. Bei–spielsweise kann ein *printf*-Aufruf durch einen *__builtin_puts*-Aufruf ersetzt werden. Diese Ersetzung verzerrt die Darstellung als Anweisung im CFG, weshalb auf sie verzichtet wird. Nach erfolgter Konfiguration wird der Prozess mit der *ProcessStartInfo* als Parameter gestartet und der StreamReader, welcher das CFG als Zeichenkette enthält, entgegengenommen.

Im zweiten Schritt erfolgt die Weiterverarbeitung des als Zeichenkette bereitgestellten CFG zur internen Graphenrepräsentation. Dabei wird anhand eines StringReaders die Zeichenkette zeilenweise ausgelesen und mittels Mustersuche durch regulärer Ausdrücke (engl. regular expressions, regex) inhaltlich identifiziert und in die entsprechende interne Repräsentation umgewandelt.

Bei regulären Ausdrücken handelt es sich um eine Meta-Sprache, die es ermöglicht, Muster in Zeichenketten zu beschreiben, wobei sie spezielle Metazeichen verwendet, um bestimmte Zeichenfolgenkonstrukte formal auszudrücken und zu erkennen [3]. Reguläre Ausdrücke sind ein zentraler Bestandteil der Textverarbeitung und kommen daher auch in diesem Ansatz zur Anwendung. Eine Übersicht der im Rahmen der Implementierung verwendeten Metazeichen für die Programmiersprache C# ist in Tabelle 2 zu finden.

Tabelle 2: Übersicht einer Auswahl an Metazeichen für reguläre Ausdrücke in C# (zusammengestellt nach [51])

Zeichen	Bedeutung
.	ein beliebiges Zeichen (außer \n für Newline)
\w	ein beliebiges Wortzeichen (Ziffer 0 bis 9, Buchstabe a bis z oder A bis Z)
\d	eine beliebige Ziffer von 0 bis 9
\s	ein Leerraumzeichen (Leerzeichen, Tabulator)
[*Gruppe*]	Zeichengruppe – Match bei einem/mehreren Zeichen der Gruppe
[^ *Gruppe*]	Zeichengruppe – Match, wenn Zeichen der Gruppe **nicht** vorkommen
(*Gruppe*)	Gruppierung von Zeichen, die sich mit .Group[*Zahl*] aus Match-Array auslesen lassen
?	Quantifizierer – vorheriger Ausdruck kommt null oder einmal vor
*	Quantifizierer – vorheriger Ausdruck kommt null oder mehrfach vor
+	Quantifizierer – vorheriger Ausdruck kommt ein oder mehrfach vor

Wichtig ist, dass in C# bei der Verwendung von regulären Ausdrücken in Strings Metazeichen durch das Escaping mit einem Backslash „\“ angezeigt werden müssen. Da der Backslash selbst ein Escape-Zeichen ist, wird er in C#-Strings durch einen weiteren Backslash maskiert. Daher werden in den folgenden Listings immer zwei Backslashs zusammen abgebildet.

Der Algorithmus beginnt, wie in der Konzeption beschrieben, mit der Erstellung der Start- und Endknoten und setzt die Verarbeitung anschließend mit der zeilenweisen Analyse der auf–

tretenden Muster in der Zeichenkette des StringReaders fort, um die interne Repräsentation des CFG zu generieren. Die Mustererkennung der Knoten und Kanten des CFG mittels regulärer Ausdrücke wird in Listing 8 veranschaulicht. Zur besseren Veranschaulichung und zum Vergleich kann Abbildung 10 auf Seite 31 herangezogen werden.

```
1.  //Erzeuge Knoten aus Compiler-Info vor Funktionsbeginn
2.  if (!inFunctionBody) {
3.      Match nodeNum = Regex.Match(line, ";;\\s+(\\d+)\\s+succs");
4.      if (nodeNum.Success) {
5.          Node node = new(int.Parse(nodeNum.Groups[1].Value));
6.          graph.AddNode(node);
7.  } /* gekürzt */
8.  //Prüfe Zeile auf "goto <bb ...>" und hole Blocknr. (= Nachfolger ID)"
9.  Match gotoNum = Regex.Match(line, "goto <bb\\s+(\\d+)>;");
10. if (gotoNum.Success) {
11.     int succId = int.Parse(gotoNum.Groups[1].Value);
12.     graph.AddEdge(graph.GetNode(actBlockNum), graph.GetNode(succId)); }
```

Listing 8: Erstellung von Knoten und Kanten (gekürzt)

Die Id der Knoten wird anhand des Musters von Compiler-Informationen, welche in den ersten Zeilen der Zeichenkette vorliegen, gemäß Zeile 3 mittels Gruppierung *(\\d+)* ausgelesen und zur Erzeugung eines internen Knotens gemäß Zeile 5 genutzt. Anschließend wird innerhalb der geparsten C-Funktion anhand des regulären Ausdrucks *"<bb\\s+(\\d+)>\\s+:"* der gerade durchlaufene Basisblock identifiziert und dessen Inhalt dem entsprechenden Knoten als Label hinzugefügt. Sollte gemäß Zeile 9 eine *goto*-Anweisung gefunden werden, so wird eine Kante zwischen dem Knoten des aktuell durchlaufenen Basisblocks und dem Knoten des mittels Gruppierung *(\\d+)* ausgelesenen Zielblocks, hergestellt. Nachdem der StringReader das Ende der Zeichenkette des CFG erreicht hat, endet der Algorithmus und die Erstellung der internen Repräsentation des CFG ist abgeschlossen. Für den bereits in der Konzeption erwähnten Rückgabewert der Funktion erfolgt eine separate Mustersuche gemäß Listing 9.

```
1.  //Lies den Return-Wert aus "D.01234 = (<type cast>) WERT;"
2.  Match returnValue = Regex.Match(line, "D.\\d+\\s+=\\s+\\(?[^)]*\\)?\\s?(.+);");
3.  if (returnValue.Success) {
4.      returnVal = returnValue.Groups[1].Value; } /* gekürzt */
```

Listing 9: Auslesen der temporären Return-Variable

Dabei wird im Verlauf der zeilenweisen Erstellung der Knoten und Kanten des internen Graphen jede Zeile auch auf das Vorkommen des Musters in Zeile 2 geprüft, welches den grundsätzlichen Aufbau der temporären Variable für den Rückgabewert entspricht. Sollte es zum Matching kommen, wird der Wert der temporären Variable mittels der Gruppierung *(.+)* ausgelesen und zwischengespeichert. Bevor der Algorithmus endet wird dem Knoten des letzten aktuellen Basisblocks die Return-Anweisung mit dem zwischengespeicherten Wert zugewiesen.

Der entwickelte Algorithmus konnte in anschließenden Tests die ggt-Funktion erfolgreich auswerten, den CFG korrekt erstellen und in die interne Graphenrepräsentation überführen. Weitere Tests mit Verbundausdrücken, die mehrere Einzelanweisungen bündeln, führten jedoch zu den in Listing 10 dargestellten Problemen.

Im C-Programm

```
1.  printf("compound expressions");
2.      var += 10;
3.      var -= 10;
4.      var *= 10;
5.      var /= 10;
6.      var %= 10;
```

In der internen Repräsentation

```
1.  printf ("compound expressions");
2.    var.4_9 = var;
3.    _10 = var.4_9 + 10;
4.    var = _10;
5.    var.5_11 = var;
6.    _12 = var.5_11 + -10;
7.    var = _12;
8.    var.6_13 = var;
9.    _14 = var.6_13 * 10;
```

Listing 10: Beispiel für temporäre Variablen und Static Single Assignment

Die Verwendung der gleichen Variablen *var* sowie die Nutzung von Verbundausdrücken im Programm führen während der Kompilierung zum Auftreten von Static Single Assignment (SSA) und temporären Ausdrücken im CFG. SSA ist eine Compilertechnik, bei der jeder Variablen nur einmal ein Wert zugewiesen werden kann und eine erneute Zuweisung die Erstellung einer neuen Variablen erfordert [44]. Diese Technik ermöglicht eine vereinfachte Analyse der Zustände von Variablen und unterstützt verschiedene Optimierungen, wie etwa die Eliminierung redundanter Berechnungen [52]. Temporäre Ausdrücke dienen der Speicherung von Teil–berechnungen und werden sowohl zur Eliminierung redundanter Berechnungen als auch zur Vereinfachung komplexer Berechnungen im Rahmen der Vorbereitung für die maschinen–optimierte Verarbeitung verwendet [44].

Um im Algorithmus die Verzerrungen des CFG durch beide Compilertechniken zu verhindern, wird gemäß Listing 11 wiederum durch Verwendung von regulären Ausdrücken eine Muster–suche auf die typische Struktur von SSA und temporären Ausdrücken durchgeführt.

```
1.  //Prüfe auf SSA oder temp. Var. (z.B. "var.1_2" oder "_1") und hole den Wert
2.  Match varMerge = Regex.Match(line, "(\\w*\\.?\\d*_\\d+)\\s+=\\s+(.+);");
3.  if (varMerge.Success) {
4.      regexMatches.Add(varMerge.Groups[1].Value, varMerge.Groups[2].Value);}
5.  /* gekürzt */
6.  //Ersetze Platzhalter-Variable mit Wert
7.  foreach (var regexMatch in regexMatches.OrderByDescending(rm => rm.Key)) {
8.      line = Regex.Replace(line, regexMatch.Key, regexMatch.Value);}
```

Listing 11: Rückführung von Static Single Assignments und temporären Variablen (gekürzt)

Dabei werden sowohl die Bezeichnung einer SSA-Variable oder temporären Variable mittels Gruppierung *(\\w*\\.?\\d*_\\d+)* als Key als auch die zugehörigen Werte mittels der Grup–pierung *(.+)* als Value in einem Dictionary gespeichert. Anschließend wird das Dictionary rückwärts durchlaufen und die Zeichenkette wird durch aufeinanderfolgende Ersetzungen der Keys durch ihre entsprechenden Values wieder in den Zustand vor der Optimierung zurück–geführt.

Eine abschließende Beurteilung der Lösung zur Integration des CFG in die interne Graphenrepräsentation zeigt, dass die im Konzept festgelegten Anforderungen erfüllt wurden. Die Erstellung eines CFG im GCC-Compiler und dessen Überführung zur Weiterverarbeitung gemäß der Aufgabenstellung kann für einfache CFGs erfolgreich durchgeführt werden.

Allerdings haben die aufgetretenen Komplikationen bei der Erweiterung auf komplexere Anweisungen gezeigt, dass die Optimierungen des GCC-Compilers den CFG erheblich verändern können. Infolgedessen muss die Lösung angepasst werden, um die durch den Compiler vorgenommenen Änderungen angemessen zu behandeln. Die durchgeführten Anpassungen zur Berücksichtigung der Compileränderungen im CFG sind jedoch begrenzt und können nicht den gesamten Umfang an Anweisungen und Datenstrukturen abdecken, die im CFG auftreten können. Rückblickend muss festgestellt werden, dass die Entscheidung, GCC als Compiler aufgrund leichterer Anweisungsauswertung zu bevorzugen, überdacht werden muss, da durch die Erweiterungen bezogen auf SSA und temporäre Variablen eine LLVM ähnliche Auswertung in GCC notwendig werden lässt und somit eine Umsetzung auch auf Basis von LLVM möglich sein kann.

4.3 Nutzung von XML basiertem Importformat zur Graph-Generierung

Gemäß dem Gesamtkonzept erfolgt im nächsten Schritt die Implementierung der zweiten Komponente, welche die Integration eines XML-Parsers in die .NET-Bibliothek umfasst. Dieser Parser überführt den in der Webanwendung draw.io erstellten Programmablaufplan in die interne Graphenrepräsentation.

Um die externe XML-Datei in der Bibliothek verfügbar zu machen, wird eine Methode bereit–gestellt, die mittels eines XMLReaders die Datei unter Verwendung der Parameter *xmlPath* und *settings* einliest. Bei *xmlPath* handelt es sich um eine Kombination des in der JSON-Konfiguration der .NET-Anwendung bereitgestellten Ordnerpfads und des Dateinamens, der beim Start der Anwendung als Eingabeparameter angegeben werden muss. Der Parameter *settings* enthält den Dateipfad zur XSD-Datei, gegen die die eingegebene XML-Datei validiert werden soll. Dieser Pfad ist in der JSON-Konfiguration der Bibliothek hinterlegt.

Für die Erzeugung der XSD-Dateien dieses Ansatzes wird das kostenlose Web-Tool Freeformatter verwendet, das nach dem Hochladen einer XML-Datei und der Angabe eines XSD-Designs automatisch eine XSD-Datei generiert [53]. Für das XSD-Design wird das Format „RussianDoll" gewählt, da es eine tief verschachtelte Struktur der XML-Elemente bietet, die der Hierarchie der XML-Datei von draw.io ähnelt.

Mit Hilfe des XMLReader werden die eingelesenen Daten in der XmlDocument-Klasse in eine Document Object Model Repräsentation überführt. Diese Repräsentation ermöglicht es, XML-Elemente als Objekte zu behandeln und so einen schnellen und strukturierten Zugriff auf einzelne Elemente zu erhalten. Dadurch wird es gemäß Listing 12 möglich, mit der Methode *SelectNodes* direkt XML-Elemente auszulesen, die mithilfe von XPath, einer Abfragesprache für XML, lokalisiert und gefiltert werden können. Im konkreten Fall wird in XPath mit *//** die Auswahl aller Elemente durchgeführt und mit dem Filter *[@vertex='1']* bzw. *[@edge='1']* diese auf die Elemente mit dem Attribut *vertex='1'* bzw. *edge='1'* beschränkt.

```
1. //Gehe aus von Root und hole alles mit Attribut "vertex=1"/"edge=1"
2. XmlNode root = document.DocumentElement;
3. XmlNodeList vertices = root.SelectNodes("//*[@vertex='1']");
4. XmlNodeList edges = root.SelectNodes("//*[@edge='1']");
```

Listing 12: Extraktion der XML-Datei

Auf diese Weise werden die Elemente, welche die Knoten und Kanten ergeben sollen, in den ge–trennten Listen *vertices* und *edges* erfasst und entsprechend Listing 13 zur Erzeugung von Knoten und Kanten durchlaufen. Wie konzeptionell festgelegt, erfolgt bei der Erstellung der Knoten eine Neuzuweisung der Id in Form einer rudimentären, inkrementell fortlaufenden Ganzzahl *graphId*, um eine bessere Lesbarkeit bei der Analyse der Knoten im FC zu ermöglichen. Damit diese aber bei der Erzeugung von Kanten korrekt berücksichtigt werden kann, muss über das Dictionary *idMapping* eine Rückverfolgbarkeit zwischen alter und neuer Id sichergestellt werden. Die Funktion *PrepareLabel* bereitet die aus dem Attribut *Value* gewonnenen Anwei–sungen für eine Übernahme als Knotenlabel vor und wird anschließend gesondert vorgestellt.

```
1.  //Mappe alte ID und erstelle Knoten mit Labelinhalt „value“
2.  foreach (XmlElement vertex in vertices) {
3.      idMapping.Add(vertex.GetAttribute("id"), graphId);
4.      graph.AddNode(new(graphId, PrepareLabel(vertex.GetAttribute("value"))));
5.      graphId++;
6.  };
7.  //Erstelle Kanten mit den neuen IDs aus dem XML->interne ID Mapping
8.  foreach (XmlElement edge in edges) { /* gekürzt */
9.      int sourceId = idMapping[edge.GetAttribute("source")];
10.     int targetId = idMapping[edge.GetAttribute("target")];
11.     graph.AddEdge(graph.GetNode(sourceId), graph.GetNode(targetId))}};
```

Listing 13: Erstellung des internen FC (gekürzt)

Nach dem Durchlauf beider Listen, liegt der interne FC zur weiteren Verwendung mit allen Knoten und Kanten vor und der Import der externen XML-Datei ist abgeschlossen.

In diesem Ansatz wird eine XML-Datei anhand ihrer Attribute ausgewertet. Dadurch ist es möglich, die draw.io-Tags durch eigene Tags zu ersetzen, solange die Attribute *vertex* und *value* für Knoten bzw. *source*, *target* und *edge* für Kanten erhalten bleiben. Dies ermöglicht die flexible Erstellung eigener XML-Formate, wie beispielsweise dem in Listing 14 vorgestellten, und eine Unabhängigkeit von dem von draw.io bereitgestellten XML-Format, wie im Konzept gefordert.

```
1.  <graph>
2.      <vertex id="0"  value="Start"       vertex="1"/>
3.      <vertex id="1"  value="End"         vertex="1"/>
4.      <vertex id="2"  value="y!=0"        vertex="1"/>
5.      <vertex id="5"  value="x!=y"        vertex="1"/>
6.      <vertex id="8"  value="return x"    vertex="1"/>
7.      <vertex id="12" value="x&gt;y"      vertex="1"/>
8.      <edge   id="4"  edge="1"      source="0"  target="2"/>
9.      <edge   id="7"  edge="1"      source="2"  target="5" />
10.     <edge   id="9"  edge="1"      source="2"  target="8" />
11. </graph>
```

Listing 14: Erstellung einer eigenen Graph XML (gekürzt)

In einem eigenen XML-Format kann die Lesbarkeit weiter verbessert werden, indem auf unnötige Tags, wie den Tags für die Bestimmung der Geometrie, verzichtet wird und die Bezeichnungen direkt ihrer Entsprechung im Graph gewählt werden. Die Validierung eines solchen XML-Formats erfordert zudem eine neue, eigene XML-Schema-Definition, wie sie in Listing 15 veranschaulicht wird.

```
1.    <xs:sequence>
2.      <xs:element name="vertex" maxOccurs="unbounded" minOccurs="0">*gekürzt*</>
3.      <xs:element name="edge"   maxOccurs="unbounded" minOccurs="0">
4.      <xs:complexType mixed="true">
5.        <xs:attribute type="xs:string" name="id"     use="required"/>
6.        <xs:attribute type="xs:string" name="source" use="required"/>
7.        <xs:attribute type="xs:string" name="target" use="required"/>
8.        <xs:attribute type="xs:byte"   name="edge"   use="required"/>
9.      </xs:complexType>
10.     </xs:element>
11.   </xs:sequence>
```

Listing 15: Erstellung eigener XML-Schema-Definition (gekürzt)

Durch die Möglichkeit die zu verwendende XSD in der JSON-Konfiguration festzulegen, kann beliebig zwischen draw.io-Format und eigenem Format gewechselt werden. Das Ziel dieses Ansatzes bleibt aber weiterhin die Nutzung der Vorteile einer visuellen Repräsentation des PAPs bei der Erstellung mittels draw.io.

Nach der Implementierung des XML-Parsers und des eigenen XML-Formats wurden Tests durchgeführt, um verschiedene in draw.io erstellte visuelle Gestaltungselemente mit unterschiedlicher Formatierung zu verarbeiten und daraus einen internen Graphen zu erstellen. Dabei wurde beim Auftreten von Vergleichsoperatoren in den Anweisungen festgestellt, dass diese HTML-kodiert sind, um Überschneidungen mit den Tags zu vermeiden. Zudem führt die Beschriftung von Gestaltungselementen mit mehrzeiligen Anweisungen oder Textformatierungen dazu, dass HTML-Tags in den Knotenlabels enthalten sind, was deren Verwendung im Abgleich mit den Knotenlabels des CFG unbrauchbar macht.

Um diese sowohl von HTML-Kodierung als auch von Formatierungs-Tags zu bereinigen, wird in der *PrepareLabel* Methode, deren Inhalt Listing 16 zu entnehmen ist, vor der Erstellung des Knotens eine Bereinigung unter Verwendung der *WebUtilities* der .NET-Umgebung und regulärer Ausdrücke durchgeführt.

```
1. //Entferne HTML-Encoding / HTML-Tags aus "value"-Attribut
2. string decodedString = WebUtility.HtmlDecode(source);
3. decodedString = Regex.Replace(decodedString, "<[/]?div>", ";");
4. decodedString = Regex.Replace(decodedString, "<br>", ";");
5. List<string> expressions = [.. decodedString.Split(";",
6.                              StringSplitOptions.RemoveEmptyEntries)];
7.
8. for (int i=0; i<expressions.Count; i++) {
9.     expressions[i] = Regex.Replace(expressions[i], "<[^<.]+>", "").Trim();
```

Listing 16: draw.io spezifische Anpassung der Labelinhalte

Mit Hilfe der *WebUtilities* wird eine Dekodierung sämtlicher HTML-Encodings innerhalb des Labels vorgenommen, sodass beispielsweise Vergleichsoperatoren wieder in ihrer ursprünglichen Form vorliegen. Die regulären Ausdrücke in Zeile 3 und 4 ersetzen *<div>*-Container und Zeilenumbrüche, welche bei der Verwendung von mehreren Zeilen für Anweisungen innerhalb eines Gestaltungselements vorkommen können, einheitlich durch Semikolons. Diese werden im nächsten Schritt als Separatoren verwendet, um die Anweisungen in eine Listen-Datenstruktur zu überführen, die anschließend von allen Formatierungs-Tags und deren Inhalten bereinigt wird. Die Label-Liste wird abschließend dem Knoten als Knoteninhalt hinzugefügt. Auf diese Weise entstehen Knoten mit mehreren Anweisungen, die mit der vorgestellten Methode *ExpandToMaxGraph* zu einem Splitting auf mehrere Knoten mit nur einer Anweisung führen.

Nach Behebung der auftretenden Probleme mit den Knoteninhalten kann abschließend eine Bewertung der Umsetzung dieser Komponente erfolgen. Die Implementierung gemäß Konzept kann als erfolgreich bezeichnet werden, da der Import von sowohl draw.io- als auch eigener XML-Formate und die Überführung in den internen FC mit der .NET-Bibliothek durchgeführt werden kann. Die Bereinigung der Knotenlabels sorgt dabei dafür, dass Veränderungen über webspezifische Formatierungen beseitigt und Knotenlabels korrekt verwendet werden können.

4.4 Bestimmung des Maximum Common Connected Subgraphs

Die dritte Komponente im Rahmen der Implementierung umfasst den Abgleich von CFG und FC und die Bildung des Maximum Common Connected Subgraph mit Hilfe eines Zustandsraum-Suchalgorithmus.

Wie bereits in der Konzeption beschrieben, ist es notwendig, einen Abgleich zwischen allen Knotenpaaren von CFG und FC durchzuführen. Dies ist erforderlich, da der FC aufgrund möglicher Fehler bei der Erstellung des PAP nicht zwingend zusammenhängend sein muss. Somit wäre eine einfache Traversierung des Graphen, beginnend von der Quelle, nicht ausreichend, um alle Knoten bei der Bildung des MCCS zu erfassen. Aufgrund dessen wird der gemäß Listing 17 entwickelte Algorithmus im Falle, dass zwischenzeitlich keine vollständige Übereinstimmung beider Graphen in Form eines vollständigen Matchings gefunden werden kann, für jedes Knotenpaar ausgeführt. Der dabei ermittelte Zustandsraum, als Ergebnis jedes Durchlaufs in Form eines Common Connected Subgraphs, wird auf Vollständigkeit bezogen auf die Knotenanzahl von CFG und FC geprüft und der größte gefundene CCS gespeichert. Wenn während des Abgleichs ein vollständiges Matching gefunden wird, was auf eine Isomorphie der beiden Graphen hinweist, wird dieses Matching zurückgegeben und der Algorithmus beendet.

```
1.  private static HashSet<(Node, Node)> FindMCCS(cfgNode, fcNode, visitedEdges) {
2.
3.      var mapping = new HashSet<(Node, Node)>();
4.
5.      if (!AreLocallyEqual(cfgNode, fcNode)) { //Prüfe Label + Nachbarschaft
6.          return mapping;
7.      }
8.      mapping.Add((cfgNode, fcNode));
9.
10.     foreach (var cfgSucc in cfgNode.GetSuccessors()) {
11.       foreach (var fcSucc in fcNode.GetSuccessors()) {
12.         var edge = ((cfgNode, cfgSucc), (fcNode, fcSucc));
13.
14.         if (!visitedEdges.Contains(edge) && AreLocallyEqual(cfgSucc, fcSucc)) {
15.             visitedEdges.Add(edge);
16.             var subgraph = FindMCCS(cfgSucc, fcSucc, visitedEdges);
17.             foreach (var nodePair in subgraph) {
18.                 mapping.Add(nodePair);}}} //Füge rekursiv Gefundenes hinzu
19.     }
20.     return mapping   ;}
```

Listing 17: Rekursive Ermittlung des MCCS mittels Tiefensuche

Grundlage der rekursiven Methode *FindMCCS* ist eine auf dem CFG und FC gleichzeitig statt–findende Tiefensuche, welche für das jeweils aktuell verglichene Knotenpaar innerhalb der *AreLocallyEqual* Methode prüft, ob die Knoten sowohl bezüglich ihrer Labels als auch ihrer Vorgänger und Nachfolger übereinstimmen. Sollte keine Übereinstimmung vorliegen, wird der aktuelle Ablauf abgebrochen und der bis dahin gefundene CCS zurückgegeben. Dies stellt die Abbruchbedingung der Rekursion dar. Sollte lokale Übereinstimmung vorliegen, werden beide Knoten in den aktuellen CCS, welcher als *mapping* bezeichnet wird, aufgenommen. Des Weiteren werden die Nachfolger beider Knoten durch die gleiche Prüfmethode auf lokale Gleichheit untersucht, bevor die Methode im Übereinstimmungsfall mit diesen rekursiv

aufgerufen wird. Zur Verhinderung von mehrfachen Traversierungen gleicher Kanten und damit auftretenden Endlosschleifen, findet eine Markierung durch Aufnahme in ein HashSet *visitedEdges* statt. Nach erfolgter Rekursion werden die darin ermittelten Matches im aktuellen *mapping* aufgenommen und bauen so sukzessiv den CCS auf. Sollte im Zuge der Traversierung kein vollständiges Matching erzeugt werden, ergibt sich der MCCS aus dem bis zum Ende des Vergleichs von Knotenpaaren größten bis dahin ermittelten CCS.

Nach der Implementierung des Suchalgorithmus wurde dieser anhand des CFG aus dem exemplarischen ggt-Programm getestet. Der CFG wurde dazu dupliziert und in verschiedenen Szenarien, wie mit fehlenden Knoten bzw. Kanten, vollständig oder auch komplett leer, mit dem Original verglichen. Die Ergebnisse bestätigten die Erwartungen, dass korrekte MCCS erstellt werden.

Zur Erweiterung der Testreihe wurden anschließend zufällig erstellte Kontrollflussgraphen beliebiger Größe mit Knotenlabels in Form eines beliebigen Großbuchstabens auf die gleiche Weise getestet. Dabei wurden Anomalien entdeckt. Die ermittelten MCCS bei Original und Duplikat ohne Veränderung waren nicht immer vollständige Matchings, obwohl Isomorphie erwartet wurde. Stattdessen wiesen sie vereinzelt eine geringere oder größere Anzahl an Knoten auf.

Die geringere Anzahl kann darauf zurückgeführt werden, dass im Zuge der Traversierung innerhalb eines Durchgangs die Reihenfolgen, in denen Kanten durchlaufen werden, in beiden Graphen unterschiedlich sind und somit nicht alle Knotenpaare besucht werden. Dies stellt jedoch kein Problem dar, da ein kleineres Matching lediglich dazu führt, dass im Folgeschritt bei der Ermittlung der Graph Edit Distance mehr Rest-Knoten zu untersuchen sind. Diese sind strukturell in beiden Graphen gleich und verursachen somit keine zusätzlichen Editkosten.

Problematischer sind jedoch MCCS, die mehr Matches aufweisen als Knoten vorhanden sind, da diese in der GED-Berechnung nicht erfasst werden können. Der Grund für ein solches Auftreten liegt in der begrenzten Betrachtung der Bedingungen für Isomorphie in diesem Ansatz. Es werden lediglich Knotenlabels und die lokale Nachbarschaft für den Abgleich berücksichtigt. Da es aber in großen Graphen, die innerhalb der Testreihe entstehen, zu gleichen Nachbarschaften und Knotenlabels für mehrere Knoten innerhalb eines Graphen kommen kann, werden auch mehrfache Matches von einem Knoten des CFG zu mehreren gleichen Knoten des FC erfasst. Dadurch wird die Bijektivität, als Voraussetzung für Isomorphie, verletzt und damit aufgezeigt, dass der bisher verfolgte Ansatz nicht geeignet ist, um Isomorphie eindeutig nachzuweisen.

Um das Problem des Over-Matchings einzugrenzen ist eine Erweiterung des Algorithmus um eine globale Betrachtung der Knoten erforderlich. Der verfolgte Ansatz besteht darin, dass die globale Position eines Knotens durch die kürzesten Pfade zu allen anderen Knoten im Graphen bestimmt und in den Vergleich der Knoten mit einbezogen wird. Eine Lösung diese Pfade zu berechnen, stellt für gleichgewichtete Graphen die Breitensuche dar und wird gemäß Listing 18 in diesen Ansatz integriert.

```
1.  foreach (var node in AdjacencyList.Values) {
2.      distances.Add(node, 1000); //Setze Distanzen auf „unerreichbar“
3.  }
4.  queue.Enqueue((startNode));
5.  visited.Add(startNode);
6.  distances[startNode] = 0;
7.
8.  while (queue.Count > 0) {
9.      var currentNode = queue.Dequeue();
10.
11.     foreach (var succ in currentNode.GetSuccessors()) {
12.
13.         if (visited.Contains(succ)) continue;
14.         visited.Add(succ);
15.         distances[succ] = distances[currentNode] + 1;
16.         queue.Enqueue(succ); } };
```

Listing 18: Berechnung der kürzesten Wege mittels Breitensuche

Dabei wird ausgehend vom betrachteten Knoten als *startNode* bei der Traversierung für jeden Knoten der gleichen Ebene eine um 1 erhöhte Distanz zum Vorgänger ermittelt, wie in Zeile 15 erkennbar. Nachdem der Algorithmus alle Knoten durchlaufen und die Distanzen berechnet hat, wird die Liste *distances* mit allen Distanzen zurückgegeben. Um die Distanzen in eine vergleich–bare Form zu überführen, werden daraufhin gemäß Listing 19 die Summen über alle Distanzen beider Knoten ermittelt und als Differenz eines Positionsvergleichs innerhalb beider Graphen ausgegeben.

```
1.  private static int DiffGraphPosition(Graph g1, Graph g2, Node n1, Node n2) {
2.
3.      int totalDifference;
4.      var pathLength1 = g1.ComputeShortestPaths(n1).Values.Sum();
5.      var pathLength2 = g2.ComputeShortestPaths(n2).Values.Sum();
6.      totalDifference = Math.Abs(pathLength1 - pathLength2);
7.
8.      return totalDifference;}
```

Listing 19: Positionsvergleich von zwei Knoten innerhalb ihrer Graphen

Die Differenz drückt dabei den Unterschied in der Position der Knoten zueinander aus. Eine große Differenz bedeutet, dass die Knoten weit voneinander entfernt innerhalb der Graphen liegen, während eine Differenz von 0 bedeutet, dass sie an gleicher Position liegen und somit global betrachtet identisch sind.

Im nächsten Schritt muss überlegt werden, an welcher Stelle der Positionsvergleich in den Algorithmus eingefügt werden muss, um eine effiziente Erweiterung des Vergleichs zu ermöglichen und das Over-Matching Problem zu lösen. Dafür wurden in zwei Ansätzen Positionen ausfindig gemacht.

Der erste Ansatz besteht darin, den Positionsvergleich in die rekursive *FindMCCS* Methode zu integrieren und für jedes Match von Knotenpaaren während der Traversierung durchzuführen. Dabei wird für jeden Knoten nur einmalig das Match mit der geringsten Differenz in das *mapping* aufgenommen.

Der zweite Ansatz dagegen, setzt den Positionsvergleich nach der Ermittlung des MCCS bei Auftreten von Over-Matching gemäß Listing 20 um.

```
1. var currentCCS = FindMCCS(cfgNode, fcNode, visitedEdges);
2.
3. if (currentCCS.Count > cfg.NodeCount && currentCCS.Count > fc.NodeCount) {
4.   foreach (var match in currentCCS) {
5.      var diffPosition = DiffGraphPosition(cfg, fc, match.Item1, match.Item2);
6.      if (diffPosition != 0) {
7.         /*Lösche Match mit max(diffPos), wenn cfgNode und FcNode > 1 im CCS*/
8.      };
```

Listing 20: MCCS Ermittlung mit Positionsvergleich bei Over-Matching (gekürzt)

Dabei wird nach der Ausführung der MCCS-Ermittlung im Fall von Over-Matching (Zeile 3) der Positionsvergleich für alle im CCS enthaltenen Matchings durchgeführt. Bei vorliegender Posi–tionsverschiedenheit (Zeile 6) wird das Matching mit der höchsten Differenz gelöscht, an dem die beiden Knoten beteiligt sind. Voraussetzung für die Löschung ist jedoch, dass die beteiligten Knoten noch in einem anderen Matching im CCS vorliegen, um diese nicht vollständig aus dem MCCS zu entfernen. Für beide Methoden wurden Testreihen mit zufällig erstellten Graphen und ihren Duplikaten angelegt und die Laufzeiten der MCCS-Ermittlung beider Ansätze gemessen. Die Ergebnisse sind in Abbildung 16 dargestellt, während die exakten Messreihen in der Tabelle im Anhang A3 zu finden sind.

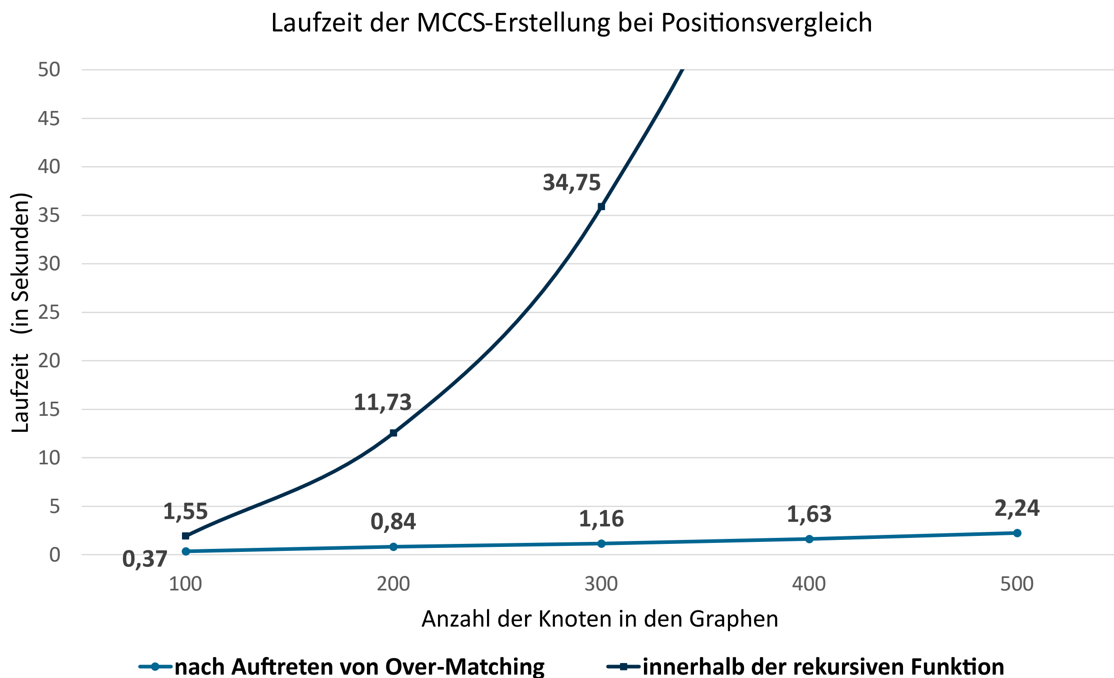


Abbildung 16: Laufzeit Positionsvergleich

Dem Diagramm lässt sich entnehmen, dass der zweite Ansatz aufgrund der wesentlich geringeren Laufzeit auch bei größer werdenden Graphen vorzuziehen und in den Algorithmus aufzunehmen ist.

Nach erfolgter Erweiterung des Algorithmus um den globalen Positionsvergleich traten keine Over-Matchings mehr in den Testergebnissen auf, wie Abbildung 17 zu entnehmen.

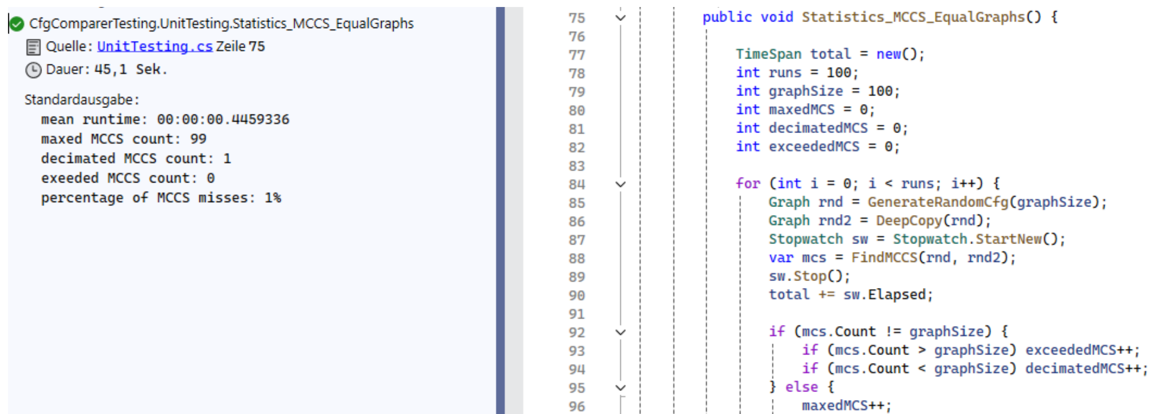


Abbildung 17: Häufigkeit von Abweichungen bei MCCS Bestimmung isomorpher Graphen

Als Ergebnis der Implementierung der dritten Komponente lässt sich festhalten, dass gemäß Konzept, ein Algorithmus zur Berechnung des MCCS erfolgreich integriert werden konnte. Für kleine Graphen im Sinne der Aufgabenstellung ist somit eine Berechnung des MCCS für vollständige und unvollständige Matchings möglich. Tests mit größeren Graphen, welche durch eine Begrenzung des Labels auf nur einen Buchstaben in ihrer Verschiedenheit eingeschränkt waren, zeigten aber, dass ein Knotenvergleich auf Basis der Knotenlabels und lokalen Nachbarschaft nicht ausreichend sind für einen Isomorphie-Nachweis. Aus diesem Grund wurde der Vergleichsansatz, zumindest bei der Berechnung des MCCS, um einen globalen Positionsvergleich ergänzt. Mit diesem wird die Möglichkeit auftretender Over-Matchings minimiert und der Algorithmus robuster in seiner Vergleichslogik gemacht. Das auftretende Problem hat den theoretischen Sachverhalt deutlich gemacht, dass der Nachweis von Isomorphie nur durch vollständigen Eigenschaftsabgleich erbracht werden kann.

4.5 Implementierung eines Algorithmus für die Graph Edit Distance

Die Weiterverarbeitung des gewonnenen MCCS wird im Rahmen der Implementierung der vierten Komponente fortgeführt. Das Ziel dieser Komponente besteht darin, die Ähnlichkeit zwischen dem C-Programm und dem Programmablaufplan abschließend zu quantifizieren. Dies geschieht durch die Berechnung der Graph Edit Distance, welche die Transformationskosten für die Umwandlung des FC in den CFG erfasst.

Ausgangspunkt der Berechnung sind die in der MCCS-Ermittlung identifizierten Matchings. Die Knoten dieser Matchings werden mit der Anzahl der Knoten im CFG und im FC gemäß Listing 21 abgeglichen, um zu überprüfen, ob der MCCS bereits beide Graphen abdeckt. Liegt ein vollständiges Matching vor, wird aufgrund der Isomorphie angenommen, dass sowohl das Programm als auch der PAP identisch sind. In diesem Fall gibt die Berechnung der GED in Zeile 4 direkt Transformationskosten in Höhe von 0 zurück. Die Rückgabe erfolgt als Tupel, bestehend aus den Transformationskosten und einem weiteren Tupel, das diese Kosten auf die Arten der Veränderung aufschlüsselt. Dies ist in Zeile 7 am endgültigen Rückgabewert nach Abschluss der GED-Berechnung im anderen Fall zu sehen.

```
1. var matches = FindMCCS(cfg, fc);  //Berechne den MCCS
2.
3. if (matches.Count == cfg.NodeCount && matches.Count == fc.NodeCount)
4.     return (0, (0, 0, 0, 0, 0)); //Gebe bei vollst. Matching keine Kosten aus
5.
6. //Rückgabe nach Ablauf der Funktion
7. return (totalCosts, (costsNodeInsert, costsNodeDelete, costsNodeRelabel,
   costsEdgeInsert, costsEdgeDelete));
```

Listing 21: Rückgabe bei vollständigem Matching

Sollte ein unvollständiges Matching vorliegen, werden im Folgeschritt CFG und FC um die übereinstimmenden Knoten bereinigt und auf Basis der Rest-Graphen die weitere Berechnung fortgesetzt. Zunächst werden dabei die Knoten des CFG nacheinander mit den Knoten des FC verglichen, bevor anschließend die Gegenprüfung durch die Knoten des FC erfolgt. Dadurch wird gewährleistet, dass auch bei unterschiedlicher Knotenanzahl in beiden Graphen eine voll–ständige Betrachtung stattfinden kann.

Im Zuge des Knotenvergleichs wird im ersten Schritt, wie in Listing 22 dargestellt, in der Methode *NodeExistsInGraph*, anhand des Labels festgestellt, ob der aktuell betrachtete Knoten des CFG eine Entsprechung im FC besitzt.

```
1. var matchNode = NodeExistsInGraph(restFc, node); //Prüfe nur anhand Label
2. if (matchNode != null) {
3.     foreach (var successor in node.GetSuccessors()) {
4.         var matchSuccNode = NodeExistsInGraph(restFc, successor);
5.         if (matchSuccNode != null) {
6.             if(!EdgeExistsInGraph(restFc, node, successor)) {
7.                 costsEdgeInsert += EdgeInsertCost;
8.                 editSteps.Add($"Insert Edge ... "); } }
9.     }
10.    processed.Add(matchNode); //Markiere Knoten in FC als „bearbeitet“
```

Listing 22: Behandlung bei Match von Knoten mit gleichem Label

Sollte ein solcher Knoten *matchNode* existieren, wird auf die gleiche Weise geprüft, ob die Nachfolger des CFG-Knotens auch im FC vorhanden sind und ob im positiven Fall die Kante zwischen ihnen auch im FC abgebildet ist. Das Fehlen einer Kante führt dazu, dass diese im FC eingefügt werden muss und somit entsprechende Editkosten gemäß Zeile 7 zu den *costsEdgeInsert*, als Teilkosten der Transformationskosten für das Hinzufügen von Kanten, addiert werden. Zudem wird der *matchNode* durch die Aufnahme in die Liste *processed* als bereits „bearbeitet" markiert.

Für den Fall, dass kein *matchNode* auf Basis des Knotenlabels existiert, findet im zweiten Schritt gemäß Listing 23 mit der Methode *FindMatchingNodeInGraph* die Suche nach einem Knoten mit derselben Nachbarschaft im FC statt, um zu prüfen, ob ein Matching nur aufgrund falscher Labels ausgeschlossen wurde.

```
1.  //Finde einen Knoten mit gleicher Nachbarschaft (Label ignoriert)
2.  var matchingNode = FindMatchingNodeInGraph(restFc, node);
3.
4.  if (matchingNode == null) {
5.      costsNodeInsert += NodeInsertCost;
6.
7.      /* editSteps.Add($"Insert Node ...");  (vereinfacht)*/
8.
9.      foreach (var successor in node.GetSuccessors()) {
10.         if (editSteps.Add($"Insert Edge ...")) {
11.             costsEdgeInsert += EdgeInsertCost;};}
```

Listing 23: Einfügen eines Knotens bei kompletter Abweichung

Falls auch diese Suche erfolglos bleibt, wird der CFG-Knoten zusammen mit seiner vollständigen Nachbarschaft in den FC eingefügt, wobei dies in Zeile 9 bis 11 nur exemplarisch für die Nach–folger angegeben ist. Die Kosten dafür werden zu den entsprechenden Teilkosten addiert.

Wurde dagegen ein Knoten im FC mit passender Nachbarschaft gefunden, so wird dieses Matching zunächst einer Liste mit möglichen Kandidaten für ein Relabeling *relabelCandid* hinzugefügt, wie in Listing 24 dargestellt. Ein direktes Relabeling erfolgt an dieser Stelle nicht, um mögliche exakte Matchings, die folgen könnten, zu bevorzugen. Erst nachdem alle Knoten des CFG abgeglichen wurden, erfolgt das Relabeling, nachdem gemäß Zeile 5 geprüft wurde, ob die Kandidaten noch nicht „bearbeitet" wurden und damit noch kein exaktes Matching für den Knoten existiert.

```
1.          relabelCandid.Add((matchingNode,node)); //Markiere als Relabel-Kandidat
2.      }
3.  }
4.  foreach(var relCand in relabelCandid) {
5.      if (!processed.Contains(relCand.source)){//Relabel, wenn nicht „bearbeitet“
6.          costsNodeRelabel += NodeRelabelCost;
7.          editSteps.Add($"Relabel Node ...");
```

Listing 24: Relabeling eines Knoten bei gleicher Nachbarschaft

Anschließend werden die Knoten des FC auf gleiche Weise zunächst über den Labelvergleich und im Falle ausbleibender Matchings über die Prüfung gleicher Nachbarschaften mit dem CFG abgeglichen. Der Unterschied zum Durchlauf der CFG-Knoten besteht darin, dass die Knoten und

Kanten bei ausbleibendem Matching im FC entfernt und entsprechend Teilkosten für das Löschen hinzugefügt werden.

Der Algorithmus endet, wie bereits in Listing 21 in Zeile 7 gezeigt, mit der Rückgabe der Transformationskosten und ihrer aufgeschlüsselten Teilkosten. Dies geschieht, nachdem alle Knoten im CFG und FC abgeglichen wurden und die Editkosten für jeden Knoten entsprechend den spezifischen Teilkosten erfasst wurden. Zusätzlich wird über einen Out-Parameter die Liste *editSteps*, welche die Änderungsschritte erfasst, ausgegeben. Das erlaubt die unabhängige Verarbeitung und Ausgabe von Editkosten und Änderungsschritten.

Nach der Implementierung wurden Tests zur Prüfung der korrekten Berechnungen durch–geführt. Dabei wurde festgestellt, dass bei verschiedenen Vergleichssituationen, beispielsweise bei vorkommenden Knoten- oder Kantendifferenzen, unterschiedlichen Labels sowie dem Vergleich mit identischen oder leeren FC, zwar korrekte Berechnungen und Schrittauf–zeichnungen erfolgen, aber die Transformationskosten nicht optimal ermittelt werden.

Das kann dazu führen, dass ein CFG-Knoten und ein FC-Knoten, welche sich geringfügig in den Knotenlabels und der Nachbarschaft unterscheiden, statt durch minimale Änderungen angepasst zu werden, der FC-Knoten gelöscht und der CFG-Knoten eingefügt werden, was höhere Kosten verursacht. Dadurch kommt es im gesamten Abgleich von C-Programm und PAP zu überbewerteten Kosten, was die Ähnlichkeit verzerrt. Aus diesem Grund muss eine Anpassung des Algorithmus erfolgen, um die Transformationskosten korrekt zu minimieren.

Ein Ansatz zur Anpassung könnte darin bestehen, beim Abgleich jedes CFG-Knotens mit den FC-Knoten das kostenminimale Matching auf Basis eines Knotenlabel- und Nachbarschaftsver–gleichs zu ermitteln und die auftretenden Editkosten den Transformationskosten hinzuzufügen. Die dabei im Matching erfassten FC-Knoten werden als „bearbeitet" markiert, um mehrfache Matchings auf den gleichen FC-Knoten zu verhindern. Sollten nach Bearbeitung der CFG-Knoten noch unmarkierte FC-Knoten vorhanden sein, werden diese gelöscht und die entsprechenden Editkosten zu den Transformationskosten addiert. Abschließend werden die so minimierten Transformationskosten ausgegeben. Die Aufzeichnung der Bearbeitungsschritte kann dabei parallel zur Ermittlung der minimalen Kosten erfolgen.

Eine vollständige Implementierung dieses Ansatzes sowie die Durchführung von Tests konnten aufgrund der zeitlichen Beschränkungen in der Bearbeitung nicht abgeschlossen werden. Dies führte dazu, dass das ursprüngliche Ziel der optimalen Berechnung der Graph Edit Distance nicht vollständig erreicht werden konnte.

4.6 Integration einer zeichenketten- und strukturbasierten Metrik

In der abschließenden Implementierung der fünften Komponente findet die Integration des Knotenlabelvergleichs statt, der für die Bestimmung des MCCS und die Berechnung der GED von Bedeutung ist. Der Konzeption entsprechend wird zur Berechnung der Ähnlichkeit von Knoten eine multidimensionale Metrik implementiert, welche die Ähnlichkeit der Knotenlabels sowohl als Zeichenketten als auch als Anweisungen eines Programms berücksichtigt. Dabei wurden die drei Bestandteile der Metrik, String-Ähnlichkeit, syntaktische Ähnlichkeit und semantische Ähnlichkeit, formal festgelegt und in dieser zu einer Gesamtähnlichkeit zusammengeführt.

Der erste Bestandteil, die String-Ähnlichkeit, wird mit Hilfe der Levenshtein-Distanz und dyna–mischer Programmierung gemäß Listing 25 berechnet. Dabei ist zu berücksichtigen, dass die Labels vor dem Vergleich um Leerzeichen bereinigt werden, damit versehentlich falsche Leerzeichensetzungen im Zuge der Beschriftung die Ähnlichkeit nicht verzerren.

```
1.  for (int i = 0; i <= label1.Length; i++) { costMatrix[i, 0] = i; }
2.  for (int j = 0; j <= label2.Length; j++) { costMatrix[0, j] = j; }
3.
4.  for (int i = 1; i <= label1.Length; i++) {
5.      for (int j = 1; j <= label2.Length; j++) {
6.          int replaceCost = (label1[i - 1] == label2[j - 1]) ? 0 : 1;
7.
8.          costMatrix[i, j] = Math.Min(Math.Min(
9.                      costMatrix[i - 1, j] + 1,                // Löschen
10.                     costMatrix[i, j - 1] + 1),               // Einfügen
11.                     costMatrix[i - 1, j - 1] + replaceCost); }} // Ersetzen
12. }
13. return costMatrix[label1.Length, label2.Length];
```

Listing 25: Berechnung der String-Distanz

Anschließend wird die in Kapitel 2.2.1 der Theorie bereits vorgestellte Standardberechnung verwendet, die eine Distanzmatrix *costMatrix* als Grundlage nutzt, welche zu Beginn gemäß den Zeilen 1 und 2 mit den Basisfällen gefüllt wird. Im Verlauf des Algorithmus wird dann zeilenweise von links nach rechts der Inhalt der Distanzmatrix an der Stelle *costMatrix[i, j]* durch den minimalen Kostenwert bestimmt, der in den Zeilen 8 bis 11 ermittelt wird. Nachdem der Algorithmus das Ende der letzten Zeile erreicht hat, wird der Eintrag an der untersten rechten Stelle der Matrix als Levenshtein-Distanz zurückgegeben.

Im nächsten Schritt wird die Levenshtein-Distanz gemäß Listing 26 und Formel (20) zur prozentualen String-Ähnlichkeit *literalSimilarity* umgerechnet, indem die Distanz auf das längere der beiden Labels normalisiert und anschließend der Kehrwert als Prozentsatz berechnet wird.

```
1. int stringDistance = CalculateStringDistance(label1, label2);
2.
3. double literalSimilarity =
   (1 -(stringDistance / Math.Max(label1.Length, label2.Length))) * 100;
```

Listing 26: Berechnung der String-Ähnlichkeit

Der zweite und dritte Bestandteil erfordern, dass die Knotenlabels als Anweisungen interpretiert und die zugrundeliegenden Strukturen dieser Anweisungen analysiert werden. Anweisungen einer Programmiersprache, die Bestandteile ganzer Programme bilden, können durch die

statische Analyse im Compilerbau untersucht werden. Auf dieser Grundlage können Syntaxbäume generiert werden, deren Auswertung die Basis für die Berechnungen der beiden Metrikbestandteile bildet.

Im .NET-Umfeld gibt es die .NET Compiler Platform-SDK, auch als Roslyn-API bekannt, die hauptsächlich für C# und Visual Basic entwickelt wurde und Zugriff auf die vom .NET-Compiler erstellten Modelle zur Codeanalyse gewährt [54]. Der erklärte Nutzen der API liegt darin, Compiler-Informationen bereitzustellen und darauf aufbauend Unterstützung bei der Erstellung von Analysetools und Codefehlerbehebungen für eigene Codebasen und Programmierrichtlinien zu geben [54].

Im Rahmen dieses Ansatzes kann die Roslyn-API, die in der *Microsoft.CodeAnalysis* Bibliothek bereitgestellt wird, zur Erstellung von Syntaxbäumen verwendet werden. Diese basieren auf *SyntaxNodes*, *SyntaxToken* und *SyntaxTrivia* [55]. *SyntaxTrivia* stellen Quelltextinformationen wie Leerzeichen oder Kommentare dar, die für diesen Ansatz jedoch keine Relevanz haben und deshalb nicht weiter berücksichtigt werden. *SyntaxNodes* und *SyntaxToken* ähneln den klassischen Nichtterminalen und Terminalen des Compilerbaus und sind die Hauptelemente der Analyse.

Eine Besonderheit der Roslyn-API ist, dass auch dann ein Syntaxbaum generiert wird, wenn das Knotenlabel keine syntaktisch korrekte Anweisung darstellt. Fehlen bei der Erstellung Token, ergänzt der Parser die erwarteten mit dem *IsMissing = true* Flag oder überspringt sie, bis er bearbeitbare Elemente findet [55]. Tests mit der API haben gezeigt, dass bei abstrakten Beschreibungen innerhalb der Labels neben der Verwendung von *SyntaxToken* auch die Nutzung von *SyntaxNodes* sinnvoll ist. Dies liegt daran, dass das beschriebene Parserverhalten oft dazu führt, dass leere Token oder gar keine Token gesetzt werden und somit nur aus den Nodes Informationen gewonnen werden können. Entgegen der ursprünglichen Konzeption werden daher nicht die abstrakten Syntaxbäume, sondern die konkreten Syntaxbäume der C#-Sprache für die Untersuchung herangezogen. Innerhalb des Ansatzes wird daher, wie in Listing 27 gezeigt, für jedes Knotenlabel ein Syntaxbaum erstellt.

```
1. public static (Nodes, Token) ExtractLabelTree(string label) {
2.
3.     var root = CSharpSyntaxTree
4.                 .ParseText(label).GetRoot()
5.                 .DescendantNodes().First().DescendantNodes().First();
6.
7.     return (root.DescendantNodesAndSelf(), root.DescendantTokens());}
```

Listing 27: Erstellung des Syntaxbaums eines Labels

Wie in Zeile 5 ersichtlich, wird der Syntaxbaum jedoch tiefer gewurzelt, als es die Roslyn-API vorgibt. Der Grund dafür ist, dass die ursprüngliche Wurzel in allen Knotenlabels identisch ist und zudem gleiche Trivia als Kindknoten enthält. Bei der Analyse syntaktisch gleicher Elemente würde dies zu einer immer wiederkehrenden Basis-Ähnlichkeit führen.

Mit Hilfe der aus dem Syntaxbaum extrahierten Nodes und Token können im Folgenden die beiden Bestandteile der Metrik, syntaktische und semantische Ähnlichkeit, berechnet werden.

Dabei werden die in der Konzeption definierten Formeln um einen gleichwertigen Beitrag der *SyntaxNodes* zur Berechnung der Ähnlichkeit erweitert.

Im Rahmen der syntaktischen Ähnlichkeit führt dies zu einer Erweiterung der Formel (21) für die normierte Berechnung des Durchschnitts von Nodeanzahl und Tokenanzahl, die in Listing 28 dargestellt wird. Die Gewichtung von *SyntaxNodes* und *SyntaxToken* wird dabei als gleich angenommen, um Berechnungen und Feinabstimmungen bei der Parameterwahl für die über–geordnete strukturelle Ähnlichkeit nicht unnötig zu erschweren.

```
1. double treeSynSimilarity =
2.    ((double)(nodeCount1 + nodeCount2) /
3.             (2*Math.Max(nodeCount1, nodeCount2))   * 0.5 +
4.     (double)(tokenCount1 + tokenCount2) /
5.             (2*Math.Max(tokenCount1, tokenCount2)) * 0.5) * 100;
```

Listing 28: Berechnung der syntaktischen Ähnlichkeit

Für die Berechnung der semantischen Ähnlichkeit ergibt sich analog eine Erweiterung der Formel (22) aus der Konzeption zu der gemäß Listing 29 in Zeile 7 angegebenen Formel für den Jaccard-Index. Dabei wird in den vorhergehenden Zeilen 1 und 4 exemplarisch die Berechnung des Durchschnitts semantisch gleichartiger Token und die Vereinigung über alle Tokenarten aufgezeigt.

```
1. var equalTokenKinds = treeData1.Token.Select(t1 => t1.Kind())
2.                          .Intersect(treeData2.Token.Select(t2 => t2.Kind()));
3.
4. var totalTokenKinds = treeData1.Token.Select(t1 => t1.Kind())
5.                          .Union(treeData2.Token.Select(t2 => t2.Kind()));
6.
7. double treeSemSimilarity = (((double)equalNodeKinds.Count() /
8.                                      totalNodeKinds.Count()) * 0.5 +
9.                             ((double)equalTokenKinds.Count() /
10.                                     totalTokenKinds.Count()) * 0.5) * 100;
```

Listing 29: Berechnung der semantischen Ähnlichkeit

Nach Berechnung der String-Ähnlichkeit, der syntaktischen Ähnlichkeit und der semantischen Ähnlichkeit erfolgt abschließend gemäß Listing 30 unter Verwendung der in der JSON-Konfiguration angegebenen Gewichtungen für die strukturelle Ähnlichkeit in Form des *QualtoQuantWeight* und für den Einfluss der String-Ähnlichkeit auf die Gesamtähnlichkeit in Form des *LiteralWeight* die Ermittlung zunächst der strukturellen Ähnlichkeit und infolgedessen der Gesamtähnlichkeit.

```
1. double structuralSimilarity =
2.          treeSemSimilarity * QualtoQuantWeight +
3.          treeSynSimilarity * (1 - QualtoQuantWeight);
4.
5. double totalSimilarity =
6.          literalSimilarity * LiteralWeight +
7.          structuralSimilarity * (1 - LiteralWeight);
```

Listing 30: Berechnung der strukturellen Ähnlichkeit und Gesamtähnlichkeit

Im letzten Schritt wird die Gesamtähnlichkeit *TotalEQ* mit dem festgelegten Schwellenwert *EqualThreshold* gemäß Listing 31 abgeglichen. Bei Überschreitung des Schwellenwerts wird entschieden, dass die untersuchten Knotenlabels als gleich angesehen werden, während sie bei Unterschreitung als nicht gleich betrachtet werden.

```
1. public static bool LabelIsEqual(Node node1, Node node2) {
2.
3.     var (TotalEQ, _, _, _) =
4.         CalculateLabelEquality(node1.GetLabel()[0],node2.GetLabel()[0]);
5.     return TotalEQ >= (EqualThreshold * 100);
6. }
```

Listing 31: Labelvergleich mit Schwellenwert

Nach der Implementierung wurden mehrere einfache Tests mit verschiedenen Vergleichs–szenarien für Knotenlabels durchgeführt und ausgewertet. Ein exemplarisches Beispiel einer Testauswertung ist in Abbildung 18 dargestellt. In diesem Beispiel wurde eine gleichmäßige Verteilung der Gewichtungen über alle Parameter vorgenommen, um die unterschiedlichen Einflüsse der einzelnen Parameter auf das Maß der Ähnlichkeit zu verdeutlichen. Die String-Ähnlichkeit hat einen großen Einfluss, wenn es darum geht, exakte Inhalte, wie zum Beispiel die Werte bei einer Zuweisung, zu erkennen und abzugleichen. Bei der Spezifizierung eines Bezeichners, etwa von „a“ zu „Anzahl“ wie im ersten Beispiel, kann sie jedoch dazu führen, dass die Vergleichbarkeit deutlich sinkt. Um in solchen Vergleichen Raum für Flexibilität zu schaffen, kann mit der Erhöhung des Einflusses der strukturellen Ähnlichkeit die Vergleichbarkeit hergestellt werden. Darüber hinaus kann beispielsweise eine Erhöhung der Gewichtung der semantischen Ähnlichkeit die Unterscheidbarkeit von Anweisungen mit Operatoren wie „=“ und „<=“ klarer herausarbeiten. Aus den Tests ergab sich noch keine eindeutige Festlegung der Parameter.

```
Standardausgabe:
  Equality Measurements:
  Total: 062,50 Literal: 025,00 Syn: 100,00 Sem: 100,00  -- [a = 19] <> [Anzahl = 9]
  Total: 032,71 Literal: 000,00 Syn: 087,50 Sem: 043,33  -- [return] <> [Rückgabe]
  Total: 044,79 Literal: 014,29 Syn: 087,50 Sem: 063,10  -- [if 1] <> [Wenn(1)]
  Total: 072,22 Literal: 044,44 Syn: 100,00 Sem: 100,00  -- [a = 19] <> [Anzahl= 19]
  Total: 070,00 Literal: 040,00 Syn: 100,00 Sem: 100,00  -- [x=x-y] <> [y=y-x]
  Total: 080,00 Literal: 080,00 Syn: 100,00 Sem: 060,00  -- [a = 19] <> [a <= 19]
  Total: 073,91 Literal: 071,43 Syn: 100,00 Sem: 052,78  -- [a[8] = 10] <> [a(8) = 10]
```

Abbildung 18: Durchführung exemplarischer Ähnlichkeitsmessung

Die Beurteilung der Implementierung zeigt, dass alle im Konzept definierten Vorgaben erfolgreich umgesetzt wurden. Allerdings musste der Betrachtungshorizont im Bereich der strukturellen Bestandteile erweitert werden, indem auch die *SyntaxNodes* berücksichtigt wurden. Dies war notwendig, um bei abstrakteren Beschreibungen innerhalb der Labels einen aussagekräftigeren Syntaxbaum zu erhalten. Die Ergebnisse der Tests zeigen, dass eine Vergleichbarkeit hergestellt werden kann, jedoch die Anpassung der Parameter sich als komplex herausstellt. Weitere Untersuchungen sind erforderlich, um geeignete Parametereinstellungen zu ermitteln.

4.7 Konsolenanwendung und Gesamtbewertung

In diesem letzten Abschnitt wird die Integration aller fünf entwickelten Komponenten in die prototypische .NET-Konsolenanwendung vorgestellt. Sie dient als Schnittstelle zum Nutzer und soll im Zuge des Abgleichs von C-Programm und Programmablaufplan alle Informationen bereit–stellen, die im Rahmen der Erstellung und Verarbeitung der beiden als Parameter übergebenen Dateien gewonnen werden. Darüber hinaus führt sie, mit der Berechnung innerhalb eines einfachen Bewertungssystems, den letzten Schritt in der Bearbeitung aus und liefert dem Nutzer eine Grundlage, um die Bewertung einer studentischen Leistung vornehmen zu können.

Der Ablauf bei der Ausführung der Konsolenanwendung lässt sich dabei wiederum klassisch mit dem EVA-Prinzip beschreiben. Zunächst wird die ausführbare Datei *CFGComparer* auf der Kommandozeile gestartet, wie in Abbildung 19 veranschaulicht.

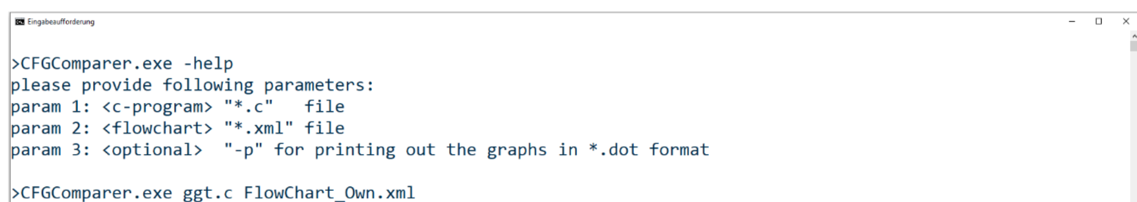


Abbildung 19: Windows-Ausführung der Konsolenanwendung

Dabei werden die beiden notwendigen Eingabeparameter, das C-Programm und der PAP im XML-Format, übergeben, wodurch die Weitergabe an die Datenverarbeitung eingeleitet wird. Zusätzlich ist über den Standard-Parameter *-help* der Aufruf einer Kurzerklärung der verfüg–baren Parameter abrufbar, um Eingabefehler durch den Nutzer zu minimieren.

In der Datenverarbeitung wird, gemäß Listing 32 in vereinfachter Form, zunächst über die Funktion *ImportCompilerCFGRaw* der GCC-Compiler aufgerufen und im Zuge der Kompilierung die CFG-Zeichenkette in die Anwendung importiert. Im nächsten Schritt wird in Zeile 3 die interne CFG-Graphenrepräsentation erzeugt und der Graph so erweitert, dass in jedem Knoten nur eine Anweisung enthalten ist.

```
1.  string compilerCfg = ImportCompilerCfgRaw(CFileDir + args[0]);
2.
3.  Graph cfgGraph = ExpandToMaxGraph(GenerateGraphFromRaw(compilerCfg));
4.
5.  Graph fcGraph = ExpandToMaxGraph(GenerateGraphFromXML(XmlFileDir + args[1]));
6.
7.  var (totalCosts, splitCosts) = CalculateGED(cfgGraph, fcGraph, out editSteps);
8.
9.  double maxPoints = 2 * cfgGraph.NodeCount + cfgGraph.EdgeCount;
10. double receivedPoints = maxPoints - totalCosts;
```

Listing 32: Grundlogik der Konsolenanwendung (vereinfacht)

Im Anschluss wird in Zeile 5 die XML-Datei in die interne FC-Graphenrepräsentation überführt und der Graph ebenfalls erweitert. Da sowohl CFG als auch FC in vergleichbarer Form vorliegen, kann die Berechnung der Ähnlichkeit durchgeführt werden. Die Funktion *CalculateGED* erzeugt zunächst den MCCS und berechnet anschließend die Transformationskosten der Rest-Graphen. Als Rückgabewert wird ein Tupel bereitgestellt, das sowohl die Transformationskosten als auch

deren Aufsplittung auf die fünf Editierarten enthält. Zudem wird die Liste *editSteps* mit den durchgeführten Editierschritten befüllt. Den letzten Bearbeitungsschritt stellt die Berechnung der Bewertung in den Zeilen 9 und 10 dar. Hierbei wird die maximal erreichbare Punktzahl, gemäß Konzeption, als Summe der doppelten Knotenanzahl und der Kantenanzahl des CFG ermittelt und anschließend durch Subtraktion der Transformationskosten die erreichte Punkt–zahl berechnet.

Die Ergebnisse werden nach ihrer Berechnung auf der Konsole ausgegeben, wie in Abbildung 20 dargestellt. Es erfolgt eine textuelle Ausgabe der Graphenstrukturen von CFG und FC, der Matchings des MCCS, einer Übersicht der Matchings der Labelvergleichsberechnungen, der *editSteps* Liste mit den nachvollziehbaren Änderungen im Rahmen der GED-Berechnung und deren Kosten sowie der finalen Bewertung.

```
Eingabeaufforderung

>CFGComparer.exe ggt.c FlowCha
Graph: Control Flow Graph
------------------------
0->[2]
Outdegree: 1
0<-
Indegree: 0
["Start"]
------------------------
1->
Outdegree: 0
1<-[7]
Indegree: 1
["End"]
------------------------
2->[6][7]
Outdegree: 2
2<-[0]
Indegree: 1
["y!=0"]
```

```
Eingabeaufforderung

# nodes/edges in Flow Chart : 7 / 8

=========================================
Node mapping in Maximum Common Connected

Node CFG   |Node FC
0          |0
2          |2
7          |4
1          |1

=========================================
Found Label Matchings based on threshold

Equality: 100,00% | [Start] <---> [Start]
Equality: 100,00% | [End] <---> [End]
Equality: 100,00% | [y!=0] <---> [y!=0]
Equality: 100,00% | [x>y] <---> [x>y]
Equality: 100,00% | [y=y-x] <---> [y=y-x]
Equality: 100,00% | [x!=y] <---> [x!=y]
Equality: 100,00% | [return x] <---> [ret
```

```
Eingabeaufforderung
======================================
Edits made in flow chart to fit control

- Insert Node ["x=x-y"] into flow chart
- Insert Edge from ["x>y"] to ["x=x-y"]
- Insert Edge from ["x=x-y"] to ["x!=y"

Calculation of GED:

Node add costs : 2
Node del costs : 0
Node rel costs : 0
Edge add costs : 2
Edge del costs : 0

**TOTAL costs**: 4
======================================

evaluation result: 22/26
percentage: 84,62%
```

Abbildung 20: Ausgabe der Konsolenanwendung

Eine abschließende Bewertung der Implementierung aller Komponenten ergibt, dass die in der Konzeption festgelegten Anforderungen mit Ausnahme der optimalen Berechnung der GED in Komponente 4 umgesetzt werden konnten. Die prototypische Anwendung kann aufgrund dessen voll funktional arbeiten, bietet aber im Hinblick auf die Ergebnisse noch beschränkte Aussagekraft bezüglich der Ähnlichkeit von C-Programm und PAP, da die Transformations–kosten, welche als Grundlage der Berechnung dienen als zu hoch angesetzt werden könnten. Daher ist eine Anpassung des GED-Algorithmus erforderlich, um die vollständige Funktionalität der Bibliothek und Anwendung zu gewährleisten und die Berechnungen weiter zu optimieren.

5 Fazit

5.1 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung eines Ansatzes, um automatisch C-Programme und zugehörige Programmablaufpläne auf Korrektheit zu vergleichen und im Fall von Teilkorrektheit eine Bewertung der Ähnlichkeit vorzunehmen. Dieser Ansatz sollte in einer .NET-Bibliothek für die zukünftige Nutzung in anderen Anwendungen bereitgestellt und in einer prototypischen Konsolenanwendung geprüft werden.

Zur Erreichung dieses Ziels wurde die Problemstellung auf den Bereich der Graphentheorie und den Vergleich der Ähnlichkeit zweier Kontrollflussgraphen übertragen. Dafür wurden zu Beginn die notwendigen theoretischen Konzepte und Begrifflichkeiten vorgestellt. Anschließend wurden die Rahmenbedingungen für den Vergleich sowie die technische Umsetzung des Prototyps erarbeitet und ein Gesamtkonzept für die Lösung erstellt.

Im Rahmen der Implementierung wurde das Konzept im letzten Schritt umgesetzt. Dabei wurden zunächst sowohl der Kontrollflussgraph des Programms mit dem GCC-Compiler als auch der Programmablaufplan über eine draw.io XML-Datei importiert und in die interne Graphen–repräsentation überführt. Im Anschluss wurden beide Graphen anhand eines Algorithmus zur Bestimmung des Maximum Common Connected Subgraphs verglichen. Ein vollständiger MCCS bedeutete dabei, dass der Programmablaufplan im Hinblick auf das Programm korrekt war. Ein unvollständiger MCCS deutete dagegen auf Abweichungen hin, die im nächsten Schritt durch die Ermittlung der Graph Edit Distance zur Bestimmung der Ähnlichkeit untersucht wurden. Die errechneten Transformationskosten, die als Maß für die Ähnlichkeit dienen, flossen dabei in die Bewertung im Rahmen des Vergleichs ein. Abschließend wurde eine Konsolenanwendung entwickelt, die auf Basis der Bibliothek die Ergebnisse der Berechnungen und die Bewertung ausgab.

Als Ergebnis der Implementierung konnte festgehalten werden, dass der Import des Kontrollflussgraphen und des Programmablaufplans erfolgreich durchgeführt wurde und die Erstellung eines korrekten Maximum Common Subgraphs möglich war. Die anschließende Berechnung der GED ergab allerdings, dass zwar Transformationskosten ermittelt wurden, diese aber nicht minimal waren. Dadurch wird die Lösung in ihrer Verwendbarkeit eingeschränkt, da die Ähnlichkeitsbestimmung auf Basis der GED von den minimalen Transformationskosten abhängt. Um diese Einschränkung zu beheben, muss der GED-Algorithmus im Zuge einer Weiterentwicklung korrigiert werden. Trotzdem konnte die Funktionalität der Komponenten bei der Berechnung einer noch eingeschränkten Ähnlichkeit zwischen Programm und Programm–ablaufplan bestätigt werden.

5.2 Ausblick

Die erstellte .NET-Bibliothek erlaubt eine eingeschränkte Berechnung der Ähnlichkeit von C-Programmen und Programmablaufplänen. Diese zu optimieren und den Algorithmus dahin–gehend anzupassen, die tatsächlich minimalen Transformationskosten zu ermitteln, stellt den Hauptfokus der Weiterentwicklung dar.

Sobald die Bibliothek uneingeschränkt nutzbar ist, kann eine Erweiterung in Erwägung gezogen werden. Es ist vorstellbar, neben der inhaltlichen Prüfung auf Korrektheit von PAP und C-Programm auch den Aufbau des PAP an sich auf korrekte Verwendung der Kontrollstrukturen hin zu prüfen. Die von draw.io zur Verfügung gestellte XML-Datei weist dafür bereits mit Informationen zur geometrischen Form innerhalb eines Tags das notwendige Potenzial auf. Dies würde ein vertiefendes Lernen zur Thematik Programmablaufplan ermöglichen.

Alternativ könnte auch eine eigene Anwendung zur grafischen Erstellung der PAPs mit einer Schnittstelle zur Bibliothek geschaffen werden, um ein vollumfängliches Tool-Set bereitstellen zu können, welches unabhängig von draw.io und im Umfang an die PAP-Erstellung angepasst ist. Dies könnte auch innerhalb der bestehenden .NET-Anwendung geschehen, in die die Bibliothek zukünftig integriert werden soll.

Denkbar ist auch die Implementierung einer Funktion innerhalb der Bibliothek, welche es ermöglicht aus Kontrollflussgraphen grafische Programmablaufpläne zu generieren, um diese in der Lehre bei der Wissensvermittlung einzusetzen.

Darüber hinaus könnte die Bibliothek um Funktionen zur Visualisierung der Kontrollflussgraphen und der Berechnungsschritte der Graph Edit Distance erweitert werden. Dies würde für eine bessere Nachverfolgbarkeit bei der Ähnlichkeitsberechnung von C-Programm und PAP sorgen.

Auch eine Erweiterung um die Auswahl zusätzlicher Metriken der Ähnlichkeitsbestimmung sind denkbar, um ein breiteres Spektrum an verschiedenen Kriterien neben der Nachbarschaft und den Knoteninhalten bei der Bestimmung abdecken und somit flexibel auch andere Graphen–arten untersuchen zu können.

Die Herausforderung, die jedoch bei allen Arten von Erweiterungen bestehen bleibt, ist die effiziente Bearbeitung von großen Graphen. Die .NET-Bibliothek ist für den Vergleich einfacher, kleiner Graphen gedacht, daher sind die zugrunde liegenden Algorithmen auf Einfachheit und Wartbarkeit ausgelegt. Große Graphen erfordern aufgrund des Isomorphie-Problems optimierte Algorithmen, um effektiv die Ähnlichkeit bestimmen zu können.

Literaturverzeichnis

References

[1] J. M. Leimeister, *Einführung in die Wirtschaftsinformatik*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021.

[2] C. Büsing, *Graphen- und Netzwerkoptimierung*. Heidelberg: Spektrum Akad. Verl., 2010. [Online]. Available: http://www.springerlink.com/content/n167n2

[3] H. Ernst, J. Schmidt, and G. Beneken, *Grundkurs Informatik*. Wiesbaden: Springer Fachmedien Wiesbaden, 2020.

[4] H. Knebl, *Algorithmen und Datenstrukturen*. Wiesbaden: Springer Fachmedien Wiesbaden, 2021.

[5] H.-P. Gumm, *Programmierung, Algorithmen und Datenstrukturen*. Berlin: De Gruyter, 2016. [Online]. Available: http://gbv.eblib.com/patron/FullRecord.aspx?p=4707913

[6] L. Priese and K. Erk, *Theoretische Informatik: Eine umfassende Einführung,* 4th ed. Berlin: Springer Vieweg, 2018.

[7] V. Turau and C. Weyer, *Algorithmische Graphentheorie: Deterministische und randomisierte Algorithmen,* 5th ed. Berlin, Boston: De Gruyter, 2024.

[8] W. Dörfler and J. Mühlbacher, *Graphentheorie für Informatiker*. Berlin: De Gruyter, 1973.

[9] O. Deiser, "Einführung in die Mathematik 2.1: Elementare Zahlentheorie und Graphentheorie," Oct. 2022. Accessed: Nov. 17 2024. [Online]. Available: https://www.aleph1.info/?call=Puc&permalink=ema21

[10] M. Grohe and P. Schweitzer, "The graph isomorphism problem," *Commun. ACM*, vol. 63, no. 11, pp. 128–134, 2020, doi: 10.1145/3372123.

[11] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," Jan. 2013. [Online]. Available: http://arxiv.org/pdf/1301.1493v1

[12] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento, "A Comparison of Algorithms for Maximum Common Subgraph on Randomly Connected Graphs," in *Lecture Notes in Computer Science, Structural, Syntactic, and Statistical Pattern Recognition*, G. Goos et al., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 123–132.

[13] E. Duesbury, J. D. Holliday, and P. Willett, "Maximum Common Subgraph Isomorphism Algorithms," The University of Sheffield, 2016. Accessed: Nov. 17 2024. [Online]. Available: http://match.pmf.kg.ac.rs/content77n2.htm

[14] P. Vismara and B. Valery, "Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms," in *Communications in Computer and Information Science, Modelling, Computation and Optimization in Information Systems and Management Sciences*, H. an Le Thi, P. Bouvry, and T. Pham Dinh, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 358–368.

[15] F. Serratosa, "Redefining the Graph Edit Distance," *SN COMPUT. SCI.*, vol. 2, no. 6, 2021, doi: 10.1007/s42979-021-00792-5.

[16] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems," in *Proceedings of the International Conference on Pattern Recognition Applications and Methods*, Lisbon, Portugal, 2015, pp. 271–278.

[17] S. O. Krumke and H. Noltemeier, *Graphentheoretische Konzepte und Algorithmen*. Wiesbaden: Vieweg+Teubner Verlag, 2012.

[18] M. Ferrer, F. Serratosa, and K. Riesen, "Improving bipartite graph matching by assessing the assignment confidence," *Pattern Recognition Letters*, vol. 65, pp. 29–36, 2015, doi: 10.1016/j.patrec.2015.07.010.

[19] J. Kim, "Efficient Top-k Graph Similarity Search With GED Constraints," *IEEE Access*, vol. 10, pp. 79180–79191, 2022, doi: 10.1109/ACCESS.2022.3194559.

[20] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Anal Applic*, vol. 13, no. 1, pp. 113–129, 2010, doi: 10.1007/s10044-008-0141-y.

[21] H.-P. Gumm and M. Sommer, *Formale Sprachen, Compilerbau, Berechenbarkeit und Komplexität*. Berlin, Boston: De Gruyter, 2019. [Online]. Available: https://www.degruyter.com/isbn/9783110442397

[22] I. S. Putra, S. A. Rukmono, and R. S. Perdana, "Abstract Syntax Tree (AST) and Control Flow Graph (CFG) Construction of Notasi Algoritmik," in *2021 International Conference on Data and Software Engineering (ICoDSE)*, Bandung, Indonesia, 2021, pp. 1–6.

[23] U. Meyer, *Grundkurs Compilerbau,* 2nd ed. Bonn: Rheinwerk Verlag; Rheinwerk Computing, 2024.

[24] K. Wang, M. Yan, H. Zhang, and H. Hu, "Unified abstract syntax tree representation learning for cross-language program classification," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, Virtual Event, 2022, pp. 390–400.

[25] V. Suttichaya, N. Eakvorachai, and T. Lurkraisit, "Source Code Plagiarism Detection Based on Abstract Syntax Tree Fingerprintings," in *2022 17th International Joint Symposium on Artificial Intelligence and Natural Language Processing (iSAI-NLP)*, Chiang Mai, Thailand, 2022, pp. 1–6.

[26] Ultipa, *Jaccard Similarity - Analytics & Algorithms - Ultipa Graph.* [Online]. Available: https://www.ultipa.com/document/ultipa-graph-analytics-algorithms/jaccard-similarity/v5.0 (accessed: Nov. 18 2024).

[27] S. Fletcher and M. Z. Islam, "Comparing sets of patterns with the Jaccard index," *AJIS*, vol. 22, 2018, doi: 10.3127/ajis.v22i0.1538.

[28] S. Ioffe, "Improved Consistent Sampling, Weighted Minhash and L1 Sketching," in *2010 IEEE International Conference on Data Mining*, Sydney, Australia, 2010, pp. 246–255.

[29] B. P. Gond, M. Shahnawaz, Rajneekant, and D. P. Mohapatra, "NLP-Driven Malware Classification: A Jaccard Similarity Approach," in *2024 IEEE International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS)*, Bangalore, India, 2024, pp. 1–8.

[30] Y. Chen, Z. Fan, Z. Chen, and Y. Zhu, "CA-Jaccard: Camera-aware Jaccard Distance for Person Re-identification," in *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Seattle, WA, USA, 2024, pp. 17532–17541.

[31] T. P. Rinjeni, A. Indriawan, and N. A. Rakhmawati, "Matching Scientific Article Titles using Cosine Similarity and Jaccard Similarity Algorithm," *Procedia Computer Science*, vol. 234, pp. 553–560, 2024, doi: 10.1016/j.procs.2024.03.039.

[32] M. Mohr, "Systematic Approaches to Advanced Information Flow Analysis – and Applications to Software Security," Dissertation, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie (KIT), Karlsruhe, 2023.

[33] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev, "Generating Customised Control Flow Graphs for Legacy Languages with Semi-Parsing," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Limassol, Cyprus, 2022, pp. 523–532.

[34] P. K. Tiwari, "Malware Detection Using Control Flow Graphs," in *2024 2nd International Conference on Device Intelligence, Computing and Communication Technologies (DICCT)*, Dehradun, India, 2024, pp. 216–220.

[35] Deutsches Institut für Normung, *Software-Entwicklung, Programmierung, Dokumentation: Normen,* 3rd ed. Berlin: Beuth, 1989.

[36] OMG, *About the Unified Modeling Language Specification Version 2.5.1.* [Online]. Available: https://www.omg.org/spec/UML/2.5.1/About-UML (accessed: Dec. 5 2024).

[37] A. Chopade, V. Shingde, A. Chavare, and T. Bhagwat, "Code Insight - Flowchart Generator," in *2024 2nd International Conference on Computer, Communication and Control (IC4)*, Indore, India, 2024, pp. 1–6.

[38] D. Hooshyar, R. B. Ahmad, Md Nasir, Mohd Hairul Nizam, and W. C. Mun, "Flowchart-based Approach to Aid Novice Programmers: A Novel Framework," in *2013 Conference on Future Internet Communications (CFIC 2013): Coimbra, Portugal, 15 - 16 May 2013*, 2013.

[39] M. Mhatre, A. Pandey, H. Rane, and S. Sahu, "A Novel Approach for Creating Flowcharts using Generative AI," in *2024 Asia Pacific Conference on Innovation in Technology (APCIT)*, MYSORE, India, 2024, pp. 1–7.

[40] C. Niebler, "Overcome learning obstacles in circuit network analysis with flowcharts," in *2023 32nd Annual Conference of the European Association for Education in Electrical and Information Engineering (EAEEIE)*, Eindhoven, Netherlands, 2023, pp. 1–6.

[41] Ellson, John: Gansner, Emden, Y. Hu, S. North, M. Jacobsson, M. Fernandez, and M. Hansen, *Graphviz*: Open Source. Accessed: Nov. 21 2024. [Online]. Available: https://graphviz.org/

[42] GCC Team, *G++ and GCC (Using the GNU Compiler Collection (GCC)).* [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/G_002b_002b-and-GCC.html (accessed: Nov. 22 2024).

[43] LLVM Admin Team, *The LLVM Compiler Infrastructure Project.* [Online]. Available: https://llvm.org/ (accessed: Nov. 22 2024).

[44] GCC Team, *A GNU Manual.* [Online]. Available: https://gcc.gnu.org/onlinedocs/ (accessed: Nov. 17 2024).

[45] Gaudenz Alder, *draw.io*: JGraph Ltd and draw.io AG. Accessed: Nov. 21 2024. [Online]. Available: https://www.drawio.com/about

[46] Microsoft Corporation, *MS Visio Professional 2021*: Microsoft Corporation. Accessed: Nov. 22 2024. [Online]. Available: https://www.microsoft.com/de-DE/microsoft-365/visio/flowchart-software

[47] Friedrich Folkmann, *PapDesigner*: Georg-Simon-Ohm Berufskolleg in Köln. Accessed: Nov. 22 2024. [Online]. Available: http://friedrich-folkmann.de/papdesigner/Hauptseite.html

[48] GCC Team, *Developer Options (Using the GNU Compiler Collection (GCC)).* [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html (accessed: Nov. 15 2024).

[49] GCC Team, *C Dialect Options (Using the GNU Compiler Collection (GCC)).* [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html#index-fno-builtin (accessed: Nov. 22 2024).

[50] GCC Team, *Other Builtins (Using the GNU Compiler Collection (GCC)).* [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (accessed: Nov. 30 2024).

[51] Adegeo, *Sprachelemente für reguläre Ausdrücke – Kurzübersicht - .NET.* [Online]. Available: https://learn.microsoft.com/de-de/dotnet/standard/base-types/regular-expression-language-quick-reference (accessed: Nov. 21 2024).

[52] A. Koch, "Optimierende Compiler: Kontrollflussgraphen und Static Single Assignment-Form," Darmstadt, SoSe 2008. Accessed: Nov. 17 2024. [Online]. Available: https://www.esa.informatik.tu-darmstadt.de/archive/twiki/pub/Lectures/OptimierendeCompiler11De/cfgssa-handout.pdf

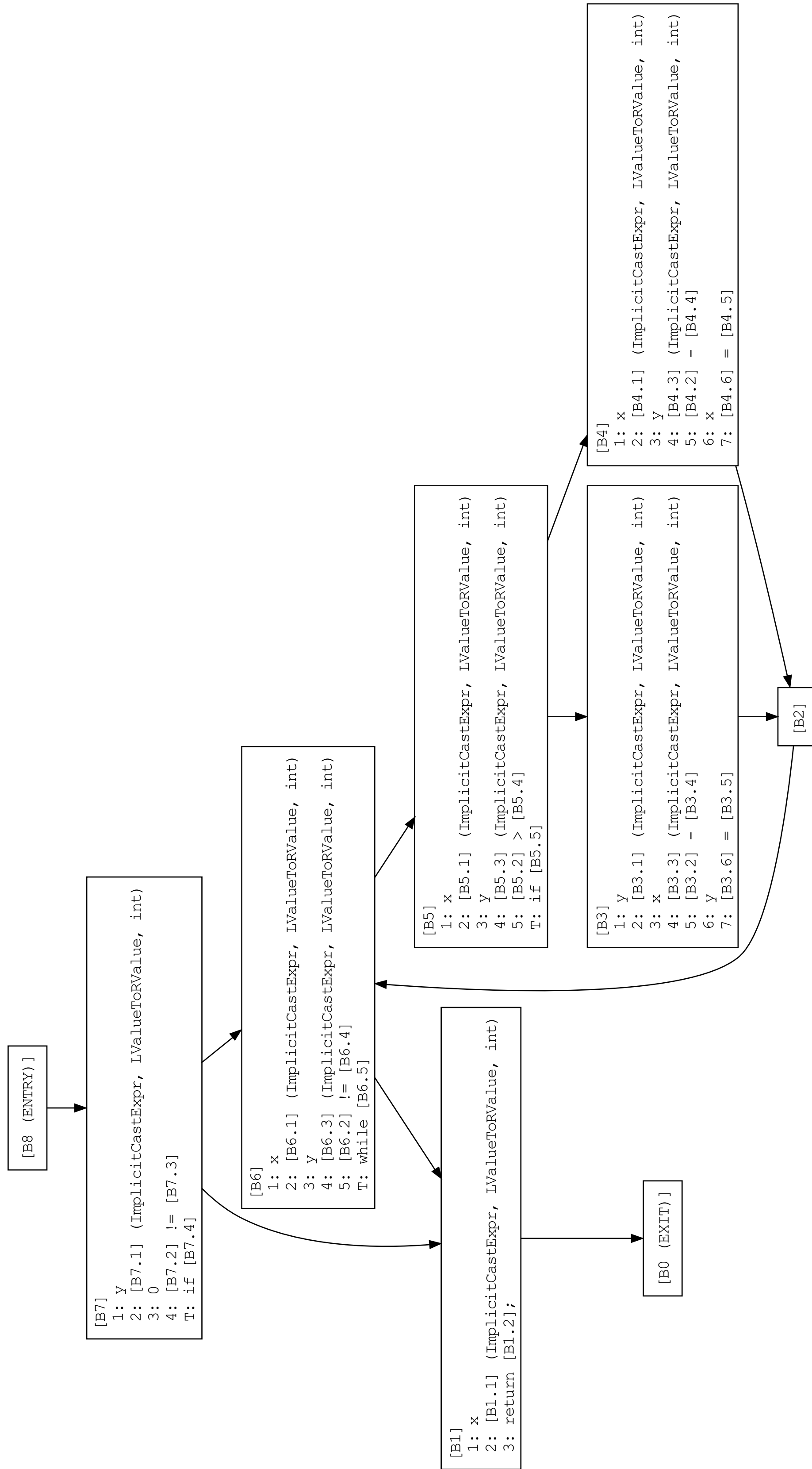
[53] FREEFORMATTER, *Free Online XSD/XML Schema Generator - FreeFormatter.com.* [Online]. Available: https://www.freeformatter.com/xsd-generator.html (accessed: Dec. 1 2024).

[54] BillWagner, *Das .NET Compiler Platform SDK (Roslyn APIs).* [Online]. Available: https://learn.microsoft.com/de-de/dotnet/csharp/roslyn-sdk/ (accessed: Dec. 4 2024).

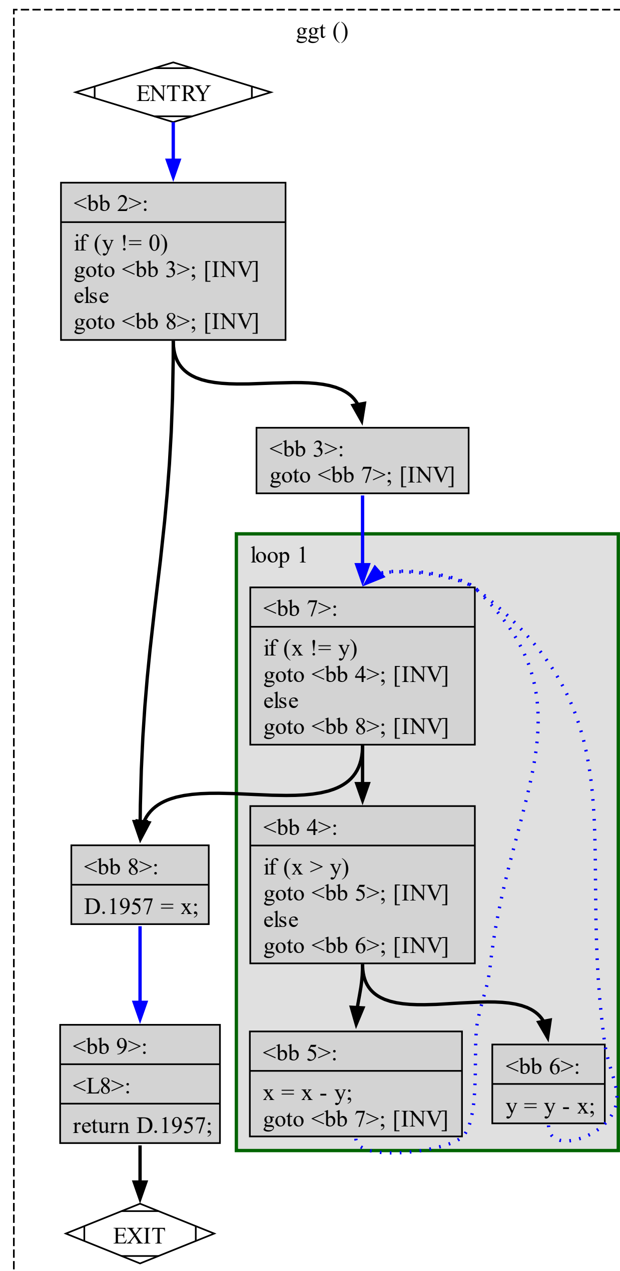
[55] BillWagner, *Verwenden des .NET Compiler Platform SDK-Syntaxmodells - C#.* [Online]. Available: https://learn.microsoft.com/de-de/dotnet/csharp/roslyn-sdk/work-with-syntax (accessed: Dec. 4 2024).

Anhang

A1 Vollständiger Kontrollflussgraph im LLVM (Clang)-Compiler



A2 Vollständiger Kontrollflussgraph im GCC-Compiler



A3 Testreihen zur Laufzeitermittlung von Positionsvergleichen

Durchlauf	Knoten #	Ansatz 2 (s)	Ansatz 1 (s)	Knoten #	Ansatz 2 (s)	Ansatz 1 (s)	Knoten #	Ansatz 2 (s)	Ansatz 1 (s)	Knoten #	Ansatz 2 (s)	Ansatz 1 (s)	Knoten #	Ansatz 2 (s)	Ansatz 1 (s)
1	100	1,23	1,77	200	1,66	10,71	300	2,20	36,19	400	2,65	86,10	500	3,32	197,02
2	100	0,33	1,56	200	0,81	11,51	300	1,12	30,38	400	1,78	84,28	500	2,54	196,74
3	100	0,34	1,77	200	0,74	11,07	300	1,06	29,11	400	1,63	76,43	500	1,99	177,43
4	100	0,41	1,62	200	0,91	16,40	300	1,00	30,37	400	1,61	81,25	500	2,35	191,36
5	100	0,36	1,60	200	1,03	11,14	300	0,97	28,87	400	1,62	75,54	500	2,26	202,92
6	100	0,34	1,47	200	0,82	12,30	300	1,06	33,51	400	1,35	77,26	500	2,47	190,88
7	100	0,37	1,47	200	0,97	14,00	300	1,10	31,15	400	1,66	79,05	500	2,42	209,66
8	100	0,42	1,69	200	0,67	10,62	300	1,10	35,74	400	1,65	80,20	500	2,28	205,94
9	100	0,35	1,53	200	0,72	10,71	300	1,00	39,66	400	1,57	78,53	500	2,24	195,13
10	100	0,33	1,35	200	0,71	10,03	300	1,21	36,47	400	1,54	72,22	500	2,24	186,30
11	100	0,39	1,64	200	0,83	10,81	300	1,21	36,76	400	1,55	79,81	500	2,25	192,68
12	100	0,31	1,53	200	0,81	12,60	300	1,24	31,43	400	1,35	80,88	500	2,36	198,93
13	100	0,40	1,77	200	0,68	10,45	300	1,18	28,84	400	1,65	77,47	500	2,03	188,60
14	100	0,35	1,66	200	0,71	9,56	300	1,06	30,65	400	1,61	73,10	500	2,50	199,52
15	100	0,28	1,37	200	0,92	11,75	300	1,17	32,17	400	1,64	82,42	500	2,24	182,31
16	100	0,41	1,60	200	0,76	11,62	300	1,10	29,67	400	1,60	89,86	500	2,41	185,46
17	100	0,37	1,46	200	0,71	9,74	300	1,00	27,45	400	1,65	80,75	500	2,45	197,92
18	100	0,37	1,53	200	0,70	9,82	300	1,01	26,27	400	1,57	87,49	500	2,34	192,85
19	100	0,33	1,40	200	0,69	9,91	300	1,11	30,62	400	1,53	87,97	500	2,27	180,60
20	100	0,39	1,51	200	0,68	9,66	300	1,14	41,04	400	1,68	77,71	500	2,38	191,58
21	100	0,35	1,42	200	0,82	11,80	300	1,14	37,95	400	1,25	82,44	500	2,46	220,58
22	100	0,32	1,45	200	0,82	11,25	300	1,07	37,43	400	1,48	90,14	500	2,39	210,60
23	100	0,30	1,35	200	0,66	9,60	300	1,11	35,59	400	1,60	75,47	500	2,33	197,18
24	100	0,35	1,56	200	0,82	10,53	300	1,52	36,79	400	1,37	71,51	500	2,28	189,29
25	100	0,34	1,36	200	0,72	10,66	300	0,91	29,49	400	1,40	76,55	500	2,26	177,73
26	100	0,33	1,48	200	0,80	11,89	300	1,21	39,53	400	1,84	77,69	500	1,74	185,83
27	100	0,38	1,50	200	0,65	10,88	300	1,11	40,85	400	1,57	82,40	500	2,42	196,07
28	100	0,36	1,71	200	1,11	11,77	300	1,16	38,35	400	1,61	74,09	500	2,07	177,03
29	100	0,38	1,71	200	0,78	13,51	300	1,13	38,19	400	1,73	78,90	500	2,07	175,78
30	100	0,36	1,63	200	0,97	10,33	300	1,06	38,28	400	1,88	85,14	500	2,27	189,88
31	100	0,39	1,52	200	0,79	12,97	300	1,22	36,48	400	1,69	89,78	500	2,24	194,69
32	100	0,31	1,30	200	1,00	11,20	300	1,27	38,92	400	1,68	84,85	500	1,75	154,09
33	100	0,31	1,47	200	0,71	12,61	300	1,12	31,88	400	1,43	81,43	500	2,09	167,57
34	100	0,37	1,57	200	1,08	11,73	300	1,19	35,07	400	1,74	93,28	500	1,88	168,27
35	100	0,32	1,53	200	0,85	12,50	300	1,27	36,98	400	1,77	89,55	500	2,16	180,82
36	100	0,38	1,54	200	0,98	12,74	300	1,09	37,16	400	1,47	77,62	500	2,39	177,86
37	100	0,34	1,55	200	0,76	12,54	300	1,32	38,85	400	1,60	89,61	500	2,11	170,79
38	100	0,34	1,81	200	1,05	11,49	300	1,22	33,60	400	1,58	85,29	500	2,41	182,35
39	100	0,30	1,36	200	0,72	13,12	300	1,13	33,33	400	1,94	91,74	500	1,97	171,62
40	100	0,39	1,65	200	0,90	12,31	300	1,18	34,61	400	1,69	82,95	500	1,88	159,84
41	100	0,36	1,52	200	0,77	12,95	300	1,18	32,40	400	1,44	76,29	500	1,99	183,68
42	100	0,37	1,52	200	0,86	12,43	300	1,06	33,46	400	1,64	87,22	500	2,15	171,42
43	100	0,35	1,76	200	0,76	13,26	300	1,15	36,93	400	1,56	82,88	500	2,29	182,93
44	100	0,39	1,63	200	1,01	13,01	300	1,20	38,08	400	1,67	73,60	500	2,29	194,10
45	100	0,35	1,63	200	0,78	14,41	300	1,10	33,78	400	1,71	88,78	500	1,99	178,86
46	100	0,40	1,60	200	1,25	12,39	300	1,19	41,49	400	1,69	76,67	500	2,12	171,97
47	100	0,40	1,69	200	0,90	13,67	300	1,17	34,72	400	1,55	78,97	500	2,05	174,22
48	100	0,31	1,41	200	0,77	12,51	300	1,04	34,71	400	1,50	77,30	500	2,39	176,17
49	100	0,31	1,31	200	0,82	11,45	300	1,10	35,76	400	2,03	89,36	500	2,14	178,99
50	100	0,38	1,61	200	0,74	10,61	300	1,28	40,73	400	1,72	89,10	500	2,03	170,69
Mittelwert		**0,37**	**1,55**		**0,84**	**11,73**		**1,16**	**34,75**		**1,63**	**81,78**		**2,24**	**185,89**

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als die angegebenen verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Halle (Saale), 10.12.2024________________ ____________________________________

(Ort, Datum) (Unterschrift Sören Taube)