Bachelorarbeit

TheobaldSubdue: Compression-based Algorithm for Subgraph Mining of Edit Operations in Model Repositories

vorgelegt von

David Theobald

Betreuer:	Prof. Dr. Fuchß
Zweitgutachter:	Prof. Dr. Link
Beginn der Arbeit:	30.03.2021
Abgabedatum:	12.08.2021

Betreuung:
Saarland University
Christof Tinnes

Inhaltsangabe

Aufgrund der immer weiter zunehmenden Komplexität von Softwaresystemen ist es erforderlich, neue Lösungsansätze zur Komplexitätsreduktion zu entwickeln. Insbesondere im Model Driven Engineering wird versucht, durch weitere Abstraktionsgrade anhand von Modellen die Entwicklung und Wartung von Softwaresystemen zu vereinfachen. Bearbeitungsschritte dieser Modelle in einem Model Repository werden durch Edit Operations abgebildet, was zu einer Versionshistorie der Modelle führt. Über alle Versionen in einem Model Repository hinweg kann das Subgraph Mining dazu verwendet werden, die interessantesten Edit Operations zu identifizieren. Mit diesem Ansatz kann ein Generator für Edit Operations entwickelt werden, der dem Entwickler eine Auswahl von Vorschlägen für verschiedene Edit Operations anbietet, sodass der Entwickler weniger intellektuellen Aufwand aufwenden muss, um Änderungen an Modellen vorzunehmen. Somit soll ein Mehrwert geschaffen werden, um die Komplexität von Softwaresystemen zu reduzieren. In der vorliegenden Arbeit wird der Subgraph Mining Algorithmus Subdue für das Identifizieren interessanter Edit Operations herangezogen. Der Bereich des Graph Minings kann recht komplex erscheinen, insbesondere existiert wenig Literatur die versucht, die Thematik auf einfache Art und Weise verständlich zu machen. Aus diesem Grund wird zunächst der kompressionsbasierte Ansatz des Subdue Algorithmus möglichst einfach dargestellt. Zudem wird ein Debugging Tool entwickelt, mit dem die einzelnen Bearbeitungsschritte Schritt für Schritt nachvollzogen werden können. Es existieren offizielle Implementierungen des Subdue Algorithmus in den Programmiersprachen C und Python, die jedoch unterschiedliche Ergebnisse liefern. Daher wird eine erste Pilotstudie über die Performance beider Implementierungen geführt. Anschließend werden Änderungen an dem Subdue Algorithmus vorgenommen, die zu besseren Ergebnissen für den Anwendungsfall des Subgraph Minings von Edit Operations in Model Repositories führen. Insbesondere wird eine neue Metrik vorgeschlagen, die auf Model Repositories bessere Ergebnisse erzielen kann. Diese Änderungen führen zu dem in der Arbeit vorgelegten TheobaldSubdue Algorithmus. In einem Experiment auf simulierten Model Repositories werden die Ergebnisse des TheobaldSubdue mit dem des Subdue und dem Frequent Subgraph Mining Algorithmus Gaston gemessen. Die Ergebnisse hierbei zeigen, dass der TheobaldSubdue auf dem verwendeten Datensatz bessere Ergebnisse für die Identifikation von Edit Operations in Model Repositories als der Subdue erzielen kann, allerdings bleibt der Ansatz weiterhin minimal schlechter als der Gaston Algorithmus.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Aufgrund der immer komplexer werdenden Softwaresysteme ist es notwendig, die Komplexität dieser Systeme zu bewältigen und neue Lösungsansätze zur Komplexitätsreduktion zu entwickeln. Hierfür eignen sich insbesondere die Ansätze aus dem Model Driven Engineering, die mit Modellen statt mit Quellcode arbeiten. Die wichtigsten Aspekte auf einen Blick des Model Driven Engineering lassen sich aus der Publikation *The State of Practice in Model-Driven Engineering* [1] entnehmen. Mithilfe der Modelle lassen sich dann weitere Artefakte wie bspw. den Quellcode und Dokumentationen für das Softwaresystem generieren. Die Modelle werden in sogenannten Model Repositories verwaltet, die ebenfalls wie gängige Code Repositories eine Versionierung aufweisen. Die Versionierung gibt Aufschluss darüber, wie sich die Modelle über die Zeit verändert haben. In der Publikation *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] von C. Tinnes wird ein Ansatz vorgeschlagen, wie sich aus diesen Model Repositories Edit Operations ermitteln lassen. Edit Operations repräsentieren dabei die Editiervorgänge an den Modellen. Mit dem Ansatz von C. Tinnes lassen sich die Modellunterschiede in den Model Repositories als Graphen extrahieren, womit sich anschließend mittels Graph Mining die interessantesten Edit Operations identifizieren lassen. Das Ziel von Graph Mining, insbesondere dem Subgraph Mining ist es, signifikantes und nützliches Wissen aus Graphdatensätzen zu generieren. Für diese Aufgabe existieren bereits zahlreiche Subgraph Mining Algorithmen, mit denen sich interessante Muster und somit nützliches Wissen aus Daten ableiten lässt. Eine Übersicht über die bekanntesten Subgraph Mining Algorithmen kann aus der Publikation *Frequent Subgraph Mining Algorithms - A Survey and Framework for Classification* [3] entnommen werden. Für das Identifizieren von Edit Operations in Model Repositories wird in der vorliegenden Arbeit der kompressionsbasierte SUBDUE Algorithmus untersucht, der in der Veröffentlichung *Discovery of Inexact Concepts from Structural Data* [4] von L. Holder und D. Cook vorgestellt wird. Anders als andere Subgraph Mining Algorithmen, berechnet der SUBDUE Algorithmus für die Bestimmung der interessanten Muster einen Kompressionswert. Aus der Publikation *The Minimum Description Length Principle* [5] von P. Grünwald lässt sich entnehmen, dass Kompression ebenfalls ein Maß dafür, wie interessant ein Muster in den Daten ist. Bisherige Untersuchungen von C. Tinnes haben gezeigt, dass ein kompressionsbasierter Ansatz gut geeignet ist, um Edit Operations zu identifizieren. Daher scheint der SUBDUE Algorithmus ein geeigneter Kandidat für das Subgraph Mining von Edit Operations in Model Repositories zu sein. Diese Annahme wird in dieser

Arbeit durch Anwendung des SUBDUE Algorithmus auf simulierte Model Repository Datensätze in einem Experiment untersucht. Es ist keine Literatur zu finden, die den SUBDUE Algorithmus im Detail einfach verständlich vorstellt. Daher werden die Konzepte und das Verfahren des SUBDUE Algorithmus in dieser Arbeit vorgestellt. Zudem wird ein Debugging Tool für den SUBDUE Algorithmus entwickelt, mit dem sich das Verfahren schrittweise nachvollziehen lässt. Der Anwendungsfall von Subgraph Mining in Model Repositories bringt einige besondere Eigenschaften mit sich. Diese können ausgenutzt werden, um die Mining Ergebnisse zu optimieren. Zu diesem Zweck wird in dieser Arbeit der THEOBALDSUBDUE Algorithmus vorgeschlagen, der auf dem SUBDUE Algorithmus basiert. In einem abschließenden Experiment werden die Ergebnisse und die Performance des THEOBALDSUBDUE, sowie des SUBDUE Algorithmus gemessen und miteinander verglichen.

1.1 Motivation

Schon in den späten 60er Jahren waren die Anzeichen deutlich, dass es erhebliche Aufwände mit sich bringt, große und skalierbare Softwaresysteme zu entwickeln. Diese Erfahrung wurde von P. Naur und B. Randell 1979 in dem Bericht der NATO *Software Engineering: As it was in 1968* [6] festgehalten. Dort wurde unter anderem berichtet, dass ein Softwaresystem schwer zu beherrschen ist. Wenn die Komplexität nicht ausreichend reduziert wird, gefährdet dies die erfolgreiche Realisierung des Systems. Um diese Herausforderung bestmöglich zu bewältigen, kann man sich unterschiedlicher Techniken und Methoden bedienen. Ein effektiver Software Reuse Ansatz ist hierfür eine leistungsstarke Methode, um gegen Systemkomplexität vorzugehen. Der Software Reuse Ansatz wird in der Publikation *Software Reuse* [7] von C. Krueger vorgestellt. Das Ziel von Software Reuse ist, schon existierende Komponenten aus anderen Softwaresystemen für die Entwicklung eines neuen Softwaresystems einzusetzen, um somit die Komplexität zu reduzieren und die Produktivität zu steigern. Insbesondere jene Software Reuse Ansätze mit dem Merkmal der Abstraktion nehmen eine zentrale Rolle in der Komplexitätsreduktion ein. Aktueller Forschungsgegenstand der Dissertation von C. Tinnes am Lehrstuhl für Software Engineering an der Universität des Saarlandes ist, diese Systemkomplexität für den Softwareentwickler in seiner Tätigkeit des Programmierens mithilfe von Software Reuse und neuen Abstraktionsmodellen zu reduzieren. Wenn ein Softwareentwickler seiner Arbeit nachgeht, muss er einen gewissen intellektuellen Aufwand erbringen. Soll bspw. eine neue Funktion in das Softwaresystem implementiert werden, lässt sich dieser kognitive Aufwand wie eine Distanz betrachten, die vom Startpunkt bis hin zum Zielpunkt durchlaufen werden muss. Die grundsätzliche Annahme hinter dem Ansatz von C. Tinnes lautet, dass sich durch die große kognitive Distanz zwischen Quellcode und dem fertigen Softwaresystem noch weitere Abstraktionskonzepte befinden, die noch nicht ausreichend erforscht sind. Ein übergreifendes Konzept über alle Abstraktionslayer hinweg könnten sogenannte Edit Operations darstellen. Edit Operations sind in der Lage, die Bearbeitungsschritte von Softwareentwicklern abzubilden, die in einem Softwaresystem angewandt werden, um Änderungen durchzuführen. Eine Edit Operation kann dabei ein beliebig großes Konzept über eine oder mehreren beliebigen Abstraktionsstufen zusammenfassen. Eine kleine Edit Operation auf der Ebene des Quellcodes kann bspw. das einfache Einfügen eines Statements zu einer Methode sein. Es sind aber auch weitaus größere Edit Operations vorstellbar auf

höheren Abstraktionsebenen wie bspw. das Zusammenfassen aller Komponenten für eine Datenbankanwendung, welches als vollständiges Konzept auf Architekturebene in ein neues Softwaresystem integriert werden kann, ohne die Implementierungsdetails kennen zu müssen. Mit anderen Worten, mithilfe von Edit Operations könnte es ermöglicht werden, beliebige Konzepte aus Softwaresystemen abzubilden, um damit die Entwicklung von neuen Softwarekomponenten und Systemen effizienter zu gestalten. Ein Konzept im oben genannten Sinne kann also mit einer Menge von Edit Operations dargestellt werden. Dem Softwareentwickler könnte so auf Basis erlernter Konzepte, die durch Edit Operations abgebildet werden, Vorschläge generieren, sodass sich wiederholende Konzepte mit geringerem intellektuellem Aufwand implementieren lassen. Das bedeutet, ein Softwareentwickler muss ein Konzept, welches zuvor erlernt wurde, nicht von Grund auf neu implementieren, sondern bekommt während dem Programmieren erkannte Konzepte angezeigt, die er nun wiederverwenden kann. Realisieren ließe sich das bspw. mit einem Generator, der anhand bisher erlernter Konzepte anhand von Edit Operations, dem Programmierer geeignete Vorschläge anzeigt. Der Programmierer kann den für sich zutreffende Vorschlag auswählen, um somit die Implementierung des Konzepts durchzuführen.

Im Model Driven Engineering verfolgt man schon seit jeher den Ansatz, Komplexität von Softwaresystemen beherrschbar zu machen, indem mithilfe von Modellierungstechniken versucht wird, Softwareentwicklungsprozesse zu strukturieren, zu vereinheitlichen und automatisiert auszuführen. Dies bestätigen auch S. Sendall und W. Kozaczynski in ihrer Veröffentlichung *Model Transformation – the Heart and Soul of Model-Driven Software Development* [8].

> The motivation behind model-driven software development is to move the focus of work from programming to solution modeling. The model-driven approach has a potential to increase development productivity and quality [...]
>
> (S. Sendall und W. Kozaczynski)

Mit den Werkzeugen im Model Driven Engineering werden Modelle erzeugt, mit denen Softwaresysteme abstrahiert werden können, sodass weniger Detailwissen über das System und deren Komponenten notwendig ist. Die generierten Modelle können auch wiederverwendet werden und dienen somit als Ansatz zur Wiederverwendung von Softwarekomponenten in Model Repositories. Im Unterschied zu einem Software Repository, das nur Quellcode beinhaltet, enthält ein Model Repository weitere Abstraktionsstufen auf Basis weiterer Artefakte. Mithilfe dieser erweiterten abstrakten Sicht wird sowohl die Fähigkeit unterstützt, die Funktionalität des Softwaresystems für alle am System Beteiligten verständlicher zu machen, als auch die Möglichkeiten, Dokumentation und Analyse automatisch durchzuführen. Ein Model Repository verwaltet also nicht nur den Quellcode und dessen lose gekoppelte Artefakte, sondern kann alle relevanten Artefakte zusätzlich miteinander verknüpfen. Dies kann dazu führen, dass die Modelle im Laufe der Zeit zu umfangreichen Modellen anwachsen, weshalb Model Repositories in kürzester Zeit zu großen Datenkomplexen werden. Dies zeigt auch die Fallstudie eines Softwaresystems aus der Industrie, die in *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] von C. Tinnes durchgeführt wird. Das dort vorhandene Model

Repository ist in mehr als 6 Jahren auf eine Gesamtgröße von 300GB herangewachsen, was in diesem Model Repository ca. 1,2 Millionen Elementen entspricht. Die Arbeit mit solch großen Model Repositories kann durch diese Größe ineffizient werden, da zur lokalen Entwicklung immer wieder eine große Menge an Daten heruntergeladen werden müssen. Dies führt nicht nur zu langen Wartezeiten während der Entwicklung, sondern auch zu einem erhöhtem Datenverkehr, der wiederum zusätzliche Kosten verursacht. Der oben erwähnte Ansatz zur Erkennung übergeordneter Konzepte durch von Edit Operations könnte zu interessanten Optimierungen in Model Repositories führen, da Model Driven Engineering häufig für die Entwicklung domänenspezifischer Systemen angewandt wird. Model Repositories beinhaltet dadurch eine große Menge an Artefakten und Daten der Domäne. Aus diesem Grund ist es durchaus interessant, Edit Operations in Model Repositories ausfindig zu machen, da die gefundenen Konzepte mit hoher Wahrscheinlichkeit von domänenspezifischer Natur sind. Schafft man es, diese domänenspezifischen Konzepte zu erlernen und dem Entwickler zum richtigen Zeitpunkt vorzuschlagen, könnte die Produktivität durch den verringerten intellektuellen Aufwand gesteigert werden. Durch diesen neu gewonnenen Software Reuse Ansatz in Model Respositories könnte ebenso die kognitive Distanz in der Entwicklung und die Systemkomplexität reduziert werden. Die gefundenen Konzepte durch Edit Operations könnten sich zudem positiv auf die schon erwähnten Nachteile der enormen Größe von Model Repositories auswirken. Das Herunterladen der Modelle zur lokalen Entwicklung könnte ebenfalls optimiert werden, indem die Modelle auf Basis der gelernten Edit Operations komprimiert werden. Dies würde die Wartezeiten für das Herunterladen der Modelle reduzieren, sowie den dadurch verursachten Datenverkehr minimieren. Um diese Annahmen zu verifizieren, bedarf es weiterer Forschungstätigkeiten und sind nicht Teil dieser Arbeit.

Der Ansatz regelmäßige Konzepte in Daten zu erkennen ist nicht neu und wurde schon in den 90er Jahren unter dem Begriff des Concept Learnings von L. Holder und D. Cook in ihrer Arbeit *Discovery of Inexact Concepts from Structural Data* [4] und den daraus folgenden wissenschaftlichen Arbeiten publiziert. Forschungsgegenstand dieser Arbeiten ist die Identifizierung von sich wiederholenden Substrukturen in strukturierten Daten, die konzeptionell interessant erscheinen. Schon damals wurde in deren Veröffentlichung *An Empirical Study of Domain Knowledge and Its Benefits to Substructure Discovery* [9] das Potenzial erkannt, Substrukturen in Quellcode ausfindig zu machen, um regelmäßige und interessante Konzepte zu finden und zu modularisieren, sodass die Komplexität in der Entwicklung von Softwaresystemen reduziert werden kann. Diese Beobachtung stellt auch die Grundlage für die Dissertation von C. Tinnes. Ziel ist es, die Idee des Concept Learnings und der Identifizierung von Substrukturen aufzugreifen und auf Model Repositories anzuwenden. Somit soll die schon erwähnte Komplexitätsreduktion explizit in Model Repositories im Model Driven Engineering vorangetrieben werden, sodass auf dieser Basis interessante Edit Operations in einem Model Repository erkannt werden können.

1.2 Ziel der Thesis

Im Rahmen der Forschungstätigkeiten der Forschergruppe um L. Holder und D. Cook wurde auch ein neuer Algorithmus für das Subgraph Mining vorgeschlagen. Der SUBDUE Algorithmus identifiziert sich wiederholende und interessante Substrukturen in strukturellen Daten. Ursprünglich wurde der SUBDUE Algorithmus dazu entwickelt Muster unter Verwendung von Domänenwissen zu erkennen. Allerdings hat sich in der durchgeführten Evaluation in *An empirical study of domain knowledge and its benefits to substructure discovery* [9] herausgestellt, dass er nicht nur Konzepte in den konkreten Domänen Quellcode, CAD-Schaltungen, Analyse chemischer Verbindungen und Szenenanalyse identifizieren kann, sondern auch zuverlässig in General Purpose Datensätze interessante Ergebnisse liefern kann. Weitere Bemühungen mittels des Concept Learnings mittels Subgraph Mining, insbesondere mithilfe des SUBDUE Algorithmus Konzepte in Softwaresystemen zu erkennen, um damit Softwareentwickler in ihrer Arbeit zu unterstützen, scheint nicht weiter verfolgt worden zu sein. Es konnten auch keine Hinweise darauf gefunden werden, warum dieser Ansatz nicht weiter aufgegriffen wurde. Mit der zuvor genannten Annahme, dass das Vorschlagen von Edit Operations mit einem Generator die kognitiven Aufwände für Softwareentwickler deutlich reduziert und somit die Entwicklung und Wartung von Softwaresystemen effizient gesteigert werden kann, verspricht der Ansatz des Subgraph Minings für Edit Operations durchaus interessant zu sein. Aus diesem Grund stellt der Ansatz des Concept Learnings durch Subgraph Mining von Edit Operations im Model Driven Engineering eine besonders interessante Möglichkeit dar, Konzepte in Model Repositories zu identifizieren. Infolgedessen sollen die Forschungstätigkeiten von L. Holder und D. Cook aufgegriffen werden, um deren Ideen auf Model Repositories im Model Driven Engineering weiter zu untersuchen.

Trotz der scheinbar geeigneten Eigenschaften des SUBDUE Algorithmus für das Concept Learning in Model Repositories konnte er nicht die erhofften Erwartungen erfüllen. Dies geht auch aus der oben genannten Arbeit von C. Tinnes hervor. In einer ersten Pilotstudie hat sich gezeigt, dass andere Subgraph Mining Algorithmen auf Model Repositories wesentlich besser performen, als der SUBDUE Algorithmus. In dieser Studie werden unterschiedliche Graph Mining Algorithmen auf Model Repositories getestet, mit dem Ergebnis, dass der GASTON Algorithmus bisher am besten bzgl. der Laufzeit und der identifizierten Ergebnisse abschneidet. Der GASTON Algorithmus wurde erstmals von S. Nijssen und J. Kok in der Publikation *The Gaston Tool for Frequent Subgraph Mining* [10] im Jahr 2005 veröffentlicht. Der GASTON ist ein Frequent Subgraph Mining Algorithmus und identifiziert interessante Pattern anhand deren Häufigkeit im Eingabegraph. Aufgrund der unterschiedlichen Ansätze wird es interessant zu sehen sein, wie der GASTON im Vergleich zu dem kompressionsbasierten SUBDUE abschneidet. Die Annahme, dass der SUBDUE als geeignet erscheint, basiert darauf, dass er im Vergleich zu anderen Graph Mining Algorithmen auf dem Prinzip der Minimum Description Length zur Mustererkennung aufbaut. Das Buch *The Minimum Description Length Principle* [5] von P. Grünwald gilt als Standard Referenz für den Minimum Description Length Ansatz. Aus dem Standardwerk ist zu entnehmen, dass interessante Muster in Daten genau jene sind, die eine größtmögliche Komprimierung auf den Daten hervorrufen. Dieses Prinzip ist des-

wegen für den Anwendungsfall auf Model Repositories so interessant, da wie zuvor erwähnt, das oftmals domänenspezifisches Wissen in Model Repositories vorhanden ist. Dies könnte dazu führen, dass interessante domämenspezifische Konzepte in den Model Repositories solche sind, die zu einer hohen Kompression der Modelle führen würden.

Mit der vorliegenden Arbeit soll ein Beitrag zur Optimierung des SUBDUE Algorithmus für das Subgraph Mining von Edit Operations in Model Repositories geleistet werden. Es wird die Annahme getroffen, dass der SUBDUE mit entsprechenden Anpassungen für Model Repositories bessere Ergebnisse erzielen kann. Diese Anpassungen am SUBDUE führen zum Vorschlag des THEOBALDSUBDUE Algorithmus, mit dem das Subgraph Mining von Edit Operations in Model Repositories auf Basis des SUBDUE verbessert werden soll. Mit der im Anschluss folgenden Evaluation soll die Frage geklärt werden, ob der THEOBALDSUBDUE Algorithmus zu einem besseren Subgraph Mining Resultat führt oder ob auch mit den durchgeführten Anpassungen die Performance nicht gesteigert werden kann.

Um interessante Muster in Graphen zu finden haben sich besonders zwei Ansätze als erfolgreich erwiesen. Aus der Publikation *Graph-Based Knowledge Discovery: Compression vs. Frequency* [11] lässt sich entnehmen, dass unterschiedliche Arten von Graph Mining Algorithmen existieren. Jene die auf dem Prinzip des Frequent Subgraph Minings basieren und die interessante Subgraphen anhand derer Häufigkeit lernen. Die andere Klasse an Algorithmen identifiziert interessante Subgraphen auf Basis von Kompression. Die Veröffentlichung *The Minimum Description Length Principle* [5] erklärt, dass, je höher der Grad der Kompression ist, die durch den Subgraph am Graphen erwirkt wird, desto interessanter ist er. C. Tinnes führte in der Publikation *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] ebenfalls eine Pilotstudie durch, um einen ersten Überblick über mehrere Subraph Mining Algorithmen zu gewinnen. In seiner Pilotstudie werden Algorithmen beider Klassen getestet mit dem Ergebnis, dass der Frequent Subgraph Mining Algorithmus GASTON die beste Performance auf Model Respositories aufweist. Diese Pilotstudie wird aufgegriffen, in dem die Frequent Subgraph Mining Algorithmen GASTON, sowie der kompressionsbasierte Algorithmus SUBDUE mit dem vorgeschlagenen THEOBALDSUBDUE auf Performance verglichen werden. Hierfür werden die Laufzeiten und die Korrektheit der erkannten Pattern gemessen.

1.3 Kontext und Umfeld

Um die Ziele der vorliegenden Arbeit zu erreichen, sowie die Evaluation des THEOBALDSUBDUE im Vergleich zu anderen Subgraph Mining Algorithmen durchzuführen, werden die Performance und die Mining Ergebnisse auf vorhandene Datensätze getestet. Für die Evaluationen werden simulierte Model Repositories verwendet, die C. Tinnes für die Evaluation in seiner Arbeit generiert hat. Dazu wurden Datensätze generiert, auf die Subgraph Mining Algorithmen angewendet werden können, um Edit Operations zu identifizieren. Da die Datensätze selbst generiert wurden, ist die zu identifizierende Edit Operation bekannt, sodass die erfolgreiche Identifizierung

der korrekten Edit Operation gemessen werden kann. Für die Datensätze wurden eine Reihe von Model Repositories wie folgt generiert. Ein Datensatz beinhaltet mehrere simulierte Model Repositories, denen jeweils ein Komponentenmodell auf Basis eines Metamodells zugrunde liegt. Die zu identifizierenden Edit Operations wurden iterativ auf dieses Komponentenmodell angewendet. Zusätzlich wurde mit einer bestimmten Wahrscheinlichkeit eine zufällige Störung angewendet, um weitere Edit Operations auf dem Komponentenmodell zu simulieren. Die Anwendung dieser Edit Operations bewirkt dadurch mehrere Modelltransformationen am Komponentenmodell, die zu unterschiedlichen Versionen des Komponentenmodells führen. Auf zwei aufeinanderfolgenden Versionen wird die Differenz gebildet, die nun als Datenbasis für das Subgraph Mining dient. Jedes der simulierten Model Repositories beinhaltet entweder 10 oder 20 dieser Differenzen, die zu einem Graphen zusammengebaut werden, auf den man anschließend Subgraph Mining Algorithmen anwenden kann, um interessante Edit Operations zu identifizieren.

1.4 Forschungsfragen

In einer ersten Pilotstudie von C. Tinnes in *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] wurde festgestellt, dass der SUBDUE Algorithmus im Vergleich zu anderen Subgraph Mining Algorithmen eine schwächere Leistung beim Erkennen von Edit Operations in Model Repositories erzielt. In dieser Arbeit wird gezeigt, dass der SUBDUE mit Anpassungen gezielt für Model Repositories optimiert werden kann. Hierfür wird eine neue Metrik zur Berechnung der Kompression vorgeschlagen, die zu einem besseren Mining Ergebnis führt. Zudem wird eine Umstrukturierung vorgenommen, mit der sich überflüssig Berechnungen einsparen lassen. Es wurden auch weitere Anpassungen ausprobiert, die jedoch noch nicht im Rahmen dieser Arbeit zu einer erfolgreichen Optimierung geführt haben. Durch die erfolgreich durchgeführten Veränderungen resultiert der vorgeschlagene THEOBALDSUBDUE. Dies führt zu folgenden Forschungsfragen:

RQ1 Wie funktioniert der SUBDUE Algorithmus und wie verhält er sich auf Model Repositories?

RQ2 Gibt es Unterschiede in den offizielle SUBDUE Implementierungen in Python und in C Implementierung und falls ja, welche Auswirkungen haben die Unterschiede auf die erzielten Ergebnisse?

RQ3 Kann der THEOBALDSUBDUE Algorithmus im Vergleich zum SUBDUE und zum GASTON Algorithmus bessere Ergebnisse auf Model Respositories erzielen? Hierfür wird überprüft, wie oft die Algorithmen das korrekte Pattern identifizieren konnten.

Um die Forschungsfrage **RQ1** und **RQ2** zu beantworten, wird eine erste Pilotstudie durchgeführt, in der die Performance der SUBDUE Implementierungen in den Programmiersprachen C und Python evaluiert wird. Hierfür wird die benötigte Laufzeit und der Speicherplatzverbrauch gemessen. Ebenso wird geprüft, ob beide Implementierungen die gleichen zu identifizierenden Edit Operations erkennen. Zudem wird für die Forschungsfrage **RQ1** ein Debugging Tool entwickelt, mit dem sich jeder

einzelne Schritt des SUBDUE Algorithmus nachvollziehen lässt. Zur Beantwortung der Forschungsfrage **RQ3**, werden die Ergebnisse des implementierten THEOBALDSUBDUE und des SUBDUE auf simulierte Model Repository Datensätzen gemessen. Auch hier wird die Laufzeit gemessen und ob eine korrekte Erkennung der zu identifizierenden Edit Operation erreicht wird.

1.5 Ergebnisse

Im Rahmen der vorliegenden Arbeit werden die notwendigen Grundlagen, sowie das Verfahren des SUBDUE Algorithmus auf einfache Art und Weise dargestellt. Dies dient zur Ergänzung der SUBDUE Literatur, da diese zu diesem Zeitpunkt noch als unzureichend für einen einfachen Verständnisaufbau eingestuft werden kann. Zudem wird ein Debugging Tool für die Beam Search des SUBDUE implementiert, wodurch sich der Algorithmus Schritt für Schritt nachvollziehen lässt. Mit diesem Gesamtpaket wird die Forschungsfrage **RQ1** in dieser Arbeit beantwortet.

Für die Beantwortung von **RQ2** werden die SUBDUE Implementierungen in C und Python in einem Experiment untersucht. Hierfür werden die Mining Ergebnisse gemessen und miteinander verglichen. Aus den Messungen kann entnommen werden, dass es zu signifikanten Unterschieden in den Ergebnissen kommt, auch bei gleichen Parametereinstellungen. Im Rahmen dieser Arbeit konnte noch nicht vollständig geklärt werden, aus welchen Gründen diese Unterschiede auftreten. Darauf wird entsprechend im Ausblick in Kapitel 8 eingegangen. Die Auswirkungen der Unterschiede kann jedoch aus dem ersten Experiment entnommen werden.

Die Messungen aus dem dritten Experiment werden in der Tabelle 1.1 an dieser Stelle vorab dargestellt, um die Forschungsfrage **RQ3** zu beantworten. Aus der Tabelle ist zu entnehmen, dass der vorgeschlagene THEOBALDSUBDUE auf dem angewandten Datensatz zu besseren Ergebnissen gelangt, als der SUBDUE. Somit konnte der SUBDUE für das Subgraph Mining von Edit Operations in Model Repositories erfolgreich in dieser Arbeit verbessert werden. Nichtsdestotrotz erzielt der GASTON Algorithmus weiterhin bessere Ergebnisse. Im Kapitel 8 wird ein Ausblick auf weitere Optimierungsansätze des THEOBALDSUBDUE geworfen, mit deren Umsetzung sich die Ergebnisse weiter verbessern könnten.

Algorithm	**Runtime** (s)	**Match** (%)
TheobaldSubdue	3,5333	91,19
Subdue Python (Heuristic)	2,9656	89,79
Subdue Python (Size)	3,1184	89,99
Subdue C (MDL)	0,4370	89,89
Subdue C (Size)	0,4233	90,19
Gaston	1,1924	94,79

Tabelle 1.1: TheobaldSubdue im Vergleich.

1.6 Herausforderungen

Es haben sich einige Herausforderungen ergeben, die hier Erwähnung finden. Dieser Abschnitt hält zudem die wichtigsten Erkenntnisse fest, die neben der Beantwortung der Forschungsfragen festgestellt wurden.

Unterschiedliche Formate für Graphen

Eine zeitintensive Angelegenheit ergibt sich aus der Vielzahl der verschiedenen Formate für Graphen. Es gibt unzählige Formate und es hat sich bisher kein einheitlicher Standard durchgesetzt. Dadurch tritt das Problem auf, dass oftmals die Algorithmen unterschiedliche Formate für ihre Graphenrepräsentation auswählen. Darüber hinaus fügen einige Algorithmen zusätzliche Informationen in ihre Darstellung ein, wodurch noch weitere Formate außerhalb einiger gängigen Formate entstehen. Werden nun mehrere Algorithmen miteinander verglichen oder kombiniert, führt dies unweigerlich zu dem Problem, dass die Formate nicht kompatibel miteinander sind. Dies führt dazu, dass viele Konverter notwendig sind, die von einem Format zum anderen konvertieren. Dabei muss das entsprechende Mapping der Formate mühsam in Handarbeit geschrieben werden. Vermutlich werden viele andere im Bereich des Graph Mining auf ein ähnliches Problem stoßen, sodass es hilfreich wäre, ein geeignetes Tool zu haben, das alle möglichen Formate konvertieren kann.

Verständnis heuristischer Algorithmen

Das Nachvollziehen des Subdue Algorithmus hat sich als viel komplizierter herausgestellt als erwartet. Zum einen gibt es nicht genügend Literatur darüber, wie der Subdue Algorithmus im Detail funktioniert und zum anderen ist es oft schwierig die Gründe für Designentscheidungen bei der Implementierung des Algorithmus zu verstehen. Der Subdue Algorithmus verwendet viele Heuristiken, die in den Veröffentlichungen nicht immer detailliert ausformuliert sind. Das ist auch einer der Gründe, weshalb im Rahmen der vorliegenden Arbeit ein Debugging Tool entwickelt wurde, um den Subdue Schritt für Schritt nachvollziehen zu können.

Fehlerhafte Ergebnisse

Fehlerhafte Ergebnisse waren nicht immer deutlich und sofort zu erkennen. Fehler im Code führten zwar oftmals zu schlechteren Ergebnissen, jedoch war nicht immer einfach ersichtlich, dass die Ergebnisse auch fehlerhaft waren. Es konnte nicht immer von vornherein ausgeschlossen werden, dass das Subdue aus irgendeinem

Grund keine schlechten Ergebnisse liefert und somit die Ergebnisse doch korrekt sind. Das Debuggen von Graph Mining Algorithmen, insbesondere dem SUBDUE Algorithmus hat sich als aufwändiger erwiesen, als zuvor angenommen, was die Interpretation der Ergebnisse erschwerte. Hier ist es wichtig, einen geplanten Entwicklungszyklus mit ausreichendem Testen einzuhalten, um Fehler rechtzeitig zu erkennen. Eine erfolgreiche Methode war unter anderem, die Ergebnisse des Python SUBDUE und des THEOBALDSUBDUE mit einer weiteren C Implementierung des SUBDUE zu überprüfen. Dabei haben Ausreißer in den Ergebnissen oftmals auf Fehler im eigenen Code hingewiesen.

Kapitel 2

Grundlagen

Im folgenden Kapitel der Grundlagen werden zunächst die maßgeblichen Grundbausteine besprochen, um das vorausgesetzte Wissen zusammenzufassen, sodass das notwendige Verständnis zur Herleitung der in der Einleitung genannten Motivation geschaffen wird. Um die Herausforderung in Gänze zu verstehen, wird im weiteren Verlauf der Ursprung des wesentlichen Konzepts von Software Reuse und des Model Driven Engineerings erläutert. Dabei wird verständlich gemacht, dass die Beherrschung von großen Softwaresystemen tatsächlich noch immer nicht gelöst ist. Auch 50 Jahre nach dem Aufkommen des Software Engineerings ist das Ziel, den Aufwand bei der Entwicklung und Wartung von Softwaresystemen beherrschbar zu machen, noch nicht erreicht. Durch welche Konzepte, zu welchem Zeitpunkt und ob dieser Anspruch überhaupt erreicht werden kann, wird selbst in Fachkreisen noch diskutiert. Es ist jedoch klar, dass noch viel Aufwand in das Forschungsthema Software Engineering investiert werden muss, um die Komplexität von Softwaresystemen deutlich zu reduzieren.

Daran anknüpfend werden in den weiteren Kapiteln des Grundlagenteils die theoretischen und technischen Aspekte des Graph Mining behandelt. Algorithmische Probleme auf Graphen sind nicht immer trivial und können eine sehr komplexe Angelegenheit darstellen. Diese Komplexität für das Subgraph Mining spielt in der Umsetzung der Erkennung interessanter Edit Operations eine wesentliche Rolle und wird im Weiteren näher erläutert. Die vorhandene Komplexität muss von allen Subgraph Mining Algorithmen bewältigt werden, unabhängig davon, welchen Prinzipien sie folgen. Um den TheobaldSubdue von anderen Subgraph Mining Algorithmen abzugrenzen, werden die beiden wichtigsten Ansätze im Subgraph Mining vorgestellt. Hierfür wird das Prinzip des Frequent Subgraph Minings und der Minimum Description Length betrachtet. Der TheobaldSubdue nutzt den kompressionsbasierten Ansatz der Minimum Description Length für das Erkennen von Subgraphen, während andere Subgraphen Mining Algorithmen wie der Gaston das Prinzip des Frequent Subgraphen Minings verwenden. Diese beiden Ansätze gilt es gegenüberzustellen, um die Unterschiede erkenntlich zu machen.

2.1 Software Reuse

Aufgrund der sich abzeichnenden und unkontrollierbaren Komplexität von Softwaresystemen rief das NATO Science Committee im Jahr 1968 zur ersten Softwa-

re Engineering Conference auf, die zum Ziel hatte, die Gründe der Komplexität von Softwaresystemen zu verstehen und im Weiteren Lösungsansätze zur Komplexitätsreduktion zu erarbeiten. Auch wenn die Anforderungen im Vergleich zu heute andere waren, erkannte man schon zur damaligen Zeit, dass mit wachsender Größe eines Softwaresystems die Komplexität rasant zunimmt. Diese Beobachtung führte zu einem Aufruhr in den Fachkreisen der Informatik, weshalb diese Zeit als Softwarekrise in die Geschichte einging. Daraus resultierte auch die Feststellung, dass die Entwicklung von Software einer eigenen Ingenieurwissenschaft zuzuordnen ist, dem Software Engineering. Die Erkenntnisse aus der NATO Software Engineering Conference wurden von P. Naur und B. Randell in dem Bericht *Software Engineering* [6] festgehalten. Obwohl mit zunehmender Zeit immer größere und noch komplexere Systeme entwickelt wurden, hat sich an der Substanz der Problematik im Software Engineering nicht sonderlich viel geändert. Trotz immer weiter wachsenden Verständnis, verbesserter Methoden und modernen Werkzeugen ist es weiterhin nicht gänzlich gelungen, diese Komplexität maßgeblich zu reduzieren, sodass sich ein Softwaresystem einfach und effizient kontrollieren lässt. Aus diesem Grund gilt die Softwarekrise in den Fachkreisen weiterhin als ungelöstes Problem in der Informatik. Zu diesem Schluss kommt auch B. Randell 50 Jahre nach seiner Teilnahme an der NATO Software Engineering Conference.

> But some large software projects in the latter bespoke category still suffer from problems that are all too reminiscent of those that, in 1968, gave rise to discussion of a “software crisis”.
>
> (B. Randell)

Infolgedessen ist es unabdingbar, weitere Bestrebungen in die Forschung zur Komplexitätsreduktion von Softwaresystemen zu investieren. Da die Komplexität von Softwaresystemen noch immer nicht zureichend beherrscht werden kann, stellt sich also weiterhin die Frage, wie sich diese Komplexität noch weiter reduzieren lässt. Der naive Gedanke, dass die Komplexität nach der initialen Entwicklung des Softwaresystems abnimmt, trifft ebenso wenig zu, wie die Beherrschbarkeit der Komplexität in der Planungs- und Entwicklungsphase. Vielmehr ist genau das Gegenteil der Fall, denn oftmals steigt die Komplexität im weiteren Verlauf des Betriebs und der Wartung weiter an. Dies beobachtet auch M. Lehman, der mit seiner publizierten Arbeit *Programs, Life Cycles, and Laws of Software Evolution* [12] die Gesetze der Software Evolution aufstellt, anhand dessen sich einige Gründe und Auslöser der Komplexität von Softwaresystemen entnehmen lassen. Die zwei bedeutendsten Gesetze Continuing Change und Increasing Complexity besagen, dass ein Softwaresystem einem kontinuierlichen Wandel unterliegt, was unweigerlich aufgrund der wachsenden Entropie in einem Softwaresystem zu einem Anstieg der Komplexität führt. Diese Evolution ist auch in dem Buch *Software Engineering at Google* [13] beschrieben. Dort wird auch das Problem erwähnt, dass durch die Schnittstellen offener Systeme oftmals deren Komplexität weitergegeben wird, was zu einem fortlaufendem Übermaß an Komplexität führt. Dieses Phänomen ist unter dem Begriff Hyrum’s Law bekannt. Die Auswirkungen dieser Gesetzmäßigkeiten tragen ihren Teil dazu bei, dass durch die stetig wachsende Anzahl an Entitäten innerhalb eines Softwaresystems die Anzahl an Wechselwirkungen exponentiell zunimmt, was wiederum den Kontrollverlust über das gesamte Softwaresystem über die Zeit immer

Jahr	**Succeeded** (%)	**Challenged** (%)	**Failed** (%)
1994	16	53	31
1996	27	33	40
1998	26	46	28
2000	28	49	23
2002	34	51	15
2004	29	53	18
2006	35	46	19
2009	32	44	24
2010	31	47	22
2011	29	49	22
2012	27	56	17
2013	31	50	19
2014	28	55	17
2015	29	52	19

Tabelle 2.1: Erfolg und Misserfolg von IT-Projekten.

wahrscheinlicher macht.

Die Systemkomplexität ist damals wie heute einer der Gründe, warum die Quote der unzureichenden und fehlgeschlagenen Softwaresysteme immer noch so hoch ist. Einen guten Überblick über das Thema Erfolg und Misserfolg von Softwaresystemen bietet das IT-Forschungsberatungsunternehmen Standish Group. Die Standish Group ist bekannt für ihre Chaos-Studie, die sich schon seit 1994 mit den Erfolgs- und Misserfolgsfaktoren in IT-Projekten beschäftigt. Somit gilt die Chaos-Studie mit über 40.000 Daten aus verschiedenen IT-Projekten als eine der wichtigsten Langzeitstudien, die den Erfolg und Misserfolg von Softwaresystemen empirisch untersucht. In der Tabelle 2.1 werden die Erfolgs- und Misserfolgsquoten über zwei Jahrzehnte dargestellt. Ausgehend vom Jahr 1994 scheint ein leichter Trend erkennbar zu sein, der die Erfolgsrate zu Beginn etwas ansteigen, sowie die Misserfolgsrate etwas sinken lässt. Es ist jedoch klar zu erkennen, dass in den letzten zwei Jahrzehnten keine signifikanten Fortschritte der Komplexitätsreduktion in der Entwicklung von Softwaresystemen zu verzeichnen waren.

In der Studie *Critical success factors for software projects: A comparative study* [14] werden zusätzlich insgesamt 43 Veröffentlichungen aus den Jahren 1990 bis 2010 analysiert, in denen sich signifikante Beiträge finden lassen, die einen Hinweis darauf geben, was die kritischen Faktoren für den Erfolg von Softwareprojekten sind. Auch wenn die Studie zeigt, dass ein Großteil der kritischen Faktoren von nicht-technischer Natur sind, sollte beachtet werden, dass viele der nicht-technischen Faktoren durch die Systemkomplexität verursacht werden. Aus der Studie geht hervor, dass bspw. ein unrealistischer Zeitplan einer der treibenden Faktoren für das Scheitern von Softwaresystemen ist, der mit einer Häufigkeit von 53,5% auftritt. Aber auch andere Faktoren wie ein unrealistischer Kostenrahmen (44,2%), Unvertrautheit mit Technologie (34,9%), sowie Komplexität und Projektgröße (23,3%) resultieren maßgeblich aus der Komplexität von Softwaresystemen.

Die wohl leistungsstärkste Methode um gegen Systemkomplexität vorzugehen ist ein Software Reuse Ansatz. Es ist eines der wichtigsten Fundamente, um die Komplexität und Aufwände zu reduzieren. Es bildet somit ein unerlässliches Werkzeug im Software Engineering und ist Grundlage dafür, dass ein Softwaresystem nicht mehr von Grund auf neu entwickelt werden muss, sondern aus schon vorhandenen Software Artefakten zusammengesetzt werden kann. Es beschreibt also die Methodik der Wiederverwendung von schon existierenden Komponenten in einer neuen Einsatzumgebung. Dabei ist es in der Regel nicht wichtig, in welchem Kontext die Komponente zuvor eingesetzt wurde. Wenn die Komponente den Software Reuse Anforderungen entspricht, dann kann durch die Wiederverwendung die Produktivität des zu entwickelnden Softwaresystems erhöht werden. Der Einsatz von Wiederverwendung unter Voraussetzung von standardisierten Komponenten führt in den meisten Fällen zu einer Kosteneinsparung in der Entwicklung, da nicht mehr alle Komponenten von Grund auf neu entwickelt werden müssen. Eine erfolgreiche Integration kann allerdings nur dann gelingen, wenn die Komponente zuvor für eine Wiederverwendung ausgelegt ist.

Das in diesem Zusammenhang wahrscheinlich wichtigste Werk trägt den Titel *Software Reuse* [7] und wurde von C. Krüger publiziert. Das wichtigste Merkmal von Software Reuse ist Abstraktion, welche eine zentrale Rolle zur Komplexitätsreduktion einnimmt. Aus diesem Grund wird im Weiteren das Konzept der Abstraktion näher erläutert. Denn Abstraktion bietet nicht nur die Möglichkeit die Komplexität während der initialen Entwicklung von neuen Softwaresystemen zu minimieren, sondern auch die von M. Lehman erkannten Probleme resultierend aus der Software Evolution vorzubeugen. Die Schaffung eines höheren Abstraktionsgrades ist die fundamentale Zielsetzung hinter dem Subgraph Mining von Edit Operations in Model Resposito-ries, weshalb ein detaillierter Blick auf die Abstraktion notwendig ist.

2.1.1 Abstraktion

H. Vliet beschreibt die Eigenschaft der Abstraktion in seiner Publikation *Software engineering - principles and practice* [15] wie folgt:

> Abstraction means that we concentrate on the essential features and ignore, abstract from, details that are not relevant at the level we are currently working.
>
> (H. Vliet)

Abstraktion bedeutet also, dass kein Wissen über die tatsächliche Funktionalität eines Prozesses vorhanden sein muss. Es reicht lediglich zu wissen, dass das Verfahren auch das gewünschte Ergebnis liefert. Ein einfaches Beispiel aus der realen Welt zeigt, wie mächtig das Werkzeug der Abstraktion ist. In fast jedem Haushalt befindet sich eine Steckdose, die elektrische Geräte mit Strom versorgen kann. Um das elektrische Gerät zu nutzen, muss der Nutzer nicht wissen, wie genau der Strom erzeugt und zu seinem Haushalt transportiert wird. Es genügt lediglich zu wissen, dass Strom verfügbar ist, wenn ein elektrisches Gerät angeschlossen wird. Der Nutzer muss sich nicht um das Detailwissen bemühen, sondern kann einfach die

Funktionalität der Steckdose und die des Stromes nutzen, ohne jegliche Kenntnisse über die dahinter stehende Physik zu haben. Ebenso verhält sich die Abstraktion im Software Engineering. Durch Abstraktion können komplexe Vorgänge hinter Schnittstellen versteckt werden, sodass die Komplexität vor dem Benutzer verborgen werden kann. Hierfür werden oftmals Probleme in kleinere Teilprobleme zerlegt, die dann wiederum abstrahiert werden. Dadurch ergeben sich viele kleinere Module, deren Funktionalität zwar jeweils selbst komplex sein kann, die jedoch nach außen hin einfache Schnittstellen anbieten, sodass das Wissen über das Detail nicht notwendig ist. Durch immer weitere Abstraktionsgrade ist es somit möglich, komplexe Softwaresysteme zu entwickeln, da stets auf vorhandene Module aufgebaut werden kann. Mit jedem weiteren Layer an Abstraktion wird das benötigte Detailwissen über das darunterliegende Layer irrelevant. Es ist lediglich notwendig zu verstehen, wie man das darunterliegende Layer ansprechen muss, um eine gewünschte Funktionalität hervorzurufen. Ein weiteres Beispiel, bei dem sich das Prinzip der Abstraktion als sehr erfolgreich erwiesen hat, ist die Funktionalität heutiger Computersysteme. Die Abbildung in 2.1 zeigt einen Ausschnitt mehrerer Abstraktionslayer eines Computersystems. Sie zeigt auch, dass ein Entwickler einer einfachen Anwendung kein Wissen mehr darüber haben braucht, wie die darunterliegende Hardware funktioniert. Dieses Maß an Komplexitätsreduktion ist ebenfalls notwendig, um Softwaresysteme beherrschbar zu machen. Weitere Untersuchungen in diesem Bereich sind daher von dringender Notwendigkeit, sodass komplexe Softwaresysteme einfacher zu entwickeln und effizienter zu warten sind.

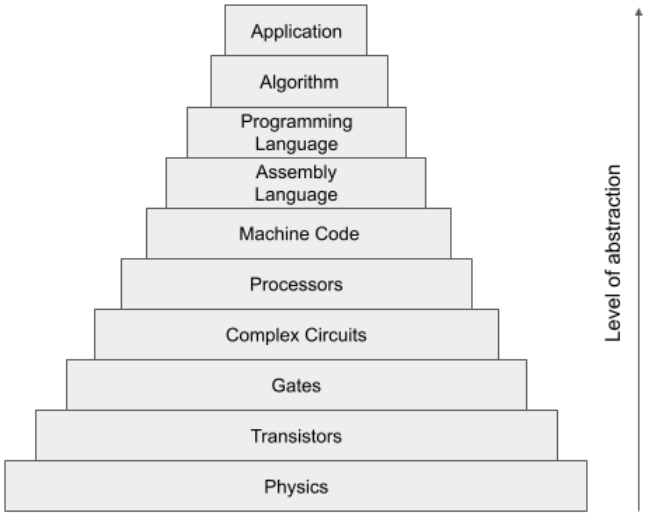

Abbildung 2.1: Ausschnitt einiger Abstraktionslayer eines Computersystems.

2.2 Model Driven Engineering

Das Model Driven Engineering ist ein Paradigma innerhalb der Softwareentwicklung. Mit diesem Paradigma soll unter anderem die Komplexität von Softwaresystemen minimiert und die daraus resultierenden Probleme reduziert werden. Das wichtigste Werkzeug im Model Driven Engineering sind Modelle, mithilfe derer die Entwicklung eines Softwaresystems auf ein neues Abstraktionsniveau gehoben werden soll. Durch das Mittel der Abstraktion soll die Systemkomplexität für Softwareentwickler beherrschbarer werden, indem es den Fokus der Softwaresystementwicklung von der reinen Entwicklung von Quellcode auf den Modellentwurf verlagert. Im Gegensatz zu Quellcode von Programmiersprachen, die oftmals sehr allgemein formuliert sind, können Modelle speziell für die Domäne der Anwendungen entworfen werden. Damit rückt das für die Entwicklung von Softwaresystemen erforderliche technische Detailwissen weiter in den Hintergrund. In anderen Worten: Ein Modell wird dazu verwendet, um das System in einer domänenspezifischen Sprache zu beschreiben, sodass keine technischen Kenntnisse erforderlich sind. Modelle können also zur Abstraktion von Softwaresystemen verwendet werden, um das gesamte System aus einer vereinfachten Sicht darzustellen. In der Veröffentlichung *Model-driven engineering: A survey supported by the unifiedconceptual model* [16] beschreibt das A. Rodrigues da Silva wie folgt: Mithilfe von Modellen könne eine gemeinsame Vision und gemeinsames Wissen zwischen technischen und nicht-technischen Beteiligten geteilt werden, um somit die Kommunikation zu erleichtern.

> Models allow sharing a common vision and knowledge among technical and non-technical stakeholders, facilitating andpromoting the communication among them.
>
> (A. Rodrigues da Silva)

Ein Teilbereich des Model Driven Engineering beinhaltet, dass aus den domänenspezifischen Modellen nachträglich der Quellcode generiert wird. Mithilfe dieser Modelle können auch tendenziell nicht-technische Anwender an dem Softwaresystem arbeiten, da nur das Domänenwissen vorhanden sein muss und der Quellcode automatisch auf Basis der Modelle erstellt wird. Modelle werden also nicht mehr nur als Artefakte zur Dokumentation verwendet, sondern werden zunehmend zu einem zentralen Bestandteil von Softwaresystemen.

Aus der Veröffentlichung *The State of Practice in Model-Driven Engineering* [1] lässt sich entnehmen, dass Model Driven Engineering zwar grundsätzlich ein bewährter Ansatz für die Entwicklung einzelner komplexer Teilbereiche ist, jedoch in der Praxis nur selten zur Erstellung ganzer Softwaresysteme zum Einsatz kommt. Model Driven Engineering deckt daher oft nur einen kleinen Anwendungsbereich im Entwicklungszyklus eines Softwaresystems ab und spielt daher oft nur eine untergeordnete Rolle. Die Veröffentlichung zeigt auch interessante Ergebnisse der dort durchgeführten Studie. Diese besagen, dass durch die Einführung von Model Driven Engineering die meisten befragten Unternehmen eine Produktivitätssteigerung zwischen 20% und 30% erfahren haben. Die Autoren J. Whittle, J. Hutchinson und M. Rouncefield geben an, dass diese Steigerung nicht signifikant genug sei, um Model Driven Engineering mit aller Gewalt in einem Unternehmen einzuführen. Zumal

nach Angaben der Studie die wesentlichen Vorteile bisher durch die Dokumentation anhand von Modellartefakten und nicht durch die Codegenerierung getrieben wird. Im Gegensatz zu den Autoren J. Whittle, J. Hutchinson und M. Rouncefield der zuvor genannten Veröffentlichung beschreibt B. Selic in seiner Arbeit *The Pragmatics of Model-Driven Development* [17], dass Modelle, die am Ende nur zur Dokumentation dienen, nur von begrenztem Wert seien. Dokumentationen weichen oft von der Realität der Softwaresysteme im Einsatz ab und müssen aufwendig gewartet und gepflegt werden. B. Selic ist der Meinung, dass die automatisierte Generierung von Quellcode aus ihren Modellen das grundlegende Prinzip von dem Model Driven Engineering Ansatz darstellt. Auch wenn die Generierung von Quellcode in den frühen Anfängen nur mit begrenztem Erfolg umgesetzt werden konnte, sieht B. Selic gute Chancen die Produktivität in Zukunft signifikant steigern zu können. Grund zu dieser Annahme sind die immer ausgereifteren Automatisierungstechnologien, mit deren Hilfe sich auch vollständige Softwaresysteme aus Modellen generieren lassen.

Im Bereich Software Engineering, insbesondere im Model Driven Engineering scheint die Weiterentwicklung in Bezug auf Produktivität und Zuverlässigkeit deutlich langsamer voranzuschreiten als erwünscht. Dies geht auch aus der Publikation *Fifty Years of Software Engineering - or - The View from Garmisch* [18] hervor. So hat sich seit der Einführung der Programmiersprachen der dritten Generation im Jahr 1950 die auf strukturierten und prozeduralen Konzepten basieren, nicht viel geändert. Das Abstraktionsniveau unserer heutigen Programmiersprachen haben sich dahingehend kaum weiterentwickelt und basieren noch immer auf den historisch gewachsenen Ansätzen von damals.

Ein idealer Umbruch, der eine neue Abstraktionsstufe bringen würde, ist wie folgt zu verstehen: Analog zur Verdrängung der Assemblersprachen durch Programmiersprachen der dritten Generation wie Java oder C++ könnten auch diese Programmiersprachen durch Programmiertechnologien aus dem Model Driven Engineering ersetzt werden. Auch wenn diese Entwicklung noch in weiter Ferne zu liegen scheint, ist sie unerlässlich, um die Produktivität zu steigern und die Komplexität beherrschbar zu machen. Diese Thesis soll einen Beitrag zur Weiterentwicklung in diesem Bereich leisten, sodass neue Programmiertechnologien mit höheren Abstraktionsebenen entwickelt werden können, um die Effizienz zukünftig signifikant steigern zu können. Aus diesem Grund knüpft diese Arbeit an den Konzepten aus dem Model Driven Engineering an mit dem Ziel, die Generierung von Quellcode anhand von Modellen besser zu verstehen. Mit dem Vorschlag des TheobaldSubdue soll zudem die Identifikation von interessanten Mustern in den Modellen verbessert werden, sodass eine automatisierte Generierung auf Basis der gefundenen Muster durchgeführt werden kann.

2.2.1 Model Repository

Ein Model Repository enthält alle möglichen Artefakte, die mit Werkzeugen des Model Driven Engineering erstellt und bearbeitet werden können. Die Untersuchungen in dieser Arbeit werden nicht direkt an einem Model Repository durchgeführt, sondern es werden dafür simulierte Model Repositories mit einfacher Struktur verwendet, damit die geplanten Experimente kontrolliert durchgeführt werden können.

Um mehr über relevante Model Repository Technologien und Werkzeuge zu erfahren, findet sich hierzu eine Kurzübersicht in der Tabelle 2.2, die aus der Publikation *Collaborative Repositories in Model-Driven Engineering* [19] entnommen ist. Innerhalb eines Model Repository werden die Artefakte systematisch strukturiert und verknüpft. Dadurch wird das Model Repository mit den darin enthaltenen Modellen gekoppelt, sodass ein Prozess für die Systementwicklung implementiert werden kann. Mit den entsprechenden Entwicklungswerkzeugen für Model Repositories lässt sich somit eine Standardisierung innerhalb des Model Repositories schaffen. Dadurch kann eine kollaborative Zusammenarbeit zwischen den beiden Hauptanwendern eines Model Repositories gewährleistet werden. Die beiden Anwendergruppen werden in der Veröffentlichung *A Model-Driven Approach for Developing a Model Repository: Methodology and Tool Support* [20] definiert. Der Reuse Producer entwickelt neue Artefakte für das Model Repository und speichert diese dort ab. Der Reuse Consumer wiederum benutzt die schon vorhandenen Artefakte zur Wiederverwendung. Diese Zusammenarbeit kann mithilfe eines Model Repositories und einem entsprechenden Entwicklungswerkzeug verbessert werden.

Trotzdem stecken die Entwicklungswerkzeuge noch in den Kinderschuhen und es gibt noch einige Herausforderungen zu überwinden, um die Produktivität bei Model Repositories zu erhöhen. Diese Herausforderungen sind ebenfalls der Arbeit *Collaborative Repositories in Model-Driven Engineering* [19] zu finden. Während das Tooling rund um die eigentlichen Modelle bereits recht gut ausgestattet ist, fehlt es oft noch an optimaler Unterstützung für andere Modellierungsartefakte, Modellierungseditoren, Modelltransformationen und Codegeneratoren. Zudem fehlt es noch an effizienteren Abfragemechanismen. Zu Beginn der Thesis wurde bereits erwähnt, dass Model Repositories zu extrem großen Datenspeicher heranwachsen können. Gesamtgrößen von mehreren 100 GB mit Millionen von Einzelelementen sind keine Seltenheit, daher ist das effiziente Abrufen von Artefakten ein wesentlicher Bestandteil, um die Produktivität auf Model Repositories zu steigern. Um diese Herausforderungen zu meistern, bedarf es weiterer Forschungstätigkeiten. Insbesondere mit dem Ansatz des Subgraph Minings von Edit Operations könnten effizientere Abfragemechanismen entwickelt werden, dass nicht mehr die vollständigen Modelle abgefragt werden müssten. Wenn wichtige Konzepte auf den Modellen identifiziert werden können, könnte das Erlernen dieser Konzepte es ermöglichen, die Modelle zu komprimieren, um bspw. die Effizienz von Abfragemechanismen zu optimieren.

2.2.2 Edit Operation

Eine zentrale Rolle im Model Driven Engineering sind Model Transformationen, mithilfe derer sich die Evolution von Modellen durchführen, sowie abbilden lässt. Modelle können somit erstellt, gelöscht, bearbeitet oder zusammengeführt werden und bilden somit eine Klasse an Modellierungsoperationen. Die Klasse der Edit Operations ist eine Unterklasse der Model Transformationen. Mithilfe einer Edit Operation kann bspw. ein Entwickler eine Änderung an einer Modellinstanz vornehmen. Somit kann eine Modelländerung innerhalb eines Model Repositories durchgeführt werden, was zu einer neuen Version des Model Repositories führt. Auf diese Weise können Modellunterschiede (Diffs) zwischen zwei Versionen durch Edit Operations beschrieben werden.

Werkzeug	**Artefakte**	**Kollaborativ**	**Schnittstellen**
Rational Rhapsody Designer Manager	SysML, UML Modelle	Ja	XMI
Enterprise Architect	SysML, UML, Modelle	Ja	XMI
Visual Paradigm	UML Modelle	Ja	XMI
MagicDraw	UML Modelle	Ja	XMI
Modellio	Modelle	Ja	XMI
GenMyModel	UML Modelle	Ja	XMI
CDO	Modelle, Metamodelle	Ja	EMF
EMFStore	Modelle, Metamodelle	Ja	EMF
MORSE	Modelle	Ja	Service
ModelBus	Modelle, Metamodelle, Transformationen	Ja	Adapter
AMOR	Modelle, Metamodelle	Ja	XMI
ReMoDD	Modelle, Metamodelle, Transformationen	Nein	-
GME	Modelle, Metamodelle	Nein	COM
MDEForge	Modelle, Metamodelle, Transformationen, Editoren	Nein	REST-API

Tabelle 2.2: Bekannte Entwicklungswerkzeuge für Model Repositories.

Mit Edit Operations können viele Operationen auf Modelle automatisiert werden. In der Publikation *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] von C. Tinnes sind die Operationen auf Modellen genannt, die sich mithilfe von Edit Operations abbilden lassen. Diese Auflistung zeigt deutlich, dass Edit Operations ein mächtiges Entwicklungswerkzeug darstellen. Das Subgraph Mining und dem daraus resultierenden Lernen von interessanten Edit Operations könnte Optimierungen für diese Anwendungsfälle hervorbringen. Somit verspricht dieser Ansatz, ein breites Spektrum an Einsatzmöglichkeiten zu finden:

- Repair, quick-fix generation, auto completion
- Slicing
- Model editors
- Operation-based merging

- Model refactoring
- Model optimization
- Model Co-Evolution and meta-model evolution
- Artifact co-evolution in general
- Semantic lifting of model differences
- Mutation-based fuzzing
- Delta-oriented development in model-based product line engineering
- Model generation

2.2.3 Modellierungssprache Henshin

Henshin ist eine Modellierungssprache für das Eclipse Modeling Framework, mit der sich Modelltransformationen beschreiben lassen. Somit können auch Edit Operations mithilfe von Henshin ausgedrückt werden. Henshin selbst basiert ebenfalls auf einem Meta Model, welches in Abbildung 2.2 dargestellt ist. Diese Abbildung ist aus der Veröffentlichung *Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations* [21] entnommen. Dieses Paper geht detaillierter auf das Konzept von Henshin ein. Im Folgenden sollen nur die für diese Thesis wichtigsten Aspekte berücksichtigt werden. Eine Modelltransformation ist nichts anderes als eine Regel, die die Transformation von einem Eingangsgraphen in einen Ausgangsgraphen beschreibt. Die Modelltransformation einer Modellinstanz beschränkt sich hierbei auf die Beziehungen, die auf dem Meta Model der Modellinstanz zur Verfügung stehen. Das bedeutet, dass eine Modelltransformation auf der Grundlage eines Meta Models definiert wird. Eine Modellinstanz kann sich also nur auf der Basis ihres Meta Models ändern.

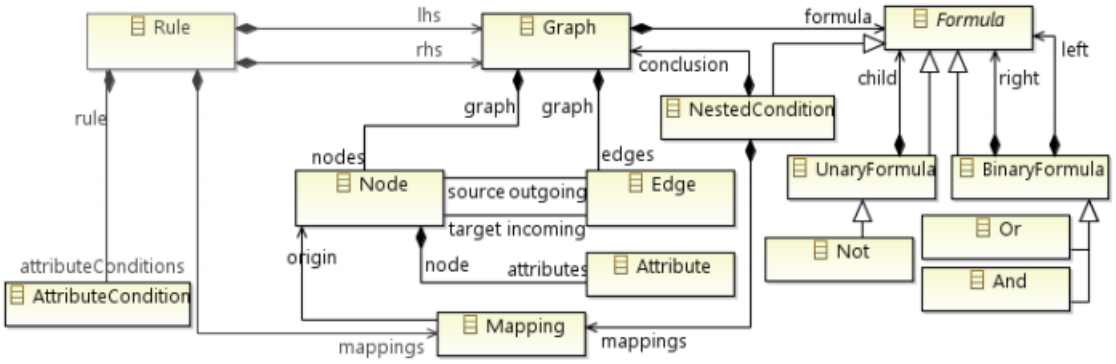

Abbildung 2.2: Meta Model der Modellierungssprache Henshin.

Um eine Transformationsregel in Henshin zu erstellen, können Parameter, Knoten, Kanten, Attribute und Aktionen für die Modelltransformation definiert werden. Für

den Zweck dieser Thesis sind insbesondere die Knoten, Kanten und Aktionen wichtig. Weitere Elemente der Henshin Modellierungssprache werden in dieser Arbeit nicht berücksichtigt.

Aktionen

Die Knoten und Kanten einer Edit Operation sind je nach dem Typ der Änderung mit einem Stereotyp beschriftet. Eine Edit Operation kennt die drei unten stehenden Aktionen, die zu einer Modelländerung auf einer Modellinstanz führen.

1. Die *Create* Aktion annotiert einen Knoten oder eine Kante mit dem Label $<< create >>$, wenn diese durch die Modelltransformation der Modellinstanz neu hinzugefügt werden soll und somit in der neuen Version der Modellinstanz verfügbar ist.

2. Die *Delete* Aktion annotiert einen Knoten oder eine Kante mit dem Label $<< delete >>$. Mit dieser Aktion wird das versehene Objekt durch die Ausführung der Modelltransformation entfernt und ist somit nicht mehr in der neuen Version der Modellinstanz verfügbar.

3. Knoten oder Kanten, die mit dem Label $<< preserve >>$ versehen sind, werden nicht direkt einer Aktion unterzogen. Objekte mit diesem Label werden nicht durch die Model Transformation geändert, sie werden ohne Veränderung in die neue Version der Modellinstanz mit übergehen.

Beispiel einer Henshin Modelltransformation

Das Beispiel in Abbildung 2.3 zeigt eine Modelltransformation auf einer Modellinstanz auf Basis eines Meta Models. Die Modelltransformation ist mithilfe von Henshin abgebildet und kann als Edit Operation interpretiert werden. Der Einfachheit halber wird hier eine genauere Beschreibung des Hinzufügens weggelassen, die über Parameter hätten beschrieben werden können, um bspw. dem Studenten einen Namen und eine Matrikelnummer zuzuweisen. Die Edit Operation bildet also nur die durchgeführte Operation ab, ohne die konkreten Details der Operation zu beschreiben. Sie beschreibt lediglich das Hinzufügen eines neuen Studenten zur Modellinstanz. Hierfür wird ein neues *Student* Objekt mithilfe von $<< create >>$ beschrieben, zudem werden auch die benötigten Beziehungen zur *University* und zur *Faculty* dargestellt, die für das Hinzufügen des Studenten notwendig sind. Die mit $<< preserve >>$ gekennzeichneten Objekte werden dabei unverändert mit in die nächste Version der Modellinstanz übernommen, sind also nicht von Änderungen der Modelltransformation betroffen.

Diese vereinfachte Beschreibung einer Edit Operation in Henshin wird in der vorliegenden Arbeit verwendet, um die Modelltransformationen auf den Datensätzen bestehend aus Model Repositories zu beschreiben. In den durchgeführten Experimenten dieser Arbeit werden die ausgeführten Edit Operations, die sich in den Model Repositories manifestiert haben, als Graphen repräsentiert. Diese Edit Operations können mithilfe von Subgraph Mining Algorithmen identifiziert und gelernt werden. Somit kann auf einem Model Repository, auf dem viele unterschiedliche

Edit Operations angewandt wurden, die interessantesten Edit Operations erkannt werden.

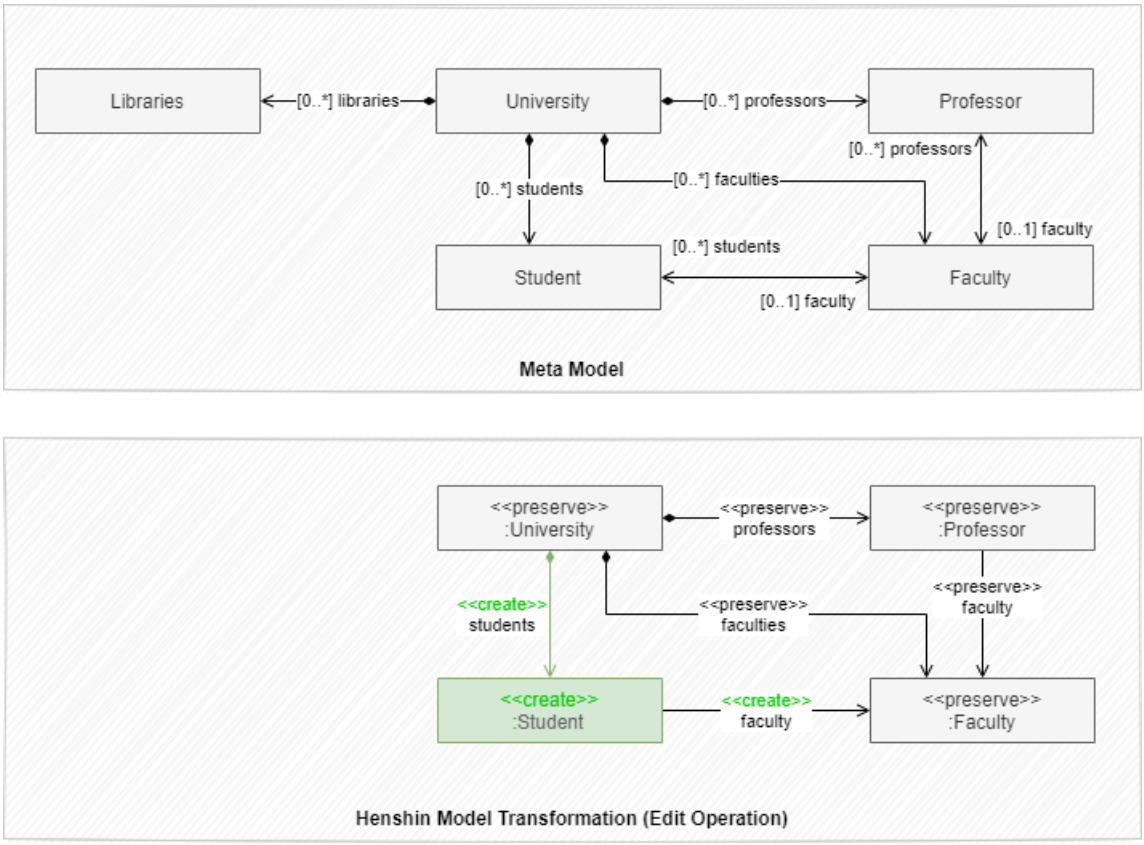

Abbildung 2.3: Beispiel einer Henshin Modelltransformation auf Basis des angegebenen Meta Models.

2.3 Graph Mining

Das Hauptaugenmerk dieser Thesis liegt auf der Identifikation von Konzepten in Graphdatensätzen. Da die Daten eines Model Repository schon aufgrund der Beschaffenheit strukturiert sind, bietet sich die Aufbereitung der Daten als Graphdarstellung besonders an. Modelle in Model Repositories können sogar mehr Struktur aufweisen als Graphen. Die Arbeiten von G. Taentzer und K. Ehrig rund um das Eclipse Modeling Framework, insbesondere die Publikation *The EMF Model Transformation Framework* [22] beschreiben diese Eigenschaften. Daher lassen sich die Modelle in Graphen umwandeln, da für eine Graphenrepräsentation nicht mehr Struktur notwendig ist, als die Modelle ohnehin schon mit sich bringen. Graphdarstellungen sind in der Regel recht unkompliziert einzusetzen und eignen sich daher hervorragend als Repräsentation von komplexen Zusammenhängen. Darüber hinaus existieren viele Forschungsarbeiten rund um das Thema Graph Mining, weshalb auf einen großen Wissensfundus für die Implementierung der Identifikation von Konzepten in Graphen zurückgegriffen werden kann. In der Arbeit *Graph mining: A survey of graph mining techniques* [23] wird darauf näher eingegangen. Diese Erkenntnisse

lassen sich somit ebenso auf Model Repositories anwenden, die als Graphen abgebildet werden. Ziel dieses Kapitels ist es, die theoretischen sowie technischen Grundlagen von Graph Mining verständlich zu machen. Obwohl die Repräsentation von Daten als Graphen ein unkompliziertes Mittel ist, um komplexe Verknüpfungen einfach verständlich darzustellen, so ist der algorithmische Aufwand auf Graphen nicht immer trivial und kann sich als komplexe Herausforderung darstellen. Dies kann aus der Veröffentlichung *Reducibility Among Combinatorial Problems* [24] entnommen werden. Doch bevor im weiteren Verlauf auf die Komplexität von Graph Mining Algorithmen eingegangen wird, folgt zunächst eine Einführung über Graph Mining mit einem Blick auf die grundlegenden Definitionen aus der Graphentheorie.

Graph Mining kann dazu eingesetzt werden, Muster aus Daten zu extrahieren, um im Zuge weiterer Verarbeitungsschritte die zugrundeliegenden Daten zu interpretieren und auszuwerten. Dabei werden die Eigenschaften und Strukturen von Graphen genutzt, um zusätzlich interessante Informationen aus den Daten zu generieren. Die Anwendungsfälle von Graphen sind zahlreich, so können komplexe Datensätze bspw. Freundeskreise in sozialen Netzwerken, Proteinstrukturen in der Biologie oder Verbundstoffe und Zusammensetzungen in der Chemie und vieles mehr als Graphen abgebildet werden. Mit der Erkenntnis, dass sehr viele Probleme in der echten Welt mithilfe von Graphen abgebildet werden können, stiegen auch die Bemühungen, immer bessere Graph Mining Algorithmen zu entwickeln. Dadurch wurde die Erkennung von Mustern viel effizienter und die Interpretation und Auswertung der Daten immer hochwertiger.

Es existieren verschiedene Anwendungsgebiete im Bereich Graph Mining. Diese Thesis behandelt insbesondere das Subgraph Mining auf Basis des Pattern Growth Ansatzes, mithilfe dessen sich bestimmte Muster innerhalb eines Graphen identifizieren lassen. Pattern Growth bezeichnet damit das Verfahren, die Subgraphen Schritt für Schritt zu erweitern, um sich somit immer näher an das beste Pattern zu bewegen. Die bekanntesten Subgraph Mining Algorithmen dieser Art lassen sich aus der Tabelle 2.3 entnehmen, die aus der Veröffentlichung *Frequent Subgraph Mining Algorithms - A Survey and Framework for Classification* [3] stammt.

2.3.1 Definitionen

Die Graphentheorie befasst sich mit dem mathematischen Modell eines Graphen und deren Eigenschaften sowie die Beziehungen der darin enthaltenen Elemente. Ein Graph G besteht aus einer Menge von Knoten V und Kanten E, die zueinander in Beziehung stehen. Der Graph selbst, aber auch die Knoten und Kanten können bestimmte Eigenschaften besitzen. Viele Probleme aus der realen Welt lassen sich in eine Graphendarstellung überführen. Die Anwendungsgebiete sind zahlreich, so lassen sich bspw. soziale Netzwerke, chemische Moleküle, Versorgungsnetze, Routenplanungen und vieles mehr untersuchen. Werden diese Daten als Graphen abgebildet, können spezielle Graphalgorithmen ausgeführt werden, um Lösungen für die verschiedensten Problemstellungen aus den unterschiedlichsten Anwendungsgebieten berechnet werden. Auch in dieser Thesis werden die Model Repositories als Graphen repräsentiert und besitzen somit Eigenschaften aus der Graphentheorie,

Algorithmus	**Pattern Auswahl**	**Limitierung**
Subdue	Kompression nach dem Minimum Description Length Prinzip	Geringe Anzahl von Pattern werden identifiziert
gSpan	Tiefensuche und lexikografisch Reihenfolge	Probleme in der Skalierbarkeit
Close Graph	Tiefensuche und lexikografisch Reihenfolge	Bringt Overhead an Laufzeit mit
Gaston	Anzahl an Vorkommen eines Patterns	Interessante Pattern gehen möglicherweise verloren
TSP	Travelling salesman problem tree	Bringt Overhead an Laufzeit mit
MOFA	Tiefensuche und lexikografisch Reihenfolge	Berechnung für interessante Pattern möglicherweise nicht genau
RP-FP	Tiefensuche und lexikografisch Reihenfolge	Bringt Overhead an Laufzeit mit
RP-GD	Tiefensuche und lexikografisch Reihenfolge	Bringt Overhead an Laufzeit mit
JPMiner	Tiefensuche und lexikografisch Reihenfolge	Gelegentlich wird nur eine geringe Anzahl an Pattern erkannt
MSPAN	Tiefensuche und lexikografisch Reihenfolge	-

Tabelle 2.3: Bekannte Subgraph Mining Algorithmen auf Basis des Pattern Growth Ansatzes.

die Subgraph Mining Algorithmen ausnutzen, um bessere Ergebnisse zu erzielen.

Subgraph

Sei Graph G gegeben, dann ist S ein Subgraph, wenn S aus einer Teilmenge an Knoten und Kanten aus G besteht. Somit ist S stets mindestens einmal in G enthalten. *Graph* $S = (V', E')$ ist Subgraph von *Graph* $G = (V, E)$, wenn gilt: $V' \subseteq V$, $E' \subseteq E$ und $((v_1, v_2) \in E' \rightarrow v_1, v_2 \in V')$.

Labeled Graph

Die Knoten und Kanten eines Labeled Graph werden einer Beschriftung zugeordnet. Die Beschriftung konkretisiert die Beziehung der Elemente im Graphen. Anders als bei einem Weighted Graph, der nur Kantenbeschriftungen einer geordneten Menge bspw. von reellen Zahlen erlaubt, so gelten für den Labeled Graph keine solchen Einschränkungen. Die Menge der Beschriftungen kann also beliebig gewählt werden.

Mixed Graph

Die Kanten eines Graphens können entweder gerichtet oder ungerichtet sein. Eine gerichtete Kante gibt eine bestimmte Orientierung mit einer klaren Richtung vor,

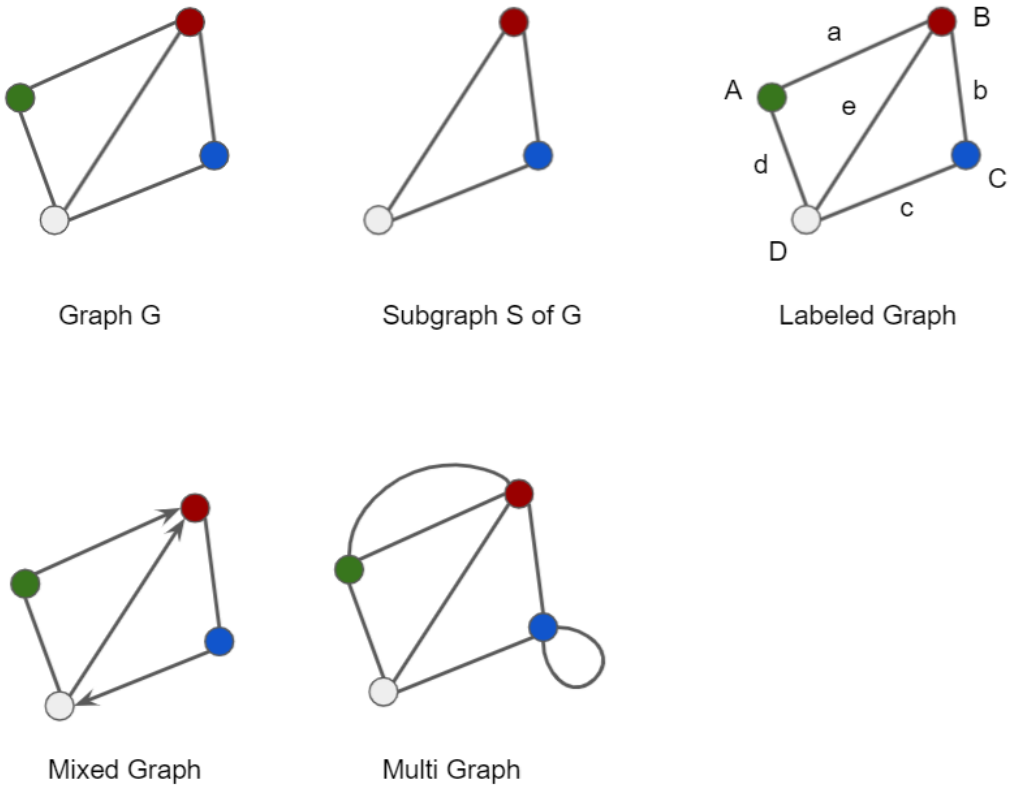

Abbildung 2.4: Graph Definitionen

hingegen die Orientierung einer ungerichteten Kante beidseitig gilt. Gegeben seien die Knoten $u, v \in V$, so wird die Orientierung einer gerichteten Kante wie folgt definiert: (u, v), wobei u der Beginn der Orientierung und v das Ende der Pfeilrichtung definiert. Eine ungerichtete Kante ohne Orientierung wird $[u, v]$ notiert. Ein Mixed Graph hingegen kann beide Arten von Kanten besitzen, die enthaltenen Kanten können also gerichtet oder ungerichtet sein. Der *Mixed Graph* $G = (V, E, A)$ enthält also zusätzlich zu den ungerichteten Kanten E noch eine Menge an ungerichteten Kanten A.

Multi Graph

Der Multi Graph erlaubt, dass zwei Knoten auch durch mehrere Kanten verbunden werden dürfen. Somit werden parallele Kanten zugelassen, also Kanten, die denselben Anfangsknoten u und denselben Endknoten v haben für die $u \neq v$ gilt. Zusätzlich sind Schleifen erlaubt, d.h. Kanten dürfen denselben Anfangsknoten u und denselben Endknoten v haben für die $u = v$ gilt.

2.3.2 Komplexität

Problemstellungen in der Informatik können in verschiedene Komplexitätsklassen eingeteilt werden. Die Komplexitätsklasse gibt an, wie schwierig ein Problem zu lösen ist. Das bedeutet, dass es Probleme gibt, die schwieriger sind und mehr Aufwand erfordern, als andere. Eines der bedeutendsten Werke im Bereich der Komple-

xitätstheorie *The Complexity of Theorem-Proving Procedures* [25] wurde 1971 von S. Cook publiziert. Zur gleichen Zeit forschte L. Levin an der gleichen Sache und veröffentlichte ähnliche Forschungsergebnisse wie S. Cook, weshalb die Erkenntnisse heute auch unter dem Cook–Levin Theorem bekannt sind. Das Resultat der Ergebnisse begründet eine neue Klasse von Problemen, genannt NP-complete. Probleme, die in diese Klasse fallen, gehören zu den schwierigsten Problemen und können vermutlich nicht effizient gelöst werden. Ob die Probleme in NP-complete wirklich nicht effizient gelöst werden können, ist eine noch ungelöste Frage in der Informatik. Eine formale Beschreibung wurde ebenfalls von S. Cook in *The P versus NP problem* [26] veröffentlicht.

Graph Isomorphismus

Für das Identifizieren von Edit Operations in Model Repositories spielt das P versus NP Problem eine bedeutende Rolle. Um Edit Operations mit dem SUBDUE Algorithmus identifizieren zu können, muss regelmäßig überprüft werden, ob zwei Graphen isomorph zueinander sind. Das Graphen-Isomorphismus-Problem liegt in der Komplexitätsklasse NP. Bisher wurde noch kein NP-complete Beweis gefunden, es ist derzeit aber auch noch nicht bekannt, ob es in polynomieller Zeit gelöst werden kann. Auch wenn die Klassifizierung des Graphen-Isomorphismus-Problem noch eine offene Frage ist und noch kein allgemein gültiger effizienter Algorithmus gefunden wurde, existieren in der Praxis approximative Algorithmen, die die Isomorphie von Graphen annähernd effizient lösen können. Dies kann aus der Publikation *Practical Graph Isomorphism* [27] aus dem Jahr 1981 von B. McKay entnommen werden.

Der Graph Isomorphismus ist wie folgt definiert. Gegeben seien zwei Graphen G und H. Beide Graphen sind zueinander isomorph, wenn sie die gleiche Anzahl an Knoten und Kanten besitzen, wobei für alle Kanten gilt, dass die Kantenverbindungen zwischen den Knoten in G auch in H stets erhalten bleiben. Sind die Graphen zueinander isomorph, so können zwei beliebige Knoten u und v aus dem Graphen G ausgewählt werden. u und v sind benachbarte Knoten in G, wenn $f(u)$ und $f(v)$ ebenfalls in H benachbart sind. Es existiert also eine Bijektion zwischen den Knoten- und Kantenmengen von beiden Graphen G und H. In anderen Worten, zwei Graphen sind zueinander isomorph, wenn sie strukturell identisch sind. Ein anschauliches Beispiel kann aus der Abbildung 2.5 entnommen werden.

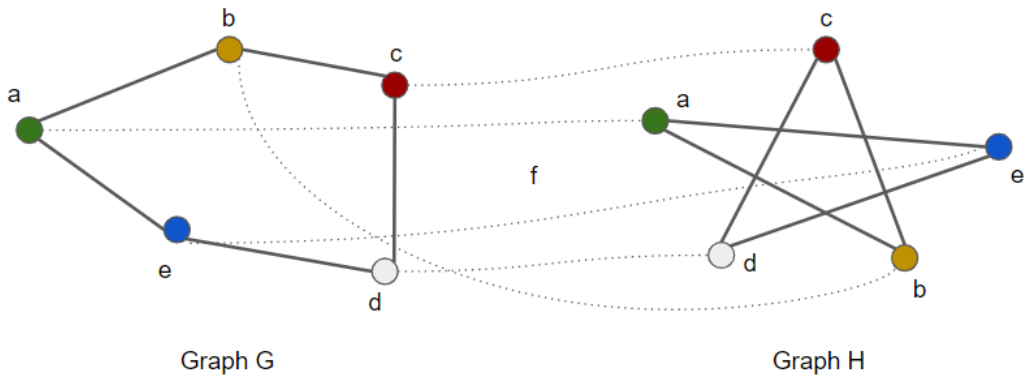

Abbildung 2.5: Graph Isomorphismus

Subgraph Isomorphismus

Im Gegensatz zum Graph-Isomorphismus-Problem, bei dem nicht bekannt ist, ob es in Polynomialzeit gelöst werden kann oder ob es in der Komplexitätsklasse NP-complete liegt, ist das Subgraph-Isomorphismus-Problem der Komplexitätsklasse NP-complete zuzuordnen und ist somit von Natur aus schwierig zu lösen. Nachvollziehen lässt sich dies in der Veröffentlichung Computers and Intractability; A Guide to the Theory of NP-Completeness [28], in der unter anderem das Subgraph-Isomorphismus-Problem verdeutlicht wird. Dies hat zur Folge, dass es vermutlich keinen Algorithmus geben kann, der dieses Problem effizient berechnet, da die Rechenzeit exponentiell in Bezug auf die Anzahl der Knoten zunimmt. Es ist also kein effizienter Algorithmus bekannt, der in Polynomialzeit prüfen kann, ob ein Subgraph-Isomorphismus auf einem Graphen gegeben ist.

Ein Subgraph Isomorphismus ist wie folgt definiert. Gegeben seien zwei Graphen G und H, wobei H' ein beliebiger Subgraph von H ist. Der Graph G ist subgraphisomorph zu H, wenn H' isomorph zu G ist. Anders als für den Graph-Isomorphismus ist hier keine bijektive Abbildung zwischen den beiden Graphen gefordert, es muss lediglich eine Bijektion zwischen einem Graph und dem Subgraph, zu dem er Subgraph isomorph ist, existieren.

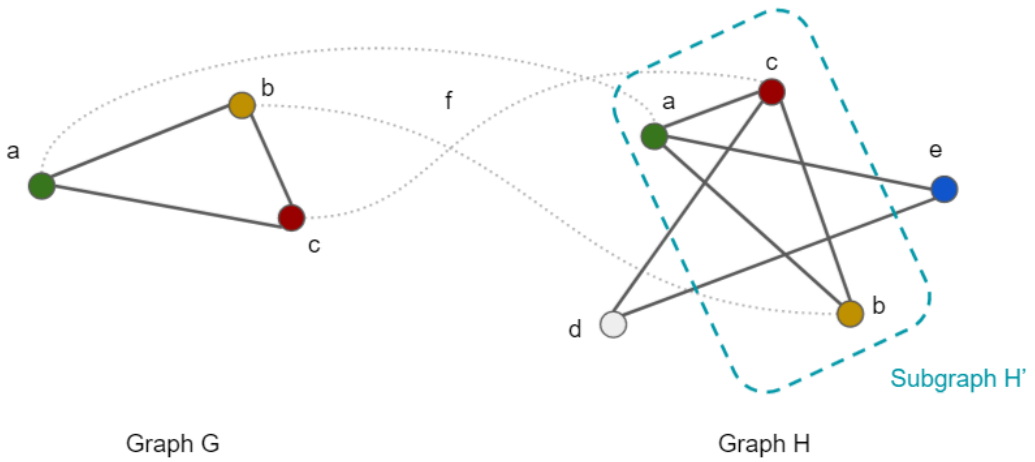

Abbildung 2.6: Subgraph Isomorphismus

2.3.3 Frequent Subgraph Mining

Frequent Subgraph Mining Algorithmen sind eine Klasse von Algorithmen, mit denen sich interessante Subgraphen auf Graphen identifizieren lassen. Mit dem Frequent Subgraph Mining verfolgt man das Ziel, alle Subgraphen in einem Graphen zu identifizieren, deren Häufigkeit über einem festgelegten Schwellenwert liegt. Je häufiger ein Subgraph im Graphen vorkommt, desto mehr Informationen liefert er dem Graphen, weshalb er interessanter zu sein scheint als andere Subgraphen. Innerhalb der Frequent Subgraph Mining Algorithmen gibt es wiederum unterschiedliche Vorgehensweisen. Ein interessanter Frequent Subgraph Mining Ansatz, mit dem sich

ebenfalls das Identifizieren von interessanten Edit Operations in Model Repositories umsetzen ließe, ist das Prinzip des Pattern Growth. Dies ist auch jener Ansatz, mit dem C. Tinnes in *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] Edit Operations identifiziert. Diese Thesis beschäftigt sich wiederum mit einem kompressionsbasierten Verfahren, welches sich zwar dem Frequent Subgraph Mining ähnelt, sich jedoch hauptsächlich in der Metrik für die Bestimmung der interessanten Pattern unterscheidet.

2.3.4 Minimum Description Length

Im Folgenden wird die grundlegende Problematik aufgegriffen, wie sich Daten anhand von Modellen beschreiben lassen, um damit das Prinzip der Minimum Description Length für das Subgraph Mining zu motivieren. Durch die Minimum Description Length wird ein kompressionsbasierter Ansatz beschrieben mit dem sich Subgraphen in Graphen erkennen lassen. Dieser Ansatz wird für den zugrundeliegenden SUBDUE verwenden, auf dem der THEOBALDSUBDUE aufbaut. Die Minimum Description Length wurde 1978 von J. Rissanen in der Veröffentlichung *Modeling By Shortest Data Description* [29] erstmalig in der Informationstheorie eingeführt. Die Grundidee der Minimum Description Length basiert auf dem heuristischen Occam's Razor Prinzip. Zurückzuführen ist dieser Ansatz auf den Philosophen Wilhelm von Ockham, der im 14. Jahrhundert gelebt hat. Eine ausführlichere Beschreibung dieses Prinzips und seiner Grundideen finden sich in dem Buch *Maximum Entropy and Bayesian Methods* [30]. Für den Zweck dieser Thesis genügt es eine einfache Vorstellung davon zu haben. Occam's Razor besagt, dass unter einer Menge von Theorien, die versuchen den gleichen Sachverhalt zu erklären, jene die beste ist, die auch die einfachste darstellt. In anderen Worten ausgedrückt, die einfachste Theorie für ein Erklärungsmodell ist den anderen vorzuziehen. Dieses heuristische Prinzip ist fundamental für das Prinzip der Minimum Description Length.

Wenn aus einem Datensatz neue Informationen gewonnen werden sollen, muss ein geeignetes Modell gefunden werden, um die Informationen zu beschreiben. Eine wesentliche Aufgabe im Data Mining ist es deshalb, Modelle zu erstellen, um anschließend mithilfe dieser Modelle eine Beschreibung der Daten zu erwirken. Die Schwierigkeit folgt aufgrund der wechselseitigen Wirkung von den Daten und den Beschreibungsmodellen. Die Beschreibungslänge $L(D)$ der Daten setzt sich aus den folgenden zwei Termen zusammen. Zum einen aus der Länge $L(M)$ für die Beschreibung des Modells und zum anderen Kosten $L(D|M)$ für die Beschreibung der Daten, die mithilfe dem Modell durchgeführt wird. Daraus lässt sich die Beziehung zwischen den Daten und seinem Beschreibungsmodell veranschaulichen: $L(D) = L(M)+L(D|M)$. Ziel ist es, die Daten bestmöglich zu beschreiben und dabei die Kosten für das Beschreibungsmodell und die Kosten für die Anwendung des Beschreibungsmodells auf den Daten minimal zu halten. Für ein optimales Ergebnis muss also die Summe aus $L(M) + L(D|M)$ bestmöglich minimiert werden.

P. Grünwald erwähnt in seinem Buch *The Minimum Description Length Principle* [5], dass die Minimum Description Length erfolgreich eingesetzt wird, um induktive Inferenzprobleme zu lösen. Das bedeutet, mithilfe der Minimum Description Length können Schlussfolgerungen getroffen werden, die trotz fehlender Sachlage

sehr wahrscheinlich sind, indem Verallgemeinerungen getroffen werden. Es hat sich herausgestellt, dass das Prinzip der Minimum Description Length insbesondere gute Ergebnisse liefert, um die oben genannte Schwierigkeit der Modellauswahl für die Beschreibung von Daten zu treffen. Die Gründe lassen sich anhand des Schaubilds 2.7 erklären, welches ebenfalls in dem Buch von P. Grünwald zu finden ist. Ein geeigneter Kompromiss ist es, eine Näherung zu erreichen, mit der versucht wird, einen guten Trade-off zu erzielen. Ein Beschreibungsmodell für Daten muss also nicht unbedingt komplex sein, um wertvolle Informationen aus den Daten abzuleiten. Dass ein solcher Kompromiss auch in der Praxis für reale Daten zu besseren Ergebnissen führt, wurde in mehreren Publikationen nachgewiesen und kann ebenfalls in dem Buch von P. Grünwald nachgelesen werden. Das Lösen von induktiven Inferenzproblemen, insbesondere der Modellauswahl, kann im Subgraph Mining genutzt werden, um interessante Subgraphen in Graphen ausfindig zu machen.

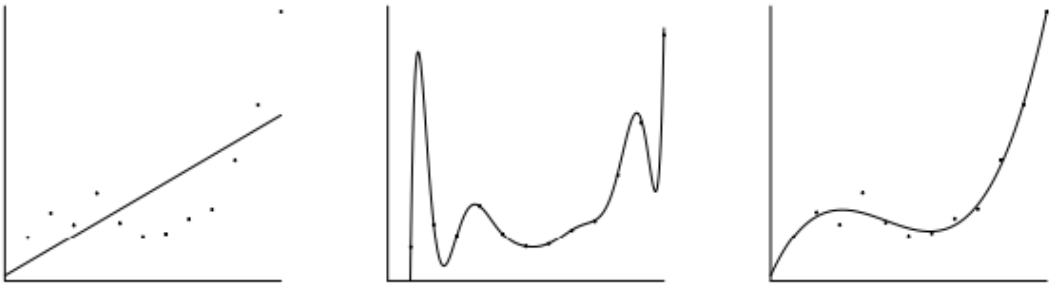

Abbildung 2.7: Minimum Description Length Trade-off.

Im Folgenden wird der Ansatz der Kompression näher betrachtet, die auf der Minimum Description Length basiert. Mithilfe einer Kompression ist es möglich, die Daten durch ihre zugrundeliegende Ordnung und die darauf gefundenen Regelmäßigkeiten zu komprimieren. Obwohl die Kompression die Datenmenge reduziert, bleibt der Informationsgehalt der Daten gleich. Diese Kompression ist sogar verlustfrei, wie aus der Publikation *Minimum description length vs. maximum likelihood in lossy data compression* [31] zu entnehmen ist. Die Minimum Description Length ist also eine Methode, mit der durch Kompression Daten komprimiert werden. Je besser sich die Daten komprimieren lassen, desto größer ist die zugrundeliegende Ordnung in den Daten, was wiederum zu einer besseren Identifizierung von interessanten Konzepten führt. Wird also eine Kompression auf Daten angewandt, so tritt ein gewisser Lerneffekt auf. Diese Schlussfolgerung mag zwar auf den ersten Blick nicht trivial erscheinen, aber das folgende intuitive Beispiel aus der realen Welt veranschaulicht die Wirksamkeit der Kompression in Bezug auf das Lernen von Konzepten. Tatsächlich kann das Lernen eines Kindes auch mit der Methodik der Kompression beschrieben werden. Kinder werden von Grund auf mit Konzepten vertraut gemacht, die sich die Menschheit ausgedacht und in Worte gefasst hat. Im Alltag denken wir oft gar nicht über die genaue Bedeutung unserer Wörter und der ihnen zugrunde liegenden Konzepte nach. Diese Wörter werden Kindern von Anfang an beigebracht. Jedes Kind weiß, was ein Wald und was ein Baum ist, ohne wirklich alle Eigenschaften

und Informationen über einen Wald kennen zu müssen. So muss ein Kind nicht wissen, wie Fotosynthese funktioniert, obwohl das ein grundlegender Bestandteil eines Waldes ist. Detailwissen ist also nicht unbedingt notwendig, um ein Konzept zu verstehen. Das Wort Wald ist ein von Menschen erfundenes Konzept, um alle darunterliegenden Konzepte, Eigenschaften, Gegenstände und Erfahrungen in einem Wort zu beschreiben. Streiche man nun das Wort Wald aus unserem Wortschatz, so würde man Wörter wie Bäume, Blätter, Sträucher, Tiere, Blumen, usw. nutzen, um das Konzept Wald zu beschreiben. Diese Worte sind ebenfalls Konzepte, die sich in weitere Wörter zerlegen lassen. Ein Konzept scheint also aus einer Hierarchie von weiteren Informationen zu bestehen. Mithilfe eines Konzeptes werden also zugrundeliegende Informationen abstrahiert, ohne dass man sich um mehr Detailwissen bemühen muss. Am Beispiel des Waldes und seiner Hierarchie an Konzepten ist leicht zu erkennen, dass für das Konzept des Waldes eine Kompression von Informationen stattgefunden hat.

Doch an welchen Kriterien lässt sich entscheiden, ob eine Kombination an Informationen zu einem Konzept wird? Es scheint, dass Menschen diesen Prozess intuitiv im Allgemeinen sehr effektiv anwenden. In der Abbildung 2.8 sind drei Schaubilder dargestellt. Dort sind zwei Konzepte abgebildet, die wohl jedem bekannt sind: Ein Wald und ein Baum. Das dritte Bild zeigt einen Baum, in dessen Nähe Pilze wachsen. Für diesen Informationsgehalt existiert kein eigenes Konzept. Offenbar ist dieses Konzept nicht interessant genug, um ihm einen eigenständigen Begriff zu geben. Diese menschliche Intuition lässt sich mit dem Prinzip der Minimum Description Length erklären. Wie bereits erwähnt, sind diejenigen Konzepte interessant, die die Daten und somit die Informationen am besten komprimieren. Übertragen auf das obige Beispiel bedeutet dies, dass das Konzept des Baumes mit den Pilzen die Informationen im Vergleich zu anderen Konzepten schwächer komprimiert. Pilze wachsen nicht neben jedem Baum, aber jeder Wald besteht aus Bäumen und jeder Baum besteht aus einem Stamm und einer Krone.

Abbildung 2.8: a) Konzept Wald b) Konzept Baum c) Kein versprachlichtes Konzept.

Diese Eigenschaft der Kompression wird in der statistischen Inferenz genutzt, um empirisch aus Daten zu lernen. Auch P. Grünwald beschreibt in seinem Buch, dass das Finden und Lernen von interessanten Mustern in Daten mit der Fähigkeit zur Kompression gleichgesetzt werden kann.

> The goal of statistical inference may be cast as trying to find regularity in the data. “Regularity” may be identified with “ability to compress.”

> MDL combines these two insights by viewing learning as data compression: it tells us that, for a given set of hypotheses H and data set D, we should try to find the hypothesis or combination of hypotheses in H that compresses D most.
>
> (P. Grünwald)

Der SUBDUE Algorithmus ist ein Subgraph Mining Algorithmus, der mithilfe der Minimum Description Length und dem Ansatz der Kompression interessante Subgraphen in Graphen erkennen kann. Somit ist der SUBDUE in der Lage, signifikante Konzepte in Daten zu erkennen. Dass das Subgraph Mining auf Basis von Kompression in der Praxis auch tatsächlich erfolgreich eingesetzt wird, zeigen die zahlreichen Anwendungen und Evaluationen des SUBDUE Algorithmus, die in Veröffentlichungen von L. Holder publiziert wurden.

Zuvor wurde bereits das Prinzip des Frequent Subgraph Minings besprochen, das auf Grundlage von Häufigkeit interessante Subgraphen identifiziert. Eine weitere Klasse von Subgraph Mining Algorithmen sind die kompressionsbasierten Algorithmen, die mithilfe von Kompression interessante Subgraphen erkennen. Für den kompressionsbasierten Ansatz kann das Prinzip der Minimum Description Length angewendet werden. Beide Ansätze zum Subgraph Mining unterscheiden sich in den zugrunde liegenden Prinzipien, was Auswirkungen auf die Leistung mit sich bringt und somit das Problem der Subgraphen Erkennung unterschiedlich performant gelöst werden kann. Dies bestätigt auch die Studie *Graph-Based Knowledge Discovery: Compression Versus Frequency* [11] von W. Eberle und L. Holder. Aufgrund der wesentlichen Unterschiede in den Prinzipien kann es zu starken Leistungsunterschieden in der Laufzeit und in der Genauigkeit der Mustererkennung kommen.

Kapitel 3

Subdue

Der Subdue Algorithmus wurde erstmals von L. Holder 1988 in seiner Veröffentlichung *Substructure Discovery in SUBDUE* [32] vorgestellt und ist somit eines der ersten Arbeiten im Bereich Graph Mining, um Muster und Konzepte in strukturellen Daten zu identifizieren. Aufgrund der immer größer werdenden Datenmenge wird es immer interessanter, neue Muster und Konzepte in Daten zu erkennen. Oft liegen die Daten in einer strukturierten Form vor. Es stehen also nicht nur die eigentlichen Datenwerte zur Verfügung, sondern aufgrund der strukturierten Form, in der die Daten gespeichert sind, können weitere Informationen anhand der Struktur abgeleitet werden. Aufgrund der enormen Menge an Daten wird es allerdings immer schwieriger neue Zusammenhänge auf Basis der Struktur zu erkennen. Diese Strukturen zu untersuchen kann zu neuen Erkenntnissen und zu mehr Wissen über die Daten führen. Um dieses neue Wissen zu generieren, muss zunächst ein System vorhanden sein, das interessante Muster auf Basis der Strukturen erkennen kann. Strukturelle Daten können aufgrund ihrer Eigenschaften in eine Graphdarstellung überführt werden, was Graph Mining zu einem idealen Ansatz macht, um neues Wissen aus den vorhandenen Strukturen zu generieren. Subdue ist ein Algorithmus, der interessante Substrukturen in Graphen identifizieren kann, um neue und wesentliche Konzepte in den vorhandenen Daten zu entdecken. Die Schwierigkeit dieses Entdeckungsprozesses liegt darin, zwischen wichtigen und unwichtigen Informationen zu unterscheiden und im weiteren Verlauf nur die jeweils wichtigsten Informationen für die Identifizierung der Substrukturen zu verwenden. L. Holder beschreibt die Wichtigkeit dieser Eigenschaft wie folgt: So wie der Mensch auch in der Lage sei, wichtige von unwichtige Informationen zu unterscheiden, so müsse auch ein maschinelles Lernsystem diese Fähigkeit besitzen. Ist ein solches System in der Lage, die wirklich wichtigen und interessanten Dinge aus einer Vielzahl an strukturierten Daten zu erkennen, so können die wirklich wesentlichen Konzepte in den Daten identifiziert werden. Dabei werden unwichtige Strukturen ignoriert und nicht weiter verfolgt.

> Machine learning systems that operate in such a detailed structural environment must be able to abstract over unnecessary detail in the input and determine which attributes are relevant to the learning task.
>
> (L. Holder)

Mit dem Subdue Algorithmus schlägt L. Holder eine Möglichkeit vor, dieses Verfahren eines maschinellen Lernsystems anzuwenden, um damit die wirklich interessanten Substrukturen in Graphen zu finden. Die Funktionalität des Subdue Algorith-

mus ist für den weiteren Verlauf dieser Arbeit von wesentlicher Bedeutung, da der TheobaldSubdue Algorithmus auf dem Subdue basiert. Daher werden im Folgenden die algorithmischen Konzepte erläutert, die der Subdue für die Identifizierung interessanter Subgraphen verwendet und somit ebenfalls im TheobaldSubdue untergebracht sind.

3.1 Algorithmische Konzepte

Der Subdue Algorithmus bedient sich mehrerer algorithmischer Konzepte, die in diesem Unterkapitel erläutert werden. Durch die Kombination dieser Konzepte ist es dem Subdue möglich, die interessante Substrukturen auf Basis von Kompression zu erkennen.

3.1.1 Minimal Expansion

Um die interessantesten Substrukturen erkennen zu können, benötigt es einen Mechanismus, der schon zuvor gefundene Substrukturen erweitert. Die dadurch erzeugten Substrukturen können im weiteren Verlauf einer Prüfung unterzogen werden, ob diese durch das Hinzufügen weiterer Substruktur interessanter geworden sind. Dieser Mechanismus wird unter anderem in der Veröffentlichung *Substructure Discovery Using Minimum Description Length and Background Knowledge* [33] beschrieben und wird im Folgenden als Minimal Expansion bezeichnet. Die Minimal Expansion beschreibt den Vorgang einer minimalen Erweiterung einer Substruktur. Das bedeutet, dass für eine ausgewählte Substruktur alle kleinstmöglichen Erweiterungen erzeugt werden und daraus eine Liste aller möglichen Erweiterungen generiert wird. Jede einzelne Erweiterung der Substruktur wird also um genau eine weitere Substruktur der Größe eins erweitert. Für das Verfahren der Minimal Expansion im Subdue Algorithmus wird ein Subgraph übergeben der anschließend um alle möglichen Erweiterungen mit genau einer benachbarten Kante oder einem benachbarten Knoten erweitert wird. In der Abbildung 3.1 wird die Minimal Expansion auf einem Graph beispielhaft dargestellt. Ausgangspunkt ist die Kante a, die zuvor bereits als interessanter Subgraph ausgewählt wurde und nun erweitert werden soll. Im nächsten Schritt werden alle benachbarten Knoten der Kante a für die Erweiterung berücksichtigt. Dies wird für alle Instanzen der des Patterns durchgeführt. In dem abgebildeten Beispiel existieren zwei unterschiedliche Möglichkeiten, die Kante a zu erweitern. Es wird also beiden Varianten nachgegangen und jeweils ein erweiterter Subgraph aus der Kante a und dem jeweiligen Knoten erstellt. Das Resultat der vorliegenden Minimal Expansion sind zwei neue Subgraphen, die aus dem vorherigen Subgraph generiert wurden.

3.1.2 Kompression

Kompression ist das essenzielle Konzept, mit der interessante Subgraphen auf dem Eingabegraph identifiziert werden. Wie bereits im Abschnitt 2.3.4 erwähnt, nutzt der Subdue Algorithmus das Prinzip der Minimum Description Length, um den Informationsgehalt der Subgraphen zu gewichten, sodass die wirklich relevanten Subgraphen erkannt werden können. Dadurch werden wichtige Subgraphen von unwichtigen

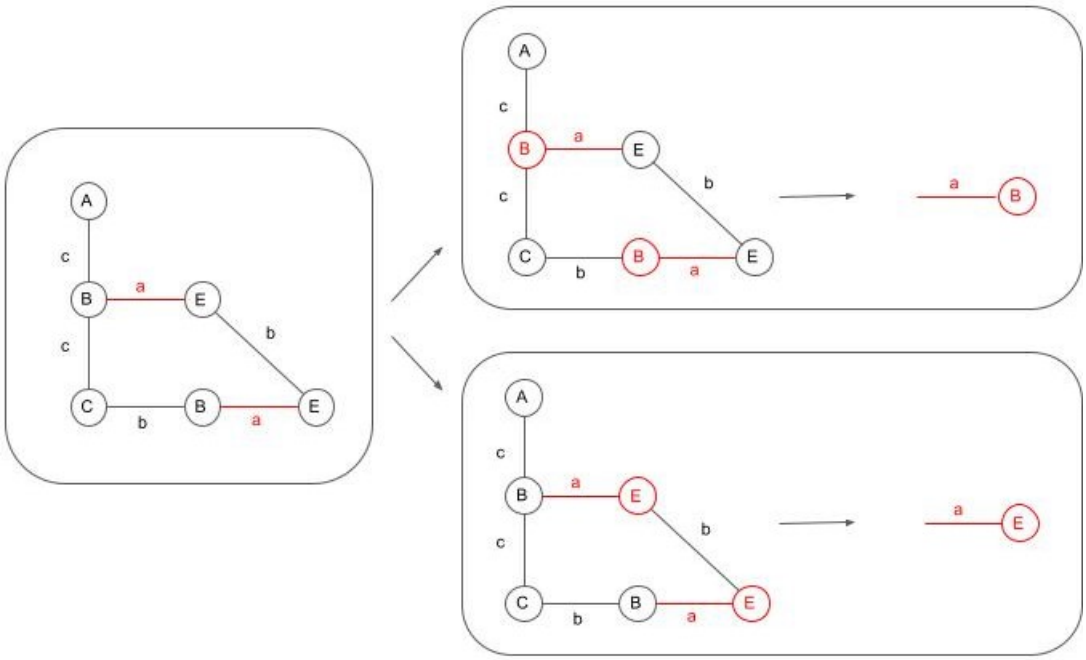

Abbildung 3.1: Minimale Expansion einer Kante in einem Graph.

getrennt und nur die jeweils wichtigsten Subgraphen fließen in die Berechnung zur Identifizierung mit ein. Der Kompressionswert der Subgraphen ist die entscheidende Metrik dafür, wie interessant ein einzelner Subgraph ist. Je höher die Kompression, desto mehr Informationsgehalt verbirgt sich hinter dem Subgraphen und desto wichtiger ist er. Der SUBDUE Algorithmus kann aus mehreren Iterationen bestehen, die jeweils zum Abschluss den Subgraph mit dem höchsten Kompressionswert dafür verwendet, den Eingabegraph zu komprimieren. Dieser komprimierte Graph ist wiederum der Eingabegraph für weitere Iterationen, um neue interessante Subgraphen auf dem komprimierten Graph zu finden. Die Abbildung 3.2 zeigt eine beispielhafte Kompression auf einem gegebenen Graph. Das Pattern $B - a - E$ besitzt den höchsten Kompressionswert auf dem Eingabegraph, weshalb alle Vorkommen dieses Patterns durch ein neues Konzept X ersetzt werden.

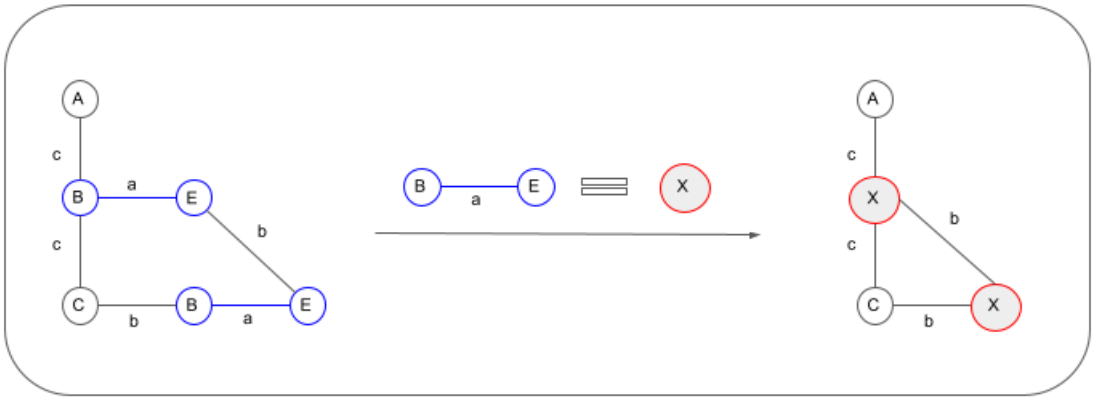

Abbildung 3.2: Anwendung einer Kompression mit einem Subgraphen auf einem Eingabegraph.

3.1.3 Beam Search

Das Suchverfahren nach den besten Pattern des SUBDUE Algorithmus basiert auf der Beam Search, die erstmalig von R. Reddy und B. Lowerre im Jahr 1976 in der Veröffentlichung *The Harpy Speech Understanding* [34] erwähnt wurde.

> The beam search technique used by the Harpy system is a heuristic search technique [...]. Beam search is a technique in which a group of near-miss alternatives around the best path are examined. By searching many alternatives simultaneously, this method avoids the need for backtracking.
>
> (R. Reddy und B. Lowerr)

Im Gegensatz zur Greedy Search, die jeweils nur das beste gefundene Objekt für weitere Suchiterationen auswählt, kann die Beam Search also eine beliebige Menge n von Objekten auswählen. Darüber hinaus berücksichtigt die Beam Search die gesamte Sequenz der bisher ausgewählten Objekte, weshalb nicht nur das beste lokale Ergebnis berücksichtigt wird, sondern die besten Ergebnisse in der Kombination der bisher ausgewählten Objekte. Die Berücksichtigung mehrerer Objekte und dessen Sequenz wirkt sich auf die Kosten des Rechenaufwands und der Ausführungszeit aus. Die Beam Search wählt in jeder Iteration die n interessantesten Subgraphen aus dem Eingabegraph aus. Die Metrik für diese Auswahl ist der Wert der Kompression, den der Subgraph auf dem Eingabegraph bewirkt. Es werden also jene Subgraphen ausgewählt, die einen hohen Kompressionswert aufweisen. Das Verfahren der Beam Search ist in Abbildung 3.3 dargestellt. Die dort abgebildete Beam Search beträgt drei Iterationen mit jeweils einer Suchbreite von zwei. In der ersten Iteration werden Subgraphen für die Auswahl bestimmt, die nur aus einer Kante bestehen. Aufgrund der Suchbreite von zwei und auf Basis der zuvor errechneten Kompression der Subgraphen auf dem Eingabegraph werden die zwei besten Subgraphen ausgewählt und bilden somit die Grundlage der nächsten Iteration. Beide ausgewählten Subgraphen werden nun einer Minimal Expansion und einer erneuten Berechnung der Kompression unterzogen. Aus den daraus resultierenden Subgraphen werden erneut die besten zwei Subgraphen mit der höchsten Kompression ausgewählt, um anschließend in die nächste Iteration überzugehen. Dieser Vorgang wird so lange wiederholt, wie Iterationen für die Beam Search definiert sind.

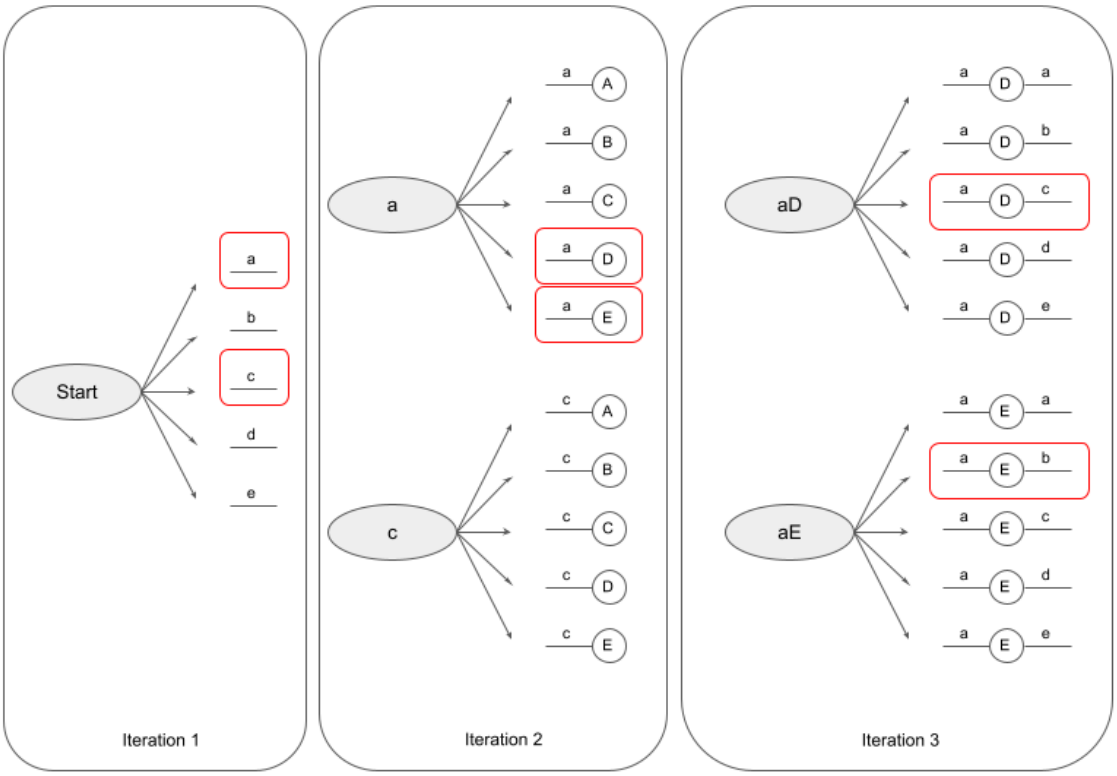

Abbildung 3.3: Verwendung der Beam Search, um eine Auswahl an interessantesten Subgraphen für weitere Iterationen bestimmen.

3.2 Parameter

Der SUBDUE Algorithmus kann durch eine Auswahl an Parametern konfiguriert werden, um die Identifizierung interessanter Subgraphen zu modifizieren. Je nach Parametereinstellung können unterschiedliche Resultate gefunden werden, zudem kann sich die Konfiguration stark auf die Laufzeit auswirken. Die Parameter lassen sich in zwei Kategorien einteilen. Zum einen lässt sich der Suchraum durch eine entsprechende Parameterauswahl einschränken und zum anderen können zusätzliche Funktionen aktiviert werden, um bestimmte Subgraphen zu identifizieren. Es lassen sich keine allgemeingültigen Regeln für die Parameterauswahl aufstellen, mit denen die besten Resultate erzielt werden können. Für die Nutzung in einem gegebenen Anwendungsfall ist es daher sinnvoll, eine empirische Studie mit unterschiedlichen Parametereinstellungen durchzuführen.

BeamWidth b

Mit dem BeamWidth Parameter kann die Breite der Beam Search konfiguriert werden. Für jede neue Iteration werden nur die besten b Pattern beibehalten, die im nächsten Durchlauf erneut einer Minimal Expansion unterzogen werden. Dadurch kann der Suchraum eingeschränkt werden, um die Laufzeit zu verbessern. Die Einschränkung hat jedoch zur Folge, dass eventuell interessante Pattern übersehen werden. Ein solch eingeschränkter Suchraum benachteiligt Pattern, die zu Beginn einen geringen Kompressionswert besitzen, die aber durch die Minimal Expansion zu ei-

nem immer höheren Kompressionswert anwachsen. Bevorzugt werden Pattern, die zu Beginn eine hohe Kompression aufweisen und mit wachsender Größe an Kompression zunehmen, aber auch jene, die mit zunehmender Größe nicht mehr viel an Kompression gewinnen.

Iterations i

Dieser Parameter gibt die Anzahl an Iterationen im Erkennungsprozesses der Pattern vor, die der SUBDUE Algorithmus insgesamt durchläuft. Nach jeweils einer Iteration wird das Pattern, das die höchste Kompression hervorruft, auf dem Eingabegraph angewendet. Der daraus resultierende Graph wird als neuer Eingabegraph für die nächste Iteration genutzt. Bsp: Für $i = 1$ wird nur eine Iteration durchlaufen, was keiner Kompression entspricht. Für $i = 0$ wird so lange iteriert, bis keine Kompression mehr möglich ist. Für alle anderen Fälle $i > 1$ wird der Erkennungsprozess mit Anwendung von Kompressionen mehrmals durchlaufen.

Limit l

In Abgrenzung zum Paramater Iterations bezieht sich die Konfiguration des Limits auf die Anzahl der Durchläufe der Minimal Expansion und der Beam Search innerhalb einer Iteration des Erkennungsprozesses. Limit bestimmt also die Anzahl an Iterationen eines einzelnen Erkennungsprozesses. Der Parameter Limit gibt also an, dass l Pattern insgesamt in einem Substructure Discovery Durchlauf berücksichtigt werden. Je nach Parameterwahl kann die Laufzeit stark beeinflusst werden und die Ergebnisse können stark variieren. Generell lässt sich sagen, dass Limit ein Trade-off zwischen der Laufzeit und der Qualität des identifizierten Patterns regelt.

MaxSize max

Die maximale Größe max gibt die obere Schranke für die Größe der zu identifizierenden Pattern an.

MinSize min

Die minimale Größe min gibt die untere Schranke für die Größe der zu identifizierenden Pattern an.

NumBest n

Bestimmt die Anzahl an interessanten Pattern, die am Ende des SUBDUE Algorithmus ausgegeben werden. Der SUBDUE Algorithmus identifiziert die interessantesten n Pattern und gibt diese aus.

Overlap o

Der SUBDUE Algorithmus speichert alle gefundenen Instanzen eines Patterns im Eingabegraphen. Mit dem Overlap lässt sich die Überlappung dieser Instanzen eines Patterns bestimmen. Wenn keine Überlappung erlaubt ist, kann dies zu einer geringeren Anzahl von Instanzen des Musters führen, wenn sich Instanzen überlappen.

Die Überlappung o kann entweder deaktiviert, nur für Knoten oder für Knoten und Kanten aktiviert werden. $o \in \{none, vertex, edge\}$.

Prune r

Mithilfe des Prunings lässt sich der Suchraum des SUBDUE Algorithmus verkleinern. Der Parameter Prune kann entweder aktiviert $r = True$ oder deaktiviert $r = False$ werden. Ist das Pruning aktiviert, so werden Pattern p, die aus einem Parent Pattern pp erzeugt werden direkt verworfen, falls der Kompressionswert c des Parent Patterns größer ist. Das Pattern p wird also verworfen, wenn gilt: $pp.c > p.c$. Dies kann jedoch zu einem schlechteren Ergebnis führen, da Pattern benachteiligt werden, die in einem frühen Stadium eine geringe Kompression aufweisen, aber mit zunehmender Größe eine erhebliche Kompression erreichen. Auch für diesen Paremeter gilt der Trade-off zwischen der Laufzeit und der Qualität des identifizierten Patterns.

3.3 Datenstrukturen

Um das Verfahren der SUBDUE Implementierung nachzuvollziehen, werden im folgenden Abschnitt die zwei wichtigsten Datenstrukturen vorgestellt, die in der Implementierung von L. Holder verwendet werden. Der SUBDUE Algorithmus verwendet eine eigene Implementierung für die Graph Datenstruktur, die folgende Eigenschaften besitzt. Der Graph ist Attributed, was bedeutet, dass Knoten und Kanten mit einer beliebigen Menge an Attributen beschriftet werden können. Zudem ist der Graph ein gemischter Graph, was bedeutet, dass er sowohl gerichtete als auch ungerichtete Kanten haben kann. Zuletzt hat es die Eigenschaft eines Multigraphen, sodass zwei Knoten auch durch mehrere Kanten verbunden sein können. In der Abbildung 3.4 ist das Model des Graphen zu sehen. Jeder Graph besteht aus Knoten und Kanten, jeder Knoten besitzt dabei eine einzigartige ID, einen Zeitstempel und eine beliebige Menge an Attributen. Ebenso besitzt jede Kante eine einzigartige ID, einen Zeitstempel und eine beliebige Menge an Attributen, gefolgt von der ID des Quellknotens, der ID des Zielknotens und einem booleschen Wert, ob die Kante gerichtet oder ungerichtet ist.

Ein Pattern ist nichts anderes als ein Subgraph, der ein gefundenes Konzept auf dem SUBDUE Eingabegraph repräsentiert. Das Pattern Model ist in Abbildung 3.5 dargestellt, es beinhaltet die Datenstruktur des Subgraphen selbst, einen Kompressionswert der auf Basis des SUBDUE Eingabegraphens berechnet wird und eine beliebige Menge an Graphinstanzen, die aus Kanten und Knoten bestehen. Ein Pattern ist genau dann ein sinnvolles Pattern, wenn der Subgraph mindestens ein weiteres Mal auf dem SUBDUE Eingabegraph vorkommt. Die Graphinstanzen eines Patterns speichern hierfür alle gefundenen Übereinstimmungen, die dem Subgraphen strukturell gleich sind. Ein Pattern beinhaltet also die zugrundeliegende Struktur als Subgraph und alle gefundenen strukturellen Übereinstimmungen. Der Kompressionswert für ein Pattern gibt auf einer Skala von 0 bis 1 an, wie gut die Struktur des Subgraphens den SUBDUE Eingabegraph komprimiert, wobei 0 für keine Kompression und 1 für die ideale Kompression steht. Die Kanten und Knoten der Graphinstanzen werden in einem OrderedSet gespeichert. Ein OrderedSet ist eine Mischung aus einer List

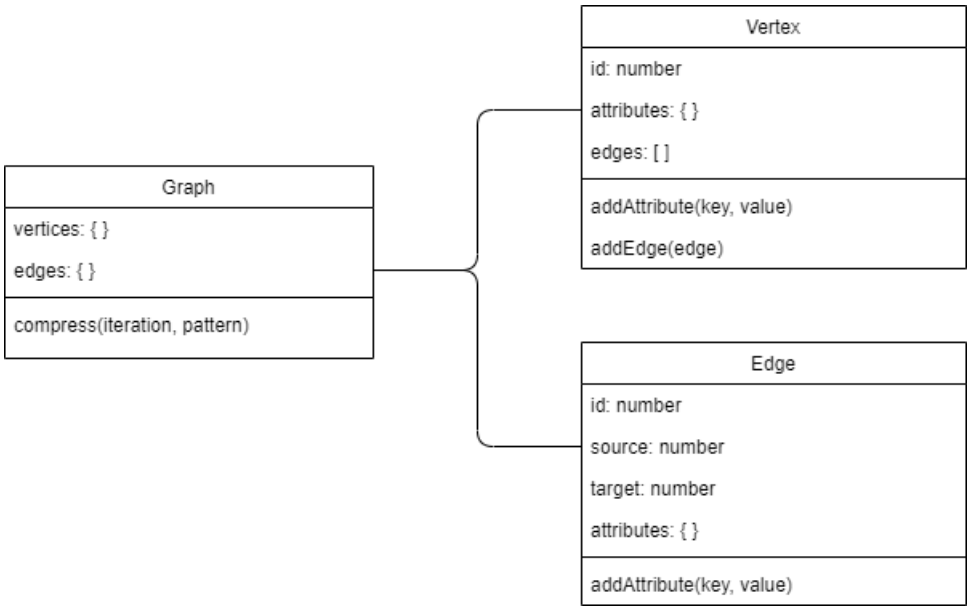

Abbildung 3.4: UML Diagramm der SUBDUE Graph Implementierung.

und einem Set und kann sich die Reihenfolge mithilfe von Indexnummern merken, in denen die Objekte eingefügt werden.

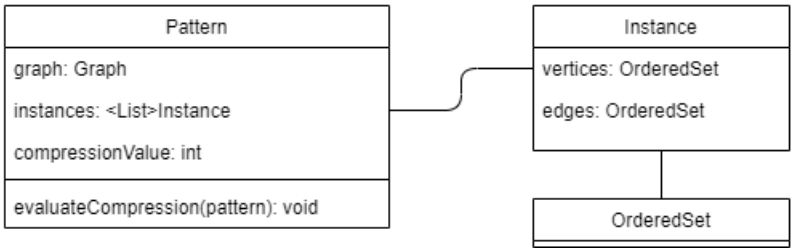

Abbildung 3.5: UML Diagramm der SUBDUE Pattern Implementierung.

3.4 Verfahren

Wie bereits angesprochen ist der SUBDUE ein kompressionsbasierter Algorithmus, der die Minimum Description Length als Metrik verwendet, um jeweils die beste Kompression auf dem Eingabegraph durchzuführen, sodass die interessantesten Pattern entdeckt werden können. Um eine solche Kompression durchführen zu können, müssen zunächst interessante Pattern gefunden und deren Kompressionswert berechnet werden. Im Abschnitt der Minimum Description Length wurde erwähnt, dass ein Pattern umso interessanter ist, je höher sein Kompressionswert auf dem Eingabegraph ist. Zunächst müssen erste Pattern auf dem Graph ausgewählt werden, die als Ausgangslage für den ersten Durchlauf der Minimal Expansion zur Verfügung stehen. Anschließend werden die Kompressionswerte der erzeugten Subgraphen berechnet, die dann den verfügbaren Suchraum für die Beam Search liefern. Dieser Vorgang kann beliebig oft wiederholt werden.

Hierbei lässt sich der SUBDUE Algorithmus in fünf verschiedene Phasen einteilen, die in der Abbildung 3.6 dargestellt wird. Den Hauptbestandteil des Algorithmus bilden dabei zwei Programmschleifen, die in Abhängigkeit der Parametereinstellung durchlaufen werden. Die Substructure Discovery ist die Komponente, in der die eigentliche Identifizierung von Pattern stattfindet. Ein detailliertes Aktivitätsdiagramm der Substructure Discovery kann aus der Abbildung 3.7 entnommen werden. Im Folgenden werden die einzelnen Phasen und das Zusammenwirken für die Substructure Discovery erläutert.

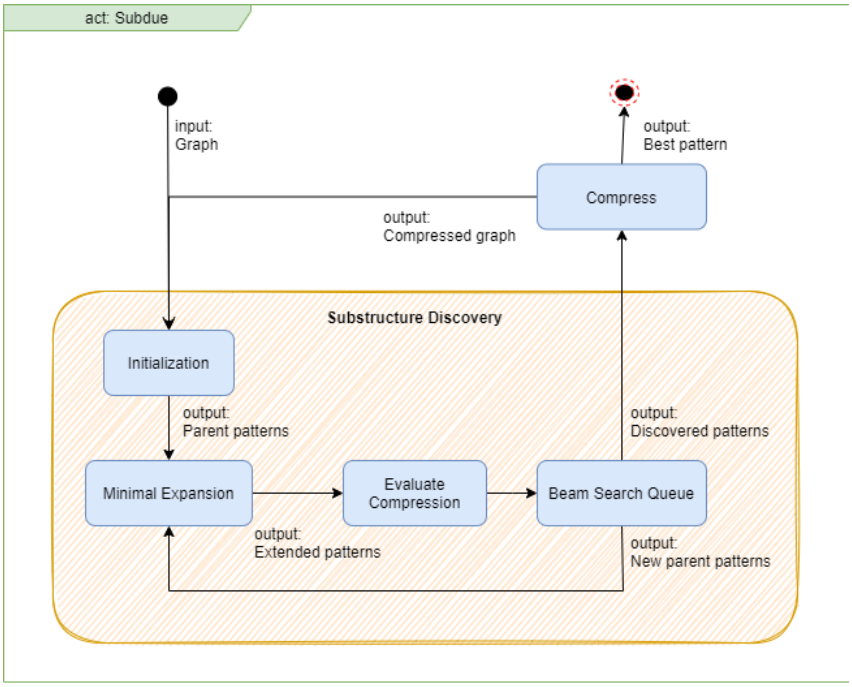

Abbildung 3.6: Subdue Verfahren.

Phase 1: Initialization

Damit der SUBDUE Algorithmus mit der Substructure Discovery und somit mit einer Erweiterung der Pattern mittels Minimal Expansion Verfahren beginnen kann, muss zunächst eine Liste von initialen Pattern erstellt werden. Diese initialen Pattern sind sogenannte 1-Edge Graphen, sie bestehen jeweils aus genau einer Kante. Aus dem Eingabegraph werden alle 1-Edge Graphen generiert, die anschließend jeweils auf einen Isomorphismus und optional auf ein Overlapping geprüft werden. Falls ein Isomorphismus festgestellt wird, so wird dem Pattern eine neue Instanz des Patterns beigefügt. Dadurch speichert jedes Pattern nicht nur sich selbst, sondern auch alle weiteren Vorkommen. Am Ende der Initialisierung ist eine Liste von ersten interessanten Pattern vorhanden, die im weiteren Verlauf des SUBDUE Algorithmus als Basis für alle kommenden Iterationen der Minimal Expansion dienen.

Für den Isomorphismus Test in der Initialization, aber auch in anderen Phasen der Substructure Discovery, enthält der SUBDUE einen Graph Matcher, der dafür zuständig ist, zwei Eingabegraphen G und H auf ihre strukturelle Gleichheit zu prüfen. Es wurde bereits im Kapitel der Graph Mining Grundlagen erwähnt, dass

die Untersuchung auf strukturelle Gleichheit von zwei Graphen ein schwieriges graphentheoretisches Problem darstellt. Aus diesem Grund verwendet diese SUBDUE Implementierung einen approximativen Graphen Isomorphismus. Die verwendete Approximation ist polynomial beschränkt, sodass die Effizienz verbessert werden kann, während eine angemessene Genauigkeit beibehalten wird. Der Isomorphismus Algorithmus ist auf Basis der Kanten E im Graph G mit E^2 beschränkt. Das bedeutet nach maximal E^2 Schritten bricht der Algorithmus ab und gibt zurück, dass kein Isomorphismus gefunden wurde. Mit dem Aufruf des Graph Matchers wird der approximative Algorithmus für das Graph-Isomorphismus-Problem auf den Graphen G und H gestartet. Zu Beginn werden die Anzahl der Knoten und Kanten in beiden Graphen verglichen. Wenn diese nicht gleich sind, dann kann trivialerweise kein Isomorphismus gegeben sein, da die Graphen strukturell unterschiedlich sind. Falls im Graphen G nur Knoten und keine Kanten enthalten sind, wird geprüft, ob jeweils zwei Knoten identisch sind. Wenn die Attribute der beiden Graphen übereinstimmen und beide Knoten den gleichen Kantengrad haben, werden beide Knoten als identisch klassifiziert.

Phase 2: Minimal Expansion

Die Minimal Expansion ist der Beginn jeder Substructure Discovery Verfahrens. Die Minimal Expansion erwartet als Eingabe eine Liste von Pattern, die im weiteren Verlauf als Parent Patterns genutzt werden, um jeweils jedes dieser Pattern um genau eine weitere Kante zu erweitern. Somit verfolgt der SUBDUE demselben Pattern Growth Ansatz, wie andere Frequent Subgraph Mining Algorithmen. Die zu prüfenden Pattern werden dadurch immer größer. Nachdem das jeweilige Parent Pattern erweitert wurde, wird dieses ebenfalls wieder auf einen Isomorphismus, auf ein Overlapping geprüft und entsprechend in die Extended Pattern Liste aufgenommen. In der Extended Pattern Liste befinden sich somit alle erweiternden Pattern auf Basis eines Parent Patterns.

Phase 3: Evaluate Compression

Für jedes generierte Pattern aus der Minimal Expansion, welches den Anforderungen für das zu suchende Pattern entspricht, wird ein Kompressionswert berechnet. Hierfür stehen unterschiedliche Metriken zur Verfügung, die im Abschnitt 3.5 beschrieben sind. Der Kompressionswert gibt an, mit welcher Qualität das Pattern den Eingabegraphen komprimiert. Je höher dieser Wert ausfällt, desto interessanter scheint das Pattern nach dem Minimum Description Length Ansatz 2.3.4 zu sein.

Phase 4: Beam Search Queue

In der Beam Search Queue werden alle Pattern der Beam Search gehalten, die für den weiteren Verlauf der Substructure Discovery benötigt werden. Es werden jeweils die bisher besten Pattern mit den höchsten Kompressionswerten gespeichert, aber auch weitere interessante Pattern, die als Kandidaten in Betracht gezogen werden.

Phase 5: Compress

Die Compress Phase ist nicht mehr der Substructure Discovery zugeordnet. Das beste Pattern der aktuellen Iteration wurde bereits in der Substructure Discovery identifiziert und befindet sich in der Beam Search Queue. Dieses Pattern wird nun verwendet, um den Eingabegraphen zu komprimieren. Hierfür werden alle Vorkommen des Patterns durch einen neuen Knoten ersetzt. Alle mit dem Pattern verbundenen Kanten werden an den neuen Knoten geführt, sodass die Beziehungen über das Pattern hinaus beibehalten werden. Der daraus resultierende Graph kann wiederum als neuer Eingabegraph für die nächste Iteration des SUBDUE Algorithmus genutzt werden. Somit ist es möglich, eine Hierarchie von in sich verschachtelte Pattern zu identifizieren, die wiederum höhere Kompressionswerte aufweisen können. In dieser Thesis wird dieses Verfahren jedoch nicht weiter betrachtet, da der Schwerpunkt auf der Identifizierung einfacher Edit Operationen liegt.

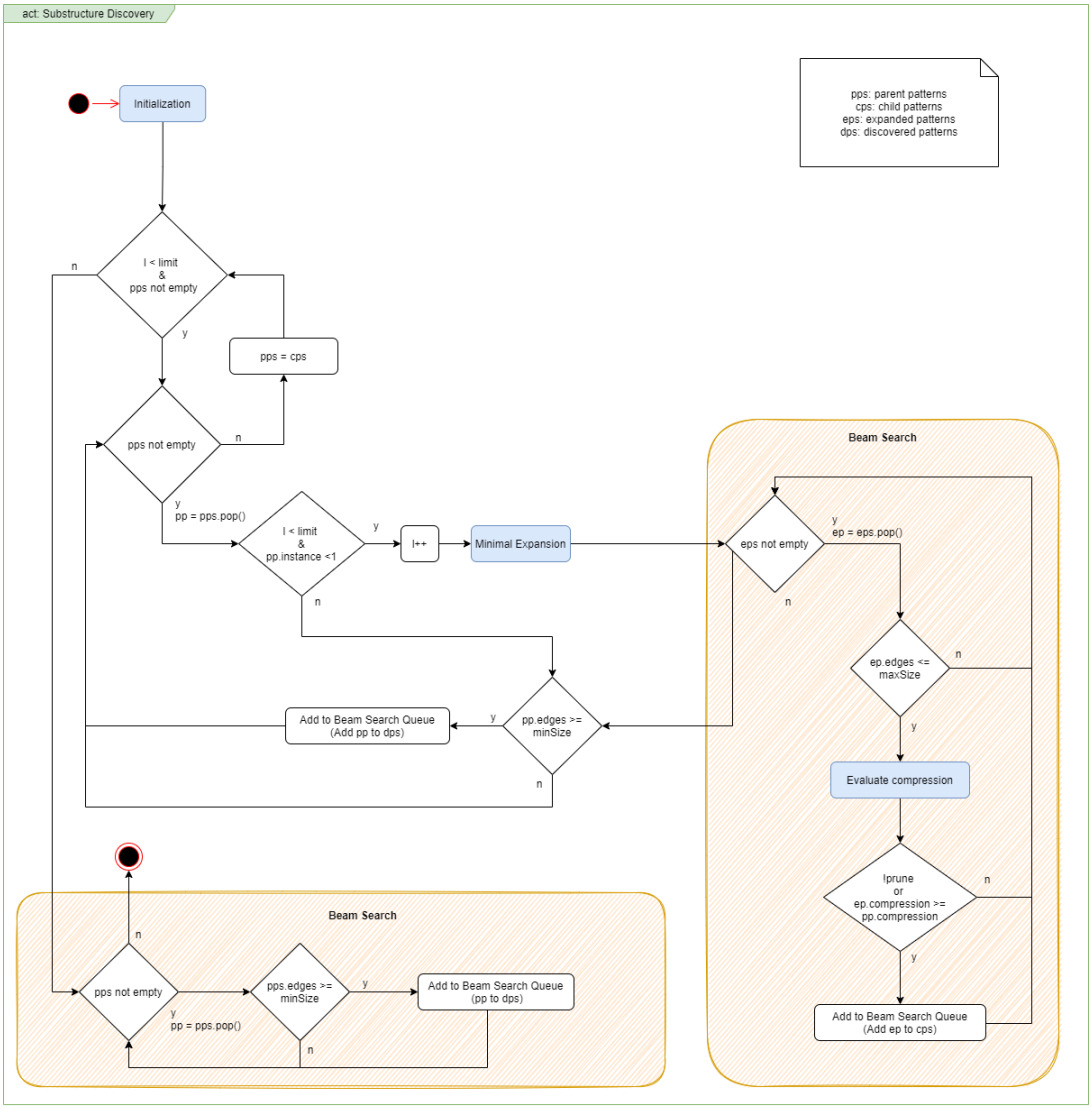

Abbildung 3.7: Aktivitätsdiagramm der SUBDUE Substructure Discovery.

3.5 Implementierung

Zur Verfügung steht jeweils eine SUBDUE Implementierung in der Programmiersprache Python und in C. Beide Implementierungen stammen aus dem SUBDUE Forscherteam, das von L. Holder und D. Cook angeführt wird. Die Implementierungen unterscheiden sich jedoch nicht nur in der Programmiersprache, in denen sie implementiert wurden, sondern auch in anderen technischen Details. Diese Unterschiede haben Auswirkungen auf die Performance des Algorithmus. Dadurch variiert nicht nur die Laufzeit, sondern es treten auch unterschiedliche Ergebnisse des identifizierten besten Patterns auf. Aus diesem Grund wird im Anschluss dieses Kapitels eine Pilotstudie folgen, um die Performance beider Implementierungen auf Model Repositories zu messen.

Metriken

Die beiden Implementierungen in Python und C unterscheiden sich nicht nur in den Programmiersprachen, sondern weisen auch unterschiedliche Ansätze im Verfahren des SUBDUE Algorithmus auf. In der Veröffentlichung des SUBDUE und in der ursprünglichen C Implementierung wird die Kompression mithilfe der Minimum Description Length vorgestellt. Die Beschreibungslängen DL des Patterns P und den damit zu komprimierenden Eingabegraphens G werden für die Berechnung Bits dargestellt. Die Beschreibungslänge des mit P komprimierten Eingabegraphens G wird mit dem Ausdruck $DL(G|P)$ abgebildet.

$$compression\_mdl(P, G) = \frac{DL(G)}{DL(P) + DL(G|P)}$$

Graph $G|P$: Der mit dem Pattern P komprimierte Graph G

Darüber hinaus bietet die C SUBDUE eine weitere Möglichkeit einer Kompressionsmetrik, die auf der vorkommenden Anzahl von Knoten und Kanten im Eingabegraphen G und dem Pattern P beruht, anstelle der Beschreibungslängen durch Bits. Die Kompression anhand der Anzahl an Knoten und Kanten ist ein weniger genaues Maß für die Kompression des Graphens G, die durch das Pattern P herbeigeführt wird. Allerdings hat sich herausgestellt, dass das Maß an Ungenauigkeit dieser Metrik zu vernachlässigen ist und die berechneten Kompressionswerte im Regelfall konsistent mit den Kompressionswerten der Minimum Description Length sind. Diese Konsistenz der Ergebnisse konnte in einem Experiment nachgemessen werden, welches in der Veröffentlichtung *Graph-based Data Mining in Dynamic Networks: Empirical Comparison of Compression-based and Frequency-based Subgraph Mining* [35] durchgeführt wurde.

$$compression\_size(P, G) = \frac{size(G)}{size(P) + size(G|P)}$$

$$size(Graph) = \#Graph.vertices + \#Graph.edges$$

Die Python SUBDUE Implementierung verwendet für die Berechnung der Kompression eine andere Metrik, welche die relative Kompression ausdrückt. Auf welcher Grundlage diese Heuristik basiert, konnte nicht nachvollzogen werden.

$$compression\_heuristic(P, G) = \frac{\#P.instance * \#P.edges}{\#G.edges}$$

Durch die unterschiedlichen Metriken, die für die Berechnung der Kompression zur Verfügung stehen, wird eine unterschiedliche Performanceleistung je nach Auswahl der Metrik erwartet. Die Auswirkungen der Metrikauswahl auf die Performance wird ebenfalls in der Pilotstudie aufgegriffen. Die Ergebnisse dieser Pilotstudie dienen ebenfalls als Ausgangslage für die Kompressionsmetrik, die für den THEOBALD-SUBDUE verwendet wird.

3.6 Nachteile in Model Repositories

Wie bereits von C. Tinnes in der Arbeit *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] erwähnt, sind die Ergebnisse und die Performanceleistung des SUBDUE Algorithmus im Vergleich zu anderen Subgraph Mining Algorithmen auf Model Repositories noch wesentlich schwächer. Ein wesentlicher Unterschied zu anderen Anwendungsgebieten ist, dass ein Model Repository eine Abbildung einer Instanz eines Meta Models enthält. Diese Instanz wurde durch eine Folge von Modelltransformationen generiert und verändert. Eine Modelltransformation betrifft typischerweise nicht das gesamte Model Repository, sondern nur eine Teilmenge der enthaltenen Elemente. Es existieren also Elemente, die nicht direkt von Modelltransformation betroffen sind. Diese Preserve Elemente haben sich selbst nicht verändert, sie können lediglich durch Edit Operationen neue Beziehungen zu anderen Elementen erhalten. Dabei können auch mehrere Edit Operationen Auswirkungen auf ein und dasselbe Preserve Element haben. Ein Preserve Element kann daher Bestandteil von mehreren Edit Operationen sein. Hingegen sind alle nicht Preserve Elemente immer genau einer Edit Operation zuzuordnen. Die aktuelle SUBDUE Implementierung berücksichtigt mit dem Overlap Parameter alle enthaltenen Elemente des Graphens. Für den Anwendungsfall auf Model Repositories ist allerdings eine Überlappung von nicht Preserve Elementen nicht möglich, weshalb die Annahme getroffen wird, dass diese Einschränkung zu einer besseren Performance führt.

Diese Eigenschaft der Model Repositories hat ebenfalls Auswirkungen auf die Berechnung der Kompressionsmetrik. Da die Preserve Elemente nicht direkt von der Modelltransformation betroffen sind und sich durch die Preserve Elemente der Eingabegraph nicht verändert, sind diese Elemente nicht relevant für die Kompression. Der Kompressionswert drückt aus, wie stark ein Pattern den Eingabegraph komprimiert. Wenn die Preserve Elemente bei der Berechnung nicht berücksichtigt werden, werden die Edit Operationen mit allen durch die Modelltransformationen veränderten Elementen stärker gewichtet. Durch das Herausrechnen der Preserve Elemente für die Berechnung der Metrik könnten bessere Ergebnisse erzielt werden.

Der SUBDUE Algorithmus bietet die Möglichkeit, mehrere interessante Pattern in absteigender Reihenfolge nach den Kompressionswerten zu identifizieren. Ein weiterer Nachteil ist hierbei, dass die so identifizierten Pattern sehr ähnlich sind. Sollen nun mehrere interessante Pattern identifiziert werden, so werden insbesondere in Model Repositories oftmals nur ähnliche Edit Operationen erkannt. Die Pattern unterscheiden sich entweder geringfügig oder sind Subgraphen des besten Patterns. Mit dem Ziel vor Augen, einem Programmierer auf Basis eines Model Repositories die interessantesten Edit Operationen an die Hand zu geben, sollte das Identifizieren möglichst unterschiedliche Pattern ausgeben. Erst wenn dem Programmierer eine breite Palette an interessanten Edit Operationen vorliegt, kann er die für sich richtige Edit Operation auswählen, um somit das Reuse Prinzip für Model Repositories zu optimieren. Dabei sollten die Edit Operationen möglichst divers sein. Es lassen sich zwar mehrere interessante Pattern mit dem SUBDUE zwischenspeichern, jedoch kann momentan nur das beste Pattern davon ausgegeben werden. Diese Ergebnisse lassen sich auch in der Veröffentlichung *A Survey of Frequent Subgraph Mining Algorithms* [36] entnehmen. Für einen besseren Reuse Ansatz in Model Repositories sollte es daher möglich sein, mehrere unterschiedliche Edit Operationen zu identifizieren und auszugeben, die sich mit möglichst großer Distanz unterscheiden.

Kapitel 4

Experiment 1 - Subdue Pilotstudie

Um eine erste Übersicht über die Performance der verfügbaren SUBDUE Implementierungen zu gewinnen, wird im Folgenden eine erste Pilotstudie durchgeführt. Ziel der Pilotstudie ist es, den aktuellen Ist-Zustand des SUBDUE Algorithmus für das Subgraph Mining von Edit Operationen in Model Repositories festzustellen. Da sich die zur Verfügung stehenden Implementierungen des SUBDUE Algorithmus nicht nur in der Programmiersprache, sondern auch in Implementierungsdetails unterscheiden, wird die Performance beider Implementierungen in einem ersten Experiment gemessen.

4.1 Fragestellung

E1Q1 Welche Ergebnisse erzielen unterschiedlichen Parametereinstellungen und Metriken zur Berechnung der Kompression zur Identifizierung von interessanten Pattern auf Model Repositories?

E1Q2 Welche Implementierung erzielt die besten Ergebnisse auf Model Repositories?

4.2 Aufbau

Für die Pilotstudie werden 100 zufällig ausgewählte Model Repositories aus einem zuvor generierten Datensatz ausgewählt. Dieser Datensatz besteht aus 2000 simulierten Model Repositories, die auf Basis des Meta Models in Abbildung 4.1 erzeugt wurden. Jedes Model Repository besteht aus einer zufälligen Instanz des Meta Models und einer Änderungshistorie dieses Komponentenmodells. Um diese Änderungshistorie zu erzeugen, wurde eine Edit Operation mit einer bestimmten Wahrscheinlichkeit iterativ auf die Komponentenmodelle angewandt. Zudem wurden noch weitere Edit Operationen ebenfalls mit einer gewissen Wahrscheinlichkeit auf den Komponentenmodellen ausgeführt, um den Störfaktor für das Subgraph Mining zu erzeugen. Die simulierten Model Repositories in dem Datensatz unterscheiden sich in genau den beiden erwähnten Punkten, in der Anzahl an durchgeführten Edit Operationen und der Wahrscheinlichkeit des Auftretens einer weiteren Edit Operationen als Störung, die ungleich der tatsächlich interessanten Edit Operation ist, die zuvor durchgeführt wurde. Dadurch entstehen mehrere Modelltransformationen, die zu unterschiedlichen Versionen des Komponentenmodells innerhalb eines

Model Repositories führen. Die damit generierte Änderungshistorie simuliert somit ein Model Repository, auf dem mehrere Modelltransformationen ausgeführt wurden.

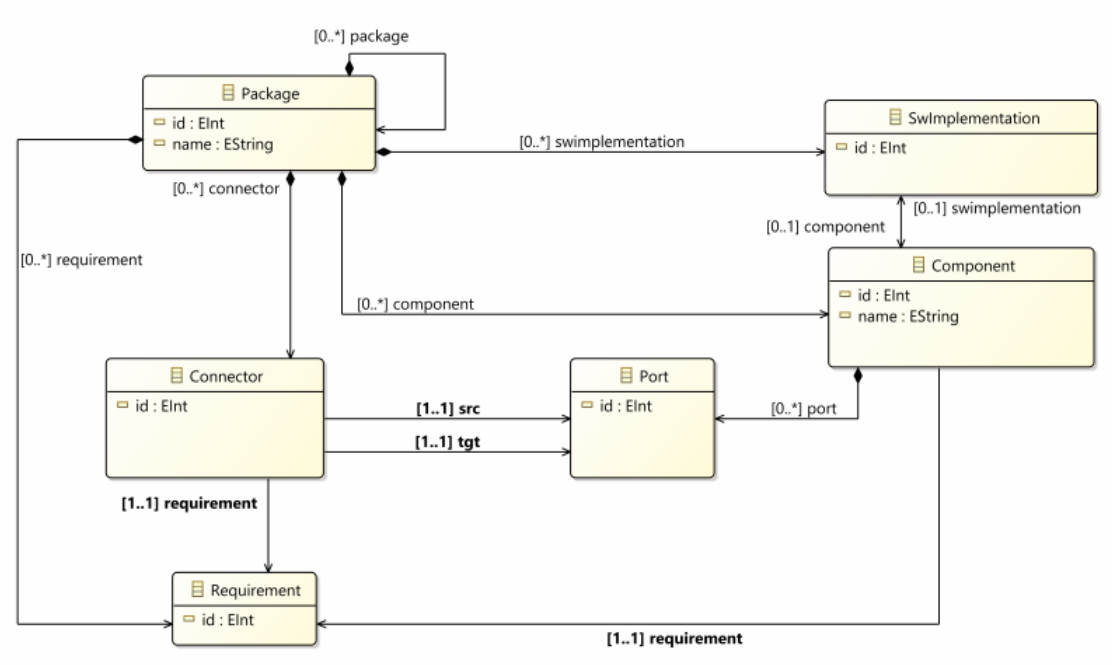

Abbildung 4.1: Meta Model der simulierten Model Repositories.

Die Edit Operation aus der Abbildung 4.2 besteht aus 7 Knoten und 7 Kanten, die jeweils mit einem Label versehen sind. Alle Knoten und Kanten des gesamten Model Repositories, insbesondere die Edit Operationen sind mit einer Henshin ähnlichen Sprache per Prefix markiert. Die Modellierungssprache Henshin aus dem Abschnitt 2.2.3 definiert mehrere Aktionen einer Modelltransformation. In den simulierten Model Repositories werden die Elemente einer Revision in zwei Zuständen eingeteilt. Ein Element kann entweder durch eine Edit Operation neu in die aktuelle Version des Model Repositories hinzugefügt worden sein oder das Element hat sich zur vorherigen Version nicht geändert. Im ersten Fall wird das Element mit dem Prefix *Add* markiert, im zweiten Fall erhält das Element den Prefix *Preserve*. Am Beispiel der Edit Operation in der Abbildung 4.2 lässt sich erkennen, dass zwischen den beiden *Preserve_Component*s und dem *Preserve_Package* weitere Elemente hinzugefügt wurden.

Das Ziel des Experiments ist es, mit den beiden SUBDUE Implementierungen die angewandte Edit Operation auf den jeweils 100 Model Repositories als interessantestes Pattern zu identifizieren. Dabei sollen alle anderen möglichen Pattern vernachlässigt und als schlechtere Pattern klassifiziert werden, sodass genau die Edit Operation aus der Abbildung 4.2 aus dem simulierten Model Repository gelernt wird. Ein simuliertes Model Repository ist dabei nichts anderes als ein Graph, der aus jeweils den Graphen der einzelnen Versionen des Model Repositories besteht. In diesem Graphen befinden sich die gesamte Änderungshistorie, insbesondere alle ausgeführten Edit Operationen. Dieser Graph ist in für die aktuell simulierten Model Repositories gelabelt, ausschließlich ungerichtet und ist kein Multi Graph. Der SUBDUE Algorith-

mus wird auf diesem Graphen ausgeführt und soll die Edit Operation in Abbildung 4.2 erkennen.

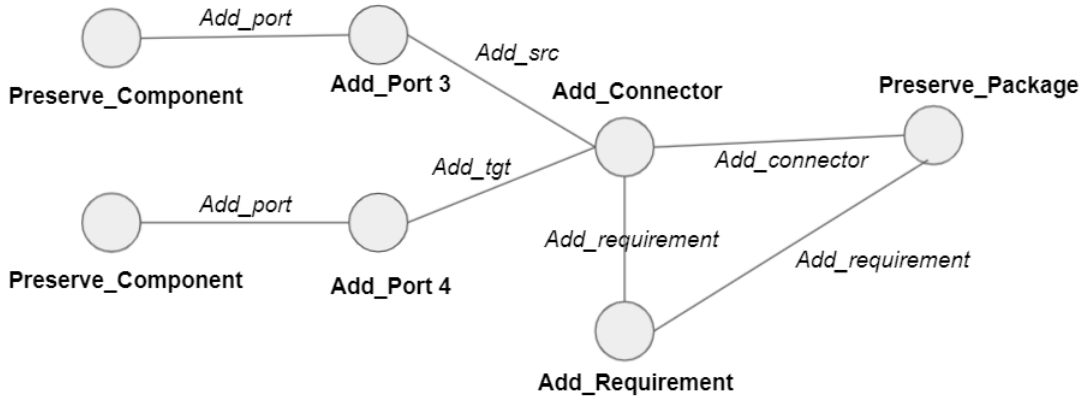

Abbildung 4.2: Korrekte Edit Operation des ersten Datensatzes.

4.3 Durchführung

Die Pilotstudie ist in mehrere Versuchsläufe unterteilt, die sich in unterschiedlichen Parametereinstellungen für die Ausführung des SUBDUE Algorithmus unterscheiden. Die Parametereinstellungen für beide SUBDUE Implementierungen pro Versuchslauf sind dieselben. Jeder Versuch findet auf demselben Datensatz statt und wird auf derselben Hardware mit den Spezifikationen in der Abbildung 4.1 durchgeführt.

Während der Ausführung der SUBDUE Implementierungen auf den Model Repositories werden sowohl die Laufzeit als auch die Heapsize gemessen. Zu diesem Zweck wird die Startzeit zu Beginn jeder Ausführung und die Endzeit am Ende des Algorithmus gemessen. Aus beiden Werten wird die Laufzeit ermittelt. Der wichtigste Messwert für die Performance ist der Matchwert. Mit der Durchführung des SUBDUE Algorithmus wird das für den Algorithmus interessanteste Pattern aus dem Model Repository ausgegeben. Da die simulierten Model Repositories konstruiert sind ist bekannt, welche Edit Operation für die Erstellung der Model Repositories am häufigsten ausgeführt wurde und somit das interessanteste Pattern darstellt. Diese Edit Operation wird als Benchmark genutzt. Es lässt sich nun einfach prüfen, ob die vom SUBDUE identifizierte Edit Operation mit dem Benchmark übereinstimmt. Falls ja, wird für den Versuchsdurchlauf ein korrekter Match identifiziert. Die Anzahl an korrekten Matches ist somit der ausschlaggebende Messwert, wie gut die SUBDUE Implementierung auf dem Datensatz mit den gegebenen Parametereinstellungen performt.

Versuch 1 und 2

Für die beiden Versuche 4.2 und 4.3 wird der Parameter für das *Limit* und der $MaxSize$ dem zufällig ausgewählten Wert 50 zugewiesen. Mit diesem Wert gibt es

OS	Windows 10
RAM	16GB
Processor	Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz 2.40GHz
Hard Drive	Samsung V-Nano SSD 860 EVO

Tabelle 4.1: Die für alle Experimente verwendete Hardware.

Iterations	1	**BeamWidth**	4
Limit	50	**Overlap**	False
MaxSize	50	**Prune**	False
MinSize	1	**ValueBased**	False

Implementation	**Metric**	**Runtime** (s)	**Heap** (byte)	**Match** (%)
Subdue C	MDL	0,49862	136,76	43
Subdue C	Size	0,47111	135,62	33
Subdue Python 1.4	Heuristic	3,5969	143,23	28
Subdue Python 1.4	Size	3,3870	134,39	57

Tabelle 4.2: Experiment 1, Versuch 1: Prüfung des Overlap Parameters.

eine obere Schranke für die Anzahl an Pattern, die in der Substructure Discovery berücksichtigt werden. Je nach Komplexität des Eingabegraphens werden für dieser Einstellungen wichtige Pattern nicht weiter verfolgt, was zu einem schlechteren Ergebnis führen kann. Ziel dieser beiden Versuche ist es, den *Overlap* Parameter zu testen und die Performanceunterschiede für diesen Parameter zu messen. Für den zweiten Versuch wird bei sonst gleichbleibenden Parametereinstellungen der *Overlap* Parameter von $False$ auf $True$ gesetzt.

Versuch 3

Da die Versuchsreihe auf simulierten Model Repositories ausgeführt wird, ist das korrekt zu identifizierende Pattern bereits bekannt. In diesem Versuchsdurchlauf 4.4 wird daher versucht, das korrekte Pattern mit gezielten Anpassungen der Parametereinstellungen zu identifizieren. Die $MaxSize$ und die $MinSize$ werden entsprechend dem zu suchenden Pattern angepasst, sowie das *Limit* weiter reduziert.

Versuch 4 und 5

In den beiden Versuchen 4.5 und 4.6 wird nochmals die Performance des *Overlap* Parameters getestet. Die beiden Parameter *Limit* und $MaxSize$ werden nun nicht mehr statisch gesetzt, sondern werden anhand der Größe des Eingabegraphs $|E|/2$ berechnet. Dadurch wird die obere Schranke für die Anzahl der berücksichtigten Pattern erhöht, was tendenziell zu besseren Ergebnissen führen kann.

Versuch 6

Im letzten Versuch 4.3 dieses Experiments wird die Auswirkung des $BeamWith$ Parameters gemessen. Der SUBDUE Algorithmus wird jeweils mit unterschiedlicher $BeamWidth$ zwischen $1 - 6$ ausgeführt.

Iterations	1	**BeamWidth**	4
Limit	50	**Overlap**	True (Vertex)
MaxSize	50	**Prune**	False
MinSize	1	**ValueBased**	False

Implementation	**Metric**	**Runtime** (s)	**Heap** (byte)	**Match** (%)
Subdue C	MDL	0,7352	135,76	92
Subdue C	Size	0,5996	137,13	92
Subdue Python 1.4	Heuristic	5,2636	140,19	92
Subdue Python 1.4	Size	6,1459	139,56	90

Tabelle 4.3: Experiment 1, Versuch 2: Prüfung des Overlap Parameters.

Iterations	1	**BeamWidth**	4
Limit	30	**Overlap**	True (Vertex)
MaxSize	7	**Prune**	False
MinSize	2	**ValueBased**	False

Implementation	**Metric**	**Runtime** (s)	**Heap** (byte)	**Match** (%)
Subdue C	MDL	0,3621	134,88	77
Subdue C	Size	0,2986	137,13	76
Subdue Python 1.4	Heuristic	3,0306	139,74	58
Subdue Python 1.4	Size	0,1585	138,28	92

Tabelle 4.4: Experiment 1, Versuch 3: Parameter angepasst auf korrektes Pattern.

Iterations	1	**BeamWidth**	4
Limit	$\|E\|/2$	**Overlap**	False
MaxSize	$\|E\|/2$	**Prune**	False
MinSize	2	**ValueBased**	False

Implementation	**Metric**	**Runtime** (s)	**Heap** (byte)	**Match** (%)
Subdue C	MDL	0,8341	135,38	42
Subdue C	Size	0,8079	136,27	33
Subdue Python 1.4	Heuristic	25,4635	145,57	29
Subdue Python 1.4	Size	18,2103	145,57	32

Tabelle 4.5: Experiment 1, Versuch 4: Prüfung des Overlap Parameters.

Iterations	1	**BeamWidth**	4
Limit	$\|E\|/2$	**Overlap**	True (Vertex)
MaxSize	$\|E\|/2$	**Prune**	False
MinSize	2	**ValueBased**	False

Implementation	**Metric**	**Runtime** (s)	**Heap** (byte)	**Match** (%)
Subdue C	MDL	26,5711	130,17	90
Subdue C	Size	21,7632	131,34	87
Subdue Python 1.4	Heuristic	24,0363	144,11	92
Subdue Python 1.4	Size	33,6395	137,92	89

Tabelle 4.6: Experiment 1, Versuch 5: Prüfung des Overlap Parameters.

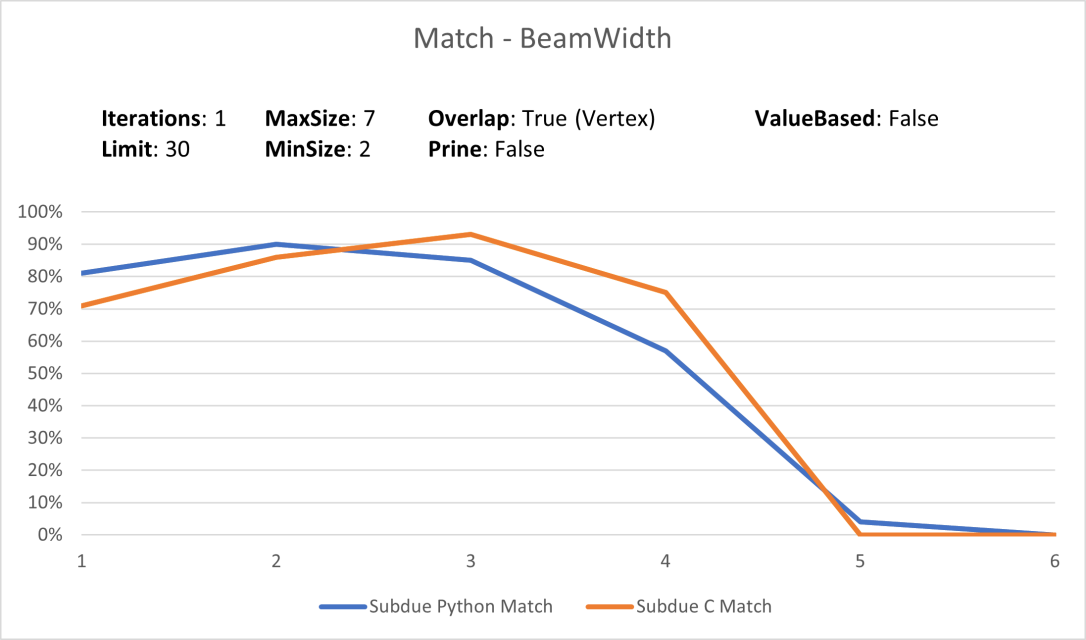

Abbildung 4.3: Experiment 1, Versuch 6: Prüfung des BeamWith Parameters.

4.4 Evaluation

Die Ergebnisse aus der Versuchsdurchführung für die Beantwortung von **E1Q1** zeigen, dass der SUBDUE Algorithmus signifikant bessere Ergebnisse erzielt, wenn der *Overlap* Parameter aktiviert ist. Dies ist auch nicht weiter verwunderlich, da sich die Instanzen der identifizierten Pattern in Model Repositories stark überlappen können. Anders als in anderen Anwendungsbereichen, in denen sich die gefundenen Pattern nicht überlagern, so können die gefundenen Pattern in einem Model Repository zu unterschiedlichen Edit Operationen gehören. Es ist auch nicht weiter verwunderlich, dass die Laufzeit in den Versuchen fünf und sechs deutlich angestiegen ist. Durch die erhöhte obere Schranke $|E|/2$ werden deutlich mehr Pattern in der Substructure Discovery berücksichtigt, was zu der längeren Laufzeit führt. An den Messungen ist auch zu erkennen, dass der SUBDUE für unterschiedliche Metriken in den jeweiligen Implementierungen C und Python zu verschiedenen Ergebnissen führt. Um **E1Q2** zu beantworten, werden die Matching Ergebnisse aller Versuche berücksichtigt. Im Schnitt erzielt die C Implementierung 1,87% bessere Ergebnisse. Aus den Messungen fällt allerdings ein unvorhergesehenes Verhalten auf. Die Matchingrate reduziert sich auf nahezu 0% herab, wenn die $BeamWidth$ über 4 gesetzt wird.

Aus den Ergebnissen ergeben sich folgende neue Fragestellungen, die im weiteren Verlauf der vorliegenden Arbeit zu einer näheren Untersuchung des SUBDUE Algorithmus führen:

E1Q3 Warum gibt es Abweichungen in den Matches je nach Implementierung trotz gleicher Parameterauswahl?

E1Q4 Warum werden die Anzahl an Matches mit einem höheren BeamWidth Parameter schlechter?

Die Frage **E1Q3** ergibt sich aus der unerwarteten Messung, dass die Matches trotz gleicher Metrik in den beiden unterschiedlichen Implementierungen teilweise stark voneinander abweichen können. Auch die unterschiedlichen Metriken innerhalb derselben Implementierung weisen teilweise eine stärkere Abweichung auf, als vermutet.

Auch die Frage **E1Q4** stellt sich aus der unerwarteten Beobachtung, dass die Trefferquote des korrekt zu identifizierten Patterns mit einer größeren BeamWidth tendenziell abnimmt. Die BeamWidth gibt die Suchbreite der Suche nach interessanten Pattern an. Je größer die BeamWidth, desto mehr Pattern werden im gesamten Suchprozess berücksichtigt. Eine größere BeamWidth hat also eine Auswirkung auf die Anzahl an Pattern, die in jede Iteration der Minimal Expansion übergeht. An dieser Stelle lohnt sich nochmals ein Blick auf die Abbildung 3.6, in der die Phasen des SUBDUE Algorithmus dargestellt sind. Ebenfalls kann sich mit der Abbildung 3.3 nochmals ins Gedächtnis gerufen werden, wie sich eine Beam Search beispielhaft vorzustellen ist. Zusammengefasst bedeutet das, dass umso mehr Pattern in der Beam Search Queue gehalten werden, je größer die BeamWidth ist. Daraus erschließt sich die Erwartung, dass mit einer größeren BeamWidth potenziell mehr Matches resultieren, da die Suchbreite weniger eingeschränkt ist und daher mehr Kombinationen aus der Minimal Expansion in der Beam Search Queue gehalten werden. Dieser Erwartung trifft jedoch auf dem verwendeten Datensatz für jeweils

beide SUBDUE Implementierungen nicht zu. Die beiden Fragen **E1Q3** und **E1Q4** sind aus Zeitgründen nicht mehr Teil dieser Arbeit, bietet jedoch Potenzial für weitere Untersuchungen, die im Kapitel Ausblick 8 erläutert werden.

Kapitel 5

Experiment 2 - Subdue Beam Search Debugging

Motiviert durch die Ergebnisse des ersten Experiments wird in diesem weiteren Experiment die Python SUBDUE Implementierung detaillierter untersucht. Die Ergebnisse deuten darauf hin, dass zum einen die Python Implementierung eine schlechtere Erkennung der interessantesten Pattern erzielt, im Vergleich zur C Implementierung. Zum anderen sind die Ergebnisse beider Implementierungen, sowohl in C als auch in Python aktuell schlechter als die Ergebnisse, die C. Tinnes mit dem Frequent Subgraph Mining Algorithmus GASTON erzielt. Aus diesem Grund wird für die Beam Search in der Substructure Discovery des Python SUBDUE ein Debugger implementiert, sodass sich die Zustände der Beam Search Queue zu jedem Zeitpunkt einfach einsehen lassen.

5.1 Fragestellung

E2Q1 Wie verhält sich die Beam Search Queue auf einem Datensatz, auf dem das zu suchende Pattern nicht gefunden wird?

E2Q2 An welcher Stelle befinden sich nur noch schlechte Pattern in der Beam Search Queue, die durch weitere Minimal Expansion Iterationen nicht mehr zu dem gesuchten Pattern führen können und unter welchen Umständen tritt dieses Problem auf?

5.2 Aufbau

Der Beam Search Debugger besteht aus 7 unterschiedlichen Anzeigen, die jeweils die Inhalte der aktuell lokalen Queue und dessen interessante Inhalte aufzeigen. In Abbildung 5.2 ist das Aktivitätsdiagramm der Substructure Discovery dargestellt. Ebenfalls markiert sind die entsprechenden Positionen, an denen der Debugger die Beam Search Inhalte ausgibt. Des Weiteren ist in der Abbildung 5.1 die Ausgabe des Debuggers innerhalb eines Durchlaufs der Substructure Discovery beispielhaft abgebildet. Jeder Durchlauf r der Substructure Discovery beginnt mit einer Liste von Parent Patterns. Es werden alle Pattern aus dieser Liste nacheinander in der Iteration l ausgewählt. Dieses ausgewählte Parent Pattern wird im weiteren Ablauf entsprechend der Substructure Discovery verarbeitet. Alle Pattern werden als Graphen

ausgegeben und enthalten jeweils den Kompressionswert auf dem Eingabegraph, sowie die Anzahl aller Vorkommen an Instanzen des Patterns im Eingabegraph. Durch die jeweils kleinschrittigen Ausgaben des Debuggers lässt sich die Funktionalität der Substructure Discovery und somit auch der Beam Search im Detail nachvollziehen. Zudem kann somit herausgefunden werden, zu welchen Zeitpunkten der SUBDUE Algorithmus schlechte Entscheidungen für die Identifizierung des korrekten Patterns getroffen hat.

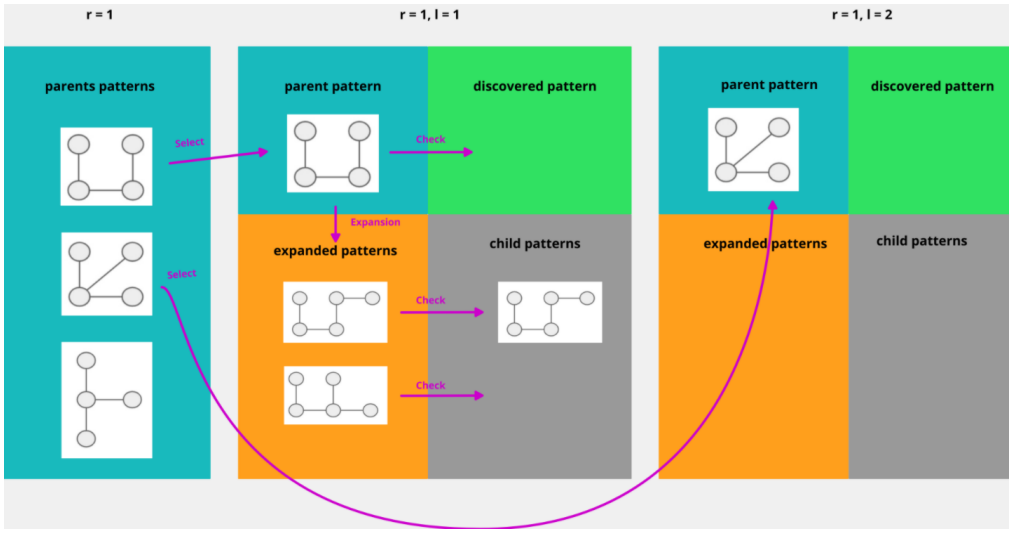

Abbildung 5.1: Beispielhafte Ausgabe des Beam Search Debuggers.

Im Folgenden werden die einzelenen Ausgaben des Debuggers und die darin enthaltenen Pattern der Beam Search Queue erläutert.

Debugger: Init

Nach der Initialisierung des SUBDUE werden alle gefundenen 1-Edge Pattern über den Debugger aufgelistet. Die gefundenen 1-Edge Pattern werden in die Parent Pattern Liste geschrieben. Somit kann gezeigt werden, auf welcher Basis der weitere Verlauf der Substructure Discovery fortgeführt wird.

Debugger: Parents

Jeder Durchlauf der Substructure Discovery beginnt mit einer Liste an Parent Patterns und jeder dieser Durchläufe iteriert über die gesamte Parent Pattern Liste. Dadurch wird jedes Parent Pattern begutachtet und durch die Minimal Expansion erweitert. Die Parent Patterns zeigen also den aktuellen Basiszustand der Beam Search Queue an. Alle nachfolgenden Pattern der Beam Search Queue sind Subgraphen dieser Parent Patterns.

Debugger: Parent

Das Parent Pattern zeigt an, welches Pattern in der aktuellen Iteration ausgewählt wurde. Dieses Pattern ist Ausgangslage für die Minimal Expansion. Zudem wird

das Parent Pattern in die Discovered Pattern aufgenommen, falls die Discovered Pattern Liste noch nicht komplett gefüllt ist oder das Parent Pattern einen höheren Kompressionswert besitzt, als die Pattern in der Liste.

Debugger: Expansion

Die Expanded Pattern Liste enthält alle Pattern, die auf Basis des aktuell ausgewählten Parent Patterns durch die Minimal Expansion erweitert wurden. Zu diesem Zeitpunkt befinden sich alle erweiterten Pattern in der Liste. Der Kompressionswert wurde noch nicht berechnet und somit wurde noch keine Expanded Pattern mit einem schlechteren Kompressionswert als sein Parent Pattern aussortiert.

Debugger: Evaluation

Zu diesem Zeitpunkt wurden die Kompressionswerte für alle Expanded Patterns berechnet. Expanded Patterns mit einem schlechteren Kompressionswert als deren Parent Pattern wurden aussortiert. Ein Expanded Pattern in dieser Liste ist somit ein potenziell neues Child Pattern, vorausgesetzt, die Liste der Child Patterns ist noch nicht komplett gefüllt oder der Kompressionswert ist größer als die bisherigen Child Patterns. Ist letzteres der Fall, wird das Child Pattern mit einem schlechteren Child Pattern aus der Liste getauscht.

Debugger: Childs

Alle Child Patterns werden zum Ende der Iteration, wenn alle Parent Patterns durchlaufen sind, zu den neuen Parent Patterns, die erneut durch die Minimal Expansion Iteration laufen werden. Die Child Patterns stellen also die Basis für die kommende Iteration bereit. Mit der BeamWidth Parametereinstellung kann die Länge dieser Liste konfiguriert werden. Je größer die BeamWidth, desto mehr Pattern werden in den Beam Search Iterationen berücksichtigt.

Debugger: Discovered

Die Discovered Patterns werden mit dem Abschluss des Subdue Algorithmus ausgegeben und sind somit die interessantesten Pattern, die auf dem Eingabegraph gefunden wurden.

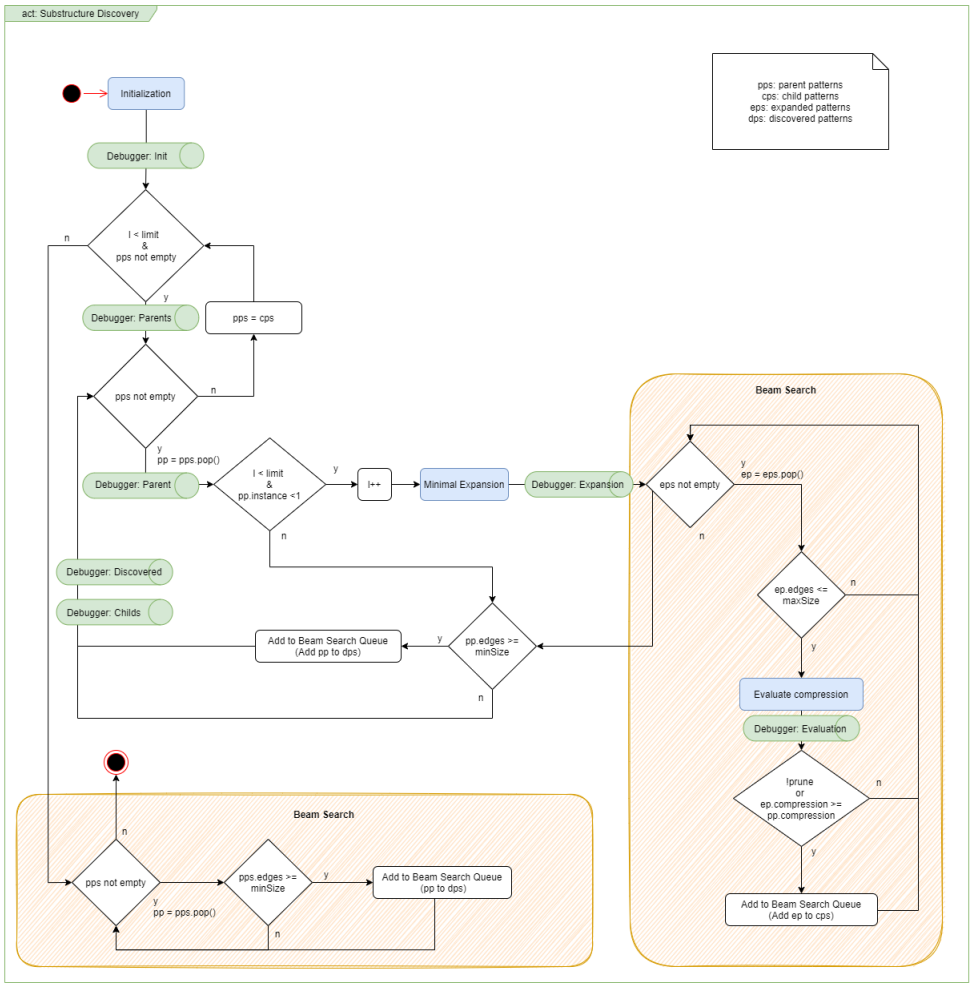

Abbildung 5.2: Subdue Debugger Implementierung.

5.3 Durchführung

Um die Fragestellungen beantworten zu können, wird der Debugger auf einem simulierten Model Repository durchgeführt, auf dem der SUBDUE das korrekte Pattern nicht identifizieren kann. Die Ergebnisse, ob der SUBDUE auf einem Model Repository das korrekte Pattern erkennen kann, sind aus dem ersten Experiment bekannt. Das simulierte Model Repository stammt daher auch aus demselben Datensatz, der im ersten Experiment verwendet wurde und ist dementsprechend gleich aufgebaut. Das korrekte Pattern, welches zu identifizieren ist, ist in der Abbildung 4.2 abgebildet. Für die Durchführung des Versuchs auf dem Model Repository wird ein Test Case angelegt, der den SUBDUE Algorithmus mit der Debugger Implementierung auf dem Model Repository ausführt. Der Debugger erstellt während der Ausführung des SUBDUE für jeden Zustand der Beam Search Queue eine Ausgabe der Pattern Listen mit den jeweils darin enthaltenden Pattern. Hierfür erstellt der Debugger einen neuen Ordner in dem Datensatz, in dem alle Ausgaben gespeichert werden.

Versuch 1

Mit diesem Versuch soll beantwortet werden, wie sich die Beam Search Queue auf Datensätzen verhält, auf denen das korrekte Pattern nicht gefunden wird. Das hierfür

Iterations	1	**BeamWidth**	10
Limit	$\|E\|/2$	**Overlap**	True (Vertex)
MaxSize	$\|E\|/2$	**Prune**	False
MinSize	4	**ValueBased**	False
Eval	Heuristic	**NumBest**	3

Tabelle 5.1: Experiment 2, Versuch 1: Debugger auf falsch identifiziertem Pattern.

ausgewählte Model Repository beinhaltet 10 Diffs, auf die jeweils 97 Edit Operationen angewandt wurden. Eine zusätzliche Störung wurde mit einer Wahrscheinlichkeit von 0,5 ausgeführt. Auf diesem simulierten Model Repository wird der SUBDUE mit den Parametereinstellungen in der Tabelle 5.1 ausgeführt. Es wurden weitere Versuche mit anderen Parametereinstellungen vorgenommen, jedoch konnte in keinem dieser Durchläufe das korrekte Pattern identifiziert werden.

5.4 Evaluation

Mithilfe des Debuggers kann die Beam Search Schritt für Schritt nachvollzogen werden, um die Funktionalität des SUBDUE Algorithmus zu überprüfen. Im Nachfolgenden wird die Ausgaben des Debuggers für den durchgeführten Versuch untersucht, sodass **E2Q1** und **E2Q2** beantwortet werden können. Im Anhang der vorliegenden Arbeit befinden sich detailliertere Abbildungen zu den Ausgaben des Debuggers auf dem Datensatz. Der SUBDUE Algorithmus identifiziert die Pattern in der Abbildung 5.3. Diese Pattern befinden sich zum Abschluss des SUBDUE Algorithmus in der Discovered Pattern Liste. Anhand dessen erschließt sich ein erster Hinweis, warum das korrekte Pattern aus Abbildung 4.2 nicht gefunden wurde. Alle drei identifizierten Pattern enthalten zwei Knoten mit dem Label *Preserve_Port*, welches sich nicht in dem korrekten Pattern befindet. Mit diesem Anhaltspunkt kann die Ausgabe des Debuggers durchsucht werden. Der nun gesuchte Zustand der Beam Search Queue ist jener, in dem sich in der Parent Pattern Liste nur noch Pattern mit mindestens einem *Preserve_Port* Knoten befindet. Denn zu diesem Zeitpunkt ist es dem SUBDUE nicht mehr möglich, das korrekte Pattern zu finden, da nun alle folgenden Minimal Expansions stets Pattern generieren, die einen *Preserve_Port* Knoten enthalten und diese können nicht mehr zu dem korrekten Pattern führen.

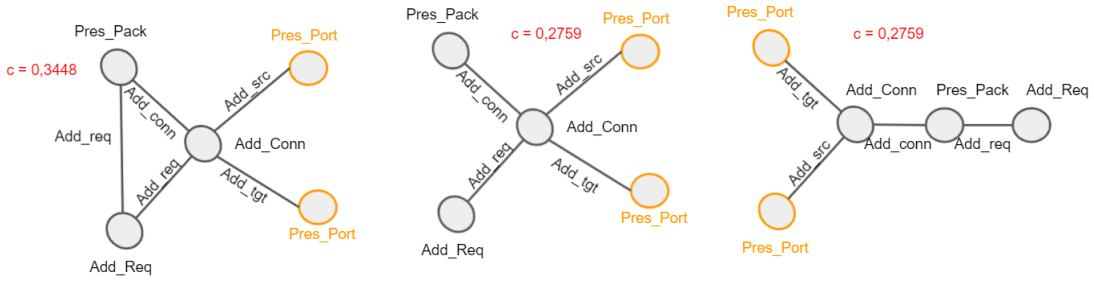

Abbildung 5.3: Identifizierte Pattern mit dem Kompressionswert c. Die orange markierten Knoten kommen nicht in dem korrekten Pattern vor.

Folgende Abbildung 5.4 zeigt den letzten Zeitpunkt der Beam Search Queue, an dem sich noch ein Subgraph des korrekten Pattern in der Queue befindet und somit noch potenziell das korrekte Pattern gefunden werden kann. In der Parent Pattern Liste der Iteration $r = 3$ befindet sich noch genau ein Pattern, welches durch weitere Expansion zum korrekten Pattern werden kann. Die restlichen Pattern in der Liste enthalten zu diesem Zeitpunkt schon jeweils mindestens einen *Preserve_Port* Knoten. Für $l = 19$ wird das potenziell interessante Pattern ausgewählt und anschließend durch die Minimal Expansion um eine Kante erweitert. An dieser Stelle bricht der Subdue, denn durch die Expansion entstehen zwei Pattern, die jeweils einen *Preserve_Port* Knoten enthalten und somit nicht mehr zum korrekten Pattern führen können. Ab diesem Zeitpunkt befinden sich sowohl in der aktuellen als auch in allen zukünftigen Child Pattern und Parent Pattern Listen nur noch Pattern, die nicht mehr zum korrekten Pattern erweitert werden können. An dieser Stelle stellt sich die Frage, warum durch die Minimal Expansion diese zwei schlechte Pattern generiert werden. Durch die Verwendung des simulierten Model Repositories ist auch bekannt, dass auf jeden Fall mehrere korrekte Pattern in dem Eingabegraph vorkommen müssen. Die Minimal Expansion sollte nach Definition des SUBDUE Algorithmus das Parent Pattern um alle infrage kommenden Kanten erweitern und für jede dieser Möglichkeiten ein Expanded Pattern ausgeben. Zu erwarten wären also auch Expanded Pattern, die weiterhin ein Subgraph von dem korrekten Pattern sind. Da dieses Verhalten auf den ersten Blick nicht erklärbar ist, sind weitere Untersuchungen am SUBDUE Algorithmus notwendig, um anschließend entsprechende Maßnahmen ergreifen zu können, um die Performance des SUBDUE Algorithmus auf Model Repositories zu optimieren.

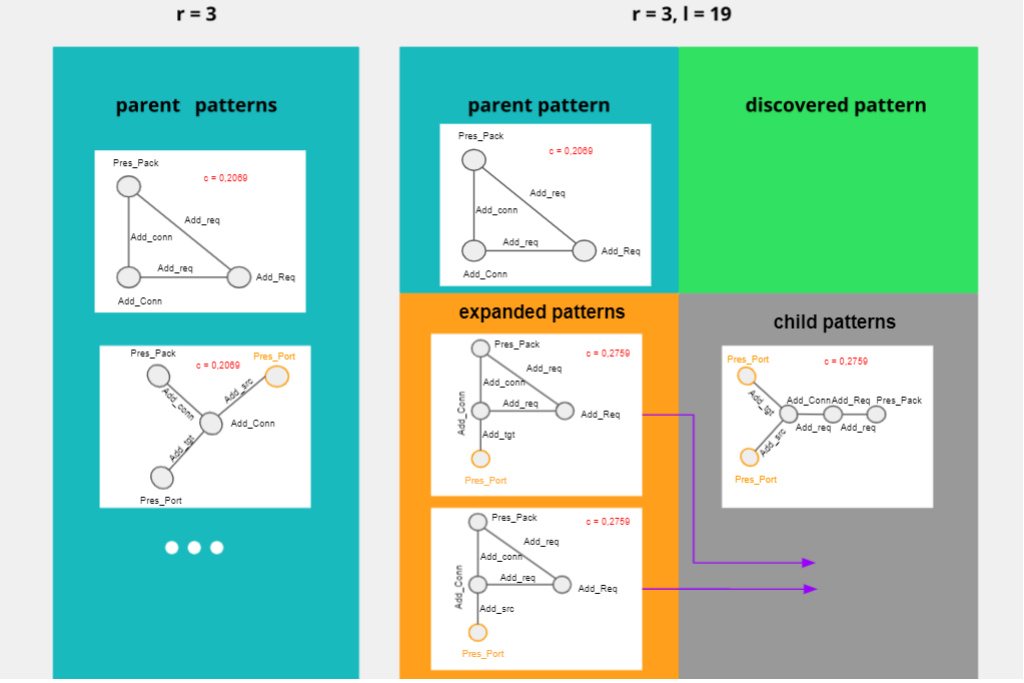

Abbildung 5.4: Zeitpunkt, an dem der SUBDUE nur noch falsche Pattern in der Beam Search Queue hält.

Kapitel 6

TheobaldSubdue

Die zuvor durchgeführten Experimente haben gezeigt, dass die aktuellen SUBDUE Implementierungen auf Model Repository Datensätze noch unzureichende Ergebnisse liefern. Zum einen findet das Mining mittels SUBDUE im Vergleich zum GASTON Algorithmus aus der Pilotstudie aus *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] weniger korrekte Pattern. Zum anderen wurde auch festgestellt, dass die Laufzeit mit zunehmender Komplexität des Eingabegraphens anwächst, weshalb der SUBDUE im Vergleich schlechter abgeschnitten hat, als die anderen Algorithmen in der Pilotstudie. Aus diesen Gründen ist der SUBDUE Algorithmus in seiner aktuellen Form noch nicht ideal für den Anwendungsfall auf Model Repositories geeignet. Im folgenden Kapitel werden nun Änderungen am SUBDUE Algorithmus vorgenommen, mit dem Ziel, die Performance und die Mining Ergebnisse für das Subgraph Mining von Edit Operations in Model Repositories zu steigern und die genannten Nachteile zu reduzieren. Als Ausgangslage für vorgeschlagenen THEOBALDSUBDUE dient die offizielle Python Implementierung des SUBDUE Algorithmus. Es werden Änderungen an der bestehenden Implementierung vorgenommen, sowie neue Designentscheidungen getroffen.

6.1 Umstrukturierung

Im Rahmen der Untersuchung der SUBDUE Implementierungen wurde festgestellt, dass der Ablauf der Substructure Discovery durch eine Umstrukturierung optimiert werden kann. Zur Erinnerung: In der inneren Schleife der Substructure Discovery wird ein Parent Pattern aus der Liste der Parent Pattern ausgewählt. Dieses Parent Pattern wird dann mithilfe der Minimal Expansion um die Größe eins erweitert werden. Dazu werden alle Möglichkeiten der Erweiterung um genau eine Kante erzeugt, um anschließend den Kompressionswert des neu erzeugten Expansion Patterns zu evaluieren.

In der SUBDUE Implementierung werden alle auf diese Weise erstellten Expansion Pattern *ep* daraufhin überprüft, ob die Anzahl an Kanten *ep.edges* des Patterns kleiner gleich der definierten maximalen Größe des zu suchenden Patterns in den SUBDUE Parametern entspricht. Somit wird die Überprüfung $ep.edges <= maxSize$ für jedes durch die Minimal Expansion erzeugte Pattern ausgeführt. Bei sehr großen Graphen, die besonders dicht sind, kann die Minimal Expansion eine Vielzahl an neuen Pattern generieren, die alle auf ihre Größe überprüft werden müssen. Da die

erzeugten Pattern aus der Minimal Expansion alle dieselbe Größe besitzen, werden nach der Überprüfung des ersten Expansion Patterns alle folgenden Überprüfung redundant ausgeführt. Optimieren lässt sich diese Abfrage, indem die Größe der Expansion Pattern vor der eigentlichen Durchführung der Minimal Expansion ausgeführt wird. Da die Minimal Expansion das Parent Pattern *pp* um genau eine Kante erweitert, ist die zu erwartende Größe aller Expansion Patterns nach der Minimal Expansion $pp.edges + 1$. Somit kann schon vor der Minimal Expansion berechnet werden, ob für die erzeugten Expansion Pattern $ep.edges <= maxSize$ gilt, indem vor der Minimal Expansion auf $pp.edges + 1 <= maxSize$ geprüft wird. Somit wird die Anzahl an Überprüfungen von n, wobei n die Anzahl an generierten Expansion Patterns durch die Minimal Expansion ist, auf 1 reduziert. Diese Umstrukturierung ist in der Abbildung 6.1 schematisch dargestellt.

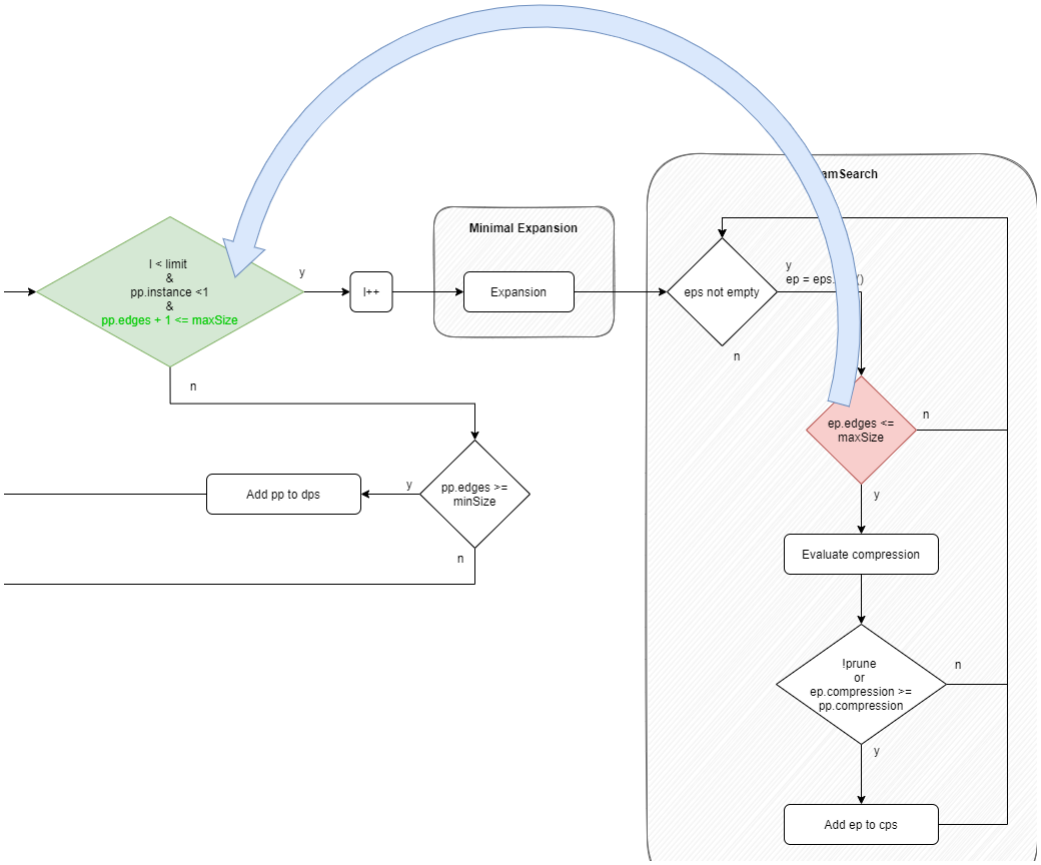

Abbildung 6.1: Optimiertes If-Statement durch Umstrukturierung der Substructure Discovery.

6.2 Overlap für Preserve Knoten

Des Weiteren wird der Overlap Mechanismus für das Subgraph Mining von Edit Operations spezialisiert. Auch hier wird die Eigenschaft der Preserve Knoten genutzt, um den Overlap für diesen konkreten Anwendungsfall anzupassen. In der Abbildung 6.2 ist ein beispielhafter Graph mit zwei Edit Operations abgebildet.

Eine Edit Operation besteht sowohl aus Preserve, als auch aus Nicht-Preserve (in der Farbe Rot gekennzeichnet) Elementen. Alle Nicht-Preserve Elemente sind dabei konkrete Veränderungen, die durch die Edit Operation abgebildet werden. In diesem Beispiel wird jeweils ein *Port* und ein *Connector* zwischen den *Component*s und dem *Package* hinzugefügt. Dabei ist das Hinzufügen des *Ports* und des *Connectors* eine konkrete Änderung an dem zugrundeliegenden Model, weshalb sie mit dem Label *Add* gekennzeichnet werden. Die Knoten *Component* und dem *Package* sind dabei nicht von einer konkreten Änderung betroffen, weshalb diese als *Preserve* gelabelt werden. Der Nicht-Preserve Teil ist mit dem restlichen Graphen über mindestens einen Preserve Knoten verbunden. Ist dies nicht der Fall so kann leicht gezeigt werden, dass der gesamte Graph aus einer einzigen Edit Operation besteht. Mehrere Edit Operationen können sich also nur an den Preserve Knoten überlappen. Diese Eigenschaft führt zu der Annahme, dass die Ergebnisse für diesen Anwendungsfall verbessert werden können, wenn der Overlap Mechanismus nur Preserve Knoten berücksichtigt. Aus diesem Grund wird die neue Strategie *preserve_vertex* für den Overlap Parameter eingeführt, der nur eine Überlappung für Preserve Knoten zulässt.

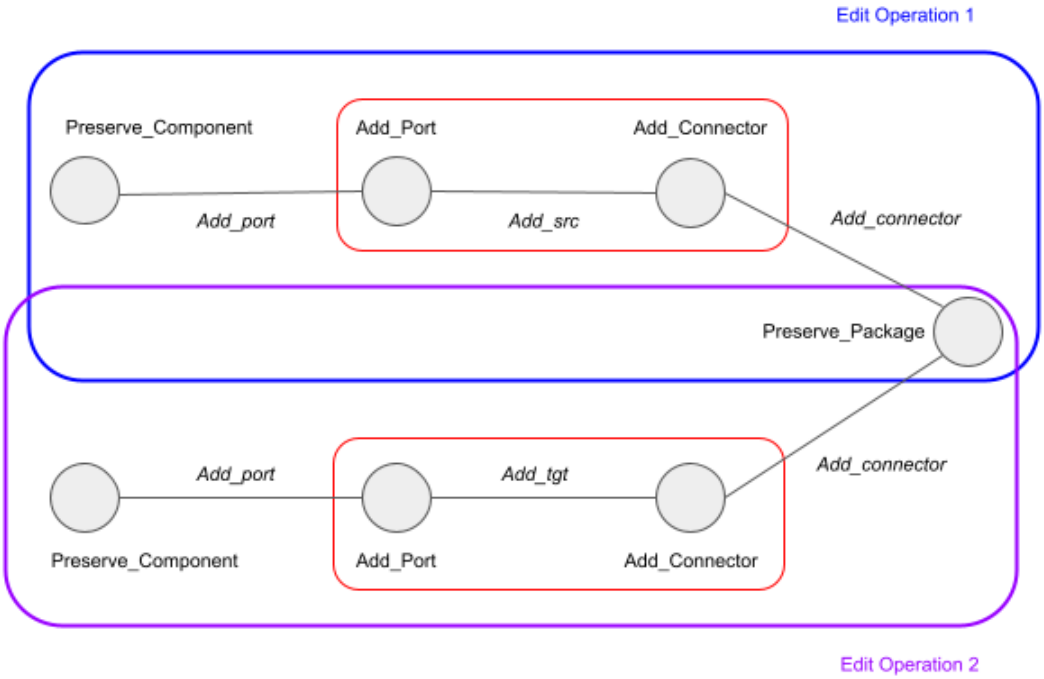

Abbildung 6.2: Overlap für Edit Operations.

6.3 Kompressionsmetrik ohne Preserve Elemente

Aktuell bieten die SUBDUE Implementierungen unterschiedliche Metriken zur Berechnung der Kompression an, siehe Abschnitt 3.5. Es wird die Annahme getroffen, dass eine auf den Anwendungsfall spezialisierte Metrik für Model Repositories die Mining Ergebnisse verbessern kann. Wie bereits im Abschnitt 3.6 erwähnt ist die Annahme, dass das Verwerfen der Preserve Elemente bei der Berechnung der Kompression zu besseren Ergebnissen führt. Das Herausrechnen der Preserve Elemente

kann hierbei nicht auf die *compression_heuristic* angewandt werden. Diese Metrik verwendet lediglich die Anzahl gefundenen Instanzen des Patterns und die Anzahl an Kanten. Eine Instanz eines Patterns besitzt kein eigenes Label und kann daher weder Preserve, noch Nicht-Preserve sein. Das Herausrechnen von Preserve Elementen aus den Instanzen führt zudem zu keiner Veränderung der Anzahl an Instanzen eines Patterns. Die Kanten einer Edit Operation können nicht mit dem Prefix Preserve gelabelt sein. Eine Kante kann nur dann Preserve sein, wenn beide Knoten ebenfalls Preserve sind. Dann sind aber sowohl Kanten und Knoten Preserve und somit ist dieser Graph keine Edit Operation, da er keine Modelltransformation beschreibt.

Die *compression_size* eignet sich für das Vorhaben, die Preserve Elemente aus der Berechnung der Kompression herauszunehmen. Diese Metrik berechnet die Kompression auf Basis der Größe des Patterns, des Eingabegraphens und des komprimierten Eingabegraphens, die jeweils Knoten enthalten können, die mit dem Prefix Preserve gelabelt sind. Für den TheobaldSubdue wird daher folgende neue Metrik zur Berechnung der Kompression verwendet mit dem Ziel, die Ergebnisse auf Model Repositories zu verbessern.

$compression\_size\_no\_preserve(P, G) = \frac{size(G)}{size(P) + size(G|P)}$ wobei für die Knoten von P und G gilt: $\#vertices = \#vertices - \#preserve\_vertices$

Kapitel 7

Experiment 3 - TheobaldSubdue

Um den TheobaldSubdue mit dem Subdue Algorithmus vergleichen zu können, wird im Folgenden ein weiteres Experiment durchgeführt. In diesem Experiment werden hierfür sowohl die Laufzeiten, als auch die Anzahl an erfolgreichen Identifikationen des korrekten Patterns gemessen. Die vorgeschlagenen Änderungen, die zum TheobaldSubdue führen, werden dabei auf einem Datensatz ausgeführt und gemessen. In einem abschließenden Versuch wird der TheobaldSubdue mit all seinen Änderungen gegen die Subdue Implementierungen und gegen den Frequent Subgraph Mining Algorithmus Gaston getestet, der von C. Tinnes bisher in *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] eingesetzt wird.

7.1 Fragestellung

E3Q1 Erzielt die vorgeschlagene Metrik, die die Kompression ohne die Preserve Elemente berechnet, bessere Ergebnisse auf Model Repositories?

E3Q2 Erzielt das Overlapping nur für Preserve Knoten bessere Ergebnisse?

E3Q3 Kann der TheobaldSubdue Algorithmus im Vergleich zum Subdue und dem Gaston mehr korrekte Pattern auf Model Repositories identifizieren?

7.2 Aufbau

Auch für dieses Experiment wird derselbe Aufbau der simulierten Model Repositories aus den vorherigen Experimenten genutzt. Für dieses abschließende Experiment wird allerdings der gesamte Datensatz aller simulierten Model Repositories verwendet, die jeweils eine Versionshistorie von 10 besitzt. Für jedes Model Repository existieren also auch hier 10 Versionen des Modells, auf die jeweils eine unterschiedliche Anzahl der Edit Operation 4.2 angewandt wurde. Ebenso wurden weitere Edit Operationens als Störfaktor einberechnet.

7.3 Durchführung

Dieses Experiment ist in drei unterschiedlichen Versuchen gegliedert, die sowohl die einzelnen Änderungen des TheobaldSubdue messen, als auch das Zusammenspiel

Iterations	1	**BeamWidth**	4
Limit	50	**Overlap**	True
MaxSize	50	**Prune**	False
MinSize	3	**ValueBased**	False

Algorithm	**Metric**	**Match** (%)
TheobaldSubdue	Size without Preserve	91,19
Subdue Python	Heuristic	89,79
Subdue Python	Size	89,99
Subdue C	MDL	89,89
Subdue C	Size	90,19

Tabelle 7.1: Experiment 3, Versuch 1: Kompressionsmetrik ohne Preserve Elemente.

aller Änderungen im Vergleich zum SUBDUE und dem GASTON.

Versuch 1

In Versuch 1 wird die vorgeschlagene Metrik zur Berechnung der Kompression ohne Preserve Elemente überprüft. Hierfür wird als Benchmark sowohl die Python als auch die C Implementierung des SUBDUE mit den unterschiedlichen zur Verfügung stehenden Metriken auf dem Datensatz ausgeführt. Anschließend wird der THEOBALDSUBDUE auf dem gleichen Datensatz gemessen.

Versuch 2

Versuch 2 testet die neue Strategie für die Überlappung nur für Preserve Knoten. Aus dem Versuch 1 ist bekannt, dass die vorgeschlagene Metrik bessere Ergebnisse erzielt. Daher wird als Benchmark der THEOBALDSUBDUE mit der Metrik ohne Preserve Elemente herangezogen. Der THEOBALDSUBDUE wird einmal mit dem neuen Preserve Vertex Overlap und einmal mit dem Vertex Overlap aus dem SUBDUE auf dem Datensatz ausgeführt.

Versuch 3

Aus Versuch 2 geht hervor, dass die vorgeschlagene Overlap Strategie keine besseren Ergebnisse liefert. Daher wird hier der THEOBALDSUBDUE nur mit der neu vorgestellten Metrik und der Umstrukturierung auf dem Datensatz ausgeführt. Als Benchmark dienen hier ebenso die beiden SUBDUE Implementierungen in C und in Python, sowie der GASTON Algorithmus. Zusätzlich wird noch die Laufzeit gemessen. Diese ist allerdings mit Vorsicht zu genießen, da die Implementierungen in unterschiedlichen Programmiersprachen entwickelt sind, weshalb diese reine Laufzeit der Ausführung aufgrund der Sprachenabhängigkeit nicht fair verglichen werden kann.

Algorithm	**Metric**	**Overlap**	**Match** (%)
TheobaldSubdue	Size without Preserve	Preserve_Vertex	88,99
TheobaldSubdue	Size without Preserve	Vertex	91,19

Tabelle 7.2: Experiment 3, Versuch 2: Overlap für Preserve Knoten.

Iterations	1	**BeamWidth**	4
Limit	50	**Overlap**	True
MaxSize	50	**Prune**	False
MinSize	3	**ValueBased**	False

Algorithm	**Runtime** (s)	**Match** (%)
TheobaldSubdue	3,5333	91,19
Subdue Python (Heuristic)	2,9656	89,79
Subdue Python (Size)	3,1184	89,99
Subdue C (MDL)	0,4370	89,89
Subdue C (Size)	0,4233	90,19
Gaston	1,1924	94,79

Tabelle 7.3: Experiment 3, Versuch 3: TheobaldSubdue im Vergleich.

7.4 Evaluation

Aus den Messungen des ersten Versuchs lässt sich entnehmen, dass die neu vorgeschlagene Metrik zur Berechnung der Kompression erfolgreich mehr korrekte Pattern identifiziert. Die Berechnung der Kompression ohne die Preserve Elemente kann daher erfolgreich zur Optimierung eingesetzt werden und beantwortet somit die Frage **E3Q1**. Die neu implementierte Strategie für den Overlap Mechanismus, der nur noch die Preserve Knoten berücksichtigt, hat nicht den erhofften Effekt gezeigt. Die Mining Ergebnisse wurden dadurch minimal schlechter, sodass die Frage **E3Q2** verneint werden muss. Dieses Resultat entspricht nicht der getroffenen Annahme. In dieser Arbeit lässt sich der Grund für das Fehlschlagen nicht aufklären. Daher sind weitere Untersuchungen notwendig, um folgende neue Fragestellung zu beantworten:

E3Q4 Warum erzielt das Overlapping nur für Preserve Knoten entgegengesetzt der getroffenen Annahme schlechtere Mining Ergebnisse?

Diese Frage wird ebenfalls nochmals im Ausblick zu weiteren Untersuchungs- und Optimierungsansätzen erwähnt. Die letzte offene Frage **E3Q3** kann durch die Messungen aus dem dritten Versuch beantwortet werden. Diese zeigen, dass der TheobaldSubdue bessere Ergebnisse erzielt als der Subdue. Die Laufzeitkosten haben sich dafür minimal erhöht. Jedoch bleibt der TheobaldSubdue bzgl. den korrekten Ergebnissen unter dem Gaston Algorithmus. Es bleibt zu klären, ob TheobaldSubdue den Gaston mit weiteren, im Kapitel 8 Ausblick erwähnten Anpassungen übertreffen kann.

Kapitel 8

Ausblick

Es konnte zudem gezeigt werden, dass der vorgeschlagene TheobaldSubdue ein minimal verbessertes Ergebnis für das Subgraph Mining auf Model Repositories erzielt. Im Vergleich zum Gaston Algorithmus schneidet der TheobaldSubdue, wie auch der Subdue minimal schlechter mit weniger Matches des korrekten Patterns auf dem vorliegenden Datensatz ab. Um die Ergebnisse weiter zu optimieren, werden Ansatzpunkte für eine Weiterentwicklung des TheobaldSubdue aufgeführt.

8.1 Untersuchung der Minimal Expansion

Aus dem Ergebnis aus Experiment 2 wird ersichtlich, dass die Minimal Expansion des Subdue Algorithmus ungünstige Entscheidungen treffen kann. Dies kann dazu führen, dass das beste Pattern nicht mehr identifiziert werden kann und somit nicht die interessanteste Edit Operation erkannt wird. Im Rahmen dieser Thesis konnte nicht festgestellt werden, wann und aus welchen Gründen diese ungünstigen Entscheidungen auftreten. Die Experimente 1 und 3 zeigen allerdings, dass dieses Verhalten im Schnitt nicht übermäßig häufig auftritt. Um die Ergebnisse auf Model Repositories weiter zu optimieren, ist die Untersuchung der Minimal Expansion ein erster Ansatzpunkt.

8.2 Untersuchung der BeamWidth

Die Resultate von Experiment 1 zeigen, dass der Subdue Algorithmus sehr sensibel auf Veränderungen des BeamWidth Parameters reagiert. Ab einer BeamWidth von 4 fällt die Rate der korrekten Ergebnisse signifikant ab. Die BeamWidth bestimmt den Suchraum der Identifikation des interessantesten Patterns. Die Erwartung ist, dass ein größerer Suchraum zu tendenziell besseren Ergebnissen führt, da mehr Pattern in der Beam Search berücksichtigt werden. Entgegen den Erwartungen sinkt die Trefferquote jedoch erheblich ab der Parametereinstellung $BeamWidth = 4$. Dies ist in beiden Implementierungen des Subdue Algorithmus zu beobachten, sodass die Annahme getroffen werden kann, dass dieses Verhalten auf das Verfahren und nicht auf die Implementierung zurückzuführen ist. Um diese Annahme zu überprüfen, sind weitere Untersuchungen notwendig. Hierfür lässt sich der im Rahmen dieser Thesis entwickelte Debugger einsetzen. Des Weiteren wäre es interessant zu erfahren, ob dieses Verhalten verstärkt auf Model Repositories auftritt, insbesondere in den

verwendeten Datensätzen dieser Arbeit oder auch auf anderen Datensätzen anderer Anwendungsfälle.

8.3 Untersuchung der unterschiedlichen Ergebnisse der Subdue Implementierungen

Ebenso kann aus dem Experiment 1 entnommen werden, dass die SUBDUE Implementierungen in C und Python trotz gleicher Parametereinstellungen zu unterschiedlichen Mining Ergebnissen führen. An dieser Stelle ergibt sich die Frage nach den Ursachen für dieses gezeigte Verhalten. Im Rahmen dieser Arbeit konnte dies nicht endgültig geklärt werden, weshalb weitere Untersuchungen erforderlich sind. Der THEOBALDSUBDUE basiert auf der SUBDUE Python Implementierung. Aus dem Experiment ist ersichtlich, dass die C Implementierung im Durchschnitt minimal bessere Ergebnisse liefert. Wenn die Ursache dafür gefunden wird, dann könnte der THEOBALDSUBDUE angepasst werden, sodass er noch bessere Ergebnisse erzielt.

8.4 Untersuchung des Overlaps für Preserve Knoten

Die Messungen aus dem Experiment 3 haben gezeigt, dass die für den THEOBALDSUBDUE entwickelte Overlap Strategie, entgegengesetzt der Annahme, zu keinen besseren Mining Ergebnissen geführt hat. Im Rahmen der Arbeiten konnten keine Einwände gefunden werden, weshalb die vorgeschlagene Overlap Strategie nicht die erhofften Erwartungen erfüllen konnte. Daher sind weitere Untersuchungen notwendig, um Implementierungsfehler vollends ausschließen zu können. Falls keine Fehler vorliegen sollten, so bliebe weiterhin die Frage offen, weshalb diese Strategie zu schlechteren Ergebnissen führt.

8.5 Implementierung einer Pattern Distanz

In einer Pilotstudie von C. Tinnes im Rahmen der Publikation *Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining* [2] wird erkannt, dass der SUBDUE Algorithmus nur einige geringe Anzahl an unterschiedlichen Pattern identifizieren kann. Zu demselben Entschluss gelangen auch die Autoren in der Veröffentlichung *A Survey of Frequent Subgraph Mining Algorithms* [36]. Um zukünftig einen Vorschlagsgenerator für Edit Operations auf einem Model Repository entwickeln zu können, sollte eine möglichst vielfältige Auswahl an Pattern identifiziert werden, die sich mit möglichst großer Distanz unterscheiden. Um dies zu gewährleisten, muss die Beam Search des THEOBALDSUBDUE Algorithmus angepasst werden, sodass nicht nur den Pattern mit der größten Kompression nachgegangen wird, sondern dass sich ebenfalls die Distanz der Pattern in der Beam Search Queue größtmöglich unterscheiden.

Anhang

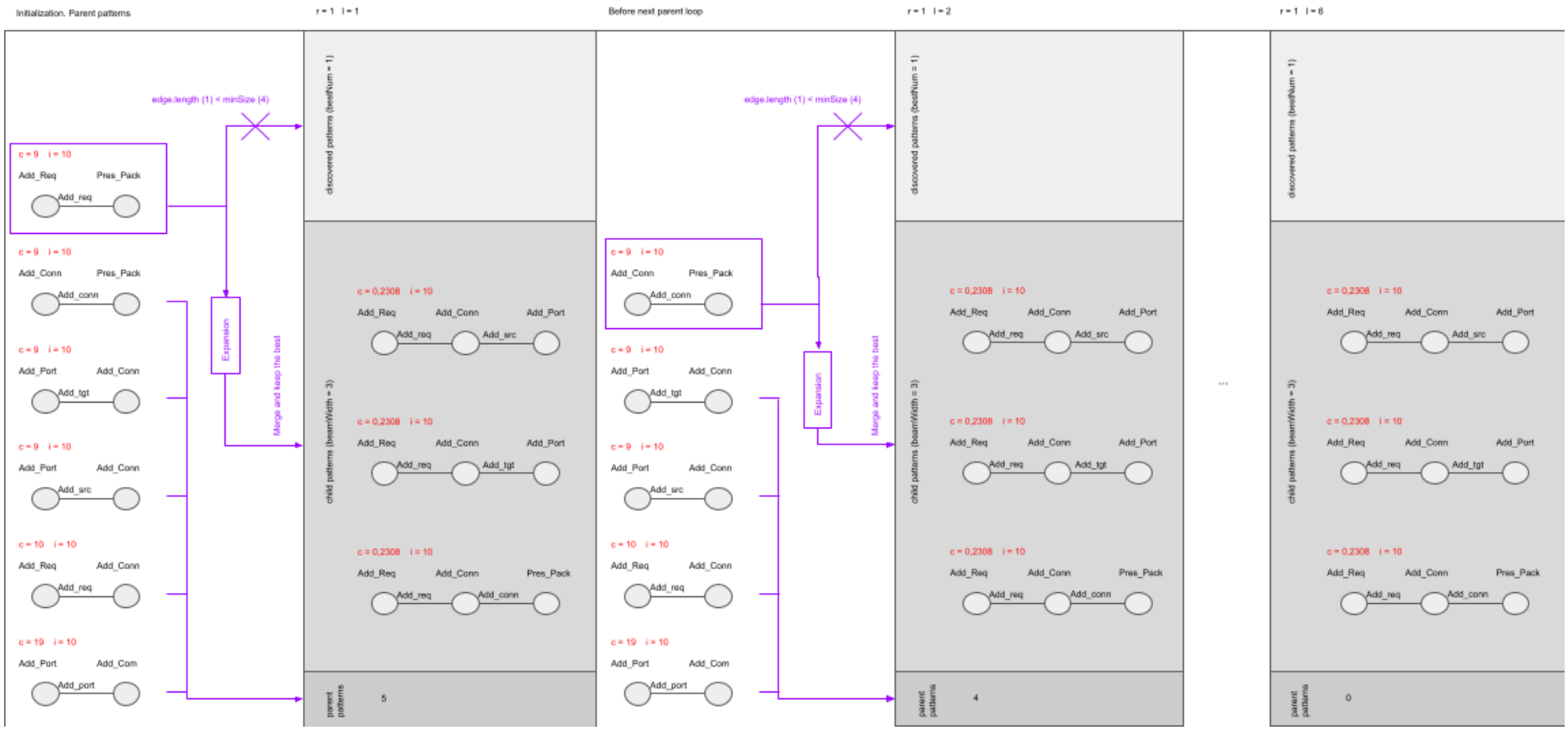

Abbildung 8.1: Initialisierung mit 1-Edge Graphen und anschließender Beam Search Queue in der ersten Iteration.

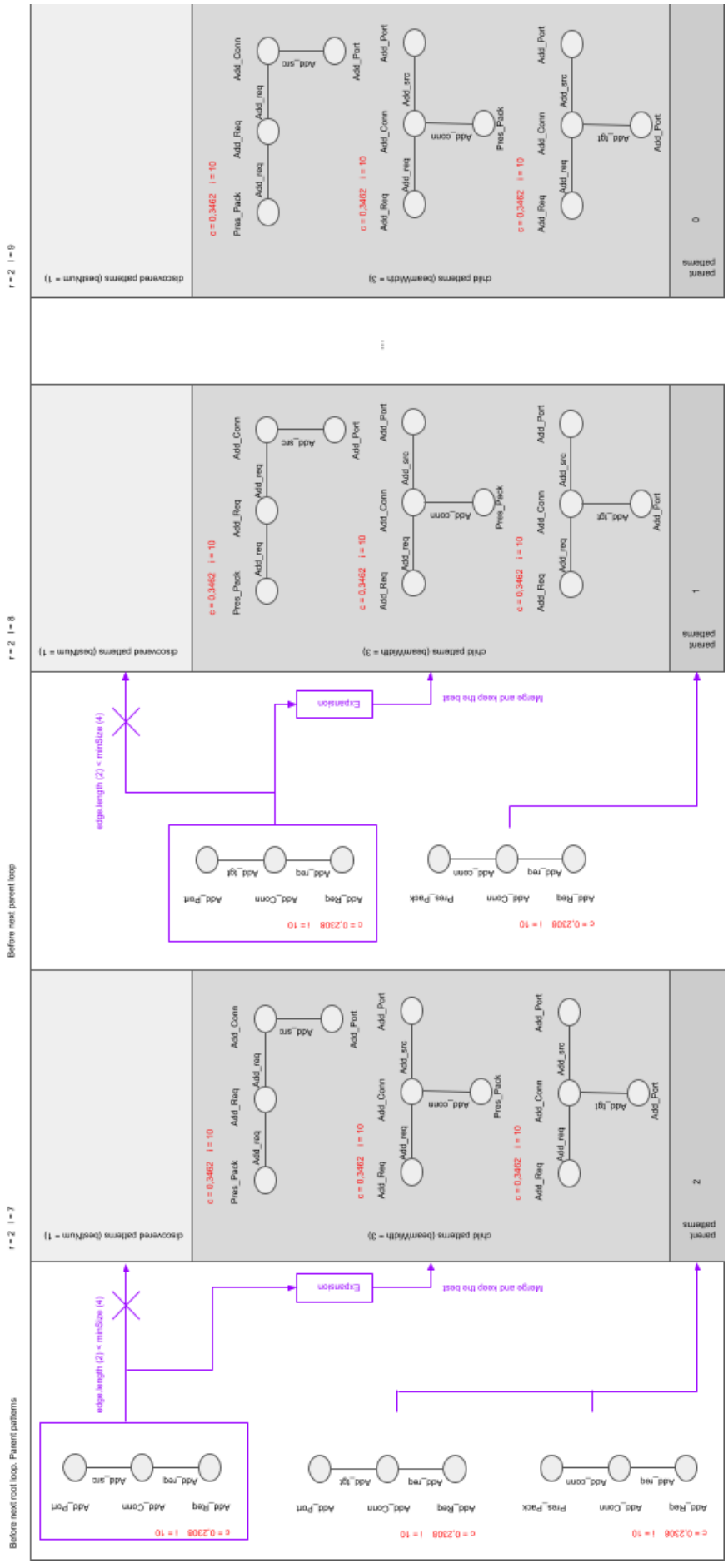

Abbildung 8.2: Beam Search Queue in der zweiten Iteration.

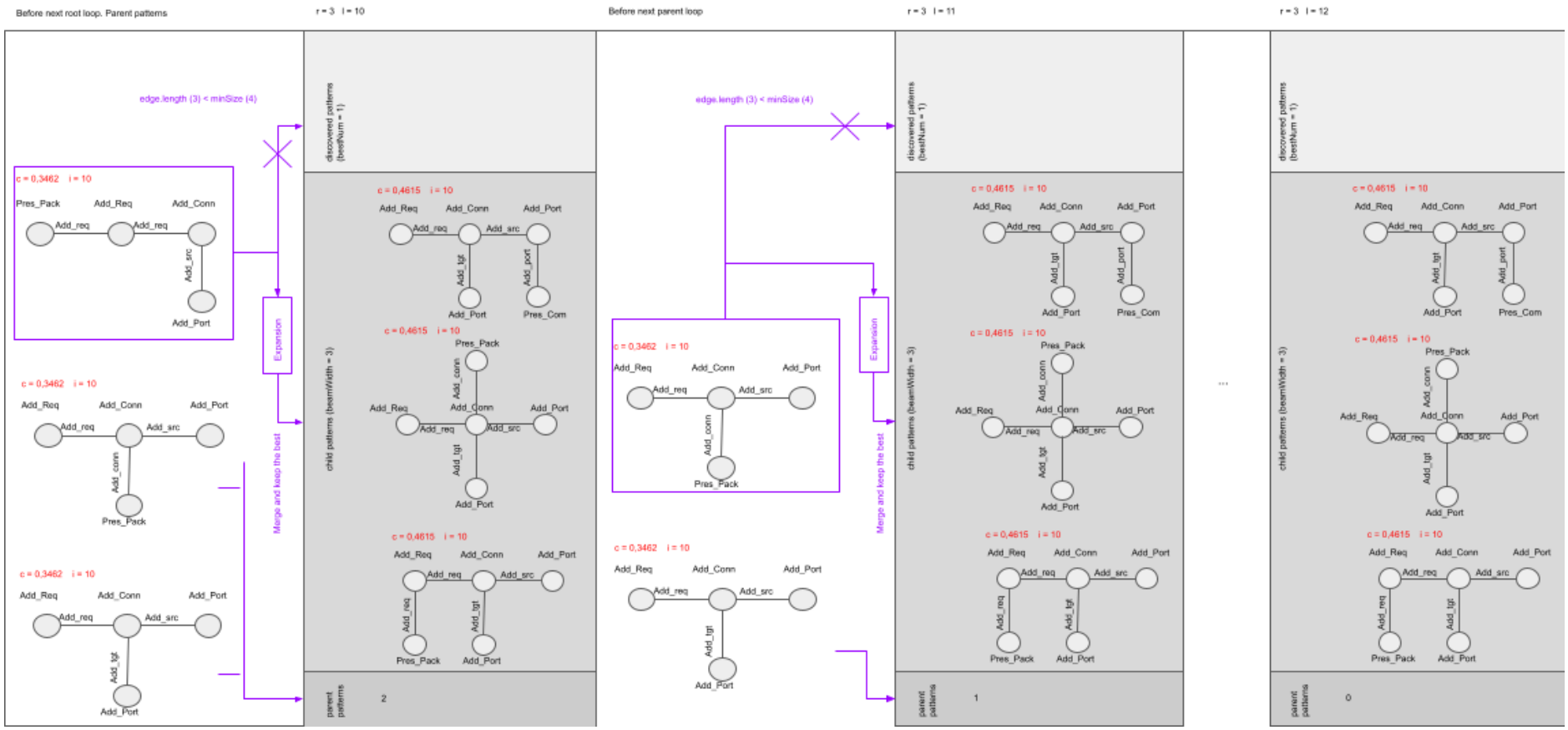

Abbildung 8.3: Beam Search Queue in der dritten Iteration.

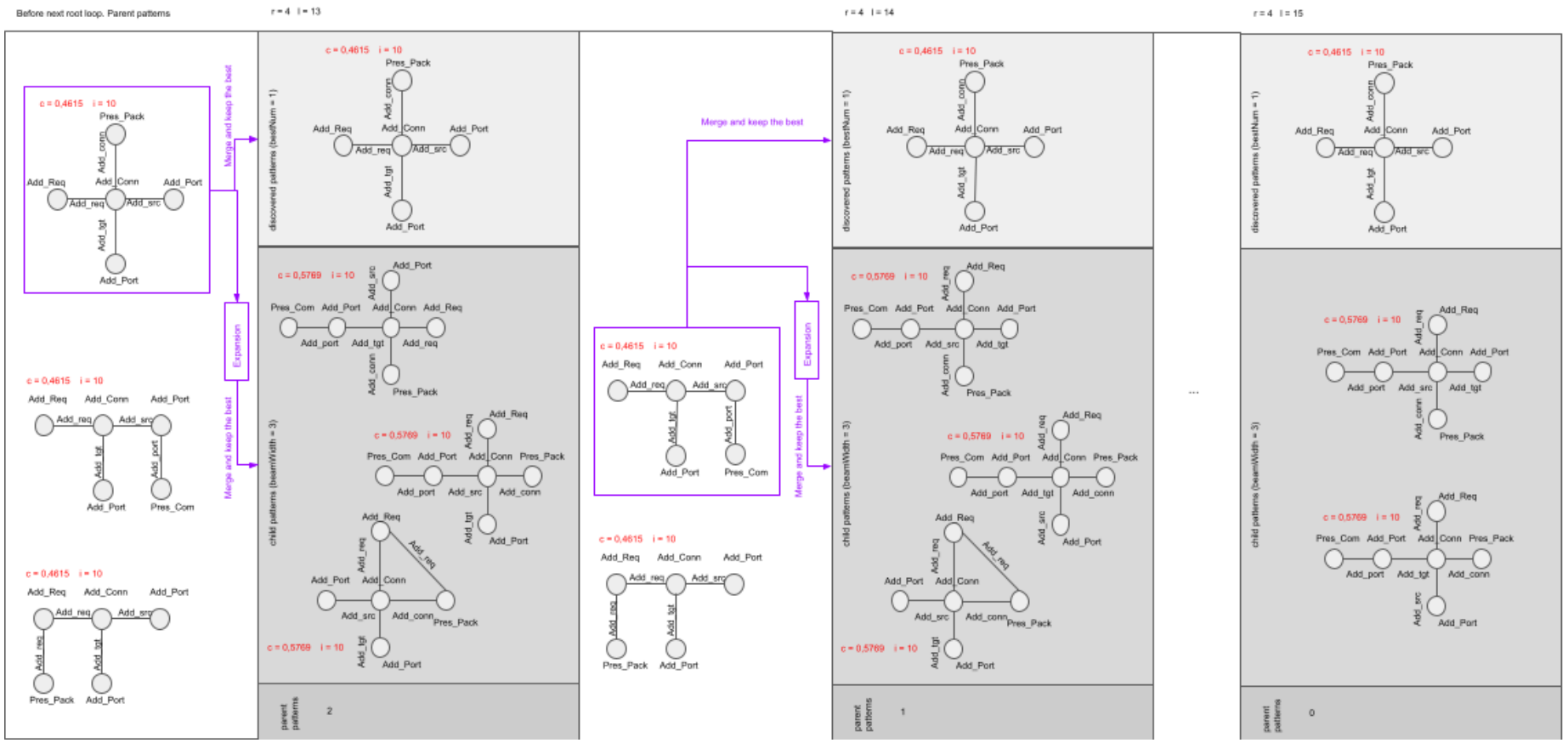

Abbildung 8.4: Beam Search Queue in der vierten Iteration.

Literatur

[1] J. Whittle, J. Hutchinson und M. Rouncefield, "The State of Practice in Model-Driven Engineering," *IEEE Software*, 2014.

[2] C. Tinnes, T. Kehrer, M. Joblin, U. Hohenstein, A. Biesdorf und S. Apel, "Learning Domain-Specific Edit Operations from Model Repositories with Frequent Subgraph Mining," *arXiv preprint arXiv:2108.01001*, 2021.

[3] L. Kesavan, "Frequent Subgraph Mining Algorithms - A Survey and Framework for Classification," *Computer Science Information Technology*, 2012.

[4] L. B. Holder und D. J. Cook, "Discovery of Inexact Concepts from Structural Data," *IEEE Educational Activities Department*, 1993.

[5] P. Grünwald, "The Minimum Description Length Principle," 2007.

[6] B. Randell, "Software Engineering: As it was in 1968.," Jan. 1979.

[7] C. Krueger, "Software reuse," *ACM Comput. Surv.*, 1992.

[8] S. Sendall und W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development," *IEEE Softw.*, 2003.

[9] S. Djoko, D. J. Cook und L. B. Holder, "An empirical study of domain knowledge and its benefits to substructure discovery," *IEEE Transactions on Knowledge and Data Engineering*, 1997.

[10] S. Nijssen und J. Kok, "The Gaston Tool for Frequent Subgraph Mining," *Electronic Notes in Theoretical Computer Science*, 2005.

[11] W. Eberle und L. Holder, *Graph-Based Knowledge Discovery: Compression vs. Frequency*, 2011.

[12] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, 1980.

[13] H. Wright, T. D. Winters und T. Manshreck, "Software Engineering at Google," *Software Engineering at Google*, 2020.

[14] M. Hairul, M. Md Nasir und S. Sahibuddin, "Critical success factors for software projects: A comparative study," *Scientific Research and Essays*, 2011.

[15] H. Vliet, "Software engineering - principles and practice," 1993.

[16] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems Structures*, 2015.

[17] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, 2003.

[18] B. Randell, "Fifty Years of Software Engineering - or - The View from Garmisch," *CoRR*, 2018.

[19] J. Di Rocco, D. Di Ruscio, L. Iovino und A. Pierantonio, "Collaborative Repositories in Model-Driven Engineering [Software Technology]," *IEEE Softw.*, 2015.

[20] B. Hamid, "A model-driven approach for developing a model repository: Methodology and tool support," *Future Generation Computer Systems*, 2017.

[21] T. Arendt, E. Biermann, S. Jurack, C. Krause und G. Taentzer, "Henshin: advanced concepts and tools for in-place EMF model transformations," 2010.

[22] E. Biermann, K. Ehrig, C. Ermel, C. Krause und G. Taentzer, "The EMF Model Transformation Framework," 2007.

[23] S. u. Rehman, K. Ullah und S. Fong, "Graph mining: A survey of graph mining techniques," 2012.

[24] R. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations*, 1972.

[25] S. Cook, "The complexity of theorem-proving procedures," *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[26] S. Cook, "The P versus NP problem," *The millennium prize problems*, 2001.

[27] B. D. McKay, "Practical Graph Isomorphism," 1981.

[28] M. R. Garey und D. S. Johnson, "Computers and Intractability; A Guide to the Theory of NP-Completeness," 1990.

[29] J. Rissanen, "Modeling By Shortest Data Description*," 1978.

[30] A. J. M. Garrett, "Ockham's Razor," 1991.

[31] M. Madiman, M Harrison und I. Kontoyiannis, "Minimum description length vs. maximum likelihood in lossy data compression," 2005.

[32] L. Holder, "Substructure Discovery in SUBDUE," 1988.

[33] D. Cook und L. Holder, "Substructure Discovery Using Minimum Description Length and Background Knowledge," 1999.

[34] R. D. R. Bruce Lowerre, "The Harpy Speech Understanding," 1976.

[35] C. H. You, L. B. Holder und D. J. Cook, "Graph-Based Data Mining in Dynamic Networks: Empirical Comparison of Compression-Based and Frequency-Based Subgraph Mining," *2008 IEEE International Conference on Data Mining Workshops*, 2008.

[36] C. Jiang, F. Coenen und M. Zito, "A Survey of Frequent Subgraph Mining Algorithms," *The Knowledge Engineering Review*, 2004.