technische universität
dortmund

Erweiterte Substruktursuche in Moleküldatenbanken und ihre Integration in Scaffold Hunter

Nils Kriege

Algorithm Engineering Report
TR10-1-001
Februar 2010
ISSN 1864-4503

Fakultät für Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

technische universität
dortmund

Diplomarbeit

Erweiterte Substruktursuche in Moleküldatenbanken und ihre Integration in Scaffold Hunter

Nils Kriege

18. Dezember 2009

Betreuer:
Prof. Dr. Petra Mutzel
Dipl.-Inform. Karsten Klein

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de

An dieser Stelle möchte ich mich bei Frau Professor Petra Mutzel, an deren Lehrstuhl ich diese Diplomarbeit durchführen durfte, für die Betreuung und kooperative Unterstützung bei der Themenfindung bedanken.
Mein ganz besonderer Dank gilt jedoch Karsten Klein, der mir jederzeit ein kompetenter Ansprechpartner und verlässlicher Ratgeber war.
Ferner danke ich Stefan Wetzel, der das Thema der Arbeit angeregt und zur Klärung von Sachfragen aus dem Bereich der Chemie beigetragen hat.

Inhaltsverzeichnis

Kapitel 1

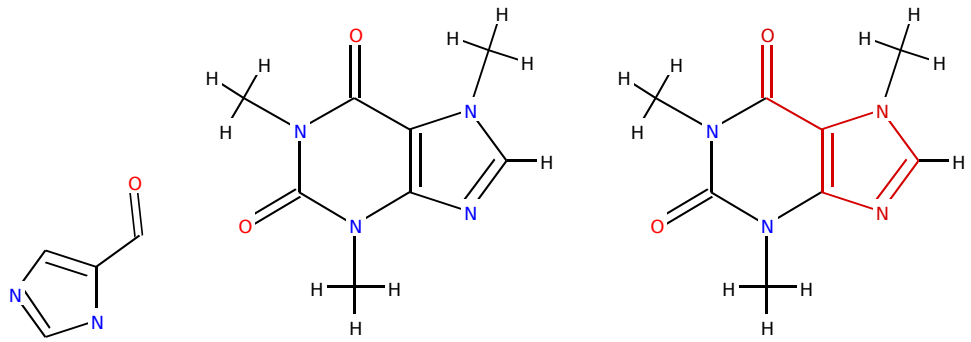
Einleitung

Die vorliegende Diplomarbeit beschreibt die Entwicklung und Implementierung eines Systems zur Substruktursuche in Moleküldatenbanken, das in Scaffold Hunter, einer Software zur Erkundung des chemischen Strukturraums, integriert wurde. In diesem Kapitel werden die Grundlagen der Substruktursuche erläutert (Abschnitt 1.1) und es wird der Zusammenhang zwischen chemischen Verbindungen und Graphen hergestellt (Abschnitt 1.2). Abschnitt 1.3 gibt einen Überblick über den weiteren Aufbau der Arbeit.

1.1 Suche in Moleküldatenbanken

Moleküle bestehen aus Atomen, die durch Atombindungen miteinander verbunden sind. Die Anzahl möglicher Moleküle ist gewaltig und es ist eine Aufgabe der Chemie, ihren Aufbau aufzudecken, Zusammenhänge zwischen ihnen herzustellen und ihre Umwandlung durch chemische Reaktionen zu erforschen. Um die dabei anfallenden Daten zu verwalten, kommen computergestützte Informationssysteme zum Einsatz. Eine in der Chemie übliche Darstellung von Molekülen sind 2-dimensionale Strukturformeln, die Atome und ihre Bindungen veranschaulichen. Diese Form der Darstellung kommt nicht nur der tatsächlichen Natur von chemischen Verbindungen nahe, sondern ist außerdem gut geeignet, um mit Hilfe von Computern verwaltet zu werden. Eine Datenbank, in der Moleküle mit ihrem Aufbau und den verschiedensten Eigenschaften gespeichert sind, wird als Moleküldatenbank bezeichnet.

Eine elementare Voraussetzung für den sinnvollen Einsatz von Moleküldatenbanken ist es, dass die gewünschten Informationen auf angemessene Weise abgerufen werden können. Dazu werden häufig zwei intuitive Möglichkeiten zur Suche angeboten [68]: Die Struktursuche ermöglicht es, die Datenbank nach einer vom Benutzer vollständig angegebene Struktur zu durchsuchen und, falls sich die Struktur in der Datenbank befindet, die zugehörigen Eigenschaften abzurufen. Bei der Substruktursuche gibt der Benutzer ein Molekülfragment an und das Informationssystem liefert alle Moleküle, die dieses Fragment enthalten. Ab-



(a) (b) (c)

Abbildung 1.1: Das Suchmuster (a) ist in dem Koffeinmolekül (b) als Substruktur enthalten. Diese ist auf Bild (c) rot hervorgehoben.

bildung 1.1 zeigt die Strukturformel von Koffein sowie ein mögliches Suchmuster und das Vorkommen des Suchmusters als Substruktur im Koffeinmolekül. Da Substrukturen eines Moleküls in Zusammenhang mit bestimmten chemischen und biologischen Eigenschaften gebracht werden können, stellt diese Form der Suche eine sinnvolle Art der Recherche in Molekül datenbanken dar. Eine Erweiterung der Substruktursuche besteht darin, dass das Suchmuster nicht exakt angegeben werden muss, sondern bestimmte Atome oder Bindungen vom Benutzer als variabel gekennzeichnet werden können. Variable Elemente können auf unterschiedliche Atom- bzw. Bindungstypen abgebildet werden. Diese Form der Suche in Molekül datenbanken wird in dieser Diplomarbeit behandelt. Der Vollständigkeit halber sei erwähnt, dass weitere Möglichkeiten der Datenbanksuche in der Chemie angewandt werden. Dazu zählt die Ähnlichkeitssuche, bei der das Informationssystem Moleküle, sortiert nach ihrer Ähnlichkeit zu einer vom Benutzer angegebenen Struktur, ausgibt [70].

Struktur- und Substruktursuche ist seit über 50 Jahren Gegenstand der Forschung und erste Ansätze gehen auf Kirsch et al. [53] zurück. Es gibt zahlreiche Anbieter kommerzieller Systeme, allerdings kaum geeignete frei verfügbare Software, die in ein Open-Source-Programm wie Scaffold Hunter integriert werden könnte.

Die Suche nach Fragmenten in Molekülen ist nicht mit der verbreiteten Suche in Textdokumenten zu vergleichen, sondern gestaltet sich wesentlich aufwendiger. Moleküle bestehen aus Objekten, die in komplexer Weise miteinander in Beziehung stehen - die formalen Grundlagen zur Beschreibung derartiger Strukturen liefert die Graphentheorie.

1.2 Graphentheorie in der Chemie

Strukturformeln lassen sich durch Graphen repräsentieren. Die hier verwendeten Definitionen von Graphen basieren auf [21].

Definition 1.1 (Gerichteter Graph). Ein *gerichteter Graph* G ist ein Tupel (V, E), wobei V eine endliche Menge und $E \subseteq V \times V$ eine binäre Relation auf V ist. Die Elemente von V werden als *Knoten*, die Elemente von E als *Kanten* des Graphen G bezeichnet.

Definition 1.2 (Ungerichteter Graph). Ein *ungerichteter Graph* G ist ein Tupel (V, E) bestehend aus der Knotenmenge V und einer Kantenmenge $E \subseteq \{X \subseteq V || X| = 2\}$, einer Teilmenge aller 2-elementigen Teilmengen von V.

Auch für eine Kante $e = \{u, v\}$ in einem ungerichteten Graphen wird im Folgenden die Notation (u, v) verwendet, wobei (u, v) und (v, u) die gleiche Kante meinen. Existiert eine Kante $e = (u, v)$, so heißen die Knoten u und v *adjazent* oder *benachbart* und werden als *inzident* zu der Kante e bezeichnet. Der *Grad* $deg(u)$ eines Knotens u ist die Anzahl der zu ihm inzidenten Kanten. In einem gerichteten Graphen führt die Kante $e = (u, v)$ *von* u *nach* v und wird als von u *ausgehend* und in v *eingehend* bezeichnet. Der *Ausgangsgrad* $deg^+(u)$ des Knotens u ist die Anzahl der von u ausgehenden Kanten und $deg^-(u)$ die Anzahl der in u eingehenden Kanten. Eine Kante $e = (u, u)$ wird als *Schleife* bezeichnet. Ungerichtete Graphen enthalten keine Schleifen.

Definition 1.3 (Gelabelter Graph). Ein *gelabelter Graph* G ist ein gerichteter oder ungerichteter Graph $G = (V, E)$ zusammen mit einer Funktion $l : V \cup E \to \Sigma$. Für einen Knoten u (eine Kante e) wird $l(u)$ ($l(e)$) als *Label* des Knotens (der Kante) bezeichnet.

Eine Strukturformel, wie sie in Abbildung 1.1 zu sehen ist, kann durch einen ungerichteten gelabelten Graphen beschrieben werden. Die Atome des Moleküls entsprechen den Knoten des Graphen und die Atombindungen den Kanten. Der Atomtyp bzw. die Art der Bindung wird durch die Label des Graphen repräsentiert. Für eine Einfachbindung wird im Folgenden das Label "`-`" verwendet, für eine Doppelbindung das Label "`=`". Der Atomtyp wird durch das Symbol des Elements (`N`, `O`, ...) repräsentiert. Der aus einer Strukturformel abgeleitete ungerichtete gelabelte Graph wird nachfolgend als *Molekülgraph* bezeichnet.

Bei dieser Repräsentation von Molekülen durch ihren Molekülgraphen können gewisse Informationen, wie die geometrische Anordnung der Atome zueinander, nicht berücksichtigt werden. Für die Substruktursuche in Moleküldatenbanken ist diese Vereinfachung jedoch geeignet. Das grundlegende Problem, ob ein Molekülfragment in einem Molekül enthalten ist, entspricht also der Frage, ob ein gelabelter Graph in einem anderen enthalten ist. Diese Fragestellung wird als *Teilgraph-Isomorphie-Problem* bezeichnet.

1.3 Aufbau der Arbeit

Der Ablauf einer Substruktursuche ist schematisch in Abbildung 1.2 dargestellt. Die Eingabe ermöglicht es dem Nutzer, eine Molekülfragment anzugeben, das gesucht werden soll.

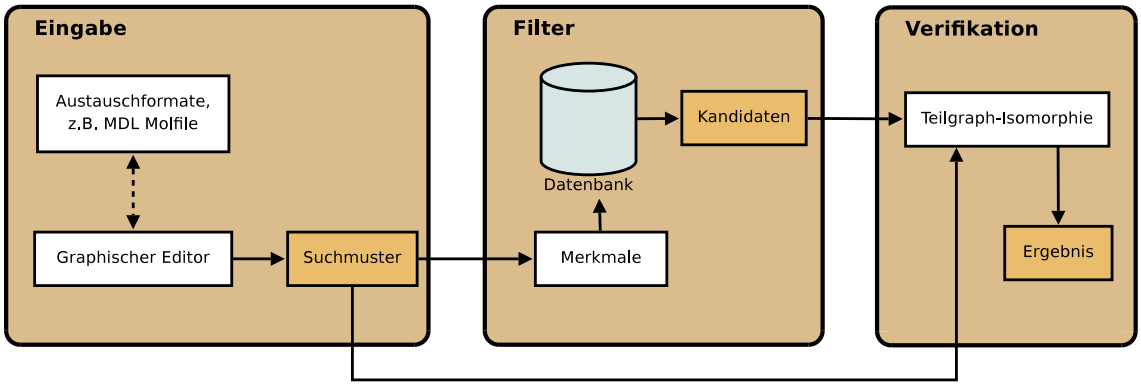


Abbildung 1.2: Schematischer Ablauf einer Substruktursuche.

Dazu dient ein graphischer Strukturedítor, mit dem Suchmuster neu erstellt oder aus einem in der Chemie üblichen Austauschformat geladen und anschließend editiert werden können. Typischerweise werden in einem nächsten Schritt, der Filterphase, bestimmte charakteristische Merkmale des Suchmusters verwendet, um Moleküle, die das Suchmuster auf keinen Fall enthalten, vorab auszuschließen. Das Ergebnis dieses Schritts ist eine Menge von Kandidaten, die im letzten Schritt daraufhin überprüft werden, ob sie das Suchmuster tatsächlich enthalten.

Der Aufbau dieser Arbeit folgt der Einteilung der Substruktursuche in die einzelnen Phasen: In Kapitel 2 wird die Software Scaffold Hunter vorgestellt, die Anwendungen der Substruktursuche im Kontext des Programms und die Integration eines graphischen Strukturedítors beschrieben. Kapitel 3 befasst sich mit dem grundlegenden graphentheoretischen Problem der Substruktursuche, dem Teilgraph-Isomorphie-Problem. Hier werden in der Literatur beschriebene Lösungsansätze vorgestellt und darauf aufbauend ein neuer Algorithmus entwickelt. Dieser nutzt die speziellen Eigenschaften von Molekülgraphen zur Verkürzung der Laufzeit aus, wie am Ende des Kapitels in Experimenten gezeigt wird. In Kapitel 4 wird die der Verifikation vorgeschaltete Filterphase untersucht. Es werden bekannte Ansätze erläutert und ein neues Verfahren vorgestellt, das für die Substruktursuche in Scaffold Hunter implementiert wurde. Wiederum wird in einem experimentellen Vergleich das neue Verfahren mit bekannten Ansätzen verglichen. Die Arbeit endet mit einer Zusammenfassung und einem Ausblick in Kapitel 5.

Kapitel 2

Scaffold Hunter

Scaffold Hunter ist ein Tool zur Visualisierung chemischer Strukturen und ihrer Zusammenhänge. Die Software ist ein Kooperationsprojekt zwischen dem Max-Planck-Institut für molekulare Physiologie und dem Lehrstuhl für Algorithm Engineering der Technischen Universität Dortmund und wurde im Rahmen der Projektgruppe 504 im Wintersemester 2006/2007 und Sommersemester 2007 unter der Leitung von Karsten Klein und Stefan Wetzel begonnen [33]. Das Programm ist in Java implementiert und zusammen mit der am Max-Planck-Institut entwickelten Software *Scaffold Tree Generator* unter der GNU General Public License veröffentlicht worden[1].

2.1 Motivation

In diesem Abschnitt wird die Zielsetzung und Motivation des Programms basierend auf der Dissertation von Stefan Wetzel [67] zusammengefasst.

Als *chemischer Strukturraum* wird die Gesamtheit aller theoretisch möglichen organischen Moleküle bezeichnet. Die Größe des chemischen Strukturraums wird auf bis zu 10^{160} geschätzt. Nur einen geringen Teil des Strukturraums machen Moleküle aus, die als pharmazeutische Wirkstoffe in Frage kommen. Die Zahl möglicher Wirkstoffkandidaten wird auf 10^{60} geschätzt. Die Dimension dieser Zahlen macht deutlich, dass eine Suche nach neuen Wirkstoffen gezielt erfolgen muss und sich auf einen möglichst kleinen, relevanten Teil beschränken sollte. Ein Datensatz bestehend aus chemischen Strukturen wird als *Molekülbibliothek* bezeichnet. Eine Kernaufgabe der pharmazeutischen Forschung ist es, eine Auswahl von Strukturen zu Molekülbibliotheken mit hoher biologischer Relevanz zusammenzustellen. Diese können experimentell in Hochdurchsatzverfahren auf ihre Eignung als Wirkstoff getestet werden. In der größten Moleküldatenbank, Chemical Abstracts Service, sind bereits über 24 Millionen zyklischer Strukturen registriert, was allerdings nur einen kleinen Teil der möglichen Wirkstoffe ausmacht.

[1] Die Software steht unter `http://www.scaffoldhunter.com` zur Verfügung.

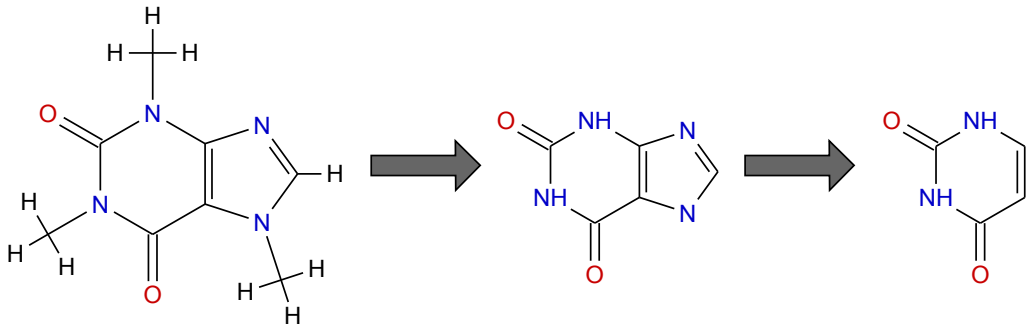


Abbildung 2.1: Einordnung von Koffein im Baumdiagramm: Das linke Bild zeigt das vollständige Molekül, das mittlere Bild das zugehörige Molekülgerüst und rechts ist das im Baum vorausgehende Molekülgerüst zu sehen.

Ein zentrales Problem ist also, relevante bislang unerforschte Teile des chemischen Strukturraums ausfindig zu machen. Bei dieser Aufgabe sind Erkenntnisse der Theoretischen Chemie und damit verbunden die Möglichkeiten der *Cheminformatik* von großem Nutzen. Das Zusammenführen experimenteller Daten, ihre computergestützte Analyse und die Verifikation daraus gewonnener Erkenntnisse in weiteren Experimenten erfordert eine enge Verbindung aller Disziplinen. Scaffold Hunter stellt ein benutzerfreundliches Tool dar, das helfen kann, die Kluft zwischen theoretischer und experimenteller Wissenschaft zu verringern. Auf der Grundlage eines Konzeptes zur hierarchischen, strukturbasierten Klassifikation von Substanzen visualisiert Scaffold Hunter die Zusammenhänge von Strukturen, erlaubt eine intuitive Navigation im Strukturraum und bietet die Möglichkeit, diesen mit experimentell ermittelten Eigenschaftswerten anzureichern.

2.2 Kartierung des chemischen Strukturraums

Als Grundlage für Scaffold Hunter dient ein am Max-Planck-Institut entwickeltes Konzept zur Kartierung des chemischen Strukturraums, das in [58] im Detail beschrieben ist. Jedes Molekül wird nach dem *Scaffold-Tree-Algorithmus* zunächst auf sein Molekülgerüst (*Scaffold*) reduziert, indem Seitenketten entfernt werden, bis das größte zusammenhängenden Ringsystem übrigbleibt. Ein Scaffold wird weiter zerlegt, indem iterativ jeweils eine Ringstruktur entfernt wird, bis es nur noch aus einem einzigen Ring besteht.

Abbildung 2.1 zeigt den Prozess für das Molekül von Koffein. Im ersten Schritt werden die Seitenketten mit Ausnahme der Doppelbindungen vom Molekül entfernt, so dass das Molekülgerüst entsteht. Zur weiteren Zerlegung muss einer der beiden Ringe entfernt werden. Die Wahl wird stets deterministisch anhand von Regeln der Chemie getroffen und hat zum Ziel, die "am wenigsten charakteristischen" Ringe zuerst zu entfernen. Ein Scaffold, das durch Entfernen eines Rings aus einem anderen Scaffold hervorgegangen ist, ist

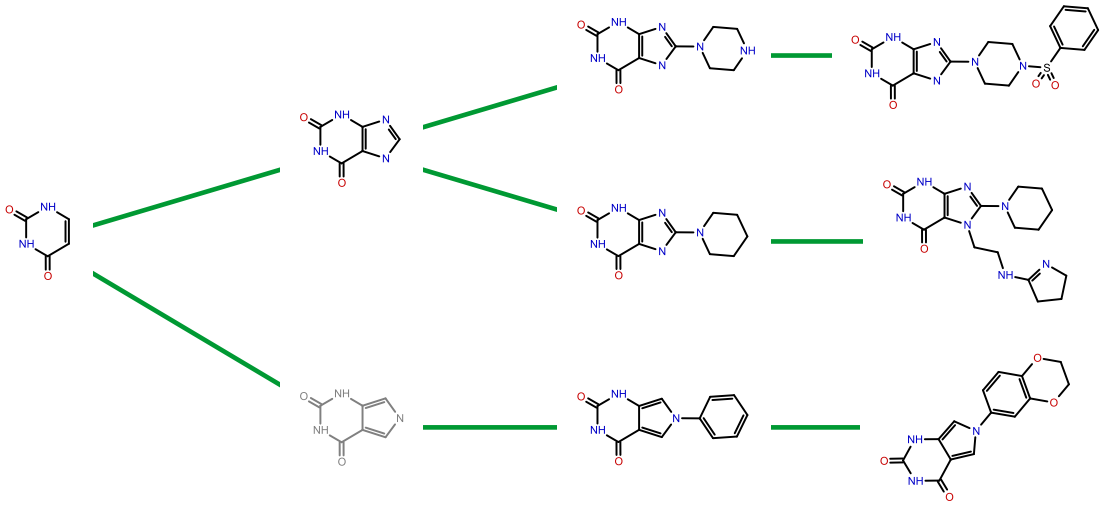
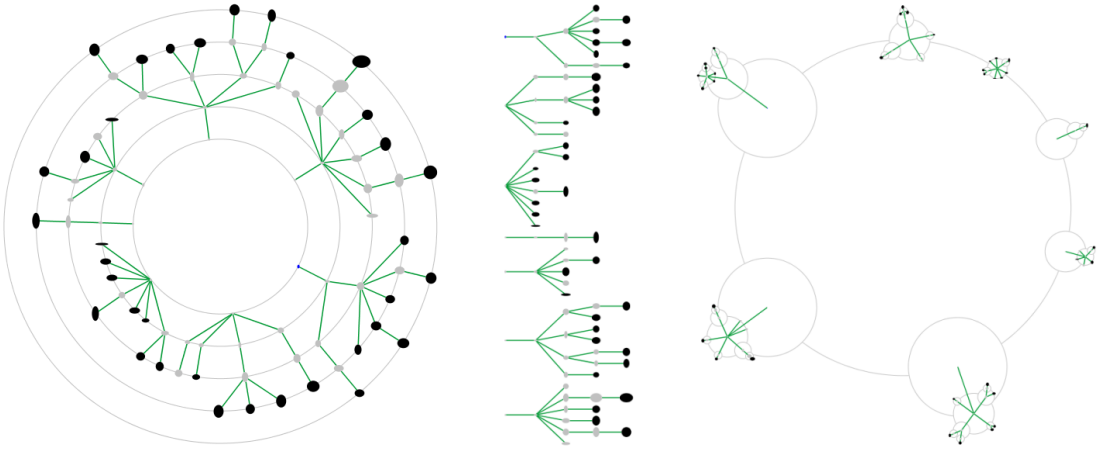


Abbildung 2.2: Das Molekülgerüst von Koffein im Baumdiagramm. Virtuelle Scaffolds sind ausgegraut.

sein direkter Vorgänger im Baumdiagramm. Auf diese Weise erhält man eine Vielzahl von Elter-Kind-Beziehungen, die zu einem Baum zusammengefasst werden können.

In Abbildung 2.2 ist das bekannte Molekülgerüst von Koffein im Baumdiagramm eingeordnet. Während Koffein sowie einige andere ähnliche Moleküle direkt mit diesem Scaffold assoziiert werden können, existieren im verwendeten Datensatz weitere Moleküle, deren Molekülgerüst das von Koffein vollständig enthält. Im Baum sind dies die Nachfolger des Koffein-Gerüstes. Charakteristisch ist, dass die Molekülgerüste mit jeder Ebene im Baum komplexer werden und einen zusätzlichen Ring aufweisen. Bei geeigneter Wahl eines Regelsystems zur Bestimmung der Vorgänger-Scaffolds bleiben bestimmte Eigenschaften entlang der Pfade im Baum erhalten, so dass die Zusammenhänge von chemischer Bedeutung sind. Bei der Erzeugung des Baums können Molekülgerüste entstehen, mit denen kein einziges Molekül assoziiert ist. Diese Scaffolds werden als virtuell bezeichnet und können interessante Ausgangspunkte für neue chemische Verbindungen sein, die von der zu Grunde liegenden Molekülbibliothek nicht abgedeckt werden [67].

Das Verfahren zur Erzeugung ist in der Software Scaffold Tree Generator implementiert und ermöglicht es, neben dem vorgegebenen Regelsatz, wie er in [58] beschrieben ist, neue Regeln hinzuzufügen und mit unterschiedlicher Priorität anzuwenden. Die Software ist in der Lage, Moleküldatensätze mit Eigenschaften im SDfile-Format einzulesen, daraus den Baum der Molekülgerüste zu erstellen und in einer Datenbank abzuspeichern. Aus den Eigenschaftswerten der Moleküle werden Eigenschaftswerte der Molekülgerüste abgeleitet, indem z.B. der Mittelwert oder das Minimum und Maximum berechnet werden.



(a) (b) (c)

Abbildung 2.3: Unterschiedliche Layouts des gleichen Baums: (a) Radial Layout, (b) Linear Layout, (c) Balloon Layout.

2.3 Visualisierung des Baumdiagramms

Das Programm Scaffold Hunter ermöglicht es, den Baum der Molekülgerüste – oder einen durch Filterregeln bestimmten Teile des Baums – aus der Datenbank zu laden und interaktiv zu visualisieren. Dazu kann das Baumdiagramm in verschiedenen Layouts betrachtet werden, Teilbäume können auf Wunsch jederzeit ein- oder ausgeklappt werden und es können die Eigenschaftswerte der Molekülgerüste visualisiert werden. Das Programm verwendet das Konzept des *Zoomable User Interface*, das es dem Nutzer erlaubt, das Baumdiagramm aus verschiedenen Zoomstufen in einer unterschiedlich detaillierten Darstellung zu betrachten und den angezeigten Ausschnitt zu verschieben (*Pan*). Die Steuerung kann dabei sowohl mit der Maus als auch mit der Tastatur erfolgen. Auf diese Weise wird eine einfache und intuitive Navigation im Baum ermöglicht, die das Erforschen von Ästen strukturell verwandter Molekülgerüste in besonderer Weise unterstützt.

Die automatische Berechnung eines übersichtlichen Layouts für das Baumdiagramm fällt in den Bereich des *Graphenzeichnens*. Unter einem Layout versteht man die geometrische Anordnung der Objekte des Diagramms nach ästhetischen Kriterien. Scaffold Hunter bietet drei verschiedene Layoutverfahren, die unterschiedliche Anwendungsfälle berücksichtigen und zwischen denen jeder Zeit gewechselt werden kann. In Abbildung 2.3 ist ein Baum in unterschiedlichen Layouts zu sehen. Das Radial Layout ähnelt der manuell erstellten Vorgabe aus [42] und ist leicht zugänglich. Das Balloon Layout trennt einzelne Äste klarer voneinander und erlaubt es, diese mit allen Hierarchieebenen zu betrachten. Das Linear Layout ist besonders geeignet, wenn die Molekülgerüste nach einem bestimmten Kriterium (wie Ähnlichkeit oder Eigenschaftswerten) sortiert dargestellt werden. Die Animation von Layoutänderungen beim Ein- und Ausklappen von Ästen machen diese

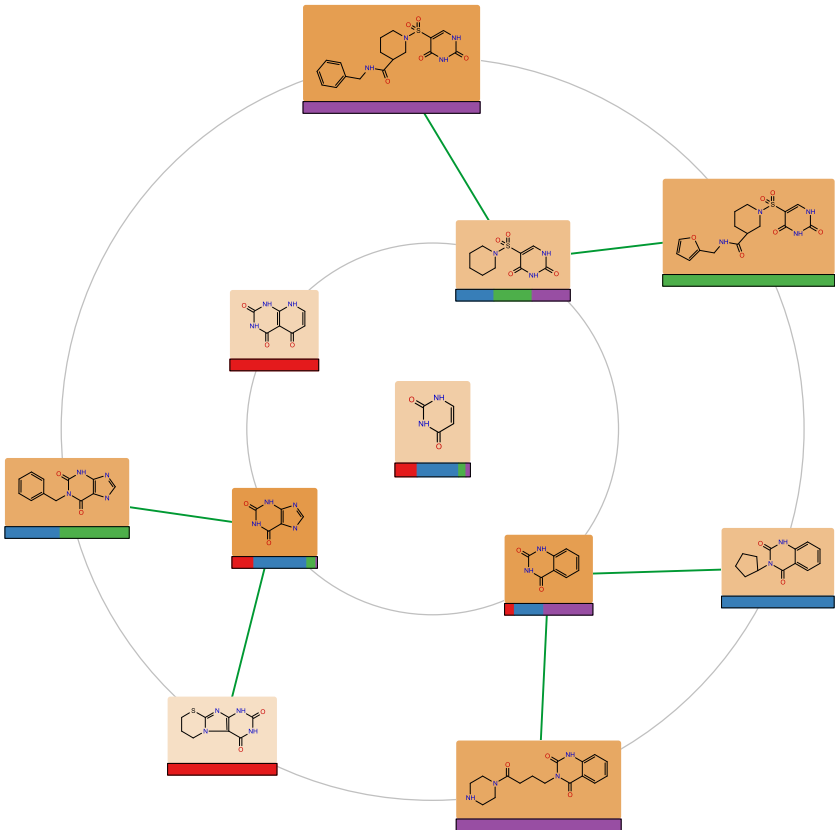


Abbildung 2.4: Einfärbung des Koffein-Astes nach Eigenschaftswerten in Kombination mit Property-Bins.

leicht nachvollziehbar. Die Orientierung und Navigation im Diagramm wird durch eine Übersichtskarte des gesamten Baumdiagramms unterstützt und Änderungen des sichtbaren Ausschnitts können ebenfalls animiert werden.

Die Darstellung zugehöriger Eigenschaftswerte kann auf unterschiedliche Weise erfolgen: Zunächst kann eine Auswahl der in der Datenbank hinterlegten Daten in einem Tooltip-Fenster angezeigt werden. Um Eigenschaftswerte intuitiv und schnell erfassbar zu machen, können Molekülgerüste im Diagramm farbig hervorgehoben werden. Die Stärke der Einfärbung kann sich dabei nach dem Wert einer Eigenschaft richten. Abbildung 2.4 zeigt diese Funktion angewandt auf einen kleinen Teilbaum. Ferner bietet Scaffold Hunter die Möglichkeit, die Zusammensetzung der Moleküle hinter den Molekülgerüsten zu visualisieren. Dazu können Wertebereiche zu Property-Bins zusammengefasst werden. Die Aufteilung der Moleküle eines Gerüstes in die Property-Bins wird dann graphisch dargestellt, wie ebenfalls in Abbildung 2.4 zu sehen ist.

Eigenschaften werden weiterhin verwendet, um die Größe des Baumdiagramms zu beschränken. Mit Hilfe eines Filterdialogs kann das Diagramm auf solche Molekülgerüste beschränkt werden, deren Eigenschaftswerte bestimmten Anforderungen genügen. Scaffold Hunter generiert dynamisch einen zusammenhängenden Baum aus den Molekülgerüsten,

die den Filter passieren. Auf diese Weise kann der Nutzer von vornherein uninteressante Teile des Baumdiagramms ausblenden und sich auf relevante Bereiche konzentrieren. Eine iterative Wiederholung des Filterprozess wird genau so ermöglicht, wie die manuelle Auswahl bestimmter Molekülgerüste zur Darstellung in einer neuen Ansicht. Dazu werden Tabs verwendet, in denen ausgewählte oder vollständige Teilbäume gesondert angezeigt werden können.

Scaffold Hunter bietet zahlreiche Funktionen, um die praktische Arbeit mit der Software zu erleichtern. Dazu zählt ein Lesezeichensystem, mit dem bekannte Molekülgerüste schnell im Baumdiagramm lokalisiert werden können, und die Möglichkeit, Daten in verschiedenen Formaten (SVG, PNG, Listen von SMILES[2]) zu exportieren.

2.3.1 Technische Umsetzung

Scaffold Hunter setzt frei verfügbare Software ein, um die beschriebene Funktionalität zu realisieren. Zur Visualisierung wird das Toolkit Piccolo [9] verwendet, das die Funktionalität einer skalierbaren Benutzeroberfläche zur Verfügung stellt. Zur Darstellung von Molekülen und Molekülgerüsten sowie dem Export von Grafiken werden skalierbare Vektorgrafiken (SVG) verwendet, die im Gegensatz zu Formaten für Rastergrafik eine qualitativ hochwertige Darstellung in jeder Zoomstufe ermöglichen. Hierbei wird auf die Bibliothek Batik[3] zurückgegriffen.

Als Datenbank kommt ein MySQL Server zum Einsatz, der sämtliche von Scaffold Tree Generator erzeugten Daten bereitstellt. Scaffold Hunter fungiert als Client und ruft nur die erforderlichen Daten vom Server ab, die visualisiert werden sollen. Eine MySQL Datenbank kann zentral installiert sein und als Datenquelle für eine Vielzahl von Clients dienen oder lokal genutzt werden.

2.4 Integration der Substruktursuche

Wetzel erwähnt in [67] die Integration eines Systems zur Substruktursuche als eine von vielen Nutzern gewünschte Erweiterung von Scaffold Hunter. Diese würde die Nutzbarkeit des Programms zur Erforschung des chemischen Strukturraums erhöhen, wobei zwei Anwendungsfälle hervorgehoben werden: Die Suche von Molekülgerüsten, die eine gegebene Substruktur enthalten, und die Suche in der Gesamtheit der darunterliegenden Moleküle. Substrukturen in Molekülgerüsten beeinflussen die pharmakologische Wirkung, so dass mit Hilfe der Substruktursuche Scaffolds von Interesse auf intuitive Weise gefunden und angesteuert werden können. Die Suche in den darunterliegenden Molekülen erlaubt es, nach

[2] *Simplified Molecular Input Line Entry Specification* - eine verbreitete Zeichenketten-Notation für Moleküle.

[3] `http://xmlgraphics.apache.org/batik/`

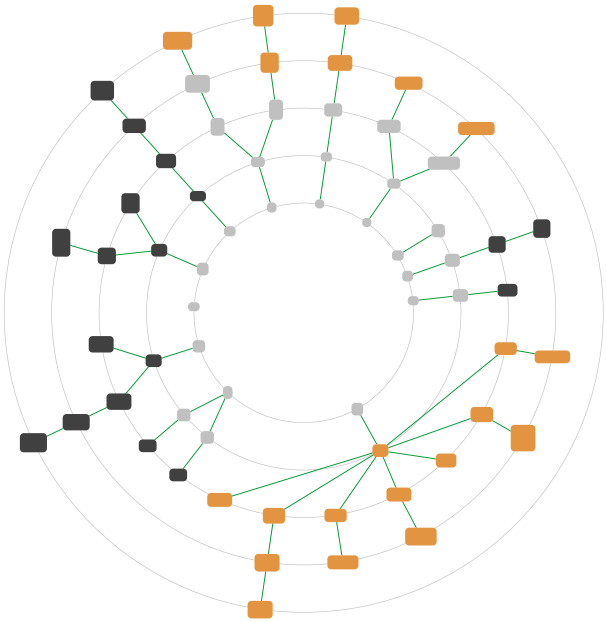
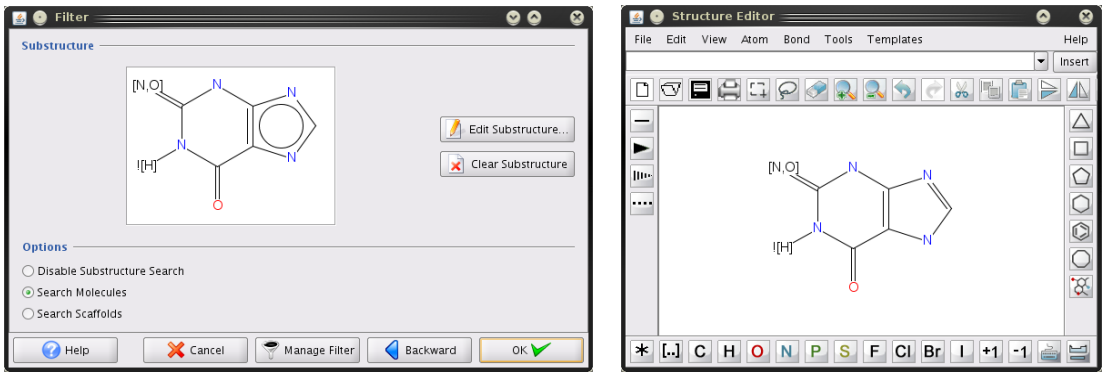


Abbildung 2.5: Alle Molekülgerüste, die das Koffein-Gerüst enthalten, wurden eingefärbt. Das Koffein-Gerüst selbst ist die Wurzel des vollständig eingefärbten Unterbaums. Die gleiche Struktur wurde jedoch auch noch in anderen Ästen gefunden.

Strukturmotiven zu suchen, die im Gegensatz zu Molekülgerüsten Seitenketten beinhalten. Auch diese können die pharmakologische Wirkung entscheidend beeinflussen.

Die Substruktursuche ist so integriert worden, dass die bereits bestehenden Mechanismen sinnvoll ergänzt werden. Dadurch können sich Benutzer, die bereits mit dem Programm vertraut sind, schnell zurechtfinden und eine insgesamt konsistente Bedienung bleibt gewährleistet. Das in Abbildung 2.6(a) zu sehende Eingabefeld wurde dem Filterdialog hinzugefügt. Auf diese Weise kann das visualisierte Diagramm auf Molekülgerüste begrenzt werden, welche das angegebene Suchmuster enthalten. Bei der Suche in den darunterliegenden Molekülen werden nur Gerüste zugelassen, die mindestens ein Molekül aufweisen, welches das Suchmuster enthält. Filterregeln bezüglich numerischer Eigenschaften können mit der Substruktursuche kombiniert werden.

Zusätzlich dient die Substruktursuche als Ergänzung der bestehenden Einfärbefunktion: Anstatt nach Eigenschaften zu filtern und die Molekülgerüste hervorzuheben, die den Filterregeln entsprechen (siehe Abbildung 2.4), werden die Molekülgerüste hervorgehoben, die das vom Nutzer vorgegebene Suchmuster aufweisen. Die farbig hervorgehobenen Molekülgerüste können so leicht bereits in einer niedrigen Zoomstufe erkannt werden, Häufungen in bestimmten Ästen sind leicht auszumachen und die gefundenen Moleküle können zur genaueren Untersuchung gesondert in einer neuen Ansicht geöffnet werden. Abbildung 2.5 zeigt ein Baumdiagramm, in dem alle Scaffolds eingefärbt wurden, die das Koffein-Gerüst



(a) Dialogfenster zur Substruktursuche (b) Der Strukturediteor JChemPaint

Abbildung 2.6: Eingabe eines Suchmusters.

enthalten. Dieses findet sich nicht nur in dem zugehörigen Ast des Baumdiagramms, sondern außerdem an anderen Stellen, die auf diese Weise schnell gefunden werden können.

2.4.1 Strukturediteor

Um dem Benutzer eine komfortable Möglichkeit zur Eingabe eines Suchmuster zu bieten, wurde ein Strukturediteor in Scaffold Hunter integriert. Strukturediteoren erlauben es, auf einfache Art und Weise 2D-Strukturformeln zu editieren. Es stehen verschiedene Atom- und Verbindungstypen sowie vorgefertigte Motive wie Ringe zur Verfügung, aus denen eine Struktur zusammengesetzt werden kann. Ein solches Tool hat zahlreiche Anwendungen in der Chemie und es existiert eine Vielzahl von Software für diese Zwecke [31]. Unter einer Open-Source-Lizenz stehen hingegen die wenigsten verfügbaren Strukturediteoren. Einer davon ist JChempaint [43], der für die Integration in Scaffold Hunter ausgewählt wurde. Die leichte Bedienbarkeit und die aktive Weiterentwicklung sprechen für diese Software. Hinzu kommt, dass JChemPaint auf die Java-Bibliothek CDK (Chemistry Development Kit) [60, 61] aufbaut, die bereits von Scaffold Tree Generator verwendet wird.

Der Editor JChemPaint ist in Abbildung 2.6(b) zu sehen. Neben den üblichen Elementen, wie Ringen unterschiedlicher Größe und verschiedenen Atomtypen, können größere Strukturen aus einer Vorlagen-Bibliothek geladen werden. Darüber hinaus werden zahlreiche Formate zum Laden vorgefertigter Moleküle unterstützt, die anschließend weiter editiert werden können. Dazu zählen unter anderem die weit verbreiteten Formate MDL Molfile und SMILES. Das integrierte automatische Layoutverfahren erzeugt eine Strukturformel nach den in der Chemie üblichen Konventionen.

Optionen wie Wildcards, die für Suchmuster von Interesse sind, werden in der aktuellen Entwicklungsversion (November 2009) nur unzureichend unterstützt. Eine Erweiterung um diese Funktionalität ist jedoch für zukünftige Versionen geplant. Im Rahmen der Diplomarbeit wurde daher die letzte verfügbare JChemPaint Version dahingehend angepasst,

dass Wildcards für beliebige Atome und Bindungen sowie Atomlisten und Negativ-Listen von Atomen, die nicht an der angegebenen Stellen auftreten dürfen, unterstützt werden. Dabei steht das Zeichen `*` für beliebige Elemente, `[`$s_1, s_2, ..., s_n$`]` für eine Atomliste und `!`[$s_1, s_2, ..., s_n$]` für eine Negativ-Liste, wobei $s_i$ die erlaubten bzw. ausgeschlossenen Symbole sind. Abbildung 2.6 zeigt ein Beispiel, bei dem die Wildcards `[N,O]` und `![H]` verwendet wurden. Außerdem ist es möglich, unzusammenhängende Suchmuster anzugeben, wobei alle Komponenten im Molekül auftreten müssen, dort aber auf beliebige Art und Weise verbunden sein können.

Kapitel 3

Das Teilgraph-Isomorphie-Problem

Dieses Kapitel befasst sich mit dem Teilgraph-Isomorphie-Problem, das in Abschnitt 3.1 formal definiert wird. Im Abschnitt 3.2 werden bekannte Ansätze vorgestellt und darauf aufbauend in Abschnitt 3.3 ein neuer Algorithmus entwickelt, der in Scaffold Hunter implementiert wurde. Abschnitt 3.4 schließt das Kapitel mit einem experimentellen Vergleich. Einige Details zur Implementierung sind im Anhang zu finden.

3.1 Theoretische Grundlagen

Mit dem Nachweisen von Teilgraph-Isomorphie ist ein NP-vollständiges Entscheidungsproblem integraler Bestandteil der Substruktursuche. Für diese Klasse von Problemen sind keine Algorithmen mit polynomieller Laufzeit bekannt. In diesem Abschnitt wird das Problem formal definiert, verwandte Problemstellungen werden vorgestellt und komplexitätstheoretisch eingeordnet.

3.1.1 Beziehungen zwischen Graphen

In diesem Abschnitt wird die bis jetzt verwendete unscharfe Formulierung, dass ein Graph in einem anderen "enthalten" ist, formal gefasst. Die Begriffe *Teilgraph* und *induzierter Teilgraph* beschreiben eine unmittelbare Beziehung zwischen zwei Graphen.

Definition 3.1 (Teilgraph [21]). Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$, geschrieben $G' \subseteq G$. G wird dann als *Obergraph* von G' bezeichnet.

Definition 3.2 (Induzierter Teilgraph [21]). Ein Teilgraph $G' = (V', E')$ von $G = (V, E)$ wird als *induziert* (von V' in G) bezeichnet, wenn für alle $u, v \in V'$ gilt $(u, v) \in E \Rightarrow (u, v) \in E'$.

Sind zwei Graphen gegeben, so liefern strukturerhaltende Abbildungen von einem Graphen auf den anderen eine Möglichkeit, diese zu vergleichen und eine Beziehung zwischen

ihren Knoten und Kanten herzustellen. Die folgenden Definitionen beziehen sich zunächst nur auf gerichtete und ungerichtete Graphen und werden später auf gelabelte Graphen erweitert.

Definition 3.3 (Graph-Isomorphismus (GI) [22]). Für zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißt eine bijektive Abbildung $\varphi : V_1 \to V_2$ *Graph-Isomorphismus*, wenn für alle $u, v \in V_1$ gilt $(u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$. Existiert ein Graph-Isomorphismus zwischen G_1 und G_2, so werden diese als *isomorph* bezeichnet, geschrieben $G_1 \simeq G_2$. Ist $G_1 = G_2$, so nennt man φ einen *Automorphismus* von G_1.

Für die nach 1.1 und 1.2 definierten Graphen ergibt sich aus der Abbildung $\varphi : V_1 \to V_2$ der Knotenmengen direkt die Abbildung der Kantenmengen $\phi : E_1 \to E_2$ zu $\phi((u, v)) = (\varphi(u), \varphi(v))$. Sind zwei Graphen isomorph, weisen sie die gleiche Struktur auf und sind in diesem Sinne identisch.

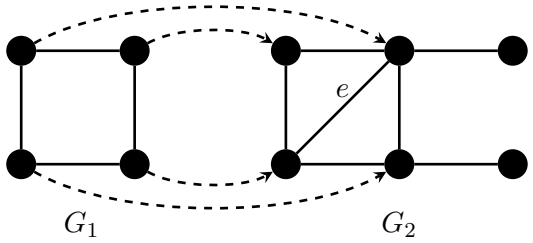
Der Vergleich von Graphen mittels Abbildungen lässt sich mit dem Konzept von Teilgraphen kombinieren. Von einem *(induzierten) Teilgraph-Isomorphismus* $\varphi : V_1 \to V_2$ von G_1 auf G_2 spricht man, wenn φ ein Graph-Isomorphismus zwischen G_1 und einem (induzierten) Teilgraphen von G_2 ist. Die folgenden Definitionen vereinfachen diese Beziehung.

Definition 3.4 (Teilgraph-Isomorphismus (TGI) [74]). Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei Graphen. Eine injektive Abbildung $\varphi : V_1 \to V_2$ heißt *Teilgraph-Isomorphismus* von G_1 auf G_2, wenn für alle $u, v \in V_1$ gilt $(u, v) \in E_1 \Rightarrow (\varphi(u), \varphi(v)) \in E_2$. Gibt es einen Teilgraph-Isomorphismus von G_1 auf G_2, wird geschrieben $G_1 \lesssim G_2$ und $G_1 \lnsim G_2$ sonst.

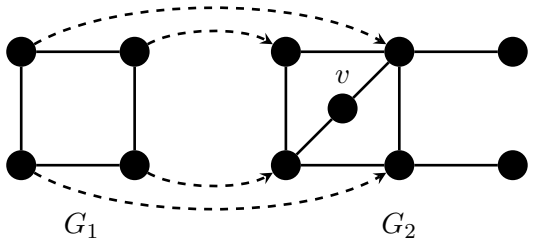
Eine andere in der Literatur verbreitete Bezeichnung für TGI ist *Graph Monomorphismus* [16]. Für einen induzierten Teilgraph-Isomorphismus muss die Bedingung an die Abbildung verschärft werden.

Definition 3.5 (Induzierter-Teilgraph-Isomorphismus (ITGI) [74]). Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei Graphen. Eine injektive Abbildung $\varphi : V_1 \to V_2$ heißt *induzierter Teilgraph-Isomorphismus* von G_1 auf G_2, wenn für alle $u, v \in V_1$ gilt $(u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$.

Abbildung 3.1 verdeutlicht den Unterschied zwischen TGI und ITGI. Bild (a) zeigt einen möglichen TGI von G_1 auf G_2. Um einen ITGI handelt es sich bei der durch die gestrichelten Linien angedeuteten Abbildung φ nicht, da der durch die vier betroffenen Knoten in G_2 induzierte Teilgraph die Kante e enthält. Eine entsprechende Kante in G_1 existiert nicht. Auf Bild (b) ist die Kante e durch den Knoten v unterteilt und der induzierte Teilgraph enthält weder v noch die zu v inzidenten Kanten. Demzufolge ist die angedeutete Abbildung hier ein ITGI und – da jeder ITGI auch ein TGI ist – gleichzeitig ein TGI.



(a) Teilgraph-Isomorphismus



(b) (Induzierter-)Teilgraph-Isomorphismus

Abbildung 3.1: Bild (a) zeigt einen TGI von G_1 auf G_2, der kein ITGI ist. Bild (b) zeigt ein Beispiel für einen ITGI. Die gestrichelten Pfeile entsprechen der Abbildung φ.

In der Literatur werden TGI und ITGI nicht streng voneinander getrennt, sondern oft unterschiedliche Definitionen unter der Bezeichnung *subgraph isomorphism* verstanden. Welche Definition verwendet wird, unterscheidet sich je nach Anwendungsgebiet. Während im Bereich der Mustererkennung ITGI die verbreitetere Variante zu sein scheint, wird in der Cheminformatik TGI verwendet [6]. Im Folgenden werden Algorithmen aus unterschiedlichen Disziplinen vorgestellt, weshalb beide Varianten hier klar voneinander getrennt werden.

Die vorgestellten Definitionen lassen sich auf gelabelte Graphen erweitern, indem für die Abbildung der Knotenmenge $\varphi : V_1 \rightarrow V_2$ und der zugehörigen Abbildung der Kantenmenge $\phi : E_1 \rightarrow E_2$ folgende Bedingungen zusätzlich gefordert werden:

$$\forall v \in V_1 \quad l(v) = l(\varphi(v)) \tag{3.1}$$

$$\forall e \in E_1 \quad l(e) = l(\phi(e)) \tag{3.2}$$

Dies bedeutet für Molekülgraphen, dass nur identische Atom- und Bindungstypen aufeinander abgebildet werden dürfen. Bei der Substruktursuche soll es erlaubt sein, Wildcards im Suchmuster zu verwenden. Diese können als spezielle Label des Suchmusters G_1 gesehen werden, die mehreren unterschiedlichen Labeln des Graphen G_2 entsprechen. Zur Vereinfachung der Notation wird im Folgenden dennoch die Gleichheit der Label als Bedingung verwendet. Die Übertragbarkeit ist für Teilgraph-Isomorphie-Algorithmen in der Praxis problemlos möglich und wird für Graph Indizes in Abschnitt 4.3 gesondert behandelt.

3.1.2 Verwandte Probleme und Anwendungsgebiete

Die vorgestellten Konzepte werden häufig zusammen mit weiteren verwandten Problemen allgemein als *Graph Matching* bezeichnet. Eine Verallgemeinerung des TGI-Problems ist das Maximum Common Subgraph Problem, das kurz in Abschnitt 3.2.1 vorgestellt wird. Nach Conte et al. [16] lassen sich diese Probleme in die Kategorie des *Exakten Matchings* einordnen, da eine exakte Übereinstimmung der Struktur zwischen einem Graphen und einem anderen Graphen (bzw. eines Teilgraphen davon) gefordert wird. Unter *Inexaktes Matching* fallen Probleme, bei denen unterschiedliche Objekte aufeinander abgebildet werden dürfen. Dabei können beispielsweise Strafkosten für solche Abbildungen angegeben oder eine Editierdistanz für Graphen definiert werden. Das Ziel ist es dann, eine Abbildung mit minimalen Kosten zu finden. Ein solches Optimierungsproblem kann optimal oder approximativ gelöst werden. Exakte und inexakte Methoden finden eine Anwendung in der Mustererkennung, der Analyse von Bildern, der Identifikation biometrischer Daten oder der Erkennung von Schriftzeichen [16]. Weitere Anwendungen sind das Testen integrierter Schaltungen oder die Analyse Sozialer Netzwerke.

Im Bereich der Chemie findet das TGI-Problem eine direkte Anwendung bei der Substruktursuche in Moleküldatenbanken. Die Suche einer vollständigen Struktur in einer Moleküldatenbank entspricht dem GI-Problem. Ein eng verwandtes Problem ist das Finden des *kanonischen Bezeichners* für einen Graphen. Ein kanonischer Bezeichner ist eine Zeichenkette, die einen Graphen eindeutig repräsentiert. Zwei Graphen sind genau dann isomorph, wenn ihre kanonischen Bezeichner identisch sind. Auf diese Weise können Moleküle durch eine eindeutige Identifikationsbezeichnung in einer Datenbank registriert werden. Die Struktursuche reduziert sich dann auf das Generieren der Identifikationsbezeichnung der gesuchten Struktur und der Suche dieser Zeichenkette in der Datenbank. Die Suche einer Zeichenkette kann z.B. mittels Hashing oder Suchbäumen effizient erledigt werden. Die Substruktursuche gestaltet sich hingegen vergleichsweise aufwendig.

3.1.3 Komplexitätstheoretische Einordnung

Die Fragestellung, ob für zwei gegebenen Graphen ein GI/TGI/ITGI existiert, wird als GI/TGI/ITGI-Problem bezeichnet. Diese Probleme sind nicht nur in der Praxis von großer Bedeutung, sondern auch aus theoretischer Sicht von besonderem Interesse. In diesem Abschnitt wird eine kurze komplexitätstheoretische Einordnung der Probleme vorgenommen.

Für das GI-Problem ist weder ein Polynomialzeitalgorithmus bekannt noch ein Beweis der NP-Vollständigkeit. Für spezielle Klassen von Graphen kann das Problem jedoch in Polynomialzeit gelöst werden. Dazu zählen unter anderem planare Graphen und Graphen mit beschränktem Grad. Molekülgraphen sind bis auf wenige Ausnahmen planar und haben stets beschränkten Grad, der sich auf die Wertigkeit chemischer Elemente zurückführen lässt. Darauf aufbauend zeigt Faulon [26], dass GI und weitere verwandte Probleme wie

das Finden kanonischer Bezeichner für Molekülgraphen in Polynomialzeit gelöst werden können. In der Praxis wird meist ein Verfahren verwendet, das auf dem Algorithmus von Morgan aufbaut, der jedoch für einige Molekülgraphen uneindeutige Ergebnisse liefert [31].

Das TGI-Problem ist hingegen NP-vollständig und kann als Verallgemeinerung zahlreicher wichtiger graphentheoretischer Probleme wie dem Finden eines Hamiltonkreises oder einer Clique gesehen werden. Polynomialzeitalgorithmen sind nur für stark eingeschränkte Klassen von Graphen bekannt. Matousek et al. [48] untersuchen u.a. das TGI-Problem auf Graphen mit beschränkter Baumweite. Hat G_2 Baumweite k, so ist das TGI-Problem in Polynomialzeit lösbar, wenn zusätzlich der Grad von G_1 beschränkt ist oder G_1 k-fach zusammenhängend ist. Eppstein beschreibt einen Linearzeitalgorithmus für Suchmuster konstanter Größe in planaren Graphen [25]. Beispiele für die Anwendbarkeit dieser Erkenntnisse in der Praxis konnten in der Literatur nicht gefunden werden und die Algorithmen scheinen kaum für eine effiziente Substruktursuche geeignet zu sein[1].

3.2 Bekannte Ansätze

In der Literatur sind zahlreiche Algorithmen zur Lösung des TGI-Problems beschrieben. Erste Algorithmen für die Substruktursuche wurden bereits in den 60er und 70er Jahren von Sussenguth [62] und Figueras [29] entwickelt. Beide Ansätze partitionieren die Knotenmengen der Graphen zunächst nach bestimmten Eigenschaften wie dem Atomtyp, so dass Mengen von Knoten entstehen, die möglicherweise aufeinander abgebildet werden. Diese Partitionen werden in späteren Schritten durch weitere Unterteilung verfeinert, wozu die Nachbarschaftsbeziehung der Knoten betrachtet wird. In empirischen Untersuchungen zur Substruktursuche haben sich diese frühen Ansätze im Vergleich zu dem direkten Backtracking-Algorithmus von Ullmann (siehe Abschnitt 3.2.2) als weniger gut geeignet erwiesen [6, 69]. Im Folgenden wird daher der Algorithmus von Ullmann zusammen mit aktuelleren Ansätzen vorgestellt, die zum Teil für anderen Anwendungsgebiete entwickelt worden sind.

Besonders berücksichtigt wurden Verfahren, die in quelloffenen Cheminformatik Toolkits zum Einsatz kommen und sich dadurch als praxistauglich erwiesen haben. In der Bibliothek CDK [60, 61] ist das in Abschnitt 3.2.1 vorgestellte Verfahren implementiert. Der Algorithmus von Ullmann (siehe Abschnitt 3.2.2) wird von den Toolkits CDL [63] und RDKit [44] verwendet. Das Toolkit Frowns [41] verwendet den in Abschnitt 3.2.3 vorgestellten VF2-Algorithmus. Die Bibliothek OpenBabel [1] benutzt einen Backtracking-Algorithmus, der sich nicht direkt einem in der Literatur beschriebenen Verfahren zuordnen lässt.

[1] Persönliche Verständigung mit David Eppstein.

3.2.1 Transformation auf das Maximum Common Subgraph Problem

Eine Verallgemeinerung des ITGI-Problems ist das *Maximum Common Subgraph* Problem.

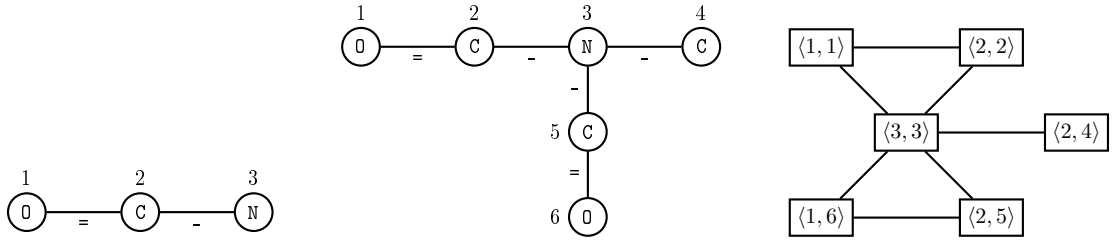
Definition 3.6 (Maximum Common Subgraph (MCS)[17]). Ein Graph $G = (V, E)$ heißt *common subgraph* von G_1 und G_2, wenn es einen ITGI[2] von G auf G_1 und von G auf G_2 gibt. G heißt *maximum common subgraph*, wenn es keinen *common subgraph* $G' = (V', E')$ mit $|V'| > |V|$ gibt.

Ein Algorithmus für das MCS-Problem kann verwendet werden, um das ITGI-Problem zu lösen, indem geprüft wird, ob der größte gemeinsame induzierte Teilgraph mit G_1 übereinstimmt. MCS hat zudem zahlreiche Anwendungen in der Chemie und anderen Disziplinen. Willet et al. [54] erläutert verschiedene Varianten, ihre Zusammenhänge und verwendete Algorithmen im Detail. Hier soll nur das grundlegende Prinzip eines bekannten Verfahrens beschrieben werden.

Ein häufig verwendeter Ansatz geht auf Levi [45] und Barrow [7] zurück und wird auch zur Lösung des ITGI-Problems vorgeschlagen. Das Verfahren generiert in einem ersten Schritt einen *Assoziationsgraphen*. Für die Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ist der Assoziationsgraph $G_A = (V_A, E_A)$ der ungerichtete Graph mit der Knotenmenge $V_A = V_1 \times V_2$, bei dem zwei Knoten $\langle u_1, u_2 \rangle$ und $\langle v_1, v_2 \rangle$ genau dann durch eine Kante verbunden sind, wenn entweder $(u_1, v_1) \in E_1$ und $(u_2, v_2) \in E_2$ oder $(u_1, v_1) \notin E_1$ und $(u_2, v_2) \notin E_2$. Ein Knoten $\langle u_1, u_2 \rangle \in V_A$ korrespondiert mit der Abbildung des Knotens $u_1 \in V_1$ auf $u_2 \in V_2$. Die Konstruktion der Kantenmenge sichert, dass im Assoziationsgraphen nur solche Knoten miteinander verbunden werden, deren zugehörige Knoten in V_1 und V_2 in gleicher Beziehung zueinander stehen, d. h. entweder beide über eine Kante miteinander verbunden sind oder beide nicht verbunden sind. Eine Clique in einem ungerichteten Graphen bezeichnet eine Teilmenge der Knoten, in der jeder Knoten mit jedem anderen über eine Kante verbunden ist. Es existiert ein direkter Zusammenhang zwischen den gemeinsamen induzierten Teilgraphen von G_1 und G_2 und den Cliquen des Assoziationgraphen: Die Knoten der Clique liefern die Paare von Knoten aus V_1 und V_2, die aufeinander abgebildet werden können. Diese Abbildung ist ein GI der durch die Knoten in G_1 bzw. G_2 induzierten Teilgraphen. Ein Clique maximaler Größe korrespondiert also mit einem größten gemeinsamen induzierten Teilgraphen.

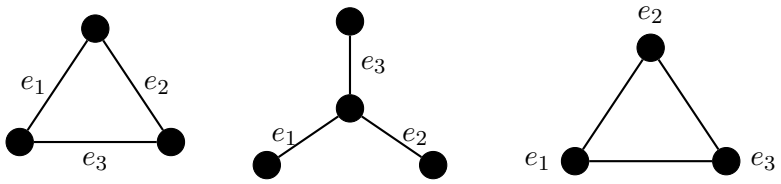
Das Verfahren lässt sich für gelabelte Graphen anpassen, indem nur dann Knoten im Assoziationsgraphen generiert werden, wenn die zugehörigen Knoten in beiden Graphen kompatible Label haben. Kantenlabel können beim Erstellen der Kantenmenge E_A be-

[2] Auch hier ist die Definition in der Literatur uneinheitlich und es wird gelegentlich ein TGI gefordert. Valiente unterscheidet *Maximum Common Subgraph Isomorphism* von *Maximum Common Induced Subgraph Isomorphism* [66]. Willet et al. [54] differenzieren zwischen Maximum Common Induced Subgraph und *Maximum Common Edge Subgraph*.



(a) G_1 (b) G_2 (c) Assoziationsgraph

Abbildung 3.2: Zwei Graphen und der zugehörige Assoziationsgraph.



(a) G (b) G' (c) $L(G) = L(G')$

Abbildung 3.3: Die beiden unterschiedlichen Graphen (a) und (b) haben den gleichen Line Graphen (c) (Beispiel übernommen aus [54]).

rücksichtigt werden. Abbildung 3.2 zeigt zwei Graphen und den zugehörigen Assoziationsgraphen. Es sind Cliquen der Größe drei zu finden: Die Knoten $\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle$ und $\langle 1,6\rangle, \langle 2,5\rangle, \langle 3,3\rangle$ sind untereinander paarweise durch eine Kante verbunden. Da die Größe der Clique der Anzahl Knoten von G_1 entspricht, liegt also in beiden Fällen ein ITGI vor. Kleinere Cliquen wie $\{\langle 3,3\rangle, \langle 2,4\rangle\}$ entsprechen kleineren gemeinsamen induzierten Teilgraphen. Das ITGI-Problem kann also auf das Finden einer größtmöglichen Clique reduziert werden, was als *Maximum-Clique*-Problem bezeichnet wird und wiederum NP-vollständig ist.

Tonnelier et al. [64] haben das Verfahren angepasst, um gemeinsame Substrukturen in Molekülen und Reaktionsgraphen zu suchen. Dazu wird nicht der Molekülgraph direkt verwendet, sondern der zugehörige *Bond Graph* (in der Graphentheorie üblicherweise als *Line Graph* bezeichnet). In diesem werden die Kanten des Molekülgraphens durch Knoten mit Labeln der Form "Atomtyp;Bindungstyp;Atomtyp" repräsentiert und zwei Knoten sind adjazent, wenn sich die durch sie repräsentierten Kanten im Molekülgraphen berühren. Zusätzliche Bedingungen bei der Konstruktion des Assoziationgraphen (dort *Compatibility Graph* genannt) sichern, dass das Ergebnis der Berechnung vom Line Graphen wieder auf den Molekülgraphen übertragen werden kann. Zum Finden von Cliquen wird ein Backtracking-Algorithmus verwendet.

Ein Problem bei der Berechnung auf den Line Graphen stellen die in Abbildung 3.3 zu sehenden unterschiedlichen Strukturen dar, die denselben Line Graphen aufweisen. Tritt

dieser Fall nicht auf, so entspricht ein auf den Line Graphen berechneter ITGI einem TGI im Ursprungsgraphen [54]. Das Verfahren von Tonnelier et al. liefert also die in der Cheminformatik gewünschte TGI-Variante.

Der Algorithmus ist in der Bibliothek CDK implementiert und wird dort auch zur Suche von Substrukturen genutzt. Der Fall, dass zwei unterschiedliche Graphen identische Line Graphen aufweisen, wird nicht behandelt, so dass die beiden Graphen 3.3(a) und 3.3(b) als isomorph erkannt werden.

3.2.2 Algorithmus von Ullmann

Der Algorithmus von J. R. Ullmann (1974) [65] ist bis heute einer der am häufigsten verwendeten Algorithmen für das Teilgraph-Isomorphie-Problem [52] und hat sich für die Substruktursuche in Untersuchungen als besonders geeignet erwiesen [6, 69]. Der Algorithmus basiert auf Backtracking und formuliert das TGI-Problem mit Hilfe von Matrizen. Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei Graphen mit den zugehörigen Adjazenzmatrizen $A = [a_{i,j}]$ und $B = [b_{i,j}]$. Eine $|V_1| \times |V_2|$-Matrix mit Elementen aus $\{0, 1\}$, bei der jede Zeile genau eine 1 enthält und jede Spalte höchstens eine 1, kann als injektive Abbildung von V_1 auf V_2 aufgefasst werden. Ullmann definiert eine Matrix $C = [c_{ij}] = M(MA)^T$, wobei T die Transposition bezeichnet. Die durch M definierte Abbildung ist genau dann ein TGI, wenn gilt:

$$\forall i, j \in \{0, ..., |V_1| - 1\} : a_{i,j} = 1 \Rightarrow c_{i,j} = 1 \tag{3.3}$$

Der Backtracking-Algorithmus arbeitet auf der Matrix M. Initial wird ein Element der Matrix $m_{i,j}$ auf 1 gesetzt, wenn der Grad des i-ten Knotens in G_1 nicht größer ist als der Grad des j-ten Knotens in G_2. Im Verlauf der Backtrackingsuche werden Elemente auf 0 gesetzt, bis die Matrix eine injektive Abbildung repräsentiert und die Bedingung (3.3) erfüllt ist oder alle Möglichkeiten erfolglos getestet wurden. Dabei geht der Algorithmus systematisch vor, wobei in jedem Schritt alle Elemente einer Zeile außer einem auf 0 gesetzt werden. Führt dies nicht zu einem TGI, erfolgt ein Backtrackingschritt, der alte Zustand wird wieder hergestellt und im nächsten Versuch ein anderes Element der Zeile auf 1 belassen.

Eine entscheidende Verbesserung gegenüber bis dahin bekannten Verfahren erreicht der Algorithmus von Ullmann durch die Anwendung einer *Refinement Procedure*, durch die der Suchraum eingegrenzt wird. Wenn M einen TGI repräsentiert, bei dem $m_{i,j} = 1$ ist, so müssen auch alle Nachbarn von v_i auf Nachbarn von v_j abgebildet werden. Diese einfache Beobachtung wird genutzt, um frühzeitig Einsen in der Matrix M zu eliminieren. Immer wenn ein Element von M auf 0 gesetzt wurde, wird für jedes Element $m_{i,j}$ der Matrix folgende Bedingung geprüft:

$$\forall x \in \{0, ..., |V_1| - 1\} : (a_{i,x} = 1 \Rightarrow \exists y \in \{0, ..., |V_2| - 1\} : m_{x,y} b_{y,i} = 1) \tag{3.4}$$

Ist die Bedingung nicht erfüllt, so wird $m_{i,j}$ auf 0 gesetzt. Die Refinement Procedure wird so oft wiederholt, bis keine Veränderung der Matrix mehr eintritt, und macht dann eine Überprüfung der Bedingung 3.3 überflüssig. Dadurch dass Einsen durch Nullen ersetzt werden, wird die Anzahl der Möglichkeiten in folgenden Iterationen verringert und, sofern Zeilen entstehen, die keine einzige 1 mehr aufweisen, kann umgehend ein Backtrackingschritt erfolgen. Die Wirkung der Refinemente Procedure soll verstärkt werden, indem die Zeilen der Matrix in einer Reihenfolge bearbeitet werden, die der Sortierung der zugehörigen Knoten nach absteigendem Grad entspricht.

Bei der Implementierung des Verfahrens müssen Vorkehrungen getroffen werden, um den Zustand der Matrix nach einem Backtrackingschritt wieder herstellen zu können. Für ein Suchmuster mit n Knoten und einen Graphen mit m Knoten wird in [51] die Worst-Case-Laufzeit des Algorithmus mit $O(m^n n^2)$ angegeben, der benötigte Speicherbedarf kann durch $O(m^3)$ abgeschätzt werden [52].

3.2.3 VF2

Der VF2-Algorithmus basiert genau wie der Algorithmus von Ullmann auf Backtracking. Es wird wiederum versucht, eine partielle Abbildung schrittweise auszubauen, wobei jetzt jedoch stets benachbarte Knoten hinzugefügt werden. Durch zusätzliche Bedingungen, die sich mit geringem Aufwand auswerten lassen, soll dabei der Suchraum beschränkt werden. Der Algorithmus wurde 1999 von Vento et al. vorgestellt [20, 18] und nachfolgend verbessert [19, 52]. Durchgeführte Vergleichsstudien zeigen, dass der VF2-Algorithmus eine deutlich bessere Laufzeit als der Algorithmus von Ullmann aufweist [30]. Mit der Bibliothek *vflib* steht eine freie Implementierung des Algorithmus zur Verfügung. Während die Bibliothek verschiedene Varianten des Graph-Matchings unterstützt, beziehen sich die Veröffentlichungen in erster Linie auf GI und ITGI[3].

Der VF2 Algorithmus beschreibt den Suchraum durch Zustände, wobei jedem Zustand s eine partielle Abbildung $M(s) \subset V_1 \times V_2$ zugeordnet wird. Initial wird ein Zustand s_0 erzeugt mit $M(s_0) = \emptyset$. Ein Zustand wird in einen neuen überführt, indem ein Paar (p, h) mit $p \in V_1$ und $h \in V_2$ der Abbildung hinzugefügt wird. Für alle weiteren Zustände, die aus diesem entstehen, gilt also, dass der Knoten p auf h abgebildet wird. Dabei werden nur Zustände generiert, deren partielle Abbildung der geforderten Matching-Bedingung genügt. Ferner werden Zustände vermieden, deren partielle Abbildung zwar nicht direkt die Bedingung verletzt, für die aber ausgeschlossen werden kann, dass sie sich zu einer zulässigen Lösung erweitern lassen (*Pruning*).

Der Pseudocode 1 wurde [52] entnommen und beschreibt die rekursiv verwendete Funktion MATCH. Wird ein Zustand erreicht, in dem alle Knoten aus V_1 auf einen Knoten von

[3] In den Veröffentlichungen zu VF2 wird eine abweichende Definition verwendeten: Was dort als "graph-subgraph isomorphism" bezeichnet wird, entspricht der hier verwendeten Definition des ITGI. Die Bibliothek bietet TGI unter der Bezeichnung "monomorphism", vgl. Abschnitt 3.1.1.

Algorithmus 3.1 : MATCH(s)-Funktion des VF2-Algorithmus.

Eingabe : Zustand s mit Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ und partieller Abbildung $M(s)$; initial s_0 mit $M(s_0) = \emptyset$

Ausgabe : Abbildung $M(s)$ von G_1 auf G_2

```
1 if M(s) enthält alle Knoten von G1 then
2     Gib M(s) aus                                   ▷ Abbildung gefunden
3 else
4     P(s) ← generateCandidatePairs(s)
5     forall p ∈ P(s) do
6         if isFeasible(p,s) then
7             s' ← Zustand, der sich durch Hinzufügen von p zu M(s) ergibt
8             match(s')
9     Datenstruktur wiederherstellen
```

G_2 abgebildet werden, wurde eine zulässige Lösung gefunden (Zeile 2). Ansonsten wird versucht, die Abbildung zu vervollständigen. Wird ein geeignetes Paar zur Erweiterung gefunden, erfolgt ein rekursiver Aufruf der MATCH-Funktion mit dem Zustand, der sich durch Hinzufügen des Paares ergibt (Zeile 8). Ansonsten erfolgt ein Backtrackingschritt, wobei zuvor Änderungen an der Datenstruktur rückgängig gemacht werden müssen (Zeile 9).

Die Funktionen generateCandidatePairs und isFeasible stellen sicher, dass jede zulässige Lösung gefunden werden kann und erfolglose Teilbäume frühzeitig als solche erkannt werden. Dazu werden *Terminalmengen* verwendet: Für einen Zustand s ist $T_1^{in}(s) \subset V_1$ die Menge der Knoten, die noch nicht in die partielle Abbildung $M(s)$ aufgenommen wurden, aber mit einem Knoten in $M(s)$ über eine ausgehende Kanten verbunden sind. $T_1^{out}(s)$ ist analog für Knoten definiert, die über eine eingehende Kante verbunden sind. $T_2^{in}(s)$ und $T_2^{out}(s)$ sind die Terminalmengen für G_2. Die Menge der Knoten, die nicht unmittelbar mit der partiellen Abbildung verbunden sind, wird als $N_1(s)$ bzw. $N_2(s)$ bezeichnet. Abbildung 3.4 verdeutlicht den Zusammenhang für einen Zustand, in dem bereits vier Knoten eines Graphen abgebildet wurden.

Die Funktionen generateCandidatePairs findet neue Kandidaten, die der Abbildung hinzugefügt werden können. Dazu werden zunächst die Terminalmenge beider Graphen betrachtet, so dass die Abbildung möglichst um benachbarte Knoten erweitert wird. Falls $T_1^{out}(s)$ nicht leer ist, so ist $P(s) = \{min(T_1^{out}(s))\} \times T_2^{out}(s)$, wobei $min(T_1^{out}(s))$ das minimale Element aus $T_1^{out}(s)$ bezüglich einer beliebigen, festen totalen Ordnung meint. Sonst wird $P(s)$ analog aus $T_1^{in}(s)$ und $T_2^{in}(s)$ gebildet. Sind beide Terminalmengen leer (der Graph ist also unzusammenhängend), so wird $P(s)$ analog aus den Knoten außerhalb von $M(s)$ bestimmt. Aufgrund der totalen Ordnung steht in jedem Zustand fest, welcher

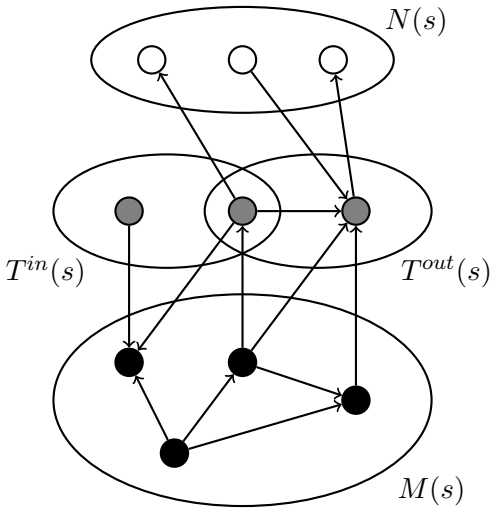


Abbildung 3.4: Die Knoten in der partiellen Abbildung $M(s)$ sind schwarz dargestellt, die daraus über eine Kante erreichbaren Knoten in den Terminalmengen $T^{in}(s)$ bzw. $T^{out}(s)$ grau. Nicht direkt verbundene Knoten der Menge $N(s)$ sind weiß dargestellt.

Knoten aus V_1 hinzugefügt wird, wodurch vermieden wird, dass eine partielle Abbildung über unterschiedliche Wege mehrmals erzeugt wird.

Die Funktion isFeasible prüft, ob das Hinzufügen des Paares die Matching-Bedingung verletzt und prüft mittels einer "k-Look-Ahead"-Regel, ob nach k Schritten noch ein konsistenter Folgezustand existieren kann. Die Implementierung unterscheidet sich je nach Matching-Variante. Für den Fall von ITGI werden Regeln aufgestellt, die mittels Terminalmengen einen 2-Look-Ahead für ein Paar (p, h) erreichen:

0-Look-Ahead: Für alle Kanten, die von $p \in V_1$ zu einem Knoten p' in der partiellen Abbildung führen, existiert eine entsprechende Kante von h zu einem h' mit $(p', h') \in M(s)$ und umgekehrt.

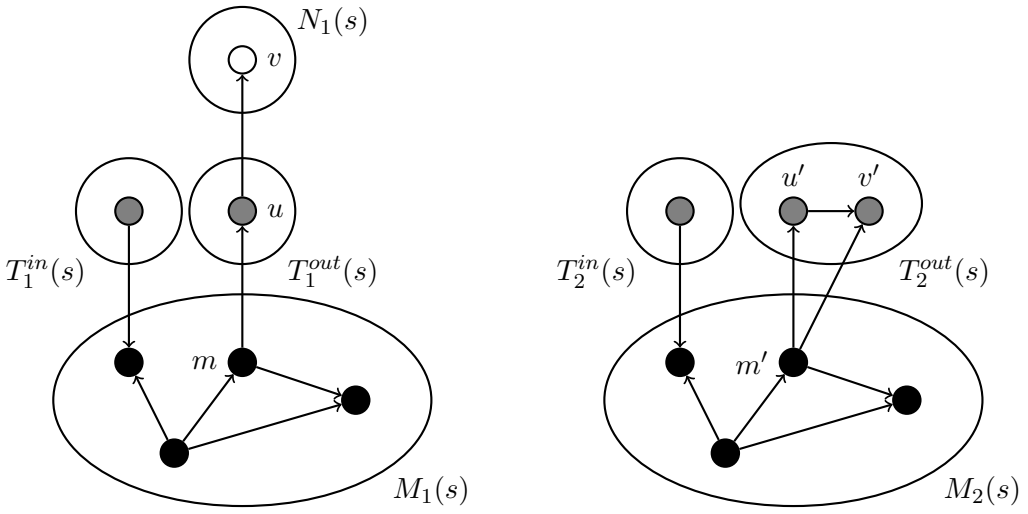
1-Look-Ahead: Die Anzahl der Nachbarn von p in T_1^{in} ist kleiner gleich der Anzahl der Nachbarn von h in T_2^{in}. Gleiches muss für die Anzahl der Nachbarn in T_1^{out} und T_2^{out} gelten.

2-Look-Ahead: Die Anzahl der Nachbarn von p in $N_1(s)$ ist kleiner gleich der Anzahl der Nachbarn von h in $N_2(s)$.

Außerdem wird geprüft, ob Knoten- und Kantenlabel zueinander passen.

Teilgraph-Isomorphie Für das TGI-Problem müssen die Look-Ahead-Regeln der Funktion isFeasible angepasst werden. Dem Quellcode der vflib lassen sich folgende Änderungen gegenüber den Regeln für ITGI entnehmen:

0-Look-Ahead: Der umgekehrte Fall wird nicht mehr geprüft.



(a) G_1 (b) G_2

Abbildung 3.5: Von dem Graphen G_1 (a) lässt sich ausgehend vom Zustand s ein TGI auf G_2 (b) ableiten, wobei u auf u' und v auf v' abgebildet werden. Da G_2 eine Kante (m', v') enthält, während eine Kante (m, v) in G_1 nicht existiert, befinden sich v und v' in unterschiedlichen Mengen.

1-Look-Ahead: Entspricht der Regel für ITGI.

2-Look-Ahead: Sei n_1 die Anzahl der Nachbarn von p in $N_1(s)$, i_1 in $T_1^{in}(s)$ und o_1 in $T_1^{out}(s)$, sowie n_2, i_2, o_2 analog für h. Es muss gelten $n_1 + i_1 + o_1 \leq n_2 + i_2 + o_2$.

Die 2-Look-Ahead-Regel muss also abgeschwächt werden, damit keine zulässigen Lösungen ausgeschlossen werden. Im Fall von ITGI konnten Knoten aus $T_1^{in}(s)$, $T_1^{out}(s)$ und $N_1(s)$ nur auf Knoten der entsprechenden Menge von G_2 abgebildet werden. Abbildung 3.5 verdeutlicht, dass dieser Zusammenhang für einen TGI nicht gilt, da Knoten aus $N_1(s)$ jetzt auf Knoten aus $T_2(s)$ abgebildet werden können.

Implementierung Neben der partiellen Abbildung $M(s)$ werden mit dem Zustand die Terminalmengen für G_1 und G_2 verwaltet. Dazu werden sechs Felder verwendet: `core_1` und `core_2` haben die Dimension von $|V_1|$ bzw. $|V_2|$ und speichern die Abbildung. Wenn $p \in V_1$ auf $h \in V_2$ abgebildet wird, ist `core_1[p]=h` und `core_2[h]=p`, wobei die Knoten durch ihre Indizes identifiziert werden. Die vier Terminalmengen werden ebenfalls mit Feldern gespeichert.

Wird ein Paar der Abbildung hinzugefügt, müssen die Abbildung und die Terminalmengen entsprechend angepasst werden. Um den Speicherbedarf gering zu halten, wird bei dem rekursiven Aufruf der MATCH-Funktion kein komplett neuer Zustand erzeugt, sondern alle Zustände teilen sich die sechs Felder. Dadurch wird es erforderlich, dass beim Backtracking Änderungen rückgängig gemacht werden (Algorithmus 3.3, Zeile 9). Um genau die Knoten aus den Terminalmengen entfernen zu können, die im Schritt zuvor eingefügt

worden sind, wird in den Feldern die Tiefe des Suchbaums (bzw. Größe der partiellen Abbildung) gespeichert, bei der der Knoten in die Terminalmenge aufgenommen worden ist. Somit wird nur wenig Speicher für einen Zustand und rekursiven Aufruf benötigt.

Für die Laufzeit des Algorithmus wird $O(m!m)$ angegeben und der Speicherbedarf durch $O(m)$ abgeschätzt, wobei $m = |V2| \geq |V_1|$ ist. Aufgrund des geringen Speicherbedarfs eignet sich der VF2-Algorithmus auch für sehr große Graphen.

3.2.4 Suchplanoptimierung

Backtracking-Algorithmen für das TGI-Problem erweitern eine partielle Abbildung meist schrittweise, indem Elemente (Knoten und Kanten) des Suchmusters auf passende Elemente des Zielgraphen so abgebildet werden, dass die TGI-Bedingung erfüllt bleibt. Dabei kann besonders für gelabelte Graphen die Reihenfolge, in der Elemente des Suchmusters in die Abbildung aufgenommen werden, einen starken Einfluss auf die Größe des Suchraums und somit auf die Laufzeit haben. Unter dem Begriff Suchplanoptimierung können Ansätze zusammengefasst werden, die durch vorherige Analyse der Graphen versuchen, eine möglichst günstige Reihenfolge zu finden. Heuristiken, um eine günstige Reihenfolge zu ermitteln, wurde vor allem im Bereich der *Graph Transformation* untersucht.

Batz [8] formuliert *primitive matching operations* für einen gerichteten gelabelten Graphen und definiert einen gültigen Suchplan als Folge solcher Operationen, die, der Reihe nach angewandt, das gesamte Suchmuster abdecken. Als *lookup operation* wird eine Operation bezeichnet, die ein Element des Suchmusters, das nicht direkt mit einem bereits abgebildeten Element verbunden ist, auf den Zielgraphen abgebildet wird. Eine *extension operation* beschreibt die Erweiterung der partiellen Abbildung um einen adjazenten Knoten und der zugehörigen Kante. Ziel ist es, einen Suchplan zu finden, bei dem für die Matchingoperationen möglichst wenige Auswahlmöglichkeiten bestehen. Dazu werden den Operationen Kosten in Höhe der durchschnittlichen Anzahl Kandidaten im Zielgraphen zugeordnet, wozu dieser zuvor analysiert werden muss. Mit Hilfe eines *Plan Graphs* wird anschließend ein Suchplan ermittelt, bei dem günstige Operationen möglichst früh ausgeführt werden und unvermeidbare, teure Operationen so spät wie möglich. Besonders für Graphen mit geringer Anzahl Kanten und einer Vielzahl unterschiedlicher Label verspricht die Methode gute Ergebnisse.

Andere Modelle wurden zuvor von Dörr und Zünndorf im Zusammenhang mit Graph Rewriting Tools vorgestellt. Dörrs [23] Ansatz ermöglicht es, das TGI-Problem für gelabelte Graphen in Linearzeit zu lösen, wenn ein Suchplan mit Matchingoperationen ohne Auswahlmöglichkeiten existiert. Zünndorf beschreibt eine heuristische Optimierung des Suchplans für das Graph Rewriting Tool Progres [78].

3.2.5 Constraintprogrammierung

Constraintprogrammierung ist ein allgemeiner Ansatz zur Lösung kombinatorischer Probleme, der es ermöglicht, ein Problem formal mit Hilfe von Constraints zu beschreiben und automatisiert zu lösen. Nach [56] ist ein *Constraint Satisfaction Problem* (CSP) durch eine Menge von Variablen X mit Angabe ihrer *Domain* D und einer Menge von Bedingungen C definiert. Die Lösung eines CSP besteht in einer Belegung der Variablen, die alle Bedingungen erfüllt. Existiert keine Lösung, wird das CSP als unerfüllbar bezeichnet. Das CSP ist im Allgemeinen NP-vollständig und kann wiederum mit einem Backtracking-Algorithmus gelöst werden.

Die folgende direkte Formulierung des TGI-Problems (ohne Berücksichtigung von Labeln) als CSP ist [74] entnommen: Für jeden Knoten $v \in V_1$ wird eine Variable x_v definiert mit $D(x_v) = V_2$. Bedingungen der Form

$$(u, v) \in E_1 \Rightarrow (x_u, x_v) \in E_2 \tag{3.5}$$

sichern, dass jede Kante von G_1 auf eine entsprechende Kante in G_2 abgebildet wird. Durch die Bedingungen $\forall i, j \in V_1 : i \neq j \Rightarrow x_i \neq x_j$ wird sichergestellt, dass jede zulässige Variablenbelegung eine injektive Abbildung liefert.

Ein Vorteil der Constraintprogrammierung ist es, dass zahlreiche allgemeine Optimierungsverfahren für CSPs auf diese Weise eine direkte Anwendung bei der Lösung des TGI-Problems finden: So entspricht das bei der Lösung von CSPs übliche Konzept der *arc consistency* für die Bedingungen (3.5) der Refinement Procedure des Algorithmus von Ullmann (siehe Abschnitt 3.2.2) [74]. Die Reihenfolge, in der Variablen ein Wert zugewiesen wird, ist ebenfalls Thema bei der Optimierung von CSPs und weist starke Parallelen zur Suchplanoptimierung auf [8]. Darüber hinaus können die Symmetrien (Automorphismen) beider Graphen genutzt werden, um symmetrische Abläufe bei der Suche nach einer Lösung des CSPs zu vermeiden (siehe Abschnitt 3.2.6).

Der Ansatz der Constraintprogrammierung ist äußerst flexibel, leicht erweiterbar und eignet sich besonders, um neue Ideen zur Verbesserung auf ihre Tauglichkeit hin zu überprüfen. Gegenüber eines direkten Backtracking-Algorithmus ist jedoch mit einem gewissen Mehraufwand zu rechnen. In experimentellen Vergleichen mit Graphen ohne Label erwies sich der CSP-Ansatz besonders für schwierige Instanzen als geeignet, während für leichte Instanzen der VF2-Algorithmus bessere Ergebnisse lieferte [74]. Es ist davon auszugehen, dass Graphen mit Labeln, wie sie bei der Substruktursuche auftreten, eher zu den leichten Instanzen zu zählen sind.

3.2.6 Ausnutzung von Automorphismen

Eine allgemeine Technik, die mit unterschiedlichen Ansätzen kombiniert wurde, ist das Ausnutzen von Automorphismen der beteiligten Graphen. Die grundlegende Idee besteht

darin, dass die Automorphismen der Graphen zu symmetrischen Abläufen bei der Suche führen. Sind die Automorphismen der Graphen bekannt, können symmetrische Fälle bei der Suche vermieden und die Laufzeit dadurch verkürzt werden.

Die Software *nauty* nutzt Automorphismen, um effizient kanonische Bezeichner für Graphen zu berechnen [49] und kann somit auch zur Lösung des GI-Problems verwendet werden. Speziell für das GI-Problem stellten López-Presa et al. 2009 einen neuen Algorithmus vor, der ebenfalls Automorphismen ausnutzt [46]. Der Algorithmus verfolgt einen Partitionierungsansatz: Beginnend mit einer Partition von Knoten mit gleichem Grad werden diese iterativ verfeinert. Dabei werden alle Zellen der Partition so unterteilt, dass nur solche Knoten in derselben Teilmenge verbleiben, die die gleiche Nachbarschaftsbeziehung zu einem gewählten Knoten (*Vertex Refinement*) oder einer gewählten Menge von Knoten aufweisen (*Set Refinement*). Der Algorithmus arbeitet zunächst nur auf einem Graphen und berechnet eine Folge von Partitionen, die durch Verfeinerungsschritte entstehen. Existiert keine Zelle in der Partition, die zur Verfeinerung genutzt werden kann, so wird ein beliebiger Knoten aus einer Zelle gewählt und mit diesem ein Vertex-Refinement durchgeführt. Diese Operation wird in der Verfeinerungsfolge als Backtracking gekennzeichnet. Die gleiche Folge von Verfeinerungsschritten wird anschließend versucht auf den anderen Graphen anzuwenden. Nur wenn dessen Knotenmenge mit der gleichen Verfeinerungsfolge partitioniert werden kann, sind beiden Graphen isomorph. Bei der Rekonstruktion der Verfeinerungsfolge auf dem zweiten Graphen müssen bei den mit Backtracking gekennzeichneten Operationen alle Knoten einer Zelle getestet werden. Die Anzahl solcher Fälle soll durch die Erkennung von Automorphismen verringert werden. Zusammen mit der Partitionierungsfolge werden äquivalente Knoten bestimmt, die ein Automorphismus aufeinander abbildet. Backtrackingschritte an äquivalenten Knoten werden anschließend eliminiert, wodurch die Suche beschleunigt wird. Die Übertragung auf das TGI-Problem ist nicht ohne weiteres möglich, da einerseits der Partitionierungsansatz in dieser Art nicht funktioniert und sich andererseits die Automorphismen beider Graphen unterscheiden können, wenn ein TGI existiert.

Für die Suche von Mustern in biologischen Netzwerken beschreiben Kellis et al. [34] ein Verfahren, das die Automorphismen des Suchmusters ausnutzt. Die Problemstellung unterscheidet sich von der Substruktursuche dadurch, dass hier alle Vorkommen des Musters im Netzwerk gesucht werden. Das Verfahren wurde in einen Backtracking-Ansatz integriert, der mit dem Algorithmus von Ullmann vergleichbar ist. Die Automorphismen des Suchmusters werden genutzt, indem zusätzliche Bedingungen an eine erlaubte Abbildung aufgestellt werden: Sei $n : V \to \mathbb{Z}$ eine Nummerierung der Knoten eines Graphen. Für eine Äquivalenzklasse von Knoten des Suchmusters $\{v_0, ..., v_k\}$ wird die Bedingung $n(\varphi(v_0)) < \min\{n(\varphi(v_1)), ..., n(\varphi(v_k))\}$ an eine gültige Abbildung φ gestellt. Derartige Bedingungen werden generiert, bis kein Automorphismus des Suchmusters, der alle Bedingungen erfüllt, mehr existiert, außer der identischen Abbildung $\varphi(v_i) = v_i$. Die zusätzlichen

Bedingungen werden bei der späteren Suche im Netzwerk bzgl. dessen Knotennummerierung direkt geprüft. Eine Beschleunigung resultiert vor allem daraus, dass es nur noch eine einzige Möglichkeit gibt, das Suchmuster auf einen bestimmten Teilgraphen des Netzwerks abzubilden und jedes Vorkommen des Musters somit nur ein einziges Mal gefunden wird.

In der Constraintprogrammierung können die Symmetrien eines CSPs verwendet werden, um die Berechnung der Lösungen zu beschleunigen (*Symmetry-Breaking*). Zampelli [74] untersuchte die entstehenden Symmetrien des als CSP modellierten TGI-Problems. Dabei besteht ein direkter Zusammenhang zwischen den Automorphismen des Suchmusters und den Symmetrien der Variablen. Die Automorphismen des Zielgraphen spiegeln sich in den Symmetrien der Domain der Variablen wieder. Zusätzlich werden lokale Symmetrien aus den Automorphismen von Teilgraphen des Suchmusters bzw. des Zielgraphen abgeleitet. Alle Arten von Symmetrien können bei der Lösung eines CSPs genutzt werden, um den Suchraum für Lösungen einzuschränken.

3.3 Ein neuer Backtracking-Algorithmus

Gegen die Verwendung der vflib-Implementierung des VF2-Algorithmus in Scaffold Hunter spricht, dass diese nur in C++ verfügbar ist. Eine Integration in ein bestehendes Java Projekt gestaltet sich schwierig und lässt sich kaum mit der gewünschten Plattformunabhängigkeit vereinbaren. Außerdem arbeitet die Implementierung mit gerichteten Graphen, so dass ungerichtete Graphen nur verwendet werden können, indem für jede ungerichtete Kante zwei gerichtete Kanten eingefügt werden, wodurch vermeidbarer Mehraufwand entsteht. Das im Java-Toolkit CDK verwendete MCS-basierte Verfahren (siehe Abschnitt 3.2.1) löst eine Verallgemeinerung des TGI-Problems und erfordert den Aufbau eines möglicherweise großen Assoziationsgraphen. Vorab durchgeführte Testläufe deuten darauf hin, dass die Laufzeit des Verfahrens für die Substruktursuche kaum akzeptabel ist.

In diesem Abschnitt wird daher ein neuer Algorithmus entwickelt, der speziell an die vorliegende Problemstellung angepasst ist und Ideen mehrerer im Abschnitt 3.2 beschriebener Ansätze aufgreift. Die Idee, eine partielle Abbildung Schritt für Schritt um ein benachbartes Knotenpaar zu erweitern, wurde vom VF2-Algorithmus übernommen. Auf das frühzeitige Erkennen aussichtsloser Äste im Suchraum mit Hilfe von Terminalmengen wurde verzichtet, da die Look-Ahead-Regeln für TGI weniger restriktiv sind (siehe Abschnitt 3.2.3). Der Mehraufwand zur Verwaltung der Terminalmengen und zur Auswertung der Look-Ahead-Regeln ist außerdem beträchtlich: Während bei der Suche eines ITGI die zur Auswertung notwendigen Berechnungen zum Teil mit der Überprüfung der Existenz von Kanten von G_2 in G_1 kombiniert werden können, muss diese Überprüfung bei der Suche eines TGI entfallen. Die Berechnungen nur zur Auswertung der Look-Ahead-Regeln beizubehalten, würde zusätzlichen Mehraufwand bedeuten, der möglicherweise nicht durch die Wirkung gerechtfertigt werden kann.

Der neue Algorithmus nutzt aus, dass das gleiche Suchmuster nacheinander in einer Vielzahl von Graphen gesucht wird, indem das Suchmuster zunächst einem Vorverarbeitungsschritt unterzogen wird, der in Abschnitt 3.3.1 beschrieben wird. Der Algorithmus selbst wird in Abschnitt 3.3.2 beschrieben. Ideen der Suchplanoptimierung können auf einfache Weise integriert werden, wie in Abschnitt 3.3.4 gezeigt wird. Abschnitt 3.3.5 gibt einen Ausblick auf weitere Optimierungsmöglichkeiten, die u.a. der Constraintprogrammierung entlehnt sind.

3.3.1 Vorverarbeitungsschritt

Eine spezieller Umstand, der bei der Substruktursuche auftritt und zur Beschleunigung ausgenutzt werden kann, ist, dass das gleiche Suchmuster nacheinander in einer Vielzahl von Graphen gesucht wird. Dadurch erscheint es lohnend, zunächst einmalig einen Vorverarbeitungsschritt auszuführen, von dem alle anschließenden Suchvorgänge profitieren.

Der Vorverarbeitungsschritt berechnet eine Folge $(v_1, ..., v_n)$ der Knoten V des Suchmusters $G = (V, E)$. In dieser Reihenfolge sollen die Knoten bei der späteren Suche der partiellen Abbildung hinzugefügt werden. Dabei ist es wünschenswert, wenn der abgebildete Teilgraph zusammenhängend ist.

Notation (Maximal zusammenhängende Folge). Für einen Graphen $G = (V, E)$ wird eine Folge $(v_1, ..., v_n)$ mit $n = |V|$ von Knoten der Menge V hier als *maximal zusammenhängend* bezeichnet, wenn für alle $j \in \{1, ..., n\}$ gilt:

$$\exists i < j : (v_i, v_j) \in E \tag{3.6}$$

$$\vee \quad \forall k \geq j : \nexists i < j : (v_i, v_k) \in E \tag{3.7}$$

Die Gleichung (3.6) stellt sicher, dass jeder Knoten zu einem in der Folge vorausgehenden Knoten adjazent ist. Diese Bedingung kann für den ersten Knoten v_1 der Folge niemals erfüllt werden, was durch Gleichung (3.7) abgedeckt wird. Zudem bleibt die Anforderung für Graphen erfüllbar, die nicht zusammenhängend sind. Die einzelnen Knoten einer Zusammenhangskomponente befinden sich aber zusammen in einem "Abschnitt" der Folge. Die Beschränkung auf maximal zusammenhängende Folgen ist keine Voraussetzung für die Korrektheit des Algorithmus. Sie ist allerdings sinnvoll, um die Laufzeit möglichst zu verkürzen, wie in Abschnitt 3.3.4 anhand eines Beispiels erläutert wird.

Bei der Vorverarbeitung sollen außerdem zu jedem Knoten v_j alle Kanten ermittelt und gespeichert werden, die die Gleichung (3.6) erfüllen.

Notation (Verbindungskante). Für eine Folge $(v_1, ..., v_n)$ zu einem Graphen $G = (V, E)$ wird eine Kante $(v_i, v_j) \in E$ hier als *Verbindungskante* von v_j bezeichnet, wenn $i < j$ gilt. $E_{v_j} = \{(v_i, v_j) \in E | i < j\}$ sind die Verbindungskanten von v_j.

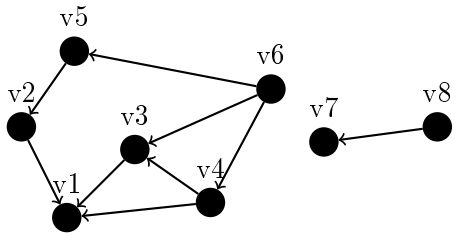


Abbildung 3.6: Eine mögliche maximal zusammenhängende Folge für einen ungerichteten Graphen mit zwei Zusammenhangskomponenten. Um die Klassifikation der Kanten zu verdeutlichen, wurden diese von dem Knoten mit der größeren Nummer zu dem mit der kleineren gerichtet. Die ausgehenden Kanten eines Knotens sind also seine Verbindungskanten.

Abbildung 3.6 zeigt eine mögliche maximal zusammenhängende Folge und die zugehörigen Verbindungskanten für einen ungerichteten Graphen mit zwei Zusammenhangskomponenten. Eine solche Folge zusammen mit den Verbindungskanten kann leicht mit Hilfe einer Breitensuche ermittelt werden. Der Algorithmus 2 verwendet in Anlehnung an [21] die Farben Weiß, Grau und Schwarz, um den Fortschritt der Bearbeitung festzuhalten.

Um sicherzustellen, dass jeder Knoten besucht wird, werden in der äußeren Schleife (Zeile 5) alle Knoten des Graphen durchlaufen und an jedem noch nicht besuchten Knoten (weiß) eine Breitensuche gestartet. Dazu wird der Knoten der Queue hinzugefügt und gilt damit als gefunden (grau). Sobald in der Schleife ein Knoten aus der Queue entnommen wurde, wird er als bearbeitet (schwarz) gekennzeichnet und erhält die nächste Position in der Folge (Zeile 12). Es werden allen Nachbarn des Knotens betrachtet und solche, die noch nicht gefunden wurden, der Queue hinzugefügt. Ist ein Nachbar als schwarz gekennzeichnet, so wird die zugehörige Kante als Verbindungskante gespeichert.

Die Folge, die der Algorithmus ausgibt, erfüllt die Bedingungen (3.6) und (3.7), da durch jede Breitensuche eine Zusammenhangskomponente des Graphen vollständig bearbeitet wird und die zugehörigen Knoten korrekt nummeriert werden: Wenn ein Knoten v aus der Queue entnommen wird, ist er entweder der erste einer Zusammenhangskomponente (3.7) oder wurde über (mindestens) eine Kante (w, v) gefunden und dann in die Queue eingefügt. In dem Fall ist dem Knoten w bereits eine kleinere Position in der Folge zugeordnet worden. Durch die Existenz der Kante (w, v) ist also die Bedingung (3.6) erfüllt. Der Algorithmus liefert also eine maximal zusammenhängende Folge.

Wenn die Nachbarn eines Knotens v durchlaufen werden (Zeile 12), hat v eine Position c in der Folge erhalten und alle Knoten mit einer Position $c' < c$ wurden als schwarz markiert. Somit wird jede Verbindungskante gefunden und E_v korrekt berechnet. Die Laufzeit beträgt insgesamt $O(|V| + |E|)$.

Algorithmus 3.2 : Vorverarbeitungsschritt TGI.

Eingabe : Graph $G = (V, E)$

Ausgabe : Folge $(v_1, ..., v_n)$ mit Verbindungskanten E_v für alle $v \in V$

```
1  forall u ∈ V do                                          ▷ Initialisiere
2      farbe[u] ← weiß
3  c ← 1
4  Q ← ∅
5  forall u ∈ V do
6      if farbe[u] = weiß then
7          ENQUEUE(Q, u)
8          farbe[u] ← grau
9          while Q ≠ ∅ do
10             v ← DEQUEUE(Q)
11             farbe[v] ← schwarz
12             v_c ← v
13             c ← c + 1
14             forall w ∈ Adj(v) do
15                 if farbe[w] = weiß then
16                     ENQUEUE(Q, w)
17                     farbe[w] ← grau
18                 else if farbe[w] = schwarz then
19                     E_v ← E_v ∪ {(v, w)}
```

3.3.2 Algorithmus

Der Algorithmus sucht den Graphen G_1 in dem Graphen G_2, indem inkrementell eine partielle Abbildung $\varphi : V_1 \rightsquigarrow V_2$ erweitert wird. Dazu wird nach und nach versucht, für jeden Knoten $v \in V_1$ einen Knoten $k \in V_2$ zu finden, so dass für alle Knoten in der Abbildung die TGI-Bedingung gilt. Während die Knoten aus V_1 in der vorberechneten Reihenfolge in die Abbildung aufgenommen werden, werden die in Frage kommenden Knoten aus V_2 in jedem Schritt dynamisch ermittelt. Dazu werden die ermittelten Verbindungskanten genutzt, die zudem geeignet sind, die Überprüfung der TGI-Bedingung zu beschleunigen. Der Pseudocode Algorithmus 3.3 zeigt den Ablauf des Verfahrens.

Die Funktion MATCH() wird dabei rekursiv aufgerufen, verwendet aber stets die gleiche Datenstruktur, bestehend aus den Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, der Folge $v_1, ..., v_n$ der Knoten aus V_1 mit den Verbindungskanten E_v für alle $v \in V_1$ und eine partielle Abbildung $\varphi : V_1 \rightsquigarrow V_2$. Nur φ wird über die Funktionsaufrufe verändert und ist initial

Algorithmus 3.3 : MATCH()-Funktion TGI.

Daten : Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, eine Folge $(v_1, ..., v_n)$ der Knoten von G_1 mit Verbindungskanten E_v für alle $v \in V_1$ und eine initial leere partielle Abbildung $\varphi : V_1 \rightsquigarrow V_2$.

Ausgabe : `true`, wenn ein TGI von G_1 zu G_2 existiert, `false` sonst

```
1  if SIZE(φ) = n then                                   ▷ Abbildung gefunden
2      return true
3  v ← v_{SIZE(φ)+1}                                     ▷ Nächster Knoten der Folge
4  if E_v = ∅ then
5      forall k ∈ V_2 do
6          if ISFEASIBLE(v, k) then
7              φ(v) ← k                                  ▷ Abbildung erweitern
8              if MATCH() then return true
9              φ(v) ← NIL                                ▷ Erweiterung rückgängig machen
10 else
11     j ← φ(h) für ein beliebiges h mit (h, v) ∈ E_v
12     forall k ∈ Adj(j) do
13         if ISFEASIBLE(v, k) and ISFEASIBLECONNECTED(v, k, E_v) then
14             φ(v) ← k
15             if MATCH() then return true
16             φ(v) ← NIL
17 return false                                          ▷ Kein Kandidat war erfolgreich
```

leer. SIZE(φ) bezeichnet die Anzahl der abgebildeten Knoten $v \in V_1$ mit $\varphi(v) \neq$ `NIL`, so dass initial SIZE(φ)= 0 gilt. Ist die Bedingung in Zeile 1 nicht erfüllt, wird versucht, die Abbildung um ein Paar (v, k) zu erweitern, wobei v der nächste Knoten der Folge ist (Zeile 1).

Welche Knoten $k \in V_2$ als Kandidaten in Frage kommen, hängt davon ab, ob sich bereits ein Nachbar von v in G_1 in der aktuellen partiellen Abbildung befindet und es somit mindestens eine Verbindungskante von v gibt. Ist $E_v = \emptyset$ werden alle Knoten $k \in V_2$ als mögliche Kandidaten betrachtet (Zeile 5). Die Funktion ISFEASIBLE(v,k) (Zeile 6 bzw. 13) gibt genau dann `true` zurück, wenn

- noch kein anderer Knoten auf k abgebildet wird ($\forall w \in V_1 : \varphi(w) \neq k$) und
- die Label beider Knoten zueinander passen ($l(v) = l(k)$) und
- $deg(v) \leq deg(k)$ gilt.

Während die ersten beiden Bedingungen notwendig sind, damit die Erweiterung der partiellen Abbildung um (v, k) ein TGI bleibt, ist die dritte Bedingung notwendig, damit sich die partielle Abbildung anschließend auf ganz V_1 erweitern lässt. Dadurch sollen frühzeitig Zustände vermieden werden, die nicht zum Erfolg führen können.

Gibt es Verbindungskanten E_v, so lässt sich die Menge der Kandidaten einschränken (Zeile 12): Jede Verbindungskante führt zu einem Knoten h mit $\varphi(h) \neq$ `NIL`. Da für jede Verbindungskante (h, v) eine entsprechende Kante in G_2 existieren muss, kommen nur solche Knoten k in Frage, die Nachbarn von $\varphi(h)$ sind. Für alle Kandidaten wird wieder geprüft, ob die Erweiterung der partiellen Abbildung die TGI-Bedingung verletzt. Jetzt müssen neben den oben beschriebenen Bedingungen der Funktion isFeasible(v,k) zusätzlich *alle* Verbindungskanten auf eine Kante von G_2 abgebildet werden können. Dies prüft die Funktion isFeasibleConnected (v,k,E_v) (Zeile 13). Sie gibt genau dann `true` zurück, wenn gilt

$$\forall e_1 = (u, v) \in E_v : \quad \exists e_2 = (\varphi(u), \varphi(v)) \in E_2 \quad \wedge \quad l(e_1) = l(e_2).$$

Wenn die partielle Abbildung erweitert wurde, wird die Funktion match() erneut aufgerufen. Die Tiefe der Rekursion entspricht also der Größe der partiellen Abbildung. Enthält die partielle Abbildung alle Knoten aus V_1, wird `true` zurückgegeben (Zeile 1 und 8 bzw. 15).

3.3.3 Analyse

In diesem Abschnitt wird die Korrektheit des Algorithmus 3.3 begründet und die Laufzeit analysiert.

Korrektheit

Zunächst werden einige Hilfssätze genannt, mit denen anschließend die Korrektheit des Algorithmus gezeigt wird. Lemma 3.7 und Lemma 3.8 wurde von Valiente übernommen und sind mit Beweis in [66] zu finden.

Lemma 3.7 (Knotengrad [66]). *Sei $\varphi : V_1 \to V_2$ ein TGI von $G_1 = (V_1, E_1)$ auf $G_2 = (V_2, E_2)$. Dann gilt $deg(v) \leq deg(w)$ für alle $v \in V_1$ mit $\varphi(v) = w$.*

Lemma 3.8 (Erweiterung einer partiellen Abbildung [66]). *Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei ungerichtete Graphen, $V_1' \subseteq V$ und $V_2' \subseteq V_2$, sowie $\varphi : V_1' \to V_2'$ ein GI zwischen dem durch V_1' in G_1 induzierten Teilgraphen und einem Teilgraphen von G_2 mit Knotenmenge V_2'. Für alle Knoten $v \in V_1 \setminus V_1'$ und $w \in V_2 \setminus V_2'$ ist die Erweiterung*

φ' von φ mit $\varphi'(v) = w$ ein GI von dem durch $V_1' \cup \{v\}$ in G_1 induzierten Teilgraphen auf einen Teilgraphen von G_2 mit Knotenmenge $V_2 \cup \{w\}$, genau dann wenn gilt[4]

$$\forall x \in V_1', y \in V_2' \text{ mit } \varphi(x) = y : (v, x) \in E_1 \Rightarrow (w, y) \in E_2 \tag{3.8}$$

Lemma 3.9. *Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei Graphen und $\varphi : V_1 \to V_2$ ein TGI von G_1 auf G_2. Dann existiert für alle $V_1' \subseteq V_1$ ein TGI φ' von dem durch V_1' in G_1 induzierten Teilgraphen $G_1' = (V_1', E_1')$ auf G_2.*

Satz 3.10. *Algorithmus 3.3 gibt genau dann* `true` *aus, wenn ein TGI von G_1 auf G_2 existiert.*

Beweis. Der Beweis lässt sich in zwei Schritten führen: Zunächst wird gezeigt, dass ein TGI existiert, wenn der Algorithmus `true` zurückgibt, anschließend, dass der Algorithmus stets `true` ausgibt, wenn ein TGI existiert.

Der Algorithmus gibt nur dann `true` aus, wenn eine Abbildung der Größe $|V_1| = n$ generiert wurde (Zeile 1). Da nur vor jedem rekursiven Aufruf ein Element der Abbildung hinzugefügt wird, welches bei Misserfolg anschließend wieder entfernt wird, muss die Rekursionstiefe $n + 1$ erreicht worden sein, wobei im i-ten Methodenaufruf ein Paar (v_i, k) hinzugefügt worden ist. Für die Folge $(v_1, ..., v_n)$ sei G_1^i der durch $V_1^i = \{v_j \in V_1 | j \leq i\}$ in G_1 induzierte Teilgraph. Die Korrektheit lässt sich durch vollständige Induktion zeigen: Initial ist $G_1^0 = (\emptyset, \emptyset)$ und die leere Abbildung φ ein TGI auf jeden Graphen G_2. Im i-ten Schritt wird das Paar (v_i, k) der Abbildung hinzugefügt, wobei G_1^{i-1} nach Induktionsvoraussetzung isomorph zu einem Teilgraphen von G_2 ist. Da durch den Test isFeasible(v_i,k) und isFeasibleConnected(v_i,k,E_{v_i}) die Bedingungen des Lemmas 3.8 erfüllt sind, ist auch G_1^i isomorph zu einem Teilgraphen von G_2. Somit ist schließlich $G_1^n = G_1$ isomorph zu einem Teilgraphen von G_2.

Die andere Richtung der Äquivalenzbeziehung erfordert den Nachweis, dass, wenn ein TGI von G_1 auf G_2 existiert, der Algorithmus `true` ausgibt. Sei φ ein solcher TGI von G_1 auf G_2. Nach Lemma 3.9 existiert dann für jeden induzierten Teilgraphen von G_1 ein TGI auf G_2. Es gilt also insbesondere, dass es für alle $0 \leq i \leq n$ einen TGI von G_1^i auf G_2 gibt. Nach Lemma 3.8 existiert für jedes $i \leq n$ eine Erweiterung (v_i, k) mit $v \in V_1$ und $k \in V_2$ mit der G_1^{i-1} zu G_1^i erweitert wird. Der Algorithmus findet die zulässige Lösung, wenn die zugehörige Folge von Erweiterungen gefunden wird. Es genügt also zu zeigen, dass der Algorithmus keine Erweiterungen ausschließt, die zur gewünschten Lösung führt. Die Funktion isFeasible(v_i, k) gibt auch für Kandidaten k mit $deg(v_i) > deg(k)$ `false` zurück. Nach Lemma 3.7 können dadurch keine Erweiterungen vernachlässigt werden, die zu zulässigen Lösungen führen. Der Algorithmus beschränkt außerdem die Kandidaten

[4] Hierbei wurde die Korrektur nach `http://www.lsi.upc.edu/~valiente/algorithm/errata.html` berücksichtigt und das Lemma für ungerichtete Graphen angepasst.

$k \in V_2$ auf Nachbarn eines Knotens $j \in V_2$ mit $j = \varphi(h)$, falls $(h, v) \in E_v$ (Zeile 12). Dabei handelt es sich um eine notwendige Bedingung für isFeasibleConnected(v_i, k, E_{v_i}), so dass auch dadurch keine erlaubten Erweiterungen unberücksichtigt bleiben. □

Laufzeit

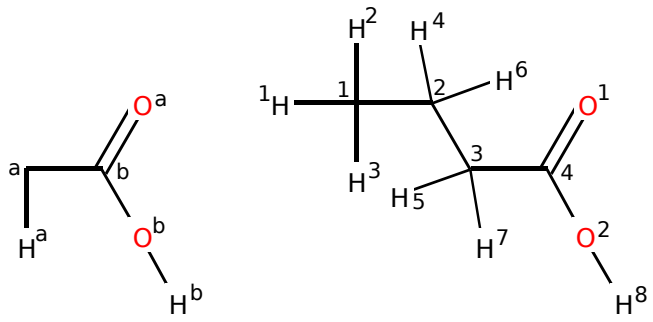
Die Funktion isFeasible(v, k) hat die Laufzeit $O(1)$, wenn zusätzlich zu φ die Umkehrfunktion φ^{-1} mitgeführt wird. Die Laufzeit der Funktion isFeasibleConnected(v, k, E_v) hängt von der Laufzeit einer Anfrage nach der Existenz einer Kante (u, v) ab. Mit Hilfe von Adjazenzlisten kann die Anfrage in $O(deg(v))$ beantwortet werden, mit einer Adjazenzmatrix in $O(1)$. In dem Fall weist die Funktion eine Gesamtlaufzeit von $O(|E_v|)$ auf. Da die bei der Substruktursuche auftretenden Graphen einen kleinen, beschränkten Grad ausweisen, wurden für die Implementierung dennoch Adjazenzlisten verwendet.

Sei $n = |V_1|$, $m = |V_2|$. Für zusammenhängende Graphen mit beschränktem Knotengrad d lässt sich die Worst-Case-Laufzeit durch $O(md^n)$ abschätzen. Die Anzahl der Kandidaten beträgt m für den ersten Knoten aus V_1 und maximal d für weitere Knoten. Die Anzahl betrachteter partieller Abbildungen beträgt somit maximal md^{n-1}. Jeder partiellen Abbildung geht ein Aufruf von isFeasible() und ggf. isFeasibleConnected() voraus, die zusammen die Laufzeit $O(1 + |E_v|) = O(d)$ aufweisen.

3.3.4 Suchplanoptimierung

In Abschnitt 3.3.1 wurde der Vorverarbeitungsschritt des Algorithmus dargestellt, der eine Reihenfolge $(v_1, ..., v_n)$ der Knoten des Suchmusters erzeugt, in der die Knoten später in die partiellen Abbildung aufgenommen werden. Diese Folge ist mit den in Abschnitt 3.2.4 beschriebenen Suchplänen vergleichbar. In diesem Abschnitt wird eine Heuristik zur Optimierung entwickelt, die ebenfalls implementiert wurde. Bei der Optimierung wird auf eine Analyse einzelner Zielgraphen verzichtet und stattdessen allgemeine Informationen über die Häufigkeit von Knotenlabeln in Molekülgraphen verwendet. Dies hat den Vorteil, dass der gleiche Suchplan für alle TGI-Tests mit dem gleichen Suchmuster verwendet werden kann, und den Nachteil, dass er nicht für alle Instanzen optimal geeignet sein kann.

Der folgende Abschnitt verdeutlicht den Einfluss von derartigen Optimierungen im Zusammenhang mit dem vorgestellten Algorithmus anhand eines Beispiels. Nachfolgend wird gezeigt, wie sich Informationen über die Häufigkeit von Knotenlabeln im Vorverarbeitungsschritt des Algorithmus zur Optimierung nutzen lassen. In Abschnitt 3.4.1 werden typische Probleminstanzen analysiert, woraus die Häufigkeit von Knotenlabeln in den Molekülgraphen des Datensatzes abgeleitet werden kann.



(a) Suchmuster (b) Molekül

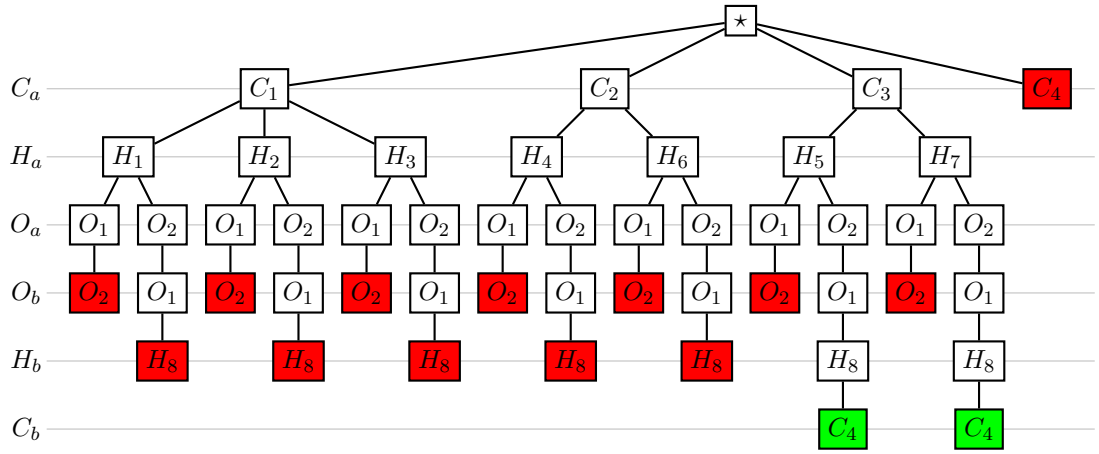
Abbildung 3.7: Beispiel für zwei Eingabeinstanzen: (a) zeigt ein Suchmuster, das in dem Molekül (b) vorhanden ist. Zur Unterscheidbarkeit sind Atome gleichen Typs mit Buchstaben bzw. Zahlen gekennzeichent.

Die Größe des Suchraums

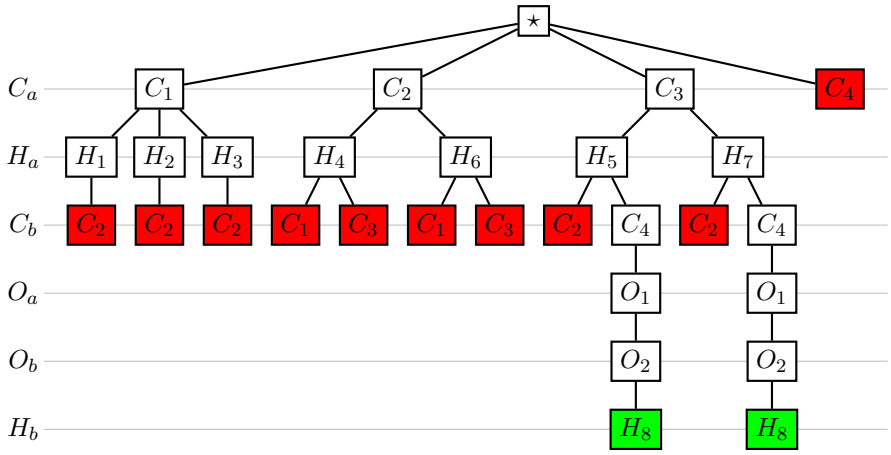
Der entscheidende Einfluss des Suchplans auf die zu erwartende Laufzeit soll anhand des Beispiels aus Abbildung 3.7 verdeutlicht werden. Dazu werden für drei unterschiedliche Folgen die vollständigen Suchbäume angegeben (Abbildung 3.8). Der komplette Suchbaum würde von Algorithmus 3.3 nur dann betrachtet, wenn nach dem Finden eines TGI nicht abgebrochen würde. Da der Suchraum nach dem Prinzip der Tiefensuche betrachtet wird, ist es im besten Falle möglich, dass ein Blatt, das einem zulässigen TGI entspricht, vollkommen ohne Backtracking gefunden wird. Existiert hingegen kein TGI, muss der Algorithmus jeden Knoten des Suchbaums betrachten. Daher ist es angemessen, einen generellen Zusammenhang zwischen der Größe des Suchbaums und der Laufzeit herzustellen.

Der i-ten Ebene des Baums kann der i-te Knoten der Folge $(v_1, ..., v_n)$ zugeordnet werden. Ein Knoten im Baum mit der Bezeichnung k eines Knotens im Zielgraphen auf der Ebene v_i steht dabei für die Abbildung von v_i auf k. Jedem Knoten des Suchbaums lässt sich eine partielle Abbildung zuordnen, die sich eindeutig durch den Pfad vom Knoten zur Wurzel ergibt. Die mit $\star$ gekennzeichnete Wurzel repräsentiert die leere Abbildung. Die Verzweigung eines Knotens entspricht der Anzahl Möglichkeiten, den nächsten Knoten der Folge der Abbildung hinzuzufügen, ohne dass die durch isFeasible(v,k) und isFeasibleConnected (v,k,E_v) geprüften Bedingungen verletzt werden. Jede Kante des Baums entspricht also einem möglichen Aufrufe der Funktion match(). Gibt es für einen Knoten keine solche Möglichkeit, handelt es sich um ein Blatt des Suchbaums (in Abbildung 3.8 rot eingefärbt). Blätter auf der untersten Ebene entsprechen solchen Abbildungen, die zu einem TGI gehören (in Abbildung 3.8 grün eingefärbt).

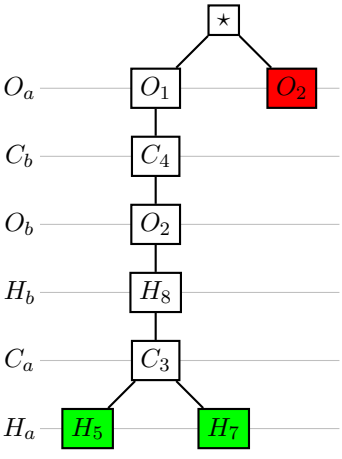
Wie der Suchbaum durch die Reihenfolge der Knoten beeinflusst wird, soll für das Beispiel in Abbildung 3.7 gezeigt werden. In Abbildung 3.8 sind die Suchbäume für drei unterschiedliche Suchpläne zu sehen. Abbildung 3.8(a) zeigt den Suchbaum für die Folge



(a)



(b)



(c)

Abbildung 3.8: Die vollständigen Suchbäume zu den Eingabeinstanzen aus Abbildung 3.7: (a) zeigt den Suchbaum zur Folge $(C_a, H_a, O_a, O_b, H_b, C_b)$, (b) zur maximal zusammenhängenden Folge $(C_a, H_a, C_b, O_a, O_b, H_b)$ und (c) für die optimierte maximal zusammenhängenden Folge $(O_a, C_b, O_b, H_b, C_a, H_a)$. Blätter, deren zugehörige partielle Abbildung nicht erweiterbar ist, sind rot markiert. Grün markierte Blätter entsprechen einem TGI.

$(C_a, H_a, O_a, O_b, H_b, C_b)$. Dabei handelt es sich nicht um eine maximal zusammenhängende Folge, da O_a und O_b mit keinem Knoten verbunden sind, der ihnen in der Folge vorausgeht. Für solche Knoten kommen alle Knoten des Zielgraphens in Frage, die die Funktion isFeasible(v,k) erfüllen. Da zwei O-Atome im Zielgraphen vorhanden sind, findet sich im Suchbaum eine Verzweigung auf der Ebene H_a. Ein noch wesentlich größerer Verzweigungsgrad würde entstehen, wenn die Folge mit unverbundenen, im Zielgraphen häufig auftretenden Atomtypen beginnen würde, wie z.B. $(H_a, H_b, ...)$.

Der Suchbaum der maximal zusammenhängenden Folge $(C_a, H_a, C_b, O_a, O_b, H_b)$ ist in Abbildung 3.8(b) zu sehen. Für die ersten beiden Ebenen ist er - genau wie die ersten zwei Elemente der Folge - mit dem des ersten Beispiels identisch. Für den vierten Knoten O_a existiert jetzt die Verbindungskante (O_a, C_b). Ein Knoten des Zielgraphen, auf den O_a abgebildet werden darf, muss eine solche Kante aufweisen, weshalb alle Erweiterungen für partielle Abbildungen mit $\varphi(C_b) \neq C_4$ fehlschlagen.

Abbildung 3.8(c) zeigt den Suchbaum, der sich zu der maximal zusammenhängenden Folge $(O_a, C_b, O_b, H_b, C_a, H_a)$ ergibt. Dieser Suchbaum ist offensichtlich günstig, da es keine starken Verzweigungen gibt und keine großen partiellen Abbildungen generiert werden können, die sich nicht zu einem TGI erweitern lassen.

Alle Suchbäume haben genau zwei grüne Blätter, die den beiden zulässigen Lösungen entsprechen. Die Anzahl der Knoten variiert hingegen stark. Die Beobachtungen motivieren einerseits die Verwendung maximal zusammenhängender Folgen, wie sie in Abschnitt 3.3.1 beschrieben wurden. Andererseits zeigen Abbildung 3.8(b) und 3.8(c), dass es auch für derartige Folgen erhebliche Unterschiede gibt. Günstig ist es offenbar, mit einem Knoten zu beginnen, dessen Label nur wenige Knoten des Zielgraphen aufweisen, wodurch der Verzweigungsgrad an der Wurzel minimiert wird. Gleiches gilt für den ersten Knoten einer neuen Zusammenhangskomponente. Die maximale Verzweigung auf der Ebene eines Knoten v mit einer Verbindungskante (u, v) ist durch die Anzahl der Nachbarn k von $\varphi(u)$, die isFeasible(v,k) erfüllen, bestimmt und somit durch den Grad des Knotens $\varphi(u)$ beschränkt. Auch hier erscheint es sinnvoll, einen Knoten mit einem Label, das im Zielgraphen nur selten vorkommt, einem Knoten mit einem häufigen Label vorzuziehen. Dies beruht allerdings auf der Annahme, dass Knoten, die relativ selten vorkommen, auch unter den Nachbarn eines festen Knotens $\varphi(u)$ vergleichsweise selten sind. Dies ist nicht zwangsläufig der Fall: Es ist durchaus denkbar, dass zwei unterschiedliche Atomtypen überproportional häufig benachbart auftreten. Daher soll die Wirksamkeit der Optimierung, die auf diesen Beobachtungen basiert, später in empirischen Experimenten überprüft werden.

Optimierung des Vorverarbeitungsschritts

Die Häufigkeit von Knotenlabeln soll genutzt werden, um heuristisch einen Suchplan zu erstellen, der möglichst für viele Zielgraphen gut geeignet ist. In Abschnitt 3.3.4 wurden

anhand eines Beispiels Ideen erarbeitet, wie eine Reihenfolge von Knoten optimiert werden kann. Auf dieser Grundlage wird der Vorverarbeitungsschritt aus Abschnitt 3.3.1 angepasst, um eine Knotenfolge mit günstigen Eigenschaften zu generieren.

Algorithmus 3.4 : Optimierter Vorverarbeitungsschritt TGI.

Eingabe : Graph $G = (V, E)$, Prioritätsfunktion $p : V \to \mathbb{R}$

Ausgabe : Folge $(v_1, ..., v_n)$ mit Verbindungskanten E_v für alle $v \in V$

Daten : Prioritätswarteschlange Q, wobei ein Knoten $v \in V$ Priorität $p(v)$ hat

▷ *Initialisiere wie Algorithmus 2, Zeile 1 bis 3*

1 $V' \leftarrow V$ absteigend sortiert nach Priorität

2 $Q \leftarrow \emptyset$ ▷ *Q ist Prioritätswarteschlange*

3 **forall** $u \in V'$ *in sortierter Reihenfolge* **do**

4 ⌊ ... ▷ *Unverändert gegenüber Algorithmus 2*

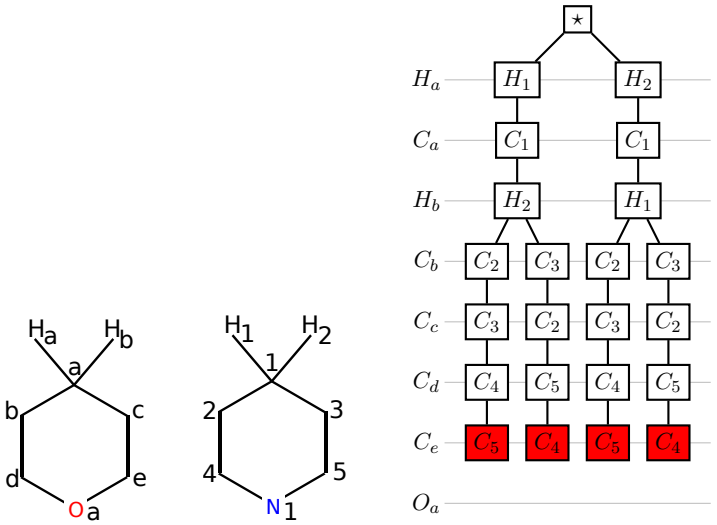
Ziel ist es, eine maximal zusammenhängende Folge zu generieren, bei der Knoten mit seltenen Labeln möglichst vor Knoten mit häufigen Labeln auftreten. Algorithmus 4 zeigt die angepasste Variante des Vorverarbeitungsschritts. Bei der ursprünglichen Version spielte es keine Rolle, in welcher Reihenfolge die Knoten durchlaufen werden (Algorithmus 2, Zeile 5) und welcher Knoten zuerst aus der Queue entnommen wird (Zeile 10). In diesen Fällen werden Knoten jetzt nach einer festen Priorität gewählt.

Sei $h : \Sigma \to \mathbb{N}$ eine Funktion, die einem Knotenlabel l die Anzahl der Knoten mit diesem Label im gesamten Datensatz zuordnet. Über eine Funktion $p : V \to \mathbb{R}$ sollen den Knoten Prioritäten zugeordnet werden. Eine sinnvolle Definition ist $p(v) = h(l(v))^{-1}$ für einen Knoten v mit Label $l(v)$[5]. Für Suchmuster mit Wildcards kann die Definition so erweitert werden, dass die Anzahl aller zutreffender Label berücksichtigt wird. Sei $c = \sum_{l \in \Sigma} h(l)$ die Anzahl aller Label, $L = \{s_1, s_2, ..., s_n\} \subseteq \Sigma$ eine Menge von Labeln. Es ergeben sich die Prioritäten

$$p(v) = \begin{cases} c^{-1} & \text{falls } v \text{ hat } \texttt{*}\text{-Wildcard} \\ (\sum_{l \in L} h(l))^{-1} & \text{falls } v \text{ hat } \texttt{[}s_1, s_2, ..., s_n\texttt{]}\text{-Wildcard} \\ (c - \sum_{l \in L} h(l))^{-1} & \text{falls } v \text{ hat } \texttt{![}s_1, s_2, ..., s_n\texttt{]}\text{-Wildcard} \\ h(l(v))^{-1} & \text{sonst.} \end{cases} \tag{3.9}$$

Die Laufzeit der vorausgehenden Sortierung der Knotenmenge beträgt $O(|V| \log |V|)$. Jeder Knoten wird ein einziges mal in die Prioritätswarteschlange eingefügt und einmal entnommen, wofür insgesamt ebenfalls die Laufzeit $O(|V| \log |V|)$ genügt, wenn die Priori-

[5] Für $h(l(v)) = 0$ sei $p(v) = \infty$.



(a) G_1 (b) G_2 (c) Suchbaum

Abbildung 3.9: Automorphismen der Graphen und symmetrische Äste des Suchbaums.

tätswarteschlange z.B. mit Heaps[6] implementiert wird [21]. Somit beträgt die Gesamtlaufzeit des optimierten Vorverarbeitungsschritts $O(|V| \log |V| + |E|)$.

3.3.5 Weitere mögliche Optimierungen

Die in diesem Abschnitt beschriebenen Optimierungen wurden im Rahmen der Diplomarbeit nicht in den Algorithmus integriert, sollen aber dennoch kurz vorgestellt werden.

Die in Abschnitt 3.2.6 beschriebene Ausnutzung von Automorphismen wurde in erster Linie verwendet, wenn alle Vorkommen des Suchmusters in einem Graphen gefunden werden sollen. Abbildung 3.9 zeigt das Suchmuster $G_1 = (V_1, E_1)$, einen Zielgraphen G_2 und den zugehörigen Suchbaum, der vom Algorithmus vollständig betrachtet würde. Für die Substruktursuche ist es von besonderem Interesse, die Automorphismen des Suchmusters zu bestimmen, da diese für alle späteren TGI-Tests verwendet werden können. Ein Automorphismus $\varphi : V_1 \to V_1$ von G_1 ist $\varphi(H_a) = H_b$, $\varphi(H_b) = H_a$ und $\varphi(v) = v$ für alle anderen Knoten v. Dieser Automorphismus könnte genutzt werden, um einen der symmetrischen Teilbäume des Suchbaums auf der Ebene H_b zu entfernen: Wenn kein TGI mit $\varphi(H_a) = H_1$ und $\varphi(H_b) = H_2$ existiert, kann auch kein TGI mit $\varphi(H_a) = H_2$ und $\varphi(H_b) = H_1$ existieren und umgekehrt.

Eine solche Optimierung wurde nicht in den Algorithmus eingebaut, da der Implementierungsaufwand den voraussichtlichen Nutzen kaum rechtfertigt: Zur Bestimmung der Automorphismen des Suchmusters könnte der entwickelte TGI-Algorithmus in leicht angepasster Form verwendet werden. Diese Methode ist hierfür allerdings nicht optimal geeignet

[6] Eine Heap-basierte Implementierung ist Teil der Java Runtime Environment.

und wäre für Graphen mit vielen Automorphismen nicht effizient. Eine Beschleunigung der Suche ist nur für stark symmetrische Suchmuster zu erwarten. Durchgeführte Test mit den vorhandenen Suchmustern haben ergeben, dass besonders viele Automorphismen nur benachbarte Wasserstoffatome "vertauschen" wie im Beispiel zu Abbildung 3.9(a). Aufgrund der verwendeten Suchplanoptimierung stehen diese häufigen Atome in der Regel am Ende der Folge, so dass keine tiefen symmetrischen Teilbäume eingespart werden würden.

Eine andere Lösung könnte darin bestehen, Wasserstoffatome in allen Molekülgraphen zu entfernen. Da diese stets Grad 1 haben, könnte die Anzahl der benachbarten Wasserstoffatome als Eigenschaft zu einem Atom gespeichert werden. Ein Knoten mit x Wasserstoffatomen dürfte dann nur auf einen Knoten mit y Wasserstoffatomen abgebildet werden, wenn $x \leq y$ gilt. Dadurch könnten die Molekülgraphen deutlich verkleinert werden und alle benachbarten Wasserstoffatome würden ohne eine Erkennung von Automorphismen immer als äquivalent betrachtet werden.

3.4 Experimenteller Vergleich

In diesem Abschnitt soll der neue Backtracking-Algorithmus experimentell untersucht werden, indem ein Vergleich mit frei verfügbaren Toolkits bzw. der vflib angestellt wird. In Abschnitt 3.4.1 werden der verwendete Moleküldatensatz und die Suchmuster beschrieben. Abschnitt 3.4.2 liefert einen Vergleich unterschiedlicher Algorithmen und in Abschnitt 3.4.3 wird die Wirksamkeit der Suchplanoptimierung überprüft. Im Abschnitt 3.4.4 wird der Einfluss von Wildcards auf die Laufzeit thematisiert.

Alle Experimente wurden auf einem Linux System mit einer 3 GHz Intel Core 2 Duo CPU und 4 GB RAM ausgeführt. Es wurden Java 1.6 und GCC 4.2 für C++-Implementierungen verwendet. Eine Optimierung zur Nutzung mehrerer Prozessorkerne wurde nicht vorgenommen.

3.4.1 Verwendete Instanzen

In diesem Abschnitt werden die Moleküle der Datenbank analysiert und die verwendeten Suchmuster beschrieben. Es wurde versucht, solche Instanzen auszuwählen, die in der Praxis auftretende Eingaben gut abdecken.

Analyse des Datensatzes

Der Moleküldatensatz besteht aus 187.266 Molekülen, die routinemäßig getestet werden, um Vorläufer von Wirkstoffen zu identifizieren, und umfasst somit realistische Instanzen, wie sie in der Praxis auftreten[7]. Es wurde die Häufigkeit von Knotenlabeln (bzw.

[7] Alle Instanzen wurden von Stefan Wetzel, Max-Planck-Institut für molekulare Physiologie in Dortmund, zur Verfügung gestellt.

Atomtypen), von Kantenlabeln (bzw. Bindungsarten) und der Knotengrade ermittelt (siehe Tabelle 3.4.1). Die Häufigkeit der Atomtypen ist aufgrund der Suchplanoptimierung von besonderem Interesse. Tabelle 3.1(a) ist zu entnehmen, dass der größte Teil der Atome Wasserstoff- oder Kohlenstoffatome sind. Ebenfalls relativ häufig kommen Sauerstoff- und Stickstoff vor, während kein anderes Elemente mehr als 1% aller Atome ausmacht. Die Verteilung erscheint günstig im Hinblick auf die Suchplanoptimierung, die in Abschnitt 3.4.3 experimentell untersucht wird. Tabelle 3.1(c) gibt Aufschluss über die Knotengrade der Molekülgraphen. Fast alle Knoten weisen einen Grad ≤ 4 auf und der durchschnittliche Grad von ca. 2 ist sehr gering. Betrachtet man die Laufzeit des neuen Algorithmus von $O(md^n)$ (siehe Abschnitt 3.3.3) wird deutlich, dass dieser vermutlich stark von dem geringen Grad der Instanzen profitiert.

Die Verteilung der Molekülgraphen nach Knoten- bzw. Kantenanzahl ist Abbildung 3.4.1 zu entnehmen. Die meisten Molekülgraphen des Datensatzes haben etwa 50 Knoten, einige weisen jedoch bis zu 200 Knoten auf. Die Verteilung nach Kantenanzahl ähnelt erwartungsgemäß der Verteilung nach Knotenanzahl.

Suchmuster

Die Suchmuster wurden ebenfalls zur Verfügung gestellt. Diese sind aus den Molekülen des Datensatzes generiert worden, indem Seitenketten gekürzt und anschließend eine Zerlegung nach dem Scaffold-Tree-Algorithmus (siehe Abschnitt 2.2) durchgeführt wurde. Auf diese Weise entstehen Molekülteile unterschiedlicher Größe mit kurzen Seitenketten, die realistische Suchmuster darstellen[8].

Suchmuster mit Wildcards mussten im Rahmen der Diplomarbeit erstellt werden. Diese sind anhand von nachfolgend erläuterten Regeln nach Absprache mit Stefan Wetzel aus den vorhandenen, festen Suchmustern generiert worden. Bei der Umformung zu variablen Suchmustern werden einige Atome durch die Wildcards `*`, `[N,O,S]`, `[C,H]`, `[Cl,Br,I]`, `![C,H]` und `![H]` ersetzt, indem zunächst zufällig eine der Wildcards ausgewählt wird, wobei `*` nur mit sehr geringer Wahrscheinlichkeit gezogen wird. Im nächsten Schritt wird ein Atom des Suchmusters ausgewählt, dessen Label durch die Wildcard ersetzt werden soll. Dabei werden nur solche Label ausgesucht, die der Wildcard entsprechen, d.h. z.B., dass `[C,H]` nur ein bestehende `C` oder `H` Label ersetzen darf. Eine weitere Bedingung lässt sich aus den Regeln der Chemie ableiten: Da ein H-Atom z.B. nur eine Bindung eingehen kann, ergibt die Wildcard `[C,H]` nur an einem Atom im Suchmuster Sinn, das höchstens eine Bindung aufweist. Aufgrund dessen sind alle Wildcards nur an Atomen mit einer Bindung erlaubt mit Ausnahme von `*` (beliebiges Atom) und `[N,O,S]` (Atom mit bis zu zwei Bindungen). Zusätzlich werden bestimmte Atome in der Nachbarschaft einer `[N,O,S]`-Wildcard ausgeschlossen, wenn derartige Muster im Allgemeinen nicht in Molekülen auftreten. Für die

[8] Persönliche Verständigung mit Stefan Wetzel.

Tabelle 3.1: Eigenschaften der Moleküle des Datensatzes.

(a) Häufigkeit der Atomtypen.

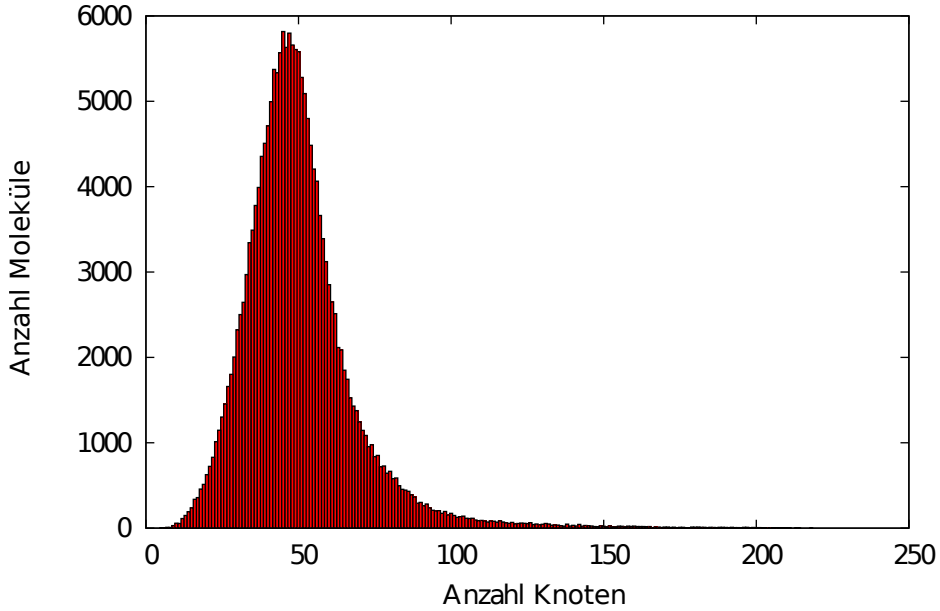
Element	Vorkommen	Anteil (in %)	Element	Vorkommen	Anteil (in %)
H	4238982	45,122833	Ge	24	0,000255
C	3834810	40,820530	Sb	13	0,000138
O	610408	6,497630	Pb	10	0,000106
N	489704	5,212769	Cr	8	0,000085
S	87881	0,935470	Bi	8	0,000085
F	60154	0,640323	W	7	0,000075
Cl	49092	0,522571	Ru	4	0,000043
Br	13355	0,142160	Ti	4	0,000043
Si	3138	0,033403	Mo	3	0,000032
P	3063	0,032605	Ga	3	0,000032
I	2242	0,023865	Tl	3	0,000032
B	545	0,005801	In	2	0,000021
Se	225	0,002395	Zr	2	0,000021
Sn	147	0,001565	V	1	0,000011
Al	42	0,000447	Hf	1	0,000011
As	30	0,000319	Rh	1	0,000011
Hg	30	0,000319	Tc	1	0,000011
Te	29	0,000309			

(b) Häufigkeit der Bindungstypen.

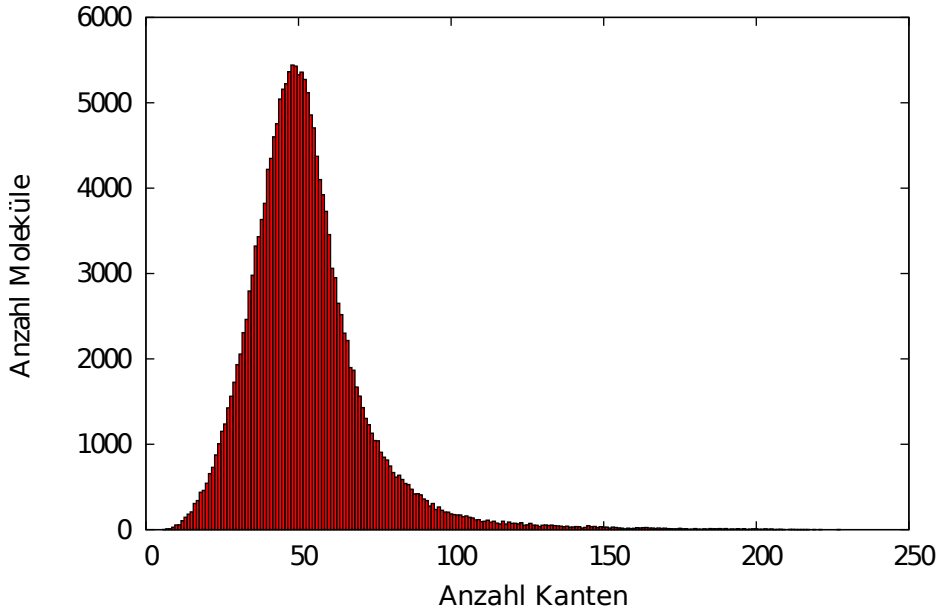
Bindung	Vorkommen	Anteil (in %)
aromatisch	2261621	23,093655
einfach	7068569	72,177918
doppel	449985	4,594845
dreifach	13082	0,133582

(c) Häufigkeit der Knotengrade (durchschnittlich 2,085).

Grad	Vorkommen	Anteil (in %)
1	4727249	50,320305
2	472717	5,031946
3	2863632	30,482599
4	1330674	14,164670
5	31	0,000330
6	14	0,000149



(a) Verteilung der Molekülgraphen nach Anzahl Knoten (durchschnittlich $50,2$)



(b) Verteilung der Molekülgraphen nach Anzahl Kanten (durchschnittliche $52,3$)

Abbildung 3.10: Verteilung der Molekülgraphen des Datensatzes.

gewählte Wildcard wird zufällig eine erlaubte Position ausgewählt und das Label durch die Wildcard ersetzt. Existiert keine erlaubte Position, wird eine neue, andere Wildcard gezogen. Ferner werden für einige Doppelbindungen beliebige Bindungstypen erlaubt. Die Anzahl der Wildcards richtet sich nach der Größe des Suchmusters. Für ein Suchmuster mit n Knoten werden $\lceil n/9 \rceil$ Wildcards für Knoten eingefügt und $\lfloor d/3 \rfloor$ Wildcards für Kanten, wobei d die Anzahl der Doppelbindungen ist. Auf diese Weise sollen Suchmuster mit Wildcards entstehen, die praxisrelevante Eingaben so gut wie möglich nachbilden. Die restriktiven Vorgaben erlauben es kaum, die Anzahl der Wildcards stark zu variieren. Alle Wildcard-Suchmuster wurden nach dieser Vorgehensweise generiert.

3.4.2 Vergleich der Verfahren

Der neue Algorithmus wird in diesem Abschnitt mit anderen Algorithmen aus verfügbaren Cheminformatik Toolkits und der vflib verglichen. Im CDK ist ein auf MCS-basierender Algorithmus enthalten, der in Abschnitt 3.2.1 beschrieben ist. Außerdem wurde die C++-Bibliothek CDL zu Testzwecken herangezogen, die eine Implementierung des Algorithmus von Ullmann (Abschnitt 3.2.2) beinhaltet. Die C++-Bibliothek vflib wurde ebenfalls in die Experimente einbezogen. Dazu wurden die Moleküle mit Hilfe des Toolkits OpenBabel eingelesen und die Datenstruktur für Graphen der vflib daraus aufgebaut. Die ungerichteten Kanten der Molekülgraphen wurden dabei durch zwei gerichtete Kanten repräsentiert. Der in OpenBabel implementierte Backtracking-Algorithmus konnte nicht getestet werden, da dieser Suchmuster nur im SMARTS-Format akzeptiert. Eine Schwierigkeit bei der Vergleichbarkeit ist, dass unterschiedliche Bibliotheken das Austauschformat für Moleküle nicht vollkommen identisch interpretieren bzw. aromatische Bindungen auf unterschiedliche Weise erkennen. Hinzu kommt, dass der verwendete Suchplan im einfachen Fall von der Reihenfolge der Knoten im Graphen beeinflusst wird, die wiederum davon abhängt, wie das Molekül eingelesen wird. Da sich Bibliotheken auch hier unterscheiden können und der Suchplan vermutlich großen Einfluss auf die Laufzeit hat (vgl. Abschnitt 3.3.4) wurde die Reihenfolge der Knoten des Graphen nach dem Einlesen mit Hilfe einer Zufallsfunktion neu bestimmt. Zur besseren Vergleichbarkeit des neuen Algorithmus mit dem VF2-Algorithmus wurde dieser zusätzlich in Java implementiert und so angepasst, dass er direkt mit ungerichteten Graphen arbeitet und von der neu entwickelten Suchplanoptimierung ebenfalls profitieren kann.

Aus dem Molekül datensatz wurden 2000 Moleküle zufällig ausgewählt und aus den Suchmustern ohne Wildcards je 20 der Größe 5, 8, 11, 14, 17, 20, 25, 30, 40, 60, 80, 100 zufällig bestimmt. Jedes Suchmuster wurde in allen 2000 Molekülen gesucht, so dass insgesamt für jeden Algorithmus 480.000 TGI-Tests durchgeführt wurden. Besonders realistische Suchmuster dürften eher 30 oder weniger Kanten aufweisen, weshalb mehr Gruppen in diesen Bereich fallen. Größere Suchmuster sind vermutlich nur in wenigen der Molekül-

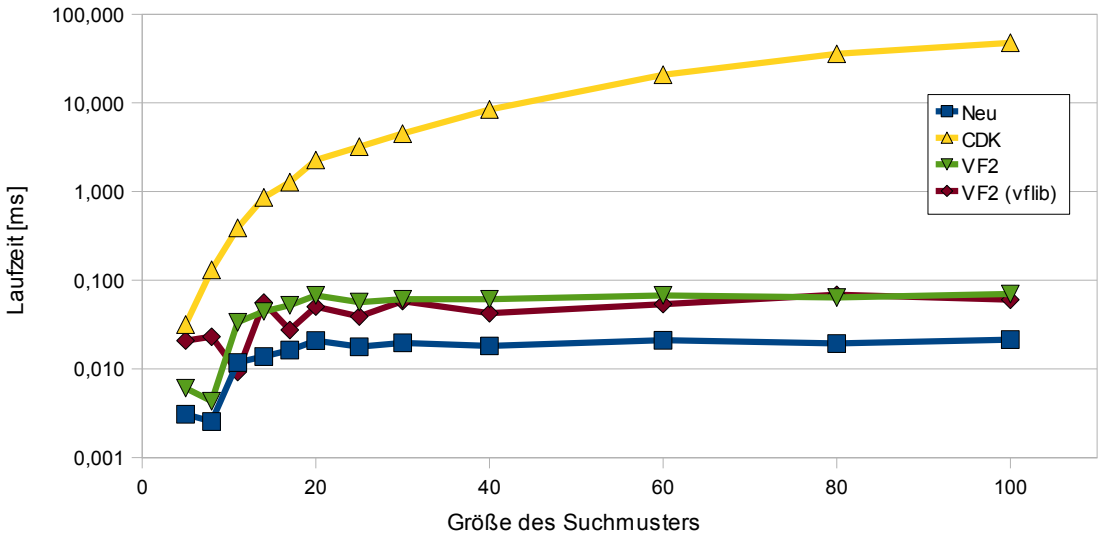


Abbildung 3.11: Laufzeit der TGI-Algorithmen (Durchschnittswert für einen TGI-Test).

graphen vorhanden, sind aber für die Entwicklung der Laufzeit von Interesse. Keiner der Algorithmen prüft die Größe der Instanzen vorab, um einen TGI auszuschließen[9], so dass die Messergebnisse hierdurch nicht verfälscht werden.

Abbildung 3.11 zeigt die Laufzeiten der unterschiedlichen Algorithmen. Aufgrund der starken Unterschiede in der Laufzeit wurde eine logarithmische Skalierung der y-Achse gewählt. Es zeigt sich, dass der CDK-Algorithmus deutlich langsamer ist als die anderen Verfahren. Zudem nimmt die Laufzeit mit zunehmender Größe des Suchmusters stark zu. Die eigene VF2-Implementierung und die der vflib weisen insgesamt eine sehr ähnliche Laufzeit auf, wobei die vflib-Version etwas besser abschneidet, obwohl die Modellierung ungerichteter Kanten mit Hilfe von zwei gerichteten Kanten zu einem Mehraufwand führen sollte. Dies kann möglicherweise mit der höheren Effizienz von C++ gegenüber Java begründet werden. Generell liegen beide Kurven jedoch nah zusammen, was für die gute Vergleichbarkeit der Implementierungen spricht. Der neue Algorithmus schneidet im Vergleich am besten ab und seine Laufzeit liegt deutlich unterhalb der von VF2. Für die eigene VF2-Implementierung und den neuen Algorithmus wurde die Anzahl der generierten partiellen Abbildungen ermittelt. Im Durchschnitt aller TGI-Tests des Experiments beträgt dieser Wert für den VF2-Algorithmus $57,9$ und für den neuen Algorithmus bei $60,5$. Der bessere Wert des VF2-Algorithmus lässt sich auf das Pruning mittels Terminalmengen zurückführen. Der Unterschied fällt jedoch gering aus, was für die schlechte Wirksamkeit dieser Pruningstrategie für TGI in Molekülgraphen spricht. Betrachtet man die deutlich schlechtere Laufzeit des VF2-Algorithmus, scheint sich der Mehraufwand für den Aufbau der Terminalmenge und das Auswerten der Look-Ahead-Regeln kaum zu lohnen.

[9] Für die vflib wurde diese Überprüfung deaktiviert.

Alle direkten Backtracking-Algorithmen weisen eine Zunahme der Laufzeit bis zu einer Größe des Suchmusters von ca. 20 auf, während die Laufzeit danach konstant bleibt. Dies unterscheidet sich von dem Verhalten des CDK-Algorithmus. Eine Erklärung für dieses Verhalten ist, dass der MCS-basierte CDK Algorithmus einen Assoziationsgraphen aufbauen muss, dessen Größe stark von der Größe des Suchmusters beeinflusst wird. Die konstante Entwicklung der Laufzeit für die Backtracking-Algorithmen kann damit begründet werden, dass es häufig ausreicht, einen kleinen Teil des Suchmusters zu betrachten, um einen TGI auszuschließen. Die Größe dieses relevanten Teils liegt möglicherweise meist unter 20, so dass eine Vergrößerung des Suchmusters nicht zu einem Mehraufwand führt.

Der Algorithmus von Ullmann in der Implementierung der CDL wurde nach über 10 Stunden Laufzeit abgebrochen, ohne dass der Testlauf beendet werden konnte. Diese ausgesprochen schlechte Laufzeit ist überraschend angesichts der Tatsache, dass der Algorithmus häufig für die Substruktursuche empfohlen wird. Daher wurde zusätzlich die Ullmann-Implementierung der vflib[10] unter den gleichen Bedingungen getestet. Auch hier konnte der Test nach über 10 Stunden nicht abgeschlossen werden. Eine Erklärung für die schlechte Laufzeit könnte sein, dass der Algorithmus keine festen Vorgaben über die zu verwendende Knoten-Reihenfolge macht. In der Veröffentlichung [65] werden die Knoten in den Experimenten nach absteigendem Grad sortiert, wodurch die Wirkung der Refinement Procedure erhöht werden soll. Dies mag bei dichten Graphen sinnvoll sein, für dünne Graphen scheint es jedoch angemessener, stets zusammenhängende Knoten der Reihe nach abzuarbeiten. Dies wird bei beiden Ullmann-Implementierungen nicht berücksichtigt, was die sehr schlechten Laufzeiten zur Folge haben könnte.

3.4.3 Wirksamkeit der Suchplanoptimierung

Während der neue Algorithmus in den Experimenten bisher eine maximal zusammenhängende Folge von Knoten verwendet hat, die nicht optimiert worden ist, wird in diesem Kapitel der Einfluss der Optimierung nach Abschnitt 3.3.4 untersucht. Die Optimierung ist für den neuen Algorithmus und auch für die eigene VF2-Implementierung anwendbar. Andere Algorithmen werden in diesem Abschnitt nicht mehr berücksichtigt.

Als Grundlage der Optimierung dient anwendungsspezifisches Wissen über die Häufigkeit von Knotenlabeln. Es ist jedoch nicht angemessen, zu erwarten, dass die Werte des Datensatzes aus Tabelle 3.1(a) für alle Moleküldatenbanken zutreffend sind. Dennoch kann man davon ausgehen, dass gewisse Übereinstimmungen übertragen werden können. Für die hier durchgeführten Experimente wurde die Häufigkeitsfunktion auf einem Teil des Datensatzes, bestehend aus 5000 zufällig ausgewählten Molekülen, ermittelt. Die Experimente wurden anschließend auf 2000 anderen Molekülen des Datensatzes ausgeführt. Die

[10] Die Implementierung ist eine für ITGI angepasste Variante des unter Abschnitt 3.2.2 beschriebenen Algorithmus und deshalb nicht exakt vergleichbar.

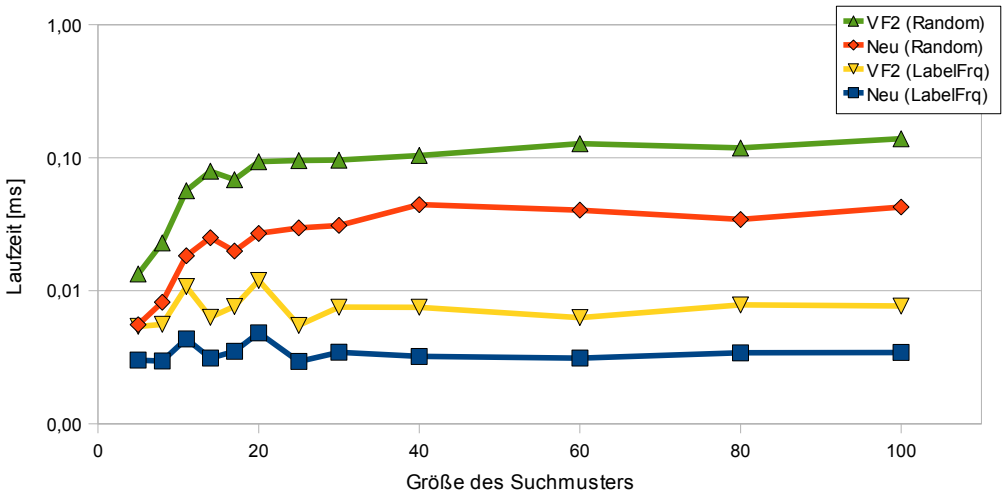
aus dem Teildatensatz ermittelte Häufigkeitsfunktion ähnelt der Tabelle 3.1(a) insofern, dass die Reihenfolge der Elemente, sortiert nach Häufigkeit, für die ersten 14 Einträge identisch ist. Unterschiede treten nur bei den sehr seltenen Elemente auf, wobei viele gar nicht im Teildatensatz auftreten.

In den folgenden Experimenten wurden 2000 Instanzen des Moleküldatensatzes zufällig ausgewählt, wobei nur Instanzen der Größe 15, 20, 25,..., 115 verwendet und zu jeder Größe maximal 100 Instanzen gewählt wurden. Diese Auswahl spiegelt nicht die für den gesamten Datensatz typische Verteilung wieder (siehe Abbildung 3.10(b)), ist aber für eine genauere Analyse der Abhängigkeit zwischen Laufzeit und der Größe des Zielgraphen sinnvoll. In diesen Experimenten wurde jeder TGI-Test 10 mal wiederholt und die Laufzeit gemittelt, um genauere Ergebnisse zu erhalten.

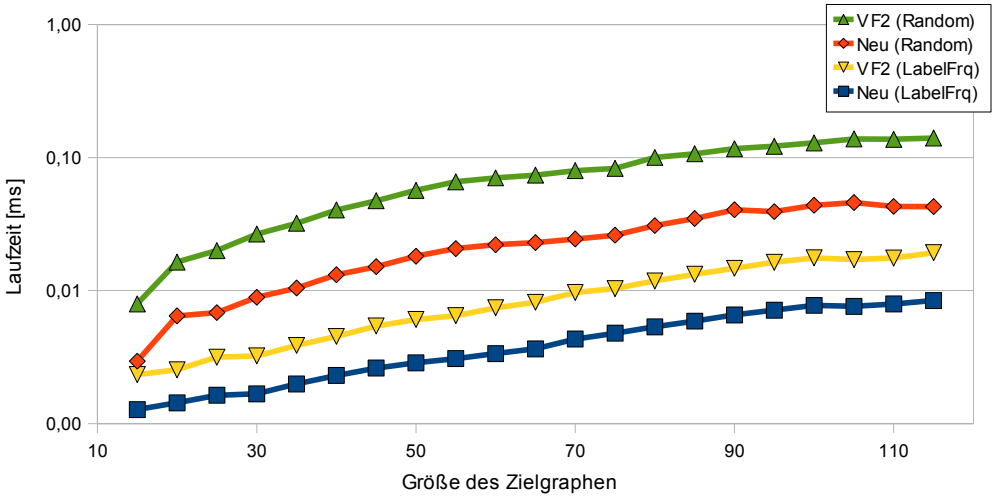
Abbildung 3.12 zeigt die ermittelten Laufzeiten in Abhängigkeit zur Größe des Suchmusters bzw. des Zielgraphen für beide Algorithmen mit optimiertem Suchplan (LabelFrq) und zufälligem Suchplan (Random). Es wurde wiederum eine logarithmische Skalierung der y-Achse gewählt, um dem großen Unterschieden in der Laufzeit Rechnung zu tragen. Es zeigt sich, dass der VF2-Algorithmus genau wie der neue Algorithmus stark von der Suchplanoptimierung profitieren. Abbildung 3.12(a) ist zu entnehmen, dass die Größe des Suchmusters kaum mehr einen Einfluss auf die Laufzeit hat, wenn die Optimierung angewandt wird. Dies lässt sich wiederum mit dem bereits im vorausgehenden Abschnitt 3.4.2 angeführten Argumente begründen, dass nur ein kleiner Teil des Suchmusters vom Algorithmus betrachtet werden muss, um einen TGI auszuschließen. Durch die Suchplanoptimierung wurde dieser relevante Teil des Suchmusters offenbar deutlich verkleinert. Dies steht im Einklang mit der Vermutung aus Abschnitt 3.3.4, dass es sinnvoll ist, die partielle Abbildung mit Knoten zu beginnen, die ein "seltenes" Label aufweisen.

In Abbildung 3.12(b) ist die Abhängigkeit der Laufzeit zur Größe des Zielgraphen zu sehen. Für alle Varianten nimmt die Laufzeit mit der Größe des Zielgraphen auf ähnliche Weise zu. Dies entspricht dem zu erwartenden Verlauf, da ein größerer Zielgraph mehr Möglichkeiten für partielle Abbildungen bietet, die betrachtet werden müssen. Beide Diagramme bestätigen die Wirksamkeit der Suchplanoptimierung und die gute Laufzeit des neuen Backtracking-Algorithmus.

Außerdem wurde eine Unterscheidung danach durchgeführt, ob ein TGI gefunden wurde oder nicht. Bei der zufälligen Wahl einer maximal zusammenhängenden Folge konnte festgestellt werden, dass die Laufzeit für positiv getesteten Instanzen eher geringer ausfällt als für Instanzen, die das Suchmuster nicht enthalten. Dies kann damit begründet werden, dass der Algorithmus abgebrochen werden kann, wenn das Suchmuster gefunden wurde. Für große Suchmuster scheint sich diese Tendenz umzukehren, was aufgrund der geringen Anzahl an positiv getesteten Instanzen für Suchmuster dieser Größenordnung jedoch genauer untersucht werden müsste. Eine solche Entwicklung erscheint jedoch schlüssig, wenn



(a) Zusammenhang mit der Größe des Suchmusters



(b) Zusammenhang mit der Größe des Zielgraphen

Abbildung 3.12: Einfluss der Suchplanoptimierung.

man bedenkt, dass der Algorithmus für viele Instanzen, die das Suchmuster nicht enthalten, einen TGI ausschließen kann, bevor eine große partielle Abbildung aufgebaut wurde.

Wird die Reihenfolge zuvor optimiert, so weisen die positiv getesteten Instanzen generell eher eine höhere Laufzeit auf, die aber immer noch weit unter der bei einer nicht optimierten Reihenfolge liegt. Dies kann damit begründet werden, dass durch die Optimierung solche Instanzen, die das Suchmuster nicht enthalten, sehr früh ausgeschlossen werden können. Die Beobachtungen treffen auf den neuen Backtracking Algorithmus und VF2 gleichermaßen zu. Eine Schwierigkeit bei der Unterscheidung nach positiv und negativ getesteten Instanzen besteht darin, dass besonders große Suchmuster im zufällig gewählten Datensatz nur in wenigen Molekülgraphen gefunden werden können, was bei 240 unterschiedlichen Suchmustern unvermeidlich ist. Aufgrund dessen wäre es von Interesse, diese Tendenzen in weiteren Experimenten genauer zu untersuchen.

3.4.4 Suchmuster mit Wildcards

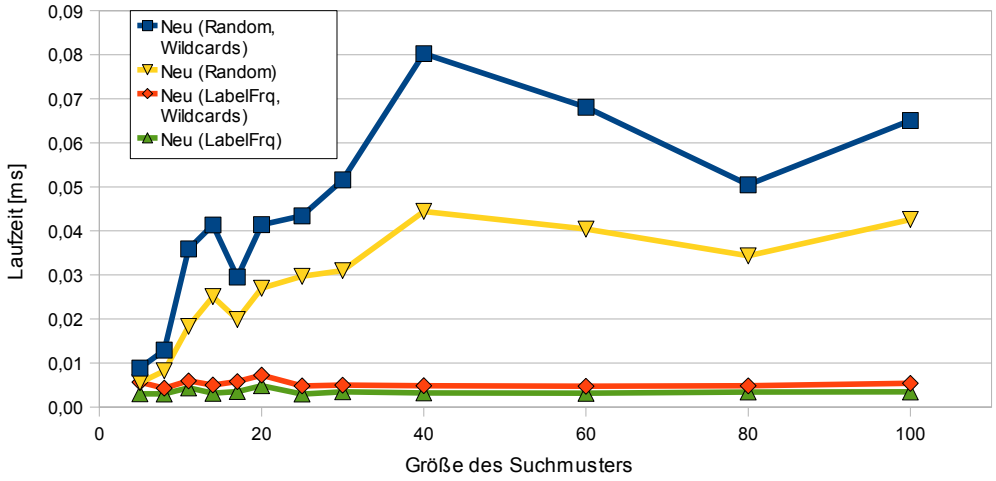
Die Experimente des letzten Abschnitts wurden wiederholt, wobei die Suchmuster diesmal nach dem in Abschnitt 3.4.1 erläuterten Verfahren mit Wildcards versehen wurden.

Abbildung 3.13 zeigt die gemessenen Laufzeiten sowie die Ergebnisse aus dem vorherigen Experiment ohne Wildcards für den neuen Backtracking-Algorithmus. Durch das Hinzufügen von Wildcards hat sich die Laufzeit erhöht, was leicht damit zu erklären ist, dass variable Suchmuster mehr mögliche partielle Abbildungen erlauben. Die Laufzeit verschlechtert sich für den zufälligen Suchplan mit zunehmender Größe des Zielgraphen (siehe Abbildung 3.13(b)) besonders stark. Auffällig ist, dass der Unterschied bei einem nichtoptimierten Suchplan sehr deutlich ausfällt, während für die optimierte Variante kaum eine Veränderung eintritt. Dies kann damit begründet werden, dass den Knoten mit einem Wildcard-Label im optimierten Vorverarbeitungsschritt angemessene Prioritäten zugeteilt werden und diese ggf. in der Reihenfolge nach hinten verschoben werden. Die Suchplanoptimierung nutzt die verbleibenden markanten Label so gut, dass kaum eine Verschlechterung der Laufzeit eintritt.

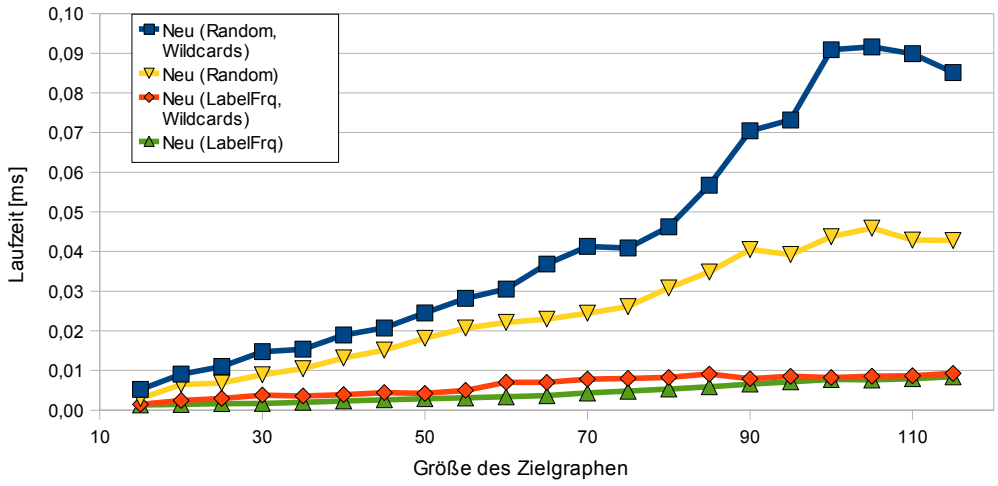
3.4.5 Weitere Laufzeiten

Der neue Backtracking-Algorithmus berechnet in einem Vorverarbeitungsschritt einen Suchplan, der zusätzlich optimiert werden kann. Abbildung 3.14 zeigt die Laufzeit für beide Varianten der Vorverarbeitung. Durch die Optimierung verlängert sich die Laufzeit deutlich. Dennoch ist die benötigte Zeit von unter $0,03$ Millisekunden insgesamt sehr gering und zahlt sich schnell aus, da der Vorverarbeitungsschritt nur ein einziges Mal durchgeführt werden muss und alle späteren TGI-Test davon profitieren.

Für den praktischen Einsatz ist außerdem die Zeit von Interesse, die benötigt wird, um einen Molekülgraphen zu laden. Die Moleküle sind im SDfile-Format gespeichert und wer-



(a) Zusammenhang mit der Größe des Suchmusters



(b) Zusammenhang mit der Größe des Zielgraphen

Abbildung 3.13: Laufzeit für Suchmuster mit Wildcards.

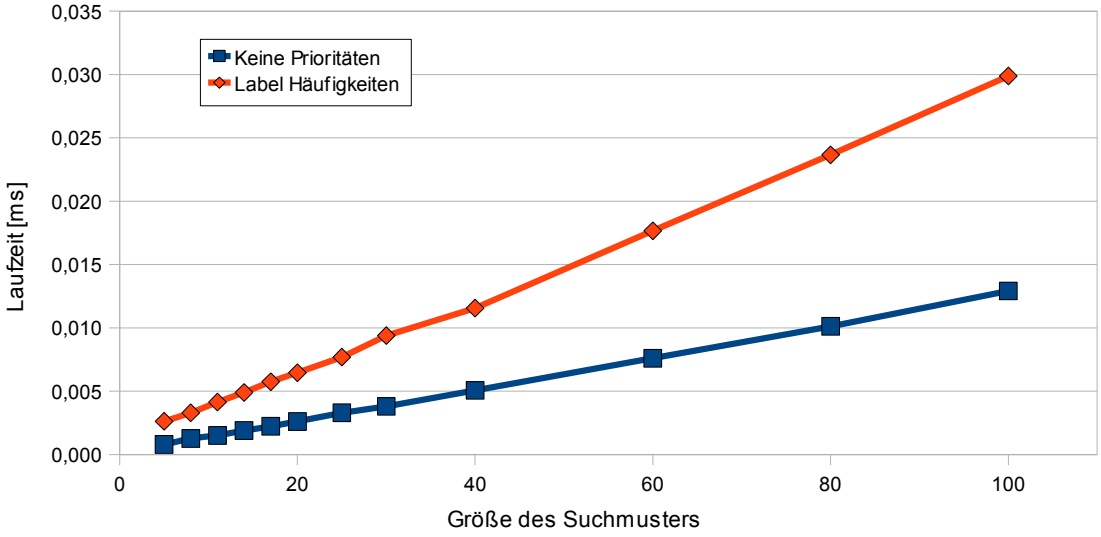


Abbildung 3.14: Laufzeit des Vorverarbeitungsschritts.

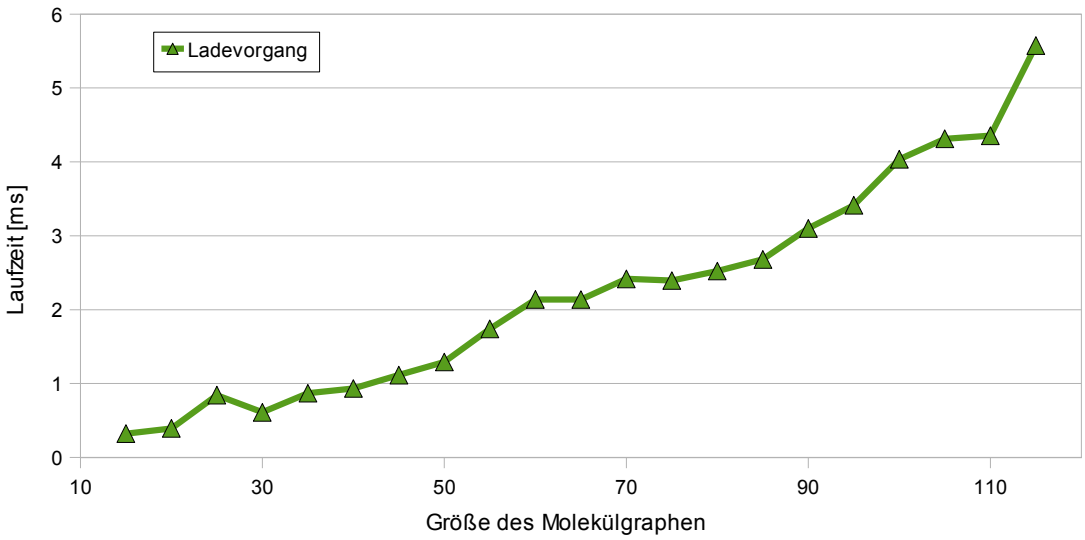


Abbildung 3.15: Laufzeit zum Laden eines Molekülgraphen.

den mit Hilfe der Bibliothek CDK ausgelesen, wobei zudem die Aromatizität der Moleküle berechnet wird. Anschließend wird daraus eine Datenstruktur für einen Molekülgraphen berechnet. Abbildung 3.15 kann die dazu benötigte Laufzeit entnommen werden. Diese überschreitet deutlich die Laufzeit eines TGI-Tests und bremst die Substruktursuche dadurch aus. Eine genauere Analyse hat ergeben, dass die schlechte Laufzeit in erster Linie auf die Erkennung aromatischer Bindungen zurückzuführen ist. Diese Angaben sind dem SDfile nicht direkt zu entnehmen und müssen daher berechnet werden. Ein SDfile beinhaltet außerdem Informationen (wie z.B. Koordinaten von Atomen), die für den Aufbau eines

Graphen nicht erforderlich sind. Daher scheint das Format nicht optimal geeignet zu sein, um Moleküle für die Substruktursuche zu speichern. Da die Datenbank von Scaffold Hunter jedoch dieses Format verwendet, müssen die Ladezeit vorerst in Kauf genommen werden. Tests mit dem wesentlich kompakteren SMILES-Format ergaben zudem keine deutliche Verbesserung der Laufzeit. Eine erhebliche Beschleunigung könnte durch ein neues Format erreicht werden, das alle für die Substruktursuche benötigten Informationen bereit hält, und effizient geladen werden kann.

3.4.6 Fazit

Der experimentelle Vergleich macht deutlich, dass der neue Backtracking-Algorithmus sehr gute Laufzeiten bei der Suche in Molekülgraphen liefert. Keiner der Algorithmen konnte im Vergleich eine ähnlich gute Laufzeit erzielen. Ferner konnte gezeigt werden, dass die erarbeitete Suchplanoptimierung die Laufzeit erheblich verbessert. Besonders vorteilhaft wirkt sich der optimierte Vorverarbeitungsschritt bei Instanzen mit Wildcards aus. Insgesamt hat die Untersuchung ergeben, dass das NP-vollständige TGI-Problem auf Molekülgraphen in der Praxis effizient gelöst werden kann, wenn die speziellen Eigenschaften dieser Graphen wie der geringe Knotengrad und die Verteilung der Label ausgenutzt werden. Aufgrund der günstigen Laufzeit von TGI-Tests wird das Laden des Molekülgraphen zum ausschlaggebenden Faktor bei der Substruktursuche.

Kapitel 4

Indizes für die Suche in Graphen

Im vorangehenden Kapitel 3 wurde ein Algorithmus vorgestellt, der zwei Graphen darauf überprüft, ob der eine in dem anderen enthalten ist. Werden alle Graphen einer Datenbank gesucht, die ein gegebenes Suchmuster aufweisen, so muss der Algorithmus für jeden Graphen einmal ausgeführt werden. Hier wird die Größe der Datenbank zum entscheidenden Faktor für die Zeit, die zur Beantwortung einer Suchanfrage benötigt wird. Betrachtet man ein Suchmuster und eine Strukturformel, fällt es in in einigen Fällen leicht, einen Teilgraph-Isomorphismus auszuschließen, z.B. weil das Suchmuster einen Atomtyp aufweist, der in der Strukturformel an keiner Stelle vorkommt. In diesem Kapitel wird ein Verfahren vorgestellt, das auf ähnliche Weise vorgeht, um ohne großen Aufwand einen Teil der Datenbankgraphen von vornherein auszuschließen. Dies steigert die Effizienz der Suche und verspricht auch in sehr großen Datenbanken akzeptable Laufzeiten.

Im Abschnitt 4.1 werden grundlegende Ideen erläutert, auf denen viele bekannte Verfahren aufbauen. Abschnitt 4.2 stellt einige Ansätze detaillierter vor und Abschnitt 4.3 liefert eine Einschätzung der Möglichkeiten, diese so zu erweitern, dass variable Suchmuster verarbeitet werden können. In Abschnitt 4.4 wird ein neues Verfahren beschrieben, das schließlich im Abschnitt 4.5 experimentell untersucht wird.

4.1 Theoretische Grundlagen

Die in der Literatur beschriebenen Methoden verwenden mehrheitlich ein als *Filter-Verifikations-Ansatz* bekanntes Verfahren [32, 73, 77]: In einem ersten Schritt werden Graphen herausgefiltert, die auf keinen Fall das Suchmuster enthalten. Alle anderen Graphen bilden die Kandidatenmenge. Diese umfasst sämtliche Datenbankgraphen, die den gesuchten Graphen enthalten, und zusätzlich möglicherweise weitere, die das Suchmuster nicht enthalten (*False-Positives*). Daraus resultiert die Notwendigkeit der Verifikation: Alle Graphen der Kandidatenmenge werden mittels eines TGI-Tests daraufhin überprüft, ob sie das Suchmu-

ster tatsächlich enthalten. Abbildung 1.2 veranschaulicht den Ablauf der Substruktursuche und ordnet die Filterphase ein.

Die effiziente Berechnung der Kandidatenmenge wird dadurch ermöglicht, dass in einem Vorverarbeitungsschritt alle Graphen der Datenbank in einer Datenstruktur, dem *Index*, aufgenommen werden. Dieser Vorgang wird nur ein einziges Mal ausgeführt und der Index kann anschließend genutzt werden, um alle späteren Suchanfragen zu beschleunigen. Eine Aktualisierung ist nur notwendig, wenn neue Graphen der Datenbank hinzugefügt werden.

Im Folgenden wird die Auswirkung auf die Laufzeit der Suche formalisiert. Dabei wird diese Notation verwendet:

$\mathcal{G}$ - Menge aller Graphen

$\mathcal{D} \subset \mathcal{G}$ - Menge der Graphen in der Datenbank

$Q \in \mathcal{G}$ - Suchmuster

$\mathcal{D}_G = \{H \in \mathcal{D} | G \lesssim H\}$ - die Menge der Datenbankgraphen, die G enthalten

$\mathcal{C}_Q \subseteq \mathcal{D}$ - Kandidatenmenge für das Suchmuster Q

$\mathcal{D}_Q \subseteq \mathcal{C}_Q$ - Ergebnismenge

4.1.1 Kostenmodell

Für den Nutzer eines Systems zur Substruktursuche ist die Antwortzeit auf eine Suchanfrage entscheidend - also die Zeit, die das System benötigt, um das Ergebnis der Anfrage zu liefern. Mit dem vorgestellten Filter-Verifikations-Ansatz hängt diese Größe nicht mehr ausschließlich von der Anzahl der Graphen in der Datenbank und den beteiligten Graphen ab, sondern auch von der Größe der Kandidatenmenge. Yan et al. stellen in [73] folgende Formel zur Berechnung der Antwortzeit auf, die dies berücksichtigt:

$$T = T_s + |\mathcal{C}_Q|(T_{io} + T_{tgi}) \tag{4.1}$$

Dabei ist T_s die Laufzeit der Filterphase, T_{io} die Zeit, die das Laden eines Graphen aus der Datenbank durchschnittlich beansprucht, und T_{tgi} die durchschnittliche Laufzeit des TGI-Algorithmus. Die Experimente in Abschnitt 3.4 erlauben eine Einschätzung der Größen T_{tgi} und T_{io}.

Eine echte Ersparnis durch den Einsatz eines Filter-Verifikations-Systems liegt also nur dann vor, wenn auf so viele I/O-Operationen und TGI-Tests verzichtet werden kann, dass die dadurch eingesparte Zeit die Laufzeit für die Suche im Index T_s aufwiegt. Ein wesentlicher Faktor dabei ist die Größe der Kandidatenmenge $\mathcal{C}_Q$. Die Größe der Ergebnismenge $\mathcal{D}_Q$ ist eine untere Schranke für die Größe der Kandidatenmenge, da $\mathcal{D}_Q \subseteq \mathcal{C}_Q$ gilt. Da üblicherweise $|\mathcal{D}_Q| \ll |\mathcal{D}|$ ist, kann mit einer Verkürzung der Antwortzeit gerechnet werden,

wenn in solchen Fällen auch $|\mathcal{C}_Q| \ll |\mathcal{D}|$ gilt. Die Eigenschaft eines Index, eine Kandidatenmenge mit wenigen "falsch positiven" Graphen zu liefern, wird als seine *Effektivität* bezeichnet.

In der Literatur wird die Effektivität auf unterschiedliche Art und Weise gemessen. Zou et al. [77] verwenden ein Maß, das als $\frac{|\mathcal{D}|-|\mathcal{C}_Q|}{|\mathcal{D}|-|\mathcal{D}_Q|}$ definiert ist und *Pruning Power* genannt wird. Ein optimaler Index erreicht dabei den Wert 1, niedrigere Werte sprechen für eine schlechtere Effektivität. He und Singh verwenden in [36] ein Maß, das von der Größe der Datenbank unabhängig ist und die Größe der Kandidatenmenge ins Verhältnis zur Größe der Ergebnismenge setzt. Die *Accuracy* der Kandidatenmenge wird dort nach der Formel $\frac{|\mathcal{D}_Q|}{|\mathcal{C}_Q|}$ berechnet. Yan et al. [73] verwenden den Kehrwert der Accuracy $\frac{|\mathcal{C}_Q|}{|\mathcal{D}_Q|}$ und bezeichnen den Wert als *Answer Set Ratio*. Hierbei weist ein hoher Wert auf eine schlechte Effektivität hin. Betrachtet man eine Datenbank mit 1000 Graphen und zwei unterschiedliche Suchmuster Q_1 mit $\mathcal{D}_{Q_1} = 1$ und $\mathcal{C}_{Q_1} = 50$ sowie Q_2 mit $\mathcal{D}_{Q_2} = 501$ und $\mathcal{C}_{Q_1} = 550$, so beträgt die Pruning Power für Q_1 ca. $0,95$ und für Q_2 nur ca. $0,90$. Die Pruning Power spricht also dafür, dass der Index die Anfrage nach Q_1 besser beantwortet hat. Verwendet man die Answer Set Ratio erhält man für Q_1 den Wert 50 und für Q_2 ca. $1,1$, was der Bewertung durch die Pruning Power widerspricht. Trotz der möglicherweise unterschiedlichen Bewertung zweier Suchanfragen erscheinen die vorgestellten Maße geeignet, um Indizes bei gleicher Auswahl von Suchmustern miteinander zu vergleichen, da eine kleinere Kandidatenmenge stets zu einem besseren Wert führt. Im Folgenden wird $\frac{|\mathcal{C}_Q|}{|\mathcal{D}_Q|}$ als Maß für die Effektivität verwendet.

Die Qualität eines Index wird also einerseits durch die benötigte Laufzeit T_s zum Generieren der Kandidatenmenge bestimmt und andererseits durch die Effektivität. Zwischen beiden Eigenschaften kann dabei je nach Verfahren eine gegenseitige Abhängigkeit bestehen: Häufig ist es einfach, die Effektivität zu erhöhen, indem zusätzliche Informationen indiziert werden, wodurch jedoch die Laufzeit T_s bei der Auswertung erhöht wird. Hier gilt es, einen Kompromiss zwischen beiden Größen zu finden.

4.1.2 Notwendige Bedingungen für Teilgraph-Isomorphie

Indizes können realisiert werden, indem notwendige Bedingungen für die Existenz eines Teilgraph-Isomorphismus gefunden werden. Erfüllt ein Graph eine notwendige Bedingung nicht, kann er aus der Kandidatenmenge entfernt werden, ohne dass das TGI-Problem gelöst werden muss. Geeignete Bedingungen sollten mit geringem Aufwand berechnet werden können und sind im Idealfall nur dann erfüllt, wenn tatsächlich ein TGI existiert. Da dies nicht miteinander zu vereinbaren ist, muss man sich mit Bedingungen zufrieden geben, die häufig auch dann erfüllt sind, wenn kein TGI existiert.

Definition 4.1 (Notwendige Bedingung für TGI). Seinen Q und G zwei Graphen. $B : \mathcal{G} \times \mathcal{G} \to \{\texttt{true}, \texttt{false}\}$ ist eine *Notwendige Bedingung für Teilgraph-Isomorphie*, wenn gilt $Q \lesssim G \Rightarrow B(Q, G) = \texttt{true}$.

Die Kontraposition liefert, dass, wenn $B(Q,G)$ nicht erfüllt ist, kein TGI von Q auf G existiert. Andererseits kann, wenn $B(Q,G)$ erfüllt ist, keine Aussage darüber getroffen werden, ob ein TGI tatsächlich vorliegt. Ein Index kann eine Vielzahl solcher Bedingungen verwenden und es genügt bereits eine einzige unerfüllte Bedingung, um einen Graphen gefahrlos aus der Kandidatenmenge zu entfernen.

Während in der Literatur zahlreiche notwendige Bedingungen für Graph-Isomorphismus (*Graph-Invarianten*) zu finden sind, sind für TGI nur wenige bekannt, die zudem deutlich weniger selektiv sind.

Vergleich von Zahlenwerten

Irniger und Bunke haben ein Filterverfahren für GI entwickelt, das eine Kandidatenmenge mit Hilfe eines Entscheidungsbaums erzeugt [38]. In [39, 40] wird das Verfahren auf TGI übertragen. Die Grundlage dieser Methode sind notwendige Bedingungen, die sich durch den Vergleich von Zahlenwerten überprüfen lassen. Für die Graphen in der Datenbank werden Eigenschaften vorberechnet. Bei der Suche werden die Werte des gesuchten Graphen mit denen der Datenbankgraphen verglichen und solche verworfen, die die zugehörige Bedingung nicht erfüllen. Folgende notwendige Bedingungen für die Existenz eines TGI von $Q = (V_Q, E_Q)$ auf $G = (V_G, E_G)$ werden verwendet:

- die Anzahl der Knoten: $|V_Q| \leq |V_G|$
- die Anzahl der Knoten mit einem bestimmten Label:

$$\forall n \in \Sigma : |\{v \in V_Q | l(v) = n\}| \leq |\{v \in V_G | l(v) = n\}|$$

- die Summe der Kanten von Knoten mit einem bestimmten Label:

$$\forall n \in \Sigma : \sum_{v \in V_Q \wedge l(v)=n} deg(v) \leq \sum_{v \in V_G \wedge l(v)=n} deg(v)$$

- die Anzahl Knoten mit einem bestimmten Grad:

$$\forall d \in \{0, ..., \max_{v \in V_Q}(deg(v))\} : |\{v \in V_Q | deg(v) \geq d\}| \leq |\{v \in V_G | deg(v) \geq d\}|$$

- die Anzahl Knoten mit einem bestimmten Label und einem bestimmten Grad:

$$\forall n \in \Sigma : \forall d \in \{0, ..., \max_{v \in V_Q}(deg(v))\} :$$
$$|\{v \in V_Q | deg(v) \geq d \wedge l(v) = n\}| \leq |\{v \in V_G | deg(v) \geq d \wedge l(v) = n\}|$$

In einem anderen Zusammenhang werden in [12] folgende Bedingungen vorgeschlagen:

- maximale Kreislänge: Die maximale Länge eines Kreises in Q muss kleiner gleich der maximalen Länge eines Kreises in G sein
- minimale Kreislänge: Die minimale Länge eines Kreises in Q muss größer gleich der minimalen Länge eines Kreises in G sein

Enthalten von Teilgraphen

Während die vorgestellten Bedingungen auf dem Vergleich von Zahlenwerten beruhen, kann auch das Konzept der Teilgraph-Isomorphie selbst verwendet werden, um notwendige Bedingungen zu formulieren.

Beobachtung 4.2 (Transitivität). Seien Q und G zwei Graphen mit $Q \lesssim G$. Für alle Graphen F mit $F \lesssim Q$ gilt $F \lesssim G$.

Diese Beobachtung ist die Grundlage zahlreicher Indizierungsverfahren von Graphen: Es wird vorab eine Menge von Graphen $\mathcal{F}$ festgelegt, die als *Merkmalsmenge* bezeichnet wird. Bei der Suche wird für alle $F \in \mathcal{F}$ die Bedingung

$$B_F(Q,G) = \begin{cases} \texttt{false} & \text{falls } F \lesssim Q \text{ und } F \not\lesssim G, \\ \texttt{true} & \text{sonst} \end{cases} \tag{4.2}$$

gefordert. Alle solchen Bedingungen lassen sich zusammenfassen zu

$$B(Q,G) = \begin{cases} \texttt{false} & \text{falls } \{F \in \mathcal{F} | F \lesssim Q\} \not\subseteq \{F \in \mathcal{F} | F \lesssim G\} \\ \texttt{true} & \text{sonst.} \end{cases} \tag{4.3}$$

Die Bedingung ist also nur dann erfüllt, wenn G mindestens alle Merkmale enthält, die Q aufweist. Dies lässt sich vergleichsweise effizient auswerten, wenn im Vorverarbeitungsschritt für jeden Graphen G der Datenbank die Menge der enthaltenen Merkmale $\{F \in \mathcal{F} | F \lesssim G\}$ bestimmt wird. Bei einer Suchanfrage nach dem Graphen Q müssen also lediglich die Menge $\{F \in \mathcal{F} | F \lesssim Q\}$ ermittelt und die Teilmengenbeziehungen überprüft werden.

Bekannte Indizierungsverfahren unterscheiden sich im Wesentlichen in der Auswahl der Merkmalsmenge und der verwendeten Datenstruktur zur Überprüfung der Teilmengenbeziehung.

4.1.3 Merkmalsmengen

Eine Möglichkeit, die Merkmalsmenge zu beschränken, besteht darin, nur bestimmte Klassen von Graphen zuzulassen: So kann die Merkmalsmenge beispielsweise auf alle Pfade reduziert werden, die eine bestimmte Länge nicht überschreiten.

Im Folgenden werden einige Klassen von Graphen definiert, die als Merkmale von Interesse sind. Die Definitionen sind weitgehend aus [21] übernommen.

Definition 4.3 (Pfad). Sei $G = (V, E)$ ein Graph. Ein *Pfad* der Länge n in G ist eine Folge von Knoten $(v_0, ..., v_n)$ für die gilt $\forall 0 \leq i < n : (v_i, v_{i+1}) \in E$. v_0 und v_n werden als *Endknoten* bezeichnet. Ein Pfad wird als *einfach* bezeichnet, wenn alle Knoten des Pfades paarweise verschieden sind.

Definition 4.4 (Kreis). Ein Pfad $K = (v_0, ..., v_n)$ der Länge $n > 1$, mit $v_0 = v_n$ wird als *Kreis* bezeichnet. Ein Kreis ist *einfach*, wenn ausschließlich die beiden Endknoten identisch sind.

Definition 4.5 ((Freier) Baum). Ein ungerichteter Graph $G = (V, E)$ wird als *Baum* bezeichnet, wenn $|E| = |V| - 1$ gilt und G keinen Kreis enthält.

Um Merkmalsmengen zu verwalten und schnell prüfen zu können, ob ein gegebener Graph einem Merkmal entspricht, d. h. isomorph zu einem Graphen der Merkmalsmenge ist, ist es zweckmäßig, Graphen durch einen kanonischen Bezeichner eindeutig zu identifizieren.

Einem Pfad P lässt sich beispielsweise der Bezeichner $B(P) = l(v_0)l((v_0, v_1))l(v_1)...l(v_n)$ zuordnen, der in dieser Arbeit *Label-Pfad*[1] genannt wird. Den beiden einfachen Pfaden $P_1 = (v_0, ..., v_n)$ und $P_2 = (v_n, ..., v_0)$ in einem ungerichteten Graphen $G = (V, E)$ kann der identische Teilgraph $G' = (V', E')$ mit $V' = \{v_0, ..., v_n\} \subseteq V$ und der Kantenmenge $E' = \{(v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_n)\} \subseteq E$ zugeordnet werden. Die Label-Pfade $B(P_1)$ und $B(P_2)$ können sich aber möglicherweise unterscheiden. Werden stets beide Varianten verwendet, bleibt das Vorgehen korrekt, die Anzahl der Merkmale nimmt jedoch zu. Darum ist es sinnvoll, nur eine der beiden Varianten zu speichern. Welche der beiden Möglichkeiten gewählt wird, darf nicht dem Zufall überlassen werden, sondern muss eindeutig bestimmt sein. Dies kann geschehen, indem stets der lexikographisch kleinere Label-Pfad gewählt wird. Eine so gewählte Zeichenkette, die für zwei Graphen genau dann identisch ist, wenn diese isomorph sind, wird *kanonischer Bezeichner* genannt.

Während es für einfache Pfade trivial ist, einen kanonischen Bezeichner zu finden, kann dies für komplexere Graphen deutlich aufwendiger sein. Für allgemeine Graphen ist kein Algorithmus mit polynomieller Laufzeit bekannt. Dies folgt direkt daraus, dass sich das Graph-Isomorphie-Problem, für das kein Polynomialzeitalgorithmus bekannt ist (vgl. Abschnitt 3.1.3), leicht auf das Finden kanonischer Bezeichner polynomiell reduzieren lässt. Somit ist die Beschränkung auf bestimmte Klassen von Graphen einerseits von Interesse, um die Anzahl möglicher Merkmale zu reduzieren und andererseits, um Merkmalsmengen effizient verwalten zu können.

[1] Eine Folge von Knotenlabeln wird in [32] als *label-path* bezeichnet. Um Unklarheiten zu vermeiden wird bei der hier verwendeten Definition vereinfachend angenommen, dass jedes Knoten- und Kantenlabel aus einem einzigen Zeichen besteht.

4.1.4 Klassifikation von Ansätzen

In der Literatur beschriebene Ansätze lassen sich auf unterschiedliche Weise in Klassen einteilen. Ein wesentlicher Unterschied ist, auf welche Weise Merkmale gewonnen werden. Hier kann zwischen Data-Mining-basierten Ansätzen und Nicht-Data-Mining-basierten Ansätzen unterschieden werden [77]. Data-Mining-basierte Ansätze analysieren zunächst die Graphen in der Datenbank und extrahieren Merkmale, von denen man sich eine hohe Effektivität verspricht, nach bestimmten Kriterien. Wird die Graphdatenbank anschließend verändert, indem neue Graphen hinzugefügt oder vorhandene gelöscht werden, kann die Qualität der Merkmalsauswahl nachlassen, so dass Maßnahmen zur Aktualisierung des Index getroffen werden sollten. Im Bereich der Chemie besteht außerdem die Möglichkeit, Strukturen nach anwendungsspezifischen Kriterien auszuwählen. Dies hat den Nachteil, dass die Auswahl nicht für alle Einsatzgebiete gute Ergebnisse liefert und für neue Stoffgruppen ungeeignet sein kann [11].

Nicht-Data-Mining-basierte Ansätze kommen ohne eine vorherige Analyse des Datenbestandes aus. Hier werden die Merkmale direkt aus den einzelnen Graphen gewonnen, ohne die Gesamtheit der Graphen in der Datenbank zu betrachten. Dabei werden in der Regel alle Teilgraphen einer bestimmten Klasse von Graphen als Merkmal verwendet. Nachträgliches Löschen und Einfügen von Graphen beeinträchtigt die Qualität der Indexstruktur somit nicht.

Eine weitere Möglichkeit der Klassifizierung stellt die Auswahl von Merkmalen in den Mittelpunkt. So können Methoden nach der verwendeten Klasse von Graphen unterschieden werden: Einige Verfahren verwenden Pfade als Merkmale, andere Bäume oder Graphen. Während pfadbasierte Methoden meist sämtliche Pfade einer Struktur indizieren, wird bei Graphen oft eine Auswahl getroffen, um die Anzahl der Merkmale zu begrenzen.

Deutliche Unterschiede der Verfahren lassen sich auch in der verwendeten Datenstruktur des Index ausmachen: Ein einfacher Ansatz ist es, zu jedem Graphen der Datenbank die Menge der enthaltenen Merkmale zu speichern. In der Filterphase können dann alle Merkmalsmengen sequentiell durchlaufen und daraufhin überprüft werden, ob alle Merkmale des Suchmusters enthalten sind. In dem Fall wird der zugehörige Graph in die Kandidatenmenge aufgenommen. Eine andere Möglichkeit besteht darin, zu jedem Merkmal die Menge der Datenbankgraphen (identifiziert durch eine Referenz oder ID) zu speichern. Dieses Verfahren wird als *invertierter Index* bezeichnet. In der Filterphase wird dann die Kandidatenmenge aus dem Schnitt aller Mengen gebildet, die zu Merkmalen gehören, die im Suchmuster enthalten sind: Sei $\mathcal{F}_Q = \{F \in \mathcal{F} | F \lesssim Q\} \cup \{(\emptyset, \emptyset)\}$. Die Kandidatenmenge kann dann zu

$$\mathcal{C}_Q = \bigcap_{F \in \mathcal{F}_Q} \mathcal{D}_F$$

bestimmt werden. Enthält $\mathcal{F}_Q$ nur den leeren Graphen - also kein einziges indiziertes Merkmal - so gilt $\mathcal{C} = \mathcal{D}$. Da die Filterphase dann wirkungslos ist, sollte ein guter Index diesen Fall vermeiden.

4.2 Bekannte Ansätze

In diesem Abschnitt werden einige in der Literatur beschriebene Verfahren im Detail vorgestellt. Diese folgen weitgehend den im vorangehenden Abschnitt vorgestellten Prinzipien.

4.2.1 Fingerprints in der Chemie

Zur Beschleunigung der Substruktursuche in großen Moleküldatenbanken wurden *Molecular Fingerprints* entwickelt. Bei dem Fingerprint eines Moleküls handelt es sich um einen Bitvektor fester Länge, der charakteristische Merkmale des Moleküls erfasst. Für alle Moleküle der Datenbank wird der Fingerprint vorab in einem Vorverarbeitungsschritt berechnet und in der Datenbank gespeichert. Bei der Suche wird der Fingerprint des Suchmusters berechnet und mit denen in der Datenbank verglichen.

Die Bedeutung der einzelnen Bits ist unterschiedlich und hängt von der verwendeten Methode ab. In [2, 11] werden zwei Klassen von Fingerprints unterschieden, die im Folgenden beschrieben werden.

Structure-Key Fingerprints

Structure-Key Fingerprints arbeiten mit einer vordefinierten Merkmalsmenge, wobei jedem Merkmal ein Bit des Fingerprints zugeordnet ist. Ein Bit wird genau dann auf 1 gesetzt, wenn das Molekül das zugehörige Merkmal aufweist. In der Regel handelt es sich bei Merkmalen um das Enthalten bestimmter Strukturen. Darüber hinaus können vordefinierte Eigenschaften in einem Structure-Key Fingerprint kodiert werden wie z.B. “enthält 4 oder mehr Sauerstoffatome” oder “enthält einen beliebigen Ring der Größe 5”.

Es sind verschiedene Zusammenstellungen von Merkmalen bekannt wie z.B. MACCS/ MDL Keys [24], bestehend aus 166 Strukturen, oder PubChem Substructure Fingerprint [4] mit 880 Strukturmerkmalen. Feste Zusammenstellungen von Merkmalen sind oft für einen bestimmten Zweck optimiert und haben den Nachteil, für andere Bereiche weniger geeignet zu sein. Für neue Anwendungen sind die erfassten Merkmale möglicherweise nicht hinreichend.

Hash-Key Fingerprints

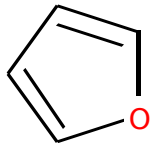
Hash-Key Fingerprints zeichnen sich dadurch aus, dass keine vordefinierte Merkmalsmenge benötigt wird, sondern Merkmale direkt aus den Molekülen generiert werden. Im Gegensatz zu Structure-Key Fingerprints kann dieser Ansatz deshalb als Nicht-Data-Mining-basiert

bezeichnet werden. Daylight Chemical Information Systems entwickelte ein Index System, bei dem alle Pfade, die eine bestimmte Länge nicht überschreiten, als Merkmal dienen [2]. Da die Menge möglicher Pfade groß ist, wird nicht für jeden ein unterschiedliches Bit reserviert, sondern der zugehörige Label-Pfad über eine Hashfunktion auf eine Position im Fingerprint abgebildet. Das Bit an dieser Position wird auf 1 gesetzt. Es können mehrere Bits des Fingerprints für jeden Pfad auf 1 gesetzt werden, um die Information redundant abzuspeichern.

Aufgrund der Verwendung einer Hashfunktion besteht hier keine Eins-zu-Eins-Beziehung zwischen einem Merkmal und der Position des Bits im Fingerprint: Es ist möglich, dass zwei unterschiedliche Label-Pfade von einer Hashfunktion auf die gleiche Position abgebildet werden. In diesem Fall spricht man von einer *Kollision*. Treten viele Kollisionen auf, wird dadurch die Effektivität des Index verschlechtert, es werden jedoch niemals korrekte Antworten verworfen.

Die Wahrscheinlichkeit für Kollisionen kann verringert werden, indem die Länge des Fingerprints erhöht wird. Dies hat jedoch den Nachteil, dass die Größe des Index und die Laufzeit der Filterphase zunehmen. Eine andere Möglichkeit wäre, die maximale Länge von Pfaden zu verringern, so dass weniger unterschiedliche Pfade gefunden werden. Dies hat wiederum den Nachteil, dass kürzere Pfade weniger selektiv sind. Das Festlegen der Parameter hat also großen Einfluss auf die Qualität der Filterphase. In [11] wird eine Pfadlänge zwischen 0 und 7 als üblich angegeben. Die freie Implementierung eines pfadbasierten Hash-Key Fingerprints der Bibliothek CDK verwendet als Standardeinstellung eine maximale Pfadlänge von 8 und Fingerprints mit 1024 Bit.

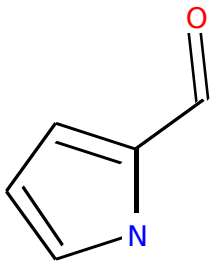
Abbildung 4.1 zeigt ein Beispiel mit zwei Strukturen und den zugehörigen Fingerprints, anhand derer ausgeschlossen werden kann, dass die eine Struktur in der anderen enthalten ist. In Abbildung 4.1(a) ist die gesuchte Struktur, die darin gefundenen Pfade mit maximaler Länge 3 und die durch eine Hashfunktion bestimmte Position in einem 32-Bit Fingerprint zu sehen. Zwischen den Pfaden `O` und `C=C-C=C` liegt eine Kollision vor, da beide auf die Position 11 des Fingerprints abgebildet werden. Abbildung 4.1(b) zeigt eine Struktur, wie sie in der Datenbank vorkommen könnte, und den zugehörigen Fingerprint. In dieser größeren Struktur werden 19 Pfade gefunden, wodurch der Fingerprint dichter mit Einsen besetzt ist. Hier treten bereits drei Kollisionen auf. Abbildung 4.1(c) zeigt, wie anhand der beiden Fingerprints ein Teilgraph-Isomorphismus ausgeschlossen werden kann: Mit Hilfe einer bitweisen Und-Verknüpfung beider Fingerprints kann leicht festgestellt werden, dass im Fingerprint des Suchmusters Bits an Positionen auf 1 gesetzt sind, die im Fingerprint des Datenbankgraphen eine 0 enthalten. Dies kann nur dann der Fall sein, wenn das Suchmuster einen Pfad enthält, der im Datenbankgraphen nicht vorhanden ist, wodurch ein Teilgraph-Isomorphismus ausgeschlossen werden kann.



C	7	C-C=C	16	C-C=C-O	30
O	11	C-O-C	13	C-O-C=C	28
C-C	25	C=C-O	0	C=C-C=C	11
C-O	2				
C=C	12				

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0

(a) Ein Suchmuster, die darin enthaltenen Pfade bis zur Länge 3 und die ihnen durch eine Hashfunktion zugeordnete Position im Fingerprint. Es liegt eine Kollision zwischen den Pfaden `O` und `C=C-C=C` vor, die im Fingerprint rot gekennzeichnet ist.



C	7	C-C-N	19	C-C-N-C	6
N	10	C-C=C	16	C-C=C-C	9
O	11	C-C=O	7	C-C=C-N	31
C-C	25	C-N-C	16	C-N-C=C	4
C-N	3	C=C-N	1	C=C-C=C	11
C=C	12			C=C-C=O	30
C=O	17			N-C-C=O	29

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	0	1	1	0	1	1	0	1	1	1	1	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1

(b) Ein Molekül, in dem das Muster gesucht werden soll. Es liegen drei Kollisionen vor: Die Pfade `C` und `C-C=O`, `O` und `C=C-C=C`, `C-C=C` und `C-N-C` werden jeweils auf identische Positionen abgebildet.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0
AND	0	1	0	1	1	0	1	1	0	1	1	1	1	0	0	0	1	1	0	1	0	0	0	0	0	1	0	0	0	1	1	1
=	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0

(c) Vergleich zweier Fingerprints: Ein Teilgraph-Isomorphismus kann ausgeschlossen werden, da das Ergebnis der bitweisen Und-Verknüpfung nicht mit dem Fingerprint des Suchmusters identisch ist. D. h., dass das Suchmuster mindestens einen Pfad enthält, der nicht im Molekül enthalten ist.

Abbildung 4.1: Beispiel eines pfadbasierten 32-Bit Hash-Key Fingerprints als Filter bei einer Substruktursuche. Abbildung (a) zeigt die gesuchte Struktur zusammen mit ihrem Fingerprint, (b) ein Molekül, in dem die Struktur gesucht wird. Aufgrund der Fingerprints kann ein Teilgraph-Isomorphismus ausgeschlossen werden, wie Abbildung (c) zeigt.

Fingerprints zur Bestimmung der Ähnlichkeit

Neben dem Einsatz von Fingerprints zur Beschleunigung einer Substruktursuche werden Fingerprints in der Chemie zur Bestimmung der Ähnlichkeit zweier Moleküle verwendet. Es gibt eine Vielzahl unterschiedlich definierter Ähnlichkeitsmaße, die sich aus Fingerprints berechnen lassen. Zu den am häufigsten verwendeten gehört der Tanimoto-Koeffizient [11]. Dieser ist definiert als

$$T = \frac{c}{a + b - c}, \tag{4.4}$$

wobei a die Anzahl 1-Bits im ersten und b im zweiten Fingerprint ist. c ist die Anzahl der Bits, die in beiden Fingerprints gemeinsam auf 1 gesetzt ist. Für das Beispiel aus Abbildung 4.1 ergibt sich damit ein Ähnlichkeitswert von $T = \frac{6}{10+16-6} = 0,3$.

Fingerprints, die speziell für derartige Anwendungen entwickelt wurden, sind nicht unbedingt zur Beschleunigung einer Substruktursuche geeignet: *Extended-connectivity fingerprints* [55] erfassen beispielsweise zu jedem Atom die vollständige Umgebung bis zu einer bestimmten Entfernung. Diese Klasse von Fingerprints hat sich experimentell als geeignet zur Erfassung von Ähnlichkeit erwiesen. Sie ist jedoch nicht geeignet, um eine Substruktursuche zu beschleunigen, da sich die Umgebungen von Atomen, die durch einen TGI aufeinander abgebildet werden, unterscheiden können.

Die mit Hilfe von Fingerprints ermittelte Ähnlichkeit beruht nicht auf einer streng formalen Definition, sondern bezieht sich nur auf die im Fingerprint erfassten Merkmale. Zur exakten Bestimmung der Ähnlichkeit wird unter anderem die Größe eines Maximum Common Subgraphs (siehe Definition 3.6) verwendet [54].

4.2.2 GraphGrep

GraphGrep wurde 2002 als anwendungsunabhängige Methode zur Suche in Graphen vorgestellt [32, 59] und beinhaltet eine an SMILES/SMARTS angelehnte Anfragesprache namens Glide. Diese unterstützt Wildcards für Knoten mit beliebigem Label, Pfade beliebiger Länge, optionale Knoten und Pfade der Länge ≥ 1.

Die Filterphase folgt ebenfalls einem pfadbasierten Ansatz. In der ursprünglichen Version werden alle Pfade[2] bis zu einer bestimmten Länge generiert und die zugehörigen Label-Pfade dienen als Schlüssel für eine Hashtabelle. Diese enthält für jeden Label-Pfad ein Feld mit der Anzahl der Vorkommen in jedem Graphen der Datenbank. In der Filterphase werden alle Graphen herausgefiltert, die für mindestens einen der Label-Pfade des Suchmusters eine geringere Anzahl an Vorkommen aufweisen. Variable Knoten und Kanten werden aus dem gesuchten Graphen entfernt und nur die Pfade in dem möglicherweise unzusammenhängenden Restgraphen verwendet.

GraphGrep unterstützt zudem die Suche in sehr großen Datenbankgraphen in besonderer Weise. Hierzu werden die Label-Pfade des Suchmusters verwendet, um nur den für das

[2] In späteren Veröffentlichungen werden ausschließlich einfache Pfade verwendet [10].

Suchmuster relevanten Teil des Datenbankgraphen zu laden. Die Verifikationsphase wird mit einem eigens hierfür entwickelten TGI-Algorithmus gelöst, der alle möglichen Kombinationen von überlappenden Pfaden bildet. Dieser Ansatz wurde jedoch später durch den VF2-Algorithmus der vflib (siehe Abschnitt 3.2.3) ersetzt [27, 28, 10].

Mit GraphFind [28] wurde das System 2008 für biomedizinische und chemische Datenbanken vorgestellt. Eine wesentliche Erweiterung besteht darin, dass der Index verkleinert wird: Für zwei Merkmale, die in allen Graphen der Datenbank mit gleicher Häufigkeit vorkommen, wird nur noch ein Eintrag in der Hashtabelle verwaltet. Darüber hinaus können Merkmale, die *fast* mit gleicher Häufigkeit auftreten, zusammengefasst werden, wodurch die Größe zusätzlich reduziert wird, aber evtl. die Effektivität des Index nachlässt. Gleiche oder ähnliche Einträge werden effizient mit einem auf Min-Hashing [15] basierenden Algorithmus gefunden.

2009 wurde eine weitere Verbesserung des Systems präsentiert, die anstatt mit einer Hashtabelle mit einem Suffixbaum arbeitet [10]. Die Label-Pfade werden in einem Suffixbaum verwaltet. Jeder Knoten repräsentiert den Label-Pfad, der sich auf dem Pfad von der Wurzel zum Knoten ergibt und enthält ein Feld, das - genau wie zuvor die Hashtabelle - für jeden Graphen der Datenbank die Anzahl der Vorkommen des Label-Pfades enthält. Bei der Suche nach einem Graphen wird für diesen ein neuer Suffixbaum erstellt, der alle im Graphen enthaltenen Label-Pfade repräsentiert. Dieser Baum wird anschließend mit dem Index (Suffixbaum der Datenbankgraphen) abgeglichen. Die Kandidatenmenge wird wie zuvor ermittelt, indem die Anzahl der Vorkommen von Label-Pfaden verglichen wird. Hierbei genügt es, die Knoten des Suffixbaums zu betrachten, die zu *maximalen Pfaden* im Suchmuster gehören. Diese zeichnen sich dadurch aus, dass sie entweder die maximale Länge erreicht haben, oder kürzer sind und nicht erweitert werden können. Die Beschränkung auf solche Suffixbaum-Knoten verkürzt die Laufzeit der Filterphase.

4.2.3 GIndex

GIndex [72, 73] basiert auf Merkmalen, die mittels Techniken des *Frequent Graph Minings* gewonnen werden und lässt sich somit den Data-Mining-basierten Verfahren zuordnen. Als Merkmale werden *discriminative frequent structures* verwendet: Der *Support* eines Graphen G ist definiert als $|\mathcal{D}_G|$ und G wird bezogen auf die Datenbank $\mathcal{D}$ als *frequent* bezeichnet, wenn $\frac{|\mathcal{D}_G|}{|\mathcal{D}|} \geq k$ gilt, wobei k ein festzulegender Parameter zwischen 0 und 1 ist. Ein Graph ist also dann häufig, wenn er in mindestens $k|\mathcal{D}|$ Datenbankgraphen enthalten ist. Die Wahl des Parameters k beeinflusst direkt die Anzahl häufiger Graphen und somit die Größe der Merkmalsmenge. Eine sehr große Merkmalsmenge führt zu einem sehr großen Index, enthält aber dafür mit hoher Wahrscheinlichkeit Merkmale, die das Suchmuster gut erfassen und führt somit zu einer guten Kandidatenmenge. Wird k niedrig gewählt, gestaltet sich das Finden der häufigen Strukturen - besonders in großen Datenbanken - schwierig, da es

exponentiell viele Teilgraphen gibt, von denen ein großer Teil die Häufigkeitsforderung erfüllt.

Daher wird k nicht fest gewählt, sondern über eine monoton steigende Support-Funktion $\psi(l)$ ausgedrückt, die von der Größe $l = |E|$ des Merkmalsgraphen $G = (V, E)$ abhängt. Dies hat zur Folge, dass kleine Graphen eher in die Merkmalsmenge aufgenommen werden. Graphen, die aus einem einzigen Knoten bestehen, sollen stets Einzug in der Merkmalsmenge halten, so dass garantiert werden kann, dass zu jedem Datenbankgraphen mindestens ein Merkmal indiziert wird. Dies wird erreicht, indem $\psi(0) = 1$ gesetzt wird. Graphen, die nur aus einem Knoten bestehen, gelten also bereits dann als häufig, wenn sie in einem einzigen Datenbankgraphen enthalten sind.

Die zweite Anforderung an ein mögliches Merkmal G ist, dass dieses bezogen auf eine bestehende Merkmalsmenge $\mathcal{F}$ *discriminative* ist: Sei $\mathcal{F}_G$ die Menge aller echten Teilgraphen von G, die in der Merkmalsmenge enthalten sind. Dann beschreibt folgende Formel die *dicriminative ratio* von G:

$$\gamma = \frac{|\bigcap_{H \in \mathcal{F}_G} \mathcal{D}_H|}{|\mathcal{D}_G|}. \tag{4.5}$$

Ein Graph wird dann als *discriminative* bezeichnet, wenn $\gamma > g > 1$ ist, wobei g ein festzulegender Parameter ist. Ist $\gamma \gg 1$, so wird G durch seine in $\mathcal{F}$ enthaltenen Teilgraphen nur unzureichend erfasst und ist dementsprechend wertvoll für die Merkmalsmenge. Ist γ kaum größer als 1, kann auf G als Merkmal verzichtet werden und wird dementsprechend als *redundant* bezeichnet. Der Parameter l hat also wiederum direkten Einfluss auf die Größe der Merkmalsmenge und die Effektivität des Index.

Organisiert wird die Merkmalsmenge in einer Baumstruktur, wobei jeder Graph aus $\mathcal{F}$ durch einen kanonisches Bezeichner identifiziert wird. Die Baumstruktur erleichtert einerseits das Generieren der Merkmalsmenge eines Suchmusters und andererseits das effiziente Berechnen der Kandidatenmenge.

Die Suche läuft in drei Schritten ab: Zunächst werden alle Merkmale des Suchmusters ermittelt, dann wird mit Hilfe des Index die Kandidatenmenge berechnet und im letzten Schritt mit einem TGI-Algorithmus verifiziert. Die Merkmalsmenge des Suchmusters wird berechnet, indem Teilgraphen des Suchmusters nacheinander in aufsteigender Größe bis zu einer maximalen Größe $maxL$ gebildet werden. Anhand der Baumstruktur des Index können dabei frühzeitig solche Teilgraphen erkannt werden, die sich nicht zu einem Graphen der Merkmalsmenge erweitern lassen.

Die Kandidatenmenge wird gebildet, indem der Durchschnitt aller ID-Mengen der Merkmale des Suchmusters berechnet wird. Das Prinzip ist also das eines invertierten Index. Die Baumstruktur wird verwendet, um die Anzahl notwendiger Durchschnittsoperationen zu reduzieren: Enthält das Suchmuster zwei Graphen F_1 und F_2 der Merkmalsmenge mit $F_1 \lesssim F_2$, so kann das Merkmal F_1 ignoriert werden, da die zu F_1 gespeicherte Menge von

Graphen $\mathcal{D}_{F_1}$ eine Obermenge von $\mathcal{D}_{F_2}$ ist und somit bei der Berechnung des Durchschnitts unwirksam bleibt.

Werden nachträglich Graphen eingefügt oder gelöscht, so wird die Merkmalsmenge nicht verändert, sondern nur die ID-Listen entsprechend modifiziert. Einige wenige Einfüge- und Löschoperationen haben keine große Auswirkung auf die Häufigkeit von Teilgraphen, so dass keine Änderung notwendig ist. Für große Datenbanken wird empfohlen, nur einen Teil der Datenbankgraphen zu verwenden, um die Merkmalsmenge zu bestimmen. Dies ist notwendig, da Frequent Graph Mining aufwendig ist und nach Möglichkeit die gesamte Datenmenge im Hauptspeicher gehalten werden sollte.

In Experimenten wurde GIndex mit GraphGrep verglichen. Dabei wurden die Parameter $g = 2$ und $maxL = 10$ für GIndex gewählt. Als größenabhängige Support-Funktion wurde

$$\psi(l) = \begin{cases} 1 & \text{falls } l < 4, \\ 0.1|\mathcal{D}|\sqrt{\frac{l}{maxL}} & \text{sonst} \end{cases} \tag{4.6}$$

verwendet. Es wurden also sämtliche Teilgraphen mit bis zu 3 Kanten als Merkmal verwendet. Die maximale Pfadlänge bei GraphGrep wurde auf 10 gesetzt. Als Testinstanzen diente unter anderem ein Datensatz mit Molekülen aus der AIDS-Forschung. Sowohl bei häufig vorkommenden Suchmustern als auch bei seltenen weist GIndex eine deutlich höhere Effektivität auf. Die Unterschiede sind besonders bei Suchmustern der Größe 15 und 20 deutlich: Für GIndex liegt der nach $\frac{|\mathcal{C}_Q|}{|\mathcal{D}_Q|}$ berechnete Wert im Durchschnitt etwa bei 2, für GraphGrep zwischen 9 und 11. Die Laufzeit, um den Index zu generieren, liegt auf einem 1.5GHz System mit 1 GB RAM für die größte getestete Datenbank mit 10000 Graphen zwischen 6 und 12 Minuten, abhängig von der Wahl des Parameters g .

4.2.4 TreePi

TreePi [75] verfolgt den gleichen Ansatz wie GIndex. Da Frequent Graph Mining jedoch aufwendig ist, wird hier stattdessen nach häufigen Bäumen gesucht, die als Merkmale dienen können. *Frequent Tree Mining* ist generell einfacher und effizienter umsetzbar. Motiviert wird die Entscheidung ferner dadurch, dass Bäume fast die gleichen Strukturinformationen wie beliebige Graphen erfassen können, Bäume generell einfacher zu handhaben sind und die Position von Teilbäumen in einem Graphen einfach gespeichert werden kann.

Die Merkmalsmenge $\mathcal{F}$ wird im Wesentlichen nach den gleichen Kriterien ausgewählt wie bei GIndex, beschränkt sich allerdings auf Bäume. Als größenabhängige Support-Funktion wird

$$\psi(l) = \begin{cases} 1 & \text{falls } l \leq \alpha, \\ 1 + \beta l - \alpha\beta & \text{falls } \eta \geq l > \alpha, \\ +\infty & \text{falls } l > \eta \end{cases} \tag{4.7}$$

vorgeschlagen, wobei α, β und η wählbare Parameter sind. Die Merkmalsmenge $\mathcal{F}$ wird weiter verkleinert, indem die bereits von GIndex bekannte Forderung, dass Merkmale *discriminative* sein müssen, angewandt wird.

Für jeden Baum der Merkmalsmenge wird der *Center* (ein Knoten oder eine Kante) ermittelt und für jedes seiner Vorkommen in einem Graphen der Datenbank die Position des Centers gespeichert. In einem invertierten Index wird zu jedem Merkmal festgehalten, in welchen Datenbankgraphen es enthalten ist. Zur Beantwortung einer Anfrage wird das Suchmuster zunächst heuristisch wiederholt vollständig in Bäume aus $\mathcal{F}$ zerlegt. Dabei werden jedoch nicht in jedem Fall alle Bäume aus $\mathcal{F}$ gefunden, die im Suchmuster enthalten sind. Nur die auf diese Weise im Suchmuster gefundenen Merkmale werden für die Filterphase verwendet. Die Kandidatenmenge ergibt sich aus den Datenbankgraphen, die alle solchen Merkmale enthalten und lässt sich schnell über den Index finden.

In einer zweiten Phase wird die Kandidatenmenge weiter gefiltert, indem die Lage der Merkmale im Suchmuster zueinander berücksichtigt wird. Dazu wird für zwei Merkmalsbäume die Distanz zwischen ihren Centern betrachtet. Ein Datenbankgraph kann herausgefiltert werden, wenn darin alle Vorkommen zweier Merkmale eine größere Distanz aufweisen als im Suchmuster. Es muss also geprüft werden, ob mindestens eine Kombinationen von Merkmalsvorkommen im Datenbankgraphen die Distanzbedingung erfüllt.

Die Verifikationsphase kommt ohne einen direkten Algorithmus für TGI aus, indem ein isomorpher Teilgraph direkt aus den Merkmalen mit Hilfe der Positionen ihrer Center zusammengesetzt wird.

In Experimenten mit einem Moleküldatensatz wurde gezeigt, dass TreePi weniger Merkmale als GIndex indiziert und dennoch eine höhere Effektivität aufweist. Dies kann auf die zusätzliche Filterphase zurückgeführt werden. Darüber hinaus erstellt TreePi den Index schneller als GIndex und weist deutlich schnellere Antwortzeiten auf.

4.2.5 Weitere Ansätze

Die hier vorgestellten Ansätze zeigen weitere Ideen auf, die zur Indizierung von Graphen verwendet werden können, jedoch für die vorliegende Aufgabenstellung weniger geeignet erscheinen.

GDIndex [71] ist ein Verfahren, bei dem sämtliche induzierten Teilgraphen indiziert werden. Diese werden in einem *Directed Acyclic Graph* (DAG) geordnet: Jeder Knoten entspricht einem induzierten Teilgraphen und hat eine ausgehende Kante zu allen Knoten, die einen Graphen repräsentieren, der genau einen zusätzlichen Knoten aufweist. Einzige Quelle ist ein Knoten, der den leeren Graphen repräsentiert, und Senken entsprechen Graphen aus der Datenbank. Innere Knoten können ebenfalls Graphen aus der Datenbank entsprechen. Isomorphe Graphen werden nur durch einen einzigen Knoten im DAG repräsentiert. Über kanonische Bezeichner, die in einer Hashtabelle verwaltet werden, kann

schnell auf jeden einzelnen Knoten des DAG zugegriffen werden. Eine Anfrage bzgl. ITGI kann beantwortet werden, indem der zugehörige Knoten im DAG mittels der Hashtabelle gefunden wird und alle Nachfolger des Knotens im DAG gesucht werden. Alle so gefundenen Knoten, die Graphen aus der Datenbank repräsentieren, bilden die Ergebnismenge. Eine Verifikation ist nicht notwendig. Das Verfahren eignet sich besonders für dichte, sehr kleine Graphen. Als maximale Knotenanzahl wird ca. 20 angegeben, was mit der exponentiell steigenden Anzahl induzierter Teilgraphen begründet werden kann. Daher ist das Verfahren für Molekül datenbanken ungeeignet.

Closure-Tree [36] ordnet alle Graphen der Datenbank in einem Suchbaum. Die Blätter des Suchbaums entsprechen den Graphen der Datenbank und ein innerer Knoten der *Graph Closure* aller seiner Kinder. Graph Closure ist ein neu eingeführtes Konzept, das mehrere Graphen durch einen neuen Graphen repräsentiert, der diese vollständig umfasst. Dabei können identische Teile der Graphen in der Graph Closure zusammengefasst werden, und solche Teile, die nicht in allen eingeschlossenen Graphen enthalten sind, als solche gekennzeichnet werden. Eine TGI-Anfrage wird beantwortet, indem der Suchbaum ausgehend von der Wurzel durchlaufen wird. Enthält die Graph Closure eines inneren Knotens das Suchmuster nicht, so braucht der komplette Teilbaum nicht weiter betrachtet zu werden. Dies erfordert das Lösen zahlreiche TGI-Probleme und auch die Konstruktion des Index mit der Forderung nach möglichst "kleinen" Graph Closures ist ein komplexitätstheoretisch anspruchsvolles Problem. Beides wird nur approximativ gelöst. Experimente deuten auf eine sehr lange Laufzeit der Filterphase hin [77].

GCode [77] ist ein Verfahren, das Erkenntnisse der *Spectral Graph Theory* verwendet, um neue notwendige Bedingungen für Teilgraph-Isomorphie zu formulieren. Für jeden Knoten eines Graphen in der Datenbank wird eine Signatur ermittelt. Diese enthält Informationen bezüglich seines Labels, der Label seiner Nachbarn und der von ihm ausgehenden Pfade bis zur Länge n, die zusammen einen *Level-n Path Tree* bilden. Für die Adjazenzmatrix des Level-n Path Trees werden die t größten Eigenwerte berechnet. Für einen Teilgraph-Isomorphismus muss der Knoten auf einen anderen Knoten abgebildet werden, dessen Level-n Path Tree den des anderen enthält. Dies ist nur dann möglich, wenn die Eigenwerte der zugehörigen Adjazenzmatrix bestimmten Anforderungen genügen. Aus der Signatur der Knoten eines Graphen wird eine Signatur für den Graphen selbst berechnet. Gefiltert wird in zwei Schritten: Im ersten wird nur die Signatur der Graphen verwendet, im zweiten die Signatur der Knoten. Anschließend wird für alle übrig gebliebenen Graphen das TGI-Problem exakt gelöst. Als Vorteil des Verfahrens wird angegeben, dass eine gute Effektivität erreicht wird und das Filtern mit Zahlenwerten schnell ist. Die Filterphase gewinnt dadurch an Effektivität, dass nicht nur zwei Graphen direkt verglichen werden, sondern auch die Signatur der darin enthaltenen Knoten. Dadurch wird die Filterphase jedoch auch deutlich aufwendiger.

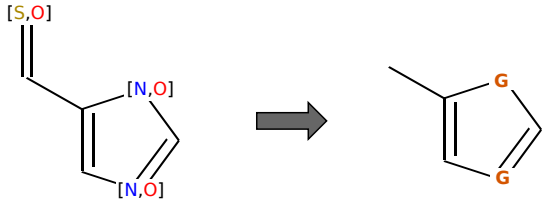
4.3 Erweiterbarkeit um Wildcards

Eine wünschenswerte Eigenschaft ist es, als Suchmuster nicht nur einen Graphen mit fest vorgegebenen Knoten- und Kantenlabeln zuzulassen, sondern es dem Nutzer zu ermöglichen, für bestimmte Elemente Wildcards anzugeben, die für variable Knoten- und Kantenlabel stehen. Neben Wildcards für beliebige Typen, ist die Angabe von Listen erlaubter Typen möglich. Dieses Konzept ist bei der Substruktursuche üblich und wird beispielsweise von SMARTS unterstützt.

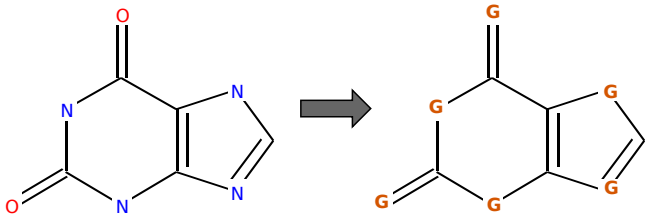
Die Verwendung von Wildcards wird im Zusammenhang mit den bekannten Ansätzen aus Abschnitt 4.2, wenn überhaupt, nur am Rande behandelt. GraphGrep unterstützt vergleichbare Operationen, indem Knoten und Kanten mit Wildcards im Suchmuster entfernt werden, bevor darin nach Merkmalen gesucht wird [59]. Dadurch entstehen kleinere, unzusammenhängende Fragmente. Den gleichen Ansatz verfolgt Daylight Chemical Information Systems in der kommerziellen Oracle Datenbank Erweiterung DayCart [57]. Hierbei wird eine zusätzliche Optimierung für Atomlisten verwendet: Um zu vermeiden, dass nur sehr kleine Fragmente nach der Löschung variabler Knoten und Kanten übrigbleiben, werden stattdessen alle Möglichkeiten enumeriert und mehrere Fingerprints berechnet. Die Kandidatenmenge setzt sich dann aus der Vereinigung aller Kandidatenmengen der enumerierten Fingerprints zusammen. Die Verwendung mehrerer Knoten mit Wildcards führt dabei schnell zu einer sehr großen Anzahl an Möglichkeiten. Diese wird begrenzt, indem Knoten mit vielen möglichen Atomtypen, die die Anzahl an Möglichkeiten stark erhöhen, doch vollständig entfernt werden. Das System beinhaltet zusätzliche Optimierungen, die die Variabilität durch Anwendung von Regeln der Chemie reduzieren sollen.

Das Löschen variabler Knoten (zusammen mit alle inzidenten Kanten) im Suchmuster, so dass keine Merkmale gefunden werden, die variable Knoten beinhalten, wurde ausschließlich bei Nicht-Data-Mining-basierten Verfahren angewandt. Die Übertragbarkeit auf Data-Mining-basierte Verfahren wie z.B. GIndex (siehe Abschnitt 4.2.3) scheint fraglich: Eine Suchanfrage kann nur dann gut beantwortet werden, wenn das verbleibende Suchmuster gut durch Merkmale erfasst wird. Hier kann sich die Beschränkung auf eine kleinere Auswahl von Merkmalen negativ auswirken. Das Löschen variabler Knoten führt möglicherweise zu Suchmustern, die nur schlecht durch indizierte Merkmale abgedeckt werden. Dies könnte eine starke Verschlechterung der Effektivität bei Anfragen mit Wildcards zur Folge haben.

Eine andere Möglichkeit könnte darin bestehen, Knoten- und Kantenlabel nur teilweise oder gar nicht mit den Merkmalen zu erfassen: Ein Graph mit Labeln kann nur dann in einem anderen Graphen mit Labeln enthalten sein, wenn der zugehörige Graph *ohne* Label in dem anderen Graphen *ohne* Label enthalten ist. Es ist also möglich, einen Index komplett ohne Label aufzubauen, der zusätzlich zum vorhandenen Index verwendet wird. Dieser wird im Allgemeinen eine deutlich schlechtere Effektivität aufweisen, ist aber



(a) Suchmuster



(b) Molekül

Abbildung 4.2: Zusammenfassen von Knotenlabeln: Die Label N und O werden zur Gruppe G zusammengefasst. Im Suchmuster muss nur noch der Knoten mit der Liste $[S, O]$ gelöscht werden, der Ring bleibt erhalten.

dann nützlich, wenn Suchmuster viele Wildcards enthalten. Es wäre darüber hinaus möglich, Knoten- und Kantenlabel zu Gruppen zusammenzufassen und neue Label für solche Gruppen einzuführen. Dies könnte besonders für Moleküle sinnvoll sein, da Atomtypen mit ähnlichen Eigenschaften wahrscheinlich häufig in Listen aufgezählt werden und sinnvoll in Gruppen zusammengefasst werden können. Das Suchmuster würde dann in einem ersten Schritt auf das Abstraktionsniveau des Index gebracht werden, indem Label und Listen von Labeln, die vollständig in einer Gruppe liegen, durch das Label der Gruppe ersetzt würden. Knoten mit Listen von Labeln aus unterschiedlichen Gruppen müssten weiterhin gelöscht werden. Anschließend kann eine Kandidatenmenge mit einem beliebigen Index berechnet werden, in dem zuvor alle Graphen auf dem gleichen Abstraktionsniveau indiziert wurden. Abbildung 4.2 zeigt ein Beispiel mit einem Suchmuster aus der Chemie.

Während die zuletzt genannte Methode eine mögliche Erweiterung des bekannten Verfahrens darstellt und genauer auf seine Tauglichkeit untersucht werden müsste, wurde für die Implementierung in Scaffold Hunter auf die erste Variante zurückgegriffen, die sich auch bei der Suche in Molekül datenbanken bewährt hat. Auf eine Optimierung durch Enumeration wird verzichtet, stattdessen wurde ein neuer Fingerprint entwickelt, der im nachfolgenden Abschnitt beschrieben wird. Eine Verbesserung bei der Suche mit Wildcards verspricht der Fingerprint dadurch, dass komplexere Merkmale erfasst werden. Zerfällt ein Suchmuster durch das Löschen von variablen Knoten in viele kleine Fragmente, haben diese vermutlich die Struktur eines Baums. Der neue Fingerprint kann diese Fragmente besser erfassen als ein pfadbasierter Fingerprint.

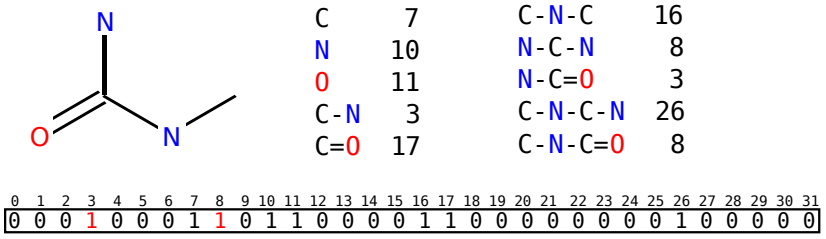
4.4 Ein neuer Hash-Key Fingerprint

Für die Integration in Scaffold Hunter wurde ein neuer Hash-Key Fingerprint entwickelt, der genau wie der in Abschnitt 4.2.1 beschriebene Ansatz ohne Data-Mining auskommt. Neben den erwähnten Schwierigkeiten bei der Unterstützung von Wildcards gibt es mehrere Gründe, die gegen die Verwendung eines Data-Mining-orientierten Ansatzes wie GIndex oder TreePi sprechen: Zunächst wäre der deutlich höhere Aufwand der Implementierung und Integration zu nennen, da für keines der Verfahren freie Software in Java zur Verfügung steht. Während der erste Schritt zur Generierung der Merkmalsmenge, das Frequent Graph Mining, z.B. mit der frei verfügbaren Bibliothek ParMol [50] realisiert werden könnte, bleibt der Aufwand für das Herausfiltern redundanter Merkmale und die Implementierung des invertierten Index erheblich. Scaffold Tree Generator erlaubt es, nachträglich Moleküle in einen bestehenden Scaffold-Baum einzufügen. Dies ist bei Data-Mining orientierten Verfahren problematisch, da die Merkmalsmenge evtl. angepasst werden sollte und der gesamte Index neu generiert werden müsste. Bei großen Datenbanken ist Frequent Graph Mining im gesamten Datensatz zudem nicht effizient realisierbar. Die Client-Server-Architektur von Scaffold Hunter spricht dafür, die Filterphase auf Serverseite zu realisieren. Ein invertierter Index, wie ihn GIndex und TreePi benutzen, ist nur schwierig effizient mit einer MySQL Datenbank umzusetzen, während das Filtern mit Fingerprints leicht zu realisieren ist wie in Abschnitt 4.4.4 gezeigt wird.

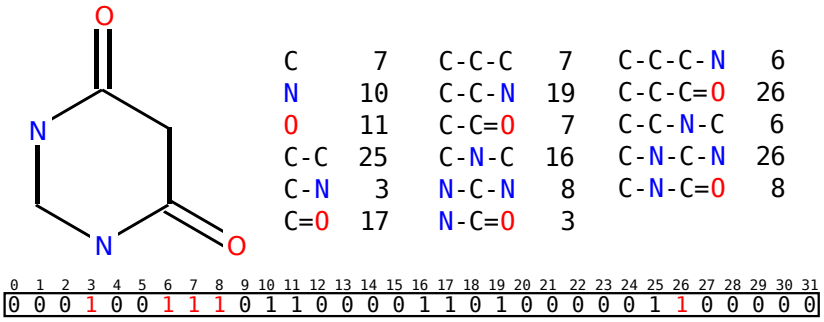
Die Vorteile von Verfahren wie GIndex und TreePi gegenüber pfadbasierten Verfahren wie GraphGrep oder den üblichen Hash-Key Fingerprints beruhen wesentlich darauf, dass komplexere Merkmale indiziert werden. In diesem Abschnitt wird ein neuer Hash-Key Fingerprint vorgestellt, der Bäume und Kreise an Stelle von Pfaden als Merkmale verwendet. Da jeder Pfad auch ein Baum ist, wird die Merkmalsmenge gegenüber dem bekannten Ansatz erweitert. Durch Bäume können komplexere Teilstrukturen eines Moleküls erfasst werden, wodurch die Effektivität der Filterphase erhöht werden soll.

Das Beispiel in Abbildung 4.3 zeigt einen Fall, in dem Pfade unzureichend sind, um einen Teilgraph-Isomorphismus auszuschließen: Alle Label-Pfade des Suchmusters treten auch im Molekül auf. Es ist jedoch leicht zu sehen, dass das Suchmuster dennoch nicht im Molekül enthalten ist. Der Baum bestehend aus dem O-Atom, den beiden N-Atomen und dem dazwischenliegenden C-Atom ist nicht im Molekül vorhanden. Hier genügt also ein Baum mit vier Knoten als Merkmal, um eine Struktureigenschaft zu erfassen, durch die ein Teilgraph-Isomorphismus ausgeschlossen werden kann.

Das Beispiel verdeutlicht also den klaren Vorteil von Bäumen gegenüber Pfaden. Es muss jedoch andererseits beachtet werden, dass wesentlich mehr Teilbäume als Pfade in einem Molekül enthalten sind, wodurch die Merkmalsmenge vergrößert wird. Ob sich der zusätzliche Aufwand lohnt, soll im nachfolgenden Abschnitt 4.5 durch experimentelle Vergleiche ermittelt werden.



(a) Suchmuster



(b) Molekül

Abbildung 4.3: Erfolgloses Filtern mit einem pfadbasierten Hash-Key Fingerprint: Das Molekül kann nicht aus der Kandidatenmenge entfernt werden, da alle Label-Pfade des Suchmusters auch als Label-Pfad im Molekül vorkommen, obwohl das Suchmuster nicht im Molekül enthalten ist.

Zunächst wird im Detail beschrieben, wie Teilbäume und Kreise in einem Graphen gefunden und als eindeutige Merkmale verwendet werden können.

4.4.1 Kanonische Bezeichner für Bäume

Bei dem pfadbasierten Ansatz wurden nur die lexikographisch kleineren Label-Pfade als Merkmal verwendet, damit zwei isomorphe Teilgraphen nicht als zwei unterschiedlichen Label-Pfade erfasst werden. Das Prinzip, dass alle isomorphen Graphen durch eine eindeutige Form repräsentiert werden, wird allgemein als *Kanonisierung* bezeichnet. Aus der kanonischen Form eines Graphen lässt sich ein eindeutiger Bezeichner ableiten, der kanonischer Bezeichner genannt wird. Während für allgemeine Graphen kein polynomieller Algorithmus bekannt ist, ist das Problem für Bäume in Zeit $O(n)$ lösbar, wie im Folgenden erläutert wird. Bei der Suche nach Merkmalen ist damit zu rechnen, dass für sehr viele, eher kleine Bäume ein kanonischer Bezeichner berechnet werden muss. Daher kommt es gelegen, dass ein Algorithmus für Bäume nicht nur eine theoretisch gute Laufzeit bietet, sondern auch leicht effizient implementiert werden kann.

Abbildung 4.4 zeigt drei isomorphe Bäume wie sie als Teilstrukturen eines Moleküls auftreten können. Für die Korrektheit der Filterphase ist es von entscheidender Bedeutung, dass identische Merkmale, d. h. isomorphe Bäume, durch einen eindeutigen Bezeichner

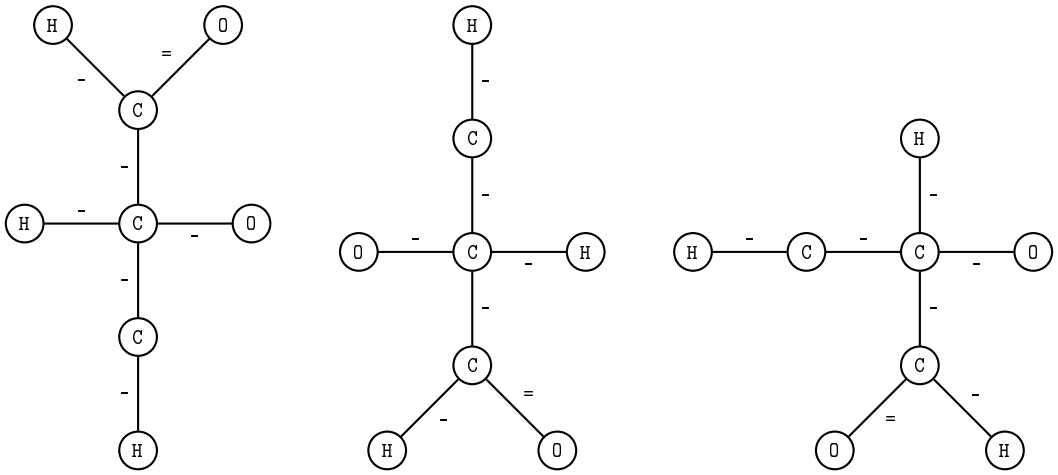


Abbildung 4.4: Drei isomorphe Bäume.

identifiziert werden. Dabei darf es beispielsweise keine Rolle spielen, in welcher Reihenfolge die Nachbarn eines Knotens in seiner Adjazenzliste gespeichert sind.

Das Problem wurde im Bereich des *Frequent Tree Minings* untersucht und der vorgestellte Algorithmus folgt der Beschreibung in [14]. Die kanonische Form eines Baums lässt sich mit Hilfe eines eindeutig festgelegten *geordneten Baums* angeben.

Definition 4.6 (Gewurzelter Baum [21]). Ein gewurzelter Baum ist ein Baum $B = (V, E)$ mit einem ausgezeichneten Knoten $r \in V$, der als *Wurzel* bezeichnet wird. Folgt u in dem Pfad von v zu r direkt auf v, so wird u als *Vater* von v bezeichnet und v als *Kind* von u. Kinder mit dem selben Vater sind *Geschwister*.

Definition 4.7 (Geordneter Baum [21]). Ein *geordneter Baum* ist ein gewurzelter Baum, bei dem die Kinder jedes Knotens geordnet sind.

Die Vorgehensweise des Algorithmus lässt sich in drei Schritte gliedern: Zunächst wird der Baum in einen gewurzelten Baum überführt (im Sonderfall entstehen zwei gewurzelte Bäume), dann wird eine Ordnung der Kinder jedes Knotens bestimmt und zuletzt wird ein kanonischer Bezeichner generiert. Die drei Schritte werden im Folgenden genauer beschrieben.

Bestimmen einer Wurzel

Um eine Wurzel eindeutig festzulegen, wird der *Center* eines Baums bestimmt. Dazu werden schrittweise alle Blätter des Baums mit den zugehörigen Kanten entfernt, bis nur noch ein oder zwei Knoten übrig bleiben. Bleibt ein Knoten übrig, so wird dieser als *Center* bezeichnet. Bleiben zwei Knoten übrig, spricht man von *Bicenter* [13]. Entscheidend ist,

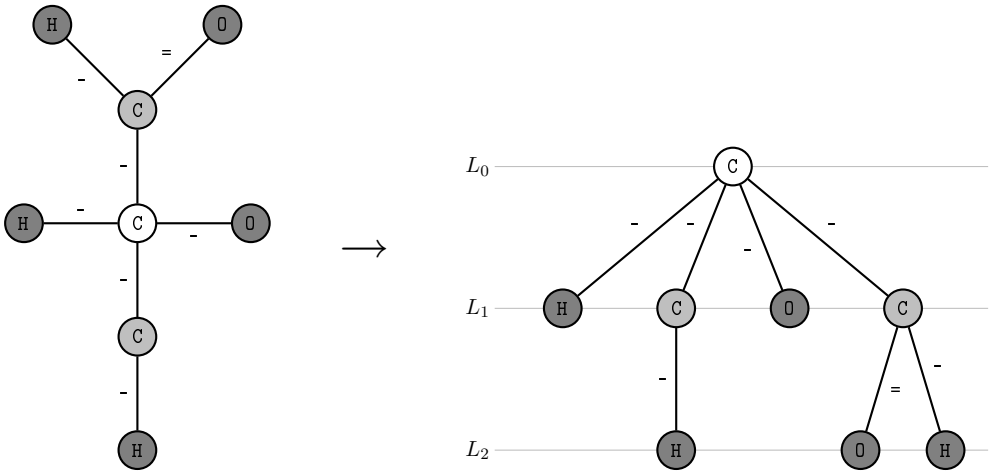


Abbildung 4.5: Der weiße Knoten ist Center des Baums. Die Blätter der ersten Iteration sind dunkelgrau gekennzeichnet, die Blätter der zweiten hellgrau. Die rechte Abbildung zeigt eine typische Zeichnung des entstandenen gewurzelten Baums, bei dem die Wurzel ganz oben abgebildet ist und Kinder stets eine Ebene unter ihrem Vater.

dass, wenn ein Graph-Isomorphismus φ zwischen zwei Bäumen existiert, φ stets den Center des einen Baums auf den des anderen abbildet bzw. die Bicenter aufeinander abbildet. Abbildung 4.5 veranschaulicht das Verfahren für den Baum aus Abbildung 4.4(a).

In dem Fall, dass nur ein Center existiert, wird dieser Knoten als Wurzel gewählt. Liegt ein Bicenter vor, so wird der Baum zwischen diesen beiden Knoten in zwei Teilbäume zerlegt, die getrennt behandelt werden. Für beide Teilbäume wird der Knoten als Wurzel verwendet, der ursprünglich einer der Bicenter war. Das Ergebnis dieses Schritts ist also entweder ein gewurzelter Baum oder zwei gewurzelte Bäume, die als getrennter Sonderfall behandelt werden. Die Laufzeit hierfür beträgt $O(n)$.

Festlegen einer Ordnung

Ein gewurzelter Baum kann in einen geordneten Baum überführt werden, indem die Ordnung der Kinder jedes Knotens festgelegt wird. Wieder muss die Ordnung für alle isomorphen Bäume zu dem gleichen Ergebnis führen. Zur Vereinfachung werden Kantenlabel nicht explizit behandelt. Man kann ohne weiteres jedem Knotenlabel das Label der Kante zu seinem Vater voranstellen, so dass Kantenlabel für die Ordnung ignoriert werden können.

Zunächst wird eine Ordnungsrelation auf den Knoten eines Baums definiert, die es erlaubt, Geschwister untereinander zu sortieren, indem die an ihnen beginnenden Teilbäume verglichen werden.

Definition 4.8. Seien u und v zwei Knoten in einem Baum und $c_1^u, c_2^u, ..., c_n^u$ die geordneten Kinder von u bzw. $c_1^v, ..., c_m^v$ von v. Es gilt $u < v$, genau dann, wenn

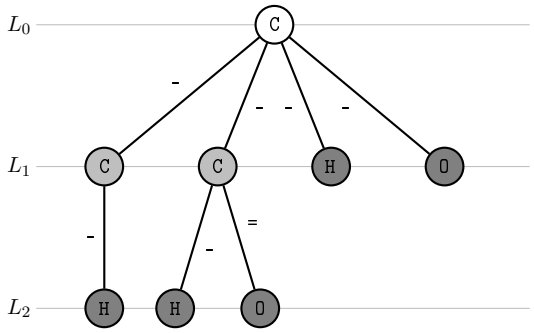


Abbildung 4.6: Die kanonische Form des Baums von Abbildung 4.5.

- $l(u) < l(v)$ (lexikographisch kleiner) ist oder
- $l(u) = l(v)$ und $\exists k : c_k^u < c_k^v \wedge \forall i < k : c_i^u = c_i^v$ oder
- $l(u) = l(v)$ und $\forall i \leq n : c_i^u = c_i^v$ und $n < m$.

Es gilt $u = v$ genau dann, wenn $l(u) = l(v)$ und $\forall i : c_i^u = c_i^v$ und $n = m$.

Zwei Geschwister in einem Baum werden also geordnet, indem zunächst ihre Label miteinander verglichen werden. Sind diese identisch, werden rekursiv die bereits sortierten Kinder (und ggf. deren Kinder usw.) von links nach rechts miteinander verglichen, bis die Ordnung entschieden werden kann. In dem Fall, dass zwei Geschwister die Wurzeln identischer (d. h. isomorpher) Teilbäume sind, spielt die Sortierung keine Rolle.

Abbildung 4.6 zeigt den geordneten Baum, der sich aus dem Baum aus Abbildung 4.5 ergibt, wenn alle Geschwister entsprechend der Ordnung sortiert werden. Ein so sortierter Baum wird als *kanonische Form* bezeichnet.

Der Algorithmus löst das Problem bottom-up, so dass in jedem Schritt bereits die Ordnung aller Kinder bekannt ist. Eine einfache Implementierung hat den Nachteil, dass der Vergleich zweier Knoten nicht in Zeit $O(1)$ möglich ist und ein vergleichsbasierter Sortiervorgang von k Geschwistern Zeit $\Omega(k \log k)$ beansprucht [21]. Hopcroft et al. beschreiben in [5] einen Algorithmus, der eine vergleichbare Ordnung für Bäume ohne Label in Zeit $O(n)$ erzeugt. In [47] wird ein darauf aufbauendes Verfahren vorgestellt, das den Algorithmus auf Bäume mit Labeln erweitert und ebenfalls in Linearzeit läuft. Entscheidend für die Verbesserung der Laufzeit ist, dass hier zunächst alle Knoten, die sich auf derselben Ebene befinden, untereinander sortiert und durchnummeriert werden. Diese Nummerierung wird verwendet, um die nächst höhere Ebene wiederum zu ordnen und zu nummerieren. Zur Sortierung kann hierbei Bucketsort verwendet werden.

Da die hier auftretenden Bäume relativ klein sind, wurde auf die Implementierung eines Linearzeitalgorithmus verzichtet und die einfache Variante gewählt.

Generieren des kanonischen Bezeichners

Ist die kanonische Form eines Baums bekannt, lässt sich daraus leicht eine Zeichenkette ermitteln, die den Baum eindeutig beschreibt. Dazu wird der Baum mittels Tiefensuche durchlaufen, Knoten- und Kantenlabel ausgegeben und Backtrackingschritte durch das zusätzliche Zeichen $ kenntlich gemacht, das nicht als Label auftreten darf. Mit diesem Verfahren ergibt sich für den Baum aus Abbildung 4.6 der kanonische Bezeichner `C-C-H$$-C-H$=O$$-H$-O$$`, der den Baum eindeutig und vollständig beschreibt.

In dem Fall, dass ein Baum die zwei Center u und v hat, wurde der Baum zunächst zerteilt. Die beiden Teilbäume T_u und T_v werden getrennt geordnet und von beiden wird der kanonische Bezeichner berechnet. Sei $B(T_u)$ der kanonische Bezeichner von T_u und $B(T_v)$ der von T_v und o. B. d. A. $u \leq v$ bezüglich der unter 4.8 definierten Ordnung. Der kanonische Bezeichner des gesamten Baums ist dann $B(T_u)l((u,v))B(T_v)$ - zwischen den beiden kanonischen Bezeichnern wird also das Label der Kante, welche die beiden Wurzeln der Teilbäume verbindet, eingefügt.

4.4.2 Enumerieren aller Teilbäume

Es sollen alle Bäume generiert werden, die eine gegebene maximale Größe nicht überschreiten und in dem gegebenen Graphen enthalten sind. Für jeden solchen Baum wird dann sein kanonischer Bezeichner ermittelt und als Merkmal abgespeichert.

Algorithmus 4.1 : Generieren aller Teilbäume.

Daten : Graph $G = (V_G, E_G)$, maximale Baumgröße $maxB$
Ausgabe : Menge aller Teilbäume $\mathcal{R}$

```
1 forall v ∈ V_G do
2     B ← ({v}, ∅)
3     R ← R ∪ {B}
4     extendTree(B)
```

Der verwendete Algorithmus 4.1 beginnt mit allen möglichen Bäumen, die aus einem einzigen Knoten bestehen. Jeder generierte Baum wird durch die Prozedur extendTree schrittweise um alle möglichen Kanten erweitert. Um sicherzustellen, dass der auf diese Weise generierte Teilgraph ein Baum bleibt, dürfen nur Kanten gewählt werden, von denen genau ein Knoten bereits im Baum enthalten ist (Algorithmus 4.2, Zeile 1). Wäre keiner der beiden Knoten im Teilgraphen enthalten, würde ein unzusammenhängender Teilgraph entstehen. Wären beide bereits im Teilgraph enthalten, würde ein Kreis geschlossen. In beiden Fällen wäre der Teilgraph kein Baum mehr. Durch die Abfrage in Zeile 3 wird verhindert, dass identische Teilbäume (mit gleicher Kantenmenge) mehrmals gene-

Algorithmus 4.2 : ExtendTree(B)

Eingabe : Baum $B = (V_B, E_B)$
Daten : Graph $G = (V_G, E_G)$, maximale Baumgröße $maxB$
Ausgabe : Teilbäume $\mathcal{R}$

```
1 forall e = (u, v) ∈ E_G mit u ∈ V_B und v ∉ V_B do
2     B' ← (V_B ∪ {v}, E_B ∪ {e})
3     if B' ∉ R then
4         R ← R ∪ {B'}
5         if |E_B'| < maxB then extendTree(B')
```

riert werden. Identische Teilbäume würden sonst immer dann entstehen, wenn die Kanten des Teilbaums in unterschiedlicher Reihenfolge hinzugefügt wurden.

4.4.3 Erfassen von Ringstrukturen

Bäume sind in der Lage, viele Strukturmerkmale von Molekülen gut zu erfassen. In der Chemie sind besonders zyklische Strukturen von großer Bedeutung, die von Bäumen jedoch nicht vollständig wiedergegeben werden können. Um diesen Nachteil auszugleichen, werden einfache Kreise als zusätzliche Merkmale erfasst und ebenfalls im Index gespeichert.

Wie bei Bäumen stellt sich auch bei einem Kreis das Problem, dass dieser durch einen eindeutigen kanonischen Bezeichner repräsentiert werden soll. Eine Zeichenkette kann leicht generiert werden, indem der Kreis entweder im oder gegen den Uhrzeigersinn durchlaufen wird und beginnend an einem beliebigen Knoten nacheinander die Knoten- und Kantenlabel ausgegeben werden. Für einen Ring mit n Knoten existieren dann $2n$ mögliche Bezeichner. Wieder kann der lexikographisch kleinste Bezeichner als kanonischer Bezeichner gewählt werden.

Eine Zeichenkette der Länge n, bei der das erste Zeichen als Nachfolger des n-ten Zeichens angesehen wird, bezeichnet man als *Circular String*. Das "Auftrennen" der Zeichenkette an einer beliebigen Stelle, so dass die resultierende Zeichenkette die lexikographisch kleinste aller n möglichen ist, wird als *Circular String Linearization*-Problem bezeichnet und lässt sich z.B. mit Hilfe von Suffixbäumen in Zeit $O(n)$ lösen [35]. Somit kann für Kreise mit n Knoten ein kanonischer Bezeichner in Zeit $O(n)$ gefunden werden. Auf die Implementierung eines Linearzeitalgorithmus wurde wiederum verzichtet, da ein einfacher Ansatz, der alle $2n$ Möglichkeiten vergleicht, für die auftretenden einfachen Kreise in ausreichender Geschwindigkeit zu einem kanonischen Bezeichner führt.

Abbildung 4.7 zeigt eine Struktur, die insgesamt drei einfache Kreise enthält, wobei der "äußere" Ring der Größe neun zwei kleinere Ringe vollständig umfasst. Da nur einfache

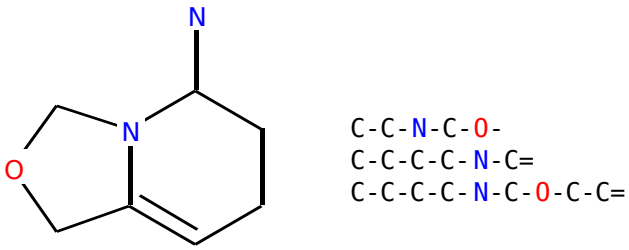


Abbildung 4.7: Eine Struktur und die kanonischen Bezeichner der drei darin enthaltenen einfachen Kreise. Im Beispiel ist das Zeichen "-" lexikographisch kleiner als "=".

Kreise erfasst werden sollen, existieren keine weiteren Merkmale. Die zu den drei Ringen zugehörigen kanonischen Bezeichner sind neben der Abbildung zu finden.

Das Suchen von Ringen in einem Molekül ist ein häufiges Problem der Chemie und die Bibliothek CDK bietet diese Funktionalität. Statt einer neuen Implementierung eines Algorithmus zur Suche von Ringen wurde daher auf die entsprechende Methode der Bibliothek zurückgegriffen. Um die erforderliche Laufzeit und die Anzahl der Merkmale zu beschränken, wird die maximale Anzahl Knoten eines einfachen Kreises durch einen Parameter begrenzt.

4.4.4 Datenstruktur

Als Datenstruktur werden Hash-Key Fingerprints verwendet wie sie in Abschnitt 4.2.1 vorgestellt wurden und in der Chemie häufig Anwendung finden. Alle generierten kanonischen Bezeichner von Bäumen und Ringstrukturen werden über eine Hashfunktion auf ein Bit im Fingerprint abgebildet. Um Ringen eine stärkere Gewichtung zu verleihen, wurden für jeden Ring zwei Bits im Fingerprint gesetzt. Dadurch wird der Einfluss von Kollisionen auf diese Merkmale verringert.

Nach Gleichung 4.3 setzt sich die Kandidatenmenge aus allen Graphen zusammen, deren Merkmalsmenge eine Obermenge der Merkmalsmenge des Suchmusters ist. Das Problem, alle Mengen in einer Datenbank zu finden, die eine gegebene Menge enthalten, ist für zahlreichen Anwendungen von Bedeutung und wurde u.a. im Bereich relationaler Datenbanken und der Indizierung von Texten untersucht[3]. Theoretische und experimentelle Resultate [76, 37] sprechen stark für die Verwendung eines invertierten Index. Die durchgeführten Experimente beziehen sich jedoch eher auf Suchmengen mit geringer Kardinalität, wie sie für die Text- und Datenbanksuche in diesem Kontext typisch sind. Die Laufzeit zur Beantwortung einer Anfrage hängt bei der Verwendung eines invertierten Index jedoch stark von der Kardinalität der Suchmenge ab, während diese bei Fingerprints kaum ei-

[3] In Veröffentlichungen aus diesem Forschungsbereich wird eine mit Hash-Key Fingerprints vergleichbare Datenstruktur als *Signature File* bezeichnet.

ne Rolle spielt. In der hier verwendeten Form kann die gesuchte Teilmenge jedoch viele Elemente enthalten.

Ein entscheidender Vorteil von Fingerprints ist, dass sie einfach zu implementieren sind und sich gut in die vorhandene Architektur von Scaffold Hunter integrieren lassen: Wie in Abschnitt 2.3.1 beschrieben wurde, verwendet Scaffold Hunter einen MySQL Server, von dem selektiv die Daten geladen werden, die visualisiert werden sollen. Der komplette Datenbestand ist also nur auf Serverseite verfügbar und eine Übertragung der Daten zum Client sollte nur erfolgen, wenn die Daten tatsächlich benötigt werden. Das Filtern der Kandidatenmenge lässt sich bei der Verwendung von Fingerprints mit einfachen Bitoperationen implementieren, die serverseitig von MySQL ausgeführt werden können [3]. Dazu werden die Fingerprints in 64-Bit große Segmente zerlegt, die als MySQL Datentyp `BIGINT` gespeichert werden können. Alle Fingerprints der Moleküle können also direkt in der Datenbank gespeichert werden und die Filterphase, bei der in der Regel der größte Teil der Graphen ausgeschlossen werden kann, wird vollständig von der Datenbank ausgeführt, zu der nur der Fingerprint des Suchmusters übertragen wird.

4.4.5 Beispiel

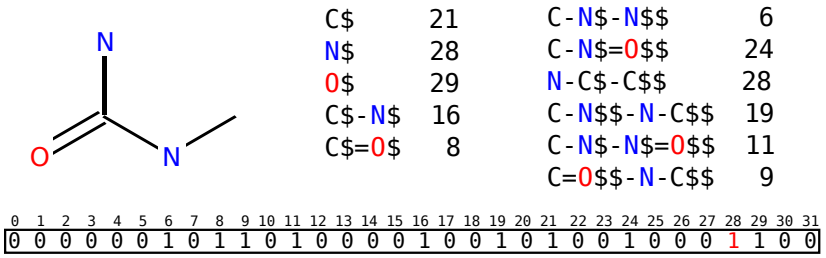
Die Funktionsweise des Verfahrens soll an den bereits von Abbildung 4.3 bekannten Strukturen veranschaulicht werden, bei denen das Filtern mit einem pfadbasierten Fingerprint erfolglos blieb. Abbildung 4.8 zeigt die Strukturen zusammen mit den kanonischen Bezeichnern aller darin enthaltenen Bäume mit bis zu drei Kanten. Anhand der Fingerprints kann jetzt ausgeschlossen werden, dass das Suchmuster in dem Molekül enthalten ist: Das Bit an der Position 11 ist im Fingerprint des Suchmusters gesetzt, jedoch nicht im Fingerprint des Moleküls, das demzufolge herausgefiltert würde. Der kanonische Bezeichner, der auf die Position 11 im Fingerprint des Suchmusters abgebildet wurde, ist `C-N$-N$=O$$` und beschreibt das C-Atom zusammen mit den drei benachbarten Atomen. Diese Struktur konnte durch Pfade nicht erfasst werden.

4.4.6 Analyse

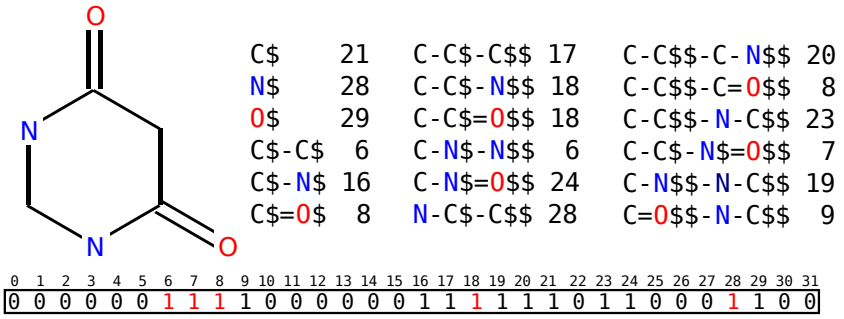
Aufgabe der Filterphase ist es, eine Kandidatenmenge zu erzeugen, und es ist wünschenswert, dass möglichst viele der Kandidaten auch tatsächliche Treffer sind. Es gibt zwei Gründe, warum False-Positives auftreten:

1. Die verwendeten Merkmale genügten nicht, um einen Teilgraph-Isomorphismus auszuschließen.

2. Aufgrund von Kollisionen konnten unterschiedliche Merkmale nicht mehr unterschieden werden.



(a) Suchmuster



(b) Molekül

Abbildung 4.8: Das Beispiel zeigt ein Suchmuster und ein Molekül zusammen mit den kanonischen Bezeichnern der darin enthaltenen Bäume mit bis zu 3 Kanten. Das Molekül könnte bereits in der Filterphase ausgeschlossen werden und würde nicht in die Kandidatenmenge aufgenommen.

Beide Fälle wirken sich ungünstig auf die Qualität des Index aus. Für die Korrektheit ist es entscheidend, dass keine Moleküle herausgefiltert werden, die das Suchmuster tatsächlich enthalten (*False-Negatives*).

Satz 4.9 (Keine False-Negatives). *Sei Q ein Suchmuster. Für alle nach dem oben genannten Verfahren herausgefilterten Graphen $G \in \mathcal{D}$ gilt $Q \lesssim G$.*

Beweis. Angenommen, es gibt einen Graphen G mit Fingerprint F_G, der bei der Suche nach Q mit Fingerprint F_Q herausgefiltert wurde, obwohl $Q \lesssim G$ gilt. Graphen werden nur dann aus der Kandidatenmenge entfernt, wenn im Fingerprint des Suchmusters ein Bit an einer Position auf Eins gesetzt ist, das im Fingerprint des Graphen auf Null gesetzt ist. F_Q enthält also an mindestens einer Position eine Eins, die in F_G auf Null gesetzt ist. Sei c die Position eines solchen Bits im Fingerprint.

Beim Generieren des Fingerprints F_Q muss also ein kanonischer Bezeichner B erzeugt worden sein mit $h(B) = c$. Da eine Hashfunktion deterministisch arbeitet, kann beim Generieren des Fingerprints F_G der kanonische Bezeichner B nicht erzeugt worden sein, da dann auch in F_G das Bit an Position c gesetzt worden wäre.

Angenommen, es handelt sich bei B um den kanonischen Bezeichner eines Baums. Der Algorithmus 4.1 findet alle in den Graphen Q bzw. G auftretenden Teilbäume bis zu einer bestimmten Größe. Alle isomorphen Teilbäume werden aufgrund der Konstruktion von

kanonischen Bezeichnern durch den gleichen Bezeichner identifiziert. Hieraus folgt, dass es einen Baum T mit $T \lesssim Q$ gibt, der nicht in G enthalten ist, also $T \lesssim G$. Aufgrund der Transitivität von Teilgraph-Isomorphie (Beobachtung 4.2) folgt somit, dass $Q \lesssim G$ gelten muss, was der Annahme widerspricht.

Sei B der kanonische Bezeichner eines einfachen Kreises, so folgt wiederum aus der Tatsache, dass alle einfachen Kreise gefunden werden und isomorphe Kreise durch kanonische Bezeichner eindeutig identifiziert werden, dass Q einen Kreis enthält, der in G nicht enthalten ist. Auch in diesem Fall liegt ein Widerspruch zur Annahme vor. □

4.5 Experimenteller Vergleich

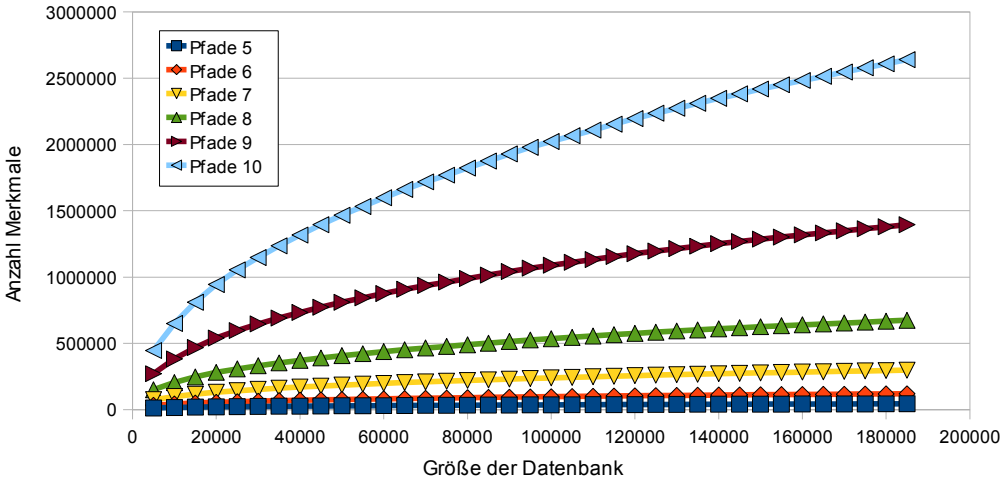
Anhand von experimentellen Vergleichen soll in diesem Abschnitt der neue Fingerprint mit dem bekannten pfadbasierten Ansatz verglichen werden. Alle Experimente wurden mit den unter Abschnitt 3.4.1 analysierten Instanzen und Suchmustern durchgeführt.

Aus den zur Verfügung gestellten Suchmustern wurden Gruppen gleicher Größe (Anzahl Kanten) gebildet. Dazu wurden wieder Gruppen der Größe 5, 8, 11, 14, 17, 20, 25, 30, 40, 60, 80, 100 erzeugt, indem je 100 Strukturen dieser Größe zufällig aus dem vorhandenen Datensatz mit Suchmustern ausgewählt wurden. Große Suchmuster sind eher von geringerem Interesse, da diese sehr spezifisch sind und nur in sehr wenigen Datenbankgraphen vorkommen. Suchmuster mit einer größeren Ergebnismenge führen zwangsläufig zu einer großen Kandidatenmenge und haben eine aufwendige Verifikationsphase zur Folge. Hier ist eine kleine Kandidatenmenge besonders wünschenswert. Daher wurde bei der Auswahl der Gruppen ein besonderes Gewicht auf Suchmuster mit weniger als 30 Kanten gelegt.

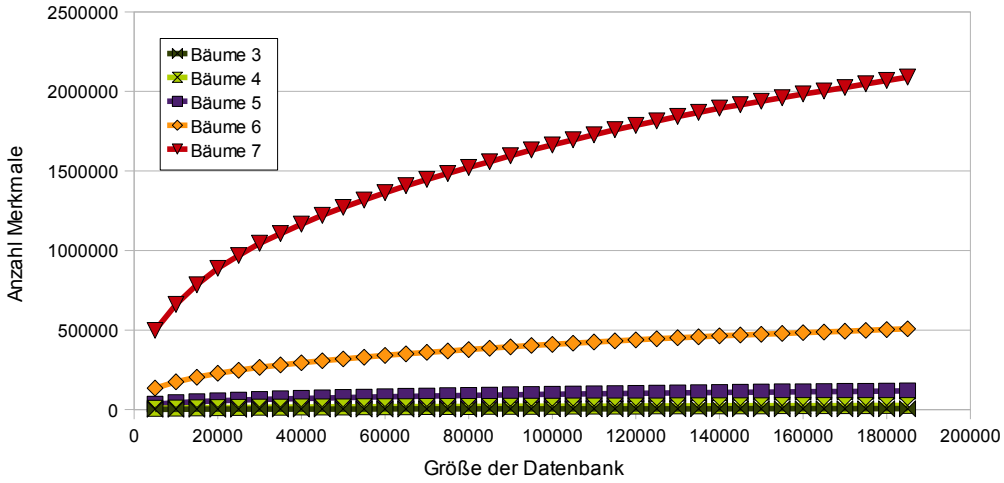
4.5.1 Wahl der Parameter

Die Effektivität eines Index, der mit Fingerprints arbeitet, hängt wesentlich von den verwendeten Parametern ab. Wie bereits erwähnt spielt die Anzahl der Kollisionen eine wichtige Rolle. Viele Kollisionen führen dazu, dass die Kandidatenmenge viele False-Positives enthält. Kollisionen sind jedoch unvermeidlich, wenn mehr unterschiedliche Merkmale existieren als der Fingerprint Bits zur Verfügung stellt. Das Verhältnis zwischen der Größe der Merkmalsmenge und der Länge des Fingerprints sollte also so gewählt werden, dass die Verschlechterung der Kandidatenmenge durch Kollisionen akzeptabel bleibt. In diesem Abschnitt werden anhand von Experimenten günstige Parameter ermittelt.

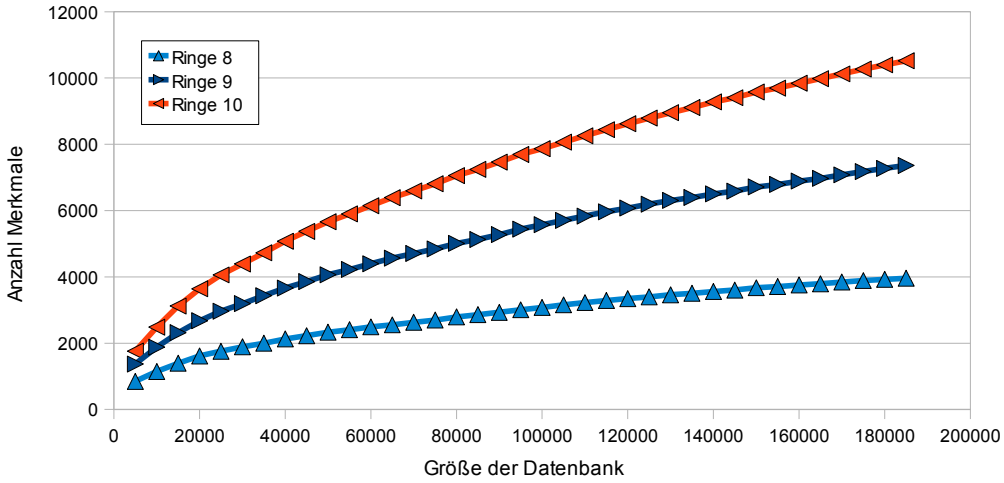
Die Größe des Fingerprints kann prinzipiell beliebig gewählt werden, hat jedoch direkten Einfluss auf die Größe des Index und die Laufzeit der Filterphase. Die Anzahl der extrahierten Merkmale lässt sich indirekt beeinflussen, indem die maximale Größe der Merkmale verändert wird. Der Zusammenhang zwischen der Größe der Merkmalsmenge und der Größe der Datenbank ist für verschiedene Klassen von Merkmalen ermittelt worden (Abbildung 4.9). Für alle Merkmalsklassen ist zu beobachten, dass das Wachstum



(a) Anzahl Pfade bei unterschiedlicher maximaler Länge



(b) Anzahl Bäume bei unterschiedlicher maximaler Größe



(c) Anzahl Ringe bei unterschiedlicher maximaler Größe

Abbildung 4.9: Wachstum der Merkmalsmenge in Abhängigkeit zur Größe der Datenbank.

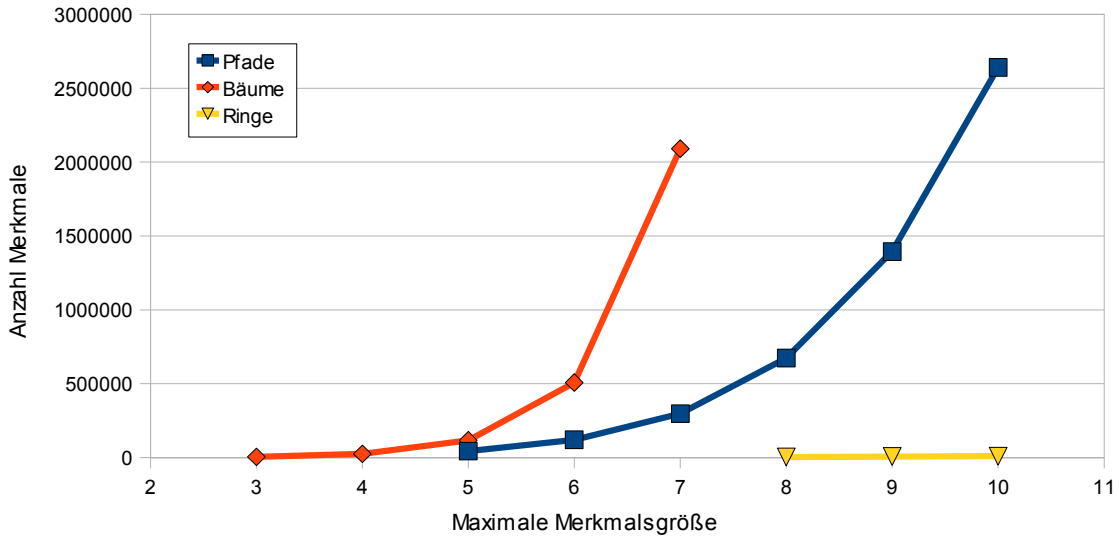


Abbildung 4.10: Wachstum der Merkmalsmenge in Abhängigkeit zur maximalen Merkmalsgröße für den vollständigen Datensatz mit 187.266 Molekülen.

besonders zu Beginn stark ist und mit zunehmender Größe der Datenbank nachlässt. Eine solche Entwicklung war zu erwarten, da sich viele Merkmale wiederholen. Eher unerwartet hingegen ist, dass bei fast 200.000 Graphen noch keine obere Schranke erreicht zu sein scheint, die die Größe der Merkmalsmenge begrenzt. Dies spricht für die große Vielfalt unterschiedlicher Merkmale in Molekülgraphen. Während die Zahl der unterschiedlichen Ringe relativ gering ist, fällt die Zahl unterschiedlicher Pfade und besonders die Zahl unterschiedlicher Bäume ausgesprochen groß aus und verdeutlicht die Unvermeidbarkeit von Kollisionen.

Abbildung 4.10 setzt das Wachstum der Merkmalsmenge ins Verhältnis zur maximalen Merkmalsgröße (gemessen in der Anzahl Kanten). Die Anzahl der Bäume nimmt erwartungsgemäß schneller zu als die Anzahl der Pfade.

Ein fairer Vergleich zwischen Bäumen und Pfaden als Merkmalsmenge soll im Folgenden durchgeführt werden, indem eine feste Fingerprintgröße von 2048 Bit festgelegt wird. Abbildung 4.10 legt nahe, dass für Bäume eine kleinere Merkmalsgröße angemessen ist als für Pfade. Welche Größe sich als optimal erweist, soll experimentell ermittelt werden.

Dabei ist ein weiterer Aspekt von Bedeutung, der die Qualität des Fingerprint-Index beeinflusst: Graphen mit dicht besetzen Fingerprints, die fast vollständig aus 1-Bits bestehen, können nur selten bei der Suche ausgeschlossen werden. Das Verhältnis von 1-Bits zur Anzahl der Bits insgesamt gibt also Hinweise darauf, ob die Parameter günstig gewählt sind.

Notation (Auslastung). Für einen Fingerprint F_G mit s Bits, c 1-Bits und $s - c$ 0-Bits bezeichnet $A(F_G) = \frac{c}{s}$ die *Auslastung* des Fingerprints.

Die Auslastung hängt einerseits von der Anzahl s der Bits im Fingerprint ab und andererseits von der Anzahl an Merkmalen, die ein einziges Molekül aufweist.

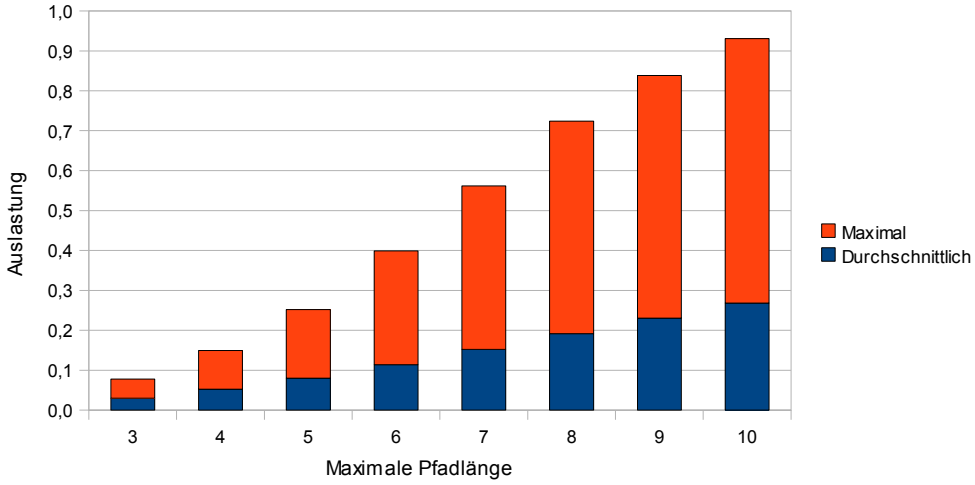
Erstellen der Indizes

Um sowohl für pfad- als auch für baumbasierte Indizes eine geeignete maximale Merkmalsgröße zu ermitteln, wurden mehrere Indizes für den vollständigen Moleküldatensatz erstellt: Für pfadbasierte Indizes wurden alle Parameter für die maximale Länge zwischen 3 und 10 verwendet, für Bäume wurde die maximale Baumgröße (Anzahl der Kanten) zwischen 3 und 8 gewählt. Obwohl Ringe als alleinige Indexmerkmale unzureichend sind, wurde auch ein Index mit Ringen mit maximaler Größe zwischen 7 und 10 erstellt. Dadurch lässt sich der Nutzen der Kombination von Pfaden und Ringen bzw. Bäumen und Ringen abschätzen.

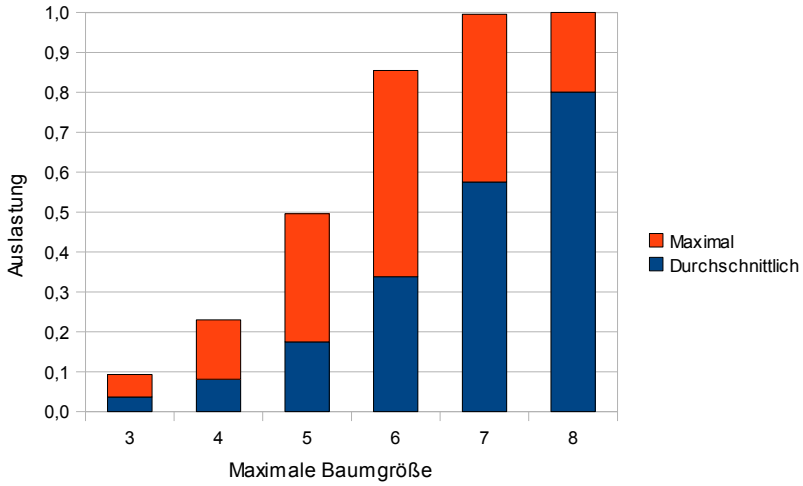
Abbildung 4.11 zeigt, wie sich die Auslastung des 2048-Bit Fingerprints für Pfade bzw. für Bäume in Abhängigkeit zur maximale Merkmalsgröße verändert. Die durchschnittliche Auslastung über alle Moleküle liegt für Pfade aller Länge unter $0,3$ (Abbildung 4.11(a)). Die maximale Auslastung liegt für Pfade mit maximaler Länge 10 bei über $0,9$. Moleküle, deren Fingerprints eine derart hohe Auslastung aufweisen, können nur schwer herausgefiltert werden.

Für Bäume (Abbildung 4.11(b)) nimmt die Auslastung des Fingerprints wesentlich schneller zu: Bei einer maximalen Baumgröße von 7 treten bereits Fingerprints auf, die fast vollständig aus 1-Bits bestehen. Graphen mit solchen Fingerprints können nie aus der Kandidatenmenge ausgeschlossen werden, so dass hier eine deutlich verschlechterte Effektivität zu erwarten ist. Die Auslastung gibt einen Anhaltspunkt zur Einschätzung der Merkmalsanzahl einzelner Moleküle, wobei jedoch zu beachten ist, dass die Auslastung nicht linear mit der Anzahl gefundener Merkmale zunimmt: Wenn bereits eine hohe Auslastung erreicht ist, ist die Abbildung eines hinzukommenden Merkmals auf ein 1-Bit sehr wahrscheinlich, wodurch sich die Auslastung nicht verändern würde. In diesen Fällen ist also mit einer sehr großen Anzahl von Kollisionen im Fingerprint zu rechnen. Obwohl in der Datenbank mehr unterschiedliche Pfade bis zur Länge 10 existieren als unterschiedliche Bäume mit maximaler Größe 7 (Abbildung 4.9), liegt die Auslastung der Fingerprints bei einer maximalen Baumgröße von 7 deutlich höher. Dies ist damit zu begründen, dass in jedem einzelnen Molekül viele unterschiedliche Merkmale existieren, die sich in anderen Molekülen wiederholen.

Für Ringe wurde die Auslastung nicht dargestellt, da sie erwartungsgemäß gering ausfällt. Der durchschnittliche Wert liegt stets unter $0,003$, der maximale unter $0,02$. Dies spricht dafür, dass Ringe mit den anderen Merkmalen kombiniert werden können, ohne die Auslastung stark zu beeinflussen.



(a) Auslastung durch Pfade als Merkmal



(b) Auslastung durch Bäume als Merkmal

Abbildung 4.11: Auslastung eines 2048-Bit Fingerprints, Maximal- und Durchschnittswerte bezogen auf die gesamte Datenbank mit 187.266 Molekülen.

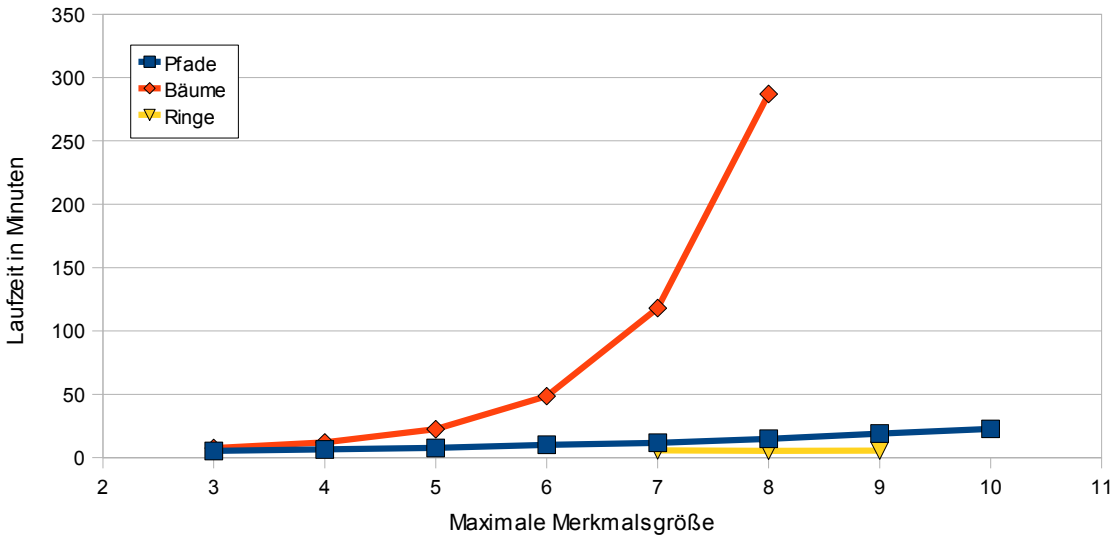


Abbildung 4.12: Benötigte Laufzeit zur Erfassung der Merkmale in allen 187.266 Molekülen der Datenbank.

Für die Laufzeit ist es entscheidend, wie viele Merkmale in einem Molekül gefunden werden - unabhängig davon, ob diese unterschiedlich oder identisch sind. Abbildung 4.12 zeigt, wie sich die Laufzeit für jede der drei Merkmalsklassen mit steigender maximaler Merkmalsgröße entwickelt. Während für Pfade und Ringe ein annähernd lineares Wachstum zu beobachten ist, wächst die Laufzeit für Bäume exponentiell. Dies ist einerseits mit der Merkmalsanzahl zu begründen: Es gibt nur wenige Ringe und deutlich weniger Pfade als Bäume. Hinzukommt, dass die Berechnung kanonischer Bezeichner für Bäume deutlich aufwendiger ist als für Pfade. Aufgrund der stark ansteigenden Laufzeit für Bäume scheint es sinnvoll, die maximale Größe kleiner als 7 zu wählen. Gerade in diesem Bereich verspricht auch die Auslastung der Fingerprints die besten Ergebnisse.

Qualität der Kandidatenmenge

Die erstellten Indizes sollen in diesem Abschnitt auf ihre Effektivität überprüft werden. Für pfad- und baumbasierte Indizes sollen die günstigsten Parameter ausgewählt und die Indizes anschließend im Detail miteinander verglichen werden. Außerdem soll das Potential der Kombination mit Ringmerkmalen untersucht werden. Um einen Überblick über die Effektivität der Indizes zu erhalten, wurden Suchanfragen nach allen 1200 Suchmustern (je 100 gleicher Größe) mit allen Indizes berechnet. Als Vergleichswert für die Qualität der Kandidatenmenge wurde zusätzlich ein exakter Teilgraph-Isomorphie-Test durchgeführt. Es sind also für alle Anfragen $n = 1, ..., 1200$ die Mengen $\mathcal{D}_n$ und für die 14 Indizes $i = 1, ..., 14$ die Kandidatenmengen $\mathcal{C}_n^i$ ermittelt worden. Um eine Übersicht über die Effektivität der Indi-

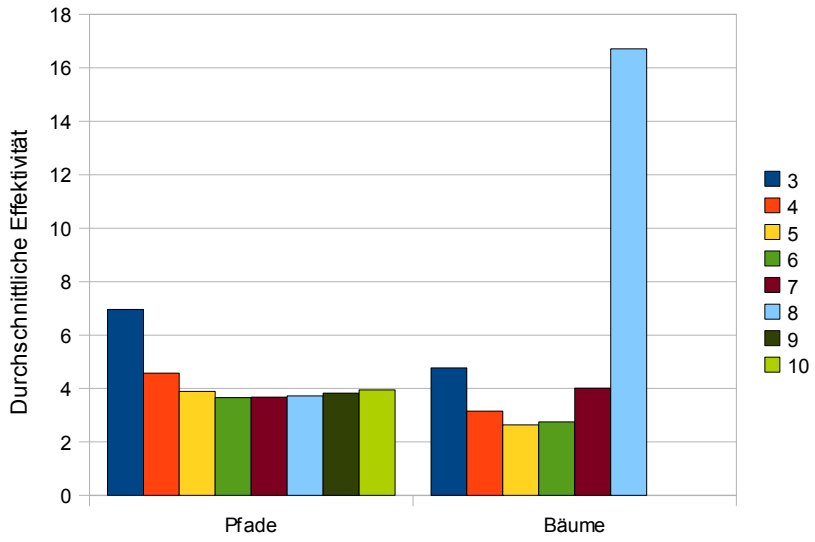


Abbildung 4.13: Durchschnittliche Effektivität von pfad- und baumbasierten Indizes mit unterschiedlicher maximaler Merkmalsgröße.

zes zu gewinnen, werden die Messwerte für jeden Index zur durchschnittlichen Effektivität E_i nach folgender Formel zusammengefasst:

$$E_i = \frac{|\mathcal{C}_1^i| + ... + |\mathcal{C}_n^i|}{|\mathcal{D}_1| + ... + |\mathcal{D}_n|} \tag{4.8}$$

Die Berechnung entspricht dem in [73] verwendeten Mittelwert $\frac{AVG(|\mathcal{C}_Q|)}{AVG(|\mathcal{D}_Q|)}$. Ein optimaler Index würde einen Wert von 1 erreichen, höhere Werte sprechen für eine schlechtere Effektivität.

Abbildung 4.13 zeigt die ermittelten Werte: Eine schlechte Effektivität liefert ein Index mit maximaler Baumgröße 8. Dies kann mit der ausgesprochen hohen Auslastung der Fingerprints bei diesem Index begründet werden (siehe Abbildung 4.11(b)). Ebenfalls schlecht schneidet der Pfad-Index mit maximaler Länge 3 ab. Dies hängt offenbar damit zusammen, dass Moleküle durch diese Merkmale nur unzureichend erfasst werden können. Sowohl bei Pfaden als auch bei Bäumen ist die Tendenz zu erkennen, dass die Effektivität mit niedriger Merkmalsgröße zunächst schlecht ist, sich mit zunehmender Merkmalsgröße verbessert und letztendlich wieder verschlechtert. Dies spricht dafür, dass für die feste Fingerprintgröße von 2048-Bit die relevanten Merkmalsgrößen getestet worden sind: Eine höhere Effektivität darf weder für kleinere noch für größere maximale Merkmalsgrößen erwartet werden. Für Pfade scheinen die Werte 7 und 8 optimal zu sein, für Bäume 5 und 6.

Im Vergleich zwischen Pfaden und Bäumen deutet sich an, dass Bäume eine höhere Effektivität bei deutlich kleinerer Merkmalsgröße ermöglichen. Bereits ein Index mit Bäumen bis zur Größe 4 bieten durchschnittlich eine bessere Effektivität als alle getesteten pfadbasierten Indizes.

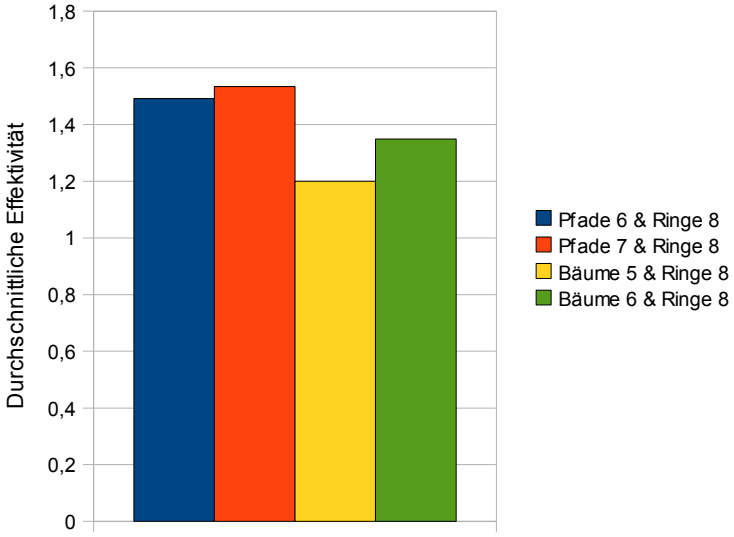


Abbildung 4.14: Durchschnittliche Effektivität von pfad- und baumbasierte Indizes mit unterschiedlicher maximaler Merkmalsgröße, kombiniert mit Ringmerkmalen bis zur Größe 8.

Es wurde ferner die durchschnittliche Effektivität der ringbasierten Indizes ermittelt, die zwischen $68,7$ und $65,1$ liegt und damit erwartungsgemäß schlecht ausfällt. Es ist aber denkbar, dass gerade die "falsch positiven" Moleküle, die allein durch Pfade oder Bäume nicht herausgefiltert werden, durch Ringmerkmale ausgeschlossen werden können. Daher werden die hier als optimal ermittelten Indizes mit Ringen kombiniert. Dazu wurden den beiden vielversprechenden pfad- bzw. baumbasierten Indizes Ringe bis zur Größe 8 als Merkmal hinzugefügt.

Abbildung 4.14 zeigt die mit diesen Indizes ermittelte durchschnittliche Effektivität. Es zeichnet sich sowohl für pfad- als auch für baumbasierte Indizes eine deutliche Verbesserung ab, wobei letztere weiterhin besser abschneiden. Der beste Index mit Bäumen bis zur Größe 5 und Ringen mit maximaler Größe 8 liefert eine durchschnittliche Effektivität von $1,2$ und liegt damit nah an dem optimalen Wert von 1.

Die Laufzeit, um die Kandidatenmenge zu generieren, unterscheidet sich erwartungsgemäß nicht wesentlich. Sie liegt meist deutlich unter einer Sekunde und der Mittelwert über alle Instanzen beträgt $0,1$ Sekunden. Abbildung 4.15 ist zu entnehmen, dass die Laufzeit mit zunehmender Größe der Kandidatenmenge etwas ansteigt. Mit einer Verschlechterung der Laufzeit muss möglicherweise gerechnet werden, wenn der Index eine Größe erreicht, die von MySQL nicht mehr im Speicher gehalten wird. Die Größe der Indizes hängt ausschließlich von der Größe der Fingerprints ab und beträgt bei 2048-Bit Fingerprints insgesamt 46,6 MB für 187.266 Moleküle. Dieser Wert beinhaltet neben den 2048 Bits für jedes Molekül auch einen Integer-Wert als ID, über die das zugehörige Molekül identifiziert werden kann.

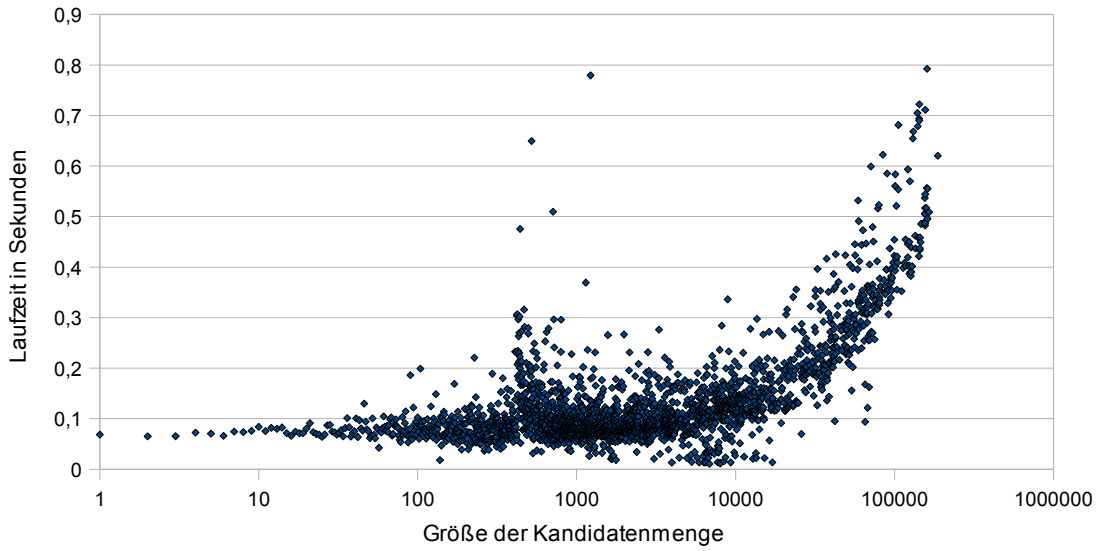


Abbildung 4.15: Die Laufzeit der Filterphase.

4.5.2 Detaillierter Vergleich

In dem vorherigen Abschnitt wurde gezeigt, dass der neue Fingerprint, der auf Bäumen und Ringen basiert, eine gute Effektivität liefert, wenn die Parameter für die Größe der Merkmale angemessen gewählt werden. Die Bibliothek CDK ist in der Lage, einen Fingerprint zu erstellen, der ausschließlich auf Pfaden basiert und zu Vergleichszwecken herangezogen werden könnte. In den Standardeinstellungen werden Fingerprints der Größe 1024 und Pfade bis zur Länge 8 verwendet. Der Fingerprint entspricht weitgehend der eigene Implementierung eines pfadbasierten Fingerprints. Die Ergebnisse des vorherigen Abschnitts legen nahe, dass der CDK-Fingerprint mit den Standardeinstellungen kaum konkurrenzfähig sein dürfte. Aus diesem Grund werden hier die vielversprechendsten Indizes des vorherigen Abschnitts genauer verglichen: Der Index, der Pfade bis zur Länge 6 verwendet und Ringe bis zur Größe 8, kurz "P6R8", und der baumbasierte Index mit Ringen "B5R8".

Die bisherigen Experimente geben keinen Aufschluss darüber, in welcher Weise die Qualität des Index mit der Größe des Suchmusters zusammenhängt. Für die Gruppen von Suchinstanzen unterschiedlicher Größe wurde jeweils der Mittelwert der Kardinalität der Kandidatenmengen und der Ergebnismengen gebildet.

Abbildung 4.16 zeigt die ermittelten Werte. Die Kurve der Ergebnismenge ist eine untere Schranke für die Kurven der Kandidatenmenge. Es ist wünschenswert, dass die Kurve der Kandidatenmenge möglichst nah an der Kurve der Ergebnismenge liegt. Es lässt sich zunächst feststellen, dass die durchschnittliche Größe der Ergebnismenge mit zunehmender Größe des Suchmusters stark abnimmt. Während Suchmuster der Größe 5, 8 und 11 durchschnittlich in weit über 1000 Molekülen gefunden werden, treten Suchmuster mit mehr als 30 Kanten im Durchschnitt in weniger als 2 der 187.266 Moleküle auf. Dies bestätigt die

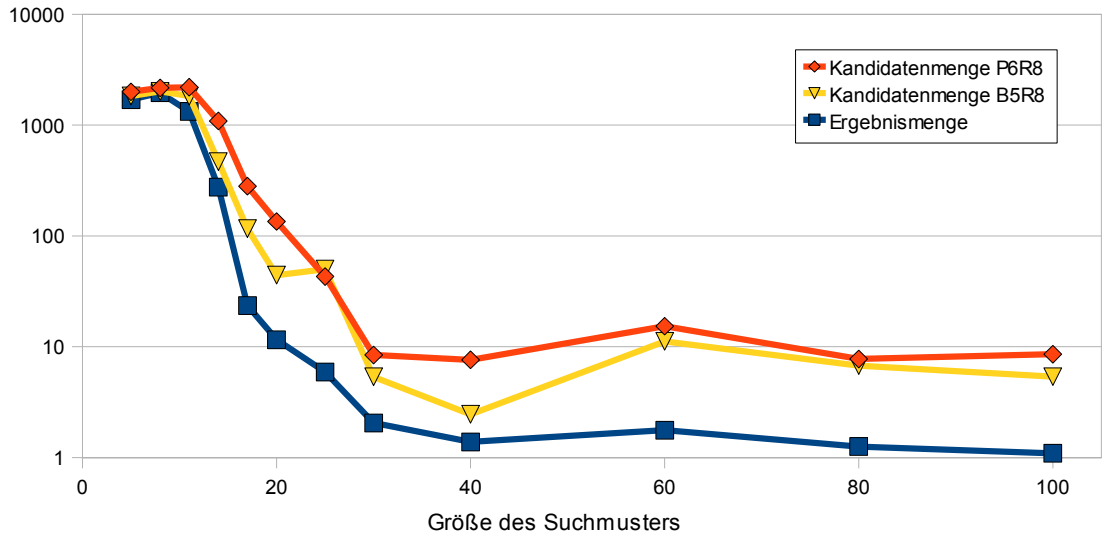


Abbildung 4.16: Größe der Ergebnis- und Kandidatenmenge.

Annahme, dass große Suchmuster nur selten in den Datenbankgraphen vorkommen. Die Auswahl der Suchmuster deckt besonders den wichtigen Bereich gut ab, in den die häufig auftretende Suchmuster fallen. Die Kandidatenmengen liegen besonders in diesem Bereich nah an der Ergebnismenge. Bei seltenen Suchmustern liegt die Größe der Kandidatenmenge für beide Indizes im Durchschnitt fast um einen Faktor 10 über der Größe der Ergebnismenge. Für die Antwortzeit des Systems sind Suchmuster kritisch, die häufig in der Datenbank auftreten, da diese zwangsläufig eine große Kandidatenmenge und somit eine lange Verifikationsphase zur Folge haben. Gerade in diesem Bereich ist es wünschenswert, dass die Kandidatenmenge wenige False-Positives enthält. Eine Kandidatenmenge der Größe 10, die nur einen tatsächlichen Treffer enthält, führt hingegen kaum zu einer Beeinträchtigung.

Die Größe der Kandidatenmengen der beiden Indizes verhält sich insgesamt ähnlich. In den meisten Fällen weist der Index B5R8 bessere Werte auf. Beachtet man die logarithmische Skalierung, fällt der Unterschied besonders bei Suchmustern mit einer Größe zwischen 17 und 40 (mit einer Ausnahme) deutlich aus. Um den Einfluss der Größe der Ergebnismenge genauer zu untersuchen, wurden die Messdaten in drei Gruppen eingeteilt: Die erste Gruppe umfasst alle “seltenen” Suchmuster mit bis zu 20 Treffern, die zweite Gruppe umfasst Suchmuster, die eine Ergebnismenge zwischen 20 und 80 Molekülen liefert und die dritte Gruppe alle Suchmuster mit einer größeren Ergebnismenge. Abbildung 4.17 zeigt die für jede Gruppe nach Formel (4.8) ermittelten Werte. Es zeigt sich, dass der Index B5R8 besonders bei der ersten und zweiten Gruppe deutlich besser abschneidet: Für die erste Gruppe ist die Kandidatenmenge im Durchschnitt um einen Faktor 3 kleiner, für die zweite Gruppe um mehr als einen Faktor 2. Für die besonders wichtige Gruppe der

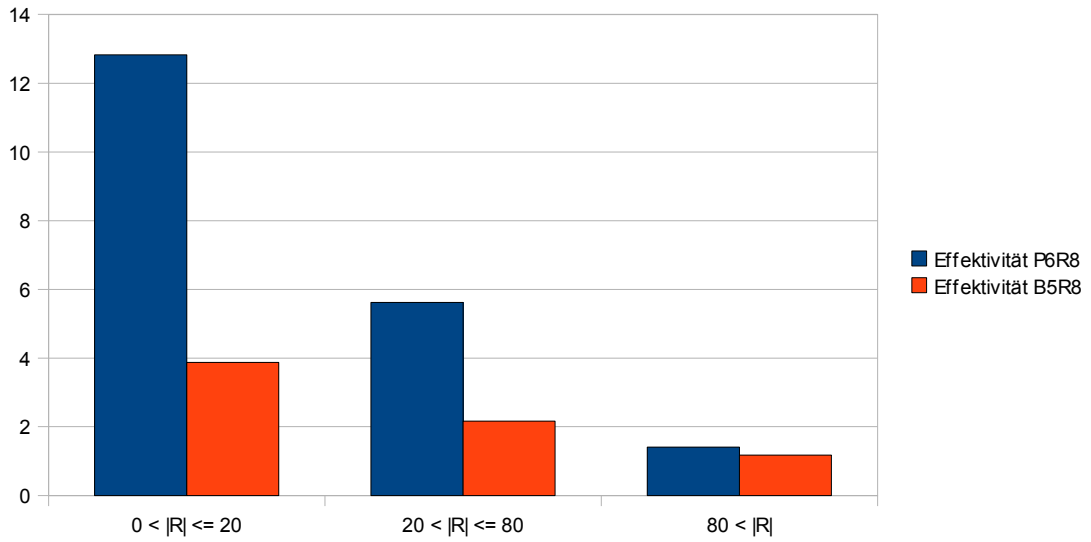


Abbildung 4.17: Mittlere Effektivität in Abhängigkeit zur Größe der Ergebnismenge $R = \mathcal{D}_Q$.

häufigen Suchmuster fällt der Unterschied geringer aus: Beide Indizes liefern hier ein gutes Ergebnis.

Berechnet man die Effektivität nach der Formel (4.8), so wird der Wert maßgeblich von Suchmustern bestimmt, die eine große Ergebnis- und Kandidatenmenge aufweisen. Dies ist durchaus sinnvoll, da gerade für solche Suchmuster eine gute Kandidatenmenge wichtig ist. Die Unterteilung der Messdaten nach Häufigkeit der Suchmuster legt nahe, dass die Unterschiede besonders für seltene Suchmuster groß ausfallen. Um dies genauer zu untersuchen, wurde der Mittelwert der Effektivität für Gruppen gleicher Größe nach folgender Formel berechnet:

$$E_i = \frac{1}{n} \sum_{i=1}^{n} \frac{|\mathcal{C}_n^i|}{|\mathcal{D}_n|} \tag{4.9}$$

Abbildung 4.18 zeigt die Resultate für beide Indizes. Es zeigt sich, dass der Index B5R8 in allen Bereichen besser abschneidet und nah am Optimum liegt. P6R8 zeigt bei Instanzen der Größe 17 und 20 eine deutliche Schwäche. Diese ist überwiegend auf einige wenige Suchmuster zurückzuführen, bei denen der Index eine sehr schlechte Kandidatenmenge berechnet hat. Eine genauere Untersuchung hat ergeben, dass diese fast ausschließlich aus Kohlen- und Wasserstoffatomen bestehen und nur wenige charakteristische Atome aufweisen. Diese werden durch Bäume offenbar besser ausgenutzt als durch Pfade. Dass dieses Problem bei Suchmustern der Größe 17 und 20 auftritt, könnte damit zusammenhängen, dass größere Suchmuster generell mehr charakteristische Merkmale liefern und kleinere Suchmuster ohne viele charakteristische Merkmale auch tatsächlich zu einer großen Ergebnismenge führen. Suchmuster, die nur häufig auftretende Merkmale enthalten, aber

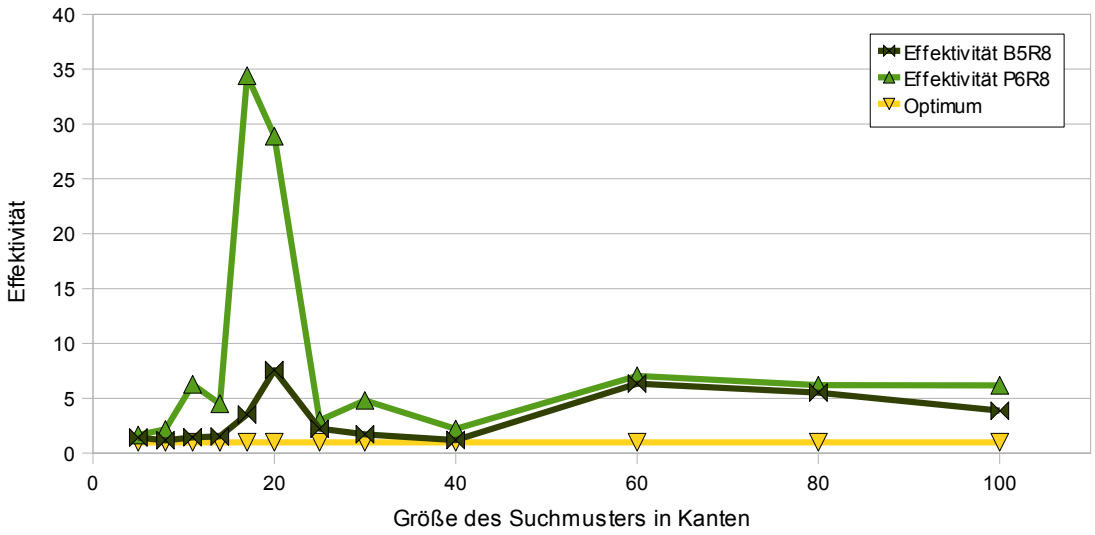


Abbildung 4.18: Mittlere Effektivität in Abhängigkeit zur Größe des Suchmusters.

dennoch eine spezifische, seltene Struktur aufweisen, stellen ein Problem dar. Suchmuster mit einer Größe von 17 oder 20 sind groß genug, um eine sehr spezielle Struktur aufzuweisen, die von den deutlich kleineren Merkmalen nicht mehr erfasst werden kann. Um diese Hypothese zu untermauern, wären weitere Experimente mit einer größeren Anzahl Suchmustern dieser Größenordnung notwendig.

4.5.3 Suchmuster mit Wildcards

Für die beiden ausgewählten Fingerprints wurden Experimente mit den in Abschnitt 3.4.1 beschriebenen Wildcard-Instanzen durchgeführt. Alle variablen Elemente des Suchmusters wurden, wie in Abschnitt 4.3 erläutert, vor der Berechnung des Fingerprints entfernt. Abbildung 4.19(a) zeigt die durchschnittliche Größe der Kandidatenmengen und der Ergebnismengen für die Suchmuster unterschiedlicher Größe. Im Vergleich zu den Ergebnissen mit festen Suchmustern (vgl. Abbildung 4.16) ist nun ein deutlicher Unterschied zwischen der Kandidatenmenge und der tatsächlichen Ergebnismenge zu erkennen. Dieser zeigt sich bei beiden Indizes, wobei der Index B5R8 etwas besser abschneidet.

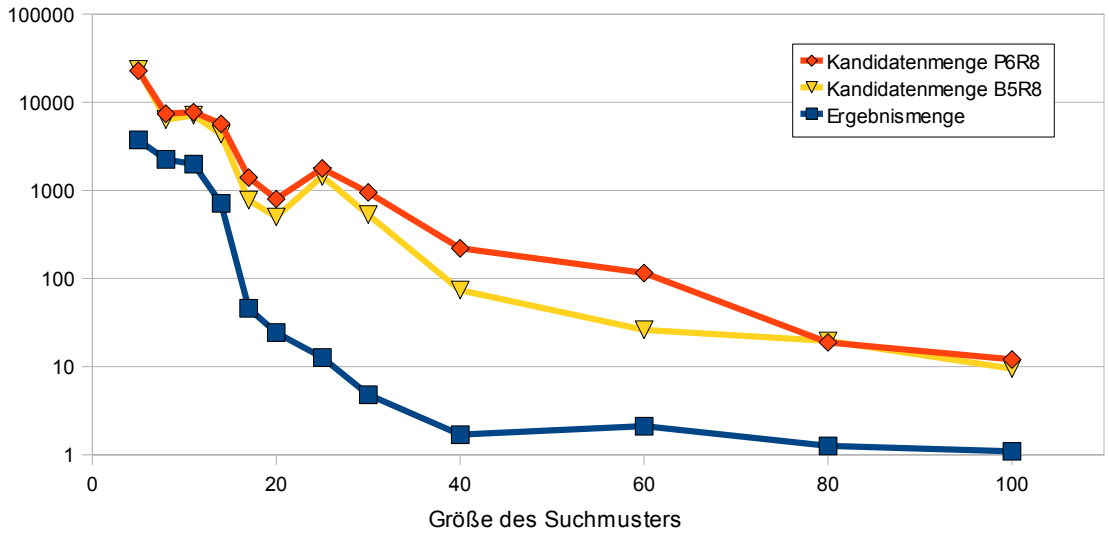
Abbildung 4.19(b) zeigt die analog zum vorangehenden Abschnitt berechnete Effektivität der Indizes. Auch hier zeigt sich die deutliche Verschlechterung der Effektivität bei Instanzen mit Wildcards, wobei wiederum der baumbasierte Fingerprint etwas bessere Ergebnisse liefert. Eine besonders drastische Abweichung ist für Instanzen der Größe 15-40 zu beobachten. Eine genauere Analyse der Daten ergab, dass die schlechten Effektivitätswerte vor allem auf einige wenige Instanzen zurückzuführen sind, die eine sehr kleine Ergebnismenge und eine sehr große Kandidatenmenge aufweisen. Solche Instanzen weisen vermutlich Charakteristika auf, die in nur wenigen Molekülen vorkommen und bei der

Löschung variabler Elemente verloren gehen. Für mehr als 50% der Suchmuster Q mit Wildcards ist für beide Indizes der Wert $\frac{|\mathcal{C}_Q|}{|\mathcal{D}_Q|}$ kleiner als 3.

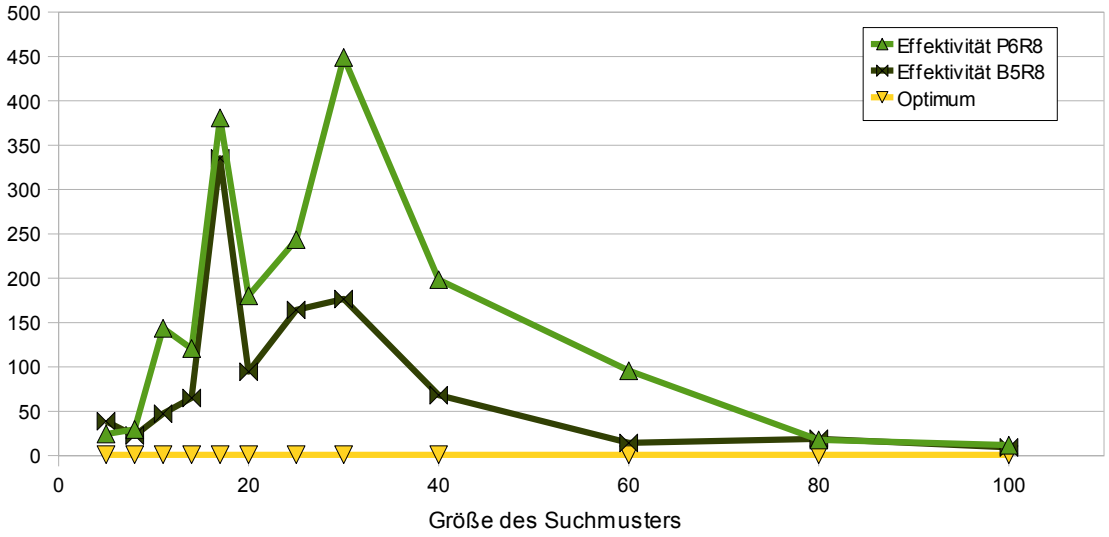
4.5.4 Fazit

Der experimentelle Vergleich zeigt, dass sich Bäume als Merkmale nutzen lassen, um einen Hash-Key Fingerprint mit guter Filterwirkung zu erhalten. Als Nachteil von Bäumen gegenüber Pfaden ist zu nennen, dass die Laufzeit zum Generieren des Index höher ist. Die Anzahl der Teilbäume nimmt stark mit der erlaubten Maximalgröße zu. Für Bäume bis zu einer relevanten maximalen Größe ist sowohl die Laufzeit als auch die Größe der Merkmalsmenge akzeptabel. Ein klarer Vorteil liegt in der besseren Filterwirkung von baumbasierten Fingerprints. Bereits relativ kleine Bäume ermöglichen eine bessere Effektivität als Pfade größerer Länge. Das Indizieren zusätzlicher Ringstrukturen hat sich als nützliche Ergänzung erwiesen. Für Suchmuster, die Wildcards enthalten, kann die Filterwirkung der getesteten Indizes stark nachlassen. Während für einen großen Teil der Anfragen weiterhin zufriedenstellende Ergebnisse berechnet werden konnten, wurde für einige Anfragen eine unzureichende Filterwirkung ermittelt. Die Vermutung, dass ein baumbasierter Fingerprint hier deutlich bessere Ergebnisse liefert, konnte nicht bestätigt werden. Das Entfernen aller variablen Elemente im Suchmuster lässt zu oft Fragmente zurück, die keine ausreichend selektiven Merkmale bieten. Das in Abschnitt 4.3 beschriebene Verfahren zur schrittweisen Abstraktion der Label im Graphen könnte eine Verbesserung herbeiführen, wurde aber nicht experimentell untersucht. Insgesamt stellt ein baumbasierter Fingerprint eine geeignete Alternative zu pfadbasierten Fingerprints dar.

Ein Vergleich mit Data-Mining-basierten Verfahren wie GIndex konnte im Rahmen der Diplomarbeit nicht durchgeführt werden. GIndex ist in der Lage, komplexere Merkmale zu indizieren, beschränkt sich aber auf eine kleine Auswahl der möglichen Merkmale. Der hier entwickelte Ansatz erfasst kleinere Merkmale vollständig, büßt aber durch Kollisionen im Fingerprint einen Teil der damit möglichen Effektivität ein. Ob sich der deutlich höhere Aufwand von GIndex durch eine höhere Effektivität bei der Suche in Molekül datenbanken auszahlt, bleibt unbeantwortet.



(a) Größe der Ergebnis- und Kandidatenmenge



(b) Mittlere Effektivität in Abhängigkeit zur Größe des Suchmusters

Abbildung 4.19: Ergebnisse mit Wildcard-Suchmustern.

Kapitel 5

Zusammenfassung und Ausblick

Um Informationen aus einer Moleküldatenbank auf angemessene Weise abrufen zu können, ist ein System zur Substruktursuche von großem Nutzen. Obwohl derartige Systeme in der Chemie häufig eingesetzt werden und seit geraumer Zeit Gegenstand der Forschung sind, gestaltet sich die Implementierung eines solchen Systems nicht einfach: Es gibt kaum geeignete frei verfügbare Software, die in ein Open-Source-Projekt wie Scaffold Hunter integriert werden könnte und die benötigte Funktionalität sowie die für die Suche notwendige Effizienz bietet.

Der direkte Zusammenhang zwischen chemischen Verbindungen und Graphen erlaubt es, die Substruktursuche als graphentheoretisches Problem zu betrachten. Die Suche in Graphen wird üblicherweise nach dem Filter-Verifikations-Ansatz gelöst. Somit lassen sich zwei Teilprobleme identifizieren, die beide in dieser Diplomarbeit behandelt wurden.

Die Verifikationsphase korrespondiert mit dem Teilgraph-Isomorphie-Problem. Die große praktische Relevanz dieses Problems spiegelt sich in der großen Anzahl publizierter Verfahren zu diesem Thema wieder. Für die Suche in Molekülgraphen wird häufig der Algorithmus von Ullmann (1974) empfohlen. Für andere Anwendungsgebiete sind nachfolgend zahlreiche neue Verfahren vorgestellt worden. In dieser Arbeit ist ein neuer Algorithmus entwickelt worden, der Ideen unterschiedlicher aktueller Ansätze aufgreift, und speziell die Anwendung bei der Substruktursuche unterstützt. Dazu wird vorab in einem Vorverarbeitungsschritt das Suchmuster analysiert und eine Folge möglichst zusammenhängender Knoten berechnet. Von diesem einmaligen Vorverarbeitungsschritt können anschließend alle TGI-Tests mit dem gleichen Suchmuster profitieren. Bei der Vorverarbeitung können auf einfache Weise Informationen über die Häufigkeit von Knotenlabeln in Molekülgraphen genutzt werden, um die Folge zu optimieren. Diese Idee ist mit Verfahren zur Suchplanoptimierung vergleichbar. Der Algorithmus baut mit Hilfe der vorberechneten Knotenfolge eine partielle Abbildung schrittweise aus, wobei die Idee, möglichst benachbarte Knoten der Abbildung hinzuzufügen, dem Vorgehen des VF2-Algorithmus entspricht. Die vorberechnete Datenstruktur wird zudem genutzt, um die Anzahl möglicher Kandidaten für einen

Knoten des Suchmusters auf einfache Weise bestimmen zu können und auf die Nachbarn eines bereits abgebildeten Knotens zu beschränken. Dies ist besonders für Graphen mit geringem Grad wie Molekülgraphen vorteilhaft. Außerdem unterstützen der Algorithmus und die Suchplanoptimierung die Verwendung von Wildcards. In einem experimentellen Vergleich konnte gezeigt werden, dass der neu entwickelte Algorithmus deutlich bessere Laufzeiten als alle verfügbaren getesteten Implementierungen anderer Verfahren liefert und die verwendete Suchplanoptimierung für Molekülgraphen sehr gut geeignet ist. Die ausgesprochen gute Laufzeit des neuen Algorithmus verdeutlicht, dass das NP-vollständige TGI-Problem für Molekülgraphen in der Praxis effizient gelöst werden kann, wenn die besonderen Eigenschaften dieser Graphen genutzt werden.

Für die Filterphase wurde ein neues Verfahren entwickelt, das die in der Chemie gängige Nutzung von Hash-Key Fingerprints mit der Verwendung komplexer Merkmale kombiniert. Während pfadbasierte Hash-Key Fingerprints in der Chemie verbreitet sind, ist die Nutzung komplexerer Merkmale wie Bäume oder Graphen im Zusammenhang mit einer beschränkten Merkmalsmenge vorgeschlagen worden, die mit Hilfe von Methoden des Frequent Graph Minings berechnet werden muss. Die bessere Effektivität solcher Verfahren mit komplexen Merkmalen motiviert die Entwicklung eines neuen Hash-Key Fingerprints in dieser Arbeit. Dieser erfasst sämtliche Bäume und Ringe bis zu einer bestimmten Größe, die in einem Molekülgraphen enthalten sind, und damit deutlich komplexere Strukturen als Pfade. Basierend auf bekannten Verfahren zur Berechnung kanonischer Bezeichner wird beschrieben, wie derartige Merkmale aus Graphen gewonnen und in einem Fingerprint repräsentiert werden können. Die höhere Effektivität des neuen Fingerprints gegenüber pfadbasierten Fingerprints konnte in experimentellen Vergleichen gezeigt werden. Die Indizierung von Ringen hat sich dabei als sinnvolle Ergänzung zu Bäumen herausgestellt. Die höhere Laufzeit zur einmaligen Erzeugung eines baumbasierten Fingerprint-Index kann durch die verbesserte Filterwirkung gerechtfertigt werden. Das Problem der Unterstützung von Wildcards im Suchmuster konnte mit dem neuen Fingerprint nicht vollständig zufriedenstellend gelöst werden. Es wurden jedoch Ideen zur Lösung dieser Problematik erarbeitet. Eine sinnvollen Umsetzung erfordert genaue Informationen über typische Wildcards und war im Rahmen der Diplomarbeit nicht möglich, so dass die Wirksamkeit in der Praxis offen gelassen werden muss.

Die Entwicklung der neuen Verfahren und ihre Implementierung ermöglichen in Kombination mit einem Strukturediteor ein effizientes und benutzerfreundliches System zur Substruktursuche, das in Scaffold Hunter integriert wurde. Die Verwendung eines graphischen Strukturediteors und seine Modifikation zur Unterstützung von Wildcards erlauben eine komfortable Eingabe von variablen Suchmustern. Der vorhandene Filterdialog wurde um eine Funktion zur Suche in Molekülen und Molekülgerüsten erweitert. Ferner können mit Hilfe der Suchfunktion Moleküle im Baumdiagramm hervorgehoben werden, die bestimmte Strukturmerkmale aufweisen. Hierdurch wird die Nutzbarkeit des Programms zur Er-

forschung des chemischen Strukturraums erhöht. Eine geplante Erweiterung, die auf der geleisteten Arbeit aufbaut, ist die Realisierung einer Funktion zur Ähnlichkeitssuche.

Weitere Informationen

In diesem Kapitel wird ein grober Überblick über die Struktur der Implementierung gegeben. Wesentliche Designentscheidungen werden anhand von UML-Diagrammen kurz erläutert.

Datenstruktur für Molekülgraphen

Abbildung 1 zeigt den Aufbau der Datenstruktur für Graphen und das Zusammenspiel mit dem Toolkit CDK. Dieses stellt für chemische Verbindungen, bestehend aus Atomen (`IAtom`) und Bindungen (`IBond`), Klassen bereit, die jedoch keine effiziente Traversierung der Datenstruktur erlauben. Diese Datenstruktur ist grundlegend für die Funktionalität von CDK und kann aus einer Vielzahl von Austauschformaten geladen werden. Die Struktur eines gelabelten Graphen wird durch die allgemeinen Klassen `Graph`, `Node` und `Edge` bereitgestellt und unterstützen eine effiziente Traversierung des Graphen. Die davon erbenden Klassen repräsentieren Molekülgraphen und erweitert die Oberklassen um die Verbindung zu der Datenstruktur des CDK. Instanzen der Klassen `MolGraph`, `MolNode` und `MolEdge`

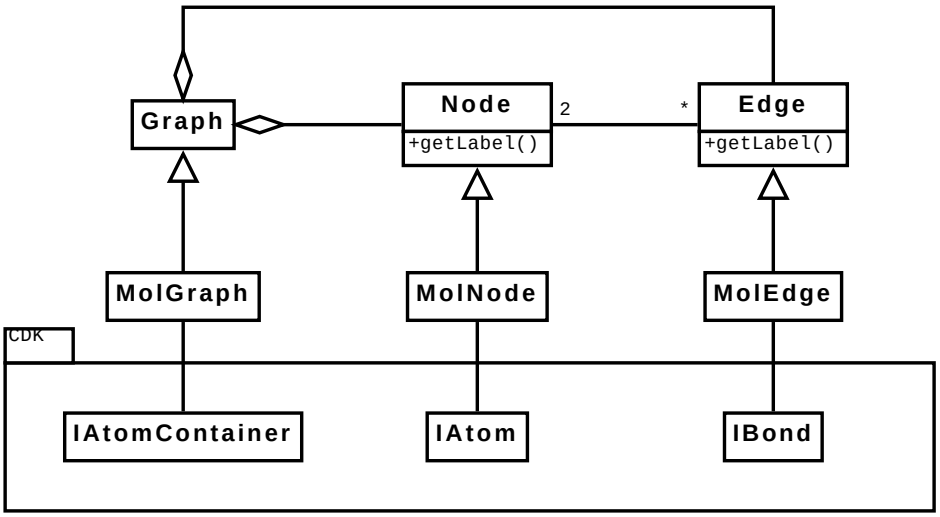


Abbildung 1: Klassendiagramm Molekülgraph

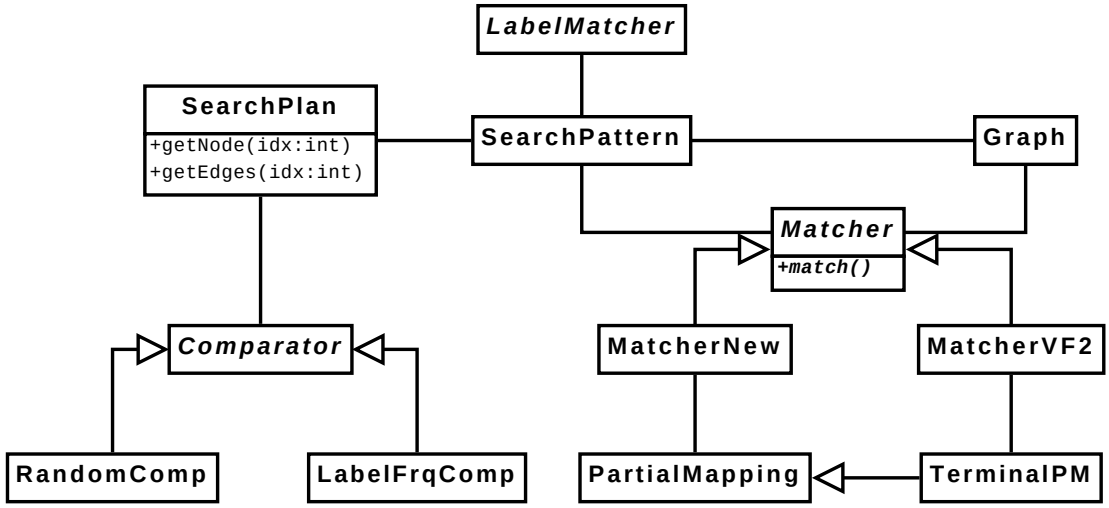


Abbildung 2: Klassendiagramm TGI-Algorithmus

werden direkt aus der CDK-Datenstruktur aufgebaut, wobei das Label durch den Atom- bzw. Bindungstyp bestimmt wird.

Implementierung des TGI-Algorithmus

Abbildung 2 zeigt Klassen, die für den TGI-Algorithmus relevant sind. Die abstrakte Klasse `Matcher` ist zentral und repräsentiert die Funktionalität, zwei Graphen aufeinander abzubilden. Dabei wird das Suchmuster durch die Klasse `SearchPattern` zur Verfügung gestellt. Diese Klasse speichert neben der Referenz auf den eigentlichen Graphen den zugehörigen, im Vorverarbeitungsschritt berechneten Suchplan. Die Klasse `SearchPlan` implementiert den Vorverarbeitungsschritt des Algorithmus und erlaubt eine integrierte Optimierung. Diese kann durch beliebige Klassen gesteuert werden, die das Java Interface `Comparator` implementieren. Hierdurch wird die Priorisierung für Knoten des Suchmusters im Rahmen der Freiräume, die bei der Erzeugung einer maximal zusammenhängenden Folge bestehen, erlaubt. `LabelFrqComp` verfügt über eine Liste der Häufigkeit von Knotenlabeln und berechnet die Prioritäten wie in Abschnitt 3.3.4 beschrieben. Das Interface `LabelMatcher` ermöglicht den Vergleich von Labeln und kann auf verschiedene Weise implementiert werden. Listen und Negativ-Listen von Labeln werden in eine Datenstruktur überführt, die eine effiziente Überprüfung erlaubt.

Für die abstrakte Klasse `Matcher` wurden zwei Implementierungen erstellt. Die erbende Klasse `MatcherNew` enthält den neu entwickelten Backtracking-Algorithmus, die Klasse `MatcherVF2` eine Implementierung des VF2-Algorithmus. Beide Algorithmen bauen schrittweise eine partielle Abbildung aus. Diese wird durch die Klasse `PartialMapping` repräsentiert.

sentiert. Für den VF2-Algorithmus wurde die Unterklassen `TerminalPM` angefertigt, die zusätzlich zu der eigentlichen Abbildung die Terminalmengen verwaltet.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *Open Babel: The Open Source Chemistry Toolbox.* http://openbabel.org.

[2] *Daylight Theory Manual v4.9.* http://www.daylight.com/dayhtml/doc/theory, Januar 2008.

[3] *MySQL 5.1 Reference Manual.* Internet, 2009.

[4] *PubChem Substructure Fingerprint v1.3.* Internet, 2009.

[5] Aho, Alfred V. und John E. Hopcroft: *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[6] Barnard, John M.: *Substructure searching methods: Old and new.* Journal of Chemical Information and Computer Sciences, 33(4):532–538, 1993.

[7] Barrow, Harry G. und Rod M. Burstall: *Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques.* Inf. Process. Lett., 4(4):83–84, 1976.

[8] Batz, Gernot Veit: *An Optimization Technique for Subgraph Matching Strategies.* Technischer Bericht 2006-7, Universität Karlsruhe, IPD Goos, April 2006.

[9] Bederson, Benjamin B., Jesse Grosjean und Jon Meyer: *Toolkit Design for Interactive Structured Graphics.* IEEE Trans. Softw. Eng., 30(8):535–546, 2004.

[10] Bonnici, V, R Di Natale, A Ferro, R Giugno, M Mongiovi, G Pigola, A Pulvirenti und D Shasha: *Enhancing Graph Database Indexing By Suffix Tree Structure*, 2009.

[11] Brown, Nathan: *Chemoinformatics - an introduction for computer scientists.* ACM Comput. Surv., 41(2), 2009.

[12] Callaghan, Steve, Gerard Ellis und James Harland: *Filters for Graph Matching.* In: *CATS*, Seiten 121–136, 1998.

[13] Chi, Y., R. Muntz, S. Nijssen und J. Kok: *Frequent subtree mining – an overview*, 2005.

[14] Chi, Yun, Yirong Yang und Richard R. Muntz: *Indexing and Mining Free Trees.* Data Mining, IEEE International Conference on, 0:509, 2003.

[15] Cohen, Edith, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D. Ullman und Cheng Yang: *Finding Interesting Associations without Support Pruning.* IEEE Trans. Knowl. Data Eng., 13(1):64–78, 2001.

[16] Conte, D., P. Foggia, C. Sansone und M. Vento: *Thirty Years Of Graph Matching In Pattern Recognition.* International Journal of Pattern Recognition and Artificial Intelligence, 2004.

[17] Cook, Diane J. und Lawrence B. Holder: *Mining Graph Data.* John Wiley & Sons, 2006.

[18] Cordella, L. P., P. Foggia, C. Sansone und M. Vento: *Performance Evaluation of the VF Graph Matching Algorithm.* In: *ICIAP '99: Proceedings of the 10th International Conference on Image Analysis and Processing*, Seite 1172, Washington, DC, USA, 1999. IEEE Computer Society.

[19] Cordella, L. P., P. Foggia, C. Sansone und M. Vento: *An Improved Algorithm for Matching Large Graphs.* In: *3rd IAPR-TC15 Workshop on Graph-based Representations*, 2001.

[20] Cordella, L.P., P. Foggia, C. Sansone, F. Tortorella und M. Vento: *Graph Matching: a Fast Algorithm and its Evaluation.* In: *In Proc. of the 14th International Conference on Pattern Recognition*, Seiten 1582–1584, 1999.

[21] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[22] Diestel, Reinhard: *Graphentheorie.* Springer-Verlag, Heidelberg, 3 Auflage, September 2006.

[23] Dörr, Heiko: *Bypass Strong V-Structures and Find an Isomorphic Labelled Subgraph in Linear Time.* In: *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, Seiten 305–318, London, UK, 1995. Springer-Verlag.

[24] Durant, Joseph L., Burton A. Leland, Douglas R. Henry und James G. Nourse: *Reoptimization of MDL Keys for Use in Drug Discovery.* Journal of Chemical Information and Computer Sciences, 42(5):1273–1280, 2002.

[25] Eppstein, David: *Subgraph isomorphism in planar graphs and related problems.* J. Graph Algorithms & Applications, 3(3):1–27, 1999.

[26] Faulon, Jean-Loup: *Isomorphism, Automorphism Partitioning, and Canonical Labeling Can Be Solved in Polynomial-Time for Molecular Graphs.* Journal of Chemical Information and Computer Sciences, 38(3):432–444, 1998.

[27] Ferro, A., R. Giugno, M. Mongiovi, A. Pulvirenti, D. Skripin und D. Shasha: *Graphblast: Multi-Feature Graphs Database Searching.* In: *Workshop On Network Tools And Applications In Biology - NETTAB*, 2007.

[28] Ferro, A., R. Giugno, M. Mongiovì, A. Pulvirenti, D. Skripin und D. Shasha: *GraphFind: enhancing graph searching by low support data mining techniques.* BMC bioinformatics, 9 Suppl 4, 2008.

[29] Figueras, J.: *Substructure search by set reduction.* Journal of Chemical Documentation, 12:237–244, 1972.

[30] Foggia, P., C. Sansone und M. Vento: *A performance comparison of five algorithms for graph isomorphism.* Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, Seiten 188–199, 2001.

[31] Gasteiger, Johann und Thomas Engel (Herausgeber): *Chemoinformatics: A Textbook.* Wiley VCH, October 2003.

[32] Giugno, R. und D. Shasha: *GraphGrep: A fast and universal method for querying graphs.* In: *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, Band 2, Seiten 112–115 vol.2, 2002.

[33] Gorecki, Adalbert, Anke Arndt, Arbia Ben Ahmed, Andre Wiesniewski, Cengizhan Yücel, Gebhard Schrader, Henning Wagner, Michael Rex, Nils Kriege, Philipp Büderbender, Sergej Rakov und Vanessa Bembenek: *ChemBioSpace Explorer: PG504 Endbericht*, 2008.

[34] Grochow, Joshua A. und Manolis Kellis: *Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking.* In: *RECOMB*, Seiten 92–106, 2007.

[35] Gusfield, Dan: *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[36] He, Huahai und Ambuj K. Singh: *Closure-Tree: An Index Structure for Graph Queries.* In: *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Helmer, Sven und Guido Moerkotte: *A performance study of four index structures for set-valued attributes of low cardinality.* The VLDB Journal, 12(3):244–261, 2003.

[38] Irniger, Christophe und Horst Bunke: *Graph Matching : Filtering Large Databases of Graphs Using Decision Trees.* In: *Proc. 3rd IAPR-TC15 Workshop on Graph-Based Representations in Pattern Recognition*, Seiten 239–249, 2001.

[39] Irniger, Christophe und Horst Bunke: *Decision Tree Structures for Graph Database Filtering.* In: *SSPR/SPR*, Seiten 66–75, 2004.

[40] Irniger, Christophe und Horst Bunke: *Graph Database Filtering Using Decision Trees.* In: *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 3*, Seiten 383–388, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Kelley, Brian: *FROWNS.* http://frowns.sourceforge.net.

[42] Koch, Marcus A., Ansgar Schuffenhauer, Michael Scheck, Stefan Wetzel, Marco Casaulta, Alex Odermatt, Peter Ertl und Herbert Waldmann: *Charting biologically relevant chemical space: a structural classification of natural products (SCONP).* Proceedings of the National Academy of Sciences of the United States of America, 102(48):17272–17277, November 2005.

[43] Krause, Stefan, Egon Willighagen und Christoph Steinbeck: *JChemPaint - Using the Collaborative Forces of the Internet to Develop a Free Editor for 2D Chemical Structures.* Molecules, 5(1):93–98, 2000.

[44] Landrum, G.: *RDKit.* http://www.rdkit.org.

[45] Levi, G: *A note on the derivation of maximal common subgraphs of two directed or undirected graphs.* Calcolo, Jan 1973.

[46] López-Presa, José Luis und Antonio Fernández Anta: *Fast Algorithm for Graph Isomorphism Testing.* In: *SEA*, Seiten 221–232, 2009.

[47] Luccio, F., A. Enriquez, P. Rieumont und L. Pagli: *Bottom-up subtree isomorphism for unordered labeled trees*, 2004.

[48] Matousek, Jirí und Robin Thomas: *On the complexity of finding iso- and other morphisms for partial k-trees.* Discrete Mathematics, 108(1-3):343–364, 1992.

[49] McKay, Brendan: *Practical graph isomorphism.* In: *Numerical mathematics and computing, Proc. 10th Manitoba Conf., Winnipeg/Manitoba 1980, Congr. Numerantium 30*, Seiten 45–87, 1981.

[50] Meinl, Thorsten, Marc Wörlein, Olga Urzova, Ingrid Fischer und Michael Philippsen: *The ParMol Package for Frequent Subgraph Mining.* ECEASST, 1, 2006.

[51] Messmer, Bruno T.: *Efficient Graph Matching Algorithms.* Doktorarbeit, 1995.

[52] P. Cordella, Luigi, Pasquale Foggia, Carlo Sansone und Mario Vento: *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs.* IEEE Trans. Pattern Anal. Mach. Intell., 26(10):1367–1372, 2004.

[53] Ray, Louis C. und Russell A. Kirsch: *Finding Chemical Records by Digital Computers.* Science, 126(3278):814–819, 1957.

[54] Raymond, John W. und Peter Willett: *Maximum common subgraph isomorphism algorithms for the matching of chemical structures.* Journal of Computer-Aided Molecular Design, 16(7):521–533, 2002.

[55] Rogers, D., R. D. Brown und M. Hahn: *Using Extended Connectivity fingerprint with Laplasian Modified Bayesian analysis in high-throughput-screening follow-up.* J. Biomol. Screen, 10(7):628–686, 2005.

[56] Rossi, Francesca, Peter van Beek und Toby Walsh: *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., New York, NY, USA, 2006.

[57] Sayle, Roger: *Improved SMILES Substructure Searching.*

[58] Schuffenhauer, A., P. Ertl, S. Roggo, S. Wetzel, M. A. Koch und H. Waldmann: *The Scaffold Tree - Visualization of the Scaffold Universe by Hierarchical Scaffold Classification.* J. Chem. Inf. Model., 47(1):47–58, January 2007.

[59] Shasha, Dennis, Jason und Rosalba Giugno: *Algorithmics and Applications of Tree and Graph Searching.* In: *Symposium on Principles of Database Systems*, Seiten 39–52, 2002.

[60] Steinbeck, C., Y. Han, S. Kuhn, O. Horlacher, E. Luttmann und E. Willighagen: *The Chemistry Development Kit (CDK): an open-source Java library for Chemo- and Bioinformatics.* J Chem Inf Comput Sci, 43(2):493–500, 2003.

[61] Steinbeck, C., C. Hoppe, S. Kuhn, M. Floris, R. Guha und E. L. Willighagen: *Recent developments of the chemistry development kit (CDK) - an open-source java library for chemo- and bioinformatics.* Current pharmaceutical design, 12(17):2111–2120, 2006.

[62] Sussenguth, Edward H.: *A Graph-Theoretic Algorithm for Matching Chemical Structures.* J. Chem. Doc., 5(1):36–43, 1965.

[63] Sykora, Vladimir J. und David E. Leahy: *Chemical Descriptors Library (CDL): A Generic, Open Source Software Library for Chemical Informatics.* J. Chem. Inf. Model., September 2008.

[64] Tonnelier, Christian, Philippe Jauffret, Thierry Hanser und Gérard Kaufmann: *Machine learning of generic reactions: 3. an efficient algorithm for maximal common substructure determination.* Tetrahedron Computer Methodology, 3, 1990.

[65] Ullmann, J. R.: *An Algorithm for Subgraph Isomorphism.* J. ACM, 23(1):31–42, 1976.

[66] Valiente, Gabriel: *Algorithms on Trees and Graphs.* Springer-Verlag, Berlin, 2002.

[67] Wetzel, Stefan: *Similarity in chemical and protein space: finding novel starting points for library design.* Doktorarbeit, Technische Universität Dortmund, 2009.

[68] Willett, Peter: *A review of chemical structure retrieval systems.* Journal of Chemometrics, 1(3):139–155, 1987.

[69] Willett, Peter: *Matching of Chemical and Biological Structures Using Subgraph and Maximal Common Subgraph Isomorphism Algorithms.* The IMA Volumes in Mathematics and its Applications, 108:11–38, 1999.

[70] Willett, Peter, John M. Barnard und Geoffrey M. Downs: *Chemical Similarity Searching.* Journal of Chemical Information and Computer Sciences, 38(6):983–996, November 1998.

[71] Williams, David W., Jun Huan und Wei Wang: *Graph Database Indexing Using Structured Graph Decomposition.* Data Engineering, International Conference on, 0:976–985, 2007.

[72] Yan, Xifeng, Philip S. Yu und Jiawei Han: *Graph indexing: a frequent structure-based approach.* In: *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, Seiten 335–346, New York, NY, USA, 2004. ACM.

[73] Yan, Xifeng, Philip S. Yu und Jiawei Han: *Graph indexing based on discriminative frequent structure analysis.* ACM Trans. Database Syst., 30(4):960–993, 2005.

[74] Zampelli, Stéphane: *A constraint programming approach to subgraph isomorphism.* Doktorarbeit, Universite catholique de Louvain, 2008.

[75] Zhang, Shijie, Meng Hu und Jiong Yang: *TreePi: A Novel Graph Indexing Method.* In: *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, Seiten 966–975, 2007.

[76] Zobel, Justin, Alistair Moffat und Kotagiri Ramamohanarao: *Inverted files versus signature files for text indexing.* ACM Trans. Database Syst., 23(4):453–490, 1998.

[77] Zou, Lei, Lei Chen, Jeffrey Xu Yu und Yansheng Lu: *A novel spectral coding in a large graph database*. In: *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, Seiten 181–192, New York, NY, USA, 2008. ACM.

[78] Zuendorf, Albert: *A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES*. Technischer Bericht AIB-05-1993, RWTH Aachen, 1993.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 18. Dezember 2009

Nils Kriege