Graph-Rekonstruktion im Rahmen chemischer Strukturrepräsentationen

Diplomarbeit

am Lehrstuhl für Informatik 1
der Bayerischen Julius-Maximilians-Universität Würzburg

Le Thuy Bui Thi

Betreuer:
Prof. Dr. Hartmut Noltemeier
Dr. Marc Zimmermann

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Wie jede andere naturwissenschaftliche Disziplin ist die Chemie stark auf experimentelle Beobachtungen und damit auf *Daten* angewiesen. Bis vor einigen Jahren waren Publikationen in Fachzeitschriften oder Buchform der normale Weg, um Neuentwicklungen in der Wissenschaft zu veröffentlichen. Mit dem immensen Anstieg bekannter chemischer Strukturen und der daraus resultierenden Informationsflut wurde die Einrichtung zentraler Datenbanken als Haupt-Informationsquelle unumgänglich.

Die zwei wichtigsten Chemie-Datenbanken der Welt sind die Beilstein- und CAS-Datenbanken.

Die Beilstein-Datenbank vom *Beilstein-Institut* mit Sitz in Frankfurt am Main – im Bereich der organischen Chemie die älteste Datenbank – ist eine der größten Faktendatenbanken der Welt. Sie umfaßt die wissenschaftliche Literatur von 1771 bis zur Gegenwart und enthält über 9,3 Millionen Strukturen mit über 35 Millionen Faktendatensätzen, einschließlich ihrer Originalzitate. Erfaßt werden chemische, physikalische, pharmakologische und physiologische Eigenschaften als numerische Werte, Stichworte oder als Texteinträge. Der Datenbankinhalt beschreibt Synthese und chemisches Verhalten der Verbindungen. Dieses Wissen ermöglicht Chemikern, erfolgreiche Wege zur optimalen Synthese bekannter Verbindungen zu finden und anhand von Analogien neue Verbindungen herzustellen [30].

Die CAS Datenbank von der *American Chemical Society* mit Sitz im Columbus, Ohio, ist eine weitere große Datenbank im Bereich der Chemie. Seit 1907 hat CAS Artikel von mehr als 40.000 wissenschaftlichen Zeitschriften indiziert und zusammengefaßt, zusätzlich zu Patenten, Konferenzberichten und anderen Veröffentlichungen aus dem Bereich der Chemie, der Life Sciences und angrenzender Gebiete. Insgesamt sind Informationen zu mehr als 24 Millionen chemischen Verbindungen via CAS online verfügbar [56]. Als zentrale Anlaufstelle zum Zugriff auf diese und ähnliche Datenbanken

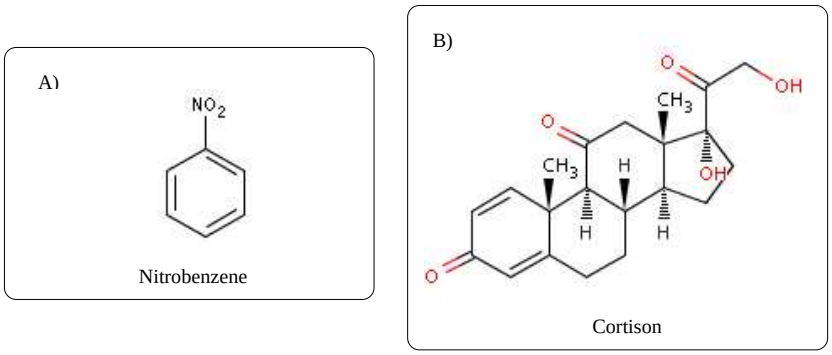


Abbildung 1.1: Zwei der verwendeten Teststrukturen für das System KEKULÉ, wie in [23] angegeben.

hat sich SCIFINDER (www.cas.org/SCIFINDER/SCHOLAR) etabliert.

Im Gegensatz zu manch anderem Fachgebiet steht die Chemie bei der Einrichtung solcher Datenbanken jedoch vor einer besonderen Aufgabe. Die relevante Information liegt oft nicht in Form von Zahlen, Tabellen oder Text vor, sondern in Form von graphischen Darstellungen. Diese graphischen Darstellungen sind die universale Sprache zur Beschreibung von Molekülen in der Chemie weltweit.

Ein Molekül besteht aus Atomen, die durch Bindungen verknüpft sind. Anzahl und Art der Bindungen sind charakteristische Atomeigenschaften. Die Kenntnis der Molekülstruktur und ihrer Zusammenhänge mit bestimmten physikalischen Eigenschaften sowie biologischen Aktivitäten ist von entscheidender Bedeutung in der Chemie. Mit Hilfe dieses Wissens lassen sich Stoffe mit bestimmten Eigenschaften gezielt synthetisieren.

Wie bereits angedeutet, haben Wissenschaftler bereits zahlreiche dieser Strukturen erforscht und beschrieben. Das Problem besteht nun darin, dass die Molekülstrukturen meist in einem Dateiformat abgelegt sind, welches ausschließlich die graphischen Eigenschaften speichert. Das heißt, in einer solchen Datei steht, *wie* das Molekül dargestellt werden soll, nicht aber *was* es darstellt. Damit wird ein Durchsuchen der Datenbanken de facto unmöglich. Um dieses Problem zu lösen, müssen die Graphiken in ein Format übersetzt werden, dass auch von Maschinen verstanden werden kann. Da eine manuelle Umwandlung der eingescannten oder mittels unterschiedlicher Zeichenprogramme erstellter Graphiken aus Kosten- und Zeitgründen nicht praktikabel erscheint, drängt sich eine maschinelle oder zumindest maschinen-unterstützte Umwandlung auf.

Obwohl dieses Problem nicht neu ist, wurde es, soweit uns bekannt, bisher noch wenig behandelt. J.R. McDaniel et al. haben Anfang der 90er Jahre in [41] und [42] von einem interaktiven System – KEKULÉ – berichtet. Aus [23] haben wir entnommen, dass KEKULÉ für kleine und einfache Strukturen

wie in Abbildung 1.1 A) rechte gute Ergebnisse liefert, die wenig Nachbearbeitungen vom Benutzer erfordern; bei komplexeren Strukturen Abbildung 1.1 B) und/oder mit niedriger Bildauflösung als 300 dpi war eine Rekonstruktion jedoch praktisch unmöglich. Außer den referenzierten Quellen waren leider keine weiteren Informationen zu KEKULÉ zu finden. Es ist uns insbesondere nicht bekannt, ob KEKULÉ zur Zeit überhaupt als Softwarepaket verfügbar ist. Ein weiteres interaktives System haben Ibison et al. in [26] vorgestellt. Das Projekt mit dem Namen CLiDE (Chemical Literature Data Extraction) wurde Anfang der 90er Jahre an der University of Leeds, GB, gestartet. Die entstandene Software wird nun von der Firma Symbiosys Inc. in Canada vertrieben [29].

In einer vom Fraunhofer Institut SCAI durchgeführten Evaluierung hat CLiDE jedoch nur in 50 % der Testfälle die chemischen Strukturen richtig erkannt. In den anderen Fällen war jeweils mindestens eine Nachkorrektur des Benutzers erforderlich [20]. Besondere Schwierigkeiten hat CLiDE damit, sogenannte Chiralbindungen zu erkennen; statt durch einfache Linien sind diese durch ausgefüllte oder gestrichelte Keile repräsentiert. Ebenso haben sich Atomsymbole wie etwa Fluor (F), Chlor (Cl) und Stickstoff (N) als problematisch erwiesen. Dies zeigt, dass das Problem, chemische Strukturen in Bilddokumenten zu erkennen und diese in ein maschinenlesbares Format umzuwandeln, nach wie vor besteht. Gleichzeitig war dies Anregung für ein neues Projekt am Fraunhofer Institut SCAI, das sich mit diesem Thema auseinander setzt. Die vorliegende Diplomarbeit ist innerhalb dieses Projekts entstanden.

1.2 Aufgabenstellung

Ziel dieser Diplomarbeit war die Ausarbeitung eines Konzepts für die Rekonstruktion chemischer Strukturen aus Bilddokumenten (Bitmap-Dateien) und ihre Abspeicherung in einem maschinenlesbaren Format (SDfile).

Dabei sollten folgende Punkte berücksichtigt werden:

1. Ausarbeitung eines Konzepts zur Rekonstruktion chemischer Strukturen aus Bilddokumenten (Bitmap-Dateien).
2. Implementierung eines Prototyps des entwickelten Konzepts in der objekt-orientierten Programmiersprache JAVA.
3. Ausgabe einer Bewertung der rekonstruierten Struktur.
4. Erkennen und Lernen eines neuen Strukturfragmentes.
5. Integretation einer manuellen Nachbearbeitung durch den Benutzer.
6. Automatische Abspeicherung der rekonstruierten Struktur im SDfile-Format.

1.3 Überblick

Die Erkennung von Symbolen in Bilddateien ist ein Thema, mit dem sich verschiedene Gruppen im Bereich der Mustererkennung und Dokumentenanalyse seit mehreren Jahren intensiv beschäftigt haben. Dabei wurde anfänglich vor allem die Mustererkennung in Bilddokumenten aus Bereichen wie Architektur [1, 2], Elektrotechnik [51], Maschinenbau, Kartografie, Musik usw. behandelt. Jedes dieser Fachgebiete hat seine eigene Symbolsprache erfunden; ein Symbol ist dabei jeweils eine anwendungsspezifische graphische Einheit. Dementsprechend gibt es fachspezifische Varianten der Mustererkennungsverfahren. Einen guten Überblick über die häufig angewandten Verfahren zur Symbolerkennung in den verschiedenen Fachgebieten geben [14, 35].

Noch heute gibt es große Mengen graphischer Dokumente, die nicht in maschinenlesbarem Format vorliegen. Abgesehen von der leichteren Archivierung sind es vor allem die Weiterverarbeitung und die automatisierte Informationsgewinnung, die eine Umwandlung in ein maschinenlesbares Format erforderlich machen. Anstelle einer Beschreibung der einzelnen Bildpunkte soll die Datei die Symbole auf einem höheren Abstraktionsniveau enthalten, beispielsweise durch explizite Benennung der Linien. Anstatt nur zu beschreiben *wie* das Symbol dargestellt wird, soll beschrieben werden *was* das Symbol darstellt. Diese Aufgabe zerfällt in zwei Schritte. Zum einen müssen die graphischen Einheiten in der Bilddatei identifiziert werden; zum anderen muss ein geeignetes Dateiformat gefunden werden, um die so erhaltene Information abspeichern zu können. Das Dateiformat sollte dabei die Weiterverarbeitung möglichst einfach machen. Für jeden einzelnen Teilschritt gibt es eine große Zahl von Vorschlägen und Verfahren. Aufgrund der Schwierigkeit eine allgemeine und robuste Methode zu finden, die sich in allen Gebieten gleichermaßen bewährt, müssen jedoch oft fachgebiets- oder problemspezifische Eigenschaften der Symbole ausgenutzt werden [35]. Ein Vergleich sowohl der allgemeinen als auch der fachspezifischen Verfahren wird auch dadurch erschwert, dass es keine Menge von Standard-Testdaten gibt. Dieses Manko wurden von verschieden Autoren erkannt und in den jüngsten Jahren hat es vermehrt Bemühungen gegeben, dieses Defizit zu beheben [63, 64].

Methodisch lassen sich die Verfahren zur Symbolerkennung in zwei Hauptrichtungen unterscheiden:

- Im *statistischen* Verfahren wählt man eine Anzahl n von Merkmalen, anhand derer die Symbole klassifiziert werden. Jedem Symbol wird dann ein n-dimensionaler Merkmalsvektor zugeordnet. Außerdem wird der gesamte n-dimensionale Vektorraum mittels einer geeigneten Funktion partitioniert. Dies soll so geschehen, dass *ähnliche* Muster jeweils in der gleichen Partition des Vektorraums liegen. Offensichtlich ist bei der statistischen Klassifikation die Wahl der geeigneten

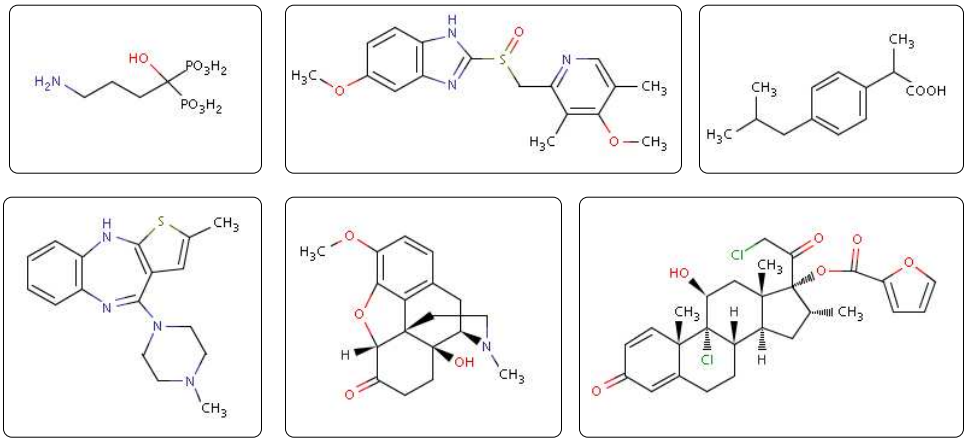


Abbildung 1.2: Strukturdiagramme verschiedener chemischer Verbindungen.

Merkmale ebenso wie das Auffinden einer geeigneten Funktion zur Merkmalsraumpartitionierung entscheidend. Der Hauptvorteil dieses Verfahrens liegt in der niedrigen Zeitkomplexität. Ein großer Nachteil besteht jedoch darin, dass die Relationen zwischen den Merkmalen vernachlässigt werden, wie zum Beispiel die Distanz zweier Objekte zueinander [31].

- Bei der *strukturellen* Symbolerkennung werden symbolische Datenstrukturen wie Strings, Bäume und Graphen anstatt numerischer Vektoren zur Repräsentation von Objekten eingesetzt. Symbolische Datenstrukturen erlauben eine explizite Beschreibung von Relationen zwischen Objekten. Diese Relationen können unterschiedlicher Natur sein, z.B. geometrischer, räumlicher usw. Weiterhin ermöglichen es diese Datenstrukturen, den hierarchischen Aufbau von Objekten aus vielen einfachen Teilstrukturen zu beschreiben. Die Erkennung eines unbekannten Objekts wird in diesem Verfahren durch einen Vergleich dessen symbolischer Repräsentation mit einer Anzahl vordefinierter Modell-Objekte realisiert. Das Verfahren basiert also auf dem *Matching*-Verfahren [7].

Während die Symbolerkennung für technische Dokumente bereits sehr gründlich diskutiert wurde, ist die automatische Erkennung chemischer Strukturen erst seit Anfang der neunziger Jahre vermehrt behandelt worden.

Wie sehen nun die Zeichnungen chemischer Strukturen aus? Was für Symbole gilt es zu erkennen? Welches Dateiformat ist die geeignete Speicherungsform?

Abbildung 1.2 zeigt Strukturdiagramme einiger chemischer Verbindungen von Medikamenten. In diesen Diagrammen werden Atome abgekürzt

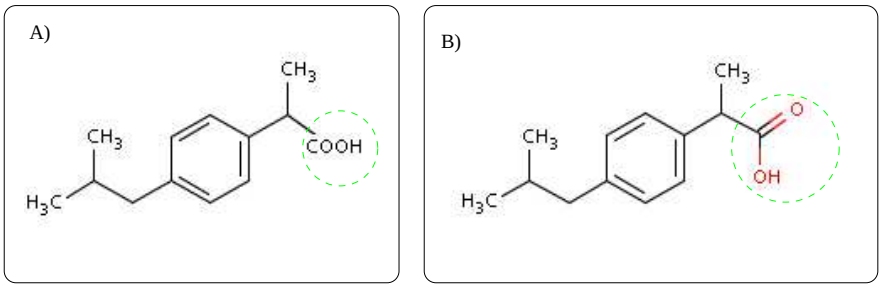


Abbildung 1.3: Die Carboxylatgruppe kann kontrahiert (A) oder expandiert (B) gezeichnet werden.

durch Buchstaben und Bindungen in der Regel durch geraden Linien dargestellt. Neben einzelnen Atomen, die durch einen großen oder einen großen und einen kleinen Buchstabe symbolisiert werden, können auch Molekülgruppen (Superatome) vorkommen; diese sind Zeichenketten wie z. B. $(-COOH)$ oder $(-PO_3H_2)$ und bezeichnen jeweils eine häufig auftretende Kombination wie etwa die Carboxylat- oder die Dihydrogenphosphitgruppe. Bei den Bindungen treten die durch zwei paralle Liniensegmente repräsentierte Doppelbindung und die Einfachbindung am häufigsten auf. Dreifachbindungen sind hingegen seltener vorhanden.

Des Weiteren werden in Strukturdiagrammen organischer Verbindungen oft noch ausgefüllte oder gestrichelte Keile eingesetzt, um die räumliche Anordnung der Atome im Molekül anzudeuten. Moleküle, die so gebaut sind, dass sie nicht mit dem Molekül, das durch ihr 2D-Spiegelbild beschrieben wird, zur Deckung gebracht werden können, nennt man chiral. Bei diesen werden werden die genannten Keile eingesetzt. Die elementaren Symbole in chemischen Strukturdiagrammen sind also Atomsymbole und verschiedene Bindungssymbole. Diese können zu größeren Symbolen wie etwa zyklischen Einheiten kombiniert werden. Die vollständige Rekonstruktion chemischer Strukturen aus Bilddarstellungen umfasst somit folgenden Aufgaben:

- Erkennung von Atomsymbolen und Zahlen (OCR); bei Superatomen ist außerdem ein Parsen in ihre elementaren Atome sowie eine Intepretation der implizit enthaltenen Bindungen erforderlich, Abbildung 1.3 gibt einen Beispiel an der Carboxylatgruppe $-COOH$;
- Erkennung von chemischen Bindungen (einfache gerade Linien, Parallelen und Keile);
- Zuordung von Atomen und Bindungen;
- Speicherung der rekonstruierten Struktur in einem geeigneten Format.

Wir wollen nun einen kurzen Überblick über die Ansätze von Kekulé und CLiDE geben.

Kekulé

Kekulé nimmt als Eingabe Rasterbilder chemischer Strukturen im TIFF-Format an und erzeugt eine Graphrepräsentation der entsprechenden Strukturen, welche dann in strukturierten Stringrepräsentationsformen wie ISIS, MOLfile, SMILES oder ROSDAL gespeichert werden kann. Alternativ unterstützt das Programm auch das Einscannen der Eingabe aus Papierzeichnungen. Die Verarbeitungschritte dieses Systems lassen sich in folgenden Schritten unterteilen:

- Lesen der Eingabe
- Vektorisierung des Eingabe-Bildes
- Suche nach Chiralbindungen
- Zeichenerkennung
- Graphgenerierung
- Nachbearbeitung
- Anzeige und Editieren

Nach der Vektorisierung des Rasterbildes versucht Kekulé alle Vektoren, die Chiralbindungen darstellen könnten, zu erkennen. Anschließend werden neuronale Netze eingesetzt, um die Atomsymbole zu identifizieren. Zu ihrer Intepretation werden unter anderem chemische Regeln herangezogen. Im nächsten Schritt werden die übriggebliebenen Vektoren als Bindungen interpretiert. Aus diesen und den erkannten Atomen wird dann ein Graph bzw. eine Verknüpfungstafel (Definition siehe Abschnitt 2.3.3) für die zu erkennende chemische Struktur erzeugt. In der Nachbearbeitungsphase wird das graphische Bild normiert. Dabei werden zum Beispiel die Bindungslängen oder die Winkeln zwischen zweier Bindungen korrigiert. Schließlich kann der Benutzer die Ausgabe in einer Benutzeroberfläche, falls nötig, korrigieren.

CLiDE

CLiDE rekonstruiert die Bilddarstellung chemischer Strukturen in drei Schritten. Im erstem Schritt wird das Bild segmentiert. Dazu werden die Bildpunkte in schwarz und weiß unterteilt. Dann wird für jeden schwarzen Bildpunkt ein Knoten eingeführt. Kanten zwischen diesen Knoten bezeichnen

die Nachbarschaften zwischen den entsprechenden Bildpunkten. Der so entstandene Graph zerfällt in Zusammenhangskomponenten. Jede dieser Zusammenhangskomponenten repräsentiert eines oder mehrere Primitive (Zeichen, graphische Elemente oder kurze Striche), die CLiDE im diesen Schritt zu identifizieren versucht. Die Trennung dieser Primitive erfolgt nun über die relative Größe der Komponente, ihren Flächeninhalt und das Verhältnis, der sie einschließenden Rechteckseiten. Die Strichmuster von Chiralbindungen werden mittles Hough-Transformation ermittelt. Die Zeichenerkennung erfolgt durch Anwendung neuronaler Netze. Alle übriggeblieben graphischen Elemente werden später als chemische Bindungen interpretiert.

Im darauf folgenden Schritt werden die erkannten Buchstaben und Zahlen zu Textgruppen angeordnet. In der letzten Phase wird versucht die Primitive im chemischen Kontext zu intepretieren. Einzelne Atomsymbole werden identifiziert. Textgruppen von Superatomen werden über eine „look-up table“ intern aufgelöst. Nachdem auch die Bindungen festgestellt wurden, werden Atome und Bindungen miteinander assoziert. Nicht verbundene Bindungen werden miteinander verknüpft, falls die Distanz ihrer Endpunkte unterhalb eines Grenzwertes liegt. Die rekonstruierte Struktur kann in Formaten wie MOLfile und ChemDraw exportiert werden. Diese Formate enthalten die in chemischen Informationssystemen oft eingesetzte *Verknüpfungstafel*. In ihr werden alle Atome und die sie verbindenden Bindungen aufgelistet. Sie ist mit der Repräsentation von Graphen als Adjazenzlisten vergleichbar. Beide werden wir im nächsten Kapitel näher betrachten.

Der in dieser Arbeit vorgestellte Ansatz basiert auf struktureller Mustererkennung. Die Erkennung der unbekannten Struktur im Eingabe-Bild erfolgt über Graph-Matching-Verfahren. Das Eingabebild wird zuerst zu einem Graph umgewandelt. Durch Graph Matching wird dann versucht, diesen mittels Strukturfragementen aus einer Bibliothek zu rekonstruieren. Es wird dabei ermittelt, welche der Modell-Graphen im Eingabe-Graphen als Subgraphen enthalten sind. Aus diesen wird dann die unbekannte Struktur zusammengefügt.

Zyklische Strukturfragmente (einfache Ringe oder kondensierte Ringsysteme), die noch nicht in der Bibliothek enthalten sind, werden automatisch erkannt. Bei Zustimmung des Benutzers werden sie in die Bibliothek aufgenommen und stehen beim nächsten Matching ebenfalls als Modelle zur Verfügung. Die rekonstruierte Struktur wird automatisch im SDfile-Format abgelegt, und zusammen mit einer Bewertung der Rekonstruktion wird sie dem Benutzer in einer Benutzeroberfläche präsentiert. Er kann das Ergebnis dann gegebenfalls korrigieren, weiterverarbeiten oder in einem anderen Format speichern. Stellen, an denen System Fehler vermutet, werden dabei gesondert markiert angezeigt.

Ein Prototypimplementierung des vorgestellten Ansatzes wurde in der Programmiersprache JAVA realisiert. Aus zeitlichen Gründen mußten wir

dabei leider auf die Implementierung eines OCR-Verfahrens verzichten. Es ist einstweilen Aufgabe des Benutzers, im Bild explizit angezeigte Atome oder Superatome zu markieren und sie anzugeben. Ebenso wurden in der aktuellen Implementierung Chiralbindungen noch nicht behandelt. Eine Erweiterung des entwickelten Systems um diese Funktionionalitäten ist für die Zukunft vorgesehen.

1.4 Aufbau des Dokumentes

Wir geben nun einen Überblick über den weiteren Aufbau des Dokumentes. In Kapitel zwei führen wir die theoretischen Grundlagen unserer Arbeit ein. In Kapitel drei präsentieren wir unser Konzept. Anschließend betrachten wir die experimentellen Ergebnisse unserer Arbeit und ihre Bewertung in Kapitel vier. Schließlich diskutieren wir in Kapitel fünf welche Perspektiven für eine Weiterentwicklung unserer Arbeit sich daraus ergeben. Der Anhang enthält Anmerkungen zur programmier-technischen Realisierung des Konzepts.

Kapitel 2

Grundlagen

Dieses Kapitel führt die graphentheoretischen Grundbegriffe ein, die in den folgenden Kapiteln benutzt werden. Es werden außerdem verschiedene Datenstrukturen sowie grundlegende Algorithmen auf Graphen vorgestellt. Im Anschluß daran werden wir die gebräuchlichsten Darstellungsformen chemischer Verbindungen betrachten.

2.1 Graphentheoretische Grundlagen

In diesem Abschnitt wird eine kurze Einführung in die Terminologie der Graphentheorie gegeben. Eine ausführlichere Einleitung in dieses Gebiet sowie vertiefendes Material findet sich zum Beispiel in [15, 16, 25, 32].

2.1.1 Grundbegriffe

Definition 2.1 ***(Ungerichteter Graph)***

Ein ***ungerichteter Graph*** *ist ein Tripel $G = (V, E, \gamma)$ aus einer nichtleeren Menge V, einer Menge E und einer Abbildung $\gamma : E \to \{X | X \subseteq V \text{ mit } 1 \leq |X| \leq 2\}$.*

Die Elemente aus V heißen die ***Ecken****, die Elemente aus E die* ***Kanten*** *des Graphen G. Für eine Kante e bezeichnet man die Elemente von $\gamma(e) = \{u, v\}$ als* ***Endpunkte*** *von e. Man sagt auch, u und v sind durch e verbunden. u und v sind dabei nicht notwendigerweise verschieden.*

Zwei Kanten e und f sind ***parallel****, wenn sie gleichen Endpunkte haben. Eine Kante e heißt* ***Schlinge****, falls ihre Endpunkte identisch sind.*

Enthält G keine parallelen Kanten und Schlingen, so heißt G ***einfach****. Man schreibt dann auch $G = (V, E)$, wobei $E \subseteq V \times V$ gilt.*

Im Folgenden werden wir nur einfache ungerichtete Graphen betrachten. Für eine Kante e werden wir anstelle von $\{u, v\}$ auch (u, v) schreiben.

Falls von mehreren Graphen gesprochen wird, schreiben wir für eine Ecken-/Kantenmenge eines Graphen G auch $V(G)/E(G)$, falls nicht schon aus dem Kontext klar ist, welche Menge gerade gemeint ist.

Definition 2.2 *Sei $G = (V, E)$ ein Graph. Zwei Ecken u und v von G heißen **adjazent** oder benachbart, wenn es eine Kante $e = \{u, v\} \in E$ gibt.*

*Eine Ecke u heißt mit einer Kante e **inzident**, wenn $u \in e$ gilt.*

Zwei Kanten $e \neq f$ sind inzident, wenn sie eine gemeinsame Endecke haben.

Sind je zwei Ecken von G benachbart, so heißt G vollständig. Ein vollständiger Graph auf n Ecken wird mit K_n bezeichnet.

Definition 2.3 *(**Teilgraph, Obergraph**)*

*Ein Graph $G' = (V', E')$ heißt **Teilgraph** eines Graphen $G = (V, E)$, in Zeichen $G' \subseteq G$, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt. G wird dann als **Obergraph** von G' bezeichnet. Informell sagen wir häufig, dass G den Graphen G' enthält.*

Definition 2.4 *(**Subgraph**)*

*Sei G ein Graph $G = (V, E)$. Ein Teilgraph $S = (V_S, E_S)$ von G heißt **induziert** (von V_S in G), wenn*

1. $V_S \subseteq V$
2. $E_S = E \cap (V_S \times V_S)$.

*Der induzierte Teilgraph S heißt dann **Subgraph** von G und wird auch mit $G[V_S]$ bezeichnet.*

Falls S ein Subgraph von G ist, bezeichnen wir mit $G - S$ denjenigen Subgraph von G, der aus der Eckenmenge $V \backslash \{V_S\}$ und allen Kanten von G besteht, die nur mit Ecken aus der Differenzmenge $V \backslash \{V_S\}$ inzident sind. Ist $V_S = \{v\}$ einelementig, so setzen wir $G - \{v\} = G - v$.

Definition 2.5 *(**Kantengraph**)*

*Sei G ein Graph $G = (V, E)$. Ein **Kantengraph** von G ist ein Graph $L(G) = (V_{L(G)}, E_{L(G)})$ mit folgenden Eigenschaften*

1. $V_{L(G)} = E$
2. $E_{L(G)} = \{(e, f) | e, f \in E \wedge e \neq f$ *ist inzident in* $G\}$

Im Kantengraphen $L(G)$ eines Graphen G wird also jede Kante aus G als eine Ecke repräsentiert. Zwei Ecken e und f aus $L(G)$ sind genau dann benachbart, wenn sie als Kanten im Graph G inzident sind.

Definition 2.6 *(**Ungerichteter attributierter Graph**)*
*Ein **ungerichteter attributierter Graph** G ist ein Quadrupel $G = (V, E, \mu, \nu)$ mit folgenden Eigenschaften:*

1. *V ist eine nicht leere Eckenmenge*
2. *$E \subseteq V \times V$ ist eine Kantemenge*
3. *$\mu : V \to L_V$ ist eine Funktion, die jeder Ecke aus V einem Attribut l zuordnet.*
4. *$\nu : E \to L_E$ ist eine Funktion, die die Kanten in G auszeichnet.*

Definition 2.7 *(**Subgraph eines attributierten Graphen**)*
*Sei G ein Graph $G = (V, E, \mu, \nu)$. Ein **Subgraph** von G ist ein Graph $S = (V_S, E_S, \mu_S, \nu_S)$ mit folgenden Eigenschaften:*

1. *$V_S \subseteq V$*
2. *$E_S = E \cap (V_S \times V_S)$.*
3. *μ_S und ν_S sind Einschränkungen von μ und ν.*

2.1.2 Wege, Kreise und Zusammenhang

Sei im Folgenden $G = (V, E)$ ein ungerichteter Graph.

Definition 2.8 *(**Weg, Kreis**)*
*Eine endliche Kantenfolge $w = \{e_1, e_2, \ldots, e_k\}$, $e_i \in E$, heißt ein **Weg** von v_0 nach v_k im G, wenn es $v_0, v_1, \ldots, v_k \in V$ gibt, so dass $e_i = (v_{i-1}, v_i)$ für $i = 1, 2, \ldots, k$ gilt. Die **Länge** $|w|$ des Weges ist die Kantenzahl k der Kantenfolge w.*

*Sind die Anfangs- und Endecke v_0 bzw. v_k von w identisch und $|w| \geqslant 3$, so heißt w ein **Kreis**.*

*Ein Weg w heißt **einfach**, wenn alle Kanten $e_i \in w$ disjunkt sind. Er heißt **elementar**, falls - bis auf den Fall, dass es sich um einen Kreis handelt - jede Ecke nur einmal in w vorkommt.*

Definition 2.9 *(**Zusammenhang**)*
*Zwei Ecken u und v sind **zusammenhängend** (in Zeichen $u \sim v$), wenn es einen Weg zwischen u und v gibt.*

*Der Graph G ist zusammenhängend, falls für je zwei Ecken u und v in G gilt, $u \sim v$. Ein maximal zusammenhängender Teilgraph von G wird als **Zusammenhangskomponente** bezeichnet.*

Die Relation $\sim$ ist eine *Äquivalenzrelation* auf V. Die Zusammenhangskomponenten von G sind die Äquivalenzklassen bezüglich $\sim$ und bilden eine *Partition* der Eckenmenge V (Beweis siehe zum Beispiel [32]).

Definition 2.10 *(**Artikulationspunkt, Brücke**)*

*Eine Ecke v eines Graphen G heißt **Artikulationspunkt** oder **trennende Ecke**, wenn durch Löschen dieser Ecke die Anzahl der Zusammenhangskomponenten von G erhöht wird. Eine solche Kante wird als **Brücke** bezeichnet.*

Eine Kante e eines Graphen G ist genau dann eine Brücke, wenn sie nicht auf einem Kreis von G liegt. (Beweis siehe [25, 66])

Definition 2.11 *(**2-fach zusammenhängende Komponente**)*

*Eine **2-fach zusammenhängende Komponente** eines Graphen G ist eine maximale Kantenmenge $E' \subseteq E$ mit der Eigenschaft, dass es für je zwei Kanten $e, f \in E'$ gilt, e und f liegen auf einem gemeinsamen einfachen Kreis in G.*

G heißt 2-fach zusammenhängend, wenn G nur eine 2-fach zusammenhängende Komponente hat.

Somit ist jede Kantenmenge $E' \subseteq E$ genau dann eine 2-fach zusammenhängende Komponente, wenn E' keine Brücke enthält.

Des Weiteren gelten folgende Eigenschaften für 2-fach zusammenhängende Komponenten :

- Jede Kante in G, die keine Brücke ist, liegt genau in einer 2-fach zusammenhängenden Komponente von G, ebenso wie jede Ecke, die weder isoliert noch eine trennende Ecke ist.

- Zwei 2-fach zusammenängende Komponenten von G haben höchstens eine gemeinsame Ecke; diese ist dann notwendigerweise eine trennende Ecke.

- Die trennende Ecken von G trennen die 2-fach zusammenhängende Komponenten von G. Falls G keine trennende Ecke besitzt, so ist G 2-fach zusammenhängend.

- Die Kanten irgendeines Kreises von G liegen alle in derselben 2-fach zusammenhängenden Komponente von G.

2.1.3 Repräsentation von Graphen

Hier stellen wir ein paar allgemein gebräuchliche Verfahren vor, Graphen als Dateien abzuspeichern. Außerdem erwähnen wir ihre Vor- und Nachteile. Bezeichne im Folgenden $G = (V, E)$ einen Graph mit Eckenmenge $V = \{v_1, v_2, \ldots, v_n\}$ und Kantenmenge $E = \{e_1, e_2, \ldots, e_m\}$.

Adjazenzmatrix-Repräsentation

Definition 2.12 ***(Adjazenzmatrix)***
Die ***Adjazenzmatrix*** *von G ist eine $n \times n$-Matrix $A(G) = (a_{ij})$ mit*

$$a_{ij} = \begin{cases} 1 & \textit{falls } e = (v_i, v_j) \in E, \\ 0 & \textit{sonst.} \end{cases}$$

Die Adjazenzmatrix A für einen Graphen G mit Eckenmenge V benötigt genau n^2 Speicherplatz. Damit ist diese Darstellung besonders gut für dichte Graphen geeignet; bei dünnen Graphen entstehen sehr viele Nulleinträge. Dabei bezeichnen wir einen Graphen als *dicht*, wenn er vollständig oder fast vollständig ist.

Inzidenzmatrix-Repräsentation

Definition 2.13 ***(Inzidenzmatrix)***
Die ***Inzidenzmatrix*** *von G ist eine $n \times m$-Matrix $I(G) = (a_{ij})$ mit*

$$a_{ij} = \begin{cases} 1 & \textit{falls } v_i \in e_j, \\ 0 & \textit{sonst.} \end{cases}$$

Die Inzidenzmatrix braucht $\Theta(n \cdot n)$ Speicherplatz.

Adjazenzlisten-Repräsentation

Die **Adjazenzlistendarstellung** von G besteht aus einem Array von n Listen, für jede Ecke aus V eine. Für jede Ecke $u \in V$ gibt es eine Liste *Adj[u]*, die alle zu u adjazenten Ecken enthält.

Der Speicheraufwand für die Adjazenzlistendarstellung beträgt genau $(n + m)$. Sie ist gut für dünne Graphen geeignet, d.h. wenn $m \ll \frac{1}{2}n(n+1)$ für einen ungerichteten Graph gilt.

2.1.4 Vergleichskriterien für Algorithmen

Unter einem *Algorithmus* verstehen wir eine Verarbeitungsvorschrift zur Lösung eines Problems. Dabei besteht das Problem darin, zu einer Menge von Eingabe-Werten (auch einfach als *Eingabe* bezeichnet) die zugehörigen Ausgabe-Werte (*Ausgabe*) zu finden.

Zur Lösung eines Problems gibt es oft verschiedene Algorithmen. Um diese charakterisieren und sie untereinander vergleichen zu können, braucht man ein Maß für die Effizienz eines Algorithmus. Dieses wird durch die *Zeit-* und *Raumkomplexität* des Algorithmus gegeben. Erstere beschreibt, wieviel Zeit zum Ausführen des entsprechenden Algorithmus benötigt wird. Letztere

gibt an, wie der Algorithmus sich hinsichtlich des Speicherplatzverbrauchs verhält.

Damit man unabhängig von der Austattung eines bestimmten Computers einen Vergleich verschiedener Algorithmen anstellen kann, werden in der Zeitkomplexität keine konkreten Zeitangaben gemacht, sondern es wird nur die Anzahl der benötigten Rechenschritte betrachtet. Unter einem Rechenschritt verstehen wir Grundoperationen wie arithmetische Operationen, Operationen zum Datentransfer (Laden, Speichern, Kopieren) sowie Vergleiche und Verzweigungen. Es wird dabei vereinfachend angenommen, dass diese Operationen auf einer gegebenen Rechenanlage jeweils konstante Zeit brauchen (*Unit-Cost RAM Modell*).

Da die Komplexität eines Algorithmus von der Größe der Eingabe abhängt, wird sie als Funktion eben dieser Eingabegröße angegeben. Dabei werden multiplikative und additive Konstanten außer Acht gelassen, weil diese bei großen Eingaben wenig Einfluss auf die jeweilige Komplexität haben. Man interessiert sich also lediglich für die *Größenordnung* der Komplexitätsfunktionen.

Bevor wir die Größenordnung von Funktionen genauer betrachten, ist es noch wichtig zu erwähnen, dass ein sinnvolles Maß für die Eingabegröße eines Algorithmus problemabhängig ist. Für manche Probleme, zum Beispiel die Sortierung eines Arrays der Länge n, ist das natürlichste Maß der Eingabegröße die Anzahl der Elemente in der Eingabe. Für viele andere Probleme, zum Beispiel die Multiplikation zweier ganzen Zahlen, ist es die Anzahl von Bits, die zur binären Darstellung der Zahlen benötigt wird. Bei Graphalgorithmen wird die Eingabegröße durch die Eckenzahl n, die Kantenzahl m oder durch die Kombination beider angegeben.

Größenordnung von Funktionen

Sei M die Menge aller reellwertigen Funktionen $f : \mathbb{N} \to \mathbb{R}$ auf den natürlichen Zahlen. Jede Funktion $g \in M$ wird damit einer der folgenden Klassen zugeordnet:

- $O(g) := \{f \in M \mid \exists c,\ n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$
- $\Omega(g) := \{f \in M \mid \exists c,\ n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$
- $\Theta(g) := O(g) \cap \Omega(g)$

Für einen Algorithmus unterscheidet man außerdem drei unterschiedliche Komplexitätsfunktionen:

- Die *worst-case Komplexität* wird durch eine Funktion von n beschrieben, die die obere Schranke für die Anzahl der benötigten Rechenschritte für alle Probleminstanzen der Größe n darstellt.

- Die *best-case Komplexität* wird durch jene Funktion von n angegeben, welche das Minimum der benötigten Rechenschritte für alle Probleminstanzen der Größe n liefert.
- Schließlich wird die *average-case Komplexität* durch jene Funktion von n beschrieben, welche das statistische Mittel der Rechenschritte eines Algorithmus für beliebige Probleminstanzen der Größe n liefert.

P und NP

Eine Funktion f wird als *von polynomieller Größenordnung* oder einfach als *polynomiell* bezeichnet, wenn es ein Polynom g gibt, so dass $f \in O(g)$ gilt. Wir schreiben auch $f \in P$. Der Begriff der polynomiellen Laufzeit ist in der Informatik von großer praktischer Bedeutung. Wenn ein Algorithmus unter dem zuvor vorgestellten Berechnungsmodell polynomielle Laufzeit hat, so darf man hoffen, mit ihm alle praktisch relevanten Eingaben verarbeiten zu können.

Es gibt jedoch noch ein anderes bedeutendes Rechenmodell, das Nichtdeterministische. Da der Platz hier für eine formale Einführung nicht ausreicht, werden wir uns auf ein paar informelle Bemerkungen beschränken. Der interessierte Leser sei auf [52] verwiesen.

Dieses Berechnungsmodell wird meistens auf *Turingmaschinen* eingeführt. Turingmaschinen sind von der Berechnungsmächtigkeit dem RAM-Modell ebenbürtig; das RAM-Modell ist jedoch enger an die Architektur tatsächlicher Rechner angelehnt. Im nicht-deterministischen Modell kann von einem Zustand der Berechnung zu einem beliebigen aus einer Menge von Folgezuständen übergegangen werden. Unter diesem Berechnungsmodell gilt ein Algorithmus als polynomiell, wenn es mindestens einen Berechnungspfad gibt, der in polynomieller Laufzeit zum Ergebnis führt. Auf diesem Pfad *rät* die Maschine in jedem Zustand den jeweils richtigen unter den erlaubten Folgezuständen. Ein Algorithmus der unter diesem Maschinenmodell polynomielle Laufzeit hat, gehört der Klasse *NP* an; *NP* steht für nichtdeterministisch polynomiell. Ob P und NP die gleichen oder verschiedene Klassen bezeichnen, ist ein großes ungelöstes Problem der theoretischen Informatik. Es wird jedoch allgemein angenommen, dass sie verschieden sind. So kennt man zum gegenwärtigen Zeitpunkt viele Probleme, für die Algorithmen mit polynomieller Laufzeit nur für nicht-deterministische Maschinenmodelle bekannt sind. Wenn man diese Algorithmen auf real existierender Hardware laufen lässt, so muss man alle möglichen Berechnungspfade sequentiell erproben. Damit sind sie im allgemeinen nicht effizient.

2.1.5 Durchsuchen von Graphen

Viele graphentheoretische Algorithmen erfordern das vollständige Durchsuchen aller Ecken oder Kanten eines Graphen. Viele Informationen über die

Struktur des Graphen können auf diese Weise gewonnen werden, zum Beispiel ob der Graph kreisfrei ist oder wieviele Zusammenhangskomponenten er hat. Zwei häufig verwendete Suchverfahren für Graphen sind die *Tiefen-* (Depth-first search, DFS) und *Breitensuche* (Breadth-first search, BFS). Wir möchten im Folgenden das Arbeitsprinzip der Tiefensuche für ungerichteten Graphen vorstellen.

Die Suche beginnt mit einer Startecke s und bewegt sich entlang der Kanten des Graphen, um alle von s aus erreichbaren Ecken zu besuchen. Dabei wird jede Ecke, sobald sie besucht wird, als *besucht/discovered* markiert. Nun wird für die jeweils aktuelle Ecke v_i ein beliebiger, noch nicht besuchter Nachbar v_j gewählt. Dieser wird dann wiederum als *besucht/discovered* markiert; außerdem merken wir uns v_i als Vorgänger. Da jede Ecke höchstens einmal entdeckt wird, gibt es für sie höchstens einen Vorgänger. Diese werden in einem Array *pred* gespeichert. $pred[v_j] = v_i$ bedeutet also, dass v_i der Vorgänger von v_j ist.

Für v_j gehen wir wiederum genauso vor. Wenn es für eine Ecke nun keinen noch nicht besuchten Nachbarn gibt, so markieren wir diese Ecke als *bearbeitet/finished*. Dann bewegen wir uns entlang der gespeicherten Vorgänger solange im Graph wieder aufwärts, bis wir eine Ecke mit einem noch nicht besuchten Nachbarn v_k finden. Für diese Nachbarecke v_k verfahren wir nun wieder wie gehabt. Wir fahren solange fort, bis alle Ecken als *bearbeitet/finished* markiert sind. Falls somit alle von s aus erreichbaren Ecken bereits als *bearbeitet/finished* markiert sind, und im Graphen noch unmarkierte Ecken übrig sind, wird irgendeine dieser Ecken als neue Startecke ausgewählt und die Suche läuft weiter wie beschrieben, bis alle Ecken durchsucht sind.

Wir bemerken, dass die Markierungen nötig sind um bei einem zyklischen Graphen nicht in eine unendliche Schleife zu geraten.

Wenn wir nun bei DFS beim ersten Entdecken einer Ecke jeweils noch eine Zeitmarke speichern, so erhalten wir durch einen Lauf von DFS eine Ordnung auf den Ecken des Graphen.

Durch DFS können viele Informationen über die Struktur eines Graphen G gewonnen werden. Eine grundlegende Eigenschaft von DFS ist, dass während dieser Suche ein Vorgängergraph $G_{pred} = (V, E_{pred})$ mit

$$E_{pred} = \{(pred[v], v) | v \in V \wedge pred[v] \neq NIL\}$$

konstruiert wird. Dieser bildet einen DFS-Wald, der aus mehreren DFS-Wurzelbäumen besteht. Die Wurzeln dieser DFS-Bäume sind die jeweiligen Startecken s. Bei ungerichteten Graphen stellen die Ecken eines solchen Wurzelbaumes eine Zusammenhangskomponente dar. Die Kanten des Graphen G werden durch DFS in zwei Arten klassifiziert:

- Zu den *Baumkanten* (tree edges) gehören alle Kanten der Menge E_{pred}.

- Als *Rückwärtskanten* (back edges) werden diejenigen Kanten (u, v) bezeichnet, die die Ecke u mit einem Vorgänger v in einem DFS-Wurzelbaum verbindet.

Falls die Adjazenlisten eines Graphen $G = (V, E)$ mit $|V| = n$, $|E| = m$ gegeben sind, kann die Tiefensuche auf G in $O(n + m)$ Zeit durchgeführt werden [15, 32]. In dieser Arbeit haben wir DFS eingesetzt, um alle 2-fach zusammenhängende Komponenten eines Graphen zu bestimmen, siehe Kapitel 3.5. Der Pseudocode des DFS ist dort gegeben.

Der Name Tiefensuche erklärt sich daraus, dass von einer Ecke aus jeweils entlang eines Nachbarn versucht wird, einen möglichst langen Pfad in den Graph hinein zu finden. Bei Breitensuche werden im Gegensatz dazu für jede Ecke v zuerst alle Nachbarn betrachtet, bevor wiederum deren Nachbarn betrachtet werden. Da es damit notwendig wird, deutlich größere Teile des Graphen im Speicher zu halten, wird womöglich der Tiefensuche der Vorzug gegeben.

2.2 Molekülstruktur

Die 2D-Darstellung chemischer Strukturen in Form von Strukturdiagrammen wird oft als die universelle Sprache der Chemiker betrachtet. Sie macht die Moleküle für uns besser vorstellbar und lässt erste Einblicke in deren chemischen Charakter zu. Unter einem Molekül versteht man dabei die kleinste Einheit einer chemischen Verbindung, die noch die für diese Verbindung typischen Eigenschaften aufweist.

Um die dreidimensionale Molekülstruktur gut im Zweidimensionalen darstellen zu können, beschränken sich die Strukturdiagramme auf das Wesentliche. Sie setzen sich aus Atomsymbolen und Linien zusammen; letztere spezifizieren dabei Bindungen zwischen den Atomen. In der organischen Chemie sind dabei Sauerstoff, Schwefel, Stickstoff, Phosphor, Chlor, Brom, Fluor und Iod neben dem Kohlenstoff die am häufigsten auftretende Atome. Bei den Bindungen handelt es sich überwiegend um *kovalente Bindungen*. Diese chemische Bindungsart kommt durch die räumliche Überlappung von jeweils zwei Elektronenorbitalen der Atome zustande. Das dabei entstandene Elektronenpaar gehört den beiden verbundenen Atomen gleichzeitig und vermittelt die Bindung. Dadurch wird eine abgeschlossene stabile Edelgaskonfiguration in den Molekülen erreicht.

Die Anzahl der Atombindungen, die ein bestimmtes Atom ausbilden kann, wird seine *Bindigkeit* genannt. Sie hängt von der Elektronenkonfiguration ab; d.h. von der räumlichen Verteilung der Elektronen auf die einzelnen Orbitale der Atome. Bei Kohlenstoff beträgt sie beispielsweise 4.

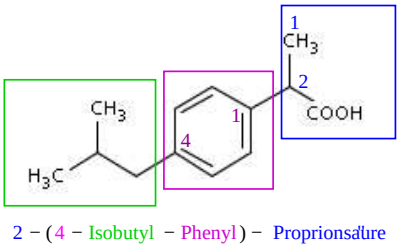


Abbildung 2.1: IUPAC-Name für Ibuprofen: 2-(4-Isobutyl-phenyl)-propionsäure.

2.3 Repräsentation chemischer Strukturen

In den Anfängen der organischen Chemie waren *Trivialnamen* das vorherrschende Mittel, um chemische Substanzen auseinander zu halten. Daher lassen sich in vielen Fällen die Namen der Strukturen auf ihre Herkunft oder Verwendung zurückführen. Beispiele hierfür sind *Penicillin* (Schimmelpilz: Penicillium notatum) oder *Cumarin* (Bohnenart, die von südamerikanischen Indianern cumaru genannt wird). Selbst in der heutigen Zeit wird diese Art der Namengebung noch angewendet, um Moleküle mit komplexen Strukturen einfach zu benennen. Als jedoch mit der Zeit die Anzahl der neuentdeckten Substanzen immer schneller zunahm, war es notwendig, chemische Verbindungen systematisch zu klassifizieren und für sie eine Nomenklatur zu entwickeln, die idealerweiser jeder Substanz einen eigenen Namen zuordnet. Ebenso war es wünschenswert, dass die Ableitung einer molekularen Struktur aus ihrem systematischen Namen und umgekehrt die Erzeugung eines solchen Namen aus der entsprechenden Struktur ermöglicht werden sollte. Von den entstandenen Nomenklatur-Systemen ist die IUPAC-Nomenklatur (**I**nternational **U**nion of **P**ure and **A**pplied **C**hemistry) die Wichtigste. Sie wurde 1922 eingeführt und basiert auf der 1892 entworfenen Genfer Nomenklatur. Noch bis heute wird dieses Nomenklatursystem ständig überarbeitet und aktualisiert [50, 55].

Das Grundprinzip der IUPAC-Nomenklatur besteht darin, dass zunächst ein *Grundgerüst* festgestellt wird. Die Atome des Grundgerüsts werden dann so durchnummeriert, dass die *Substituenten* (kleinere Seitenketten, die am Grundgerüst hängen) an Atome mit möglichst kleinen Nummern stehen. H-Atome werden bei der Nummerierung jedoch nicht berücksichtigt. Beim Name werden zuerst die Nummer der Verknüpfungsstellen der Substituenten, gefolgt von deren Namen und schließlich der Name des Grundgerüst angegeben. Abbildung 2.1 zeigt ein Beispiel der IUPAC-Nomenklatur anhand des Schmerzmittels Ibuprofen.

Dieses IUPAC-System weist allerdings Mankos auf: Zum einen ist die Nomenklatur nicht eineindeutig; d.h. mehrere gültige Namen können dersel-

ben Struktur zugeordnet werden. Außerdem kann ein IUPAC-Name unnötig lang sein. Nach dem zweiten Weltkrieg wurden verschiedene neue Notationen für chemische Verbindungen entwickelt, wie zum Beispiel die *Wiswesser* Linearnotation [22]. Diese zeichnen sich durch größere Kompaktheit aus, die Eineindeutigkeit ist auch hier im Allgemeinen nicht gegeben. In dieser Notation wird eine chemische Struktur durch eine Sequenz von Buchstaben und Zahlen dargestellt. Im Grunde genommen handelt es sich bei der IUPAC-Nomenklatur zwar auch um eine Linearnotation; im Unterschied zu ihr kann in den neueren Notationen die Topologie der Strukturen direkt an der Codierung abgelesen werden. Die semantische Interpretation durch einen Computer wird dadurch erheblich erleichtert.

Zwar wurden diese Notationen noch vor der Erfindung des Computers entworfen und waren lediglich dazu gedacht chemische Verbindungen einfacher chiffrieren zu können; dank ihrer Kompaktheit wurden sie aber in jener Zeit, als Computer-Speicher noch sehr teuer war, nicht nur zur vorherrschenden Darstellungsform beim Informationsaustausch zwischen Chemikern und sondern auch in Informationssystemen. Bis heute sind noch einige weitere Notationen entwickelt worden, von denen es wiederum mehrere Erweiterungen gibt, um jeweils spezielle Eigenschaften chemischer Strukturen darstellen zu können [22]. Wegen ihrer wichtigen Rolle in heutigen Informationssystemen, und weil wir in dieser Arbeit auch eine solche Linearnotation (SMILES) eingesetzt haben, wollen wir im nächsten Abschnitt diese Repräsentationsformen näher betrachten.

2.3.1 Linearnotation

Eine Linearnotation stellt die Struktur einer chemischen Verbindung als lineare Sequenz von Buchstaben und Zahlen dar. Sie ist kompakt und leicht zu lernen. Eine der wichtigsten unter ihnen ist die *Wiswesser Line Notation* (WLN). Diese ist allerdings so kompakt, dass die Lesbarkeit dadurch vermindert wird. Neuere Repräsentationen dieser Art, die im Vergleich zur WLN zwar weniger kompakt, aber besser lesbar sind, sind zum Beispiel ROSDAL (**R***epresentation of* **O***rganic* **S***tructures* **D***escription* **A***rranged* **L***inearly*) [22], SLN (**S***ybyl* **L***ine* **N***otation*) [3] und SMILES (**S***implified* **M***olecular* **I***nput* **L***ine* **E***ntry* **S***ystem*) [67, 68]. Letztere hat sich wohl zur wichtigsten und am weitesten verbreiteten Linearnotation entwickelt.

SMILES wurde in 1986 von David Weininger bei der US Environmental Research Laboratory, USEPA, Duluth, entwickelt. Seitdem ist sie stark erweitert worden. Die Repräsentation beruht auf der Topologie der Molekülstruktur. Zur Konvertierung einer Struktur in einen Zeichenstring wird diese so durchlaufen, dass jedes Atom nur einmal besucht wird. Man könnte beispielsweise ein Variante des Algorithmus DFS einsetzen, um die Struktur zu traversieren. Dabei werden folgende sechs Grundregeln angewandt:

1. Atome werden durch ihre Atomsymbole aus dem Periodensystem der Elemente (PSE) repräsentiert.

2. Wasserstoff-Atome saturieren automatisch die freien Valenzelektronen und werden ausgelassen.

3. Benachbarte Atome stehen direkt nebeneinander, (mit Ausnahme von Verzweigungen (siehe unten)).

4. Doppel- und Dreifach-Bindungen werden durch „=“ bzw. „ # “ angezeigt.

5. Verzweigungen werden durch Klammern angegeben. Eine öffnende Klammer zeigt den Beginn einer Verzweigung an, die schließende Klammer hinter einem bestimmten Atom bedeutet, dass die Verzweigung nach diesem Atom abgeschlossen ist.

6. Zur Beschreibung von Ringen werden Ringschlußbindungen aufgebrochen. Um die Nachbarschaft der beiden Atome anzudeuten, die durch die aufgebrochene Bindung verknüpft sind, werden sie jeweils mit der gleichen Ziffer markiert.

Die SMILES-Notation für die Struktur in Abbildung 2.1 lautet zum Beispiel: *CC(C)CC1=CC=C(C=C1)C(C)C(O)=O*.

Eine spezielle Erweiterung von SMILES ist die USMILES (*Unique SMILES*), die manchmal auch als Broad SMILES bezeichnet wird. Es handelt sich um eine kanonische Struktur-Repräsentation. Unabhängig von der internen Atomnummerierung wird durch Anwendung des sogenannten CANGEN-Algorithmus [69] gewährleistet, dass aus einer chemischen Verbindung immer die gleiche kanonische und eineindeutige Kodierung erzeugt wird.

Weiterführende Informationen über SMILES und ihrer Erweiterungen können in der oben referenzierten Literatur oder auf den Web-Seiten der Firma Daylight [27] nachgelesen werden.

2.3.2 Graphentheoretische Repräsentation

Eine weitere Möglichkeit, chemische Strukturen in einer maschinenlesbaren Form zu darzustellen, ist die Repräsentation durch im Abschnitt 2.1.3 vorgestellte Graphrepräsentationen. Eine typische Repräsentation ist der *Molekulargraph*. Es handelt sich hier um einen ungerichteten attributierten Graph. Die Ecken eines Molekulargraphen entsprechen den Atomen und die Kanten den Bindungen im Molekül. Jeder Ecke wird außerdem das zugehörige Atomsymbol als Attribut zugeordnet. Bei den Kanten sind die Bindungsarten (Einfach-, Doppel- und Dreifachbindung) die entsprechenden Attribute.

Der Molekulargraph spiegelt also die Topologie einer Molekülstruktur wieder. Wie bei allgemeinen Graphen hat man bei Molekulargraphen die

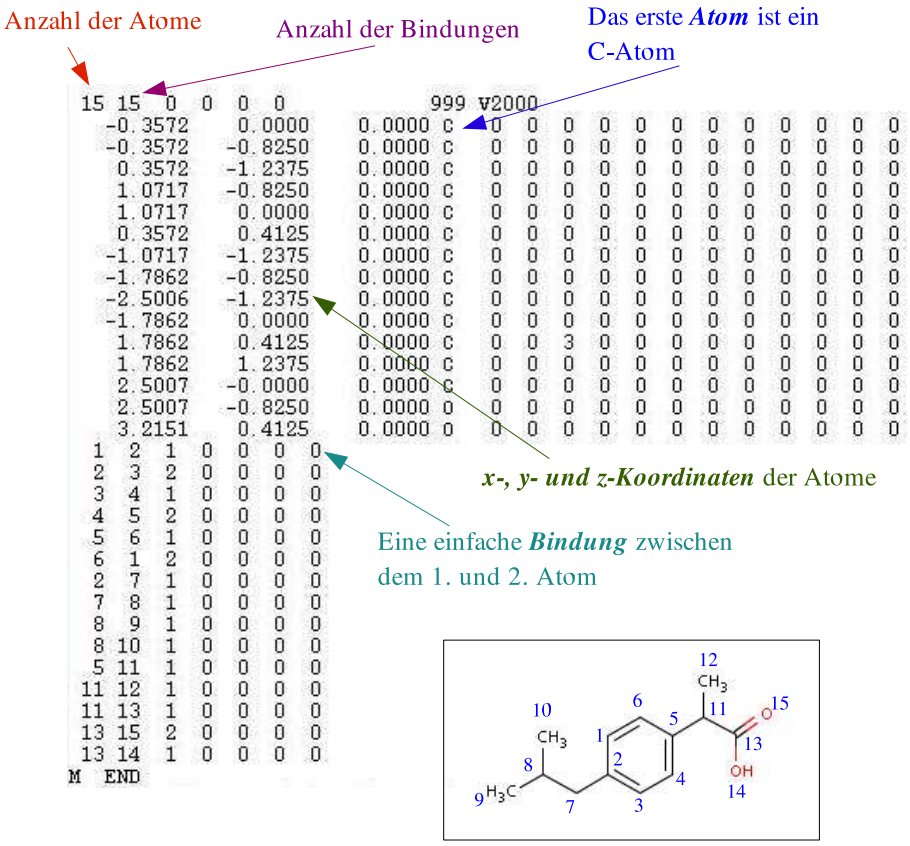


Abbildung 2.2: Verknüpfungstafel im MDL-Format (H-Atome wurden weggelassen). Die Atomnummern sind im Strukturdiagramm angegeben.

Möglichkeit, diese als Adjazenz-, Inzidenzmatrizen oder Adjazenzlisten zu speichern. Letztere Form hat den Vorteil, dass sie bei *dünnen* Graphen – d.h. wo $|E| \ll |V^2|$ gilt – weniger Speicherplatz benötigt. Diese Eigenschaft dürfte für die meisten Molekulargraphen zutreffen. Es wurden auch hier wieder Varianten der Speicherungsformen vorgeschlagen, siehe zum Beispiel [22]. Im Großen und Ganzen ergänzen sie die herkömmliche Speicherformen von Graphen um die Möglichkeit, Attribute zu Atomen und/oder Bindungen mitzuspeichern. Eine wichtige Rolle spielt dabei die sogenannte *Verknüpfungstafel (Connection Table)*. Sie hat sich seit den 1980er Jahren als Alternative für die Speicherung eines Molekulargraphen etabliert und ist heute fester Bestandteil zahlreicher Dateiformate für chemische Verbindungen. Als Beispiele sind die *Molfile*- und *SDfile*-Formate zu nennen. Beide wurden von der Firma Elsevier MDL [28] entwickelt.

2.3.3 Verknüpfungstafel (Connection Table)

Verknüpfungstafeln können als Varianten der Adjazenzlisten betrachtet werden. Auch hier gibt es wiederum unterschiedliche Darstellungsarten, die je nach Dateiformat leicht voneinander abweichen. Im Wesentlichen besteht eine Verknüpfungstafel aber aus zwei Teilen. Im ersten Teil sind die Atomsymbole zusammen mit ihren Indices nacheinander aufgeführt (*Atomlisten*). Danach werden in dem zweiten Teil alle Bindungen mit den jeweiligen Atomindices im Molekül aufgelistet (*Bindungslisten*). Im Gegensatz zu den Adjazenzlisten werden die ungerichteten Kanten in einem Molekulargraph nur einmal in der Verknüpfungstafel gespeichert.

Abbildung 2.2 zeigt eine Verknüpfungstafel vom Ibuprofen im MDL-Format. Diese Verknüpfungstafel bildet den Haupteil in dem für die Eingabe erzeugten SDfile. In solch einem SDfile können außerdem noch weitere Informationen über die betreffende Verbindung gespeichert werden, zum Beispiel ihr Synonym-Name, der Schmelzpunkt, die Dichte usw.

Kapitel 3

Konvertierung von Bitmap nach SDfile

Dieses Kapitel bildet den Kern der Arbeit. Hier wird das Konzept vorgestellt. Die einzelnen Teilaufgaben des vorgeschlagenen Lösungsansatzes werden nacheinander beschrieben.

3.1 Das Konzept

In Abbildung 3.1 sind die einzelnen Arbeitsschritte illustriert. Der Ansatz besteht aus drei Phasen: Vorverarbeitung, Graph-Matching und Rekonstruktion. In der ersten Phase – Vorverarbeitung – wird das Eingabe-Bild in eine Vektorgraphik konvertiert. Bevor aus dieser Vektorgraphik eine Graphdarstellung erzeugt wird, werden noch ein paar Nachbearbeitungen ausgeführt, um eine möglichst getreue Graphrepräsentation des Eingabe-Bildes zu erhalten. An dieser Stelle müsste außerdem die Trennung zwischen Atomsymbolen und Bindungen erfolgen. Um die Atomsymbole zu erkennen, sollte ein Zeichenerkennungsverfahren (Optical Character Recognition, OCR) implementiert werden. Im Rahmen der vorliegenden Arbeit wurde hierauf jedoch aus zeitlichen Gründen verzichtet. Statt dessen ist es einstweilen die Aufgabe des Benutzers, diese manuell zu spezifizieren. Eine Erweiterung unseres Programms um die notwendige Funktionalität ist für die Zukunft vorgesehen.

Nachdem also alle im Bild explizit angegebenen Atomsymbole festgestellt wurden, muss noch bestimmt werden, zu welchen Bindungen die Atomsymbole jeweils gehören. Aus den gewonnenen Informationen wird nun ein Graph erzeugt, der das Eingabe-Bild beschreibt.

Für die Erkennung unbekannter Strukturen wurde eine Bibliothek von häufig vorkommenden Strukturfragmenten zusammengestellt. Sie wurden ebenfalls in Form von Graphen abgelegt und dienen zur Laufzeit als Modelle für die Erkennung eines Eingabe-Graphen.

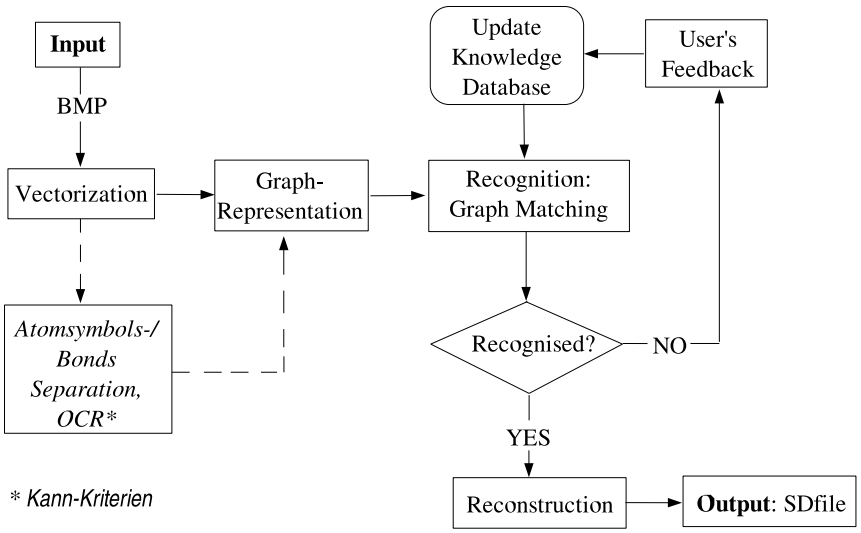


Abbildung 3.1: Ein Konzept zur Rekonstruktion chemischer Strukturen aus Bitmap-Dateien.

Die zweite Phase beschäftigt sich dann mit der Aufgabe, den Eingabe-Graph mit den vordefinierten Modell-Graphen mittels Graph-Matching Verfahren zu vergleichen. Es werden dabei alle Modell-Graphen bestimmt, die im Eingabe-Graph enthalten sind. Aus den berechneten Ergebnissen wird die unbekannte Struktur im letzten Schritt rekonstruiert.

Falls sich so eine Struktur ohne Valenzfehler (d.h. wenn alle Atome durch genau soviele Bindungen mit anderen Atomen im Molekül verbunden sind, wie ihre Bindungszahlen es vorschreiben) rekonstruieren lässt, wird sie automatisch im SDfile-Format gespeichert und in einer Benutzeroberfläche angezeigt. Ansonsten besteht für die Benutzer noch die Möglichkeit, die Rekonstruktion manuell nachzubearbeiten. Diese Nachbearbeitungsmöglichkeit ist natürlich auch dann gegeben, wenn das System seine Arbeit für einen Erfolg hält.

Parallel zum Matching wird zusätzlich überprüft, ob die im Eingabe-Graphen enthaltenen zyklischen Komponenten jeweils komplett durch einen Modell-Graph gematcht werden können. Falls nicht, werden sie als neue Modell-Graphen aufgenommen. Auf diese Weise hoffen wir, den Erkennungsprozess durch Lernen beschleunigen zu können. Die Beschränkung auf zyklische Strukturen erklärt sich aus der Tatsache, dass die Bibliothek bereits die notwendigen Fragmente enthält, um alle nicht-zyklische Strukturen zu rekonstruieren.

3.2 Vorüberlegungen

Eine grundlegende Frage im Bereich der Mustererkennung, und damit auch für die vorliegende Arbeit, ist wie sich das Eingabe-Bild geeignet repräsentieren lässt. Der Repräsentationsform kommt deshalb eine besondere Bedeutung zu, weil sich verschiedene Repräsentationen jeweils besonders für die Verarbeitung mit verschiedenen Methoden eignen.

Im ersten Abschnitt wird eine Graphdarstellung für das Eingabe-Bild erzeugt. Dies erfolgt über eine Vektorgraphik Konvertierung aus der Bitmap-Eingabe. Hierzu setzen wir die Opensource-Software AutoTrace ein [4].

Bei näherer Betrachtung der 2D-Repräsentation chemischer Strukturen lässt sich feststellen, dass eine solche Struktur aus vielen kleineren Strukturfragmenten besteht. Das kleinste Strukturfragment ist dabei ein Liniensegment, welches eine Bindung zwischen zwei Atomen darstellt. Größere Fragmente werden etwa durch zyklische Einheiten gebildet. Desweiteren kann man beobachten, dass diese Grundfragmente wiederholt in einer Struktur vorkommen können. Zum Beispiel kann eine komplexere zyklische Komponente aus mehreren 6er-Ringen bestehen oder aus einem einzigen Liniensegment kann eine beliebig lange Kette (verzweigt oder unverzweigt) zusammengesetzt werden. Die Strukturen sind dabei im allgemeinen unterschiedlich groß, in verschiedenen Lagen gezeichnet oder wurden mit Hilfe unterschiedlicher Programme mit ihrer jeweils unterschiedlichen graphischen Darstellung erstellt. Dennoch liegen ihre Komponenten in der Regel in bestimmten geometrischen Verhältnisse zueinander vor. So ist etwa das Längenverhältnis zweier Bindungen zueinander in zwei Darstellungen sicherlich bedeutend ähnlicher als die durchschnittliche Länge einer Bindung in den beiden Darstellungen.

Von diesen Merkmalen ausgehend, scheint es für uns wichtig, dass eine Datenstruktur für das Eingabe-Bild folgende Kriterien erfüllt:

- Transformationsunabhängigkeit, d.h. translations-, rotations- und skalationsunabhängig;
- Speicherungsmöglichkeit für die (geometrische) Relationen der Komponenten untereinander;
- Beschreibungsmöglichkeit der Strukturen durch ihren hierarchischen Aufbau.

Diese Anforderungen machen deutlich, dass die strukturelle Mustererkennung ein naheliegender Lösungsweg ist [6, 54].

Charakterisch für Anwendungen der strukturellen Mustererkennung ist es, dass die zu erkennende Eingabe komplexere Objekte enthält, die aus einfacheren Grundstrukturen bestehen. Ferner stehen diese Objekte/Grundstrukturen in der Regel in bestimmten Relationen zueinander, die für die Erkennung relevant sein können. Diese Relationen können unterschiedlicher Natur

sein, zum Beispiel geometrischer, räumlicher, konzeptioneller usw. Die Grundidee der strukturellen Mustererkennung liegt nun darin, die zu erkennenden Objekte mittels Datenstrukturen wie Strings oder Graphen zu beschreiben. Solche Datenstrukturen stellen Mechanismen zur Verfügung, mit denen nicht nur die Relationen zwischen Objekten beziehungsweise ihrer Grundstrukturen untereinander explizit beschrieben werden können, sondern sie erleichtern auch die Darstellung des hierarchischen Aufbaus von Objekten. Die Erkennung von unbekannten Objekten erfolgt dann im Allgemeinen durch Vergleiche der entsprechenden Objektrepräsentationen mit einer Anzahl von vordefinierten Modell-Objekten, d.h. es muss die Frage beantwortet werden, wie *ähnlich* zwei Objekte zueinander sind.

Unter den möglichen Repräsentationsformen – Strings oder Graphen – sind Strings die einfachste und bezüglich der Raum- sowie der Zeitkomplexität die Günstigste. Sie sind allerdings nicht so mächtig und flexibel wie Graphen. Diese gehören zu den allgemeinsten Formalismen zur Objekt-Repräsentation, die bisher im Bereich der strukturellen Mustererkennung vorgeschlagen worden sind. Im Gegensatz zu Strings sind Graphen auch für mehrdimensionale Objekt-Repräsentation geeignet. Dabei werden typischerweise Teile eines komplexen Objekts als Ecken, und die Relationen zwischen ihnen als Kanten eines Graphen dargestellt. Es besteht ferner die Möglichkeit, durch Einsetzen von Ecken- und Kantenattributen weitere Informationen in die Graphrepräsentation einzubeziehen. So werden oft Werte wie die Länge eines Segmentes, die Größe oder Farbe einer Bildregion, die räumliche Distanz zwischen zwei Punkten, oder der Winkel zwischen zwei geraden Linien als Attribute verwendet. Neben dem mächtigen Darstellungsformalismus haben Graphen noch eine Reihe interessanter Invarianz-Eigenschaften anzubieten. Beispielsweise kann ein Graph rotiert, skaliert oder verschoben werden, er bleibt im mathematischen Sinne derselbe Graph [6, 8]. Damit erfüllen Graphen die ersten beiden Anforderungen, die wir an unsere Repräsentationsform stellen. Für den dritten Punkt – den hierarchischen Aufbau komplexerer Strukturen – gibt es ebenfalls eine Lösung, die wir in Abschnitt 3.4.4 vorstellen.

Aufgrund der genannten Eigenschaften haben wir uns für Graphen als Repräsentationsform chemischer Strukturen entschieden. Damit lässt sich die Frage nach der Ähnlichkeit von Objekten auf die Bestimmung der Ähnlichkeit zwischen Graphen zurückführen, was im Allgemeinen als *Graph-Matching* bezeichnet wird.

Damit stellen sich zwei neue Fragen. Die erste betrifft die Zeitkomplexität des Vergleichs zweier Strukturen miteinander. Bei allen *optimalen Algorithmen* kann im schlechtesten Fall eine bezüglich der Anzahl der Ecken im Graphen exponentielle Zeit erforderlich sein. Im Gegensatz dazu haben *approximierte Algorithmen* zwar nur polynomielle Zeitkomplexität, liefern allerdings möglicherweise keine optimale Lösung [11, 70]. Da die zu rekonstruierenden Strukturen in der Regel nicht so groß sind (oft unter 100 Ecken

in der Molekulargraphdarstellung), denken wir, dass ein Vergleich zweier Strukturen durch einen optimalen Algorithmus in einer vertretbaren Zeit durchgeführt werden kann[1].

Das zweite Problem besteht darin, dass es mehrere vordefinierte Modell-Graphen geben kann, die mit einem Eingabe-Graph verglichen werden müssen, um das Eingabe-Bild rekonstruieren zu können. In diesem Fall muss zur Rekonstruktion der Eingabe-Graph sequentiell mit jedem Modell-Graph verglichen werden. Die Folge davon ist, dass die Laufzeit annähernd linear in der Anzahl der vordefinierten Modell-Graphen ist. Um diesem Problem zu begegnen, speichern wir die Modell-Graphen in einem sogenannten *Dekompositionsnetzwerk*, das von B. Messmer vorgeschlagen wurde [46]. Dieses Verfahren ermöglicht eine kompakte Darstellung der Modell-Graphen. Dazu werden sie in einem „off-line"-Prozess rekursiv in kleinere Subgraphen zerlegt und in dem Dekompositionsnetzwerk gespeichert. Dabei wird darauf geachtet, dass jeder gemeinsame Subgraph, der in einem oder mehreren Modell-Graphen mehrmals enthalten ist, nur genau *einmal* gespeichert wird. Insbesondere wird in einem solchen Dekompositionsnetzwerk explizit gespeichert aus welchen kleineren Subgraphen ein größerer Subgraph aufgebaut ist. Zur Laufzeit wird das Dekompositionsnetzwerk herangezogen, um die Vergleiche der Modell-Graphen mit dem Eingabe-Graph effizient zu gestalten. Dabei werden – beginnend mit den kleinsten Subgraphen im Dekompositionsnetzwerk – immer größere Modell-Graphen mit der Eingabe gematcht. Wir merken uns für jeden dieser Vergleiche, ob er erfolgreich war. Diese Ergebnisse dienen dann als Grundlage für weitere Vergleiche immer größerer Subgraphen aus dem Netzwerk mit dem Eingabe-Graph. Auf diese Weise werden gemeinsame Subgraphen in den Modell-Graphen nur einmal mit dem Eingabe-Graph verglichen, so dass die Laufzeit des Matchings nur sublinear von der Anzahl der Modell-Graphen abhängt.

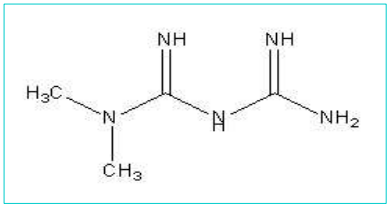
Es folgen nun die Beschreibungen der einzelnen Aufgabenabschnitte.

3.3 Vorverarbeitung des Eingabe-Bildes

Im ersten Schritt muss eine Graphdarstellung aus dem Eingabe-Bild erzeugt werden. Das Ergebnis dient dann als Eingabe für die Subgraphisomorphismen-Suche im nächsten Abschnitt.

Der Graph einer Eingabe wird jedoch nicht direkt aus deren Bitmap-Datei generiert, vielmehr geschieht dies über einen Zwischenschritt: die Konvertierung vom Bitmap zur Vektorgraphik. Dabei benutzen wir als Datei-Format der Vektorgraphik SVG (***S**calable **V**ector **G**raphics*), ein auf Xml basierendes Dateiformat [24].

[1] Diese Vermutung wurde durch unsere Testergebnisse bestätigt. Für eine vollständige Rekonstruktion komplexerer Teststrukturen waren bisher nie mehr als 1.3 Sekunden erforderlich. Siehe auch Kapitel 4.



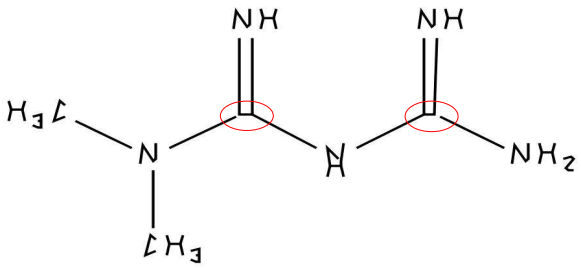


Abbildung 3.2: Eine von AutoTrace aus einer Bitmap-Datei konvertierte Vektorgraphik. Typische Rauschdaten wie zerbrochene Liniensegmente oder zusätzliche kurze Linien sind eingekreist.

3.3.1 Konvertierung von Bitmaps zu Vektorgraphiken

Für die Konvertierung von Bitmap-Dateien zu Vektorgraphiken gibt es bereits eine Vielzahl von Softwarepaketen. Also haben wir zunächst einige solche Programme getestet. Darunter waren Vector Eye [10], Vextractor [65], Potrace [53] und AutoTrace [4]. Die ersten beide Programme sind kommerzielle Produkte und laufen nur unter dem Windows Betriebssystem. Die beiden letzteren Pakete sind Open-Source und stehen für verschiedene Betriebssysteme zur Verfügung. Obwohl Vector Eye mit einigen Einstellungen eine Vektorgraphik lieferte, die auf den ersten Blick das Original recht getreu wiedergibt, ist es für unsere Arbeit leider nicht geeignet. Das Programm zerlegt nämlich lange Liniensegmente oder Kurven in sehr viele kleinere Vektoren, so dass eine Weiterverarbeitung großen Aufwand erfordert. Bei Vextractor wurden oft Teile von Buchstaben oder Zahlen nicht erkannt. Potrace lieferte insgesamt recht gute Ergebnisse. Ein Manko dieses Programms war leider, dass die erzeugten Vektoren nur den Umriß des Eingabe-Bildes zeichnen. Es besteht keine Möglichkeit, deren Mittellinien zu extrahieren. Die von AutoTrace erzeugten Umriß-Vektorgraphiken sind im Vergleich zu Potrace etwas schlechter, dafür können jedoch die Mittellinien der Bilder extrahiert werden, aus denen wir schnell eine Graphdarstellung generieren können. Deshalb haben wir uns schließlich für AutoTrace entschieden.

Abbildung 3.2 zeigt eine von AutoTrace erzeugte Vektorgraphik. Sie zeigt auch typische Fehler, die bei der Vektorisierung auftreten können. Manchmal werden an denjenigen Stellen, wo mehrere Linien zusammentref-

fen, zusätzliche Linien erzeugt; ein anderes Mal werden eigentlich verbundene Linien voneinander getrennt. Diese Fehler müssen – sofern möglich – in einem Nachbearbeitungsprozess beseitigt werden. Auf diese Fehlerbehandlung werden wir später noch eingehen.

Nach der Konvertierung wird die entstandene Vektorgraphik dem Benutzer präsentiert. Da in der vorliegenden Arbeit noch keine automatische Zeichenerkennung implementiert worden ist, ist es nun seine Aufgabe, alle im Bild vorkommenden Atomsymbole manuell zu markieren und sie im SMILES-Format anzugeben.

Jetzt wenden wir uns der Frage zu, wie sich das Eingabe-Bild geeignet durch einen Graph darstellen lässt.

3.3.2 Graph-Repräsentation des Eingabe-Bildes

Nach der Konvertierung in SVG haben wir also Vektoren vorliegen, die Atomsymbole oder chemische Bindungen repräsentieren. Die Aufgabe besteht nun darin, jede dieser unterschiedlichen Einheiten in der Graphik zu identifizieren und anschließend eine sinnvolle Struktur aus ihnen zu rekonstruieren. Da die Identifikation der Atomsymbole – ob manuell markiert und spezifiziert, oder automatisch durch ein OCR-Programm erkannt – eine gesonderte Behandlung erfordert, bleiben nur die Bindungen zu erkennen. Eine Bindung ist somit das kleinste Primitiv, das zu interpretieren ist. Jede Bindung wird daher durch eine Ecke in jenem Graph gespeichert, den wir für das Eingabe-Bild erzeugen.

Aus der Vektorgraphik eines Eingabe-Bild wird ein einfacher ungerichteter Graph $G_I = (V_I, E_I, \mu_I, \nu_I)$ generiert. Dabei werden wir die Bindungen als Ecken im Graph G_I speichern. Mehrfachbindungen werden jedoch nur durch eine Ecke dargestellt. Damit die Art der Bindung später richtig rekonstruiert werden kann, merken wir sie uns in einem Attribut.

Falls zwei Liniensegmente l_1 und l_2 – in G_I durch v_1 und v_2 repräsentiert – einen gemeinsamen Endpunkt haben, dann existiert in G_I eine Kante $e = (v_1, v_2)$. Diese Kante hat ein Attribut, welches dem Winkel zwischen l_1 und l_2 (in positiver Drehrichtung) entspricht. Diese Repräsentation entspricht also einem attributierten Kantengraph des zugehörigen Molekulargraphen.

3.3.3 Nachbearbeitungen des Eingabe-Graphen

Die resultierenden Vektorgraphiken sind leider oft fehlerbehaftet. Dies kann an einem schlechten Scanvorgang, einer bereits fehlerhaften Zeichnung oder an Problemen bei der Konvertierung von Bitmap zur Vektorgraphik liegen. Möglicherweise auftretende Fehler sind zum Beispiel:

1. Unterbrechungen eines Liniensegmentes, das durchgezogen sein müßte.

2. Zwei nicht colineare Liniensegmente, die im Original verbunden sind, sind in der erzeugten Vektorgraphik nicht mehr verbunden.

3. Zusätzliche Liniensegmente können am Treffpunkt verschiedener Liniensegmente auftreten.

4. Der Winkel zwischen zwei Liniensegmenten kann von demjenigen in einer idealen Modell-Struktur abweichen.

Die letzten drei Fehlerarten sind bei unseren bisherigen Beobachtungen häufig aufgetreten, wobei der 2. Fehler am häufigsten beobachtet wurde; in der Regel tritt er da auf, wo mehr als zwei Liniensegmente zusammentreffen. Bei der Nachbearbeitung der Vektorgraphiken versuchen wir, bereits möglichst viele dieser Fehlern zu korrigieren. Zum Aufspüren dieser Fehler wird zunächst ein sogenannter *relativer Nachbarschaftsgraph* der Endpunkte der Liniensegmente in der konvertierten Vektorgraphik generiert. Anhand der Nachbarschaften in diesem Graph werden dann diejenigen Stellen aufgedeckt, wo wahrscheinlich einer der oben beschriebenen Fehler vorliegt. Schließlich wird nach dem Testen einiger Bedingungen, auf die wir im Folgenden eingehen, entschieden, ob an diesen Stellen eine Korrektur erfolgen sollte.

Nun stellt sich zunächst die Frage, wie „relative Nachbarschaft“ definiert ist. Wir beziehen uns hier auf die Arbeit von G.T. Toussaint [61] und betrachten zwei Punkte p_i und p_j aus einer endlichen disjunkten Punktmenge $P = \{p_1, p_2, \ldots, p_n\}$ in einer Ebene als „relativ nah“ beieinander, wenn $d(p_i, p_j) \leq \max[d(p_i, p_k), d(p_j, p_k)]$ für alle $k = 1, \ldots, n$ und $k \neq i, j$ gilt. Dabei bezeichnet d die euklidische Distanz in der Ebene. Der relative Nachbarschaftsgraph ist dann wie folgt definiert:

Definition 3.1 ***(Relativer Nachbarschaftsgraph)***
Sei $P = \{p_1, p_2, \ldots, p_n\}$ eine endliche disjunkte Menge von Punkten auf einer Ebene. Ein ***relativer Nachbarschaftsgraph*** *(Relative Neighborhood Graph, RNG) ist ein Graph $G = (P, E)$ mit folgenden Eigenschaften:*

1. *Die Punkte in P stellen die Ecken des Graphen G dar,*

2. *$E = \{(p_i, p_j) | d(p_i, p_j) \leq \max[d(p_i, p_k), d(p_j, p_k)]\ \forall k = 1, \ldots, n$ und $k \neq i, j\}$.*

Den relativen Nachbarschaftsgraph erhält man also, indem man zwei Punkte p_i und p_j, für alle $i, j = 1, 2, \ldots, n, i \neq j$ genau dann durch eine Kante miteinander verbindet, wenn sie relative Nachbarn von einander sind. In [61] wurde gezeigt, dass die Anzahl der Kanten eines RNG durch $O(n)$ beschränkt ist.

Toussaint hat zwei Algorithmen für die Erzeugung eines solchen Graphen vorgestellt. Der eine erfordert $O(n^3)$ Zeit, funktioniert aber auch im

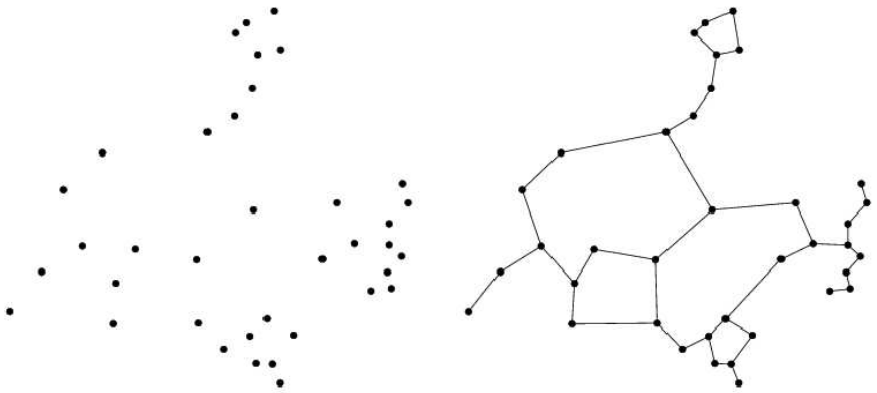


Abbildung 3.3: Relativer Nachbarschaftsgraph einer Punktmenge [61].

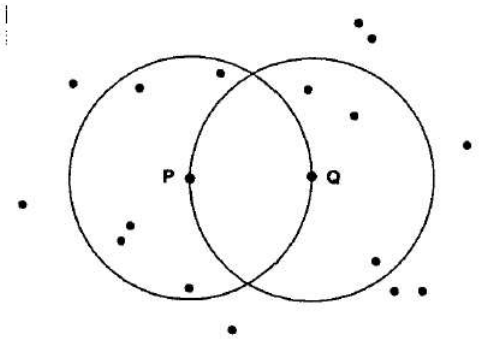


Abbildung 3.4: Zur Berechnung einer Kante im RNG: Zwei Punkte p, q werden durch eine Kante verbunden, wenn kein anderer Punkt innerhalb des von ihnen definierten Kreiszweiecks liegt.

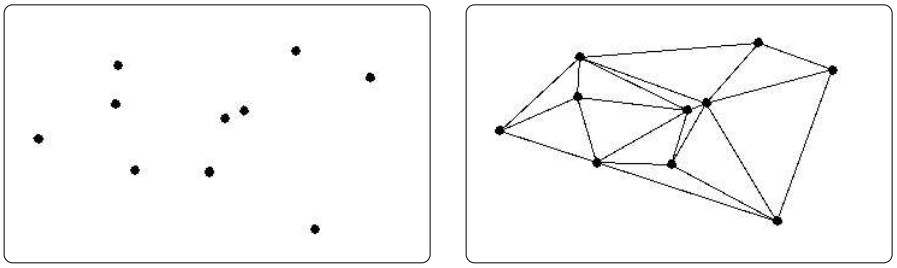


Abbildung 3.5: Delaunay Triangulation einer Punktmenge [61].

mehrdimensionalen Raum; der andere läuft in $O(n^2)$ Zeit, ist aber auf den $\mathbb{R}^2$ beschränkt. Ersterer ist ein naiver Algorithmus. Er berechnet zunächst für alle Paare von Punkten aus der Eckenmenge deren Distanz, prüft dann nach, ob es einen anderen Punkt gibt, der beiden näher liegt als sie einander (siehe Abbildung 3.4). Falls nicht, werden sie durch eine Kante verknüpft.

Im Algorithmus 3.1 ist der naive Algorithmus beschrieben. Für die erste for-Schleife werden $O(n^2)$ Operationen benötigt, um $O(n^2)$ Punktpaare zu betrachten. Die zweite und dritte for-Schleife brauchen jeweils für jedes Punktepaar $O(n)$ Rechenschritte. Somit beträgt die Zeitkomplexität des Algorithmus insgesamt $O(n^3)$.

RELATIVER NACHBARSCHAFTSGRAPH

EINGABE:

V: Eine endliche Punktmenge $V = \{p_1, \ldots, p_n\}$.

AUSGABE:

G: Ein relativer Nachbarschaftsgraph $G = (V, E)$.

ALGORITHMUS:

(1) **for all** Paare von Punkten $(p_i, p_j) \in V$ mit $i, j = 1, \ldots, n, i \neq j$ **do**
(2) Berechne $d(p_i, p_j)$
(3) **end for**
(4) **for all** Paare von Punkten (p_i, p_j) **do**
(5) Berechne $d^k_{max} = max[d(p_i, p_k), d(p_k, p_j)]$ für $k = 1, \ldots, n \neq i, j$
(6) **end for**
(7) **for all** Paare von Punkten (p_i, p_j) **do**
(8) **if** $(\nexists\ d^k_{max} : d^k_{max} < d(p_i, p_j))$ **then**
(9) Füge eine Kante $e = (p_i, p_j)$ zu G hinzu
(10) **end if**
(11) **end for**

Algorithmus 3.1: Naiver Algorithmus zur Berechnung eines relativen Nachbarschaftsgraphen.

Der zweite Algorithmus verwendet die Eigenschaft, dass der relative Nachbarschaftsgraph eine Teilmenge einer *Delaunay-Triangulation* ist. Eine Delaunay-Triangulation einer Punktmenge ist die Einteilung des Inneren der Punktmenge in Dreiecke, wobei der maximale Winkel zwischen allen Dreiecksseiten minimiert wird. Ein Beispiel ist in Abbildung 3.5 gezeigt. Statt die Distanzen aller Punktpaare explitzit zu bestimmen, berechnet man zuerst die Delaunay-Triangulation der gegebenen Punktmenge, welche $O(n)$

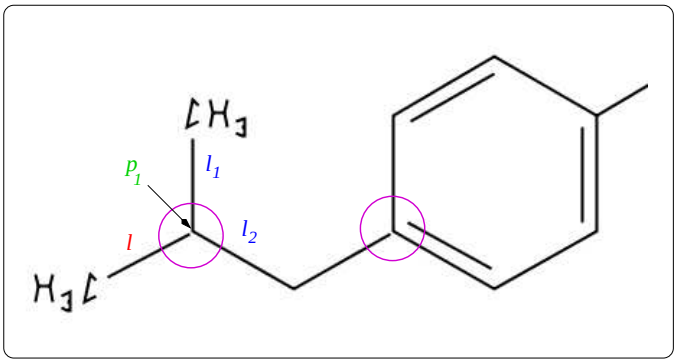


Abbildung 3.6: Zerbrochene Liniensegmente, die miteinander verbunden werden sollten.

Kanten zurückgibt. Für diese Berechnung gibt es verschiedene Algorithmen, die eine Worst-Case Zeitkomplexität von $O(n \log n)$ haben [21]. Jedem Punktepaar (p_i, p_j) wird anschließend eine Kante der berechneten Delaunay-Triangulation zugeordnet, die diese beiden Punkte als Endpunkte hat. Die darauf folgenden Schritte entsprechen denen des naiven Algorithmus, d.h. es werden all diejenigen Kanten der Triangulation entfernt, die nicht Kanten des zu erzeugenden relativen Nachbarschaftsgraphen sind.

In dieser Arbeit wurde lediglich der naive Algorithmus implementiert. Um die Laufzeit zu verbessern, sollte der zweite Algorithmus, oder aber der von K. J. Supowit in [58] vorgeschlagene Algorithmus implementiert werden. Letzterer geht ebenso von einer Delaunay-Triangulation der gegebenen Punktmenge aus. Er kann jedoch in $O(n \log n)$ nachprüfen, welche Kanten der Triangulation nicht zum relativen Nachbarschaftsgraphen gehören und diese dann aus der Kantenmenge entfernen.

Anhand der Nachbarschaften im RNG werden nicht nur die am Anfang dieses Abschnittes beschriebenen Fehler aufgespürt, sondern sie werden auch benutzt, um Mehrfach-Bindungen zu identifizieren und Atomsymbole explizit Bindungen zuzuordnen. Wir werden nun diese einzelne Nachbearbeitungen näher betrachten.

Minimierung von Vektorisierungsfehlern

Abbildung 3.4 zeigt die typischen Fehler, die bei der Konvertierung einer Bitmap-Datei zur Vektorgraphik auftreten können. Um bessere Ergebnisse beim Matching erzielen zu können, versuchen wir diese zu korrigieren. Dazu wird ein relativer Nachbarschaftsgraph der Endpunkte der Vektoren erzeugt.

Wir suchen dabei in erster Linie Fehler, bei denen mehrere Liniensegmente, die im Original in einem Punkt zusammentreffen, in der Vektorgraphik nicht mehr alle miteinander verbunden sind. Seien l_i mit $i \geq 2$ und l solche Liniensegmente, wobei die l_i den Punkt p_1 als gemeinsamen Endpunkt haben

und durch die Vektorisierung fälschlicherweise nicht mehr mit l verbunden sind (Abbildung 3.6). l hat dann in der Vektorgraphik in der Regel einen Endpunkt p_2, der so dicht bei p_1 liegt, dass diese beiden Punkten im RNG tatsächlich durch eine Kante verknüpft werden. Die Verbindung des Liniensegmentes l mit den anderen Segmenten erfolgt, falls sie eine Verlängerung um weniger als 12 Prozent der ursprünglichen Länge und eine Änderung der Lage von l im Raum um weniger als 5 Grad erfordert. Die Wahl dieser Parameter erklärt sich wie folgt: Wir konnten bisher keine Darstellung finden in der eine Korrektur von mehr als 5 Grad erforderlich gewesen wäre. Ebenso konnten wir keine Darstellung finden, in der eine Verlängerung einer Linie um mehr als 12 Prozent notwendig gewesen wäre.

In dieser Phase werden ferner auch alle Liniensegmente, die kürzer als ein Fünftel der durchschnittlichen Länge aller Liniensegmenten im Eingabe-Bild haben, aus der Eingabe entfernt. Diese sind entweder Rauschdaten oder sie sind Teil einer Chiralbindung, die eine gesonderte Behandlung erfordern.

Bestimmung von Mehrfach-Bindungen

Mehrfach-Bindungen in einer chemischen Struktur werden durch echte Parallelen dargestellt. Um diese Liniensegmente als eine Mehrfach-Bindung identifizieren zu können, werden im Allgemeinen die Nachbarschaften der zugehörigen Endpunkte im RNG betrachtet. Da diese Liniensegmente in der Regel verhältnismäßig dicht beieinander liegen, werden ihre Endpunkte durch eine Kante im RNG verbunden. Die Existenz einer RNG-Kante allein garantiert natürlich noch nicht, dass die entsprechenden Endpunkte in einer relevanten Nähe zueinander liegen. Deswegen wird zusätzlich nachgeprüft, ob die Distanz der Endpunkte unterhalb eines geeignet gewählten Schwellenwertes liegt. Falls dies zutrifft, verschmelzen wir im Falle einer positiven Parallelitätsprüfung die entsprechende Ecken zu einer Ecke und merken uns die Mehrfachbindung in einem Attribut dieser Ecke. Der Schwellenwert von einem Fünftel der durchschnittlichen Linienlänge hat sich in unseren Experimenten als gut geeignet erwiesen.

Explizite Zuordnung von Atomsymbolen und Bindungen

Atomsymbole, die in einer chemischen Struktur explizit angegeben sind, stehen in der Regel direkt am Ende der entsprechenden Bindungen (Liniensegmente) oder zwischen mehreren Bindungen. Bei einem „sauberen“ Bild kann man davon ausgehen, dass sie keine der Bindungen berühren. Um eine chemische Struktur richtig rekonstruieren zu können, müssen wir also nicht nur die Atomsymbole identifizieren, sondern es muss auch erkannt werden, zu welcher Bindung ein Atomsymbol gehört. Im Rahmen dieser Arbeit haben wir keine Zeichenerkennung implementiert. Die Atomsymbole werden vom Benutzer explizit angegeben. Es bleibt jedoch die Aufgabe, zu klären, wie

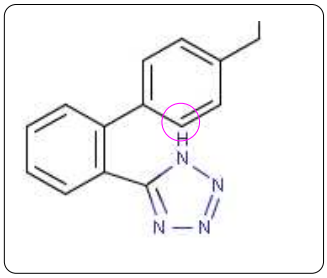


Abbildung 3.7: Zuordnung von Atomsymbolen und Bindungen. Trotz der Nähe zum Text sollen die Bindungen mit dem eingekreisten Endpunkt mit diesem assoziert werden.

sich die Zugehörigkeit eines solchen Atomsymbols zu bestimmten Bindungen feststellen lässt.

Wenn der Benutzer einen bestimmten Bereich im Bild markiert und das an dieser Stelle befindliche Atomsymbol spezifiziert, werden die entsprechende Vektor-Objekte durch ein Text-Objekt ersetzt. Das Rechteck, welches die markierten Vektor-Objekte einschließt (*bounding box*), wird ermittelt. Der Mittelpunkt dieses Rechtecks, p_A, wird durch eine Ecke im RNG repräsentiert. Wir informieren uns dann wiederum mit Hilfe der Nachbarschaften von p_A im RNG, welche Bindungen mit dem Atomsymbol im Punkt p_A verknüpft sein könnten. Wir bemerken, dass durchaus auch mehrere Atomsymbole gemeinsam auftreten können, insbesondere wenn es sich um ein sogenanntes Superatom handelt, zum Beispiel bei der Carboxylatgruppe (-COOH). Die Auswahl des Mittelpunkt des die Atomsymbole einschließenden Rechteckes ist dadurch begründet, dass die Atomsymbole in der Regel unterschiedlich lang sind und in allen möglichen Lagen auftreten können.

Sei $Adj[p_A] = \{p_1, \ldots, p_k\}$ die Menge der Nachbarn von p_A. Für jeden Nachbar $p_i \in Adj[p_A]$ wird zunächst nachgeprüft, ob die beiden Punkte *nah genug* beieinander liegen. Ihre Distanz sollte nicht mehr als ein 3/5 der Länge des entsprechenden Liniensegmentes betragen. Diese Einschränkung ist dafür da, dass wir nicht fälschlicherweise eine Bindung mit einem Atom assozieren. Der Grenzwert wurde aus Testerfahrungen ermittelt. Bei einem niedrigeren Wert wurde machmal Bindungen nicht mit Atomen verbunden, ein höherer Wert hatte anderseits dazu geführt, dass falsche Bindungen assoziert wurde. Falls also ein Nachbarpunkt p_i nicht zu weit von p_A liegt, suchen wir nach allen Liniensegmenten, die p_i ursprünglich als Endpunkt hatten. Wenn es jedoch mehr als ein solches Liniensegment gibt, ist es eher unwahrscheinlich, dass all diese Liniensegmente – die ja Bindungen darstellen – mit einem expliziten Atomsymbol verbunden werden sollten (siehe Abbildung 3.7). Deshalb wird in solch einem Fall nichts unternommen. Ansonsten wird das Atomsymbol der entsprechenden Bindung zugeordnet. Wir nehmen nun an, dass alle Liniensegmente mit Endpunkten $p_i \in Adj[p_A]$ auch tatsächlich

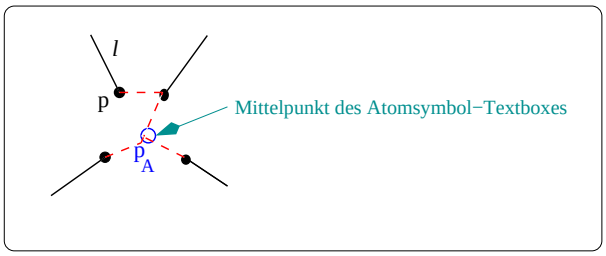


Abbildung 3.8: Zuordnung von Atomsymbolen und Bindungen: Obwohl p und p_A im relativen Nachbarschaftsgraph nicht benachbart sind, muss das Atom am Punkt p_A auch mit der Bindung l verknüpft sein.

mit dem Atomsymbol an der Stelle p_A verbunden sein sollten. Dann sind noch folgende Punkte zu beachten:

1. Falls $Adj[p_A]$ mehr als ein Element hat, müssen die drei folgende Aufgaben bearbeitet werden:
   (a) Es muss ein neuer gemeinsamer Treffpunkt p'_A für alle Liniensegmente, die jeweils einen Endpunkt $p_i \in Adj[p_A]$ haben, berechnet werden. p'_A bildet die neue Lokation für das entsprechende Atom.
   (b) Jedes Liniensegment l mit Endpunkten (p_i, p_{opp}) oder (p_{opp}, p_i) wird zu $l = (p'_A, p_{opp})$ bzw. $l = (p_{opp}, p'_A)$ korrigiert.
   (c) Alle Ecken in der Graphrepräsentation, die ursprünglich ein Liniensegment mit einem Endpunkt in $p_i \in Adj[p_A]$ darstellen, werden miteinander durch eine Kante verbunden.
2. Es könnte durchaus noch Liniensegmente geben, die mit demselben Atomsymbol an der Stelle p_A verbunden sein sollten, deren Endpunkte aber nicht mit p_A im RNG benachbart sind (siehe ABBILDUNG). Einer der Endpunkte dieser Liniensegmente ist dann in der Regel jedoch mit einem Punkt $p_i \in Adj[p_A]$ benachbart. D.h. Korrekturen wie in 1.(b-c) müssen auch für diese Liniensegmente vorgenommen werden.

Die beschriebenen Nachbearbeitungen sind notwendig, damit sowohl die Nachbarschaften im Eingabe-Graph als auch die Attribute (wie zum Beispiel der Winkel zwischen zwei Liniensegmente) sinnvoll korrigiert werden können. Dies führt oft zu einer Verbesserung der Matching-Ergebnisse.

3.4 Graph-Matching

In diesem Abschnitt geben wir zunächst einen informellen Überblick über Graph-Matching Verfahren. Anschließend präsentieren wir die Standard-Algorithmen für diese Aufgabe. Dann wenden wir uns der Frage zu, welche

Graphen wir als Basis für die Erkennung der beliebigen Eingabe-Graphen wählen. Außerdem stellen wir eine geeignete Datenstruktur für diese Modell-Graphen vor.

3.4.1 Einführung

Zu den Standard-Konzepten im Graph-Matching gehören *Graph-*, *Subgraphisomorphismen* und *maximale gemeinsame Subgraphen*. Zwei Graphen werden als isomorph bezeichnet, wenn sie eine identische Struktur aufweisen. Formeller ist ein Graphisomorphismus zwischen zwei Graphen G_1 und G_2 eine bijektive Abbildung der Ecken in G_1 auf die Ecken von G_2, wobei die Nachbarschaften und die Attribute (falls vorhanden) erhalten bleiben. Analog ist ein Subgraphisomorphismus zwischen G_1 und G_2 ein Graphisomorphismus zwischen G_1 und einem Subgraph von G_2. Subgraphisomorphismem stellen somit ein allgemeineres Konzept als Graphisomorphismen dar; jeder Graphisomorphismus ist auch ein Subgraphisomorphismus, aber nicht umgekehrt. Graph- und Subgraphisomorphismen werden angewandt, um zu überprüfen, ob zwei Objekte die gleichen sind bzw. ob ein Objekt im anderen enthalten ist. Ein maximal gemeinsamer Subgraph zweier Graphen G_1 und G_2 ist der größte Graph G, der sowohl zu einem Subgraph von G_1 als auch zu einem Subgraph von G_2 isomorph ist. Je größer also der maximal gemeinsame Subgraph von zwei Graphen G_1 und G_2 ist, desto ähnlicher sind sie.

Die genaue Komplexität des Graphisomorphismusproblems ist nach wie vor nicht bekannt. Man konnte bisher weder zeigen, dass es *NP*-vollständig ist, noch dass es in polynomieller Zeit lösbar ist. Für die beiden anderen Probleme, Subgraphisomorphismus und maximal-gemeinsamer Subgraph, wurde dagegen gezeigt, dass sie *NP*-vollständig sind. Nichtdestotrotz findet Graph-Matching wegen der Mächtigkeit des Formalismus in sehr vielen Bereichen Anwendung. Eine der frühesten Anwendungen war die (Sub-)Struktursuche in chemischen Datenbanken (zitiert aus [44]), welche bis heute nach wie vor eine sehr wichtige Rolle spielt [22]. Anwendungen jüngerer Jahre umfassen fall-basiertes Schließen, maschinelles Lernen [13, 45], usw. Weiterhin finden sich zahlreiche Anwendungen in den Bereichen Mustererkennung und „Machine Vision“ in der Literatur [8, 12]. Dazu gehören zum Beispiel die Zeichenerkennung [38], graphische Symbolerkennung [2, 39, 35], 3D-Objekterkennung, Video-Indexierung [57], biometrische Identifikation [49] usw. Umfangreiche Referenzen können in [8, 12] gefunden werden.

In jüngster Zeit war eine spezielle Ausgabe der Zeitschrift International Journal of Machine Intelligence and Image Analysis dem Thema Graph-Matching im Bereich Mustererkennung gewidmet [40]. Aktuelle Entwicklungen und Reviews über Graph-Matching Verfahren und Anwendungen im Mustererkennungsbereich sind dort zu finden.

Wir wollen nun einige formelle Definitionen zum Graph-Matching einführen.

Im Anschluß daran werden die klassische Algorithmen zur Lösung des Graph-Matching Problems vorgestellt.

Im Folgenden betrachten wir – unserer Aufgabenstellung gemäss – nur ungerichtete attributierte Graphen.

Definition 3.2 *(**Graphisomorphismus**)*
*Seien $G = (V, E, \mu, \nu)$ und $G' = (V', E', \mu', \nu')$ zwei Graphen. Ein **Graphisomorphismus** von G nach G' ist eine bijektive Abbildung $f : V \to V'$ mit den Eigenschaften*

1. *$\mu(v) = \mu'(f(v))$ für alle $v \in V$.*
2. *für jede Kante $e = (u, v) \in E$ gibt es eine Kante $e' = (f(u), f(v)) \in E'$ mit $\nu(e) = \nu'(e')$, und für jede Kante $e' = (u', v') \in E'$ gibt es eine Kante $e = (f^{-1}(u'), f^{-1}(v')) \in E$, so dass $\nu'(e') = \nu(e)$.*

G und G' heißen isomorph, in Zeichen $G \cong G'$.

Definition 3.3 *(**Subgraphisomorphismus**)*
*Eine injektive Abbildung $f : V \to V'$ heißt **Subgraphisomorphismus** von G nach G', falls es einen Subgraph $S \subseteq G'$ gibt, so dass f ein Graphisomorphismus von G nach S ist.*

Definition 3.4 *(**Maximal gemeinsamer Subgraph**)*
Gegeben seien zwei Graphen $G_1 = (V_1, E_1, \mu_1, \nu_1)$ und $G_2 = (V_2, E_2, \mu_2, \nu_2)$. Ein Graph $G = (V, E, \mu, \nu)$ heißt gemeinsamer Subgraph von G_1 und G_2 genau dann, wenn Subgraphisomorphismen $f_1 : V \to V_1$ und $f_2 : V \to V_2$ existieren.

*G heißt **maximal gemeinsamer Subgraph** (maximum common subgraph, MCS) von G_1 und G_2 genau dann, wenn es keinen anderen gemeinsamen Subgraph von G_1 und G_2 gibt, der mehr Ecken als G hat.*

3.4.2 Algorithmen für Graph Matching

Für das Graph Matching Problem gibt es bereits ein breites Spektrum von Algorithmen. Zu den Standard-Verfahren zählt der 1976 von Ullman vorgestellte Algorithmus zur Berechnung von Graph- und Subgraphisomorphismen [62]. Dieser arbeitet nach dem Backtracking Prinzip, das noch mit einem Forward-Checking kombiniert wird. Auch für die Substrukturen-Suche in chemischen Datenbanken wird er weit verbreitet eingesetzt und hat sich dabei als einer der effizientesten Algorithmen bewährt [5, 33]. Algorithmen zur Bestimmung maximal-gemeinsamer Subgraphen wurden auch in [17, 43] vorgestellt. Das erste der beiden Verfahren berechnet für zwei gegebene Graphen zunächst deren sogenannten *Verträglichkeitsgraph* (association graph) und bestimmt ihren maximal gemeinsamen Subgraph dann mittels *maximal-Clique Suche* im Verträglichkeitsgraphen. Das Verfahren

wurde im System TOPSIM (TOPological SIMilarity) eingesetzt [17]. Das zweite der beiden Verfahren – der Algorithmus von McGregor – ist eine Variante des Backtracking-Verfahrens. In [9] werden diese beiden Algorithmen auf zufällig generierten Graphen miteinander verglichen.

Die Anwendung eines der oben erwähnten Algorithmen garantiert, dass eine optimale Lösung für das jeweilige Problem gefunden wird, falls eine solche existiert. Da diese Probleme jedoch NP-vollständig sind, brauchen sie alle exponentielle Zeit. Neben diesen Algorithmen gibt es noch eine ganze Reihe suboptimaler Verfahren, die in polynomieller Zeit laufen. Dabei werden sehr unterschiedliche Methoden angewandt, wie zum Beispiel probabilistische Relaxation, neuronale Netze, genetische Algorithmen, usw. Solche Methoden können allerdings nicht garantieren, dass immer eine optimale Lösung gefunden wird. Umfangreiche Referenzen zu verschieden suboptimalen Verfahren können in [8, 12] gefunden werden. Da uns für die Rekonstruktion chemischer Strukturen eine optimale Lösung wichtig ist, haben wir suboptimale Algorithmen nicht betrachtet.

Wir werden nun im Folgenden zwei Standard-Verfahren für die Ermittlung eines Matchings vorstellen. Zwar werden wir sie zugunsten eines anderen Verfahrens nicht zur Anwendung bringen; sie sind jedoch die zumeist verwendeten Algorithmen in diesem Gebiet, so dass wir sie zur späteren Diskussion präsentieren.

Ullmans Algorithmus für Graph-/Subgraphisomorphismus-Suche

Der klassische Algorithmus für Graph- und Subgraphisomorphismen-Suche ist der *Ullman-Algorithmus* [62]. Dieser basiert auf dem Backtracking-Verfahren, das mit Forward-Checking kombiniert wird, um den Suchraum einzuschränken.

Seien zwei Graphen $G_1 = (V_1, E_1, \mu_1, \nu_1)$ und $G_2 = (V_2, E_2, \mu_2, \nu_2)$ gegeben. Gesucht ist ein Graph- oder Subgraphisomorphismus von G_1 nach G_2. Wir beginnen damit, dass wir eine Ecke $u_i \in V_1$ auf eine Ecke $v_i \in V_2$ abbilden, d.h. wir setzen $f(u_i) = v_i$, und prüfen nach, ob $\mu_1(u_i) = \mu_2(v_i)$ ist. Falls ja, definiert f einen Graphisomorphismus vom Subgraph $\{u_i\}$ von G_1 auf den Subgraph $\{v_i\}$ von G_2. Wir können f dann um die Abbildung einer Ecke u_{i+1} auf eine Ecke v_{i+1} erweitern. Im Allgemeinen kann eine Abbildung $f = \{u_i \mapsto v_i, \ldots, u_j \mapsto v_j\}$, die eine Subgraphisomorphismus repräsentiert, um eine Abbildung $u_{j+1} \mapsto v_{j+1}$ mit $u_{j+1} \neq u_k, v_{j+1} \neq v_k$ für alle $k < j+1$ erweitert werden, falls die folgende Bedingungen erfüllt sind:

1. $\mu_1(u_{j+1}) = \mu_2(v_{j+1})$,
2. für jede Kante $e_1 = (u_k, u_{j+1}) \in E_1$ mit $k \leq j+1$ muss es eine Kante $e_2 = (v_k, v_{j+1}) \in E_2$ mit $k < j+1$ geben, und es gilt $\nu_1(e_1) = \nu_2(e_2)$. Dies muss ebenso für die umgekehrte Richtung gelten.

Diese Bedingungen garantieren, dass $f = \{u_i \mapsto v_i, \ldots, u_{j+1} \mapsto v_{j+1}\}$ einen Subgraphisomorphismus von einem Subgraph H von G nach G_2 repräsen-

tiert, wobei H aus der Eckenmenge $\{u_i, \ldots, u_{j+1}\}$ sowie denjenigen Kanten zwischen diesen Ecken besteht. Falls an einer bestimmten Stelle die Ecke $u_j \in V_1$ nicht auf eine Ecke $v_j \in V_2$ abgebildet werden kann, geht der Algorithmus einen Schritt zur vorherigen Ecke u_{j-1} zurück und versucht u_{j-1} auf eine andere Ecke in V_2 abzubilden, die bisher noch nicht betrachtet wurde (Backtracking). Wenn anderseits eine Abbildung gefunden wird, deren Definitionsbereich alle Ecken von V_1 umfaßt, dann haben wir bereits einen Subgraphisomorphismus von G_1 nach G_2 vorliegen.

Eine Verbesserung der beschriebenen Prozedur bietet die Kombination mit einer Forward-Checking-Prozedur [62]. Das Grundprinzip dieser Prozedur besteht darin, für jede Abbildung $u_j \mapsto v_j$ zu testen, ob es mindestens eine weitere Abbildung von einer bisher noch nicht betrachteten Ecke $u_k \in V_1$ auf eine bestimmte Ecke $v_k \in V_2$ mit $k > j$ gibt, so dass die Kriterien für den Subgraphisomorphismus erfüllt bleiben. Falls eine Abbildung $u_j \mapsto v_j$ vorliegt, die zwar mit allen Abbildungen $u_i \mapsto v_i$ für $i < j$ konsistent ist, aber für die keine Abbildung auf eine Ecke $v_k \in V_2$ mit $k > j$ möglich ist, dann wird die Abbildung $u_j \mapsto v_j$ gleich abgelehnt, und der Algorithmus kehrt in den vorherigen Level zurück. Auf diese Weise kann man all die Zwischenschritte der Levels l mit $i < l < j$ sparen.

Die Zeitkomplexität des Ullmans Algorithmus hängt von den Größen der beiden Graphen G_1 und G_2 sowie von deren Eckenmarkierungen ab. Der beste Fall liegt vor, wenn alle Ecken unterschiedlich markiert sind und G_1 zu G_2 isomorph ist. In solch einem Fall kann jede Ecke aus G_1 auf genau eine Ecke in G_2 abgebildet werden und die Zeitkomplexität beträgt $O(n_1 n_2)$ für $|V_1| = n_1$, $|V_2| = n_2$. Falls alle Ecken in G_1 und G_2 nicht markiert und die Graphen sehr dicht sind, haben wir den schlechtesten Fall vorliegen, für den der Ullman Algorithmus $O(n_2^{n_1} n_1^2)$ Zeit benötigt. Falls k Modell-Graphen aus einer Bibliothek mit einem Eingabe-Graph G_I gematcht werden müssen, steigt die Zeitkomplexität linear mit der Zahl k an. Eine ausführlichere Analyse und Diskussion findet der interessierte Leser in [46, 62].

Berechnung eines maximal-gemeinsamen Subgraphen mittels maximaler Cliquen

Seien nun zwei Graphen $G_1 = (V_1, E_1, \mu_1, \nu_1)$ und $G_2 = (V_2, E_2, \mu_2, \nu_2)$ gegeben. Die Grundidee beim Graph-Matching mittels maximale-Clique-Suche besteht darin, zunächst jede Ecke $u \in V_1$ auf alle Ecken $v \in V_2$ abzubilden, falls $\mu_1(u) = \mu_2(v)$ gilt. Anschließend wird jede solche Abbildung als eine neue Ecke in einem sogenannten *Verträglichkeitsgraph* $G_A = (V_A, E_A)$ gespeichert. Zwei Ecken v_A und v'_A im Verträglichkeitsgraphen G_A, welche zwei Abbildungen $u \mapsto v$, $u' \mapsto v'$ für $u, u' \in V_1$ und $v, v' \in V_2$ darstellen, sind genau dann miteinander verbunden, wenn die entsprechenden Abbildungen *verträglich* sind. Sie sind verträglich miteinander, falls die betrachteten Eigenschaften – etwa Verbundenheit, Abstand und Winkel – für die Paare u, u'

und v, v' gleich sind, bzw. innerhalb definierter Toleranzschranken liegen. Somit repräsentiert die Abbildung $f : \{u \mapsto v, u' \mapsto v'\}$ einen Graphisomorphismus von einem Subgraph von G_1, dessen Eckenmenge $\{u, u'\}$ ist, auf einen Subgraph von G_2, bestehend aus der Eckenmenge $\{v, v'\}$. Genauer ausgedrückt bedeutet dies, dass jeder vollständige Teilgraph H_A von G_A, den man als *Clique* bezeichnet, einen gemeinsamen Subgraph von G_1 und G_2 darstellt. Aus diesem Zusammenhang folgt direkt, dass die maximale Clique in G_A dem maximal-gemeinsamen Subgraph von G_1 und G_2 entspricht; damit lässt sich die Suche nach einen maximal-gemeinsamen Subgraph auf die maximale-Clique-Suche zurückführen.

Wir geben nun die formelle Definition für den Verträglichkeitsgraph an. Anschließend werden die maximale-Clique-Suche und die Laufzeit des darauf basierenden Algorithmus für Graph-Matching kurz beschrieben. Für eine ausführlichere Behandlung des Algorithmus verweisen wir auf [44].

Definition 3.5 *(**Clique**)*

*Ein vollständiger Teilgraph H eines Graphen G wird als **Clique** bezeichnet. Eine Clique maximaler Ordnung heißt maximale Clique.*

Definition 3.6 *(**Verträglichkeitsgraph**)*

*Seien zwei Graphen $G_1 = (V_1, E_1, \mu_1, \nu_1)$ und $G_2 = (V_2, E_2, \mu_2, \nu_2)$ gegeben. Ein **Verträglichkeitsgraph** (association graph) für G_1 und G_2 ist ein nicht attributierter Graph $G_A = (V_A, E_A)$ mit $V_A \subseteq V_1 \times V_2$, $E_A \subseteq V_A \times V_A$ und*

1. *$V_A = \{(u, v) | u \in V_1, v \in V_2,$ und $\mu_1(u) = \mu_2(v)\}$*
2. *E_A besteht aus allen Kanten $e_A = (v_A, v'_A)$ mit $v_A = (u, v)$, $v'_A = (u', v')$, so dass das folgende Kriterium der Strukturerhaltung gilt:*
   (a) *$u \neq u'$ und $v \neq v'$*
   (b) *Falls es eine Kante $e = (u, u') \in E_1$ gibt, dann muss es eine Kante $e' = (v, v') \in E_2$ geben, und es gilt $\nu_1(e) = \nu_2(e')$;*
   (c) *Falls es keine Kante $e = (u, u') \in E_1$ gibt, dann darf es auch keine Kante $e' = (v, v') \in E_2$ geben.*

Ausgehend vom Verträglichkeitsgraphen $G_A = (V_A, E_A)$ für zwei Graphen G_1 und G_2, kann nun der maximal-gemeinsame Subgraph dieser beiden Graphen ermittelt werden, indem man eine Suche nach allen maximalen Cliquen in G_A durchführt. Ist $C = \{v_{A_1}, \ldots, v_{A_k}\}$ eine Eckenmenge, die eine Clique in G_A aufspannt, dann repräsentiert die Abbildung, die in den Ecken der Menge C kodiert ist, einen Graphisomorphismus von einem Subgraph S_1 von G_1 auf einen Subgraph S_2 von G_2 mit $S_1 = \{u | (u, v) \in C\}$ und $S_2 = \{v | (u, v) \in C\}$. Falls $S_1 = G_1$ gilt, dann haben wir einen Subgraphisomorphismus von G_1 nach G_2 gefunden. Gilt $S_2 = G_2$, so entspricht die

gefundene Clique einem Subgraphisomorphismus von G_2 auf G_1. Dieses Verfahren ebenso wie ein Verfahren zur Bestimmung aller maxmimalen Cliques in einem ungerichteten Graph wurde in [44] ausführlich beschrieben.

Die Zeitkomplexität dieses Algorithmus ist wie bei Ullmans Algorithmus im besten Fall $O(n_1 n_2)$ mit $|V_1| = n_1$, $|V_2| = n_2$. Für den schlechtesten Fall, d.h. wenn alle Ecken gleich markiert und beide Graphen vollständig sind, braucht der Algorithmus $O((n_1 n_2)^{n_1})$ [44].

Ullmans Algorithmus ist für die Subgraphisomorphismen Suche zwischen zwei unmarkierten Graphen recht effizient und eines der am häufigsten eingesetzten Verfahren bei der Suche nach chemischen Substrukturen. Auch die Suche nach maximal-gemeinsamen Subgraph wird hierzu oft zur Anwendung gebracht. Trotzdem haben wir uns aus verschiedenen Gründen entschieden, das im Folgenden detailliert vorgestellte Verfahren zu benutzen.

In [44] wurden experimentelle Ergebnisse präsentiert, die zeigen, dass beim Zusammenwirken verschiedener Faktoren ein Verfahren, das Dekompositionsnetzwerke benutzt, Ullmans Verfahren überlegen war. Faktoren, die sich als vorteilhaft für Dekompositionsnetzwerke erwiesen, waren dabei: mehr als zwei Graphen miteinander zu vergleichen; markierte Graphen (ab fünf unterschiedlichen Ecken- oder Kantenmarkierungen); vor allem eine große Zahl gemeinsamer Subgraphen in den zu matchenden Graphen.

In unserem Falle haben wir eine nicht kleine Bibliothek von Modell-Graphen, die zur Erkennung eines unbekannten Eingabe-Graphen alle mit diesem gematcht werden müssen. Die Anzahl der unterschiedlichen Markierungen liegt oft unter fünf, wenn nicht zwischen unterschiedlichen Atomsymbolen und Bindungsarten unterschieden wird. Andererseits sind in den Modell-Graphen viele gemeinsame Strukturen vorzufinden. Alles in allem erwarten wir deswegen, dass die auf Graph-Dekompositionen basierte Subgraphisomorphismen-Suche für uns die günstigere Wahl ist. Es wäre jedoch interessant, andere Methoden ebenfalls zu implementieren, um einen konkreten Vergleich bezüglich unserer Anwendung ziehen zu können.

3.4.3 Modell-Graphen

Wir beschäftigen uns nun mit der Frage, welche Modell-Graphen wir in unserer Datenbank ablegen wollen, um sie zur Laufzeit mit der Eingabe zu matchen. Wie bereits bemerkt, lässt sich eine chemische Struktur in der Regel aus vielen kleineren Strukturfragmenten zusammensetzen. Manche dieser Strukturfragmente kommen wiederholt in einer Struktur vor, oder sie finden sich in verschiedenen Strukturen wieder. Unterschiedliche Studien haben versucht, einen Zusammenhang zwischen häufig vorkommenden Substituenten chemischer Strukturen von Medikamenten und deren Wirkungen zu ziehen [19]. Dabei wurde einerseits festgestellt, dass es wohl eine ungeheure Anzahl von Substituenten gibt; andererseits scheint nur eine recht kleine Menge von Substituenten häufiger vorzukommen. Abbildung 3.8 zeigt ein Ergebnis einer

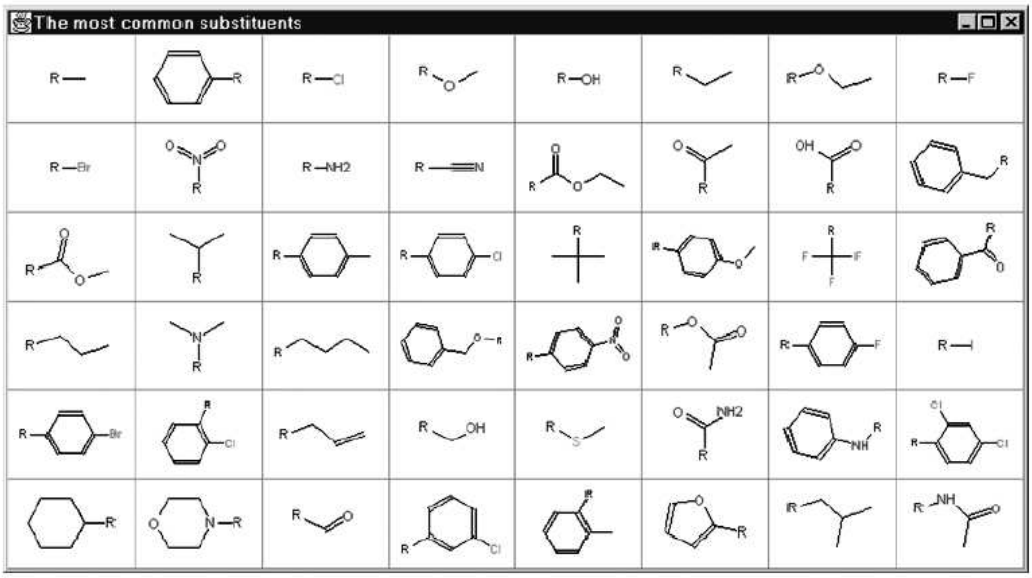


Abbildung 3.9: Häufig vorkommende Substituenten [19]. Der Buchstabe *„R“* steht für eine Restgruppe.

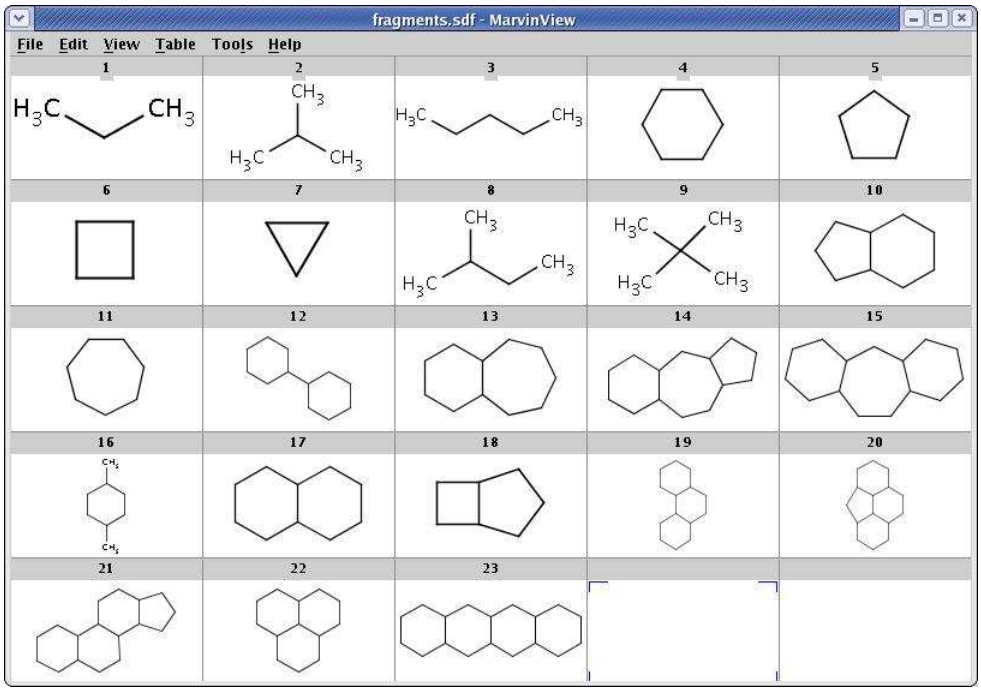


Abbildung 3.10: Strukturfragmente, die in der vorliegenden Arbeit als Modelle zur Erkennung einer unbekannten Struktur dienen.

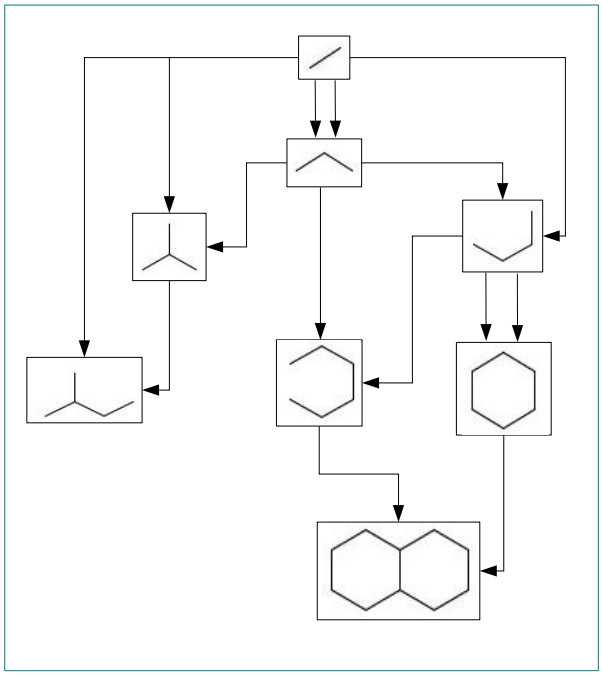


Abbildung 3.11: Repräsentation der Modell-Strukturfragmente in einem Dekompositionsnetzwerk. (Die Strukturfragmente werden eigentlich als Kantengraph gespeichert. Einfachheitshalber sind sie hier in der herkömlichen 2D-Zeichnung dargestellt.)

dieser Studien.

Von den Ergebnissen der genannten Studien und Beobachtungen der Strukturen im gegebenen Testkorpus ausgehend, haben wir eine Menge von Strukturfragmente zusammengestellt, die als Modelle zur Erkennung einer chemischen Struktur in einem unbekannten Bild dienen. Dabei haben wir die häufig vorkommenden Strukturfragmente weiter zusammengefasst, indem wir Mehrfachbindungen und Heteroatome nicht differenzieren. Damit erhalten wir eine kompakte Bibliothek, von der wir erwarten, dass sie den Großteil der auftretenden Strukturen erzeugen kann.

3.4.4 Repräsentation der Modell-Strukturfragmente

Voraussetzung für einen Vergleich zweier Graphen ist natürlich, dass diese beide Graphen vom gleichen Typ sind. Die Modell-Strukturfragmente werden daher auch in Form von Kantengraphen gespeichert. Zur Erkennung einer unbekannten Struktur könnten wir nun der Reihe nach alle vordefinierten Modell-Fragmente auf ihr Vorkommen in der Eingabe überprüfen. Aus den so gefundenen Teilfragmenten lässt sich dann die unbekannte Struktur rekonstruieren. Die Folge dieses Vorgehens wäre jedoch, dass wir jeden

Subgraph eines Modell-Strukturfragments mit dem Eingabe-Graph matchen müssten. Bei einer größeren Menge von Modellen würde dies recht hohe Kosten verursachen. Da die Modell-Strukturfragmente viele gemeinsame Teile aufweisen, lässt sich viel Arbeit sparen, wenn wir diese gemeinsamen Teile der Modell-Strukturfragmente aufspüren und sie dann nur einmal mit der unbekannten Struktur vergleichen. Für diesen Zweck setzen wir das sogenannte Dekompositionsnetzwerk zur Speicherung der vordefinierten Strukturfragmente (Modell-Graphen) ein.

Dieses Modellnetzwerk wird in einem off-line Prozess generiert. Wie gesagt ist die Hauptidee, gemeinsame Teilstrukturen in den Strukturfragmenten zu finden und sie einmalig im Netzwerk abzuspeichern. Zu diesem Zweck werden die Strukturfragmente in kleinere Komponenten zerlegt. Die kleinste vorkommende Komponente ist ein Liniensegment, welches einer einzelnen chemischen Bindung entspricht. Jede Komponente wird durch einen Knoten im Netzwerk repräsentiert. Dabei kann jeder Knoten höchstens zwei Vorgänger haben, da ein (Sub-)Graph jeweils in höchstens zwei Teilkomponenten zerlegt wird. Er kann jedoch beliebig viele Nachfolger haben, da ein Subgraph in beliebig vielen Obergraphen vorkommen kann. Falls eine bestimmte Teilstruktur in mehreren Strukturfragmente vorkommt, wird sie in nur einem Knoten des Netzwerks gespeichert. Dies führt zu einer kompakten Darstellung aller Strukturfragmente. Abbildung 3.11 illustriert den Aufbau des Dekompositionsnetzwerks an einer kleinen Menge von Modell-Strukturfragmenten.

Wir werden nun die formelle Definition für die Dekomposition einer Menge von Modell-Graphen geben.

Definition 3.7 ***(Dekomposition)***

Sei $M = \{G_1, \ldots, G_l\}$ eine Menge von Modell-Graphen. Ein ***Dekomposition*** *von M, in Zeichen $D(M)$, ist eine endliche Menge von Quadrupeln (G, G', G'', E) mit folgenden Eigenschaften:*

1. *G, G′, G″ sind Graphen, wobei $G' \subset G$ und $G'' \subset G$ gilt.*
2. *E ist eine Kantenmenge mit $G = G' \cup_E G''$.*
3. *Es gibt für jeden Graph G_i ein Quadrupel $(G, G', G'', E) \in D(M)$ mit $G = G_i, i = 1, \ldots, l$.*
4. *für jedes Quadrupel $(G, G', G'', E) \in D(M)$ gilt:*
   (a) *Es existiert kein anderes Quadrupel $(G_1, G'_1, G''_1, E_1) \in D(M)$ mit $G = G_1$.*
   (b) *Falls G' aus mehr als einer Ecke besteht, so gibt es ein Quadrupel $(G_1, G'_1, G''_1, E_1) \in D(M)$, so dass $G' = G_1$ gilt.*
   (c) *Falls G'' aus mehr als einer Ecke besteht, so gibt es ein Quadrupel $(G_2, G'_2, G''_2, G_2) \in D(M)$, so dass $G'' = G_2$ gilt.*

(d) *Falls G' nur aus einer Ecke besteht, so gibt es kein Quadrupel $(G_1, G'_1, G''_1, E_1) \in D(M)$, so dass $G' = G_1$ gilt.*

(e) *Falls G'' nur aus einer Ecke besteht, so gibt es kein Quadrupel $(G_2, G'_2, G''_2, G_2) \in D(M)$, so dass $G'' = G_2$ gilt.*

Informell gesprochen, ist eine Dekomposition eine *rekursive Partitionierung* der Modell-Graphen in kleinere Subgraphen, jeweils von einem ganzen Modell-Graph ausgehend, bis der triviale Graph, d.h. ein Graph mit nur einer Ecke, vorliegt. In einem Quadrupel (G, G', G'', E) ist das Ergebnis einer solchen Partitionierung gespeichert. G ist der Graph, der zerlegt werden sollte. G' und G'' sind die zerlegten Subgraphen von G, und E stellt diejenige Kantenmenge dar, die G' und G'' in G verbindet.

Die Bedingung 3 garantiert, dass jeder Modell-Graph zerlegt wird. Die *Eindeutigkeit* der Dekomposition folgt aus der Bedingung 4.a). Durch die darauf folgende Bedingungen wird die *Vollständigkeit* der Modell-Graphen-Dekomposition gewährleistet.

Es ist offensichtlich, dass es viele verschiedene Möglichkeiten gibt, eine gegebene Menge von Modell-Graphen zu zerlegen. Man kann nun verschiedene Dekompositionen als optimal definieren; zum Beispiel diejenige, welche eine minimale Anzahl von Quadrupeln aufweist, oder jene, bei der der größte Subgraph aller Modelle durch ein Quadrupel (G, G', G'', E) repräsentiert wird. Die Bestimmung einer solchen optimalen Dekomposition kostet uns allerdings im schlimmsten Fall exponentielle Zeit [13]. Für diese Arbeit setzen wir daher die im Folgenden beschriebene Heuristik ein:

Sei $M = \{G_1, \ldots, G_l\}$ eine Menge von Modell-Graphen, die zerlegt werden soll. Betrachte nun nacheinander alle Graphen $G_i \in M$ und zerlege sie unter Berücksichtigung folgender Regeln:

Bevor ein Modell-Graph G zerlegt und in das Dekompositionsnetzwerk D eingefügt wird, suchen wir zunächst nach dem größten Subgraph $S_{\max}$ von G, der bereits in D vorhanden ist. Dann zerlegen wir G in zwei Subgraphen $S_{\max}$ und $(G - S_{\max})$

- Falls $S_{\max}$ isomorph zu G ist, ist G bereits in D repräsentiert und der Algorithmus stoppt.

- Falls es keinen solchen Subgraph $S_{\max}$ von G in D gibt oder aber ein $S_{\max}$ gefunden wird, der aus nur einer Ecke besteht während G mehr als drei Ecken hat, so teilen wir G in zwei etwa gleich große Subgraphen G' und G''. Auf diese Weise zerlegen wir dann die beiden Subgraphen von G rekursiv in kleinere Subgraphen, bis ein Subgraph mit nur einer Ecke vorliegt.

Die beschriebene Strategie gewährleistet, dass das Dekompositionsnetzwerk relativ balanciert bleibt. Die Tatsache, dass sie nicht immer eine optimale Dekomposition liefert, hat kaum Einfluß auf die Laufzeit der Matching-

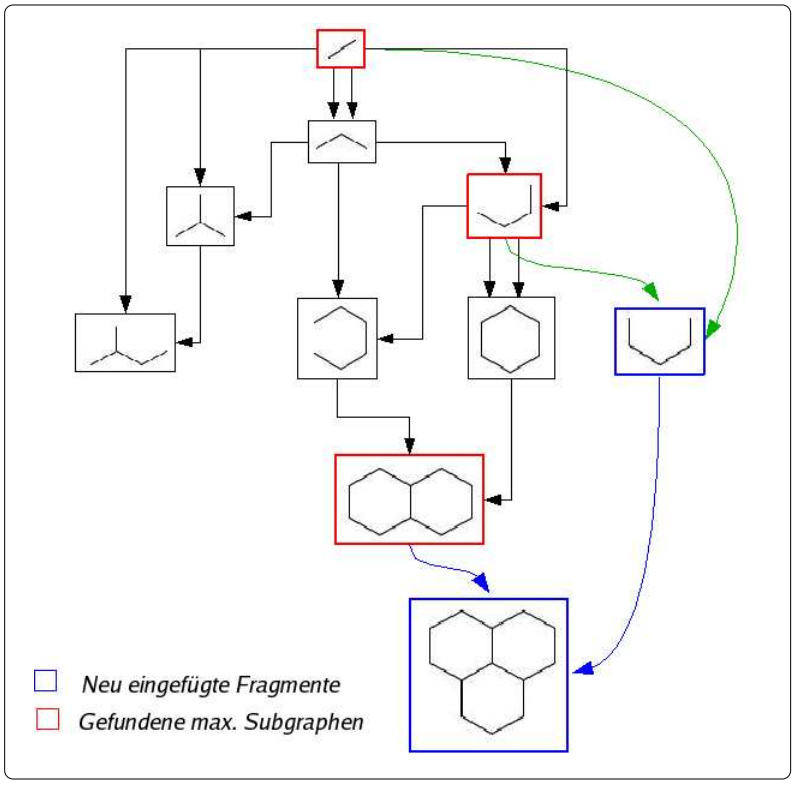


Abbildung 3.12: Einfügen eines neuen Modells G in das Dekompositionsnetzwerk. Falls bereits ein maximaler Subgraph S_{max} von G im Netzwerk vorhanden ist, wird G in S_{max} und $G - S_{max}$ zerlegt. Wenn die Subgraphen von G selbst neu im Netzwerk sind, werden sie auf die gleiche Weise rekursiv zerlegt.

Prozedur [44]. Sie bringt außerdem als wichtigen Vorteil mit, dass das Netzwerk *inkrementell erweiterbar* ist. Man kann jederzeit einen neuen Modell-Graph in das Netzwerk einfügen, ohne dass man alle vorhandenen Modell-Graphen neu zerlegen muss, um zum Beispiel das Kriterium der Optimalität zu erfüllen. Abbildung 3.12 illustriert das Einfügen eines neuen Modell-Graphen in ein bereits vorhandenes Netzwerk.

DEKOMPOSITION-ALGORITHMUS, HAUPTPROZEDUR

▷ Jeder Modell-Graph $G_i \in M$ wird nacheinander durch den Aufruf *Decompose* in kleineren Subgraphen zerlegt.

DECOMPOSITION(M):

EINGABE:

M: Menge der Modell-Graphen $M = \{G_1, \ldots, G_l\}$.

AUSGABE:

D: Ein Dekompositionsnetzwerk für alle Modell-Graphen.

ALGORITHMUS:

(1) $D \leftarrow \emptyset$
(2) **for all** $G_i \in M$ **do**
(3) $D \leftarrow$ DECOMPOSE(G_i, D) %Algorithmus 3.3
(4) **end for**

Algorithmus 3.2: Alle Modell-Graphen werden nacheinander in kleinere Subgraphen zerlegt.

Die Hauptprozedur des Dekompositionsalgorithmus ist in Algorithmus 3.2 gegeben. Für eine Menge $M = \{G_1, \ldots, G_l\}$ von Modell-Graphen ruft sie die Prozedur *Decompose* (Algorithmus 3.3) l mal auf, um jeden dieser Graphen zu zerlegen. Es ist zu beachten, dass für die *if*-Abfrage in der Zeile 5 bereits einen Algorithmus zur Subgraphisomorphismen-Suche erforderlich ist. Hierfür haben wir den im nächsten Abschnitt vorgestellten Algorithmus verwendet. Die Lauzeit für die Dekomposition ist damit von dem Algorithmus zur Subgraphisomorphismen-Suche abhängig. Diese werden wir im nächsten Abschnitt betrachten.

DEKOMPOSITION-ALGORITHMUS

▷ Ein Modell-Graph G wird rekursiv in kleinere Subgraphen zerlegt. Dabei wird zunächst geprüft, ob bereits ein maximaler Subgraph S_{max} von G im Dekompositionsnetzwerk vorhanden ist. Falls ja, wird G in S_{max} und $G - S_{max}$ zerlegt. Ansonsten ist G in zwei etwa gleich große Subgraphen zu teilen.

DECOMPOSE(G, D):

EINGABE:

G:	Ein Graph $G = (V, E, \mu, \nu)$ mit $n := \|V\|$ und $m := \|E\|$.
D:	Ein Dekompositionsnetzwerk.

AUSGABE:

D':	Ein Dekompositionsnetzwerk, in dem G durch einen neuen Knoten repräsentiert wird (falls er nicht bereits im Netzwerk gespeichert ist).

ALGORITHMUS:

(1) $S_{max} \leftarrow \emptyset$
(2) $D' \leftarrow D$
(3) **if** $(n = 1)$ **then** exit
% *Finde den maximalen Subgraph S_{max} von G in D*
(4) **for all** $(G_i, G_i', G_i'', E_i) \in D$ **do**
(5) **if** (G_i ist ein Subgraph von G und $|V_{max}| < |V_i|$) **then**
(6) $S_{max} \leftarrow G_i$
(7) **end if**
(8) **end for**
(9) **if** $(S_{max} \simeq G)$ **then** exit
% *Es wurde kein maximaler Subgraph S_{max} von G in D gefunden, oder S_{max} ist zu klein*
(10) **if** (($S_{max} = \emptyset$ und $n > 1$) oder ($|V_{max}| = 1$ und $n > 3$)) **then**
(11) Teile G in zwei etwa gleich große Subgraphen S_{max} und $G - S_{max}$
(12) $D' \leftarrow$ DECOMPOSE(S_{max})
(13) **end if**
(14) $D' \leftarrow$ DECOMPOSE$(G - S_{max})$
(15) $D' \leftarrow D' \cup \{(G, S_{max}, G - S_{max}, B)\}$, wobei B diejenigen Kanten in G sind, die S_{max} und $G - S_{max}$ verbindet.

Algorithmus 3.3: Zerlegung eines Modell-Graphen in kleinere Subgraphen.

3.4.5 Subgraphisomorphismen-Suche mittels Graph-Dekomposition

Die Repräsentation von Modell-Graphen als ein Dekompositionsnetzwerk, wie im letztem Abschnitt beschrieben, bildet die Basis für den von uns verwendeten Algorithmus zur Bestimmung von Subgraphisomorphismen zwischen diesen Modell-Graphen und dem Eingabe-Graph.

Anstatt für jeden Modell-Graph ein Graph-Matching mit dem Eingabe-Graph G_I durchzuführen, werden wir nun alle Quadrupel $(G, G', G'', E) \in D$ durchlaufen und nach Subgraphisomorphismen vom Graph G mit G_I suchen. Wir beginnen mit demjenigen Quadrupel (G, G', G'', E), bei dem G aus nur einer Ecke besteht (G' und G'' sind beide in diesem Fall leer). Die gefundenen Subgraphisomorphismen werden dann in den nächsten Schritten eingesetzt, um einen größeren Subgraphisomorphismus zu bilden. Angenommen wir betrachten gerade ein Tupel (G, G', G'', E), und es sind bereits Subgraphisomorphismen f' und f'' zwischen G' bzw. G'' und G_I gefunden worden. Damit f' und f'' nun zu einem größeren Subgraphisomorphismus f von G auf G_I kombiniert werden können, müssen zwei Bedingungen erfüllt sein:

- Erstens müssen die Bilder von f' und f'' disjunkt sein. Dies ist erforderlich, damit die aus ihnen kombinierte Funktion f auch wieder injektiv ist.
- Zweitens muss sicher gestellt sein, dass alle Kanten aus E, die G' mit G'' in G verbinden, korrekt auf Kanten von G_I abgebildet werden und umgekehrt.

Formell bedeutet dies:

1. $f'(V_{G'}) \cap f''(V_{G''}) = \varnothing$.
2. Für jede Kante $e = (v_1, v_2) \in E$ mit $v_1 \in G'$ und $v_2 \in G''$ gibt es eine Kante $e_I = (f'(v_1), f''(v_2)) \in E_I$ mit $\nu(e) = \nu_I(e_I)$, und
   für jede Kante $e_I = (v_I, v'_I) \in E_I$ mit $v_I = f'(v_1)$ und $v'_I = f''(v_2)$ gibt es eine Kante $e = (f'^{-1}(v_I), f''^{-1}(v'_I)) \in E$ mit $\nu_I(e_I) = \nu(e)$.

Um uns merken zu können, welche Subgraphen des Netzwerks bereits mit dem Eingabe-Graph gematch wurden, versehen wir jeden dieser Graphen im Netzwerk mit einer Markierung. Am Anfang sind sie alle mit *unsolved* markiert. Sobald die Suche nach Subgraphisomorphismen von einem Subgraph im Netzwerk mit der Eingabe durchgeführt worden ist, wird dieser Subgraph, abhängig vom Suchergebnis, entweder als *alive* oder *dead* markiert. Jeder Graph G des Quadrupels $(G, G', G'', E) \in D$, der noch mit *unsolved*, während seine beiden Subgraphen G' und G'' bereits als *alive* markiert sind, kann mit der Eingabe gematcht werden, falls die oben genannten Bedingungen erfüllt sind.

MATCHING-ALGORITHMUS

▷ Beginnend mit dem kleinsten Graph, der aus nur einer Ecke besteht, wird inkrementell nach Subgraphisomorphismen von einem Graphen G im Dekompositionsnetzwerk nach dem Eingabe-Graphen G_I gesucht. Sind z.B. für ein Quadrupel $(G, G', G'', B) \in D$ bereits Subgraphisomorphismen von G' und G'' nach G_I gefunden worden, so wird die Prozedur *Merge* aufgerufen, um sie – unter den dort genannten Bedingungen – zu einem Subgraphisomorphismus von G nach G_I zu kombinieren.

MATCH(D, G_I):

EINGABE:

D:	$D = \{(S_1, S_1', S_1'', B_1), \ldots, (S_N, S_N', S_N'', B_N)\}$.
G_I:	Ein Graph $G_I = (V_I, E_I, \mu_I, \nu_I)$.

AUSGABE:

F:	Alle gefundenen Subgraphisomorphismen von $S_i \in D$ nach G_I.

ALGORITHMUS:

```
(1)  F ← ∅
(2)  P ← ⋃_{i=1}^{N} {S_i, S_i', S_i''}   %Menge aller Graphen, die in D vorkommen
(3)  for all S ∈ P do
(4)      processed[S] ← unsolved
(5)  end for
(6)  for all S = (V_S, E_S, μ_S, ν_S) ∈ P mit |V_S| = 1 do
(7)      F_S ← MATCH_VERTEX(v, G_I), wobei {v} = V_S
(8)      if (F_S = ∅) then
(9)          processed[S] ← dead else
(10)         processed[S] ← alive und assoziere F_S mit S
(11)     end if
(12) end for
(13) while (∃ S ∈ P mit processed[S] = unsolved) do
(14)     if (∃ (S, S_1, S_2, B) ∈ D mit processed[S] = unsolved,
         processed[S_1] = alive und processed[S_2] = alive) then
(15)         Seien F_1 und F_2 jeweils die Menge der Subgraphisomorphismen,
             die mit S_1 bzw. S_2 assoziert sind
             F_S ← MERGE(S_1, F_1, S_2, F_2, G_I)
(16)         if (F = ∅) then
(17)             processed[S] ← dead else
(18)             processed[S] ← alive und assoziere F_S mit S
(19)             F ← F ∪ F_S
(20)         end if
(21)     end if
(22) end while
(23) return F
```

Algorithmus 3.4: Subgraphisomorphismen-Suche zwischen Graphen in einem Dekompositionsnetzwerk und dem Eingabe-Graph G_I.

MATCHING-ALGORITHMUS

▷ Jedes Attribut einer Ecke v_I im Eingabe-Graphen wird mit dem Attribut einer Ecke v eines Modell-Graphen verglichen, um einen Subgraphisomorphismus von v nach G_I zu finden.

MATCH_VERTEX(v, G_I):

EINGABE:

v:	Eine Ecke mit dem Attribut $l := \mu(v)$.
G_I:	Der Eingabe-Graph $G_I = (V_I, E_I, \mu_I, \nu_I)$.

AUSGABE:

F:	Die Menge aller Subgraphisomorphismen von v nach G_I.

ALGORITHMUS:

(1) $F \leftarrow \emptyset$
(2) **for all** $v_I \in V_I$ **do**
(3) **if** $(l = \mu_I(v_I))$ **then**
(4) $f(v) = v_I$ und $F \leftarrow F \cup \{f\}$
(5) **end if**
(6) **end for**

Algorithmus 3.5: Subgraphisomorphismen-Suche von einer Ecke v auf den Eingabe-Graphen G_I.

Algorithmus 3.4 beschreibt die Suche nach Subgraphisomorphismen von einer Menge von Modell-Graphen mit einem Eingabe-Graph. Bei der Initialisierung werden zunächst alle Graphen im Netzwerks mit *unsolved* markiert. Danach wird mit der Prozedur *Match_Vertex* (Algorithmus 3.5) versucht, all diejenige Graphen mit der Eingabe zu matchen, die nur aus einer Ecke bestehen. Dann werden die entsprechenden Graphen des Netzwerks jeweils mit der gefundenen Menge von Subgraphisomorphismen assoziert und deren Markierungen aktualisiert. In den darauf folgenden Schritten wird jedes Quadrupel $(S, S', S'', E) \in D$ betrachtet, bei dem S noch als *unsolved* und S' sowie S'' beide als *alive* markiert sind. Durch den Aufruf der Prozedur *Merge* (Algorithms 3.6) in Zeile 15 wird für jedes solche Quadrupel überprüft, ob die berechneten Subgraphisomorphismen von S' bzw. S'' auf den Eingabe-Graph G_I zu Subgraphisomorphismen von S auf G_I erweiterbar sind. Falls nicht, wird S als *dead* markiert. In der Folge können keine derjenigen Graphen im Netzwerk auf G_I gematcht werden, die S als Subgraph enthalten. Ansonsten wird S selber als *alive* markiert und die bis dahin berechneten Menge der Subgraphisomorphismen wird um die Subgraphisomorphismen

MATCHING-ALGORITHMUS

▷ Zwei Subgraphisomorphismen f_1 und f_2 von S_1 bzw. S_2 nach G_I werden zu einem größeren Subgraphisomorphismus f von $S_1 \cup_B S_2$ nach G_I kombiniert, falls die Bilder von f_1 und f_2 disjunkt sind, und die Kanten einer Menge B, die S_1 und S_2 verbinden, „richtig“ auf die Kanten von G_I abgebildet werden können.

MERGE$(S_1, F_1, S_2, F_2, B, G_I)$:

EINGABE:

S_1:	Ein Graph $S_1 = (V_1, E_1, \mu_1, \nu_1)$.
F_1:	Menge der Subgraphisomorphismen von S_1 nach G_I.
S_2:	Ein Graph $S_2 = (V_2, E_2, \mu_2, \nu_2)$.
F_2:	Menge der Subgraphisomorphismen von S_2 nach G_I.
B:	Eine Kantenmenge mit $e = (u, v) \in B$ und $u \in V_1, v \in V_2$, oder $u \in V_2, v \in V_1$.

AUSGABE:

F:	Menge aller Subgraphisomorphismen von $S_1 \cup_B S_2$ nach G_I.

ALGORITHMUS:

(1) $F \leftarrow \emptyset$
(2) **for all** Paare (f_1, f_2), $f_1 \in F_1$ und $f_2 \in F_2$ **do**
(3) Teste die Bedingungen (a) und (b)

(a) $f_1(V_1) \cap f_2(V_2) = \emptyset$

(b) Für jede Kante $e = (v_1, v_2) \in B$ gibt es eine Kante $e_I = (f'(v_1), f''(v_2)) \in E_I$ mit $\nu(e) = \nu_I(e_I)$, und für jede Kante $e_I = (v_I, v_I') \in E_I$ zwischen $f'(V_1)$ und $f''(V_2)$ gibt es eine Kante $e = (f'^{-1}(v_I), f''^{-1}(v_I')) \in E$ mit $\nu_I(e_I) = \nu(e)$.

(4) **if** beide Bedingungen (a) und (b) erfüllt sind **then**
(5) Sei ein Subgraphisomorphismus $f : V_1 \cup V_2 \to V_I$ von $S_1 \cup_B S_2$ nach G_I wie folgt definiert:

$$f(v) = \begin{cases} f_1(v) & falls\ v \in V_1 \\ f_2(v) & falls\ v \in V_2 \end{cases}$$

Füge f zu F hinzu, d.h. $F \leftarrow F \cup \{f\}$
(6) **end if**
(7) **end for**
(8) **return** F

Algorithmus 3.6: Kombiniere zwei Subgraphisomorphismen f_1 und f_2 zu einem größeren Subgraphisomorphismus f.

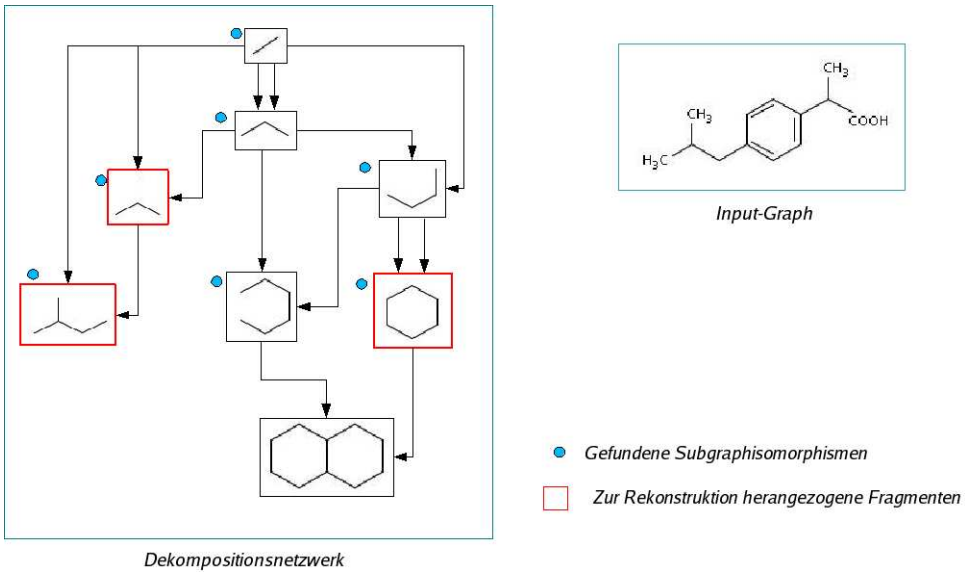


Abbildung 3.13: Beispiel für die Subgraphisomorphismen-Suche.

von S auf die Eingabe vergrößert. Danach fährt die Suche genauso weiter fort, bis entweder alle Quadrupel des Netzwerks untersucht worden sind, oder es kein Quadrupel $(S, S', S'', E) \in D$ mehr gibt, bei dem S noch als *unsolved* und S' sowie S'' beide als *alive* markiert sind.

Die Ausgabe des Algorithmus 3.4 ist eine Menge $F = \{F_1, \ldots, F_k\}$ mit $|F| \leq |D|$, $F_i \neq \emptyset$. Dabei ist jedes Element $F_i \in F$ wiederum eine Menge. Sie enthält alle gefundene Subgraphisomorphismen eines Graphen G_i im Dekompositionsnetzwerk mit dem Eingabe-Graph G_I. D.h. $F_i \in F = \{f_1, \ldots, f_l\}$ mit $f_i : V \to V_I'$ ist ein Subgraphisomorphismus für ein $G \in D$ und $V_I' \subseteq V_I$. Die Elemente F_i der Menge F sind dabei bezüglich der Größe der Graphen G in aufsteigender Reihenfolge sortiert.

Falls alle Modell-Graphen komplett unterschiedlich sind, ist die Laufzeit des Algorithmus 3.4 dieselbe wie bei Ullmans Algorithmus. Sie beträgt im besten Fall $O(knn_I)$, wobei k die Anzahl der Modelle im Netzwerk, n die Eckenzahl eines Modell-Graphen und n_I die Eckenzahl des Eingabe-Graphen bezeichnet. Im schlechtesten Fall ist sie $O(kn_I{}^n n^2)$. Im anderen Extremfall, wenn alle Modelle identisch sind, wird die Laufzeit unabhängig von der Anzahl der Modelle. Sie beträgt $O(nn_I)$ im besten Fall, und $O(n_I{}^n n^2)$ im schlechtesten Fall [44, 46].

An dieser Stelle möchten wir noch die Laufzeit des Algorithmus 3.2 angeben, welcher aus einer Menge von Modell-Graphen ein Dekompositionsnetzwerk erzeugt. Für eine Menge von Modell-Graphen $M = \{G_1, \ldots, G_k\}$ ruft dieser Algorithmus die Prozedur *Decompose* (Algorithmus 3.3) k mal auf, um jedes Modell $G_i \in M$ zu zerlegen. In der Prozedur *Decompose* wird jeweils

zunächst nach einem maximalen Subgraph S_{max} von G_i gesucht, der bereits im aktuellen Netzwerk vorhanden ist. Falls die Suche erfolgreich ist, wird G_i in S_{max} und den Differenzgraph $G_i - S_{max}$ zerlegt. Letzterer wird dann wiederum rekursiv durch die Prozedur *Decompose* zerlegt. Die Laufzeit des Algorithms zur Erzeugung eines Netzwerks ist damit abhängig von der Anzahl der Aufrufe der Prozedur *Decompose* und der Komplexität des Algorithmus zur Subgraphisomorphismen-Suche. Der beste Fall des Algorithmus liegt also vor, wenn alle Modelle identisch und ihre Ecken unterschiedlich markiert sind. In diesem Fall ist die Prozedur *Decompose* durch $O(n^2)$ beschränkt – n ist dabei die Eckenzahl in G_i. Da bei der Zerlegung eines Modells G_i die Prozedur *Decompose* $O(n)$ mal rekursiv aufgerufen werden, benötigt diese Zerlegung $O(n^3)$ Zeit. Die Erzeugung eines Dekompositionsnetzwerks für k Modelle erfordert im besten Fall somit $O(kn^3)$ Zeit. Entsprechend der worstcase Komplexität der Subgraphisomorphismen-Suche beträgt ihre Laufzeit im schlechtesten Fall $O(k^2 n^{n+3})$ [44, 46].

Aus den berechneten Subgraphisomorphismen soll nun die unbekannte chemische Struktur im Eingabe-Bild rekonstruiert werden. Dieser Aufgabe werden wir uns im nächsten Abschnitt widmen.

3.5 Rekonstruktion chemischer Strukturen

Die Subgraphisomophismen-Suche mittels Graph-Dekomposition, wie im letzten Abschnitt beschrieben, liefert uns eine Menge $F = \{F_1, \ldots, F_k\}$ mit $|F| \leq |D|$, $F_i \neq \emptyset$, und $F_i = \{f_1, \ldots, f_l\}$ mit $f_i : V \rightarrow V_I', V_I' \subseteq V_I$. Diese Menge gibt an, welche Graphen des Netzwerks als Subgraphen im Eingabe-Graphen G_I enthalten sind. Es wird ferner durch jede Abbildung $f_i : V \rightarrow V_I'$ einer Menge $F_i \in F$ genau spezifiziert, welche Eckenmenge $V_I' \subseteq V_I$ der Eckenmenge V von G entspricht, so dass der durch V_I' induzierte Subgraph die gleiche Struktur wie G darstellt. Da wir G_I als Kantengraph des Molekulargraphen repräsentiert haben (siehe Abschnitt 3.3.2), wird nun durch jede Zuordnung $f_i(v) = v_I$ eine Kante des Molekulargraphen (d.h. eine Bindung sowie die beiden zugehörigen Atome der zu rekonstruierenden Struktur) abgedeckt.

Algorithmus 3.7 zeigt die Prozedur der Rekonstruktion an. Der Algorithmus benötigt als Eingabe außer der Menge der berechneten Subgraphisomorphismen und dem Eingabe-Graph G_I noch eine Liste $L_A = \{a_1, \ldots, a_n\}$, welche während der Vorverarbeitung erstellt wurde. In jedem Element $a_i \in L_A$ sind Informationen wie Atomsymbole und Lokation des entsprechenden Atoms im Bild gespeichert.

Zur Rekonstruktion werden nun nacheinander die nach Größe der zugehörigen Graphen sortierten Mengen $F_i \in F$ betrachtet. Kandidaten für die Rekonstruktion sind all jene Subgraphisomorphismen $f_i : V \rightarrow V_I'$ aus F_i, deren Bilder bisher noch nicht zur Rekonstruktion verwendet wurden.

ALGORITHMUS ZUR REKONSTRUKTION

▷ Zur Rekonstruktion werden die berechneten Subgraphisomorphismen nacheinander, jeweils der Größe nach, betrachtet. Jede Ecke v_I in den Bilder eines Subgraphisomorphismus entspricht einer Kante $e = (u, v)$ im zu rekonstruierenden Molekulargraph G_{Mol}. Sei $f : G \rightarrow G_I$ der aktuelle Subgraphisomorphismus. Wurden die Bilder von f bisher noch nicht zur Rekonstruktion benutzt, so füge für jede Bild-Ecke v_I aus f die entsprechende Molekularecken (falls sie nicht bereits vorhanden sind) und die sie verbindende Kante in G_{Mol} ein.

EINGABE:

F:	Menge der gefundenen Subgraphisomorphismen, F ist bzgl. der Subgraphismorphismengröße aufsteigend sortiert.
G_I:	Eingabe-Graph $G_i = (V_I, E_I, \mu_I, \nu_I)$.
L_A:	Vorläufige Liste der Atome im Eingabe-Bild.

AUSGABE:

G_{Mol}:	Molekulargraph der rekonstruierten Struktur.

ALGORITHMUS:

(1) $G_{Mol} \leftarrow \emptyset$
% Liste *matched*, um uns die bereits zur Rekonstruktion verwendeten Ecken zu merken
(2) $matched \leftarrow \emptyset$
(3) **while** $(matched \neq V_I)$ **do**
% Betrachte jeweils den nächst größten Subgraphisomorphismus
(4) $F_i \leftarrow F.lastElement()$
(5) $F \leftarrow F - \{F_i\}$
(6) **for all** $f \in F_i$ **do**
(7) **if** $(matched \cap f = \emptyset)$ **then**
(8) **for all** $f(v) = v_I$ **do**
(9) $matched \leftarrow matched \cup v_I$
% Seien $a_1, a_2 \in L_A$ jene Atome, deren Bindung durch v_I in G_I dargestellt ist.
(10) $V_{Mol} \leftarrow V_{Mol} \cup \{a_1, a_2\}$
(11) $E_{Mol} \leftarrow E_{Mol} \cup \{(a_1, a_2)\}$
(12) **end for**
(13) **end if**
(14) **if** $(matched = V_I)$ **return** G_{Mol}
(15) **end for**
(16) **end while**
(17) **return** G_{Mol}

Algorithmus 3.7: Rekonstruktion einer chemischen Struktur aus den Ergebnisse der Subgraphisomorphismen-Suche.

Falls solch eine Abbildung f_i beim Durchlauf der Liste F_i gefunden wird, werden zunächst für $f_i(v) = v_I$ die beiden in v_I kodierten Atome, sowie deren Bindungen in den Molekulargraph eingefügt. Dann merken wir uns alle Ecken in V_I' als rekonstruiert. Anschließend wird die Menge F solange abgearbeitet, bis alle Ecken aus V_I rekonstruiert sind.

Bevor der vom Algorithmus 3.7 rekonstruierte Molekulargraph G_{Mol} automatisch in einem SDfile-Format abgelegt wird, prüfen wir zunächst nach, ob alle Atome im G_{Mol} genau so viele Bindungen haben wie ihre Bindigkeit (Bindungszahl) es vorschreibt. Es wird ebenfalls sichergestellt, dass keine der Bindungen eine Länge von weniger als einem Drittel der Durchschnittlänge aller Bindungen in G_{Mol} hat. Anschließend wird das Ergebnis dem Benutzer in einer Benutzeroberfläche präsentiert. Für den Fall, dass noch Valenzfehler in der rekonstruierten Struktur festgestellt worden sind, oder wenn noch verdächtig kurze Bindungen in G_{Mol} vorkommen, werden diese markiert angezeigt. Der Benutzer hat nun die Möglichkeit, alle fehlerhaften Stellen zu korrigieren. Eine Nachbearbeitung wird dem Benutzer natürlich auch dann angeboten, wenn das Programm seine Arbeit für einen Erfolg hält.

Da wir beim Matching bisher kein chemisches Fachwissen einbeziehen, kann eine tatsächliche Optimalität der Rekonstruktion nicht garantiert werden. Ein mögliches Kriterium für Optimalität wäre nun die Maximierung der durchschnittlichen Größe der zur Rekonstruktion herangezogenen Fragmente. Ebenso wäre es denkbar, deren Anzahl zu minimieren. Intuitiv klar ist nur, dass das häufige Matchen des für eine einzelne Bindung stehenden Fragments zu einer schlechten Bewertung führen sollte.

Zur Bewertung der Ausgabe werden Anzahl und Art der Korrekturen des Eingabe-Bild (Löschen, Verlängern eines Liniensegmentes, und Verschmelzen der Mehrfachbindungen) ausgegeben. Außerdem wird die Güte der Rekonstruktion durch einen Wert zwischen 0 und 1 bewertet. Dieser wird wie folgt berechnet:

$$s = \frac{|B_R|}{|B_I|}$$

Dabei bezeichnet $|B_I|$ die Anzahl der Bindungen in der Eingabe-Struktur, und $|B_R|$ die Anzahl aller Bindungen, die mit Graphen aus dem Modell-Netzwerk gematcht und auch tatsächlich zur Rekonstruktion herangezogen wurden. Nicht berücksichtigt werden dabei Bindungen, die mit der einzelnen Bindung im Modell-Netzwerk gematcht wurden. Diese kann jede Bindung matchen und damit würde die Bewertung immer eins betragen. Wir betrachten also in B_R nur solche Bindungen, die mit zusammengesetzten Modellen aus der Bibliothek gematcht wurden.

Man beachte, dass wir uns damit auf keines der zuvor vorgeschlagenen Maße für die Optimalität festgelegt haben. Die Bewertungsfunktion liefert

uns ausschließlich den Prozentsatz an Bindungen, die nicht durch das singuläre Fragment rekonstruiert wurden.

Bei dem zur Rekonstruktion eingesetzten Algorithmus haben wir jeweils den größtmöglichen Subgraphisomorphismus herangezogen. Dies kann die Anzahl der Matchings mit dem singulären Fragments womöglich erhöhen.

Der Algorithmus zur Rekonstruktion benötigt $O(kn^3)$ Zeit, wobei k die Anzahl der Graphen im Dekompositionsnetzwerk und n die Anzahl der Ecken im Eingabe-Graph ist. Die *while*-Schleife wird höchstens k mal durchlaufen. Die äußere *for*-Schleife in Zeile 6 ist durch $O(n^2)$ beschränkt, da $O(n^2)$ Subgraphisomorphismen von einem Graph im Netzwerk mit dem Eingabe-Graph gefunden werden können. Schließlich erfordert die innere *for*-Schleife $O(n)$ Berechnungsschritte.

3.6 Lernen einer neuen Grundstruktur

Für die Erkennung einer neuen Struktur steht uns bereits eine Bibliothek von Modell-Fragmenten zur Verfügung, aus denen sich eine Vielzahl von Strukturen zusammensetzen lässt. Insbesondere können aus den azyklischen Modellfragmenten beliebig lange Ketten (verzweigt oder unverzweigt) gebildet werden.

Neben azyklischen Ketten kommen in chemischer Strukturen oft Kreise (Ringe) unterschiedlicher Länge vor. Solche Ringe unterschiedlicher Größen können wiederum zu größeren zyklischen Gebilden kombiniert werden.

Um die Eingabe-Graphen schnell erkennen und rekonstruieren zu können, sollte die Bibiothek möglichst viele der auftretenden Teilstrukturen enthalten. Andererseits darf die Bibliothek nicht beliebig groß werden, weil dies zu einer längeren Laufzeit beim Matching führt.

Diese konträren Anforderungen führen zu folgenden Überlegungen: Auf die nachträgliche Aufnahme azyklischer Strukturen in die Bibliothek wird prinzipiell verzichtet. Um die Aufnahme zyklischer Strukturfragmente in die Bibliothek zu ermöglichen, wurde das Matchingverfahren wie folgt erweitert.

Im Dekompositionsnetzwerk sind all diejenigen Modell-Graphen, die zyklische Strukturen darstellen, gesondert gekennzeichnet. Bevor das Matching zwischen einem Eingabe-Graph und den Modell-Graphen beginnt, suchen wir alle *zweifach zusammenhängenden* Komponenten in der zu erkennenden Eingabe-Struktur. Im Rahmen des Matching merken wir uns nun, welche dieser Komponenten komplett durch eine zyklische Struktur aus dem Dekompositionsnetzwerk erkannt wurden. Somit erhalten wir alle zweifach zusammenhängenden Komponenten in der Eingabe-Struktur, die noch nicht in der Bibliothek enthalten sind.

Dabei ist die Erweiterung der Bibliothek um azyklische Ketten nicht prinzipiell ausgeschlossen; sie wird denjenigen Benutzern überlassen, die über chemisches Fachwissen verfügen und die Entscheidung treffen können,

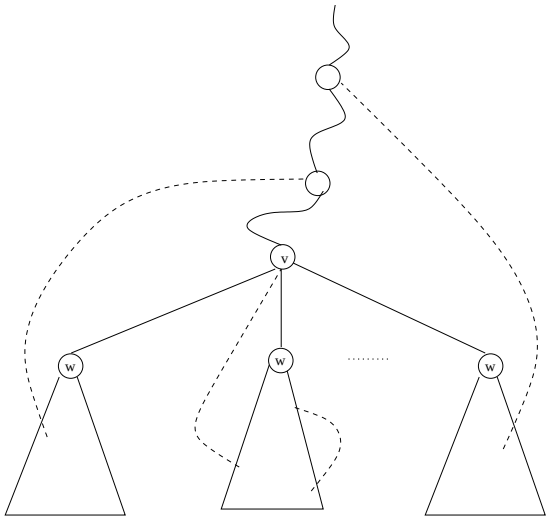


Abbildung 3.14: Ein Teilbaum des DFS-Wurzelbaumes T kann nur durch Rückwärtskanten mit in T „höhergelegenen“ Vorgänger-Ecken verbunden sein (außer dem einen Pfad aus Baumkanten, auf dem er entdeckt wurde).

ob eine solche Erweiterung sinnvoll ist.

Um nun alle 2-fach zusammenhängende Komponenten eines Graphen $G = (V, E)$ zu bestimmen, suchen wir mittels DFS dessen trennende Ecken und untersuchen, wie diese die 2-fach zusammenhängende Komponenten von G trennen.

Zuerst rufen wir ein paar Eigenschaften von DFS in Erinnerung. Für einen Graph G werden durch DFS mehrere DFS-Wurzelbäume erzeugt. Jede Ecke $v \in V$ hat in solch einem Baum höchstens einen Vorgänger $u \in V$ ($pred[v] = u$), von dem aus v entdeckt wurde. Ferner werden die Entdeckungszeiten aller Ecken $v \in V$ in einem Array $discovered[v]$ gespeichert. Verfolgen wir nun einen Pfad in einem DFS-Baum T, so haben die Ecken entlang dieses Pfades aufsteigende Entdeckungszeiten. Durch DFS werden außerdem die Kanten $e \in E$ in Baum- und Rückwärtskanten klassifiziert. Baumkanten sind all diejenigen Kanten, die im DFS-Wald vorkommen. Rückwärtskanten sind solche Kanten (u, v), bei denen v ein Vorgänger von u im DFS-Baum ist.

Aus den beschriebenen Eigenschaften folgt, dass ein Teilbaum des DFS-Wurzelbaumes T nur durch Rückwärtskanten mit in T „höhergelegenen“ Vorgänger-Ecken verbunden sein kann; außer dem einen Pfad aus Baumkanten, auf dem er entdeckt wurde. Abbildung 3.14 illustriert diesen Sachverhalt. Außerdem lässt sich Folgendes beobachten:

1. Die Wurzelecke s eines DFS-Wurzelbaumes ist genau dann eine trennende Ecke, wenn s mehr als zwei direkte Nachfolger hat.

2. Jede innere Ecke v eines DFS-Wurzelbaumes ist eine trennende Ecke, falls folgende Bedingungen zutreffen:

   - v hat einen Nachfolger;
   - weder v noch einen Nachfolger u von v im DFS-Wurzelbaumes ist durch eine Rückwärtskante mit einem Vorgänger von v verbunden.

Um zu überprüfen, ob ein Teilbaum durch eine solche Rückwärtskante mit einem 'höheren' Vorgänger im DFS-Wurzelbaum verbunden ist, greifen wir auf die Entdeckungszeiten zurück und definieren ein weiteres Array $low[v]$ mit Einträgen für alle $v \in V$.

$$low[v] = min\{discover[v], discover[w] \mid (u,w) \text{ ist eine Rückwärtskante für irgendeinen Nachfolger } u \text{ von } v\}$$

Wenn eine Ecke v bei DFS entdeckt wird, wird zuerst $discover[v]$ gesetzt und $low[v]$ bekommt den gleichen Wert zugewiesen. Anschließend werden alle Nachbarecken u von v bearbeitet. Der Wert $low[v]$ wird dann wie folgt aktualisiert:

- $low[v] := min(low[v], low[u])$, falls u bisher noch nicht entdeckt wurde;
- $low[v] := min(low[v], discover[u])$, falls u schon entdeckt wurde und $pred[v] \neq u$ gilt.

Im zweiten Fall ist (v,u) eine Rückwärtskante. Falls v eine benachbarte Ecke u hat, so dass $low[v] \geq discover[u]$ gilt, so ist v eine trennende Ecke. Somit lassen sich die trennenden Ecken in G während eines DFS-Durchlaufs ermitteln.

Um die Kanten aller 2-fach zusammenhängenden Komponenten zu finden, wird ein Stapel S verwendet. Wenn eine Kante (u,v) bei DFS durchlaufen wird – entweder um die Ecke v von u aus zu besuchen oder falls (u,v) eine Rückwärtskante ist –, wird sie auf den Stapel S abgelegt. Falls u später mittels oben beschriebener Methode als eine trennende Ecke identifiziert wird, werden solange Kanten aus dem Stapel entfernt, bis auch (u,v) entfernt wird. Wird dabei mehr als eine Kante entfernt, so bilden die entfernten Kanten eine 2-fach zusammenhängende Komponente von G. Andernfalls ist (u,v) eine Brücke.

Die Hauptprozedur des Algorithmus zur Bestimmung aller 2-fach zusammenhängenden Komponenten eines Graphen G ist im Algorithmus 3.8 gegeben. Nach den Initialisierungschritten in der ersten for-Schleife wird für eine Startecke $s \in V$ die Prozedur *BCC-Visit* (Algorithmus 3.9) aufgerufen. Durch jeden Aufruf *BCC-Visit(v)* für eine Ecke $v \in V$ werden alle von v

2-FACH ZUSAMMENHÄNGENDE KOMPONENTEN, HAUPTPROZEDUR

BCC(G):

EINGABE:

G:	Ein Graph $G = (V, E)$ in Adjazenzlistendarstellung mit $n := \|V\|$ und $m := \|E\|$.

AUSGABE:

C:	Menge der 2-fach zusammenhängenden Komponenten von G $C = \{C_1, \ldots, C_l\}, C_i \subseteq E, C_i \cap C_j = \emptyset$ für $i \neq j$.

ALGORITHMUS:

```
(1)  for all v ∈ V do
(2)      pred[v] ← NIL
(3)      discover[v] ← 0
(4)      low[v] ← 0
(5)  end for
(6)  time ← 0
(7)  S ← ∅    % Stapel für Kanten
(8)  for all v ∈ V do
(9)      if (discover[v] = 0) then
(10)         BCC-VISIT(v)
(11)     end if
(12) end for
```

Algorithmus 3.8: Anwendung von DFS zur Berechnung 2-fach zusammenhängenden Komponenten eines Graphen G.

aus erreichbaren Ecken, die noch nicht besucht wurde, durch rekursive Aufrufe derselben Prozedur abgearbeitet. Der Eintrag im Array *discover* dient gleichzeitig als Markierung einer Ecke. Ein Eintrag $discover[v] = 0$ gibt an, dass v noch nicht besucht wurde.

Wie bei DFS kann der Algorithmus zur Bestimmung aller 2-fach zusammenhängenden Komponenten eines Graphen G in $O(n+m)$ Zeit implementiert werden, wenn G in Form von Adjazenzlisten gegeben ist.

PROZEDUR ZUR BERECHNUNG 2-FACH ZUSAMMENHÄNGENDER KOMPONENTEN

BCC-VISIT(v):

EINGABE:

v: Die zu untersuchende Ecke.

ALGORITHMUS:

```
(1)  time ← time + 1
(2)  discover[v] ← time
(3)  low[v] ← time
(4)  for all u ∈ Adj[v] do
(5)     if (e = (v, u) ∉ S) then
(6)        S.push(e)
(7)     end if
(8)     if (discover[u] = 0) then
(9)        pred[u] ← v
(10)       BBC-Visit(u)
(11)       if (low[u] ≥ d[v]) then
 %         v ist eine trennende Ecke
(12)          C_i ← ∅
(13)          while (e = (v, u) ∈ S) do
(14)             C_i ← C_i ∪ {S.pop}
(15)          end while
(16)          if (|C_i| > 1) then
(17)             C ← C ∪ {C_i}
(18)          end if
(19)          low[v] ← min[low[v], low[u]]
(20)       else if (pred[u] ≠ v) then
 %         (u, v) ist eine Rückwärtskante
(21)          low[v] ← min[low[v], discover[u]]
(22)       end if
(23)    end if
(24) end for
```

Algorithmus 3.9: Prozedur zur Berechnung 2-fach zusammenhängender Komponenten eines Graphen G.

3.7 Erweiterung des Konzeptes

In Folgenden wollen wir zwei Erweiterungen des bisher beschriebenen Konzepts vorstellen.

3.7.1 Verfeinerung des Matching-Verfahrens

Das Eingabe-Bild kann viele Rauschdaten und Verzerrungen enthalten, die eine korrekte Rekonstruktion verhindern. Wir haben zwar versucht, diese fehlerhaften Daten im erzeugten Eingabe-Graph zu eliminieren, dies ist jedoch nicht vollständig möglich. Für Verzerrungen des Winkels zwischen zwei Bindungen wurde bei der bisher beschriebenen Subgraphisomorphismen-Suche bereits eine Abweichung von 5 Grad toleriert. Testergebnisse haben aber gezeigt, dass größere Abweichungen nicht so selten sind. Die Folge ist, dass Teilstrukturen, die im Vergleich zu den idealen Modellen verzerrt sind, nicht gemacht werden können.

Um diesem Problem zu begegnen, ist es daher notwendig, eine *fehlerkorrigierende* Subgraphisomorphismen- Suche einzusetzen. Beim diesem Verfahren werden *Edit Operationen* eingeführt, um die Transformation eines idealen Modell-Graphen in einen Subgraph des verzerrten Eingabe-Graphen zu modellieren. Die erlaubten *Edit-Operationen* umfassen dabei typischerweise Insertionen, Deletionen und Substitutionen von Ecken und Kanten. In unserem Kontext sollte auch eine Veränderung des Winkels zwischen zwei Bindungen erlaubt sein. Das zentrale Problem dieses Ansatzes besteht darin, dass zu klären ist, wieviele solcher *Edit-Operationen* erlaubt sind; es muß bestimmt werden, ab wann zwei Graphen nicht mehr ähnlich genug sind. Dafür wird üblicherweise eine geeignete Kostenfunktion für jede erlaubte Operation eingeführt. Die Subgraph-Distanz von einem Modell zu einem Eingabe-Graph ist nun definiert über die minimale Summe aller Kosten, die entstehen, wenn das Modell so editiert wird, dass es mit einem Subgraph der Eingabe gematcht werden kann. Dabei werden alle möglichen Folgen von solchen Edit-Operationen berücksichtigt. Je kleiner die Distanz ist, umso ähnlicher sollten die beiden Subgraphen sein. Die Bestimmung geeigneter Schwellenwerte für die Ähnlichkeit erfolgt üblicherweise experimentell, etwa mit Hilfe von Machine Learning. In [48] wurde beispielsweise ein Verfahren vorgestellt, Graph Edit Kosten mittels *„self-organizing map“* zu lernen.

Aus Zeitgründen war eine Implementierung fehler-korrigiernden Matchings im Rahmen dieser Arbeit leider nicht möglich. Wir wollen den Ansatz im Folgenden aber kurz skizzieren.

Um diese Verfahren zum Einsatz zu bringen, sollte in der vorliegenden Implementierung der Matching-Algorithmus 3.4 entsprechend angepasst werden. Dazu muß die Prozedur *Match_Vertex* zum Matchen einzelner Ecken (Algorithmus 3.5) verändert werden. Diese Prozedur findet Subgraphisomorphismen zwischen dem Modell-Fragment, dass für eine einzelne Bindung

steht, und dem Eingabe-Graph. Sie soll abhängig von den bisher aufgelaufenen Edit-Kosten auch das Löschen einer Ecke im Modell-Graph gestatten. Das Einfügen können wir außer Betracht lassen, da wir einen Subgraphisomorphismus zwischen dem Modell und der Eingabe suchen. 'Überflüssige' Ecken im Eingabe-Graph werden von diesem Verfahren ignoriert. Desweiteren muß der Algorithmus *Merge* (Algorithmus 3.6), welcher zwei bereits gefundene Subgraph-Isomorphismen zu einem größeren zu kombinieren versucht, angepasst werden. Ebenfalls in Abhängigkeit von den bisher aufgelaufenen Edit-Kosten soll er alle drei möglichen Kanten-Edit-Operationen, die Kantenattributsubstitution (Winkelattribut), Kantendeletion und Kanteninsertion, ermöglichen. Die Buchführung über die aufgelaufenen Kosten muß in den Matching-Algorithmus integriert werden. Um die kombinatorische Explosion bei der Kombination von Subgraphisomorphismen durch die nun stark angewachsene Anzahl an Eingaben in den Griff zu bekommen, sind besondere Vorkehrungen nötig. Dazu werden für jeden Graph des Netzwerks jeweils eine *open*- und *closed*-Liste eingeführt. Wenn ein Subgraphisomorphismus eines Graphen G des Netzwerks mit dem Eingabe-Graph gefunden wird, wird er in die *open*-Liste von G aufgenommen. Dann wird in einem Schritt jeweils das kosten-minimale Element aller *open*-Listen bestimmt. Dieses wird dann mit allen Elementen aller *closed*-Listen kombiniert und selbst in die entsprechende *closed*-Liste eingefügt. Wenn eine solche Kombination erfolgreich war, wird ihr Ergebnis in die passende *open*-Liste eingefügt. Dieses Verfahren garantiert, dass stets nur kosten-minimale Kombinationen erprobt werden.

Eine formellere Beschreibung des fehlerkorrigierenden Subgraphisomorphismus Verfahren mittels Graph-Dekomposition findet sich in [46].

3.7.2 OCR-Verfahren zur Erkennung von Atomsymbolen

Die Erkennung von Atomsymbole können auch über ein fehlerkorrigierendes Graphisomorphismus Verfahren erfolgen. Das Verfahren besteht aus zwei Phasen. In der ersten, der Lernphase, werden verschiedene repräsentative Schriftmuster gelernt. Dazu soll für jedes Zeichen eine Menge representativer Bilddarstellungen zusammengestellt werden. Diese definieren dann eine Äquivalenzklasse für das entsprechende Zeichen. Aus ihren Bildern werden die entsprechenden Graphenrepräsentationen erzeugt. Knick- und Endpunkte im Bild werden dabei durch Ecken, Liniensegmente durch Kanten des Graphen repräsentiert. Für jede Klasse wird danach ihr *Median* berechnet, der als Repräsentant diese Klasse vertritt. Ein Median ist dabei definiert als jener Graph, für den die Summe der Distanzen zu allen anderen Graphen der Klasse minimal ist.

In der Erkennungsphase werden die zu erkennenden Atomsymbole nun eingelesen und als Graphen repräsentiert. Wir gehen hier davon aus, dass die Trennung zwischen Atomsymbolen und Bindungen bereits erfolgt ist. Mit-

tels fehlerkorrigierender Graphisomorphismus-Suche werden die Repräsentanten gematcht. Der Isomorphismus mit den geringsten Edit-Kosten kann zur Identifikation des Atomsymbols herangezogen werden [32].

Zur Trennung von Atomsymbolen und Bindungen könnte beispielsweise eine Variante des Kasturi-Algorithmus [59] zur Anwendung gebracht werden.

Kapitel 4

Auswertung

Für das im letzten Kapitel beschriebenen Konzept wurde ein Prototyp in der Programmiersprache Java implementiert. Dieses Kapitel befasst sich mit den Testergebnissen der Implementierung und deren Auswertung.

Für die durchgeführten Tests wurde folgende Konfiguration verwendet:

- Intel(R) Pentium(R) 4 CPU 3.06GHz
- 1 GB Hauptspeicher
- Betriebssystem Fedora Core 2.4.22
- Java 1.4.2-6

4.1 Testkorpus

Der Testkorpus besteht aus den Molekülstrukturen der meistverkauften Medikamente aus dem Jahr 2002. Er wurde ursprünglich für die Evaluierungsstudie [20] des Programmes CLiDE [29] zusammengestellt. Die Namen der Molekülstrukturen wurden der RxList [36] entnommen. Die Moleküle wurden dann mit dem Programm ChemDraw nach den Vorlagen in [47] gezeichnet. Die Bitmap-Dateien sind die Screenshots dieser Zeichnungen. Kriterien für die Zusammenstellung waren: unterschiedliche Strukturgrößen, wichtigste Atome in der organische Chemie, häufig auftretende Ringsysteme und unterschiedliche Bindungsarten [20]. Abbildung 4.2 zeigt einen Ausschnitt aus dem Korpus.

Der Testkorpus enthält 38 Strukturen, in denen Chiralbindungen vorkommen. Sie wurden nicht getestet, da diese Bindungsart in der Implementierung noch nicht berücksichtigt wurde. Aus dem gleichen Grund haben wir beim Testen jene Bitmap-Dateien außer Acht gelassen, bei denen Atomsymbole und Bindungen sich überlappen. Von den 97 Strukturen in diesem Korpus haben wir somit 49 getestet.

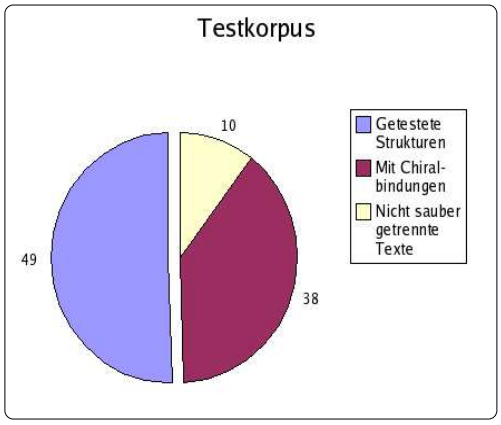


Abbildung 4.1: Übersicht des gegebenen Testkorpus.

Bei der Evaluierung unseres Programms interessieren wir uns für die folgenden Fragen:

- Werden die Strukturen korrekt rekonstruiert?
- Werden die Mehrfachbindungen erkannt?
- Werden Atomsymbole den richtigen Bindungen zugeordnet?
- Werden neue Ringssysteme erkannt und in die Modellbibliothek aufgenommen?
- Ist die rekonstruierte Struktur für automatische Speicherung im SDfile-Format geeignet?
- Wie verhält sich die Bewertungsfunktion?

4.2 Ergebnisse

Wir haben 49 Strukturen getestet, davon wurden 35 richtig rekonstruiert. Abbildung 4.3 zeigt einen Überblick der Testergebnisse.

Korrektheit der Rekonstruktion

Abbildung 4.3 schlüsselt die fehlerhaften Rekonstruktionen nach der Art der aufgetretenen Fehler auf. Dabei ist zu erkennen, dass aufgebrochene Bindungen der häufigste Fehler sind. Dieser Fehler tritt vor allem dann auf, wenn ein Atom an vier oder mehr Bindungen beteiligt ist, wobei mindestens eine von ihnen eine Doppelbindung ist. Das häufige Auftreten dieses Fehlers lässt sich durch zwei Beobachtungen erklären. Erstens hat das Bildverarbeitungsprogramm AutoTrace es an solchen Stellen besonders schwer, das

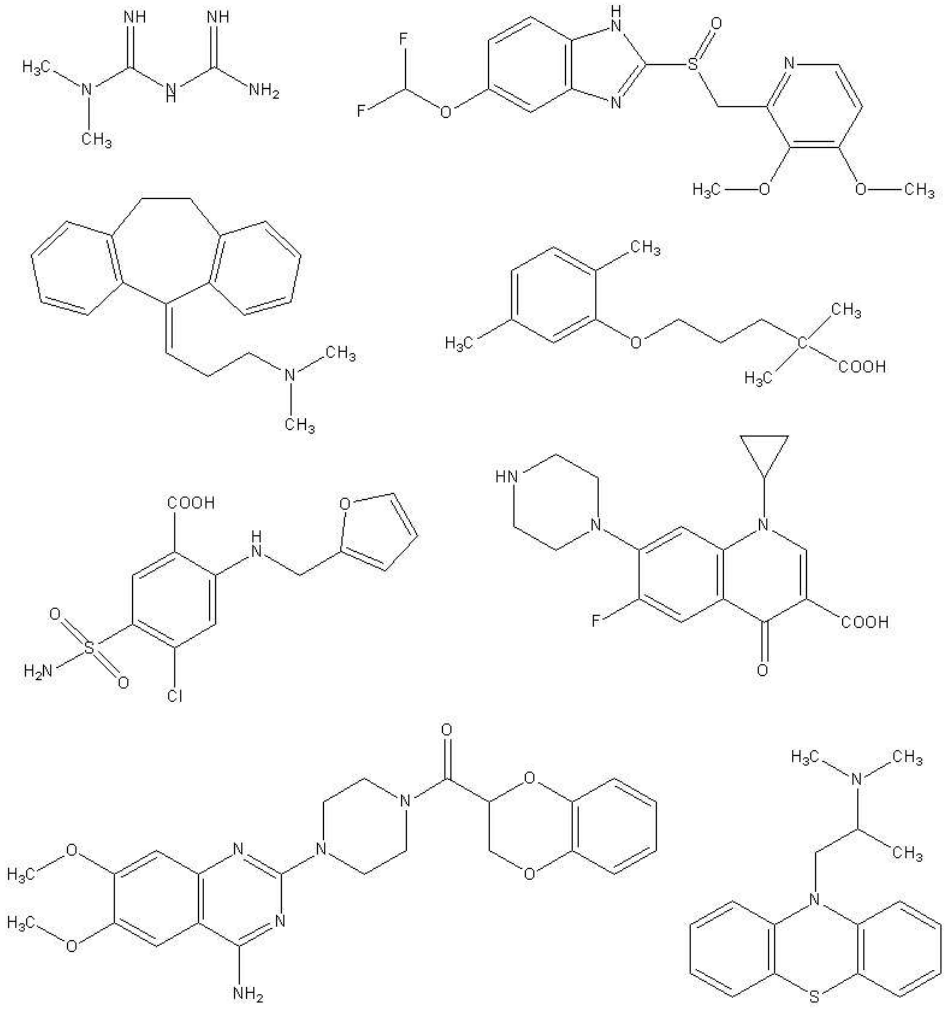


Abbildung 4.2: Einige der getesteten Strukturen.

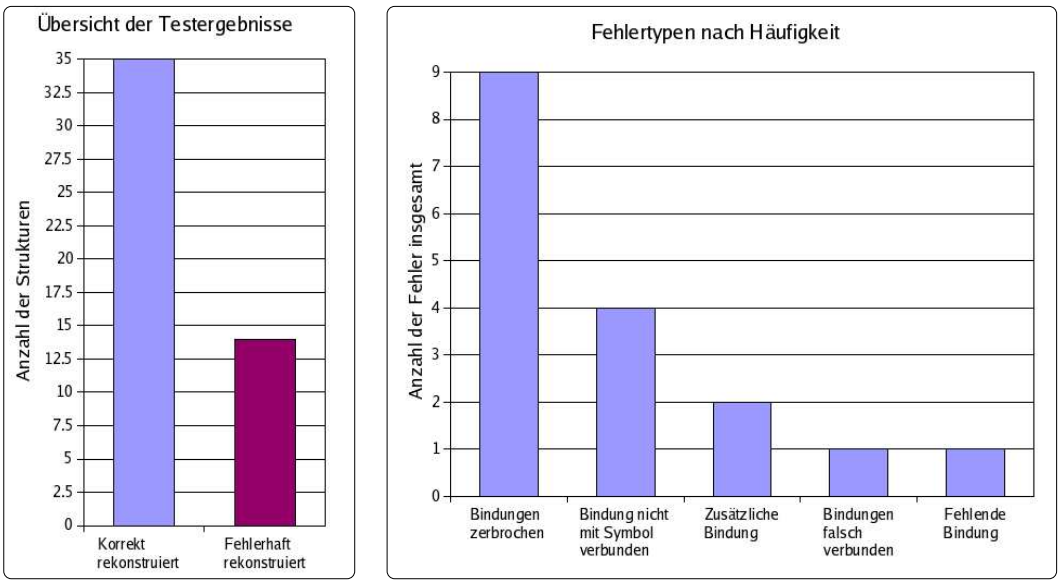


Abbildung 4.3: Übersicht der Testergebnisse.

Bild richtig zu segmentieren. Oft werden mehrere zusätzliche kleine Segmente erzeugt und/oder Objekte, die im Original verbunden sind, werden getrennt. Zweitens wird beim Verschmelzen von Mehrfachbindungen oft die Lage der Linien verändert, so dass bei der Nachbearbeitung der konvertierten Vektorgraphik weniger genaue Informationen über die Nachbarschaften zur Verfügung stehen. Die zusätzlichen kleinen Segmente werden bei der Nachbearbeitung entfernt, so dass wiederum getrennte Objekte entstehen. Wir versuchen zwar, fälschlicherweise getrennte Objekte bei der Nachbearbeitung wieder zu verbinden; wenn jedoch durch verschmolzene Mehrfachbindungen die Genauigkeit der Nachbarschaften leidet, häufen sich die Fehler.

Derartige Fehler lassen sich aber beseitigen, insbesondere wenn mehr chemisches Wissen bei der Interpretation mit einbezogen wird. Wenn man zum Beispiel zusätzlich die Bindigkeit des betroffenen Atoms betrachten würde, könnte man mitunter fehlende Bindungen erkennen. Dann könnte man eine *k-Nearest Neighbor* Suche durchführen, um einen wahrscheinlichen Partner zu identifizieren. Da die Bindigkeit bei den einzelnen Atomen variiert und außerdem Wasserstoff-Atome sowohl in den Strukturdiagrammen als auch im SMILES-String in der Regel nicht extra angegeben werden, erfordert eine Einbindung dieser Information mehrere Fallunterscheidungen.

Behandlung von Mehrfachbindungen und Atomsymbolen

Wie in obiger Abbildung außerdem zu erkennen ist, sind andere Fehlerarten nur selten vorgekommen. Die Mehrfachbindungen wurden in allen Fällen

korrekt erkannt. Manchmal wurden zusätzliche Bindungen eingefügt oder nicht alle Bindungen, die zu einem bestimmten Atom gehören, werden mit diesem verbunden. Diese fehlerhaften Rekonstruktionen erfordern jedoch oft nur jeweils einen einzigen korrigierenden Eingriff des Benutzers, um eine originalgetreue Struktur zu erhalten. Auch hier gilt wieder, dass der Einbezug von mehr chemischen Fachwissen in die Rekonstruktion diese Fehler vermeiden helfen kann.

Aufnahme neuer zyklischer Strukturen

Die Identifikation der zweifach-zusammenhängenden Komponenten im Molekulargraph des unbekannten Struktur funktionierte korrekt. Wenn jedoch solche zyklische Komponenten in der Eingabe nicht direkt mit einem entsprechenden zyklischen Modell-Strukturfragment gematcht werden konnten, so ergab sich folgendes Bild: In 18 von 20 Fällen waren die nicht erkannten Strukturfragmente bereits in der Bibliothek enthalten. Da sie jedoch im Eingabegraph verzerrt im Vergleich zur Bibliothek vorlagen, konnten sie nicht gematcht werden. Eine solche Verzerrung ist ausschlaggebend, wenn der Winkel zwischen zwei Bindungen in der Eingabe mehr als fünf Grad vom Winkel zwischen den entsprechenden Bindungen im Modellstrukturfragment abweicht. Als Ausweg aus dieser unbefriedigenden Situation schlagen wir vor, die Modellstrukturfragmente zu „normalisieren“. Es sollten je Modell-Strukturfragment eine größere Zahl von Darstellungen genommen werden, um ihren *Median* zu berechnen. Diese durchschnittliche Darstellung des Modell-Fragments wird dann in die Bibliothek aufgenommen. Wir erwarten so die Anzahl der fälschlicherweise nicht gematchten zyklischen Strukturenfragmente reduzieren zu können. Das Verfahren könnte auch um eine Lernkomponente erweitert werden, indem fälschlicherweise nicht gematchte Strukturfragmente gewichtet in den Median eingerechnet werden.

Automatische Speicherung als SDfile

Die automatische Speicherung der rekonstruierten Graphen als SDfile hat sich als weitgehend problemlos erwiesen. Ausschließlich in jenen Fällen in denen entweder Valenzfehler vorliegen oder Bindungen um ein Drittel kürzer als die durchschnittliche Bindung sind, wird der Graph nicht automatisch gespeichert. Statt dessen werden diese Stellen dem Benutzer markiert zur Korrektur vorgelegt. Zum einen sind solche Fälle nahezu immer Zeichen für Fehler; andererseits können solch fehlerhafte SDfiles den Editor für chemische Strukturen – Marvin – zum Absturz bringen.

Bewertung der Rekonstruktionen

In Abbildung 4.4 werden die Bewertungen der Rekonstruktionen wiedergegeben. Wie auf den ersten Blick ersichtlich ist, liegen die Bewertungen für

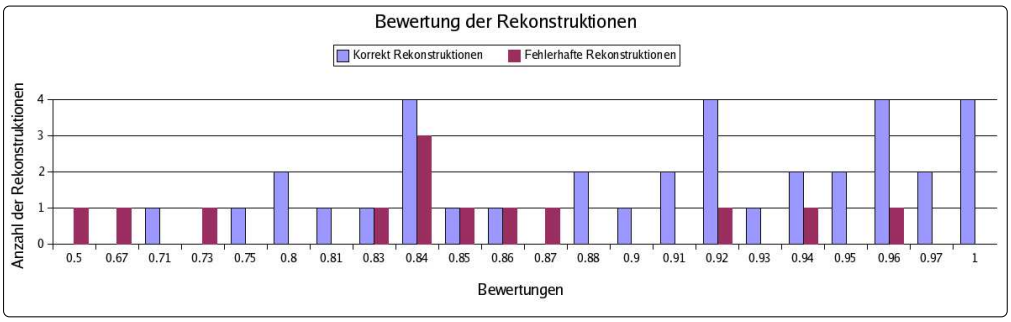


Abbildung 4.4: Bewertung der Rekonstruktionen.

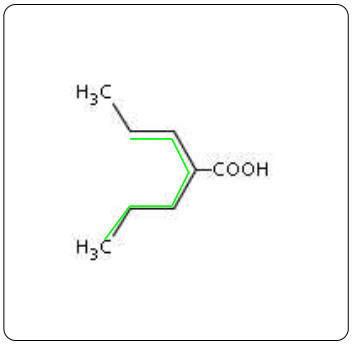


Abbildung 4.5: Beispiel einer sehr kleinen Struktur, die trotz ihrer korrekten Rekonstruktion eine schlechte Bwertung erhalten hat. Das markierte, zur Rekonstruktion verwendete Strukturfragment erzwingt die zweimalige Benutzung des singulären Struktur-Fragments.

korrekte und fehlerhafte Rekonstruktionen recht nah beieinander. Die durchschnittliche Bewertung der korrekten Rekonstruktionen beträgt 0.93, die der fehlerhaften 0.82. Die recht hohe Bewertung der fehlerhaften Rekonstruktionen erklärt sich daraus, dass zumeist nur ein einzelner Fehler vorhanden ist. Die auffälligen korrekten Rekonstruktionen mit sehr niedrigen Bewertungen sind allesamt Fälle, bei denen für sehr kleine Eingabestrukturen eine Substruktur nicht in der Bibliothek enthalten war. Diese wurde dann aus dem Modell-Fragment, das für eine einzelne Bindung steht, rekonstruiert. Wie bereits oben erwähnt, können diese Teile der Rekonstruktion in der Bewertungsfunktion nicht berücksichtigt werden.

Laufzeit

In Abbildung 4.6 sind die Laufzeiten der einzelnen Rekonstruktionen der Teststrukturen angezeigt. Die Zeitmessung umfasst dabei die Umwandlung des Eingabe-Bildes in einen Graph, das Graph-Matching und die Rekonstruktion. Die Spezifizierung der Atomsymbole durch den Benutzer wurde

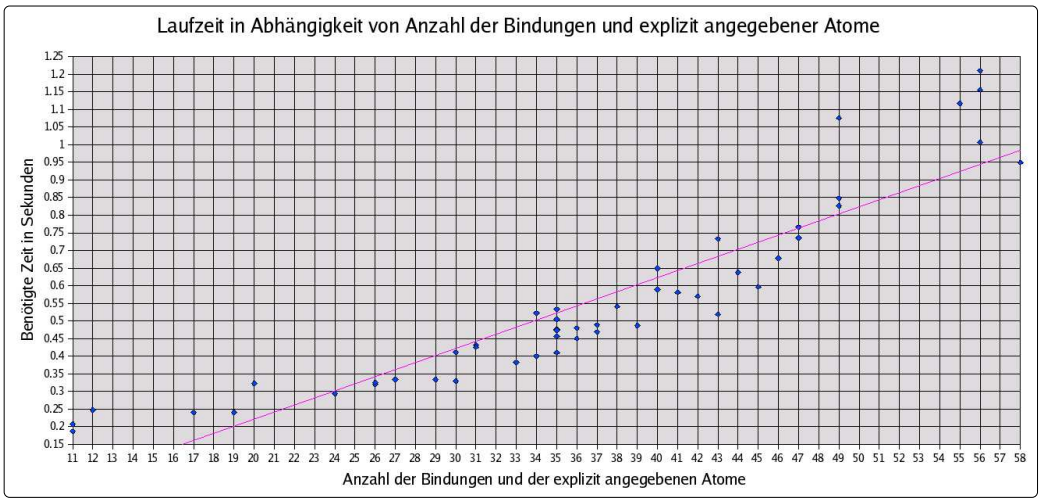


Abbildung 4.6: Laufzeiten je Rekonstruktion in Abhängigkeit von der Anzahl der Bindungen sowie explizit angegebener Atome.

natürlich nicht berücksichtigt.

Da die Vorbearbeitungszeit des Eingabe-Bildes nicht nur von der Anzahl der Bindungen, sondern auch von der Anzahl der explizit angegebenen Atome abhängt, haben wir die Laufzeit in der Zeichnung in Abhängigkeit von diesen beiden Parametern aufgetragen. Dabei werden Mehrfachbindungen mehrfach gezählt, da sie ebenso wie die Atomsymbole die Zeit der Vorbearbeitung beeinflusst. Aus der Abbildung wird deutlich, dass die Laufzeit etwa linear zur Anzahl der Bindungen und der explizit angegebenen Atome ansteigt.

Die gemessenen Laufzeiten zeigen eine lineare Abhängigkeit von der Anzahl der Bindungen und der explizit angegebenen Atome. Dies heißt aber nur, dass die exponentielle worst-case Komplexität des eingesetzten Verfahrens bei den getesteten Strukturen nicht zum Tragen gekommen ist. Umfangreiche Tests, die ein Maß für die praktische Bedeutung der exponentiellen Komplexität liefern können, sind für die Zukunft geplant.

Kapitel 5

Zusammenfassung und Ausblick

Ziel der Diplomarbeit war die Ausarbeitung eines Konzepts zur Rekonstruktion chemischer Strukturen aus Bilddokumenten und ihre Abspeicherung in einem maschinenlesbaren Format.

In dieser Arbeit wurde einen Lösungsansatz vorgestellt, der auf struktureller Mustererkennung basiert. Dazu wurde eine Bibliothek von Modell-Strukturfragmenten zusammengestellt, die zur Erkennung der unbekannten Struktur im Eingabe-Bild herangezogen wird. Die Repräsentation der Modell-Strukturfragmente und des Eingabe-Bildes erfolgt durch ungerichtete attributierte Graphen. Mittels Subgraphisomorphismen-Suche werden die Modell-Strukturfragmente mit der unbekannten Struktur verglichen. Aus den so ermittelten Substrukturen wird diese dann rekonstruiert.

Obwohl organische Verbindungen in großer Diversität vorliegen, lässt sich feststellen, dass bestimmte Strukturfragmente in vielen Verbindungen als Teil vorkommen. Wir haben versucht, diese Strukturfragmente in die Modell-Bibliothek aufzunehmen. Um zu verhindern, dass beim Matching gemeinsame Teilstrukturen der Modelle unnötig mehrfach mit einer unbekannten Struktur verglichen werden, wurden die Modelle in einem off-line Prozess zerlegt und in einem Dekompositionsnetzwerk gespeichert. In diesem Netzwerk sind Teilstrukturen, die mehrmals in verschiedenen Modellen oder in einem Modell enthalten sind, durch nur einen Knoten dargestellt. Das Matching lässt sich zur Laufzeit dann effizienter gestalten, indem jede Teilstruktur im Netzwerk höchstens einmal mit der unbekannten Struktur verglichen werden.

Das Modell-Netzwerk kann inkrementell erweitert werden. So werden zum Beispiel zur Laufzeit automatisch neue Ringe/kondensierte Ringsysteme ermittelt. Dazu werden die 2-fach zusammenhängenden Komponenten im Molekulargraph bestimmt. Bei Zustimmung des Benutzers werden sie automatisch in das Netzwerk integriert, und stehen bei der nächsten Rekon-

struktionen ebenfalls als Modelle zur Verfügung.

Die Rekonstruktion wird automatisch im SDfile-Format abgelegt, in welchem die Topologie des Moleküls strukturiert gespeichert ist. Über die Schnittstelle mit MARVIN – einem Editor für chemische Strukturen – wird dem Benutzer eine manuelle Nachbearbeitung angeboten.

Es wurde ein Prototyp des vorgestellten Ansatzes in der Programmiersprache JAVA implementiert. Leider war es innerhalb der beschränkten Zeit einer Diplomarbeit nicht möglich, ein vollautomatisches System zur Rekonstruktion chemischer Strukturen zu liefern. Das System kann jedoch dazu erweitert werden. In einigen anderen Aspekten kann die Implementierung ebenfalls verbessert werden.

Im ersten Schritt, der Umwandlung des Eingabebildes zur Vektorgraphik, steckt noch deutliches Verbesserungspotenzial. Folgende Möglichkeiten bieten sich an:

- Der Einsatz von Machine-Learning-Verfahren, um die optimalen Parameter für AUTOTRACE zu ermitteln.
- Implementierung eines eigenen spezialisierten Konverters.

Die Implementierung eines OCR-Verfahrens zur automatischen Erkennung der Atomsymbole ist ein weiterer Schritt. Dazu kann beispielsweise die im Abschnitt 3.7.2 kurz vorgestellte Methode angewandt werden.

Die Erkennung und Interpretation von Chiral-Bindungen ist ebenso eine notwendige Erweiterung.

Von unserem Standpunkt aus betrachtet ist die Verfeinerung des Graph-Matching zu einem fehlertolerierten Verfahren die interessanteste der Erweiterungen. Wir glauben nicht, dass Verbesserungen bei der Konvertierung zur Vektorgraphik ausreichen werden, um fehlerfreie Eingabe-Graphen zu erhalten. Graph-Matching in Kombination mit Edit-Operationen wäre außerdem in der Lage kontextsensitiv verschiedene Parameter für die Korrekturen zur Anwendung zu bringen. Außerdem lassen die Testergebnisse darauf schließen, dass chemisches Fachwissen in das Matching-Verfahren und damit auch in das Modell-Netzwerk integriert werden muss. Hier sehen wir das größte Verbesserungspotenzial.

Anhang A

Implementierung

Bei der Entwicklung wurde folgende Software verwendet:

- die Laufzeitumgebung Sun Java Software Development Kit 1.4.2,
- die Entwicklungsumgebung Eclipse 3.0 [18],
- die Opensource Software AutoTrace 0.31.1 [4] zur Konvertierung von Bitmap-Bilder in Vektorgraphiken (SVG),
- Batik 1.5 [60] zum Parsen von SVG nach Java2D,
- Marvin 3.5.1 [37] zur Darstellung und Nachbearbeitung der rekonstruierten chemischen Strukturen.

Das entwickelte Programm besteht aus fünf verschiedenen Paketen. Die Abhängigkeit der Pakete voneinander ist der Abbildung A.1 zu entnehmen. Im Paket `graph` sind die Datenstrukturen zum Speichern von Graphen enthalten. Dabei werden Graphen in Adjazenzlistendarstellung in einer HashMap abgelegt. Diese vordefinierte Java-Klasse unterstützt besonders gut die dynamischen Änderungen von Knoten und Kanten. Die Algorithmen zur Bestimmung 2-fach zusammenhängender Komponenten (DFS) und des relativen Nachbarschaftsgraph finden sich im Paket `graph.algorithms`. Ersterer wird auch für die Bestimmung der Zusammenhangskomponenten und der Artikulationspunkte ungerichteter Graphen eingesetzt. Im Paket `preprocessing` findet sich die Schnittstelle zu AutoTrace, der SVG-Parser und der graphische Editor, in dem der Benutzer die Atomsymbole spezifiziert. Die Algorithmen zur Dekomposition der Modell-Graphen sowie zum Graph-Matching befinden sich im Paket `matching`. Die Hauptklasse des Programms, die die main-Methode enthält, ist im Paket `reconstruction` enthalten. Dort findet sich auch die Klasse `MolReconstructor`, welche die Funktionalität zur Rekonstruktion einer Struktur aus den gefundenen Subgraph-Isomorphismen zur Verfügung stellt. Die Schnittstelle zu Marvin findet sich schließlich im Paket `marvinInterface`.

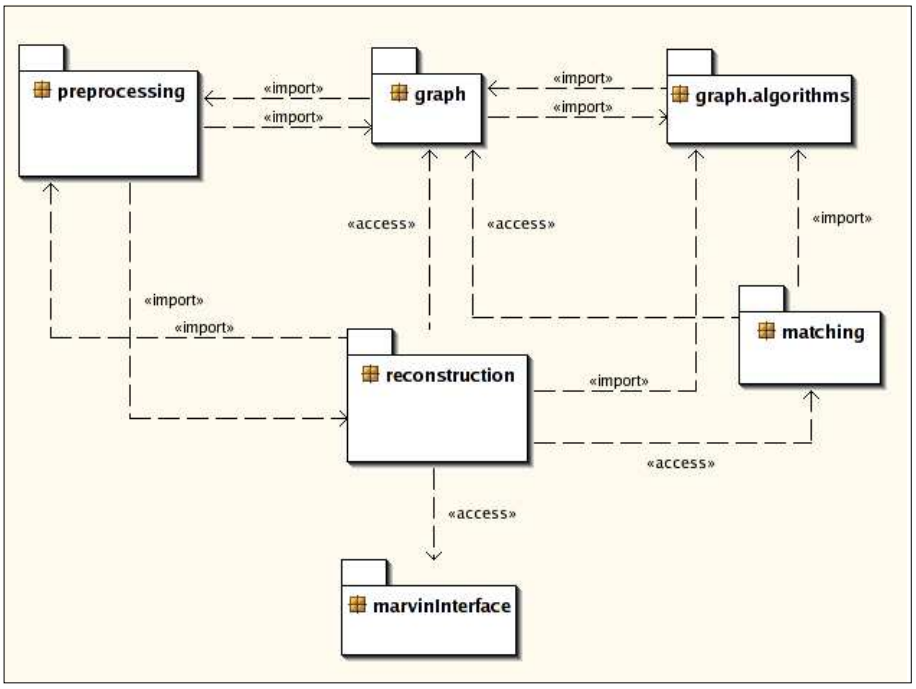


Abbildung A.1: Übersicht der verschiedenen Pakete und ihrer Abhängigkeit voneinander.

Eine Übersicht der Programmstruktur mit den wichtigsten Klassen aus den verschiedenen Paketen wird in Abbildung A.2 als UML-Diagramm gezeigt. Diesem lässt sich die Beziehungen zwischen wichtigen Klassen der vorliegenden Implementierung entnehmen.

Weiter unten ist ein Screenshot der Benutzeroberfläche zu sehen. Im rechten Teil des Bildes wird das Ergebnis der Rekonstruktion angezeigt. Die Eingabe wird im linken Teil der Oberfläche wiedergegeben. Die gesamte Funktionalität des chemischen Struktur-Editors Marvin steht dem Benutzer zur Verfügung.

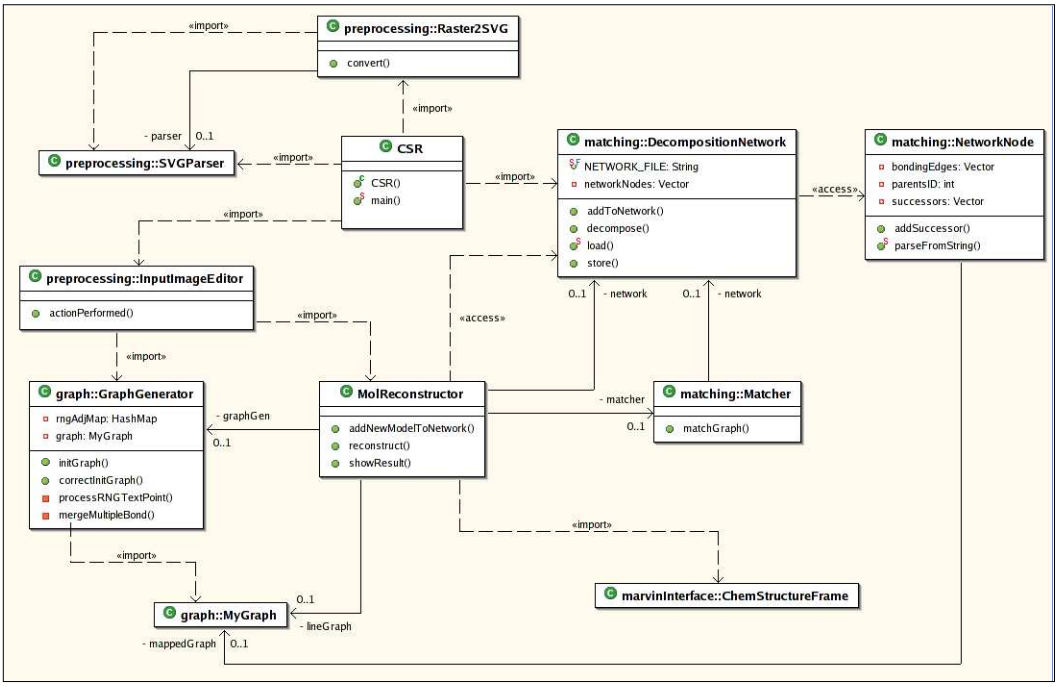


Abbildung A.2: Die wichtigsten Klassen des entwickelten Prototyps in Übersicht.

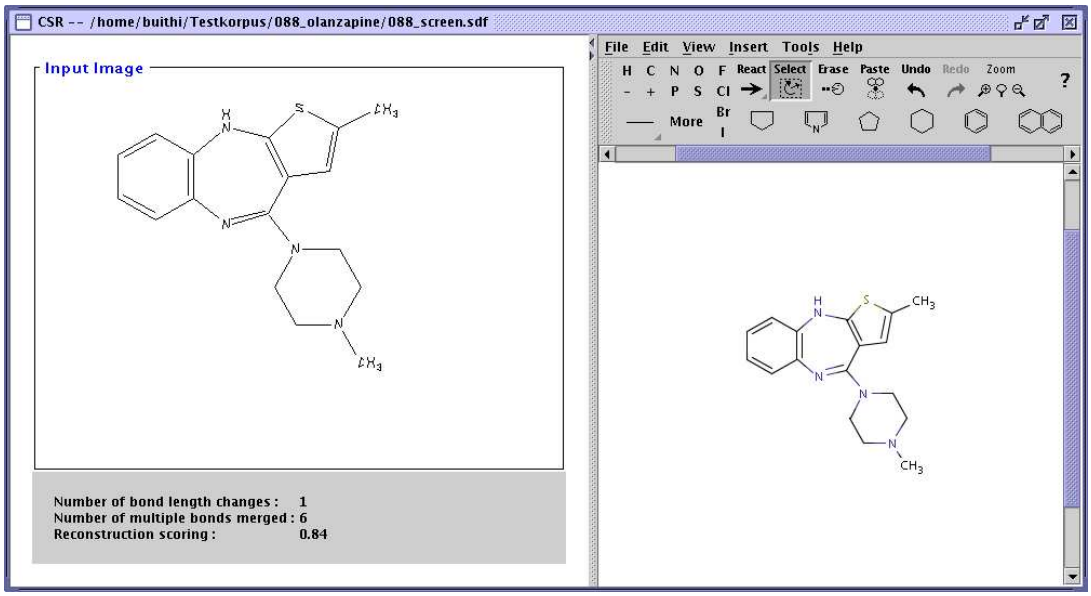


Abbildung A.3: Die Benutzeroberfläche mit der rekonstruierten Struktur.

Literaturverzeichnis

[1] Christian Ah-Soon and Karl Tombre. Network-based recognition of architectural symbols. In *SSPR '98/SPR '98: Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition*, pages 252–261. Springer-Verlag, 1998.

[2] Christian Ah-Soon and Karl Tombre. Architectural symbol recognition using a network of constraints. *Pattern Recognition Letters*, 22(2):231–248, 2001.

[3] Sheila Ash, Malcolm A. Cline, R. Webster Homer, Tad Hurst, and Gregory B. Smith. SYBYL Line Notation (SLN): A Versatile Language for Chemical Structure Representation. *Journal of Chemical Information and Computer Sciences*, 37(1):71–79, 1997.

[4] AutoTrace. http://autotrace.sourceforge.net. Letzter Zugriff am 14.02.2005.

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

[6] Horst Bunke, editor. *Syntactic and Structural Pattern Recogntion*. World Scientific Publishing, 1990.

[7] Horst Bunke. *Handbook of Pattern Recognition and Computer Vision*, chapter 1.5, pages 163–209. World Scientific Publishing, 1995.

[8] Horst Bunke. Recent developments in graph matching. In *ICPR*, pages 2117–2124, 2000.

[9] Horst Bunke, Pasquale Foggia, C. Guidobaldi, Carlo Sansone, and Mario Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 123–132. Springer-Verlag, 2002.

[10] Celinea. http://www.celinea.com/. Letzter Zugriff am 14.02.2005.

[11] William J. Christmas, Josef Kittler, and Maria Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):749–764, 1995.

[12] D. Conte, P. Foggia, C. Sansone, and M. Vento. Graph matching applications in pattern recognition and image processing. In *ICIP03*, pages II: 21–24, 2003.

[13] Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res. (JAIR)*, 1:231–255, 1994.

[14] Luigi P. Cordella and Mario Vento. Symbol and shape recognition. In *GREC*, pages 167–182, 1999.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[16] R. Diestel. *Graphentheorie*. Springer Verlag, 2nd edition, 2000.

[17] P. J. Durand, R. Pasari, J. W. Baker, and Chun che Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2, 1999.

[18] Eclipse. http://www.eclipse.org. Letzter Zugriff am 14.02.2005.

[19] Peter Ertl. Cheminformatics analysis of organic substituents: Identification of the most common substituents, calculation of substituent properties, and automatic identification of drug-like bioisosteric groups. *Journal of Chemical Information and Computer Sciences*, 43(2):374–380, 2003.

[20] T. Fey. Validation of chemical structure recognition software for 2d drawings and a following graphical error curing. Master's thesis, Fachhochschule Bonn-Rhein-Sieg, 2004.

[21] Steven Fortune. *Handbook of Discrete and Computational Geometry*, chapter 20, pages 377–388. CRC Press, 1997.

[22] Johann Gasteiger, editor. *Handbook of Chemoinformatics*. Wiley-VCH, 1st edition, 2003.

[23] G.V. Gkoutos, H.R. Rzepa, O.Adjei R.M. Clark, and H. Johal. Chemical machine vision: Automated extraction of chemical metadata from raster images. *Journal of Chemical Information and Computer Sciences*, 43(5):1342–1355, 2003.

[24] Scale Vector Graphics. http://www.w3.org/graphics/svg/. Letzter Zugriff am 17.02.2005.

[25] Frank Harary. *Graphentheorie*. R. Oldenbourg Verlag GmbH, München, 1974.

[26] P. Ibison, M. Jacquot, F. Kam, A. G. Neville, Richard W. Simpson, Christian A. G. Tonnelier, T. Venczel, and A. Peter Johnson. Chemical literature data extraction: The CLiDE Project. *Journal of Chemical Information and Computer Sciences*, 33(3):338–344, 1993.

[27] Daylight Information Systems Inc. http://www.daylight.com/smiles. Letzter Zugriff am 14.02.2005.

[28] Elsevier MDL Information Systems Inc. http://www.mdl.com/downloads/public/ctfile/ctfile.jsp. Letzter Zugriff am 15.02.2005.

[29] Symbiosys Inc. http://www.symbiosys.ca/clide. Letzter Zugriff am 10.02.2005.

[30] Beilstein Institut. http://www.beilstein-institut.de. Letzter Zugriff am 10.02.2005.

[31] Anil K. Jain, Robert P. W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(1):4–37, 2000.

[32] S.O. Krumke, H. Noltemeier, H.-C. Wirth, and I. Demgensky. Graphentheoretische Konzepte und Algorithmen Skript zur Vorlesung am Lehrstuhl für Informatik 1 der Universität Würzburg, 2003.

[33] Andrew R. Leach and Valerie J. Gillet. *An Introduction to Chemoinformatics*. Kluwer Academic Publishers, 2003.

[34] Josep Lladós and Young-Bin Kwon, editors. *Graphics Recognition, Recent Advances and Perspectives, 5th InternationalWorkshop, GREC 2003, Barcelona, Spain, July 30-31, 2003, Revised Selected Papers*, volume 3088 of *Lecture Notes in Computer Science*. Springer, 2004.

[35] Josep Lladós, Ernest Valveny, Gemma Sánchez, and Enric Martí. Symbol recognition: Current advances and perspectives. In *GREC '01: Selected Papers from the Fourth International Workshop on Graphics Recognition Algorithms and Applications*, pages 104–127. Springer-Verlag, 2002.

[36] RxList LLC. http://www.rxlist.com. Letzter Zugriff am 15.02.2005.

[37] Chemaxon Ltd. www.chemaxon.com/marvin. Letzter Zugriff am 14.02.2005.

[38] Si Wei Lu, Ying Ren, and Ching Y. Suen. Hierarchical attributed graph representation and recognition of handwritten chinese characters. *Pattern Recogn.*, 24(7):617–632, 1991.

[39] Yan Luo and Liu Wenyin. Interactive recognition of graphic objects in engineering drawings. In Lladós and Kwon [34], pages 128–141.

[40] D.Conte M. Vento and et. al. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence.*, 18:265 – 298, 2004. In Press.

[41] Joe R. McDaniel and Jason R. Balmuth. Kekule: OCR-optical chemical (structure) recognition. *Journal of Chemical Information and Computer Sciences*, 32(4):373–378, 1992.

[42] Joe R. McDaniel and Jason R. Balmuth. Automatic interpretation of chemical structure diagrams. In *Selected Papers from the First International Workshop on Graphics Recognition, Methods and Applications*, pages 148–158. Springer-Verlag, 1996.

[43] J.J. McGregor. Backtracking search algorithms and the maximal common subgraph problem. *Software Practice and Experience*, 12:23–34, 1982.

[44] Bruno T. Messmer. *Efficient Graph Matching Algorithms for Preprocessed Models Graphs*. PhD thesis, Inst. of Comp. Science and Appl. Mathematics, University of Bern, 1996.

[45] Bruno T. Messmer and Horst Bunke. Automatic learning and recognition of graphical symbols in engineering drawings. In Rangachar Kasturi and Karl Tombre, editors, *GREC*, volume 1072 of *Lecture Notes in Computer Science*, pages 123–134. Springer, 1995.

[46] Bruno T. Messmer and Horst Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):307–323, 2000.

[47] Martin Negwer and Hans-Georg Scharnow. *Organic-Chemical Drugs and Their Synonyms*. Wiley-VCH, 1st edition, 2001.

[48] Michel Neuhaus and Horst Bunke. Self-organizing graph graph edit distance. In Edwin R. Hancock and Mario Vento, editors, *GbRPR*, volume 2726 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2003.

[49] Michel Neuhaus and Horst Bunke. An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In Ana L. N. Fred, Terry Caelli, Robert P. W. Duin, Aurélio C. Campilho, and Dick de Ridder, editors, *SSPR/SPR*, volume 3138 of *Lecture Notes in Computer Science*, pages 180–189. Springer, 2004.

[50] International of Pure and Applied Chemistry. http://www.iupac.org. Letzter Zugriff am 14.02.2005.

[51] Akio Okazaki, Shou Tsunekawa, Takashi Kondo, Kazuhiro Mori, and Eiji Kawamoto. An automatic circuit diagram reader with loop-structure-based symbol recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(3):331–341, 1988.

[52] Christos H. Papadimitriou. *Computer Complexity.* Addison Wesley, 1994.

[53] Potrace. http://potrace.sourceforge.net. Letzter Zugriff am 23.02.2005.

[54] Alberto Sanfeliu, René Alquézar, J. Andrade, Joan Climent, Francesc Serratosa, and Jaume Vergés-Llahí. Graph-based representations and techniques for image processing and image analysis. *Pattern Recognition*, 35(3):639–650, 2002.

[55] Karl Schwister, editor. *Taschenbuch der Chemie.* Fachbuchverlag Leipzig, 2nd edition, 1999.

[56] Chemical Abstracts Service. http://www.cas.org. Letzter Zugriff am 10.02.2005.

[57] Kim Shearer, Horst Bunke, and Svetha Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognition*, 34(5):1075–1091, 2001.

[58] Kenneth J. Supowit. The relative neighborhood graph, with an application to minimum spanning trees. *J. ACM*, 30(3):428–448, 1983.

[59] Karl Tombre, Salvatore Tabbone, Loïc Pélissier, Bart Lamiroy, and Philippe Dosch. Text/graphics separation revisited. In *DAS '02: Proceedings of the 5th International Workshop on Document Analysis Systems V*, pages 200–211. Springer-Verlag, 2002.

[60] Batik SVG Toolkit. http://xml.apache.org/batik/. Letzter Zugriff am 14.02.2005.

[61] Godfried T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12:261–286, 1980.

[62] J. R. Ullman. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[63] Ernest Valveny and Philippe Dosch. Symbol recognition contest: A synthesis. In Lladós and Kwon [34], pages 368–386.

[64] Ernest Valveny and Philippe Dosch. Performance evaluation of symbol recognition. In Simone Marinai and Andreas Dengel, editors, *Document Analysis Systems*, volume 3163 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 2004.

[65] VextraSoft. http://www.vextrasoft.com/. Letzter Zugriff am 14.02.2005.

[66] Lutz Volkmann. *Graphen und Digraphen: Eine Einführung in die Graphentheorie*. Springer Verlag, 1991.

[67] David Weininger. SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules. *J. Chem. Inf. Comput. Sci.*, 28(1):31–36, 1988.

[68] David Weininger. SMILES. 3. depict. graphical depiction of chemical structures. *J. Chem. Inf. Comput. Sci.*, 30(3):237–243, 1990.

[69] David Weininger, Arthur Weininger, and Joseph L. Weininger. SMILES. 2. algorithm for generation of unique SMILES notation. *Journal of Chemical Information and Computer Sciences*, 29(2):97–101, 1989.

[70] Richard C. Wilson and Edwin R. Hancock. Structural matching by discrete relaxation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(6):634–648, 1997.

Danksagung

Ich möchte mich bei allen herzlich bedanken, die zum Entstehen dieser Arbeit beigetragen haben.

Zuallererst gilt dies meinen Betreuern Herrn Prof. Dr. Hartmut Noltemeier und Herrn Dr. Marc Zimmermann, die mir in vielen Diskussionen zahlreiche Anregungen gegeben haben. Des Weiteren danke ich Herrn Dr. Martin Hoffman für sein großes Vertrauen und seine Motivationskünste.

Zuletzt gilt mein besonderer Dank meiner Familie und C., die mich während meines Studiums uneingeschränkt unterstützt haben.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel verwendet habe.

Würzburg, 29.03.05