Eberhard Karls
Universität
Tübingen

Algorithmen zur Graphmustersuche für das Netzwerkanalysetool BiNA

Diplomarbeit

Erstkorrektor (Informatik)
Prof. Michael Kaufmann
Zweitkorrektor (Biologie)
Prof. Rolf Reuter
außerdem betreut von
Prof. Oliver Kohlbacher, Dipl.-Inf. Torsten Blum

vorgelegt von
Carsten Henneges

Fakultät für Informations- und Kognitionswissenschaften
Wilhelm-Schickard Institut für Informatik
Arbeitsbereich Paralleles Rechnen

Tübingen, 19. Oktober 2006

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegeben Hilfsmittel benutzt zu haben.

Tübingen, 19. Oktober 2006

Zusammenfassung

In dieser Diplomarbeit werden Algorithmen zur Graphmustersuche für das Netzwerkanalysetool BiNA entwickelt. Es handelt sich dabei um ein Visualisierungstool für Daten im BN++-Modell, das diese mit Hilfe von Graphen darstellt und mit den Layout-Algorithmen der *yfiles* zeichnet. Die anzeigbaren Daten liegen in einer, in Tübingen gepflegten, MySQL-Datenbank vor und stammen aus elf verschiedenen frei zugänglichen biologischen Datenbanken.
Für dieses Tool wurden neue Algorithmen entwickelt und implementiert, die Graphen auf Gleichheit testen, das Vorkommen eines Graphen in einem anderen suchen oder für einen gegebenen Graphen randomisiert alle Teilgraphen bestimmter Größe zu bestimmen.
Mit Hilfe dieser Algorithmen wurden daraufhin die biologischen Netzwerke der Organismen *Arabidopsis thaliana*, *Drosophila melanogaster*, *Sarachomyces cerevisiae* und *Escherichia coli* analysiert. Die erhaltenen Ergebnisse zeigen, dass die randomisierte Teilgraphsuche stark vom verwendeten Datenmodell und der verwendeten Teilgraphdefinition abhängig ist.
Der in dieser Arbeit entwickelte Graphisomorphiealgorithmus zeigt eine neue Richtung auf, mit der dieses Problem gelöst werden kann. Seine Leistungsfähigkeit erweist sich für den Gebrauch in Datenstrukturen, wie der „Move-to-Front“-Liste, die in dem randomisierten Algorithmus verwendet wird, als ausreichend.

Abstract

This masterthesis develops algorithms for the network analysis tool BiNA. BiNA is an visualization tool for BN++-data, which uses graphs for representation and the *yfiles* graph-layouting algorithms for drawing. The available data is stored in an MySQL database in Tübingen and contains data from eleven free biological databases.
For this tool, new algorithms for testing the equality of graphs, searching one graph in another and calculating randomized all subgraphs of an certain size for an given graph were developed.
Using these algorithms, the biological networks of *Arabidopsis thaliana*, *Drosophila melanogaster*, *Sarachomyces cerevisiae* and *Escherichia coli* were analyzed. The results show, that the randomized subgraph search mainly depends on the used data model as on the used subgraph definition.
The developed algorithm for graph isomorphism shows a new direction to solve the problem. The achieved performance proved to be adequate for use in data structures like move-to-front lists, which are used by the random sampling algorithm.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BFS	Breath First Search
CSI	Common Subgraph Isomorphism
DFS	Depth First Search
ESI	Exact Subgraph Isomorphism
FPTAS	Fully Polynomial Time Approximation Scheme
SQL	Structured Query Language

Einleitung

Moderne molekularbiologische Methoden und groß angelegte Forschungsprojekte mit hohen Budgets liefern im Bereich der Biologie eine wahre Flut an Daten, wie Swiss-Prot [3], RCSB [23], NCBI und andere zeigen. Damit haben Wissenschaftler nun zunehmend die Möglichkeit, die Zusammenhänge und deren Strukturen zu erforschen und zu modellieren. Je komplexer sich jedoch die Zusammenhänge darstellen, umso wichtiger sind auch leistungsfähige Visualisierungswerkzeuge, um den Überblick über die sich anhäufenden Daten zu behalten. Sie sollten einen einfachen und schnellen Zugang zu den ihnen zugrunde liegenden Daten bereitstellen, ohne aber Einschränkungen bei der Informationsdarstellung vorzunehmen. Zudem müssen sie in der Lage sein, Datensätze eventuell heterogener Datenquellen einzulesen und für den Menschen geeignet aufzubereiten.
Als eine der Möglichkeiten zur Modellierung haben sich Graphen herausgestellt. In der Informatik gehören sie zu den mächtigsten und flexibelsten Datenstrukturen überhaupt. Einige Varianten sind in der Lage, turing-mächtige Rechnungen durchzuführen, während andere zur Lösung analytischer Probleme verwendet werden können. Allen gemeinsam ist, dass sie aus zwei allgemeinen Grundelementen konstruiert sind: Knoten und Kanten. Dabei ist es in Erweiterungen möglich, diese Grundelemente je nach Bedarf mit Beschriftungen (*Labels*) und anderen Eigenschaften auszustatten. Beispiele solcher Graphen sind z.B. Protein-Interaktionsnetzwerke, Genregulations-Netzwerke und metabolische Pfade, wie sie unter anderem in KEGG [30] dargestellt werden, aber auch die Petri-Netze Petri-Netze [35] der theoretischen Informatik.
Zu den oben angesprochenen Visualisierungstools gehören Cytoscape [45], Osprey [5], VisAnt [24] und BiNA (Biological Network Analysis), das Visualisierungstool zu BN++ [40]. Sie alle visualisieren Daten mit Hilfe von Graphen und bieten Zugang zu den unterschiedlichsten Datenquellen. In Bezug zu den Datenquellen wird generell zwischen *Browser*-, *Mediator*- und *Data Warehouse*-Architektur unterschieden, die in Kapitel 1 erläutert werden. Zu der letzteren Kategorie zählen BN++ und BiNA, die Gegenstand der vorliegenden Arbeit sind.
Ziel dieser Diplomarbeit war es, BiNA um Analysefunktion zur Graphmustersuche zu erweitern, sowie die in der BN++-Datenbank gespeicherten Informationen auf häufig auftretende Muster zu untersuchen. Dazu wurden neue Algorithmen im Bereich der Subgraph- und Graphisomorphie entwickelt und anschließend mit einem randomisierten Algorithmus zur Mustersuche kombiniert.
Die Diplomarbeit ist wie folgt strukturiert: Zunächst wird der Hintergrund der Graphmustersuche kurz motiviert, dann ein Einblick in das BN++-Umfeld gegeben. Es werden verschiedenen Softwarearchitekturen vorgestellt, die zur Bereitstellung von Informationen existieren. Danach werden BN++ und BiNA beschrieben, bevor in Kapitel 2 eine Beispiel BN++-Modellierung, sowie die verwendeten Datensätze vorgestellt werden. Das Hauptkaptitel 3 befasst sich mit den neu entwickelten Algorithmen. Die einzelnen Unterkapitel enthalten jeweils einen Abschnitt mit formalen Betrachtungen, sowie eine vorgezogene Diskussion des beschriebenen Verfahrens. Der zugehörige Pseudocode ist jeweils im Anhang zu finden. In Kapitel 4 werden die Laufzeiten und Analyseergebnisse vorgestellt und diskutiert. Kapitel 5 schließt die Arbeit mit einer Diskussion der gesamten Arbeit ab.

Kapitel 1

Hintergrund

Seit einiger Zeit setzt sich in der Bioinformatik ein neuer Trend durch: die Systembiologie. Diese Teildisziplin beschäftigt sich mit den Prinzipien ganzer biologischer Systeme. Dabei werden nicht nur molekulare Interaktionssysteme betrachtet, sondern auch mikro- und makroskopische Systeme, wie z.B. das neuronale Netzwerk von *C.elegans* oder Nahrungsketten verschiedener Ökosysteme (vgl. [2]). Nicht mehr das Protein oder das Gen stehen im Mittelpunkt der Forschung, sondern deren Kombination in einem zuverlässig funktionierenden System.
Die besonderen Herausforderungen bestehen dabei nicht darin, ausreichend Daten zu beschaffen (das wird von anderen, groß angelegten Forschungsprojekten erledigt), vielmehr bestehen sie darin, diese Daten sinnvoll miteinander abzugleichen und in geeigneten Modellen zu kombinieren. Dazu müssen die Daten aus den verschiedenen verfügbaren Quellen zusammengetragen und in einem einheitlichen Datenmodell gespeichert werden, ohne evtl. benötigte Information zu verlieren. Auf der Grundlage eines solchen Datenmodells können dann die gespeicherten Systeme analysiert und für die Erstellung analytischer Modelle verwendet werden. Diese müssen anschließend wieder mit experimentellen Ergebnissen verifiziert werden.
Als Modellierungsansätze werden, neben der traditionell mathematischen Methode mit Differentialgleichungen, auch Graphen eingesetzt. Diese verknüpfen miteinander interagierende Systemkomponenten durch Kanten und stellen eine sehr flexible und anschauliche Alternative dar. Für die Visualisierung sind Graphen sehr gut geeignet, da es mittlerweile eine große Anzahl leistungsfähiger Layout-Algorithmen gibt, mit denen sie übersichtlich dargestellt werden können. Diese Algorithmen können u.a. auch dazu verwendet werden, bestimmte Merkmale des Netzwerks hervorzuheben und so wichtige Hinweise für eine Modellierung liefern.
Eine weitere Aufgabe besteht darin, den Komponenten der Graphen eine, dem untersuchten Problem angepasste, Bedeutung zu geben, so dass die verschiedenen Analysemethoden aussagekräftige Ergebnisse liefern, die auch überprüft werden können bzw. Hinweise liefern, die im Labor von Nutzen sind. Zudem sollten sie auch einen formalen Zugang zu den dargestellten Systemen bieten, so dass auch hypothetische Systeme konstruiert und experimentell überprüft werden können.
Auch die automatisierte Analyse dieser Graphen erwächst aus diesem Zusammenhang. Dabei steht im Mittelpunkt die Frage, ob die betrachteten Netzwerke aus Grundkomponenten mit definierter Funktionalität bestehen. Dies kann nur in Netzwerken mit geeigneter Semantik der Komponenten erfolgreich sein. Hier ist zu definieren, was eigentlich gesucht wird, bzw. welche Eigenschaften die Ergebnisse haben müssen, um einen Erkenntnisgewinn darzustellen. In diesem Kontext hat die Suche nach den häufigen Teilgraphen (*Frequent (Sub)Pattern*) einer Netzwerkmodellierung begonnen, von denen erhofft wird, dass sie diese Erwartungen erfüllen. Bisher bekannte Ergebnisse [2] lassen diese Hoffnungen als durchaus berechtigt erscheinen, obwohl auf diesem Gebiet noch viel Forschungsbedarf besteht.

Kapitel 2

Material und Methoden

2.1 Einleitung

Es ist schon seit langem bekannt, dass die Bewältigung einer wahren Datenflut, die durch z.T. automatisierte High-Throughput-Projekte mit den neuesten molekularbiologischen Methoden und hohen Budgets entstand, eines der zentralen Themen der Bioinformatik ist. Diese Datenflut hat Ausmaße, die es Wissenschaftlern nicht mehr ermöglicht, den kompletten Überblick über den aktuellen Stand der Forschung zu behalten und birgt weiterhin das Risiko, dass erfasste Zusammenhänge unausgewertet bleiben und somit kein Erkenntnisgewinn aus ihnen gezogen wird.
Programme wie Cytoscape [45], Osprey [5], VisAnt [24] und ähnliche bemühen sich, den Zugriff und den Umgang mit dieser Datenflut zu vereinfachen, indem sie ausgefeilte Visualisierungtechniken, wie z.B. Graphlayoutalgorithmen verwenden, um biochemische Netzwerke übersichtlich darzustellen. Dabei gibt es im Allgemeinen drei Grundarchitekturen, denen diese Systeme folgen: *Navigatoren*, *Mediatoren* und *Data warehouses*. Alle drei sollen im Folgenden kurz vorgestellt werden:

Navigators Diese Architektur stellt lediglich einen rudimentären Zugang zu der zugrunde liegenden Datenquelle bereit. Meist besteht dieser in Form eines Web-Interfaces mit der Möglichkeit zur Suche nach Schlüsselbegriffen, sowie einer einfachen Navigation durch die Ergebnisdaten. Die wohl populärsten Beispiele dieser Architektur sind SRS [17], BioNavigator [14] und Entrez [20].

Mediators Diese Architektur bietet ein komfortables Interface zu der angebotenen Datenquelle. Dabei müssen Halter der Datenbank und Hersteller des Mediators nicht dieselbe Organisation sein. Durch die öffentliche Bereitstellung der Daten im Internet können beliebige Programme Web-Anfragen generieren und die erhaltenen aktuellen Abfrageergebnisse auswerten. Dabei muss allerdings oft das Datenmodell des Anbieters verwendet werden, wodurch das Zusammenspiel mit anderen Datenquellen erschwert wird. Zusätzlich ist diese Architektur von der angebotenen Performance des Datenhalters abhängig. Beispiele derartiger Anwendungen sind Discovery Link [21], TAMPIS [47] und BioMediator [12].

Data warehouses Diese Klasse integriert komplette Datenbestände heterogener Datenquellen unter Berücksichtigung der entsprechenden Datemodelle in einer eigenen, lokalen Datenbank mit eigenem Datenmodell. Dies ermöglicht nicht nur eine bessere Performancekontrolle, sondern bietet zudem die Möglichkeit einer integrativen Analyse dieser Datensätze, so dass tatsächlich neue Zusammenhänge erschlossen werden können, die bei einer einzelnen Betrachtung der Ausgangsdatenquellen nicht möglich waren. Nachteil dieser Architektur ist die konsistente Datenintegration durch Kuratoren, sowie die Notwendigkeit des Datenabgleiches. Ein weiterer zusätzlicher Aufwand entsteht bei Änderungen im Daten-

modell der Anbieter, so dass komplette Module dieser Architektur evtl. neu geschrieben werden müssen, oder komplett entfallen. Beispiele in dieser Kategorie sind GUS [9], Biozon [4] und BN++ [40].

2.2 BN++

Das BN++ ging aus dem *BioMiner*-Projekt [40] der Universität Saarland mit der Universität Tübingen hervor. Ziel des Open-Source-Projektes, das unter Sourceforge [1] erhältlich ist, war es, die verschiedenen heterogenen Datenbanken in einem einheitlichen Daten-Modell, *BioCore* aus dem später das BN++-Modell wurde, zu speichern. Dieses objekt-orientierte Modell besteht aus einer C++/Java-Klassenhierarchie, welche die notwendigen Methoden für das Laden und Speichern in bzw. aus einer SQL-Datenbank bereitstellen. Es sollte in der Lage sein, jeden möglichen molekularbiologischen Zusammenhang beschreiben zu können.
Auf der Server- bzw. der Datenbankseite waren Analyseprogramme vorgesehen, welche aus den gespeicherten Daten weiterführende Informationen berechnen. Eines der ersten dieser Programme war *PathFinder*, mit dessen Hilfe aus neusequenzierten Organismen metabolische Pfade berechnet werden konnten. Zur Zeit sind weitere Analysetools in der Entstehung, zu denen auch der Mustersuche-Algorithmus dieser Diplomarbeit zählt.
Damit die aus den heterogenen Datenquellen, wie z.B. KEGG [30] mittels Adaptoren importierten Informationen konsistent in der Datenbank der Universität Tübingen gespeichert werden können, wurden Programme zum Zusammenführen mehrerer Datenbanken zu einer größeren entwickelt. Ziel dieser Programme ist es, den Datenbestand in einem konsistenten Zustand zu halten, was sich angesichts der vielen verschiedenen Quellen als große Herausforderung erwies.
Bisher wurden folgende Datenbestände importiert:

- Sequenzdatenbanken: SwissProt [3], RefSeq [43]
- Pfaddatenbanken: KEGG [30], BioCyc [32], TransPath [34]
- Proteininteraktionsdatenbanken: DIP [44], MINT [51], IntAct [22], HPRD [16]
- Transkriptionsfaktor-Datenbank: TransFac [37]
- Proteinklassifikations-Datenbank: InterPro [41]

Zur Visualisierung dieser sollte ein plattform-unabhängiger Java-Client die in der Datenbank gespeicherten Daten laden und mittels der bekannten und mächtigen Visualisierungsbibliothek, den *yfiles* [50] darstellen. Zudem war eine XML-Kommunikationsschnittstelle des Clients mit dem Datenbankserver geplant. In der Diskussion steht mittlerweile SOAP als Kommunikationsprotokoll, mit dessen Hilfe die Datenbankobjekte über das Internet verschickt werden sollen.
Abbildung 2.1 veranschaulicht die BN++-Architektur, deren Kern das BN++-Datenmodell ist, welches im Kapitel 2.4 beschrieben werden soll.

2.3 BiNA

Bei BiNA handelt es sich um das zu BN++ gehörende Visualisierungstool, das aus dem früheren *PathViewer*-Programm hervorging. Es wird an der Universität Tübingen von Andreas Gerasch entwickelt. BiNA bietet dem Benutzer die Möglichkeit, interaktiv die importierten Datenbestände zu erforschen. Dabei kommen die in den *yfiles* angebotenen Layout-Algorithmen zur Anwendung, mit deren Hilfe der gerade betrachtete Datensatz optimal dargestellt wird, so dass Zusammenhänge leichter wahrgenommen werden können. Weiterhin bietet die BiNA-Plattform einen zusätzlichen Abgleich mit aktuellen

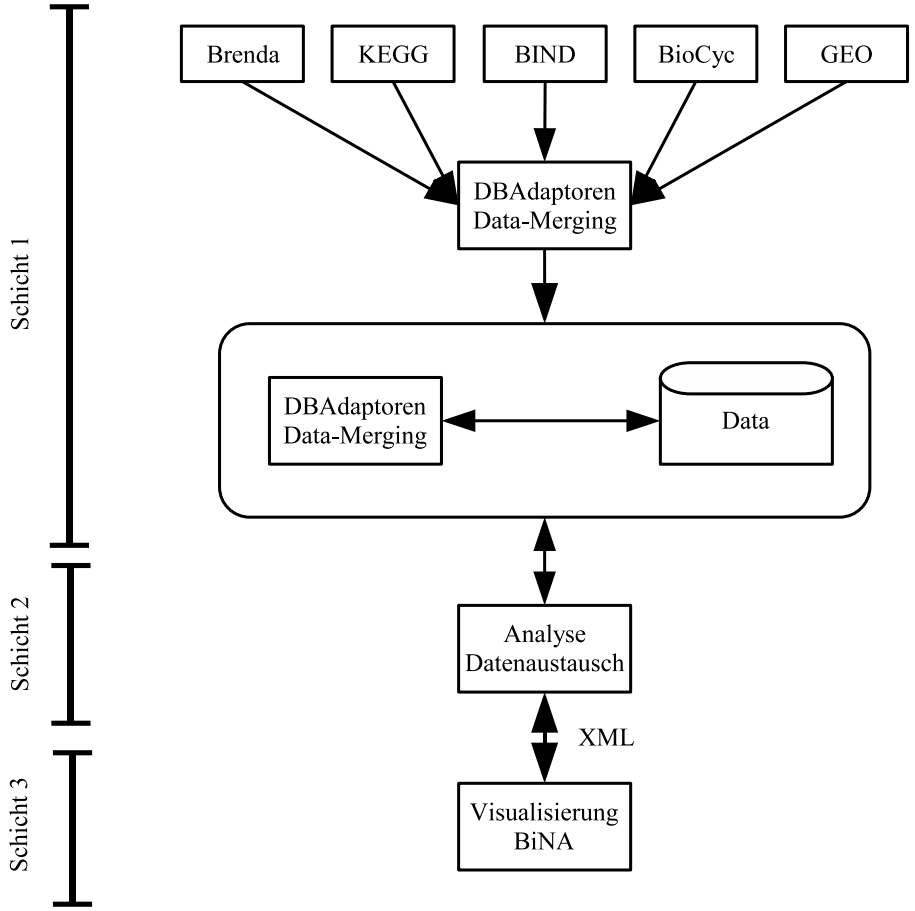


Abbildung 2.1: BN++-Architektur

Datenquellen aus dem Internet. So werden automatisch Hintergrundinformationen von PDB [23] und ähnlichen Quellen heruntergeladen und angezeigt. Auch die Visualisierung von Genexpressionsleveln ist inzwischen möglich. So werden Proteine mit höherer Expression mit anderen Farbschattierungen unterlegt als Proteine mit geringer oder keiner Expression. Abbildung 2.2 zeigt einen Screenshot von BiNA. Für Entwickler stellt BiNA eine ausgefeilte und flexible Plugin-Architektur zur Verfügung. So lassen sich unterschiedliche Graph-Bibliotheken über ein globales Interface in BiNA einbinden, ohne dass dadurch andere Plugin-Entwickler beeinträchtigt werden. Die im Rahmen dieser Diplomarbeit implementierten Algorithmen sind in BiNA als *MotifPlugin* integriert worden und können somit auf jeden beliebigen geladenen Datensatz angewendet werden.

2.4 Das BN++-Datenmodell

Das BN++-Datenmodell, das aus dem Vorgängerdatenmodell *BioCore* entstanden ist, stellt, wie oben schon erwähnt, den zentralen Kern des Projektes dar. Es handelt sich dabei um eine Klassenhierarchie, mit deren Hilfe biologisch relevante Daten modelliert und in einer Datenbank abgespeichert werden können. Die Modellierung ist darauf ausgelegt, möglichst alle biologisch relevanten Informationen in Bezug zueinander speichern zu können. Auch hier wird von der Idee Gebrauch gemacht, Graphen zur Modellierung zu verwenden.

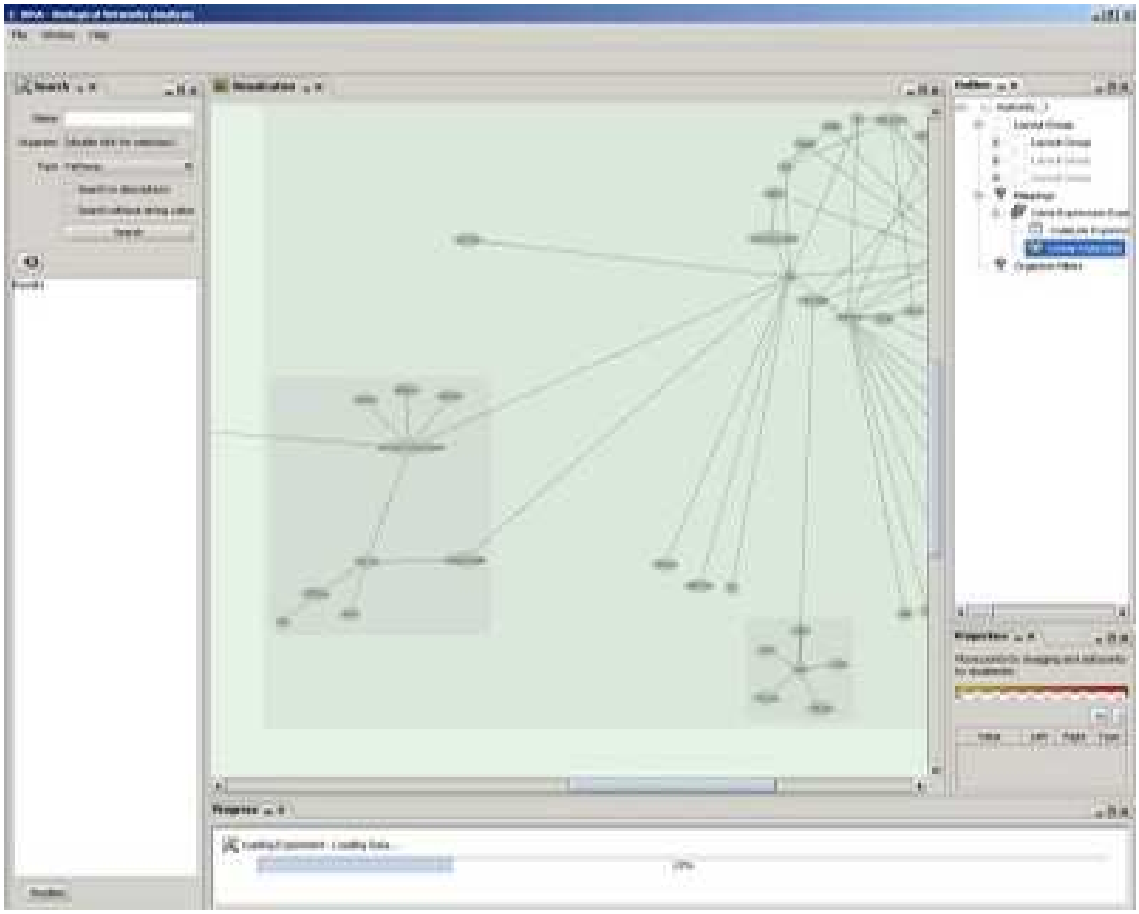


Abbildung 2.2: BiNA-Screenshot

Das BN++-Modell beschreibt einen sogenannten *Hypergraphen*, also einen Graphen dessen Knoten unterschiedlichen Typs sind. In BN++ stehen zwei Typen zur Auswahl: auf der einen Seite die *Participants* und auf der anderen Seite die *Events*. Beide Entitäten sind mittels gerichteter Kanten eines Typs, den *Roles*, miteinander verbunden, wobei die Events zusätzlich auch untereinander über Kanten verknüpft werden können, um eine Ereignisshierarchie modellieren zu können.
Grundidee ist, dass alles in der Biologie über Ereignisse, den *Events*, und deren Teilnehmern, den *Participants*, beschrieben werden kann. Dabei nimmt jeder Teilnehmer eines Ereignisses in einer bestimmten Rolle, der *Role*, daran teil. Diese Rollen stellen die gerichteten Kanten des Hypergraphen dar und beschreiben die Art und Weise, in welcher der Teilnehmer in dem Ereignis wirkt. Abbildung 2.3 illustriert diesen Zusammenhang als UML-Diagramm.
In dem Diagramm wird deutlich, dass jedes Objekt von *Thing* ableitet. Über das *Role*-Objekt wird eine 1:1-Relation zwischen *Participants* und *Events* hergestellt. Da aber *Event* ebenfalls von *Participant* ableitet, kann also auch eine 1:1-Relation zwischen *Events* beschrieben werden. Von diesen Basisklassen leiten speziellere Klassen zur näheren Beschreibung ab. Schlussendlich stellt die oberste Basisklasse *PersistentObject* die Grundlage für die Speicherung in der SQL-Datenbank bereit.

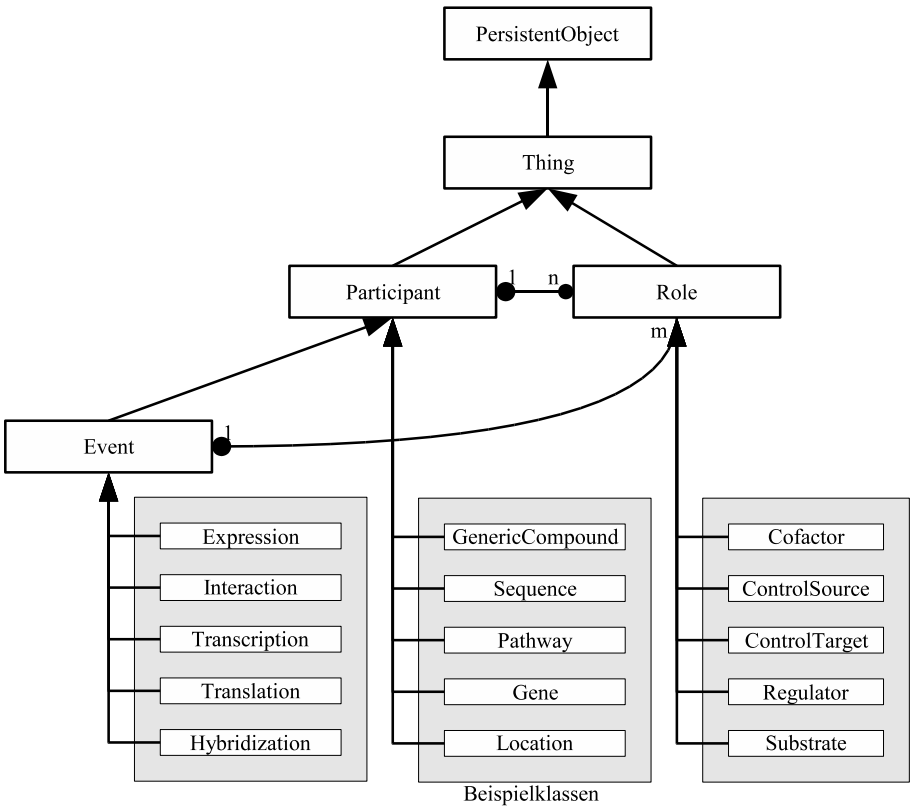


Abbildung 2.3: UML-Diagramm der BN++-Klassenhierarchie

2.5 Eine Beispielmodellierung

In diesem Abschnitt wird das BN++-Modell mit einer Beispielmodellierung vorgestellt. Es wird der Fall modelliert, in dem die Transkriptionsfaktoren A und B einen Komplex bilden, der an einen Genabschnitt C bindet. Daraufhin wird die Transkription des Genes in die mRNA C veranlasst. Diese wird vom Ribosom in ein Protein C übersetzt, das wiederum in der Lage ist, an Gen C zu binden und dadurch als Repressor zu agieren. Es handelt sich also um einen negativen Feedback-Loop, der in Abbildung 2.4 als Hypergraph dargestellt ist.
Dabei sind drei Knotensorten hervorgehoben: Die roten Knoten stehen für `Events`, an denen die blauen und grünen `Participant`-Knoten teilnehmen. Um zu verdeutlichen, dass der BN++-Graph differenzierter modelliert ist, sind die Proteinkomplexe in Grün von den anderen `Participants` in Blau, hauptsächlich Protein und Transkriptionsfaktoren, unterschieden. Diese Knoten sind über gerichtete Kanten, den Rollen (`Role`) verbunden. Aus einem `Event` austretende Kanten stehen für die Rolle als `Product`, in ein `Event` eingehende Kanten für die Rolle `Educt`. Auch diese können über abgeleitete Klassen weiter differenziert werden. Zudem kann derselbe `Participant`, wie das bei den Polymerasen der Fall ist, mit verschiedenen Rollen am gleichen `Event` teilnehmen.
An diesem Beispiel wird die Leistungsfähigkeit des BN++-Modells deutlich. Es zeigt sich jedoch, dass die Hypergraphen sehr groß werden, selbst wenn nur wenige Informationen darzustellen sind. So haben die im Ergebnis-Kapitel angegebenen Datensätze zu *E.coli*, *S.cerevisiae*, *D.melanogaster* und *A.thaliana*, den Modellorganismen der Molekularbiologie, die in Tabelle 2.1 angegebenen Größen. Bei dem anderen untersuchten Datensatz handelt es sich um den *E.coli*-Datensatz von Uri Alon ohne Autoregulation (vgl. Kapitel 3.3.2), sowie um die Protein-Interaktionsnetzwerke für die vier oben angegebenen Organsimen. In den letzteren existiert genau dann eine Kante zwischen zwei Proteinen, wenn diese miteinander

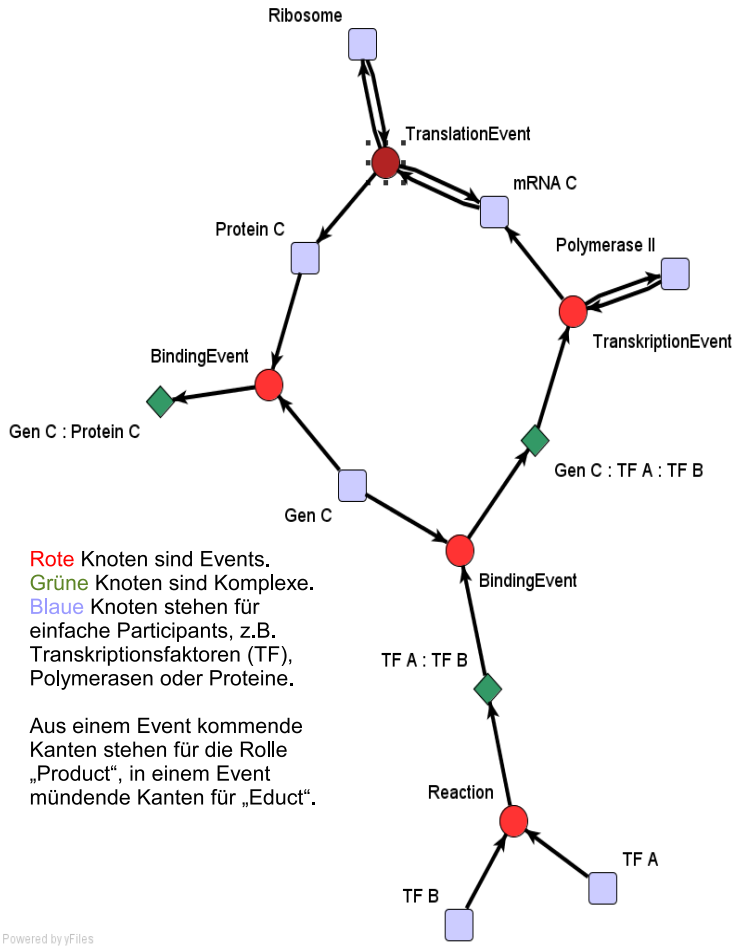


Abbildung 2.4: Die Beispielmodellierung verdeutlich, dass im BN++-Modell biologische Regulation sehr viel differenzierter dargestellt werden kann, als in anderen bekannten Modellierungen. Es wird in dem modellierten negativen Feedback-Loop zwischen roten Events, grünen Komplexen und blauen Participants unterschieden.

interagieren.

Datensatz	Knoten $\mid N \mid$	Kanten $\mid E \mid$
E.coli (BN++)	3014	3474
S.cerevisiae (BN++)	1174	1246
D.melanogaster (BN++)	2037	2562
A.thaliana (BN++)	1626	2265
E.coli (Protein-Interaktion)	2083	4156
S.cerevisiae (Protein-Interaktion)	757	1648
D.melanogaster (Protein-Interaktion)	1332	6544
A.thaliana (Protein-Interaktion)	1132	8943
E.coli ohne Autoregulation	424	519

Tabelle 2.1: Größen der biologischen Eingabe-Graphen

Kapitel 3

Algorithmen

In den folgenden Abschnitten werden die drei entwickelten Algorithmen nun näher beschrieben. Es handelt sich dabei ausschließlich um Algorithmen, die von allgemeinen, unbeschrifteten und ungerichteten Graphen ausgehen und keine Voraussetzungen über Art und Struktur derselben benötigen. Allerdings sind sie alle in dem Sinne generisch, indem sie ohne Weiteres zusätzliche Informationen in ihren Ablauf einbeziehen und dadurch Vorteile erhalten können.
Bei der Entwicklung wurde von dem Standpunkt ausgegangen, dass der erste Schritt in Richtung der Entwicklung eines Subgraphisomorphie- oder Frequent Subpattern-Algorithmus ein Algorithmus zur Graphisomorphie sein muss. Dies erwies sich als nur bedingt richtig. Viele der dort verwendeten Prinzipien werden auch in den beiden anderen Algorithmen eingesetzt, obwohl die Graphisomorphie-Lösungsstrategie sich dort als schwer anwendbar erwies.
Die eingesetzten Prinzipien beinhalten eine neue Art der Klassifikation von Knoten und Kanten, die Interpretation des Eingabeproblems mittels eines funktionalen Programms, dessen Ausgabe eine neue Art „stellen-loser“ Zahlen, sogenannter *Merkmalszahlen*, ist, sowie den Gebrauch einer Backtracking-Strategie. Alle diese Prinzipien finden sich im Graphisomorphie-Algorithmus, für den sie auch entwickelt wurden. Daher wird mit dessen Beschreibung begonnen.
Der Pseudocode zu allen Algorithmen ist im Anhang zu finden. In den formalen Betrachtungen sind immer zwei Graphen $G = (V_G, E_G)$, das *Muster*, und $H = (V_H, E_H)$, der sogenannte *Host*, gegeben. Es wird dabei immer das Muster mit dem Host verglichen, bzw. dessen Existenz im Host gesucht. Lediglich bei der Frequent Subpattern-Suche ist nur das Muster G gegeben.

3.1 Graphisomorphie

3.1.1 Einleitung

Das Problem, zwei Graphen auf Gleichheit zu prüfen, besteht im Prinzip darin, zu prüfen, ob es eine Abbildung aus der Menge der Knoten des einen Graphen in die Knotenmenge des anderen gibt, so dass die Abbildungen zweier benachbarter Knoten im anderen Graphen auch benachbart sind. Es sind einige hinreichende Kriterien für Ungleichheit bekannt, wie zum Beispiel die *Gradfolge der Knoten*. Diese kann benutzt werden, nicht isomorphe Kandidaten zu erkennen, nicht aber um die Gleichheit wirklich nachzuweisen.
Bisher ist auch über die Komplexitätsklasse dieses Problems nur soviel bekannt, dass man noch nicht weiß, ob es *NP*-vollständig ist. Von einer Unterklasse der Graphisomorphie, nämlich der Baumisomorphie, konnte gezeigt werden, dass sie in P liegt und in $O(n)$ lösbar ist. Der dort verwendete Ansatz besitzt Ähnlichkeiten mit dem im folgenden vorgestellten Verfahren. Trotzdem verwenden bisherige Programme, die Graphen vergleichen (z.B. `nauty` [38]), komplizierte Regeln, Verfeinerungs- und Aus-

schlussverfahren, die einem Backtrackingverfahren Informationen bei der Alternativenauswahl liefern, so dass sie im Mittel das Problem schnell lösen können.
Der folgende Graphisomorphiealgorithmus benutzt im Kern eine lokale Klassifikation der Knoten. Daher beginnt das Kapitel zunächst mit den verschiedenen Knotenklassifikationsmöglichkeiten. Mit Hilfe der erhaltenen Klassifikationen wird ein generisches Verfeinerungsverfahren beschrieben, dass die Eigenschaften aller Knoten parallel genauer auftrennt, so dass alle Knoten globale Positionsinformation besitzen. Dieses ist genau genommen ein rekursives, funktionales Programm, dass die Eingabegraphen mit Hilfe eines Zahlensystems interpretiert und eine Abbildung von Knoten nach Zahlen liefert. Die dahinter stehende Idee ist, dass gleiche Eingaben zur selben Ausgabe führen. Dazu wird ein Merkmalszahlensystem verwendet, dessen Sinn in einer Entkopplung von der nicht feststehenden Reihenfolge der Knotennachbarn zu sehen ist. Sollten die Ausgaben übereinstimmen, besteht der letzte Schritt in dem Versuch einer Aufzählung mittels eines Backtrackingverfahrens. Dieses verwendet die Ausgabe der funktionalen Interpretation für seine Entscheidungen. Interessanterweise enthüllt die funktionale Interpretation sogar dem Graph inhärente Symmetrieeigenschaften. Das Kapitel endet mit der Beschreibung des Backtrackingalgorithmus und einer Betrachtung der Graphsymmetrien.
Die in dieser Arbeit verwendete Definition eines Graphen lautet folgendermaßen:

Definition 1. *Ein ungerichteter **Graph** ist ein Tupel $G = (V,E)$ mit V, der endlichen **Knotenmenge**, und $E = \{\{u,v\} \mid u \in V, v \in V\}$, der endlichen **Kantenmenge** ungerichteter Kanten.*

3.1.2 Die Knotenklassifikation

Eine der ersten Ideen die man versuchen könnte, wenn man sich mit Graphisomorphie beschäftigt, wäre für jeden Knoten aus G alle möglichen Knoten in H zu speichern, und dann sukzessive solche Knoten zu entfernen, die nicht mehr passen. Es ist leicht einzusehen, dass ein solches Verfahren enorm viel Speicherplatz benötigen und wahrscheinlich eine geringe Performance haben würde. Dennoch hat es den Vorteil, dass zumindest zu Beginn nur lokale Entscheidungen nötig sind. Alles, was dazu nötig wäre, ist eine *Klassifikation* der Knoten in beiden Graphen, so dass Knoten, die in G und in H dieselbe *Klasse* haben, lokal aufeinander abgebildet werden dürfen. Formal:

Definition 2. *Ein **Klassifikator** ist eine Funktion $c : V \longrightarrow M$ aus einer Knotenmenge V in eine beliebige Merkmalsmenge M, so dass gilt*

$$\forall u,v : c(u) = c(v) \Longleftrightarrow u \text{ und } v \text{ sind bezüglich } M \text{ lokal isomorph} \tag{3.1}$$

Die Knoten u und v müssen dabei nicht unbedingt aus unterschiedlichen Graphen stammen. Im folgenden seien M^G und M^H die zu G und H gehörigen Merkmalsmengen. Die Elemente dieser Merkmalsmengen werden mit $m^G \in M^G$ und $m^H \in M^H$ bezeichnet.

Der *Grad* eines Knotens ist sicherlich ein solches *Merkmal*, was auch aus dem Kriterium der Gradfolge ersichtlich wird. Allerdings sagt es nichts über die Topologie der Nachbarn aus. Will man dennoch eine lokale Betrachtung anstreben, so kann man sich auf den durch die direkten Nachbarn eines Knotens induzierten Teilgraphen beschränken und diesen zur Klassifikation verwenden. Wir definieren also:

Definition 3. *Die **Nachbarschaft** eines Knotens ist gegeben als*

$$N(u) := \{v \in V \mid \exists \{u,v\} \in E\}. \tag{3.2}$$

*Der **induzierte Nachbarschaftsgraph** $G(u)$ ist der Teilgraph, der nur aus den Knoten $N(u)$ und den zwischen diesen existierenden Kanten besteht. Er zerfällt in **einzelne Knoten**, **Pfade** und **Zykel**.* [1]

[1] Hier sind Zykel ohne Schlaufen gemeint.

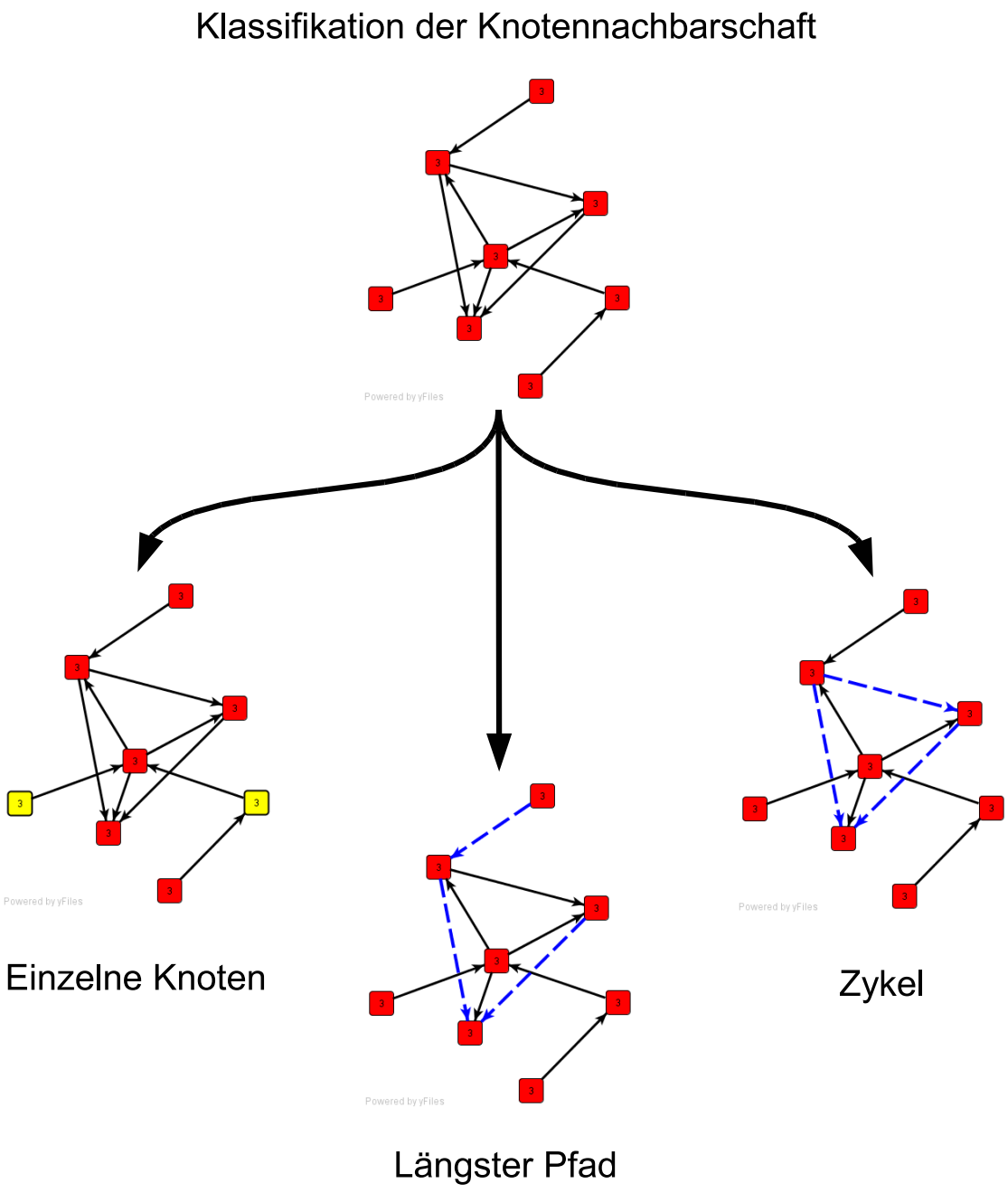


Abbildung 3.1: Die Topologie der Nachbarn von Knoten 1 zerfällt in einzelne Knoten, (längste) Pfade und Zykel. Diese Merkmale können bei der Abbildung isomorpher Knoten aufeinander verwendet werden.

Damit stehen uns nun zum Beispiel die Anzahl der freien Knoten, die Länge des längsten Pfades und die Zykelgrößen des induzierten Nachbarschaftsgraphen zur Verfügung, mit denen ein Knoten näher beschrieben werden kann. Gleichzeitig beinhalten sie Informationen über die topologische Umgebung des Knotens, wie auch die Information des Grades. Die Berechnung des längsten Pfades eines Graphen ist NP-vollständig, kann aber für kleine Graphen mit vertretbarem Aufwand mittels einer vollständigen Aufzählung (*Exhaustive Search*) berechnet werden. Wahrscheinlich ist es diese „teure Information“ die zu der guten Performance des Graphisomorphiealgorithmus beiträgt.
Alternativen, wie z.B. der Grad oder der *Konnektivitätskoeffizient*, der als Quotient von realisierten zu möglichen Kanten definiert ist, scheinen ebenfalls zu funktionieren, sollten aber in Zukunft noch genauer untersucht werden. Zudem bietet sich die Möglichkeit die Knotenklassifikation durch Kombination und Integration mit anderen Informationen zu verbessern. Denkbar wären biologische Merkmale wie „Gen“, “Transkriptionsfaktor“, „DNA“,“RNA“ und viele andere, die ein Merkmal m für einen Knoten definieren könnten.
Diese Abstraktionsschicht der Klassifkation ist der Kern, der den Algorithmen dieser Arbeit gemeinsam ist. Im folgenden seien die Merkmale also wie folgt definiert:

Definition 4. *Das **Nachbarschaftsmerkmal** für $u \in V$ sei gegeben als*

$$m_u := (s, p, c) \in M \tag{3.3}$$

wobei s die Anzahl der einzelnen Knoten, p die Länge des längsten Pfades und $c = \{c_i\}$ die Menge der Zykelgrössen c_i im induzierten Nachbarschaftsgraphen und M die gesamte Merkmalsmenge ist.

Für deren Berechnung siehe Algorithmus *classify-node* im Anhang A.2.1.

3.1.3 Die Rekursion

Hat man nun jedem Knoten ein Merkmal $m \in M$ zuordnen können, so könnte man sagen, dass jeder Knoten „lokales Wissen“ besitzt. Mit diesem könnte also eine korrekte lokale Zuordnung gemacht werden, nicht aber eine globale. Wie kann aber „lokales Wissen“ in „globales Wissen“ überführt werden?
Es muss nicht lange gesucht werden, um ein Netzwerk zu finden, in dem Knoten über „globales Wissen“ verfügen: Das Internet (siehe A.1) stellt ein prominentes Beispiel dafür dar. Jeder Router verfügt über genaue Informationen über die an ihn angeschlossenen Rechner. Über andere Router erfährt er in jeder Update-Runde von weiter entfernten Computern. Bis er, sofern man das Internet für den Augenblick mal als statisch annimmt, globales Wissen über die Topologie des Internets bezüglich seiner Position im Netzwerk besitzt.
Wenn man von dem Gedanken absieht, für jeden Knoten einen neuen Graphen erzeugen zu wollen, um diese dann auf Gleichheit zu prüfen, bleiben dennoch einige Konzepte, die sich sinnvoll nutzen lassen:

- Zu Beginn hat jeder Knoten lokales Wissen in seinem Informationsspeicher
- Jeder Knoten gibt sein Wissen an alle seine Nachbarn weiter, dadurch fliessen Informationen von einem Ende des Netzwerkes zum anderen
- Informationen werden parallel in Updaterunden ausgetauscht, die auf das Wissen zu Beginn der Runde zurückgreifen
- Irgendwann erhalten die Knoten keine neuen Informationen mehr, dann besitzt jeder Knoten globales Wissen

Ein ganz besonders wichtiger Aspekt dieses Verfahrens ist, dass es von der Topologie des Graphen abhängt, da Informationen nur mit den Nachbarknoten ausgetauscht werden. Und gerade die Topologie ist es, die das Problem an sich darstellt. Formal lässt sich das obige Verfahren folgendermaßen beschreiben:

Definition 5. *Die **Rekursion** ist gegeben als*

$$g^0(u) := m \in M \tag{3.4}$$

$$g^{i+1}(u) := h(\{g^i(u)\} \cup \{d(g^i(v)) \mid v \in N(u)\}) \tag{3.5}$$

*mit $g : V \to M$ **Informationsabbildung**, die jedem Knoten u eine Informationseinheit m aus einer abzählbar unendlichen Menge M zur Zeit i zuordnet. Die injektive **Distanzfunktion** $d : M \to M$ berücksichtigt eine Änderung der Information entlang einer Kante und $h : P(M) \to M$ ist die sogenannte **Transferfunktion**, mit der die von den Nachbarn erhaltenen Informationen zu einer neuen verarbeitet und gespeichert werden. Dabei ist zu beachten, dass*

$$P(M) := \{A \in \mathcal{P}(M) :\mid A \mid \leq MaxDegree\} \tag{3.6}$$

*gilt. $P(M)$ ist also eingeschränkt auf die Elemente der Potenzmenge von M mit Kardinalität kleiner als ein maximaler Knotengrad. Die Rekursion gilt als **konvergiert**, falls*

$$g^{i+1}(u) \equiv_M h(\{g^i(u)\} \cup \{d(g^i(v)) \mid v \in N(u)\}) \forall u \in V \tag{3.7}$$

für eine Äquivalenzrelation $\equiv_M$ auf M.

Bei Betrachtung des Aufrufs von g innerhalb der Argumente von h wird klar, dass diese Definition ein „funktionales Programm“ darstellt, das den Graphen aus der Sicht eines Knotens interpretiert. Man beachte dabei, dass u selbst wieder Nachbar von v ist, d.h. u's Information ihm von v zurückgereicht wird. Weiterhin ist wichtig, dass u seine Informationen nicht verliert, was im Parameter $g^i(u)$ für h zum Ausdruck kommt.
Wie sich durch die Implementierung zeigte, ist die Distanzfunktion nicht unbedingt nötig und kann deshalb als Identität definiert, bzw. weggelassen werden. Für sie sollte nur Injektivität gelten.
Wichtiger ist dagegen die Transferfunktion h. Ihre Eingabe sind Mengen aus der Potenzmenge von M mit einer beschränkten maximalen Kardinalität. Dabei ist wichtig, dass verschiedene Argumentmengen auf verschiedene Informationen der nächsten Iteration abgebildet werden.

Bedingung 1. *Für beliebige Transferfunktion $h : P(M) \longrightarrow M$ soll gelten:*

$$\forall A, B \in P(M) : A \neq B \Longleftrightarrow h(A) \neq h(B) \tag{3.8}$$

Diese Forderung kann, aufgrund der Endlichkeit von $P(M)$ leicht erfüllt werden, da M abzählbar unendlich ist.

Es ist nötig, dass das Argument von h eine ungeordnete Menge ist, deren Größe zwar beschränkt ist, aber nicht feststeht. Da die Nachbarn eines Knotens und deren Informationen keine Reihenfolge haben, muss diese Funktion unabhängig vom Knotengrad sein. Des weiteren wird ein geeigneter Informationsraum benötigt, der ausreichend viele Elemente für die Bedürfnisse der Transferfunktion enthält.

3.1.4 Die Merkmalszahlen

Wenn man die Routing-Analogie des vorhergehenden Abschnitts weiterverfolgen will, müsste man nun beginnen, eine Datenstruktur analog einer Routingtabelle zu entwerfen. Das würde jedoch die Vergabe von Identitäten für Knoten und Kanten implizieren. Doch die sind nicht gegeben. Jeder Knoten aus G kann auf jeden Knoten aus H abgebildet werden, sofern beide gleiche Merkmale besitzen. Würde man nun beginnen, die Kanten eines Knotens mit Identitäten zu belegen, um sich - analog zum Routing-Beispiel - zu merken, über welche Kante welches Merkmal kam, so würde man auch den Kantennachbarn Identitäten zuweisen. Würde man nur die Merkmale speichern, die einen Knoten erreichen, so könnte es

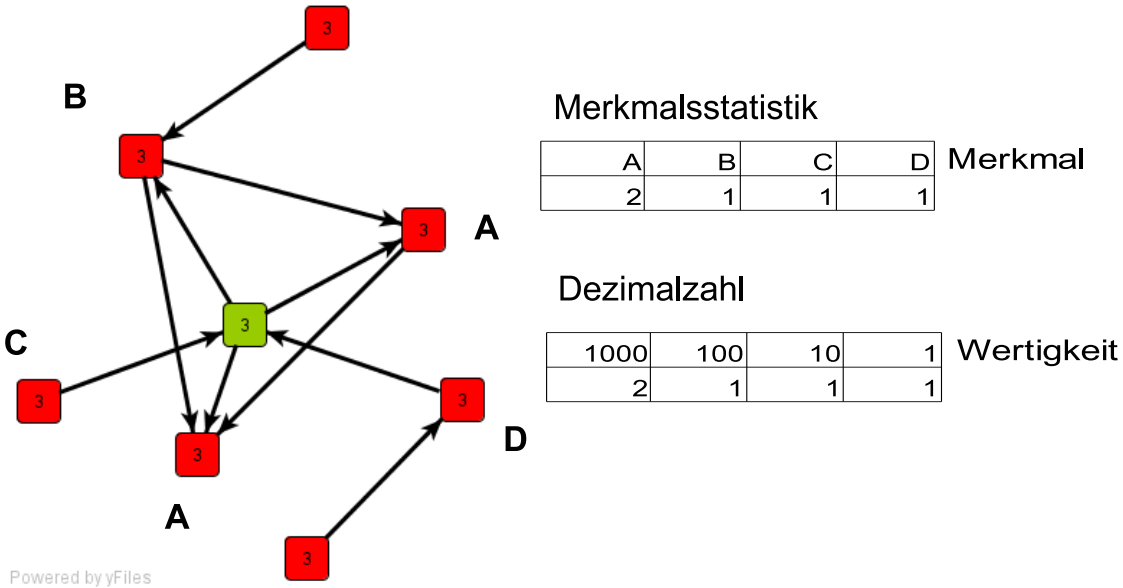


Abbildung 3.2: Jeder Nachbar des Knoten 3 besitzt eines von 4 verschiedenen Merkmalen. In der Statistik bilden diese die Spaltenüberschriften, während die Zahlen in den Zellen stehen. Die Reihenfolge der Spaltenüberschriften ist dabei beliebig, zudem gibt es Ähnlichkeiten mit Zahlensystemen.

passieren, dass jeder Knoten alle Merkmale erhält und man dadurch nichts gewonnen hätte. Topologisch gesehen, spielen die Distanzen eine Rolle, welche die Merkmale zurücklegen, um einen Knoten zu erreichen. Was ist aber, wenn es zwischen zwei Knoten mehrere Pfade gibt? Soll nur der Kürzeste zählen, oder sollten die „Eintreffzeitpunkte“ gespeichert werden? Das würde implizieren, dass die Merkmale mit „Zeitstempeln“ versehen und ihr Ursprung identifizierbar gemacht werden muss. Auch hier erscheinen (variable) Knotenidentitäten die einzige Lösung zu sein. Doch diese bringen nur das ursprüngliche kombinatorische Problem in etwas anderer Form mit sich.
Was kann man also unternehmen, um Identitäten zu vermeiden, aber das Wissen und die Fähigkeit der Zuordnung der Knoten aus G und H zueinander zu erweitern? Eine Statistik über die Merkmalsverteilung in der Nachbarschaft für einen Knoten würde sicher einen Fortschritt bedeuten. Dadurch würden einige Knoten herausfallen, die zwar vom Merkmal her passen könnten, deren Nachbarschaft aber nicht die richtige Zusammensetzung aufweißt. Diese Statistik würde in jedem Fall aber nicht erfordern, dass Knotenidentitäten eingeführt werden müssten. Genau genommen, würde sie für jeden Knoten „implizites Wissen“ erwerben, das den Versuch einer direkten Zuordnung vereinfachen würde.
Wenn man die Rekursion noch einmal näher betrachtet, wäre eine Merkmalstatistik sicher für dem Schritt von g^0 nach g^1 vertretbar. Die Transferfunktion würde dann für jede Merkmalsklasse die Anzahl der Vorkommnisse in der direkten Nachbarschaft zählen und als g^1-Wert zurückgeben.
Abbildung 3.2 veranschaulicht eine solche Statistik. Weist man jedem Merkmal eine Wertigkeit zu, so erhält man eine Zahl in einem Zahlensystem, dessen Stellen über die Merkmale definiert sind. [2] Bevor das Merkmalszahlensystem genauer definiert wird, sollen vorher noch zwei Eigenschaften verdeutlicht werden:

1. Es besteht keine Ordnung zwischen den Stellen und deshalb auch nicht zwischen den Zahlen.

2. Merkmale können nicht ineinander umgerechnet werden, da sie „atomare Information“ darstellen.

[2] Die Idee ein Zahlensystem mit gemischter Basis (*mixed-radix number system*) zur Lösung eines Problems heran zu ziehen, findet sich u.a. auch bei „Generic number systems and haplotype analysis“ [26].

Im Falle einer Rechnung kann also kein Übertrag entstehen.

Diese beiden Beobachtungen reflektieren nur die oben angeführten Schwierigkeiten bei der Einführung von Knotenidentitäten und die Flexibilität der Klassifikation. Ein Zahlensystem jedoch bringt die letzte benötigte Komponenten mit: die Addition. Bisher waren g als Speicherabbildung und d als Identität definiert und nur h, die Transferfunktion, musste noch gesucht werden. Diesen Part wird jetzt die Addition im Merkmalszahlensystem übernehmen und damit die Rekursion vervollständigen.
Zunächst werden die Elemente des Merkmalszahlensystems definiert:

Definition 6. *Sei $M = \{m\}$ eine Menge von Merkmalen und $M(x) \subseteq M$, die von x als Basis verwendete Teilmenge von M, dann ist eine **Merkmalszahl** x_M definiert als:*

$$x_M := \bigcup_{m \in M(x)} \{(c_m, m)\} \tag{3.9}$$

*mit $c_m \in \mathbb{N}$ **Koeffizient** von m. Per Konvention sei $c_m = 0$, falls $m \notin M(x)$. Die Menge aller Merkmalszahlen über M sei mit $\mathbb{M}_M$ bezeichnet.*

Für h definieren die die Addition wie folgt:

Definition 7. *Seien $x, y \in \mathbb{M}_M$ beliebig. Die **Addition** sei wie folgt definiert:*

$$x +_M y := \bigcup_{m \in M(x) \cup M(y)} (c_m^x + c_m^y, m) = \bigcup_{m \in M} (c_m^x + c_m^y, m) \tag{3.10}$$

Die Koeffizienten werden also stellenweise ohne Übertrag addiert, wenn die Addition für h verwendet wird. Damit ergibt sich:

Definition 8. *Die **Merkmalsrekursion** ist definiert als:*

$$g^0(u) := (1, c(u)) = (1, m) \in \mathbb{M}_M \tag{3.11}$$

$$g^{i+1}(u) := g^i(u) + \sum_{v \in N(u)} g^i(v) \tag{3.12}$$

mit c ein beliebiger Klassifikator.

Den Rekursionsfuß bildet also die Knotenklassifikation selbst. Ansonsten summiert man jeweils die Merkmalszahlen der Nachbarn auf. Doch wie oft soll das geschehen? Es wird eine Äquivalenzrelation für die Merkmalszahlen als Abbruchkriterium benötigt, die für die Merkmalsrekursion feststellt, ob noch neue Information berechnet wird. Diese lässt sich wie folgt definieren:

Definition 9. *Seien $x, y \in \mathbb{M}_M$ beliebig. Es gilt*

$$x \equiv_M y \Longleftrightarrow M(x) = M(y). \tag{3.13}$$

Damit endet die Merkmalsrekursion, wenn kein Knoten neue „atomare Information“ mehr erhält. Es ist zwar nicht sichergestellt, dass dadurch jedes Merkmal komplett durch den Graphen geflossen ist und jeder Knoten damit wirklich globales Wissen erhalten hat, aber diese Definition hat sich als zweckmäßig erwiesen und gute Ergebnisse geliefert. Daher ist sie die einzige Konvergenzrelation, die für den Abbruch bisher eingesetzt wird. Eine mögliche Alternative wäre $g^{|V|}$ zu berechnen. Vermutlich kann sogar nachgewiesen werden, dass immer $|\,V\,|$ viele „Transfers“ nötig sind (vergleiche Kapitel 3.1.6).

3.1.5 Der Backtrackingalgorithmus

Was wurde nun mit der Merkmalsrekursion erreicht? Da durch die Rechnung nur „nichtpassende“ Elemente aussortiert werden, gilt für die Rekursion die Invariante:

Invariante 1. *Knoten mit gleichen Merkmalen können in einer Isomorphieabbildung zwischen G und H aufeinander abgebildet werden, bzw. isomorph sein.*

Alles was erreicht wurde, ist eine Verfeinerung des globalen Wissens der einzelnen Knoten, die bezüglich der gewählten Äquivalenzklasse nicht mehr verbessert werden kann. Damit verbleibt nun immer noch die Arbeit, die Isomorphie zwischen G und H konkret nachzuweisen. Als ersten Test bleibt einem dazu der Vergleich der Merkmalshistogramme, was mindestens einem verbesserten Vergleich der Gradfolge gleichkommt:

Definition 10. *Ein **Merkmalshistogramm** ist definiert als $\mathcal{H}_G : \mathbb{M}_M \to \mathbb{N}$, die jeder Merkmalszahl aus $\mathbb{M}_M$, bzw. jedem Merkmal aus M deren Häufigkeit im betrachteten Graphen G zuordnet. Das Histogramm kann sowohl für Knoten- als auch für Kantenmerkmale gebildet werden. Das Merkmalshistogramm für die Informationsabbildung g^n und Graph G wird mit $\mathcal{H}_G^n$ bezeichnet.*

Da gleiche Graphen unter der Rekursion zu gleichen Ergebnissen führen müssen, gilt folgender Satz, der leicht durch Widerspruch bewiesen werden kann:

Satz 1.

$$H^G \neq H^H \Longrightarrow G \neq H \tag{3.14}$$

Sind also die Knotenhistogramme gleich, so kann man versuchen, eine Isomorphieabbildung aufzuzählen. Da noch nicht sicher feststeht, dass G und H tatsächlich isomorph sind, soll nun ein *Backtracking*-Verfahren mit *Branch & Bound*-Charakter verwendet werden, um eine oder alle möglichen Isomorphieabbildungen zu berechnen.
Dazu wird als erstes ein „markanter Knoten“ u mit $\mathcal{H}_G(g^n(u))$ minimal in G ausgewählt. Von diesem ausgehend wird mittels BFS eine Aufzählungsliste der Kanten aus G berechnet, die jede genau einmal aufzählt, wobei Kreuzkanten vor Vorwärtskanten aufgelistet werden (siehe Algorithmus *calculate-visiting-order* und Abbildung 3.3). Weiterhin werden die Elemente so aufgezählt, dass sie zu jeder Zeit eine *Zusammenhangskomponente* darstellen. Dies setzt die folgende Annahme voraus:

Annahme 1. *Der Graph G, und damit auch H, ist zusammenhängend.*

Der Backtracking-Algorithmus versucht nun eine konsistente Zuordnung in dieser Reihenfolge, wobei er schon zugeordnete Knoten und Kanten markiert. Die folgende Abbildung und der Pseudocode im Anhang A.2.2 illustrieren das Vorgehen.

3.1.6 Formale Betrachtungen zur Graphisomorphie

Da die vorhergehenden Kapitel sehr anschaulich geschrieben waren, wird sich dieses Kapitel formaler mit der Rekursion beschäftigen. Dabei werden einige ihrer Eigenschaften nachgewiesen, die zeigen, dass das Backtracking-Verfahren ohne Rückschritte sofort eine Isomorphieabbildung berechnen kann. Dazu wird mit einer kurzen Zusammenfassung aller beweisrelevanten Eigenschaften und Definitionen begonnen. Anschliessend wird mit einigen kurzen Lemmata über weitere Eigenschaften der Rekursion der Grundstein für den Beweis des ersten Satzes gelegt, der die Existenz einer mindestens lokal isomorphen Abbildung nachweist. Ausgehend von diesem Satz ist es ein kurzer Schritt zu dem zweiten Satz, der zeigt, dass eine global konsistente Isomorphieabbildung effizient durch das Backtracking-Verfahren berechnet wird. Zuletzt wird noch auf die Zeit-Komplexität des Verfahrens eingegangen.
Der Ausgangspunkt für die folgenden Beweise lautet also:

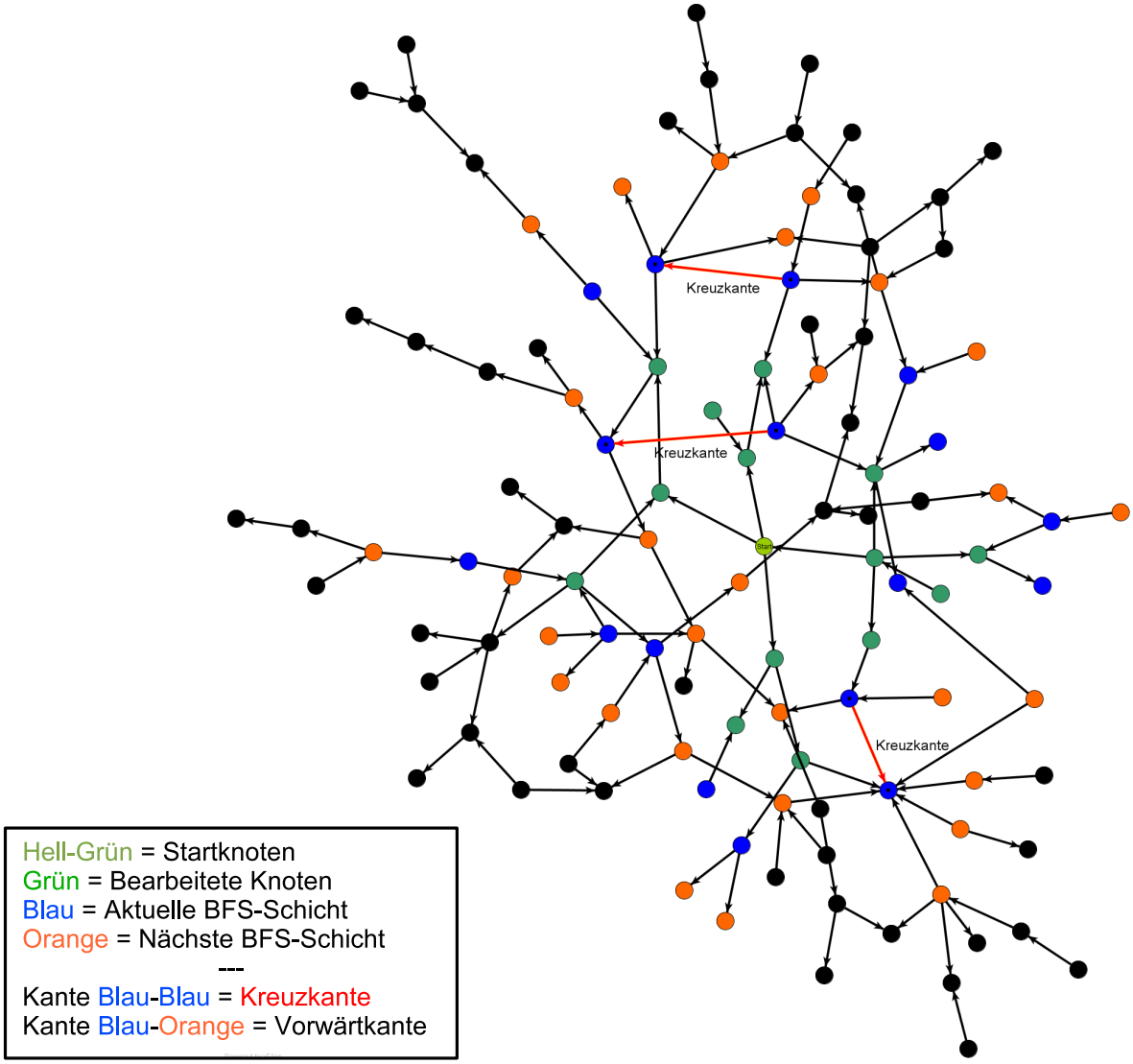


Abbildung 3.3: Ausgehend von dem grünen Startknoten wurden die gelben Knoten mittels BFS erreicht und bearbeitet. Die nächsten beiden BFS-Schichten werden durch die blauen und orangefarbenen Knoten gestellt. Die Durchwanderungsreihenfolge stellt sicher, dass Kreuzkanten zwischen den blauen Knoten vor den Vorwärtskanten (Blau nach Orange) bearbeitet werden. So wird erreicht, dass nicht passende Abbildungen so früh wie möglich erkannt werden.

Definition 11 (Zusammenfassung)**.** *Sei M eine abzählbar unendliche Merkmalsmenge, z.B. $\mathbb{M}_X$, und $G = (V,E)$ ein beliebiger ungerichteter Graph. Die Nachbarschaft eines Knotens sei*

$$N(u) := \{v \in V : \exists \{u,v\} \in E\}$$

Sei $c : M \longrightarrow V$ ein Klassifikator, für den gilt

$$\begin{aligned} \forall u,v \in V : c(u) = c(v) &\Longleftrightarrow u,v \text{ sind lokal isomorph} \\ &\Longleftrightarrow N(u), N(v) \text{ sind gleich} \end{aligned} \tag{3.15}$$

Die Rekursion lautet

$$\begin{aligned} g^0(u) &:= c(u) = m \in M \\ g^{i+1}(u) &:= h(\{g^i(u)\} \cup \{d(g^i(v)) : v \in N(u)\}) \end{aligned} \tag{3.16}$$

und deren Eigenschaften sind

$$\begin{aligned} g &: V \longrightarrow M \\ d &: M \longrightarrow M, \forall a,b \in M : a \neq b \Longleftrightarrow d(a) \neq d(b) \\ P(M) &:= \{X \in \mathcal{P}(M) :\mid X \mid \leq MaxDegree(V)\} \\ h &: P(M) \longrightarrow M, \forall A,B \in P(M) : A \neq B \Longleftrightarrow h(A) \neq h(B) \end{aligned} \tag{3.17}$$

*Außerdem gilt g^{n+1} ist **konvergiert** unter M, wenn*

$$g^{n+1}(u) \equiv_M g^n(u) \forall u \in V \tag{3.18}$$

gilt, also die Nachfolgefunktion g^{n+1} auf ein anderes, aber äquivalentes Merkmal abbildet.

Mit diesen Definitionen folgt aus 3.17 zunächst:

Lemma 1. *Seien $A,B \in P(M)$ mit $A = B$ beliebig, dann gilt $h(A) = h(B)$, sowie für beliebiges $x \in A$ oder $x \in B$ auch $h(A \setminus \{x\}) = h(B \setminus \{x\})$.*

Aus 3.17 folgt ausserdem

Lemma 2.

$$g^{i+1}(u) = g^{i+1}(u^*) \Longrightarrow g^i(u) = g^i(u^*) \forall i \tag{3.19}$$

Beweis (Widerspruch) Angenommen es gelte $g^{i+1}(u) = g^{i+1}(u^*)$, aber nicht $g^i(u) = g^i(u^*)$ für beliebiges i. Wähle $i = 0$, dann ist $c(u) = g^0(u) \neq g^0(u^*) = c(u^*)$ und deshalb $g^1(u) \neq g^1(u^*)$, da die Nachbarschaften nach 3.15 nicht isomorph sein dürfen, sich also durch die Merkmale unterscheiden müßen. Damit kann, wegen 3.17 für größere i nicht mehr $g^{i+1}(u) = g^{i+1}(u^*)$ gelten. ■
Weiterhin gilt

Lemma 3 (Existenz eines Nachbars). *Für beliebige Knoten u, u^* gilt:*

$$g^{i+1}(u) = g^{i+1}(u^*) \Longleftrightarrow \forall v \in N(u) \exists v^* \in N(u^*) : g^i(v) = g^i(v^*) \tag{3.20}$$

Beweis Es ist

$$\begin{aligned} & g^{i+1}(u) = g^{i+1}(u^*) \\ \Leftrightarrow_{3.16}\ & h(\{g^i(u)\} \cup \{d(g^i(v)) : v \in N(u)\} = h(\{g^i(u^*)\} \cup \{d(g^i(v^*)) : v^* \in N(u^*)\} \\ \Leftrightarrow_{3.17}\ & h(M) = h(M^*) \Leftrightarrow M = M^* \end{aligned}$$

Wähle nun $v \in N(u) : m = d(g^i(v)) \in M$ beliebig, dann gibt es $m^* \in M^* : m = m^*$ und wegen Lemma 1 ein $v^* \in N(u^*) : m^* = d(g^i(v^*))$. ■

Nun kann der grundlegenden Satz 2 für die Rekursion bewiesen werden. Die Aussage dieses Satzes ergibt zusammen mit Lemma 2, dass für alle Knoten im Abstand von $i+1$ eine Abbildung auf lokal isomorphe Knoten existiert.

Satz 2 (**Nachbarn in Entfernung** n)**.** *Seien u, u^* beliebige Knoten mit $g^{i+1}(u) = g^{i+1}(u^*)$. Dann gilt*

$$\forall v : dist(u,v) \leq i+1 \exists v^* : dist(u^*,v^*) = dist(u,v) = k \wedge g^k(v) = g^k(v^*) \tag{3.21}$$

Beweis (Konstruktiv) Der Beweis geht wie folgt vor: Zunächst wird ein Algorithmus A und dessen Invariante I angegeben. Indem nachgewiesen wird, dass diese für A gilt und er sicher terminiert, vervollständigen wir den Beweis. A wird eine mögliche Lösung konkret berechnen. Dabei wird deutlich, dass dies unter den gegebenen Vorraussetzungen immer möglich ist.

Algorithm 3.1.1. *Algorithmus* $\mathsf{A}(g, i+1, u, u^*)$

```
1  k ← i+1;
2  comment: Invariante I: A enthält Tripel (k,v,v*) : g^k(v) = g^k(v*) für 0 ≤ k ≤ i+1
3  A ← {(k,u,u*)};
4  while 0 ≤ k do
5      for each (k,v,v*) ∈ A do
6          for all w ∈ N(v) : ∄(a,b,c) ∈ A : b = w do
7              Sei w* ∈ N(v*) : g^{k-1}(w) = g^{k-1}(w*) ∧ ∄(a,b,c) ∈ A : c = w*
8              A ← A ∪ {(k-1,w,w*)};
9          od
10     od
11     k ← k-1;
12 od
```

Initialisierung: Die Datenstruktur $\mathcal{A}$ wird mit der Eingabe $(i+1, u, u^*)$ des Programms initialisiert, da $k = i+1$. Für diese gilt die Annahme $g^{i+1}(u) = g^{i+1}(u^*)$ und daher gilt in Zeile 3 die Invariante vor Beginn der Hauptschleife.

Iteration: In jedem Schleifendurchlauf wird für jedes Tripel $(k,v,v^*) \in \mathcal{A}$ getestet, ob v einen Nachbarn $w \in N(v)$ besitzt, für den in $\mathcal{A}$ noch kein Tripel $(a,b,c) : b = w$ existiert.

Sei $w \in N(v)$ ein solcher Nachbar, dann gibt es ein $w^* \in N(v^*) : g^{k-1}(w) = g^{k-1}(w^*)$, da wegen I gilt $g^k(v) = g^k(v^*)$. Denn $g^k(v) = g^k(v^*) \Leftrightarrow h(M) = h(M^*)$ nach 3.17. Die Mengen M, M^* lassen sich nun aber zerlegen in

$$\begin{aligned} M &= M_{\mathcal{A}} \cup M_{\neg\mathcal{A}} = \{w \in N(v) : \exists (a,b,c) \in \mathcal{A} : b = w\} \cup \{w \in N(v) : \nexists (a,b,c) \in \mathcal{A} : b = w\} \\ M^* &= M^*_{\mathcal{A}} \cup M^*_{\neg\mathcal{A}} = \{w^* \in N(v^*) : \exists (a,b,c) \in \mathcal{A} : c = w^*\} \cup \{w^* \in N(v^*) : \nexists (a,b,c) \in \mathcal{A} : c = w^*\} \end{aligned}$$

Da die Knoten aber immer paarweise in $\mathcal{A}$ gespeichert werden und die Invariante I für diese gilt, ergibt sich

$$(M = M^* \wedge M_{\mathcal{A}} = M^*_{\mathcal{A}}) \Rightarrow M_{\neg\mathcal{A}} = M^*_{\neg\mathcal{A}}$$

Sei also

$$g^{k-1}(w) = m \in M_{\neg\mathcal{A}} \Rightarrow \exists w^* \in N(v^*) : g^{k-1}(w^*) = m^* \in M^*_{\neg\mathcal{A}}$$

Also kann A immer ein solches w^* auswählen und in $\mathcal{A}$ als $(k-1, w, w^*)$ speichern, so dass die Invariante erhalten bleibt. Denn es kann für $k-1$ aufgrund 3.17 immer ein solches w^* gefunden werden. Damit gilt I in Zeile 11 und somit auch in Zeile 5.
Termination: Da in Zeile 1 k mit $i+1$ initialisiert und in Zeile 11 dekrementiert wird, die Schleife aber bei $k < 0$ abbricht, terminiert der Algorithmus A sicher. Da die Invariante während der gesamten Hauptschleife aufrecht erhalten wird, gilt sie auch am Ende und in $\mathcal{A}$ befindet sich eine Abbildung mit den gesuchten Eigenschaften.
Da es sich im Inneren der Hauptschleife um eine Breitensuche handelt, gilt für jedes Tripel

$$(k, v, v^*) \in \mathcal{A} : dist(u, v) = dist(u^*, v^*) = i + 1 - k$$

Womit auch die Aussage über die Distanzen gezeigt und Satz 2 bewiesen ist. ■
Es bleibt noch zu zeigen, dass bei gleichen Histogrammen $\mathcal{H}^n_G$ für unter $\equiv_M$ konvergierte g^n immer eine echte Isomorphie-Abbildung gefunden werden kann. Also

Satz 3 (Gleiche Histogramme). *Es gilt*

$$g^{n+1}(u) \equiv_M g^n(u) \forall u \wedge \mathcal{H}^n_G = \mathcal{H}^n_H \Longrightarrow \exists I : I = \{(u, u^*) : u \mapsto u^*\} \textit{ Isomorphieabbildung} \tag{3.22}$$

Beweis Betrachtet man die Aussage von Satz 2 noch einmal genauer, so sichert dieser Satz aufgrund von Lemma 2 zu, dass sicher *mindestens* eine Abbildung für alle Nachbarn $v \in N(u) : dist(u, v) \leq i+1$ auf die Nachbarn $v^* \in N(u^*) : dist(u^*, v^*) \leq i+1$ gefunden werden kann, die *mindestens* die lokale Isomorphie erhält. Es lässt sich also definieren:

Definition 12. *Sei $g^n(u)$ beliebig, dann ist die Menge $\mathcal{L}$ aller lokal isomorphen Abbildungen I definiert als*

$$\mathcal{L}_{g^n}(u) := \{I : I \textit{ kann von Algorithmus A von u ausgehend gefunden werden}\} \tag{3.23}$$

Auf diesen Mengen lässt sich die Operation $\bowtie$ wie folgt definieren:

Definition 13 ($\bowtie$-Operation)**.** *Sei V Knotenmenge mit Abbildungsmenge $\mathcal{L}_{g^n}(V) = \{I_V\}$, sowie Knoten $u \notin V$ mit Abbildungsmenge $\mathcal{L}^*_{g^n}(u) = \{I_u : (u, u^*) \in I_u)\}$, dann ist*

$$\mathcal{L}_{g^n}(V \cup \{u\}) := \mathcal{L}_{g^n}(V) \bowtie \mathcal{L}^*_{g^n}(u) := \{I \in \mathcal{L}_{g^n}(V) \cup \mathcal{L}^*_{g^n}(u) : (u, u^*) \in I\} \tag{3.24}$$

Dieser Operator fügt den Knoten u also in V ein, selektiert aber auf die Abbildungen, die u auf u^* abbilden, bzw. das Paar (u, u^*) enthalten. Er macht damit genau das, was der Backtracking-Algorithmus A.2.2 macht: er erweitert seine gehaltene Knotenpartition (gegeben durch die zugeordneten Kanten) um einen neuen Knoten, verringert aber evtl. die Menge der ihm im nächsten Schritt zur Verfügung stehenden Abbildungen. Es wird klar, dass, sollte es eine Isomorphieabbildung geben, diese in dieser Restmenge enthalten sein muss. Folglich muss zunächst gezeigt werden, dass im Falle gleicher Histogramme die verbleibende Menge niemals leer wird. Danach kann man das Argument aus Kapitel 3.2.5 anführen, bzw. mittels Widerspruch beweisen, dass die gefundene Abbildung eine Isomorphieabbildung ist.
Dazu werden folgende Annahmen benötigt:

Annahme 2.

$$n = \mid V \mid \tag{3.25}$$

Diese Annahme gilt besonders dann, falls g schon bei kleinerem n konvergiert ist, da Konvergenz bedeutet, dass es keine Information mehr gibt, durch die Knoten mit gleichem g^n auf unterschiedliche Merkmale abgebildet werden.
Seien also $W \subset V$ die vom Backtracking-Algorithmus bereits erreichte zusammenhängende Teilmenge und $\mathcal{L}_{g^n}(W)$ die verbleibenden Abbildungen. Sei $v \in W$ der Knoten, von dem aus $u \notin W$ erreicht wird. Da $dist(u,v) = 1 \leq n$ gibt es in $\mathcal{L}_{g^n}(W)$ Abbildungen, die u als Urbildknoten enthalten. Andererseits muss es, wegen Annahme 2, in $\mathcal{L}_{g^n}(u)$ mindestens eine Abbildung geben, die W so abbildet, wie sie durchwandert wurde, da A denselben Pfad verwenden könnte wie der Backtracking-Algorithmus und im Beweis von Satz 2 keinerlei Annahmen über die Art der Durchwanderung gemacht wurden. Damit besitzen sowohl $\mathcal{L}_{g^n}(W)$, als auch $\mathcal{L}^*_{g^n}(u)$ Abbildungen, die den jeweiligen anderen Operanden berücksichtigen, so dass $\mathcal{L}_{g^n}(W \cup \{u\}) = \mathcal{L}_{g^n}(W) \bowtie \mathcal{L}^*_{g^n}(u)$ nicht leer sein kann.
Nun stellt sich die Frage, ob Algorithmus A und das Backtracking-Verfahren derart vorgehen können, dass zwar lokal isomorphe Abbildungen übrig bleiben, aber keine einzige Isomorphieabbildung? Der folgende Widerspruchsbeweis zeigt, dass immer eine Isomorphieabbildung übrig bleiben wird. Angenommen, $\mathcal{L}(V)$ wäre die verbleibende, nicht-leere Menge an lokal isomorphen Abbildungen, die der Backtracking-Algorithmus berechnen kann und sie enthalte keine Isomorphieabbildung. Da der Algorithmus aber nur mindestens lokal isomorphe Knoten einander zuweist und für jeden Knoten $u \in V$ ein u^* gefunden wurde, kann diese Situation nur durch Unterschiede in der Kantenmenge zustande kommen. Damit muss es Paare (u,u^*) und (v,v^*) geben, die aufeinander abgebildet wurden, sich aber in den Kanten unterscheiden. Konkret: es gibt die Kante (u,v), aber nicht (u^*,v^*). Damit wären aber weder u,u^* noch v,v^* lokal isomorph und es würde gelten $g^n(u) \neq g^n(u^*)$, bzw. $g^n(v) \neq g^n(v^*)$. Das wäre ein Widerspruch zur Korrektheit des Algorithmus A, sowie des Backtracking-Algorithmus. Damit gilt das Gegenteil und $\mathcal{L}(V)$ enthält mindestens eine Isomorphieabbildung I. Aus der Gleicheit der konvergierten Histogramme folgt also die Gleichheit der Graphen. ■

Zum Schluss stellt sich die Frage, wieviel die Berechnung eines Histogramms kostet? Zunächst einmal werden in jeder Runde über jede der $\mid E \mid$ Kanten die Merkmale ausgetauscht. Eine Runde dauert also $O(\mid E \mid)$ Zeit. Da es im schlimmsten Fall $\mid V \mid$ viele unterschiedliche Merkmale geben kann, die dann auch genauso viele Runden benötigen können, bis sie bei jedem Knoten bekannt sind, werden im schlimmsten Fall $O(\mid E \mid \cdot \mid V \mid)$ viele Runden benötigt. Ein Histogrammvergleich hingegegen benötigt im schlimmsten Fall $O(\mid V \mid)$, wenn jeder Knoten ein eigenes Merkmal erhalten hat. Insgesamt benötigt ein Isomorphievergleich damit nur $O(\mid E \mid \cdot \mid V \mid + \mid V \mid))$ Zeit mit diesem Algorithmus, wobei sich dieser Wert durch Vorberechnung der Histogramme auf $O(\mid V \mid)$ verringern lässt.

Satz 4 (Komplexität des Graphisomorphiealgorithmus)**.** *Mit dem Graphisomorphie-Algorithmus können zwei Graphen in Zeit*

$$O(\mid E \mid \cdot \mid V \mid + \mid V \mid)$$

verglichen werden.

3.1.7 Symmetrische Knoten

Nach der Betrachtung des Verfahrens, stellt sich nun die Frage, wie das „implizite Wissen“ aussieht, das über die Merkmalsrekursion berechnet wird. Ein Ansatz dazu ist alle Knoten mit gleicher Merkmalszahl anzufärben. In Abbildung 3.4 ist ein Beispielgraph entsprechend gefärbt worden.
An diesem Bild fällt auf, dass Knoten, die symmetrisch an den Graphen angeschlossen sind, dieselbe Farbe erhalten. Es sind Knoten, die von einem Automorphismus untereinander ausgetauscht werden können. Das heißt, sucht man eine Isomorphieabbildung von dem Graphen auf sich selbst, so können Knoten mit derselben Merkmalszahl $g^n(u)$ aufeinander abgebildet werden, ohne das eine Abbildung unmöglich wird. Dies führt uns zu folgendem Korollar:

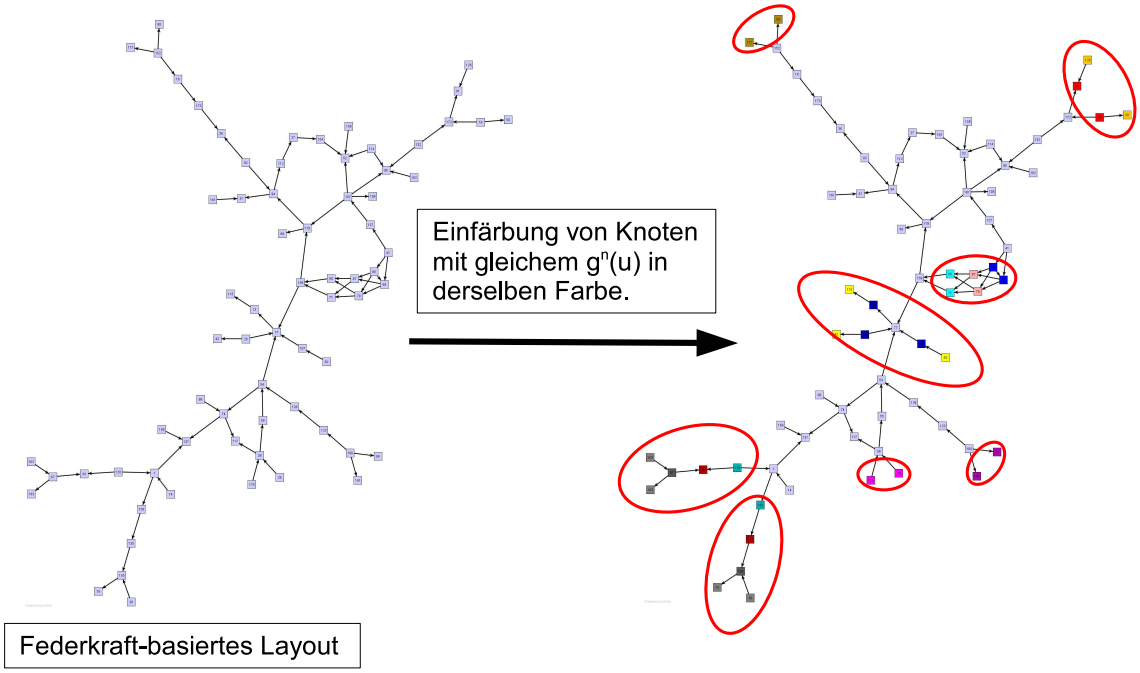


Abbildung 3.4: Das Anfärben von Knoten mit gleichem $g^n(u)$ enthüllt Symmetrien. Auffällig ist dabei, dass viele der symmetrischen Knoten auch über ein federkraftbasiertes Layout gefunden werden könnten.

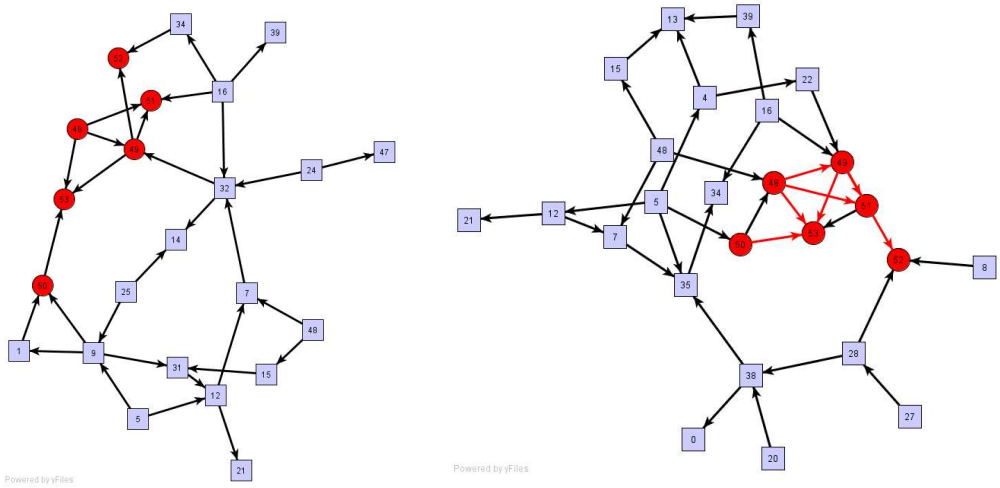
Korollar 1. *Knoten mit gleicher Merkmalszahl $g^n(u)$ sind isomorph bezüglich Automorphie. Sie heissen* ***topologisch symmetrisch****.*

Für den Algorithmus bedeutet das allerdings, dass die Suche einer Isomorphieabbildung mittels des Backtracking-Verfahrens der Suche einer Automorphieabbildung gleich kommt.
In der Literatur findet sich bei „Geometric symmetry in graphs“ [36] die Behauptung, dass die Suche nach jeder axialen (rotationaler bzw. zentraler) Symmetrie ein NP-vollständiges Problem ist. Da es dem Autor nicht möglich war, die entsprechenden Beweise einzusehen, sollten auch hier weitere Nachforschungen angestellt werden.
Außer der Berechnung symmetrischer Knoten, bietet der Graphisomorphietest die Möglichkeit, Datenstrukturen zu implementieren, die über Topologien als Schlüssel auf Werte zugreifen. Das verwendet der Frequent Pattern-Algorithmus aus Kapitel 3.3.4.

3.2 Subgraphisomorphie

3.2.1 Einleitung

Bei der Betrachtung von Graphen stellt sich oft die Frage, ob ein Graph in einem anderen als Teilgraph enthalten ist. Allerdings bedarf diese Frage weiterer Präzision. So ist es einerseits möglich, nach einer „Kopie“ des Musters zu suchen, die in den anderen Graphen „eingeklebt“ wurde (*ESI*), oder nach einer, einen zusammenhängenden Teilgraphen bildenden, Kantenmenge zu fragen, deren Topologie der des Musters entspricht (*CSI*). Abbildung 3.5 verdeutlicht diesen Unterschied. Es ist offensichtlich, dass *ESI* ein vereinfachender Spezialfall von *CSI* ist. In diesem Kapitel soll es darum gehen, *CSI*-Probleme zu lösen. Allerdings ist, im Gegensatz zur Graphisomorphie, von *CSI* die NP-Vollständigkeit nachgewiesen. Damit kommen nur noch Algorithmen in Frage, die Strategien zur Vermeidung von unnötiger Rechnung einsetzen: randomisierte Algorithmen (Monte Carlo/Las Vegas), Approximationsalgorithmen



Exact Subgraphisomorphy (ESI) *Common Subgraphisomorphy (CSI)*

Abbildung 3.5: Während bei *Exact Subgraphisomorphy* die genaue Kopie eines Graphen als eingebetteter Teilgraph gesucht wird, muss bei *Common Subgraphisomorphy* eine Kantenmenge ausgewählt werden, die den Mustergraphen nachbildet.

(FPTAS), parametrisierte Algorithmen, Branch & Bound-Verfahren oder Dynamisches Programmieren, um nur einige Möglichkeiten zu nennen.
Obwohl, angesichts der NP-Vollständigkeit, randomisierte und approximative Verfahren durchaus an Reiz gewinnen, scheint es schwer möglich zu sein, die im Graphisomorphiealgorithmus entwickelten Methoden unter diesen Umständen einzusetzen. Auch der Ansatz des Dynamischen Programmierens, obwohl er vielversprechende Ansätze bietet, erwies sich als wenig zugänglich. Somit blieb nur noch die Strategie, so intelligent wie möglich bei der Suche nach Subgraphen vorzugehen und unpassende Abbildungen so früh wie möglich zu erkennen.
Im folgenden Kapitel wird nun der entwickelte Subgraphisomorphiealgorithmus vorgestellt. Er verwendet, wie schon angekündigt, Ideen des Graphisomorphiealgorithmus und diesen letzten Endes sogar selbst, um die allgemeine Form (*CSI*) des Subgraphisomorphieproblems zu lösen. Dazu wird zunächst die „Enthaltenseins“-Relation über den Merkmalen definiert. Diese wird zum Finden der möglichen Startpunkte des modifizierten Backtracking-Algorithmus (Kapitel 3.2.4) verwendet. Bevor es um diesen geht, wird noch eine Klassifikation der Kanten definiert, die dabei hilft, unnötigen Aufwand zu vermeiden. Das Kapitel endet mit einer Diskussion der Probleme, die sich beim Versuch ergaben, die Merkmalsrekursion auf das Subgraphisomorphieproblem anzuwenden. Hier finden sich auch interessante Ansätze für weiterführende Betrachtungen.

3.2.2 Die „Enthaltenseins“-Relation

In Kaptiel 3.1.2 wurde eine erweiterte Klassifikation der Knoten vorgeschlagen. Diese basiert auf der Zerlegung der Knotennachbarschaft in einzelne Knoten, (längste) Pfade und Zykel. Angesichts der Aufgabe eine „große Enthaltenseins“-Relation zu suchen, wäre eine Idee, diese auch für die Merkmale, den „Atomen“ des Graphisomorphiealgorithmus, zu definieren. Vielleicht wäre dann eine Anpassung der Merkmalsrekursion möglich, so dass dasselbe Prinzip der „Gleichheit unter Interpretation“ anwendbar wird. Alternativ könnte die implizite Information der Merkmalszahlen in „Enthaltenseins“-Information umgerechnet werden.

Obwohl sich diese Ansätze im Verlauf der Entwicklung scheinbar als nahezu unbrauchbar herausgestellt haben, bildet die Definition einer „Enthaltenseins"-Relation unter den Merkmalen einen ersten Schritt zur Vermeidung unnötiger Rechnungen. Allerdings würde die Berechnung *exakten* Enthaltenseins eines Nachbarschaftsgraphen in einem anderen wieder rekursiv zum Subgraphisomorphieproblem zurückführen, ohne dass viel gewonnen wäre. Daher wurde der vorsichtige Begriff „Enthaltenseins"-Relation, in Symbolen $\blacktriangleleft$-Relation und nicht „Element"-Relation, $\in$-Relation, gewählt. Die im Folgenden definierte Relation ist über den Merkmalen und nicht über den konkreten Nachbarschaftsgraphen gegeben und sagt nichts darüber aus, ob eine Nachbarschaft tatsächlich in einer anderen enthalten ist. Vielmehr liefert sie nur die Aussage, dass sie es sein könnte.
Die Definition der $\blacktriangleleft$-Relation lautet also:

Definition 14. *Sei c ein bliebiger Klassifkator und u,v beliebige Knoten mit $c(u) = m_u$ und $c(v) = m_v$. Sei weiterhin der induzierte Nachbarschaftsgraph von u mit $G(u) = (V(u),E(u))$, analog $G(v) = (V(v),E(v))$, bezeichnet und es sei für beliebige Graphen $G = (V,E), G' = (V',E')$ die Vereinigung $\cup$ gegeben als $G \cup G' = (V \cup V', E \cup E')$ und die Differenz $\setminus$ als $G \setminus G' = (V \setminus V', E \setminus E')$. Dann gilt*

$$m_u \blacktriangleleft m_v \Longleftrightarrow \exists G_u^\Delta, G_v^\Delta : c(G(u) \cup G_u^\Delta) = c(G(v) \setminus G_v^\Delta) \tag{3.26}$$

Das heißt, ein Merkmal m_u ist in m_v enthalten, gdw. wenn es eine Menge von Knoten und Kanten gibt, mit denen der induzierte Nachbarschaftsgraph von u erweitert und der von u reduziert werden kann, so dass deren Klassifikation durch c zu demselben Merkmal führt.

Dabei ist zu beachten: Es dürfen bei u nur Kanten und Knoten hinzugefügt und bei v nur abgezogen werden. Es stellt sich nun die Frage, ob Umformungsoperationen simuliert werden müssen.
Man betrachte noch einmal die Merkmalsattribute aus Kapitel 3.1.2: Wenn zwei Merkmale gegeben sind, müssen nur Knoten addiert werden, um die einzelnen Knoten abzugleichen. Dabei müssen natürlich die Knoten des längsten Pfades mit berücksichtigt werden. Auch müssen nur Knoten und Kanten hinzugefügen werden, um den längsten Pfad abzugleichen. Bei den Zykelgrößen genügt es dagegen, die größten Zykel zu ermitteln und miteinander zu vergleichen, denn ein großer Zykel kann durch Einfügen geeigneter Kanten zum Erzeugen kleinerer Zykel verwendet werden. Damit lässt sich also eine geeignete $\blacktriangleleft$-Relation für die Nachbarschaftsmerkmale effizient implementieren (siehe Algorithmus 1). Bild 3.6 verdeutlicht den besprochenen Zusammenhang noch einmal.

3.2.3 Die symmetrische Klassifikation der Kanten

Im Folgenden soll eine ungerichtete Kante zunächst ganz naiv betrachtet werden. Eine Kante besitzt zwei Knoten. Die Nachbarn dieser Knoten können entweder dem einen Ende oder dem anderen Ende der Kante oder beiden zugeordnet werden. Damit ergeben sich fünf Knotenmengen, deren Attributzusammensetzungen als Unterscheidungsmerkmal verwendet werden können. Allerdings ist Vorsicht geboten. Da vom ungerichteten Fall ausgegangen wurde, muss ein Enthaltenseins-Test die Symmetrie entlang der Kante beachten. Das folgende Bild 3.7 veranschaulicht diesen Zusammenhang.
Die beiden Knoten heißen u und sein Nachbar $\bar{u}$. Jeder der beiden besitzt eine Menge von Nachbarn, die nur mit ihm verbunden sind, nämlich S für u und $\bar{S}$ für $\bar{u}$. Zusätzlich ist ihnen die Nachbarknotenmenge M gemeinsam. Aus diesen Mengen lässt sich ein 5-Tupel an Attributmengen $(S,u,M,\bar{u},\bar{S})$ zur Beschreibung der Kante bilden.

Definition 15 (Kanten-5-Tupel). *Sei $e = \{u,v\}$ beliebige Kante und $c : V \longrightarrow M$ ein beliebiger Klassifikator. Sei weiterhin für eine Knotenmenge V die Menge aller Merkmale gegeben als $K(V) = \{c(w) : w \in V\}$. Dann ist $c : E \longrightarrow \mathcal{P}(M) \times M \times \mathcal{P}(M) \times M \times \mathcal{P}(M)$ eine **Kantenklassifikator** mit*

$$c(e) = (S,t,M,\bar{t},\bar{S}) \textbf{ Kanten-5-Tupel} \tag{3.27}$$

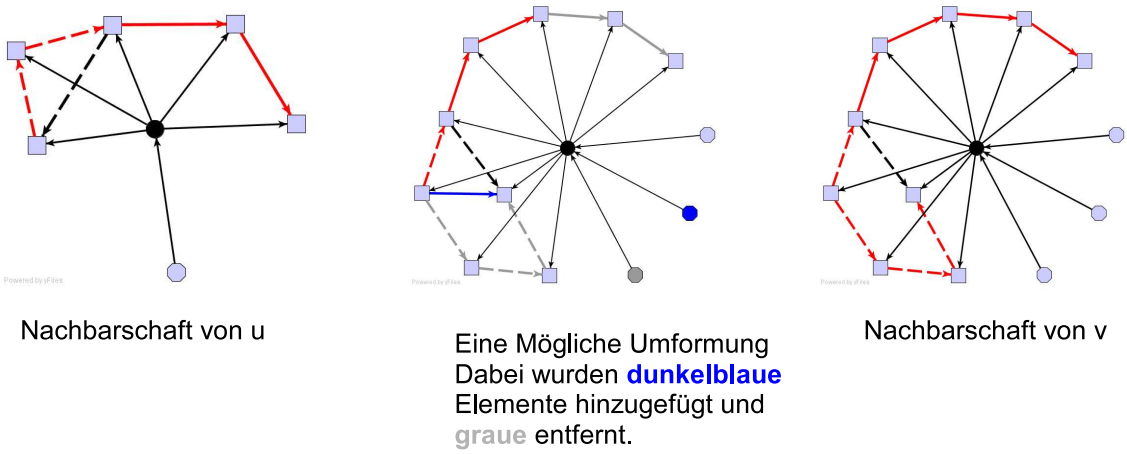


Abbildung 3.6: Die Nachbarschaften von u und v unterscheiden sich bezüglich einzelner Knoten, des längsten Pfades, sowie des Zykels. Durch das Hinzufügen blauer Elemente, sowie das Entfernen von Grauen, lässt sich aus beiden ein für beide gleicher Zwischengraph konstruieren. Damit ist die Nachbarschaft von u, gemäss der $\blacktriangleleft$-Relation, in der von v enthalten.

Dabei ist $t = c(u)$, $\bar{t} = c(v)$, *die beiden Knoten gemeinsame Merkmalsmenge* $M = K(N(u) \cap N(v))$ *und* $S = K(N(u)) \setminus M$, $\bar{S} = K(N(v)) \setminus M$.

Da wir mit dem Prinzip der Klassifikation arbeiten und die Knoten bis zur Zuordnung keine Identität besitzen dürfen, enthalten diese Mengen die Klassifikation der jeweiligen Knoten.
Bevor nun der Backtracking-Algorithmus der Subgraphisomorphie beschrieben wird, muss noch die $\blacktriangleleft$-Relation für Kanten-5-Tupel definiert werden:

Definition 16. *Seien* e *und* e' *zwei beliebige Kanten und* c *ein Kantenklassifikator mit* $c(e) = t = (S,t,M,\bar{t},\bar{S})$ *und* $c(e') = t' = (S',t',M',\bar{t}',\bar{S}')$. *Sei zudem eine Untermengenrelation* $\subseteq_M$, *welche die* $\blacktriangleleft$*-Relation beachtet, für Merkmalsmengen* A,B *wie folgt definiert:*

$$A \subseteq_{\blacktriangleleft} B \Leftrightarrow \forall x \in A \exists f(x) = y : x \blacktriangleleft y \;\&\; f : A \to B \text{ injektiv} \tag{3.28}$$

Dann ist die $\blacktriangleleft$*-Relation über den Kanten-5-Tupel wie folgt definiert:*

$$\begin{aligned} t \blacktriangleleft t' \Longleftrightarrow & (S \subseteq_{\blacktriangleleft} S' \;\&\; t \blacktriangleleft t' \;\&\; M \subseteq_{\blacktriangleleft} M' \;\&\; \bar{t} \blacktriangleleft \bar{t}' \;\&\; \bar{S} \subseteq_{\blacktriangleleft} \bar{S}') \vee \\ & (S \subseteq_{\blacktriangleleft} \bar{S}' \;\&\; t \blacktriangleleft \bar{t}' \;\&\; M \subseteq_{\blacktriangleleft} M' \;\&\; \bar{t} \blacktriangleleft t' \;\&\; \bar{S} \subseteq_{\blacktriangleleft} S') \end{aligned} \tag{3.29}$$

Mit diesem Hintergrund kann nun zum Backtracking-Verfahren übergegangen und sich mit den Unterschieden zur Graphisomorphie beschäftigt werden.

3.2.4 Der Backtrackingalgorithmus

Mit Hilfe der $\blacktriangleleft$-Relation können sehr schnell alle in Frage kommenden Startpunkte für das Backtrackingverfahrens gefunden werden. Obwohl man meinen könnte, dass diese Relation aufgrund ihrer konservativen Definition relativ wenig Nutzen hat, ist sie dennoch in der Lage Knoten auszuschließen. Besitzt u z.B. drei freie Knoten, so kann er nicht mehr in der Nachbarschaft $N(v)$ enthalten sein, wenn diese zwei freie Knoten erfordert. Ähnliche Überlegungen ergeben sich für den längsten Pfad und die Zykelgrößen. Damit ist die $\blacktriangleleft$-Relation trotz ihres konservativen Verhaltens brauchbar.

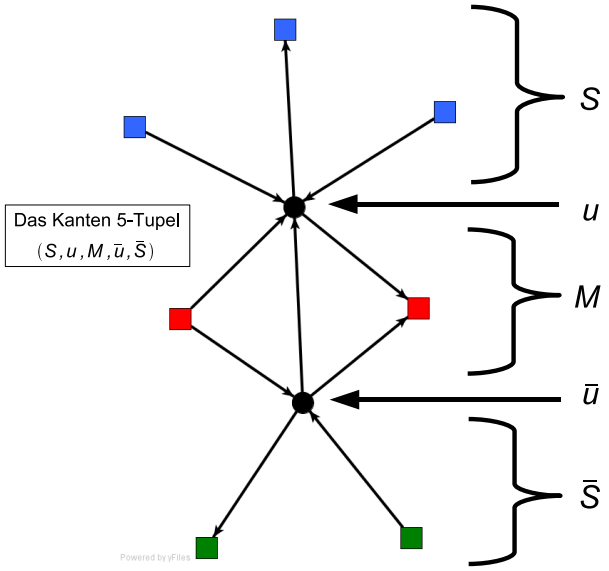


Abbildung 3.7: Jeder Knoten einer Kante stellt eine Menge u dar. Mit dieser ist die zu ihm gehörende Nachbarschaftsknotenmenge S assoziiert. Da es sich um ungerichtete Kanten handelt, besitzt dieser Knoten den Komplementknoten $\bar{u}$. Daher hat die Nachbarmenge S die Komplementmenge $\bar{S}$. Allein die gemeinsamen Knoten M besitzen keine Gegenmenge.

Aus ihr ergibt sich der erste Schritt des Subgraphisomorphiealgorithmus als Suche nach möglichen Startknoten. Diese gestaltet sich wie folgt: Durch die Berechnung der Merkmalsrekursion für das Muster kann ein möglichst markanter Startknoten in diesem gesucht werden. Der Startknoten sollte so gewählt werden, dass möglichst wenig andere Knotem mit derselben Merkmalszahl in G existieren. Dieser Knoten wird nun als *Ankerknoten* mit u_G^a bezeichnet. Von ihm ausgehend wird einen Durchwanderungspfad für G in derselben Art berechnet, wie es beim Backtracking des Graphisomorphiealgorithmus der Fall ist. Nun werden alle Knoten u_H des Hosts H durchwandert und mittels der $\blacktriangleleft$-Relation getestet (siehe Appendix *subgraph-isomorphy*). Damit werden alle möglichen Abbildungen berechnet, die zwischen u_G^a und den Knoten aus H bestehen können.

Definition 17. *Das Paar* $(u_G^a, u_H) = (u_G^a, u_H^a) : u_G^a \blacktriangleleft u_H^a$ *wird* ***Ankerpaar*** *genannt.*

Von diesen Ankerpaaren ausgehend kann eine Abbildung von G nach H versucht werden.
Allerdings sind vorher noch das Kantenhistogramm $\mathcal{E}_H^0$ für g^0 und die Kanten, deren Abstand zu u_H^a kleiner ist, als die maximale BFS-Tiefe in G, in H zu berechnen. Dieses wird mit dem Kantenhistogramm $\mathcal{E}_G^0$ (grob) verglichen, um festzustellen, dass sich ein Backtracking-Versuch wirklich lohnen könnte. Dafür reicht es aus, lediglich das Enthaltensein jeder Kategorie aus $\mathcal{E}_G^0$ zu überprüfen, und ob es für diese in $\mathcal{E}_H^0$ ausreichend Kanten gibt.
Wie auch beim Graphisomorphiealgorithmus werden Knoten bei der Rekursion entlang eines Durchwanderungspfades mit der Merkmalsklassifikation und der $\blacktriangleleft$-Relation getestet und bei Verwendung markiert.
Der Pseudocode für dieses Verfahren findet sich im Anhang A.3.2. Um die Belegung abschließend zu testen, wird für die erstellte Kantenzuordnung noch ein verkürzter Isomorphietest durchgeführt. Dieser wird im nächsten Kapitel besprochen.

3.2.5 Der verkürzte Isomorphietest

Wenn der Rekursionsfuß im Algorithmus A.3.2 erreicht wird, ist eine Abbildung von $E_G \longrightarrow E_H$ berechnet worden. Diese stellt einen zusammenhängenden Teilgraphen dar und alle Knotenmerkmale in G sind bezüglich der $\blacktriangleleft$-Relation in denen assoziierten Knoten in H enthalten. Da die $\blacktriangleleft$-Relation als keine wirkliche Enthaltenseins-Relation anzusehen ist, sollte das Ergebnis an diesem Punkt noch einmal einer Prüfung unterzogen werden. Allerdings sollte diese so kurz und genau wie möglich sein, um im Falle eines Fehlschlages nicht zuviel Rechenzeit aufzuwenden.
In Kapitel 3.1.4 wurde schon erwähnt, dass die Merkmalsrekursion wahrscheinlich $\mid V \mid$ viele Rekursionsschritte benötigt, um eine Auftrennung der Knoten zu erhalten, die es ermöglicht, eine Graphisomorphieabbildung aufzuzählen. Doch in diesem Augenblick wird nur eine Bestätigung der Isomorphie benötigt, möglichst ohne viele Iterationen. Also stellt sich die Frage: Geht es schneller?
Man betrachte die Situation einmal etwas näher! Es wurden exakt soviele Kanten in H ausgewählt, wie es Kanten in G gibt. Daraus und aus der Tatsache, dass diese eine Zusammenhangskomponente darstellen, folgt auch, dass wir genau so viele Knoten aus H, wie in G abgebildet haben. Es ist also einzig zu prüfen, ob diese korrekt verbunden sind. Genau genommen, ist die topologische Korrektheit der Abbildung zu testen. Und dazu kann wieder das Prinzip der *Gleichheit unter Interpretation* verwendet werden, wie es die Merkmalsrekursion bietet.
Geht man davon aus, dass zwei Graphen mit gleicher Knoten- und Kantenzahl gegeben und weiterhin jedem Knoten ein Merkmal zugeordnet ist, so dass die Merkmalshistogramme für g^0 beider Graphen gleich sind. So stellt sich die Frage, auf welcher Rekursionsstufe topologische Unterschiede frühestmöglich erkannt werden können? Das ist direkt nach dem ersten Rekursionsschritt mit dem Histogramm für g^1 der Fall. Im Folgenden werden die Ideen zum Beweis dieser Behauptung, die sich als recht zuverlässig erwiesen hat, skizziert.

Beweisskizze Es gibt zwei mögliche Fehler: Die falsche Zuordnung zweier Merkmale oder die falsche Verknüpfung zweier Knoten, die G und H unterscheiden können, obwohl die Anfangshistogramme Gleichheit aufweisen. Nehmen wir an, es gebe nur einen einzigen Fehler dieser Art zwischen G und H. Um die Auswirkung dieser Fehler zu betrachten, müssen nur die betroffenen Knoten und ihre nähere Umgebung untersucht werden, da sich der Rest bei nur einem Fehler korrekt verhält. Es ist klar, dass es sich dabei um Austauschfehler handelt, d.h. es gibt zwei Orte in H, an denen sich Unterschiede bemerkbar machen müssen: In der Umgebung der beiden Knoten, zwischen denen der Merkmalsaustausch statt gefunden hat oder an den Knoten, die eine Kante verloren bzw. dazu bekommen haben. Der Merkmalsaustausch macht sich dadurch bemerkbar, dass alle Knoten in der Nachbarschaft eines der betroffenen Knoten im Rekursionsschritt eine abweichende Merkmalsstelle in ihrer Merkmalszahl erhalten. Der Kantenaustausch macht sich darin bemerkbar, dass bei den betroffenen Knoten weniger, bzw. mehr der richtigen Merkmale ankommen, diese sich also im Merkmalskoeffizienten unterscheiden. Die folgende Abbildung 3.8 soll diese Fälle verdeutlichen. In beiden Fällen werden die Unterschiede mit hoher Wahrscheinlichkeit erkannt und sind eventuell sogar lokalisierbar, was fortgeschrittenen Algorithmen weitere Möglichkeiten eröffnet. Ein Beweis hierfür findet sich in Kapitel 3.1.6. Pseudocode findet sich im Anhang A.3.3.

3.2.6 Die Probleme der Rekursion

Abschließend stellt sich die Frage, warum die Rekursion an sich nicht hilfreich beim Aufspüren von Subgraphen ist? Man betrachte einmal das grundlegende Entwurfsprinzip dieser Gleichung: Sie transformiert wie auch immer gestaltetes konkretes, lokales Wissen über die Topologie der Nachbarschaft in implizites, globales Wissen über die topologische Platzierung eines Knotens. Es wurde auch schon bemerkt, dass dieses in den Koeffizienten der Merkmalszahlen „gespeichert“ ist, da den Knoten keine

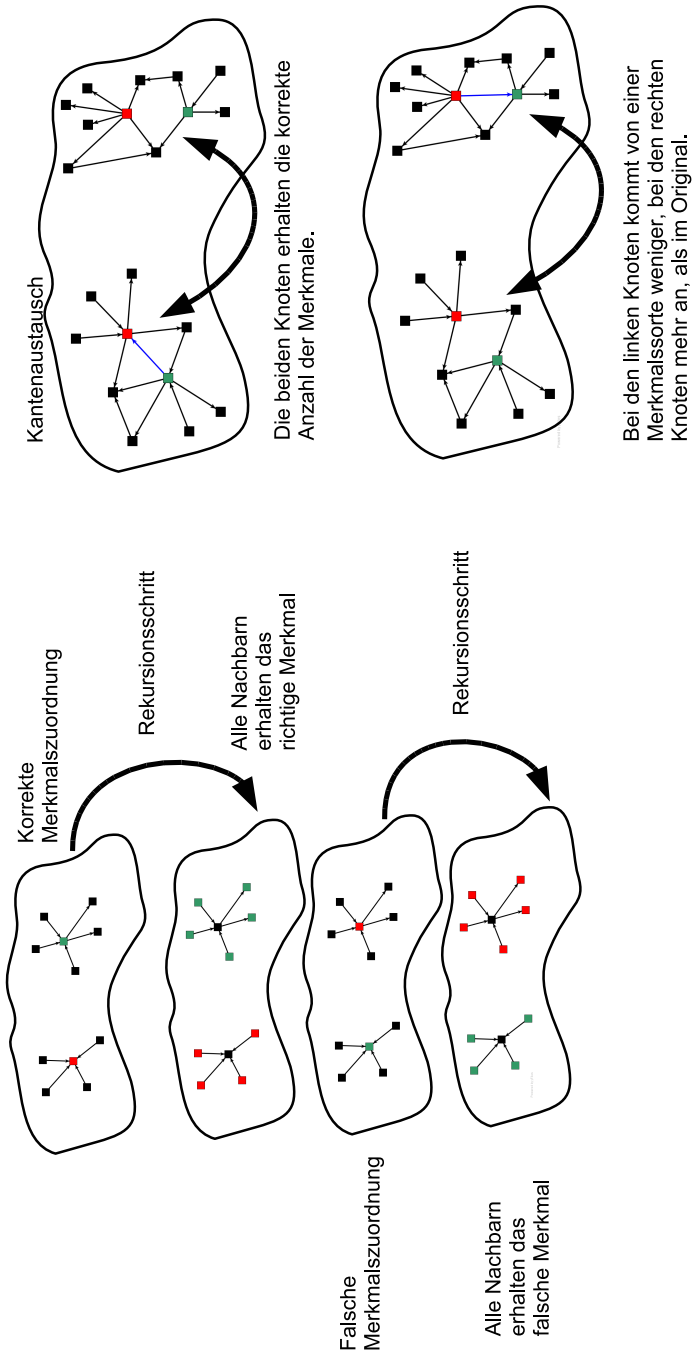


Abbildung 3.8: Diese Abbildung stellt die Fälle dar, die beim Test einer möglichen Isomorphieabbildung auftreten können. Da es sich nur um Austauschfehler von Merkmalen oder Kanten handeln kann, werden immer zwei Positionen in einem Graphen betrachtet. Die obere Hälfte zeigt dabei, die Situation ohne Fehler, während in der unteren Hälfte jeweils der Austausch von Merkmalen (linke Hälfte) oder Kanten (rechte Hälfte) stattgefunden hat. Die Auswirkungen des Austauschs sind farbig hervorgehoben.
Auf der linken Seite werden in der unteren Hälfte das rote und das grüne Merkmal vertauscht, mit der Folge, dass die jeweiligen Nachbarschaften bzgl. der oberen Situation die falschen Merkmale erhalten. Auf der rechten Seite wird die blaue Kante an einen anderen Ort verlegt, so dass sich durch den fehlenden „Merkmalsfluß“ unterschiedliche Merkmalszahlen für die Nachbarschaften des roten und grünen Knotens bzgl. der oberen Situation ergeben müssen, bei sonst korrekter Abbildung der Merkmale.

Identitäten gegeben und somit keine darauf basierenden Datenstrukturen verwenden werden konnten. Allerdings wurde damit Speicherplatz gespart.
Um also die ◄-Relation, die für die Merkmale gefordert wird, auf die Merkmalszahlen übertragen zu können, müssten deren Operationen teilweise rückgängig gemacht werden können. Dazu ist jedoch topologisches Wissen sowie Kenntnis der Knoten und Kanten nötig, deren Einfluss herausgerechnet werden soll. Weiterhin muss beachtet werden, dass mit zunehmender Rekursionstiefe die Auswirkung entfernter Knoten, sowie Rückkopplungseffekte zunehmen. Im ersten Fall steigt die Komplexität mit der Anzahl der einflussnehmenden Knoten, im zweiten mit den Rückkopplungseinflüssen. Schließlich erhalten die Knoten in der darauffolgenden Runde genau die Information von ihren Nachbarn zurück, die sie vorher an diese weitergegeben haben, ganz im Gegensatz zum echten Routing, um wieder auf die Einstiegsanalogie (Internet) zurückzukommen. Beide Effekte auseinander zuhalten erweist sich als fast unmöglich, bzw. führt auf die Knotenidentifikationsproblematik und der damit verbunden Komplexität zurück.
Allerdings wurde beim Versuch eine interessante Beobachtung gemacht, die Ziel künftiger formaler Untersuchungen sein könnte und hier nun vorgestellt werden soll. Man betrachte einmal den „Merkmalszuwachs“, den ein Knoten in einem Rekursionsschritt erhält und definiere ihn als Ableitung. Die Merkmalsrekursion, wie aus Definition 8, lautet dann:

$$
\begin{aligned}
g^0(u) &:= (1, m) \\
g^{i+1}(u) &:= g^i(u) + \sum_{v \in N(u)} g^i(v) \\
\Longrightarrow g^{i\prime}(u) &= g^{i+1}(u) - g^i(u) = \sum_{v \in N(u)} g^i(v) \quad \textbf{Merkmalsableitung}
\end{aligned}
$$

Damit ist nun offensichtlich, dass die Ableitung von g von der Topologie abhängig ist, was allgemeine Differentialrechnungen kompliziert bis unmöglich macht. Hinzu kommt, was bisher noch nicht erwähnt wurde, dass die Merkmalszahlen alles andere als ein mathematischer Körper sind, sich aber mit etwas Geschick zu einem Vektorraum erweitern lassen. Auch hier können weiterführende theoretische Betrachtungen zu interessanten Ergebnissen führen.

3.3 Frequent Subpattern

3.3.1 Einleitung

Nachdem Graphen verglichen und Subgraphen gefunden werden können, stellt sich nun die Frage, ob es in einem gegebenen Graphen G häufige Teilgraphen, sogenannte *Motife* gibt. Diese Motife würden sozusagen die „Grundbausteine“ darstellen, aus denen G konstruiert wurde. Die Kenntnis solcher Motife kann damit zum Verständnis der Konstitution von G beitragen. So wäre es möglich die Darstellung von G zu vereinfachen: Motife könnten als Module klassifiziert und eigenständig betrachtet werden. Sie könnten dann auch bei der Darstellung einheitlich markiert werden und so zu weiterem Verständnis beitragen. Auch die Frage, ob ein Graph nach einer bestimmten Vorschrift konstruiert wurde, könnte besser diskutiert werden, wenn die Motife eines Graphen bekannt wären. Eine weitere Idee könnte sein, Graphen anhand der Motife auf Ähnlichkeit zu überprüfen.
Gerade diese Fragen sind es, welche die Bioinformatik in der letzten Zeit zunehmend beschäftigen. Ist es möglich, das biologische Netzwerke aus Bausteinen aufgebaut sind? Und wenn ja, nach welchen Prinzipien funktionieren sie? Wie und aus welchem Grund arbeiten sie zusammen? Wie sind sie entstanden?
Die Arbeiten von Uri Alon [42] und Falk Schreiber [18] setzen sich mit diesem Thema auseinander. Dort werden ein *Exhaustive Search*-Algorithmus und ein randomisierter Sampling-Algorithmus vorgestellt, die beide vom Autor dieser Arbeit getestet wurden. Allerdings fiel die Wahl auf den randomisierten Algorithmus, da dieser dem Autor wesentlich leistungsfähiger erschien. In den Arbeiten von Uri Alon

finden sich darüber hinaus auch Erklärungen zu den gefundenen Motifen.
Bevor es nun zur Suche übergeht, muss erst einmal Klarheit darüber bestehen, was Motife genau sind. Die möglichen Ansätze und Definitionen finden sich in [18] und werden im Kapitel 3.3.2 besprochen. Danach wird der Sampling-Algorithmus von Uri Alon [42] in Kapitel 3.3.3 beschrieben. In Kapitel 3.3.4 werden dann die an diesem Algorithmus vorgenommenen Erweiterungen erklärt und im darauffolgenden Kapitel anhand eines stochastischen Urnen-Modells veranschaulicht. Dieser Teil endet schließlich in einer Diskussion der Verfahren in Kapitel 3.3.6.

3.3.2 Häufige Subgraphen

Der Begriff *Motif* bezeichnet zunächst einmal lediglich einen Teilgraphen. Darüber, ob und wie häufig dieser Teilgraph sein sollte, bzw. ob er eine funktionale Komponente darstellt, herrscht keine eindeutige Meinung. Geht man z.B. von einem funktionalen Standpunkt aus, so ist es nicht unbedingt nötig, dass der Teilgraph häufig im Netzwerk vorkommt. Allerdings geht man oft davon aus, dass häufige Komponenten auch funktional sind, so dass dieser Motif-Begriff auf jeden Fall berücksichtigt werden sollte. Hier soll im Folgenden von *häufigen Teilgraphen* (*Frequent Subpattern*) ausgegangen werden, wenn von Motifen die Rede ist.
In [18] werden vier Häufigkeitskonzepte (vgl. Tabelle 3.1) vorgestellt, die bei der Suche nach Motifen verwendet werden können. Diese regulieren, ob die gefundenen Muster gemeinsame Komponenten haben dürfen. Dabei ergeben sich drei praktikable Konzepte, die verwendet werden können. Konzept $\mathcal{F}^*$

	Gemeinsame Elemente	
Konzept	Knoten	Kanten
$\mathcal{F}_1$	Ja	Ja
$\mathcal{F}_2$	Ja	Nein
$\mathcal{F}^*$	Nein	Ja
$\mathcal{F}_3$	Nein	Nein

Tabelle 3.1: Die Frequenz-Konzepte regeln, ob Muster gemeinsame Komponenten besitzen dürfen oder nicht. Sie spielen hauptsächlich bei der Reduktion der Laufzeit eine Rolle.

entfällt, da Motife mit gemeinsamer Kante auch gemeinsame Knoten haben müssen. Diese können dann in einem Algorithmus verwendet werden, der alle möglichen Instanzen eines Musters für ein bestimmtes Konzept aufzählt, was natürlich sehr aufwändig werden kann. Allerdings stellt sich die Frage, ob es die Wiederverwendung von Graphkomponenten ist, die unterschieden werden muss. Ein Grund dafür ist sicherlich die Verringerung des Suchaufwands. Ob das darüber hinaus sinnvoll ist, steht noch zur Diskussion. Die Ergebnisse eines solchen Verfahrens sind für die Interaktionsdaten von *E.coli* in „Towards Motif Detection“ [18] veröffentlicht. Der dort verwendete Graph besitzt genau dann eine Kante zwischen zwei Knoten, die Proteine darstellen, wenn diese miteinander interagieren. Auf eine Analyse der biochemischen Hintergründe wurde dort verzichtet.
In [42] wird von einer anderen Eigenschaft der Motife ausgegangen:

> Ein häufiges Motif wird bei der zufälligen Auswahl von Teilgraphen wahrscheinlicher gefunden, als andere Teilgraphen derselben Größe.

In diesem Fall muss man sich keine Gedanken mehr über Häufigkeitskonzepte machen. Es bleibt nur die Wahl, ob schon einmal gezogene Komponenten zugelassen werden oder nicht, und wie die Teilgraphen aussehen, die gezogen werden. Da Teilgraphen zusammenhängend sein sollten, bleibt nur die Wahl, ob zufällig Knoten oder Kanten ausgewählt werden, um eine Stichprobe zu ermitteln. Das führt zu zwei Strategien:

knotenbasiert Ein aufspannender Baum wird im Graph ausgewählt. Alle Kanten zwischen den Knoten dieses Baumes bilden den ausgewählten Teilgraphen.

kantenbasiert Eine zusammenhängende Kantenmenge wird ausgewählt. Der ausgewählte Teilgraph besteht aus genau diesen Kanten und den zu ihnen adjazenten Knoten.

Dieser Unterschied ist für die Ergebnisse und deren Auswertung des in Kapitel 3.3.4 vorgeschlagenen Algorithmus bedeutsam. Da es bisher nur diese beiden Algorithmen randomisierter Natur gibt, die Graphen auf Motife hin untersuchen, steht eine Diskussion dieser Strategien noch aus.
Wenn also mit einer der beiden Strategien genügend Stichproben an Teilgraphen ausgewählt werden, so konvergieren - nach dem *schwachen Gesetz der großen Zahlen* [31] - die gezählten Häufigkeiten der gefundenen Topologien gegen deren Wahrscheinlichkeiten, da es sich um einen diskreten Wahrscheinlichkeitsraum handelt.
Umgekehrt könnten berechnete Wahrscheinlichkeiten als Schätzwert für die Häufigkeiten angesehen werden. Dabei ist zu beachten, dass der Wahrscheinlichkeitsraum von dem betrachteten Graphen G abhängt, der über seine Topologie die Häufigkeiten der Motife mit gesuchter Grösse, bzw. die Wahrscheinlichkeit für das Auffinden eines Motifs bestimmt. Allerdings kann ein und dasselbe Motif mit unterschiedlichen Wahrscheinlichkeiten gefunden werden, je nach Häufigkeit und topologischer Einbettung. In diesem Zusammenhang entsteht die Problematik, dass Knoten mit großem Grad eine höhere Wahrscheinlichkeit haben, mit einem Teilgraphen gefunden zu werden, als andere. Diese Knoten werden als *Hubs* bezeichnet.
Bevor es zum algorithmischen Teil übergehen, sollen einige der bekannten Motife vorstellt werden. Es handelt sich dabei um die *Autoregulation*, den *Feed-forward-loop*, das *Single-input-module* und die *Dense-overlapping-region*. Diese sind in Abbildung 3.9 dargestellt und in [2] ausführlich beschrieben.

Autoregulation Die Autoregulation kommt in zwei Varianten vor: Dem *positiven* und dem *negativen Feedback-Loop*. Sie ist in Organismen dann zu finden, wenn es um darum geht, eine schnelle Antwort zu erhalten. In biochemischen Modellen erreichen Gene unter Autoregulation schneller ihre Sättigungskonzentration, als ohne.

Feed-forward-loop Der *Feed-forward-loop* besteht aus drei Komponenten, die so miteinander verschaltet sind, dass es einen direkten und einen indirekten Regulationspfad für ein bestimmtes Ziel gibt. Die Regulation an sich kann dabei jeweils positiv oder negativ sein. Es gibt 14 Varianten, die sich in *kohärente* und *inkohärente* Schleifen unterteilen, je nachdem, ob das Vorzeichen der beiden Pfade gleich ist oder nicht. Der Feed-forward-loop beschleunigt in einigen Fällen das Erreichen eines bestimmten Konzentrationslevels und ermöglicht es, Signalsschwankungen zu glätten. In anderen Varianten ist dieses Motif in der Lage, Oszillationen hervorzurufen. Der Feed-forward-loop ist in fast jeder bisher untersuchen Netzwerkvariante zu finden. Sei es in Interaktionsnetzwerken, in Genregulationsnetzwerken oder in neuronalen Netzwerken, wie in dem von *C.elegans*. Es scheint ausreichend viele Gründe für die häufige unabhängige evolutionäre Entwicklung dieses Motifs in den verschiedensten Netzwerken zu geben.

Single-input-module Das *Single-input-module* ist im Prinzip ein Multi-output Feed-forward-loop und besitzt zum großen Teil auch dessen Eigenschaften. Zusätzlich ist es in der Lage, zeitgesteuerte Abläufe zu erzeugen, je nach Parameter der regulierten Gene für die beiden steuernden Komponenten. Dabei werden so regulierte Gene häufig in einer anderen Reihenfolge eingeschaltet, als sie ausgeschaltet werden.

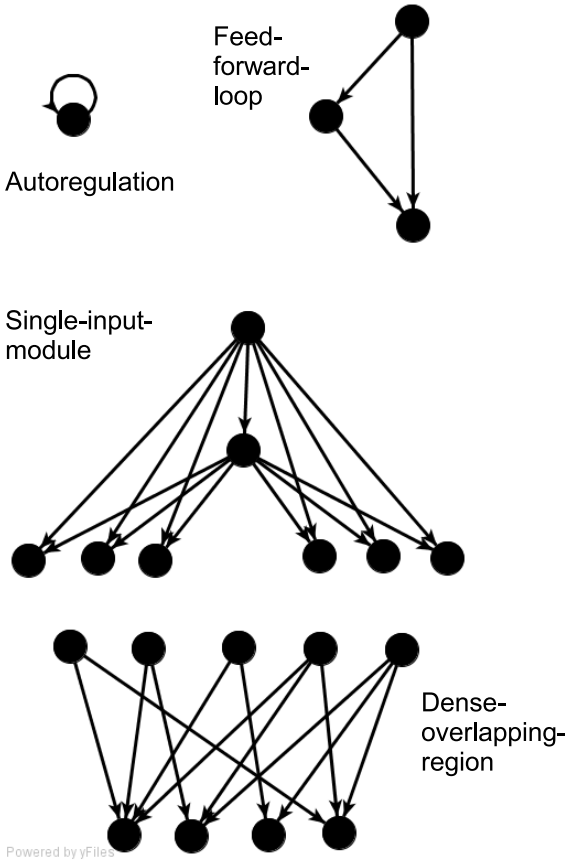


Abbildung 3.9: Bekannte Motife

Dense-overlapping-region Die *Dense-overlapping-region* besteht aus einem bipartiten Graphen. Es gibt eine Menge an regulativen Komponenten, die eine Menge von Zielkomponenten mit einem hohen Konnektivitätsgrad regulieren. Dieses Motif ist hauptsächlich als Entscheidungsfunktion zu finden. Seine biochemische Modellierung weist mathematische Parallelen zu Klassifikatorfunktionen aus Machine-Learning-Algorithmen. Diese Klassifikatorfunktionen bestehen meistens aus einem Gewichtsvektor für eine Menge an Eingabesignalen. Je nachdem, ob das Skalarprodukt dieses Vektors mit einem mehrdimensionalen Eingabevektor größer oder kleiner Null ist, wird die Eingabe mit 0 oder 1 klassifiziert. In den gefundenen biologischen Netzwerken übernehmen die Reaktionsparameter der teilnehmenden Transkriptionsfaktoren die Rolle der Signalgewichte. Mit diesem Netzwerkmotif kann die Natur also lernen. Bei den beiden letzteren Motifen handelt es sich um sogenannte *generalisierte Motife*, d.h. für diese kann eine Struktur der Vernetzung angegeben werden, nicht aber eine konkrete Topologie. Damit ist deren Detektion mit jedem der oben erwähnten Konzepte kaum möglich.

3.3.3 Der Sampling-Algorithmus

Hier wird nun die ursprüngliche Variante des Sampling-Verfahrens von Uri Alon vorgestellt. Der Efficient Sampling-Algorithmus [42] arbeitet wie in Listing A.4.1 dargestellt. Er berechnet Schätzungen für die Subgraphkonzentrationen, die wie folgt definiert sind:

Definition 18. *Die **Subgraphkonzentration** C_i eines Subgraphen vom Typ i im Graph G ist definiert als*

$$C_i = \frac{N_i}{\sum_i N_i} \tag{3.30}$$

mit N_i der Anzahl an Instanzen vom Typ i in G.

Zur Berechnung dieser geht er dabei wie folgt vor: In der Hauptschleife werden erweiternde Kanten aus G ausgewählt, bis diese eine Knotenmenge von n Knoten bilden. Diese Knoten und alle Kanten zwischen ihnen bilden die gefundene Instanz. Für diese Instanz wird die Wahrscheinlichkeit P_i für deren Auswahl berechnet. Dazu werden alle aufspannenden Bäume dieser Instanz aufgezählt und die Wahrscheinlichkeit berechnet, mit der diese Kantenmengen gefunden werden. Genau genommen wird

$$P_i = \sum_{\sigma \in S_m} \prod_{E_j \in \sigma} Pr[E_j = e_j \mid (E_1, \ldots, E_{j-1}) = (e_1, \ldots, e_{j-1})], \tag{3.31}$$

mit S_m als Menge aller $(n-1)$-Kantenpermutationen des gefundenen Subgraphen, die zu dessen Auswahl führen können und E_j die j.te Kante einer bestimmten Permutation, berechnet. Nachdem dies k mal berechnet und L Subgraphtypen gefunden wurden, wird für jeden Subgraphtypen i die geschätzte Konzentration

$$C_i = \frac{S_i}{\sum_{k=1}^{L} S_k}, \text{ mit } S_i = \sum \frac{1}{P_i} \tag{3.32}$$

berechnet, wobei S_i aus den Wahrscheinlichkeitswerten gebildete *Scores* sind. Diese Konzentration kann schon ab ca. 5000 Stichproben als guter Schätzwert angesehen werden.
Die Laufzeit dieses Verfahrens hängt hauptsächlich von der Größe der Stichprobe, sowie des Auswahlverfahrens ab, nicht aber von der topologischen Komplexität des Graphen selbst.

3.3.4 Die Erweiterung

Während das oben beschriebene Verfahren eine Stichprobe an Teilgraphen nimmt, und durch die von der Topologie abhängigen, berechneten Wahrscheinlichkeiten einen Schätzwert für die Konzentration eines Motifs berechnet, versucht der hier angegebene Algorithmus eine vollständige Zerlegung des Graphen in Teilgraphen bestimmter Größe und deren relative Häufigkeit zu berechnen.
Er geht dabei wie in Listing A.4.2 beschrieben vor: Der Algorithmus wählt in der Hauptschleife analog zum vorher besprochenen Verfahren zufällig Kanten des Graphen aus, die einen zusammenhängenden Subgraphen bilden. [3] Als nächstes testet er, ob diese Topologie schon einmal gefunden wurde. Dazu wird der schnelle Graphisomorphiealgorithmus aus Kapitel 3.1 verwendet. Da Graphen keine Ordnung besitzen, werden die Graphen in einer Liste mit „Move-to-Front"-Regel gehalten, da diese im schlechtesten Fall nachweisbar zwei mal so langsam ist, wie die optimale Liste. Wurde diese Topologie noch nicht entdeckt, so wird sie in die Liste aufgenommen und der Zähler der Hauptschleife auf k zurückgesetzt. Wurde die Topologie schon einmal gefunden, wird deren Häufigkeit um eins inkrementiert. Am Ende werden aus den gezählten absoluten Häufigkeiten h der Topologien relative Häufigkeiten h_T für diese Stichprobe berechnet und ausgegeben.
Es handelt sich bei diesem Verfahren also um einen Monte-Carlo-Algorithmus, der mit einer gewissen Wahrscheinlichkeit alle tatsächlichen Subgraphen mit n Kanten, sowie deren Häufigkeiten aufzählt. Damit hat man eine Zerlegung von G in Komponenten der Größe n berechnet, also die „Grundbausteine" von G ermittelt.
Der Algorithmus läuft insgesamt solange, bis er k mal unabhängig hintereinander keine neue Topologie in G entdeckt hat. Damit ist k ein Laufzeit- und Güteparameter zugleich. Er, die Mustergröße und die Topologie von G bestimmen im wesentlichen die Laufzeit des Algorithmus.

[3] Als Nebenprodukt berechnet er ebenfalls die Wahrscheinlichkeit für das Auffinden der aktuellen Instanz. Allerdings wird diese in der tatsächlichen Ausgabe nicht berücksichtigt.

3.3.5 Formale Betrachtungen zum Frequent Subpattern Algorithmus

Nachdem die vorherigen Kapitel relativ informell gehalten waren, soll nun das Frequent Subpattern-Problem einmal aus stochastischer Sicht betrachtet werden. Es ist also ein Graph $G = (V,E)$ und eine Grösse $k \in \mathbb{N} : k \leq \mid E \mid$ gegeben. Der Parameter k gibt an, wie viele Knoten, bzw. Kanten ein ausgewählter, zusammenhängender Subgraph haben muss, um als Muster zu gelten. Da G eine Struktur endlicher Mengen ist, folgt, dass G nur aus endlich vielen Teilgraphen besteht, die in endlich viele unterschiedliche *Topologien* $\mathcal{T}_i = (V_i, E_i) :\mid E_i \mid = k$ unterschieden werden können. Für jede Topologie gibt es also eine *Häufigkeit* h_i mit der $\mathcal{T}_i$ in G vorkommt. Diese Häufigkeit ist zugleich auch die Wahrscheinlichkeit p_i, mit der $\mathcal{T}_i$ ausgewählt wird. Es handelt sich hierbei um einen diskreten Laplace'schen Wahrscheinlichkeitsraum mit unbekannter Verteilung, aber festem Erwartungswert und fester Varianz. Abbildung 3.10 veranschaulicht diese Situation. Das bedeutet auch, dass nach dem *schwachen Gesetz der großen Zahlen* [31] die beobachteten Häufigkeiten mit wachsender Stichprobengröße gegen die tatsächliche Wahrscheinlichkeit stochastisch konvergieren. Seien also h_i^* die beobachteten Häufigkeiten, dann gilt

$$\lim_{n\to\infty} \frac{h_i^*}{\sum_j h_j^*} = h_i = Prob[\mathcal{T}_i] = \sum_{l=1}^{h_i} Prob[I_i^l] \tag{3.33}$$

mit I_i^l als konkreter *Instanz* von $\mathcal{T}_i$ in G. Dabei hängen die $Prob[I_i^l]$ von der Topologie des Graphen ab. Es gibt also zwei Möglichkeiten, die Wahrscheinlichkeit einer Topologie und damit deren Häufigkeit zu schätzen.

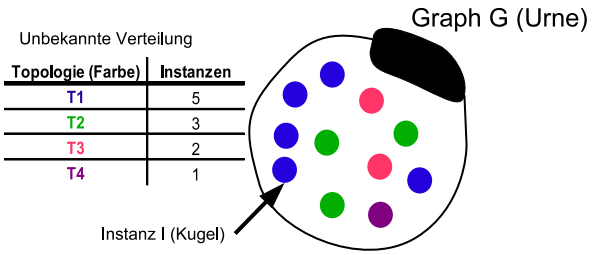


Abbildung 3.10: Der Graph G (Urne) bildet bezüglich der unterschiedlichen Topologien T_i (Kugelfarben), die durch ihre Instanzen I_j (Kugeln) vertreten werden, einen diskreten Laplace'schen Wahrscheinlichkeitsraum mit festem Erwartungswert und fester Varianz.

3.3.6 Diskussion

Auf den ersten Blick scheinen beide Verfahren dasselbe zu leisten: Sie wählen Teilgraphen aus, berechnen Wahrscheinlichkeiten für Instanzen, und geben eine Menge an Topologien und Werte aus, die eine Schätzung für deren Wahrscheinlichkeit sind. Doch es bestehen grundlegende Unterschiede zwischen beiden Verfahren.
Zunächst einmal unterscheiden sie sich in der Definition der Teilgraphen. Algorithmus A.4.1 verwendet eine knotenbasierte Definition, während A.4.2 eine kantenbasierte Defintion benutzt. Beide Definitionen sind zunächst einmal legitim, führen aber scheinbar, wie im Ergebnisteil deutlich wird, zu unterschiedlichen Ergebnissen.
Der größere Unterschied ist allerdings folgender: In dem Verfahren von Uri Alon wird eine Stichprobe fester Größe ausgewählt und, mit Hilfe der aus der Topologie von G berechneten Wahrscheinlichkeiten, eine Schätzung für die Häufigkeit der einzelnen Topologien abgegeben, mit der diese vorkommen.
Bei dem Verfahren dieser Diplomarbeit wird versucht, *alle* vorkommenden Topologien zu ermitteln und aus den *tatsächlich* gezählten Häufigkeiten einer Topologie einen Schätzwert zu berechnen. Dieser ist

unabhängig von der Topologie von G, da er aus der Stichprobe berechnet wird.
Bezüglich des Urnen-Modells in Abbildung 3.10 bedeutet dies: Algorithmus A.4.1 wählt i Kugeln aus, schaut aber jedes Mal, wenn er eine Kugel zieht, wie wahrscheinlich der Griff zu dieser war. Algorithmus A.4.2 hingegen zieht solange Kugeln aus der Urne, bis er k-mal hintereinander keine neue Farbe gefunden hat. Es sind also zwei grundlegend verschiedene Experimente, die hier durchgeführt werden.

Kapitel 4

Ergebnisse

Dieses Kapitel stellt die Ergebnisse der Laufzeitmessungen und der Frequent-Pattern-Analyse vor. Zunächst wurden Laufzeitmessungen auf einem `Intel Pentium 4` Prozessor mit `1.80GHz` und `512 MB` Arbeitspeicher vorgenommen. Diese Messungen wurden mittels der in Tabelle 4.1 angegebenen Datensätze durchgeführt.
Zunächst werden die Zugriffszeiten der Move-to-Front-Liste für die Graphen des Frequent Pattern Algorithmus, die ab jetzt mit *GraphMap* bezeichnet wird, betrachtet. Danach werden die Laufzeiten des Frequent Pattern Algorithmus, des Graphisomorphie-Algorithmus und der Subgraphsuche angegeben. Anschließend werden die Anzahl der gefundenen Mustergraphen sowie die relativen Häufigkeiten betrachtet. Danach werden für jeden Organismus und jeden Datensatz die wahrscheinlichsten Muster bezüglich der Häufigkeiten angegeben und die Organismen abschließend untereinander vergleichen.

Datensatz	Knoten $\mid N \mid$	Kanten $\mid E \mid$
Planar (klein)	50	80
Planar (mittel)	100	200
Planar (groß)	100	300
Random (klein)	50	80
Random (mittel)	100	200
Random (groß)	100	300
Random (dünn)	200	300

Tabelle 4.1: Größen der Eingabe-Graphen zur Laufzeitmessung

4.1 Laufzeiten

4.1.1 Zugriffszeit *GraphMap*

Wie schon erwähnt, handelt es sich bei der *GraphMap* um eine Move-to-Front-Liste, die Tupel aus Graphen und Werten speichert. Dabei werden die Graphen als Schlüssel für den Zugriff verwendet. Wurde ein Graph bei einem Zugriff gefunden, wird sein Eintrag an den Anfang der Liste gehängt, um direkt nachfolgende Zugriffe effizienter zu machen. In „Amortized Efficency of List Update Rules“ [46] wird bewiesen, dass diese Zugriffsmethode maximal doppelt so schlecht ist, wie die optimal sortierte Liste.
In Abbildung 4.1 sind zwei Diagramme zu sehen, welche die mittlere Zugriffszeit in Abhängigkeit von Listengröße (links) und Kantenzahl (rechts) der gespeicherten Graphen darstellen. Im linken Diagramm ist ein deutlicher Anstieg der mittleren Zugriffszeit bei einer Listengröße von ca. 200 Einträgen zu erkennen. Danach werden die Zugriffszeiten deutlich unkorrelierter. Dies kann mit der Move-to-Front-Regel

erklärt werden. Prinzipiell ist diese Aufspreizung als positiv zu werten, da so auch kurze Zugriffszeiten bei langen Listen möglich sind. Auch sehr gut zu erkennen ist, dass die Zugriffszeit im schlechtesten Fall linear mit der Listenlänge wächst. So befinden sich die meisten Datenpunkte unter der roten Geraden, welche die Datenpunkte des schlechtesten Falles interpoliert.
Im rechten Diagramm ist die Abhängigkeit der mittleren Zugriffszeit von der Kantenzahl der Schlüssel-Graphen dargestellt. Diese Kurve erweckt den Eindruck einer exponentiellen Abhängigkeit und ist einer der Gründe, warum die Analyse der Interaktions-Graphen nur für Größen $n \leq 9$ durchgeführt wurde. Allerdings muss erwähnt werden, dass bei jedem Vergleich zweier Graphen in der verwendeten Implementierung der komplette Isomorphiealgorithmus bis zum Histogrammvergleich ausgeführt wurde. Das erforderte, dass jeweils für mindestens einen der zu vergleichenden Graphen die Merkmalsfunktion komplett neu berechnet werden musste, was wie schon angedeutet ein Aufwand von $O(\mid E \mid \cdot \mid V \mid)$ im schlimmsten Fall bedeutet. Ist l die aktuelle Listenlänge, so ergeben sich $O(l \cdot \mid E \mid \cdot \mid V \mid)$ viele Vergleiche, was diesen rapiden Anstieg ebenfalls erklärt.
Da sich die Histogramme für gespeicherte Graphen nicht mehr ändern, sollten in zukünftigen Implementierungen die Histogramme beim Einfügen in die Liste vorberechnet und bei einem Vergleich nicht neu berechnet werden. Damit ließe sich die Zugriffszeit im schlimmsten Fall auf $O(l \cdot \mid V \mid)$ bei $O(\mid V \mid)$ verschiedenen Merkmalen der Histogramme reduzieren. Ebenfalls zu untersuchen wäre, ob aus den Histogrammen eine Ordnung für Graphen ableitbar ist und so Zugriffe mittels binärer Suche möglich werden.

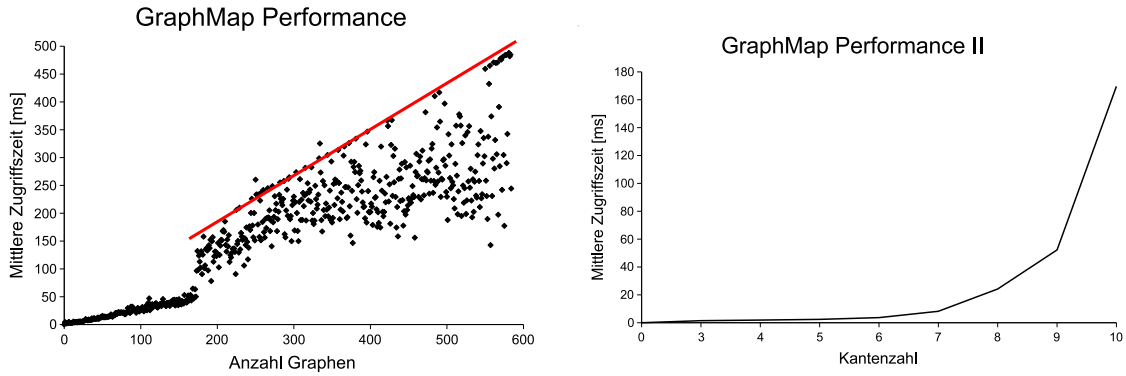


Abbildung 4.1: *GraphMap*-Zugriffszeit in Abhängigkeit der Listenlänge (links) und der Kantenzahl der gespeicherten Graphen (rechts). Im linken Diagramm ist ein deutlicher Effizienzverlust bei ca. 200 Einträgen mit folgender Aufspreizung der Zugriffszeiten zu erkennen. Im rechten Diagramm erkennt man deutlich, dass die mittlere Zugriffszeit scheinbar exponentiell von der Kantenzahl der als Schlüssel verwendeten Graphen abhängt.

4.1.2 Laufzeit Frequent Pattern-Algorithmus

In diesem Abschnitt werden der Frequent Pattern-Algorithmus und dessen Laufzeit in Abhängigkeit der Rundenzahl, sowie der Mustergröße auf dem heterogenen Datensatz der Zufallsgraphen aus Tabelle 2.1 betrachtet. Dabei ist zubeachten, dass dieser Algorithmus stark von den Zugriffszeiten der *GraphMap*, sowie der Graphtopologie (vgl. Urnenmodell in Abbildung 3.10) abhängig ist. In Abbildung 4.2 sind die Laufzeiten des Algorithmus graphisch dargestellt.
Im linken Bild ist die mittlere Laufzeit in Abhängigkeit der Abtastrunden visualisiert. Man sieht deutlich, dass die Laufzeit scheinbar unabhängig von der Rundenzahl variiert. Hier offenbart sich die Randomisierung des Frequent Pattern Algorithmus und dessen Abhängigkeit von der Topologie des Eingabegraphen G, welche die Laufzeit bestimmen.

Im rechten Bild ist die Abhängigkeit der Laufzeit von der gesuchten Mustergrösse dargestellt. Es ist wieder ein scheinbar exponentieller Trend erkennbar. Allerdings lässt sich dies mit der Abhängigkeit des Algorithmus von den Zugriffszeiten der *GraphMap* erklären, die eine ähnliche Charakteristik aufweist. Auch hier ist zu erwähnen, dass die Implementierung der *GraphMap* noch verbessert werden kann.

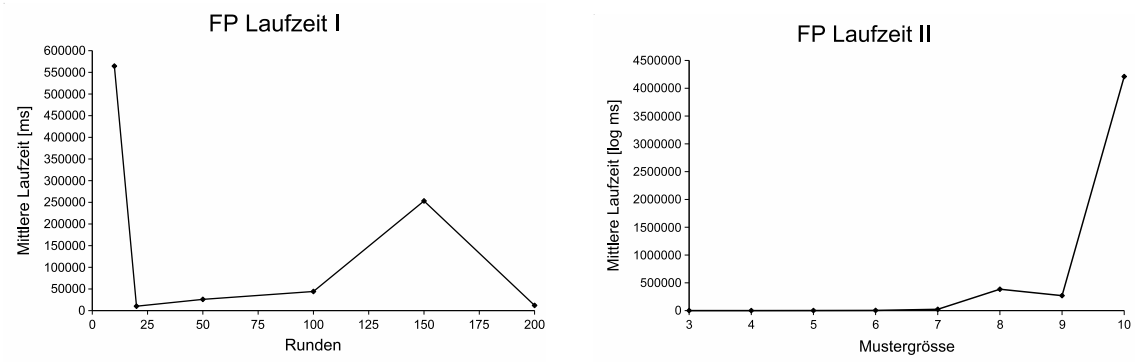


Abbildung 4.2: Die Laufzeit des Frequent Pattern Algorithmus in Abhängigkeit der Runden und der gesuchten Mustergröße. Zu erkennen ist, dass die Laufzeit bezüglich der Runden, stark schwankt, während sie bezüglich der Mustergröße exponentiell abhängig zu sein scheint.

4.1.3 Laufzeit Graphisomorphie-Algorithmus

Um einige Laufzeitmittelwerte für den Graphisomorphietest zu erhalten, wird zusätzlich zu einem kompletten Vergleich der Graphen untereinander noch die durch den Frequent Pattern-Algorithmus erhaltene Teilgraphzerlegungen als Testobjekt verwendet. Auch diese werden untereinander komplett verglichen. Um die Parametermenge klein zu halten, wurden nur bei gefundener Gleichheit die mittleren Laufzeiten bezüglich der Kantenzahl unterschieden. Für die Tests, bei denen Ungleichheit gefunden wurde, bilden wir lediglich den Mittelwert.
Im oberen Abschnitt der Tabelle 4.2 werden die Laufzeiten für die Tests unter den Ergebnisgraphen mit $6 \leq \mid E \mid$ des Frequent Pattern Laufes angegeben. Dort ist ein deutlicher Anstieg der mittleren Laufzeit bei den Werten 9 und 10 auf ca. $4ms$ erkennbar. Die mittlere Laufzeit liegt hier bei ca. $2ms$ für Gleichheit, für Ungleichheit bei $0.7464ms$. Auch bei den größeren Datensätzen liegt die Laufzeit im *ms*-Bereich, wobei größere Graphen länger brauchen, als kleinere.
Zu diesen Ergebnissen muss gesagt werden, dass die Laufzeit hauptsächlich durch die Berechnung von g bestimmt wird. Diese benötigt im schlimmsten Fall Zeit $O(\mid V \mid)$ zur Konvergenz, was auch den Sprung der Laufzeit zwischen Kantengröße 8 und 9 erklären kann. Da es sich bei der Implementierung um eine Java-Implementierung handelt, wurden die Laufzeiten noch nicht mit hochoptimierten C-Programmen, wie `nauty`, verglichen. Dies würde eine C++/C-Implementierung erfordern, um eine annähernd praktische Vergleichbarkeit der Algorithmen zu gewährleisten.

4.1.4 Laufzeit Subgraphisomorphie-Algorithmus

Zuletzt soll noch die Laufzeit des Subgraphisomorphie-Algorithmus untersucht werden. Dabei muss beachtet werden, dass bei einer vollständigen Aufzählung aller Vorkommen die Größe der Automorphismengruppe des Musters die Laufzeit mitbestimmt. Hier bedeutet das: Hat der Algorithmus das erste Vorkommen gefunden, findet er bei den darauf folgenden Backtracking-Schritten alle Automorphismen des Musters, was schneller geht, da es sich dann um „sichere Treffer“ handelt. Aus diesem Grund wurde hier nur die Zeit bis zum Auffinden des ersten Vorkommens gemessen und in Tabelle 4.3 angegeben. Auch hier wurden wieder die Ergebnisse des Frequent Pattern-Algorithmus als Suchmuster verwendet.

Kanten/Kategorie	Mittlere Laufzeit [ms]
6	1.0476
7	0.7500
8	1.2955
9	4.5000
10	4.3333
Gleichheit	1.1951
Ungleichheit	0.7464
Planar (klein)	23
Planar (mittel)	81
Planar (gross)	90
Random (klein)	22
Random (mittel)	65
Random (gross)	133
Random (dünn)	226
Ungleichheit	93.9286

Tabelle 4.2: Diese Tabelle führt im oberen Teil die Laufzeiten des Graphisomorphiealgorithmus bei Gleichheit in Abhängigkeit der Kantenzahl auf. Dabei wurden die vom Frequent Pattern Algorithmus gefundenen Muster untereinander verglichen. Nach den Eintrag „Gleichheit“ und „Ungleichheit“, welche die mittleren Gesamtlaufzeiten für diese Fälle angeben, werden die Laufzeiten für die Gleichheitstests der Eingabegraphen untereinander, sowie der Mittelwert bei Ungleichheit angegeben.

Tabelle 4.3 zeigt im oberen Abschnitt die mittlere Laufzeit bis zum Finden eines Musters abhängig von der Musterkantengröße. Im unteren Abschnitt wird die mittlere Laufzeit bei Enthaltensein und Nicht-Enthaltensein angegeben. Alle Werte befinden sich im Bereich zwischen $0.8ms$ und $1.5ms$, was relativ schnell anmutet. Allerdings muss dabei beachtet werden, dass die Eingabe aus Teilgraphen bestand, von denen bekannt ist, dass sie im Host *häufig* vorkommen. Daher ist klar, dass diese Laufzeitangaben den optimalen Fall repräsentieren und die realen Laufzeiten im allgemeinen Fall wahrscheinlich schlechter sind.

Muster-Kanten/Kategorie	Mittlere Laufzeit [ms]
3	0.8158
4	0.9090
5	1.0324
6	1.1383
7	1.2725
8	1.3937
9	1.4590
Enthalten	1.0786
Nicht-Enthalten	0.8409

Tabelle 4.3: Diese Tabelle zeigt im oberen Abschnitt die mittlere Laufzeit bis zum Finden des ersten Vorkommens in Abhängigkeit der Mustergröße. Im unteren Abschnitt ist die mittlere Gesamtlaufzeit bei Enthaltensein, bzw. Nicht-Enthaltensein angegeben.

4.2 Analyseergebnisse

4.2.1 Gefundenen Muster vs. Iterationsrunden

Bevor es an die eigentliche Auswertung geht, soll zunächst die Musterausbeute des Frequent Pattern Algorithmus betrachtet werden. Aus Platzgründen beschränkt sich dieser Abschnitt auf die Ausbeute in dem Protein-Interaktions- und dem BN++-Datensatz für *A.thaliana*, *D.melanogaster*, *S.cerevisiae* und *E.coli* in Tabelle 4.4 und 4.5. An diesen beiden Tabellen wird auch die Auswirkung unterschiedlicher Klassifikatoren deutlich.
Der Protein-Interaktions-Datensatz 4.4 wurde mit dem in Definition 3.3 erwähnten Klassifikator untersucht, bei dem es sich um einen rein topologischen Klassifikator handelt. Für ihn spielt die unterschiedliche Beschriftung der Knoten keine Rolle, was dazu führt, dass in allen Organismen für jede Kombination aus Mustergröße und Rundenzahl ähnlich viele Muster gefunden werden. Das hängt damit zusammen, dass der Wahrscheinlichkeitsraum, aus dem die Stichproben gezogen werden, aufgrund des Klassifikators noch relativ klein ist und so durch den Runden-Parameter noch sicher erschlossen werden kann. Bei Verwendung eines Klassifikators, der zusätzlich noch die Beschriftung der Knoten beachtet, wie beim BN++-Datensatz in Tabelle 4.5 vergrößert sich der zu betrachtende Raum und es wird deutlich, dass die vier Organismen mit ihren Netzwerken die zur Verfügung stehende Menge möglicher Topologien (mit Beschriftung) in sehr unterschiedlichem Maße ausnutzen. Hierbei fällt auf, dass *E.coli* in scheinbar jeder Größenordnung die wenigsten Topologien verwendet. *D.melanogaster* hingegen übertrifft alle anderen Organismen ausnahmslos an topologischer Komplexität.
Zuletzt zeigt sich noch, dass die nötige Rundenzahl mit der alle verwendeten Topologien detektiert werden können, wie zu erwarten war, eindeutig von der Mustergröße abhängt. So sind in Tabelle 4.4 bei Mustergröße 8 mindestens 200 Runden nötig, bis man sicher sein kann, dass alle Topologien entdeckt wurden. Allerdings ist auch diese Mindestrundenzahl sicher von dem verwendeten Klassifikator abhängig.
Für diese Ergebnisse lässt sich also abschließend sagen, dass sowohl der eingesetzte Klassifikator, die Mustergröße, wie auch die Rundenzahl entscheidend für das Ergebnis sind.

Size	Rounds	A.thaliana	D.melanogaster	S.cerevisiae	E.coli
3	10	14	13	14	15
3	20	13	15	15	15
3	50	14	15	15	15
3	100	15	15	15	15
3	150	15	15	15	15
3	200	15	15	15	15
3	250	15	15	15	15
4	10	17	21	21	20
4	20	23	22	23	24
4	50	22	22	25	25
4	100	25	23	25	25
4	150	25	24	25	25
4	200	25	24	25	25
4	250	25	24	25	25
5	10	43	40	48	53
5	20	47	50	56	55
5	50	54	56	55	58
5	100	56	57	59	58
5	150	57	58	58	58

Size	Rounds	A.thaliana	D.melanogaster	S.cerevisiae	E.coli
5	200	58	59	58	60
5	250	59	57	59	58
6	10	89	91	108	120
6	20	112	108	122	127
6	50	128	129	137	133
6	100	132	140	144	143
6	150	145	142	145	147
6	200	142	141	148	145
6	250	141	146	147	147
7	10	162	201	248	253
7	20	246	257	288	306
7	50	315	306	351	349
7	100	352	355	364	371
7	150	372	363	386	377
7	200	365	375	384	385
7	250	375	373	380	384
8	10	328	464	596	587
8	20	568	608	795	757
8	50	770	868	982	983
8	100	897	973	1007	1031
8	150	948	998	1048	1053
8	200	1023	1031	1070	1081
8	250	1012	1041	1073	1065
9	10	842	910	1511	1305
9	20	1295	1343	2114	2000
9	50	1996	2364	2651	2536
9	100	2609	2819	2936	2902
9	150	2686	2999	3064	3095
9	200	2786			3106

Tabelle 4.4: Diese Tabelle listet abhängig von der gesuchten Mustergröße und Rundenzahl die Anzahl der gefundenen Topologien den Protein-Interaktions-Datensätzen auf.

Size	Rounds	A.thaliana	D.melanogaster	S.cerevisiae	E.coli
3	10	43	79	65	25
3	20	50	123	90	29
3	50	56	187	138	39
3	100	134	192	171	41
3	150	197	291	207	47
3	200	188	309	185	55
3	250	199	348	213	49
4	10	75	227	159	55
4	20	95	318	240	73
4	50	131	526	386	81

Size	Rounds	A.thaliana	D.melanogaster	S.cerevisiae	E.coli
4	100	220	792	436	93
4	150	366	859	528	97
4	200	459	892	532	131
4	250	540	998	606	123
5	10	125	369	252	97
5	20	179	737	430	140
5	50	238	1359	830	161
5	100	344	2130	1140	174
5	150	734	2493	1252	191
5	200	890	2834	1418	212
5	250	1054	2776	1478	213
6	10	269	997	542	220
6	20	364	1639	777	294
6	50	655	3498	1973	375
6	100	824	5350	2504	404
6	150	1407	6616	2798	472
6	200	1739	7490	3195	464
6	250	2179	8113	3417	501
7	10	522	2190	895	357
7	20	796	3561	1875	539
7	50	1319	8419	3340	813
7	100	1766	13928	5451	991
7	150	2908	16630	6164	1039
7	200	3484	19301	7256	1058
7	250	3285			

Tabelle 4.5: Diese Tabelle listet abhängig von der gesuchten Mustergröße und Rundenzahl die Anzahl der gefundenen Topologien in den BN++-Datensätzen auf.

4.2.2 Die wahrscheinlichsten Muster

Dieses Kapitel stellt für jeden Datensatz die Topologien mit dem größten h_i-Wert vor. Dabei wird keine Unterscheidung zwischen den Runden gemacht, sondern die beste relative Häufigkeit angegeben, mit der dieses Muster in irgendeinem Programmlauf vorkam.
Die Netzwerke für Protein-Interaktion und BN++-Modell wurden für die vier Organismen, *E.coli*, *S.cereivisiae*, *D.melanogaster* und *A.thaliana*, jeweils für die Mustergrößen drei bis acht, sowie dem Runden-Abbruchkriterien $10, 20, 50, 100, 150, 200, 250$ durch den Frequent Pattern-Algorithmus aus Kapitel 3.3.4 jeweils fünf mal getestet.
Im BN++-Modell enthielten die Netzwerke Markierungen für die Klassen `Protein`, `ProteinComplex` und `Reaction`, die von einem erweiterten Klassifikator, der diese Markierungen mit der Klassifikation aus Definition 3.3 kombinierte, berücksichtigt wurden. In diesen Ergebnissen sind Proteine als abgerundete, blaue Rechtecke, Protein-Komplexe durch Diamant-förmige, grüne Symbole und Reaktionen durch rote Punkte dargestellt.
Der Protein-Interaktionsdatensatz wurde mit Klassifikator 3.3 ohne Berücksichtigung der Markierungen untersucht. Allerdings werden in den Ergebnisdarstellungen Protein-Komplexe durch ihre rechteckige Form von den rund dargestellten Proteinen unterschieden, auch wenn der Analyse-Algorithmus dies

nicht berücksichtigte.
Zuletzt wurde der, von Autoregulationskanten bereinigte, *E.coli*-Datensatz von Uri Alon ebenfalls mit dem einfachen Klassifikator 3.3 analysiert. In diesen Ergebnissen finden sich zusätzlich die Namen einer Beispielinstanz, welche diese Musterklasse repräsentiert.
Unter jedem Muster findet sich die Angabe des h_i-Wertes (`hi`) und der dazugehörigen Rundenzahl (`r`), bei der dieser Wert erreicht wurde. Aufgrund der Größe der Ergebnismenge, wurde zur Auswahl der Muster folgendermaßen vorgegangen: Zunächst wurden die h_i-Werte für alle Topologien eines Datensatzes absteigend sortiert. Die besten 200 Topologien wurden ausgewählt und untereinander auf Gleichheit getestet, bei Gleichheit wurde eine der beiden Instanzen entfernt. Die angegebenen Mustermengen zeigen also die Zusammensetzung der besten 200 Topologien, unabhängig von Mustergröße und Rundenkriterium. Wenn ein Muster also eine relative Häufigkeit von 0.05 bei einer Rundenzahl von 50 hat, bedeutet das, dass es bei allen anderen Runden mit einem geringeren `hi`-Wert oder gar nicht gefunden wurde.

Betrachtet man nun den BN++-Datensatz, so fällt relativ schnell auf, dass die Topologien von

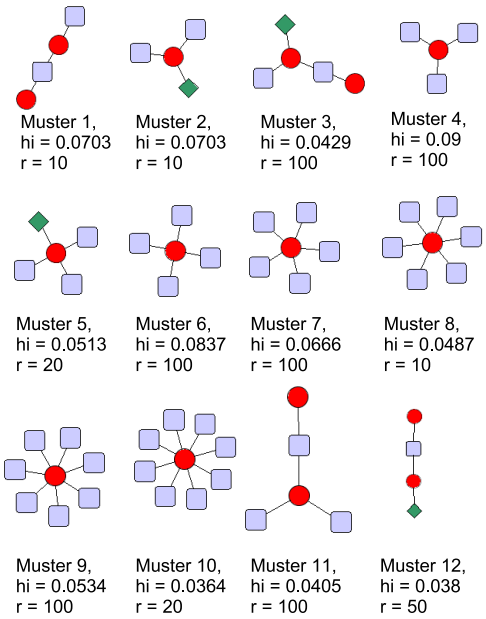


Abbildung 4.3: Ergebnisse *A.thaliana* (BN++)

D.melanogaster und *A.thaliana* fast identisch sind. Bis auf Muster elf in 4.3 sind alle Muster dort auch in 4.4 enthalten. Es handelt sich dabei hauptsächlich um Stern-Topologien, bzw. kurze Pfade. Daran wird deutlich, dass höhere Organismen am häufigsten Reaktionen mit vielen Teilnehmern verwenden.
Die Erklärung dafür steht in „An Introduction into Systems Biology“ [2], Kapitel 6: Diese Topologien haben bezüglich der Informationsverarbeitung die Eigenschaften einer *Entscheidungsfunktion*. Durch die vielen Reaktionspartner werden die komplexen Bedingungen festgelegt, die für eine Antwort nötig sind. Es ist also nicht verwunderlich, hauptsächlich sternförmige Topologien bei den höheren Organismen zu finden.

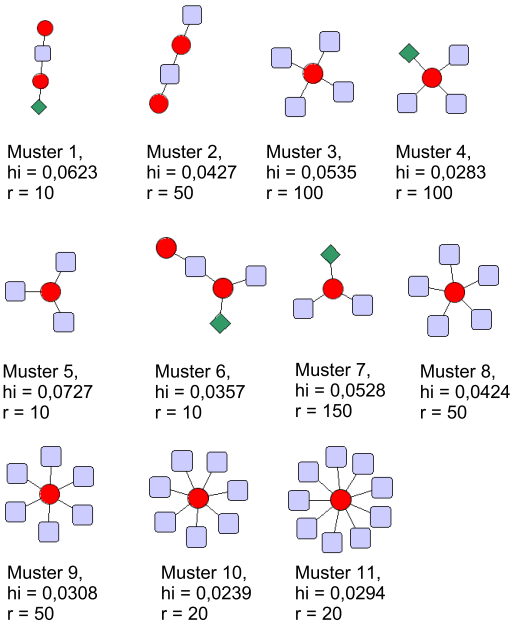


Abbildung 4.4: Ergebnisse *D.melanogaster* (BN++)

Betrachtet man nun *S.cerevisiae* und *E.coli*, so fällt auf, dass in zunehmendem Maße auch zyklische Topologien vorkommen. Diese werden deshalb häufiger gefunden, da weniger Stern-Topologien vorkommen. Allerdings besitzt *S.cerevisiae* trotzdem einen etwas größeren Satz an Stern-Topologien als *E.coli*, was sicher mit der höheren Komplexität der Eukaryoten gegenüber den Prokaryoten zusammenhängt.
Nun soll die Analyse der Protein-Interaktionsnetzwerke betrachtet werden. Diese sind im Anhang A.5 zu finden. In diesen existiert zwischen zwei Knoten genau dann eine Kante, wenn diese miteinander interagieren. Auf den ersten Blick fällt auf, dass nun bei *A.thaliana* und *D.melanogaster* hauptsächlich zykelfreie Strukturen mit längeren Pfaden auftreten. Während bei *S.cerevisiae* und *E.coli* zunehmend Strukturen mit Zykel zu finden sind.
Wie ist das zu erklären, wo doch beiden Netzwerken derselbe Datensatz zugrunde liegt? Der Unterschied liegt im verwendeten Datenmodell. Beim BN++-Model handelt es sich um einen *Hypergraphen*, bei dem die `Events` die `Participants` stern-förmig konzentrieren. Da ein `Event` eine Interaktion darstellt, muss bei der Umrechnung des Datensatzes zwischen allen `Participants` eines `Events` eine Kante erstellt werden, um diese Beziehung darzustellen. Damit ist das BN++-Modell von vornherein „sternlastiger“, als das Protein-Interaktionsmodell, das dafür *Cliquen*-lastiger wird.[1] Diese Umrechnung erhöht auch den durchschnittlichen Grad eines Knotens: Hatte im BN++-Model ein Knoten nur eine Kante zu einem `Event`, so erhalten er und die anderen Teilnehmer des `Events` im Protein-Interaktionsmodell so viele Kanten, wie es Teilnehmer des `Events` gibt.
Da *A.thaliana* und *D.melanogaster* offensichtlich viele `Events` mit vielen Teilnehmern haben, ist das umgerechnete Protein-Interaktionsnetzwerk ein sehr dichtes Netzwerk. Die Netzwerke von *S.cerevisiae*

[1] Bei einer *Clique* handelt es sich um einen Teilgraph, in dem jeder Knoten eine Kante zu jedem anderen besitzt.

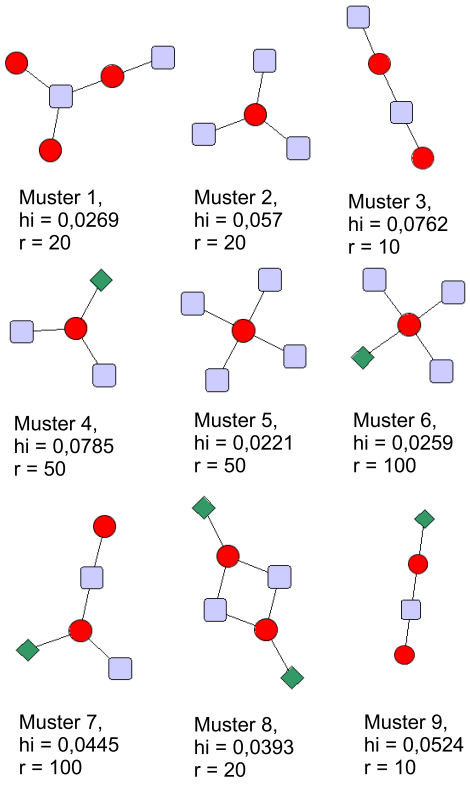


Abbildung 4.5: Ergebnisse *S.cerevisiae* (BN++)

und *E.coli* sind demnach weniger dicht als die anderen beiden. Die abgebildeten Topologien lassen demnach zunächst die Vermutung aufkommen, dass es in dichten Netzwerken wahrscheinlicher ist, einen Pfad zu finden, als einen Zykel. Nimmt man diese Vermutung an, so können die Pfade als Bestätigung der Aussage, dass *A.thaliana* und *D.melanogaster* eine höhere Regulationsdichte haben als *S.cerevisiae* und *E.coli*, angesehen werden.
Allerdings muss auch beachtet werden, dass der Algorithmus, wie schon besprochen, eine *kantenabhängige Teilgraphdefinition* verwendet. Eine *knotenabhängige Teilgraphdefinition* würde hingegen in einem solchen dichten Netzwerk wahrscheinlich mehr Zykel finden.
Damit wird letzten Endes nur klar, dass bei der Suche nach häufigen Teilgraphen in biologischen Netzwerken unbedingt die Modellierung der Daten mit der verwendeten Graphdefinition sinnvoll abgeglichen werden muss, will man eine biologisch relevante Aussage machen. Allerdings ist bei Verwendung der vorgestellten randomisierten Verfahren nicht ausgeschlossen, dass etwas Interessantes gefunden werden kann.[2]
Zuletzt bleibt noch die Diskussion der Ergebnisse des *E.coli*-Datensatzes von Uri Alon. Auch diese finden sich im Anhang A.5. Diese sind in zwei Abbildungen aufgeteilt worden. In Abbildung A.7 finden sich verschiedene Topologien, bei denen es sich scheinbar hauptsächlich um *Dense overlapping regions* (DORs) handelt. Auch hier werden häufig Stern-Topologien gefunden, was mit der oben angesprochenen Eigenschaft der *Entscheidungsfunktion* zusammenhängen könnte.
In Abbildung A.8 ist der zweite Teil der gefundenen Topologien zusammengefasst. Diese enthalten alle

[2]Der Autor ist z.B. fasziniert von Muster 8 in 4.6, das auch als Muster 9 in 4.7 vorkommt.

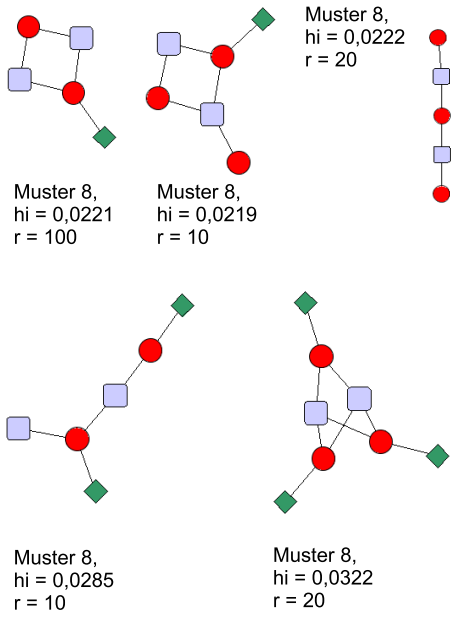


Abbildung 4.6: Ergebnisse *S.cerevisiae* (BN++)

crp, bei dem es sich um einen scheinbar hoch-regulierten Knoten handelt. Allerdings verdeutlicht dieses häufige Auffinden auch den Einfluss von *Hubs* auf den Frequent Pattern-Algorithmus.
Abschließend lässt sich sagen, dass mit diesem Algorithmus zwar interessante Muster gefunden werden können, deren biologische Interpretation nur durch Experimente, nicht aber durch die `hi`-Werte, über die diese Auswahl getroffen wurde, möglich ist. Wahrscheinlich enthalten die nicht dargestellten seltneren Motife weitere interessante Muster, die ebenfalls praktisch untersucht werden sollen. Schließlich wird eine Zerlegung des gesamten Netzwerkes in Komponenten berechnet und die häufigsten Topologien machen nur einen kleinen Teil des Gesamten aus, wenn man die `hi`-Werte betrachtet.

4.3 Integration in BiNA

In diesem Kapitel sollen kurz die vier Bedienelemente des *MotifPlugin* für BiNA vorgestellt werden. Diese sind in den Abbildungen 4.8 und 4.9 zu finden. Das erste Tab mit der Überschrift „Compare Graphs“ bietet die Möglichkeit, zwei Teilgraphen des im Hauptfenster geladenen Netzwerks miteinander zu vergleichen. Dazu werden mittels Selektion im Hauptfenster und dem jeweiligen *Load*-Button des Tabs die Teilgraphen in das jeweilige Vorschaufenster geladen. Wird nun der *Compare*-Button gedrückt, wird ein Vergleich der beiden Netzwerke durchgeführt und es erscheint in der rechten unteren Ecke eine Meldung. Bei diesem, wie auch bei den anderen Verfahren, berücksichtigt die Klassifikation der Knoten zusätzlich die BN++-Klassenhierarchie. Das bedeutet, dass z.B. ein BN++-`Gene` nicht auf ein BN++-`Protein` abgebildet werden darf.

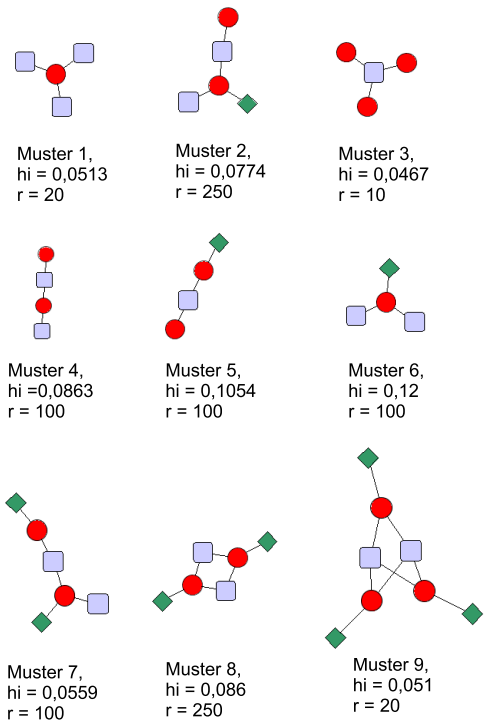
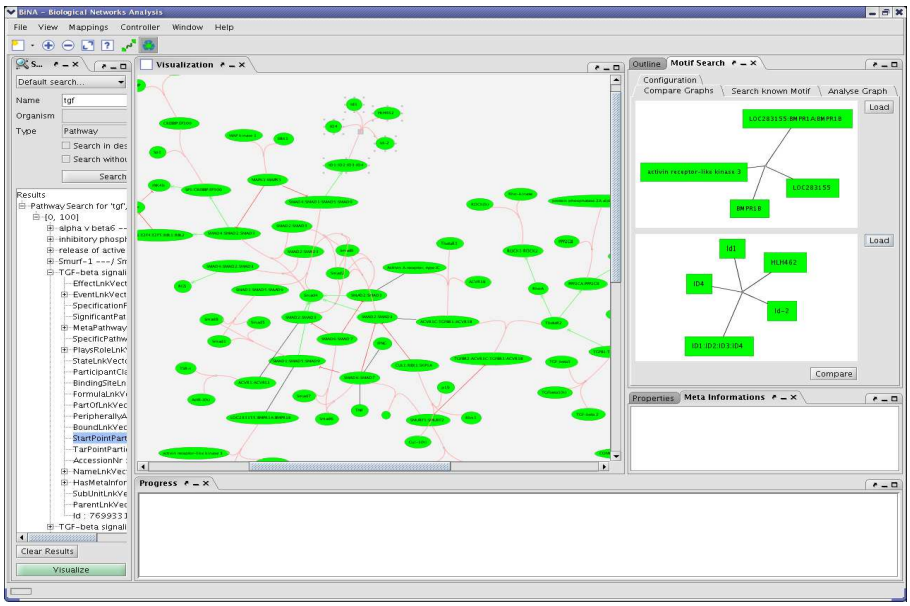


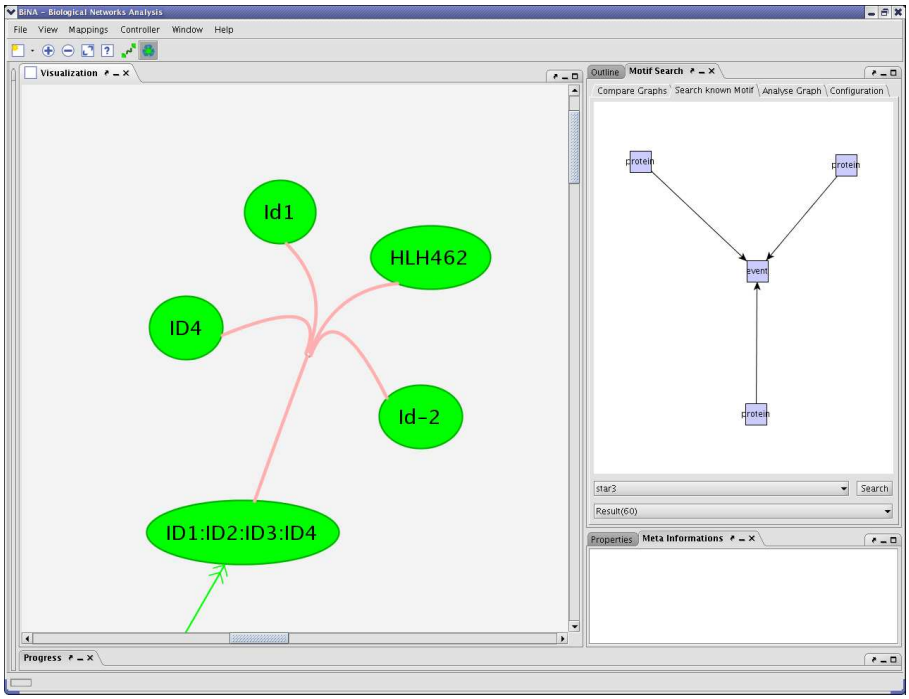
Abbildung 4.7: Ergebnisse *E.coli* (BN++)

Das zweite Tab mit der Überschrift „Search Known Motifs“ bietet eine Auswahlliste mit bekannten Mustern, die in dem aktuell geladenen Netzwerk gesucht werden können. Die angezeigten Vorlagen befinden sich als GML-Dateien in einem Plugin-Unterverzeichnis. Als Label können die Knoten dabei BN++-Klassennamen tragen, um das Muster genauer zu definieren. Auch bei dieser Suche wird die BN++-Klassenhierarchie beachtet. Bei der Auswahl eines Musters aus der Pull-Down-Liste, wird dieses in dem Vorschaufenster angezeigt. Nun kann der *Search*-Button gedrückt werden, um die Suche zu starten. Aus Zeit- und Speichergründen wurde die Anzahl der zu suchenden Muster beschränkt. Die Suche endet, nachdem die eingestellte Anzahl an Vorkommen gefunden wurde. Diese Anzahl kann im Konfigurations-Tab geändert werden. Nach Beendigung der Suche ist die Ergebnisliste mit neuen Einträgen gefüllt. Durch Anwählen eines entsprechenden Eintrags, zoomt das Hauptfenster an den Ort des Vorkommens. Dieser Graph ist dann zur weiteren Bearbeitung selektiert.
Das dritte Tab mit der Überschrift „Analyse Graph“, ermöglicht das Finden häufiger Subgraphen im aktuell geladenen Netzwerk. Dazu muss nur der *Search*-Button gedrückt werden. Die gefundenen Muster werden daraufhin in die Ergebnisliste eingefügt und können durch entsprechende Auswahl im Vorschaufenster angezeigt werden. Zusätzlich zur Topologie des Musters werden die berechneten relativen Häufigkeiten im Textfeld unter der Vorschau angegeben. Mit Hilfe des *Save*-Buttons kann das angezeigte Muster in einer GML-Datei gespeichert werden. Dabei werden die BN++-Klassen als Labels verwendet, so dass diese, wenn sie im Musterverzeichnis gespeichert werden, nach einem Plugin-Neustart sofort zur Suche verwendet werden können.
Das letzte Tab „Configuration“ dient zur Konfiguration der drei Algorithmen. Hier wird die maximale Ergebniszahl eingestellt und die Anzahl der Runden, die zum Abbruch der Suche nach häufigen Sub-

graphen führt, eingestellt. Mit Hilfe zweier Checkboxen kann man festlegen, ob die Graphen gerichtet oder ungerichtet interpretiert werden sollen und ob nur ein schneller Isomorphietest, also lediglich ein Histogrammvergleich, durchgeführt werden soll.

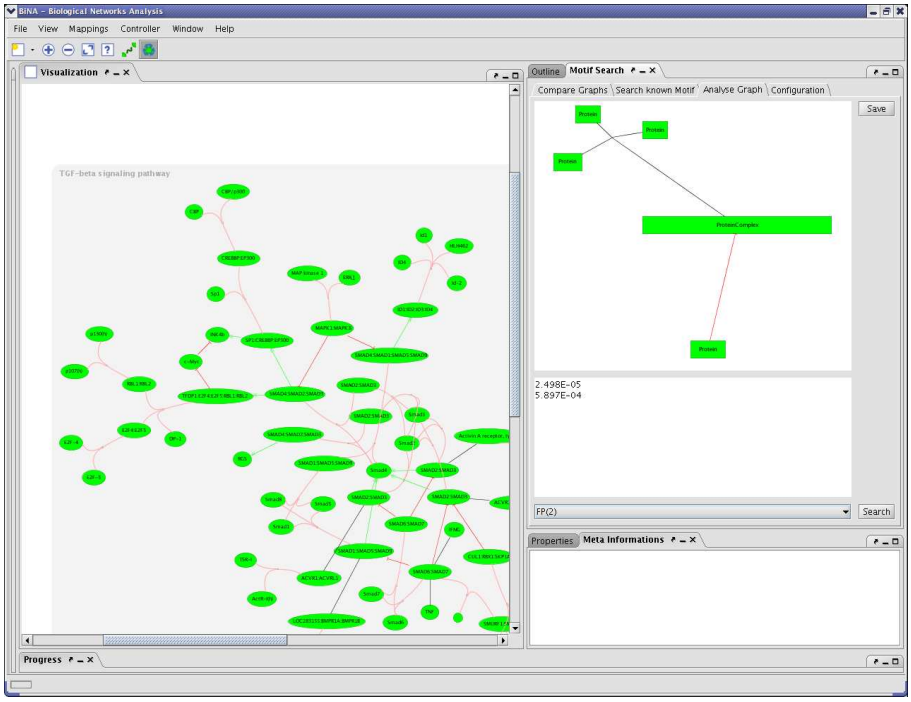


Graphenvergleich

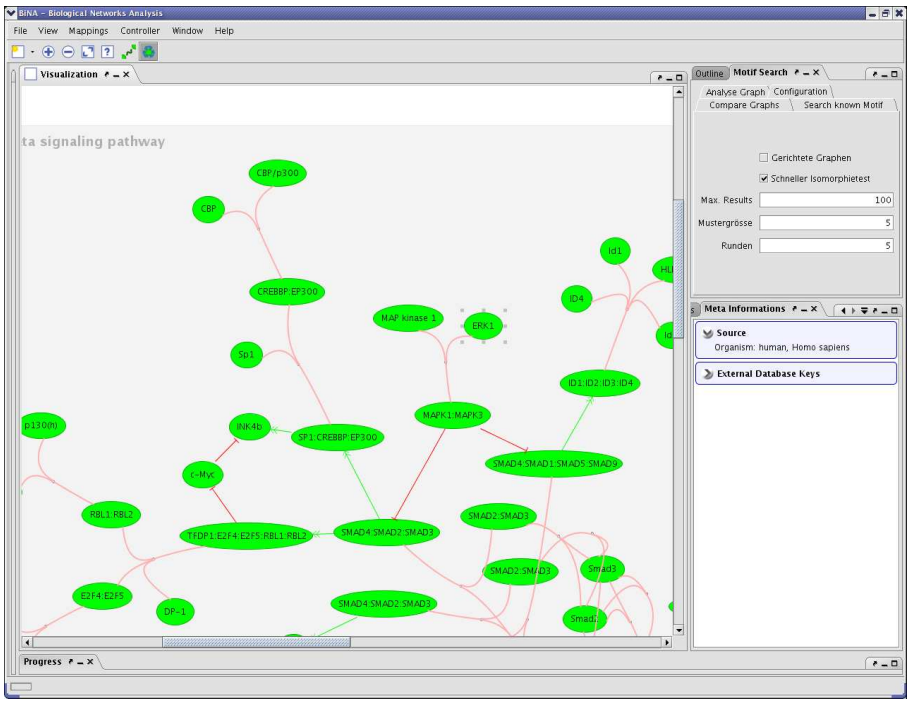


Mustersuche

Abbildung 4.8: Bedienelemente des *MotifPlugin* I



Suche häufiger Graphen



Konfiguration

Abbildung 4.9: Bedienelemente des *MotifPlugin* II

Kapitel 5

Diskussion

In dieser Arbeit wurden drei Algorithmen entwickelt. Mit diesen lassen sich Graphen auf Gleichheit testen, in einem anderen Graphen suchen und ein gegebener Graph analysieren. Diese Algorithmen wurden dazu verwendet, um biologische Netzwerke zu analysieren. Diese wurde in zwei verschiedenen Modellen betrachtet: Dem BN++-Modell und einem Modell, das nur die Protein-Interaktion berücksichtigt. Zusätzlich wurde ein fremder Datensatz, der *E.coli*-Datensatz von Uri Alon untersucht. Dabei wurden von einem Rechner-Cluster ca. 200GB Daten berechnet, deren vollständige Auswertung über die bereits vorgestellten Ergebnisse hinaus eine noch ausstehende Aufgabe für den Autor dieser Arbeit ist.
Auf der biologischen Seite konnten weder neue Erkenntnisse gewonnen werden, noch die von Uri Alon publizierten Ergebnisse bestätigt werden, da die verwendeten Teilgraphdefinitionen voneinander abwichen. Welche Teilgraphdefinition richtig oder falsch ist, bleibt ein offener Punkt für zukünftige Untersuchungen. Wichtig ist dabei, dass sich deren Auswirkungen offensichtlich im Ergebnis niederschlagen.
Auf der theoretischen Seite wurde ein neuer, schneller Algorithmus zur Graphisomorphie vorgestellt, dessen vollständige Untersuchung ebenfalls für den Autor noch aussteht. Hier wurden Beweise und Beweisideen angegeben und z.T. formal ausgearbeitet. Auch wurde herausgefunden, dass die Funktionsweise des Algorithmus in der Lage ist, symmetrische Komponenten in Graphen aufzufinden. Der modulare Aufbau des Algorithmus lässt darüber hinaus noch viel Spielraum für Erweiterungen, so wurde zwar exemplarisch ein Klassifikator (3.3) erwähnt, aber es sind noch viele weitere möglich. So sollte z.B. der Zusammenhang zwischen Laufzeit, Korrektheit des Ergebnisses und einem Klassifikator, der den Knotengrad verwendet, untersucht werden. Oder die Verwendung anderer Knotenmerkmale, wie z.B. dem Konnektivitätsgrad. Auch hier eröffnet sich eine Forschungsmöglichkeit. Schließlich ermöglicht der Graphisomorphietest selbst die Entwicklung neuer Algorithmen und Datenstrukturen, von denen die *GraphMap* nur ein Beispiel ist.
Auch der vorgestellte Algorithmus zur Subgraphisomorphie scheint schnell zu sein. Allerdings müssen auch hier noch Laufzeituntersuchungen und Tests mit diversen Klassifkatoren durchgeführt werden. Des Weiteren kam in der vorliegenden Diplomarbeit die Idee der Kantenklassifikation auf, die ebenfalls neue Perspektiven eröffnet.
Gleiches gilt für den Frequent Pattern-Algorithmus. Hier ist zu prüfen, wie sich die beiden möglichen Teilgraphdefinitionen unterscheiden und auswirken. Auch eine mathematische Bearbeitung des Urnen-Modells aus Abbildung 3.10 steht noch aus. In diesem Kontext sollte der Zusammenhang mit dem Runden-Parameter untersucht werden.
Auf der praktischen Seite verbleibt, neben der Verbesserung der Implementierungen, die Erweiterung des BiNA-Plugins zur Motif-Suche. So sollte es einfacher möglich sein, neue Muster zu laden, bzw. einzugeben. Derzeit ist das nur mittels einer entsprechenden GML-Datei möglich.

Danksagung

Prof. Michael Kaufmann, Prof. Oliver Kohlbacher, Prof. Rolf Reuter, Andreas Gerasch, Torsten Blum, Nina Lehmann, Martin Siebenhaller, Markus Geyer, Nadav Kashtan, Uri Alon danke ich für ihren Beitrag zu dieser Arbeit.
Außerdem geht Dank an die Programmierer der *yfiles*, durch deren Bibliothek diese Arbeit erst möglich wurde.
Ganz besonders danke ich meinen Korrekturlesern: Ingrid u. Helmut Henneges, Markus Bloos, Thomas Karschney und Kathrin Doppler.

Anhang A

Zusatzinformationen

A.1 Routing und das Internet

Dieser Abschnitt soll eine kurze Einführung in das Internet und Routing, soweit es die in Kapitel 3.1 verwendete Analogie betrifft, sein.
Das Internet ist ein weltumspannendes Computernetzwerk mit der Aufgabe der Datenübertragung. Eines der obersten Designziele war der Umgang mit einer sich ändernden Netzstruktur. Neben dem Umgang mit sich öffnenden, ändernden und schließenden Kommunikationswegen zwischen den Computern, musste auch eine Methode entwickelt werden, mit der Nachrichten an Rechner geliefert werden konnten, zu denen der Weg noch nicht bekannt war. Die Strategie des Internets ist, dass jeder teilnehmende Computer Nachrichten weiterleiten muss. Heute handelt es sich dabei fast ausschließlich um so genannte Router, die auf die Aufgabe der Nachrichtenvermittlung spezialisiert sind. Da das Netzwerk dynamisch ist, können jederzeit Computer erreichbar, bzw. unerreichbar werden. Für die Vermittlung von Nachrichten bedeutet das, dass die Teilnehmer am einen Ende des Netzwerks keine sichere Information über die Struktur des Netzwerkes am anderen Ende haben, also nicht wissen können, über welchen Weg sie eine Nachricht senden sollen. Doch genau das ist aus Effizienzgründen nötig, wenn das Netz nicht mit der zu sendenden Nachricht geflutet werden soll. Aus diesem Grund tauschen die Router von Zeit zu Zeit oder durch Ereignisse ausgelöst Informationen in Form so genannter *Routing-Tabellen* über die ihnen bekannten Teilnehmer aus, so dass diese Informationen nach einer gewissen Zeit auch in entlegenen Bereichen des Netzwerks verfügbar sind. In diesen Routing-Tabellen ist der kürzeste Pfad zu allen bekannten Rechnern vermerkt, so dass auch diese Rechner entscheiden können, über welchen Weg sie einen gesuchten Computer am besten kontaktieren.

A.2 Graphisomorphie

A.2.1 Überblick der Arbeiten zur Graphisomorphie

Das Problem der Graphisomorphie beschäftigt die Informatik schon seit den frühesten Anfängen und wird in jeder Publikation als fundamentales Problem der Graphentheorie, wie auch der Komplexitätstheorie aufgeführt (Handbook of Graph Theory [28]) . Bisher wurde erwartet, dass der Isomorphietest in $O(\exp(\log^2 n))$ oder vielleicht in $O(\exp(\log n \log\log n))$ gelöst werden kann, während der schnellste bekannte Algorithmus eine Zeit von $O(\exp(n^{1/2+O(1)}))$ benötigt. Diese Algorithmen verwenden im Prinzip auch die Techniken des in 3.1 vorgestellten Algorithmus, wie Backtracking, die Verfeinerungstechnik (*refinement-technique*) oder die Zuordnung von Attributen, so genannten *kanonischen Markierungen (canonical labels)*. Auch die schrittweise Erweiterung einer möglichen Isomorphieabbildung wird benutzt. Allerdings führten diese Ideen sehr oft knapp an einer allgemeinen Lösung vorbei, da übersehen wurde,

dass die Kanten das Problem darstellen, wie es z.B. auch bei der Entwicklung des Planaritätstests für Graphen (Boyer,Myrvold [27]) der Fall war. Für spezielle Graphen, wie Bäume (mit beschränkter Breite), planare Graphen, Intervallgraphen, zirkuläre Graphen, Graphen mit beschränktem Genus oder Grad konnten dennoch effiziente Algorithmen erarbeitet werden. Zum *State of the art* gehört auch `nauty` [38], dass zusätzlich gruppentheoretische Ansätze verwendet. Denn im engen Zusammenhang mit der Isomorphie steht auch die Frage nach den Automorphismen, also den Isomorphieabbildungen eines Graphen, die diesen auf sich selbst abbilden. Diese bilden eine Gruppe und stellen so einen interessanten Zusammenhang zu diesem Gebiet der Mathematik her. Ein anderer hochinteressanter Zusammenhang ist der, dass das Graphisomorphieproblem als gleichwertig zum so genannten *Termgleichheitstest (term equality problem)* angesehen wird, bei dem es darum geht, zu entscheiden, ob zwei Terme mit kommutativen und/oder assoziativen Funktionen und kommutativen Variablen gleich sind. Wahrscheinlich stellt dieses Problem den Bezug zu der in Kapitel 3.1.3 angegebenen Rekursion her.
Der wohl einfachste Isomorphiealgorithmus wird in „Algorithms on Trees and Graphs“ [49] beschrieben. Dabei wird eine anfangs leere Isomorphieabbildung nach und nach in einem Backtracking-Schema erweitert. Dieses Verfahren benötigt im schlimmsten Fall $O((n+1)!)$ Zeit. In dem genannten Buch von Gabriel Valiente findet sich auch ein effizienter Algorithmus zur Baumisomorphie, der Ansätze aus der Rekursion in 3.1.3 zeigt. Hier erhalten Baumknoten Beschriftungen mit den lexikographisch geordneten Tiefen der angeschlossenen Teilbäume.
In „An Efficient Algorithm for Graph Isomorphism“ [10] wird eine Verfeinerungsprozedur angegeben, die insofern nahe an 3.1 herankommt, als dass zu Beginn des Verfahrens die Knoten mittels Indices nach Grad klassifiziert werden. Diese Klassifikationen wird zu Attributlisten verarbeitet, die sich aus den Attributen jeweiligen Nachbarn zusammensetzen. Durch Sortieren dieser Listen entsteht eine neue, verfeinerte Klasseneinteilung, so dass neue Indices vergeben und von vorne begonnen werden kann. Allerdings verfehlt dieses Verfahren es, die in 3.1.7 besprochenen symmetrischen Knoten zu berechnen. Dies kann damit erklärt werden, dass es sich um einen lokalen Prozess handelt, bei dem Knoten nur im besten Fall „globales Wissen“ erwerben. Aus dieser Verfeinerung werden nun verschiedene Graphen berechnet, welche den Ursprungsgraphen repräsentieren und leichter zu vergleichen sind.
Eine andere interessante Arbeit „A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices“ [13] schlägt ebenfalls ein Verfeinerungsverfahren vor, das für jeden Knoten weiter entfernte Knoten zur Klassifikation berücksichtigt. Hier findet sich auch, dass zwei Graphen nur dann isomorph sein können, wenn deren Klassifikationsvektoren (oder *Histogramme*) gleich sind. Auch dieser Algorithmus besitzt eine Komplexität von $O(N \cdot N!)$ im schlimmsten Fall.
Einer der Algorithmen, der dem Algorithmus in in 3.1 am ähnlichsten war, findet sich in „GIT - A Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorphism“ [48]. Dieses Verfahren partitioniert Knoten nach Graden in *possible node pairing lists (PNPL*, ausgesprochen *“pineapple“)*. Danach erhalten alle Knoten einer *PNPL* eine einzigartige Nummer, die *set number*, zugewiesen, die dann über eine Erweiterungsfunktion mit den Nummern der Nachbarn verrechnet wird, um eine verbesserte Aufteilung zu erhalten. Hier wird die Beobachtung gemacht, dass für dieses Verfahren sehr unterschiedliche Funktionen verwendet werden können, wie z.B. die Summe oder die Multiplikation mit Primzahlen [1]. Das Problem dieses Ansatzes besteht darin, dass lokale Isomorphieinformation mit der Information über die globale topologische Position eines Knotens vermischt wird. So kann am Ende des Verfahrens nicht unterschieden werden, ob zwei Knoten in der gleichen Klasse sind, weil ihre Nachbarschaft lokal isomorph ist, oder ob sie ähnliche globale Positionen oder evtl. sogar keines von beiden besitzen. In 3.1 wird dies über die Aufteilung der Information in die Merkmale und deren Koeffizienten erreicht.

[1] Eine Idee, die der Autor anfangs auch verfolgte.

A.2.2 Pseudocode Graphisomorphie

Knotenklassifikation Um einen Knoten *node* zu klassifizieren gehen wir wie folgt vor: Wir berechnen den induzierten Nachbarschaftsgraphen des Knotens, also den Graphen, der nur aus den Knoten besteht, die mit *node* verbunden sind, sowie den Kanten, die zwischen diesen Knoten existieren. Dabei wird *node* nicht mit aufgenommen.
Danach führen wir eine Tiefensuche auf diesem Graphen aus und zählen die einzelnen Knoten, ohne Kanten. Hat ein Knoten Kanten, startet die Tiefensuche von diesem Knoten aus und aktualisiert jeweils den längsten gefundenen Pfad, wie auch die gefundenen Zykelgrössen.

Algorithm A.2.1.

```
1  proc classify-node(node) ≡
2    Sei G' = (V',E') der induzierte Nachbarschaftsgraph G(node)
3    global var ⟨singleNodes ← 0; longestPath ← 0; cycleSizes ← ∅; ⟩
4              for all n ∈ V'
5                  do if 0 = degree(n)
6                        then comment: Einzelne Knoten hochzählen
7                             singleNodes ← singleNodes + 1;
8                        else comment: Längsten Pfad und Zykelgrössen berechnen
9                             depths[n] ← 0;
10                            call dfs(n,0,depths,⊥); [a]
11                 fi
12             od
13             return (singleNodes,longestPath,cycleSizes);
14   end
15
16   proc dfs(node,depth,depths,precc) ≡
17     comment: precc ist der Knoten des vorherigen Aufrufs
18     nextDepth ← depth + 1;
19     longestPath ← max{longestPath,depth};
20     comment: Vorwärtsschritte
21     for all n ∈ neighbors(node)
22         do if n ≠ precc
23               then if ¬visited(n)
24                       then depths[n] ← nextDepth;
25                            call dfs(n,nextDepth,depths,node);
26                       else
27                            cycleLength ← depth − depths[n] + 1;
28                            cycleSizes ← cycleSizes ∪ {cycleLength};
29                    fi
30         fi
31     od
32   end
```

[a] Das Symbol ⊥ steht für `undefined`

Durchwanderungsreihenfolge Die Funktion *calculate-visiting-order* berechnet eine Kantenreihenfolge für das Muster, so dass diese zu jedem Zeitpunkt eine Zusammenhangskomponente darstellen. In dieser Reihenfolge kommen Kreuzkanten *vor* den Vorwärtskanten, so dass beim Backtracking Falsch-Zuweisungen möglichst früh erkannt werden.

Algorithm A.2.2.

```
1  proc calculate-visiting-order(G, H_G)
2    comment: Sei H_G : M_M ⟶ ℕ das Histogramm von G
3    Sei start ∈ V Knoten mit H_G(start) minimal und degree(start) maximal
4    queue ← (start);
5    mark(start) ← true ;
6    list ← ();
7    while ¬ empty(queue)
8          do
9          node ← removeFirst(queue);
10         append ← ();
11         for e ∈ AdjacentEdges(node)
12             do if ¬visited(e)
13                   then
14                         Sei e = {u,v}
15                         if ¬visited(u) ∧ ¬visited(v)
16                           then comment: Kreuzkante vorne anhängen
17                                append ← (e) ++ append;
18                            else comment: Vorwärtskante hinten anhängen
19                                if ¬visited(u)
20                                  then visited(u) ← true ;
21                                       queue ← queue ++ (u);
22                                fi
23                                if ¬visited(v)
24                                  then visited(v) ← true ;
25                                       queue ← queue ++ (v);
26                                fi
27                                append ← append ++ (e);
28                         fi
29                         visited(e) ← true ;
30             fi
31         od
32         list ← list ++ append;
33   od
34   return (list);
35 end
```

Backtracking Der Backtracking-Algorithmus versucht eine Aufzählung der Isomorphie-Abbildung durchzuführen. Dazu wandert er über den Kantenaufzählungspfad *list* des Musters und zählt alle noch freien Erweiterungs-Möglichkeiten für jede Kante auf. Dabei aktualisiert er jeweils die gerade gehaltene Knoten- und Kante-Abbildung zwischen den beiden Graphen.

Algorithm A.2.3.

```
1  proc backtrack(list, i)
2    if size(list) = i
3      then comment: Rekursionsfuss
4           output (Isomorphieabbildung gefunden !);
5       else comment: Rekursionsschritt
6           Sei e^G = {u^G, v^G} die i.te Kante aus list
7           if iso(e_i^G) = ⊥
8             then
9                  if ¬visited(u^G, v^G)
10                   then comment: Teste alle Kanten mit passenden
11                        comment: Merkmalszahlen m(u^H), m(v^H) ∈ M_M
12                        for all e^H = {u^H, v^H} ∈ E : m(u^G) = m(u^H) ∧ m(v^G) = m(v^H)
13                            do
14                            iso(e^G) ← e;
15                            call backtrack(list, i+1);
16                            iso(e^G) ← ⊥;
17                        od
18                 elsif one-is-visited(u^G, v^G)
19                      then Sei u^G der besuchte und u^H der ihm zugeordnete Knoten
20                           for all e^H ∈ AdjacentEdges(u^H) : m(v^G) = m(v^H)
21                               do
22                               iso(e^G) ← e^H;
23                               call backtrack(list, i+1);
24                               iso(e^G) ← ⊥;
25                           od
26                       else
27                           Seien u^H, v^H die u^G, v^G zugeordneten Knoten
28                           if exists-edge(u^H, v^H)
29                             then
30                                  Sei e^H = {u^H, v^H} diese Kante
31                                  iso(e^G) ← e^H;
32                                  call backtrack(list, i+1);
33                                  iso(e^G) ← ⊥;
34                           fi
35                 fi
36             end
```

Isomorphietest Testet die Graphen G und H auf Gleichheit. Dabei werden erst die beiden Merkmalsrekursionen g_G^n und g_H^n bis zur Konvergenz und daraus die Histogramme $\mathcal{H}_G$ und $\mathcal{H}_H$ berechnet. Sind diese gleich, wird das *backtracking*(*list*,0) gestartet.

Algorithm A.2.4.

```
1  proc graph-isomorphy(G,H)
2    Berechne konvergente Transferfunktionen g_G^n, g_H^n
3    Berechne H_G^n,H_H^n für g_G^n, g_H^n
4    if H_G^n = H_H^n
5      then
6           list ← calculate-visiting-order(G, H_G^n);
7           backtrack(list,0);
8       else
9           output (G und H sind nicht isomorph !)
10   fi
11 end
```

A.3 Subgraphisomorphie

A.3.1 Überblick der Arbeiten zur Subgraphisomorphie

Im Gegensatz zu Graphisomorphie wurde das Problem der Subgraphisomorphie als NP-vollständig nachgewiesen. Das hat zur Folge, dass natürlich alle Strategien, die auch in 3.2 erwähnt werden, in der Literatur zu finden und ausprobiert worden sind. Hauptsächlich wird versucht, den Suchraum einer *exhaustive search* durch weitere Informationen einzuschränken. Häufig wird dabei die Hilfe von vorgegebenen Knoten und Kantenbeschriftungen in Anspruch genommen. Der Suchraum ist dabei meist der komplette Raum der Isomorphieabbildungen, die über eine Permutationsmatrix zwischen den Adjazenzmatrizen abbilden. Dies ist u.a. bei der grundlegenden Arbeit von J.R.Ullmann „An Algorithm for Subgraph Isomorphism“ [29] der Fall. Auch hier wird mittels einer *refinement procedure* versucht, die Abbildungen aus dem Suchbaum herauszunehmen, die auf keinem Fall zu einem Ergebnis führen können. Dieses Vorgehen entspricht ungefähr dem Test mit der ◄-Relation in 3.2.2 und dem Test der Kantenhistogramme in 3.2.3.
Allerdings gibt es auch Verfahren, wie in „Algorithms on Trees and Graphs“ [49], die analog zu 3.2.4 alle möglichen Abbildungen direkt aufzählen und erweitern. Diese verwenden, wie im Fall des *VF2-Algorithmus* aus „A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs“ [8] oder des *CSI-Algorithmus* aus „Common subgraph isomorphism detection by backtracking search“ [33], semantische und syntaktische Prädikate, um zu entscheiden, ob die aktuell gehaltene Abbildung erweitert werden soll, oder nicht. Der in 3.2.4 beschriebene Algorithmus unterscheidet sich nur insofern vom VF2-Algorithmus bzw. dem CSI-Algorithmus, als dass der Backtrackingpfad innerhalb des Musters vorberechnet und die Merkmale mittels der ◄-Relation als Prädikat verwendet werden, um unnötige Startpunkte und Erweiterungen zu vermeiden.
Zuletzt soll hier noch ein dritter Ansatz, „Efficient Subgraph Isomorphism Detection: A Decomposition Approach“ [39], erwähnt werden. In diesem Verfahren werden in einem Vorverarbeitungsschritt die zu suchenden Mustergraphen binär in (möglichst wenige) kleinere Graphen zerlegt, aus denen sie zusammengesetzt werden können. Soll nun gegen einen Eingabegraphen getestet werden, so werden alle Vorkommen der kleinsten Graphen gesucht und dann schrittweise getestet, ob diese zu den Mustern zusammengesetzt werden können. Allerdings wird auch mit diesem Algorithmus keine subexponentielle Laufzeit im schlechtesten Fall erreicht.
Während die erwähnten Algorithmen auf allgemeinen Graphen arbeiten und im schlechtesten Fall exponentielle Laufzeit haben, muss allerdings erwähnt werden, dass es für spezielle Graphen effiziente Algorithmen gibt, wie z.B. für planare Graphen („Subgraph Isomorphism in Planar Graphs and Related Problesm“ [15]). Dabei werden diese z.B. in Bäume bestimmter Breite zerlegt und dadurch die Laufzeit beschränkt.
Fazit dieses Überblicks ist, dass es keinen subexponentiellen Algorithmus für dieses Problem gibt.

A.3.2 Pseudocode Subgraphisomorphie

◄-Relation Die ◄-Relation ist wie folgt auf den Merkmalstripeln, die *classify-node* zurück gibt, definiert

```
1 funct ◄ (X, Y)
2   Seien X=(a,b,c) und Y=(x,y,z)
3   if a ≤ x+y ∧ ∃ i ∈ z : max{c} ≤ i
4     return true
5 else
6     return false
7   fi
8 end
```

Subgraphisomorphietest Der Subgraphisomorphietest zählt alle Vorkommen von G in H mittels *exhaustive search* und einem modifizierten Backtracking-Verfahren auf. Dabei wird die $=$-Relation im Backtracking durch die ◄-Relation ersetzt. Außerdem wird im Rekursionsfuß (Zeile 4) ein verkürzter Isomorphietest durchgeführt, bei dem die Merkmale aus G den zugeordneten Knoten aus H zugewiesen und dann ausschließlich über die ausgewählten Kanten transferiert werden, um das Merkmalshistogramm $\mathcal{H}_H^1$ zu berechnen, das für den abschließenden Konsistenztest verwendet wird.

Algorithm A.3.1.

```
1  proc subgraph-isomorphy(G,H)
2    Berechne (konvergente) Transferfunktionen g_G^0, g_G^1, g_G^n, g_H^0
3    Berechne H_G^1, H_G^n für g_G^1, g_G^n
4    Berechne 5-Tupel-Kantenhistogramm E_G^0 aus g_G^0
5    Berechne maximale BFS-Tiefe bfsMax für G
6    list ← calculate-visiting-order(G, H_G^n);
7    Sei n_G^a der Startknoten von list
9    for all n_H^a ∈ E(H) : m(n_G^a) ◄ m(n_H^a)
10     do
11     Berechne 5-Tupel-Kantenhistogramm E_H^0 für alle Kanten e = {u,v}
12     mit u,v haben BFS-Tiefe kleiner bfsMax
13     if E_G^0 ◄ E_H^0
14       comment: Muss nur grob abgeschätzt werden, z.B. alle Merkmale
15       comment: enthalten (◄) und ausreichende Häufigkeiten
16       then
17           call modified-backtrack(list, 0);
18     fi
19   od
20 end
```

Algorithm A.3.2.

```
1  proc modified-backtrack(list, i)
2    if size(list) = i
3      then
4           comment: Rekursionsfuss
5           if short-isomorphism-check(iso, g^n, G, H)
6             then output (Isomorphieabbildung gefunden !);
7           fi
8       else comment: Rekursionsschritt
9           Sei e^G = {u^G, v^G} die i.te Kante aus list
10          if iso(e_i^G) = ⊥
11            then
12                 if ¬visited(u^G, v^G)
13                   then
14                        comment: Teste alle Kanten mit passenden
15                        comment: Merkmalszahlen m(u^H), m(v^H) ∈ M_M
16                        for all e^H = {u^H, v^H} ∈ E : m(u^G) ◀ m(u^H) ∧ m(v^G) ◀ m(v^H)
17                            do
18                            iso(e^G) ← e^H;
19                            call modified-backtrack(list, i+1);
20                            iso(e^G) ← ⊥;
21                        od
22                 elsif one-is-visited(u^G, v^G)
23                      then
24                           Sei u^G der besuchte und u^H der ihm zugeordnete Knoten
25                           for all e^H ∈ AdjacentEdges(u^H) : m(v^G) ◀ m(v^H)
26                               do
27                               iso(e^G) ← e^H;
28                               call modified-backtrack(list, i+1);
29                               iso(e^G) ← ⊥;
30                           od
31                       else
32                           Seien u^H, v^H die u^G, v^G zugeordneten Knoten
33                           if exists-edge(u^H, v^H)
34                             then
35                                  Sei e^H = {u^H, v^H} diese Kante
36                                  iso(e^G) ← e^H;
37                                  call modified-backtrack(list, i+1);
38                                  iso(e^G) ← ⊥;
39                           fi
40                 fi
41            end
```

Algorithm A.3.3.

```
1  proc short-isomorphism-check(iso, g^n, G, H)
2    comment: iso stellt neben der Kantenzuweisung,
3    comment: auch eine Knotenzuweisung bereit
4    Berechne ĝ^1_H aus ĝ^0_H(u) = g^0_G(iso^{-1}(u))
5    Berechne Kantenhistogramm Ê^1_H aus ĝ^1_H
6    if E^1_G = Ê^1_H
7      then
8           return (true )
9       else
10          return (false )
11   fi
12 end
```

A.4 Motif-Suche

A.4.1 Überblick der Arbeiten zur Motif-Suche

In diesem Abschnitt soll ein kurzer Überblick über die bisherigen Veröffentlichungen zum Thema der Motif-Suche, bzw. der Suche nach häufigen Subgraphen gegeben werden. Dieser Überblick, wie auch die anderen, erhebt nicht den Anspruch, umfassend zu sein, ist aber bemüht, die wichtigsten Arbeiten vorzustellen.
Eine der ersten Arbeiten über die Suche nach häufigen Subgraphen ist „Diagonally Subgraphs Pattern Mining“ [7]. Hier wird auf zwei algorithmische Verfahrensweisen verwiesen: Dem Apriori-Ansatz und der DFS-Suche.
Der Apriori-Ansatz stellt im wesentlichen eine Breitensuche *Breath-First-Search (BFS)* dar, indem er aus häufigen Mustern der Größe k Muster der Größe $k+1$ erstellt und anhand einer Größe *minSup*, dem minimalen Support eines Musters, feststellt, ob ein Muster als häufig zu betrachten ist. Diese Ansätze unterscheiden sich hauptsächlich darin, was als Grundbaustein verwendet wird: Knoten, Kanten oder kantendisjunkte Pfade.
Bei der Tiefensuche werden gefundene Muster von einem Graphen direkt abgeleitet. Für jedes dieser Muster wird eine einzigartige kanonische Markierung berechnet, die dazu verwendet wird, eine (lexikographische) Ordnung über alle möglichen Muster und damit eine Reihenfolge herzustellen, in der der Mustersuchraum durchwandert wird. Dieser Artikel schlägt zudem einen Hybrid-Ansatz vor, der Ideen aus beiden Verfahren verwendet, um Suchbereiche, die wenig Erfolg versprechen, auszusparen.
Eine Anwendung und Erweiterung des erwähnten Apriori-Ansatzes findet sich in „Towards Motif Detection in Netzworks: Frequency Concepts and Flexible Search“ [18], welcher in Kapitel 3.3.2 besprochen wird. Die hierzu gehörende Anwendung ist in „MAVisto: a tool for the eploration of network motifs“ [19] beschrieben. Um Muster nun tatsächlich suchen zu können, erweiterten diese Autoren in *PatternGravisto* [6] die Definition eines Such-Musters als Graph mit Regulären Ausdrücken als Kantenbeschriftungen, um alle Vorkommen mit diesem Ansatz suchen zu können. Hier wird auch beschrieben, wie mit Hilfe eines kräfte-basierten Layout-Verfahrens eine übersichtliche Visualisierung der Motife zu erreichen ist.
Zuletzt existiert ein anderer neuartiger Ansatz in „Discovering frequent topological structures from graph datasets“ [25]. Dieser sucht ebenfalls häufige Strukturen in Graphen, wobei aber die Beziehung des topologischen Minors [11] beachtet wird. Dies geschieht mit Hilfe einer *Relabeling*-Funktion, mit deren Hilfe *unabhängige Pfade* zu einer einzelnen Kante kontrahiert und mit einer neuen Markierung versehen werden. Anhand dieser kontrahierten Pfade können die gefundenen Strukturen kategorisiert werden.
Die für diese Diplomarbeit wirklich relevante Arbeit ist jedoch „Efficient sampling algorithm for estimating subgraph concentrations and detecting netork motifs“ [42]. Diese wird in Kapitel 3.3.3 näher beschrieben. Es geht dabei um die Abschätzung der Subgraphhäufigkeiten mit Hilfe von zufällig gewähl-

ten Stichproben-Teilgraphen, sowie der Berechnung der Wahrscheinlichkeit für deren Entdeckung. Diese wird dann zu einem Schätzwert für die tatsächliche Häufigkeit verrechnet. Es handelt sich dabei um ein relativ leistungsfähiges und schnelles Verfahren, dessen Laufzeit nicht von der Größe des Eingabegraphen, wohl aber von der gesuchten Mustergröße abhängt. Diese Arbeit wertet zusätzlich zu der Beschreibung des Algorithmus noch die gefundenen Muster im biologischen Kontext vorheriger Arbeiten U.Alons aus. So finden sich hier die schon vorher publizierten *Feed-forward-loops*, *Single-input-modules* und konkrete Formen der *Dense-overlapping-regions*, sowie diverse Mischformen derselben. Außerdem wird auf die Frage eingegangen, wie groß eine Stichprobe gewählt werden muss, damit gute Ergebnisse erzielt werden. Auch findet sich hier die Idee eines konvergenten Sampling-Verfahrens, die von dieser Diplomarbeit aufgegriffen wurde.

A.4.2 Pseudocode Frequent Subpattern

Uri Alons Sampling-Algorithmus Dieser Algorithmus wählt k mal Teilgraphen mit n Knoten aus G aus und berechnet dann deren *Sampling*-Wahrscheinlichkeit. Diese wird für jeden Subgraphtypen zu einem *Score* zusammen addiert, der als Schätzwert für die Häufigkeit (oder Konzentration) dieses Subgraphtypen ausgegeben wird.

Algorithm A.4.1.

proc *sampling*(G,n)
  Sei $G=(V,E)$
  for $i \leftarrow 1$ **to** k **do**
    Wähle zufällig $e=\{u,v\} \in E$;
    var $\langle V_S \leftarrow \{u,v\};\ E_S \leftarrow \{e\};\ \rangle$
      while $|\ V_S\ | \neq n$ **do**
        neighbours $\leftarrow$ *neighbour-edges*(E_S);
        Wähle zufällig $e=\{u,v\}$ aus *neighbours* aus
        $\langle V_S \leftarrow V_S \cup \{u,v\};\ E_S \leftarrow E_S \cup \{e\};\ \rangle$
      od
      Berechne Wahrscheinlichkeit
      $Pr[\text{subgraph}\{V_S\} \text{ gefunden}] = \sum_{\sigma \in S_m} \prod_{E_j \in \sigma} Pr[E_j = e_j \mid (E_1,\ldots,E_j-1) = (e_1,\ldots,e_j-1)]$
      für Auswählen des n-Knoten-Subgraph. S_m ist Menge aller Permutationen
      der Kanten $(e_1,\ldots,e_m)$, die zur Auswahl führen können
    od
    Summiere die Scores
      $W(\{v_1,\ldots,v_n\}) = \frac{1}{Pr[\text{subgraph}\{V_S\} \text{ gefunden}]}$
    für Subgraphtyp i zu Score
      $S_i = \sum W(subgsraph_i)$
    auf und berechne geschätzte $subgraph_i$-Konzentration
      $C_i = S_i / \sum S_k$
  end

Konvergenz-Sampling Beim verbesserten Sampling wird von dem schnellen Graphisomorphietest Gebrauch gemacht, um auf die in einer *Move-to-Front*-Liste gehaltenen Tupel (T_i,h_i) mit T_i als i.ter Subgraphtopologie und h_i deren Häufigkeit, mit der diese gefunden wurden, zuzugreifen. Dabei werden solange Teilgraphen ausgewählt, bis k-mal hintereinander keine neue Topologie gefunden wurde. Die gefundenen relativen Häufigkeiten werden dann als Schätzwert für die tatsächliche Häufigkeit der entsprechenden Topologie ausgegeben. Dadurch ist es wahrscheinlicher, eine vollständige Zerlegung von G

in seine Komponenten der Grösse k zu erhalten. Der Unterschied zu dem vorherigen Verfahren ist auch, dass der gezogene Subgraph nur aus den k Kanten und den adjazenten Knoten besteht.

Algorithm A.4.2.

```
1  proc convergence-sampling(G,n)
2     L_MF ← ∅;
3     r ← k;
4     while 0 ≤ r
5           do T ← sample-topology(G,n);
6           if T ∉ L_MF
7              then L_MF ← L_MF ∪ {(T,1)}
8                   r ← k;
9              else (T,h) ← get(L_MF,T);
10                  h ← h + 1;
11                  r ← r − 1;
12          fi
13    od
14    Berechne für jede Topologie
15    h_T = h_T / Σ_{(X,h_X)∈L_MF} h_X
16 end
17
18 proc sample-topologie(G,n)
19    Wähle zufällig e = {u,v} ∈ E;
20    var ⟨E_S ← {e}; ⟩
21       while | E_S |≠ n do
22             neighbours ← neighbour-edges(E_S);
23             Wähle zufällig e aus neighbours aus
24             ⟨E_S ← E_S ∪ {e}; ⟩
25       od
26       return (E_S);
27    end
```

A.5 Weitere Ergebnisse

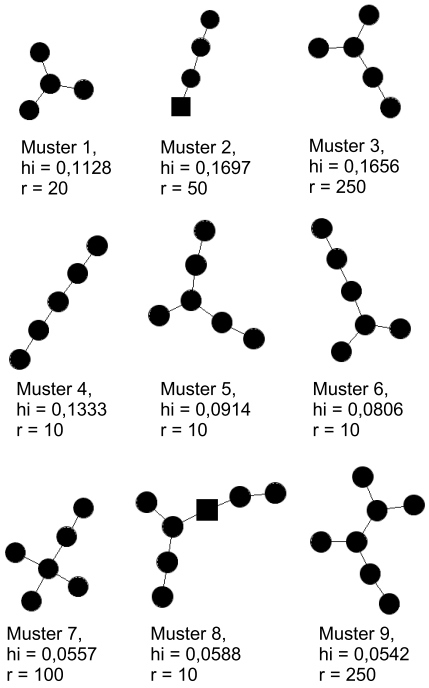


Abbildung A.1: Ergebnisse *A.thaliana* (Protein-Interaktion)

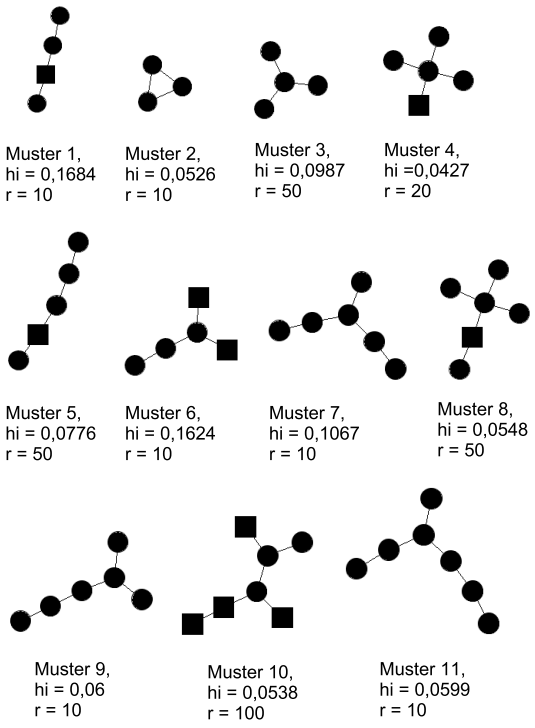


Abbildung A.2: Ergebnisse *D.melanogaster* (Protein-Interaktion)

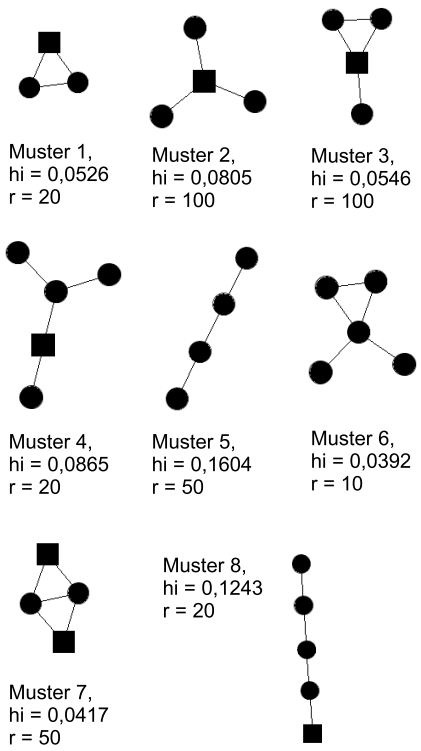


Abbildung A.3: Ergebnisse *S.cerevisiae* (Protein-Interaktion)

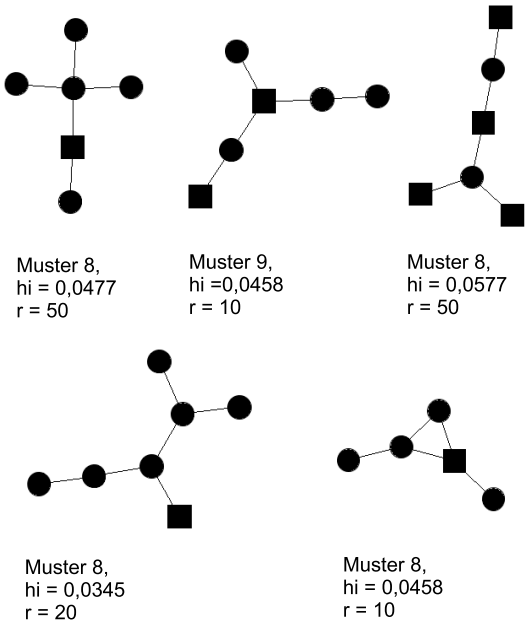


Abbildung A.4: Ergebnisse *S.cerevisiae* (Protein-Interaktion)

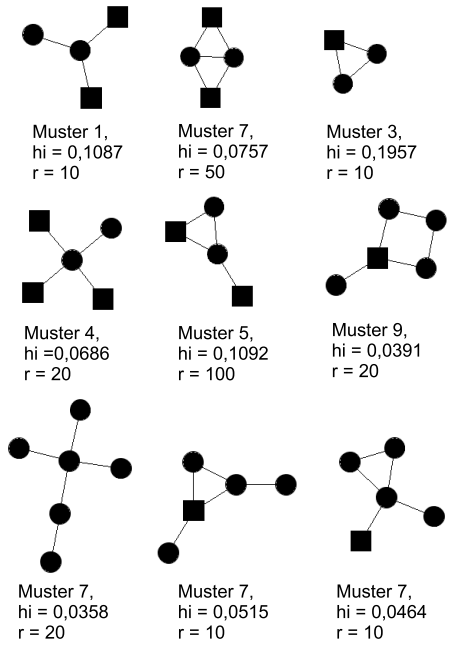


Abbildung A.5: Ergebnisse *E.coli* (Protein-Interaktion)

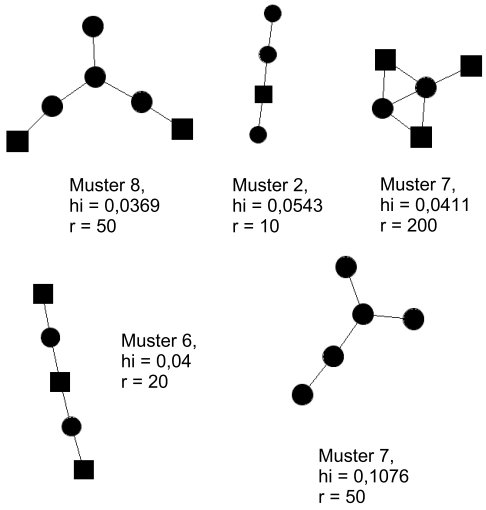


Abbildung A.6: Ergebnisse *E.coli* (Protein-Interaktion)

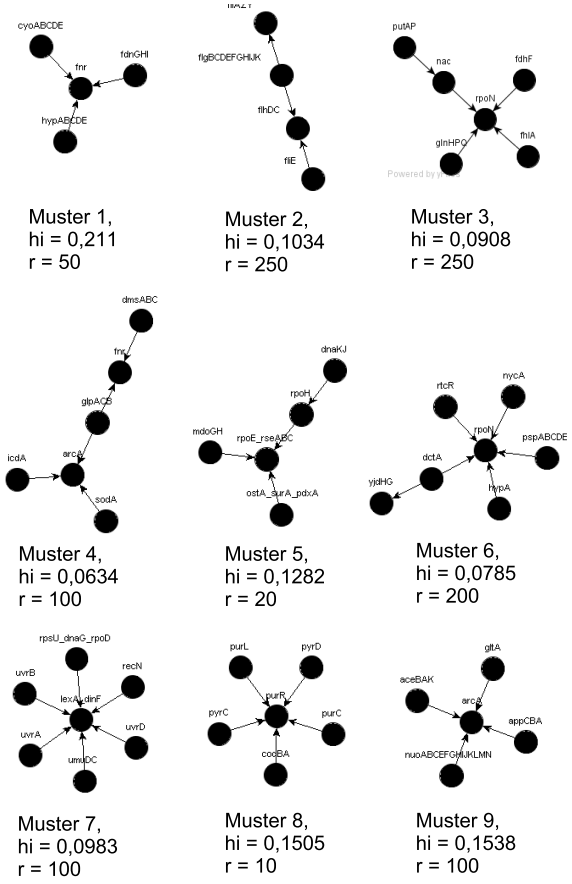


Abbildung A.7: Ergebnisse *E.coli* (Uri Alon Datensatz)

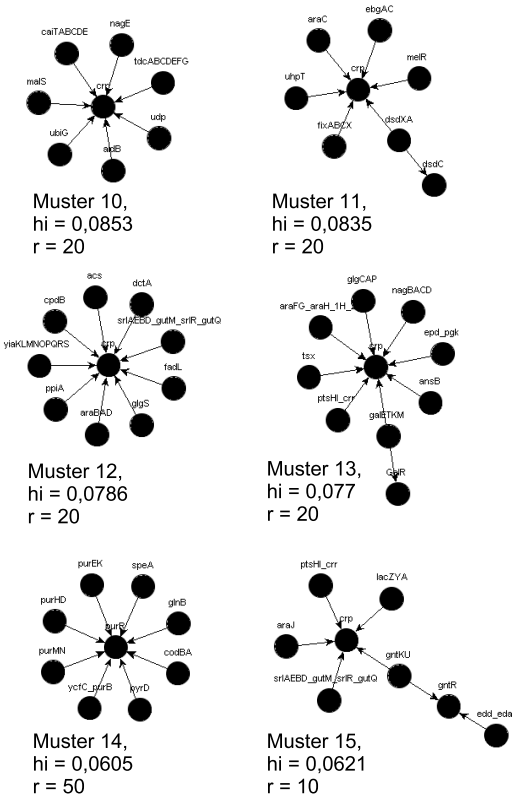


Abbildung A.8: Ergebnisse *E.coli* (Uri Alon Datensatz)

Literaturverzeichnis

[1] http://www.ncbi.nlm.nih.gov/Entrez.

[2] Uri Alon. *An Introduction to Systems Biology*. Chapman & Hall/CRC, 2006.

[3] Bairoch and Apweiler. The swiss-prot protein sequence database and its supplement trembl in 2000. *Nucleic Acids Res.*, 28(1):45–48, 2000.

[4] A Birkland and G Yona. Biozon: a system for unification, management and analysis of heterogenous biological data. *BMC Bioinformatics*, 7(70), 2006.

[5] BJ. Breitkreutz, C. Stark, and Tyers M. Osprey: A network visualization system. *Genome Biology*, 4(3):R22, 2003.

[6] Falk Schreiber Christian Klukas, Dirk Koschützki. Graph pattern analysis with *PatternGravisto*. *Journal of Graph Algorithms and Applications*, 9(1):19–29, 2005.

[7] Moti Cohen and Ehud Gudes. Diagonally subgraphs pattern mining. In *DMKD '04: Proceedings of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 51–58, New York, NY, USA, 2004. ACM Press.

[8] Luigi P. Cordella, Pasqualle Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[9] S B Davidson, J Crabtree, B P Brunk, J Schug, V Tannen, G C Overton, and C J Stoeckert. K2/kleisli and gus: Experiments in integrated access to genomic data sources. *IPM Systems Journal*, 40(2):512–530, 2001.

[10] C.C.Gotlieb D.G.Corneil. An efficient algorithm for graph isomorphism. *Journal of the Association for Computing Machinery*, 17(1):51–64, 1970.

[11] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000.

[12] L Donelson, P Tarczy-Hornoch, P Mork, C Dolan, J A Mitchell, M Barrier, and H Mei. The biomediator system as a data integration tool to answer diverse biologic queries. *Medinfo*, 11(2):768–772, 2004.

[13] Larry E. Druffel Douglas C. Schmidt. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the Association for Computing Machinery*, 23(3):433–445, 1976.

[14] Entigen. http://www.antigen.com/library, 2001.

[15] David Eppstein. Subgraph isomorphism in planar graphs and related problesm. *Journal of Graph Algorithms and Applications*, 3(3):1–27, 1999.

[16] S Peri et al. Development of human protein reference database as initials platform for approaching systems biology in humans. *Genome Research*, 13:2363–2371, 2003.

[17] T Etzold and P Argos. Srs - an indexing and retrieval tool for flat file data libraries. *Computer applications in the biosciences*, 9(1):51–60, 2004.

[18] Henning Schwöbbermeyer Falk Schreiber. Towards motif detection in networks: Frequency concepts and flexible search. *Biology (NETTAB'04)*, pages 91–102, 2004.

[19] Henning Schwöbbermeyer Falk Schreiber. Mavisto: a tool for exploration of network motifs. *Bioinformatics Application Note*, 21(17):3572–3574, 2005.

[20] National Center for Biotechnology Information. Entrez - search and retrieval system. http://www.ncbi.nlm.nih.gov/Entrez, 2006.

[21] L M Haas, P M Schwarz, P Kodali, E Kotlar, J E Rice, and W C Swope. Discoverylink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511, 2001.

[22] H Hermjakob, L Montecchi-Palazzi, C Lewington, S Mudali, S Kerrien, S Orchard, M vingron, B Roechert, P Roepstorff, A Valencia, H Margalit, J Armstrong, a Bairoch, G Cesareni, D Sherman, and R Apweiler. Intact - an open source molecular interaction database. *Nucleic Acid Research*, 32:D452–D455, 2004.

[23] H.M.Berman, J.Westbrook, Z.Feng, G.Gilliland, T.N.Bhat, H.Weissig, I.N.Shindyalov, and P.E.Bourne. The protein data bank. *Nucleic Acids Research*, 28:pp. 235–242, 2000.

[24] Z. Hu, J. Mellor, J. Wu, and C. DeLisi. Visant: an online visualization and analysis tool for biological interaction data. *BMC Bioinformatics*, 5(17), 2004.

[25] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal. Discovering frequent topological structures from graph datasets. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 606–611, New York, NY, USA, 2005. ACM Press.

[26] Pak Chung Sham Jing Hua Zhao. Generic number systems and haplotype analysis. *Computer Methods and Programs in Biomedicine, Elsevier*, 70:1–9, 2003.

[27] Wendy J. Myrvold John M. Boyer. On the cutting edge: A simplified $o(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[28] Gross L. Gross Jonathan L. Gross, Jay Yellen. *Handbook of Graph Theory*. CRC Press, 2004.

[29] J.R.Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, 1976.

[30] M. Kanehisa. A database for post-genome analysis. *Trends Genet.*, 13:375–376, 1997.

[31] Ulrich Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik*, pages 57f, Satz 3.16. Vieweg, 2002.

[32] C J Krieger, P Zhang, L A Mueller, A Wang, S Paley, M Arnoud, J Pick, S Y Rhee, and P D Karp. Metacyc: a multiorganism database of metabolic pathways and enzymes. *Nucleic Acid Research*, 34:D438–D4422, 2004.

[33] Evgeny B. Krissinel and Kim Henrick. Common subgraph isomorphism detection by backtracking search. *Software - Practice and Experience*, 34:591–607, 2004.

[34] M Krull, S Pistor, N Voss, A Kel, I Reuter, D Kronenberg, H Michael, K Schwarzer, A Potapov, C Choi, O Kel-Margoulis, and E Wingender. Transpath: An information resource for storing and visualizing signaling pathways and their pathological aberrations. *Nucleic Acid research*, 34(D546-D551), 2006.

[35] L.Priese and H.Wimmel. *Petri-Netze*. Springer, 2003.

[36] J. Manning. *Geometric symmetry in graphs*. PhD thesis, Purdue Univ., 1990.

[37] V Matys, O Kel-Margoulis, E Fricke, I Liebich, S Land, A Barre-Dirrie, I Reuter, D Checkmenev, M Krull, K Hornischer, N Voss, P Stegmaier, B Lewicki-Potapov, H Saxel, A Kel, and E Wingender. Transfac and its module transcompel: transcriptional gene regulation in eukaryotes. *Nucleic Acid Research*, 34:D108–D110, 2006.

[38] Brendan D. McKay. Nauty - no automorphism, yes ? http://cs.anu.edu.au/people/bdm/nauty/.

[39] 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.

[40] M.Sirava, T. Schäfer, M. Eiglsperger, M. Kaufmann, O. Kohlbacher, E. Bornberg-Bauer, and H. P. Lenhof. Biominer - modelling, analyzing, and visualizing biochemical pathways and networks. *Bioinformatics*, 18(2):S219–S230, 2002.

[41] N. J. Mulder, R. Apweiler, T. K. Attwood, A. Bairoch, A. Bateman, D. Binns, P. Bradley, P. Bork, P. Bucher, L. Cerutti, R. Copley, E. Courcelle, U. Das, R. Durbin, W. Fleischmann, J. Gough, D. Haft, N. Harte, N. Hulo, D. Kahn, A. Kanapin, M. Krestyaninova, D. Lonsdale, R. Lopez, I. Letunic, M. Madera, J. Maslen, J. McDowall, A. Mitchell, A. N. Nikolskaya, S. Orchard, M. Pagni, C. P. Ponting, E. Quevillon, J. Selengut, C. J. Sigrist, V. Silventoinen, D. J. Studholme, R. Vaughan, and C. H. Wu. Interpro, progress and status in 2005. *Nucleic Acids Research*, 33:D201–D205, 2005.

[42] R.Milo N.Kashtan, S.Itzkovitz and U.Alon. Efficient sampling algorithm for estimating concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, 2004.

[43] K D Pruitt, T Tatusova, and D R Maglott. Ncbi reference sequence (refseq): a curated nonredundant sequence database of genomes, transcripts and proteins. *Nucleic Acid Research*, 33(1):D501–D504, 2005.

[44] L Salwinski, C S Miller, A J Smith, F K Pettit, J U Bowie, and D Eisenberg. The database of interacting proteins: 2004 update. *Nucleic Acid Research*, 32:D449–D451, 2004.

[45] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, , and Trey Ideker. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13:2498–2504, 2003.

[46] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update rules. *ACM*, 4:488–492, 1984.

[47] R Stevens, P Baker, S Bechhofer, G Ng, A Jacoby, N W Paton, C A Goble, and A Brass. Tambis: transparent access to multiple bioinformatics information sources. *Bioinformatics*, 16(2):184–185, 2000.

[48] Stephen H. Unger. Git - a heuristic program for testing pairs of directed line graphs for isomorphism. *Communications of the ACM*, 7(1):26–34, 1964.

[49] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer, 2002.

[50] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfiles : Visualization and automatic layout of graphs. *In 11th Symposium on Graph Drawing (GD'01)*, 2001.

[51] A Zanzoni, L Montecchi-Palazzi, M Quondam, G Ausiello, M Helmer-Citterich, and G Cesareni. Mint: a molecular interaction database. *FEBS Letters*, 513(1):135–140, 2002.