Bachelorarbeit zum Thema

Subgraphisomorphie
- auf Knowledge-Graphen -

im Wintersemester 2018/2019

vorgelegt von Tim Steinbach
Freitag, 26. April 2019

Studienfach:	Bachelor Mathematik
Matrikelnummer:	5244072
E-Mail-Adresse:	SteinbT0@Smail.Uni-Koeln.de

– Dr. Jens Dörpinghaus –

– Dr. Vera Weil –

Fraunhofer SCAI

SCAI
Institute for Algorithms and Scientific Computing
Fraunhofer Gesellschaft

MNF
Department Mathematik / Informatik
Universität zu Köln

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Köln, Freitag den 26. April 2019

_____
(Tim Steinbach)

Inhaltsverzeichnis

II. Verzeichnisse

1. Einleitung & Hintergrund

Diese Bachelorarbeit befasst sich mit dem *Subgraphisomorphie-Problem* auf *Knowledge-Graphen*. Dieses Problem ist folgendermaßen definiert: Bei zwei gegebenen Graphen G und H behandelt es die Frage, ob H zu einem Subgraph von G isomorph ist. Wie später in Abschnitt 3.4 gezeigt wird, ist dieses Problem $\mathcal{NP}$-vollständig.

Das Subgraph-Problem kann auf die verschiedensten Graphklassen angewendet werden, so auch auf die der Knowledge-Graphen. Diese Bachelorarbeit beantwortet die Frage, ob in einem Knowledge-Graph, also in einer Graphdatenbank, bestimmte Graphstrukturen wiederzufinden sind. Dazu werden Suchanfragen als Digraph codiert und als Subgraph im Knowledge-Graph gesucht. Das Problem wird damit als Subgraph-Problem behandelt.

1.1. Motivation

[In diesem Abschnitt wird sich an 1, 2, orientiert]

Der Subgraphisomorphismus ist eine wichtige und sehr allgemeine Form der Mustererkennung, die in Bereichen wie computergestütztem Design, Bildverarbeitung, Graphgrammatiken, Graphtransformation und Biocomputing praktische Anwendung findet. Die Nutzung in all diesen Bereichen liefert eine nicht mathematische Motivation sich mit dem Subgraph-Problem zu beschäftigen. Aus mathematischer bzw. informatischer Sicht sind $\mathcal{NP}$-vollständige Probleme immer eine Herausforderung, nicht nur für die theoretische Informatik, der man sich gerne stellt.

Es gibt verschiedene Arten der Mustererkennung, wie etwa die Stringsuche in Texten, Sequenzabgleiche, Baumvergleiche oder eben Musterabgleiche in Graphen. In der immer spezieller werdenden Hierarchie von Mustererkennungs- und damit verwandten Problemen kommt der Subgraphisomorphismus direkt nach dem Graphisomorphismus an zweiter Stelle.

1. Graphisomorphie
2. **Subgraphisomorphie**
3. Maximaler, gemeinsamer Subgraph
   - 3.1. Maximaler, gemeinsamer induzierter Subgraph:
     Ein Graph, der ein induzierter Subgraph von zwei gegebenen Graphen ist und maximal viele Knoten hat.
   - 3.2. Maximaler, gemeinsamer Kanten-Subgraph:
     Ein Graph, der ein Subgraph von zwei gegebenen Graphen ist und möglichst viele Kanten aufweist.

Hierbei suchen diese drei Beispiele nach exakten Übereinstimmungen zwischen zwei Graphen. Der Graphisomorphismus überprüft dabei, ob diese zwei Graphen komplett Übereinstimmen. Der Subgraphisomorphismus vergleicht den Eingabegraph mit Teilen des Supergraphen. Die in Punkt drei erwähnten Musterabgleiche, suchen ebenfalls nach Übereinstimmungen bei Subgraphen, allerdings haben diese zusätzlich speziellere Eigenschaften. Es existieren auch

Algorithmen die nach Näherungen suchen, allerdings wird in dieser Bachelorarbeit nicht näher auf diese eingegangen.

Des Weiteren ist auch das Konzept des Knowledge-Graphen interessant, sowie dessen Entwicklung aus dem Semantic Web und seinem Zusammenhang zur semantischen Suche. Dies alles sind junge Gebiete in direkter Beziehung zum Internet. Wie diese Beziehung genau besteht wird später in Abschnitt 1.3 deutlich.

1.2. Zielsetzung der Arbeit

Die Naturwissenschaften sind ein interessantes Themenfeld, welches schon immer von Menschen erforscht wurde. Allerdings war es in der Vergangenheit schwerer sich über bestimmte Erkenntnisse zu informieren und diese dann ggf. weiter zu untersuchen. Forschungsergebnisse und das darin enthaltene „Wissen“, konnte z. B. im Mittelalter nur langsam von Mönchen kopiert und in Bibliotheken gesammelt werden.

Viele der Informationen in Büchern, sind zum größten Teil auch heute noch unzugänglich. Mit Hilfe der semantischen Suche soll versucht werden, Wissen wieder greifbarer und Literatur zugänglicher zu machen.

Im Rahmen dieser Bachelorarbeit wird ein Microservice für die bestehende Suchmaschine SCAIView implementiert. Dieser Microservice soll möglichst schnell Wissensabfragen im Knowledge-Graph ermöglichen. Der Knowledge-Graph ist eine Sammlung verschiedener Datenbanken mit medizinischem Kontext, der Daten in einer Graphstruktur abspeichert. Abfragen im Knowledge-Graph werden dementsprechend als Graph codiert. Um diese Abfragen effizient zu gestalten ist es notwendig das Subgraph-Problem zu betrachten. Für die Lösung dieses Problems werden verschiedene Algorithmen implementiert und miteinander verglichen. Im Verlauf der Bachelorarbeit wird auch darauf eingegangen, ob Algorithmen existieren, die effizienter abschneiden können und wo die Grenzen der Algorithmen liegen. Schließlich wird auf das Potential und den Nutzen in der Praxis hingewiesen.

1.3. Ursprung des Problems

Mit dem Beginn des Web-Zeitalters in den 1990er Jahren begannen Unternehmen und Personen Homepages online zu stellen und füllten diese mit Text, Videos und Bildern. Ins tägliche Leben zog das Internet bei vielen Nutzern aber erst mit seiner sozialen Nutzung im *Social Web* ein und das Internet wurde eine Welt der Interaktion. Für die privaten Nutzer waren Seiten wie Wikipedia, YouTube und Facebook in den 2000er Jahren die am häufigsten besuchten Seiten. Das Web, welches sich ständig weiterentwickelt, ist zu diesem Zeitpunkt noch syntaxorientiert, weshalb sowohl diese Seiten, wie auch alle anderen Homepages, nur direkt aufgerufen oder gefunden werden konnten. Wollte man z. B. über Google gefunden werden, mussten im Metatext der Seite bestimmte Stichwörter auftauchen. [14]

Metatexte sind notwendig damit Computer Inhalte besser finden und verarbeiten können, denn viele Suchbegriffe sind mehrdeutig und werden in unterschiedlichen Disziplinen benutzt. Dadurch sind auch die Suchergebnisse vielfältig und bieten nicht unbedingt die Information, die sich der Suchende erhofft hat. Computer müssen also nicht nur in der Lage sein die Eingabe zu verstehen, sondern diese auch in Beziehung zu anderen Deutungen setzen. Es entwickelte sich das *Semantic Web* als Internet der Bedeutungen. [14]

Semantic Web bezeichnet eine Technologie, bei der die Daten einer konventionellen Webseite um strukturierte Daten angereichert werden, um die Bedeutung der Information für Maschinen leichter auswertbar zu machen. Durch diesen Ansatz können Menschen und Computer wesentlich besser miteinander kooperieren und es können intelligentere Webservices kreiert werden. Die erste Ausprägung dieses neuen Web ist die *Semantic Search*, wie sie 2013 von Google eingeführt wurde. [15]

Für die praktische Umsetzung des Semantic Web kommen unterschiedliche Technologien zum Einsatz. Entsprechende Webstandards, viele davon vom *World Wide Web Consortium*[1] entwickelt, bieten die Voraussetzungen, dass die Informationen und deren Beziehungen von Computern interpretiert und ausgewertet werden können. Zum einen können mit dem *Ressource Description Framework*[2] Webseiten näher beschrieben werden, zum anderen werden via *Rule Interchange Format*[3] die Regeln bestimmt, denen die Maschine folgen soll, um Bedeutungszusammenhänge zu erkennen. Außerdem gibt es noch *Dublin Core*[4], mit dem digitale Inhalte mit maschinenlesbaren Meta-Angaben versehen werden können. Eventuelle Bedeutungsschwierigkeiten im Semantic Web werden durch solche Modelle der Taxonomie und der Klassifizierung ausgeräumt. [16]

Ein weiterer wesentlicher Bestandteil des Semantic Web ist die *Web Ontology Language*[5]. OWL basiert technisch auf der RDF-Syntax, geht aber über die Ausdrucksmächtigkeit hinaus. Zusätzlich zu RDF werden weitere Sprachkonstrukte eingeführt, die es erlauben, Ausdrücke ähnlich der Prädikatenlogik zu formulieren.

Die Datenbank-Abfragesprache *SPARQL*[6] ist ein Datenzugriffsprotokoll für das Semantic Web. Es wird seit 2006 von der RDF Data Access Working Group[7] des W3C entwickelt und standardisiert. SPARQL ist der Nachfolger mehrerer Abfragesprachen, wie z. B. der RDF Query Language. Es funktioniert für jede Datenquelle, die in RDF abgebildet werden kann. [17]

Eine SPARQL-Abfrage besteht aus einem 3-Tupel, in dem jedes Element (Subjekt, Prädikat und Objekt) variabel sein darf. Lösungen für die Variablen werden gefunden, indem die Muster in der Abfrage mit Tripeln in den Datensätzen abgeglichen werden. SPARQL stellt vier Möglichkeiten zur Abfrage bereit. Es kann verwendet werden um:

- ASK: Fragen, ob mindestens eine Übereinstimmung des Anfragemusters in den RDF-Diagrammdaten vorhanden ist.
- SELECT: Alle oder einige dieser Übereinstimmungen in Tabellenform auswählen.
- CONSTRUCT: Einen RDF-Graphen konstruieren, indem die Variablen in den Übereinstimmungen ersetzt werden.
- DESCRIBE: Die gefundenen Übereinstimmungen beschreiben, indem ein relevanter RDF-Graph erstellt wird.

SPARQL ist genau wie Cypher[8] eine sehr leistungsfähige Abfragesprache. Sie basiert ebenfalls auf Graphen und ist auf den ersten Blick sehr ähnlich zu dem was der Microservice `Subgraph`

[1] W3C: World Wide Web Consortium – `https://www.w3.org/`
[2] RDF: Ressource Description Framework – `https://www.w3.org/RDF/`
[3] RIF: Rule Interchange Format – `https://www.w3.org/TR/rif-overview/`
[4] Dublin Core – `http://dublincore.org/`
[5] OWL: Web Ontology Language – `https://www.w3.org/OWL/`
[6] SPARQL: Simple Protocol And RDF Query Language – `https://www.w3.org/2001/sw/wiki/SPARQL`
[7] DAWG: Data Access Working Group
[8] *siehe* Abschnitt 3.3.1: Neo4j & Cypher

später leisten soll. Allerdings besteht ein wesentlicher Unterschied darin, dass mit Cypher zusätzlich auch *Kanteneigenschaften* abgefragt und berücksichtigt werden können. Das ist mit SPARQL nicht möglich.

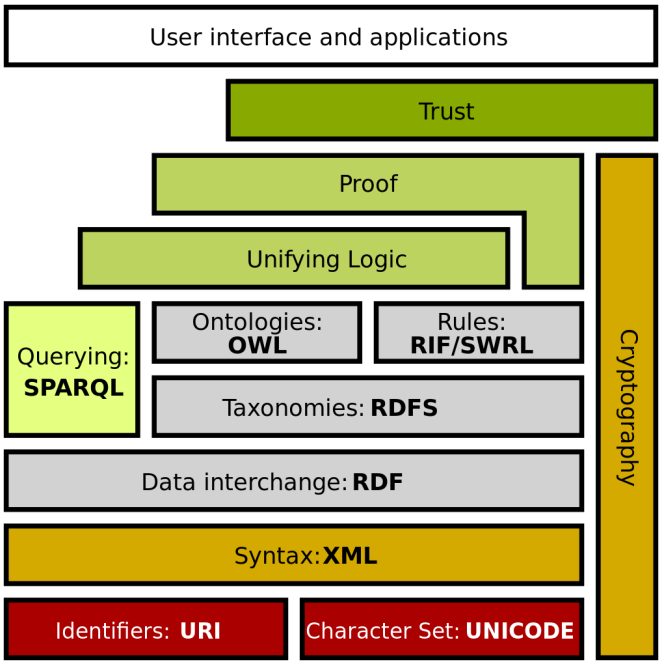


Abbildung 1.1.: Semantic-Web-Stack [18]

Mit Hilfe dieser Protokolle und Technologien ist es im Semantic Web möglich nach ganzen Sätzen zu suchen. Es kann z. B. gefragt werden „Wer hat das Telefon erfunden“ oder „Wie alt ist Merkel“. [19]

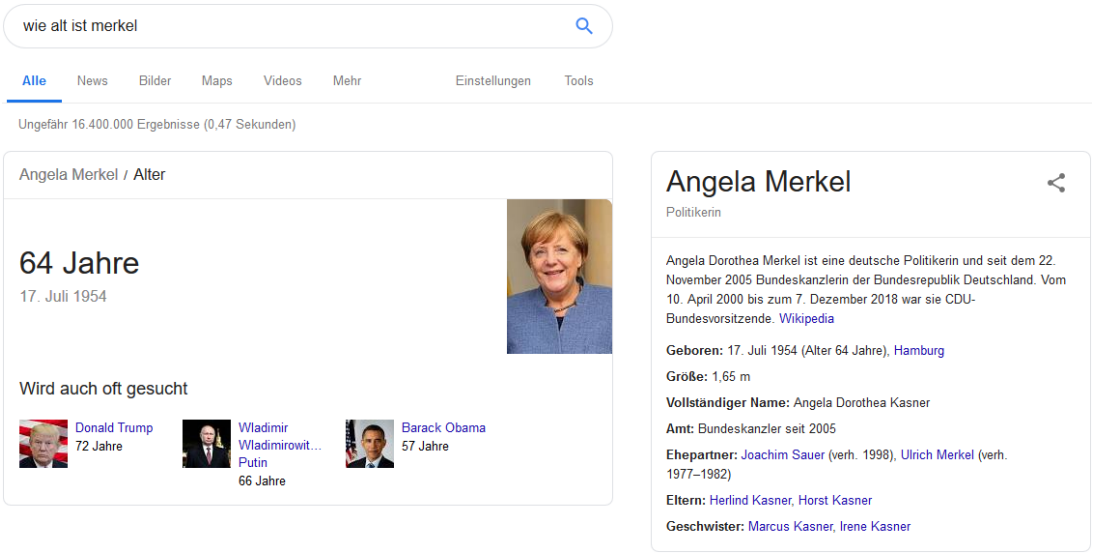


Abbildung 1.2.: Sinnvolle Antwort auf die Frage „Wie alt ist Merkel“

Mit diesem Abschnitt sollte die Entstehung und Bedeutung des Semantic Web erklärt werden. Der Zusammenhang zum Thema der Bachelorarbeit lässt sich zusammenfassend so darstellen:

$$\text{Semantic Web} \rightsquigarrow \text{Semantic Search} \rightsquigarrow \text{Knowledge-Graph} \rightsquigarrow \text{Subgraphisomorphie}$$

1.4. Aufbau der Arbeit

Zunächst wird in Kapitel 2 ein Überblick über die wichtigsten Begriffe gegeben und das notwendige mathematische und graphentheoretische Wissen erarbeitet, auf dem die späteren Modelle und Lösungen beruhen. Dazu gehören graphentheoretische Grundlagen wie gerichtete und ungerichtete Graphen, Subgraphen und der Knowledge-Graph, sowie die anschauliche Erklärung von Graph- und Subgraphisomorphismus. Außerdem werden die wichtigsten Komplexitätsbegriffe eingeführt.

Im nachfolgenden Kapitel 3 wird kurz erläutert was SCAIView ist und worin der Zusammenhang zum Microservice `Subgraph` besteht. Es wird das konkrete Problem dieser Bachelorarbeit und das Vorgehen zur Lösung beschrieben, sowie die Anforderungen an das Programm und die Komplexität des Problems.

In Kapitel 4 werden Algorithmen zur Lösung des Subgraph-Problems vorgestellt. Im Vordergrund stehen zwei exakte Algorithmen: Einer der beiden ist ein Brute-Force Algorithmus, welcher sukzessive alle möglichen Subgraphen ausprobiert, der andere ist ein Backtracking Algorithmus, der den Subgraph schrittweise mit dem Knowledge-Graph abgleicht.

In Kapitel 5 werden die implementierten Algorithmen aus Kapitel 4 analysiert. Dazu werden die Testumgebungen beschrieben und exemplarisch gezeigt, dass die Algorithmen korrekt funktionieren. Im Anschluss werden die Algorithmen auf den Knowledge-Graphen angewendet und miteinander verglichen.

Im letzten Kapitel „Fazit“ wird überprüft, ob die Zielsetzung der Bachelorarbeit erfüllt wurde. Es wird bewertet, ob die Algorithmen auch in der Praxis sinnvoll auf den Knowledge-Graphen angewendet werden können. Der Ausblick auf mögliche Verbesserungen und Erweiterungen des Microservice schließt die Bachelorarbeit ab.

2. Grundlagen & Vorüberlegungen

Dieses Kapitel behandelt die grundlegenden Begriffe dieser Arbeit. Zunächst wird kurz erläutert, was man unter einem *Graphen* versteht und darauf aufbauend, folgt eine nähere Betrachtung von *Knowledge-Graphen*. Anschließend wird darauf eingegangen wie Graphen untereinander verglichen werden können und das Konzept der *Graph- bzw. Subgraph-isomorphie* erläutert, da es für das weitere Verständnis dieser Arbeit benötigt wird. Im letzten Abschnitt wollen wir uns noch mit der *Komplexität* von Problemen befassen, damit wir später das SUBGRAPH-Problem einordnen können.

2.1. Definitionen & Notationen

[In diesem Abschnitt wird sich an 3–6, orientiert]

2.1.1. Graphen

Verknüpfte Daten sind heutzutage überall zu finden. Beispiele hierfür sind unter anderem Straßen- und Logistiknetze sowie Transport- oder Kommunikationssysteme. Das World-Wide-Web verfügt ebenfalls über eine Vielzahl miteinander verbundener Datensätze, wie z. B. das soziale Netzwerk von Facebook oder die Suchmaschine Google. Aber auch biologische Netzwerke, Epedemie- bzw. Seuchennetzwerke oder chemische Netzwerke sind passende Beispiele. Alle diese Systeme können als *Graph* modelliert werden. Generell werden Graphen in der Informatik häufig zur Modellierung verwendet. Sie sind zugleich anschaulich und gut abstrahierbar. Wir geben nun eine Einführung in die grundlegenden Begriffe und Eigenschaften von Graphen.

Definition 2.1 (Graph):
Ein **Graph** G *ist ein Paar* $G = (V, E)$ *mit* $E \subseteq V \times V$. *Dabei ist* V *eine Menge von* **Knoten** (engl. *vertices*) *und* E *eine Menge von* **Kanten** (engl. *edges*), *welche jeweils zwei Knoten miteinander verbinden.*

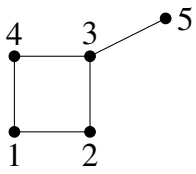
Bemerkung 2.1:
Um Verwechslungen bei mehreren Graphen zu vermeiden, versehen wir V und E von $G = (V, E)$ oft mit dem „Namen“ des Graphen im Index, schreiben also V_G und E_G.

Definition 2.2 (ungerichteter Graph):
Wir nennen $G = (V, E)$ *einen* **ungerichteten Graph**, *wobei*

$$E \subseteq \big\{\{u, v\} : u, v \in V,\ u \neq v\big\}$$

die Menge der **ungerichteten Kanten** *ist, also* $\{u, v\} = \{v, u\}$ *gilt.*

Beispiel 2.1 (ungerichteter Graph):



$$V = \{1,2,3,4,5\}$$

$$E = \big\{\{1,2\},\{2,3\},\{3,4\},\{4,1\},\{3,5\}\big\}$$

Bemerkung 2.2:
Zwei Knoten, die durch eine Kante verbunden sind, oder zwei Kanten, die einen gemeinsamen Knoten besitzen, nennt man **benachbart** oder **adjazent**. Gehört ein Knoten zu einer Kante, so nennen wir die beiden **inzident**.

Zwei Kanten $e, f \in E$ heißen **parallel**, wenn $e = \{u,v\} = f$. Wir sprechen dann auch von einer **Mehrfachkante**. Eine Kante $e \in E$ mit $e = \{u,u\}$ heißt **Schlinge**.

Graphen ohne Mehrfachkanten und Schlingen heißen **einfach**.

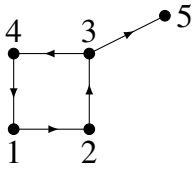
Ein Graph heißt **endlich**, wenn die Knotenmenge V endlich ist, sonst spricht man von einem **unendlichen Graph**.

Definition 2.3 (gerichteter Graph):
Wir nennen $D = (V, A)$ einen **gerichteten Graph** *oder* **Digraph**, *wobei*

$$A \subseteq \big\{(u,v) \colon u,v \in V,\ u \neq v\big\}$$

die Menge der **gerichteten Kanten** *ist, also $(u,v) \neq (v,u)$ gilt.*

Beispiel 2.2 (gerichteter Graph):



$$V = \{1,2,3,4,5\}$$

$$E = \big\{(1,2),(2,3),(3,4),(4,1),(3,5)\big\}$$

Oft werden den Knoten und Kanten noch weitere Eigenschaften zugeschrieben, weshalb wir auch den *attributierten Graphen* einführen wollen.

Definition 2.4 (attributierter Graph):
Ein **gelabelter** *oder* **attributierter Graph** *ist ein Tripel $G = (V, E, \ell)$, mit einer Menge von Knoten V, einer Menge von Kanten $E \subseteq V \times V$ und einer Funktion $\ell \colon V \cup E \to L$, welche den Knoten und Kanten Attribute aus der Menge L zuweist.*

Bemerkung 2.3:
Im Rahmen dieser Arbeit werden wir uns auf die Betrachtung von Graphen beschränken, welche *endlich*, *einfach*, *gerichtet* und *attributiert* sind.

Definition 2.5 (Subgraph & Supergraph):
Sei G ein Graph. Der Graph H heißt **Subgraph**, **Untergraph** *oder* **Teilgraph** *von G* (Notation: $H \subset G$), *falls gilt*:

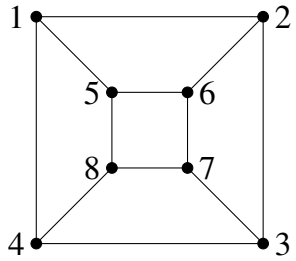
$$V_H \subset V_G \qquad \text{und} \qquad E_H \subset E_G.$$

H entsteht aus G durch entfernen der Knoten in $V_G \setminus V_H$ und der Kanten in $E_G \setminus E_H$. G **enthält** *H und G ist* **Supergraph** *oder* **Obergraph** *von H.*

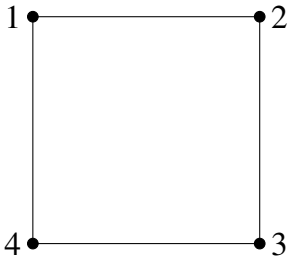
H heißt **aufspannender Subgraph** *von G, falls $V_H = V_G$, d. h. wenn aus G nur Kanten und keine Knoten entfernt werden.*

Beispiel 2.3 (Subgraph):
Sei G ein Graph mit $V_G = \{1,2,3,4,5,6,7,8\}$ und
$E_G = \big\{\{1,2\},\{1,4\},\{1,5\},\{2,3\},\{2,6\},\{3,4\},\{3,7\},\{4,8\},\{5,6\},\{5,8\},\{6,7\},\{7,8\}\big\}$.

Dann bildet $H \subset G$ einen Subgraph mit $V_H = \{1,2,3,4\} \subset V_G$ und
$E_H = \big\{\{1,2\},\{1,4\},\{2,3\},\{3,4\}\big\} \subset E_G$.



G



H

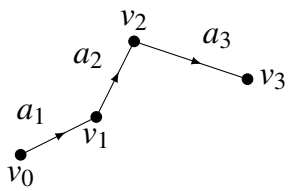
Definition 2.6 (Kantenfolge, Weg & Länge):
Sei $D = (V, A)$ ein Digraph. Dann nennen wir $P = (v_0, a_1, v_1, a_2, \ldots, a_m, v_m)$ eine **Kantenfolge**, *wenn $v_0, \ldots, v_m \in V$, $a_1, \ldots, a_m \in A$ mit $a_i = (v_{i-1}, v_i)$. v_0 bezeichnen wir dabei als* **Startknoten** *und v_m als* **Endknoten** *von P.*

Sei $l\colon A \to \mathbb{Z}_{\geq 0}$ eine **Längenfunktion**, *dann ist die* **Länge** *von P definiert als*

$$l(P) = \sum_{i=1}^{m} l(a_i).$$

Wenn $|\{v_0, \ldots, v_m\}| = m + 1$, wobei $v_i \neq v_j$ für $i \neq j$, dann heißt P **v_0-v_m-Weg** *oder kurz* **Weg**.

Beispiel 2.4 (Kantenfolge & Weg):



Die Kantenfolge $(v_0, a_1, v_1, a_2, v_2, a_3, v_3)$ ist auch ein v_0-v_3-Weg.

Bemerkung 2.4:
Mit $|\cdot|$ ist die **Mächtigkeit** der Menge gemeint. Bei einer endlichen Menge X bezeichnet die Mächtigkeit die Anzahl der Elemente von X.

Bemerkung 2.5:
$u, v \in V$ heißen **wegzusammenhängend**, falls ein u-v-Weg existiert. Die Relation „wegzusammenhängend“ ist eine Äquivalenzrelation und ihre Äquivalenzklassen heißen **Zusammenhangskomponenten**. G heißt **zusammenhängend**, wenn $\forall s, t \in V$ mit $s \neq t$ ein s-t-Weg existiert, d. h. nur eine Zusammenhangskomponente existiert.

2.1.2. Knowledge-Graphen

Unter dem Begriff *Knowledge-Graph* versteht man ein System, das wichtige Informationen sucht und miteinander verknüpft. Man spricht von einer Datenbank, die häufig gesuchte Keywords auf Grundlage des bereits vorhandenen Contents sammelt. Mit dem Knowledge-Graph werden z. B. Informationen über Menschen oder Orte gesammelt, die auf die eine oder andere Weise miteinander verbunden sind. In der Regel ist mit der Bezeichnung „Knowledge-Graph“ das Einführen einer solchen Systematik wie bei Google oder Facebook gemeint. [20]

Definition 2.7 (Entität):
Als **Entität** (*auch Informationsobjekt genannt,* engl. *entity*) *wird in der Datenmodellierung ein eindeutig zu bestimmendes Objekt bezeichnet, über das Informationen gespeichert oder verarbeitet werden sollen.*

Die Entität wird durch ihre Attribute bzw. Eigenschaften bestimmt. Die bei einer bestimmten Entität auftretenden Werte sind Attributwerte.

Bemerkung 2.6:
Eine Entität kann materiell oder immateriell, konkret oder abstrakt sein.

Beispiel 2.5:

1. Jedes beliebige, erkennbare Konzept kann eine Entität sein. In Java bspw. eine Klasse.
2. Bei der Datenmodellierung sind Entitäten alle Dinge, die für die Datenbank relevant sind. Eine Entität entspricht bspw. einer Tabelle im relationalen Modell.

Definition 2.8 (Knowledge-Graph):
Unter dem Begriff **Knowledge-Graph** (*im Folgenden mit $\mathcal{G}$ notiert*) *versteht man zunächst ganz allgemein eine Systematik, mit der Informationen gesucht und miteinander verknüpft werden.*

Konkrete aufbereitete und kompilierte Suchergebnisse zu bestimmten Themengebieten und Entitäten bilden dann den „Knowledge-Graph“. Je nach Anwendung gibt es Unterschiede hinsichtlich Komplexität und Art des Knowledge-Graphen.

Mit Hilfe des Knowledge-Graph von Google werden neben den Suchergebnissen relevante Informationen zu der entsprechenden Anfrage zusammengefügt. Auf diese Weise bietet Google dem Nutzer ausreichend Information zu seiner Anfrage und ein weiterer Webseitenbesuch wird ggf. überflüssig, da Google die möglicherweise gewünschte Information schon mit anzeigt. [20]

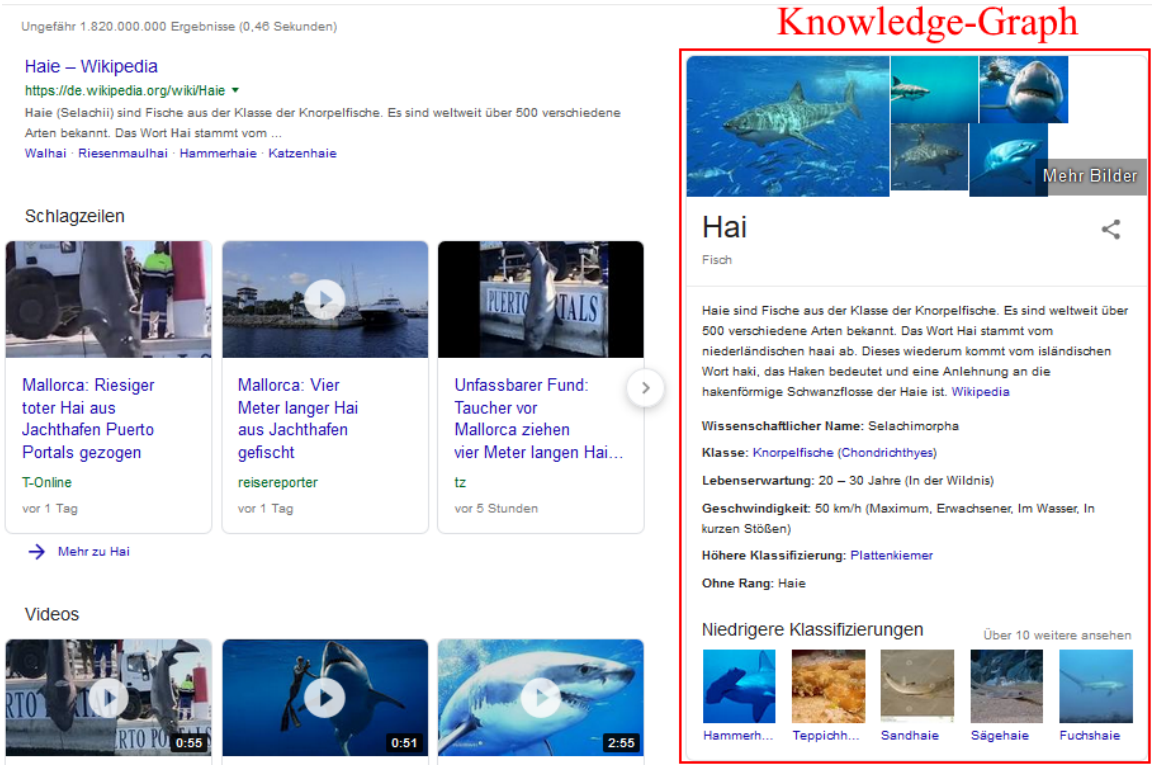


Abbildung 2.1.: Der Knowledge-Graph von Google – rechts neben den Standardsuchergebnissen werden die Kerninformationen zum Suchbegriff „Hai“ aufgelistet.

2.1.3. Isomorphismen

Im Folgenden wollen wir uns mit der Struktur von Graphen und deren Vergleichbarkeit beschäftigen. Dazu benutzen wir *Isomorphismen*, konkret Graph- und Subgraphisomorphismen. In der Graphentheorie ist ein Graph- bzw. Subgraphisomorphismus eine Bijektion zwischen den Knotenmengen zweier Graphen und beschreibt die Eigenschaft, ob zwei Graphen die gleiche Struktur besitzen.

Definition 2.9 (Graphisomorphismus):
Zwei Graphen G und H heißen **isomorph** *(Notation: $G \cong H$), wenn eine Bijektion $\varphi\colon V_G \to V_H$ existiert mit*

$$(u,v) \in E_G \iff \big(\varphi(u),\varphi(v)\big) \in E_H.$$

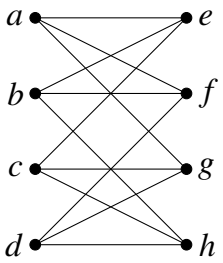
Die Abbildung φ heißt dann **Isomorphismus** *zwischen G und H.*

Falls $V_G = V_H$, dann nennen wir die Abbildung φ **Automorphismus**.

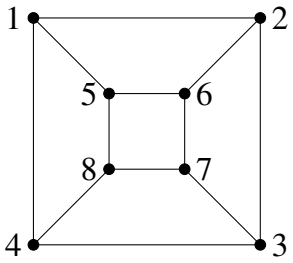
Isomorphe Graphen besitzen dieselben Strukturen und unterscheiden sich nur in der Benennung von Knoten und Kanten. Ein solcher Graph repräsentiert eine Äquivalenzklasse isomorpher Graphen.

Beispiel 2.6 (Graphisomorphie):
Sei G ein Graph mit $V_G = \{a, b, c, d, e, f, g, h\}$ und
$E_G = \big\{\{a,e\},\{a,f\},\{a,g\},\{b,e\},\{b,f\},\{b,h\},\{c,e\},\{c,g\},\{c,h\},\{d,f\},\{d,g\},\{d,h\}\big\}$.

Und sei H ein Graph mit $V_H = \{1, 2, 3, 4, 5, 6, 7, 8\}$ und
$E_H = \big\{\{1,2\},\{1,4\},\{1,5\},\{2,3\},\{2,6\},\{3,4\},\{3,7\},\{4,8\},\{5,6\},\{5,8\},\{6,7\},\{7,8\}\big\}$.



G

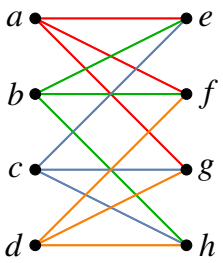


H

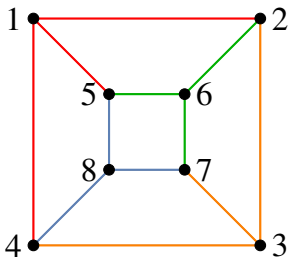
Weil ein Isomorphismus $\varphi\colon V_G \to V_H$ existiert, mit

$$a \mapsto 1$$
$$b \mapsto 6$$
$$c \mapsto 8$$
$$d \mapsto 3$$
$$e \mapsto 5$$
$$f \mapsto 2$$
$$g \mapsto 4$$
$$h \mapsto 7 \text{ gilt:}$$

$$\{\{a,e\},\{a,f\},\{a,g\}\} \xrightarrow{\varphi} \{\{1,5\},\{1,2\},\{1,4\}\}$$
$$\{\{b,e\},\{b,f\},\{b,h\}\} \xrightarrow{\varphi} \{\{6,5\},\{6,2\},\{6,7\}\}$$
$$\{\{c,e\},\{c,g\},\{c,h\}\} \xrightarrow{\varphi} \{\{8,5\},\{8,4\},\{8,7\}\}$$
$$\{\{d,f\},\{d,g\},\{d,h\}\} \xrightarrow{\varphi} \{\{3,2\},\{3,4\},\{3,7\}\}.$$



G



H

Also sind G und H isomorph: $G \cong H$.

Definition 2.10 (Subgraphisomorphismus):
Seien G und H zwei Graphen und sei $I \subset G$ ein Subgraph von G mit $V_I \subseteq V_G$ und $E_I \subseteq E_G \cap (V_I \times V_I)$. Ein **Subgraphisomorphismus** *von I nach H ist eine Bijektion*

$$\varphi \colon V_I \to V_H$$

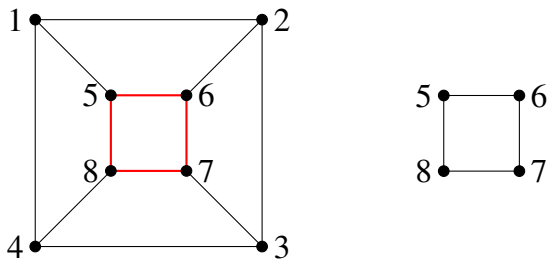
mit der Eigenschaft, dass aus $(u,v) \in E_I$ auch $\big(\varphi(u), \varphi(v)\big) \in E_H$ folgt. Wenn ein solcher Subgraphisomorphismus zwischen zwei Graphen H und $I \subset G$ existiert, dann nennen wir die Graphen H und I **isomorph** *und schreiben $I \simeq H$.*

Beispiel 2.7 (Subgraphisomorphie):
Seien G und H so definiert wie im Beispiel 2.6 zur Graphisomorphie und sei I der Graph mit $V_I = \{5,6,7,8\}$ und $E_I = \big\{\{5,6\},\{5,8\},\{6,7\},\{7,8\}\big\}$.

1. Weil $V_I \subset V_H$ und $E_I \subset E_H$ gilt, sieht man leicht, dass $I \subset H$ und $I \simeq H$ gilt. Wobei der zugehörige Subgraphisomorphismus

$$\begin{aligned} \varphi \colon V_I &\to V_H \\ v &\mapsto \mathrm{id}(v) \text{ ist.} \end{aligned}$$

   Also sind I und H isomorph: $I \simeq H$.

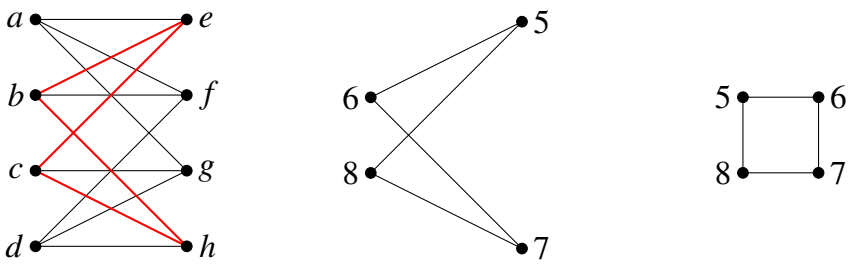


H $\qquad$ I

2. Um zu zeigen, dass $I \simeq G$ gilt müssen wir einen Subgraphisomorphismus φ finden, der $(u,v) \in E_I$ auf $\big(\varphi(u), \varphi(v)\big) \in E_G$ abbildet. Mit

$$\begin{aligned} \varphi \colon V_I &\to V_G \\ 5 &\mapsto e \\ 6 &\mapsto b \\ 7 &\mapsto h \\ 8 &\mapsto c \text{ gilt:} \end{aligned}$$

$$\big\{\{6,5\},\{6,7\}\big\} \xrightarrow{\varphi} \big\{\{b,e\},\{b,h\}\big\}$$
$$\big\{\{8,5\},\{8,7\}\big\} \xrightarrow{\varphi} \big\{\{c,e\},\{c,h\}\big\}.$$



G $\qquad$ I $\qquad$ I

Wir haben also ein zulässiges φ gefunden und damit sind I und G isomorph: $I \simeq G$.

2.1.4. Komplexitätsklassen

Die Komplexitätstheorie, als Teilgebiet der *theoretischen Informatik*, befasst sich mit der Komplexität algorithmisch behandelbarer Probleme auf verschiedenen formalen Rechnermodellen. Die Komplexität von Algorithmen wird in deren Ressourcenverbrauch gemessen, meist *Rechenzeit* oder *Speicherplatzbedarf*. Die Komplexität eines Problems ist wiederum die Komplexität desjenigen Algorithmus, der das Problem mit dem geringstmöglichen Ressourcenverbrauch löst. [7]

Zwei der wichtigsten Komplexitätsklassen sind $\mathcal{P}$ und $\mathcal{NP}$, welche wir nun formal einführen wollen.

Die Komplexitätsklasse $\mathcal{NP}$ wurde als erste systematisch auf vollständige Probleme untersucht. Inzwischen sind unzählige algorithmische Problemstellungen als $\mathcal{NP}$-vollständig klassifiziert worden, von denen viele praktische Bedeutung besitzen. [8]

Definition 2.11 (Komplexitätsklasse $\mathcal{P}$):
Die **Komplexitätsklasse $\mathcal{P}$** *ist die Menge der Sprachen, für die es eine deterministische Turing-Maschine gibt, die in höchstens $p(|x|)$ Schritten entscheiden kann, ob $x \in L$. Dabei ist p ein beliebiges Polynom. Also*

$$\mathcal{P} = \bigcup_{k \in \mathbb{N}_0} \mathtt{TIME}(n^k).$$

Bemerkung 2.7:
Komplexitätsklassen sind Sprachfamilien, also Mengen von Sprachen. Das bedeutet, dass man, streng genommen, nur bei Entscheidungsproblemen über ihre Zugehörigkeit zu einer bestimmten Komplexitätsklasse sprechen kann. Such-, Optimierungs- und Optimalwertprobleme lassen sich jedoch auf Entscheidungsprobleme zurückführen.

Definition 2.12 (Komplexitätsklasse $\mathcal{NP}$):
Die **Komplexitätsklasse $\mathcal{NP}$** *ist die Klasse aller Sprachen, die von nichtdeterministischen Turing-Maschinen in polynomieller Zeit akzeptiert werden. Also*

$$\mathcal{NP} = \bigcup_{k \in \mathbb{N}_0} \mathtt{NTIME}(n^k).$$

Bemerkung 2.8:
Nichtdeterministische Turing-Maschinen sind mächtiger als deterministische Turing-Maschinen, was in $\mathcal{P} \subset \mathcal{NP}$ resultiert.

Definition 2.13 ($\mathcal{NP}$-vollständig):
Eine Sprache S ist **$\mathcal{NP}$-vollständig**, *wenn*:

1. $S \in \mathcal{NP}$ *und*
2. S **$\mathcal{NP}$-schwer** *ist, d. h. für alle $L \in \mathcal{NP}$: $L \preceq S$.*

Bemerkung 2.9:
Eine Sprache S ist also $\mathcal{NP}$-vollständig, wenn für alle anderen Sprachen $L \in \mathcal{NP}\colon L \preceq S$ gilt. Das bedeutet, dass sich alle anderen Sprachen **polynomiell auf S reduzieren** lassen.

2.2. Erste Überlegungen

2.2.1. Invarianten

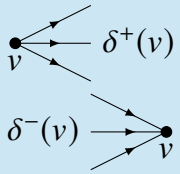
Es gibt verschiedene Möglichkeiten zwei Graphen auf Isomorphie zu untersuchen. Eine Möglichkeit ist es mit *Invarianten* zu arbeiten, d. h. mit Größen, die bei isomorphen Graphen übereinstimmen. Drei erwähnenswerte Invarianten sind die Anzahl der Knoten, die Anzahl der Kanten und die „Verteilung“ der Knotengrade. Stimmen zwei Graphen nicht in allen diesen Größen überein, können sie nicht isomorph sein.

Definition 2.14 (Knotengrad für gerichtete Graphen):
Sei $D = (V, A)$ ein gerichteter Graph. Für $U \subseteq V$ sei

$$\delta^+(U) = \{(v,w) \in A\colon v \in U,\ w \notin U\}$$
$$\delta^-(U) = \{(v,w) \in A\colon v \notin U,\ w \in U\}.$$

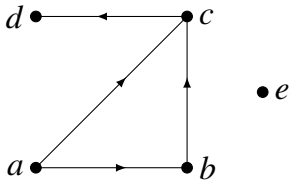
Für einen Knoten $v \in V$ schreiben wir verkürzt

$$\delta^+(v) = \delta^+(\{v\}) \qquad \delta^-(v) = \delta^-(\{v\}).$$



Ferner sei $d^+(v) = |\delta^+(v)|$ der **Außengrad**,
$d^-(v) = |\delta^-(v)|$ *der* **Innengrad** *und*
$d(v) = d^+(v) + d^-(v)$ *der* **Grad**.

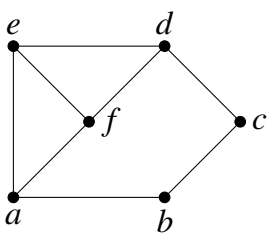
Beispiel 2.8 (Knotengrade):



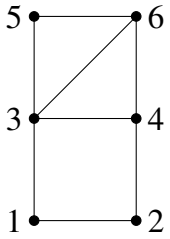
Knoten v	Grad $d(v)$
a	$2 + 0 = 2$
b	$1 + 1 = 2$
c	$1 + 2 = 3$
d	$0 + 1 = 1$
e	$0 + 0 = 0$

Bemerkung 2.10:
Ein Graph, bei dem alle Knoten den konstanten Grad k haben, heißt ***k*-regulär**. Einen Knoten vom Grad 0 nennen wir **isoliert**.

Beispiel 2.9 (Graphisomorphie mit Invarianten):



G



H

Man sieht sofort, dass $G \ncong H$. Eine mögliche Begründung warum G und H nicht isomorph sind ist, dass im rechten Graphen H der Knotengrad 4 vorkommt, und im linken Graphen G nicht.

Graph-Invariante	G	H
Knoten	6	6
Kanten	8	8
Knotengerade	2-mal 2	3-mal 2
	4-mal 3	2-mal 3
	0-mal 4	1-mal 4

2.2.2. Diameter

Ein Maß für die Abschätzung, ob überhaupt ein Subgraph existieren kann, ist der *Diameter* des Graphen.

Definition 2.15 (Abstand):
Sei $D = (V, A)$ ein Digraph und seien $s, t \in V$. Der **Abstand** $\operatorname{dist}(s,t)$ *von s zu t ist*

$$\operatorname{dist}(s,t) = \min l(P) \text{ mit } P \text{ } s\text{-}t\text{-Weg}.$$

Definition 2.16 (Exzentrizität):
Die **Exzentrizität $\varepsilon(v)$** *eines Knotens v ist der maximale Abstand zu allen anderen Knoten des Graphen*

$$\varepsilon(v) = \max_{u \in V} \operatorname{dist}(v,u).$$

Bemerkung 2.11:
Man beachte, dass in gerichteten Graphen der Abstand von der Richtung des Weges abhängt. Insbesondere kann es sein, dass nur in eine Richtung ein gerichteter Weg existiert.

Definition 2.17 (Diameter):
Den größten Abstand zwischen zwei Knoten in einem Graphen G nennt man den **Durchmesser** *oder* **Diameter** *des Graphen und bezeichnet ihn mit $D(G)$. Der Durchmesser ist also das Maximum aller Exzentrizitäten der Knoten in G und $D(G)$ ist damit die größte Distanz zwischen allen Knotenpaaren*

$$D(G) = \max_{v \in V} \varepsilon(v).$$

Für einen nicht zusammenhängenden Graphen definieren wir $D(G) = \infty$.

2.2.3. Algorithmen

Es gibt verschiedene algorithmische Ansätze das Subgraph-Problem zu lösen. Die möglichen Algorithmen lassen sich grob in zwei Klassen unterteilen, einmal in *exakte Algorithmen* und einmal in *approximative Algorithmen*. Da nach genauen Übereinstimmungen gesucht werden soll, werden exakte Algorithmen verwendet.

Für Testzwecke wurde sich für die Implementierung eines *Brute-Force Algorithmus* entschieden. Für das Lösen des Subgraph-Problems auf dem Knowledge-Graph wurde ein *Backtracking Algorithmus*, in zwei unterschiedlichen Varianten, implementiert.

Für weitere Beispiele exakter Algorithmen oder Beispiele für approximative Algorithmen sei an dieser Stelle auf [9] verwiesen.

Generell lassen sich Algorithmen durch Heuristiken verbessern.

Definition 2.18 (Heuristik):
In der Optimierung wird unter einer **Heuristik** *eine methodische Anleitung verstanden, welche versucht, mit begrenztem Wissen das Optimum des Optimierungsproblems zu finden oder zumindest eine Verbesserung zu erreichen. Heuristiken werden in der Optimierung immer dann eingesetzt, wenn rein mathematische Verfahren nicht eingesetzt werden können oder ineffektiv sind.*

Bemerkung 2.12:
Topologische Optimierung von Knowledge-Graph-Strukturen ist ein Feld, in dem Heuristiken Anwendung finden. Generell basieren viele Verfahren der Optimierung teilweise oder ganz auf Heuristiken.

3. Problemstellung

Heutzutage werden täglich Milliarden von Suchanfragen im Internet gestellt. Alleine Google verzeichnet seit 2016 jährlich über 3 Billionen Suchanfragen. Jeder Nutzer sucht durchschnittlich 3.4 Mal pro Tag nach etwas. [21]

Das Problem nach etwas Bestimmtem zu suchen ist also allgegenwärtig. Dabei ist es wichtig, dass Suchmaschinen nicht nur die eingegebenen Schlüsselwörter finden und als Treffer markieren, sondern selber versuchen den „Sinn“ der Frage zu verstehen um dadurch gezielter bzw. relevanteren Inhalt anzeigen zu können. Das führt uns zum Begriff der *semantischen Suche*, welche die Bedeutung einer Suchanfrage in den Mittelpunkt stellt.

Bei keywordbasierten Suchmaschinen wird nur nach „zusammenhanglosen“ Wörtern gesucht. Durch die Verwendung von Hintergrundwissen wird bei einer semantischen Suche die inhaltliche Bedeutung der Suchanfrage berücksichtigt, wodurch die Suchanfrage präziser erfasst und mit den inhaltlich relevanten Texten in Verbindung gebracht werden kann. Somit werden inhaltlich „sinnvolle“ Suchergebnisse bereitgestellt. Die semantische Suche imitiert gewissermaßen das menschliche Gehirn, indem Wissen und Assoziationen zur Suche genutzt werden. [22]

3.1. SCAIView

SCAIView ist eine semantische Suchmaschine zur Wissensfindung und zum Wissensabruf. Es ist ein Informationsabrufsystem, das semantische Suchen in großen Textsammlungen, durch die Kombination von Freitexten und ontologischen Darstellungen von automatisch erkannten biologischen Entitäten, ermöglicht. Es wurde entwickelt, um wissenschaftliche Literatur besser zugänglich zu machen. SCAIView ist ein microservicebasiertes[1] System. Dadurch lassen sich einfach Microservices in die Verarbeitungspipeline hinzufügen oder aus dieser entfernen. So auch der Microservice `Subgraph` der im Rahmen dieser Bachelorarbeit implementiert wurde. Die Benutzeroberfläche von SCAIView ist ein webbasierter Microservice, der auf Apache Tomcat läuft und über ReST-API-Aufrufe[2] mit dem Backend kommuniziert. API-Aufrufe[3] werden über den Nachrichtenbroker Apache ActiveMQ an SCAIView übermittelt. [10]

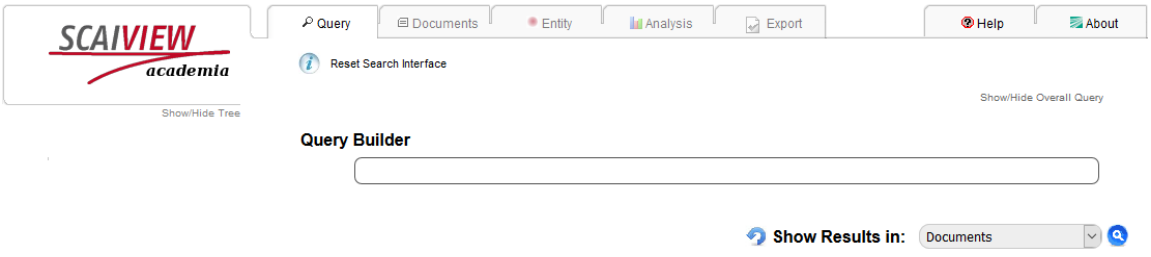


Abbildung 3.1.: SCAIView-Webseite: `http://academia.scaiview.com/`

[1] Microservice – unabhängiger, entkoppelter Dienst für kleinere Aufgaben
[2] ReST: Representational State Transfer
[3] API: Application Programming Interface – Anwendungsschnittstelle

Im Kern erfüllt SCAIView folgende Aufgaben:

- Bereitstellen von Daten
- Abrufen von Daten
- Verarbeitung von Daten

Der `Subgraph`-Microservice kommuniziert innerhalb von SCAIView nur mit der API und der Neo4j[1]-Graphdatenbank. Die über die GUI[2] eingegebene Suchanfrage wird von der API im JSON-Format[3] an `Subgraph` übergeben. Die daraus entstehende Cypher-Query wird direkt, ungefiltert und ohne Zwischenlayer, in Neo4j abgefragt. Diese Direktverbindung zur Graphdatenbank ist eher unüblich und stellt eine Besonderheit dar. Das entstandene Neo4j-Result-Set wird an `Subgraph` zurückgegeben, dort in das JSON-Format konvertiert und weiter an die API gereicht. Von dort können die Suchergebnisse nun an die GUI weitergegeben und angezeigt werden.

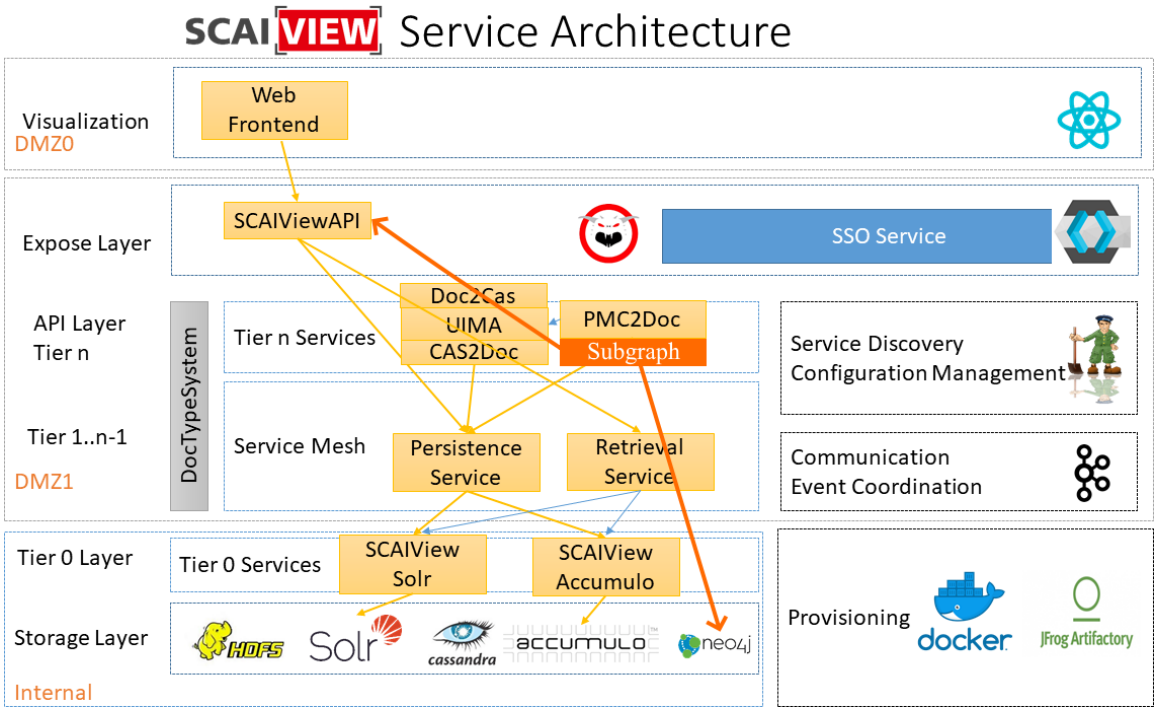


Abbildung 3.2.: SCAIView-Architektur [11]

3.2. Konkretes Problem

In unserem Fall werden keine allgemeinen Suchanfragen gestellt, sondern es wird spezifisch nach biomedizinischem Kontext gesucht. Das können z. B. Autoren, Dokumente, Medikamente, deren Wirkung oder Zusammensetzung, oder beliebige MeSHTerms[4] sein. Dazu wurde ein Knowledge-Graph aus verschiedenen medizinischen Datenbanken aufgebaut.

Als Graphdatenbank, welche die Informationen aus diesen Datenbanken zusammenführt, wurde Neo4j gewählt. In diese Neo4j-Graphdatenbank sind alle Informationen der medizinischen Datenbanken eingelesen worden.

[1] *siehe* Abschnitt 3.3.1: Neo4j & Cypher
[2] GUI: Graphical User Interface – Grafische Benutzeroberfläche
[3] JSON: JavaScript Object Notation – *siehe* Abschnitt 3.3.2: Format
[4] MeSH: Medical Subject Headings

Die wichtigsten Quellen zum Aufbau des Knowledge-Graph $\mathcal{G}$ waren dabei:

- Arztberichte
- Biomedizinische Literatur
- Patente

Die Datenbanken, deren Informationen bisher in $\mathcal{G}$ gespeichert wurden, sind:

- PubMed
- PubMed central
- MeSH hierarchy
- Gene Ontology

Bemerkung 3.1:
Der Knowledge-Graph kann als eine weitere Etappe auf dem Weg zu einer semantischen Suche interpretiert werden.

3.2.1. Eingabe

Im Allgemeinen stellen Nutzer Anfragen in Textform. Da eine beliebig gestellte Frage so aber über wenig Struktur verfügt, wird der Einfachheit halber die Eingabe als gerichteter Graph $D = (V, A) \subset \mathcal{G}$ codiert. Dieser besteht aus Knoten und Kanten und wird nach einem bestimmten Format[1] eingegeben.

Die Knoten enthalten die konkreten Daten über das Suchobjekt wie etwa, dass nach dem `Author` mit dem Namen x gesucht wird. Ob es sich beim Namen um den Vor- oder Nachnamen handelt spielt dabei keine Rolle. Sobald mehrere Knoten spezifiziert wurden, können diese mit Kanten verbunden werden, welche ebenfalls mit Eigenschaften versehen werden können. So lässt sich bspw. mit Hilfe der `isAuthor`-Relation klären, welcher Autor welches `Document` verfasst hat.

3.2.2. Ausgabe

Es lassen sich verschiedene Arten von Problemstellungen untersuchen:

- Entscheidungs-Problem:
  Beantworte ob H zu einem Subgraph von G isomorph ist.
- Such-Problem:
  Gib *einen* Subgraph von G zurück, der zu H isomorph ist.
- Zähl-Problem:
  Gibt die Anzahl aller Subgraphen von G zurück, die zu H isomorph sind.
- Aufzählungs-Problem:
  Gibt *alle* Subgraphen von G zurück, die zu H isomorph sind.

[1] *vgl.* Abschnitt 3.3.2: Format

Wir versuchen einmal das *Such-Problem* und einmal das *Aufzählungs-Problem* zu lösen, womit sich natürlich auch die beiden anderen Problemstellungen beantworten lassen. Die API bietet dementsprechend zwei Möglichkeiten zur Suchanfrage. Einmal erhält man mit der Methode `searchGraph` einen Graphen, also den erstbesten Treffer, als Antwort oder mit der Methode `searchAllGraph` alle möglichen Graphen, die zur Suchanfrage passen.

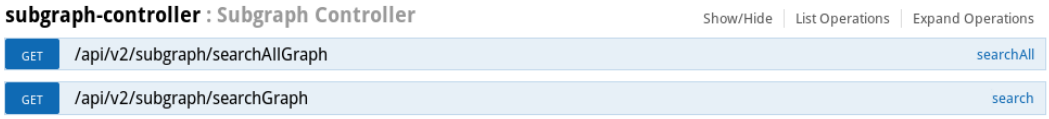


Abbildung 3.3.: SCAIView-API mit beiden Suchmöglichkeiten

Die Ausgabe zur Suchanfrage ist eine Menge von Graphen $\{H : H \simeq D \subset \mathcal{G}\}$ welche zum Eingabegraph D isomorph sind. Analog zur Eingabe werden diese in einem bestimmten Format[1] ausgegeben.

3.2.3. Vorgehen

Um effizient nach etwas suchen zu können, ist es wichtig eine geeignete Datenstruktur zu verwenden. Da sowohl die Eingabe als auch die Ausgabe ein gerichteter Graph ist, wurde auch in Java die *Digraphstruktur* umgesetzt. Es gibt eine Klasse `Vertex`, welche die Knoten repräsentiert und eine Klasse `Edge`, welche die Kanten darstellt. Zusammen werden beide über die Klasse `Digraph` verwaltet.

Die Suche nach dem Eingabegraph D läuft nach folgendem Schema ab:

1. Der JSON-Input(-String) wird in einen Digraph geparst.
2. Aus dem Digraphen wird eine Cypher-Query erstellt.
3. Die Cypher-Query wird in Neo4j abgefragt.
4. Die Ergebnis-Record-List aus Neo4j wird in Digraphen umgewandelt.
5. Die Digraphen werden in das JSON-Format geschrieben.

3.3. Technologien & Anforderungen

3.3.1. Neo4j & Cypher

Neo4j ist eine in Java implementierte Open-Source-*Graphdatenbank*. Die Entwickler beschreiben Neo4j als eine eingebettete, transaktionale Datenbank-Engine, die Daten anstatt in Tabellen in Graphen abspeichert. Neo4j Version 1.0 wurde im Februar 2010 freigegeben. [23]

In Neo4j wird alles entweder als Kante, als Knoten oder als Attribut gespeichert. Jeder Knoten hat eine beliebige Anzahl von Attributen. Knoten und Kanten können eine Beschriftung (Label) tragen. [23]

[1] *vgl.* Abschnitt 3.3.2: Format

Cypher ist eine deklarative Abfragesprache für Graphen, die expressive und effiziente Abfragen und Aktualisierungen auf sogenannten Property-Graphen ermöglicht. Cypher ist eine relativ einfache und dennoch mächtige Sprache. Selbst sehr komplizierte Datenbankabfragen können mit Hilfe von Cypher einfach formuliert werden. [24]

Der Knowledge-Graph $\mathcal{G}$ liegt als Neo4j-Graphdatenbank vor und enthält die in Tabelle 3.1 abgebildeten Eigenschaften.

Tabelle 3.1.: Knoten- und Kanten-Kategorien des Knowledge-Graphen

NodeLabels:

- Affiliation
- Annotation
- Author
- Concept
- ConceptNeo4JImpl
- Document
- Journal
- MeSHTerm
- PublicationType
- Source
- Terminology
- TerminologyNeo4JImpl

RelationshipTypes:

- C_HAS_PARENT
- T_HAS_CONCEPT
- hasAffiliation
- hasAnnotation
- hasDocument
- hasTerm
- isAuthor
- isOfType
- isSource

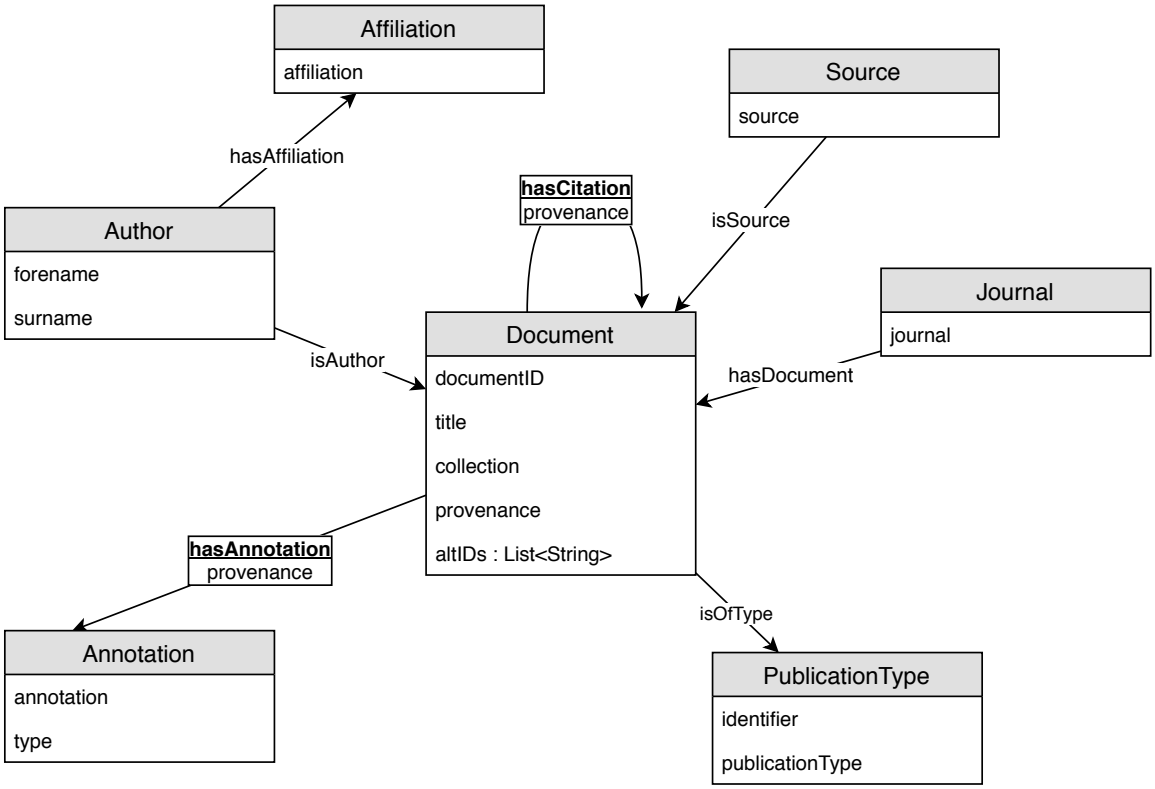


Abbildung 3.4.: Darstellung des Datenmodells der Neo4j-Graphdatenbank [12]

3.3.2. Format

Der Graph $D = (V, A) \subset \mathcal{G}$ wird im JSON-Graph-Format[1] eingegeben. Dabei wurde sich auf das folgende Schema geeinigt:

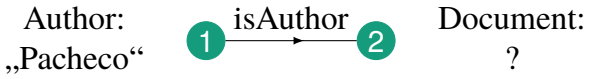
```
{"graph":{
 "nodes":[
  {"id":1,"label":"Knoten 1"},
  {"id":2,"label":"Knoten 2"}],
 "edges":[
  {"from":1,"to":2,"properties":[{...}]}]
 }
}
```

Bemerkung 3.2:
Die verwendeten Knoten-IDs im Format dienen lediglich als Notationshilfe. Es gibt keinen direkten Zusammenhang zu den Graphdatenbank-IDs.

Beispiel 3.1 (Eingabe mit $|V| = 2$ & $|V| = 3$):

1. Möchte man wissen welche Dokumente von „Pacheco“ geschrieben wurden, könnte man folgenden Eingabegraphen verwenden:

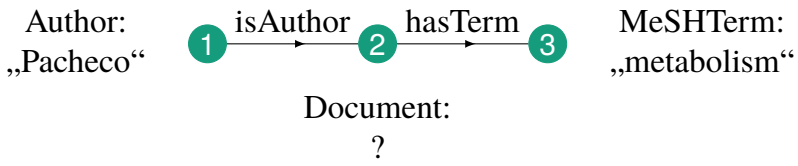
```
{"graph":{
 "nodes":[
  {"id":1,"Author":"Pacheco"},
  {"id":2,"Document":""}],
 "edges":[
  {"from":1,"to":2,"properties":[{"":"isAuthor"}]}]
 }
}
```



2. Möchte man alle Dokumente die von „Pacheco“ geschrieben wurden und die den MeSHTerm „metabolism“ enthalten, könnte man den Eingabegraphen folgendermaßen erweitern:

```
{"graph":{
 "nodes":[
  {"id":1,"Author":"Pacheco"},
  {"id":2,"Document":""},
  {"id":3,"MeSHTerm":"metabolism"}],
 "edges":[
  {"from":1,"to":2,"properties":[{"":"isAuthor"}]},
  {"from":2,"to":3,"properties":[{"":"hasTerm"}]}]
 }
}
```

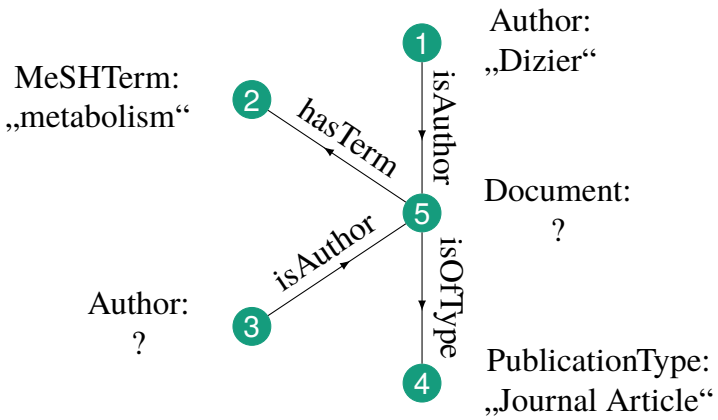
[1] JSON: JavaScript Object Notation – *vgl.* `http://visjs.org/network_examples.html`



Beispiel 3.2 (Eingabe für komplexere Graphen):

1. Fragt man sich wer zusammen mit „Dizier“ einen „Journal Article“ verfasst hat, in dem der MeSHTerm „metabolism“ vorkommt, so könnte man die Frage folgendermaßen codieren:

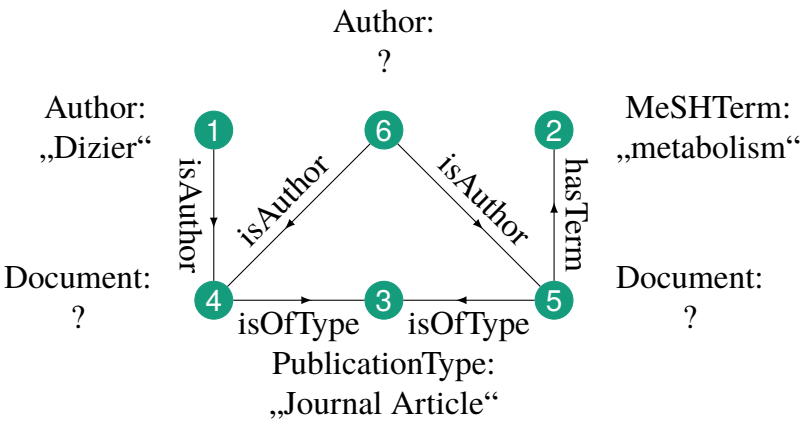
```
{"graph":{
 "nodes":[
  {"id":1,"Author":"Dizier"},
  {"id":2,"MeSHTerm":"metabolism"},
  {"id":4,"PublicationType":"Journal Article"},
  {"id":5,"Document":""},
  {"id":3,"Author":""}],
 "edges":[
  {"from":1,"to":5,"properties":[{"":"isAuthor"}]},
  {"from":3,"to":5,"properties":[{"":"isAuthor"}]},
  {"from":5,"to":2,"properties":[{"":"hasTerm"}]},
  {"from":5,"to":4,"properties":[{"":"isOfType"}]}]
 }
}
```



2. Fragt man sich andererseits wer zusammen mit „Dizier“ einen „Journal Article“ verfasst hat und gleichzeitig Dokumente geschrieben hat, die den MeSHTerm „metabolism“ enthalten, so macht man folgende Eingabe:

```
{"graph":{
 "nodes":[
  {"id":1,"Author":"Dizier"},
  {"id":2,"MeSHTerm":"metabolism"},
  {"id":3,"PublicationType":"Journal Article"},
  {"id":4,"Document":""},
  {"id":5,"Document":""},
  {"id":6,"Author":""}],
 "edges":[
  {"from":1,"to":4,"properties":[{"":"isAuthor"}]},
  {"from":6,"to":4,"properties":[{"":"isAuthor"}]},
  {"from":4,"to":3,"properties":[{"":"isOfType"}]},
```

```
    {"from":6,"to":5,"properties":[{"":"isAuthor"}]},
    {"from":5,"to":2,"properties":[{"":"hasTerm"}]},
    {"from":5,"to":3,"properties":[{"":"isOfType"}]}]
  }
}
```



3.4. Komplexität

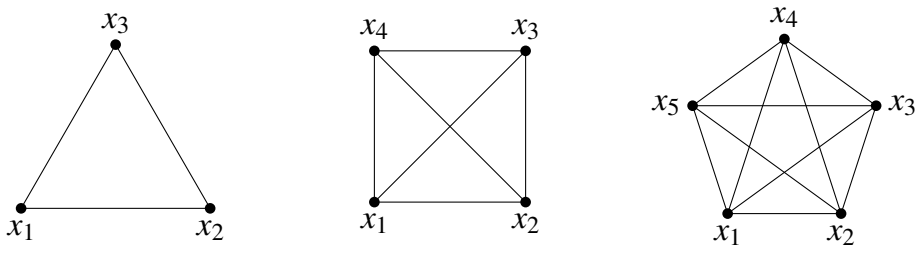
Wir wollen uns kurz in Erinnerung rufen was das Cliquen-Problem ist und wie es in der Entscheidungsvariante lautet.

Definition 3.1 (induzierter Subgraph):
Sei $G = (V, E)$ ein Graph, $W \subset V$ nicht leer und $E(W) = \{(u, v) \in E : u, v \in W\}$. Dann ist $E(W)$ die Menge der Kanten von G, die beide Endknoten in W haben. Wir bezeichnen mit $G(W) = (W, E(W))$ den von W **(knoten)induzierten Subgraphen** *von G.*

Bemerkung 3.3:
Mit $G - W$ notieren wir den induzierten Subgraph $G(V \setminus W)$. $G - W$ ist also der Subgraph, der aus G entsteht, indem wir alle Knoten von W und alle mit W inzidenten Kanten entfernen. Für $G - \{v\}$ schreiben wir kurz $G - v$.

Definition 3.2 (vollständiger Graph):
Ein **vollständiger Graph K_n** *ist ein ungerichteter Graph ohne Mehrfachkanten mit n Knoten und genau $\binom{n}{2} = \frac{n(n-1)}{2}$ Kanten für $n > 1$. In einem vollständigen Graphen ist jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden.*

Beispiel 3.3:



K_3 K_4 K_5

Definition 3.3 (Clique):
Sei G ein ungerichteter Graph ohne Mehrfachkanten und U eine Teilmenge von V. Man bezeichnet U als **Clique** *von G, wenn für je zwei beliebige verschiedene Knoten u und v aus U gilt, dass sie durch eine Kante miteinander verbunden sind.*

Bemerkung 3.4:
Der Einfachheit halber verwenden wir hier ungerichtete Graphen. Die Komplexität lässt sich aber auf gerichtete Graphen übertragen.

Definition 3.4 (Cliquen-Problem):
Gegeben ist ein Graph G und eine natürliche Zahl $n \in \mathbb{N}$. Wir bezeichnen mit ***Clique*** *die Frage danach, ob es eine Clique, also einen vollständigen (Sub-)Graphen, der Größe n in G gibt.*

Es gibt eine Vielzahl $\mathcal{NP}$-vollständiger Graphprobleme, von denen es in dieser Bachelorarbeit speziell um das Subgraph-Problem geht.

Definition 3.5 (Subgraph-Problem):
Gegeben sind zwei Graphen G und H. Wir bezeichnen mit ***Subgraph*** *die Frage danach, ob ein Subgraph von G isomorph zu H ist.*

Die $\mathcal{NP}$-Schwere ist offensichtlich, da es eine Verallgemeinerung des Clique-Problems ist, dort ist H auf vollständige Graphen K_n beschränkt.

Haben beide Graphen die gleiche Anzahl an Knoten, so ergibt sich der Spezialfall des Graphisomorphie-Problems, also die Frage, ob G selbst zu H isomorph ist. Dies ist eines der wenigen Probleme, für die es bislang nicht gelungen ist die $\mathcal{NP}$-Vollständigkeit oder die Zugehörigkeit zu $\mathcal{P}$ zu zeigen. [8]

Lemma 3.1:
Sei S eine $\mathcal{NP}$-vollständige Sprache. Dann ist eine Sprache T $\mathcal{NP}$-vollständig, wenn:

1. *$T \in \mathcal{NP}$ und*
2. *$S \preceq T$.*

Behauptung:
Das Subgraph-Problem ist $\mathcal{NP}$-vollständig.

Beweis (nach Lemma 3.1):

1. Subgraph ist in $\mathcal{NP}$:
   Seien G und H zwei Graphen. Es gibt einen Verifzierer für Subgraph. Hierzu wird die Abbildung $\varphi\colon V_H \to V_G$ geraten. Der Verifizierer überprüft nun für jede Kante $\{u, v\} \in E_H$, ob die Kante $\{\varphi(u), \varphi(v)\} \in E_G$ ist. Die Laufzeit dieses Algorithmus ist höchstens quadratisch in der Eingabelänge und damit polynomiell zeitbeschränkt.
2. Subgraph ist $\mathcal{NP}$-schwer:
   Wir wissen, dass Clique $\mathcal{NP}$-vollständig ist. Wenn sich nun Subgraph polynomiell auf Clique reduzieren lässt, also Clique $\preceq$ Subgraph gilt, dann folgt, dass Subgraph $\mathcal{NP}$-schwer ist.

Wir reduzieren also Clique $\leq$ Subgraph mit der Reduktionsfunkion

$$f(K, n) = (K_n, K)$$

wobei K_n der vollständige Graph auf n Knoten ist.

Beweis der Korrektheit:
1. Fall: Sei $(K, n) \in$ Clique. Dann besitzt K eine Clique der Größe n. Dann ist K_n ein Subgraph von K und daraus folgt $(K_n, K) \in$ Subgraph.
2. Fall: Sei $(K_n, K) \in$ Subgraph. Dann besitzt K eine n-Clique und damit ist $(K, n) \in$ Clique.

Laufzeit der Reduktion:
Um einen vollständigen Graphen der Größe n zu konstruieren, wird quadratische Laufzeit in Abhängigkeit von $n \leq |V|$ benötigt.

□

Bemerkung 3.5:
Subgraph generalisiert also Probleme wie Clique. Weitere verwandte Probleme sind IndependentSet und HamiltonianPath.

4. Implementierung

Dieses Kapitel beschreibt die Implementierung des zuvor konzipierten Microservices anhand ausgewählter Codebeispiele. Dazu gehen wir zunächst auf die genutzten Technologien und die Implementierung von SCAIVIEW ein.

4.1. Einbettung in die SCAIVIEW-Implementierung

Alle Projekte sind in GitLab[1] angelegt. Das Hauptprojekt „scaiview-happysearching“ arbeitet mit dem Framework Spring Boot[2]. Deshalb ist es notwendig im Javaprojekt eine `pom.xml` zu erstellen. Diese `pom.xml` ist für das Maven[3]-Build-System, das Abhängigkeiten zu Spring Boot und vor allem den Spring Boot Startern enthält. Die Abhängigkeiten zu Spring Boot ziehen nun alle benötigten Libraries nach und stellen so eine vollständige Laufzeitumgebung zur Verfügung.

Quellcode 4.1: Extra-Dependencies (Libraries) für `Subgraph`

```
<!-- https://mvnrepository.com/artifact/org.neo4j/neo4j-jdbc-driver -->
<dependency>
        <groupId>org.neo4j</groupId>
        <artifactId>neo4j-jdbc-driver</artifactId>
        <version>3.1.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple -->
<dependency>
        <groupId>com.googlecode.json-simple</groupId>
        <artifactId>json-simple</artifactId>
        <version>1.1.1</version>
</dependency>
```

Wie bereits kurz in den Abschnitten 3.1 und 3.2 beschrieben kommuniziert `Subgraph` über die SCAIVIEW-API. Die Klasse `GraphReader` stellt dabei die beiden Methoden `searchGraph` und `searchAllGraph` per JMS[4] bereit. Wird dort ein Graph im JSON-Format von der API übergeben, ruft die Klasse `Graph` die Methode `LinkedList<String> searchGraph(String search, boolean all)` auf, für `searchAllGraph` mit `all = true` und für `searchGraph` mit `all = false`. Von nun an werden die fünf Schritte des Schemas, wie im Abschnitt 3.2.3 beschrieben, durchlaufen.

1. Der JSON-Input(-String) wird in einen Digraph geparst:
   Aus dem JSON-Objekt werden `nodes` und `edges` entnommen und separat verarbeitet. Zuerst wird allen `nodes` der Eingabe mit `void initialiseGraphID()` eine Graph-ID zugewiesen. Dazu wird überprüft, ob der Knoten über Eigenschaften verfügt oder nur ein Platzhalter ist. Falls Eigenschaften existieren werden diese mittels Cypher in der Graphdatenbank abgefragt und gesucht. Existiert ein Knoten mit den gewünschten

[1] Anwendung zur Softwareentwicklung: Projektplanung, Projektüberwachung & Quellcodeverwaltung `https://gitlab.com` und hier speziell `http://gitlab.scai.fraunhofer.de`
[2] Spring bietet ein umfassendes Programmier- und Konfigurationsmodell für Java-basierte Anwendungen auf verschiedenen Implementierungsplattformen, `https://spring.io/projects/spring-boot`
[3] Build-Tool zum Erstellen und Verwalten von Java-basierten Projekten, `https://maven.apache.org/`
[4] JMS: Java Message Service – `https://www.jcp.org/en/jsr/detail?id=914`

Eigenschaften in der Graphdatenbank, so wird dem Digraphknoten die eindeutige Graphdatenbank-ID zugewiesen, womit die Isomorphiefunktion

$$\varphi\colon D \to \mathcal{G}$$
$$v \mapsto \mathrm{id}(v)$$

festgelegt wird. Besitzt der Knoten keine Eigenschaften oder ist in der Graphdatenbank nicht zu finden, so wird ihm die Graph-ID 0 zugewiesen. Für den Fall, dass die gewünschten Eigenschaften des Knotens mehreren Knoten der Graphdatenbank zuzuordnen sind, werden diese mit einem vorangestellten `"M"` gekennzeichnet und alle „passenden" IDs gespeichert. Später werden daraus mittels `LinkedList<Digraph> createMultigraph(Digraph d)` mehrere Suchdigraphen aufgebaut. Nachdem allen Knoten IDs zugewiesen wurden, werden die Kanten verarbeitet und die Knoten dementsprechend verknüpft.

2. Aus dem Digraphen wird eine Cypher-Query erstellt:
   Die Methode `String createCypherQuery(Digraph d)` übernimmt den Aufbau der Cypher-Query. Sie beginnt damit alle `nodes`, die über eine *positive* Graph-ID verfügen, als Variable festzulegen. Anschließend wird das Schlüsselwort `MATCH` eingebaut und für alle `edges` ein Pfad von ihrem jeweiligen Startknoten zum Endknoten erzeugt.

3. Die Cypher-Query wird in Neo4j abgefragt:
   Die erstellte Query wird mittels `Neo4JConnection.getInstance().runQuery(q)` in der Graphdatenbank abgefragt und als `StatementResult` gespeichert. Für die bessere Verarbeitung wird dieses in eine `List<Record>` umgespeichert.

4. Die Ergebnis-Record-List aus Neo4j wird in Digraphen umgewandelt:
   Die Methode `Digraph createGraphFromResult(String result)` erstellt für jedes Listenelement einen Digraph. Dazu werden die in `Record` zurückgegebenen Pfade wieder zu einem zusammenhängenden Digraphen zusammengesetzt.

5. Die Digraphen werden in das JSON-Format geschrieben:
   Um ein JSON-Objekt an die API zurück geben zu können, wird mit `JSONObject createJSONFromGraph(Digraph d)` der Digraph ins Ausgabeformat geschrieben. Dabei wird als Knoten-ID die Graph-ID verwendet und alle Informationen über den Knoten als langer String im Label gespeichert.

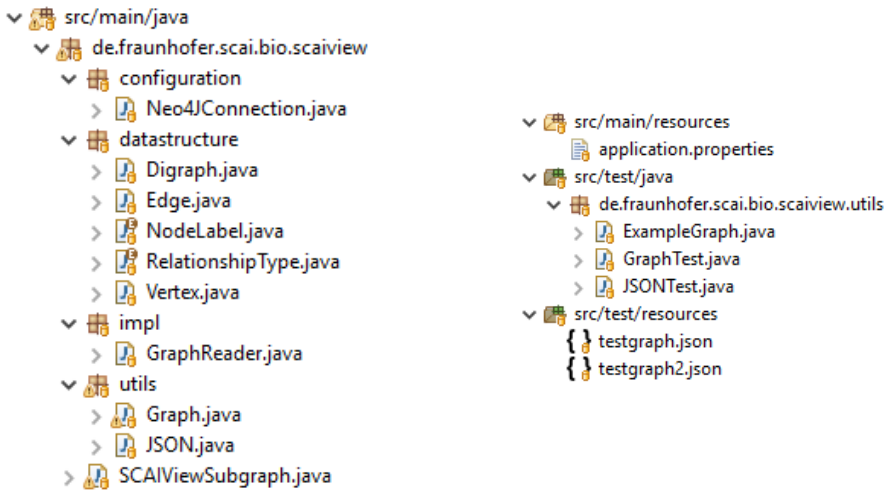


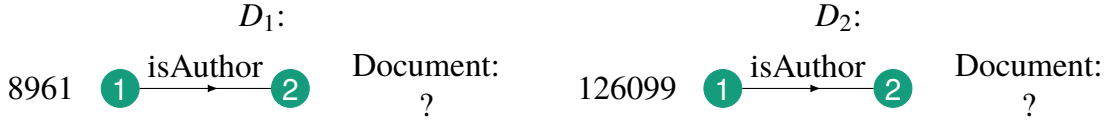
Abbildung 4.1.: Projektstruktur von `Subgraph`

Die wichtigsten Schritte dieses Vorgehens sind Schritt eins und zwei. In Schritt eins wird mit der dort beschriebenen Methode `void initialiseGraphID()` die Isomorphiefunktion festgelegt. Es wird Standardmäßig die Identität, also $\varphi = \mathrm{id}$, verwendet und die Knoten des Subgraphen mit den IDs des Knowledge-Graphen identifiziert. Später werden wir in Abschnitt 6.2 darauf eingehen, wie man diese Funktion variabler gestalten kann. Im zweiten Schritt wird mit `String createCypherQuery(Digraph d)` die Cypher-Query erstellt. Diese Query bestimmt die Art der Lösungsmethode des Subgraph-Problems maßgeblich mit, denn die Implementierung der verschiedenen Algorithmen setzt genau hier an.

Quellcode 4.2: Methode mit Festlegung der Isomorphiefunktion

```
1  public void initialiseGraphID() {
2          StringBuilder sb = new StringBuilder();
3          for (Map.Entry<String, String> entry : properties.entrySet()) {
4              String key = entry.getKey();
5              String value = entry.getValue();
6              if (!value.equals("")) {
7                  sb.append("MATCH (n:").append(key).append(") WHERE");
8                  for (NodeLabel nl : NodeLabel.values())
9                      if (nl.getNodeLabel().equals(key)) {
10                         for (String s : nl.getNodeKeys()) {
11                             sb.append(" n.").append(s).append(" = '").append(value).append(
                                   "' OR");
12                         }
13                         sb.setLength(sb.length() - 3);
14                         sb.append(" return id(n)");
15                     }
16             }
17         }
18         String query = sb.toString();
19         StatementResult result = Neo4JConnection.getInstance().runQuery(query);
20         if (result != null) {
21             List<Record> results = result.list();
22             if (results.size() == 0)
23                 graphID = "0";
24             else if (results.size() == 1)
25                  graphID = "" + results.get(0).get(0).asInt();
26             else if (results.size() >1) {
27                 graphID = "M";
28                 for (Record r : results) graphID += " "+r.get(0).asInt();
29             }
30         }
31         else
32             graphID = "0";
33     }
```

Beispiel 4.1 (Entstehung mehrerer Suchgraphen):
Bei der Eingabe von Beispiel 3.1.1. würde bei der Auswertung des Knoten 1 mit der Information `“Author“:“Pacheco“` die eindeutige Zuordnung in der Graphdatenbank nicht funktionieren. Der Nachname „Pacheco“ taucht dort mehrfach als „Author“ auf. Beispielsweise als „Henri“ mit der zugehörigen Graphdatenbank-ID „8961“ oder als „R.C.“ mit der ID „126099“. In diesem Fall würde `“M 8961 126099“` als Graph-ID festgelegt und somit zwei Digraphen D_1 und D_2 entstehen, wobei $8961 \in V_{D_1}$ und $126099 \in V_{D_2}$.



Die anschließende Suche berücksichtigt beide Subgraphen und liefert als Ergebnis

$$\{H : H \simeq D_1 \subset \mathcal{G}\} \cup \{H : H \simeq D_2 \subset \mathcal{G}\}.$$

4.2. Testumgebung

Um sich mit Neo4j & Cypher vertraut zu machen wurde zu Beginn die Bestimmung des Diameter, als Existenzkriterium für einen Subgraph, implementiert. Die Implementierung der Testumgebung wurde darauf ausgelegt die Methoden zur Ermittlung des Diameter, wie in Quellcode 4.3 angegeben, zu testen. Hierbei ermittelt `int diameter()` den globalen Diameter der gesamten Graphdatenbank und `int diameter(String type, String val)` den lokalen Diameter vom spezifizierten (Start- bzw. End-)Knoten aus.

Quellcode 4.3: Methoden zur Ermittlung des Graphdiameter

```
1  public int diameter() {
2   try (Session session = driver.session()) {
3    try (Transaction tx = session.beginTransaction()) {
4     String q = "MATCH p=(start:Node)-[:REL*]->(end:Node) " + "RETURN length(p) "
5       + "ORDER BY length(p) DESC " + "LIMIT 1";
6     StatementResult res = tx.run(q);
7     return res.single().get(0).asInt() + 1;
8    }
9   }
10 }
11
12 public int diameter(String type, String val) {
13  try (Session session = driver.session()) {
14   try (Transaction tx = session.beginTransaction()) {
15    String q = "MATCH p=(start:Node)-[:REL*]->(end:Node) " + "WHERE start." + type + "=\""
          + val
16      + "\" OR end." + type + "=\"" + val + "\" " + "RETURN length(p) " + "ORDER BY length(
            p) DESC "
17      + "LIMIT 1";
18    StatementResult result = tx.run(q);
19    return result.single().get(0).asInt() + 1;
20   }
21  }
22 }
```

Auf einem Testrechner[1] wurde eine kleine, lokale Instanz einer Neo4j-Graphdatenbank erstellt und mittels Quellcode 4.4 mit mehreren 1000 durchnummerierten Knoten befüllt. Die jeweilige Nummer wurde den Knoten als einzige Eigenschaft zugewiesen. Anschließend wurden Kanten, ohne weitere Eigenschaften, eingefügt und alle Knoten i mit ihrem jeweiligen Nachfolger $i + 1$ verbunden, so dass ein *linearer Graph* bzw. *Pfadgraph* entstand. Aus dieser langen, zusammenhängenden Kette wurden anschließend Kanten so entfernt, dass man die genaue Länge der entstandenen Kettenstücke kannte. Die dazu verwendeten Methoden sind in Quellcode 4.5 zu finden. Anschließend konnten die Methoden `int diameter()` und `int diameter(String type, String val)` erfolgreich getestet werden.

Die Anpassung der Diametermethoden auf den Knowledge-Graph hat allerdings zu Problemen geführt. Im Testgraphen ist nur eine Relation „`REL`“ vorhanden und es lässt sich leicht eine beliebig lange Kette dieser Relation mittels „`[REL*]`“ in der Graphdatenbank finden. Im Knowledge-Graph sind verschiedene Relationen, also RelationshipTypes wie in Tabelle 3.1 angegeben, definiert, welche alle abgefragt bzw. berücksichtigt werden müssen. Die einfachste Anpassung auf `[*]` hat leider nicht zum Erfolg geführt. Verschiedene Kombinationen aller RelationshipTypes, wie z. B. `[(hasDocument*|hasTerm*|isAuthor*|...)]` oder `[(hasDocument|hasTerm|isAuthor|...)*]`, haben ebenfalls nicht funktioniert. Der Diameter als Subgraph-Existenzkriterium wurde daher verworfen und in der Praxis nicht weiterverfolgt.

[1] Die Spezifikationen des Testrechners sind in Tabelle 4.1 auf Seite 31 zu finden

Quellcode 4.4: Erstellung einer Test Graphdatenbank mit $|V| = 5000$

```
1  public void createTestDB() {
2   try (Neo4J2 b = new Neo4J2()) {
3    System.out.print("Neo4J.createTestDB()..");
4    for (int i = 0; i < 5000; i++)
5     b.setKnoten("" + (i + 1));
6    for (int i = 0; i < 5000 - 1; i++)
7     b.setKanten((i + 1));
8    b.deleteRel("3000");
9    b.deleteRel("300");
10   b.deleteRel("91");
11   System.out.println(".fertig");
12  } catch (Exception e) {}
13 }
```

Quellcode 4.5: Methoden zum erstellen des Testgraphen

```
1  public void setKnoten(String n) {
2   try (Session session = driver.session()) {
3    try (Transaction tx = session.beginTransaction()) {
4     tx.run("CREATE (a:Node) " + "SET a.name = $n", parameters("n", n));
5     tx.success();
6    }
7   }
8  }
9
10 public void setKanten(int i) {
11  try (Session session = driver.session()) {
12   try (Transaction tx = session.beginTransaction()) {
13    String q = "MATCH (a:Node),(b:Node)" + "WHERE a.name = '" + i + "' AND b.name = '" + (i
          + 1) + "'"
14      + "CREATE (a)-[r:REL]->(b)";
15    tx.run(q);
16    tx.success();
17   }
18  }
19 }
20
21 public void deleteRel(String val) {
22  try (Session session = driver.session()) {
23   try (Transaction tx = session.beginTransaction()) {
24    tx.run("Match (:Node {name:\"" + val + "\"})-[r:REL]->(:Node) Delete r");
25    tx.success();
26   }
27  }
28 }
```

Tabelle 4.1.: Spezifikationen des Testrechners

Prozessor (CPU):	Intel® Core™ i7-3630QM @@ 2.40 GHz 2.40 GHz
Arbeitsspeicher (RAM):	8.00 GB (7.89 GB verwendbar)
Systemtyp:	64-Bit-Betriebssystem

4.3. Algorithmen

Für Testzwecke wurde ein *Brute-Force Algorithmus* implementiert. Dieser dient primär der Überprüfung der später implementierten Algorithmen. Für das Lösen des Subgraph-Problems auf dem Knowledge-Graph wurde ein *Backtracking Algorithmus*, in zwei unterschiedlichen Varianten, implementiert. Wie bereits erwähnt setzen die eigentlichen Lösungsalgorithmen in Schritt zwei bei `String createCypherQuery(Digraph d)` an.

4.3.1. Brute-Force Algorithmus

Der Algorithmus beginnt damit die Graphdatenbank-IDs numerisch zu sortieren. Anschließend wird über alle Knoten iteriert, wobei mit dem mit der kleinsten ID begonnen wird. Bei jedem Knoten wird nun über die Anzahl der Kanten iteriert, wobei nur auslaufende Kanten betrachtet werden. Mittels Cypher werden in einem Schritt alle Subgraphen, vom Startknoten s aus, mit n Kanten zurückgegeben. Ist die Rückgabe für ein n leer, so ist der Knoten abgearbeitet und es wird der mit der nächst höheren ID betrachtet.

Ähnlich zur Testumgebung des Diameter haben wir nur ein NodeLabel `Node` und einen RelationshipType `REL`, womit sich eine einfache Cypher-Query

```
(s:Node)-[:REL n]->(:Node)
```

bilden lässt. Als Testumgebung wurden hier keine Pfadgraphen, sondern Graphen $G \subset K_n$ mit $n \leq 20$ gewählt.

Der leere Graph $(\emptyset, \emptyset)$ und Graphen ohne Kanten $(V, \emptyset)$ wurden dabei nicht als Subgraphen gezählt und dementsprechend nicht berücksichtigt.

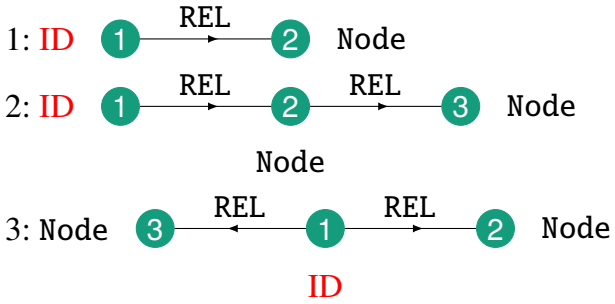
4.3.2. Backtracking Algorithmus

Eigene Variante

In der eigenen Variante wurde versucht, vor der Übergabe der Query an Cypher, im Java-Code selber mehr algorithmisch zu leisten. Es sollte also weniger Arbeit an Cypher ausgelagert werden. Dazu wurde der Suchgraph nicht direkt komplett als Query umgesetzt, sondern sukzessive, Knoten für Knoten, zusammengesetzt und in jedem Schritt erneut gesucht. Auch hier wurden der leere Graph $(\emptyset, \emptyset)$ und Graphen ohne Kanten $(V, \emptyset)$ nicht berücksichtigt und genau wie in der cypherbasierten Variante mit Knoten begonnen, die über Eigenschaften verfügen und nicht Platzhalter sind.

Wir beginnen mit einem Graphen wie unter 1. von Beispiel 4.2. Ist die Rückgabeliste von Cypher nicht leer, so existiert mindestens ein Treffer für den Graphen und wir setzen die Suche bspw. mit 2. oder 3. fort, je nach Kantenverbindungen des Suchgraphen. Diese Kantenerweiterung setzen wir solange fort, bis entweder der vollständige Suchgraph abgebildet ist oder Cypher eine leere Ergebnisliste zurück gibt. Im ersten Fall haben wir unsere Suche erfolgreich beendet. Im zweiten Fall muss der Graph nicht weiter vervollständigt werden, da mindestens eine Anforderung an den Ergebnisgraphen nicht erfüllt wird. Dessen können wir uns sicher sein, da Cypher immer die ganze NodeLabel-Kategorie abfragt.

Beispiel 4.2:



Cypherbasiert

Dieser Algorithmus nutzt parallelisiertes Backtracking durch Cypher. Der komplette Suchgraph wird dazu in einer einzigen Cypher-Query codiert und alle Knoten und Kanten werden zeitgleich abgefragt.

Quellcode 4.6: Erstellen der Query des vollständigen Suchgraphen

```
1  private String createCypherQuery(Digraph d) {
2          ArrayList<Vertex> vertices = d.getVertices();
3          // fuer k = beliebig
4          StringBuilder sb = new StringBuilder();
5          sb.append("START ");
6          for (Vertex v : vertices) {
7              if (!v.getGraphID().equals("0"))
8                  sb.append("n" + v.getId() + "=NODE(" + v.getGraphID() + "), ");
9          }
10         if (sb.length() == 6)
11             sb = new StringBuilder();
12         else
13             sb.setLength(sb.length() - 2);
14         sb.append(" MATCH ");
15         int i = 1;
16         for (Vertex v : vertices) {
17             for (Edge e : v.getEdges())
18                 for (Map.Entry<String, String> edgeentry : e.getProperties().entrySet()) {
19                     sb.append("p" + i++ + "=(n" + v.getId() + ")-[:" + edgeentry.getValue()
                               + "]->(n"
20                                 + e.toVertex().getId() + "), ");
21                 }
22         }
23         if (vertices.size() > 0)
24             sb.setLength(sb.length() - 2);
25         sb.append(" RETURN [");
26         while (i > 1)
27             sb.append("p" + --i + ",");
28         sb.setLength(sb.length() - 1);
29         sb.append("]");
30         // sb.append(" LIMIT 1");
31         logger.info("Query for any k created");
32         return sb.toString();
33     }
```

4.4. Unit-Tests

Für die beiden `utils`-Klassen `Graph` und `JSON` wurden Unit-Tests erstellt. Die Klasse `JSON` ist als Hilfsklasse für das Lesen und Schreiben aus und in das JSON-Graph-Format, wie in Abschnitt 3.3.2 festgelegt, zuständig. Der folgende Unit-Test erzeugt dabei genau das standard JSON-Graph-Format als JSONObject. Würde dieses JSONObject also in eine Datei geschrieben, so enthielt diese exakt den in Abschnitt 3.3.2 angegeben Inhalt.

Quellcode 4.7: JSON-Graph-Format durch Java-Code erzeugt

```
1      public static JSONObject standardGraph() {
2          // creating JSONObject
3          JSONObject jo = new JSONObject();
4          LinkedHashMap<String, JSONArray> graph = new LinkedHashMap<String, JSONArray>(2);
5          JSONArray nodes = new JSONArray();
6          JSONArray edges = new JSONArray();
7
8          graph.put("nodes", nodes);
9          graph.put("edges", edges);
10         jo.put("graph", graph);
11
12         // for data, create LinkedHashMap
13         LinkedHashMap<String, Object> n1 = new LinkedHashMap<String, Object>(2);
14         n1.put("id", 1);
```

```
15          n1.put("label", "Knoten 1");
16          LinkedHashMap<String, Object> n2 = new LinkedHashMap<String, Object>(2);
17          n2.put("id", 2);
18          n2.put("label", "Knoten 2");
19
20          nodes.add(n1);
21          nodes.add(n2);
22
23          LinkedHashMap<String, Object> e1 = new LinkedHashMap<String, Object>(3);
24          e1.put("from", 1);
25          e1.put("to", 2);
26          JSONArray properties = new JSONArray();
27          properties.add(new LinkedHashMap<Object, Object>());
28          e1.put("properties", properties);
29
30          edges.add(e1);
31
32          return jo;
33      }
```

Die Klasse `Graph` stellt alle Methoden für die Suche in der Graphdatenbank bereit. Mit entsprechenden Unit-Tests kann z. B. geprüft werden, wie die NodeKeys der definierten NodeLabel lauten oder ob überhaupt Daten in der Graphdatenbank vorhanden sind.

```
1       @Test
2       public void testDBConnection() {
3           logger.info("testDBConnection");
4           logger.info("NodeLabels:");
5           for (NodeLabel nl : NodeLabel.values()) {
6               logger.info(nl.getNodeLabel() + "-Keys: " + nl.getNodeKeys());
7           }
8           logger.info("RelationshipTypes");
9           for (RelationshipType rt : RelationshipType.values()) {
10              logger.info(rt.getRelationshipType());
11          }
12      }
13
14      @Test
15      public void testgetDBSize() {
16          logger.info("testgetDBSize");
17          Graph g = new Graph();
18          int s = g.getDBSize();
19          logger.info("DB-Size: " + s);
20          assertTrue("Size > 0", s > 0);
21      }
```

Damit dies korrekt funktioniert, ist es wichtig die Graphdatenbank vor den Tests zu verbinden. Über die Methode `void setUp()`, welche mit der Annotation `@Before` gekennzeichnet ist, wird die Verbindung hergestellt.

Quellcode 4.8: Verbindung zur Datenbank herstellen

```
1       @Before
2       public void setUp() {
3           Neo4JConnection.getInstance("bolt://delorean:15087", "neo4j", "keins123");
4       }
```

Über die Testklasse kann auch im Knowledge-Graph gesucht werden. Wie in Quellcode 4.9 kann eine vorher erstellte JSON-Datei eingelesen und nach dem entsprechenden Subgraphen gesucht werden. Während der Entwicklung haben die Unit-Test-Methoden so oft den Umweg über die API ersetzt.

Quellcode 4.9: Der in „testgraph.json“ definierte Subgraph wird mittels `searchGraph`-Methode im Knowledge-Graph gesucht

```
1     @Test
2     public void testGraph1() {
3         logger.info("testGraph from File");
4         try {
5             Graph g = new Graph();
6             JSONObject jo = (JSONObject) new JSONParser()
7                     .parse(new FileReader(new File("src/test/resources/", "testgraph.json")
                          ));
8             g.searchGraph(jo.toJSONString(), false);
9         } catch (Exception e) {
10            logger.error("Exception", e);
11        }
12    }
```

5. Analyse

In diesem Kapitel vergleichen wir die im vorherigen Kapitel beschriebenen Algorithmen. Dabei werden nicht alle Methoden miteinander verglichen, sondern nur die beiden Varianten des Backtracking Algorithmus. Zu Beginn werden die Testumgebungen beschrieben und im Anschluss wird analysiert, welcher der Algorithmen das bessere Laufzeitverhalten aufweist.

5.1. Testszenarien

Für die Lösungsalgorithmen wurde, ähnlich wie in Abschnitt 4.2 für die Diameter-Methoden, eine Testinstanz geschaffen. Diese wollen wir im folgenden näher beschreiben.

Auf einem Testrechner[1] wurde eine lokale Instanz einer Neo4j-Graphdatenbank angelegt. Die Testumgebung beinhaltet nur ein NodeLabel `Node` und einen RelationshipType `REL`. Alle Knoten die in die Graphdatenbank aufgenommen werden, werden über ihren Namen identifiziert. Dieser Name ist eine Nummer zwischen 1 und $n = |V|$.

Es gibt mehrere Testszenarien für die unterschiedlichen Methoden und Algorithmen.

1. Diameter-Methoden:
   Wie in Abschnitt 4.2 beschrieben wurde die Methode auf Pfadgraphen beliebiger Länge getestet.

2. Brute-Force Algorithmus:
   Zuerst wurde die Methode auf dem vollständigen Graphen K_3 getestet. Im Anschluss wurden zufällige Graphen mit $|V| \leq 20$ geschaffen. Diese wurden meistens mit Hilfe von Wolfram Mathematica erstellt und in die Graphdatenbank eingelesen. Die Korrektheit des Algorithmus, also ob tatsächlich alle Subgraphen aufgelistet werden, wurde durch den Einsatz von Mathematica erheblich vereinfacht. Wie in Abschnitt 4.3.1 erwähnt, wurden der leere Graph $(\emptyset, \emptyset)$ und Graphen ohne Kanten $(V, \emptyset)$ nicht als Subgraphen berücksichtigt[2].

3. Backtracking Algorithmus:
   Beide Varianten des Algorithmus wurden auf den selben Test-Graphdatenbanken wie der Brute-Force Algorithmus getestet. Die cypherbasierte Variante würde zusätzlich direkt auf dem Knowledge-Graph getestet.

Beim lokalen testen und arbeiten mit Neo4j ist die Weboberfläche[3] extrem hilfreich. Hier erhält man schnell eine Übersicht über die aktuellen NodeLabels und RelationshipTypes, kann schnell und bequem Cypher-Queries auf die Graphdatenbank anwenden und sich Abfragen graphisch darstellen lassen.

[1] Die Spezifikationen des Testrechners sind in Tabelle 4.1 auf Seite 31 zu finden
[2] *vgl.* Abbildung 5.1
[3] *siehe* figs. 5.2 and 5.3

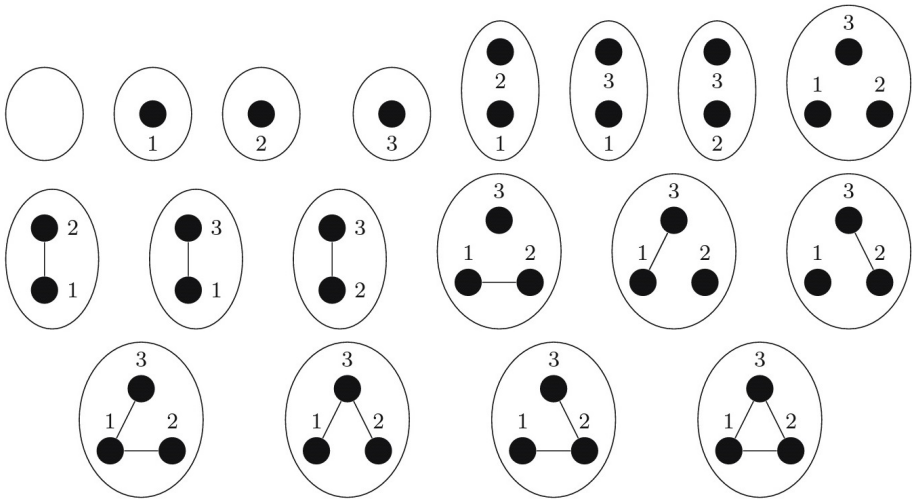


Abbildung 5.1.: Abbildungen aller 18 Subgraphen des K_3. Die 8 Subgraphen in der ersten Zeile wurden vom Brute-Force Algorithmus nicht berücksichtigt. [13, S. 229]

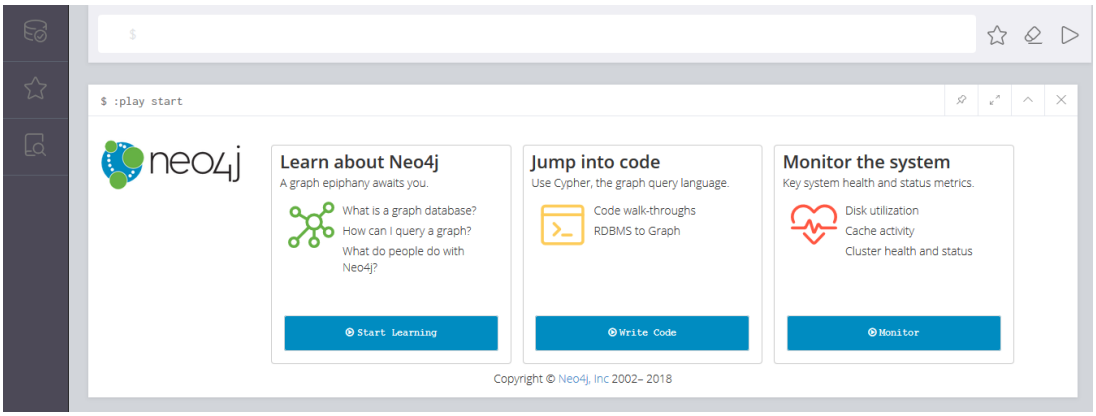


Abbildung 5.2.: Neo4j-Benutzeroberfläche: `http://localhost:7474/`

```
START  n1=NODE(108654),
       n2=NODE(108629),
       n3=NODE(110036)
MATCH  p1=(n1)-[:isAuthor]->(n4),
       p2=(n4)-[:isOfType]->(n3),
       p3=(n5)-[:isOfType]->(n3),
       p4=(n6)-[:isAuthor]->(n4)
RETURN [p4,p3,p2,p1]
 LIMIT 20
```

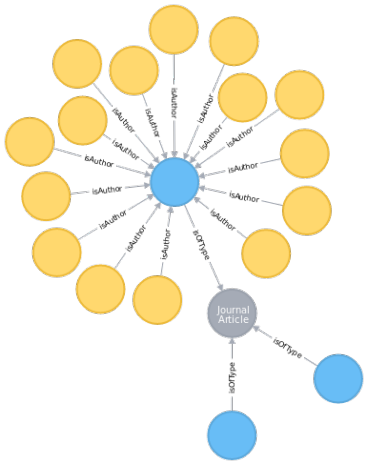


Abbildung 5.3.: Cypher-Query und zugehörige Neo4j-Ausgabe auf $\mathcal{G}$. Die Query basiert dabei auf Beispiel 3.2.2, wobei zwei Kanten entfernt wurden.

5.2. Vergleich

Der Brute-Force Algorithmus ist für größere Datenbanken, wie hier mit aktuell 60.723 Knoten, eher ungeeignet und dient primär der Überprüfung der Varianten des Backtracking Algorithmus. Diese beiden Varianten des Backtracking Algorithmus wollen wir nun miteinander vergleichen.

Zuerst wollen wir auf den Einfluss des `boolean all` der Methode `LinkedList<String> searchGraph(String search, boolean all)` eingehen. Für den Fall `all = false` wird die Suche nach dem ersten gefundenen Subgraphen beendet. Da es in diesem Fall vorkommen kann, dass nur eine Query in der Graphdatenbank abgefragt wird, wurde für die Laufzeitanalyse die Methode mit dem Parameter `all = true` aufgerufen. Es wurden also in beiden Varianten immer alle Subgraphen gesucht.

Die eigentliche Suche im Knowledge-Graph findet im Quellcode nach der Erstellung der Query und vor der Ausgabe der Query statt. In der Abbildung 5.4 also zwischen Zeile 4 und 5. Die Dauer der Suchanfrage ist in diesem Beispiel 5 m sec.

```
13:26:46.802 [main] INFO de.....Graph - Start
13:26:46.845 [main] INFO de.....Graph - Digraph from JSON created
13:26:46.931 [main] INFO de.....Graph - Multi-Digraph created
13:26:46.931 [main] INFO de.....Graph - Query for any k created
13:26:46.936 [main] INFO de.....Graph - START n1,., MATCH p1=.,. RETURN [p1,p2,.]
13:26:46.936 [main] INFO de.....Graph - Exists: YES
13:26:46.936 [main] INFO de.....Graph - Matches: 2
13:26:46.936 [main] INFO de.....Graph - Digraph from result created
13:26:46.959 [main] INFO de.....Graph - JSON from graph created
13:26:46.959 [main] INFO de.....Graph - Results: 1
```

Abbildung 5.4.: Beispielausgabe einer Suche in der eclipse-Console

Empirisch lässt sich sagen, dass beide Algorithmen das gleiche Verhalten zeigen. Beispielhaft ist es in den figs. 5.5 and 5.6 dargestellt. Eine Abfrage dauert durchschnittlich zwischen 3 m sec und 13 m sec und sie ist unabhängig von der Größe des Eingabegraphen.

Auffällig ist, dass die erste Graphdatenbank-Abfrage mit durchschnittlich 140 m sec signifikant größer ist. Dies liegt vermutlich am Verbindungsaufbau zur Graphdatenbank. Die weiteren Ausreißer sind im Gesamtbild zu vernachlässigen.

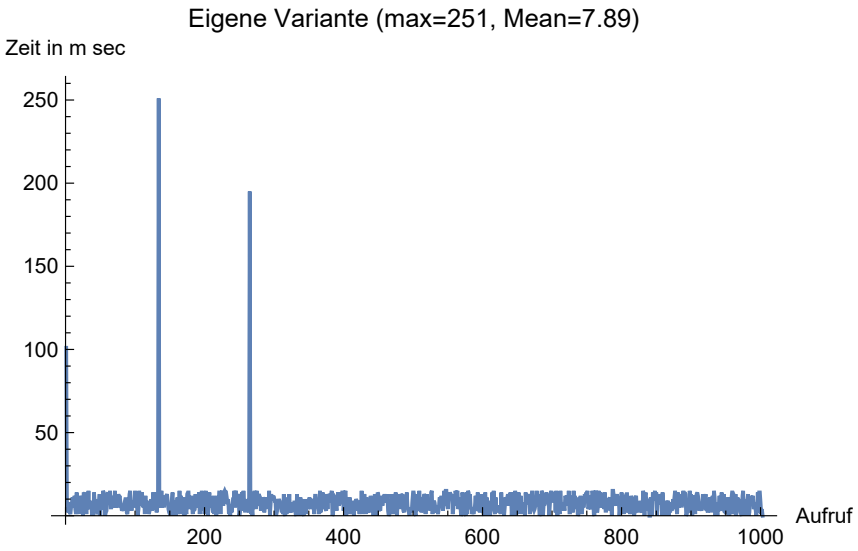


Abbildung 5.5.

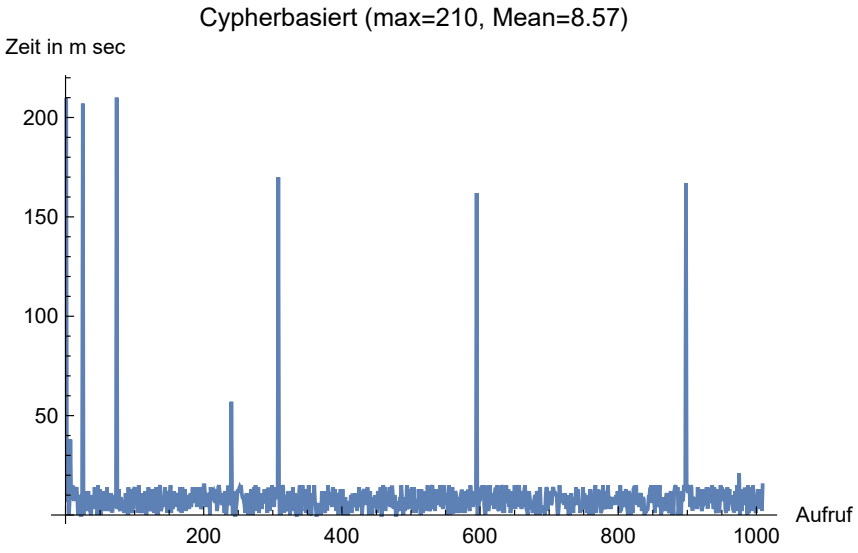


Abbildung 5.6.

Das einheitliche Verhalten lässt sich auch dadurch erklären, dass bei der eigenen Variante jede sukzessive Abfrage mit `LIMIT 1` beendet wird. Zumindest bei allen Queries die aus weniger Knoten bestehen, als im aktuellen Suchgraphen vorkommen.

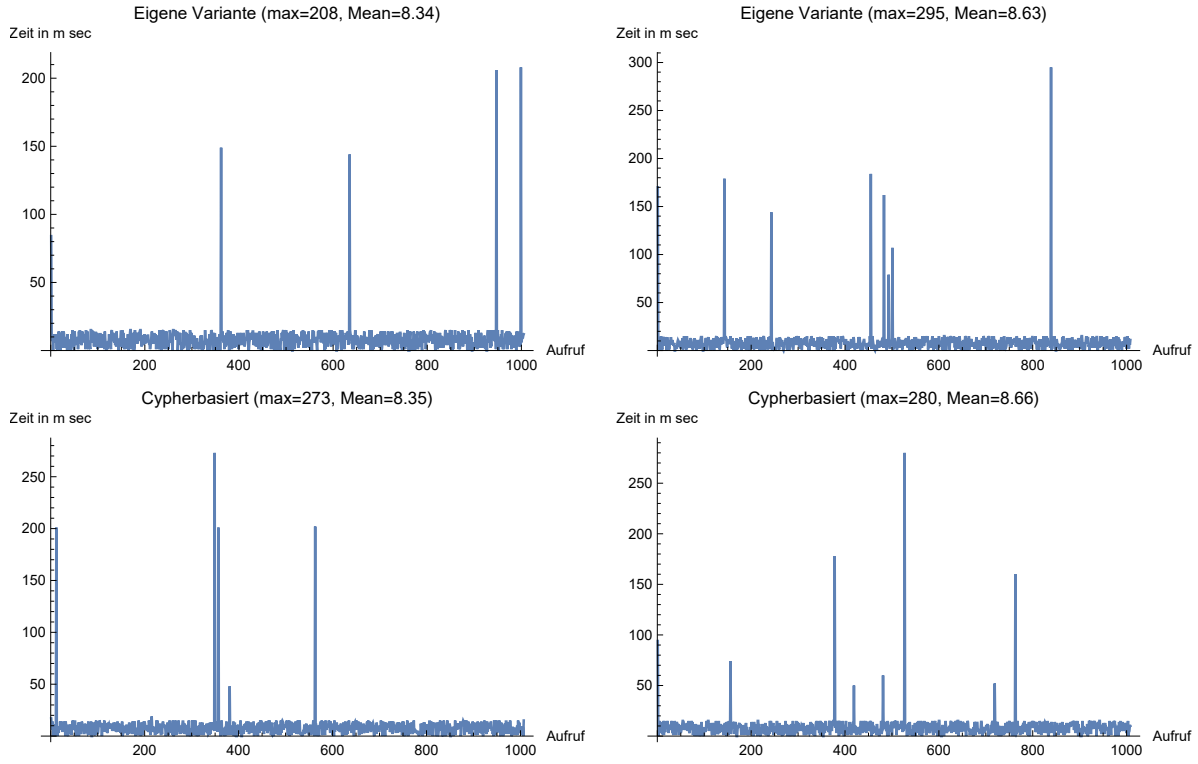


Abbildung 5.7.: Weitere Beispiele

6. Fazit

In diesem Kapitel geht es darum zu überprüfen, ob die in Abschnitt 1.2 genannten Ziele erfüllt wurden. Es wird erörtert welcher Algorithmus für die Praxis am wertvollsten ist und der praktische Nutzen des Microservice `Subgraph` herausgestellt. Im Ausblick wird auf mögliche Verbesserungen und Erweiterungen des Microservice aufmerksam gemacht.

6.1. Ergebnisse

Ein erster Ansatz war der Brute-Force Algorithmus. Dieser zählt alle Subgraphen des Obergraphen auf und vergleicht im Anschluss, ob der gesuchte Subgraph einer dieser Subgraphen ist. Ein solcher Algorithmus ist für die Praxis ungeeignet.

Beide Varianten des Backtracking Algorithmus verhalten sich in den Zeittests sehr ähnlich. Dies resultiert sicherlich aus ihrer ähnlichen Implementierung. Beide Varianten sind nahezu gleich gut und können in der Praxis eingesetzt werden. Weil der cypherbasirte Algorithmus nur eine einzige Query verwendet, statt mehrerer aufeinander aufbauender, würde ich persönlich diesen Algorithmus bevorzugen. Hervorzuheben ist an dieser Stelle nochmal, dass Cypher eine einfache und dennoch sehr mächtige Sprache ist, ohne die das suchen nach Subgraphen in der Graphdatenbank sicherlich schwieriger gewesen wäre.

Trotz der $\mathcal{NP}$-Vollständigkeit des SUBGRAPH-Problems haben sich die Varianten des Backtracking Algorithmus auch in der Anwendung als schnell und effizient herausgestellt.

Abschließend sei noch erwähnt, dass sich mit Programmen wie Cytoscape, Giphy oder Mathematica, mit wenig Aufwand JSON-Dateien visualisieren lassen. Der Microservice ist damit auch ohne API-Anbindung und ohne Frontend komplett benutzbar. Die Darstellung der einzelnen Knoten und Kanten ist dabei jedem selbst überlassen und kann beliebig formatiert werden.

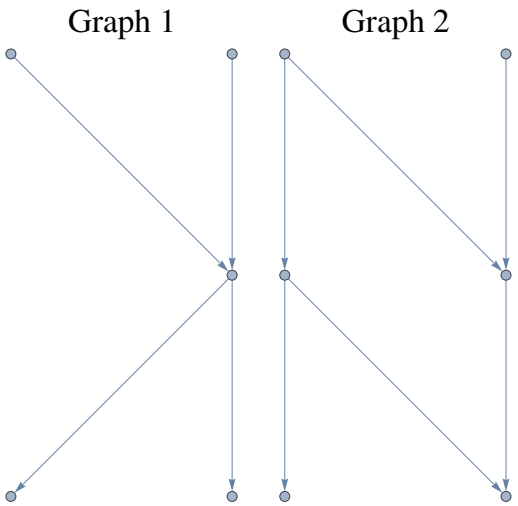


Abbildung 6.1.: Ausgabe von Beispiel 3.2 mit Mathematica, ohne Formatierung von Knoten und Kanten

6.2. Ausblick

Die im Rahmen der vorliegenden Arbeit gewonnenen Erkenntnisse lassen weiterführende Tätigkeiten zu. Dazu gehören neben der Verbesserung der untersuchten Algorithmen auch die Erweiterung des Microservice.

Knowledge-Graph

Wird in den Knowledge-Graph, z. B. bei der Erweiterung um eine Datenbank, ein neues NodeLabel bzw. ein neuer RelationshipType eingefügt, so können diese durch anpassen der Enumeration `enum NodeLabel` und `enum RelationshipType` mit abgefragt werden. Die Erweiterung von NodeKeys übernimmt das Programm bereits jetzt automatisch.

Isomorphiefunktion

Um den Extraaufwand bei der Entstehung mehrerer Suchgraphen wie in Beispiel 4.1 zu vermeiden, wäre es möglich z. B. durch die zusätzliche Angabe weiterer Information (sofern bekannt) genauer zu fragen bzw. suchen. Alternativ könnte man auch, mit Hilfe der Graphdatenbank-ID, einen Teil der Isomorphiefunktion selber festlegen.

Generell muss die Isomorphiefunktion φ nicht zwingend die id sein. Man könnte z. B. auch ein Interface verwenden und die Isomorphiefunktion damit variabel halten. Als „default“, falls das Interface nicht implementiert wird, kann weiterhin die id verwendet werden.

Heuristik

Bei den Backtracking Algorithmen könnte man statt der Tiefensuche eine Breitensuche probieren, wobei man nicht auf die Verwendung von Cypher verzichten sollte. Generell ließen sich auch noch andere Programmieransätze realisieren, wie z. B.

- Maximum Clique-Based Algorithmen oder
- Dynamische Programmierung.

Kontextmodularisierung

Die Metadaten der Dokumente beschreiben Kategorien, welche über die NodeLabel, wie in Tabelle 3.1, spezifiziert werden. Sobald weitere Datenbanken in den Knowledge-Graph aufgenommen wurden, und Kategorien wie „Protein“ oder „Organism“ verfügbar werden, wäre es möglich Proteinanwendungen auf andere Organismen zu übertragen.

`Protein:X` → `Organism:Fly` zu `Protein:X` → `Organism:Human`

Analog zur Proteinanwendung von der Fliege auf den Menschen, ließen sich auch Krankheiten auf andere Organismen übertragen.

Anstatt Eigenschaften zu übertragen, könnten diese auch generalisiert werden. Beispielsweise von

`Medikament:x` erhöht `Wert:z` zu `Medikament:` erhöht `Wert:z`

oder sogar zu `Wirkstoff:` erhöht `Wert:z`.

Generell sind diese Kategorien (also Kontext- bzw. Konzeptklassen) generisch und damit leicht erweiterbar.

Ontologie

In der Informatik ist eine Ontologie eine formale Beschreibung einer Begriffswelt. In einer Ontologie soll Information so repräsentiert werden, dass Computer damit in einer sinnvollen Weise arbeiten können.

Ontology Mapping ist ein Prozess, um Ähnlichkeiten zwischen Konzepten aus verschiedenen Ontologien herzustellen. Bei zwei Ontologien $\mathcal{A}$ und $\mathcal{B}$, bedeutet das Abbilden einer Ontologie mit einer anderen, dass für jedes Konzept in Ontologie $\mathcal{A}$, ein entsprechendes Konzept in Ontologie $\mathcal{B}$ gefunden werden muss, welches die gleiche oder eine ähnliche Semantik aufweist, und umgekehrt. Ontology Mapping ist erforderlich, um Wissensaustausch und semantische Integration in einer Umgebung mit verschiedenen zugrunde liegenden Ontologien zu erreichen.

Mit Hilfe der oben erwähnten Kontextmodularisierung könnte man versuchen Ähnlichkeiten zwischen zwei Ontologien zu finden. Speziell zwischen der genutzten im Knowledge-Graph und einer externen. Ist bereits ein Isomorphismus zwischen einer externen Ontologie zu der des Knowledge-Graph bekannt, so kann man Abfragen der externen Ontologie leicht in Abfragen für den Knowledge-Graph konvertieren.

BEL

Die *Biological Expression Language*[1] ist eine Sprache zur Darstellung naturwissenschaftlicher Erkenntnisse. Mit ihr sollen kausale und korrelative Zusammenhänge im Kontext erfasst werden. Der Kontext beinhaltet dabei Informationen über das Experiment, in dem die Zusammenhänge zu einem bekannten biologischen System beobachtet wurden, sowie die dabei zitierten Begleitpublikationen.

Ähnlich zur Verwendung einer zweiten Ontologie mit dem Knowledge-Graph, kann auch versucht werden BEL-Abfragen im Knowledge-Graph möglich zu machen. Obwohl BEL-Abfragen ähnlich aussehen wie Cypher-Queries, werden bei BEL viele eigene Methoden verwendet. Diese Methoden bieten Funktionen wie Protein-Modifikation oder Protein-Übertragung, die BEL-Struktur arbeitet mit RelationshipTypes (z. B. increases) und Annotationen.

Es gibt viele Gemeinsamkeiten zu BEL, wie RelationshipTypes und Annotationen, und auch die Funktionen von Methoden lassen sich durch kleine Erweiterungen im Knowledge-Graph umsetzen. Dennoch denke ich, das eine Erweiterung auf BEL-Queries schwer umzusetzen ist.

[1] BEL: Biological Expression Language

Verzeichnisse

A. Quellenverzeichnis

A.a. Literaturverzeichnis

[1] G. Gati. „Further annotated bibliography on the isomorphism disease“. In: *Journal of Graph Theory* 3 (1979), S. 95–109 (siehe S. 1).

[2] R.C. Read und D.G. Corneil. „The graph isomorphism disease“. In: *Journal of Graph Theory* 1 (1977), S. 339–363 (siehe S. 1).

[3] William J. Cook u. a. *Combinatorial Optimization*. New York: John Wiley & Sons, 2011. ISBN: 978-1-118-03139-1 (siehe S. 6).

[4] Reinhard Diestel. *Graphentheorie*. 3. neu bearb. u. erw. Wiesbaden: Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-21391-8 (siehe S. 6).

[5] Bernhard H. Korte und Jens Vygen. *Kombinatorische Optimierung - Theorie Und Algorithmen*. 2. deutsche Auflage. Berlin: Springer, 2012. ISBN: 978-3-540-76919-4 (siehe S. 6).

[6] Douglas B. West. *Introduction to Graph Theory*. München: Pearson, 2017. ISBN: 978-0-131-43737-1 (siehe S. 6).

[7] Ingo Wegener. *Theoretische Informatik - eine algorithmenorientierte Einführung*. 3. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2015. ISBN: 978-3-322-82204-8 (siehe S. 13).

[8] K. Rüdiger Reischuk. *Einführung in die Komplexitätstheorie - Band 1: Grundlagen Maschinenmodelle, Zeit- und Platzkomplexität, Nichtdeterminismus*. 2. völlig neu bearb. u. erw. Aufl. 1999. Berlin Heidelberg New York: Springer-Verlag, 1999. ISBN: 978-3-519-12275-3 (siehe S. 13, 25).

[9] J.W. Raymond und P. Willett. „Maximum common subgraph isomorphism algorithms for the matching of chemical structures“. In: *Journal of Computer-Aided Molecular Design* 16.7 (2002), S. 521–533 (siehe S. 16).

[10] Jens Dörpinghaus u. a. *Interne-Quelle: SCAIVIEW– A Semantic Search Engine for Biomedical Research Utilizing a Microservice Architecture*. Department of Bioinformatics, Fraunhofer Institute for Scientific Computing und Algorithms SCAI, Schloss Biringhoven, Sankt Augustin, Germany (siehe S. 17).

[11] Marc Jacobs. *Interne-Quelle: SCAIVIEW-Architecture*. Department of Bioinformatics, Fraunhofer Institute for Scientific Computing und Algorithms SCAI, Schloss Biringhoven, Sankt Augustin, Germany (siehe S. 18).

[12] Andreas Stefan (Hochschule Bonn-Rhein-Sieg). *Interne-Quelle: Neo4j-Datenmodell*. Department of Bioinformatics, Fraunhofer Institute for Scientific Computing und Algorithms SCAI, Schloss Biringhoven, Sankt Augustin, Germany, 2019 (siehe S. 21).

[13] Christoph Meinel, Martin Mundhenk und Martin Mundhenk. *Mathematische Grundlagen der Informatik - Mathematisches Denken und Beweisen. Eine Einführung*. 6. Aufl. Berlin Heidelberg New York: Springer-Verlag, 2015. Kap. 12. ISBN: 978-3-658-09885-8 (siehe S. 37).

A.b. Internetquellen

[14] *Semantic Web*. URL: https://basecamp.telefonica.de/event/web-1-0-bis-4-0-von-websites-ueber-semantik-zur-kuenstlichen-intelligenz/ (besucht am 12. 04. 2019) (siehe S. 2).

[15] *Semantic Web*. URL: https://www.advidera.com/glossar/semantic-web/ (besucht am 12. 02. 2019) (siehe S. 3).

[16] *Semantic Web*. URL: https://www.xovi.de/wiki/Semantic_Web (besucht am 12. 04. 2019) (siehe S. 3).

[17] *SPARQL*. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/ (besucht am 12. 04. 2019) (siehe S. 3).

[18] *Semantic Web Stack*. URL: https://en.wikipedia.org/wiki/Semantic_Web_Stack (besucht am 21. 02. 2019) (siehe S. 4).

[19] *Semantic Web*. URL: https://www.itwissen.info/Semantisches-Web-semantic-web.html (besucht am 12. 04. 2019) (siehe S. 4).

[20] *Was ist ein Knowledge Graph?* URL: https://backlinked.de/wiki/knowledge-graph/ (besucht am 12. 11. 2018) (siehe S. 9, 10).

[21] *Statistiken zu Suchmaschinen*. URL: https://de.statista.com/themen/111/suchmaschinen/ (besucht am 20. 03. 2019) (siehe S. 17).

[22] *Semantische Suche*. URL: https://www.empolis.com/blog/kuenstliche-intelligenz/semantische-suche/ (besucht am 12. 04. 2019) (siehe S. 17).

[23] *Neo4j*. URL: https://neo4j.com/docs/getting-started/current/ (besucht am 15. 02. 2019) (siehe S. 20).

[24] *Cypher*. URL: https://neo4j.com/docs/getting-started/current/cypher-intro/ (besucht am 21. 02. 2019) (siehe S. 21).

A.c. Weiterführende Literatur

[25] *Semantische Suche*. URL: https://de.ryte.com/wiki/Semantic_Search (besucht am 01. 04. 2019).

[26] Christian Schindelhauer. *Theoretische Informatik*. 2008. URL: http://archive.cone.informatik.uni-freiburg.de/teaching/lecture/informatik-III-w07/folien.html (besucht am 12. 03. 2019).

[27] Rainer Schrader. *Graphentheorie*. Vorlesungsfolien. Institut für Informatik, 2013.

[28] Frank Vallentin. *Operations Research*. Vorlesungsskript. Institut für Mathematik, 2014.

[29] *Semantic Web*. URL: https://www.w3.org/standards/semanticweb/ (besucht am 10. 02. 2019).

[30] *Komplexitätstheorie*. URL: https://de.wikipedia.org/wiki/Komplexit%C3%A4tstheorie (besucht am 15. 03. 2019).

[31] *Isomorphie von Graphen*. URL: https://de.wikipedia.org/wiki/Isomorphie_von_Graphen (besucht am 10. 12. 2018).

[32] *Semantische Suche*. URL: https://de.wikipedia.org/wiki/Semantische_Suche (besucht am 20. 03. 2019).

[33] *Graphentheorie – Grundbegriffe und Isomorphie*. URL: https://www.youtube.com/watch?v=b-NGTxYH6qM (besucht am 18. 03. 2019).

B. Index

-L-

-M-

-O-

-R-

-S-

-W-

-Z-

C. Definitionen

I. Subgraphisomorphie