Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

Ansätze für das Frequent-Subgraph-Problem als Anwendung für Graph Mining

Bachelorarbeit

im Studiengang Informatik

von

Maria Buling

Prüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier
Betreuerin: Dipl.-Math. Irena Schindler

Hannover, 26. September 2014

Inhaltsverzeichnis

1 Einleitung

Die Mustererkennung in typischerweise sehr großen Datenmengen bezeichnet man als Data Mining, z. B. das Finden von Häufungen und Ausreißern in Messwerten, Regressionsanalyse oder das Klassifizieren von Daten in bestimmte Kategorien. Dafür werden für jede dieser Aufgaben bestimmte Methoden angewendet mit dem Ziel, solche Muster zu finden ([HK06]). Der Wertebereich bei Data-Mining besteht meist aus reellen Zahlen oder Wertetupeln, etwa in Datenbanken.

Graph-Mining ist eine Erweiterung des Data-Minings auf Graphen. Die Problemstellungen sind deutlich anspruchsvoller bei der Untersuchung von Graphen auf Muster im Vergleich zu Zahlen, Zeichenketten oder Tupeln, weil hier unter anderem der Substruktur-Isomorphietest, d. h. das Prüfen, ob eine Struktur in einer anderen enthalten ist, NP-vollständig ist. Das bedeutet, dass es keinen Polynomialzeitalgorithmus gibt, der ausgibt, ob eine Substruktur in einer anderen Struktur vorkommt, außer es gilt P = NP.

Graphen sind in der Modellierung komplexer Strukturen wichtig, wie etwa von Schaltungen, Bildern, chemischen Verbindungen, Proteinstrukturen sowie biologischen, technischen oder sozialen Netzwerken.

Hier entstehen tendenziell große, komplexe oder viele Graphen, sodass entsprechende Mining-Methoden notwendig sind. Eine Anwendung ist z. B. das Suchen von häufig auftretenden Molekülgruppen in Proteinen. Die Knoten der Graphen repräsentieren die Atome und die Kanten die Bindungen zwischen ihnen.

In dieser Arbeit wird das so genannte *Frequent-Subgraph-Problem* betrachtet. Es geht hierbei darum, in einer gegebenen Graphenmenge häufig vorkommende Teilgraphen zu finden. Ansätze für die Lösung des Problems sind Algorithmen wie Apriori und Pattern-Growth, von denen einige Varianten vorgestellt werden und eine davon implementiert wird.

Auch werden die Vorteile und Nachteile dieser Algorithmen gegenübergestellt, deren Zeit- und Raumkomplexität untersucht und das Vorgehen bei der Implementierung erläutert.

2 Formale Grundlagen

Die vorliegende Arbeit befasst sich mit Graphen-Algorithmen. Dafür werden im Folgenden notwendige formale Begriffe eingeführt.

2.1 Notationen für Graphen

Definition 2.1. Ein *Graph* G ist ein Tupel (V, E), wobei V eine endliche nichtleere Menge von *Knoten* und die Menge $E \subseteq \{(x, y) \mid x, y \in V\}$ eine Menge von *Kanten* zwischen den Knoten ist.

Definition 2.2. Ein Graph $G = (V, E)$, für den gilt

$$(x, y) \in E \Leftrightarrow (y, x) \in E$$

heißt *ungerichtet*, ansonsten *gerichtet*.

Definition 2.3. Ein ungerichteter Graph heißt *zusammenhängend*, wenn je zwei Knoten durch eine Kantenfolge des Graphen verbunden sind.

Definition 2.4. Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ sind *isomorph*, wenn es eine Abbildung $f : V_1 \to V_2$ gibt, sodass $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$. Ist $G_1 = G_2$, heißt f *Automorphismus*.

Definition 2.5. Ein Graph $G_1 = (V_1, E_1)$ heißt *Teilgraph* oder *Subgraph* von $G = (V, E)$, falls $V_1 \subseteq V$ und $E_1 \subseteq E \cap (V_1 \times V_1)$.

Definition 2.6. Ein *markierter* Graph $G = (V, E, \alpha, \beta)$ ist ein Graph (V, E) und zwei Funktionen α, β, die die Knoten und Kanten jeweils auf die Mengen von *Labels* $L_{V(G)}$ und $L_{E(G)}$ abbilden.

Definition 2.7. Zwei markierte Graphen $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ und $G_2 = (V_2, E_2, \alpha_2, \beta_2)$ sind isomorph, wenn es eine Abbildung $f : V_1 \to V_2$ gibt, sodass $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$, $\alpha_1(v) = \alpha_2(f(v))$ und $\beta_1(u, v) = \beta_2(f(u), f(v))$.

Definition 2.8. Ein markierter Graph $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ heißt Teilgraph oder Subgraph von $G = (V, E, \alpha, \beta)$, falls $V_1 \subseteq V$ und $E_1 \subseteq E \cap (V_1 \times V_1)$ sowie $\alpha(v) = \alpha_1(v)$ für alle $v \in V_1$ und $\beta(u, v) = \beta_1(u, v)$ für alle $(u, v) \in E_1$.

Für die Kennzeichnung von Graphen werden in dieser Arbeit ähnlich zu [HK06] abweichend auch Kleinbuchstaben $g, h, \ldots$ genutzt, wenn verdeutlicht werden soll, dass ein Graph Teilgraph eines größeren ist.

2.2 Breiten- und Tiefensuche

Die folgenden Konzepte werden im Verlauf der Arbeit mehrmals benötigt.

Die **Tiefensuche** startet an einem Knoten v in einem ungerichteten Graphen mit dem Ziel, alle von v erreichbaren Knoten zu besuchen. Die Tiefensuche ist wie eine Wanderung durch ein Labyrinth, deshalb müssen die Wege markiert werden, um den Rückweg zu sichern. Vorgehen: Man verfolgt solange einen Weg, bis man auf eine Markierung oder eine Sackgasse stößt. Dann geht man zur letzten Verzweigung zurück und nimmt dort den nächsten Weg. Der aktuelle Weg wird auf einem Stack gespeichert, der maximal belegte Speicher entspricht der Weglänge zum am weitesten von v entfernten Knoten. Abbldung 2.1 veranschaulicht diesen Prozess.

Die **Breitensuche** besucht ausgehend von einem Knoten v sofort alle zu v adjazenten Knoten, danach die dazu adjazenten Knoten und so weiter, wie aus Abbildung 2.2 zu erkennen ist. Man kann dadurch jedem erreichbaren Knoten w einen „Level“ zuordnen, angefangen mit dem Level 0 für v, 1 für dessen Nachbarknoten und so weiter. Dieser Level ist die Länge des kürzesten Pfades von v nach w. Zur Einhaltung der Reihenfolge wird eine Queue benötigt, in der alle erreichten Knoten gespeichert werden. Der maximal belegte Speicher entspricht der Größe des Levels, das die meisten Knoten enthält.

Bei Algorithmus 1 steht S bei der Breitensuche für eine Queue und bei der Tiefensuche für einen Stack.

Algorithmus 1 : Breiten- bzw. Tiefensuche in Graphen

Eingabe : $G = (V, E)$, ungerichteter Graph
$v_0 \in V$, Startknoten

```
1 S ← {v_0}
2 while S enthält noch einen Knoten v do
3 |   S ← S \ {v}
4 |   foreach v_i ∈ {v_1 ... v_n} unmarkierte Nachbarn von v do
5 |   |   markiere v_i
6 |   |   S ← S ∪ {v_i}
7 |   |   output(v_i)
8 |   end
9 end
```

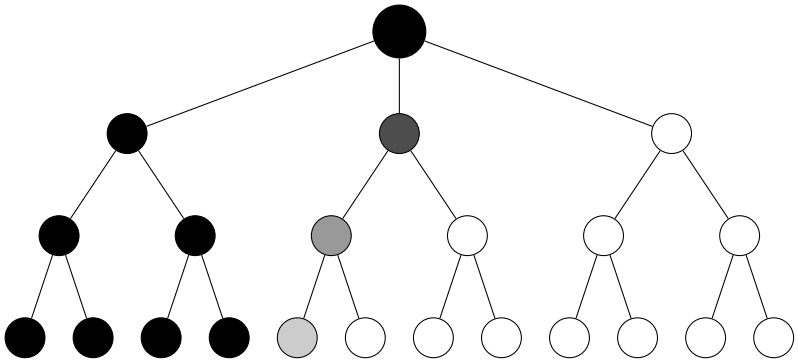


Abbildung 2.1: Tiefensuche

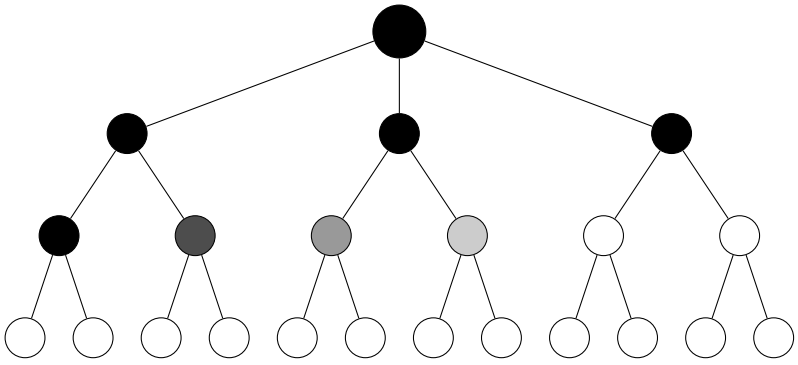


Abbildung 2.2: Breitensuche

2.3 NP-Vollständigkeit und Notwendigkeit für Heuristiken

Probleme werden als NP-*vollständig* bezeichnet, wenn sie in der Komplexitätsklasse NP liegen und zusätzlich NP-*schwer* sind.

In der Klasse NP liegen Probleme, die nichtdeterministisch in Polynomialzeit lösbar sind, d. h. es gibt eine nichtdeterministische Turingmaschine, die eine Instanz des Problems als Eingabe erhält und mit der Ausgabe 1 oder 0 hält. Ihre Laufzeit ist dabei durch ein Polynom der Eingabelänge begrenzt.

Ein Problem A heißt NP-schwer, wenn für alle Probleme $B \in \mathrm{NP}$ gilt, dass $B \leq_m^P A$, das bedeutet, B ist in Polynomialzeit reduzierbar auf A.

Ein für diese Arbeit wichtiges NP-vollständiges Problem ist das Problem des Subgraph-Isomorphismus:

$$\mathrm{SGI} := \{\langle G, h\rangle \mid G \text{ hat einen zu } h \text{ isomorphen Teilgraphen}\}$$

Kein NP-vollständiges Problem ist deterministisch in Polynomialzeit entscheidbar, außer es gilt P = NP. Man nimmt an, dass diese Klassen nicht gleich sind, so dass für NP-schwere Probleme wie z. B. das Subgraph-Isomorphie-Problem Heuristiken zum Einsatz kommen müssen.

Solche werden im Kapitel 4 vorgestellt und im Kapitel 5 wird ihr Einfluss auf die Laufzeit gezeigt.

3 Das Frequent-Subgraph-Problem

Ein wichtiges Problem im Graph Mining ist das Finden von häufigen Teilgraphen in einer großen Menge von Graphen, *Frequent-Subgraph-Mining (FSM)*. Algorithmen für FSM werden in zahlreichen Gebieten angewendet, etwa Bioinformatik, Chemie und Webanalyse. Ein Beispiel mit hoher praktischer Relevanz ist die Entdeckung von aktiven chemischen Teilstrukturen in HIV-Screening-Datensätzen ([HK06]).

Allgemein suchen FSM-Algorithmen in einer Menge von Graphen, auch *Transaktionen* genannt, nach Teilgraphen, die mit einer gewissen Mindesthäufigkeit (ihr *Support*) auftreten. Der Begriff der *Transaktion* entstand, weil die Graphenanalyse aus der Warenkorbanalyse für kommerzielle Anwendungen hervorging ([HK06]).

Nun definieren wir das Frequent-Subgraph-Problem.

Definition 3.1. Sei eine Menge $D = \{G_1, G_2, \ldots, G_n\}$ von markierten Graphen (Transaktionen) gegeben, hier und im Weiteren die *Grundmenge* genannt, und sei h ein markierter Graph. Dann gibt die Funktion $\mathbf{frequency}(h, D)$ an, in wie vielen Graphen aus D ein zu h isomorpher Teilgraph enthalten ist:

$$\mathbf{frequency}(h, D) = |\,\{G_i \in D \mid G_i \text{ hat einen zu } h \text{ isomorphen Teilgraphen }\}\,|$$

Die Funktion $\mathbf{support}(h, D)$ ergibt einen Wert zwischen 0 und 1 und gibt an, wie häufig h in D vorkommt:

$$\mathbf{support}(h, D) = \frac{\mathbf{frequency}(h, D)}{|D|}$$

Wenn D eindeutig ist, kann bei den Funktionen frequency und support der zweite Parameter weggelassen werden.

Definition 3.2. Ein Graph g heißt *häufig* bezüglich der Grundmenge D und einem minimalen Support s, wenn

$$\mathbf{support}(g) \geq s.$$

Diese Arbeit wie auch die Quellen [KK04, IWM00, Wör05] beschränken sich auf die Suche nach zusammenhängenden Teilgraphen, die in der Praxis auch relevanter sind.

Um häufige Teilgraphen zu finden, generiert man zunächst Kandidaten und ermittelt anschließend deren Häufigkeit. Dazu sind jedoch teure Tests auf Subgraph-Isomorphie nötig, für die keine effizienten Algorithmen bekannt sind. Daher unterscheiden sich die Ansätze vor allem bei der Kandidatengenerierung.

Im Folgenden werden neben der trivialen Brute-Force-Suche nach Kandidatengraphen zwei weitere Ansätze zur Kandidatengenerierung vorgestellt, *Apriori* und *Pattern Growth*.

Beispiel 3.3. Abbildung 3.1 verdeutlicht das Frequent-Subgraph-Problem. Die Teilgraphen in der Tabelle weisen den angegebenen Support bezüglich der Datenmenge $D = \{G_1, G_2, G_3\}$ auf. Die verwendeten Labelmengen sind $L_V = \{\bigcirc, \bigcirc, \bullet\}$ und $L_E = \{-, =\}$.

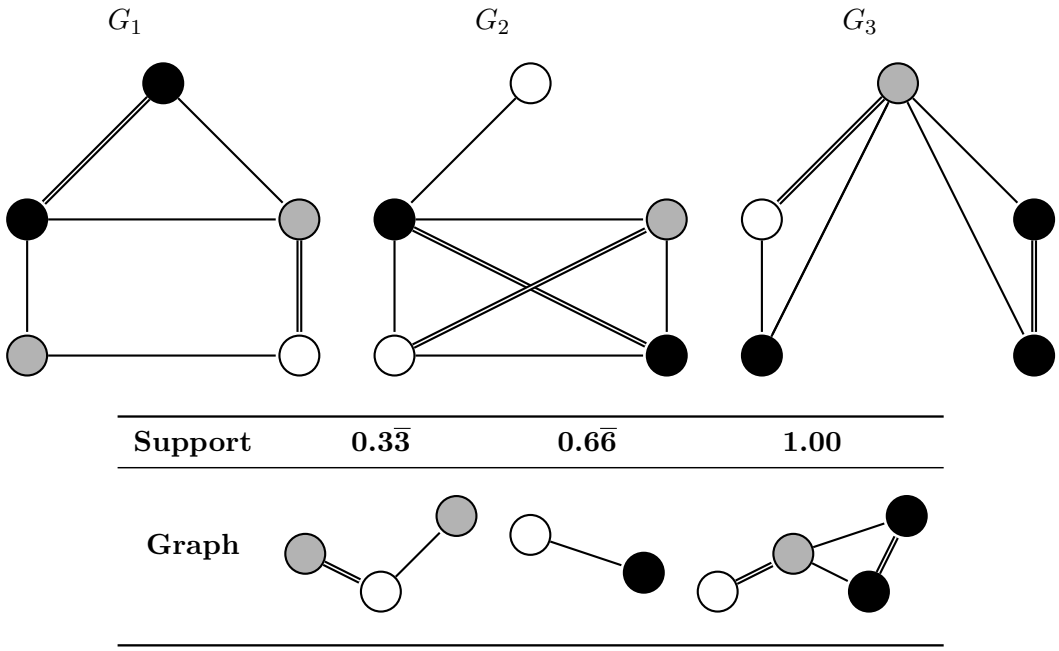


Abbildung 3.1: Beispiel für das Frequent-Subgraph-Problem

3.1 Brute-Force-Suche

Eine Möglichkeit, Kandidaten zu generieren, ist, alle Teilgraphen bis zu der gewünschten Größe zu generieren und nacheinander ihren Support zu berechnen.

Dieses Verfahren hat jedoch gravierende Nachteile:

- Duplikate (zueinander isomorphe Teilgraphkandidaten) werden nicht eliminiert.
- Unzulässige Teilgraphkandidaten (z. B. nicht zusammenhängende Graphen) werden ebenfalls nicht eliminiert.

Eine Abschätzung der Anzahl möglicher Kandidaten wäre folgende: Die Anzahl der möglichen Kanten in einem Graphen mit k Knoten ist begrenzt durch k^2. Da eine Kante vorhanden oder nicht vorhanden sein kann, gibt es beim Durchprobieren aller Kantenkonfigurationen bis zu 2^{k^2} Graphen, bei markierten Graphen noch deutlich mehr.

Die Laufzeit des Brute-Force-Algorithmus ist dadurch sehr hoch. Die nächsten Verfahren umgehen das Problem, da sie den Suchraum verkleinern können und irrelevante Teilgraphen frühzeitig eliminieren.

3.2 Das Apriori-Verfahren

Folgender Ansatz basiert auf dem sogenannten *Apriori*-Lemma.

Lemma 3.4. Wenn ein Graph h bezüglich einer Grundmenge D den Support s hat, dann haben auch alle Teilgraphen von h mindestens Support s.

Beweis. Seien $G_1, \ldots, G_n$ die Graphen aus D, die einen zu h isomorphen Teilgraphen haben, so dass h Support $\geq s$ hat. Jeder Teilgraph von h ist ebenfalls Teilgraph in $G_1, \ldots, G_n$, somit hat jeder Teilgraph von h ebenfalls Support $\geq s$.

Die Idee dabei ist, Kandidaten zu generieren, indem man nur die vorhandenen Kandidaten, die häufig sind, erweitert und somit Graphen ausschließt, die bereits nicht häufige Teilgraphen haben.

Der *Apriori-Algorithmus* generiert stufenweise Kandidaten der Größe $(k+1)$, indem immer zwei Kandidaten g, g' der Größe k miteinander verschmolzen werden, so dass der neue Kandidat zu g und g' isomorphe Teilgraphen enthält. Die *Größe* eines Graphen wird dabei abhängig von der konkreten Implementierung gemessen.

Für jeden neuen Kandidaten in der Stufe $(k+1)$ werden Isomorphietests durchgeführt, auf diese Weise werden Duplikate eliminiert und somit die Anzahl der Kandidaten pro Stufe verringert.

Außerdem werden Teilgraphen, die nicht häufig sind, bereits aus der Menge der Kandidaten der Stufe k ausgeschlossen. Dies ändert nichts an der Vollständigkeit des Suchergebnisses, denn wenn ein Graph g der Größe k nicht häufig ist, dann kann auch kein Graph h der Größe $(k+1)$ mit g als Teilgraph häufig sein. Dies ist die Aussage des Apriori-Lemmas.

So werden viele nicht relevante Graphen gar nicht erst als Kandidaten erzeugt. Dadurch ist der Apriori-Algorithmus im Hinblick auf den Zeitbedarf deutlich besser als die Brute-Force-Suche, dabei findet er trotzdem alle häufigen Teilgraphen in der Grundmenge.

Es folgt der Pseudocode des Apriori-Algoritmus in seiner allgemeiner Form.

Algorithmus 2 : Allgemeiner APRIORI-Algorithmus

Eingabe : $\langle D, s \rangle$
$D = \{G_1, \ldots, G_n\}$ ist eine Grundmenge von markierten Graphen,
$s \in \mathbb{Q}$ ist der minimale Support für häufige Teilgraphen.
Ausgabe : S, die Menge der häufigen Teilgraphen

```
1  k ← 1
2  S_1 ← Menge der häufigen markierten Teilgraphen der Größe 1
3  while S_k ≠ ∅ do
4  |  S_(k+1) ← ∅
5  |  foreach g_i ∈ S_k do
6  |  |  foreach g_j ∈ S_k do
7  |  |  |  foreach durch Verschmelzung von g_i und g_j gewonnenen Graphen m der
   |  |  |  Größe (k + 1) do
8  |  |  |  |  if support(m, D) ≥ s und kein m' ∈ S_(k+1) ist isomorph zu m then
9  |  |  |  |  |  S_(k+1) := S_(k+1) ∪ {m}
10 |  |  |  |  end
11 |  |  |  end
12 |  |  end
13 |  end
14 |  k ← k + 1
15 end
16 return S = S_1 ∪ ... ∪ S_k
```

Es gibt folgende spezielle Algorithmen, die auf dem Apriori-Ansatz basieren und sich in der Kombinationsmethode der Teilgraphen der Stufe k unterscheiden:

- *Apriori-based Graph Mining (AGM)*, knotenbasierte Verschmelzung ([IWM00]),
- *Frequent Subgraph Discovery (FSG)*, kantenbasierte Verschmelzung ([KK04]),
- *Path-join-Methode*, pfadbasierte Verschmelzung ([HK06]).

AGM und FSG werden in dieser Arbeit genauer beschrieben und FSG wird implementiert.

3.2.1 Apriori-based Graph Mining (AGM)

Der AGM-Algorithmus ist eine Methode der Kandidatengenerierung, die mit knotenbasierter Verschmelzung arbeitet. Die Größe der häufigen Teilgraphen wird in jeder Stufe um einen Knoten erhöht. Dabei werden zwei häufige Teilgraphen mit k Knoten nur dann miteinander verschmolzen, wenn sie einen gemeinsamen *Kern*, d. h. einen gemeinsamen

Teilgraphen mit genau $(k-1)$ Knoten, haben.

Der neu generierte Kandidat enthält den Kern und die zwei restlichen Knoten, die nicht im Kern liegen, aus den beiden verschmolzenen Kandidaten mit jeweils k Knoten. Es ist nicht festgelegt, ob es zwischen den beiden Knoten, die außerhalb des Kerns liegen, eine Kante geben muss. Deshalb entstehen pro Möglichkeit die Graphen zu verschmelzen zwei neue Kandidaten der Größe $(k+1)$, einer mit und einer ohne diese Kante.

Schwierig dabei ist das Finden des gemeinsamen Kerns von zwei Kandidaten. Dies ist laut [Wel11] genau wie SGI ein NP-vollständiges Problem:

$$\textsc{Common-SGI} := \{\langle G_1, G_2, k\rangle \mid G_1, G_2 \text{ haben einen gemeinsamen Kern mit } \geq k \text{ Knoten}\}$$

Für markierte Graphen ist das Problem ebenso schwer.

Wir betrachten nun den AGM-Algorithmus für die Verschmelzung zweier Graphen mit k Knoten für die Generierung neuer Kandidaten mit $k+1$ Knoten.

Algorithmus 3 : Knotenbasiertes Verschmelzen (AGM)

Eingabe : $\langle g_1, g_2\rangle$ sind häufige Teilgraphen der Stufe k, d. h. mit jeweils k Knoten
Ausgabe : M, die Menge der generierten Kandidaten

```
1  M ← ∅
2  foreach Kern h von g_1 und g_2 do
3  |   Entferne den Kern h aus g_1, g_2:
4  |   g'_i ← g_i \ h
5  |   m' ←  Graph, der durch Hinzufügen von g'_2 zu g_1 entsteht
6  |           so dass m' \ g'_1 isomorph zu g_2
7  |   M ← M ∪ {m'}
8  |   M'' ← {m'' | m'' ist Graph, der durch Hinzufügen einer Kante e zu m' entsteht,
9  |             die die Knoten aus g'_1 und g'_2 verbindet, β(e) ∈ L_E}
10 |   M ← M ∪ M''
11 end
12 return M
```

In Schritt 8 ist zu beachten, dass es mehr als eine Markierung für Kanten geben kann, d. h. dass bei der Generierung von m'' mehrere Graphen entstehen können. Bei l möglichen Labels entstehen $l+1$ neue Graphen.

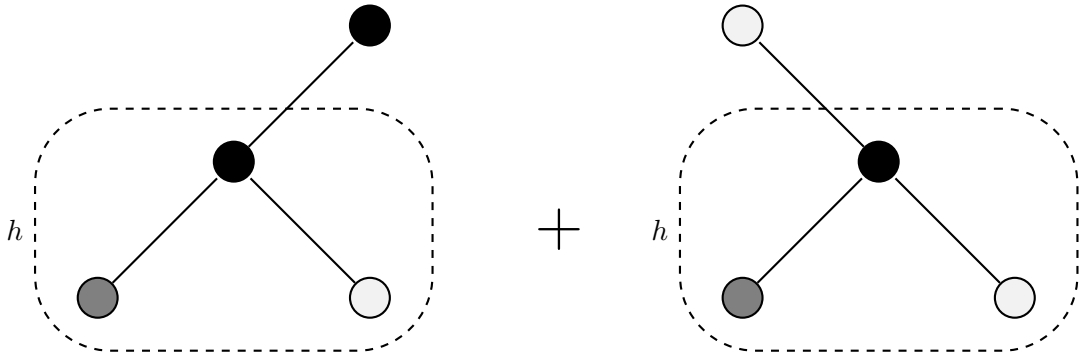


Abbildung 3.2: Zwei Graphen mit Kern h und $k = 4$ Knoten

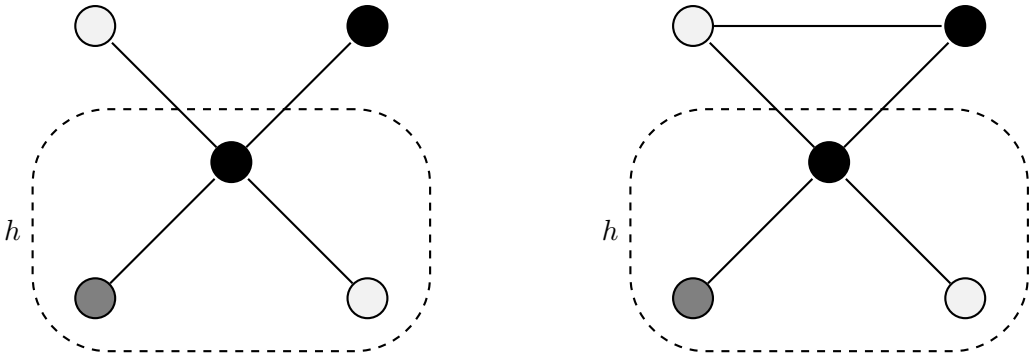


Abbildung 3.3: Neue Kandidaten mit $k + 1 = 5$ Knoten

Das Verschmelzen zweier Graphen wird an einem konkreten Beispiel dargestellt. In Abbildung 3.2 sind zwei Graphen mit $k = 4$ Knoten und mit einem gemeinsamen Kern h zu sehen. Diese werden miteinander zu den in Abbildung 3.3 gezeigten Graphen mit $k + 1 = 5$ Knoten verschmolzen.

3.2.2 Frequent Subgraph Discovery (FSG)

Im Gegensatz zu AGM ist der FSG-Algorithmus eine kantenbasierte Methode, Kandidaten zu generieren. Das heißt, dass mit jeder Stufe die Anzahl der Kanten anstatt die der Knoten erhöht wird. Dass jeweils zwei Kandidaten mit k Kanten nur dann miteinander verschmolzen werden, wenn sie einen gemeinsamen Kern haben, ist identisch zu AGM, nur dass der Kern hier $k - 1$ Kanten haben muss.

Der neu generierte Kandidat enthält den Kern und die zwei Kanten aus den beiden verschmolzenen Kandidaten mit jeweils k Kanten.

Wenn die Knoten, die außerhalb des Kerns liegen, das gleiche Label haben, gibt es mehrere Möglichkeiten, Kandidaten mit $(k + 1)$ Kanten zu generieren, da Knoten mit

gleichen Labels verschmolzen werden können. Das wird in der Abbildung 3.4 sowie 3.5 dargestellt. Knoten mit verschieden Labels dürfen nicht verschmolzen werden.

Nachdem ein Kandidat generiert wurde, wird wie auch beim AGM-Algorithmus seine Häufigkeit geprüft und er wird in die Menge der häufigen Teilgraphen eingefügt, wenn er häufig genug ist und nicht bereits ein zu ihm isomorpher Kandidat (ein *Duplikatgraph*) in der Menge enthalten ist.

Folgendes Beispiel verdeutlicht das Verschmelzen zweier Graphen mit k Kanten. Dadurch, dass die Graphen außerhalb des Kerns Knoten mit gleichen Labels besitzen, kann die Knotenanzahl stagnieren oder inkrementiert werden.

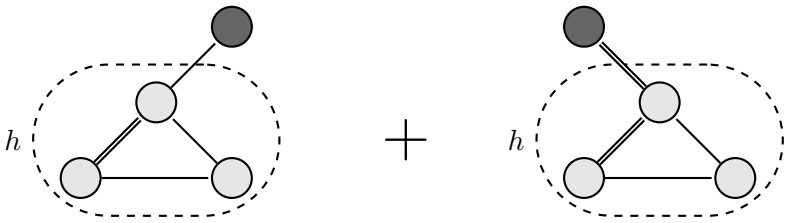


Abbildung 3.4: Zwei Graphen mit Kern h und k Kanten

In Abbildung 3.4 sind zwei Graphen mit k Kanten und mit einem gemeinsamen Kern h zu sehen. Diese werden miteinander zu den in Abbildung 3.5 gezeigten Graphen mit $k+1$ Kanten verschmolzen. Dabei besitzt der Kern mehrere Automorphismen, so dass die neue Kante zu verschiedenen seiner Knoten hinzugefügt werden kann. Es ist $L_V = \{\bigcirc, \bullet\}$ und $L_E = \{-, =\}$.

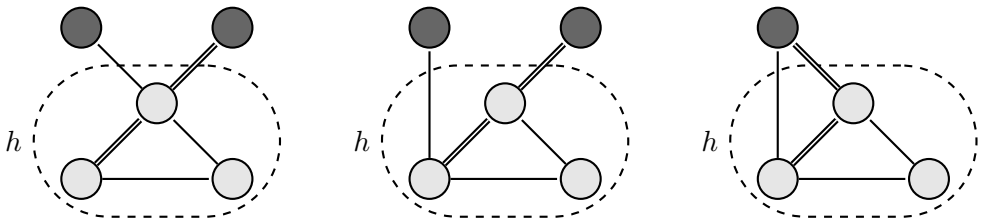


Abbildung 3.5: Neue Kandidaten mit $k+1$ Kanten

3.2.3 Verbesserungen der Apriori-Verfahren

Erweitertes Apriori-Lemma

Das Apriori-Lemma besagt, dass der Support eines Kandidaten durch den Support seiner Teilgraphen beschränkt wird. Man kann diese Aussage aber auch noch wie folgt verallgemeinern: Ein Kandidat kann nur in den Graphen als Subgraph enthalten sein, die auch alle seine Teilgraphen enthalten. Das kann man im Algorithmus ausnutzen:

Die Häufigkeitszählung eines Kandidaten soll nur in den Transaktionen der Grundmenge geschehen, in denen bereits seine beiden Elterngraphen als Subgraphen enthalten sind.

Dazu wird stets die Liste der Transaktionen, in denen ein Kandidaten-Graph häufig ist, gespeichert, in [KK04] auch *TID-Listen* genannt. Beim Mergen von zwei Kandidaten wird die Schnittmenge der Transaktionslisten gebildet und nur dort ein Test auf Subgraph-Isomorphie durchgeführt, denn ein Kandidat kann nur in Transaktionen enthalten sein, in denen alle seine Teilgraphen bereits enthalten waren und damit auch die beiden Elternkandidaten.

Primäre Kerne bei FSG

Bei der Kandidatengenerierung mit Apriori-Algorithmen werden, wie aus Abschnitt 3.2.2 bekannt, zwei häufige Kandidaten mit der Stufe k mit einem gemeinsamen Kern der Größe $(k-1)$ miteinander verschmolzen. Dabei hat jedoch jeder Kandidat durch das Weglassen von einem der k Knoten bzw. Kanten bis zu k potentielle Kerne im Bezug auf einen Verschmelzungspartner, die alle betrachtet werden.

Da für das Verschmelzen alle k Kerne eines Kandidaten mit allen Kernen des anderen Kandidaten auf Isomorphie geprüft werden, hat die Prozedur durch k^2 Isomorphietests eine sehr hohe Laufzeit.

FSG löst dieses Problem sehr effizient und elegant, indem nur zwei von allen möglichen Kernen eines Kandidaten in Betracht gezogen werden.

Sei F ein Kandidat der Größe k, und $\Theta(F) = \{H_1, H_2\}$ zwei Kerne von F, so dass H_1 das kleinste und H_2 das zweitkleinste so genannte *kanonische Labeling* haben. Dieses Labeling ist ein String, der einen Graphen eindeutig beschreibt und sich lexikographisch ordnen lässt (siehe Abschnitt 4.1.1). Diese Teilgraphen von F werden als *primäre Kerne* bezeichnet. Von jedem Kandidaten werden nun jeweils nur die beiden primären Kerne zur Generierung der Kandidaten der nächsten Stufe verwendet. Auf diese Weise werden laut [KK04] trotzdem noch alle gültigen Kandidaten der Stufe $(k+1)$ erzeugt.

Dieser Ansatz der Kandidatengenerierung reduziert die Anzahl der Duplikate und führt zu einer deutlichen Leistungsverbesserung gegenüber dem naiven Ansatz mit k Kernen.

Die Berechnung des kanonischen Labelings jedes Kernes ist zwar zeitaufwendig, aber diese kann dann für alle Isomorphietests angewendet werden, welche damit in Linearzeit

durchgeführt werden können. Dieses Verfahren wird in Abschnitt 4.1.1 erläutert.

3.3 Pattern-Growth-Ansätze

Die Apriori-Algorithmen verfolgen bei der Kandidatengenerierung das Prinzip der Breitensuche. Kandidaten der Stufe $k+1$ erhält man, indem man zwei häufige Teilgraphen der Stufe k miteinander verschmilzt (Abb. 3.6).

Bei der Pattern-Growth-basierten Suche werden, wie auch bei dem Apriori-Ansatz, zunächst Kandidaten generiert und anschließend deren Häufigkeiten geprüft. Während ein Kandidat bei Apriori durch Mergen zweier Elterngraphen entsteht, hat bei Pattern-Growth ein Kandidat G' nur einen Elterngraphen G. Hier wird G um eine Kante e erweitert, um G' zu generieren. Diese verbindet entweder zwei aus G übernommene Knoten oder einen Knoten aus G und einen neu in G' hinzugefügten Knoten. Wenn der erweiterte Teilgraph häufig ist, wird er als neuer Elterngraph verwendet, um den Algorithmus rekursiv aufzurufen (Abb. 3.7).

Im Gegensatz zu Apriori kann bei Pattern-Growth-Ansätzen sowohl die Breitensuche als auch Tiefensuche verwendet werden, um die Kandidaten im Suchraum zu generieren. Dabei verbraucht die Tiefensuche weniger Speicher, weil es ausreicht, sich die Kette π der Vorfahren des aktuellen Kandidaten zu merken und Backtracking einzusetzen, wenn der erweiterte Graph nicht mehr zu einem häufigen Teilgraphen führt (Abb. 3.7).

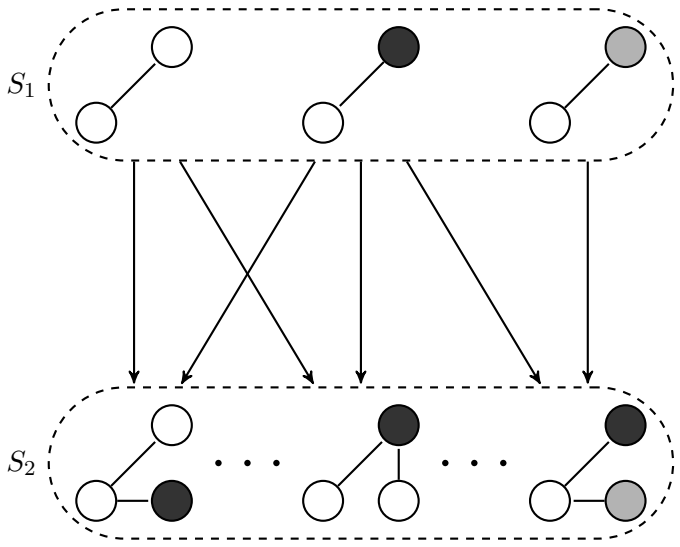


Abbildung 3.6: Prinzip der Breitensuche bei Apriori

Der allgemeine Algorithmus Pattern-Growth-Graph [HK06] erweitert rekursiv jeden

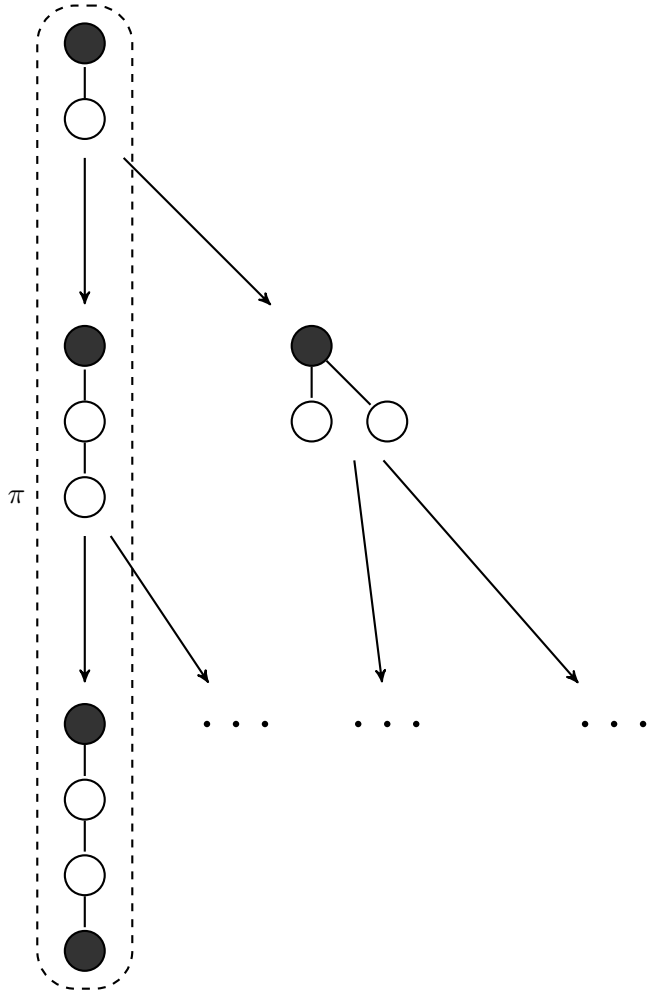


Abbildung 3.7: Prinzip der Tiefensuche bei Pattern-Growth

häufigen Teilgraphen g der Stufe k, um alle möglichen Kandidaten der nächsten Stufe zu finden, die g als Teilgraph enthalten. Der Algorithmus ist jedoch nicht effizient, da er Kandidaten auch mehrfach speichert. Das kommt dadurch zustande, dass es für einen Subgraphen mit k Kanten bis zu k Möglichkeiten gibt, aus einem Graphen mit $k-1$ Kanten entstanden zu sein.

Optimal wäre, häufige Teilgraphen so zu erweitern, dass jeder Kandidat nur einmal entsteht. Dieses Ziel wird von dem Algorithmus gSpan erreicht.

3.3.1 gSpan

Der Algorithmus gSpan implementiert die Pattern-Growth-Methode. Das Problem der unnötigen Berechnung von Duplikaten wird hier deutlich geschmälert, indem Duplikatgraphen nicht erweitert werden. Dies wird durch das Berechnen von *minimalen DFS-Codes* realisiert, die für jeden Graphen eindeutig sind.

Der minimale DFS-Code (*Depth-First-Search-Code*) eines Graphen ist eine eindeutige Darstellung und wird durch einen Tiefensuchendurchlauf gewonnen, während dessen Verlauf die Knoten nummeriert werden. Der Startknoten eines Graphen mit k Kanten wird dabei v_0 genannt, der zuletzt erreichte Knoten entsprechend v_n oder auch *rightmost vertex*. Der direkte Pfad zwischen diesen beiden Knoten wird als *rightmost path* bezeichnet.

Durch die Nummerierung wird eine Ordnung innerhalb der DFS-Codes sichergestellt, die später zum Ermitteln des minimalen Codes eines Graphen benötigt wird. Graphen, die nicht in ihrer DFS-Minimalform vorliegen, werden als Duplikatgraphen entfernt. Jede Erweiterung um eine neue Kante findet auf dem *rightmost path* statt, dies wird auch *Rechtserweiterung (rightmost extension)* genannt. Hierbei dürfen keine Multikanten entstehen. Abbildung 3.8 demonstriert dieses Prinzip. Trotz der Einschränkung auf Rechtserweiterungen werden nach [IWM00, YH02] alle häufigen Teilgraphen gefunden.

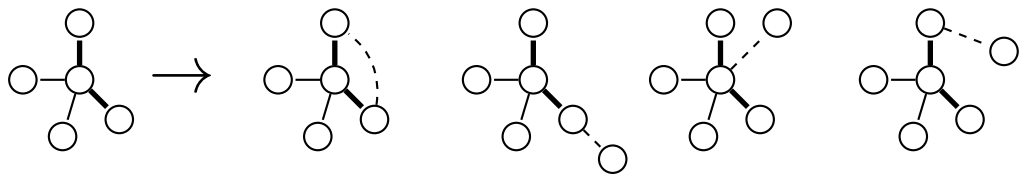


Abbildung 3.8: Mögliche Rechtserweiterungen eines Graphen

Algorithmus 4 : gSpan

Eingabe : $\langle D, s, S, g \rangle$
$D = \{G_1, \ldots, G_n\}$ ist eine Grundmenge von markierten Graphen,
$s \in \mathbb{Q}$ ist der minimale Support für häufige Teilgraphen,
$S \leftarrow \emptyset$ ist die Menge der häufigen Teilgraphen,
g ist der Ursprungsgraph.

```
1  if dfs(g) ≠ g then
2  |  return ;                                  /* g ist nicht in Minimalform */
3  end
4  S ← S ∪ {g}
5  C ← ∅ ;                                                /* Nachfahren von g */
6  foreach Kante e einer Rechtserweiterung von g do
7  |  m ← g erweitert um e
8  |  if m ist häufig then
9  |  |  C ← C ∪ {m}
10 |  end
11 end
12 Sortiere C nach DFS-Codes;
13 foreach m ∈ C do
14 |  gSpan(D, s, S, m)
15 end
```

3.4 Vergleich der existierenden Ansätze

Die Algorithmen der Apriori-Ansätze unterscheiden sich bezüglich der Laufzeit deutlich. Der FSG-Algorithmus ist wegen der Eigenschaft, dass jeder Kandidat aus den primären Kernen seiner Elterngraphen erzeugt werden kann, effizienter.

Im Vergleich dazu werden bei AGM für die Generierung der Kandidaten alle Kerne der Elterngraphen verwendet, da der Satz aus [KK04] nur für kantenbasierte Verschmelzung gilt. Außerdem entstehen mehr Duplikatgraphen, weil alle Kerne, auch zueinander isomorphe, für die Verschmelzung benutzt werden. Damit ist der Aufwand für das Entfernen der Duplikate größer.

Während FSG und gSpan bei der Anwendung auf realen Moleküldatenbanken [KK04] zufolge eine vergleichbare Laufzeit haben, ist gSpan nach [YH02] bei synthetischen, d. h. zufällig generierten, Transaktionsgraphen effizienter. Abbildung 3.9 entstammt [YH02] und zeigt die unterschiedlichen Laufzeiten auf einem synthetischen Datensatz abhängig vom gegebenen Support.

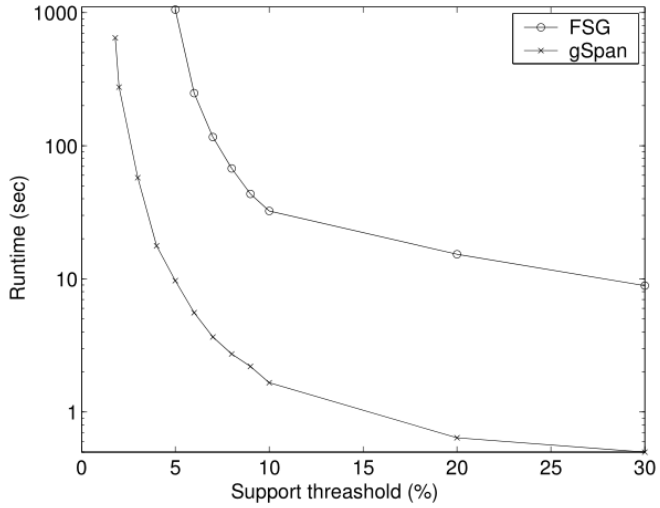


Abbildung 3.9: Vergleich der Laufzeiten von FSG und gSpan

4 Heuristiken für Isomorphietests

4.1 Isomorphie

In den vorgestellten Algorithmen ist Graphen-Isomorphie ein wichtiges Problem, da Isomorphietests beim Suchen nach häufigen Teilgraphen bei der Berechnung des Supports und beim Prüfen, ob ein neu gefundener Kandidat schon existiert, oft durchgeführt werden müssen.

Bei einem einfachen Isomorphietest werden alle möglichen Abbildungen von einer Knotenmenge auf die andere Knotenmenge dahingehend überprüft, ob sie ein Isomorphismus sind, d. h. ob sie die Kanten und Nichtkanten erhalten. Bei gelabelten Graphen müssen auch die Labels der Knoten und Kanten auf Gleichheit geprüft werden. Dieser Test hat den Nachteil, dass es $|V|!$ verschiedene Abbildungen zu prüfen gibt.

Eine mögliche Heuristik besteht darin, dass man die Knoten anhand so genannter *Invarianten* in Partitionen einteilt, d. h. anhand von Eigenschaften von Knoten, die invariant unter Isomorphieabbildungen sind. Dies hat zur Folge, dass weniger Abbildungen geprüft werden müssen, weil nur noch innerhalb der Partitionsmengen permutiert werden muss.

Eine weitere Möglichkeit, die Laufzeit zu verbessern, ist, die kanonische Kodierung eines Graphen G zu berechnen, die für alle zu G isomorphen Graphen gleich ist.

4.1.1 Kanonische Kodierungen

Definition 4.1. Eine *kanonische Kodierung* (oder auch kanonisches Labeling) cl ist eine Funktion, die einen Graphen auf ein Wort über einem Alphabet Σ eindeutig abbildet.

$$cl : \mathcal{G} \rightarrow \Sigma^*$$

Hierbei ist $\mathcal{G}$ die Menge aller ungerichteten Graphen und Σ ein beliebiges Alphabet. Σ^* ist dann die Menge aller Wörter über Σ. Die kanonische Kodierung muss dabei folgende Eigenschaft für beliebige $G, H \in \mathcal{G}$ erfüllen:

$$cl(G) = cl(H) \Leftrightarrow G \text{ ist isomorph zu } H$$

Eine kanonische Kodierung für markierte Graphen kann man z. B. über die Adjazenzmatrix definieren.

Diese kann man für markierte Graphen wie folgt kodieren:

$$\begin{array}{c} \\ v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & \dots & v_n \end{array} \\ \begin{pmatrix} - & \beta'(v_1,v_2) & \dots & \beta'(v_1,v_n) \\ \beta'(v_2,v_1) & - & \dots & \beta'(v_2,v_n) \\ \vdots & \vdots & \ddots & \vdots \\ \beta'(v_n,v_1) & \beta'(v_n,v_2) & \dots & - \end{pmatrix} \end{array}$$

Dabei ist β' definiert als

$$\beta'(u,v) = \begin{cases} 0 & \text{für } (u,v) \notin E \\ \beta(u,v) & \text{sonst} \end{cases}$$

Zunächst werden für die kanonische Kodierung die Labels aller Knoten aufgelistet. Dahinter werden alle Zeilen der oberen Dreiecksmatrix der Adjazenzmatrix ohne die Hauptdiagonale konkateniert. Es reicht die obere Dreiecksmatrix, da Adjazenzmatrizen ungerichteter Graphen immer symmetrisch sind.

$$cl(G) = \alpha(v_1) \circ \dots \circ \alpha(v_n) \circ \beta'(v_1,v_2) \circ \dots \circ \beta'(v_{n-1},v_n)$$

Wenn das Alphabet $\Sigma = L_V \cup L_E \cup \{0\}$ ist und L_V und L_E die jeweiligen Labelmengen sind, dann ist der konkatenierte Code aus Σ^*.

Damit diese Abbildung eine kanonische Kodierung ist, wählt man die Nummerierung der Knoten $v_1 \dots v_n$ so, dass der entstehende Code lexikographisch minimal ist. Dadurch ist der Code isomorpher Graphen gleich, da jeder Isomorphismus eine Knotenpermutation ist. Bei einer Durchsuchung aller Permutationen nach dem minimalen Code ist die Berechnungslaufzeit wieder mindestens $|V|!$ und lohnt sich deshalb nicht für einen einzigen Isomorphietest.

Bei den Apriori-Verfahren wird die Laufzeit trotzdem geringer, da man für jeden Graphen viele Isomorphietests durchführt. Um in die Kandidatenmenge S_k der Stufe k einen neuen Graphen einzufügen, sind $|S_k|$ Isomorphietests notwendig, für die gesamte Berechnung von S_k damit $\frac{|S_k|^2}{2}$.

Die Laufzeit mit direkten, naiven Isomorphietests beträgt für die Berechnung der Stufe S_k (ohne Laufzeit für Kandidatengenerierung):

$$T(k) = \frac{|S_k|^2}{2} \cdot O(k!)$$

Mit Vorberechnung des kanonischen Labelings ist die Laufzeit

$$T(k) = |S_k| \cdot O(k!) + \frac{|S_k|^2}{2} \cdot O(k^2).$$

Dabei ist $\frac{|S_k|^2}{2}$ die Anzahl aller Isomorphietests. Jeder Test benötigt $O(k^2)$ Schritte, da dies die maximale Länge eines kanonischen Labelings für Graphen mit k Kanten ist und die Codes verglichen werden müssen. Zusätzlich wird für jeden Graphen der Stufe k eine Laufzeit von $O(k!)$ für die Generierung des *cl*-Codes benötigt.

4.1.2 Isomorphieinvarianten

Die Berechnung des kanonischen Labelings hat bei Graphen mit k Knoten eine Laufzeit von $k!$, aber es können Invarianten genutzt werden, um Knoten in bestimmte Partitionen einzuordnen.

Invarianten sind Eigenschaften von Knoten und Kanten, die sich bei Isomorphie-Abbildungen nicht ändern. Dies sind z. B. Knotenlabels und Knotengrade, aber nicht die Knotennummerierung. Die Partitionierung reduziert die Anzahl der möglichen Permutationen drastisch. Wenn bei einem Graphen mit k Knoten die Knoten in Partitionen $(V_1, \ldots, V_m)$ eingeteilt werden, dann gibt es

$$\prod_{i=1}^{m} |V_i|!$$

Permutationen, und das ist weniger als $k!$, wenn es mindestens zwei Partitionen gibt.

Für einen Graphen mit beispielsweise 4 Knoten sind $4! = 24$ Permutationen möglich. Der Graph habe nur drei verschiedene Knotengrade. Durch die Bildung von 3 Gruppen über diese Invariante reduziert sich die Anzahl auf $1! \cdot 2! \cdot 1! = 2$ Permutationen.

Ein Nachteil dabei ist, dass bei der Berechnung des *cl*-Codes die Reihenfolge der Partitionen selbst noch permutiert werden muss, das ist ein zusätzlicher Faktor von $m!$. Dieser Worst-Case-Faktor wird in der Praxis umgangen, indem die Partitionen mit verschiedenen α-Labels bereits in eine festgelegte Reihenfolge gebracht werden.

4.1.3 Iterative Partitionierung

Eine feinere Partitionierung ist möglich, wenn man außer dem Knotenlabel die Nachbarliste des Knotens als Invariante wählt. Diese enthält Informationen über die Labels der Nachbarknoten sowie die Kantenlabels zwischen dem Knoten und seinen Nachbarn.

Basierend auf der Nachbarliste kann eine weitere Optimierung namens *iterative Partitionierung* ausgeführt werden. Die Nachbarlisten werden hier anders kodiert: Labels der Nachbarn werden ersetzt durch seine Partitionsnummer. Nur Knoten, deren Nachbarn in den gleichen Partitionen liegen, können selbst eine Partition bilden.

Zunächst werden die Knoten anhand der einfachen Invarianten, d. h. nicht auf die Nachbarn bezogen, partitioniert. Es können jedoch in einer Partition Knoten enthalten sein, die verschiedene Nachbarn haben, d. h. Nachbarn aus verschiedenen Partitionen. In

dem Fall wird rekursiv weiter partitioniert, bis alle Knoten einer Partition die gleiche Nachbarliste haben.

Die *Signatur* $\sigma(v)$ eines Knotens v ist die Zusammenfassung aller Invarianten von v.

Der folgende Algorithmus berechnet die iterative Partitionierung von Knoten eines beliebigen markierten, ungerichteten Graphen.

Algorithmus 5 : IterativePartitionierung

Eingabe : $G = (V, E, \alpha, \beta)$, ein markierter Graph mit $V = \{v_1, \ldots, v_n\}$

Ausgabe : $(V_1, \ldots, V_m,\ \sigma_1, \ldots, \sigma_m)$, Liste der Partitionen und deren Partitionssignaturen

```
1  m ← 1                                                  /* Anzahl der Partitionen */
2  Part ← int[1..n]
3  foreach v ∈ V do
4  |  Part[i] = 1
5  end
6  while ∃ i, j, i ≠ j  und Part[i] = Part[j] und σ(v_i) ≠ σ(v_j) do
7  |  m ← m + 1                                           /* Partitionen aufspalten */
8  |  foreach v_k mit σ(v_k) = σ(v_j) do
9  |  |  Part[k] = m + 1
10 |  end
11 end
12 for i = 1 to n do
13 |  V_Part[i] ← V_Part[i] ∪ {v_i}                      /* Ausgabeliste wird erstellt */
14 end
15 for i = 1 to m do
16 |  σ_i = σ(V_i)
17 end
18 return (V_1, ..., V_m, σ_1, ..., σ_m)
```

Dabei wird die Signatur $\sigma(v)$ formal definiert als

$$\sigma(v) = \left(\alpha(v); \left\{\left(Part[i_1]; \beta(v, v_{i_1})\right), \ldots, \left(Part[i_k]; \beta(v, v_{v_k})\right)\right\}\right),$$

und $\{v_{i_1}, \ldots, v_{i_k}\}$ sind die Nachbarn von v.

Beispiel 4.2. Es folgt ein Beispiel der Partitionierung eines Graphen aus Abbildung 4.1.

In der Tabelle werden die Partitionsnummern jedes Knotens des Graphen in jedem Durchlauf der **while**-Schleife aufgelistet. Da es im letzten Schritt genau so viele Partitionen gibt wie Knoten im Graphen, kann nicht weiter partitioniert werden und der Algorithmus terminiert.

Knoten	$\boldsymbol{P}[\cdot]^{(1)}$	$\boldsymbol{P}[\cdot]^{(2)}$	$\boldsymbol{P}[\cdot]^{(3)}$	$\boldsymbol{P}[\cdot]^{(4)}$	$\boldsymbol{P}[\cdot]^{(5)}$	$\boldsymbol{P}[\cdot]^{(6)}$	$\boldsymbol{P}[\cdot]^{(7)}$
$\boldsymbol{v_1}$	1	1	1	1	1	1	**1**
$\boldsymbol{v_2}$	1	2	2	2	2	2	**2**
$\boldsymbol{v_3}$	1	1	3	3	3	3	**3**
$\boldsymbol{v_4}$	1	1	1	4	4	4	**4**
$\boldsymbol{v_5}$	1	1	1	4	4	4	**7**
$\boldsymbol{v_6}$	1	1	1	1	5	5	**5**
$\boldsymbol{v_7}$	1	1	1	1	1	6	**6**

Entsprechende Signaturen der Knoten im letzten Schritt sind:

$$\sigma(v_1) = \Big(\bullet\,; \big\{\big(2\,;y\big), \big(3\,;z\big), \big(4\,;y\big), \big(7\,;y\big)\big\}\Big)$$

$$\sigma(v_2) = \Big(\textcolor{gray}{\bullet}\,; \big\{\big(1\,;y\big), \big(6\,;y\big)\big\}\Big)$$

$$\sigma(v_3) = \Big(\bullet\,; \big\{\big(1\,;z\big), \big(6\,;x\big)\big\}\Big)$$

$$\sigma(v_4) = \Big(\textcolor{gray}{\bullet}\,; \big\{\big(1\,;y\big), \big(5\,;z\big), \big(7\,;x\big)\big\}\Big)$$

$$\sigma(v_5) = \Big(\textcolor{gray}{\bullet}\,; \big\{\big(1\,;y\big), \big(4\,;x\big), \big(6\,;z\big)\big\}\Big)$$

$$\sigma(v_6) = \Big(\circ\,; \big\{\big(2\,;y\big), \big(4\,;z\big)\big\}\Big)$$

$$\sigma(v_7) = \Big(\circ\,; \big\{\big(3\,;x\big), \big(7\,;z\big)\big\}\Big)$$

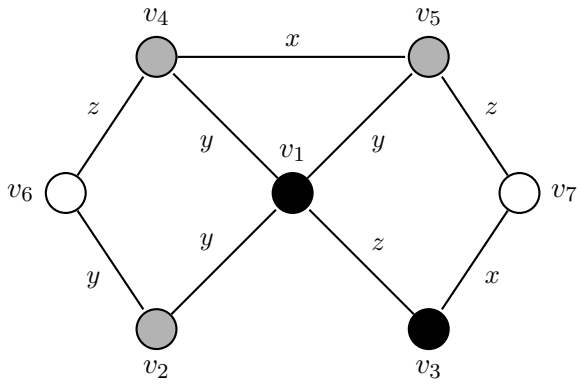


Abbildung 4.1: Ein markierter Graph G

4.2 Subgraphisomorphie

Um den Support für einen Kandidaten zu berechnen, muss geprüft werden, in wie vielen Graphen aus der Datenmenge er als Teilgraph enthalten ist.

Bei gleich großen Graphen, wie z. B. für die Eliminierung von Duplikaten der Kandidaten einer Stufe, können Isomorphie-Tests über das kanonische Labeling durchgeführt werden. Das gilt bei Subgraphisomorphietests aber nicht, die *cl*-Codes können hier nicht genutzt werden, weil diese bei Graphen verschiedener Größe nicht gleich lang sind. Zudem kommt $cl(H)$ nicht unbedingt als Teilstring in $cl(G)$ vor, selbst wenn H in G vorkommt.

Man kann einen Subgraphisomorphismus $f : H \mapsto G$ mit der Brute-Force-Methode berechnen, die für jeden Knoten aus H alle Knoten aus G als Abbildungspartner testet und eine Laufzeit von

$$\frac{|V_G|!}{(|V_G| - |V_H|)!}$$

hat.

Der Algorithmus NachbarListenPartitionierung hat das Ziel, diese Laufzeit zu verbessern, indem Knoten der Graphen nach Isomorphieinvarianten zu Partitionen zusammengefasst werden.

Im Gegensatz zu der iterativen Partitionierung werden hier die Partitionen der Nachbarknoten für die Unterscheidung der Knoten nicht betrachtet. Das bedeutet, dass in einer Partition durchaus Knoten enthalten sein können, deren Nachbarknoten in unterschiedlichen Partitionen sind. Allgemein sind die Nachbarn keine Invariante unter Subgraphisomorphie.

Die Partitionierung der Knoten wird daher nicht nach der Anzahl der Nachbarn vorgenommen, der Grad eines Knotens v aus H kann also vom Grad eines Knotens v' aus G verschieden sein, auf den v abgebildet wird. Jedoch darf der Grad von v nicht

größer sein als der von v'.

Seien $\sigma(v), \sigma(v')$ die Signaturen der Knoten v, v'. Dann schreibe $\sigma(v) \subseteq \sigma(v')$, wenn $\alpha(v) = \alpha(v')$ und alle Labelpaare $(\alpha(u), \beta(v, u))$ aus σ für Nachbarn u von v auch als Paare in σ' vorkommen. Dann muss gelten, dass $\sigma(v \in H) \subseteq \sigma(v' \in G)$ ist, wenn es einen Subgraph-Isomorphismus von H nach G gibt, der v auf v' abbildet.

Ein Brute-Force-Algorithmus, der für einen Knoten v nur passende Abbildungspartner testet, hat im Durchschnitt eine geringere Laufzeit. Es gibt jedoch Graphen, bei denen alle Knoten in einer Partition sind, daher lässt sich die Worst-Case-Laufzeit so nicht verbessern.

Der folgende Algorithmus ordnet Knoten eines beliebigen markierten, ungerichteten Graphen zu Partitionen.

Algorithmus 6 : NachbarListenPartitionierung

Eingabe : $G = (V, E, \alpha, \beta)$, ein markierter Graph mit $V = \{v_1, \ldots, v_n\}$

Ausgabe : $(V_1, \ldots, V_m,\ \sigma_1, \ldots, \sigma_m)$, Liste der Partitionen und deren Partitionssignaturen

```
1  m ← 0
2  foreach v ∈ V do
3      if ∃ i ≤ m mit σ_i = σ(v) then
4          V_i ← V_i ∪ {v}    /* Knoten mit gleicher Signatur zu einer Partition
           zusammenfassen */
5      else
6          m ← m + 1                                  /* neue Partition erzeugen */
7          V_m ← {v}
8          σ_m ← σ(v)
9      end
10 end
11 return (V_1, ..., V_m, σ_1, ..., σ_m)
```

Hier wird die Signatur $\sigma(v)$ definiert als

$$\sigma(v) = \left(\alpha(v); \left\{\Big(\alpha(u_1); \beta(v, u_1)\Big), \ldots, \Big(\alpha(u_m); \beta(v, u_m)\Big)\right\}\right),$$

dabei sind $\{u_1, \ldots, u_m\}$ die Nachbarn von v.

Beispiel 4.3. Das folgende Beispiel verdeutlicht die Partitionierung der Graphen H und G aus Abbildung 4.2.

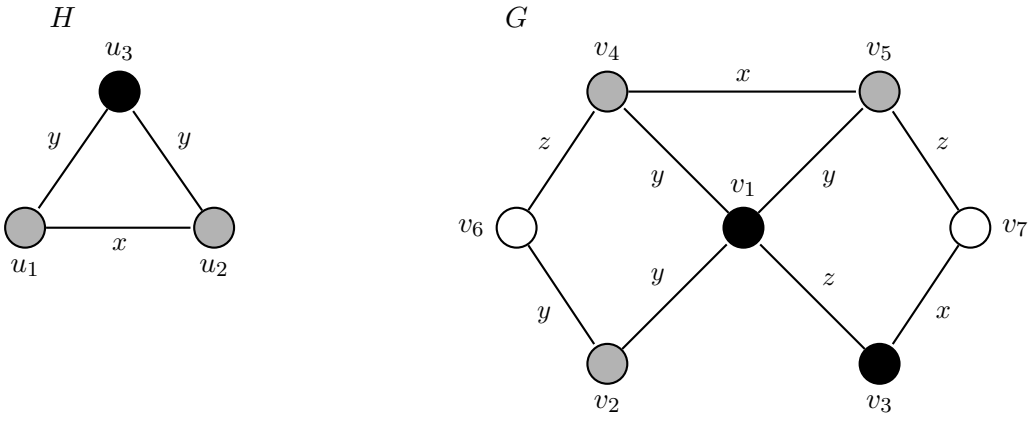


Abbildung 4.2: Eine Instanz für einen Subgraphisomorphietest

$$\sigma(u_1) = \Big(\bullet; \big\{(\bullet; y), (\bullet; x)\big\}\Big)$$
$$\sigma(u_2) = \Big(\bullet; \big\{(\bullet; y), (\bullet; x)\big\}\Big)$$
$$\sigma(u_3) = \Big(\bullet; \big\{(\bullet; y), (\bullet; y)\big\}\Big)$$

$$\sigma(v_1) = \Big(\bullet; \big\{(\bullet; y), (\bullet; z), (\bullet; y), (\bullet; y)\big\}\Big)$$
$$\sigma(v_2) = \Big(\bullet; \big\{(\bullet; y), (\circ; y)\big\}\Big)$$
$$\sigma(v_3) = \Big(\bullet; \big\{(\bullet; z), (\bullet; x)\big\}\Big)$$
$$\sigma(v_4) = \Big(\bullet; \big\{(\bullet; y), (\bullet; x), (\circ; z)\big\}\Big)$$
$$\sigma(v_5) = \Big(\bullet; \big\{(\bullet; y), (\bullet; x), (\circ; z)\big\}\Big)$$

$$\sigma(v_6) = \Big(\bigcirc ; \Big\{ \big(\bullet ; y \big), \big(\bullet ; z \big) \Big\} \Big)$$

$$\sigma(v_7) = \Big(\bigcirc ; \Big\{ \big(\bullet ; x \big), \big(\bullet ; z \big) \Big\} \Big)$$

Aus der $\subseteq$-Beziehung der Signaturen σ ergeben sich folgende Möglichkeiten, Knoten aus H auf G abzubilden:

$$\{u_1, u_2\} \mapsto \{v_4, v_5\}$$
$$\{u_3\} \mapsto \{v_1\}$$

Ist eine Partition von H (eine linke Seite) größer als die entsprechende rechte Seite, gibt es keinen Subgraph-Isomorphismus, da nicht alle Knoten der linken Seite in passende Knoten aus G abgebildet werden können. In der Implementierung wurde diese Heuristik ebenfalls berücksichtigt.

5 Implementierung des FSG-Verfahrens

In diesem Kapitel wird die Implementierung des FSG-Verfahrens vorgestellt. Ich habe mich entschieden, diese Variante der Apriori-basierten Algorithmen zu implementieren, da sie effizienter als der AGM-Algorithmus ist. Es wird auf die Herausforderungen bei der Implementierung sowie auf die Lösungen der Probleme eingegangen. Das Datenflussdiagram in Abschnitt 5.5 dient zur Veranschaulichung der Vorgehensweise bei der Implementierung und der Struktur des Programms.

5.1 Programmaufruf

Für den Aufruf des Programms muss angegeben werden, wie groß der Support der gefundenen Teilgraphen mindestens sein soll. Der Aufruf des Programms FSG sieht wie folgt aus:

java -jar FSG.jar [Dateiname] [Support]

Dabei können für den Platzhalter *[Support]* beliebige Dezimalpunkt-Zahlen in dem Wertebereich $0 < s \leq 1$ eingegeben werden. Der Platzhalter *[Dateiname]* wird durch die Datei ersetzt, die die Datenbank der Transaktionen enthält.

5.2 Ausgaben des Programms

Das Programm gibt alle Teilgraphen aus, die bezüglich der Test-Datenbank mindestens den eingegebenen Support haben. Sie werden in einem Fenster angezeigt, wie in Abbildung 5.2 gezeigt ist.

5.3 Testdaten

Als Testdatenbank wird eine Chemikaliendatenbank ([GKPWR04]) verwendet, in der Moleküle als Graphen gespeichert sind, wie zum Beispiel die in Abbildung 5.1 dargestellten. Diese wiederum sind als Knoten- und Kantenliste kodiert.

Die Entscheidung für diesen Testdatensatz erfolgte auch dadurch, dass die praktische Anwendung der Algorithmen unter Anderem in der Klassifizierung chemischer Verbin-

dungen stattfindet. Auch in Veröffentlichungen wie z. B. [IWM00] wurden diese Arten von Daten verwendet, um Laufzeitresultate zu messen.

Die 2344 Test-Transaktionen aus der Chemikaliendatenbank sind als *.sdf*-Datei gespeichert. Die Graphen besitzen zwischen 2 und 100 Kanten.

5.4 Herausforderungen

Die größte Herausforderung bei der Implementierung war es, die Isomorphie- und Subgraph-Isomorphie-Algorithmen für markierte Graphen unter Anwendung der in Kapitel 4 beschriebenen Heuristiken zu implementieren.

Die Isomorphie-Tests werden über die kanonische Kodierung eines Graphen durchgeführt. Für die Berechnung des *cl*-Codes eines Graphens werden alle Knotenpermutationen dahingehend getestet, ob ihre Adjazenzmatrix einen minimalen Code ergibt. Eine Permutation, die die Einträge der Matrix unverändert lässt, ist ein Automorphismus.

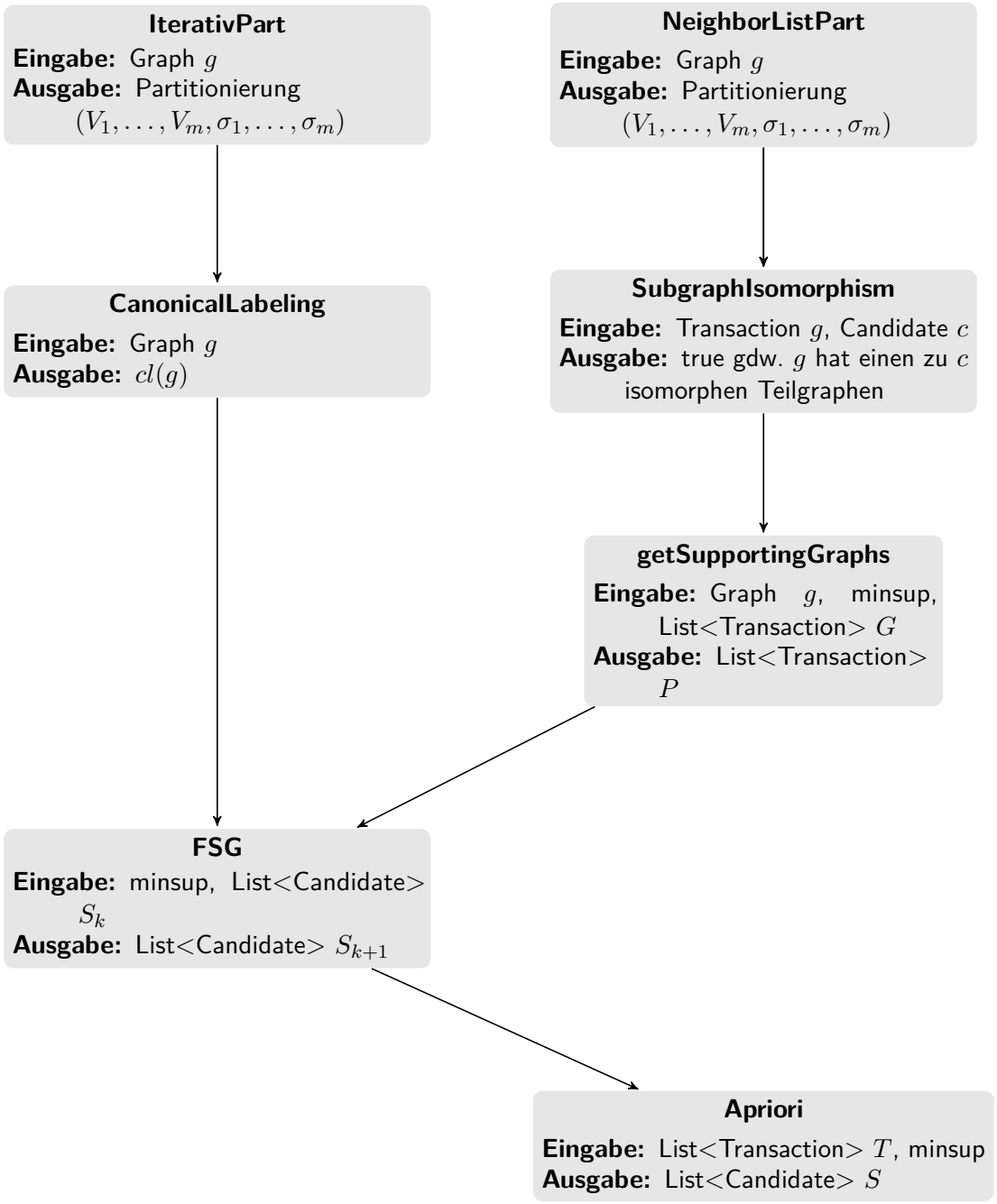
Für die Implementierung der SGI-Tests sind eigene Überlegungen eingeflossen. Dieses Problem wurde mit Hilfe der Methode NachbarListenPartitionierung gelöst, die Transaktionen und Kandidaten wurden anhand von Invarianten in Partitionen einteilt und die der Transaktionen auf die der Kandidaten abgebildet.

Die zunächst hohe Laufzeit des Programms war eine Überraschung. Nach kurzer Analyse ergab sich, dass die Berechnung der Teilgraphen der ersten Stufe sehr lange dauerte. Das Problem war, dass der Algorithmus für eine Datenbank, die n Knoten- und Kantenlabels hat, zu Beginn n^3 Graphen in der ersten Stufe testet, da er in Stufe 1 mit Kandidaten mit einer Kante und zwei Knoten beginnt.

Um die Laufzeit des Programms zu verbessern, wurde folgendes optimiert:

- Bevor Kandidaten der ersten Stufe generiert wurden, wurde der Support von einzelnen Knoten mit verschiedenen Labels berechnet. Nur die häufigen von ihnen wurden für die Bildung der ersten Stufe benutzt.
- Zudem wurden bei der Kandidatengenerierung viele Graphen doppelt erzeugt, etwa O ═ C und C ═ O .

5.5 Datenflussdiagramm



5.6 Messungen und Vergleich

In diesem Kapitel möchte ich die Aufmerksamkeit auf die graphische Auswertung des FSG-Algorithmus unter der Anwendung der in Kapitel 4 beschriebenen Heuristiken lenken.

Laufzeit von FSG in Abhängigkeit vom Support

In der Abbildung 5.3 wird durch ein Balkendiagramm die Anzahl $|S_k|$ der in der Test-Datenbank häufigen Teilgraphen in Abhängigkeit von der Stufengröße k der Kandidaten dargestellt. Die Kurve zeigt auch die Laufzeit **T** des FSG-Algorithmus für die Berechnung von S_k an.

Der für die Messung genutzte Support war 5%, die Datenbank beinhaltete 2344 Transaktionen.

Die Laufzeit weist ein exponentielles Wachstum auf. Das liegt daran, dass die Anzahl der Kandidaten in jeder Stufe wächst. Ab der Stufe 5 ist die Anzahl nahezu konstant, weil zu viele Graphen nicht mehr häufig sind. Die Laufzeit bleibt aber exponentiell und muss also auch von der Größe der Graphen abhängen.

Im Vergleich dazu zeigt das Balkendiagram in der Grafik 5.4 die Menge der häufigen Teilgraphen aus derselben Graph-Datenbank, die einen Support von 20% haben.

Bereits nach 12 Sekunden terminiert der Algorithmus und gibt die häufigen Kandidaten aus. Der Verlauf der Laufzeitkurve unterscheidet sich zu der aus der Abbildung 5.3. Da hier die Anzahl der Subgraphen ab der 5. Stufe aufgrund des hohen notwendigen Supports wieder abnimmt, hat FSG immer weniger Teilgraphen zu verarbeiten, d. h. diese miteinander zu verschmelzen sowie anschließend Isomorphie- und Subgraph-Isomorphie-Tests durchzuführen. Dadurch wird der Zeitbedarf verringert, so dass die Berechnungsdauer pro Stufe sogar wieder abnimmt und auf Null fällt, da gar keine häufigen Teilgraphen mehr gefunden werden.

Die Laufzeiten in Abbildung 5.4 folgen für die Stufen 5 bis 7 auffällig der vorangehenden Stufengröße. Hier ist die Laufzeit also proportional zur Menge der weiterzuverarbeitenden Graphen.

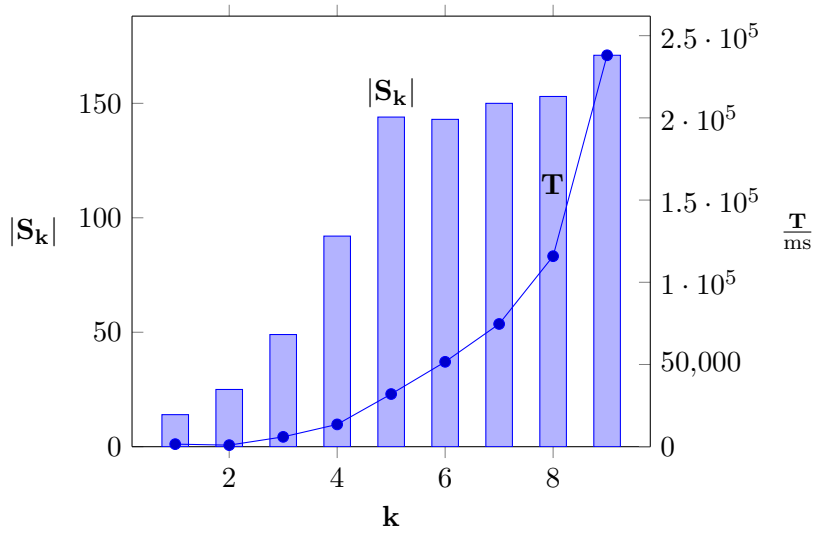


Abbildung 5.3: Vergleich von Stufengröße und Laufzeit für 5% Support

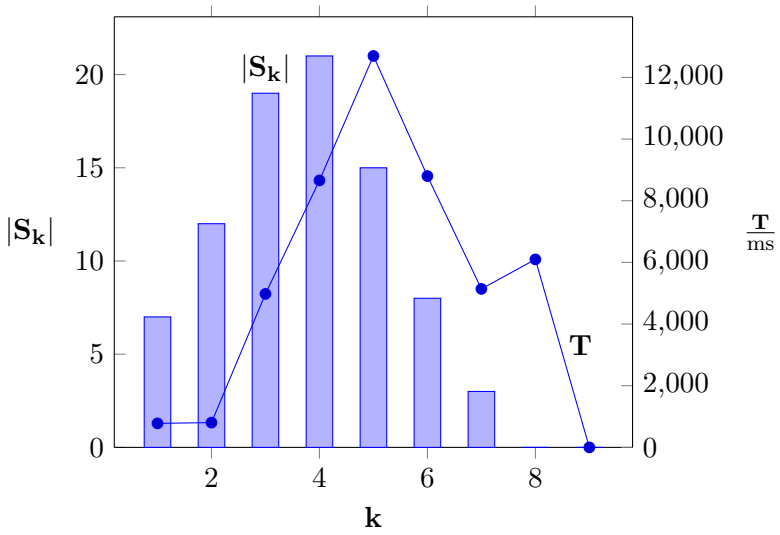


Abbildung 5.4: Vergleich von Stufengröße und Laufzeit für 20% Support

Nutzen der Heuristiken im Bezug auf die Laufzeit

Das Schaubild 5.5 gibt Auskunft über die durchschnittliche Laufzeit des Subgraph-Isomorphie-Tests im FSG-Algorithmus, d. h. über den Durchschnitt der Laufzeit für alle Instanzen der gleichen Größe. Die ungefähre Beziehung zwischen dem mittleren Zeitbedarf und der gesamten Größe des Transaktionsgraphen und des Kandidaten wird durch eine Regressionsgerade dargestellt.

Die Exponentialkurve in 5.5 macht den Unterschied zwischen der Implementierung mit und ohne Partitionierung deutlich, sie ist die durchschnittliche gemessene Laufzeit bei abgeschalteter Partitionierung.

Aus der Grafik 5.6 sind die Laufzeiten für die Berechnung der *cl*-Codes zu entnehmen. Wie man sieht, ist es eine sehr effiziente Lösung für die Reduktion der Laufzeit bei Isomorphietests.

In diesem Schaubild gibt es einen Ausreißer bei der Laufzeit für Graphen in der Stufe 9. Das heißt, dass die implementierten Heuristiken nicht immer effizient sind, insbesondere treiben Graphen, deren Knoten sich gar nicht oder nur eingeschränkt in Partitionen einteilen lassen, die Laufzeit hoch.

Solche Graphen sind *reguläre* Graphen wie Kreise und Cliquen, wie z. B. die in der Abbildung 6.1 dargestellten.

Das Fazit dieser Untersuchung ist, dass die durchschnittliche Laufzeit von FSG trotz Ausreißern meist weit unter dem Worst-Case liegt. Das heißt, dass die vorgestellten Heuristiken für Graphen sich für den praktischen Einsatz sehr gut eignen.

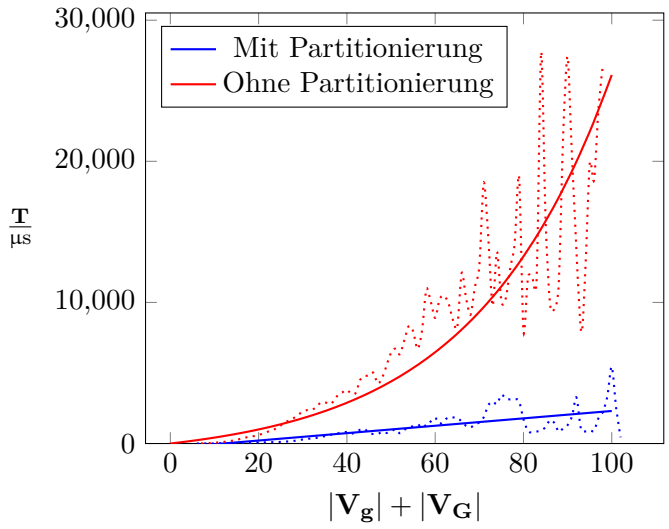


Abbildung 5.5: Durchschnittliche Laufzeit der Subgraph-Isomorphietests

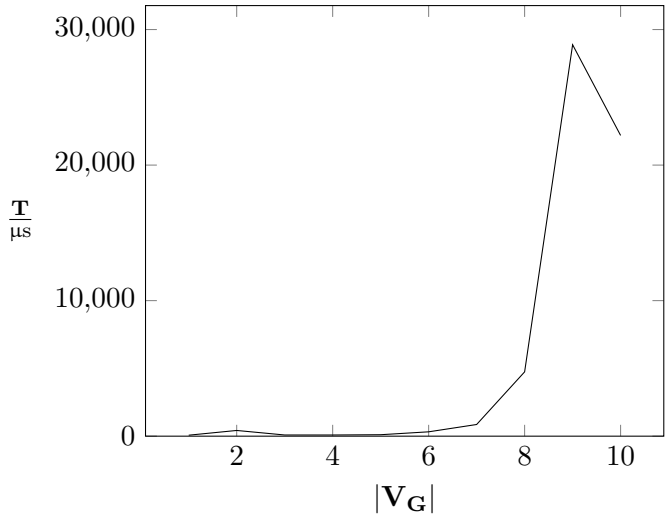


Abbildung 5.6: Durchschnittliche Laufzeit der *cl*-Berechnung

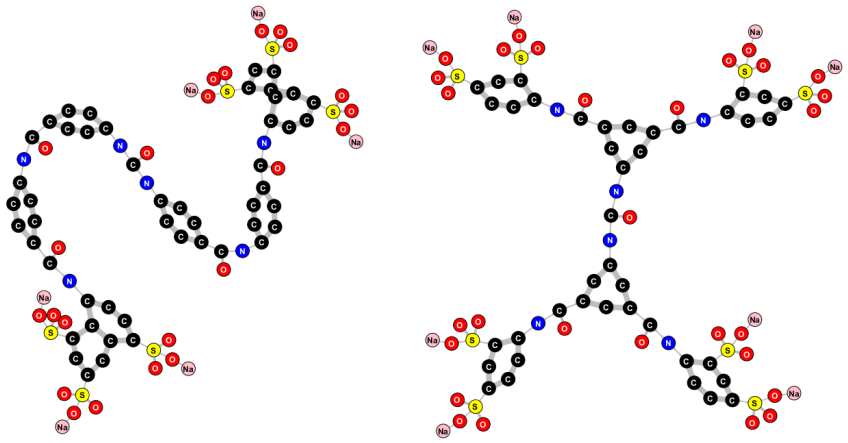


Abbildung 5.1: Transaktionen aus der Testdatenbank

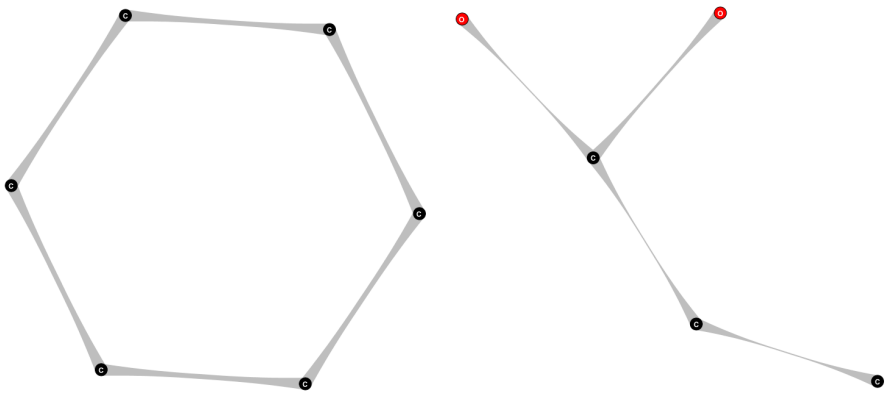


Abbildung 5.2: Einige häufige Teilgraphen mit Support >50% bzw. >10%

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde das Frequent-Subgraph-Problem als Herausforderung im Graph-Mining beschrieben. Es wurden die Algorithmen AGM, FSG und gSpan vorgestellt und verglichen. AGM war einer der ersten Algorithmen, der das Problem löste. FSG ist eine neuere Implementierung des Apriori-Patterns, die deutlich effizienter ist als AGM. Als Vertreter der Pattern-Growth-Algorithmen wurde gSpan vorgestellt, dessen Laufzeit im Bereich von FSG liegt oder diese sogar schlägt.

Im Rahmen der Arbeit wurde FSG implementiert. Dabei sind verschiedene Heuristiken aus [KK04] und [KHPM13] entnommen. Dazu gehören Partitionierungsverfahren und kanonische Kodierungen von Graphen als Heuristiken für Isomorphietests, sowie primäre Kerne und TID-Listen für das Vermeiden überflüssiger (Subgraph-)Isomorphietests.

Diese Heuristiken wurden in Abschnitt 4.2 durch eigene Überlegungen ergänzt. Insbesondere ist der Algorithmus für die Nachbarlisten-Partitionierung entstanden, sowie dessen Anwendung im Subgraph-Isomorphie-Algorithmus, um den Support effizienter zu berechnen.

Die Heuristik der Partitionierung verbessert nicht für alle Graphen die Laufzeit. Abbildung 6.1 zeigt Beispiele für so genannte reguläre Graphen, d. h. Graphen, in denen alle Knoten bezüglich beliebiger Isomorphie-Invarianten ununterscheidbar sind. Für diese Graphen ist die Partitionierung sogar eine Laufzeitverschwendung, da alle Knoten in einer Partition landen. Dies hatte zur Folge, dass in der Implementierung der Graph C_{10} nur trivial partitioniert werden konnte und das Programm bei der Berechnung des kanonischen Labelings stecken geblieben ist, da 10! Knotenpermutationen geprüft werden mussten.

Aus [KK04] folgt, dass durch eine so genannte *Vertexstabilisierung* diese Probleme umgangen werden können. Die Idee dabei ist, die Symmetrie zu unterbrechen, indem jeder Knoten einmal so behandelt wird, als ob er sich von den anderen unterscheiden würde. Damit erhält der Knoten eine entsprechende Partition. Dieses Verfahren muss zwar für jeden Knoten durchgeführt werden. Allerdings ist der Rest des Graphen sehr leicht iterativ zu partitionieren, so dass sich die Laufzeit insgesamt verbessert.

Weitere mögliche Verbesserungen wären eine Anpassung an gerichtete Graphen und die Implementierung weiterer Parser für andere Dateiformate.

Ein ähnliches Problem ist das Finden von *geschlossenen* häufigen Teilgraphen, d. h. Teilgraphen, die nicht zu größeren häufigen Kandidaten erweitert werden können. Da Teilgraphen der häufigen Graphen auch häufig sind, kann auf das Ausgeben der nicht-geschlossenen Kandidaten verzichtet werden.

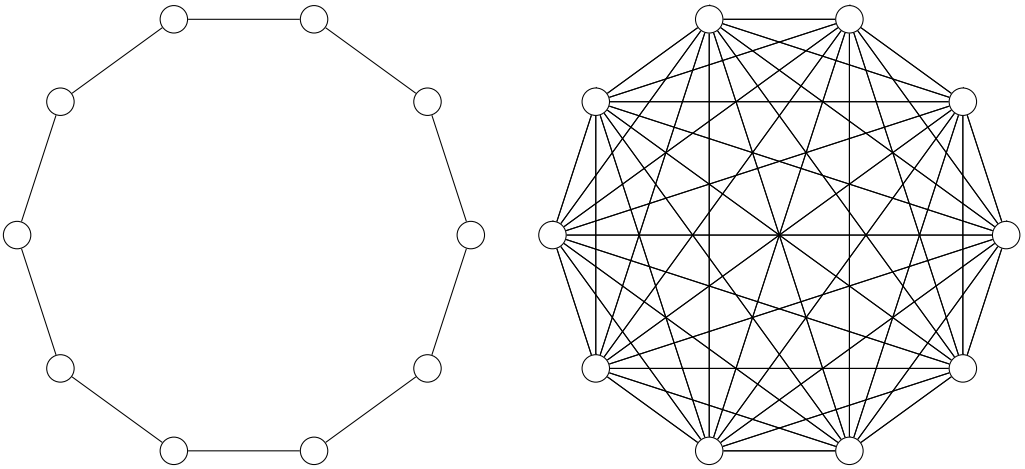


Abbildung 6.1: Reguläre Graphen C_{10} und K_{10}

Literaturverzeichnis

[GKPWR04] M. von Grotthuss, G. Koczyk, J. Pas, L. S. Wyrwicz, L.Rychlewski, *Ligand.Info: Small- Molecule Meta-Database. Comb Chem High Throughput Screen.* 2004 Dec;7(8):757-61.

[HK06] Jiawai Han, Micheline Kamber, *Data Mining: Concepts and Techniques*, 2nd edition, 2006, Morgan Kaufmann Publishers.

[IWM00] Akihiro Inokuchi, Takashi Washio, Hiroshi Motoda, *An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data*, Principles of Data Mining and Knowledge Discovery, 2000. 13-23. Springer Berlin Heidelberg.

[KHPM13] D. Kavitha, D. Haritha, V. Kamakshi Prasad, J. V. R. Murthy *An Improved Unique Canonical Labeling for Frequent Subgraph Mining*, Advanced Computing Technologies (ICACT), 2013, 15th International Conference on. IEEE.

[KK04] Michihiro Kuramochi, George Karypis, *An Efficient Algorithm for Discovering Frequent Subgraphs*, Knowledge and Data Engineering, IEEE Transactions on 16.9, 2004.

[Wel11] Ruud Welling. *A performance analysis on maximal common subgraph algorithms.* 15th Twente Student Conference on IT, Enschede, The Netherlands. University of Twente. Vol. 14. 2011.

[Wör05] Marc Sebastian Wörlein. *Vergleich von Algorithmen zur Fragmentsuche in Molekül-datenbanken (gSpan/Gaston)*, 2005. Studienarbeit.

[YH02] Xifeng Yan, Jiawei Han. *gSpan: Graph-based substructure pattern mining*, Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on. IEEE, 2002.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 26. September 2014

Maria Buling