Parallele Graph-Mining-Techniken zur Auswertung von Hypergraph-Strukturen

Bachelorthesis

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von:	Ole Fenske
Matrikelnummer:	213206164
geboren am:	12.02.1995 in Ludwigslust
Erstgutachter:	Prof.Dr.rer.nat.habil. Andreas Heuer
Zweitgutachter:	Dr.-Ing. Holger Meyer
Betreuer:	Dr.-Ing. Holger Meyer
Abgabedatum:	02.04.2018

Kurzfassung

In den letzten Jahren ist die Analyse von großen Datenmengen (*Big-Data*) immer mehr in den Fokus von Forschung und Industrie gerückt. Viele Unternehmen sehen sich mit *Big-Data* konfrontiert und müssen dementsprechend ihre Systeme auf die wachsenden Datenmengen anpassen. Das als *Big-Data-Analytics* bezeichnete Feld beschäftigt sich mit der Analyse von eben diesen großen Datenmengen, indem es unter anderem verschiedene Techniken des *Data-Mining* oder *Machine-Learning* parallelisiert. Dabei liegen die Daten meist in einer relationalen Struktur vor. Da mit Systemen, wie z.B. *Neo4j* oder auch *Hydra.PowerGraph* eine alternative Speicherung von Datensätzen als Graph oder Hypergraph möglich geworden ist, entstand im Zusammenhang mit dem *Data-Mining* das Gebiet des *Graph-Mining*. Aber auch hier ist die Datenmenge in den letzten Jahren so stark angewachsen, dass die Verfahren des *Graph-Mining* nun parallelisiert werden müssen, um weiterhin effizient berechnet werden zu können. Mit eben dieser Parallelisierung beschäftigt sich diese Arbeit. Es wird sich genauer auf das *Frequent-Subgraph-Mining* bezogen, welches als Ziel das Finden häufig auftretender Muster in einem oder mehreren Graphen hat. Dabei ist in die beiden Szenarien *Transactional-Data*, also die Verarbeitung einer Menge von Graphen und *Large-Sparse-Graph-Data*, die Verarbeitung eines einzelnen großen zusammenhängenden Graphen, zu unterscheiden. Durch den Anwendungsfall dieser Arbeit wird sich auf das letztere Szenario spezialisiert. Jedoch gibt es aktuell keine Algorithmen, die diesem Szenario und einer verteilten und parallelen Berechnung gerecht werden können. Deswegen wurde im Verlauf dieser Arbeit mit *PaSiGraM* ein eigener, paralleler, verteilter Algorithmus entworfen, der auf beliebig große Datenmengen und beliebig viele Maschinen in einem Cluster skaliert werden kann. Durch diesen Algorithmus ist es also nun möglich, *Frequent-Subgraph-Mining* zu parallelisieren und somit auch auf großen Datenmengen berechnen zu können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Definitionsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Heutzutage benutzen viele Menschen ihr Smartphone mehrere Male am Tag. Geräte sind immer besser miteinander vernetzt, in verschiedene Systeme integriert und sammeln immer mehr Daten. Vor allem das *Internet of Things* erhält in immer mehr Lebensbereiche Einzug. Die Erzeugung und das Konsumieren von Daten ist mittlerweile für viele Menschen zum Alltag geworden. In der Wirtschaft stehen Unternehmen häufig vor dem Problem, diese großen Datenmengen abspeichern und analysieren zu wollen. Das in den letzten Jahren intensivierte Forschungsfeld *Big-Data* nimmt dadurch eine immer zentralere Stellung in Industrie und Forschung ein. Im Zusammenhang damit stehen aber auch die Möglichkeiten, die sich durch die ansteigende Größe der Datenströme in der Datenauswertung ergeben. Das als *Big-Data-Analytics* bezeichnete Feld beschäftigt sich mit eben dieser Datenauswertung auf sehr großen Datenmengen. Dabei kommen Techniken aus Feldern wie der künstlichen Intelligenz, des *Data-Mining* und des *Machine-Learning* zum Einsatz. Im Gegensatz zum klassischen *Data-Mining* besteht beim Analysieren auf sehr großen Datenmengen (*Big-Data-Analytics*) das Problem darin, die Algorithmen so zu programmieren, dass diese parallel auf mehreren Computern gleichzeitig berechnet werden können, um die Ausführungszeit eines solchen Programmes zu verkürzen.
Durch die Vielfältigkeit der Daten, die täglich produziert werden, gibt es verschiedene Modelle, diese dauerhaft abzuspeichern. Eine Möglichkeit ist, die Datensätze in Graphen zu speichern. Das wird genau dann gemacht, wenn es sich dabei um verlinkte Datensätze handelt. Diese Art von Datensätzen ist mittlerweile allgegenwärtig, ohne dass die Nutzer es überhaupt merken. Ein Beispiel sind die User von Facebook, welche Freundschaften zueinander haben und somit als Graph dargestellt und auch abgespeichert werden können. Ein anderes Beispiel ist wiederum das *World-Wide-Web*, dessen einzelne Seiten miteinander verlinkt sind und dadurch ebenfalls als Graph darstellbar sind. Weitere Beispiele wären chemische Netzwerke, Unternehmensarchitekturen oder auch biologische Netzwerke. Diese Art der Darstellung und Speicherung von Daten findet in vielen Bereichen Anwendung, da sich durch die Graphstruktur an sich, die Beziehungen

zwischen den einzelnen Datensätzen darstellen lassen.
Das Gebiet, welches sich damit beschäftigt, Muster in diesen Graphen zu finden und somit neues Wissen zu generieren, wird dabei als *Graph-Mining* bezeichnet. Dazu werden auch, ähnlich wie beim *Data-Mining*, Techniken der künstlichen Intelligenz und des *Machine-Learning* benutzt. Auch hier kann man natürlich wieder in einzelne Techniken unterscheiden, aber dazu in den späteren Kapiteln mehr.
Die klassischen *Graph-Mining*-Algorithmen sind meistens darauf ausgelegt, auf einem Computer zu funktionieren. Um auch hier dem Problem sehr großer Datenmengen, wie sie z.B. bei Facebook anfallen, begegnen zu können, ist es ebenfalls nötig, solche Techniken zu parallelisieren. Das bedeutet also, man muss die einzelnen Algorithmen so umstrukturieren, dass sie auf mehreren Computern in einem Cluster gleichzeitig ausgeführt werden können.

1.1 Motivation

Wie man im Laufe der Arbeit sehen wird, gibt es schon viele Techniken, um *Graph-Mining* auf Graphstrukturen zu betreiben. Allerdings ist es aktuell Aufgabe der Forschung, diese Techniken, welche für das *Graph-Mining* benötigt werden, zu parallelisieren, um auch hier den Problemen von *Big-Data* begegnen zu können. Des Weiteren gibt es, wie diese Arbeit später noch zeigen wird, neben den klassischen Graphen auch sogenannte Hypergraphen. Und auch hier steht man vor dem Problem, Datenanalysen auf sehr großen Hypergraphen machen zu wollen. Ein Projekt, welches aktuell am *Lehrstuhl für Datenbanken und Informationssysteme* der *Universität Rostock* durchgeführt wird, nennt sich *ISEBEL* und befasst sich mit genau diesem Problem. Im folgenden Abschnitt wird kurz erläutert, worum es sich bei diesem Projekt handelt und wie diese Arbeit dazu beitragen kann, Anforderungen, die sich im Verlauf dieses Forschungsprojekts ergeben haben, zu überwinden.

1.1.1 Das WossiDiA-System

Richard Wossidlo (1859-1939) war ein deutscher Feldforscher im Bereich der Volkskunde. Er sammelte im Zuge seiner Forschungen Fakten und Daten über Legenden und Geschichten aus dem mecklenburgischen Raum. All diese Daten schrieb er auf kleine Stücken Papier und organisierte sie in Boxen. Diese Notizen beinhalten unter anderem Informationen über Erzähler, Mitwirkende, Plätze und Zeitangaben einer Legende. Aber *Richard Wossidlo* hielt nicht nur diese Informationen fest, sondern auch alle Varianten einer Legende oder Geschichte, die sich in den unterschiedlichen Regionen oder auch Orten erzählt wurden. Abbildung 1.1 zeigt ihn mit einer seiner Boxen in seinem Archiv.

Abbildung 1.1: Richard Wossidlo in seinem Archiv (Quelle: [MSH17]).

Mit der Zeit entstand das sogenannte *Wossidlo*-Archiv. Um heutige Wissenschaftler in ihrer Forschung zu unterstützen, wurden alle Notizen aus diesem Archiv auf 35mm-Mikrofilmen archiviert. Durch das *Wossidlo-Digital-Archiv*-Projekt wurden all diese gesammelten Informationen dann digitalisiert und über ein IT-System bereitgestellt, welches heute unter dem Namen *WossiDiA* bekannt ist.
Das *WossiDiA*-System beinhaltet nicht nur die digitalisierten Dokumente, sondern erlaubt auch das Managen, Verlinken, Navigieren und Stellen von Anfragen. Dabei ist für die Wissenschaftler, welche dieses System nutzen, besonders interessant, welche Legenden/Geschichten mit welchen Notizen zusammenhängen. Um all diesen Anforderungen gerecht zu werden, wurde für das *WossiDiA*-System ein graphbasierter Ansatz (*Hydra.PowerGraph*-System), in Kombination mit einer semi-strukurierten XML-Speicherung, gewählt. Dieser graphbasierte Ansatz spiegelt sich darin wieder, dass alle Dokumente in einer Hypergraph-Struktur zur Verfügung gestellt werden, um Operationen, wie z.B. das graphbasierte *Filtern* in Kombination mit *Kontraktion* und *Aggregation* von Hyperkanten, durchführen zu können. Welche Form von Graphen im *Hydra.PowerGraph*-System benutzt wird, ist Gegenstand eines folgenden Kapitels.

1.1.2 Das ISEBEL-Projekt

Das *ISEBEL*-Projekt kann als ein Projekt aufbauend auf dem *WossiDia*-System bezeichnet werden. Das Team, welches an *ISEBEL* arbeitet, setzt sich aus Forschern der *Universität Rostock*, des *Meertens Institut* in Amsterdam in den Niederlanden sowie der *Universität von Kalifornien* in Los Angeles (UCLA) in den USA zusammen. Der Titel *ISEBEL* steht dabei für „**I**ntelligent **S**earch **E**ngine for **Be**lief **L**egends“. Es geht darum, wie auch schon bei *WossiDiA*, den Wissenschaftlern die Analyse und Suche von Erzählüberlieferungen zu ermöglichen und ihnen länderübergreifend mehrere Archive über ein IT-System zur Verfügung zu stellen. Dabei sollen klassische Methoden

der Motivforschung durch die Einbeziehung von kommunikativen Kontexten und Raumbezügen einer neuen Betrachtungsweise unterzogen werden. Gleichzeitig soll das Projekt der Auftakt dazu sein, für Folklore-Archive eine internationale Umgebung zu schaffen, der sich später andere Länder anschließen werden. Dazu sollen die Inhalte der verschiedenen Folklore-Archive über ein gemeinsam genutztes Protokoll und in einem einheitlichen Datenformat zur Verfügung gestellt werden. Das Protokoll, welches dazu benutzt wird, ist *OAI-PMH*. Wie in Abb. 1.2 zu sehen, werden die Daten der jeweiligen Datenbanken (hier *WossDiA*, *Dutch Folktale* und *Danish Folktale*) in einem lokalen Exportschemata zur Verfügung gestellt, dann über das *OAI-PMH*-Protokoll eingesammelt und in einem globalen Datenschema (Global Data Schema) zur Verfügung gestellt. Das verwendete globale Datenschema ist hierbei wieder ein Hypergraphmodell.
Das *OAI-PMH*-Protokoll setzt sich hierbei aus zwei Komponenten zusammen. Zum einen gibt es den *Data-Provider*, welcher dafür zuständig ist, die Daten aus den lokalen Datenbanken einzusammeln. Die zweite Komponente, der *Service-Provider*, sammelt dann die vom *Data-Provider* bereitgestellten Daten ein und speichert diese in einer Datenbank (hier PowerGraph).
Ein weiteres Ziel ist, wie eingangs schon erwähnt, die Analyse über alle bereitgestellten Datensätze zu ermöglichen. Und genau hier setzt auch diese Arbeit an. Es soll also ermöglicht werden, mithilfe von *Graph-Mining*-Techniken, Analysen über große Datensätze machen zu können.

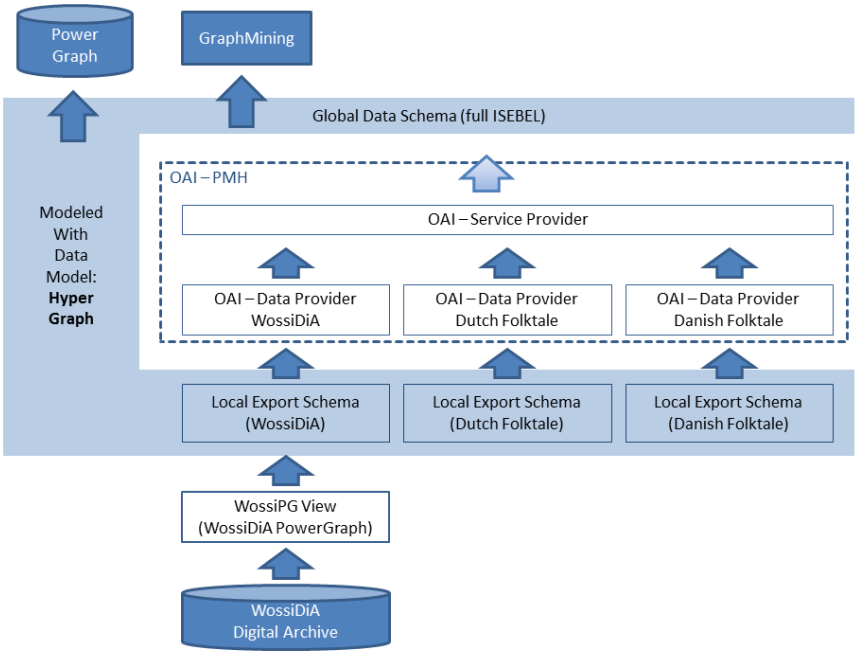


Abbildung 1.2: Die ISEBEL-Architektur.

1.2 Zielsetzung der Arbeit

Ziel dieser Bachelorarbeit ist es, zunächst eine Einführung in die Thematik rund um Graphen und Hypergraphen zu geben. Darauf aufbauend wird ein Überblick über verschiedene *Graph-Mining*-Techniken gegeben, um anschließend mit dem *Frequent-Subgraph-Mining* eine dieser Techniken näher zu erläutern. Dabei wird auf bestimmte Teilschritte des *Frequent-Subgraph-Mining* näher eingegangen, um so die einzelnen Probleme, die dabei auftreten können, verständlicher zu machen. Im Zusammenhang damit werden auch Lösungsansätze präsentiert, mithilfe derer man die unterschiedlichen Probleme adressieren kann.
Ein weiteres Ziel ist, eine *Graph-Mining*-Technik auf eine mögliche Parallelisierung hin zu untersuchen. Da sich im Laufe dieser Arbeit auf das *Frequent-Subgraph-Mining* spezialisiert wurde, wird nur für diese *Graph-Mining*-Technik eine mögliche Parallelisierung untersucht. Dabei werden die beiden Parallelisierungsplattformen *Apache Flink* und *Apache Spark* näher erläutert. Mithilfe dieser Plattformen ist es möglich verschiedene Algorithmen zu parallelisieren. Anschließend soll evaluiert werden, wie der *Stand der Technik* und der *Stand der Forschung* bezüglich paralleler *Frequent-Subgraph-Mining*-Algorithmen aussieht. Dazu wurde eine Problemanalyse, basierend auf dem Anwendungsfall und dem damit einhergehenden Datensatz, den Parallelisierungsplattformen und den schon vorhandenen Algorithmen, vorgenommen. Anschließend wurde als Ergebnis dieser Problemanalyse eine Anforderungsliste erstellt.
Im Verlauf dieser Arbeit wird sich zeigen, dass es keine Algorithmen gibt, die allen Anforderungen der Anforderungsliste gerecht werden und parallel berechnet werden können. Deswegen wurde mit *PaSiGraM* ein eigener Algorithmus für das *Frequent-Subgraph-Mining* entwickelt, der darauf ausgelegt ist alle Anforderungen zu erfüllen und parallelisiert berechnet werden kann.

1.3 Aufbau der Arbeit

Im nachfolgenden Kapitel 2 wird zunächst eine Einführung in die Thematik der Graphen gegeben. Dazu werden Graphen und Hypergraphen in den Unterkapiteln 2.1 und 2.2 erklärt. Anschließend werden unterschiedliche Vorgehensweisen zur Überführung eines Hypergraphen in einen Graphen erläutert. Abschließend wird in Unterkapitel 2.4 das Konzept der *maximal unabhängigen Menge* beschrieben, da es für das weitere Verständnis dieser Arbeit benötigt wird.
Kapitel 3 führt zunächst allgemein in verschiedene *Graph-Mining*-Techniken ein, um ein Überblick über dieses Forschungsgebiet zu geben. Anschließend wird in Unterkapitel 3.2 mit dem *Frequent-Subgraph-Mining* ein dieser Techniken genauer beschrieben. Dabei wird auf einzelne Schritte dieser Technik genauer eingegangen, um zunächst besser identifizieren zu können, welche Probleme beim *Frequent-Subgraph-Mining* auftreten und welche Vorgehensweisen es im All-

gemeinen gibt, um diese Probleme lösen zu können. Abschließend wird in Unterkapitel 3.3 eine konkreter Algorithmus für das *Frequent-Subgraph-Mining* vorgestellt.
Das Kapitel 4 beschäftigt sich mit verschiedenen Plattformen und Systemen, die für diese Arbeit von Relevanz sind. Dazu werden in den Unterkapiteln 4.1 und 4.2 die zwei Plattformen *Apache Spark* und *Apache Flink* vorgestellt. Zusätzlich dazu werden auch jeweils die dazugehörigen Bibliotheken für die Verarbeitung von Graphen erläutert. In Unterkapitel 4.5 wird das *Hydra.PowerGraph*-System erklärt, welches an der *Universität Rostock* entwickelt wurde.
In Kapitel 5 erfolgt dann die Problemanalyse. Dazu wird zunächst näher auf den mit dem Anwendungsfall verknüpften Datensatz eingegangen. In Unterkapitel 5.2 werden dann die beiden Plattformen *Apache Spark* und *Apache Flink* hinsichtlich verschiedener Kriterien miteinander verglichen. Anschließend wird in Unterkapitel 5.3 ein Überblick darüber gegeben, welche Algorithmen für das *Frequent-Subgraph-Mining* aktuell schon existieren und worauf bei dieser *Graph-Mining*-Technik generell zu achten ist. Die aus der Problemanalyse resultierende Anforderungsliste findet sich in Unterkapitel 5.4 wieder.
Kapitel 6 beschreibt dann schließlich den *PaSiGraM*-Algorithmus, welcher selber für den Anwendungsfall dieser Arbeit entwickelt wurde. Hierbei werden die einzelnen Teilschritte des Algorithmus detailliert beschrieben und mit Beispielen veranschaulicht. Zudem wird in Unterkapitel 6.6 erklärt, wie der Algorithmus mithilfe von Flink implementiert und parallelisiert werden kann. Abschließend erfolgt ein Fazit, welches nochmal alle Kernpunkte von *PaSiGraM* zusammenfasst und zeigt wo die Stärken des Algorithmus liegen.
In Kapitel 7 erfolgt dann letztendlich eine Zusammenfassung der ganzen Arbeit und ein abschließender Ausblick.

Kapitel 2

Grundlagen

Dieses Kapitel beschäftigt sich mit den grundlegenden Begriffen dieser Arbeit und wird diese in den folgenden Abschnitten behandeln. Zunächst wird nochmal kurz erläutert, was man unter einem Graphen versteht und darauf aufbauend, folgt eine nähere Betrachtung von Hypergraphen. Anschließend wird auf den Zusammenhang zwischen Graphen und Hypergraphen näher eingegangen und gezeigt, mit welchen Methoden man einen Hypergraphen in einen Graphen überführen kann. Am Ende dieses Kapitels wird das Konzept der *maximal unabhängigen Menge* erläutert, da es für das weitere Verständnis dieser Arbeit benötigt wird.

2.1 Graphen

Fortschritte in der Sozial- und Informationswissenschaft haben gezeigt, dass verknüpfte Daten überall zu finden sind. Beispiele dafür sind unter anderem Kommunikations- und Computersysteme, das *World-Wide-Web*, soziale Netzwerke wie Facebook, biologische Netzwerke, Logistik- und Transportsysteme, Epedemie- bzw. Seuchennetzwerke, chemische Netzwerke oder auch versteckte terroristische Netzwerke. All diese Systeme können als Graph modelliert werden. Dazu wird zunächst definiert, was man unter einem Graph versteht.

Definition 2.1 (Graph)**.** *Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V (engl. vertices) und einer Menge von Kanten $E \subseteq V \times V$ (engl. edges), welche zwei Knoten miteinander verbindet. Eine Kante $e = (u, v)$ besitzt zwei Knoten $u, v \in V$, wobei u und v jeweils die Endpunkte einer Kante beschreiben.*

Weiterhin kann man zwischen *gerichteten* und *ungerichteten* Graphen unterscheiden.

Definition 2.2 (Gerichteter Graph (Digraph))**.** *Gegeben sei ein Graph $G = (V, E)$. G heißt gerichtet, wenn $(u, v) \neq (v, u)$ gilt.*

Definition 2.3 (Ungerichteter Graph)**.** *Gegeben sei ein Graph $G = (V, E)$. G heißt ungerichtet, wenn $(u, v) = (v, u)$ gilt.*

Graphen können zusätzlich noch sogenannte *Labels* enthalten. Das sind Informationen die, in Form von *Attributen* an die Knoten und/oder Kanten eines Graphen angehängt werden.

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

2.2 Hypergraphen

Hypergraphen sind eine generalisierte Variante von Graphen. Sie erweitern das Konzept der einfachen Kante aus einem Graph, indem sie erlauben, dass mehr als zwei Knoten an einer Kante teilnehmen können.

Definition 2.5 (Hypergraph)**.** *Ein Hypergraph ist ein Tupel $H = (V, A)$ mit einer Menge von Knoten V und einer Menge von nicht leeren Hyperkanten $A \subseteq \mathcal{P}(V) \setminus \emptyset$. Eine Hyperkante ist wiederum ein Tupel $A = (S, D)$ mit $S, D \subseteq V$.*

Wie auch bei Graphen, kann man Hypergraphen ebenfalls in *gerichtete* und *ungerichtete* Hypergraphen unterscheiden (vgl. [GLPN93, AL17]). Die Richtung einer Hyperkante wird dann durch das Tupel $A = (S, D)$ ausgedrückt, wobei S (engl. *source*) dann die Menge der eingehenden und D (engl. *destination*) die Menge der ausgehenden Knoten ist.
Zusätzlich existieren in der Literatur unterschiedliche Definitionen, was die Kardinalitäten, den Head und den Tail der Hyperkante eines gerichteten Hypergraphen angeht. *Gallo* [GLPN93] z.B. legt fest, dass S und D *disjunkt* sind, d.h. $S \cap D = \emptyset$. Außerdem können beide Mengen, also S und D, auch die leere Menge sein. *Ausiello* [AL17] hingegen definiert eine Hyperkante als ein Tupel (S, d) mit $d \in V, S \neq \emptyset$. Er schließt nicht aus, dass $d \in S$ sein kann. Außerdem ist d auf einen Knoten begrenzt. Setzt man beide Definitionen in Beziehung kann man sagen, dass Gallo's Definition die allgemeinere ist, weil sie m:n-Beziehungen erlaubt, wohingegen Ausiello's auf m:1-Beziehungen begrenzt ist. In diesem Zusammenhang ist Ausiello ein Spezialfall von Gallo.
Meyer et al. [MSH17] greifen die Definition von *Gallo* auf und erweitern sie um das Konzept der Rollen. Dabei kann ein Knoten verschiedene Rollen haben, welche es ihm ermöglichen mehrere Male an einer Hyperkante teilnehmen zu können. Zusätzlich müssen S und D nicht *disjunkt* sein. Daraus ergibt sich dann folgende Definition:

Definition 2.6 (Gerichteter Hypergraph)**.** *Ein gerichteter Hypergraph $H = (V, A, R)$ besteht aus einer Menge von Knoten V, einer Menge von Rollen R und einer Menge von Hyperkanten $A = (S, D)$ mit $S \subseteq V \times R$ und $D \subseteq V \times R$, wobei S die Menge der eingehenden und D die Menge der ausgehenden Kante ist.*

Zusätzlich kann man noch Typisierung für die Knoten und Hyperkanten einführen, was dann zu folgender Definition führt (vgl. *Meyer et al.* [MSH17]):

Definition 2.7 (Gerichteter, typisierter Hypergraph)**.** *Ein gerichteter typisierter Hypergraph ist ein gerichteter Hypergraph (Def. 2.6) mit einer Menge von Knotentypen Γ^V, einer Menge von Hyperkantentypen Γ^A und einer Menge von Zuordnungsfunktionen $\{\alpha^V, \alpha^A, \rho^V, \rho^A, \delta\}$ mit $\alpha^V : V \mapsto \Gamma^V, \alpha^A : A \mapsto \Gamma^A, \rho^V : R \mapsto \Gamma^V, \rho^A : R \mapsto \Gamma^A$ und $\delta : R \mapsto \{-1, 0, 1\}$. δ definiert die Richtung der Rückwärtskanten mit -1, der Vorwärtskanten mit 1 und der bidirektionalen Kanten mit 0. Dabei müssen folgenden Integritätsbedingungen für die Typen von Hyperkanten eingehalten werden:*

- $\forall (v, r) \in S \cup D : \rho^V(r) = \alpha^V(v)$
- $\forall (v, r) \in S \cup D : \rho^A(r) = \alpha^A(A)$
- $\forall (v, r) \in S : \delta(r) \in \{-1, 0\}$
- $\forall (v, r) \in D : \delta(r) \in \{0, 1\}$

In den folgenden Kapiteln wird sich an der Definition 2.7 orientiert, wenn von Hypergraphen gesprochen wird. Sollte sich nach einer anderen Definition gerichtet werden, ist dies gesondert im jeweiligen Abschnitt angegeben.

2.3 Zusammenhang zwischen Graphen und Hypergraphen

Wie eingangs schon erwähnt, sind Hypergraphen eine generalisierte Variante von Graphen. Für einige Szenarien kann es durchaus nützlich sein, einen Hypergraphen als Graphen zu modellieren. So wurden zum Beispiel in den letzten Jahren viele Algorithmen für das *Mining* auf Graphen entwickelt, wohingegen für Hypergraphen nicht sehr viele Ansätze existieren. Ein Weg, wie man trotzdem die Algorithmen des *Graph-Mining* für Hypergraphen nutzen kann, ist, dass man einen Hypergraphen in einen Graphen überführt. Dazu existieren zwei Ansätze:

- Clique-Expansion
- Star-Expansion

In diesem Kapitel werden diese zwei unterschiedlichen Ansätze kurz erläutert, da sie im Laufe dieser Arbeit noch einmal aufgegriffen werden.

2.3.1 Clique-Expansion

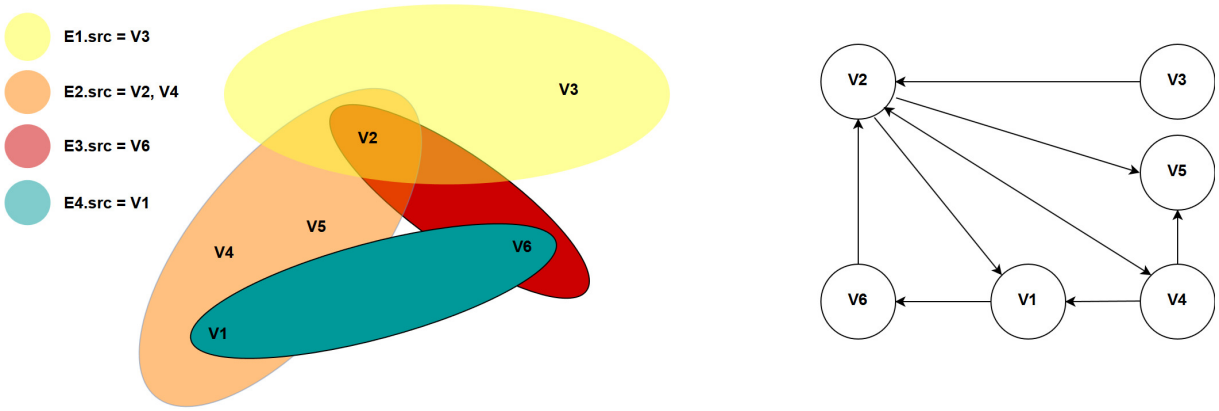


Abbildung 2.1: Ein Hypergraph und seine Darstellung als gerichteter Graph.

Bei diesem Ansatz, einen Hypergraphen in einen Graphen zu überführen, werden die Hyperkanten direkt als Kanten modelliert. Man konstruiert also einen Graphen $G^x = (V, E^x)$, mit $E^x \subseteq V \times V$, aus dem Hypergraphen $H = (V, A)$, indem jede Hyperkante $a = (u_1, ..., u_{\delta(a)}) \in A$ für $\delta(a) = |a|$ paarweise für alle durch die Hyperkante verbundenen Knoten jeweils durch eine Kante ersetzt wird:

$$E^x = \{(u, v) : u, v \in a, a \in A\}.$$

Zu beachten ist noch, dass dadurch die Knoten aus einer Hyperkante a eine Clique im Graph G^x bilden, weswegen dieser Ansatz auch *Clique-Expansion* genannt wird.

2.3.2 Star-Expansion

Bei der *Star-Expansion* werden die Hyperkanten auf die Knoten eines *bipartiten* Graphen abgebildet. Bevor erklärt wird, wie die *Star-Expansion* funktioniert, wird zunächst noch definiert, was man unter einem *bipartiten* Graph versteht.

Definition 2.8 (Bipartiter Graph)**.** *Ein Graph $G = (V, E)$ heißt bipartit, wenn sich seine Knotenmenge V in zwei disjunkte Mengen A und B aufteilen lässt und zwischen den Knoten innerhalb beider Teilmengen keine Kante verläuft. Das bedeutet:*

$$(u, v) \in E : (u \in A \wedge v \in B) \vee (u \in B \wedge v \in A)$$

A und B werden dann als Partitionsklassen und die Menge $\{A, B\}$ als Bipartition bezeichnet.

Daraus kann man sich nun die Definition für den bipartiten Graphen G^*, der aus einem Hypergraphen $H = (V, A)$ entsteht, herleiten:
$G^* = (V^*, E^*)$, wobei $V^* = A^* \cup B^*$ und $A^* = V$ und $B^* = A$. Daraus lässt sich ableiten, dass $V^* = V \cup A$. Für jeden Knoten aus B^* wird dann eine Kante $e \in E^*$ zu jedem Knoten aus A^* hinzugefügt, der an der ehemaligen Hyperkante $a \in A$ beteiligt war.

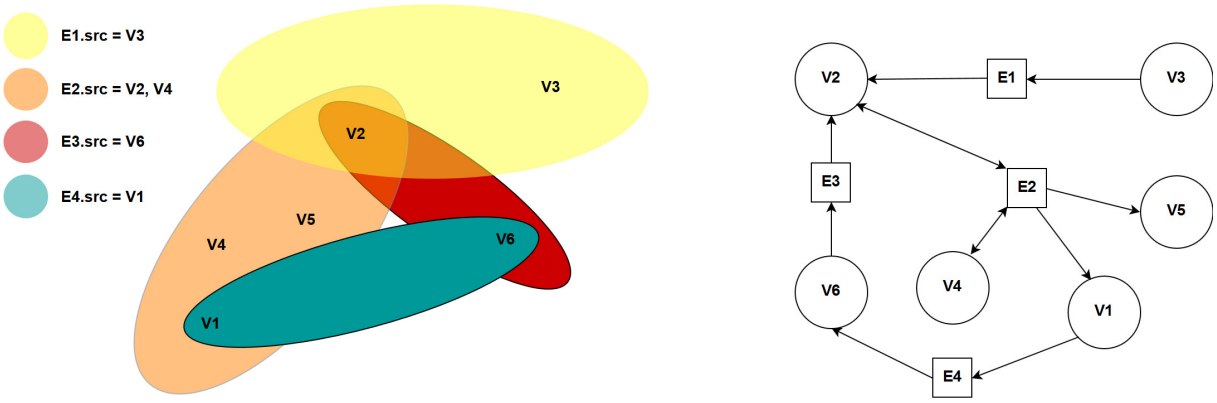


Abbildung 2.2: Ein Hypergraph und seine Darstellung als bipartiter Graph.

Man konstruiert also einen bipartiten Graphen $G^* = (V^*, E^*)$ aus dem Hypergraphen $H = (V, A)$, indem für jede Hyperkante $a \in A$ ein Knoten im bipartiten Graph G^* angelegt wird, sodass $V^* = V \cup A$. Dieser neue Knoten a (Hyperkantenknoten) wird mit jedem Knoten im Graph G^* verbunden, der Teil der ehemaligen Hyperkante war:

$$E^* = \{(u, a) : u \in a, a \in A\}.$$

Zu beachten ist noch, dass jede Hyperkante einem *Stern* (engl. *star*) im bipartiten Graph zugeordnet werden kann, weswegen dieser Ansatz auch *Star-Expansion* genannt wird.
Um die Funktionsweise dieser Methode nochmal zu illustrieren, wird nun ein Beispiel gegeben.

Beispiel 2.8.1. *Wie in Abb. 2.2 zu sehen, ist ein Hypergraph mit einer Menge von Knoten (V1-V6) und einer Menge von Hyperkanten (E1-E4), welche die Knoten miteinander verbinden, gegeben. Schaut man sich beispielhaft die Hyperkante E2 an, so sieht man, dass E2 als Knoten im nebenstehenden bipartiten Graph modelliert wurde. Zusätzlich gibt es eine Kante von E2 zu jedem Knoten, also V1, V2, V4 und V5, der an Hyperkante E2 im Hypergraphen beteiligt ist.*

Analog funktioniert die Modellierung für jede andere Hyperkante in der Abb. 2.1.

2.4 Maximal unabhängige Menge von Graphen

In diesem Kapitel wird das Konstrukt der *maximal unabhängigen Menge* eines Graphen erläutert, da diese in späteren Kapiteln noch von Bedeutung sein wird. Die *maximal unabhängige Menge* ist eine *unabhängige Menge*, die nicht Teilmenge einer anderen *unabhängigen Menge* eines Graphen ist. Deswegen wird nun zunächst erläutert, was man unter einer *unabhängigen Menge* eines Graphen versteht.
Die *unabhängige Menge* eines Graphen ist ein Begriff aus der mathematischen Graphentheorie und beschreibt eine Menge von Knoten in einem Graphen, sodass keine zwei Knoten dieser Menge *adjazent* zueinander sein können. Das bedeutet, dass keine Kanten zwischen den Knoten einer *unabhängigen Menge* vorhanden sein dürfen. In Abb. 2.3 sind die *unabhängigen Mengen* eines Graphen durch die schwarzen Knoten dargestellt. Zwischen den Knoten dieser *unabhängigen Mengen* verlaufen keine Kanten. Die *maximal unabhängige Menge* eines Graphen ist dann die *unabhängig Menge*, die nicht Teilmenge einer anderen *unabhängigen Menge* des Graphen ist. In dem Beispiel in Abb. 2.3 wäre die *maximal unabhängige Menge* die ganz rechte Menge der schwarzen Knoten, da diese drei Knoten enthält und nicht nur zwei, wie es bei den anderen beiden *unabhängigen Mengen* der Fall ist. Das Finden dieser *maximal unabhängigen Menge* eines Graphen gehört zu den NP-vollständigen Problemen in der Informatik.

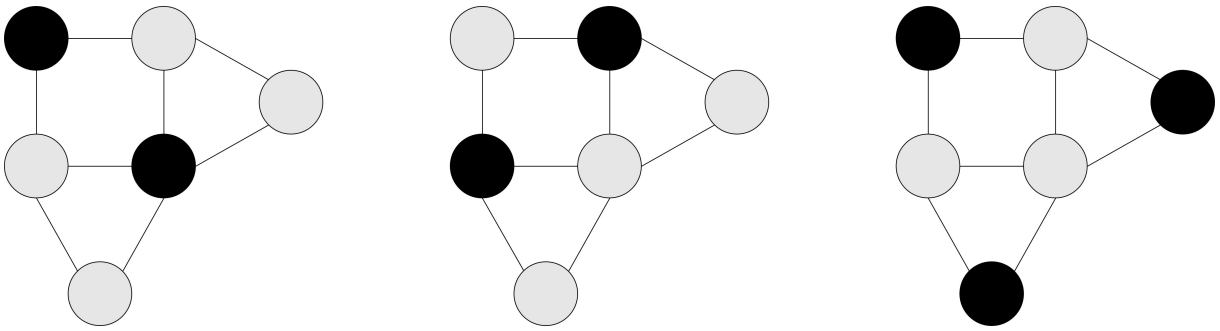


Abbildung 2.3: Die unabhängigen Mengen eines Beispielgraphen.

Kapitel 3

Graph-Mining

Viele klassische Techniken aus dem Bereich des *Data-Mining* finden auch Anwendung auf Graphen. Das Gebiet, welches sich damit beschäftigt, nennt sich *Graph-Mining*. Wie man in Unterkapitel 3.1 sehen wird, lassen sich die Algorithmen des *Graph-Mining*, wie auch beim *Data-Mining*, in verschiedene Techniken unterteilen. Es werden zwei klassische Techniken des *Graph-Mining* kurz vorgestellt, bevor in Unterkapitel 3.2 genauer auf eine dritte Technik, das *Frequent-Subgraph-Mining*, näher eingegangen wird. Anschließend wird mit *GraMi* ein Algorithmus vorgestellt, welcher dem *Frequent-Subgraph-Mining* zugeordnet wird und für den weiteren Verlauf dieser Arbeit noch von Bedeutung ist.

3.1 Graph-Mining-Techniken

Wie anfangs in diesem Kapitel erwähnt, lässt sich das *Graph-Mining* in einzelne Techniken unterteilen. Eine dieser Techniken ist das *Clustering*, welches in Abschnitt 3.1.1 erläutert wird. Anschließend stellt diese Arbeit mit der *Klassifikation* eine weitere Technik vor, die dem *Graph-Mining* zugeordnet werden kann.
Zu erwähnen ist auch, dass es neben diesen beiden Techniken auch noch weitere Algorithmen gibt, welche dem *Graph-Mining* zugehörig sind. Beispiele für solche Algorithmen sind *Keyword-Search*, *Dense-Subgraph-Discovery* oder auch *Streaming*-Algorihmen (vgl. [CCA12]). Diese Arbeit wird sich in diesem Kapitel allerdings nur auf das *Clustering*, die *Klassifikation* und das *Frequent-Subgraph-Mining* beziehen.

3.1.1 Clustering

Clustering-Algorithmen finden in vielen Fällen, in denen man mit Graphen arbeitet, verschiedene Anwendungen, wie z.B. die Überlastungserkennung (engl. *congestion detection*). Hält man sich

an die Definition von *Aggarwal*, lässt sich das *Clustering*-Problem wie folgt beschreiben (vgl. [AW10], [20117]).

Definition 3.1 (Clustering-Problem)**.** *Für eine gegebene Menge von Objekten, will man diese Menge in Gruppen ähnlicher Objekte unterteilen.*

Die *Ähnlichkeit* von Objekten wird dabei oft durch mathematische Funktionen ausgedrückt. Weiterhin unterscheidet *Aggarwal* zwischen zwei verschiedenen Typen von *Clustering*-Algorithmen:

- **Node-Clustering-Algorithmen:** In diesem Fall hat man einen großen zusammenhängenden Graphen. Für diesen Graphen will man dann die Knoten mit Hilfe von Ähnlichkeits- oder Distanzfunktionen in Cluster (bzw. Gruppen) unterteilen. Dabei sind die Kanten des Graphen mit numerischen Distanzwerten versehen. Diese Distanzwerte können dann genutzt werden, um zu ermitteln, welche Knoten nah beieinander liegen und somit einem Cluster zugeordnet werden können.

- **Graph-Clustering-Algorithmen:** In diesem Fall hat man eine (große) Menge von einzelnen Graphen, welche auf Grundlage ihrer zugrunde liegenden Struktur in mehrere verschiedene Cluster eingeteilt werden sollen. Dies stellt eine große Herausforderung dar, da man die zugrunde liegenden Strukturen der Graphen zunächst erkennen und dann die Ähnlichkeit zwischen den verschiedenen Strukturen ermitteln muss.

Für den Anwendungsfall dieser Arbeit (*ISEBEL*) wären die *Node-Clustering*-Algorithmen mit Sicherheit interessanter. Denn sie haben zum Ziel, die Konnektivität zwischen den Knoten der unterschiedlichen Cluster zu minimieren. Somit sind diese Algorithmen dazu geeignet, einen Graphen zu partitionieren und somit den gesamten Datensatz auf eine verteilte Datenbank aufteilen zu können. Das würde bei einem sehr großen Graph dazu führen, dass man bestimmte Anfragen parallelisieren und somit schneller berechnen könnte.
Der einfachste Fall dabei ist das *2-Wege-Minimum-Cut*-Problem, bei dem man den Graphen in zwei Cluster unterteilen und die Gewichte der Kanten (Konnektivität) zwischen den Clustern minimieren will. Dieses Problem ist das mathematische Äquivalent des *Maximal-Fluss-Problems* und effizient lösbar (vgl. [AW10]).
Wenn man aber den Graphen in mehr als 2 Cluster unterteilen möchte, wird das Problem NP-vollständig und somit sehr schwer zu lösen. Man sieht also, dass es nach wie vor ein Problem ist, einen Graphen ordentlich zu partitionieren, sodass die Konnektivität und somit der Netzwerkverkehr möglichst gering ist und gleichzeitig eine gute Balance zwischen den einzelnen Clustern zu haben, damit alle Knoten einer verteilten Datenbank ungefähr gleichviel Datensätze enthalten. Konkrete Algorithmen für das *Node-Clustering* sind:

- *Girvan-Newmann*-Algrithmus von *Girvan und Newmann* ([GN02])

- *Spectral-Clustering*-Algorithmen ([AW10])

3.1.2 Klassifikation

Klassifikation ist eines der zentralen Aufgaben im *Data-Mining* und *Machine-Learning*. Dadurch, dass oftmals Graphen aktuell in vielen verschiedenen Anwendungen genutzt werden, um Entitäten und ihre Beziehungen darzustellen, hat die *Klassifikation* auch im *Graph-Mining* einen hohen Stellenwert erlangt. Ein klassisches Beispiel ist, dass man ein soziales Netzwerk auf die Gesundheit unterschiedlicher Gruppen und die Struktur dieser Gruppen analysieren will. Auch die Klassifikationsalgorithmen lassen sich wieder in verschiedene Typen unterteilen. Folgt man der Unterteilung von *Tsuda und Saigo* erhält man folgende Typen (vgl. [TS10]):

- **Label-Propagation-Algorithmen:** Hier hat man einen großen Graphen, bei dem ein Teil der Knoten mit einem Label versehen ist. Das Ziel ist es, ein Modell aus den Knoten mit Label zu erlernen und auf der Grundlage dieses Modells die Knoten ohne Label zu klassifizieren, also ebenfalls mit einem Label zu versehen.
- **Graph-Klassifikations-Algorithmen:** Eine Teilmenge einer Menge von Graphen ist mit Labels versehen. Das Ziel ist auch hier ein Modell, aus der Teilmenge mit Graphen die ein Label haben, zu lernen und auf Grundlage dieses Modells die Graphen ohne Label zu klassifizieren, sprich mit einem Label zu versehen. Hierbei lässt sich noch in zwei weitere Typen unterteilen:
  - **Unüberwachtes Lernen:** Die Graphen werden auf Grundlage von *Ähnlichkeit* klassifiziert.
  - **Überwachtes Lernen:** Hier wird ein Modell aus einem *Trainingsdatensatz* gelernt. Auf Grundlage dieses Modells werden dann die restlichen Graphen klassifiziert.

Wenn man wieder den Anwendungsfall dieser Arbeit beachtet, wäre *Label-Propagation* interessanter. Denn auch hier geht man wieder davon aus, dass man einen einzigen großen Graphen hat und nicht mehrere kleine Graphen.
In früheren Studien ([20002, TN04]) wurden dazu sogenannte *Diffusion-Kernels* in Kombination mit einer *Support-Vector-Machine* genutzt. Der Grundgedanke dabei ist, die Nähe von zwei Knoten, auf Grundlage der *Pendel-Zeit* für zufällige Pfade zwischen den Knoten, zu berechnen. Für große Graphen ist dieses Verfahren allerdings nicht einsetzbar, da es $O(n^3)$ Zeit und $O(n^2)$-Speicherplatz benötigt. Aber auch dafür gibt es Methoden, die die *Sparsity* einer Adjazenzmatrix berechnen. Dies ist dann effizienter und auch für große Graphen durchführbar (vgl. [TS10]).

3.2 Frequent-Subgraph-Mining

Abbildung 3.1: Frequent Subgraph Mining und seine Anwendungen für weitere Mining-Prozesse.

Das *Frequent-Pattern-Mining* (kurz FPM) ist, wie auch die *Klassifikation*, eines der zentralen Aufgaben des *Data-Mining*. Es besitzt viele Anwendungen, wie z.B. die Warenkorbanalyse. Im Zusammenhang mit Graphen ist dann daraus das *Frequent-Subgraph-Mining* (kurz FSM) entstanden. Es hat zum Ziel, häufig vorkommende Muster in Graphen zu entdecken und somit neues Wissen aus Graphen zu generieren. Wie in Abb. 3.1 zu sehen, sind diese häufigen Muster praktisch um Graphen zu charakterisieren, zu klassifizieren, zwischen unterschiedlichen Gruppen von Graphen zu unterscheiden und um Indizes über einem Graphen zu bilden. So haben *Borgelt et al.* [BB02] mithilfe von FSM-Algorithmen, das Finden von chemischen Strukturen in einem HIV-Datensatz illustriert. *Desphande et al.* [DKWK05] nutzten oft vorkommende Teilstrukturen als zusätzliche Eigenschaft, um somit chemische Komponenten besser klassifizieren zu können. Dieses Kapitel wird sich näher mit dem *Frequent-Subgraph-Mining* befassen. Zunächst wird es eine Problemdefinition geben, gefolgt von allgemeinen Definitionen, die für das Verständnis dieses Unterkapitels benötigt werden. Anschließend wird die Arbeit einzelne Probleme von FSM-Algorithmen erläutern und zeigen, wie diese gelöst werden können.

3.2.1 Problemdefinition

Ein FSM-Problem setzt sich aus folgenden drei Eingabeparametern zusammen:

- **Datensatz** D**:** Der Graphdatensatz, in dem nach häufig vorkommenden Teilgraphen gesucht werden soll.
- **Signifikanzfunktion** $\phi(G_s)$**:** Diese Funktion gibt an, wie häufig (mit welcher *Signifikanz*) ein Teilgraph G_s in einem Datensatz auftritt.
- **Mindesthäufigkeit**[1] θ**:** Das ist die Häufigkeit, mit der ein Teilgraph G_s in einem Datensatz enthalten sein muss, damit er in die Ergebnismenge A aufgenommen wird. Diese Häufigkeit wird vom Nutzer festgelegt.

[1] entspricht dem *min-support* im klassischen *Data-Mining*

Ziel ist es dann, eine Menge A (im Folgenden als Ergebnismenge bezeichnet) zu finden, in der alle Teilgraphen G_s enthalten sind, deren *Signifikanz* höher oder gleich θ sind. Die Menge A ist dann wie folgt definiert:

Definition 3.2 (Ergebnismenge häufiger Teilgraphen)**.** $A = \{G_s \in D | \phi(G_s) \geq \theta\}$

3.2.2 Allgemeine Definitionen

Wie bei der *Klassifikation* oder dem *Clustering*, lässt sich auch das FSM in verschiedene Klassen unterteilen. Die Klassen sind dabei vom Datensatz abhängig, auf dem sie arbeiten sollen. Daraus ergibt sich folgende Aufteilung:

- **Transactional-Data:** Der Datensatz umfasst eine Menge von kleineren Graphen. Das Ziel des FSM ist hier, herauszufinden, welche Teilgraphen in der Menge von Graphen häufig zu finden sind.
- **Large-Sparse-Graph-Data:** Der Datensatz umfasst einen großen zusammenhängenden Graphen. Das Ziel des FSM ist hier, herauszufinden, welche Teilgraphen oft in dem großen zusammenhängenden Graphen zu finden sind.

In beiden Szenarien treten unterschiedliche Probleme auf. Deswegen wird in den folgenden Abschnitten zwischen diesen beiden Szenarien unterschieden. Dennoch ist der Aufbau der FSM-Algorithmen im Allgemeinen immer der Gleiche:

1. Der Algorithmus erzeugt Kandidaten G_s (Kap. 3.2.3).
2. Anschließend wird überprüft, ob die jeweiligen Kandidaten im Datensatz auftreten (Kap. 3.2.4).
3. Falls ein Kandidat in einem Graphen aus dem Datensatz enthalten ist, berechnet der Algorithmus die *Signifikanz* des Kandidaten (Kap. 3.2.5).
4. Liegt die *Signifikanz* des Kandidaten über dem festgelegten θ ($\phi(G_s) \geq \theta$), wird der Kandidat in die Ergebnismenge A aufgenommen.

Die nächsten Abschnitte befassen sich mit den einzelnen Schritten eines FSM-Algorithmus, da bei jedem Schritt unterschiedliche Probleme auftreten, welche zusätzlich auch davon abhängig sind, auf welchem Datensatz gearbeitet wird.

3.2.3 Kandidatengenerierung

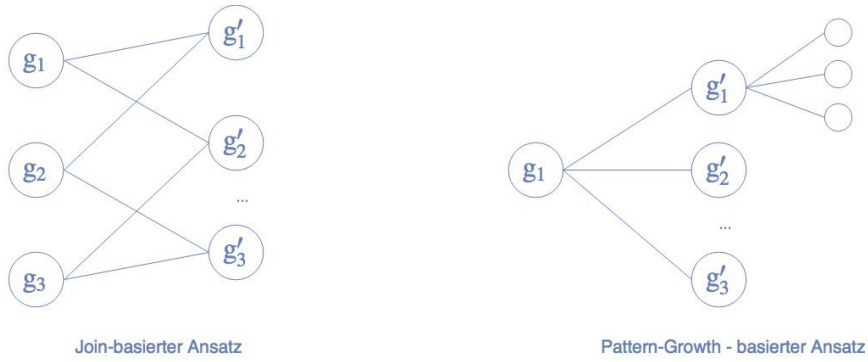


Abbildung 3.2: Die zwei unterschiedlichen Ansätze, um Kandidaten zu generieren.

Die Kandidatengenerierung ist ein fundamentaler Teil von jedem FSM-Algorithmus. Denn beim FSM wird dem Algorithmus nicht explizit vorgegeben, nach welchen Mustern bzw. welchen Teilgraphen gesucht werden soll, sondern der Algorithmus soll selbstständig Teilgraphen generieren und anschließend überprüfen, ob diese Teilgraphen häufig in dem gegebenen Datensatz vorkommen. Dazu wird zunächst definiert, was man unter einem Teilgraph versteht.

Definition 3.3 (Teilgraph). *Gegeben sei ein Graph $G = (V, E)$. Ein Graph $G_s = (V_s, E_s)$ heißt Teilgraph, wenn $V_s \subseteq V$ und $E_s \subseteq E$.*

Die Kandidatengenerierung, also die Generierung von Teilgraphen, läuft für die unterschiedlichen Datensätze (*Transactional-Data, Large-Sparse-Graph-Data*) generell gleich ab. Dennoch gibt es unterschiedliche Ansätze, um Kandidaten zu generieren (wie in Abb. 3.2 zu sehen):

- *Join*-basierter-Ansatz (auch *Apriori*-Ansatz genannt)
- *Pattern-Growth*-Ansatz

Um den Suchraum der möglichen Kandidaten einzuschränken, machen sich die Algorithmen die *Anti-Monotonie*-Eigenschaft (auch *Apriori*-Eigenschaft genannt) zunutze. Dadurch kann der Suchraum deutlich eingegrenzt werden, da nicht mehr alle Möglichkeiten durchsucht werden müssen.

Definition 3.4 (Anti-Monotonie). *Ein Teilgraph G_s ist nur dann häufig, wenn all seine Teilgraphen häufig sind.*

Dadurch, dass ein Teilgraph G_s nur dann in einem Graphen G häufig auftreten kann, wenn all seine Teilgraphen ebenfalls häufig in dem Graphen G auftreten, kann man als Grundlage für die Kandidatengenerierung auch nur Teilgraphen verwenden, von denen man schon weiß, dass sie häufig in G auftreten.

Beispiel 3.4.1. *Man hat einen Teilgraphen der Größe k (im Folgenden als k-Teilgraph bezeichnet). Dieser k-Teilgraph kann nach der Anti-Monotonie-Eigenschaft nur dann häufig in einem Graph G vorkommen, wenn alle seine $k-1$-Teilgraphen ebenfalls häufig in G vorkommen.*

Und genau diese Eigenschaft nutzen die oben aufgeführten Ansätze zur Kandidatengenerierung aus. Aber wann ist ein Teilgraph G_s häufig? Aus der Definition für die Ergbnismenge A (Def. 3.2) lässt sich die Definition für einen häufigen Teilgraphen ableiten.

Definition 3.5 (Häufiger Teilgraph)**.** *Ein Teilgraph G_s heißt häufig, wenn $\phi(G_s) \geq \theta$ erfüllt ist.*

Im Folgenden werden die beiden Ansätze zur Kandidatengenerierung im Detail vorgestellt.

Join-basierter Ansatz

Diese Strategie richtet sich sehr stark an dem *Apriori*-Algorithmus aus, welcher in dem Gebiet der Datenbanken häufig für die *Assoziationsanalyse* eingesetzt wird. Deswegen findet man diesen Ansatz auch oft in der Fachliteratur unter dem Namen *Apriori-basierter Ansatz*. Hierbei werden neue Kandidaten aus den Kandidaten generiert, für die schon überprüft wurde, dass sie häufig in einer Menge von Graphen bzw. in einem einzelnen Graph vorkommen. Dabei geht man von unten nach oben (*bottom-up*) vor. Im Detail funktioniert die Kandidatengenerierung folgendermaßen:

1. Zunächst werden alle möglichen Teilgraphen mit der Größe $k = 1$ ($k =$ Anzahl der Kanten) erzeugt.
2. Für die neu erzeugten Teilgraphen wird überprüft, ob diese häufig in einer Menge von Graphen bzw. einem einzelnen Graphen vorkommen. Ist dies der Fall, werden sie in die Ergebnismenge A aufgenommen.
3. Neue $k+1$-Teilgraphen werden durch einen *Join* von zwei ähnlichen, aber nicht vollkommen identischen, k-Teilgraphen erzeugt, welche in der Menge A enthalten sind.
4. Der Algorithmus bricht ab, wenn in einer Iteration keine neuen Teilgraphen mehr in A aufgenommen werden.

Der wichtigste Schritt hierbei ist der *Join*, also der Schritt, bei dem aus zwei ähnlichen $k-1$-Teilgraphen ein neuer k-Teilgraph entsteht. Denn es wurde festgelegt, dass aus zwei $k-1$-Teilgraphen nur dann ein neuer k-Teilgraph entstehen darf, wenn sich die beiden $k-1$-Teilgraphen *ähnlich* sind. Dazu muss natürlich aber genau definiert werden, was man unter *Ähnlichkeit* von zwei Teilgraphen versteht.

Definition 3.6 (Ähnlichkeit von zwei Teilgraphen)**.** *Seien $G_1^k = (V_1, E_1)$ und $G_2^k = (V_2, E_2)$ zwei Teilgraphen mit k Knoten, dann heißen G_1^k und G_2^k ähnlich, wenn $V(G_1^{k-1}) = V(G_2^{k-1})$ und $E(G_1^{k-1}) = E(G_2^{k-1})$ sind.*

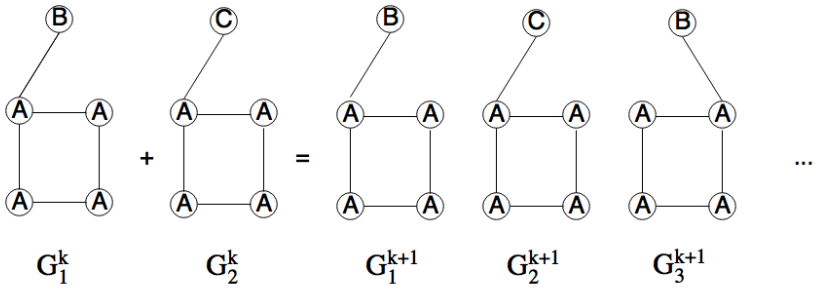


Abbildung 3.3: Der *Join* zweier *ähnlicher* Teilgraphen G_1^k und G_2^k.

Aus dieser Definition für die Ähnlichkeit von zwei Teilgraphen lässt sich also schließen, dass sie einen gemeinsamen Teilgraphen haben müssen, beide Graphen also bis auf eine Kante E identisch sein müssen, damit die beiden Teilgraphen ähnlich sind (wie in Abb. 3.3 zu sehen). Hierbei ergibt sich ein weiteres Problem, nämlich das sogenannte *Isomorphie-Problem*, welches aber im nächsten Kapitel näher behandelt wird.
Dieser wichtige *Join*, welcher ausgeführt werden muss, um neue Kandidaten zu generieren, stellt allerdings auch zugleich eine Schwäche dieses Ansatzes dar. Denn diese Operation ist sehr teuer, da man dazu ermitteln müsste, welche Teilgraphen miteinander zu einem neuen Teilgraph verbunden werden dürften. Das verursacht einen erheblichen Overhead in der Berechnung. Deswegen wurde ein anderes Verfahren entwickelt, welches keinen *Join* benötigt.

Pattern-Growth-Ansatz
Bei diesem Vorgehen werden neue Kandidaten, ebenfalls aus den schon in der Ergebnismenge A enthaltenen Teilgraphen, gewonnen. Der Unterschied zum *Join-basierten* Ansatz ist allerdings, dass bei diesem Verfahren kein *Join* benötigt wird. Im Detail funktioniert die Kandidatengenerierung dann folgendermaßen:

1. Zunächst werden alle möglichen Teilgraphen mit der Größe $k = 1$ (k = Anzahl der Kanten) erzeugt.

2. Für die neu erzeugten Teilgraphen wird überprüft, ob diese häufig in einer Menge von Graphen bzw. einem einzelnen Graphen vorkommen. Ist dies der Fall, werden sie in die Ergebnismenge A aufgenommen.

3. Neue $k+1$-Teilgraphen werden durch das zufällige Hinzufügen einer Kante zu einem k-Teilgraphen, welcher in der Menge A enthalten ist, erzeugt. Optional kann dabei auch ein neuer Knoten hinzugefügt werden, je nachdem ob die neue Kante zwei schon vorhandene Knoten miteinander verbindet oder einen schon vorhanden mit einem noch nicht vorhandenen Knoten.

4. Der Algorithmus bricht ab, wenn in einer Iteration keine neuen Teilgraphen mehr in A aufgenommen werden.

Dadurch, dass bei diesem Ansatz kein *Join* benötigt wird, entfällt der Overhead in der Berechnung und der Algorithmus ist somit schneller. Ein Problem, welches allerdings bei diesem Ansatz auftritt, ist, dass ein Teilgraph mehrere Male erzeugt werden kann.
Nach *Khan und Ranu* [KR17] kann man diesem Problem entgegenwirken, indem man die sogenannte *right-most-extension* benutzt. Dabei wird nur der Knoten eines Graphen G um eine Kante erweitert, der sich am Ende des *ganz rechten Pfads* des *Tiefensuche-Baums* von G befindet. Dieses Verfahren wird nach *Khan und Ranu* auch in dem *gSpan*-Algorithmus genutzt.
Beispiele für konkrete Algorithmen, die den *Pattern-Growth*-Ansatz nutzen, sind:

- *gSpan* von *Yan und Han* [YH02]
- *MoFa* von *Borgelt und Berthold* [BB02]
- *FFSM* von *Huan et al.* [HWP03]
- *SPIN* von *Huan et al.* [HWPY04]
- *Gaston* von *Nijssen und Kok* [NK04]
- *DESSIN* von *Li et al.* [LZY10]

3.2.4 Isomorphie

Nachdem das Problem der Kandidatengenerierung im vorherigen Abschnitt näher erläutert wurde, wird die Arbeit nun eine weitere Herausforderung aufzeigen, welche im zweiten Schritt (Überprüfung, ob ein Kandidat in einem Datensatz vorkommt) auftritt. Wie der Name des aktuellen Abschnitts schon verrät, handelt es sich dabei um das Problem der *Isomorphie* bzw. der *Teilgraph-Isomorphie*. Die Herausforderung besteht darin, festzustellen, ob ein Kandidat G_s ein Teilgraph eines anderen Graphen G ist. Dazu wird zunächst definiert, was man unter der Isomorphie von zwei Graphen versteht.

Definition 3.7 (Isomorphie)**.** *Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ sind isomorph, wenn sie die gleiche Topologie besitzen. Das heißt, es gibt eine Zuordnung von V_1 zu V_2, sodass jede Kante in E_1 einer Kante in E_2 zugeordnet werden kann. Im Fall von gelabelten Graphen muss die Zuordnung nur Kanten und Knoten auf gleich gelabelte abbilden.*

Aus Definition 3.7 lässt sich nun auch leicht die Definition der *Teilgraph-Isomorphie* ableiten (vgl. [CCA12]):

Definition 3.8 (Teilgraph-Isomorphie)**.** *Für zwei gelabelte Graphen $G = (V, E, l)$ und $G_s = (V_s, E_s, l_s)$ ist eine Teilgraph-Isomorphie eine injektive Funktion $f : V(G_s) \to V(G)$ unter der Bedingung von (s.t.):*

1. $\forall v \in V(G_s) : l_s(v) = l(f(v))$
2. $\forall (u, v) \in E(G_s) : (f(u), f(v)) \in E(G) \wedge l_s(u, v) = l(f(u), f(v))$

wobei l und l_s die Labelingfunktion von G und G_s ist. f wird dann eine Einbettung von G_s in G genannt.

Das bedeutet also, dass die Knotenmenge V, Kantenmenge E und die Labels von G_s eine Teilmenge der Knoten, Kanten und Labels von G sind und G_s somit ein Teilgraph von G ist, da eine *Teilgraph-Isomorphie* zwischen den beiden Graphen besteht ($G_s \subseteq G$). G wird dann auch Supergraph von G_s genannt.
Natürlich kann es sehr rechenaufwändig werden, für jeden Kandidaten zu überprüfen, ob eine *Isomorphie / Teilgraph-Isomorphie* zu einem anderen Graphen besteht. Denn die *Isomorphie / Teilgraph-Isomorphie* gehört zu den NP-vollständigen Problemen. Ein Weg, um zu überprüfen, ob zwei Graphen *isomorph* zueinander sind, ist das *kanonische Markieren* (engl. Canonical Labeling).

Kanonisches Markieren

Beim *Kanonischen Markieren* (vgl. [YH02]) wird jedem Graph ein eindeutiger Code, auf Grundlage seiner Knoten und Kanten, zugewiesen. Wenn man nun feststellen möchte, ob zwei Graphen gleich (*isomorph* zueinander) sind, muss man nur noch die Codes miteinander vergleichen, anstatt die Graphen vollständig zu durchlaufen.
Um jedem Graphen so einen Code zuweisen zu können, wird für jeden Graphen eine Adjazenzmatrix erstellt.

Definition 3.9 (Adjazenzmatrix)**.** *Die Adjazenzmatrix eines Graphen G mit n Knoten ist definiert als $n \times n$-Matrix, wobei die Einträge in der Diagonalen den Knoten und alle restlichen Einträge den Kanten des Graphen G entsprechen. Existiert keine Kante zwischen zwei Knoten, so wird das jeweilige Feld in der Adjazenzmatrix mit 0 belegt.*

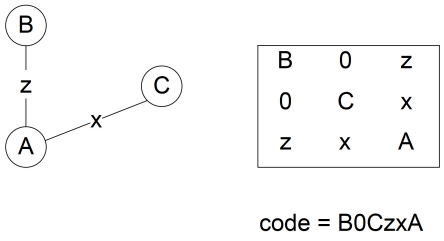


Abbildung 3.4: Die Adjazenzmatrix eines Graphen und der sich daraus ergebene Code.

Mithilfe dieser Adjazenzmatrix kann dann der Code für den jeweiligen Graphen generiert werden. Ein Beispiel für die Adjazenzmatrix eines Graphen und den daraus resultierenden Code wird in Abb. 3.4 gezeigt. In einem späteren Kapitel (6) wird noch näher auf einen Algorithmus für das *kanonische Markieren* eingegangen.

3.2.5 Signifikanzberechnung

Bei der Signifikanzberechnung eines Kandidaten muss man zwischen den zwei unterschiedlichen Szenarien von *Transactional-Data* und *Large-Sparse-Graph-Data* als Datensatz unterscheiden. Die *Signifikanz* eines Kandidaten in einer Menge von Graphen ist recht intuitiv zu berechnen. Es ergibt sich dann folgende Definition:

Definition 3.10 (Signifikanz für einen Kandidaten in einer Menge von Graphen)**.** *Sei G_s der Kandidat für den die Signifikanz aus einem Datensatz $D = \{G_1, G_2, ..., G_n\}$ berechnet werden soll, dann ergibt sich für die Signifikanz folgende Funktion*

$$\phi(G_s) = \frac{|D_{G_s}|}{|D|},$$

wobei $|D_{G_s}|$ die Anzahl der Graphen ist, zu denen G_s isomorph ist.

Für das zweite Szenario (*Large-Sparse-Graph-Data*) lässt sich die *Signifikanz* eines Kandidaten hingegen nicht so intuitiv definieren, wie es bei dem ersten Szenario der Fall war. Das liegt unter anderem daran, dass sich die verschiedenen Instanzen eines Kandidaten in einem Graphen überlappen können und dadurch die *Anti-Monotonie*-Eigenschaft verletzen könnte, welche aber ein wichtiger Bestandteil der meisten FSM-Algorithmen ist, da durch sie der Suchraum eingegrenzt werden kann. Deswegen haben sich in den letzten Jahren unterschiedliche Definitionen für die *Signifikanz* eines Kandidaten ergeben. Im Folgenden werden einige dieser Definitionen im Detail behandelt.

MIS (maximum independent set)
Kuramochi et al. [KK05] entwickelten einen Algorithmus zur Berechnung der *Signifikanz* eines Kandidaten auf Basis der *maximal unabhängigen Menge* (siehe Unterkapitel 2.4). Um diese Menge berechnen zu können, wird bei MIS ein *Überlappungsgraph* konstruiert. Zunächst werden die Definitionen für den *edge-disjoint* und die *simple Überlappung* zwischen zwei Graphen bzw. zwei Instanzen folgen, damit anschließend erläutert werden kann, wie der *Überlappungsgraph* konstruiert wird.

Definition 3.11 (Edge-disjoint von Instanzen)**.** *Zwei Graphen* $G_1 = (V_1, E_1)$ *und* $G_2 = (V_2, E_2)$ *sind edge-disjoint, wenn* $E_1 \cap E_2 = \emptyset$.

Definition 3.12 (Simple Überlappung)**.** *Gegeben ein Kandidat* $G_s = (V(G_s), E(G_s))$. *Es existiert eine simple Überlappung von zwei Instanzen* φ_1 *und* φ_2 *eines Kandidaten* G_s, *wenn* $\varphi_1(E(G_s)) \cap \varphi_2(E(G_s)) \neq \emptyset$.

Daraus lässt sich dann die Definition für den vorher erwähnten *Überlappungsgraph* erschließen:

Definition 3.13 (Überlappungsgraph)**.** *Gegeben sei die Menge aller Instanzen eines Kandidaten* G_s *in einem Graphen* G. *Der Überlappungsgraph von* G_s *in* G *ist dann der Graph, dessen Knoten nicht-identische Instanzen sind und dessen Kanten die Knoten verbindet, dessen Instanzen nicht edge-disjoint sind, also einer simplen Überlappung entsprechen.*

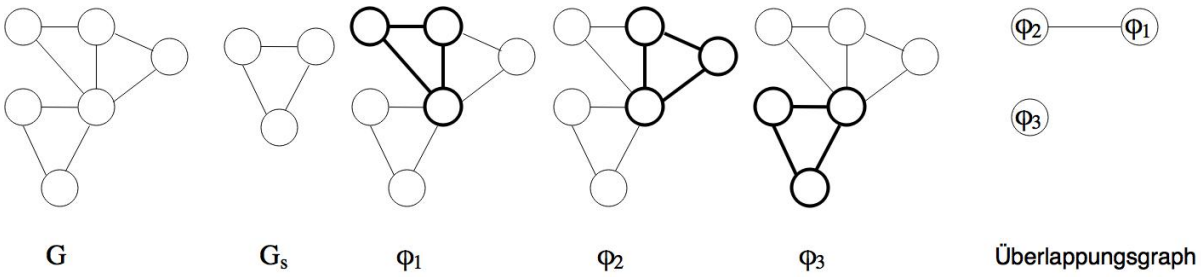


Abbildung 3.5: Die Instanzen $\varphi_1, \varphi_2, \varphi_3$ des Kandidaten G_s in einem Graph G und der sich daraus ergebene *Überlappungsgraph*.

Beispielhaft wird ein solcher *Überlappungsgraph* in Abb. 3.5 dargestellt.

Beispiel 3.13.1. *Man hat einen Graph* G *und einen Kandidaten* G_s *mit all seinen Instanzen* $\varphi_1, \varphi_2, \varphi_3$. *Der Überlappungsgraph setzt sich dann aus den jeweiligen Instanzen, welche als Knoten modelliert werden, und den Kanten, welche eine simple Überlappung repräsentieren, zusammen. In unserem Beispiel überlappen sich* φ_1 *und* φ_2, *da sie sich eine Kante* e *teilen und somit durch eine Kante im Überlappungsgraph verbunden sind.* φ_3 *hingegen teilt sich keine Kante mit einer anderen Instanz und ist dadurch mit keinem anderen Knoten verbunden.*

Die *Signifikanz* eines Kandidaten $\phi(G_s)$ ist dann definiert als die Größe der *maximale unabhängigen Menge* (engl. *maximum independent set*) des Überlappungsgraphen. Diese Definition der *Signifikanz* erfüllt die *Anti-Monotonie*-Eigenschaft.

Schädliche Überlappung (Harmful Overlap)
Eine weitere Definition für die *Signifikanz* eines Kandidaten ist die von *Fiedler und Borgelt* [FB07]. Hierbei wird, genauso wie beim *MIS*, ein Überlappungsgraph konstruiert und die Signifikanz ist dann die maximale Größe der unabhängigen Menge des Überlappungsgraphen. Allerdings haben *Fiedler und Borgelt* eine andere Art der Definition für die Überlappung zweier Instanzen festgelegt, die sogenannte *schädliche Überlappung* (engl. harmful overlap). Im Gegensatz zu der *simplen Überlappung* von *Kuramochi et al.* entsteht eine *schädliche Überlappung* nur, wenn sich zwei Instanzen einen Knoten teilen. Dadurch ergibt sich folgende Definition für die *schädliche Überlappung*:

Definition 3.14 (Schädliche Überlappung)**.** *Gegeben sei ein Graph $G = (V, E)$, ein Kandidat $G_s = (V_s, E_s)$ und zwei Instanzen φ_1, φ_2 von G_s in G. Eine schädliche Überlappung von φ_1 und φ_2 existiert dann, wenn sie isomorph (siehe Def. 3.7) zueinander sind oder ein (nichtleerer) Teilgraph G'_s von G_s existiert, sodass die G'_s-Eltern φ'_1 und φ'_2 von φ_1 und φ_2 isomorph zueinander sind.*

Würde man diese Definition nun auf das Beispiel aus Abb. 3.5 anwenden, so würde im *Überlappungsgraph* jeweils eine Kante zwischen φ_1 und φ_3 und φ_2 und φ_3 hinzukommen.
Auch die *schädliche Überlappung* erfüllt die *Anti-Monotonie*-Eigenschaft (Beweis siehe [FB07]) und liegt zudem häufig näher an der wahren *Signifikanz* eines Kandidaten als das *MIS*.

Minimum-Image-Based-Support (MNI)
Die Berechnung für die oben eingeführten Signifikanzfunktionen *MIS* und *schädliche Überlappung* sind jeweils NP-vollständig, da für die Konstruktion des Überlappungsgraphen die *Teilgraph-Isomorphie* aller Instanzen eines Musters berechnet werden muss. Um dieser Herausforderung zu begegnen, haben sich *Bringmann et al.* [BN08] eine einfachere Signifikanzfunktion ausgedacht (*MNI*), welche auf der Anzahl eindeutiger Knoten in einem Graph $G = (V, E)$ basiert, die einem Knoten eines Kandidaten $G_s = (V_s, E_s)$ zugeordnet werden können.

Definition 3.15 (Minimum-Image-Based-Support)**.** *Gegeben sei ein Graph $G = (V, E)$ und einen Kandidaten $G_s = (V_s, E_s)$, dann ist der Minimum-Image-Based-Support definiert als:*

$$\sigma_{\wedge}(G_s, G) = \min_{v \in V_s} |\varphi_i(v) : \varphi_i \textit{ ist ein Vorkommen von } G_s \textit{ in } G|.$$

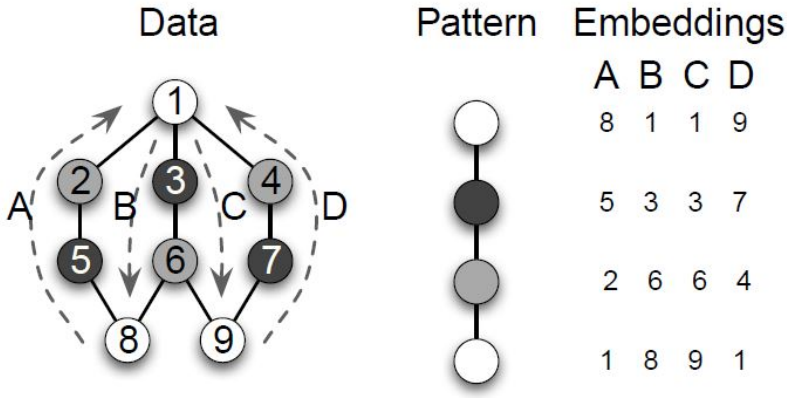


Abbildung 3.6: Ein Graph mit vier verschiedenen Instanzen eines Kandidaten. Der *Minimum-Image-Based-Support* würde hier 3 betragen. (Quelle: [BN08]).

In Abbildung 3.6 ist ein Beispiel für die *MNI*-Signifikanz zu sehen. Der Teilgraph kommt insgesamt 4 Mal im Input-Graphen vor. Seine Signifikanz beträgt allerdings nur 3, da jedem Knoten des Teilgraph nur drei unterschiedliche Knoten im Input-Graph zugeordnet werden können.
Zusätzlich hat *MNI* folgende zwei Vorteile, welche die Berechnung der *Signifikanz* wesentlich effizienter machen (vgl. [CCA12]):

1. Anstatt von $O(n^2)$ möglichen Überlappungen, wo n die mögliche Anzahl von Instanzen eines Kandidaten ist, muss die Methode nur noch eine Menge von Knoten für jeden Knoten eines Kandidaten überprüfen, was in $O(n)$-Schritten gemacht werden kann.

2. Die Methode muss kein NP-vollständiges *MIS*-Problem lösen.

Auch *MNI* erfüllt die *Anti-Monotonie*-Eigenschaft.

3.2.6 Mining von geschlossenen und maximalen Teilgraphen

Ein weiteres Problem beim Mining von häufigen Teilgraphen ist, dass häufig eine große Anzahl an Mustern gefunden wird. So kann ein Graph der Größe n, 2^n-häufige Teilgraphen enthalten (vgl. [KR17]). Die Konsequenz daraus ist, dass die Ergebnismenge A mit der Verringerung der Mindesthäufigkeit θ exponentiell anwachsen kann. Diese hohe Anzahl an häufigen Mustern macht weiterführende Analysen unnötig schwer. Deswegen wurde die Idee von *geschlossenen* und *maximalen* Teilgraphen eingeführt.

Definition 3.16 (Geschlossener Teilgraph)**.** *Ein Teilgraph G_s ist häufig und heißt geschlossen, wenn G_s häufig ist und kein Supergraph $G'_s \supseteq G_s$ existiert, sodass G'_s nicht genauso häufig ist wie G_s.*

Definition 3.17 (Maximaler Teilgraph)**.** *Ein Teilgraph G_s ist häufig und heißt maximal, wenn G_s häufig ist und kein Supergraph $G'_s \supseteq G_s$ existiert, sodass G'_s nicht häufig ist.*

Sei A^h die Menge der *häufig geschlossenen* Teilgraphen und A^{max} die Menge der *häufig maximalen* Teilgraphen. So ergibt sich folgender Zusammenhang zwischen *häufigen* A, *häufig geschlossenen* A^h und *häufig maximalen* A^{max} Teilgraphen:

$$A^{max} \subseteq A^h \subseteq A$$

Dadurch kann man die Anzahl der aus einem *Frequent-Subgraph-Mining* resultierenden Graphen einschränken und die Ergebnismenge somit übersichtlicher gestalten. Aus der Menge A^h lässt sich sogar die Menge A wieder rekonstruieren. Bei A^{max} ist das allerdings nicht möglich(vgl. [KR17]).
Es gibt aber auch noch andere Möglichkeiten, um den Ergebnisgraphen bzw. die Menge der Ergebnisgraphen einschränken zu können. So kann man z.B. alle Knoten in einen Knoten zusammenfassen, die die gleichen Attribute, Labels und Kanten zu ihren Nachbarknoten besitzen. Dadurch kann man die Größe eines Graphen minimieren. Für die Weiterverarbeitung durch andere Algorithmen ist dieses Verfahren allerdings nicht zu empfehlen, da ansonsten Informationen verloren gehen könnten.

3.3 GraMi

Algorithmen für das *Frequent-Subgraph-Mining* sind in den letzten Jahren einige entwickelt worden. Beispiel dafür sind *gSpan* von *Yan und Han* [YH02] oder auch *Gaston* von *Nijssen und Kok* [NK04]. Ein weiterer Algorithmus, der allerdings einen ganz anderen Ansatz hat, ist *GraMi* ([EASK14]). Der Algorithmus nutzt *MNI*, um die *Signifikanz* von Kandidaten zu berechnen und modelliert das Isomorphie-Problem bei der Signifikanzberechnung als ein *Constraint-Satisfaction-Problem* (kurz CSP), damit es effizienter gelöst werden kann. Zusätzlich werden bei *GraMi* unterschiedliche Heuristiken, wie das *Push-Down-Pruning* oder auch *Lazy-Search* genutzt, um den Suchraum zusätzlich einschränken zu können und die Berechnung dadurch noch effizienter zu machen.
In den folgenden Abschnitten wird die Arbeit erklären, wie *GraMi* das Isomorphie-Problem als *CSP* modelliert und wie die oben erwähnten Heuristiken funktionieren und wo sie ihre Anwendung im Algorithmus finden. Dazu wird zunächst ein kleiner Exkurs gegeben, was ein *CSP* ist und welche Techniken es gibt, um sie zu lösen.

3.3.1 Das CSP-Modell

Zunächst wird in diesem Abschnitt eine Definition gegeben, was man unter einem *CSP* versteht.

Definition 3.18 (Constraint-Satisfaction-Problem)**.** *Ein Constraint-Satisfaction-Problem (kurz CSP) ist ein Tupel (X, D, C), bei dem X eine geordnete Menge von Variablen, D eine Menge von Domänen, die den Variablen zugeorndet werden können und C die Menge von Bedingungen (constraints) zwischen den Variablen ist. Eine Lösung für ein CSP ist eine Zuordnung von Domänen aus D zu den jeweiligen Variablen in X, sodass keine Bedingungen aus C verletzt werden.*

Im Folgenden wird das spezifische *CSP*, welches *GraMi* nutzt, als *FSM-CSP* bezeichnet. Definition 3.19 legt dann fest, wie *GraMi* aus dem Isomorphie-Problem ein *FSM-CSP* modelliert. In Abbildung 3.7 sind zudem beispielhaft ein Input-Graph (a) und zwei Teilgraphen (b) und (c) zu sehen, anhand derer im Folgenden noch ein Beispiel zum besseren Verständnis erläutert wird. Auch das Lösen des *FSM-CSP* wird mithilfe dieses Beispiels beschrieben.

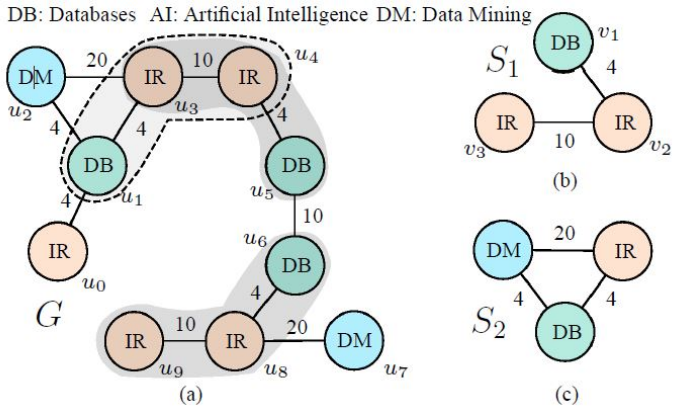


Abbildung 3.7: Ein Graph G und zwei Teilgraphen S_1 und S_2 (Quelle: [EASK14]).

Definition 3.19 (FSM-CSP). *Sei $G_s = (V_s, E_s, L_s)$ ein Teilgraph vom Graph $G = (V, E, L)$, dann ist das FSM-CSP, ein $CSP = (X, D, C)$ für das gilt:*

1. *X enthält Variablen x_v für jeden Knoten $v \in V_s$.*
2. *D ist eine Menge von Domänen für jede Variable $x_v \in X$. Jede Domäne ist eine Teilmenge von V.*
3. *Seien in C die folgenden Bedingungen enthalten:*
   (a) *$x_v \neq x'_v$, für alle distinkten Variablen $x_v, x'_v \in X$.*
   (b) *$L(x_v) = L_s(v)$ für alle Variablen $x_v \in X$.*
   (c) *$L(x_v, x'_v) = L_s(v, v')$ für alle Variablen $x_v, x'_v \in X$, sodass $(v, v') \in E_s$.*

Um die Notation zu vereinfachen, wird in dieser Arbeit, wann immer es sich aus dem Kontext ergibt, v als Knoten von G_s und seiner zugeordneten Variable x_v des *FSM-CSP* bezeichnet. Folgendes Beispiel zeigt die Notation des *FSM-CSP* aus Abb. 3.7:

Beispiel 3.19.1 (vgl. [EASK14]). *Das FSM-CSP von S_1 zu G:*

$$\left(\begin{array}{c} (v_1, v_2, v_3), \{\{u_o, ..., u_9\}, ..., \{u_0, ..., u_9\}\}, \\ \{v_1 \neq v_2 \neq v_3, L(v_1) = DB, L(v_2) = L(v_3) = IR, \\ L(v_1, v_2) = 4, L(v_2, v_3) = 10\} \end{array} \right)$$

Intuitiv ist die Lösung des *FSM-CSP* einfach nur eine Zuweisung von verschiedenen Knoten von G zu einem Knoten von G_s, sodass die Attribute der zueinander gehörigen Knoten und Kanten übereinstimmen. Für das Beispiel in Abb. 3.7 wäre also eine gültige Lösung $(v_1, v_2, v_3) = (u_1, u_3, u_4)$. Somit kann man definieren, was eine gültige Zuweisung ist.

Definition 3.20 (Gültige Zuweisung)**.** *Eine Zuweisung eines Knoten u zu einer Variable v ist gültig, wenn und wirklich nur wenn eine Lösung existiert, die u zu v zuweist. Beachte, dass jede gültige Zuweisung einer Isomorphie entspricht.*

Nun ist noch zu definieren, wann eine Zuweisung, also in unserem Fall ein Teilgraph, in die Ergebnismenge aufgenommen werden darf. Dazu benutzt *GraMi*, wie eingangs schon erwähnt, die *MNI*-Methode zur Signifikanzberechnung eines Kandidaten/Teilgraphen.

Satz 3.1. *Sei (X, D, C) das FSM-CSP. Die MNI-Signifikanz von G_s in G erfüllt die Mindesthäufigkeit θ, wenn und wirklich nur wenn jede Variable in X mindestens θ-verschiedene/distinkte Zuweisungen hat (Teilgraph-Isomorphie von G_s in G).*

3.3.2 Der Algorithmus

In diesem Abschnitt wird auf die Funktionsweise des Algorithmus im Detail eingegangen. Alle dargestellten Algorithmen entsprechen der Arbeit von *Elseidy et al.* [EASK14] und werden mit leicht veränderter Notation dargestellt, um die Konsistenz zu wahren.
Die Algorithmen *FSM* und *TeilgraphErweitern* sind für die Kandidatengenerierung. Auf diese beiden Algorithmen wird als Erstes eingegangen. Anschließend werden die Algorithmen *istHäufigCSP* und *istHäufigHeuristik* vorgestellt, welche dafür zuständig sind, die *Signifikanz* eines Kandidaten zu berechnen. *FSM* startet damit, alle häufigen Kanten in $fEdges$ zu speichern

Algorithmus 1: FSM

Input: Ein Graph G und die Mindesthäufigkeit θ
Output: Alle häufigen Teilgraphen G_s von G
$ergebnis \leftarrow \emptyset$;
Lass $fEdges$ die Menge aller häufigen Kanten von G sein ;
foreach $e \in fEdges$ **do**
    $ergebnis \leftarrow ergebnis \cup$ TeilgraphErweitern$(e, G, \theta, fEdges)$;
    Entferne e aus G und $fEdges$;
end
return $ergebnis$;

(alle Kanten, die eine höhere *Signifikanz* als θ haben). Denn basierend auf der Anti-Monotonie-Eigenschaft, können nur diese Kanten in einem häufigen Teilgraphen vorkommen. Für jede häufige Kante wird dann *TeilgraphErweitern* ausgeführt und das Ergebnis dieses Algorithmus dann mit in die Ergebnismenge *ergebnis* getan. Anschließend wird *ergebnis* zurückgegeben.

Algorithmus 2: TeilgraphErweitern

Input: Ein Teilgraph G_s, Mindesthäufigkeit θ und eine Menge von Kanten $fEdges$ von G

Output: Alle häufigen Teilgraphen von G die G_s enthalten

$ergebnis \leftarrow G_s, kandidaten \leftarrow \emptyset$;

foreach *Kante e in fEdges und jeden Knoten u aus* G_s **do**

  if *e kann genutzt werden, um u zu erweitern* **then**

    Lass *ext* die Erweiterung von G_s mit e sein;

    if *ext wurde noch nicht generiert* **then**

      $kandidaten \leftarrow kandidaten \cup ext$;

    end

  end

end

foreach $c \in kandidaten$ **do**

  if $c \geq \theta$ **then**

    $ergebnis \leftarrow ergebnis \cup$ TeilgraphErweitern$(c, G, \theta, fEdges)$;

  end

end

return *ergebnis*;

TeilgraphErweitern überprüft für jede häufige Kante e und jeden Knoten u eines Kandidaten G_s, ob dieser mit der jeweiligen Kante erweitert werden kann. Wenn dies der Fall ist und der dadurch neu entstandene Kandidat noch nicht generiert wurde, wird er der Kandidatenmenge hinzugefügt. Anschließend wird für jeden Kandidaten c aus der Kandidatenmenge überprüft, ob dieser häufig ist. Ist dies der Fall, wird *TeilgraphErweitern* für den jeweiligen Kandidaten erneut aufgerufen. Am Ende wird die Menge häufiger Kandidaten zurückgegeben.

Algorithmus 3: istHäufigCSP

Input: Graph G_s und G und die Mindesthäufigkeit θ
Output: *true* wenn G_s häufig ist, sonst $false$
Stellen sie sich ein *FSM-CSP* vor;
Wende Knoten- ud Kantenkonsistenz an;
if *Größe von jeder Domäne ist kleiner als* θ **then**
    return $false$;
end
foreach *Lösung loesung von* G_s *zum FSM-CSP* **do**
    Markiere alle Knoten von *loesung* in den zugeordneten Domänen;
    if *Alle Domänen haben mindestens* θ*-markierte Knoten* **then**
        return *true*;
    end
end
return $false$;

Die Häufigkeit, die in den bisher vorgestellten Algorithmen genutzt wurde, wird in *istHäufigCSP* genauer beschrieben. Zum Anfang wendet *istHäufigCSP* die Kanten- und Knotenkonsistenz an. Die Knotenkonsistenz entfernt ungeeignete Knoten aus der Domäne (z.B. Knoten, die andere Attribute haben) und die Kantenkonsistenz stellt die Konsistenz zwischen den Zuweisungen von zwei Knoten sicher. Den interessierten Leser, der sich näher mit Kanten- und Knotenkonsistenz beschäftigen möchte, verweise ich an dieser Stelle auf die Arbeit von *Alan K. Mackworth* ([Mac77]). Wenn nach der Knoten- und Kantenkonsistenz die Größe einer Domäne kleiner als θ ist, wird $false$ zurückgegeben. Ansonsten werden für jede Lösung des *FSM-CSP* die zugewiesenen Knoten in den Domänen markiert. Wenn alle Domänen mindestens θ-markierte Knoten haben, wird *true* zurückgeliefert.

Algorithmus 4: istHäufigHeuristik

Input: Graph G_s und G und die Mindesthäufigkeit θ
Output: *true* wenn G_s häufig ist, sonst $false$
Stellen sie sich ein *FSM-CSP* vor;
Wende Knoten- ud Kantenkonsistenz an;
foreach *Variable v mit der Domäne D* **do**
    $count \leftarrow 0$;
    Wende Kantenkonsistenz an;
    if *Größe jeder Domäne ist kleiner als* θ **then**
        return $false$;
    end
    foreach *Element u aus D* **do**
        if *u ist schon markiert* **then**
            $count++$;
        end
        if *es gibt eine Lösung loesung die u zu v zuweist* **then**
            Markiere alle Werte von *loesung* in der zugewiesenen Domäne;
            $count++$;
        else
            Entferne u aus D;
        end
        if $count = \theta$ **then**
            Fahre mit der nächsten Variablen fort (Zeile 5);
        end
    end
    return $false$;
end
return *true*;

In *istHäufigCSP* wird einfach nur über die Lösungen für ein *FSM-CSP* iteriert. Um diesen Suchprozess lenken zu können, ist der Algorithmus *istHäufigHeuristik* da. Dabei überprüft der Algorithmus pro Schritt für eine Variable, ob sie θ-zulässige Zuweisungen besitzt. Wenn diese gefunden werden, fährt der Algorithmus mit der nächsten Variable fort und wiederholt den Prozess.

3.3.3 Zusätzliche Heuristiken

In der Einleitung wurde schon erwähnt, dass bei *GraMi* auch zusätzliche Heuristiken zum Einsatz kommen, um die Berechnung effizienter und somit den gesamten Algorithmus schneller zu machen. In diesem Abschnitt werden *Push-Down-Pruning* und *Lazy-Search* vorgestellt.

Push-Down-Pruning

Der Kandidaten-Generierungs-Baum wird erzeugt, indem zu jedem Eltern-Kandidaten eine Kante hinzugefügt wird. Weil die Eltern-Kandidaten somit Teilgraphen ihrer Kinder sind, können Zuweisungen, die aus den Domänen der Eltern-Kandidaten gelöscht wurden, auch nicht mehr in den Domänen ihrer Kinder vorkommen (*Anti-Monotonie*-Eigenschaft). Wie in Abb. 3.8 zu sehen, werden S_2 und S_3 aus S_1 erzeugt. Somit sind S_2 und S_3 die Kinder von S_1. In den Domänen der Variablen v_1, v_2, v_3 wurden die Belegungen a_3, b_1, a_3 entfernt, da sie ungültig waren. Somit sind diese Belegungen auch für S_2 und S_3 ungültig und werden aus deren Domänen ebenfalls entfernt (siehe Abb. 3.8). *Push-Down-Pruning* nutzt also die Eigenschaft aus, dass jede ungültige Belegung für einen Kandidaten S, auch für seine Kinder (Supergraphen von S) ungültig ist.

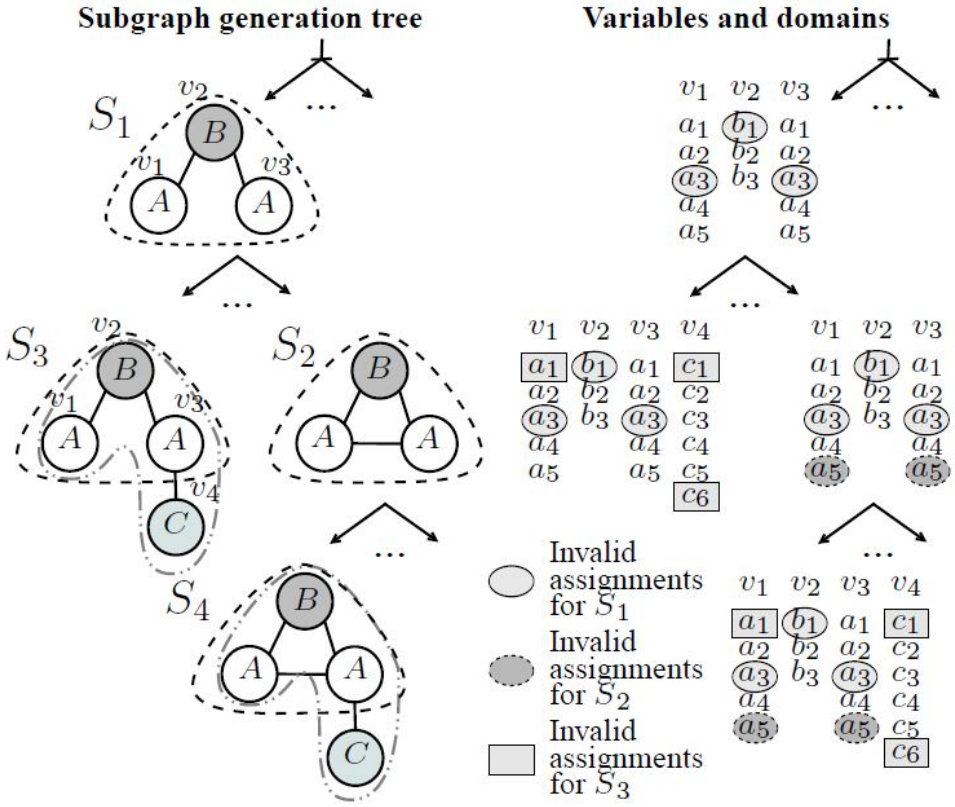


Abbildung 3.8: Ein Kandidaten-Generierungs-Baum und die Domänen der jeweiligen Variablen (Quelle: [EASK14]).

Lazy-Search
Bei dieser Heuristik wird angenommen, dass, wenn eine Suche nach einer gültigen Belegung für einen Kandidaten zu lange dauert, es für die bisher schon gemachten Belegungen wahrscheinlich gar keine Lösung gibt. Dann ist es besser, die bisher schon gemachte Belegung rückgängig zu machen und nach einer alternativen Lösung zu suchen. Dazu legt *GraMi* ein Zeitlimit fest. Wenn dieses Zeitlimit überschritten wird, probiert der Algorithmus eine andere Belegung aus. Die abgebrochene Suche wird aber trotzdem noch im Speicher gehalten. Diese gespeicherte Suche wird dann fortgesetzt, wenn die anderen Belegungen, die nicht abgebrochen wurden, keine Lösung gefunden haben, die die Mindesthäufigkeit θ erfüllen.
Bezogen auf unser Beispiel in Abb. 3.8 würde das bedeuten, wenn die Suche für S_2 abgebrochen wird, würde man mit der Suche für S_3 weitermachen.

Zu erwähnen ist, dass *GraMi* auch noch andere Heuristiken benutzt und auch erweitert werden kann, sodass der Algorithmus auch für die klassische Assoziationsanalyse bzw. Warenkorbanalyse verwendet werden kann. Aber auch hier verweise ich den interessierten Leser wieder auf die Arbeit von *Elseidy et al.* [EASK14], die weitere Heuristiken im Detail erklärt und auch eine experimentelle Evaluierung bezogen auf die Laufzeit enthält.

Kapitel 4

Graphverarbeitung und -analysen – Stand der Technik

Da eines der Ziele dieser Arbeit die Parallelisierung einer *Graph-Mining*-Technik ist, wird in diesem Kapitel ein Überblick über einige Techniken der Big-Data-Landschaft gegeben, die für die Parallelisierung geeignet sind und welche Möglichkeiten bestehen, um Graphen mit diesen Techniken analysieren zu können. Zunächst werden die beiden Frameworks *Apache Spark* (Unterkap. 4.1) und *Apache Flink* (Unterkap. 4.2) näher erläutert. Anschließend folgen zwei Unterkapitel, welche eine Bibliothek (Kap. 4.3) und ein Framework (Unterkap. 4.4) zur Analyse von Graphen näher erläutern. Abschließend wird dann das *Hydra.PowerGraph*-System (Unterkap. 4.5) beschrieben.

4.1 Apache Spark

Apache Spark ist ein Framework für das Cluster-Computing. Es ist 2009 im Laufe eines Forschungsprojekts vom *AMPLab* der *University of California* entstanden und steht seit 2010 unter einer Open-Source-Lizenz. Ab 2013 wurde das Projekt von der *Apache Software Foundation* weitergeführt und 2014 als Top-Level-Projekt eingestuft. Im folgenden Abschnitt wird zunächst erläutert, wie *Apache Spark* aufgebaut ist und welche Funktionalitäten bzw. welche Bibliotheken das Framework mitbringt. Anschließend wird die *GraphX*-Bibliothek genauer erklärt, da diese von besonderem Interesse für diese Arbeit ist.

4.1.1 Aufbau und Bibliotheken

Spark setzt sich aus mehreren und teils von einander unabhängigen Komponenten zusammen. Die Hauptbestandteile des Frameworks bilden *Spark Core* und die darauf aufbauenden Bibliotheken ([ASF17a]). Wie in Abb. 4.1 zu sehen, kann Spark aber auch auf verschiedene Da-

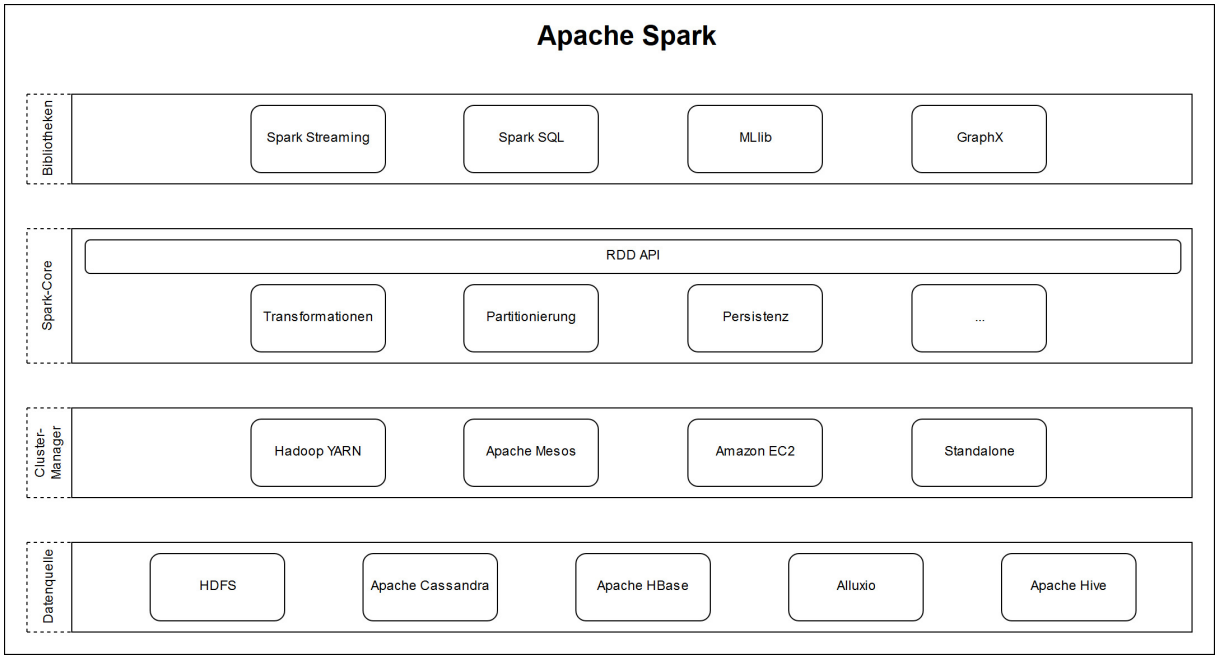


Abbildung 4.1: Der Aufbau des Spark-Frameworks.

tenquellen aufsetzen und auch unterschiedliche Cluster-Manager nutzen, um die Berechnungen zu koordinieren. Spark ermöglicht dem Programmierer, sich auf die Logik seines Programmes zu konzentrieren, ohne sich dabei noch um die Parallelisierung seiner Algorithmen kümmern zu müssen. Zusätzlich kann das Framework an unterschiedliche Speichersysteme angebunden werden. Im Folgenden wird auf die einzelnen Bestandteile von Spark näher eingegangen.

Spark Core

Spark Core stellt das Fundament des Frameworks dar. Mit der RDD API wird ein einfaches Interface für die Programmierung auf großen Datensätzen bereitgestellt.
RDD bedeutet *Resilient Distributed Dataset* und ist die Grundlage für *Spark Core*. Dabei handelt es sich um eine fehlertolerante, parallele Datenstruktur, die auf Festplatte oder im Arbeitsspeicher gespeichert werden kann (vgl. [ZCD+12]). Ein RDD kann aus einer externen Datenquelle oder als Ergebnis von Transformationen (z.B. *map*, *filter*, *join*) auf schon bestehenden RDDs erzeugt werden. Bei dieser Art der Speicherung wird versucht, eine Datenreplikation zu vermeiden, indem die verwendeten Operationen, welche zum resultierenden RDD geführt haben, in einem Log gespeichert werden. Dadurch können bei einem Fehler oder Datenverlust verlorengegangene Partitionen eines RDDs mithilfe anderer RDDs rekonstruiert werden, ohne weiteren Speicher für Replikationen verwenden zu müssen (Fehlertoleranz). Zusätzlich kann der Benutzer zwei weitere Aspekte der RDDs beinflussen: *Persistenz* und *Partitionierung*. So kann man festlegen welche

RDDs man später erneut verwenden will und die Art der Speicherung bestimmen. Ebenso lässt sich die Art der Partitionierung eines RDDs auf Grundlage eines Schlüssels beeinflussen. Spark stellt die RDDs über eine API zur Verfügung, bei der jeder Datensatz als ein Objekt repräsentiert wird und Transformationen über Methoden auf diesen Objekten ausgeführt werden.
Spark Core wurde in Scala geschrieben, bietet allerdings auch APIs für Java, Python und R an.

Bibliotheken
Spark bringt viele Bibliotheken für unterschiedliche Szenarien mit. Dadurch, dass alle Bibliotheken auf *Spark Core* implementiert sind, führen Verbesserungen von *Spark Core* auch automatisch zu Verbesserungen in den unterschiedlichen Bibliotheken. Folgende sind in Spark enthalten:

- **MLlib:** Hierbei handelt es sich um eine Bibiothek für das maschinelle Lernen, welche das Design und die Implementierung von Algorithmen aus diesem Bereich vereinfacht. So werden verschiedene Packages für Algebraoperatoren, statistische Verfahren, Modell-Training und Modell-Evaluation bereitgestellt. Im Kern besteht MLlib aus zwei Paketen: *spark.ml* und *spark.mllib*. Während *spark.mllib* auf RDDs arbeitet, benutzt *spark.ml* Data-Frames als Datenstruktur. Dadurch kann das letztgenannte Paket die Pipeline API bereitstellen, die für das Erzeugen, Debuggen und Verbessern von *Machine-Learning-Pipelines* benötigt wird. *spark.mllib* hingegen enthält Pakete für lineare Algebra, statistische Verfahren und andere Basisoperationen, die für das maschinelle Lernen benötigt werden (vgl. [ASF17c]).

- **Spark Streaming:** Diese Bibliothek ermöglicht es, mit Spark Analysen auf Stromdaten durchführen zu können. Dazu werden die Stromdaten in sogenannte Micro-Batches zerlegt, welche wiederum aus einzelnen RDDs bestehen und somit eine integrierte Fehlertoleranz mitbringen. Dadurch kann man Stromdaten-Analysen mithilfe einer Sequenz von RDD-Transformationen umsetzen. Dazu muss beim Streaming-Kontext, dem Eintrittspunkt jeder Stromdaten-Analyse, nur das Batch-Intervall festgelegt werden. Zudem kann man die Stromdatenverarbeitung mit der Batch-Verarbeitung kombinieren (vgl. [ASF17e]).

- **Spark SQL:** Diese Bibliothek führt mit den *DataFrames* eine weitere Datenabstraktion in Spark ein, welche eine Unterstützung von strukturierten, wie auch semi-strukturierten Daten mit sich bringt. Konzeptionell sind *DataFrames* nichts anderes als Tabellen in einer relationalen Datenbank. Die Bibliothek funktioniert zusätzlich als Engine für verteilte SQL-Abfragen. Ein *DataFrame* kann z.B. aus strukturierten Dateien, Tabellen in Hive oder RDDs erzeugt werden (vgl. [ASF17d]).

- **GraphX:** Das ist die Bibliothek für die Verarbeitung von Graphen. Mit ihr lassen sich parallele Graph-Analysen implementieren. Mit einer Erweiterung der RDDs, lässt sich ein Graph als *Resilient Distributed Graph* (kurz RDG) repräsentieren. Zusätzlich bringt die

Bibliothek verschiedene Operatoren für Transformation von Graphen mit, wie auch eine Menge von Algorithmen für das *Graph-Mining* (vgl. [ASF17b]).

Cluster-Manager und Datenquellen

Ein Cluster-Manager ist dazu da, um die Ressourcen eines Clusters bei der Ausführung eines Jobs zu verwalten. So kann Spark auf verschiedenen Cluster-Managern laufen (vgl. [ASF17a]):

- Hadoop YARN
- Apache Mesos
- Amazon EC2
- Standalone (eigener Cluster-Manager von Spark)

Ebenso ist es möglich, aus einem Spark-Programm auf verschiedene Datenquellen zuzugreifen bzw. als Datensenke zu definieren:

- HDFS
- Apache Cassandra
- Apache HBase
- Apache Hive
- Alluxio

Dadurch ist Spark mit vielen anderen Frameworks aus der Big-Data-Landschaft kombinierbar.

4.1.2 GraphX

Wie vorher schon erwähnt wurde, führt *GraphX* mit dem *Resilient Distributed Graph* (kurz RDG) eine weitere Datenstruktur in Spark ein. RDGs sind im Grunde genommen nur eine Erweiterung der RDDs. Die Kerndatenstruktur beruht auf einem sogenannten *Property-Graph* (kurz PG). Dabei handelt es sich um einen gerichteten Multigraphen (zwischen zwei Knoten können mehrere Kanten verlaufen) an dessen Kanten und Knoten weitere Datensätze hängen. So hat also jede Kante und jeder Knoten Attribute bzw. Eigenschaften, welche die Datensätze repräsentieren. Wie auch RDDs, sind die RDGs unveränderlich, verteilt und fehlertolerant. So wird bei jeder Transformation und Operation auf einem Graphen ein neuer Graph erstellt. Es gibt 5 Datentypen, um mit *Property-Graphen* zu arbeiten (vgl. [ASF17b]):

- *Graph*: hierbei handelt es sich um eine Abstraktion vom PG, welche konzeptionell äquivalent zu einem Paar von typisierten RDDs ist. So gibt es ein RDD für die Knoten und ein RDD für die Kanten.
- *VertexRDD*: das ist eine verteilte Menge von Knoten eines PG. Jede Kante wird durch ein Key-Value-Paar charakterisiert.
- *Edge*: die Abstraktion einer gerichteten Kante eines PG. Eine Kante setzt sich zusammen aus der ID des Quellknotens, der ID des Zielknotens und Attributen der Kante.
- *EdgeRDD*: eine verteilte Menge von Kanten.
- *EdgeTriplet*: eine Kombination aus einer Kante und den dazugehörigen Knoten.

Zusammengefasst kann man sagen, dass ein Graph in Spark als ein Paar, zusammengesetzt aus den RDDs für die Kanten und Knoten, repräsentiert wird. Diese RDDs können ebenfalls mit den Transformationen und Operationen der RDD API analysiert werden. Dadurch ist es möglich, dass der gleiche Datensatz als Paar von RDDs oder als Graph analysiert werden kann.

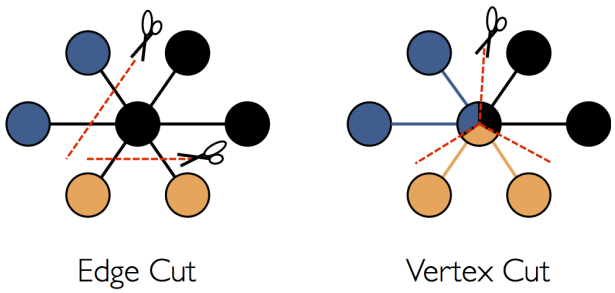


Abbildung 4.2: Die zwei Partitionierungsansätze *Edge-Cut* und *Vertex-Cut* (Quelle: [XGFS13]).

Für die Partitionierung verwendet *GraphX* den Ansatz des *Vertex-Cut*. Im Gegensatz zum *Edge-Cut* werden hier kürzere Laufzeiten erzielt (vgl. [XGFS13]). Dabei werden sogenannte *Gelenkknoten* identifiziert. Wenn man diese aus dem zu partitionierenden Graphen entfernen würde, wäre der Graph anschließend nicht mehr zusammenhängend. Ein Beispiel für einen *Edge-Cut* und einen *Vertex-Cut* ist in Abb. 4.2 zu sehen.

Operatoren

Um große Analysen ermöglichen zu können, stellt *GraphX* eine Menge von speziellen Operatoren für *Property-Graphen* zur Verfügung. Da RDGs unveränderlich sind, werden durch diese Operatoren neue Graphen erzeugt, die entweder eine andere Struktur oder veränderte Eigenschaften der Knoten/Kanten aufweisen. Die Operatoren lassen sich in unterschiedliche Gruppen unterteilen (vgl. [SDC+16]):

- *Eigenschaftsoperatoren*: Es gibt drei Operatoren, die die Eigenschaften von Kanten oder Knoten verändern können: *mapVertices, mapEdges* und *mapTriplets*. Die Struktur des Graphen wird durch diese Operatoren nicht verändert und somit können die Strukturindizes des originalen Graphen wiederverwendet werden, um den neuen Graphen zu berechnen.
- *Strukturoperatoren*: Dabei handelt es sich um Operatoren wie *reverse, subgraph, mask* und *groupEdges*. Die Eigenschaften bzw. angehängten Daten der Knoten und Kanten bleiben von diesen Operatoren unberührt.
- *Join-Operatoren:* Diese Operatoren (*joinVertices, outerJoinVertices*) können dazu genutzt werden, um Daten aus einem anderen RDD mit einem Graphen zu verbinden und somit existierende Eigenschaften zu aktualisieren oder neu hinzuzufügen.
- *Aggregationsoperatoren*: Mit diesen Operatoren kann man Daten auf Grundlage der Knotennachbarschaft aggregieren. Diese Operation ist für viele Graphalgorithmen wie z.B. den *PageRank* essenziell.

Iterative Graphverarbeitung

Um auch iterative Algorithmen zur Analyse von Graphen zu ermöglichen, bietet *GraphX* einen *Pregel-Operator* an. *Pregel* ist ein Programmiermodell von Google, welches nach dem „think like a vertex"-Prinzip arbeitet. Dabei wird auf ein Iterationsmodell, genannt *Gather-Apply-Scatter* (kurz GAS) gesetzt. Dieses Modell wurde von *Powergraph* [GLG+12] eingeführt. Dabei werden von Knoten Nachrichten entlang der Kanten zu anderen Knoten versendet. Der Nutzer muss dann drei Funktionen zur Verfügung stellen: *Gather, Apply, Scatter*. Die *GatherFunction* aggregiert alle eingehenden Nachrichten eines Knotens. Mit der *ApplyFunction* wird dann der Knotenwert des jeweiligen Knotens auf Grundlage der empfangenen Nachrichten aktualisiert. Abschließend wird mit der *ScatterFunction* der aktualisierte Knotenwert an die anderen Knoten versendet.

Dieses Verfahren wird mithilfe sogenannter *Supersteps* parallelisiert. In einem *Superstep* werden alle drei Funktionen des GAS-Modells ausgeführt. Es werden also zunächst die Nachrichten aus dem letzten *Superstep* empfangen, danach die Knotenwerte aktualisiert und anschließend werden die Nachrichten aller Knoten gleichzeitig an den nächsten *Superstep* versendet. Dadurch wird sichergestellt, dass jeder Knoten nur die Nachrichten aus dem vorherigen *Superstep* empfangen kann.

Graphalgorithmen
GraphX bringt schon eine Vielzahl an verschiedenen Graphalgorithmen mit (vgl. [ASF17b]):

- *PageRank*: Errechnet die Wichtigkeit von Knoten in Abhängigkeit der eingehenden Kanten.
- *Connected-Components* und *Strongly Connected Components*: Findet verbundene Komponenten für jeden Knoten in einem Graphen.
- *Triangle-Counting*: Analysiert für jeden Knoten, ob die Nachbarn des Knotens über eine Kante verbunden sind, um die Anzahl der Dreiecke im jeweiligen Graphen ermitteln zu können.
- *Label-Propagation*: kann benutzt werden, um *Communities* in einem Graphen zu entdecken.
- *Shortest-Path*: Findet den kürzesten Pfad von jedem Knoten zu einer gegebenen Menge von Knoten in einem Graph.

Dadurch kann der Nutzer schon auf einige Algorithmen zurückgreifen, ohne diese selber mithilfe der verschiedenen RDG-Operatoren realisieren zu müssen.

4.2 Apache Flink

Apache Flink ist ein Stream-Processing-Framework, welches von der *Apache Software Foundation* entwickelt wurde. Seinen Ursprung hat Flink im deutschen Forschungsprojekt *Stratosphere* ([ABE+14]), welches von der *Deutschen Forschungsgesellschaft* (kurz DFG) als Gemeinschaftsprojekt der *Technischen Universität Berlin*, der *Humboldt-Universität zu Berlin* und dem *Hasso-Plattner-Institut* ins Leben gerufen wurde. Flink ist dann schließlich aus der verteilten Ausführungsengine von *Stratosphere* entstanden und wurde 2014 als Inkubator-Projekt von der *Apache Software Foundation* übernommen. Im Dezember 2014 wurde es schließlich als Top-Level-Projekt von Apache eingestuft.
In dem folgenden Unterkapitel wird die Systemarchitektur von Flink näher erläutert.

4.2.1 Systemarchitektur

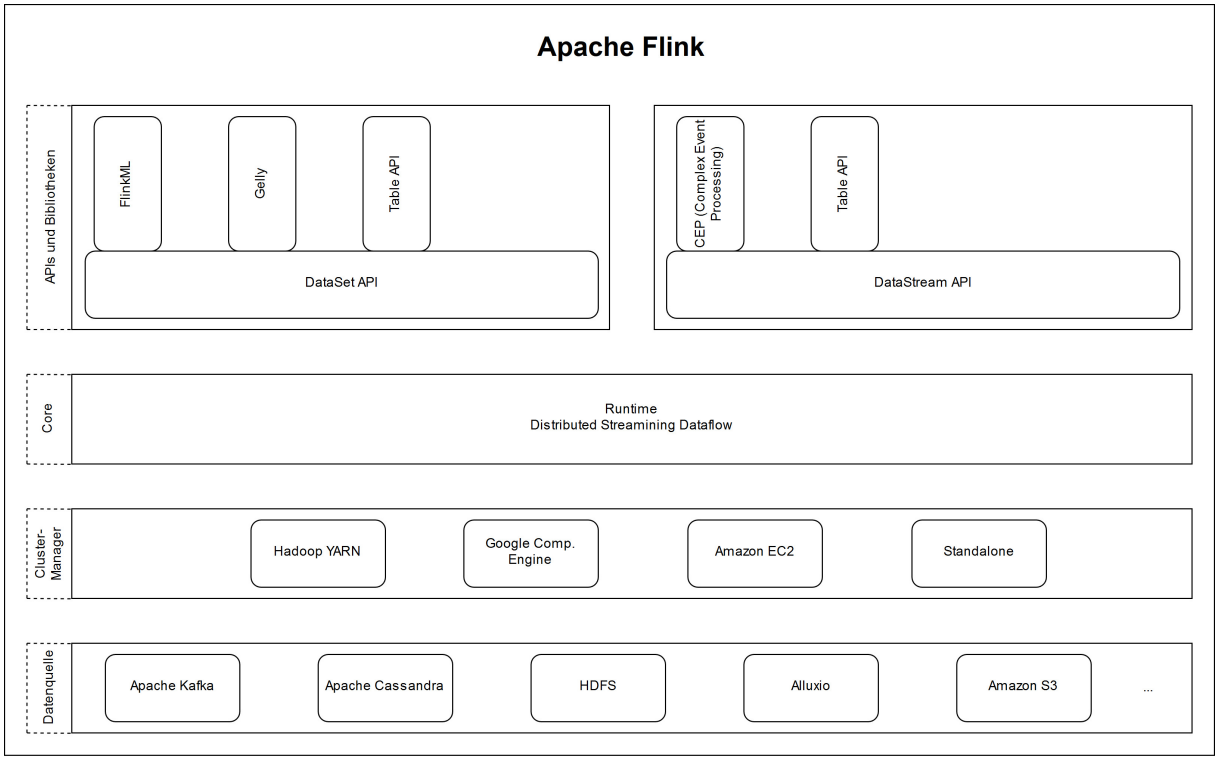


Abbildung 4.3: Der Aufbau des Flink-Frameworks.

Wie in Abb. 4.3 zu sehen, besteht Flink aus mehreren Teilen. Die 4 Hauptbestandteile dabei sind der Cluster-Manager, der Core, die APIs und die Bibliotheken (vgl. [ASF18j]). Den Kern bildet die verteilte *Dataflow*-Engine, welche die Flink-Programme ausführt. Ein Flink-Programm wird dabei als azyklischer gerichteter Graph, dessen Kanten Datenströme und dessen Knoten

Operatoren auf diesen Datenströmen sind, repräsentiert. Die zwei Kern-APIs sind *DataSet* und *DataStream*. Diese beiden APIs erzeugen dann Programme, die durch die verteilte *Dataflow*-Engine ausgeführt werden können. Die weiteren APIs und Bibliotheken bauen dann auf dieser Engine und den beiden Kern-APIs auf. In den folgenden Abschnitten wird jeder Bestandteil von Flink einzeln erläutert.

Bibliotheken und APIs

Wie auch Spark, bringt Flink ebenfalls Bibliotheken für verschiedene Analyseszenarien mit, welche im Folgenden kurz einzeln erläutert werden:

- **Table API & SQL**: Die *Table API* ist eine Anfrage-API für Scala und Java, welche es erlaubt Operatoren der relationalen Algebra auf normalen Datensätzen oder Stromdaten anzuwenden. Dazu werden die Datensätze aus der DataSet/DataStream API in eine interne relationale Abstraktion, der *Table*, überführt. Es ist ebenso möglich, direkt über SQL Anfragen an die *Tables* zu stellen. Flink's SQL-Unterstützung basiert auf *Apache Calcite*, welche den SQL-Standard implementiert (vgl. [ASF18k]).
- **FlinkML**: Diese Bibliothek bietet verschiedene fertige Algorithmen aus dem Bereich *Machine-Learning* an. Beispiele sind *Multiple lineare Regression, Support-Vector-Machines using CoCoA* und *k-Nearest-Neighbors-Join* (vgl. [ASF18h]).
- **Flink CEP**: Hierbei handelt es sich um eine Bibliothek für das *Copmplex-Event-Processing*. Dadurch hat man die Möglichkeit, Muster für komplexe Events in einem kontinuierlichen Datenstrom zu erkennen. Über die mitgelieferte *Pattern API* lassen sich eigene Muster definieren, nach denen man in den jeweiligen Datenströmen suchen möchte (vgl. [ASF18a]).
- **Gelly**: Dies ist die Graph API für Flink. Sie bringt eine Menge von Graph-Operatoren mit, welche die Entwicklung von Graphalgorithmen erleichtern soll. Zusätzlich enthält die API auch schon einige vorgefertigte Algorithmen, die für die Analyse von Graphen verwendet werden können ([ASF18i]).

Clustermanager und Datenquellen

Flink ist mit unterschiedlichen Cluster-Managern kombinierbar. So könne folgende verwendet werden:

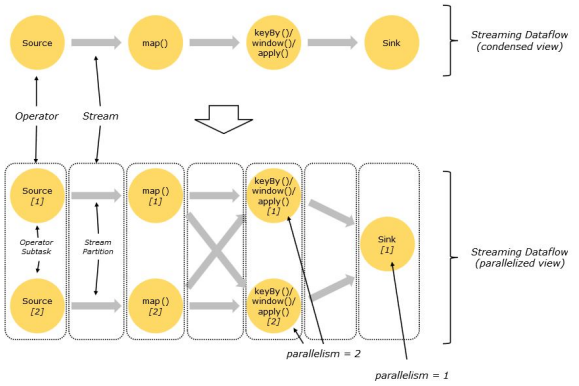
- Hadoop YARN
- Google Comp. Engine
- Amazon EC2
- Standalone (eigener Cluster-Manager von Flink)

Ebenso ist es möglich, Flink an verschiedene Datenquellen bzw. Datensenken anzubinden. Einige Beispiele (vgl. [ASF18g]):

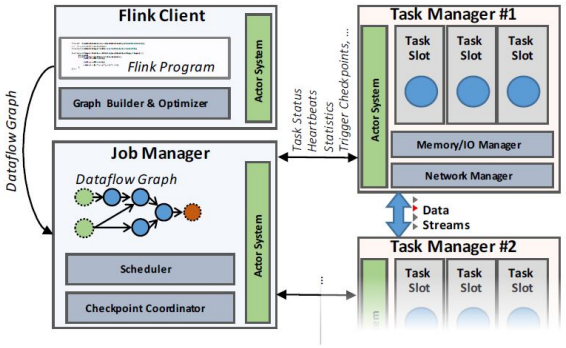
- Apache Kafka
- Apache Cassandra
- HDFS
- Alluxio
- Amazon S3

4.2.2 Flink-Programme

Wie schon erwähnt, werden Programme in Flink als *gerichtete azyklische Graphen* (kurz DAG) dargestellt. Die Knoten repräsentieren dann Operatoren und die Kanten die entsprechenden Datenflüsse, die von den Operatoren produziert wurden. Da ein Flink-Programm parallel und verteilt ablaufen soll, werden die Operatoren in eine oder mehrere Instanzen und die Datenflüsse in mehrere Partitionen aufgeteilt. Die Operatoren enthalten dann die komplette Programmlogik. Anhand des Graphen kann das Programm dann verteilt und parallelisiert werden. Dazu wird es durch einen *Client* in ein DAG übersetzt, welcher dann an den *JobManager* übermittelt wird. Dieser koordiniert dann die verteilte und parallele Berechnung des Programms. Dazu wird der Status und Fortschritt jedes Operators und Datenstroms protokolliert. *Checkpoints*, wie auch *Recovery*, werden ebenfalls vom *JobManager* koordiniert. Die Datenverarbeitung findet dann in den einzelnen *TaskManagern* statt. Dort werden dann eine oder mehrere Operationen ausgeführt, die als Ergebnis jeweils einen neuen Datenstrom weitergeben und ihren Status an den *JobManager* übermitteln. Die Fehlertoleranz von Flink-Programmen wird über die eben schon erwähnten



(a) Der DAG eines Flink-Programms (Quelle: [ASF18c]).



(b) Die Ausführung eines Flink-Programms (Quelle: [CKE+15]).

Checkpoints und partielle Neuausführung von Operationen sichergestellt. So erstellt Flink in bestimmten Abständen sogenannte *Snapshots* des aktuellen Zustands eines Operators, inklusive der aktuellen Position des Eingabestroms. Das Problem dabei liegt darin, dass von allen parallelen Operatoren konsistente *Snapshots* erstellt werden müssen. Dafür nutzt Flink einen Mechanismus, der sich *Asynchronous Barrier Snapshoting* (kurz ABS) nennt. Auf diesen Mechanismus wird hier allerdings nicht weiter eingegangen. Den interessierten Leser verweise ich deswegen an dieser Stelle auf die folgenden Quellen:

- [CKE+15]
- [CFE+]
- [ASF18b]

4.2.3 DataStream und DataSet

Mit der DataStream API hat man die Möglichkeit, Stromdaten zu verarbeiten. Dazu bringt die API verschiedene Operatoren wie z.B. *Map, Filter, KeyBy, Aggregation* oder auch *Window* mit (vgl. [ASF18f]). Bei den *Window*-Funktionen handelt es sich um besondere Operatoren, denn sie können den Datenstrom in endliche Mengen zerteilen. Die Größe der dabei entstehenden *Fenster* (engl. *Windows*) können dabei von der Zeit oder der Menge der Daten abhängig sein. Dazu werden durch die DataStream API verschiedene Mechanismen, wie *Tumbling Windows, Sliding Windows, Session Windows* oder auch *Global Windows* bereitgestellt. Anschließend können über die Fenster wieder andere Operationen berechnet werden. So ist es möglich, z.B. ein *Join* über verschiedene Fenster zu berechnen oder die Elemente eines Fensters zu *aggregieren.*
Eine weitere Möglichkeit, welche die DataStream API bietet, ist der *Side-Output.* Dieser ermöglicht es, neben den herkömmlichen Datenströmen aus Operatoren, extra Datenströme zu produzieren. Der *Side-Output* kann sich in seiner Struktur komplett zu den herkömmlichen Datenströmen unterscheiden. Das kann z.B. nützlich sein, wenn man einen Datenstrom zerteilen möchte. Denn normalerweise müsste man den Datenstrom replizieren und dann aus jedem Datenstrom die Daten herausfiltern, die man für die weitere Verarbeitung benötigt.
Mit der DataSet API ist es möglich endliche Datensätze zu verarbeiten. Das *DataSet* wird dabei intern durch einen endlichen Datenstrom repräsentiert. Auch die DataSet API bringt verschiedene Operatoren wie *Map, Filter, Join* oder *Aggregate* mit (vgl. [ASF18e]). Wie auch bei Spark, handelt es sich bei *DataSets* und *DataSteams* um fehlertolerante, parallele Datenstrukturen.

4.2.4 Das Iterationskonzept in Flink

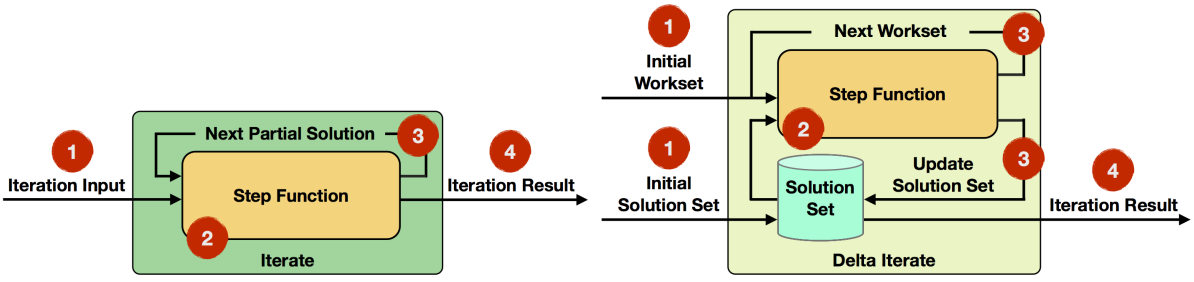
Einen gerichteten azyklischen Graph zur Ausführung von Flink-Programmen zu wählen hat auch seine Nachteile. So benötigen viele Algorithmen aus dem Bereich *Machine-Learning* und *Graph-Mining* Rekursion oder Iteration und somit einen zyklischen Ausführungsgraphen (vgl. [JPNR17]). Dazu bietet Flink zwei spezielle Operatoren an: *Bulk-Iteration* und *Delta-Iteration*.

Bulk-Iteration

Bei der *Bulk-Iteration* handelt es sich um eine simple Form der Iteration. Sie kann in vier Phasen unterteilt werden:

1. **Iterationseingabe**: das ist die initiale Eingabe für die erste Iteration, die entweder aus einer Datenquelle oder aus vorherigen Operatoren generiert werden kann.
2. **Iterationsfunktion**: diese Funktion wird in jeder Iteration ausgeführt. Sie kann Operatoren wie z.B. *map, reduce* oder auch *join* enthalten und hängt davon ab, was mit der Iteration bezweckt werden soll.
3. **Zwischenergebnismenge**: nach jeder Iteration wird das Ergebnis der *Iterationsfunktion* an den nächsten Iterationsschritt übergeben.
4. **Iterationsergebnis**: das ist das Ergebnis der letzten Iteration, welches dann in eine *Datensenke* (engl. *data sink*) geschrieben oder an den nächsten Operator übergeben wird.

Um die Abbruchbedingung einer Iteration zu definieren, kann man entweder die maximale Anzahl an Iterationen angeben oder man definiert ein bestimmtes Abbruchkriterium (vgl. [ASF18d]).



(c) Bulk-Iteration. (d) Delta-Iteration.

Abbildung 4.4: *Bulk-* und *Delta-Iteration* im Vergleich (Quelle: [ASF18d]).

Delta-Iteration

Bei der *Delta-Iteration* handelt es sich um die inkrementelle Iteration. Hier werden mit jedem Iterationsschritt selektiv nur einige Elemente der Lösung modifiziert und somit entwickelt sich im Gegensatz zur *Bulk-Iteration* die Ergebnismenge mit jeder Iteration weiter, anstatt jedes Mal komplett neu berechnet zu werden. Kann man diese Form der Iteration einsetzen, führt dies zu effizienteren Algorithmen, da nicht jedes Element der Ergebnismenge mit jeder Iteration verändert wird (vgl. [JPNR17]). Auch die *Delta-Iteration* kann wieder in vier Phasen unterteilt werden:

1. **Iterationseingabe**: das ist die initiale Eingabe für die erste Iteration, die entweder aus einer Datenquelle oder aus vorherigen Operatoren generiert werden kann.

2. **Iterationsfunktion**: diese Funktion wird in jeder Iteration ausgeführt. Sie kann Operatoren wie z.B. *map, reduce* oder auch *join* enthalten und hängt davon ab, was mit der Iteration bezweckt werden soll.

3. **Zwischenergebnismenge/aktualisiertes Iterationsergebnis**: die *Zwischenergebnismenge* wird an den nächsten Iterationsschritt übergeben und das finale Iterationsergebnis wird aktualisiert. Beide Mengen können durch unterschiedliche Operatoren aktualisiert werden.

4. **Iterationsergebnis**: das ist das Ergebnis der letzten Iteration, welches dann in eine *Datensenke* (engl. *data sink*) geschrieben oder an den nächsten Operator übergeben wird.

Die Abbruchkriterien sind dieselben wie bei der *Bulk-Iteration*, bis auf das zusätzliche Kriterium, dass die *Delta-Iteration* auch abbrechen kann, wenn die Zwischenergebnismenge leer ist (vgl. [ASF18d]).

Die bisherigen Erklärungen zu den Iterationsoperatoren von Flink haben sich nur auf „normale“ Iteration bezogen. Will man allerdings Iterationen in parallelen Berechnungen durchführen, müssen mehrere Instanzen der Iterationsfunktion parallel auf verschiedenen Partitionen eines Iterationsschrittes berechnet werden. Alle Berechnungen formen dann zusammen einen sogenannten *Superstep*, welcher gleichzeitig die Granularität der Synchronisation repräsentiert. Ein *Superstep* kann also nur abgeschlossen werden, wenn alle parallelen Berechnungen eines Iterationsschrittes terminiert sind. Danach kann dann der nächste *Superstep* berechnet werden.

4.3 Gelly

Mit *Gelly* ist es möglich, den kompletten Lebenszyklus einer Graph-Analyse abzubilden (Vorverarbeitung, Graph-Erzeugung, Graph-Analyse und Nachbereitung). In diesem Unterkapitel wird die Graph-Bibliothek nochmal gesondert vorgestellt und auf einzelne Details näher eingegangen. So bringt *Gelly* z.B. einen eigenen Datentypen für Graphen mit, auf dem dann mit speziellen Operatoren gearbeitet werden kann. Dabei baut die komplette Bibliothek auf der *DataSet API* von Flink auf. Zudem hat Gelly ein eigenes Konzept für die iterative Graphverarbeitung. Anschließend wird noch auf einzelne Graphalgorithmen eingegangen, die von der Bibliothek schon mitgebracht werden.

4.3.1 Graph API

In *Gelly* wird ein Graph durch ein *DataSet* mit Knoten und ein *DataSet* mit Kanten repräsentiert. Die Knoten des Graphen werden durch den sogenannten *Vertex*-Typ dargestellt. Ein *Vertex* ist durch eine einzigartige ID und eine Menge von Werten (die Attribute) definiert. Knoten ohne Werte können durch den *NullValue*-Wert dargestellt werden.
Die Kanten eines Graphen werden durch den *Edge*-Typ repräsentiert. Eine *Edge* definiert sich dann durch eine Quell-ID (das ist die ID des Quellknotens), eine Ziel-ID (ID des Zielknotens) und eine optionale Menge an Werten (Attribute einer Kante). In *Gelly* sind Kanten immer vom Quellknoten zum Zielknoten gerichtet. Ungerichtete Graphen können dargestellt werden, indem für jede Kante eine passende Kante existiert, die vom Zielknoten zum Quellknoten gerichtet ist (vgl. [ASF18i]).
Der Graph kann dann auf verschiedene Weisen entstehen. Entweder man erstellt ihn nur aus dem *DataSet* der Kanten oder er entsteht aus dem *DataSet* für Kanten und Knoten. Somit ist es in Gelly möglich, auch ohne einen Datensatz für alle Knoten, einen Graphen zu erstellen. Allerdings können bei dieser Variante die Knoten keine zusätzlichen Attribute enthalten, denn diese können nur im *DataSet* für die Knoten (*Vertex-Typ*) definiert werden.
Zu der Art und Weise wie *Gelly* einen Graphen partitioniert, ist, im Gegensatz zu *GraphX*, nicht viel in der Literatur zu finden. Laut [Kae17] verwendet *Gelly* eine random/hashing-Kombination, welche auf den IDs der Knoten basiert. Die Graph API bringt auch noch eigene Operatoren mit, die auf den jeweiligen Datentypen ausgeführt werden können und somit Analysen auf Graphen unterstützen. Dabei werden die Operatoren in fünf Klassen unterteilt (vgl. [ASF18i]):

- **Graph-Eigenschaften**: das sind Operationen, die sich auf Kanten und Knoten des Graphen beziehen. So kann man sich z.B. mit *getVertices* alle Knoten ausgeben lassen oder mit *getEdgeIds* eine Liste mit allen IDs der Kanten.

- **Graph-Transformationen**: hierbei handelt es sich um Operatoren, die die Struktur eines Graphen oder Eigenschaften der Kanten und Knoten verändern können. Beispiele für Operatoren dieser Klasse sind: *Map, Filter, Join*, oder auch *Union*.
- **Graph-Mutationen**: das sind alle Operatoren, mit denen man einzelne oder eine Menge von Kanten und Knoten aus einem Graphen entfernen kann.
- **Nachbarschafts-Methoden**: Diese Methoden erlauben es, dass man Aggregationen über die direkten Nachbarn eines Knotens berechnen kann. Mit *reduceOnEdges* ist es z.B. möglich, die Werte der benachbarten Kanten eines einzelnen Knotens zu aggregieren. Das Gleiche kann mit *reduceOnNeighbors* auch für die benachbarten Knoten berechnet werden.
- **Graph-Validierung**: Mit diesem Operator kann man einen Graphen auf bestimmte Eigenschaften überprüfen. So kann es z.B. interessant sein, vor einer Analyse zu wissen, ob ein Graph doppelte Kanten enthält. Um einen Graphen zu validieren, muss der Benutzer einen *GraphValidator* definieren und dessen *validate*-Methode mit der eigenen Logik überschreiben.

4.3.2 Iterative Graphverarbeitung

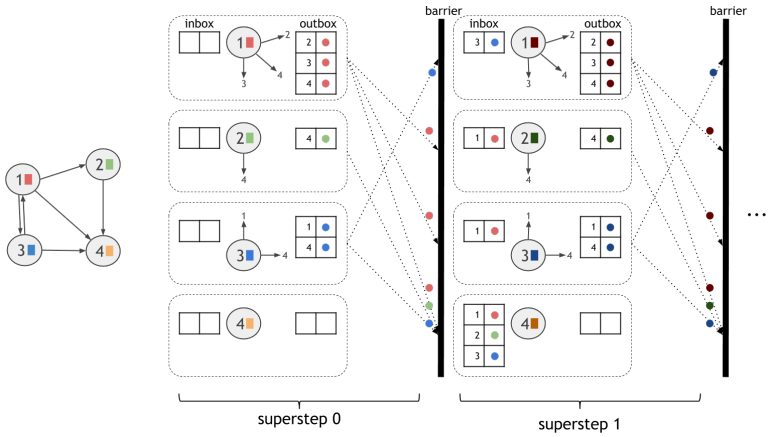


Abbildung 4.5: Das *Vertex-Centric*-Modell in zwei *Supersteps* (Quelle: [ASF18i]).

Viele Graphalgorithmen benötigen das Konzept der Iteration. Um auch solche Algorithmen implementieren zu können, bringt *Gelly* drei Modelle mit: *Vertex-Centric*, *Scatter-Gather* und *Gather-Sum-Apply* (vgl. [ASF18i]).

Vertex-Centric-Modell

Beim *Vertex-Centric*-Modell werden die Berechnungen aus der Perspektive der Knoten durchgeführt (deswegen wird dieses Modell auch oft „think like a vertex“ genannt). Die Berechnungen

laufen dann in synchronisierten Iterationsschritten (*Supersteps*) ab. In jedem *Superstep* führt jeder Knoten eine benutzerdefinierte Funktion (engl. *user-defined-funtion*, kurz UDF) aus. Knoten kommunizieren mit anderen Knoten über Nachrichten entlang der Kanten. Ein Knoten kann eine Nachricht zu jedem anderen Knoten im Graph senden, solange er die ID des jeweiligen Knotens kennt. In Abb. 4.5 sieht man ein Beispiel für das *Vertex-Centric*-Modell (vgl. [ASF18i]).

Scatter-Gather-Modell

Das *Scatter-Gather*-Modell (auch „Signal/Collect"-Modell genannt) arbeitet ebenfalls nach der „think like a vertex"-Philosophie. So werden alle Berechnungen, wie auch beim *Vertex-Centric*-Modell, in *Supersteps* und aus Sicht der Knoten ausgeführt. Die Knoten kommunizieren ebenfalls entlang der Kanten mithilfe von Nachrichten miteinander. Basierend auf diesen Nachrichten, werden die Werte der jeweiligen Knoten dann aktualisiert. Ein Hauptunterschied zwischen beiden Modellen besteht aber darin, dass das *Scatter-Gather*-Modell die Berechnung in jeweils zwei Phasen, *Scatter* und *Gather*, unterteilt:

- *ScatterFunction*: produziert die Nachrichten, die ein Knoten an andere Knoten sendet.
- *GatherFunction*: aktualisiert die Werte eines Knotens, basierend auf den eingehenden Nachrichten.

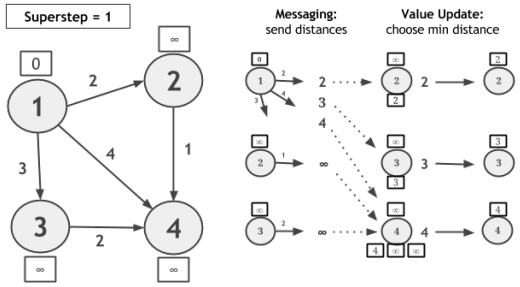


Abbildung 4.6: Das *Scatter-Gather*-Modell (Quelle: [ASF18i]).

Beide Phasen werden in einem *Superstep* ausgeführt. In Abb. 4.6 sieht man eine beispielhafte Ausführung dieses Modells anhand des *Single-Source-Shortest-Path*-Algorithmus. Zusätzlich können sich die Knoten in zwei verschiedenen Zuständen befinden: *aktiv* oder *inaktiv*. Ist ein Knoten im Zustand *inaktiv*, weil sich sein Wert in der *Gather*-Phase nicht aktualisiert hat, wird die *Scatter*-Funktion nicht ausgeführt, da seine Nachbarknoten den aktuellen Wert des Knotens kennen. Um die *Scatter-Gather*-Iteration in Flink zu implementieren, muss jeweils die *ScatterFunction* und *GatherFuntion* mit einer eigenen Logik überschrieben werden (vgl. [ASF18i]).

Gather-Sum-Apply-Modell

Gather-Sum-Apply (kurz GSA) ist ein drittes Modell für die Iteration über Graphen, welches von *Gelly* bereitgestellt wird. Es erweitert das *Scatter-Gather*-Modell zusätzlich, indem es die Berechnungen nicht in zwei, sondern in drei Phasen unterteilt (*Gather, Sum, Apply*):

- *GatherFunction*: berechnet einen Teilwert pro adjazenten Konten und inzidenter Kante eines Knotens.
- *SumFunction*: aggregiert alle Teilwerte der *GatherFunction* zu einem Wert.
- *ApplyFunction*: aktualisiert den Wert eines Knotens auf Grundlage des aggregierten Wertes der *SumFunction*.

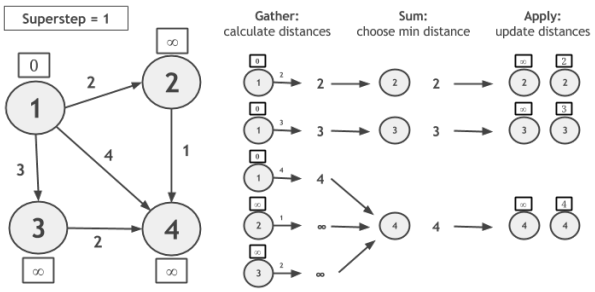


Abbildung 4.7: Das *Gather-Sum-Apply*-Modell (Quelle: [ASF18i]).

Alle drei Phasen (*Gather, Sum, Apply*) werden ebenfalls in einem *Superstep* ausgeführt. In Abb. 4.7 sieht man ein Beispiel dieser Iterationsform anhand des *Single-Source-Shortest-Path*-Algorithmus. Um die *Gather-Sum-Apply*-Iteration in Flink anwenden zu können, müssen alle drei Funktionen mit einer eigenen Logik überschrieben werden. (vgl. [ASF18i])

Wie auch bei *GraphX*, werden die Iteration mithilfe von *Supersteps* synchronisiert. Ein Beispiel dafür lässt sich in der Abb. 4.5 finden. Hier werden in ein *Superstep* die Nachrichten des vorherigen *Supersteps* empfangen, die Werte der Knoten aktualisiert und anschließend alle neue Nachrichten zur gleichen Zeit an den nachfolgenden *Superstep* gesendet.

4.3.3 Graphalgorithmen

Gelly bringt schon eine Vielzahl an verschiedenen Graphalgorithmen mit. Im Folgenden werden einige dieser Algorithmen kurz erläutert (vgl. [ASF18i]):

- **Community-Detection**: sucht nach Gruppen, dessen Knoten gut miteinander verbunden sind, aber nur wenig mit anderen Gruppen. Der Algorithmus, welcher intern genutzt wird, ist im folgenden Paper genauer beschrieben [LHLC09].

- **Label-Propagation**: *Gelly* bringt die Implementierung eines bekannten Label-Propagation-Algorithmus mit, der in dem Paper [RAK07] genauer beschrieben wird. Der Algorithmus erkennt ebenfalls Gruppen, indem er iterativ die Labels zwischen Nachbarn voraussagt. Um die Iteration durchführen zu können, wurde das *Scatter-Gather*-Modell, welches weiter oben schon beschrieben wurde, genutzt.

- **Connected Components**: Dieser Algorithmus überprüft für z.B. zwei Knoten, ob sie zur gleichen „Komponente“ gehören. Dies ist der Fall, wenn die beiden Knoten über Kanten miteinander verbunden sind, ungeachtet der Richtung dieser Kanten. Zu erwähnen ist noch, dass *Gelly* verschiedene Implementierungen für diesen Algorithmus bereitstellt.

- **Single-Source-Shortest-Path**: ausgehend von einem Startknoten, werden die kürzesten Wege zu allen anderen Knoten in einem Graph berechnet. Dabei wird intern wieder auf das *Scatter-Gather*-Modell für die iterative Berechnung der Lösung zurückgegriffen. Auch hier bietet *Gelly* wieder verschiedene Implementierungen an, sodass der Nutzer sich die Beste für seinen Anwendungsfall aussuchen kann.

- **Triangle Enumerator**: zählt die Anzahl an einzigartigen Dreiecken in einem Graph. Ein Dreieck besteht dabei aus drei Kanten, die die drei Knoten miteinander verbinden.

- **Summarization**: Oftmals hat man bei bestimmten Analysen auf einem Graph das Problem, dass der Ergebnisgraph zu groß ist, um ihn für weiterführenden Analysen nutzen zu können. *Summarization* berechnet eine verdichtete/zusammengefasste Version des Graphen, indem Knoten und Kanten auf Grundlage ihrer Werte zusammengefasst werden. Dadurch wird ein Graph übersichtlicher und kann eventuell besser ausgewertet werden.

- **Clustering**: *Gelly* bietet unterschiedliche Clustering-Algorithmen für das Clustern von Graphen an. So kann man sich den *Average-Cluster-Coefficient* berechnen lassen, der das Maß der Konnektivität eines Graphen angibt. Der Wert geht von 0,0 (keine Kanten zwischen Nachbarn) bis 1,0 (kompletter Graph/alle Knoten miteinander verbunden). Auch andere Koeffizienten wie der *Global-Cluster-Coefficient* und *Local-Clustering-Coefficient* können berechnet werden.

- **Link-Analysis**: enthält den *PageRank* von Google und die *Hyperlink-Induced-Topic-Search*. Letzterer Algorithmus berechnet die voneinander abhängigen Werte von Knoten in einem gerichteten Graph.

- **Metric**: gibt bestimmte Metriken eines Graphen, wie z.B. die Anzahl aller Kanten und Knoten, aus.

4.4 Gradoop

Gradoop (Graph Analytics on Hadoop) ist ebenfalls ein Framework für Graphanalyse, welches an der *Universität Leipzig* entwickelt wurde. Es wurde auf Flink und der NoSQL-Datenbank *HBase* implementiert. Im Moment handelt es sich noch um einen Forschungsprototypen, der allerdings schon erfolgreich mit unterschiedlichen Datensätzen getestet wurde. Dieses Unterkapitel orientiert sich in seinem Inhalt und seiner Struktur an den beiden Fachbeiträgen von *Junghanns et al.* ([JPNR17, JPT+16]).

4.4.1 Anforderungen

Junghanns et al. formulieren folgenden Anforderungen an ein Framework zur Graphanalyse (vgl. [JPNR17]):

- **Mächtiges Graphdaten-Modell**: das System sollte nicht nur homogene Graphen darstellen können, sondern auch Graphen mit heterogenen Kanten und Knoten mit verschiedenen Typen und unterschiedlichen Attributen. Zudem sollte der Benutzer nicht dazu gezwungen sein ein festes Schema für Kanten und Knoten vorgeben zu müssen.
- **Mächtige Anfrage- und Analysemöglichkeiten**: Der Benutzer sollte die Daten mit einer deklarativen Anfragesprache analysieren können. Dabei muss ebenso das Konzept der Iteration abgedeckt werden. Wenn das Framework auf einem eigenen Datenmodell basiert, sollten ebenso grundlegende Graph-Operatoren auf diesem Datenmodell zur Verfügung stehen, die die Entwicklung von Graphalgorithmen erleichtern.
- **Hohe Ausführungsgeschwindigkeit und Skalierbarkeit**: Die Analyse von Graphen sollte schnell und auf sehr große Datensätze skalierbar sein. Das setzt typischerweise die Nutzung von verteilten Clustern und *In-Memory*-Verarbeitung voraus, sodass die Algorithmen auf beliebig viele Knoten eines Clusters skaliert werden können.
- **Persistente Graphspeicherung und Transaktionsunterstützung**: Der Benutzer sollte bestimmte Graphen dauerhaft abspeichern können. Zusätzlich sollte das Framework auch OLTP-Funktionalitäten (Online Transaction Processing) mit ACID-Transaktionen bieten, um Graphdaten effizient bearbeiten zu können.
- **Einfache Benutzung/Visualisierung**: Die Benutzung des Frameworks sollte so einfach wie möglich sein. Der Nutzer sollte also in der Lage dazu sein, interaktiv Anfragen und Analysen auf Graphdaten zu formulieren. Dazu sollten mächtige Operatoren und Analysemöglichkeiten zur Verfügung gestellt werden. Zusätzlich sollte eine grafische Benutzeroberfläche zur Definition von Workflows, wie auch die Möglichkeit zur Visualisierung von Graphen, existieren.

Viele Frameworks erfüllen einige, aber nicht alle dieser Kriterien. So sind *Gelly* und *GraphX* z.B. nicht dazu in der Lage ihre Ergebnisse zu visualisieren, oder Graphen ohne ein festes Schema zu definieren. Es existieren auch keine deklarativen Anfragesprachen, die für die Analyse von Graphdaten genutzt werden können. Spark und Flink haben zwar jeweils eine SQL-API, diese kann aber nur für Anfragen auf relationale Datensätze genutzt werden.

4.4.2 Architektur

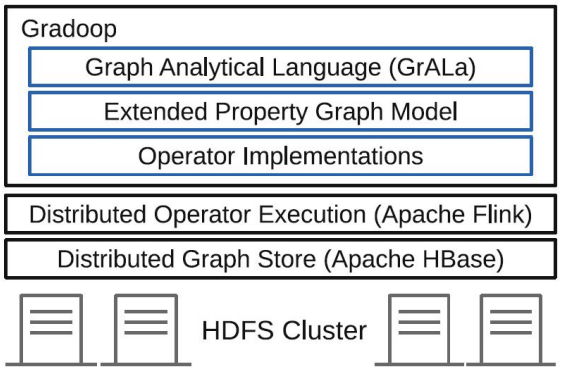


Abbildung 4.8: Die Architektur des Gradoop-Frameworks (Quelle: [JPNR17]).

Gradoop hat das Ziel ein Framework für das Management und die Analyse von Graphdaten zur Verfügung zu stellen. Um die horizontale Skalierbarkeit zu ermöglichen, läuft Gradoop auf einem *Shared-Nothing*-Cluster und benutzt Hadoop-basierte Software für die verteilte Speicherung und Verarbeitung. Der Benutzer implementiert seine Algorithmen dann mithilfe einer speziellen Anfragesprache, genannt *Graph-Analytical-Language* (GrALa). Diese Anfragesprache enthält eine Menge von Operatoren zur Analyse von einzelnen, oder auch mehreren Graphen. GrALa wurde auf Grundlage des sogenannten *Extended-Property-Graph-Modell* (EPGM) entwickelt und ähnelt in seiner Struktur der Anfragesprache *Cypher* ([JKA+17]). Das EPGM-Datenmodell wird im nachfolgenden Abschnitt genauer erklärt. Um die Programme in einer verteilten Umgebung ausführen zu können, ist *Gradoop* auf *Apache Flink* implementiert worden. Dadurch kann *Gradoop* die existierenden Techniken zur Parallelisierung von Algorithmen nutzen, ohne diese selber implementieren zu müssen. Flink übernimmt somit die Optimierung, wie auch Datenverteilung und Parallelisierung der Ausführung in einem Cluster von Computern. Zudem kann *Gradoop* einfach in Kombination mit anderen Bibliotheken von Flink genutzt werden. Der verteilte Datenspeicher wird über *Apache HBase* sichergestellt. Dabei handelt es sich um eine verteilte, nicht-relationale Datenbank die auf dem *Hadoop-Distributed-File-System* (kurz HDFS) läuft. Dadurch kann *HBase* als Quelle und Senke von Analyseprogrammen, welche mithilfe von *Gradoop* geschrieben wurden, genutzt werden. In Abb. 4.8 sieht man die Architektur des Gradoop-Frameworks.

4.4.3 Extended-Property-Graph-Model

EPGM erweitert das bekannte *Property-Graph-Model* (kurz PGM), welches auch von der Bibliothek *GraphX* in Spark genutzt wird. EPGM erweitert PGM, indem es z.B. zusätzlich eine Menge von Graphen unterstützt. Im Folgenden wird erklärt was man unter einem *Extended-Property-Graph* versteht und welche Operatoren diese Datenstruktur mitbringt.

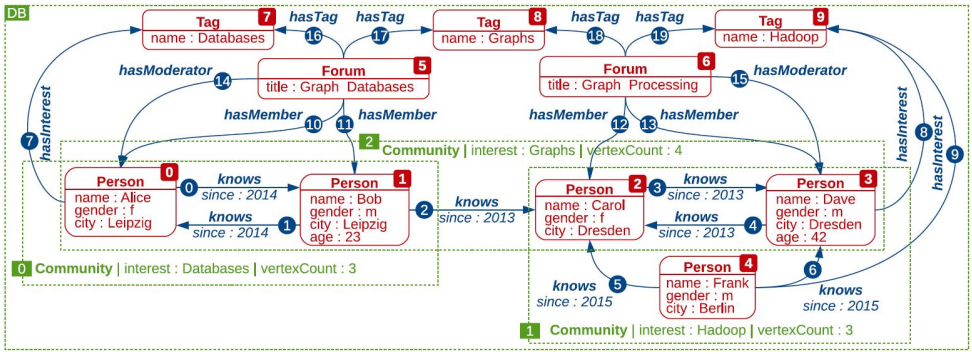


Abbildung 4.9: Ein Beispiel für ein EPGM (Quelle: [JPT+16]).

Graphrepräsentation

Ein *Property-Graph* ist ein gerichteter, attributierter und typisierter Multigraph. Um die Heterogenität auszudrücken, können Knoten und Kanten jeweils mit Typen versehen werden. Attribute sind dann Key-Value-Paare und stellen die Eigenschaften (engl. *properties)*) dar. Ein *Extended-Property-Graph* ist dann eine Datenbank aus mehreren *Poperty-Graphen*, welche *logische Graphen* genannt werden. Diese Graphen sind dann anwendungsspezifische Untermengen von einer gemeinsamen Menge an Knoten und Kanten. Daraus ergibt sich dann folgende Definition für einen *Extended-Property-Graph* (vgl. *Junghanns et al.* [JPT+16]):

Definition 4.1 (EPGM Datenbank)**.** *Eine EPGM Datenbank* $DB = (V, E, L, T, \tau, K, A, \kappa)$ *besteht aus einer Menge von Knoten* $V = \{v_i\}$*, einer Menge von Kanten* $E = \{e_k\}$ *und einer Menge von logischen Graphen* $L = (G_m)$*. Die Knoten, Kanten und logischen Graphen werden durch die Indizes* $i, k, m \in \mathbb{N}$ *gekennzeichnet. Eine Kante* $e_k = (v_i, v_j)$ *mit* $v_i, v_k \in V$ *ist von* v_i *nach* v_j *gerichtet und unterstützt Eigenreferenzierung (z.B.* $i = j$*). Es können mehrere Kanten zwischen zwei Knoten verlaufen, welche sich aber durch ihre Indizes unterscheiden. Ein logischer Graph* $G_m = (V_m, E_m)$ *ist ein geordnetes Paar von Teilmengen für die Knoten* $V_m \subseteq V$ *und für die Kanten* $E_m \subseteq E$*, bei dem gilt* $\forall (v_i, v_j) \in E_m : v_i, V_j \in V_m$*. Logische Graphen können sich überlappen, sodass* $\forall G_i, G_j \in L : |V(G_i) \cap V(G_j)| \geq 0 \wedge |E(G_i) \cap E(G_j)| \geq 0$*. Für die Definition der Typen wird das Alphabet* T *und ein Zuordnungsfunktion* $\tau : (V \cup E \cup L) \rightarrow T$ *genutzt. Ebenso sind die Eigenschaften (Key-Value-Paare) durch eine Menge von Schlüsseln* K*, eine Menge von Werten* A *und einer Zuordnungsfunktion* $\kappa : (V \cup E \cup L) \times K \rightarrow A$ *definiert.*

In Abb. 4.9 ist ein Beispiel für ein *Extended-Property-Graph* zu sehen. Hier sind die logischen Graphen durch die grünen Kästen gekennzeichnet. Jeder logische Graph hat einen Typ (z.B. „Community“) und weitere Eigenschaften (z.B. „interest: Databases“). Wie man sieht, überlappen sich in diesem Beispiel die logischen Graphen 1 und 2. Auch Knoten und Kanten haben jeweils einen Typ und Attribute.

Operatoren

Gradoop stellt Operatoren für einzelne und eine Menge von Graphen bereit, die mit der eigenen Anfragesprache GrALa genutzt werden können. Die nachfolgende Tabelle listet alle unären und binären Operatoren auf (vgl. Junghanns et al. [JPNR17], [JPR17]):

	Operator	**Output**
Unäre	Aggregation	Graph
	Transormation	Graph
	Pattern Matching	Collection
	Subgraph	Graph
	Grouping	Graph
	Selection	Collection
	Distinct	Collection
	Limit	Collection
	Sorting	Collection
Binäre	Equality	Boolean
	Combination	Graph
	Exclusion	Graph
	Overlap	Graph
	Equality	Boolean
	Difference	Collection
	Intersect	Collection
	Union	Collection

Gradoop stellt also unäre Operatoren, wie z.B. *Aggeragtion, Selection, Distinct, Limit* und auch *Sorting*, bereit. Neben diesen klassischen unären Operatoren wurden aber auch *Pattern-Matching* und *Subgraph* implementiert. In vielen Algorithmen für die Analyse von Graphen ist es der Fall, dass man nach einer Menge von Teilgraphen sucht, die isomorph zu einem gegebenen Graph-Muster sind. Auch beim *Frequent-Subgraph-Mining*, welches in dem vorherigen Kapitel 3.2 schon näher erläutert wurde, ist eine Teilaufgabe nach einem bestimmten Teilgraphen zu

suchen. Mithilfe von *Subgraph* lässt sich ein Teilgraph aus einem Graph extrahieren, der die gegebenen Kanten und Knoten enthält. *PatternMatching* hingegen kann genutzt werden, um eine Menge von Teilgraphen die bestimmte Kriterien erfüllen, aus einem Graph zu extrahieren. Mit den binären Operatoren werden unter anderem klassische Operationen aus der Mengenlehre abgedeckt. So lassen sich einzelne Graphen, oder auch eine Menge von Graphen miteinander kombinieren.
Alle EPGM-Operatoren von *Gradoop* wurden auf Basis der DataSet API von Flink implementiert. So besteht also jeder Operator aus einer Sequenz von Transformationen auf den jeweiligen *DataSets*.

4.5 Hydra.PowerGraph

In diesem Unterkapitel wird ganz kurz erläutert, worum es sich bei dem *Hydra.PowerGraph*-System handelt, da es ein fundamentaler Bestandteil des *WossiDiA*-Projekts und somit für den Anwendungsfall der Arbeit relevant ist. Das *WossiDiA*-System, welches im Kapitel 1 dieser Arbeit kurz erläutert wurde, war die Hauptmotivation, welche zu der Entwicklung des *Hydra.PowerGraph*-Systems geführt hat. Dieses System wurde ebenfalls am *Lehrstuhl für Datenbank- und Informationssysteme* der *Universität Rostock* entwickelt. Es besteht aus zwei Elementen: dem Hypergraph-Datenbanksystem *PowerGraph* und dem Framework für digitale Archive, genannt *Hydra*.
PowerGraph ist dabei nur eine Erweiterung der *objekt-relationalen* Datenbank *Postgres*. Es nutzt die räumlich/zeitliche Erweiterung und *objekt-relationale* Besonderheiten, wie z.B. UDFs und Vererbung, des *Postgres*-Systems aus.

4.5.1 Das Hypergraph-Modell

Hydra.PowerGraph nutzt als Datenmodell einen gerichteten, typisierten Hypergraphen, wie er in Def. 2.7 definiert wurde. Durch die Knoten- und Hyperkantentypen können spezielle Datenstrukturen in der Datenbank ausgedrückt werden. Dadurch kann der Benutzer auf eine Menge von typ-spezifischen Operationen zugreifen. Wie schon erwähnt nutzt *Hydra.PowerGraph* das *objekt-relationale* Datenbanksystem *Postgres*. In Abb. 4.10 ist die *objekt-relationale* Speicherstruktur eines gerichteten, typisierten Hypergraphen in *Postgres* zu sehen.
Dabei besteht die Datenbank aus zwei Teilen, einem Katalog-Schema (*catalog*) und einem Daten-Schema (*data*). Während im Katalog-Schema alle Modellinformationen gespeichert sind, werden im Daten-Schema die Datensätze an sich gespeichert. Da sich die Knoten in einem Hypergraphen nicht von den Knoten eines Graphen unterscheiden und somit leicht zu verstehen sind, ist der interessante Punkt die relationale Darstellung einer Hyperkante und der beteiligten Knoten. So gibt es eine Relation (*Hyperedge*) in der die Hyperkanten und eine Relation (*Node*) in der die

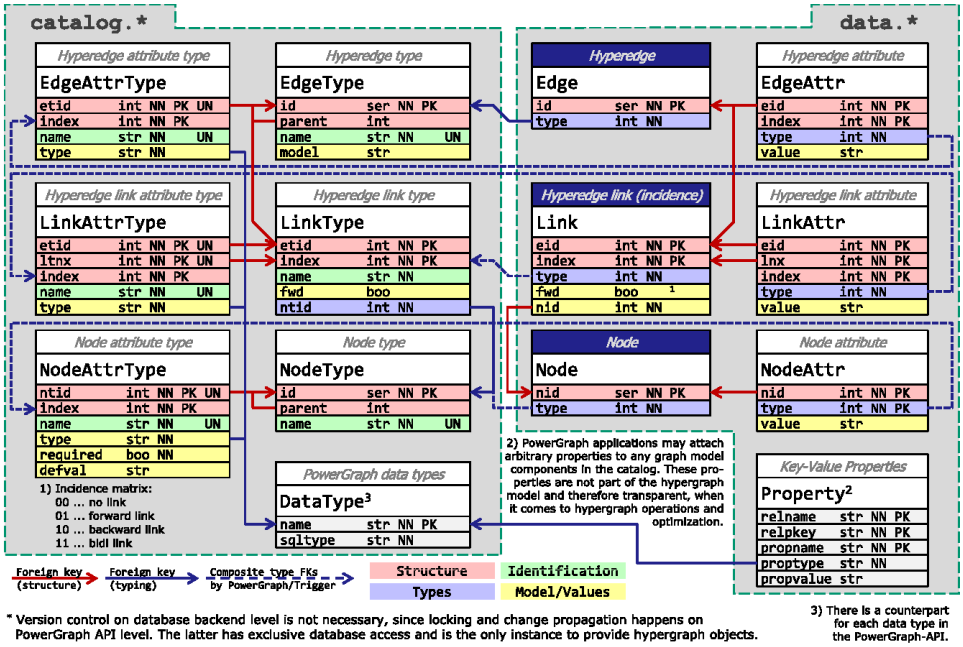


Abbildung 4.10: Die relationale Speicherstruktur eines gerichteten typisierten Hypergraphen (Quelle: *Meyer et al.* [MSH17]).

Knoten gespeichert sind. Die dritte Relation (*Hyperedge link*) gibt dann an, welche Knoten (*nid*) an welcher Hyperkante (*eid*) beteiligt sind und ob es sich bei dem Knoten um einen eingehenden oder ausgehenden Knoten handelt (*fwd*). Wenn es sich z.B. um einen eingehenden Knoten (auch Quelle genannt) handelt, wäre der Eintrag für *fwd* -1 (siehe die Definition von δ in Def. 2.7) . Für einen ausgehenden Knoten (auch Senke genannt) wäre der Wert von *fwd* dann 1.

4.5.2 Anfragen im Hydra.PowerGraph-System

PowerGraph bringt seine eigene deklarative Anfragesprache *GrafL* mit, die dadurch auch im *Hydra.PowerGraph*-System genutzt werden kann. Dabei beruht die Sprache auf einer Menge von Graph-Operatoren. Dadurch ist es möglich mit *GrafL* Knoten und Hyperkanten auf Grundlage ihrer Attribute zu selektieren. Es können aber auch komplexe Graphalgorithmen auf Grundlage der Basisoperationen von *GrafL* implementiert werden. Es gibt sogar einen extra Operator für den *k-Shortest-Path*-Algorithmus.
GrafL ähnelt der Anfragesprache *XQuery* für XML-Dokumente. Die Graphalgorithmen sind dabei in Funktionen eingekapselt. Wo *XQuery* also auf Sequenzen von XML-Fragmenten arbeitet, wird bei *GrafL* auf Knoten- und Hyperkantenmengen gearbeitet und normalerweise eine Hyperkantenmenge zurückgeliefert.

4.6 Anwendbarkeit für das Frequent-Subgraph-Mining

In diesem Unterkapitel wird sich damit beschäftigt, wie die beiden Plattformen *Apache Spark* und *Apache Flink* die Parallelisierung des *Frequent-Subgraph-Mining* unterstützen können. Dazu wird zunächst auf die beiden Bibliotheken *Gelly* und *GraphX* und auf das Framework *Gradoop* näher eingegangen.

4.6.1 Bibliotheken für die Verarbeitung von Graphen

Flink und Spark bringen jeweils eine Bibliothek für die Verarbeitung von Graphen mit. Damit soll es möglich sein verschiedene Graphen analysieren zu können. Die dazu zur Verfügung gestellten Operatoren von *GraphX* und *Gelly* wurden schon in den vorherigen Abschnitten 4.1.2 und 4.3.1 besprochen. Beide Bibliotheken bringen in etwa die selben Menge an Operationen mit. Allerdings sind diese Operatoren nur für einfache *Graph-Primitive*, also einfache Analysen auf Graphen, verwendbar. Will man ein komplexeres *Mining*-Verfahren implementieren, kommt man mit den bereitgestellten Operatoren schnell an die Grenzen beider Bibliotheken. Ein zweiter Kritikpunkt ist in diesem Zusammenhang die jeweils bereitgestellte Datenstruktur beider Bibliotheken. Denn egal ob man das *Property-Graph*-Modell von *GraphX* oder die Datenstruktur der Graph API von *Gelly* benutzt, kann man in keinem Fall eine Menge von Graphen in einer Datenstruktur speichern. Beide sind nur darauf ausgelegt einen einzelnen Graphen, nicht aber eine Menge von Graphen repräsentieren zu können. Damit sind diese beiden Bibliotheken also für die Implementierung eines *Frequent-Subgraph-Mining*-Verfahrens bisher ungeeignet.

Gradoop hingegen hat das Potenzial diese Lücke für *Apache Flink* schließen zu können. Wie in Abschnitt 4.4.3 schon erläutert, verwendet dieses Framework das *Extended-Property-Graph*-Modell (kurz *EPGM*). Mit *EPGM* ist es möglich eine Menge von Graphen in einer einzelnen Datenstruktur speichern zu können. Damit bietet es gegenüber bisherigen Bibliotheken einen Vorteil in der Speicherung und Verarbeitung mehrerer Graphen. Allerdings handelt es sich bei dem Framework aktuell noch um einen Forschungsprototypen. Auch die bisher implementierten Operatoren reichen für das *Frequent-Subgraph-Mining* aktuell alleine nicht aus. Deswegen ist auch hier momentan noch von einer Implementierung auf Basis von *Gradoop* abzusehen.

4.6.2 APIs für die Verarbeitung von Batch-Daten

Da die Bibliotheken zur Verarbeitung von Graphen bisher alle nicht nutzbar für eine Implementierung sind, bleibt noch die Möglichkeit das *Frequent-Subgraph-Mining*-Verfahren mit den herkömmlichen APIs zur Batch-Daten-Verarbeitung zu implementieren. Flink und Spark bringen mit der DataSet API und RDD API jeweils eine Möglichkeit mit, Batch-Daten verarbeiten zu

können. Dabei müsste dann aber jeweils eine eigene Datenstruktur zur Repräsentation eines Graphen entwickelt werden, die dann im Beispiel von Flink in *DataSets* gespeichert werden könnte. Spark bringt diese Möglichkeit mit seinen *RDDs* ebenfalls mit. Beide Plattformen können also eigens definierte Datenstrukturen mithilfe ihrer eigenen Datenstruktur speichern. Im Folgenden wird an dem Beispiel von Flink gearbeitet, wobei die folgenden Punkte sich auch mit Spark umsetzen lassen würden.
Mit den *DataSets* von Flink kann man also eigens definierte Datenstrukturen abspeichern. Dabei muss für das jeweilige *DataSet* mithilfe von *Generics* nur angegeben werden, um was für eine Datenstruktur es sich dabei handelt. Dadurch, dass es sich bei einem *DataSet* um eine fehlertolerante, verteilte Datenstruktur handelt, übernimmt Flink somit dann schließlich die Verteilung der in dem *DataSet* gespeicherten Daten von alleine. Somit könnte also die eigens definierte Datenstruktur von Flink verteilt und fehlertolerant gemacht werden, was für die parallele Verarbeitung unabdingbar ist. Das Gleiche würde sich, wie vorher schon erwähnt, auch mit der RDD API von Spark umsetzen lassen.
Die Operatoren der beiden APIs gleichen sich im Allgemeinen. Dadurch muss hier ebenfalls keine Unterscheidung zwischen Spark und Flink getroffen werden. Dadurch, dass man bei den verschiedenen Operatoren die Logik mit der eigenen Logik überschreiben kann, sollte dies dabei helfen die für das *Frequent-Subgraph-Mining* benötigten Operatoren zu implementieren und parallelisieren zu können.
Wenn man diese Informationen nun alle zusammenfasst, kann man sagen, dass die APIs für die Batch-Daten-Verarbeitung von Spark und Flink jeweils beide dazu geeignet wären, um mit ihnen ein *Frequent-Subgraph-Mining*-Verfahren zu implementieren. Welche der beiden Plattformen zu wählen ist, wird Inhalt von Kapitel 5 sein.

Kapitel 5

Problemanalyse

Dieses Kapitel beinhaltet eine detaillierte Problemanalyse, welche als Grundlage für die Entwicklung von *PaSiGraM*, einem eigenen FSM-Algorithmus, diente. Dazu wird zunächst der bereitgestellte Datensatz erläutert, welcher im Zusammenhang mit dem Anwendungsfall dieser Arbeit steht. Anschließend werden die beiden schon vorgestellten Parallelisierungsplattformen *Apache Spark* und *Apache Flink* anhand verschiedener Kriterien miteinander verglichen, um eine Empfehlung geben zu können, mit welcher Plattform *PaSiGraM* umgesetzt werden sollte. Danach wird ein Überblick über den Stand der Forschung gegeben und wie dieser für den eigenen FSM-Algorithmus genutzt werden kann. Am Ende dieses Kapitels wird eine Liste mit Anforderungen vorgestellt, die bei der Entwicklung von *PaSiGraM* zu berücksichtigen waren.

5.1 Anwendungsfall

Wie in Unterkapitel 1.1 schon erwähnt, soll der Algorithmus im Rahmen des *ISEBEL*-Projekts zum Einsatz kommen, um häufig auftretende Muster in dem *WossiDiA*-Datensatz zu finden und somit den Forschern auf dem Gebiet der Folkloristik neue Betrachtungsweisen in der Motivforschung zu ermöglichen. Bei dem zu Verfügung gestellten Datensatz handelt es sich um einen großen zusammenhängenden Hypergraphen. Da aktuell keine Algorithmen existieren, um *Frequent-Subgraph-Mining* auf Hypergraphen zu realisieren, ist zusätzlich zu betrachten, wie man den Hypergraphen in einen Graphen überführen kann. Die nächsten beiden Abschnitte (5.1.1 - 5.1.2) erläutern wie der Original-Datensatz aussieht und welche Methode genutzt wurde, um den Hypergraphen in einen Graphen zu überführen.

5.1.1 WossiDiA-Datensatz

Der Datensatz für den der Algorithmus zu entwerfen war, stammt aus dem *WossiDiA*-Projekt. Wie in Abschnitt 1.1.1 erwähnt, werden die Daten in einer Hypergraph-Struktur zur Verfügung gestellt. Dabei besteht der Datensatz aus ungefähr 3,1 Millionen Knoten und 1,4 Millionen Kanten. Da dieser Datensatz im *Hydra.PowerGraph*-System liegt, wird als Datenmodell ein gerichteter, typisierter Hypergraph verwendet (wie in Definition 2.7 festgelegt). Dieses Datenmodell wird allerdings auf ein *objekt-relationales* Modell abgebildet, damit es in *PostgreSQL* abgespeichert werden kann (siehe Abschnitt 4.5.1). Aus dieser Datenbank wurden alle Daten in zwei Datensätzen *nodes.csv* und *edges.csv* exportiert, welche die Menge der Knoten und Kanten enthalten. Diese beiden Dateien haben folgende Struktur:

- nodes.csv = {ID, value}
  - ID = $\{V_{ID} : ID$ ist fortlaufend nummeriert und $ID \in \mathbb{N}^+ = \{1, 2, 3, ...\}\}$
  - value = $\{V_l : l$ ist ein Attribut vom Knoten $V\}$
- edges.csv = {sourceID, destinationID, value}
  - sourceID = $\{S_{ID}$: ist die Menge der Startknoten mit $S_{ID} \subseteq V_{ID}\}$
  - destinationID = $\{D_{ID}$: ist die Menge der Zielknoten mit $D_{ID} \subseteq V_{ID}\}$
  - value = $\{E_l : l$ ist ein Attribut der Kante $E\}$

5.1.2 Der Export

Beim Export der Daten aus *PostgreSQL*, wurde nicht das Hypergraph-Modell exportiert, sondern ein attribuierter, gerichteter Graph (siehe Kapitel 2.1). In Kapitel 2.3 wurde schon erläutert, dass es zwei Ansätze gibt, um einen Hypergraphen in einen Graphen zu überführen. Für den Export des Beispieldatensatzes wurde der Ansatz der *Clique-Expansion* gewählt. Alle Hyperkanten wurden also als Kanten dargestellt. Aber auch hier sind unterschiedliche Punkte zu berücksichtigen. Unter anderem ist darauf zu achten, ob man die Richtung einer Hyperkante berücksichtigt oder nicht. In Abb. 5.1 ist zu sehen, zu welchen unterschiedlichen Ergebnissen die Berücksichtigung der Richtung von Hyperkanten führen kann. Ebenso kann es in Datensätzen, in denen nur die Menge der Startknoten explizit angegeben ist, zu unterschiedlichen Ergebnissen der *Clique-Expansion* kommen. Denn in diesem Fall kommt es zusätzlich auf die Definition des Hypergraphen an. In Kapitel 2.2 wurde erklärt, dass sich Hypergraphen in der Definition ihrer Kanten unterscheiden können (*Gallo* [GLPN93] und *Ausiello* [AL17]). Der Datensatz mit dem hier gearbeitet wird hält sich an Definition 2.7. Somit ist es möglich, dass die Mengen der Start- und Zielknoten gemeinsame Elemente enthalten können und somit nicht *disjunkt* sein müssen.

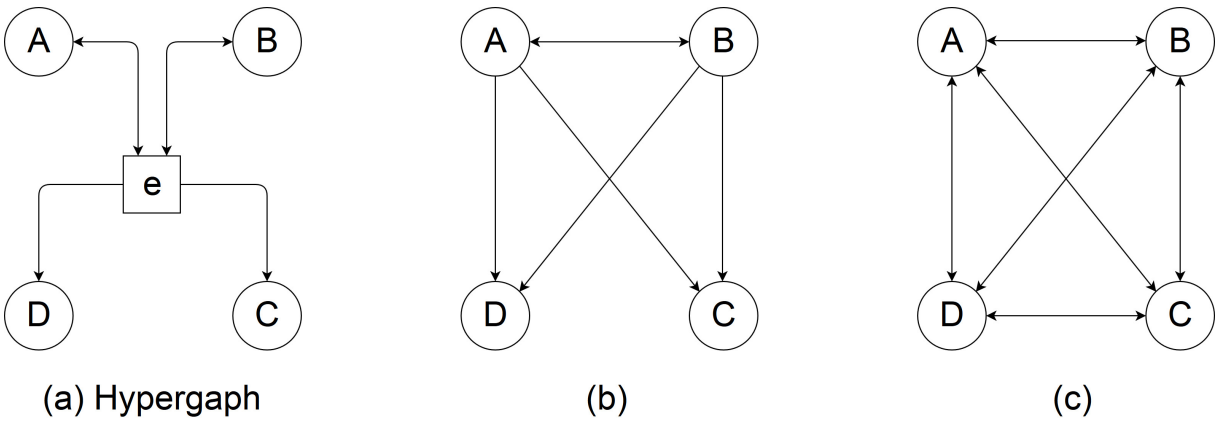


Abbildung 5.1: Der Hypergraph (a) als *Clique-Expansion* unter Beachtung (b) und Nicht-Beachtung (c) der Richtung einer Hyperkante.

5.2 Auswahl der Plattform

Ein weiterer Punkt im Zusammenhang mit der Parallelisierung von FSM-Algorithmen sind natürlich die beiden schon vorgestellten Plattformen *Apache Spark* und *Apache Flink*. In den nächsten Abschnitten werden diese anhand folgender Kriterien einem Vergleich unterzogen:

1. Laufzeitverhalten:
   (a) Eigene Messergebnisse
   (b) Messerergebnisse einer anderen Arbeit
2. Benutzerfreundlichkeit:
   (a) Dokumentation
   (b) APIs
3. Bibliotheken für das Graph-Mining:
   (a) Datenstruktur
   (b) Operatoren
   (c) Algorithmen

Die Ergebnisse des Vergleichs beruhen dabei auf einem eigenen Projekt, verschiedenen Arbeiten, die zu diesem Thema schon verfasst wurden und eigenen Erfahrungen im Umgang mit *Apache Spark* und *Apache Flink*.

5.2.1 Laufzeitverhalten

Das Laufzeitverhalten einer Parallelisierungsplattform hat einen großen Einfluss darauf, wie schnell bestimmte Berechnungen durchgeführt werden können. Dazu werden in diesem Abschnitt die Ergebnisse einer eigenen Studie, wie auch einer anderen Arbeit aufgezeigt.

Eigene Messergebnisse
Im Sommersemester 2017 wurde am *Lehrstuhl für Informations- und Datenbanksysteme* der *Universität Rostock* eine Studie ([DFS+18]) durchgeführt, welche auf einem älteren Projekt von *Stonebraker et al.* [SAD+10] basiert. Dabei sollte die Laufzeit von zwei verschiedenen Parallelisierungsplattformen verglichen werden. Letztendlich wurden Flink in der Version 1.2.0 und Spark in der Version 2.1.1 miteinander verglichen. Beide Plattformen liefen auf Hadoop in der Version 2.7.2. Als Input-Datensatz diente ein *Twitter-Follower*-Graph. Dieser enthält die IDs aller Twitter-Konten und Informationen darüber, wer wem bei Twitter folgt.
Bei der Wahl der Algorithmen wurde sich teilweise an der Arbeit [SAD+10] von *Stonebraker et al.* orientiert. Schließlich waren die Laufzeiten für drei verschiedene Algorithmen zu testen:

- **Grep-Task**: Den *Twitter-Follower*-Graph nach einer bestimmten ID durchsuchen.
- **Join-Task**: Hier musste zuvor der *PageRank* über alle Knoten berechnet und ein künstlicher *Weblog* erzeugt werden. Der Ablauf der *Join-Task* lässt sich dann in drei Schritte unterteilen:
  1. IP-Adresse ermitteln, die die meisten Twitter-Accounts in einem bestimmten Zeitraum besucht hat.
  2. Join (PageRank $\bowtie$ Weblog): Summe über *PageRanks* der von der IP-Adresse besuchten Twitter-Accounts bilden.
  3. Aggregation: Durchschnitt über Summen ausgeben.
- **kMeans**: Ein Algorithmus für das Clustern von Datensätzen.

In Tabelle 5.1 sind die verschiedenen Messwerte beider Plattformen für alle drei Algorithmen zu sehen.

Algorithmus	**Apache Flink**	**Apache Spark**
Grep-Task	100 sec.	53 sec.
Join-Task	121 sec.	140 sec.
k-Means	705 sec.	917 sec.

Tabelle 5.1: Laufzeiten dreier verschiedener Algorithmen mit Flink und Spark.

Es ist ersichtlich, dass Flink in zwei von drei Algorithmen die Berechnungen deutlich schneller abschloss als Spark. Dieses Ergebnis wird auch durch die Ergebnisse einer weiteren Arbeit gestützt ([Kae17]), die im Folgenden vorgestellt wird.

Messerergebnisse einer anderen Arbeit
In der Arbeit [Kae17] wurden fünf Hypothesen aufgestellt, wovon allerdings nur vier im Folgenden vorgestellt werden, da sich die fünfte Hypothese auf eine bestimmte Art von Graphen bezieht, welche für diese Arbeit nicht von Relevanz ist:

1. Vertex-centric Algorithmen sind unter Apache Spark und Apache Flink gleich schnell.
2. Apache Flink und Apache Spark brauchen für die x-fache Datenmenge, eine x-mal so lange Zeit für die Verarbeitung (lineares Wachstum).
3. Apache Spark ist schneller als Apache Flink, wenn das Cluster horizontal skaliert wird.
4. Apache Flink hat einen geringeren Arbeitsspeicher-Verbrauch als Apache Spark.

Für die Evaluierung der einzelnen Hypothesen auf den beiden Plattformen, wurden der *PageRank*-Algorithmus und ein Algorithmus für das *Semi-Clustering* von Graphen verwendet. Der Schwerpunkt lag also auf Graph-Algorithmen.
Schon bei der ersten Hypothese stellt sich heraus, dass Flink deutlich schneller als Spark ist. Weiterhin ist hier zu erkennen, dass mit der Menge der Daten auch die Unterschiede in den Berechnungszeiten von Spark und Flink deutlich größer werden. So erzielt Flink bei einem kleinen Graph eine bis zu 2,3-fach und bei einem größeren Graph sogar eine 5,9-fach kürzere Laufzeit als Spark.
Die zweite Hypothese trifft nur teilweise zu. Bei kleineren Eingabegraphen ist das Wachstum beider Plattformen nicht linear. Arbeitet man jedoch mit größeren Graphen, so kann man ein lineares Wachstum bei Flink beobachten, während Spark eher ein geringeres Wachstum aufweist. Dennoch erzielt Flink in seiner Ausführungszeit bessere Zeiten als Spark, obwohl das Wachstum der Ausführungszeit bei größeren Graphen bei Flink höher als bei Spark ist.
Die dritte Hypothese ist ebenfalls nicht zutreffend. Während bei Spark der *Scale-Out* von 2 auf 3 Worker in einer um 2% schnelleren Laufzeit resultierte, erzielte Flink bei gleichem *Scale-Out* eine um 60% kürzere Laufzeit. Hier ist also klar zu erkennen, dass Flink bei dem Thema Skalierbarkeit klar besser ist. Dennoch war es sehr überraschend, dass Spark bei dem *PageRank*-Algorithmus kürzere Laufzeiten erzielte als Flink.
Die vierte Hypothese konnte leider weder bestätigt, noch widerlegt werden. Bei Spark war es möglich die Auslastung des Arbeitsspeichers während der Ausführung zu messen. Flink verwaltet allerdings den Arbeitsspeicher selber, was dazu führt, dass die Runtime zur Ausführungszeit den kompletten ihr zugewiesenen Speicher belegt, unabhängig davon ob dieser wirklich benutzt

wird oder nicht. Die Flink-Runtime weist den Flink-Jobs dynamisch den benötigten Arbeitsspeicher zu und gibt den ihr zur Verfügung gestellten Speicher wieder frei, sobald die jeweiligen Berechnungen beendet sind. Dadurch konnte kein Monitoring des Arbeitsspeicherverbrauchs von Flink vorgenommen werden (vgl. [Kae17]).
Abschließend kann man sagen, dass Flink bei der Ausführungszeit meistens besser abschneidet als Spark und somit einen Vorteil bei diesem Kriterium hat. Das lässt sich vermutlich darauf zurückführen, dass Flink mit der *Dataflow*-Engine einen aktuelleren Ansatz für die Verarbeitung von Batch-Daten bietet. Für das zukünftige Anwendungsszenario einer Stromdaten-Verarbeitung im Zusammenhang mit Graphen, wäre Flink eventuell ebenfalls die bessere Wahl. Wie in Abschnitt 4.1.1 erklärt, werden Stromdaten in Spark durch Micro-Batches dargestellt, was gegenüber Flink einen Nachteil in der Stromdaten-Verarbeitung darstellen kann, da hier die Stromdaten auch als solche im System behandelt werden. Allerdings fehlt für diese Behauptung ein vorhandener Vergleich beider Plattformen, weswegen es sich nur um eine Vermutung/Hypothese handelt, die in weiteren Untersuchungen zu bestätigen bzw. widerlegen ist.

5.2.2 Benutzerfreundlichkeit

In diesem Abschnitt geht es um die Benutzerfreundlichkeit beider Plattformen. So ist es unter anderem wichtig, welche APIs bereitgestellt werden und wie die Dokumentation gepflegt wurde. Hierbei ist allerdings gleich zu erwähnen, dass an dieser Stelle nur auf die benötigten APIs, wie z.B. die DataSet API von Flink oder die RDD API von Spark, eingegangen wird.

APIs für die Batch-Verarbeitung

Beide Plattformen bringen jeweils eine API für die Verarbeitung von Batch-Daten mit. Die Operatoren dieser beiden APIs unterschieden sich nur in ganz wenigen Punkten, weswegen keine der beiden Plattformen hier besser als die jeweils Andere ist. Dennoch gibt es einen Faktor, der hier ebenfalls eine Rolle spielen kann: die Dokumentation. Bevor ein Vergleich stattfinden kann, ist zu erwähnen, dass beide Plattformen die gleichen Programmiersprachen unterstützen: Java, Python und Scala.
Die Dokumentation von Java ist für beide Plattformen gut gepflegt. Auch bei den *Programming Guides* lässt sich kein großer Unterschied ausmachen. Es ist aber auch zu erwähnen, dass Spark die deutlich ausgereiftere API für Python mitbringt und hier eine komplette Dokumentation vorhanden ist. Bei Flink befindet sich die Python API noch in der Beta-Phase und wird auch nur für die DataSet API angeboten, weswegen Spark hier einen Vorteil hat. Da Python viele Bibliotheken für die Analyse von Daten mitbringt, hat die Sprache für viele Data-Scientists in den letzten Jahren an Relevanz gewonnen. Deswegen kann dieser Fakt auch nicht einfach vernachlässigt werden.

In Flink und Spark ist es zudem möglich Batch-Daten in relationale Datensätze umzuwandeln und somit Analysen mithilfe von SQL umzusetzen. Auch hier hat Spark einen leichten Vorteil, da sich in Flink die entsprechenden APIs ebenfalls noch in der Beta befinden und somit auch noch nicht alle Funktionalitäten zu 100% umgesetzt wurden.

5.2.3 Bibliotheken für das Graph-Mining

In diesem Abschnitt geht es nicht darum, in welcher Qualität die von Flink und Spark mitgebrachten Bibliotheken dabei helfen können, *Graph-Mining*-Techniken zu parallelisieren und somit umzusetzen. Dies war Inhalt eines vorherigen Abschnitts. Trotzdem werden die Bibliotheken miteinander verglichen, da es für weiterführende Arbeiten von Interesse sein könnte. Es folgt nun also eine bloße Betrachtung der Bibliotheken und der in ihnen enthaltenen Funktionalitäten.

GraphX (Spark) vs. Gelly (Flink)
In Kapiteln 4.1.1 und 4.2.1 wurde schon erläutert, dass Spark und Flink jeweils eine Bibliothek zur Verarbeitung von Graph-Strukturen mitbringen. Diese unterteilen sich jeweils in Graph-Operatoren und fertige Algorithmen, die z.B. zum Zählen von Dreiecken oder Finden von kürzesten Pfaden genutzt werden können. In diesen Punkten unterscheiden sich beide Bibliotheken auch nicht groß voneinander, bis auf den Fakt, dass *Gelly* mehrere Varianten von Implementierungen fertiger Algorithmen mitbringt und somit eine passgenauere Auswahl auf Grundlage der Daten ermöglicht.
Das ist allerdings nur der Fall, weil *Gelly* mehrere Schnittstellen für unterschiedliche Modelle zur iterativen Verarbeitung von Graphen anbietet. Und genau dieser Fakt ist ein deutlicher Vorteil gegenüber *GraphX*, da bei *GraphX* eine solche Schnittstelle komplett fehlt(vgl. [XGFS13]). Dieser Nachteil soll durch den *Pregel-Operator* kompensiert werden, den *GraphX* mitbringt und welcher auf dem Pregel-Konzept basiert. Dadurch wird von *GraphX* allerdings nur ein einziges Modell für die iterative Verarbeitung von Graphen angeboten. Somit hat *Gelly* hier mit drei unterschiedlichen Modellen einen klaren Vorteil, der sich auch teilweise in der Performance niederschlagen kann (vgl. [Kae17]).

Gradoop
In Kapitel 4.4 wurde das Framework *Gradoop* kurz vorgestellt. Wie dort schon erwähnt, wurde es auf Flink implementiert und bietet mit seiner EPGM-Datenstruktur einen Vorteil gegenüber *Gelly* und *GraphX*. Denn die herkömmlichen Bibliotheken von Spark und Flink bieten keine Möglichkeit eine Menge von Graphen abzuspeichern. Zudem bietet *Gradoop* mit *GrALa* eine deklarative Anfragesprache mit der Analysen auf Graphen formuliert werden können. *Gradoop* wurde unter anderem mithilfe der DataSet API implementiert und nutzt somit die Parallelisierungstechniken von Flink aus. Dadurch kann Flink eine Bibliothek mehr für die Verarbeitung

von Graphen bereitstellen. Unberücksichtigt bleibt jedoch, dass es sich bei dem Framework noch um einen Forschungs-Prototypen handelt.

Somit hat Flink einen deutlichen Vorteil bei der Verarbeitung von Graphen. Das spiegelt sich zum einen in der Anzahl der Iterationsmodelle wieder, zum anderen aber auch in dem Fakt, dass mehr als eine Bibliothek für Graph-Analysen bereitgestellt wird.

5.2.4 Zusammenfassung

In Tabelle 5.2 ist ein Überblick der Ergebnisse des Vergleichs zwischen Spark und Flink dargestellt. Beim Punkt *Laufzeit* hat Flink einen Vorteil, da das Fundament dieser Plattform einen aktuelleren Ansatz bietet, als das von Spark. Bei der *Benutzerfreundlichkeit* hingegen kann sich Spark durchsetzen. Allerdings nur wegen der ausgereiften Python API. Flink gewinnt den Vergleich jedoch letztendlich dadurch, dass es mehrere Bibliotheken für die Verarbeitung von Graphen mitbringt und mehrere Iterationsmodelle anbietet. Allerdings ist hier noch zu erwähnen, dass die Bibliotheken beider Plattformen (*GraphX* und *Gelly*) bisher noch nicht darauf ausgelegt sind einen FSM-Algorithmus zu implementieren, da die mitgebrachten Datenstrukturen für die Darstellung von Graphen es nur erlauben, einen einzelnen, nicht aber eine Menge von Graphen, zu speichern. Mit *Gradoop* kann Flink in Zukunft jedoch diese Lücke schließen. Aber auch hier bleibt abzuwarten, ob sich der Forschungsprototyp zu einem stabilen Framework weiterentwickelt. Für diese Arbeit wird dem Ergebnis entsprechend erstmal mit Flink und der dazugehörigen DataSet API weitergearbeitet.

Kriterium	**Ergebnis**
Laufzeit	**Vorteil Flink**: zwei verschiedene Projekte haben gezeigt, dass Flink oftmals mit einer kürzeren Laufzeit aufwarten kann
Benutzerfreundlichkeit	**Vorteil Spark**: bietet bessere API für Python
Bibliotheken	**Vorteil Flink**: mehr Bibliotheken und Iterationsmodelle

Tabelle 5.2: Vergleich von *Apache Spark* und *Apache Flink*.

5.3 Stand der Forschung

Der letzte Punkt der für den Entwurf eines eigenen Algorithmus entscheidend war, ist der *Stand der Forschung* in diesem Bereich. In Unterkapitel 3.2 wurde schon erläutert, dass sich ein FSM-Algorithmus in die Phasen *Kandidatengenerierung* und *Signifikanzberechnung* unterteilen lässt. Beide sind essenziell für jeden FSM-Algorithmus und stellen gleichzeitig auch das Problem dar, weswegen es schwierig ist solche Algorithmen auf große Datenmengen zu skalieren (vgl. [QZL+18]):

- **Kandidatengenerierung**: Sei $N = |V|$, wobei $|V|$ die Anzahl der Knoten des Input-Graphen ist, so ergibt sich für die Kandidatengenerierung eine Komplexität von $O(2^{N^2})$.
- **Signifikanzberechnung**: Sei $N = |V|$ und $n = |V_S|$, wobei $|V|$ und $|V_S|$ die Anzahl der Knoten des Input- und des Teil-Graphen sind, so ergibt sich für die Signifikanzberechnung eine Komplexität von $O(N^n)$.

Der ganze FSM-Algorithmus hat also eine totale Komplexität von $O(2^{N^2} \times N^n)$, was exponentiell ist, wenn man den Suchraum nicht durch zusätzliche Heuristiken eingrenzt. Im nächsten Abschnitt (5.3.1) wird ein Überblick über vorhandene FSM-Algorithmen gegeben, bevor in Abschnitt 5.3.2 näher auf die einzelnen Probleme eingegangen wird, die im Zusammenhang mit dem *Freuquent-Subgraph-Mining* zu lösen sind.

5.3.1 FSM-Algorithmen

Es gibt viele Algorithmen die Heuristiken nutzen, um die Komplexität des Suchraums zu verkleinern. Aber nicht alle dieser Algorithmen sind auch darauf ausgelegt auf einem einzelnen Graphen als Input-Datensatz zu arbeiten. Ein Algorithmus, der dieses Szenario bedient und schon in Unterkapitel 3.3 vorgestellt wurde, ist z.B. *GraMi*. Ein weiterer Punkt ist die parallele Verarbeitung der Algorithmen. Hier existieren aktuell nicht viele Algorithmen, die diesem Kriterium begegnen. Ein Beispiel wäre *DistGraph* ([TZ16]). Allerdings läuft dieser Algorithmus nur auf *High-Performance-Computing*-Maschinen (kurz *HPC*), welche aber nur einem kleinen Teil von Programmierern wirklich zu Verfügung steht. Der aktuell einzige FSM-Algorithmus, der auf einem Cluster laufen kann und somit auch parallele Verarbeitung unterstützt, ist *DIMSpan* ([PJR17]). Allerdings ist dieser auf eine Menge von Daten als Input-Datensatz ausgelegt und somit für das Anwendungsszenario dieser Arbeit ungeeignet. In Tabelle 5.3 sind einige FSM-Algorithmen aufgelistet.
Es wird also deutlich, dass vorhandene Algorithmen nicht für das Anwendungsszenario dieser Arbeit geeignet sind. Der in dieser Arbeit entwickelte Algorithmus *PaSiGraM* kann diese Lücke füllen. Wie *PaSiGraM* im Detail funktioniert und welche vorhandenen Konzepte dabei genutzt wurden, wird in Kapitel 6 näher erläutert.

Algorithmus	**Input-Datensatz**	**Parallele Verarbeitung**
gSpan	Menge von Graphen	nein
Gaston	Menge von Graphen	nein
FSG	Menge von Graphen	nein
SIGRAM	einzelner Graph	nein
FFSM	einzelner Graph	nein
GraMi	einzelner Graph	nein
DIMSpan	Menge von Graphen	ja (auf Basis von Flink)
DistGraph	einzelner Graph	ja (aber nur auf *HPC*)
PaSiGraM	einzelner Graph	ja (auf Basis von Flink)

Tabelle 5.3: Verschiedene Algorithmen für das *Frequent-Subgraph-Mining* im Vergleich.

5.3.2 Probleme beim Frequent-Subgraph-Mining

In Unterkapitel 3.2 wurde schon erläutert, dass das *Frequent-Subgraph-Mining* mit mehreren NP-vollständigen Problemen einher geht. Dieser Abschnitt wird auf die einzelnen Schritte eines FSM-Algorithmus eingehen und zeigen welche Probleme bei der Implementierung und somit auch bei der Konzipierung zu lösen waren.

Kandidatengenerierung

Bei der *Kandidatengenerierung* treten unterschiedliche Probleme auf. Zum einen muss das *Isomorphie*-Problem (NP-vollständig) gelöst werden, welches schon in Abschnitt 3.2.4 näher erläutert wurde. Dadurch das Kandidaten mehrfach generiert werden könnten, muss für jeden Kandidaten überprüft werden, ob dieser schon generiert wurde oder ob es sich um einen neuen Kandidaten handelt. Zudem war die Strategie zur Generierung neuer Kandidaten so zu wählen, dass idealerweise auch alle möglichen Teilgraphen des Input-Graphen generiert werden. Dazu war abzuwiegen, welcher der beiden Ansätze (*Join* oder *Pattern-Growth*) gewählt werden sollte und was für Probleme der jeweilige Ansatz dann wieder mit sich bringen könnte. So ist der *Join* beim *Join*-basierten Ansatz eine sehr rechenaufwändige Operation. Beim *Pattern-Growth* hingegen muss geschaut werden, an welcher Stelle ein Kandidat um eine Kante erweitert werden kann, damit der Suchraum aller möglichen Erweiterungen eines Kandidaten nicht explodiert.

Signifikanzberechnung

Für die *Signifikanzberechnung* ist das *Isomorphie*-Problem ebenfalls zu lösen. Denn für jeden Kandidaten ist die Anzahl seiner Instanzen im Input-Graphen zu berechnen, um anschließend evaluieren zu können, ob es sich um einen häufigen Kandidaten handelt oder nicht. Dabei war darauf zu achten, wie man das *Teilgraph-Isomorphie*-Problem effizient lösen kann und wie man mit den Überlappungen von Instanzen eines Kandidaten im Input-Graphen umgeht.

Datenstruktur
Für einen FSM-Algorithmus ist darauf zu achten, welche Datenstruktur zur Repräsentation von Graphen genutzt wird. Dadurch, dass *PaSiGraM* auf Basis der DataSet API von Flink laufen soll, musste eine eigene Datenstruktur für Graphen kreiert werden. Das Ziel dabei sollte sein, dass möglichst wenig Informationen des Graphen verloren gehen und Operationen, wie ein *Join* oder die Erweiterung eines Kandidaten um eine Kante, auf Grundlage der Datenstruktur effizient mit den Operationen der DataSet API implementierbar sind. Ein weiterer Punkt, der die Datenstruktur betrifft, ist die eindeutige Identifizierung eines Graphen basierend auf seiner Struktur. In Abschnitt 3.2.4 wurde dazu schon kurz das *kanonische Markieren* besprochen.

5.4 Anforderungsliste

In den vorherigen Unterkapiteln (5.1 - 5.3) wurde das *Frequent-Subgraph-Mining* aus verschiedenen Perspektiven betrachtet, um Probleme und daraus resultierende Anforderungen identifizieren zu können, die bei der Entwicklung von *PaSiGraM* zu beachten waren. In Abschnitt 5.1 wurde dazu der Datensatz im Zusammenhang mit dem Anwendungsfall näher betrachtet. Daraus ist hervorgegangen, dass das Szenario eines einzelnen Graphen als Input-Datensatz zu bearbeiten ist. Der Vergleich zwischen *Apache Flink* und *Apache Spark* in Abschnitt 5.2 hat gezeigt, dass der Algorithmus zunächst auf Basis der DataSet API von Flink zu entwickeln war. In Abschnitt 5.3 wurde dann ein Überblick darüber gegeben, welche Algorithmen aktuell existieren und zu welchen Problemen es beim *Frequent-Subgraph-Mining* kommen kann. Daraus hat sich dann folgende Anforderungsliste ergeben:

1. Durch das Anwendungsszenario dieser Arbeit sollte das Konzept eines eigenen FSM-Algorithmus darauf ausgelegt sein, auf einem einzelnen Graphen als Input-Datensatz zu arbeiten.

2. Der FSM-Algorithmus sollte auf Basis von *Apache Flink* laufen. Somit ist dies auch bei der Konzipierung zu beachten. Zusätzlich kann das Konzept aber auch so aufgebaut werden, dass es mit beiden Plattformen (Flink und Spark) realisierbar ist.

3. Dadurch, dass die aktuell bereitgestellten Graph-Bibliotheken von Flink und Spark nicht für FSM-Algorithmen geeignet sind, ist bei der Konzipierung eines eigenen FSM-Algorithmus darauf zu achten, dass er mithilfe der DataSet API von Flink realisiert werden kann.

   (a) Die von der DataSet API bereitgestellten Operatoren zur Parallelisierung der Berechnungen sollten dabei bestmöglich ausgenutzt werden.

4. Es ist eine geeignete Datenstruktur zur Repräsentation von Graphen zu wählen, sodass diese mit der DataSet API umgesetzt werden kann.
   (a) Die Datenstruktur ist so zu wählen, dass möglichst wenig Informationsverlust besteht.
   (b) Operationen, die für das *Frequent-Subgraph-Mining* nötig sind, sollten auch weiterhin mit der gewählten Datenstruktur realisiert werden können.
   (c) Auf Grundlage der Datenstruktur sollte ein eindeutiger Identifikator für jeden Graphen generiert werden können, um somit der Isomorphie besser begegnen zu können.
5. Heuristiken zur Einschränkung von Suchräumen bei der *Kandidatengenerierung* und *Signifikanzberechnung* sind bei der Konzipierung zu berücksichtigen, damit die Performance des Algorithmus zusätzlich gesteigert werden kann.

Kapitel 6

PaSiGraM

In diesem Kapitel wird *PaSiGraM* vorgestellt, ein eigener paralleler FSM-Algorithmus. *PaSiGraM* bedeutet „**Pa**rallel-**S**ingle-**Gra**ph-**M**ining“. Dabei erfüllt der Algorithmus alle in Kapitel 5 identifizierten Anforderungen. In den folgenden Unterkapiteln wird eine detaillierte Betrachtung aller Teilschritte von *PaSiGraM* erfolgen und anhand von Beispielen erklärt, wie der Algorithmus die Teilprobleme vom *Frequent-Subgraph-Mining* löst. Dabei wurde sich an bestehenden Konzepten orientiert. Beispiele dafür sind *GraMi* ([EASK14]), *FSG* ([KK04]), *FFSM* ([HWP03]) oder auch *SiGraM* ([KK05]). Während der Überlegungen zu *PaSiGraM*, ist mit *SSiGraM* ([QZL+18]) ein FSM-Algorithmus erschienen, der die gleichen Ziele wie *PaSiGraM* verfolgt. Durch die gleiche Ausrichtung beider Algorithmen überschneiden sie sich in manchen Teilen. Ein detailliertere Betrachtung der Gemeinsamkeiten und Unterschiede erfolgt in den späteren Unterkapiteln.

6.1 Allgemeiner Ansatz

PaSiGraM besteht wie die meisten FSM-Algorithmen im Allgemeinen aus zwei Teilen:

- Kandidatengenerierung
- Signifikanzberechnung

Beide Teilschritte sollten möglichst gut parallelisiert werden, um die Anzahl an zur Verfügung stehenden Knoten in einem Cluster ausnutzen zu können. Deswegen ist darauf zu achten, dass möglichst wenig sequenzielle Abläufe in dem Algorithmus vorhanden sind, welche sich aber natürlich nicht vollständig umgehen lassen. Um dies dennoch gewährleisten zu können, arbeitet der Algorithmus nach dem Prinzip der *Breitensuche* (kurz BFS). Das bedeutet, dass die *Kandidatengenerierung*, wie auch die *Signifikanzberechnung Level-basiert* arbeiten. In Abb. 6.1 ist die grobe Struktur von *PaSiGraM* zu sehen. Die blauen Kästen stellen dabei die jeweiligen Datenstrukturen dar, in denen die Graphen/Teilgraphen abgespeichert werden. Bei den gelben Kreisen

handelt es sich um Funktionen, die innerhalb einer DataSet-Operation (z.B. *flatMap*) ausgeführt werden. Die beiden grünen Kästen sind die zwei Hauptbestandteile: *Kandidatengenerierung* und *Signifikanzberechnung*. Auf die einzelnen Elemente dieser Abbildung wird später genauer eingegangen.

Wichtig ist an dieser Stelle zunächst nur, dass die *Kandidatengenerierung* zunächst alle k-Teilgraphen generiert, wobei k hier die Anzahl der Kanten eines Graphen ist. Diese k-Teilgraphen werden dann an die *Signifikanzberechnung* übergeben, welche evaluiert ob die jeweiligen k-Teilgraphen häufig sind. Alle häufigen k-Teilgraphen werden dann in die Ergebnismenge aufgenommen. Anschließend, wenn die Abbruchbedingung noch nicht erfüllt ist, wird der Datensatz mit allen häufigen k-Teilgraphen wieder an die *Kandidatengenerierung* übergeben, die mithilfe dieses Datensatzes die neuen k+1-Teilgraphen generiert und wieder an die *Signifikanzberechnung* übergibt. Das wiederholt sich dann so oft, bis in einem Iterationsschritt keine weiteren häufigen Teilgraphen mehr gefunden werden können.

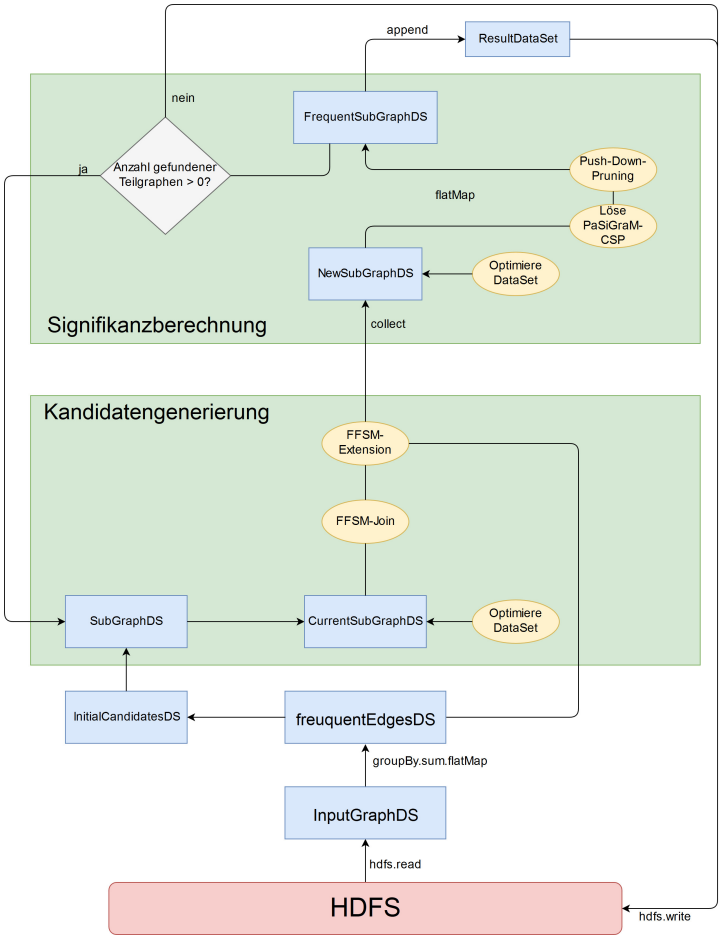


Abbildung 6.1: Eine vereinfachte Struktur des *PaSiGraM*-Algorithmus.

6.2 Datenstruktur

In diesem Abschnitt wird erklärt, in welcher Datenstruktur ein Graph abgelegt wird. Dazu wird das *CAM*-Modell (engl. *Canonical Adjacency Matrix*) von *Huan et al.* ([HWP03]) übernommen. Gemäß Definition 3.9 kann jeder Graph als *Adjazenzmatrix* dargestellt werden. In Kapitel 3.2.4 wurde schon erklärt, dass sich jedem Graph ein eindeutiger Code auf Grundlage seiner *Adjazenzmatrix* zuweisen lässt.

Definition 6.1 (Code einer Adjazenzmatrix (vlg. [QZL+18])). *Gegeben sei eine $n \times n$ Adjazenzmatrix M eines Graphen G mit n Knoten, dann wird der Code von M, bezeichnet als $Code(M)$, als eine Sequenz von Einträgen aus M generiert, indem die Einträge der Matrix von links nach rechts und von oben nach unten konkateniert werden. Im Fall einer symmetrischen Matrix gilt das Selbe, allerdings müssen nur die Einträge der unteren Dreiecksmatrix (einschließlich der Diagonale) konkateniert werden.*

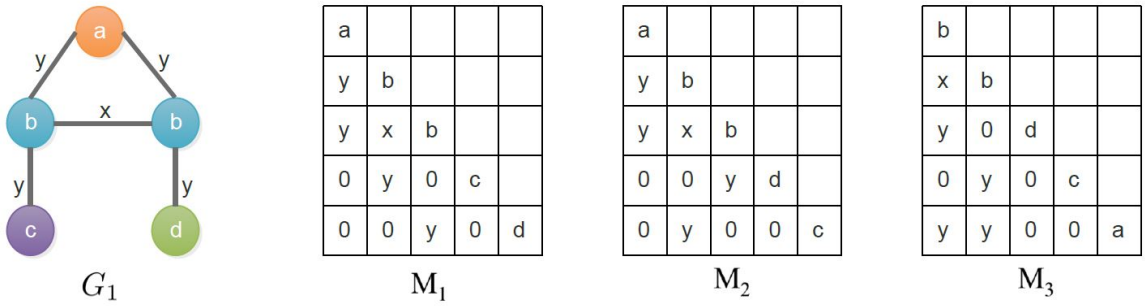


Abbildung 6.2: Ein Graph mit seinen unterschiedlichen Adjazenzmatrizen (Quelle: [QZL+18]).

Wie in Abb. 6.2 zu sehen, lassen sich aus einem Graphen allerdings unterschiedliche *Adjazenzmatrizen* generieren, die dann schließlich auch zu unterschiedlichen Codes führen würden. Der eindeutige Code, der einem Graphen dann zugewiesen werden würde, wäre der lexikographisch größte Code, der auch als *kanonische Form* eines Graphen bezeichnet wird. Zu erwähnen ist hier, dass aus den *Adjazenzmatrizen* von Graphen, die *isomorph* zueinander sind, auch die gleichen lexikographisch größten Codes generiert werden würden. Die *Adjazenzmatrix*, die dann zur *kanonischen Form* eines Graphen führt, wird im Folgenden als *CAM* bezeichnet.

Definition 6.2 (Canonical Adjacency Matrix (CAM)). *Die CAM eines Graphen G ist die Adjazenzmatrix, die den lexikographisch größten Code für G produziert.*

In dem Beispiel aus Abb. 6.2 würde die *Adjazenzmatrix* M_1 zum lexikographisch größten Code führen (M_1 : $aybyxb0y0c00y0d$ > M_2 : $aybyxb00yd0y00c$ > M_3 : $bxby0d0y0cyy00a$). Jedoch

können für einen Graphen mit $|V|$ Knoten, wobei $|V|$ die Anzahl der Knoten vom Graphen ist, $|V|!$-verschiedene Adjazenzmatrizen generiert werden, wodurch man eine Komplexität von $O(|V|!)$ hätte, um den lexikographisch größten Code in allen Permutationen zu finden (vgl. [KK04]).
Es wird also ein Verfahren benötigt, mit dem die Anzahl an Permutationen der *Adjazenzmatrix* eines Graphen minimiert werden kann, wodurch es leichter ist einen eindeutigen Code für jeden Graphen zu generieren. *Kuramochi et al.* [KK04] haben einen eigenen Algorithmus für das *kanonische Markieren* entwickelt, der auch hier angewendet wird. Dazu werden drei Heuristiken benutzt:

- Knoten-Invarianten
- Kanten-basierte Ordnung von Partitionen
- Knoten-Stabilisierung

In diesem Punkt unterscheidet sich *PaSiGraM* von dem vorher schon erwähnten *SSiGraM*-Algorithmus ([QZL+18]). *Qiao et al.* benutzen keine weitere Heuristik, um den Suchraum für die *CAM* eines Graphen einzuschränken. Dadurch rechnet *SSiGraM* mit einem NP-vollständigen Problem, was die Performance des Algorithmus negativ beeinflussen sollte. Im Folgenden werden die drei Heuristiken im Detail vorgestellt.

6.2.1 Knoten-Invarianten

Eine *Knoten-Invariante* ist eine Eigenschaft von Knoten, die sich durch die Isomorphie-Zuweisung nicht verändert. Beispiele für *Knoten-Invarianten* sind dann der *Knotengrad* oder das *Label* eines Knotens. Diese *Invarianten* können dann genutzt werden, um die Knoten in *Äquivalenzklassen* (im Folgenden werden die Begriffe *Äquivalenzklasse* und *Partition* synonym benutzt) zu unterteilen. Alle Knoten, die die gleichen Werte einer *Invariante* besitzen, werden derselben Partition zugeordnet. Diese Partitionen können schließlich dazu genutzt werden, um den lexikographisch größten Code schneller zu erzeugen. Anstatt wie bisher den maximalen Code unter allen Permutationen über alle Knoten zu suchen, wird nun nur noch in der Menge der Permutationen gesucht, die die Knoten einer Partition zusammenhalten. Zu erwähnen ist an dieser Stelle auch, dass Graphen, die *isomorph* zueinander sind, auch die gleichen Partitionen für ihre Knoten besitzen werden. In dem Algorithmus von *Kuramochi et al.* ([KK04]) werden drei Typen von *Invarianten* genutzt:

- **Typ 1: Knotengrad und Labels**: Hier werden die Knoten in *disjunkte* Partitionen zerlegt, sodass jede Partition Knoten mit gleichen Label und Knotengrad enthält. In Abb. 6.3(b) ist eine solche Partitionierung zu sehen.

- **Typ2 : Adjazenzliste**: Eine weitere Invariante nach der partitioniert werden kann, ist die *Adjazenzliste* von Knoten. Dabei wird die *Adjazenzliste* eines Knotens v durch ein Tupel $l(e), d(v), l(v)$ beschrieben, wobei $l(e)$ das Label der Kante, $d(v)$ der Grad des Knotens und $l(v)$ das Label des Knotens sind. Für jeden Knoten u wird dann eine Liste $nl(u)$ erstellt, welche die Tupel für alle von u adjazenten Knoten enthält (siehe Abb. 6.3(d)). Diese Liste kann dann genutzt werden, um Knoten in *disjunkte* Partitionen zu zerlegen, sodass zwei Knoten u und v in einer Partition sind, wenn $nl(u) = nl(v)$. Diese Invariante wird genutzt, um die von Typ 1 erstellten Partitionen noch weiter zu partitionieren. In Abb. 6.3(c) ist dies zu sehen.

- **Typ 3: Iterative Partitionierung**: Hier wird die Idee von *Adjazenzlisten* generalisiert. Anstatt eines Tupels $(l(e), d(v), l(v))$ nutzt man hier ein Paar $(P(v), l(e))$, um die Knoten in verschiedene Partitionen zu unterteilen. $P(v)$ ist dabei der Name der Partition, welcher der adjazente Knoten zugeordnet ist. Auch hier gilt, dass zwei Knoten u und v in einer Partition sind, wenn $nl(u) = nl(v)$. Diese Art der Partitionierung kann nützlich sein, wenn man einen Graphen hat, dessen Knoten und Kanten oftmals das gleiche Label besitzen.

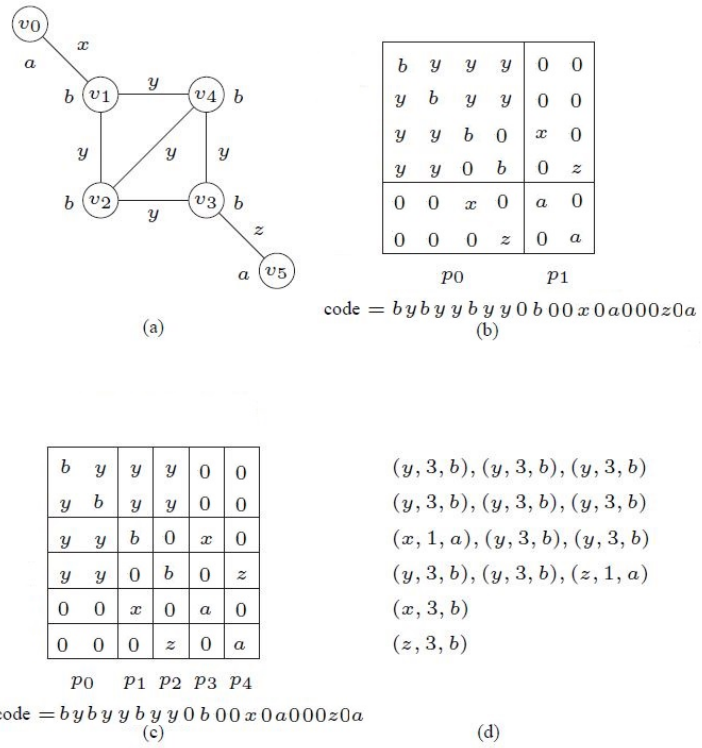


Abbildung 6.3: Die Partitionierung der *Adjazenzmatrix* eines Graphen anhand des *Knotengrades* und der *Labels* (b) und der *Adjazenzliste* seiner Knoten (c)(d) (vgl. [KK04]).

6.2.2 Kanten-basierte Ordnung von Partitionen

Zusätzlich zu *Knoten-Invarianten*, die die Knoten in Partitionen unterteilt, kann die Anzahl der Permutationen noch weiter eingeschränkt werden, indem die verschiedenen Partitionen sortiert werden. Das erlaubt einem dann frühzeitig zu erkennen, ob eine Menge von Permutationen zu einem lexikographisch kleineren als dem aktuelle Code führt oder nicht. Dadurch kann man den Suchraum weiter eingrenzen. Um dies zu erreichen, werden die Partitionen in einer absteigenden Reihenfolge, anhand der Anzahl ihrer Kanten, sortiert (vgl. [KK04]).

6.2.3 Knoten-Stabilisierung

Die *Knoten-Stabilisierung* ist nützlich, um Isomorphismen in Graphen mit regulären oder symmetrischen Strukturen zu finden. Die Kernidee dabei ist, dass man die topologischen Symmetrien aufbricht, indem man einen bestimmten Knoten in eine eigene Partition einordnet. Durch die *iterative Partitionierung* werden sich die Partitionen der restlichen Knoten dann ändern, da einer ihrer adjazenten Knoten nun in einer anderen Partition ist (vgl. [KK04]).

6.2.4 Eigenschaften der CAM

Eine Kerneigenschaft der *kanonischen Form* eines Graphen ist, dass der Präfix der *kanonischen Form* ebenso maximal ist. Daraus ergibt sich folgender Sachverhalt (vgl. [HWP03]):

Satz 6.1. *Seien ein Graph G und ein Teilgraph H von G gegeben und lass A und B die CAMs von G und H sein, dann gilt $code(A) \geq code(B)$.*

Das Konzept der *maximal korrekten Teilmatrix* ist für den weiteren Verlauf dieser Arbeit noch von Bedeutung, weswegen es an dieser Stelle genauer definiert wird.

Definition 6.3 (Maximal korrekte Teilmatrix (vgl. [HWP03]))**.** *Gegeben sei eine $n \times n$-Matrix N und eine $m \times m$-Matrix M. $m_{l,k}$ ist der letzte Eintrag von M und es gilt die Annahme, dass M zwei Kanteneinträge in der letzten Zeile hat. N wird als maximal korrekte Teilmatrix von M bezeichnet, wenn Folgendes gilt:*

1. $n = m$ *und*
2. $n_{i,j} = \begin{cases} m_{i,j} & 0 < i,j \leq n \wedge (i \neq l \wedge j \neq k) \wedge (i \neq k \wedge j \neq l) \\ 0 & sonst \end{cases}$

Wenn M nur einen Kanteneintrag in der letzten Zeile hat, dann ist N die maximal korrekte Teilmatrix von M, wenn

1. $n = m - 1$ *und*
2. $n_{i,j} = m_{i,j}$ *mit* $(0 < i,j \leq n)$

gilt.

6.3 Kandidatengenerierung

In diesem Kapitel wird genauer erklärt, wie *PaSiGraM* neue Kandidaten generiert. Aus Abschnitt 3.2.3 ist schon bekannt, dass es zwei Ansätze gibt, um potenziell häufige Teilgraphen zu generieren. Dabei nutzen sowohl der *Pattern-Growth*, wie auch der *Join-basierte* Ansatz die *Anti-Monotonie*-Eigenschaft (Definition 3.4) aus, um den Suchraum eingrenzen zu können. Bei *PaSiGraM* wird eine Kombination aus beiden Ansätzen genutzt, um die Kandidaten zu generieren. Es wird auf spezielle Operatoren zurückgegriffen, die die Darstellung eines Graphen als Adjazenzmatrix nutzen, um dadurch neue Kandidaten generieren zu können. Dabei handelt es sich um die Operatoren *FFSM-Join* und *FFSM-Extension* von *Huan et al.* ([HWP03]).

Algorithmus 5: GenerateSubgraphs

Input: Eine Menge häufiger Teilgraphen $FSGraph_{k-1}$
Output: Eine neu generierte Menge $SGraph_k$ die $FSGraph_{k-1}$ enthält
$SGraph_k \leftarrow \emptyset$
$SGraph_k \leftarrow FSGraph_{k-1}$.**map**{
$S_t \leftarrow \emptyset$
$C_{list} \leftarrow$ Kinder-Teilgraphen des jeweiligen Eltern-Teilgraphen
for s_i *in* C_{list} **do**
  for s_j *in* C_{list} **do**
    $tmp \leftarrow$ **FFSM-Join**(s_i, s_j)
    Erstelle CAM von tmp
    $S_t \leftarrow S_t \cup tmp$
  end
  $tmp \leftarrow$ **FFSM-Extension**(s_i)
  Erstelle CAM von tmp
  $S_t \leftarrow S_t \cup tmp$
end
return S_t
}.**collect**
Berechne neue Indexe für alle Teilgraphen
return $SGraph_k$

In Algorithmus 5 (vgl. [QZL+18]) ist der Pseudocode für die Kandidatengenerierung zu sehen. Zunächst wird ein leerer Datensatz ($SGraph_k$) erzeugt, in dem später alle neu generierten Kandidaten enthalten sein werden. Anschließend wird über den Input-Datensatz ($FSGraph_{k-1}$) eine *Map-Function* formuliert, in der die beiden Operatoren *FFSM-Join* und *FFSM-Extension* ausgeführt werden und als Resultat die Menge der neu generierten Teilgraphen produziert und in $SGraph_k$ abspeichert. Auf die beiden Operatoren *FFSM-Join* und *FFSM-Extension* wird in den folgenden Abschnitten genauer eingegangen. Schließlich werden alle Teilgraphen in $SGraph_k$ mit

einem neuen Index versehen, da jeder Knoten im Cluster eine eigene Indexierung für seinen Teildatensatz erstellt und ansonsten nach der Zusammenführung aller Teildatensätze zu einem einzigen Datensatz, die Indexstruktur unbrauchbar wäre.

6.3.1 Suboptimaler CAM-Baum

In Unterkapitel 6.2 wurde festgelegt, dass jeder Graph durch seine *CAM* repräsentiert wird. Die Menge der *CAMs* aller häufigen Teilgraphen wird dann in einer Baumstruktur organisiert, um Informationen zu den Eltern-Graphen eines neuen Teilgraphen abspeichern zu können. Dazu werden folgende Bedingungen an den *CAM-Baum* gestellt:

- Die Wurzel des Baumes ist eine leere Matrix.
- Jeder Knoten im Baum ist ein *distinkter* verbundener Teilgraph von G, der durch seine *CAM* repräsentiert wird.
- Für einen gegebenen Knoten N (mit seiner *CAM*), ist sein Eltern-Knoten der Teilgraph von G, der durch die *maximal korrekte Teilmatrix* von N repräsentiert wird.

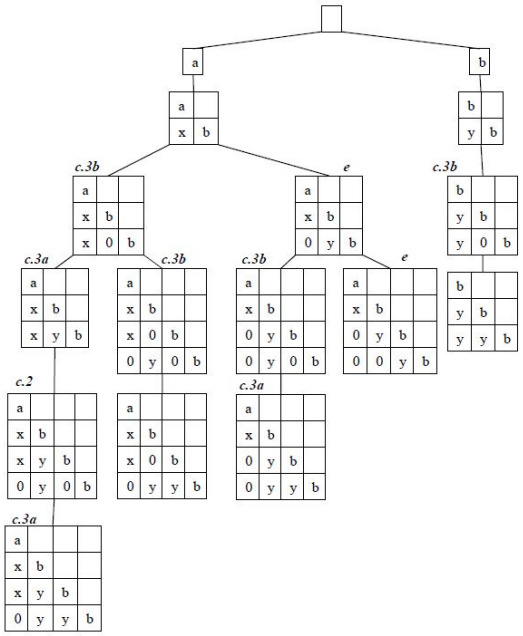


Abbildung 6.4: Ein Beispiel für einen *CAM-Baum* (Quelle: [HWP03]).

In Abb. 6.4 ist ein Beispiel für einen *CAM-Baum* zu sehen. Jeder Knoten ist die *CAM* des jeweiligen Teilgraphen und mit einer Bezeichnung versehen, welche durch die Art der Generierung (*FFSM-Join* oder *FFSM-Extension*) bestimmt wird. So ist zum Beispiel die *CAM* mit der

Bezeichnung e durch *FFSM-Extension* entstanden. Jedoch können durch die Kombination der beiden Operationen nicht alle möglichen Teilgraphen erzeugt werden. Deswegen wird der *CAM-Baum* um *suboptimale CAMs* erweitert.

Definition 6.4 (Suboptimale CAM (vgl. [HWP03]))**.** *Gegeben sei ein Graph G. Die suboptimale CAM von G ist eine Adjazenzmatrix M von G, sodass ihre maximal korrekte Teilmatrix N (Def. 6.3) die CAM des Graphen ist, welcher durch N repräsentiert wird.*

Daraus resultiert dann ein neuer Baum: der *suboptimale CAM-Baum*. Dieser wird als Datenstruktur genommen, um alle Teilgraphen zu organisieren. Zusammen mit den beiden Operationen *FFSM-Join* und *FFSM-Extension* kann man nun alle möglichen Teilgraphen generieren. In Abb. 6.5 ist ein solcher Baum zu sehen. Bei den gestrichelten Matrizen handelt es sich um die *suboptimalen CAMs*. Wenn man beide Abbildungen (6.4, 6.5) vergleicht, sieht man, dass der *suboptimale CAM-Baum* eine Erweiterung des *CAM-Baums* ist. Der Beweis, dass wirklich alle Teilgraphen im *suboptimalen CAM-Baum* enthalten sind, wird an dieser Stelle nicht erläutert. Stattdessen wird auf *Huan et al.* ([HWP03]) verwiesen, der den Beweis in seinem Paper genauer erklärt.

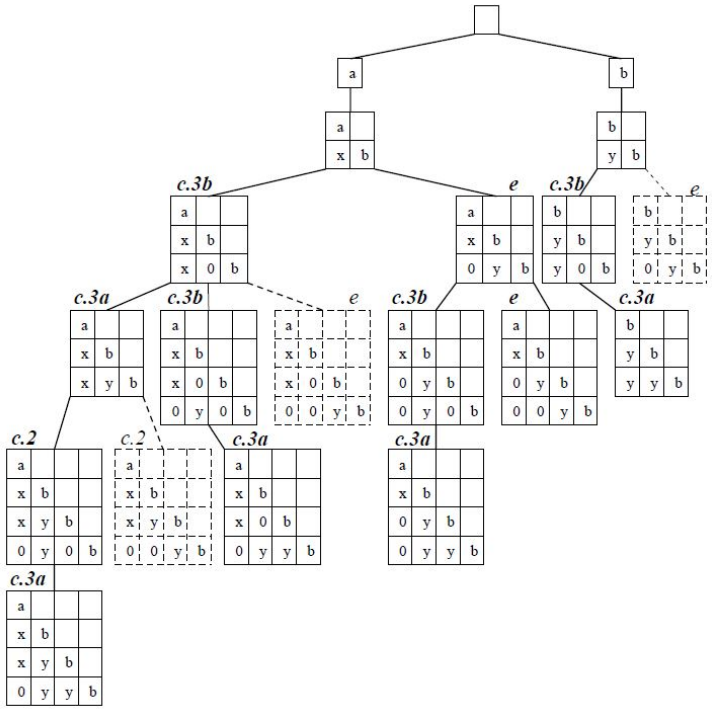


Abbildung 6.5: Ein Beispiel für einen *suboptimalen CAM-Baum* (Quelle: [HWP03]).

6.3.2 FFSM-Join

Der *FFSM-Join* ist eine von zwei Operationen, die genutzt werden, um neue Teilgraphen zu generieren. Wie eingangs schon erwähnt, handelt es sich dabei um eine Operation auf den *CAMs* zweier häufiger Teilgraphen. Bevor der *FFSM-Join* weiter erklärt werden kann, sind noch zwei weitere Begriffe einzuführen: *innere* und *äußere* Matrix (vgl. [HWP03]).

Definition 6.5 (Innere Matrix)**.** *Eine Matrix A wird als innere Matrix bezeichnet, wenn A mindestens zwei Kanteneinträge in der letzten Zeile hat.*

Definition 6.6 (Äußere Matrix)**.** *Eine Matrix A wird als äußere Matrix bezeichnet, wenn A weniger als zwei Kanteneinträge in der letzten Zeile hat.*

Darauf aufbauend wird in Algorithmus 6 der *FFSM-Join* definiert (vgl. *Huan et al.* [HWP03]). Hierbei sind drei Fälle zu unterscheiden:

- **Fall 1**: A und B sind *innere* Matrizen
- **Fall 2**: A ist eine *innere* und B eine *äußere* Matrix
- **Fall 3**: A und B sind *innere* Matrizen

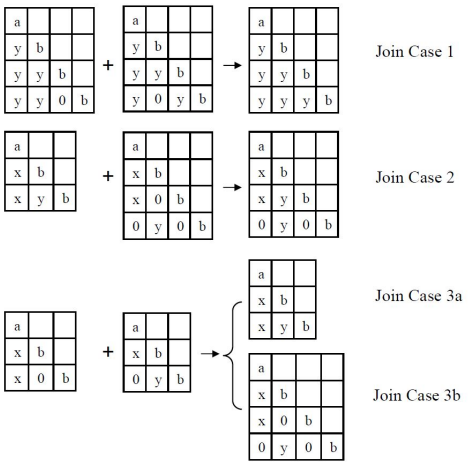


Abbildung 6.6: Beispiele für die unterschiedlichen Fälle des *FFSM-Join* (Quelle: [HWP03]).

In Abb. 6.6 sind Beispiele für jeden einzelnen Fall des *FFSM-Join*-Operators aufgeführt. Für alle 3 Fälle gilt $join(A, B) = join(B, A)$, da *FFSM-Join* zwei Ajazenzmatrizen A und B nur dann verarbeitet, wenn A und B dieselbe *maximal korrekte Teilmatrix* haben. *Fall 3b* ist allerdings eine Ausnahme. Hier gilt $join(A, B) \neq join(B, A)$.

Algorithmus 6: FFSM-Join

Input: Zwei Adjazenzmatrizen $A(m \times m)$ und $B(n \times n)$, die sich dieselbe *maximal korrekte Teilmatrix* teilen

Seien $a_{m,f}$ und $b_{n,k}$ jeweils die letzte Kante von A und B

if *Fall 1: A und B sind innere Matrizen* **then**

  if $f \neq k$ **then**

    $join(A, B) = \{C\}$, wobei C eine $m \times m$-Matrix ist

    und $c_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m, i \neq n \wedge j \neq k \\ b_{i,j} & sonst \end{cases}$

  end

  else

    $join(A, B) = \Phi$

  end

end

else if *Fall 2: A ist eine innere und B eine äußere Matrix* **then**

  $join(A, B) = \{C\}$, wobei C eine $n \times n$-Matrix ist

  und $c_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m \\ b_{i,j} & sonst \end{cases}$

end

else if *Fall 3: A und B sind innere Matrizen* **then**

  Lass D eine $(m+1) \times (m+1)$-Matrix sein (Fall 3b)

  und $d_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m \\ b_{m,j} & i = m+1, 0 < j < m \\ 0 & i = m+1, j = m \\ b_{m,m} & i = m+1, j = m+1 \end{cases}$

  if $f \neq k \wedge a_{m,m} = b_{m,m}$ **then**

    C ist eine $m \times m$-Matrix (Fall 3a)

    und $c_{i,j} = \begin{cases} a_{i,j} & 0 < i, j \leq m, i \neq n \wedge j \neq k \\ b_{i,j} & sonst \end{cases}$

    $join(A, B) = \{C, D\}$

  end

  else

    $join(A, B) = \{D\}$

  end

end

6.3.3 FFSM-Extension

Der *FFSM-Join* alleine reicht allerdings nicht aus, um alle möglichen Teilgraphen für einen *supotimalen CAM-Baum* zu generieren. Deswegen gibt es mit *FFSM-Extension* einen zweiten Operator mit dem neue Teilgraphen generiert werden können. Auch hier wird die *CAM* eines häufigen Teilgraphen genutzt, um einen neuen Teilgraphen zu erzeugen, welcher ebenfalls wieder durch seine *CAM* im *suboptimalen CAM-Baum* repräsentiert wird. Dabei wird ein Teilgraph mit k Kanten, um eine neue Kante erweitert, sodass ein neuer Teilgraph mit $k+1$ Kanten entsteht. Dabei kann es sich entweder um den Fall handeln, dass eine Kante zwei schon existierende Knoten miteinander verbindet oder mit der neuen Kante auch ein neuer Knoten eingeführt wird. Ein naiver Ansatz diese Operation umzusetzen wäre, jeden Knoten im ursprünglichen Teilgraphen G um jede mögliche Kante zu erweitern. Das würde aber in einer Komplexität von $O(\Sigma_V \times \Sigma_E \times |G|)$, wobei Σ_E und Σ_V für die Menge von verfügbaren Labels von Kanten und Knoten stehen, resultieren (vgl. [HWP03]). Deswegen nutzt *FFSM-Extension* an dieser Stelle die Idee der *right-most-extension*, welche in Unterkapitel 3.2.3 schon angeschnitten wurde.

Definition 6.7 (Ganz rechter Pfad (Right-Most-Path))**.** *Gegeben sei ein Graph G mit seinem Tiefensuche-Baum $T(G)$. Der ganz rechte Pfad von G ist dann der ganz rechte Pfad in seinem Tiefensuche-Baum $T(G)$.*

Bei der *right-most-extension* wird dann der Knoten eines Graphen G um eine Kante erweitert, der sich im *ganz rechten Pfad* von G befindet. Daraus resultiert dann folgender Algorithmus für *FFSM-Extension* (vgl. *Huan et al.* [HWP03]):

Algorithmus 7: FFSM-Extension

Input: Eine $n \times n$-Adjazenzmatrix A
Output: eine Menge S von Adjazenzmatrizen B
if *A ist eine äußere Adjazenzmatrix* **then**
  for $(n_l, e_l) \in \sum_V \times \sum_E$ **do**
    Erstelle eine $n \times n$-Matrix B
    mit $b_{i,j} = \begin{cases} a_{i,j} & 0 < i,j \leq n \\ 0 & i = n+1, 0 < j < n \\ e_l & i = n+1, j = n \\ v_l & i = n+1, j = n+1 \end{cases}$
    $S \leftarrow S \cup \{B\}$
  end
end
else
  $S \leftarrow \Phi$
end

6.4 Signifikanzberechnung – das PaSiGraM-CSP

In Unterkapitel 3.3 wurde *GraMi* vorgestellt, ein Algorithmus für das FSM auf einem einzelnen Graphen. *GraMi* nutzt dabei ein *CSP*, um das Isomorphie-Problem zu lösen. Dieser Ansatz wird auch bei *PaSiGraM* verwendet. Für den theoretischen Hintergrund von *CSPs* und wie sie in *GraMi* Anwendung finden, wird nochmals auf das Unterkapitel 3.3 verwiesen. Im Folgenden wird in Algorithmus 8 (vgl. [QZL+18]) der Pseudocode für die Signifikanzberechnung bereitgestellt.

Algorithmus 8: CalculateSupport

```
Input: Ein Teilgraph s, ein Domänen-Datensatz D_s, eine Mindesthäufigkeit θ
Output: true wenn s häufig ist, false wenn nicht
for Variable v in D_s do
    N(v) ← Nachbarknoten von v
    if N(v).size > 1 then
        for Element u in D_s(v) do
            Entferne redundante u
        end
    end
end
if D_s.size < θ then
    return false
end
for Variable v in D_s do
    count ← 0
    for Element u in D_s(v) do
        if u ist schon markiert then
            count++
        end
        else if es existiert eine Lösung die u zu v zuweist then
            Markiere die entsprechenden Werte in D_s
            count++
        end
        else
            Entferne u aus v's Domäne in D_s
        end
        if count = θ then
            Mache mit der nächsten Variable v weiter
        end
    end
end
return true
```

Da jeder Graph durch seine *CAM* repräsentiert wird, ist das *FSM-CSP* so zu modellieren, dass es mithilfe der *CAM* des jeweiligen Kandidaten und des Input-Graphen gelöst werden kann. An dieser Stelle wird nochmal daran erinnert, wie das *FSM-CSP* definiert ist (Definition 3.19). *PaSiGraM* verwendet allerdings eine andere Definition für das *FSM-CSP*, als es bei *GraMi* der Fall ist.

Definition 6.8 (PaSiGraM-CSP). *Sei $G_s = (V_s, E_s, L_s)$ ein Teilgraph vom Graph $G = (V, E, L)$, $l(v) \in L$ das Label vom Knoten $v \in V$ und $l(v_s) \in L_s$ das Label vom Knoten $v_s \in V_s$. Sei $nl(v) = \{(l(e), nv)\}$ die Adjazenzliste eines Knotens $v \in V$ und $nl(v_s) = \{(l(e_s), nv_s)\}$ die Adjazenzliste eines Knotens $v_s \in V_s$, wobei $l(e)$ bzw. $l(e_s)$ die Label der Kanten $e \in E, e_s \in E_s$ und nv bzw. nv_s die Nachbarknoten, die über die Kanten e bzw. e_s mit v, v_s verbunden sind. Dann ist das PaSiGraM-CSP, ein $CSP = (X, D, C)$ für das gilt:*

1. *X enthält Variablen x_v für jeden Knoten $v_s \in V_s$.*
2. *D ist eine Menge von Domänen für jede Variable $x_v \in X$. Jede Domäne ist eine Teilmenge von V.*
3. *Seien in C die folgenden Bedingungen enthalten:*
   (a) *$x_v \neq x'_v$, für alle $x_v, x'_v \in X$*
   (b) *$l(v) = l(v_s)$*
   (c) *$nl(v_s) \subseteq nl(v)$*

Die Variablen sind die Knoten des jeweiligen Kandidaten und die Domänen der Variablen sind die Knoten des Input-Graphen. Die Lösung des *PaSiGraM-CSP* besteht dann darin, für jede Variable gültige Belegungen in ihrer Domäne zu finden, ohne die Bedingung in C (engl. *constraint*) zu verletzen. Dabei macht *PaSiGraM* sich spezielle Eigenschaften der *CAM* zunutze, die durch die Heuristiken, welche in Unterkapitel 6.2 erklärt wurden, entstehen. So sind die Knoten jeder *CAM* in verschiedene Partitionen unterteilt, die unter anderem auf dem Knotengrad und dem Label der Knoten in der jeweiligen Partition beruhen. Sucht man nun nach einer gültigen Belegung für den Knoten eines Kandidaten, muss man nur in den Partitionen der *CAM* vom Input-Graphen schauen, die Knoten beinhalten deren Knotengrad größer oder gleich dem Knotengrad des Knotens ist, für den eine gültige Belegung gesucht wird. Um den Suchraum noch weiter zu verkleinern, können weiterhin alle Partitionen ignoriert werden, die Knoten mit einem anderen Label als dem Label des zu belegenden Knotens beinhalten. Abbildung 6.7 enthält ein Beispiel, welches im Folgenden näher erklärt wird.

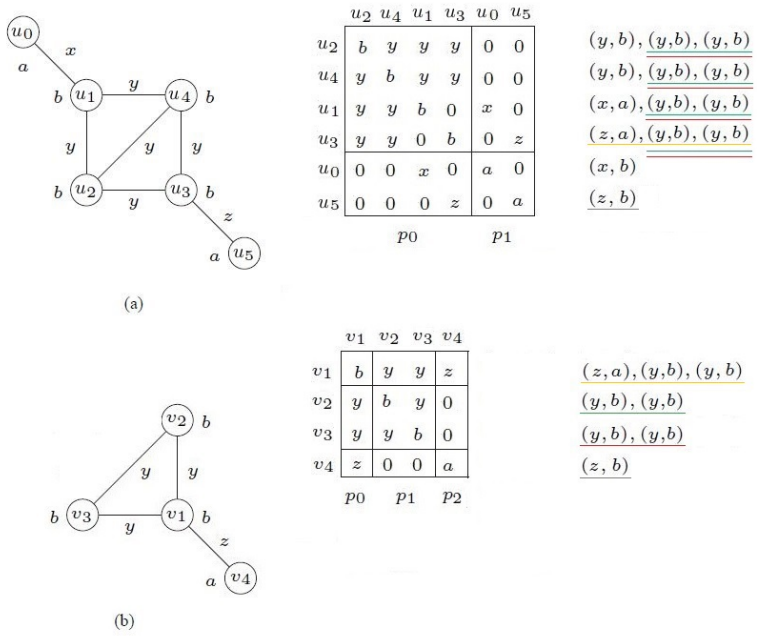


Abbildung 6.7: Input-Graph (a) und Teilgraph (b) mit ihren *CAMs*.

Zu sehen ist ein Input-Graph (a) und ein Teilgraph (b) vom Input-Graphen. Die beiden *Adjazenzmatrizen* der Graphen sind jeweils rechts daneben zu sehen. Ganz rechts in der Abbildung ist für jeden Knoten des jeweiligen Graphen die *Adjazenzliste* zu sehen. Wenn man nun das *PaSiGraM-CSP* für diese Abbildung modelliert, würde man Folgendes bekommen:

Beispiel 6.8.1 (PaSiGraM-CSP).

$$\left(\begin{array}{c}(v_1,v_2,v_3,v_4),\{\{u_o,...,u_5\},...,\{u_0,...,u_5\}\},\\ \{v_1\neq v_2\neq v_3\neq v_4, l(v_1)=l(v_2)=l(v_3)=b, l(v_4)=a\\ nl(v_1)=\{(z,a),(y,b),(y,b)\}, nl(v_2)=nl(v_3)=\{(y,b),(y,b)\}, nl(v_4)=\{(z,b)\}\}\end{array}\right)$$

So muss bei der Suche nach einer gültigen Belegung für den Knoten v_4 vom Teilgraph (b) nur in der Partition p_1 der *CAM* des Input-Graph (a) gesucht werden. Alle anderen Partitionen beinhalten nur Knoten mit anderen Label als a. Eine gültige Belegung ist dann gefunden, wenn alle Elemente der Adjazenzliste des Knotens, sich in der Adjazenzliste des jeweiligen Knotens im Input-Graph wiederfinden. In der Abbildung sind die gültigen Belegungen durch die farbigen Striche in den Adjazenzlisten der jeweiligen Knoten gekennzeichnet. Die Signifikanz entspricht dann der Mächtigkeit der kleinsten Domänen in D. In der Abbildung 6.7 hätten wir also eine Signifikanz von 1, da für den Knoten v_4 mit u_5 nur eine gültige Belegung im Input-Graphen existiert. Das entspricht dann der *MNI*-Signifikanz aus Definition 3.15. Dadurch das *PaSiGraM*

das *CSP* auf Basis der *CAM* von Graphen modelliert, unterscheidet sich der Algorithmus auch hier von *SSiGraM* und allen weiteren Algorithmen, die im Laufe dieser Arbeit erwähnt wurden. Das *PaSiGraM-CSP* ist also eine komplett neue Herangehensweise, das *Isomorphie*-Problem mithilfe eines *CSP* zu lösen. Zudem kommen bei der Lösung des *PaSiGraM-CSP* weitere Heuristiken zum Tragen, die bei *SSiGraM* keine Anwendung finden. Dadurch grenzt *PaSiGraM* den Suchraum für mögliche Lösungen des *PaSiGraM-CSP* ein und operiert nicht einfach auf dem kompletten Suchraum, so wie es bei *SSiGraM* der Fall ist. Um den Suchraum noch kleiner zu machen wird das *Push-Down-Pruning* von *GraMi* übernommen, welches in Abschnitt 3.3.3 schon näher erläutert wurde.

6.5 Optimierung von Kandidatengenerierung und Signifikanzberechnung

Zwischen den beiden Teilschritten *Kandidatengenerierung* und *Signifikanzberechnung* müssen die neuen Kandidaten immer an die jeweils andere Prozedur gesendet und somit der Datensatz, welcher die Kandidaten beinhaltet, immer wieder neu auf die verschiedenen Knoten eines Clusters aufgeteilt werden, um die beiden Teilschritte jeweils parallel berechnen zu können. Dabei kann es zu zwei Probleme kommen:

1. Der Datensatz wird ungleichmäßig verteilt. Das hätte zur Folge, dass einige Knoten im Cluster mehr Zeit für die Generierung von Kandidaten oder Berechnung der Signifikanz benötigen. Da jeder der beiden Teilschritte von *PaSiGraM* (*Signifikanzberechnung*, *Kandidatengenerierung*) erst ausgeführt werden kann, wenn der vorherige Teilschritt komplett terminiert ist, würde das den Algorithmus ausbremsen.

2. Der Datensatz, der die Kandidaten enthält, wird so ungünstig auf die Knoten im Cluster aufgeteilt, dass während der Generierung und *Signifikanzberechnung* erhöhter Kommunikationsaufwand zwischen den Knoten des Clusters entsteht, weil einigen Knoten bestimmte Teile des Datensatzes fehlen, die für die Generierung oder *Signifikanzberechnung* benötigt werden. Dann müsste ein Knoten nämlich einen Teil des Datensatzes bei einem anderen Knoten anfordern, was die Berechnung ebenfalls verlangsamen würde.

Um diesen Problemen aus dem Weg zu gehen, nutzt *PaSiGraM* die von *Apache Flink* gegebene Möglichkeit aus, die *DataSets* (die Datenstruktur, die in Flink genutzt wird, um die *CAMs* der Graphen abzuspeichern) manuell partitionieren zu können. Dazu werden verschiedene Operatoren, wie z.B. *partitionByHash, partitionByRange* oder auch *rebalance* bereitgestellt.
Die Operatoren mit denen *PaSiGraM* die Kandidaten generiert bzw. die Signifikanz berechnet, sind alle so konzipiert, dass sie immer die gleiche Menge von Graphen benötigen. So basiert z.B.

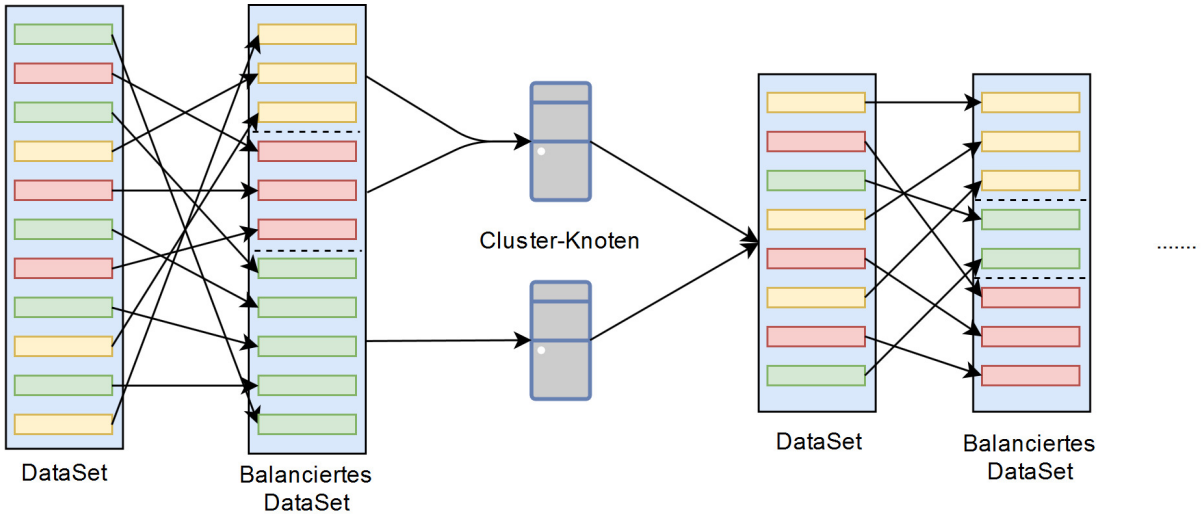


Abbildung 6.8: *PaSiGraM's* Partitionierung eines *DataSets*.

FFSM-Join darauf, nur Teilgraphen zu joinen, welche dieselbe *maximal korrekte Teilmatrix* und somit die gleichen Eltern-Graphen haben. Diese Eigenschaft der Operatoren erlaubt es *PaSiGraM* das *DataSet* effizient partitionieren zu können. So wird das *DataSet* der neuen Kandidaten auf Grundlage der Eltern-Graphen eines jeden Kandidaten partitioniert. Das bedeutet, dass die Kandidaten, die dieselben Eltern besitzen, auch derselben Partition zugewiesen werden. Anschließend wird die Anzahl der entstandenen Partitionen ermittelt, um Berechnen zu können wie viele Partitionen jedem Knoten im Cluster jeweils zugewiesen werden. Dadurch haben alle Knoten im Cluster ungefähr gleich viele Daten auf denen sie arbeiten und jeweils den Teil des *DataSets* den sie für die *Signifikanzberechnung* und *Kandidatengenerierung* benötigen. In Abbildung 6.8 ist zu sehen, wie *PaSiGraM* ein *DataSet* partitioniert. Die verschiedene Farben (grün, rot und gelb) symbolisieren dabei die Zugehörigkeit eines Kandidaten zu einem Eltern-Graph. Die Kandidaten werden dann ihrer jeweiligen Partition im *DataSet* zugeordnet, bevor die Partitionen dann anschließend nahezu gleichmäßig auf die verschiedenen Cluster-Knoten aufgeteilt werden.

6.6 Umsetzung in Flink

In diesem letzten Kapitel wird beschrieben, wie *PaSiGraM* mit Flink umgesetzt werden kann. Der Algorithmus wurde von vornherein so konzipiert, dass alle Operationen, wie auch die gewählte Datenstruktur (*CAM*), auf Basis der DataSet API von Flink realisiert werden können. Dazu wird zunächst beschrieben, welche Teile von *PaSiGraM* durch Flink parallelisiert werden können. Anschließend wird erläutert, wie das entsprechende Flink-Programm aufzubauen ist.

6.6.1 Parallelisierung

Wie schon erwähnt, ist *PaSiGraM* für die Verwendung unter *Apache Flink* konzipiert worden. Um alle Operatoren parallelisieren zu können, werden die von der DataSet API bereitgestellten Operatoren verwendet. In Abbildung 6.9 ist zu sehen, welche Teile von *PaSiGraM* somit parallelisiert werden können. Dabei ist durch die Verzweigungsknoten (graue Rechtecke) gekennzeichnet, wo eine parallele Verarbeitung durch Flink möglich ist. Die blauen Kästen stellen jeweils die Methoden auf den jeweiligen Datenstrukturen der Graphen dar. Die grünen Kästen hingegen sind die zwei Hauptkomponenten (*Kandidatengenerierung, Signifikanzberechnung*) von *PaSiGraM*. Die Parallelisierung kann mithilfe von Operatoren der DataSet API umgesetzt werden, indem die Logik dieser mit den jeweiligen Methoden überschrieben wird. Somit werden *Kandidatengenerierung* und *Signifikanzberechnung* parallelisiert. Die einzelnen Methoden (z.B. *FFSM-Join* und *FFSM-Extension*) hingegen werden weiterhin nur auf einer Maschine ausgeführt. Damit wird erreicht, dass die Hauptkomponenten, welche auf einer Menge von Graphen arbeiten, parallel berechnet und die Methoden, die jeweils immer nur auf einem einzelnen Graphen arbeiten, auf den einzelnen Cluster-Knoten bearbeitet werden.
PaSiGraM arbeitet, wie schon weiter oben erwähnt, nach dem Konzept der Breitensuche. Dadurch kann die horizontale Skalierbarkeit besser ausgenutzt und somit der Algorithmus bestmöglich parallelisiert werden.
Eine Berechnung, die noch zu komplex für eine einzelne Maschine sein könnte, wäre die Erstellung der *CAM* eines jeden Teilgraphen. Aber durch die zusätzlichen Heuristiken die *PaSiGraM* verwendet, um den Suchraum für die *CAM* einzuschränken, lässt sich auch dieser Vorgang auf einem einzelnen Cluster-Knoten realisieren. Das ist ebenfalls ein Vorteil gegenüber *SSiGraM* von *Qiao et al.* ([QZL+18]). Denn hier fehlt es an einer Erläuterung, wie die *CAM* eines Graphen berechnet wird. Deswegen ist davon auszugehen, dass hier keine zusätzlichen Heuristiken zum Einsatz kommen um den Suchraum zu verringern. Das hat zu Folge, dass die einzelnen Maschinen mehr Zeit für die *Kandidatengenerierung* benötigen und somit den Algorithmus ausbremsen.

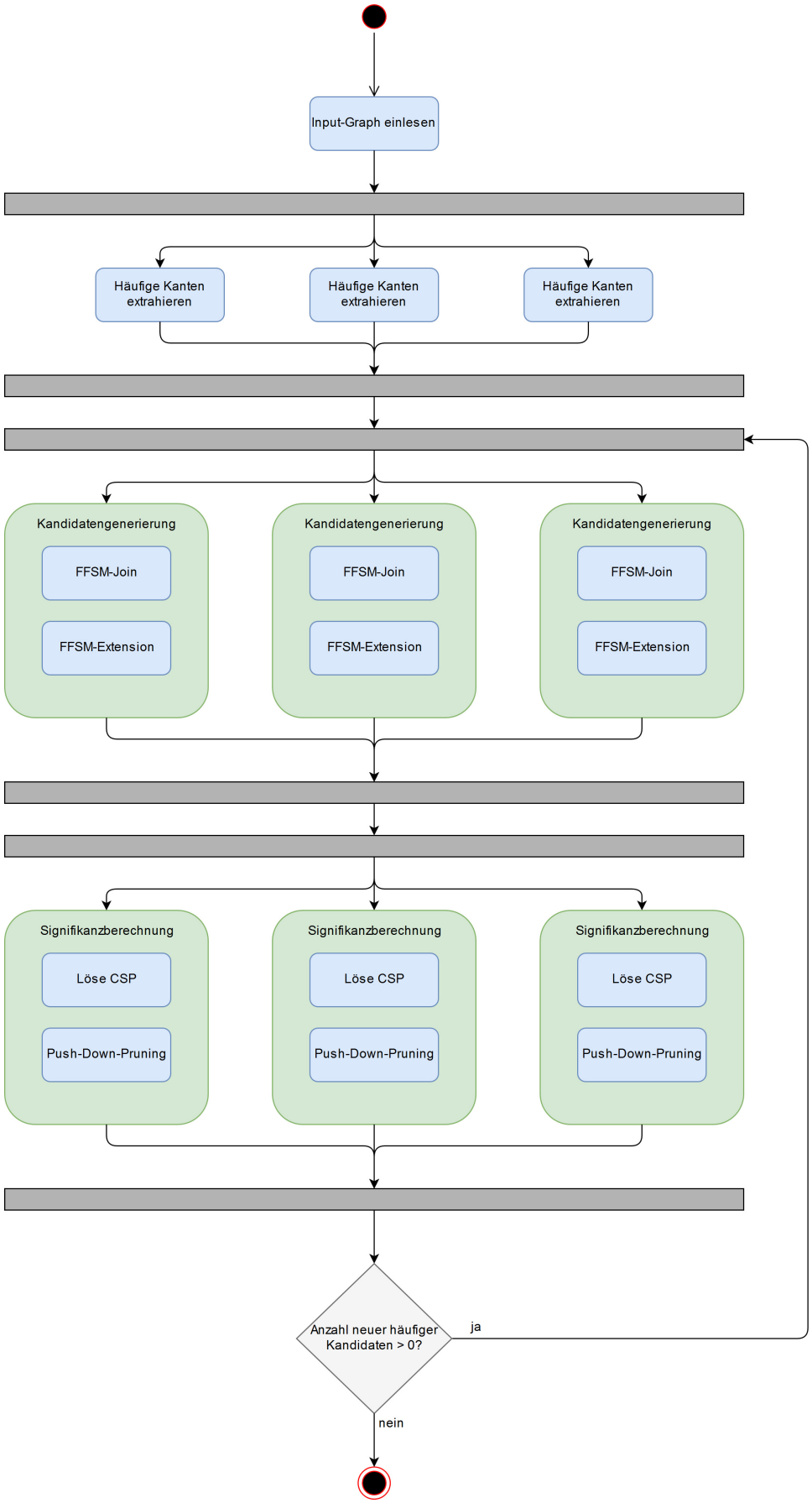


Abbildung 6.9: Parallele Ausführung von *PaSiGraM*.

6.6.2 Aufbau des Programms

In Algorithmus 9 ist der Pseudocode für eine Implementierung von *PaSiGraM* zu sehen. Zunächst werden aus dem Input-Graph G alle häufigen Kanten extrahiert mit denen dann die ersten initialen Teilgraphen S_1 berechnet werden können. Dann beginnt der iterative Teil von *PaSiGraM*. Dieser kann in Flink mithilfe des *Delta-Iteration*-Operators umgesetzt werden, welcher schon in Abschnitt 4.2.4 näher erläutert wurde. In der *while*-Schleife werden die vorher generierten häufigen Teilgraphen in S_{k-1} abgespeichert. Anschließend werden mit *GenerateSubgraphs* die neuen Teilgraphen erzeugt und in S_k abgespeichert. Mit den Operatoren *partitionCustom* und *rebalance* werden die Optimierungen aus Unterkapitel 6.5 (Partitionieren des *DataSets*) umgesetzt. Diese Optimierungen werden auch für die *Signifikanzberechnung* eingesetzt. Diese wird mithilfe von *flatMap* (eine DataSet-Operation) realisiert. Innerhalb von *flatMap* wird *calculateSupport* ausgeführt, welches dann das *PaSiGraM-CSP* für jeden Teilgraphen löst. Alle Teilgraphen aus S_k, die die Mindesthäufigkeit θ erfüllen, werden in S_{DS} abgespeichert. Am Ende der *while*-Schleife werden die neu generierten häufigen Teilgraphen dem Ergebnisdatensatz *result* hinzugefügt, bevor die Schleife wieder von vorne beginnt. *PaSiGraM* läuft dann solange bis keine neuen häufigen Teilgraphen mehr gefunden werden können.

Algorithmus 9: PaSiGraM

Input: Input-Graph G, Mindesthäufigkeit θ
Output: Ein DataSet *result* mit allen häufigen Teilgraphen
$frequentEdges \leftarrow$ alle häufigen Kanten aus G
initiale Teilgraphen $S_1 \leftarrow frequentEdges$
$S_{DS} \leftarrow S_1$
while $S_{DS}.count > 0$ **do**
  $S_{k-1} \leftarrow S_{DS}$
  $S_k \leftarrow$ **GenerateSubgraphs**(S_{k-1}.**partitionCustom**(Eltern).**rebalance**)
  $S_{DS} \leftarrow S_k$.**partitionCustom**(Eltern).**rebalance**.**flatMap**{
  Sei s der aktuelle Teilgraph
  if ***CalculateSupport***(s, G, θ) **then**
    return s
  end
  }.**collect**
  $result \leftarrow result \cup S_{DS}$
end
return *result*

Die in Unterkapitel 6.2 vorgestellte Datenstruktur *CAM* ist in Form einer eigenen Datenstruktur in Java zu implementieren. Diese kann dann, wie vorher schon erwähnt, in *DataSets* abgespeichert werden. Flink organisiert dann Verteilung und Fehlertoleranz. Dadurch wird gewährleistet, dass die Berechnungen auf dem Input-Graphen, wie auch den jeweiligen Teilgraphen parallelisiert werden können. Allerdings ist bei der Implementierung dieser Datenstruktur darauf zu achten, dass jeder Graph eine eindeutige ID besitzt, auf Grundlage derer man schließlich feststellen können muss, aus welchem anderen Teilgraph(en) der jeweilige Teilgraph generiert wurde. Denn dies ist nötig, um die Optimierungen aus Unterkapitel 6.5 umsetzen zu können.

6.7 Fazit

Mit *PaSiGraM* wurde ein Algorithmus entwickelt, der dem Anwendungsfall dieser Arbeit gerecht werden kann. Dabei wird mit der *CAM* eine Datenstruktur verwendet, die nicht nur die Graphen vollständig repräsentieren kann, sondern auch jegliche Operationen ermöglicht, die für das *Frequent-Subgraph-Mining* nötig sind. Dadurch kann *PaSiGraM* durchgängig auf der *CAM* der Graphen arbeiten und vermeidet somit unnötige Transformationen zwischen verschiedenen Datenstrukturen. Zusätzlich kommen mit *Knoten-Invarianten, Knoten-Stabilisierung* und *Kantenbasierter Ordnung von Partitionen* Heuristiken bei der Erstellung der *CAM* zum Einsatz, was die Performance des Algorithmus verbessern sollte.
Da *PaSiGraM* nach dem Prinzip der Breitensuche arbeitet, kann somit die horizontale Skalierung von Flink besser ausgenutzt werden. Bei der *Kandidatengenerierung* werden mit *FFSM-Join* und *FFSM-Extension* zwei Operatoren verwendet, die ebenfalls ausschließlich auf den *CAMs* der jeweiligen Graphen arbeiten.
Für die *Signifikanzberechnung* wurde mit dem *PaSiGraM-CSP* ein neues Verfahren entwickelt mit dem ein *CSP* auf die *CAM* eines Graphen angewendet werden kann. Durch verschiedene Eigenschaften der *CAM* wird hier der Suchraum für mögliche Lösungen eines *PaSiGraM-CSP* zusätzlich eingeschränkt, was die *Signifikanzberechnung* von *PaSiGraM* effizienter machen sollte. Durch die zusätzlichen Optimierungen für *Signifikanzberechnung* und *Kandidatengenerierung* (Unterkapitel 6.5) beinhaltet der Algorithmus ein weiteres Verfahren, was dazu beitragen kann, dass die Berechnungen noch effizienter gestaltet werden können.
Anfangs wurde erwähnt, dass mit *SSiGraM* von *Qiao et al.* ([QZL+18]) ein FSM-Algorithmus erschienen ist, der ebenfalls parallel und auf einem einzelnen Graph als Input-Datensatz arbeitet und somit in Konkurrenz zu *PaSiGraM* steht. *PaSiGraM* kann sich jedoch in vielen Punkten von diesem Algorithmus abheben. So werden bei *SSiGraM* z.B. keine zusätzlichen Heuristiken verwendet, um die Anzahl der Permutationen bei der Suche nach einer *CAM* zu verringern. Daraus resultierend besitzen die *CAMs* bei *SSiGraM* auch nicht die benötigten Eigenschaften, um den Suchraum beim Lösen des *CSPs* eingrenzen zu können. Somit rechnet *SSiGraM* mit

einem NP-vollständigen Problem, was sich negativ auf dessen Performance auswirken dürfte. *PaSiGraM* hingegen, verwendet von vornherein eine komplett andere Strategie das *CSP* zu modellieren. Hierbei wird, wie eben schon erwähnt, die *Signifikanzberechnung* direkt auf der *CAM* des jeweiligen Graphen durchgeführt. Durch diesen neuen Ansatz und die speziellen Eigenschaft der *CAMs* von *PaSiGraM* ist der Suchraum bei der *Signifikanzberechnung* deutlich kleiner, als es bei *SSiGraM* der Fall ist. Die zusätzlichen Optimierungen von *Kandidatengenerierung* und *Signifikanzberechnung*, die durch das *Clustern* von *DataSets* umgesetzt werden, fehlen bei *SSiGraM* ebenfalls komplett, was *PaSiGraM* im Vergleich zusätzlich schneller machen sollte. In den Abschnitten 6.6.1 und 6.6.2 wurde zudem noch erklärt, wie *PaSiGraM* mithilfe von Flink parallelisiert und implementiert werden kann.
Durch die Optimierungen von *PaSiGraM* ist letztendlich ein Algorithmus entstanden, der alle in Kapitel 5 identifizierten Anforderungen erfüllt. Er funktioniert auf einem einzelnen Graph als Input-Datensatz. Zudem ist der Algorithmus für *Apache Flink* optimiert. Es wäre allerdings nicht viel aufwändiger *PaSiGraM* auch auf *Apache Spark* zu implementieren. Mit der *CAM* verwendet der Algorithmus eine Datenstruktur mit der Graphen ohne Informationsverlust repräsentiert werden können. Zusätzlich können alle für das *Frequent-Subgraph-Mining* benötigten Operationen auf der *CAM* ausgeführt werden. Die verschiedenen Heuristiken und Optimierungen von *PaSiGram* tragen dazu bei, dass die Performance des Algorithmus dadurch noch besser sein sollte.

Kapitel 7

Zusammenfassung und Ausblick

In den ersten Kapiteln dieser Arbeit (2 - 3) wurde sich mit Graphen, Hypergraphen und dem *Graph-Mining* auseinander gesetzt. Dabei wurde sich in Unterkapitel 3.2 auf das *Frequent-Subgraph-Mining* spezialisiert. Es wurde gezeigt, was darunter zu verstehen ist und welche Probleme bei dieser *Graph-Mining*-Technik im Allgemeinen auftreten können. Dazu wurde das *Frequent-Subgraph-Mining* in seine einzelnen Schritte zerlegt, um ein besseres Verständnis zu schaffen und potenzielle Schwierigkeiten besser identifizieren zu können. Dabei ist herausgekommen, dass das *Frequent-Subgraph-Mining* mit vielen Problemen einher geht, welche teilweise sogar NP-vollständig sind. Im Gegenzug wurden jedoch auch Konzepte vorgestellt, mit denen diese Probleme gelöst werden können. Zu den Wichtigsten zählen hier die beiden Strategien *Join* und *Pattern-Growth* zur *Kandidatengenerierung*, wie auch der *Minimum-Image-based-Support* (kurz *MNI*) zur Berechnung der Signifikanz. In Unterkapitel 3.3 wurde zudem mit *GraMi* ein konkreter FSM-Algorithmus vorgestellt, der für den weiteren Verlauf dieser Arbeit noch wichtig sein sollte.

Da ein Ziel dieser Arbeit die Parallelisierung einer *Graph-Mining*-Technik war, wurden in Kapitel 4 unterschiedliche Plattformen zur parallelen Verarbeitung vorgestellt. Dazu zählen die beiden Plattformen *Apache Spark* und *Apache Flink*, wie auch das an der *Universität Rostock* entwickelte *Hydra.PowerGraph*-System. Die beiden Plattformen *Apache Spark* und *Apache Flink* sollten die Grundlage für die Parallelisierung einer *Graph-Mining*-Technik darstellen, weswegen sie auch detaillierter ausgeführt wurden. Das *Hydra.PowerGraph*-System hingegen wurde aufgenommen, da es im Zusammenhang mit dem Anwendungsfall dieser Arbeit steht und somit ebenfalls relevant ist.

In Kapitel 5 wurde dann schließlich eine Problemanalyse durchgeführt. Dazu war zunächst der Anwendungsfall und der mit ihm einhergehende Datensatz genauer zu untersuchen. Das Er-

gebnis war, dass der Datensatz als zusammenhängender Hypergraph bereitgestellt wurde. Dieser Hypergraph wurde dann mithilfe der *Clique-Expansion* in einen Graphen überführt, welcher dann schließlich als Input-Datensatz für den Algorithmus dienen sollte. Anschließend wurden die beiden *Parallelisierungsplattformen* Flink und Spark einem Vergleich unterzogen, damit eine Empfehlung gegeben werden konnte, mit welcher Plattform letztendlich gearbeitet werden sollte. Bei diesem Vergleich wurde unter anderem deutlich, dass beide Plattformen zwar gut zur parallelen Umsetzung von *Graph-Primitiven* sind, allerdings bei komplexeren *Mining*-Verfahren noch ausbaufähig sind. Dennoch ging *Apache Flink* als Gewinner aus diesem Vergleich hervor. Zuletzt wurde in Unterkapitel 5.3 der *Stand der Forschung* betrachtet. Dabei ist herausgekommen, dass es zu diesem Zeitpunkt keine Algorithmen für das *Frequent-Subgraph-Mining* gab, die darauf ausgelegt waren, parallel berechnet werden zu können. Das diente dann schließlich als Motivation mit *PaSiGraM* einen eigenen Algorithmus zu entwickeln, der die in Unterkapitel 5.4 vorgestellten Anforderungen erfüllen kann.

In Kapitel 6 wird dann letztendlich der *PaSiGraM*-Algorithmus im Detail beschrieben. Da im Verlauf der Entwicklung von *PaSiGraM* mit *SSiGraM* ein komplett neuer Algorithmus veröffentlicht wurde, der ebenfalls parallel berechnet werden kann, wurde *PaSiGraM* nochmals weiterentwickelt, um sich von *SSiGraM* abheben zu können. Dadurch ist schließlich ein Algorithmus entstanden, der verschiedene Vorteil gegenüber *SSiGraM* mitbringt. *PaSiGraM* ist aber nicht nur für den Anwendungsfall dieser Arbeit konzipiert. Er kann ebenso auf andere Datensätze angewendet werden, die als ein zusammenhängender Graph abgespeichert sind. Dadurch ist *PaSiGraM* aktuell einer von zwei Algorithmen, die das *Frequent-Subgraph-Mining* auf einem einzelnen zusammenhängenden Graphen parallel berechnen können.

Da im Verlauf dieser Arbeit unterschiedliche Probleme aufgetreten sind, welche in vorherigen Kapiteln schon erläutert wurden, war es in der 12-wöchigen Bearbeitungszeit leider nicht mehr möglich *PaSiGraM* zu implementieren und dementsprechend zu evaluieren. Also ist eine weiterführende Aufgabe *PaSiGraM* auf einem Cluster mit *Apache Flink* zu implementieren. Anschließend sollten unterschiedliche Messungen bezüglich der Performance durchgeführt und mit den Resultaten anderer Algorithmen wie *SSiGraM* und *GraMi* verglichen werden. Ein weiterer interessanter Vergleich wäre zudem, *PaSiGraM* ebenfalls auf *Apache Spark* zum implementieren und die Laufzeit des Algorithmus auf den beiden Plattformen zu vergleichen.
Zusätzlich könnte *PaSiGraM* noch so angepasst werden, dass der Algorithmus auch auf einer Menge von Graphen als Input-Datensatz läuft. Damit würde man das komplette Spektrum des *Frequent-Subgraph-Mining* abdecken. Zudem könnten Techniken bzw. Bibliotheken mit eingebunden werden, mit derer Hilfe man die Ergebnisdatensätze dann auch visualisieren könnte.
Ein längerfristiges Ziel könnte auch sein, ein Framework für das parallele *Graph-Mining* zu

bauen. Wie schon erwähnt, existiert so etwas aktuell noch nicht. Dabei müsste allerdings darauf geachtet werden, wie sich *Gradoop* weiterentwickelt. Momentan existiert dort mit *DIMSpan* ein Algorithmus für das *Frequent-Subgraph-Mining* auf einer Menge von Graphen als Input-Datensatz. Es bleibt allerdings abzuwarten, ob noch weitere *Graph-Mining*-Algorithmen ergänzt werden.
Im Zusammenhang mit dem *ISEBEL*-Projekt, würde das *Minen* von *Transactional-Data* mit Sicherheit auch noch ein interessanter Punkt sein. Denn dadurch könnten Gemeinsamkeiten in den unterschiedlichen Datenbanken der verschiedenen Länder ermittelt werden, die im Moment schon Teil dieses Projektes sind. Aber auch andere *Graph-Mining*-Verfahren könnten für das Projekt von Interesse sein. So könnte man anhand von *Klassifikation* unbekannte Datenpunkte auf Grundlage schon vorhandener Daten klassifizieren. Aber auch die *Klassifikation* von ganzen Graphen wäre ein denkbares Szenario. Die eben schon erwähnte Idee eines Frameworks für das parallele *Graph-Mining* könnte die Forschung in diesem Projekt mit Sicherheit gut voran bringen.

Literaturverzeichnis

[20002] *Machine Learning: Proceedings of the 1988-02 International Conferences (ICML).* Morgan Kaufmann, 2002.

[20117] *Handbook of Big Data Technologies.* Springer-Verlag GmbH, 2017.

[ABE+14] Alexandrov, Alexander, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas und Daniel Warneke: *The Stratosphere platform for big data analytics.* The VLDB Journal, 23(6):939–964, may 2014.

[AL17] Ausiello, Giorgio und Luigi Laura: *Directed hypergraphs: Introduction and fundamental algorithms—A survey.* Theoretical Computer Science, 658:293–306, jan 2017.

[ASF17a] ASF: *Apache Spark.* `https://spark.apache.org/`, 2017. Eingesehen am 09.02.2018.

[ASF17b] ASF: *Spark: GraphX Programming Guide.* `https://spark.apache.org/docs/latest/graphx-programming-guide.html`, 2017. Eingesehen am 09.02.2018.

[ASF17c] ASF: *Spark: MLlib.* `https://spark.apache.org/docs/latest/ml-guide.html`, 2017. Eingesehen am 09.02.2018.

[ASF17d] ASF: *Spark: SQL, DataFrames and DataSets.* `https://spark.apache.org/docs/latest/sql-programming-guide.html`, 2017. Eingesehen am 09.02.2018.

[ASF17e] ASF: *Spark: Streamining.* `https://spark.apache.org/docs/latest/streaming-programming-guide.html`, 2017. Eingesehen am 09.02.2018.

[ASF18a] ASF: *Flink: CEP - Complex event processing for Flink.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/libs/cep.html, 2018. Eingesehen am 11.02.2018.

[ASF18b] ASF: *Flink: Data Streamining Fault Tolerance.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/internals/stream_checkpointing.html, 2018. Eingesehen am 10.02.2018.

[ASF18c] ASF: *Flink: Dataflow Programming Model.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/concepts/programming-model.html, 2018. Eingesehen am 10.02.2018.

[ASF18d] ASF: *Flink: DataSet API: Iterations.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/batch/iterations.html, 2018. Eingesehen am 11.02.2018.

[ASF18e] ASF: *Flink: DataSet API Programming Guide.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/batch/index.html#dataset-transformations, 2018. Eingesehen am 10.02.2018.

[ASF18f] ASF: *Flink: DataStream API: Operators.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/stream/operators/index.html, 2018. Eingesehen am 10.02.2018.

[ASF18g] ASF: *Flink: Ecosystem.* https://flink.apache.org/ecosystem.html, 2018. Eingesehen am 10.02.2018.

[ASF18h] ASF: *Flink: FlinkML - Machine Learning for Flink.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/libs/ml/index.html, 2018. Eingesehen am 11.02.2018.

[ASF18i] ASF: *Flink: Gelly: Flink Graph API.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/libs/gelly/index.html, 2018. Eingesehen am 11.02.2018.

[ASF18j] ASF: *Flink: Introduction to Apache Flink.* https://flink.apache.org/introduction.html, 2018. Eingesehen am 10.02.2018.

[ASF18k] ASF: *Flink: Table API & SQL Beta.* https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/table/index.html, 2018. Eingesehen am 11.02.2018.

[AW10] Aggarwal, Charu C. und Haixun Wang: *A Survey of Clustering Algorithms for Graph Data.* In: *Managing and Mining Graph Data*, Seiten 275–301. Springer US, 2010.

[BB02] Borgelt, Christian und Michael R Berthold: *Mining molecular fragments: Finding relevant substructures of molecules.* In: *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, Seiten 51–58. IEEE, 2002.

[BN08] Bringmann, Björn und Siegfried Nijssen: *What is frequent in a single graph?* Advances in Knowledge Discovery and Data Mining, Seiten 858–863, 2008.

[CCA12] Charu C. Aggarwal, Haixun Wang (Herausgeber): *Managing and Mining Graph Data.* Springer US, 2012.

[CFE+] Carbone, Paris, Gyula Fóra, Stephan Ewen, Seif Haridi und Kostas Tzoumas: *Lightweight Asynchronous Snapshots for Distributed Dataflows.*

[CKE+15] Carbone, Paris, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi und Kostas Tzoumas: *Apache flink: Stream and batch processing in a single engine.* Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4), 2015.

[DFS+18] Dietrich, Daniel, Ole Fenske, Philipp Schweers, Stefan Schomacker und Andreas Heuer: *Stonebraker gegen Google: Das 2:0 fällt in Rostock (Ein Vergleich von Big-Data-Analytics-Plattformen).* Mai 2018. Accepted for GvDB 2018.

[DKWK05] Deshpande, M., M. Kuramochi, N. Wale und G. Karypis: *Frequent substructure-based approaches for classifying chemical compounds.* IEEE Transactions on Knowledge and Data Engineering, 17(8):1036–1050, aug 2005.

[EASK14] Elseidy, Mohammed, Ehab Abdelhamid, Spiros Skiadopoulos und Panos Kalnis: *GraMi: frequent subgraph and pattern mining in a single large graph.* Proceedings of the VLDB Endowment, 7(7):517–528, mar 2014.

[FB07] Fiedler, Mathias und Christian Borgelt: *Support Computation for Mining Frequent Subgraphs in a Single Graph.* In: *Workshop on Mining and Learning with Graphs*, 2007.

[GLG+12] Gonzalez, Joseph E., Yucheng Low, Haijie Gu, Danny Bickson und Carlos Guestrin: *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.* In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Seiten 17–30, Hollywood, CA, 2012. USENIX.

[GLPN93] Gallo, Giorgio, Giustino Longo, Stefano Pallottino und Sang Nguyen: *Directed hypergraphs and applications.* Discrete Applied Mathematics, 42(2-3):177–201, apr 1993.

[GN02] Girvan, M. und M. E. J. Newman: *Community structure in social and biological networks.* Proceedings of the National Academy of Sciences, 99(12):7821–7826, jun 2002.

[HWP03] Huan, Jun, Wei Wang und Jan Prins: *Efficient mining of frequent subgraph in the presence of isomorphism.* techreport, UNC computer science technique report TR03-021, 2003.

[HWPY04] Huan, Jun, Wei Wang, Jan Prins und Jiong Yang: *Spin: mining maximal frequent subgraphs from graph databases.* In: *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, Seiten 581–586. ACM, 2004.

[JKA+17] Junghanns, Martin, Max Kiessling, Alex Averbuch, André Petermann und Erhard Rahm: *Cypher-based Graph Pattern Matching in Gradoop.* In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems - GRADES'17.* ACM Press, 2017.

[JPNR17] Junghanns, Martin, André Petermann, Martin Neumann und Erhard Rahm: *Management and Analysis of Big Graph Data: Current Systems and Open Challenges.* In: *Handbook of Big Data Technologies*, Seiten 457–505. Springer International Publishing, 2017.

[JPR17] Junghanns, Martin, André Petermann und Erhard Rahm: *Distributed grouping of property graphs with GRADOOP.* Datenbanksysteme für Business, Technologie und Web (BTW 2017), 2017.

[JPT+16] Junghanns, Martin, André Petermann, Niklas Teichmann, Kevin Gómez und Erhard Rahm: *Analyzing extended property graphs with Apache Flink.* In: *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics - NDA '16.* ACM Press, 2016.

[Kae17] Kaepke, Marc: *Graphen im Big Data Umfeld: Experimenteller Vergleich von Apache Flink und Apache Spark.* Doktorarbeit, Hochschule für Angewandte Wissenschaften Hamburg, 2017.

[KK04] Kuramochi, M. und G. Karypis: *An efficient algorithm for discovering frequent subgraphs.* IEEE Transactions on Knowledge and Data Engineering, 16(9):1038–1051, sep 2004.

[KK05] Kuramochi, Michihiro und George Karypis: *Finding Frequent Patterns in a Large Sparse Graph*.* Data Mining and Knowledge Discovery, 11(3):243–271, sep 2005.

[KR17] Khan, Arijit und Sayan Ranu: *Big-Graphs: Querying, Mining, and Beyond.* In: *Handbook of Big Data Technologies*, Seiten 531–582. Springer International Publishing, 2017.

[LHLC09] Leung, Ian X. Y., Pan Hui, Pietro Liò und Jon Crowcroft: *Towards real-time community detection in large networks.* Physical Review E, 79(6), jun 2009.

[LZY10] Li, Shirong, Shijie Zhang und Jiong Yang: *DESSIN: Mining Dense Subgraph Patterns in a Single Graph.* In: *Lecture Notes in Computer Science*, Seiten 178–195. Springer Berlin Heidelberg, 2010.

[Mac77] Mackworth, Alan K.: *Consistency in networks of relations.* Artificial Intelligence, 8(1):99–118, feb 1977.

[MSH17] Meyer, Holger, Alf-Christian Schering und Andreas Heuer: *The Hydra.PowerGraph System.* Datenbank-Spektrum, 17(2):113–129, jun 2017.

[NK04] Nijssen, Siegfried und Joost N Kok: *A quickstart in frequent structure mining can make a difference.* In: *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, Seiten 647–652. ACM, 2004.

[PJR17] Petermann, André, Martin Junghanns und Erhard Rahm: *DIMSpan - Transactional Frequent Subgraph Mining with Distributed In-Memory Dataflow Systems.* 2017.

[QZL+18] Qiao, Fengcai, Xin Zhang, Pei Li, Zhaoyun Ding, Shanshan Jia und Hui Wang: *A Parallel Approach for Frequent Subgraph Mining in a Single Large Graph Using Spark.* Applied Sciences, 8(2):230, feb 2018.

[RAK07] Raghavan, Usha Nandini, Réka Albert und Soundar Kumara: *Near linear time algorithm to detect community structures in large-scale networks.* Physical Review E, 76(3), sep 2007.

[SAD+10] Stonebraker, Michael, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo und Alexander Rasin: *MapReduce and parallel DBMSs.* Communications of the ACM, 53(1):64, jan 2010.

[SDC+16] Salloum, Salman, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng und Joshua Zhexue Huang: *Big data analytics on Apache Spark.* International Journal of Data Science and Analytics, 1(3-4):145–164, oct 2016.

[TN04] Tsuda, K. und W. S. Noble: *Learning kernels from biological networks by maximizing entropy.* Bioinformatics, 20(Suppl 1):i326–i333, jul 2004.

[TS10] Tsuda, Koji und Hiroto Saigo: *Graph Classification.* In: *Managing and Mining Graph Data*, Seiten 337–363. Springer US, 2010.

[TZ16] Talukder, N. und M. J. Zaki: *A distributed approach for graph mining in massive networks.* Data Mining and Knowledge Discovery, 30(5):1024–1052, jun 2016.

[XGFS13] Xin, Reynold S., Joseph E. Gonzalez, Michael J. Franklin und Ion Stoica: *GraphX: A Resilient Distributed Graph System on Spark.* In: *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13.* ACM Press, 2013.

[YH02] Yan, Xifeng und Jiawei Han: *gSpan: graph-based substructure pattern mining.* In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE Comput. Soc, 2002.

[ZCD+12] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker und Ion Stoica: *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing.* In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Seiten 2–2, Berkeley, CA, USA, 2012. USENIX Association.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 29. März 2018