8. Oktober 2011

Studienarbeit

Graphenbasierte Überprüfung unvollständiger Lösungen in Modellierungsaufgaben

Arne Leitert
Diplom Informatik (6201646)
arne.leitert@uni-rostock.de

Prof. Dr. Alke Martens
Gutachter

Dennis Maciuszek
Géraldine Ruddeck
Betreuer

Universität Rostock Traditio et Innovatio

Lehrstuhl für e-Learning und kognitive Systeme
Universität Rostock
Fakultät für Informatik und Elektrotechnik

Zusammenfassung

Diese Arbeit befasst sich mit der Überprüfung von Graphen, mit dem Ziel in computerbasierten Modellierungsaufgaben die Eingabe eines Lerners mit einer Musterlösung zu vergleichen. Dazu wird das Prinzip des Graphabstands theoretisch betrachtet und Algorithmen vorgestellt, die diesen ermitteln. Die Algorithmen werden anschließend getestet, wobei die Laufzeit im Vordergrund steht. Dabei brachten ein Branch-and-Bound-Verfahren sowie ein evolutionärer Algorithmus die besten Ergebnisse. Zusätzlich wird ein Konzept vorgestellt, mit dem die Eingabe eines Lerners genauer ausgewertet werden kann.

Abstract

This thesis deals with the checking of graphs with the goal of comparing the input of a student with a reference solution for a computerbased modelling task. Therefore the principle of graph edit distance is threated theoretically and algorithms are be introduced to compute graph edit distance. Next the algorithms will be tested with the focus on runtime. In the tests a branch-and-bound-procedure and an evolutionary algorithm yielded the best results. Also this thesis introduces a concept that allows analysing the input of the student.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen verschiedener E-Learning-Szenarien gibt es die Problematik, dass ein Lerner etwas modellieren soll. Das Modell des Lerners muss dann auf seine Korrektheit überprüft werden. Bei der Modellierung soll dem Lerner zusätzlich ein hilfreiches Feedback gegeben werden.

Ein Beispiel dafür ist das Programm *ChemNom*. Dabei soll der Lerner Moleküle verschiedener Stoffe zeichnen. *ChemNom* unterstützt den Lerner, indem es beispielsweise Aussagen über noch fehlende Atome oder ungültige Bindungen macht.

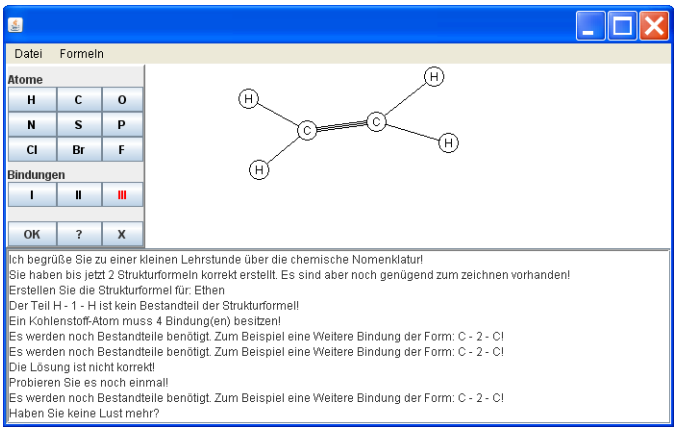


Abbildung 1.1: Das Programm *ChemNom*

Für diese Arbeit wird davon ausgegangen, dass die Modelle als Graphen vorliegen, da sich mit Graphen sehr viele Modelle darstellen lassen. Im Fall von *ChemNom* würden die Moleküle als Graphen dargestellt, indem man die Atome als Knoten interpretiert und die Bindungen als Kanten.

Des Weiteren ist eine Musterlösung gegeben. Die Eingabe des Lerners soll dann mit der Musterlösung verglichen werden. Dabei stellt sich die Frage nach der Machbarkeit einer solchen Überprüfung.

1.1 Korrektheit und Fehlergröße

Die erste Frage, die sich stellt, ist: Wann ist eine eingegebene Lösung korrekt? Da eine Musterlösung vorliegt, wird eine Lösung als korrekt definiert, wenn sie mit der Musterlösung identisch ist. Falls es mehrere richtige Lösungen gibt, kann man mehrere Musterlösungen angeben.

Bei einem Lernprozess ist es jedoch vermutlich häufiger der Fall, dass die vorgeschlagene Lösung nicht korrekt ist. Um dem Lerner ein Feedback geben zu können, ist es somit nötig zusätzliche Aussagen über eine falsche Lösung zu treffen. Ein erster Ansatz ist dabei ein Maß, um zwischen verschiedenen Graden an fehlerhaften Lösungen zu unterscheiden. In dieser Arbeit sei dieses Maß der Aufwand, der nötig ist, um die Lösung des Lerners in die Musterlösung umzuwandeln. Dieser Aufwand wird als Graphabstand bezeichnet.

Um zu überprüfen, ob eine Lösung (des Lerners) korrekt ist oder um eine Aussage über die Fehler in einer Lösung zu treffen, ist es notwendig Knoten und Kanten der Lösung den entsprechenden Knoten und Kanten in der Musterlösung zuzuordnen. Dies ist nicht trivial. In [2] wird das Problem damit umgangen, dass die Bezeichnung der Knoten in der Eingabe des Lerners mit Namenslisten abgeglichen wird, oder der Lerner selbst die Zuordnung vornimmt.

1.2 Ziele und Anforderungen

Ziel dieser Arbeit ist die Schaffung einer algorithmischen Grundlage, um in einer Modellierungsaufgabe den Lösungsvorschlag eines Lerners zu überprüfen und zu bewerten. Dabei soll die Zuordnung der Knoten zueinander automatisch erfolgen.

Für ein Verfahren zur Auswertung einer Lösung stellen sich dabei einige Anforderungen. Das Verfahren sollte möglichst unabhängig vom konkreten Modell arbeiten. Zusätzlich ist eine geringe Laufzeit wünschenswert. Im Idealfall bekommt der Lerner ohne spürbare Verzögerung eine Rückmeldung zu seiner Eingabe. Auch wenn die Berechnung länger dauert, sollte sie nicht mehr als ein paar Sekunden benötigen.

1.3 Vorhaben

Um die genannten Ziele zu erreichen, befasst sich diese Arbeit zuerst mit einigen Grundlagen im Bereich der Graphentheorie. Danach sollen verschiedene algorithmische Ansätze vorgestellt und untersucht werden. Fällt die Untersuchung positiv aus, dann wird zusätzlich ein Konzept vorgestellt, mit dem sich weitere Aussagen über die Eingabe eines Lerners treffen lassen.

Kapitel 2

Theoretische Grundlagen und Begriffe

Dieses Kapitel beinhaltet einen Überblick über Begriffe und Probleme, die in dieser Arbeit auftreten.

2.1 Graphen und Teilgraphen

Definition 2.1 (Graph) *Ein Graph G ist ein 2-Tupel $G = (V, E)$. Dabei sind*

- *V eine endliche Menge von Knoten und*
- *$E \subseteq V \times V$ die Menge an Kanten.*

Ist ein Graph ungerichtet*, dann gilt für jede Kante* $e = (u, v) \in E : (u, v) = (v, u)$*. Im* gerichteten *Fall gilt:* $(u, v) \neq (v, u)$.

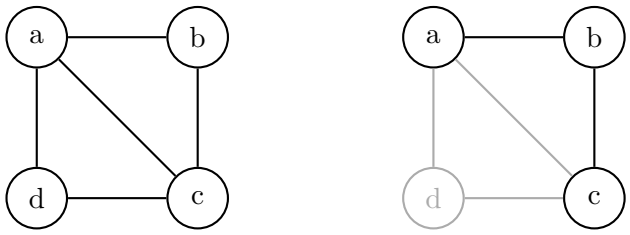
Definition 2.2 (Teilgraph) *Es sei $T = (V_t, E_t)$ ein Graph. T ist Teilgraph des Graphen $G = (V, E)$ genau dann, wenn die beiden nachfolgenden Bedingungen gelten:*

- $V_t \subseteq V$
- $e \in E_t \Rightarrow e \in E$

Gilt für alle $u, v \in V_t$ zusätzlich $(u, v) \in E \Rightarrow (u, v) \in E_t$, dann ist T ein induzierter *Teilgraph.*

Der Unterschied zwischen einem induzierten und nichtinduzierten Teilgraphen liegt darin, dass bei einem induzierten jede Kante zwischen

zwei Knoten u und v enthalten sein muss ($(u, v) \in E_t \Leftrightarrow (u, v) \in E$). Bei einem nichtinduzierten Teilgraph ist dies optional
($(u, v) \in E_t \Rightarrow (u, v) \in E$). Abbildung 2.1 stellt dies dar. Zwar ist T ein (nichtinduzierter) Teilgraph von G, jedoch fehlt die Kante (a, c), damit T auch ein induzierter Teilgraph ist.



(a) (b)

Abbildung 2.1: Der Graph G und dessen Teilgraph T

2.2 Graphisomorphie

Graphisomorphie (GI) bezeichnet vereinfacht gesagt, ob zwei Graphen gleich sind. Dies bedeutet bildlich gesprochen, dass man beide Graphen übereinander legen kann.

Definition 2.3 (Graphisomorphie) *Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ heißen isomorph, wenn eine bijektive Abbildung $\varphi : V_1 \to V_2$ existiert, so dass für alle $u, v \in V_1$ gilt: $(u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$.*

Das zur GI gehörende Entscheidungsproblem fragt, ob für zwei gegebene Graphen eine oben genannte Abbildung existiert. Eine Besonderheit ist hierbei die Komplexität. Zwar liegt GI in NP[1] [14], jedoch konnte bis zum Zeitpunkt dieser Arbeit weder bewiesen noch widerlegt werden, ob es möglich ist, GI in Polynomialzeit zu lösen oder ob GI NP-vollständig[2] ist [16, 18].

Es ist also unbekannt, ob es einen Algorithmus gibt, der mit polynomiellem Aufwand überprüft, ob zwei Graphen isomorph sind.

[1] Ein Problem liegt in NP genau dann, wenn es mit einer nichtdeterministischen (Turing-)Maschine in polynomieller Zeit gelöst werden kann.

[2] Ein Problem ist NP-vollständig genau dann, wenn es in NP liegt und sich jedes Problem in NP in Polynomialzeit darauf reduzieren lässt.

2.3 Teilgraphisomorphie

Ähnlich zur einfachen GI gibt Teilgraphisomorphie (TGI) an, ob ein Graph G_1 isomorph zu einem Teilgraph von G_2 ist.

Definition 2.4 (Teilgraphisomorphie) *Ein Graph $G_1 = (V_1, E_1)$ ist isomorph zu einem Teilgraph von $G_2 = (V_2, E_2)$ genau dann, wenn eine injektive Abbildung $\varphi : V_1 \to V_2$ existiert und für alle $u, v \in V_1$ gilt:*
$(u, v) \in E_1 \Rightarrow (\varphi(u), \varphi(v)) \in E_2$.

Analog zur GI wird auch beim TGI-Problem nach der Existenz einer in der Definition beschriebenen Abbildung gefragt. Das TGI-Problem ist NP-vollständig [5].

Anmerkung

Im Rahmen dieser Arbeit wird in dem Fall, dass G_1 isomorph zu einem Teilgraph von G_2 ist, lediglich davon gesprochen, dass G_1 ein Teilgraph von G_2 ist.

2.4 Größter gemeinsamer Teilgraph

Der größte gemeinsame Teilgraph (engl.: maximum common subgraph) zweier Graphen G_1 und G_2 beschreibt den größtmöglichen Graphen H, der Teilgraph von G_1 und G_2 ist.

Definition 2.5 (gemeinsamer Teilgraph (gTG)) *Ein Graph $H = (V_H, E_H)$ ist ein gemeinsamer Teilgraph von $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ genau dann, wenn alle der vier nachflogenden Bedingungen erfüllt sind:*

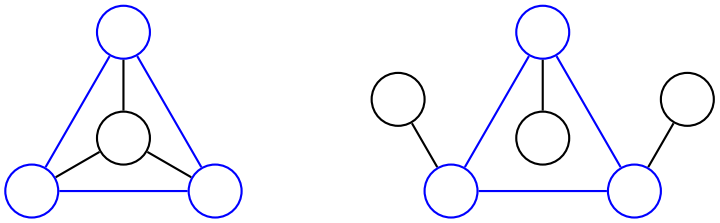
- $V_H \subseteq V_1$
- $E_H \subseteq E_1$
- *Es existiert eine injektive Abbildung $\varphi : V_H \to V_2$*
- *Für alle $u, v \in V_H$ gilt: $(u, v) \in E_H \Rightarrow (\varphi(u), \varphi(v)) \in E_2$*

Handelt es sich um einen gemeinsamen induzierten *Teilgraphen (giTG), dann gilt zusätzlich für alle $u, v \in V_H$:*
$(u, v) \in E_1 \Leftrightarrow (u, v) \in E_H \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$

2.4.1 Der induzierte Fall

Das Maß für die Größe des Teilgraphen ist abhängig davon, ob der giTG betrachtet wird oder lediglich der gTG. Im induzierten Fall ist die Anzahl der Knoten von H das Größenmaß.

Definition 2.6 (größter gemeinsamer induzierter Teilgraph) *Sei $H = (V_H, E_H)$ ein giTG von G_1 und G_2. H ist größter giTG von G_1 und G_2 genau dann, wenn kein giTG $H' = (V', E')$ von G_1 und G_2 existiert mit $|V'| > |V_H|$. Der größte giTG wird als MCS bezeichnet.*



(a) (b)

Abbildung 2.2: Der MCS (blau) zweier Graphen

Das NP-vollständige [7] MCS-Problem stellt die folgende Frage: Existiert für zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ sowie ein $k > 0$ ein giTG $H = (V_H, E_H)$, für den gilt: $|V_H| \geq k$?

2.4.2 Der allgemeine Fall

Wird lediglich der gTG betrachtet, ist die Anzahl der Kanten das Maß für die Größe. Dies liegt daran, dass ein Graph H, der nur aus den Knoten des kleineren der Graphen G_1 und G_2 besteht, immer ein gTG von G_1 und G_2 ist. Zum größten gTG zweier Graphen gehören somit immer alle Knoten des kleineren Graphen.

Satz 1 *Gegeben seien zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$, wobei $|V_1| \leq |V_2|$. $H = (V_1, \emptyset)$ ist dann ein gTG von G_1 und G_2.*

Beweis *$H = (V_H, E_H) = (V_1, \emptyset)$ ist gTG von G_1 und G_2, wenn die folgenden Bedingungen (aus Definition 2.5) gelten:*

1. $V_H \subseteq V_1$
2. $E_H \subseteq E_1$

3. *Es existiert eine injektive Abbildung* $\varphi : V_H \to V_2$

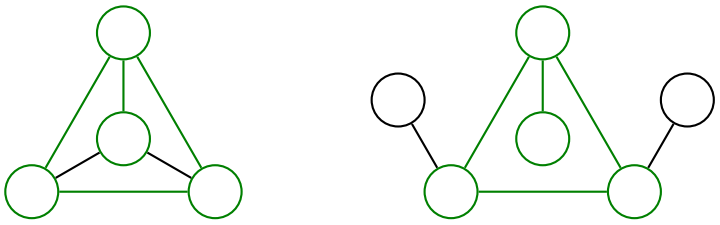
4. *Für alle* $u, v \in V_H$ *gilt:* $(u, v) \in E_H \Rightarrow (\varphi(u), \varphi(v)) \in E_2$

Die Bedingungen 1 ($V_1 \subseteq V_1$*) und 2 (*$\emptyset \subseteq E_1$*) sind offensichtlich erfüllt.*

Bedingung 3: Eine injektive Abbildung lässt sich dadurch erzeugen, dass jedem Knoten in V_1 *ein Knoten in* V_2 *zugewiesen wird. Da* $|V_1| \leq |V_2|$ *ist es möglich jedem Knoten in* V_1 *einen anderen Knoten in* V_2 *zuzuordnen. Somit existiert immer eine injektive Abbildung.*

Bedingung 4: Da H *keine Kanten besitzt (*$E_H = \emptyset$*), gilt für alle* $u, v \in V_H$*, dass* $(u, v) \in E_H$ *eine falsche Aussage ist. Somit ist die Implikation in Bedingung 4 immer wahr.*

Alle Bedingungen sind erfüllt, somit ist H *ein gTG von* G_1 *und* G_2*.* □



(a) (b)

Abbildung 2.3: Der größte gTG (grün) zweier Graphen

Definition 2.7 (größter gemeinsamer Teilgraph) *Sei* $H = (V_H, E_H)$ *ein gTG von* G_1 *und* G_2*.* H *ist größter gTG von* G_1 *und* G_2 *genau dann, wenn kein gTG* $H' = (V', E')$ *von* G_1 *und* G_2 *existiert mit* $|E'| > |E_H|$

2.5 Graphabstand

Als Graphabstand (engl.: graph edit distance) bezeichnet man die Kosten, die nötig sind, um einen Graphen so umzubauen, dass er isomorph zu einem anderen ist.

Die Definitionen in diesem Abschnitt basieren auf den entsprechenden Definitionen in [3].

Definition 2.8 (error correcting graph matching (ECGM)) *Es seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ Graphen. Ein error correcting graph matching ist eine bijektive Abbildung $\varphi : \hat{V}_1 \to \hat{V}_2$ mit $\hat{V}_1 \subseteq V_1$ und $\hat{V}_2 \subseteq V_2$.*

Die Abbildung φ stellt nun eine mögliche Bearbeitung von G_1 zu G_2 dar. Aus G_1, G_2 und φ leiten sich ab:

- $V_d := V_1 \backslash \hat{V}_1$ sind die zu löschenden Knoten,
- $V_a := V_2 \backslash \hat{V}_2$ sind die hinzuzufügenden Knoten,
- $E_d := E_1 \backslash \{(u, v) \mid u, v \in \hat{V}_1 \wedge (\varphi(u), \varphi(v)) \in E_2\}$ sind die zu löschenden Kanten und
- $E_a := E_2 \backslash \{(\varphi(u), \varphi(v)) \mid u, v \in E_1 \backslash E_d\}$ sind die hinzuzufügenden Kanten.

Definition 2.9 (Kosten eines ECGM) *Die Kosten eines ECGM φ von G_1 nach G_2 seien definiert durch*

$$c(\varphi) = \sum_{v \in V_d} c_{nd}(v) + \sum_{v \in V_a} c_{na}(v) + \sum_{e \in E_a} c_{ed}(e) + \sum_{e \in E_d} c_{ea}(e).$$

Dabei seien:

- $c_{nd}(v) \geq 0$ *die Kosten für das Löschen eines Knotens,*
- $c_{na}(v) \geq 0$ *die Kosten für das Hinzufügen eines Knotens,*
- $c_{ed}(e) \geq 0$ *die Kosten für das Löschen einer Kante und*
- $c_{ea}(e) \geq 0$ *die Kosten für das Hinzufügen einer Kante.*

Es ist möglich, die Kostenfunktion zu erweitern. Besitzt ein Graph beispielsweise eine Beschriftung oder Gewichtung (dies können z.B. Entfernungen in einem Straßennetz sein), so ist das Editieren eines Knotens oder einer Kante möglich. Dabei entfernt man die Kante nicht, sondern ändert lediglich die Beschriftung oder das Gewicht.

Definition 2.10 (Graphabstand) *Der Graphabstand $d(G_1, G_2)$ für zwei Graphen G_1 und G_2 sei definiert durch die minimalen Kosten über allen möglichen ECGMs von G_1 nach G_2.*

$$d(G_1, G_2) := \min\{c(\varphi) \mid \varphi \text{ ist ein ECGM von } G_1 \text{ nach } G_2\}$$

Graphabstand als Entscheidungsproblem

Gegeben seien zwei Graphen G_1 und G_2 sowie ein k. Existiert ein ECGM φ von G_1 nach G_2 mit $c(\varphi) \leq k$?

Garaphabstand ist NP-vollständig. [21]

2.6 Zusammenhang von MCS und Graphabstand

In [3] wird gezeigt, dass bei geeigneter Wahl der Kosten der Graphabstand zweier Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ durch die Größe des MCS $G = (V, E)$ berechnet werden kann. Es gilt dann:

$$d(G_1, G_2) = |V_1| + |V_2| - 2|V| \tag{2.1}$$

Der Graphabstand kann somit ermittelt werden, indem man den MCS zweier Graphen bildet. Knoten und Kanten, die nicht zum MCS gehören werden entfernt oder hinzugefügt.

Die Kostenfunktionen $c_{nd}(v)$, $c_{na}(v)$, $c_{ed}(e)$, und $c_{ea}(e)$, die in [3] angegeben wurden, damit (2.1) gilt, sind (angepasst an die in dieser Arbeit verwendeten Definitionen eines Graphen und Kosten eines ECGMs) wie folgt definiert:

- $c_{na}(v) = 1$
- $c_{nd}(v) = 1$
- $c_{ed}(e) = \begin{cases} \infty & e \in \hat{V}_1 \times \hat{V}_1 \\ 0 & \text{sonst} \end{cases}$
- $c_{ea}(e) = \begin{cases} \infty & e \in \hat{V}_2 \times \hat{V}_2 \\ 0 & \text{sonst} \end{cases}$

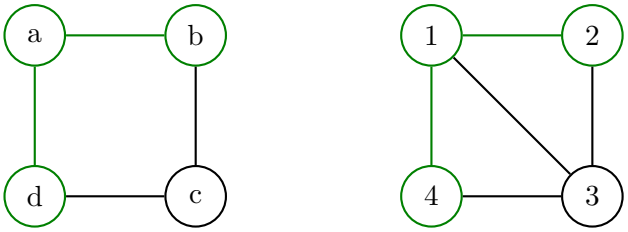
2.6.1 Probleme bei der Nutzung

Beim Ermitteln des Graphabstands mittels des MCS und der oben beschriebenen Kostenfunktion ergeben sich Probleme.

Die Kosten für das Hinzufügen und Entfernen von Kanten sind entweder 0 oder unendlich. Solange der Graphabstand endlich ist, ist er somit

unabhängig von der Anzahl der Kanten, die hinzugefügt oder entfernt werden müssen.

Ein weiteres Problem ist, dass alle Knoten, die nicht zum MCS gehören, entfernt oder hinzugefügt werden. Es kann jedoch Fälle geben, in denen das nicht erwünscht ist. Abbildung 2.4 stellt einen solchen Fall dar. Der MCS ist dabei *grün* markiert.



(a) Eingabe des Lerners

(b) Musterlösung

Abbildung 2.4: Beispiel für einen ungünstigen MCS (grün) zweier Graphen

Der Fehler des Lerners in Abbildung 2.4 ist lediglich eine fehlende Kante. Ermittelt man nun den Graphabstand mittels des MCS, so wird der Knoten c entfernt und Knoten 3 hinzugefügt. Somit wird Knoten c als fehlerhaft und Knoten 3 als fehlend betrachtet.

2.6.2 Kantengraphen

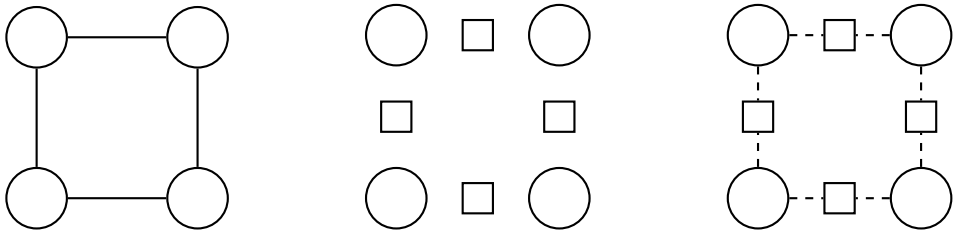
Kantengraphen sind ein für diese Arbeit erdachtes Konzept. Die Idee daran ist, in einem gegebenen Graphen die Kanten durch zusätzliche Knoten darzustellen. Der Kantengraph K eines Graphen $G = (V, E)$ ergibt sich, indem man jede Kante $e = (u, v) \in E$ zur Knotenmenge hinzugefügt und der so entstandene Knoten mit den „alten“ Knoten u und v verbunden wird. Abbildung 2.5 stellt dies dar.

Definition 2.11 (Kantengraph) *Für einen gegebenen Graphen $G = (V, E)$ sei dessen Kantengraph K wie folgt definiert:*

$$K = (V \cup E, \{(u, e), (e, v) \mid e = (u, v) \in E\})$$

Die Funktion k gibt den Kantengraph des übergeben Graphen zurück.

$$k(G) = K$$



(a) Original (b) Kanten in Knoten umwandeln (c) neue Kanten

Abbildung 2.5: Erstellen eines Kantengraphen

Mit Hilfe eines Kantengraphen lassen sich nun die oben genannten Probleme umgehen. Dazu sucht man nicht den MCS der ursprünglichen Graphen, sondern den MCS der Kantengraphen. Dabei gilt es aber zu beachten, dass man Knoten, die eine Kante darstellen und normale Knoten nicht einander zuordnet.

Ein Problem der ursprünglichen Graphen war, dass Kanten nicht bewertet wurden. Da jede Kante nun als Knoten betrachtet wird, wird somit auch jede Kante in die Kosten für den Graphabstand eingerechnet.

Ein weiteres Problem war, dass der MCS ein induzierter Teilgraph ist, wodurch Knoten ungewollt als fehlerhaft betrachtet werden können. Dieses Problem ist nun nicht mehr relevant, da jeder Teilgraph von G ein induzierter Teilgraph von dessen Kantengraph K ist. In dem Beispiel aus Abbildung 2.4 würden somit auch Knoten c und 3 zum MCS gehören.

Satz 2 *Wenn T ein Teilgraph des Graphen G ist, dann ist $k(T)$ ein induzierter Teilgraph von $k(G)$.*

Beweis *Gegeben seien die Graphen $G = (V_g, E_g)$ sowie $T = (V_t, E_t)$. T ist Teilgraph von G. $k(G) = (V_{kg}, E_{kg})$ und $k(T) = (V_{kt}, E_{kt})$ sind Kantengraphen von G und T.*

Da T Teilgraph von G ist, gilt:

$$V_t \subseteq V_g$$
$$e \in E_t \Rightarrow e \in E_g$$

Daraus folgt, dass $E_t \subseteq E_g$. Somit gilt auch:

$$V_t \cup E_t \subseteq V_g \cup E_g$$

Da dies nach Definition 2.11 jeweils die Knotenmengen der Kantengraphen $k(G)$ und $k(T)$ sind, folgt daraus, dass die Knoten von $k(T)$ eine Teilmenge der Knoten von $k(G)$ darstellen.

$$V_{kt} \subseteq V_{kg} \tag{2.2}$$

Für jede Kante $e = (u, v)$ gilt:

$$e = (u, v) \in E_t \Leftrightarrow (u, e) \in E_{kt} \text{ und } (e, v) \in E_{kt} \tag{2.3}$$

$$e = (u, v) \in E_g \Leftrightarrow (u, e) \in E_{kg} \text{ und } (e, v) \in E_{kg} \tag{2.4}$$

Da aus der linken Seite von (2.3) die linke Seite von (2.4) folgt (T ist Teilgraph von G) und die linke Seite jeweils äquivalent zur rechten ist, gilt somit, dass jede Kante ε in E_{kt} auch in E_{kg} vorhanden ist.

$$\varepsilon \in E_{kt} \Rightarrow \varepsilon \in E_{kg} \tag{2.5}$$

Aus (2.2) und (2.5) folgt, dass $k(T)$ ein Teilgraph von $k(G)$ ist. Es bleibt zu zeigen, dass es sich dabei um einen induzierten Teilgraphen handelt.

Es sei ε eine Kante in E_{kg}, dessen Knoten in V_{kt} sind. Nun gibt es zwei Fälle: $\varepsilon = (u, e)$ und $\varepsilon = (e, v)$, wobei $u, v \in V_t$ und $e = (u, v) \in E_t$.

Da $e = (u, v)$ eine Kante in T ist, sind auch (u, e) und (e, v) Kanten in $k(T)$. Somit ergibt sich, dass für alle $u^*, v^* \in V_{kt}$ gilt:
$(u^*, v^*) \in E_{kg} \Rightarrow (u^*, v^*) \in E_{kt}$. Somit ist $k(T)$ ein induzierter Teilgraph von $k(G)$. □

2.6.3 Nachteile des Kantengraphen

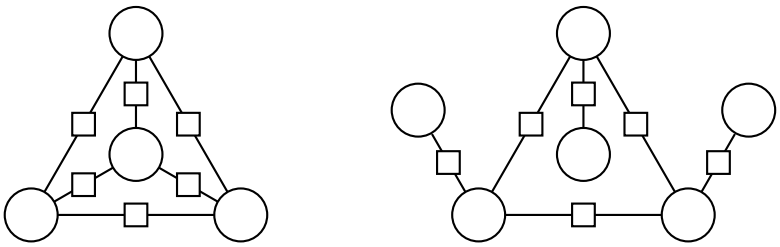
Kantengraphen besitzen zwei Nachteile. Dies sind ihre erhöhte Größe und ein Phänomen, das bei der Ermittlung eines MCS auftreten kann.

Phänomen der freien Kanten

Ermittelt man den MCS zweier Kantengraphen, gibt es ein Phänomen, das auftreten kann: *freie Kanten*.

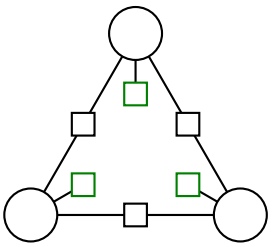
Die Kanten in einem Graphen G werden in dessen Kantengraph $k(G)$ als Knoten dargestellt. Bei der Suche nach einem MCS sind diese dann

gleichwertig mit „normalen“ Knoten. Es nun möglich, dass eine Kante e aus G, die ein Knoten in $k(G)$ darstellt, zum MCS gehört, die dazugehörigen Knoten jedoch nicht. Eine solche Kannte e sei eine *freie Kante*. Das Beispiel in Abbildung 2.6 stellt dies dar. Die freien Kanten sind *grün* markiert.



(a) Graph G_1

(b) Graph G_2



(c) MCS von G_1 und G_2

Abbildung 2.6: Das Phänomen der freien Kanten

Größe des Kantengraphen

Ein weiterer Nachteil ist, dass die Knotenzahl deutlich steigt. Da sowohl das Ermitteln des MCS als auch des Graphabstands NP-vollständig ist, gibt es vermutlich keinen Algorithmus in polynomieller Laufzeit sondern nur mit exponentieller. Ein Anstieg der Knoten und Kanten kann sich somit deutlich auf die Laufzeit auswirken.

2.7 MCS und Assoziationsgraphen

Das Finden eines MCS zweier Graphen lässt sich auf die Suche nach einer unabhängigen Menge (engl.: independent set) reduzieren [15]. Dazu erstellt man aus den gegebenen Graphen einen Assoziationsgraphen (AG). In diesem sucht man dann nach einem maximalen independent set.

2.7.1 Unabhängige Menge

Eine Unabhängige Menge in einem Graphen ist eine Teilmenge der Knoten des Graphen, wobei zwischen den Knoten keine Kante vorhanden ist.

Definition 2.12 (independent set) *Sei $G = (V, E)$ ein ungerichteter Graph. $V_i \subseteq V$ ist ein independent set in G genau dann, wenn für alle $e \in V_i \times V_i$ gilt: $e \notin E$.*

Das Ermitteln des maximum independent set (MIS) ist NP-vollständig [7].

2.7.2 Assoziationsgraph

Ein gemeinsamer induzierter Teilgraph ist eine Abbildung der Knoten von einem Graphen G_1 auf die Knoten eines Graphen G_2 (siehe Definition 2.5). Die Idee des AGs ist es, jede mögliche Zuordnung, die zu einer solchen Abbildung gehören kann, in einem Graphen darzustellen.

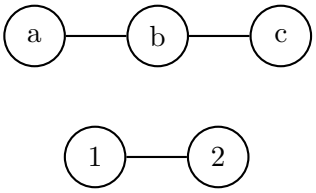


Abbildung 2.7: Die Graphen G_1 (oben) und G_2 (unten)

Die Knoten des AGs stellen die möglichen Zuordnungen von einem Knoten auf einen anderen dar. Jeder Knoten ist somit ein Paar aus einem Knoten aus G_1 und einem Knoten aus G_2. Abbildung 2.8 stellt die Kantenmenge des AGs dar, der sich aus den Graphen in Abbildung 2.7 ergibt.

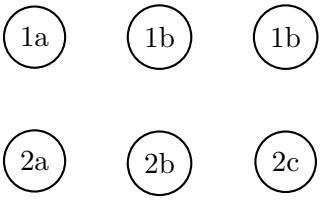


Abbildung 2.8: Die Knoten des Assoziationsgraphen von G_1 und G_2

Die Kanten des AGs (dargestellt in Abbildung 2.9) geben an, ob

Knotenpaare (also die Zuordnung eines Knotens zu einem anderen) einander ausschließen. Es gibt zwei Fälle, in denen das der Fall ist:

- Ein Knoten u aus G_1 kann nur höchstens einmal auf einen anderen Knoten in G_2 abgebildet werden. Alle anderen Knotenpaare, die den Knoten u beinhalten, sind somit nicht mehr möglich. Analog verhalten sich die Knoten aus G_2. Sie können nur einmal Ziel einer Zuordnung sein.

  Die so erzeugten Kanten sind in Abbildung 2.9 *schwarz* dargestellt.

- Hat man ein Knotenpaar, so schließt dies weitere Paare aus. Diese Paare ergeben sich durch die Betrachtung der Kanten der beiden ursprünglichen Graphen. Ein Paar (u_1, u_2) schließt ein Paar (v_1, v_2) aus, wenn es eine Kante (in G_1) zwischen u_1 und v_1 gibt, jedoch keine (in G_2) zwischen u_2 und v_2. Dies ist nötig, da ein induzierter Teilgraph dargestellt werden soll.

  Die so erzeugten Kanten sind in Abbildung 2.9 *rot* dargestellt.

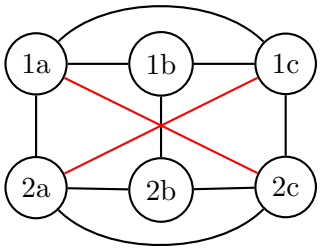


Abbildung 2.9: Der vollständige Assoziationsgraph von G_1 und G_2

Für einen AG ergibt sich somit die folgende (auf der Beschreibung in [15] basierende) Definition:

Definition 2.13 (Assoziationsgraph) *Gegeben seinen zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$. Ein Assoziationsgraph $G_A(G_1, G_2) = (V_A, E_A)$ sei wie folgt definiert:*

- $V_A = V_1 \times V_2$
- $E_A = \{((u_1, u_2), (v_1, v_2)) \mid u_i = v_i \vee ((u_1, v_1) \in E_1 \oplus (u_2, v_2) \in E_2)\}$[3]

[3] $a \oplus b \equiv (a \wedge \neg b) \vee (\neg a \wedge b)$ – „Entweder a oder b“

Jedes independent set im AG von G_1 und G_2 stellt nun einen gemeinsamen induzierten Teilgraphen von G_1 und G_2 dar [15]. Somit entspricht das MIS des AGs dem MCS von G_1 und G_2.

Kapitel 3

Algorithmen

Dieses Kapitel beschäftigt sich mit verschiedenen Ansätzen, um den Graphabstand zu ermitteln. Im ersten Abschnitt wird versucht, einen gemeinsamen induzierten Teilgraphen zu finden. Es wird dabei nicht in den ursprünglichen Graphen gesucht sondern in deren Kantengraphen. Der zweite Abschnitt beschäftigt sich mit der direkten Suche nach einem ECGM in den ursprünglichen Graphen. Dies entspricht der Suche nach dem größten gemeinsamen Teilgraph, der jedoch nicht induziert sein muss. Algorithmen, die direkt nach einem Graphabstand suchen, werden im dritten Abschnitt vorgestellt.

3.1 Algorithmen für MCS

Die Algorithmen in diesen Abschnitt suchen nach einem MCS. Dabei wird nicht der MCS der Originalgraphen ermittelt, sondern der Kantengraphen. Dies wird gemacht, weil der MCS ein induzierter Teilgraph ist und es passieren kann, dass Knoten ungewollt als fehlerhaft oder fehlend angesehen werden (siehe Abschnitt 2.6).

3.1.1 McGregor-Algorithmus

Der McGregor-Algorithmus ist ein einfacher Backtracking-Algorithmus. Er wurde 1982 erstmalig veröffentlicht und liefert eine exakte Lösung. Er wurde unter anderem auch in [4] betrachtet.

Arbeitsweise

Das Backtracking wird mittels Rekursion realisiert. Bei jedem Rekursionsaufruf ist bereits ein gemeinsamer Teilgraph vorhanden, wobei es sich dabei am Anfang um einen leeren Graphen handelt.

Nun werden von den noch möglichen Knotenpaaren der beiden Ursprungsgraphen alle durchprobiert. Dabei wird überprüft, ob es sich

```
Procedure McGregor(comSubgraph)
    For Each (u,v) ∈ V1 × V2
        If comSubgraph.IsAddablePair(u,v) Then

            comSubgraph.AddPair(u,v)

            If comSubgraph.PairCount > max.PairCount Then
                max = comSubgraph.Clone()
            End If

            // Gibt es in beiden Eingabegraphen Knoten,
               die nicht zum Teilgraphen gehören?
            If comSubgraph.IsExpandable Then
                McGregor(comSubgraph)
            End If

            // Backtracking
            comSubgraph.RemovePair(u,v)
        End If
    End For
End Procedure
```

Listing 3.1: McGregor-Algorithmus

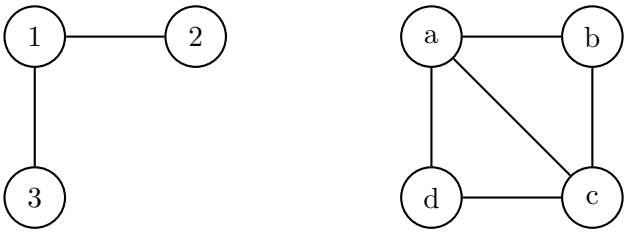
weiterhin um einen induzierten Teilgraphen handelt, wenn das aktuelle Knotenpaar zum bisher ermittelten Teilgrpahen hinzugefügt wird. Dies ist dann der Fall, wenn zwischen jedem bisherigen Knotenpaar (u_i, v_i) sowie dem aktuellen Paar (u, v) entweder in beiden Eingabegraphen eine Kante ist, oder in keinem der Graphen eine Kante ist
$((u_i, u) \in E_1 \Leftrightarrow (v_i, v) \in E_2)$. Außerdem darf keiner der Knoten bereits für ein Knotenpaar des aktuellen Teilgraphen benutzt worden sein.

Erfüllt ein Knotenpaar diese Bedingungen (is addable), dann wird es zum aktuellen Teilgrpahen hinzugefügt. Ist der Teilgraph nun größer als der bisher größte, wird er gespeichert. Als nächstes wird überprüft, ob der Teilgraph weiterhin vergrößert werden kann. Dies ist dann der Fall, wenn es in beiden Eingabegraphen noch Knoten gibt, die nicht zum gemeinsamen Teilgraph gehören. Lässt sich der Teilgraph vergrößern (is expandable), dann erfolgt ein rekursiver Aufruf, wobei der aktuelle Teilgraph übergeben wird. Abschließend wird das aktuelle Paar wieder

vom Teilgraphen entfernt, damit das nächste Paar überprüft werden kann.

Beispiel

Gegeben seien die Graphen in Abbildung 3.1. Für sie soll nun der MCS mittels McGregor-Algorithmus entwickelt werden.

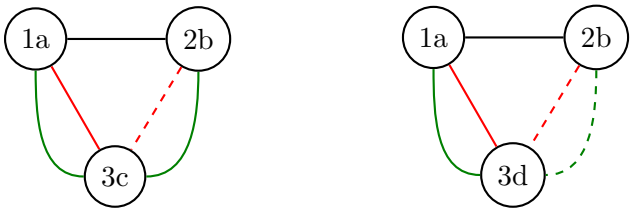


(a) Der Graph G_1 (b) Der Graph G_2

Abbildung 3.1: Die Graphen G_1 und G_2

Es folgt nun die Betrachtung eines einzelnen Aufrufs der McGregor-Prozedur, wobei schon ein Teilgraph (*comSubgraph*) vorgegeben ist. Dabei stellen *rote* Linien Kanten in G_1 dar und *grüne* Linien Kanten in G_2. Ist eine Linie gestrichelt, dann bedeutet dies, dass zwischen beiden Knoten keine Kante vorhanden ist.

Die Paare $(1, a)$ und $(2, b)$ sind bereits als Teilgraph vorgegeben. Als mögliche Paare aus V_1 und V_2, deren Knoten noch nicht verwendet wurden, bleiben noch $(3, c)$ und $(3, d)$. Abbildung 3.2 stellt die Überprüfung dar.



(a) (b)

Abbildung 3.2: Überprüfung von Paaren

Das Paar $(3, c)$ wird zuerst überprüft. Es kann jedoch nicht zum gemeinsamen Teilgraphen hinzugefügt werden, weil es eine Kante (b, c) in G_2 gibt, jedoch keine Kante $(2, 3)$ in G_1.

Als nächstes überprüft der Algorithmus das Paar $(3, d)$. Die Kanten $(1, 3)$

und (a, d) sind in G_1 und G_2 vorhanden. Außerdem sind weder $(2, 3)$ noch (b, d) in G_1 bzw. G_2 vorhanden. Somit kann $(3, d)$ zum Teilgraphen hinzugefügt werden.

Nach dem Hinzufügen des Paars zum Teilgraphen, wird der neue gemeinsame Teilgraph mit dem bisherigen größten verglichen und gegebenenfalls gespeichert. Im nächsten Schritt wird überprüft, ob der Teilgraph erweitert werden kann. Dies ist nicht der Fall, da bereits alle Knoten aus G_1 verbraucht wurden. Zuletzt wird das Paar $(3, d)$ wieder entfernt, um andere Möglichkeiten zu probieren (die es in diesem Beispiel jedoch nicht gibt).

3.1.2 MIS-basierter Algorithmus

Der folgende Algorithmus basiert auf einem Verfahren, das in [15] vorgestellt wurde. Er arbeitet mit dem Assoziationsgraphen der zu vergleichenden Graphen und sucht dort ein MIS (siehe Abschnitt 2.7).

Arbeitweise

Im ersten Schritt baut der Algorithmus den Assoziationsgraphen für die gegebenen Graphen auf. Als nächstes wird im Assoziationsgraph nach einem independent set gesucht. Dies erfolgt über rekursiv implementiertes Backtracking. Bei jedem Rekursionsaufruf ist bereits ein Menge an gewählten Knoten ($selectedNodes$), die ein independent set darstellen, vorhanden, wobei es sich dabei am Anfang um eine leere Menge handelt.

Bei jedem Rekursionsaufruf wird nun nacheinander jeder Knoten v, der nicht zur Menge der blockierten Knoten ($blockedNodes$) gehört, zur aktuellen Knotenmenge hinzugefügt. Anschließend wird v ebenfalls blockiert.

Ein Knoten wird immer dann blockiert, wenn er bereits zum aktuellen independent set gehört oder er nicht zur aktuellen Knotenmenge hinzugefügt werden kann, weil sie dann kein inependent set mehr ist.

Da ein independent set sich dadurch definiert, dass es keine Kanten besitzt, schließt die Wahl von v auch alle Knoten u aus, die über eine Kante mit v verbunden sind. Auch sie werden blockiert.

Nun erfolgt der rekursive Aufruf. Abschließend wird v wieder aus dem

```
Procedure MaxIndSet(G_1,G_2)

    (V_A,E_A) = AssoziationsGraph(G_1,G_2)
    max = ∅

    // aktuelles independent set
    selectedNodes = ∅

    // Menge der blockierten Knoten
    blockedNodes_0 = ∅

    MIS(1)

End Procedure

Procedure MIS(i)

    // Blockierung übernehmen.
    blockedNodes_i = blockedNodes_{i-1}.Clone()

    // Alle unblockierten Knoten testen.
    For Each v ∈ V_A\blockedNodes_i

        selectedNodes = selectedNodes ∪ {v}
        blockedNodes_i = blockedNodes_i ∪ {v}

        // Jeden Nachbarknoten von v blockieren.
        For Each (v,u) ∈ E_A
            blockedNodes_i = blockedNodes_i ∪ {u}
        End For

        If |selectedNodes| > |max| Then
            max = selectedNodes
        End If

        MIS(i+1)

        // Backtracking
        selectedNodes = selectedNodes\{v}

    End For
End Procedure
```

Listing 3.2: MIS-basierter Algorithmus

aktuellen independent set entfernt. Auch die Blockierungen der Knoten müssen wieder aufgehoben werden. In der in Listing 3.2 dargestellten Umsetzung ist dies dadurch gelöst, dass es für jede Rekursionstiefe eine separate Menge von blockierten Knoten gibt. Diese lassen sich jeweils über einen Index ansprechen.

Anpassung an einen Kantengraph

Handelt es sich bei den gegebenen Graphen um Kantengraphen, so lässt sich der Assoziationsgraph A vereinfachen. Jeder Knoten in A steht dann für eine Zuordnung eines Knotens von $k(G_1)$ zu einem Knoten aus $k(G_2)$. Es ist jedoch nicht möglich, zwei Knoten einander zuzuordnen, wenn einer der Knoten einen Knoten in G_1 darstellt und der andere eine Kante in G_2. Sämtliche Knoten, die eine solche Zuordnung darstellen, können somit aus dem Assoziationsgraph entfernt werden.

3.1.3 Weitere Verfahren

Es existieren noch weitere Verfahren, um einen MCS zweier Graphen zu ermitteln. Aufgrund der verwendeten Eigenschaften, ist jedoch vermutlich keine bessere Laufzeit zu erwarten.

In [1] wird ein Verfahren vorgestellt, welches mit dem Vertex Cover einer der Graphen arbeitet. Ein Vertex Cover eines Graphen ist eine Knotenmenge, bei der jede Kante des Graphen an mindestens einem Ende mit einem der Knoten verbunden ist. Ist ein Vertex Cover gefunden, wird anschließend mit dem Vertex Cover statt dem gesamten Gaphen weitergearbeitet. Insgesamt wird das ursprüngliche Problem auf drei kleinere Probleme aufgeteilt. Dies ist vermutlich[1] der Grund für die deutlich geringe Laufzeit gegenüber oben genannten Verfahren.

Dass dieses Verfahren bei einem Kantengraph den gleichen Unterschied in der Laufzeit zeigt, ist nicht zu erwarten. Das minimale Vertex Cover eines Kantengraphs entspricht der Menge der Knoten des ursprünglichen Graphen[2]. Die Folge ist, dass das Problem nicht aufgeteilt wird. Es bleibt

[1] [1] geht nicht ausreichend darauf ein.

[2] Dies gilt, falls der Graph mehr Kanten als Knoten besitzt und zusammenhängend ist. Besitzt der Graph mehr Knoten als Kanten, so ergibt sich das minimale Vertex Cover für den Kantengraph aus der Menge der Kanten. Bei nicht zusammenhängenden Graphen kann jeder zusammenhängende Teilgraph separat betrachtet werden.

in seiner ursprünglichen Größe vorhanden.

3.2 Direkte Suche nach einem ECGM

Betrachtet man einen Kantengraph statt einem normalen, so bringt dies Vorteile. Zum einen lässt sich aus dem MCS der Kantengraphen auch ein ECGM der ursprünglichen Graphen ableiten. Zum anderen bietet sich die Möglichkeit, gemeinsame freie Kanten zu erhalten, auch wenn die dazugehörigen Knoten nicht Teil eines gemeinsamen Teilgraphen sind.

Angenommen, eine solche freie Kante enthält keine zusätzlichen nutzbringenden Informationen. In einem solchen Fall kann man direkt nach einem ECGM suchen. Die Algorithmen in diesem Abschnitt ermitteln ein solches ECGM.

3.2.1 Vorüberlegungen

Der Graphabstand zweier Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ ergibt sich aus den Kosten für das Entfernen und Hinzufügen ungleicher Elemente. Folglich ist der Graphabstand minimal, wenn die Menge gemeinsamer Elemente möglichst groß ist. Diese Menge wird durch das ECGM $\varphi : \hat{V}_1 \to \hat{V}_2$ dargestellt. Dabei gelte $|V_1| \leq |V_2|$.

Satz 3 *Gegeben seien zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ mit $|V_1| \leq |V_2|$, ein ECGM $\varphi : \hat{V}_1 \to \hat{V}_2$ und dessen Kostenfunktion $c(\varphi)$ mit $c_{nd}(u) + c_{na}(v) > 0$ für alle $(u, v) \in V_1 \times V_2$. Ist $c(\varphi)$ minimal, dann ist $\hat{V}_1 = V_1$.*

Beweis *Angenommen, $c(\varphi)$ sei minimal, jedoch sei $\hat{V}_1 \subset V_1$. Dann existiert ein Knoten $u \in V_1 \backslash \hat{V}_1$, sowie ein Knoten $v \in V_2 \backslash \hat{V}_2$.*

Es sei nun $\lambda : \hat{V}_1 \cup \{u\} \to \hat{V}_2 \cup \{v\}$ ein weiteres ECGM. Dabei gelte:

$$\lambda(x) = \begin{cases} v & x = u \\ \varphi(x) & sonst \end{cases}$$

Die Kosten für λ ergeben sich nun aus den Kosten von φ, jedoch entfallen die Kosten für das Hinzufügen und Entfernen der Knoten u und v.

$$c(\lambda) = c(\varphi) - (c_{nd}(u) + c_{na}(v)).$$

Da $c_{nd}(u) + c_{na}(v) > 0$ ist, gilt somit $c(\lambda) < c(\varphi)$. Somit kann $c(\varphi)$ nicht minimal sein. □

Satz 3 sagt aus, dass in einem ECGM mit minimalen Kosten alle Knoten des kleineren Graphen G_1 vorhanden sind. Ein ECGM lässt sich somit als Permutation der Knoten in V_2 mit der Größe $|V_1|$ darstellen: Steht ein Knoten v aus V_2 an i-ter Position in der Permutation, wird er dem i-ten Knoten aus V_1 zugeordnet. Ist v nicht in der Permutation vorhanden, dann wird ihm kein Knoten zugeordnet.

Bewertung

Die Bewertung einer Permutation erfolgt über die Kanten. Dazu werden die Knotenpaare (u_1, v_1) aus G_1 mit den dazugehörigen Paaren $(u_2, v_2) = (\varphi(u_1), \varphi(v_1))$ aus G_2 verglichen. Befindet sich zwischen beiden Paaren eine Kante, dann wird (u_1, v_1) als gleiches Paar mit Kante gezählt. Ist weder zwischen (u_1, v_1) noch zwischen (u_2, v_2) eine Kante, handelt es sich lediglich um ein gleiches Paar ohne Kante. Alle anderen Paare werden als ungleich betrachtet.

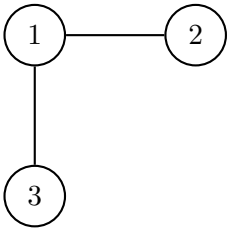
Eine Permutation P gilt als größer als eine Permutation Q, wenn die Anzahl der gleichen Paare mit Kante in P größer ist als in Q. Ist die Anzahl in beiden Permutationen gleich, so ist die Anzahl der gleichen Paare ohne Kante entscheidend.

Beispiel

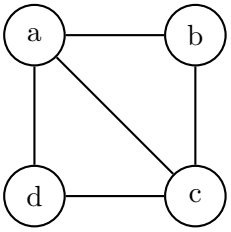
Als Beispiel seien erneut die Graphen aus Abschnitt 3.1.1 gegeben (Abbildung 3.3). Da G_1 der kleinere der beiden Graphen ist, wird eine dreielementige Permutation der Knoten aus G_2 gesucht. Insgesamt gibt es 24 mögliche Permutationen, von denen nun zwei betrachtet werden.

Die Darstellung der Permutationen erfolgt analog zu Abschnitt 3.1.1. Paare aus G_1 sind *rot*, Paare aus G_2 *grün*. Besitzen Paare eine Kante, so ist die dazugehörige Linie durchgezogen. Ist keine Kante vorhanden, dann ist die Linie gestrichelt.

Die Permutation $P_1 = (a, b, d)$ ist in Abbildung 3.4a dargestellt. Die Paare $(1, 2)$, $(1, 3)$, (a, b) und (a, d) besitzen alle eine Kante. Die Paare $(2, 3)$ und (b, d) haben beide keine Kante. Somit hat P_1 zwei gemeinsame Paare mit

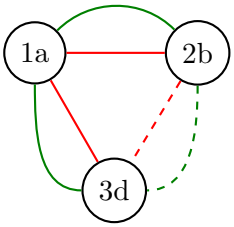


(a) Der Graph G_1

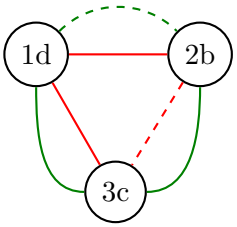


(b) Der Graph G_2

Abbildung 3.3: Die Graphen G_1 und G_2



(a) Permutation P_1



(b) Permutation P_2

Abbildung 3.4: Überprüfung von Paaren

Kanten und ein gemeinsames Paar ohne Kante.

Die Permutation $P_2 = (d, b, c)$ ist in Abbildung 3.4b dargestellt. Die Paare $(1, 3)$ und (a, c) besitzen beide eine Kante. Die Paare $(1, 2)$ und (d, b) sowie $(2, 3)$ und (b, c) haben entweder eine Kante in G_1 oder in G_2, jedoch nicht in beiden gleich. Somit hat P_2 nur ein gemeinsames Paar mit Kante.

3.2.2 Probieren aller Permutationen

Die einfachste Variante, die beste Permutation zu finden, ist es, alle Permutation zu probieren. Die Permutationen können iterativ erzeugt werden. Dabei wird eine Permutation direkt aus ihrem Vorgänger gebildet, bis die letzte Permutation erreicht ist.

Eine Alternative zu einer iterativen Erzeugung von Permutationen ist es, alle Möglichkeiten rekursiv zu durchsuchen. Dadurch wird ein Suchbaum erzeugt, dessen Blätter eine Permutation darstellen. Ein solches Verfahren kann schneller als ein iteratives sein, wenn das Erzeugen einer Permutation aus einer bestehenden rechenaufwendig wird. Die Komplexität ändert sich jedoch nicht, denn die Anzahl der Permutationen (und somit die Anzahl

```
Procedure Permut(G1,G2)
    P = firstPermut(V1,V2)
    Pmax = P
    Do Until isLastPermut(P)
        If P > Pmax Then
            Pmax = P
        End If
        P = nextPermut(P,V1,V2)
    End Do
End Procedure
```

Listing 3.3: Probieren aller Permutationen

der Blätter im Suchbaum) bleibt gleich.

3.2.3 Ein Branch-and-Bound-Verfahren

Bei einem Branch-and-Bound-Verfahren (B&B) wird ein Suchbaum aufgebaut, dessen Blätter eine mögliche Lösung darstellen. Die Suchstrategie innerhalb des Baums unterscheidet sich jedoch von einer Tiefen- oder Breitensuche. Das Verfahren wurde erstmals 1960 in [10] vorgestellt.

Arbeitsweise

Jeder Knoten erhält einen Wert, der eine obere bzw. untere (abhängig davon, ob ein Maximum oder Minimum gesucht wird) Schranke für alle Lösungen des Teilbaums angibt. Das bedeutet, dass keine der Lösungen in dem Teilbaum besser sein kann, als am aktuellen Knoten abgeschätzt wird. Bei der Wurzel des Baums beginnend wird der Suchbaum schrittweise an dem Knoten erweitert, der die beste Lösung verspricht. Ist der Knoten mit dem besten Wert ein Blatt, dann stellt er eine optimale Lösung dar.

Der Vorteil eines B&B-Verfahrens ist, dass nicht der gesamte Baum durchsucht werden muss. Aufgrund der Schranken müssen bestimmte Teilbäume nicht durchsucht werden. Nachteilig ist allerdings ein erhöhter Speicheraufwand. Jeder Knoten, der im nächsten Schritt expandiert werden kann, muss gespeichert werden. Ob der Vorteil oder der Nachteil überwiegt, hängt stark von der gewählten Schranke ab. Bildet sich eine deutliche Lösung heraus, wird der Baum nur selten erweitert. Somit sind

```
Procedure BB()

    pQ = New PriorityQueue()
    pQ.Enqueue(rootNode)

    Do Until isLeaf(pQ.Top())
        top = pQ.Dequeue()

        For Each node In expand(top)
            pQ.Enqueue(node)
        End For
    End Do
End Procedure
```

Listing 3.4: Prinzip des Branch-and-Bound-Algorithmus

Zeit- und Speicheraufwand gering. Sind alle Knoten etwa gleichwertig, muss der Baum häufiger erweitert werden. Im ungünstigsten Fall wird der Suchbaum fast komplett aufgespannt. Sowohl Zeit- wie Speicheraufwand sind dann sehr hoch.

Beispiel

Als Beispiel sei der Suchbaum in Abbildung 3.5 gegeben. In diesem soll das Minimum ermittelt werden. Es wird also das Blatt mit minimalem Wert gesucht. Der Wert der Knoten, die keine Blätter sind, gibt an, wie hoch die Werte der entsprechenden Blätter mindestens sind. Dieser Wert entspricht also einer unteren Schranke.

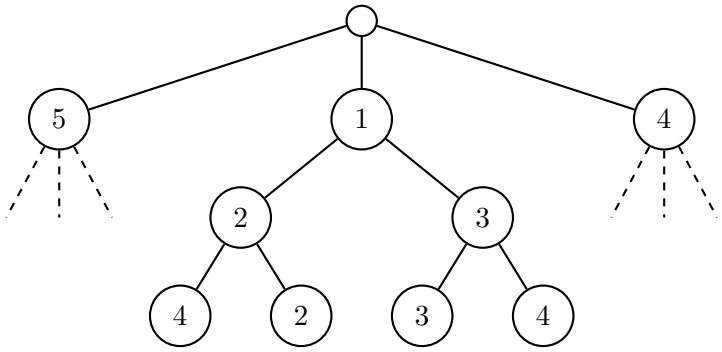


Abbildung 3.5: Beispiel-Suchbaum für das B&B-Verfahren

Abbildung 3.6 zeigt den Verlauf des Verfahrens für diesen Suchbaum. Die Knoten, die zurzeit betrachtet werden, sind *schwarz* dargestellt. Der Knoten mit dem minimalen Wert ist zusätzlich *grün* eingefärbt. Knoten, die bereits betrachtet wurden und nicht mehr im Speicher sind, sind ausgegraut.

Begonnen wird mit den Unterknoten der Wurzel: (5), (1) und (4). Da (1) das Minimum ist, wird an dieser Stelle erweitert. Dazu werden die Unterknoten zur Menge der aktuellen Knoten hinzugefügt und (1) entfernt.

Im nächsten Schritt wird der Knoten (2) erweitert. Im Speicher befinden sich danach die Knoten (5), (4), (5), (3) und (4).

Zuletzt wird der Knoten (3) erweitert, wodurch dieser entfernt wird und die Knoten (3) und (4) hinzugefügt werden. Das Minimum ist somit erneut (3). Da es sich dabei um ein Blatt handelt, ist dieser Knoten auch ein Minimum. Die Suche wird somit beendet.

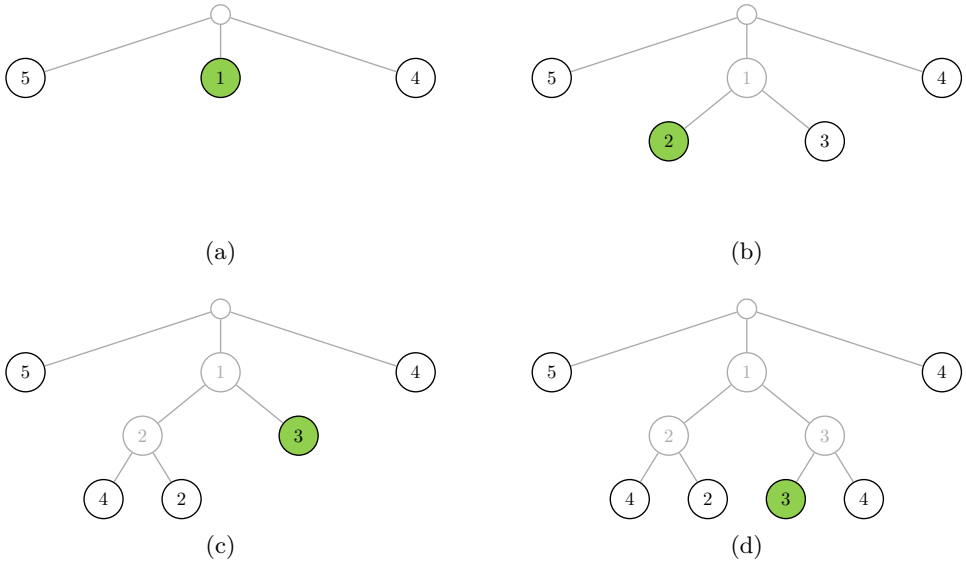


Abbildung 3.6: Suche mittels B&B-Verfahren

Ermitteln des ECGMs

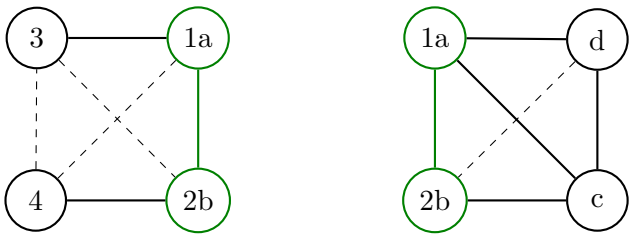
Jeder Knoten des Suchbaums beschreibt eine ECGM bzw. einen gemeinsamen Teilgraphen der Graphen G_1 und G_2. Der Wurzelknoten stellt einen leeren Graphen dar. Dessen Unterknoten entsprechen einem

Graphen mit einem Knoten. Sie geben an, worauf der erste Knoten aus G_1 abgebildet wird. Die Blätter des Suchbaums geben dann einen Teilgraphen an, bei dem jeder Knoten aus G_1 enthalten ist.

Ist ein Knoten kein Blatt, ist der durch ihn dargestellte Teilgraph somit kleiner als G_1. Folglich gibt es in beiden Graphen Knoten, die noch nicht einander zugeordnet wurden, also auch mögliche Paare, die nicht bewertet wurden. Die Anzahl der Paare mit Kanten und die Anzahl der Paare ohne Kanten beider Graphen werden nun miteinander verglichen. Die jeweils kleinere Zahl ist somit die maximale Anzahl an möglichen gemiensamen Paaren. Es wird also angenommen, dass es zu jedem Paar mit oder ohne Kante im unbetrachteten Teil von G_1 auch ein Paar mit oder ohne Kante im unbetrachteten Teil von G_2 gibt. Zusammen mit der Anzahl der gleichen Paare im betrachteten Teil bildet dies die Schranke. Beim Vergleich zweier Schranken sind in erster Linie die Paare mit Kante entscheidend. Ist ihre Zahl gleich, werden zusätzlich Paare ohne Kante betrachtet.

Beispiel

Gegeben seien die Graphen G_1 und G_2 (Abbildung 3.7). Angenommen bei der Suche nach einem ECGM mittels B&B-Verfahrens seien bereits die Zuweisungen $(1, a)$ und $(2, b)$ gegeben (grün dargestellt). Es soll nun ermittelt werden, wie viele gemeinsame Paare (mit oder ohne Kante) es maximal gibt. Paare ohne Kante sind durch gestrichelte Linien dargestellt.



(a) Der Graph G_1 (b) Der Graph G_2

Abbildung 3.7: Die Graphen G_1 und G_2 mit einem gemeinsamen Knotenpaar

Der Graph G_1 besitzt zwei Paare mit Kante: $(1a, 3)$ und $(2b, 4)$. Außerdem besitzt er drei Paare ohne Kante: $(1a, 4)$, $(2b, 3)$ und $(3, 4)$. G_2 hingegen besitzt vier Paare mit Kante $((1a, d)$, $(1a, c)$, $(2b, c)$ und $(c, d))$ sowie ein Paar ohne Kante $((2b, d))$. Von beiden Varianten (mit und ohne Kante)

wird nun das kleinere genommen, also zwei Paare mit und ein Paar ohne Kante. Somit ergibt sich (nach hinzurechnen des bereits vorhanden Paares $(1a, 2b)$) als obere Schranke: drei Paare mit und ein Paar ohne Kante. Bei keiner Kombination der Knoten ist mehr möglich.

3.2.4 Ein evolutionärer Algorithmus

Evolutionäre Algorithmen basieren auf dem Prinzip der biologischen Evolution. Dazu betrachtet man eine Population möglicher Lösungen. Jede Lösung ist ein Individuum, welches einen Wert besitzt, der die Qualität der Lösung beschreibt. Es gibt drei Mechanismen für die Suche nach einer besseren Lösung: Selektion, Rekombination und Mutation.

Selektion ist schlicht eine Auswahl der qualitativ besten Lösungen. Zu schlechte Lösungen werden aus der Population entfernt. Sie „sterben“. Mutiert ein Individuum, wird es leicht (zufällig) geändert. Bei der Rekombination zweier Individuen wird ein neues Individuum auf Basis der beiden bestehenden erzeugt. Das Neue übernimmt dabei die Eigenschaften, die beide Eltern gemeinsam hatten. Alle drei Mechanismen werden nun nacheinander ausgeführt, bis eine Abbruchbedingung erfüllt ist.

```
Procedure Evolution()
    initialisiere P
    Do Until Abbruchbedingung
        Rekombination(P)
        Mutation(P)
        Selektion(P)
    End Do
End Procedure
```

Listing 3.5: Prinzip eines evolutionären Algorithmus

Der Vorteil eines evolutionären Algorithmus ist, dass man kaum Wissen über das zu lösende Problem benötigt. Kann man eine mögliche Lösung bewerten und weitere mögliche Lösungen erzeugen, dann lässt sich auch ein evolutionärer Algorithmus implementieren. Erfahrungsgemäß erzeugen evolutionäre Algorithmen dabei eine gute Lösung. Für das Finden eines guten ECGMs werden lediglich Permutationen betrachtet. Zwar ist für die Bewertung jeder Permutation wichtig, wie die dazugehörigen Graphen aufgebaut sind, aber für Mutation und Rekombination wird lediglich die

Permutation betrachtet. Es findet kein Bezug zu den Graphen mehr statt.

Der Nachteil eines evolutionären Algorithmus ist jedoch, dass er kein exaktes Verfahren ist. Es lässt sich keine Aussage über die Qualität einer Lösung treffen. Bei einem endlichen Lösungsraum lässt sich eine Mindestwahrscheinlichkeit dafür angeben, dass die optimale Lösung gefunden wurde. Es ist jedoch im Allgemeinen nicht möglich zu sagen, ob eine Lösung ein Optimum ist oder wie weit sie vom Optimum höchstens entfernt ist.

Individuen

Ein Individuum stellt eine mögliche Lösung dar. Für die Suche nach einem ECGM zweier Graphen kann also eine Permutation der Knoten als Individuum betrachtet werden. Als Bewertung für die Selektion dient erneut die Anzahl der gleichen Paare.

Umsetzung der Mutation

Eine Mutation stellt eine kleine Veränderung des Individuums dar. Um eine Permutation zu mutieren, werden in der hier verwendeten Implementation lediglich die Abbildungen zweier zufälliger Knoten vertauscht.

Umsetzung der Rekombination

Bei der Rekombination zweier Permutationen werden gleiche Abbildungen übernommen. Alle anderen Abbildungen werden zufällig verteilt. Dadurch ist es bei Graphen unterschiedlicher Größe möglich, auch Knoten des größeren Graphen zur Lösungsmenge hinzuzufügen, die bisher nicht zum gemeinsamen Teilgraph gehörten.

3.3 Graphabstand-Algorithmen

Die Algorithmen in diesem Abschnitt werden in [12] als Algorithmen zur Ermittlung des Graphabstands beschrieben.

3.3.1 Der A*-Algorithmus

Der A*-Algorithmus ist ursprünglich zur Pfadsuche in Graphen gedacht. Er wurde 1968 in [8] vorgestellt. 1972 erfolgte eine leichte Korrektur in [9]. „Auf dem A*-Algorithmus basiert eine weit verbreitete Methode“ [12] zur

Berechnung des exakten Graphabstands.

Arbeitsweise

Bei einer Pfadsuche beginnt der A*-Algorithmus am Startpunkt des Graphen. Dieser wird als bekannt markiert. Für jeden bekannten Knoten gibt es eine Menge möglicher unmarkierter Folgeknoten, die über eine Kante erreichbar sind. Der Folgeknoten, der den kürzesten Pfad zum Ziel verspricht, wird als nächstes markiert. Dies wiederholt sich so lange bis der Zielknoten erreicht ist.

Um zu entscheiden, welcher Knoten v als nächstes markiert wird, gibt es zu jedem eine Abschätzung $f(v) = g(v) + h(v)$. Dabei gibt $g(v)$ die (bekannten) minimalen Kosten zum Erreichen von v an. Eine Abschätzung für die noch entstehenden Kosten gibt $h(v)$ an. Gewählt wird dann der Knoten, dessen $f(v)$ am kleinsten ist. Wichtig ist, dass $h(v)$ die minimalen Kosten angibt. Werden die Kosten höher angegeben als sie wirklich sind, kann dies zur Folge haben, dass der kürzeste Pfad nicht erkannt wird.

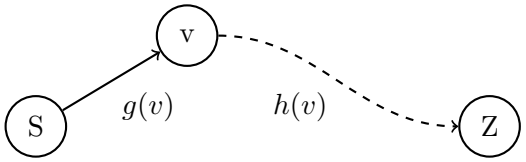


Abbildung 3.8: Prinzip des A*-Algorithmus

Der A*-Algorithmus kann beispielsweise verwendet werden, um in einem Straßennetz die kürzeste Strecke zwischen zwei Städten zu finden. Die Funktion $g(v)$ gibt dabei die Strecke an, die tatsächlich gefahren werden muss, um eine Stadt zu erreichen. Als Abschätzung für $h(v)$ kann die Luftlinie zwischen zwei Städten benutzt werden, denn die direkte Verbindung ist immer kürzer als die tatsächliche Straßenlänge.

Anpassung an Graphabstand

Zur Ermittlung des Graphabstands zweier Graphen G_1 und G_2 muss der Algorithmus leicht angepasst werden. Der Graph, in dem ein Pfad gesucht wird, ist der Suchbaum zur Ermittlung eines ECGMs. Startknoten ist die Wurzel des Baums. Zielknoten sind die Blätter. Der Wurzelknoten stellt ein „leeres“ ECGM dar. Seine Unterknoten entsprechen nun allen

möglichen Abbildungen des ersten Knotens aus G_1. Dazu gehört auch das Löschen des Knotens. [3]

Aus dem Teil-ECGM, das durch einen Knoten v des Suchbaums dargestellt wird, berechnen sich die bisherigen Kosten $g(v)$. In der in [12] vorgestellten Variante ergeben sie sich aus den Kosten für das Hinzufügen und Entfernen von Kanten. Die noch anfallenden Kosten $h(v)$ berechnen sich aus dem Vergleich der Kanten, die noch nicht im aktuellen Teil-ECGM sind. Dabei wird ermittelt, wie viele Kanten mindestens entfernt bzw. hinzugefügt werden müssen.

Vergleich mit Branch-and-Bound

Vergleicht man den A*- mit dem B&B-Algorithmus, stellt man eine starke Ähnlichkeit fest. Beide Verfahren benutzen die gleiche Strategie: In einem zu durchsuchenden Graphen (bei B&B immer ein Suchbaum) werden die bisher erreichten Knoten miteinander verglichen. An dem Knoten, dessen Summe aus den exakten und den erwarteten Kosten minimal ist, wird die Suche fortgesetzt. Der A*-Algorithmus ist dabei lediglich für eine Minimierung gedacht. Der B&B-Algorithmus ist etwas allgemeiner formuliert. Es ist auch für eine Maximierung ausgelegt. Dies ist jedoch kein relevanter Unterschied. Beim B&B-Algorithmus lassen sich Minimierung und Maximierung ineinander umwandeln, indem man die Kosten bzw. die Werte mit -1 multipliziert.

Die in [12] beschriebene Methode zur Berechnung einer unteren Schranke für den Graphabstad zweier Graphen basiert auf dem gleichen Prinzip wie die Berechnung der Schranke in Abschnitt 3.2.3. Es wird die Anzahl der verbliebenen Kanten im unbetrachteten Teil der Graphen miteinander verglichen. Das oben vorgestellte B&B-Verfahren ermittelt dabei die Anzahl der gemeinsamen Kanten. Die in [12] vorgestellte Variante zur Berechnung einer unteren Schranke (der noch anfallenden Kosten $h(v)$) berechnet die Zahl der zu löschenden Kanten. Addiert man beides, erhält man somit die Gesamtzahl der Kanten. Da diese für ein Graphenpaar konstant ist, sind auch beide Schranken äquivalent zueinander. Aus diesem Grund wird in Kapitel 4 lediglich der B&B-Algorithmus betrachtet.

[3] Aufgrund von Satz 3 (Abschnitt 3.2.1) ist es nicht nötig, das Löschen des Knotens zu betrachten, wenn G_1 der kleinere der beiden Graphen ist. Es werden dann sämtliche Knoten aus G_1 übernommen.

3.3.2 Bipartite Heuristik

Eine mögliche Variante, um den Graphabstand zweier Graphen abzuschätzen, ist ein Matching auf einem bipatiten Graphen.

Grundlagen

Ein bipartiter Graph zeichnet sich dadurch aus, dass sich dessen Knotenmenge in zwei Teilmengen unterteilen lässt, wobei Kanten immer jeweils einen Knoten der einen mit einem Knoten der anderen Teilmenge verbinden. Zwischen den Knoten innerhalb einer Teilmenge gibt es keine Kanten.

Definition 3.1 (bipartiter Graph) *Ein gewichteter bipartiter Graph G ist ein 3-Tupel $G = (V, E, \omega)$. Dabei sind:*

- $V = V_1 \cup V_2$ *mit* $V_1 \cap V_2 = \emptyset$ *eine endliche Menge an Knoten,*
- $E \subseteq V_1 \times V_2$ *die Menge der Kanten und*
- $\omega : E \to \mathbb{Q}$ *die Gewichtsfunktion der Kanten.*

G ist voll *genau dann, wenn $E = V_1 \times V_2$.*

Bei einem Matching in einem Graphen werden Knoten einander zugeordnet. Dabei müssen die Knoten über eine Kante miteinander verbunden sein. Diese Zuordnung ist eindeutig. Es wird also jedem Knoten nur maximal ein anderer Knoten zugeordnet.

Definition 3.2 (Matching) *Ein Matching in einem gewichtetem bipartiten Graphen $G = (V_1 \cup V_2, E, \omega)$ ist eine bijektive Abbildung $\mu : \hat{V_1} \to \hat{V_2}$ mit $\hat{V_1} \subseteq V_1$ und $\hat{V_2} \subseteq V_2$. Dabei gilt: $v \in \hat{V_1} \Rightarrow (v, \mu(v)) \in E$.*

Das Gewicht w eines Matchings ist dann:

$$w = \sum_{v \in \hat{V_1}} \omega((v, \mu(v)))$$

Arbeitsweise

Idee der Heuristik ist, dass man ein ECGM auch als Matching in einem bipartiten Graphen interpretieren kann. Gegeben seien die Graphen

$G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ mit $n = |V_1|$ und $m = |V_2|$. Daraus lässt sich nun ein voller bipartiter Graph $M = (V_M, E_M, \omega)$ wie folgt erstellen:

- $V_M = (V_1 \cup \{\varepsilon_{1_1}, \ \dots \ , \varepsilon_{1_m}\}) \cup (V_2 \cup \{\varepsilon_{2_1}, \ \dots \ , \varepsilon_{2_n}\})$
- $E_M = (V_1 \cup \{\varepsilon_{1_1}, \ \dots \ , \varepsilon_{1_m}\}) \times (V_2 \cup \{\varepsilon_{2_1}, \ \dots \ , \varepsilon_{2_n}\})$

Der eine Teil der Knoten in M ergibt sich aus den Knoten in G_1 sowie je einem ε-Knoten für jeden Knoten in G_2. Der zweite Teil bildet sich analog: es werden die Knoten aus G_2 genommen, sowie je ein ε-Knoten für jeden Knoten in G_1.

Die ε-Knoten dienen dazu, um das Hinzufügen oder Entfernen von Knoten ebenfalls als Matching zu ermöglichen. Wird ein Knoten hinzugefügt, so wird ihm ein ε-Knoten zugeodnet: $\varepsilon_{1_i} \mapsto v_i$. Beim Löschen ist es andersrum: $v_i \mapsto \varepsilon_{2_i}$.

Ein ECGM ist nun äquvalent zu einem Matching in M. Die Kosten c_{uv}, die mindestens entstehen, wenn zwei Knoten u und v einander zugewiesen werden, ergeben dann die Kantengewichte von M. Zur Berechnung der Kosten werden die Kanten von G_1 und G_2 mit einbezogen. Hat beispielsweise ein Knoten drei Kanten und wird auf einen Knoten mit fünf Kanten abgebildet, so müssen mindestens zwei Kanten entfernt oder hinzugefügt werden. Die Kosten können als Kosten-Matrix C dargestellt werden.

$$C = \left[\begin{array}{ccc|ccc} c_{11} & \cdots & c_{1m} & c_{1\varepsilon} & & \infty \\ \vdots & \ddots & \vdots & & \ddots & \\ c_{n1} & \cdots & c_{nm} & \infty & & c_{n\varepsilon} \\ \hline c_{\varepsilon 1} & & \infty & & & \\ & \ddots & & & 0 & \\ \infty & & c_{\varepsilon m} & & & \end{array}\right]$$

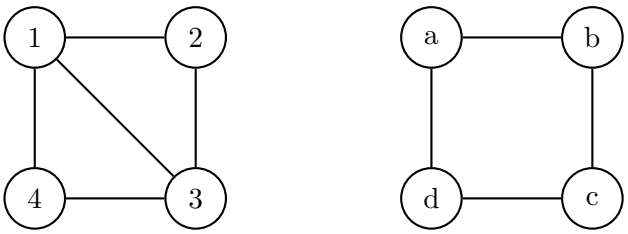
Die Matrix ist in vier Bereiche unterteilt. Oben links befinden sich die Kosten, die beim Zuweisen eines Knoten aus G_1 zu einem Knoten aus G_2 auftreten. Oben rechts und unten links sind die Kosten für das Hinzufügen oder Löschen von Knoten. Die Kosten stehen jedoch nur in den Diagonalen. Alle anderen Werte sind unendlich. Im unteren rechten Teil

sind alle Werte null. Sie stellen den Fall dar, dass zwei ε-Knoten einander zugewiesen werden.

Um nun ein ECGM zu finden, das möglichst geringe Kosten hat, wird das Matching gesucht, dessen Gewicht minimal ist. Dieses lässt sich durch ein Verfahren ermitteln, welches als Ungarische Methode[4] bekannt ist. Die Methode findet ein minimales Matching in $\mathcal{O}(n^3)$ [11].

Matching als untere Schranke

In [13] wird die Heuristik als mögliche Abschätzung für die mindestens noch entstehenden Kosten (h-Funktion) und somit als untere Schranke zur Berechnung des Graphabstands mittels eines A*-Algorithmus vorgeschlagen. Die Argumentation dabei ist, dass die Kantengewichte des bipartiten Graphen jeweils die minimalen Kosten für das Abbilden der Knoten aufeinander darstellen. Somit sei das minimale Gesamtgewicht eines Matchings auch die minimalen Kosten für den gesamten Graph. Bewiesen wird diese Behauptung nicht. Bei einer genauen Betrachtung stellt sich diese Aussage hingegen als inkorrekt heraus. Es kann passieren, dass Kosten für Kanten doppelt berechnet werden. Dies liegt daran, dass die Kosten an beiden Knoten der Kante erhoben werden.



(a) (b)

Abbildung 3.9: Graphen, die sich durch genau eine Kante unterscheiden

Die in Abbildung 3.9 dargestellen Graphen sind bis auf die Kante in der Mitte gleich. Der Graphabstand entspricht somit lediglich dem Hinzufügen oder Entfernen der Kante. Die Kosten werden allerding zweimal in die Kostenmatrix übernommen und auch zweimal in das Gesamtgewicht des Matchings eingerechnet.

Setzt man die Kosten für das Einfügen und Löschen von Knoten und

[4] Es ist auch unter Munkres Algorithmus oder Kuhn-Munkres-Algorithmus bekannt.

Kanten jeweils auf 1, so ergibt sich folgende Kostenmatrix C:

$$C = \left[\begin{array}{cccc|ccc} 1 & 1 & 1 & 1 & 3 & & \infty \\ 0 & 0 & 0 & 0 & & 2 & \\ 1 & 1 & 1 & 1 & & & 3 \\ 0 & 0 & 0 & 0 & \infty & & 2 \\ \hline 2 & & & \infty & & & \\ & 2 & & & & & \\ & & 2 & & & 0 & \\ \infty & & & 2 & & & \end{array}\right]$$

Ein minimales Matching ergibt sich nun aus folgenden Abbildungen:

$$\begin{array}{ccc} 1 & \longmapsto & a \\ 2 & \longmapsto & b \\ 3 & \longmapsto & c \\ 4 & \longmapsto & d \\ \varepsilon_{1,\ldots,4} & \longmapsto & \varepsilon_{a,\ldots,d} \end{array}$$

Die in C eingetragenen Kosten für $1 \mapsto a$ und $3 \mapsto c$ sind jeweils 1. Das minimale Gesamtgewicht ist somit 2. Der Graphabstand ist jedoch nur 1. Das Gewicht des Matchings liegt somit über den eigentlichen Kosten und kann keine untere Schranke sein.

3.4 Zusammenfassung

Es wurden nun fünf Algorithmen vorgestellt, um den Graphabstand zweier Graphen zu ermitteln. Diese verfolgen dabei unterschiedliche Ansätze. Zwei dieser Algorithmen erwiesen sich als so ähnlich, dass nur einer von ihnen weiter betrachtet wird. Zwei weitere Algorithmen sind lediglich inexakte Verfahren, bei denen nicht sicher ist, dass die ermittelte Lösung auch die beste ist.

Das nächste Kapitel vergleich nun diese Algorithmen und versucht Aussagen über deren Geschwindigkeit und (bei den inexakten Verfahren) über die Qualität der Lösungen zu treffen.

Kapitel 4

Algorithmentests

Dieses Kapitel versucht, Aussagen über die im vorherigen Kapitel beschriebenen Algorithmen zu treffen. Dabei geht es primär um die Laufzeit. Um eventuell Aussagen über mögliche Vor- und Nachteile oder besonderes Verhalten treffen zu können, wurden auch weitere Daten gesammelt.

4.1 Testverfahren

Zum Testen der Algorithmen wird eine Reihe zufällig erzeugter Graphen verwendet. Um die Qualität der Lösungen vergleichen zu können, verwenden alle Algorithmen dieselben Graphen.

4.1.1 Graph-Eigenschaften

Sämtliche Graphen sind zusammenhängend und ungerichtet. Weder Knoten noch Kanten besitzen eine Färbung, Gewichtung oder Beschriftung.

Die Graphen unterscheiden sich in zwei Eigenschaften voneinander. Dies sind die Anzahl der Knoten n und die durchschnittliche Zahl an Nachbarn eines Knotens d.

4.1.2 Durchführung

Für $n = 5$ bis $n = 20$ sowie $d = 3$ und $d = 4$ wird je ein Graph erzeugt. Für diese Graphen werden nun paarweise die ECGMs mit minimalen Kosten ermittelt. Dabei haben Paare mit einer geringeren Anzahl an möglichen ECGMs (also die Anzahl der Permutationen; siehe Abschnitt 3.2.1) Vorrang. Dies erlaubt es, Testreihen als Ganzes abzubrechen, wenn einzelne Tests bereits abgebrochen werden mussten. Die Anzahl p der Permutationen für zwei Graphen mit den Größen n und m ($n \geq m$)

berechnet sich wie folgt:

$$p = \frac{n!}{(n-m)!}$$

In Abbildung 4.1 wird die Anzahl der Permutationen für alle Paare dargestellt. Jeder Punkt entspricht dabei einem Graphenpaar. Die X-Koordinate ergibt sich aus der Sortierung der Paare nach der Zahl iher Permutationen. Paare mit einer kleineren Zahl von Permutationen sind links von Paaren mit größerer Anzahl eingezeichnet. Da die Anzahl exponentiell wächst, wurde eine logarithmische Darstellung der Y-Achse gewählt. Sie beginnt bei 10^2. Die größten Paare besitzen zwischen 10^{18} und 10^{19} mögliche Permutationen und somit mögliche ECGMs.

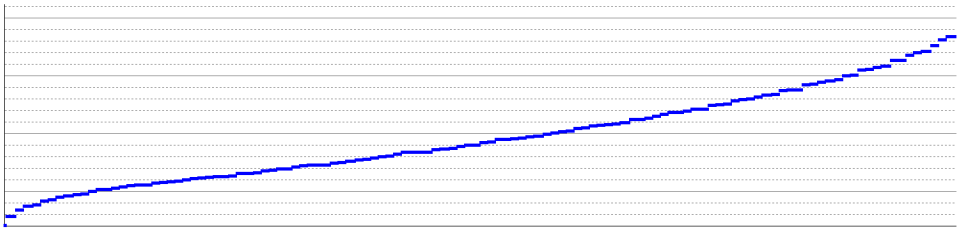


Abbildung 4.1: Anzahl der Permutationen jedes Graphenpaars

Die Tests wurden wie oben beschrieben für jeden Algorithmus ausgeführt. Das Testsystem war ein 2,4 GHz Doppelkernprozessor mit 4 GB RAM und Windows 7[1].

Dauerte ein Test länger als eine Stunde, so wurde er abgebrochen. Wurden zwei Tests hintereinander deswegen abgebrochen, so führte dies zum Abbruch der gesamten Testreihe. Aufgrund der steigenden Zahl der Permutationen ist nicht zu erwarten, dass spätere Tests erfolgreicher sind.

4.1.3 Darstellung der Ergebnisse

Um den Platz besser zu nutzen, wurde bei der Darstellung der Testergebnisse auf eine Achsenbeschriftung verzichtet. Aus diesem Grund folgt eine kurze Beschreibung der Darstellungen. Abbildung 4.2 stellt dies zusätzlich graphisch dar.

[1] Intel(R) Core(TM)2 Duo T8300 @ 2,40GHz; 4,00 GB RAM; Windows 7 Service Pack 1 (64 Bit); Windows Leistungsindex: Prozessor: 6,0 - RAM: 5,9; Laufzeitumgebung: .NET Framework 2.0

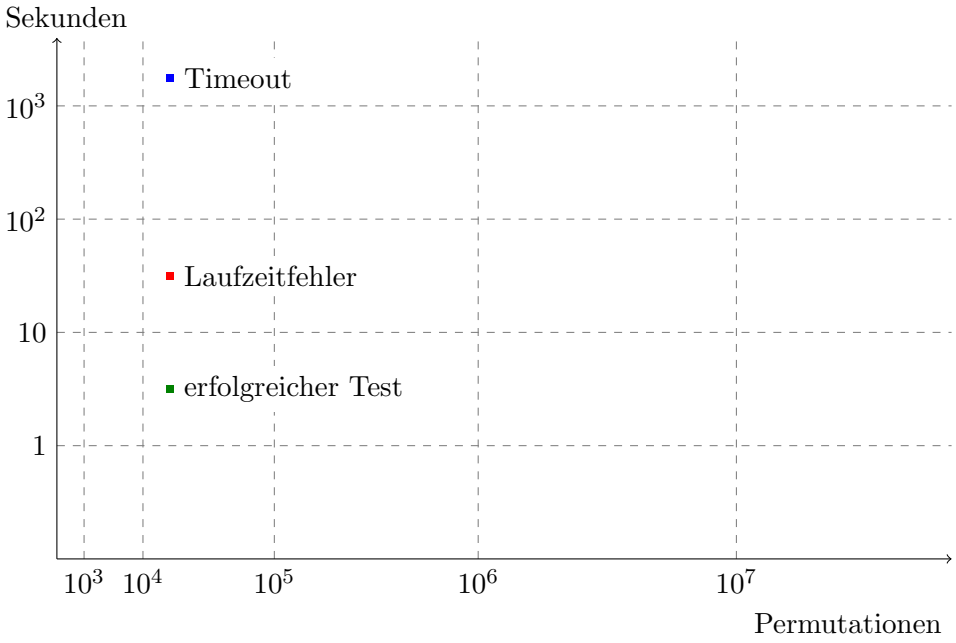


Abbildung 4.2: Das verwendete Koordinatensystem

X-Achse

Die Position auf der X-Achse ergibt sich aus der Position eines Tests in der dazugehörigen Testreihe. Da die Tests nach Anzahl ihrer Permutationen sortiert wurden (dargestellt in Abbildung 4.1), gibt die X-Achse diese auch indirekt an. Die senkrechten Hilfslinien verdeutlichen dies. Eine Hilfslinie wurde immer dann eingefügt, wenn die Anzahl der Permutationen eine ganzzahlige Zehnerpotenz überschritten hat.

Y-Achse

Die Y-Achse stellt die Zeit in Sekunden dar, die ein Test benötigt hat. Die Einteilung ist logarithmisch. Begonnen wird bei 0,1 Sekunden. Liegt die gemessene Laufzeit niedriger oder wurden gar 0,0 Sekunden gemessen, so wird die Laufzeit als 0,1 Sekunden behandelt.

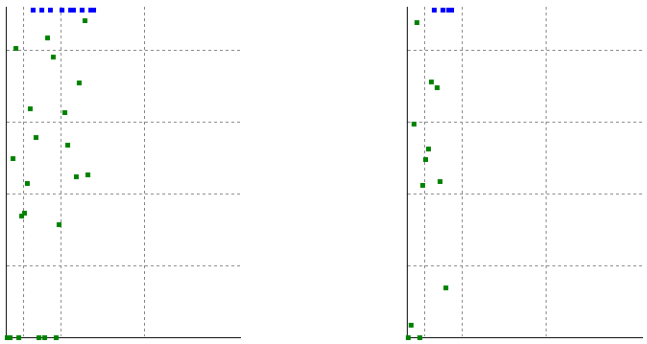
Die waagerechten Hilfslinien stellen die jeweils nächste Zehnerpotenz dar. So ist die erste Hilfslinie über der X-Achse bei 1 Sekunde, die Zweite bei 10 Sekunden, usw.

Farbgebung

Jeder durchgeführte Test wird durch einen Punkt im Koordinatensystem dargestellt. Tests können jedoch zu unterschiedlichen Ergebnissen führen. Die Farben der Punkte stellen dies dar. Ist ein Test erfolgreich, wird er in *grün* dargestellt. Ein *blauer* Punkt bedeutet, dass der Test abgebrochen wurde, weil er die maximale Laufzeit überschritten hat. Ist der Punkt *rot*, dann ist beim Test ein Fehler aufgetreten.

4.2 Testergebnisse

4.2.1 McGregor und MIS basierter Algorithmus



(a) McGregor-Alg. (b) MIS basierter Alg.

Abbildung 4.3: Test des McGregor- und des MIS basierten Algorithmus

Für beide hier dargestellten Algorithmen zeigt sich, dass sie selbst bei kleinen Graphen eine hohe Laufzeit haben. Die Testreihe wurde somit auch schon sehr früh abgebrochen.

Die Streuung lässt sich durch zwei Eigenheiten erklären. Zum einen sind beide Algorithmen in einigen Fällen in der Lage eine optimale Lösung zu erkennen. Dann wird die Suche vorzeitig abgebrochen.

Zum anderen arbeiten beide Algorithmen mit Kantengraphen. Dadurch, dass Kanten als Knoten behandelt werden und ein induzierter Teilgraph gesucht wird, ist die Anzahl der zu testenden Abbildungen eine andere als bei der Suche nach ECGMs auf „normalen“ Graphen. Somit ist es möglich, dass Graphenpaare mit mehr Abbildungen vor Paaren mit weniger getestet werden.

4.2.2 Probieren aller Permutationen

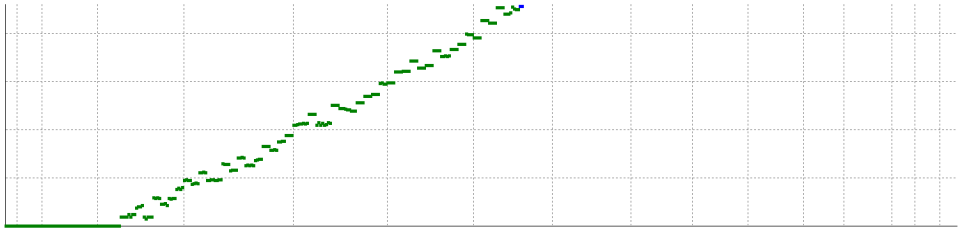


Abbildung 4.4: Probieren aller möglichen ECGMs

Der Zeitaufwand für das Durchprobieren aller Permutationen ist direkt von deren Anzahl abhängig. Da beide Achsen (annähernd[2]) logarithmisch unterteilt sind, bilden die Laufzeiten der Tests annähernd eine Gerade.

Die leichte Streuung liegt vermutlich an der iterativen Berechnung der nächsten Permutation anhand der aktuellen. Sind zwei Graphen unterschiedlich groß, so gibt es Permutationen, die das gleiche ECGM darstellen. Somit muss beim Erzeugen der nächsten Permutation darauf geachtet werden, dass die nächste Permutation auch ein neues ECGM ergibt. Dies erfordert etwas zusätzliche Rechenleistung. Ein rekursives Verfahren, welches die ECGMs mittels Backtracking ermittelt und nicht als Permutation betrachtet, könnte dazu führen, dass die Streuung geringer ist.

Bis zu einer Größe von etwa $1{,}8 \cdot 10^6$ Permutationen ($n = 10$; $m = 8$) liegt die Laufzeit bei rund einer Sekunde. Fünf Sekunden werden mit $6{,}7 \cdot 10^6$ Permutationen ($n = 11$; $m = 8$) nicht überschritten. Bei Graphenpaaren dieser Größe ist es also durchaus möglich, dieses Verfahren einzusetzen ohne die Geduld des Nutzers zu sehr zu belasten.

4.2.3 Branch-and-Bound (und A*)

Das B&B-Verfahren konnte die Testreihe vollständig durchlaufen. Kein Test musste abgebrochen werden, weil er über eine Stunde dauerte. Jedoch wurden viele Tests abgebrochen, weil die Laufzeitumgebung einen Fehler meldete (rote Punkte). Die Ursache dafür war das Verbrauchen von zu viel Speicher.

[2]Die X-Achse ist nur nährungsweise logarithmisch eingeteilt. Dies liegt daran, dass die Anzahl der Permutationen der Graphenpaare nicht gleichmäßig steigt. Bei gleichmäßigem Anstieg würde Abbildung 4.1 eine Gerade darstellen. Die Krümmung ist jedoch nur gering, weshalb die Steigerung nährungsweise als gleichmäßig angesehen werden kann.

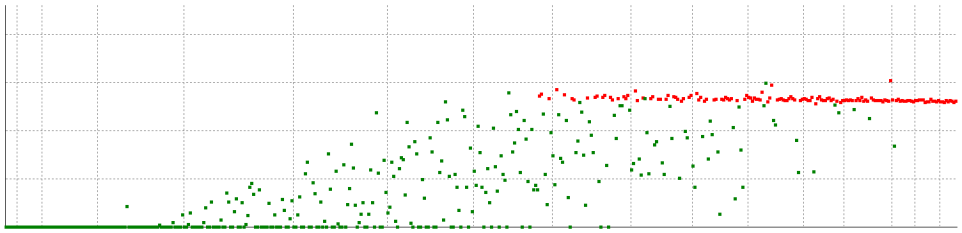


Abbildung 4.5: Test des Branch-and-Bound-Verfahrens

Auf den ersten Blick fällt die starke Streuung der Ergebnisse auf. Das Verfahren kann sehr schnell zu einem Ergebnis führen. So wurde für ein Graphenpaar mit etwa $1{,}8 \cdot 10^{14}$ Permutationen ($n = 17$; $m = 15$) in 1,4 Sekunden verglichen. Das erste Graphenpaar mit mehr als fünf Sekunden Laufzeit besitzt hingegen nur etwa $3{,}9 \cdot 10^{7}$ Permutationen ($n = 11$; $m = 10$). Es benötigte 8,0 Sekunden.

Ebenfalls auffällig ist, dass es keine Gruppenbildung gibt. Zu jeder Anzahl an Knoten n gibt es zwei Graphen mit unterschiedlicher Anzahl an Kanten ($d = 3$ und $d = 4$). Bei der Berechnung der möglichen Permutationen wird nur die Anzahl der Knoten herangezogen. Somit bilden sich in einer Testreihe Gruppen von jeweils zwei bzw. vier Graphenpaaren, welche die gleiche Anzahl an Permutationen besitzen (gut zu sehen in Abbbildung 4.4). Innerhalb einer solchen Gruppe unterscheiden sich die Paare nur durch die Anzahl der Kanten. Da die Anzahl der Kanten allerdings bei der Berechnung der Schranke des B&B-Verfahrens eine Rolle spielt, liegt es somit nahe, dass die Anzahl der Kanten einen Einfluss auf die Rechenzeit hat.

Abbildung 4.6 stellt erneut die Laufzeit des B&B-Verfahrens in einem Ausschnitt[3] dar. Die Färbung ist aber eine andere. Tests, bei denen der kleinere Graph eine geringe Kantendichte hat ($d = 3$, siehe Abschnitt 4.1.1) sind *blau* dargestellt. Tests mit einer größeren Kantendichte ($d = 4$) sind *rot*. Wegen zu hohem Speicherverbrauch abgebrochene Tests wurden nicht eingezeichnet.

Es ist deutlich zu sehen, dass blau markierte Tests in der Regel schneller sind als die roten. Beide Bereiche überlappen sich jedoch. Auch die Streuung ist bei beiden Varianten vorhanden.

[3]Die X-Achse geht von 10^{6} bis 10^{14} Permutationen, die Y-Achse von 0,1 bis 100 Sekunden.

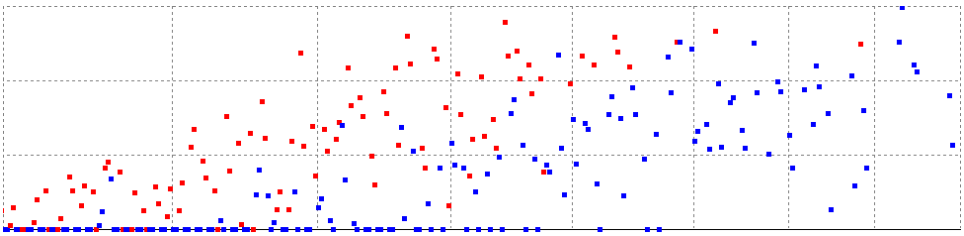


Abbildung 4.6: B&B-Laufzeit abhängig von der Kantendichte des kleineren Graphen

Dass die Kanten einen Einfluss auf die Schranke und somit auf die Laufzeit haben, führt zu einer weiteren Vermutung. Eventuell lässt sich die Laufzeit verringern, indem der Suchbaum die Knoten in Abhängigkeit ihrer anliegenden Kanten (also der Zahl ihrer Nachbarn) zuweist. Um dies zu überprüfen, wurde der Algorithmus leicht geändert. Vor der eigentlichen Suche werden die Knoten nach der Anzahl ihrer Nachbarn sortiert. Es gab zwei Testreihen. Eine begann den Suchbaum mit Knoten, die möglichst wenig Nachbarn hatten. Die Andere mit möglichst vielen. Dabei brachte die Testreihe, die Knoten mit vielen Nachbarn zuerst zuwies, den gewünschten Effekt. Abbildung 4.7 stellt das Ergebnis dar. Das vorrangige Zuweisen der Knoten mit wenig Nachbarn führte hingegen zu einer Erhöhung der Laufzeit.

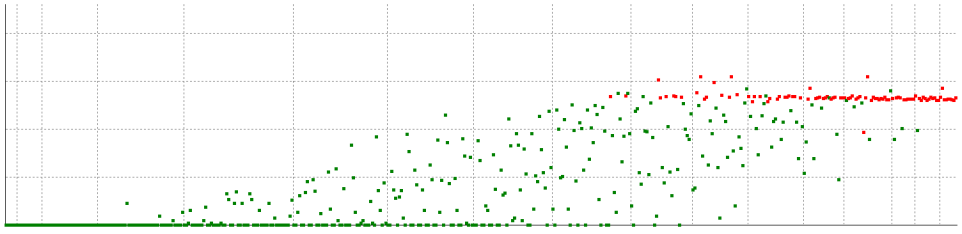


Abbildung 4.7: B&B-Laufzeit, wenn Knoten mit vielen Nachbarn zuerst zugewiesen werden

Da die Streuung weiterhin vorhanden ist, lässt sich aus Abbildung 4.7 nur schlecht sagen, ob die Verbesserung mehrheitlich ist oder nur bestimmte Graphenpaare betrifft. Abbildung 4.8 stellt deswegen die Differenz der Laufzeiten dar. *Rote* Punkte entsprechen dabei einer Verschlechterung der Laufzeit. Dies kann auch bedeuten, dass ein Test abgebrochen wurde, der vorher erfolgreich war. Hat sich die Laufzeit verbessert bzw. war ein Test erfolgreich, der zuvor abgebrochen werden musste, so wird der Punkt *grün*

dargestellt. Musste für ein Graphenpaar in beiden Testreihen der Test abgebrochen werden oder liegt die Zeitdifferenz unter 0,1 Sekunden, ist der Test nicht eingetragen.

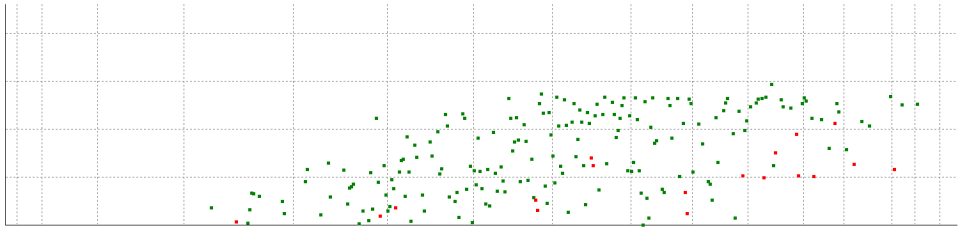


Abbildung 4.8: Laufzeitdifferenz der B&B-Testreihen

Man kann klar erkennen, dass die überwiegende Mehrheit der Tests besser verlief. Es gibt aber auch Tests, die schlechter abschnitten. Es kann also nicht von einer generellen Verbesserung gesprochen werden.

Das Fazit für das B&B-Verfahren fällt weitgehend gut aus. Es ist ein exaktes Verfahren, welches für Graphenpaare mit niedriger und mittlerer Zahl an Permutationen in der Regel schnell zu einem Ergebnis führt. Das Vorsortieren der Knoten nach der Anzahl ihrer Nachbarn erhöht dabei die Wahrscheinlichkeit für eine kurze Laufzeit. Nachteilig an B&B ist, dass es eine sehr starke Streuung besitzt. Es ist somit nur schwer vorhersagbar, wie lange die Berechnung für ein konkretes Graphenpaar dauert.

4.2.4 Evolutionärer Algorithmus

Beim Testen des evolutionären Algorithmus wurde eine Population vom 1.000 Individuen verwendet. Bei Mutation und Rekombination werden jeweils 1.000 weitere Individuen erzeugt. Die Selektion wählt aus den nun 3.000 Individuen die 1.000 Besten aus. Die Anderen „sterben“. Ein Individuum wird als Optimum betrachtet, wenn nach 20 Generationen kein besseres gefunden wurde.

Laufzeit

Mittelst des evolutionären Algorithmus konnte zu jedem Paar eine mögliche Lösung ermittelt werden. Die Laufzeit bleibt auch bei einer hohen Anzahl an Permutationen verhältnismäßig gering. Das erste Graphenpaar, welches über fünf Sekunden benötigt, besitzt etwa $1{,}3 \cdot 10^{15}$ Permutationen ($n = 15$; $m = 14$). Die höchste Laufzeit betrug 18,4 Sekunden. Das

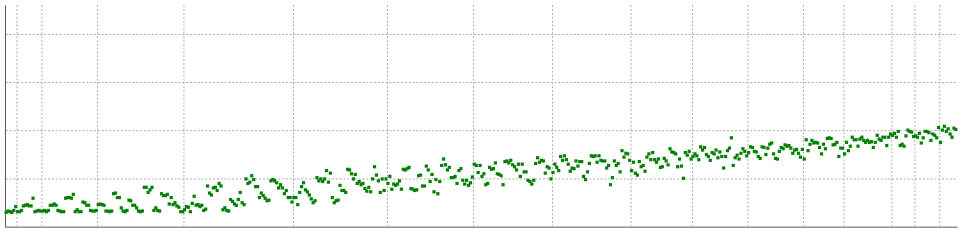


Abbildung 4.9: Laufzeiten des evolutionären Algorithmus

dazugehörige Graphenpaar hat etwa $2{,}4 \cdot 10^{18}$ mögliche Permutationen ($n = 20$; $m = 19$).

Die sichtbare Streuung könnte zwei mögliche Ursachen haben. Zum einen ist es denkbar, dass die unterschiedliche Kantendichte sich auf die Zeit auswirkt. Zum anderen werden die Paare für eine Rekombination sowie die Änderung bei einer Mutation zufällig bestimmt. Ob und wann eine bessere Lösung gefunden wird, ist somit auch vom Zufall abhängig.

Qualität der Lösung

Da es sich bei einem evolutionären Algorithmus nicht um ein exaktes Verfahren handelt, ist es nötig, die Qualität der Lösung zu betrachten. Maßgabe für die Qualität ist das Verhältnis der Anzahl gefundenen gemeinsamen Paare mit Kante in der exakten Lösung zur Anzahl in der Lösung des evolutionären Algorithmus.

Abbildung 4.10 stellt das Verhältnis zwischen beiden Lösungen dar. Es wurde erneut auf die Beschriftung verzichtet. Das Koordinatensystem unterscheidet sich jedoch von bisherigen Darstellungen. Die Y-Achse stellt nun das Verhältnis zwischen exakter und evolutionär ermittelter Lösung dar. Die Achse ist nicht mehr logarithmisch. Im Koordinatenursprung sind es 0%. Waagerechte Hilfslinien sind jeweils in 10%-Schritten eingezeichnet. Die X-Achse sowie die senkrechten Hilfslinien wurden nicht verändert. Die Farbgebung gibt zusätzlich Auskunft über die Zahl der gemeinsamen Paare ohne Kante. Stimmt sowohl die Zahl der Paare mit Kante als auch die Zahl der Paare ohne Kante mit der exakten Lösung überein, ist der Test *grün* dargestellt. Andere Tests sind *blau*.

Da nicht für alle Graphenpaare eine exakte Lösung vorliegt, ist es auch nicht möglich zu jeder Lösung des evolutionären Algorithmus eine Qualität

anzugeben. Die entsprechenden Graphenpaare sind somit nicht in Abbildung 4.10 eingetragen.

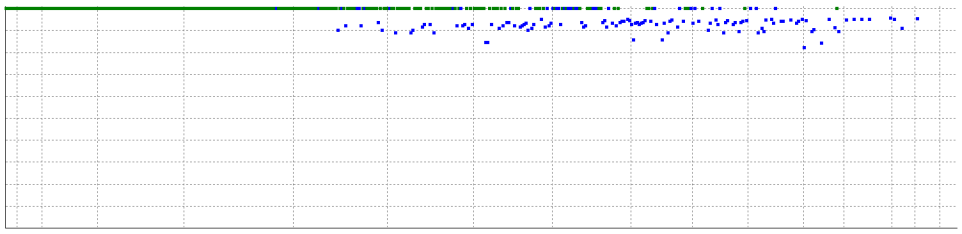


Abbildung 4.10: Qualität der Lösungen des evolutionären Algorithmus

Der Evolutionäre Algorithmus liefert durchgehend gute Lösungen. Die erste Lösung unter 100% tritt bei einem Graphenpaar mit etwa $2{,}8 \cdot 10^7$ Permutationen ($n = 20$; $m = 6$) auf. Bis etwa 10^{11} Permutationen haben jedoch ein Großteil der Lösungen eine Qualität von 100%. Die Mehrheit der Lösungen, die nicht vollständig übereinstimmen, haben eine Qualität über 90%. Auch schlechtere Lösungen unterschreiten die 80% nicht. Da nicht zu jedem Paar eine Lösung bekannt ist, lässt sich jedoch nicht sagen, ob generell die Qualität der Lösungen so gut ist. Es ist denkbar, dass die selben Eigenschaften, die dazu führen, dass B&B keine Lösung liefert, auch eine Lösung mit schlechter Qualität beim evolutionärem Algorithmus zur Folge haben.

4.2.5 Bipartites Matching

Die Ermittlung einer Lösung mittels bipartitem Matching ist im Vergleich zu den anderen getesteten Verfahren sehr schnell. Zu allen Graphenpaaren wurde in unter 0,1 Sekunden eine Lösung ermittelt. Auf eine Darstellung der Laufzeit wird deshalb verzichtet.

Interessant ist die Qualität der Lösungen, denn es handelt sich um eine Heuristik. Es wurde jedoch nicht das Gewicht des Matchings genommen, da es nur eine Schätzung ist und nicht die realen Kosten angibt. Stattdessen werden die gemeinsamen Paare genommen, die sich durch das Matching ergeben. Abbildung 4.11 stellt die Qualität des Verfahrens dar. Die Darstellung ist analog zur Qualitätsdarstellung des evolutionären Algorithmus.

Neben der starken Streuung der Ergebnisse sieht man auch, dass die Qualität bei größeren Graphenpaaren deutlich abnimmt. Eine mögliche

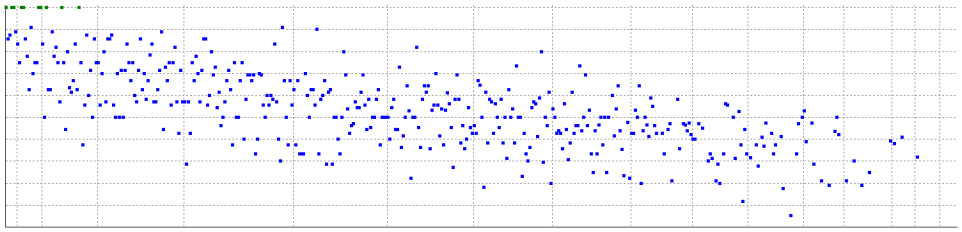


Abbildung 4.11: Qualität der Lösungen des bipartiten Matchings

Erklärung für die Abnahme ist, dass die Zahl der Kanten in den Graphen linear mit deren Größe steigt. Die Anzahl der Knotenpaare steigt jedoch quadratisch. Die Wahrscheinlichkeit, dass bei zwei Knotenpaaren auch beide eine Kante besitzen, nimmt somit ab.

Abbildung 4.12 stellt die Qualität einer weiteren Testreihe dar. Hierbei wurden bei jedem Test zehn zufällige Permutationen erstellt und die Beste dann als Lösung ausgegeben. Es ist zu sehen, dass die Qualität der mittels bipartitem Matching ermittelten Lösung sich kaum von der Qualität der zufällig erzeugten Lösungen unterscheidet.

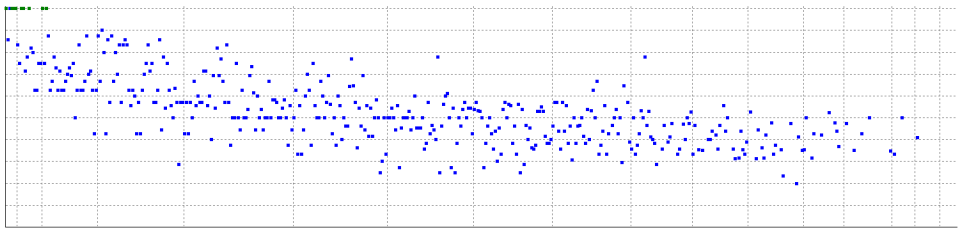


Abbildung 4.12: Qualität von zufälligen Lösungen

Das Fazit für das bipartite Matching fällt negativ aus. Zwar ist es ein sehr schnelles Verfahren, jedoch führt die Heuristik nicht zu Lösungen mit hoher Qualität. Die Qualität ist auch nicht besser als zufällig ermittelte Lösungen.

Kapitel 5

Praktische Umsetzung

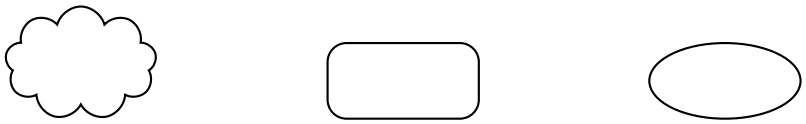
Dieses Kapitel stellt am Beispiel der Lotka-Volterra-Form (LV-Form) ein Konzept für die praktische Umsetzung dar.

5.1 Lotka-Volterra-Form

Die Lotka-Volterra-Regeln (LV-Regeln) sind ein Regelwerk „zur quantitativen Beschreibung der Populationsdynamik in Räuber-Beute-Beziehungen“ [17]. Sie wurden 1925 und 1926 von Alfred Lotka und Vito Volterra formuliert. Die LV-Form ist ein Modell, welches auf den LV-Regeln basiert.

5.1.1 Elemente

Eine LV-Form besteht aus drei Elementen: *Populationen*, *Ereignisse* und *Parameter*. Abbildung 5.1 stellt diese dar.



(a) Population (b) Ereignis (c) Parameter

Abbildung 5.1: Elemente der LV-Form

Populationen sind die Basiselemente. Sie stellen eine Gruppe von Lebewesen dar.

Ein *Ereignis* ist eine gerichtete Verbindung zweier Populationen miteinander. Es kann auch eine Population mit sich selbst verbunden werden. Ereignisse stellen Handlungen oder Geschehen in einer Population dar, die Auswirkungen auf die Ziel-Population haben.

Parameter beeinflussen Ereignisse. Sie sind somit auch immer auf

(mindestens) ein Ereignis gerichtet. Sie können zusätzlich von Populationen abhängig sein.

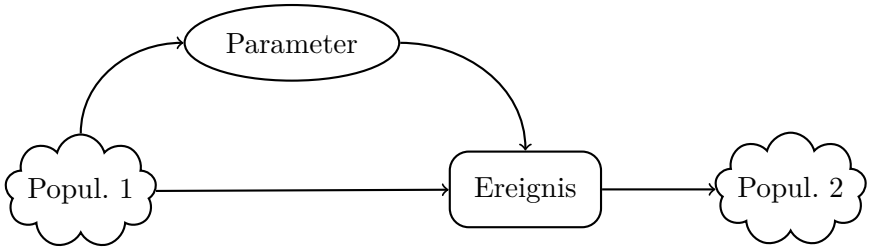


Abbildung 5.2: Mögliche Verbindungen in einer LV-Form

5.1.2 Formale Umsetzung

Mit den oben beschriebenen Regeln lässt sich eine LV-Form aus Populationen, Ereignissen und Paramtern formal beschreiben.

Definition 5.1 (Lotka-Volterra-Form) *Eine Lotka-Volterra-Form Λ ist ein 3-Tupel $\Lambda = (\Pi, \Sigma, \Phi)$. Dabei sind:*

- Π *eine endliche Menge von Populationen,*
- $\Sigma \subseteq \Pi \times \Pi$ *die Menge an Ereignissen und*
- $\Phi \subseteq \wp(\Pi) \times (\wp(\Sigma) \backslash \{\emptyset\})$ *die Menge an Parametern*[1]*.*

5.1.3 Umsetzung als Graph

Die nun formal beschriebene LV-Form lässt sich auch als Graph umsetzen. Dabei sind Populationen, Ereignisse und Parameter jeweils Knoten. Die Kanten ergeben sich direkt aus den in Definition 5.1 beschrieben Bedingungen. Der so entstehende gerichtete Graph sei dann ein Lotka-Volterra-Graph (LVG).

Definition 5.2 (Lotka-Volterra-Graph) *Gegeben sei eine LV-Form $\Lambda = (\Pi, \Sigma, \Phi)$. Der dazugehörige gerichtete Lotka-Volterra-Graph $G = (V, E)$ sei wie folgt definiert:*

Die Knotenmenge V ist die Vereinigungsmenge der Populationen, Ereignisse und Parameter.

$$V = \Pi \cup \Sigma \cup \Phi$$

[1] $\wp(M) = \{T \mid T \subseteq M\}$ ist die Potenzmenge (die Menge aller Teilmengen) von M.

Die Kantenmenge E ist die Vereinigung aus der durch Ereignisse erzeugten Kantenmenge E_Σ sowie der durch Parameter erzeugten Kantenmenge $E_{\Pi\Phi}$ und $E_{\Phi\Pi}$. Dabei stellt $E_{\Pi\Phi}$ die Kanten von Populationen zu Parametern und $E_{\Phi\Sigma}$ von Parametern zu Ereignissen dar.

$$E = E_\Sigma \cup E_{\Pi\Phi} \cup E_{\Phi\Sigma}$$

Für jedes Ereignis $e = (p, q) \in \Sigma$ ist jeweils eine Kante zwischen dem Ereignis und den beiden Populationen p und q in E_Σ enthalten.

$$E_\Sigma = \{(p, e), (e, q) \mid e \in \Sigma\}$$

In $E_{\Pi\Phi}$ befindet sich für jeden Parameter $\varphi = (P_\Pi, P_\Sigma) \in \Phi$ jeweils eine Kante zwischen den Populationen $\pi \in P_\Pi$ und dem Parameter φ.

$$E_{\Pi\Phi} = \bigcup_{\varphi \in \Phi} \{(\pi, \varphi) \mid \pi \in P_\Pi\}$$

Des weiteren ist in $E_{\Phi\Sigma}$ für jeden Parameter $\varphi = (P_\Pi, P_\Sigma) \in \Phi$ jeweils eine Kante vom Parameter φ zu jedem Ereignis $e \in P_\Sigma$ vorhanden.

$$E_{\Phi\Sigma} = \bigcup_{\varphi \in \Phi} \{(\varphi, e) \mid e \in P_\Sigma\}$$

Der so erzeugte LVG lässt sich nun mit anderen Graphen vergleichen.

5.2 Veringerung der Komplexität

Angenommen, eine LV-Form besitzt vier Populationen, pro Population zwei Ereignisse und fünf Parameter. Der dazugehörige LVG hat dann 17 Knoten. Vergleicht man den LVG mit einem dazu isomorphen Graphen, so gibt es etwa $3{,}6 \cdot 10^{14}$ ECGMs. Das B&B-Verfahren hat in dieser Größenordnung eine hohe Wahrscheinlichkeit nicht erfolgreich zu sein. Bei den entsprechenden Tests (Abschnitt 4.2.3) mussten die meisten von ihnen abgebrochen werden, weil der Speicherverbrauch zu hoch war.

5.2.1 Vorgabe von Knoten

Eine Möglichkeit, die Anzahl der ECGMs zu senken, ist es, dem Lerner Knoten vorzugeben. So könnten beispielsweise die Population vorgegeben

werden. Die entsprechenden Knoten sind dann fest den Populationsknoten in der Musterlösung zugeordnet. Die Anzahl der ECGMs sinkt dann auf etwa $6{,}2 \cdot 10^9$.

Es besteht vermutlich auch die Chance, dass sich durch die Vorgabe schneller deutliche Schranken bilden, also dass schneller abgeschätzt werden kann, wie viele Kanten der gemeinsame Teilgraph höchstens haben wird. Die Vermutung basiert darauf, dass die zugewiesenen Knoten auch Auswirkungen auf die Kanten und somit auf die benachbarten Knoten haben. Gibt es ohne Vorgabe beispielsweise für ein verbundenes Paar aus einer Population und einem Ereignis vier oder mehr mögliche Abbildungen, so gibt es mit Vorgabe nur noch ein oder zwei. Dies liegt daran, dass die Population eindeutig bestimmt werden kann. Somit bleiben als mögliche Zuordnungen für das Ereignis nur die Ereignisse der Musterlösung, die mit der Population verbunden sind.

5.2.2 Knotenfärbung

Eine weitere Möglichkeit besteht darin, die Knoten eines Graphen in Gruppen zu unterteilen, sie zu färben.

Definition 5.3 (Knotenfärbung) *Gegeben sei ein Graph $G = (V, E)$. Eine Knotenfärbung f ist eine Abbildung $f : V \to \mathbb{N}$. Die Farbe eines Knotens v ist dann $f(v)$.*

Nun erlaubt man in einem ECGM nur Zuweisungen, wenn beide Knoten die gleiche Farbe besitzen. Für einen LVG bietet es sich an, die Knoten in drei Farben zu unterteilen: je eine Farbe für Populationen, Ereignisse und Parameter. Auch die Vorgabe von Knoten lässt sich mittels Färbung umsetzen. Jeder vorgegebene Knoten bekommt eine eigene Farbe.

Mittels Färbung lässt sich nun die Zahl der möglichen ECGMs von $G_1 = (V_1, E_1)$ nach $G_2 = (V_2, E_2)$ stark reduzieren. Es seien n_i und m_i die Anzahl der Knoten in G_1 bzw. G_2, welche die Farbe i haben.

$$n_i = \big|\{v \mid v \in V_1 \wedge f(v) = i\}\big|$$
$$m_i = \big|\{v \mid v \in V_2 \wedge f(v) = i\}\big|$$

Es sei dann p_i die Anzahl der ECGMs innerhalb einer Farbe.

$$p_i = \begin{cases} \frac{n_i!}{(n_i - m_i)!} & n_i \geq m_i \\ \frac{m_i!}{(m_i - n_i)!} & \text{sonst} \end{cases}$$

Die Gesamtzahl p der ECGMs ist nun das Produkt aller p_i.

$$p = \prod_{i \in \mathbb{N}} p_i$$

Das Beispiel von oben hat dann, wenn es mit einem isomorphen Graphen verglichen wird, $4{,}8 \cdot 10^6$ mögliche ECGMs. In der Testreihe des B&B-Verfahrens, in der die Knoten nach Zahl ihrer Nachbarn sortiert wurden (siehe Abbildung 4.7), wurden alle Tests in der Größenordnung von 10^6 bis 10^7 Permutationen erfolgreich in unter 1 Sekunde beendet.

5.3 Auswertung eines ECGMs

Nach Ermittlung eines ECGMs ist der nächste Schritt die Auswertung. In der hier vorgestellten Variante wird ein Fehlergraph gebildet. Anschließend erfolgt eine Mustersuche auf Basis von TGI.

5.3.1 Fehlergraph

Die Idee an einem Fehlergraph ist es, dass man beide Graphen zusammenfügt. Zusätzlich markiert man die Knoten und Kanten abhängig davon, ob sie hinzugefügt, entfernt oder übernommen wurden. Auf diese Weise lässt sich in diesem neuen Graphen nach Mustern suchen. Der so erzeugte Graph sei ein Fehlergraph.

Definition 5.4 (Fehlergraph) *Gegeben seien zwei Graphen $G_1 = (V_1, E_2)$ und $G_2 = (V_2, E_2)$ sowie ein ECGM $\varphi : \hat{V}_1 \to \hat{V}_2$ mit $\hat{V}_1 \subseteq V_1$ und $\hat{V}_2 \subseteq V_2$. Aus G_1, G_2 und φ ergeben sich zusätzlich V_d, V_a E_d und E_a (siehe Abschnitt 2.5).*

Ein Fehlergraph von G_1, G_2 und φ ist dann ein 3-Tupel $F = (V_F, E_F, \alpha)$. Dabei sind:

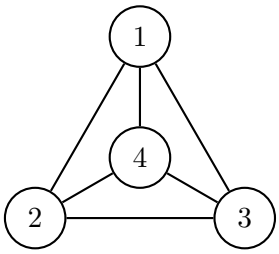
- *$V_F = V_1 \cup V_a$ die Menge der Knoten,*
- *$E_F = E_1 \cup E_a$ die Menge der Kanten und*

- $\alpha : V \cup E \to \{r, g, b\}$ *die Markierung der Knoten und Kanten.*

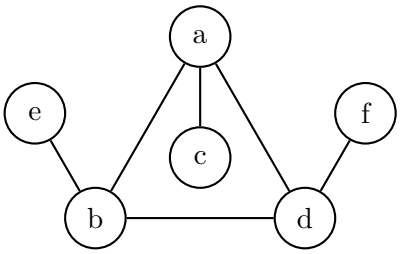
Dabei wird ein Element des Graphen (ein Knoten oder eine Kante) mit r markiert, wenn es entfernt wird. Es erhält g als Markierung, wenn es hinzugefügt wird. Ist ein Element in beiden Graphen vorhanden (wird übernommen), dann wird es mit b markiert.

$$\alpha(x) = \begin{cases} r & x \in V_d \cup E_d \\ g & x \in V_a \cup E_a \\ b & sonst \end{cases}$$

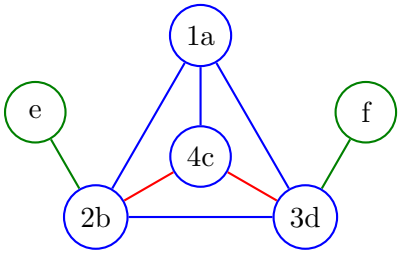
Der so erzeugte Fehlergraph F ist nun ein Graph, der alle Informationen aus den Graphen G_1 und G_2 sowie dem ECGM φ enthält. Abbildung 5.3 stellt dies dar. Die Knoten und Kanten sind dabei entsprechend ihrer Markierung in rot, grün und blau dargestellt.



(a) Eingabe des Lerners



(b) Musterlösung



(c) erzeugter Fehlergraph

Abbildung 5.3: Beispiel für einen Fehlergraph

5.3.2 Mustererkennung

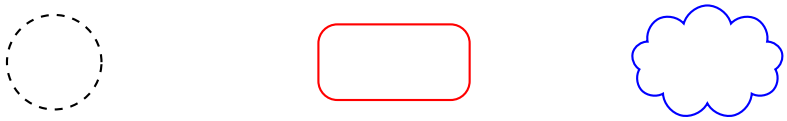
Das Erkennen von Mustern in einem Fehlergraphen lässt sich über Teilgraphisomorphie (TGI) realisieren. Dabei stellt man die Muster als Graphen dar, zu denen dann isomorphe Teilgraphen im Fehlergraphen

gesucht werden. Zwar ist TGI auch NP-vollständig, ein Muster ist aber höchstens so groß wie der Fehlergraph. Nimmt man zusätzlich an, dass die gesuchten Muster deutlich kleiner sind als die Fehlergraphen, so sollte selbst mit einfachen Algorithmen wie Backtracking die Laufzeit in verträglichem Maße bleiben.

Elementare Muster

Die zu suchenden Muster sind Graphen. Ihre Elemente sind Knoten und Kanten. Somit stellt ein Knoten bzw. eine Kante ein elementares Muster dar. Dabei ist vor allem die Markierung interessant. Die drei bei Fehlergraphen verwendeten Markierungen lassen sich erweitern. So wäre es eine Möglichkeit, Knoten und Kanten mit beliebiger Markierung zuzulassen. Abbildung 5.4 stellt mögliche elementare Varianten für Knoten dar. Ist eine beliebige Markierung zulässig, so wird der Knoten gestrichelt gezeichnet. Kanten verhalten sich dazu analog.

Dieses Prinzip lässt sich auch auf die Färbung von Knoten erweitern. So ist es denkbar, dass man in dem Fehlergraphen eines LVG etwas sucht. Ein elementares Muster wäre hier, dass ein Ereignis entfernt wird (Abbildung 5.4b).



(a) beliebiger Knoten

(b) entferntes Ereignis

(c) gemeinsame Population

Abbildung 5.4: Beispiele für elementare Knotenmuster

Einfache Muster

Mittels der elementaren Muster lassen sich nun größere Muster entwerfen. So lässt sich mit dem Muster in Abbildung 5.5 beispielsweise erkennen, dass der Lerner lediglich einen Knoten zu viel hinzugefügt hat. Betrachtet man nur die Zahl der entfernten und hinzugefügten Knoten, so würde man dem Lerner zwei überflüssige Kanten, einen überflüssigen Knoten und eine fehlende Kante anlasten. Das Muster ermöglicht somit eine andere Bewertung.

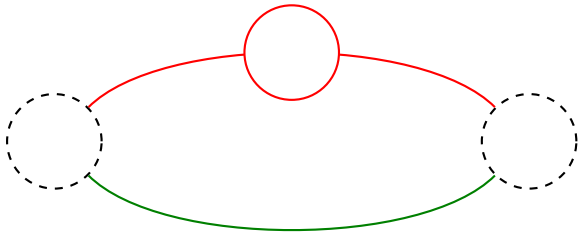


Abbildung 5.5: Knoten zu viel

Die Abbildungen 5.6 und 5.7 zeigen zwei weitere Beispiele. Im ersten Fall wurde eine Kante in die falsche Richtung eingetragen. Das zweite Beispiel verwendet zusätzlich die Knotenfärbung. Es gibt an, dass ein Ereignis auf die falsche Population gerichtet ist.

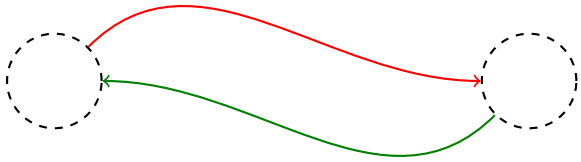


Abbildung 5.6: Falsche Richtung einer Kante

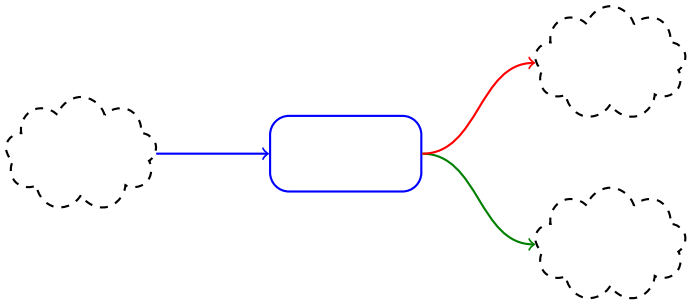


Abbildung 5.7: Ereignis zielt auf falsche Population

Musterhierarchie

Bildet man aus einfachen Mustern größere, so entsteht dabei auch eine Hierarchie. Das Erkennen dieser Hierarchie kann Vorteile für die Suche nach Mustern und die Bewertung der Eingabe des Lerners haben.

Ist ein kleines Muster Teil eines größeren, so hat dies zur Folge, dass das größere nur dann vorhanden ist, wenn auch das kleinere gefunden wurde. Mit einer Hierachie lässt sich somit der Aufwand für die Mustersuche verringern, indem zuerst kleinere Muster gesucht werden. Wurde ein kleineres Muster nicht gefunden, so ist auch kein Muster im Fehlergraphen enthalten, welches sich aus dem kleinen ergibt. Zusätzlich dienen bereits gefundene kleine Muster als Suchgrundlage für die Großen. Es ist somit nicht mehr nötig, den gesamten Fehlergraphen zu durchsuchen.

Für die Bewertung der Lerner-Eingabe kann eine Hierachie dann hilfreich sein, wenn große und kleine Muster gefunden werden. Ist ein größeres Muster im Fehlergraph vorhanden, so sind dies auch kleinere Muster, wenn sie Teil des größeren sind. Werden nun das große und das kleinere Muster bewertet, fließen die kleineren Muster mehrfach in die Bewertung ein. Dies mag nicht immer erwünscht sein.

Berechnung der Hierarchie

Das Ermitteln der Hierarchie erfolgt erneut über TGI. Dabei wird überprüft, ob ein Muster ein Teilgraph eines anderen ist. Auf diese Weise baut sich ein gerichteter azyklischer Graph (engl. directed acyclic graph; kurz DAG) auf. Dabei stellt jeder Knoten ein Muster dar. Existiert ein Pfad von einem Knoten u zu einem Knoten v, dann ist das Muster von u ein Teilgraph des Musters von v.

Geht man davon aus, dass das Überprüfen der TGI in $\mathcal{O}(\tau)$ möglich ist und es insgesamt n Muster gibt, dann lässt sich der DAG in $\mathcal{O}(n^2\tau)$ bilden. Dazu stellt man den DAG als Adjazenzmatrix der Größe $n \times n$ dar. Das Element (i, j) der Matrix hat dann den Wert 0, wenn es keine Kante gibt oder $i = j$, und den Wert 1, wenn es eine Kante gibt, also wenn Muster i Teilgraph von Muster j ist.

Wenn erwünscht, kann man redundante Kanten auch entfernen. Eine Kante (i, j) ist redundant genau dann, wenn es einen Knoten (ein Muster) k gibt, und ein Pfad von i über k zu j existiert. Um redundante Kanten zu entfernen muss man nun einfach alle paarweise verschiedenen i, j, und k überprüfen. Da sich dies einfach aus der Adjazenzmatrix auslesen lässt, ergibt sich somit ein zusätzlicher Aufwand von $\mathcal{O}(n^3)$. Um das Übersehen redundanter Kanten aufgrund von bereits entfernten zu vermeiden, sollten Kanten jedoch erstmal nur zum Löschen markiert und im Nachhinein

gelöscht werden.

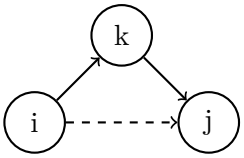


Abbildung 5.8: Redundante Kante (i, j) in der Musterhierarchie

Der gesamte Aufwand liegt dann höchstens bei $\mathcal{O}(n^2\tau + n^3)$. Insgesamt ist der Aufwand jedoch nebensächlich, denn die Hierarchie lässt sich bereits beim Entwurf einer Aufgabe ermitteln. Es ist nicht nötig dies bei jeder Eingabe des Lerners zu wiederholen.

Überschneidung von Mustern

Eine Problematik, die man bei der Mustersuche beachten muss, ist die Überschneidung von Mustern. Zwei Muster können einen gemeinsamen Teilgraphen besitzen. Werden nun beide Muster erkannt, kann es vorkommen, dass sie sich überschneiden. Dies kann zu einer fehlerhaften Bewertung führen. Erkennen lässt sich ein gemeinsamer Teilgraph indem man das ECGM für beide Muster ermittelt.

Prinzipiell bieten sich drei Varianten für den Umgang mit Musterüberschneidungen an:

- **Ignorieren:** Sollte eine Überschneidung keine relevanten Folgen haben, so ist die Suche auch nicht nötig.
- **Nachträgliche Suche:** Zuerst werden alle Muster im Fehlergraphen ermittelt. Erst danach wird überprüft, ob es zu Überschneidungen gekommen ist.
- **Sperren von Knoten:** Legt man fest, dass ein Knoten bzw. eine Kante nur zu einem Muster gehören kann, dann schließt man damit Überschneidungen aus. Dabei sollte man jedoch auf die Reihenfolge achten, mit der man nach Mustern sucht. Es kann passieren, dass ein weniger wichtiges Muster zuerst gefunden wird und somit ein wichtiges Muster sperrt.

Kapitel 6

Diskussion

Die Arbeit beschäftigte sich mit dem Vergleichen von Graphen. Ziel war es eine algorithmische Grundlage zu schaffen, um die Eingabe eines Lerners bewerten zu können.

6.1 Die untersuchten Algorithmen

Um das Ziel zu erreichen wurden verschiedene Algorithmen betrachtet. Einige erwiesen sich als gute Grundlage für weitere Betrachtungen. Andere hingegen scheiterten bereits an kleinen Graphenpaaren.

6.1.1 Kantengraphen und MCS

Ein Ansatz zum Ermitteln einer Lösung war es, die zu vergleichenden Graphen in Kantengraphen umzuwandeln. Als nächstes sollte dann der gemeinsame induzierte Teilgraph (MCS) ermittelt werden. Dieser Ansatz erwies sich jedoch als unbrauchbar. Die Laufzeit ist schlicht zu hoch. Die dafür getesteten Verfahren waren jedoch so ausgelegt, dass sie alle Möglichkeiten durchprobieren. Wäre es möglich die Laufzeit auf ein vertretbares Maß zu senken, dann ließe sich auch untersuchen, ob der Ansatz eines Kantengraphen sinnvoll ist. Dieser bietet die Möglichkeit Kanten unabhängig von Knoten zu betrachten. Es besteht die Hoffnung, dass man auf diese Weise zusätzliche Informationen erhält. Eventuell lassen sich aber die gleichen Informationen auch über Mustersuche im Fehlergraphen ermitteln.

6.1.2 Probieren aller Permutationen

Als nächstes wurde versucht ECGMs direkt zu ermitteln. Der erste Ansatz dabei war das Probieren aller möglichen Permutationen. Das Verfahren ist zwar offensichtlich eines, welches sehr schnell eine zu hohe Laufzeit besitzt, jedoch war es deutlich schneller als die Suche nach einem MCS in

Kantengraphen für die gleichen Graphenpaare. Es ist somit als Vergleichsgröße zu verstehen.

6.1.3 Das B&B-Verfahren

Das B&B-Verfahren erwies sich als ein zufriedenstellendes. Die Idee daran, abzuschätzen wo im Suchbaum sich das Optimum befindet, beschleunigt die Suche stark. Dabei wurde beiläufig festgestellt, dass es sich bei B&B sowie dem A*-Algorithmus um fast identische Verfahren handelt. Dies ist insofern interessant, weil diese Ähnlichkeit üblicherweise in entsprechender Literatur nicht erwähnt wird. Ursache dafür mag sein, dass die Intention, die zu diesen Algorithmen führte, bei beiden eine unterschiedliche war. Bei B&B ging es um die Lösung linearer Optimierungsprobleme. Der A*-Algorithmus hat seine Wurzeln in der Pfadsuche.

6.1.4 Verbesserung von B&B

Ebenfalls interessant ist die starke Streuung bei der Laufzeit des B&B-Verfahrens. Hier bleibt die Frage offen, wie sich die Streuung und die Laufzeit verringern lassen. Dabei bieten sich zwei Ansätze: das Finden einer besseren Schranke und das Erkennen und Nutzen von Eigenschaften, die sich auf den Suchaufwand auswirken.

Eine bessere Schranke

Je besser die Schranke ist, umso weniger Varianten müssen überprüft werden. Dabei gilt es jedoch zu beachten, dass eine genauere Schranke auch einen Anstieg der Komplexität und somit der Laufzeit zur Folge hat. Dies liegt daran, dass die bestmögliche Schranke keine Abschätzung mehr ist, sondern der korrekte Wert der Lösung. Die eigentliche Suche lässt sich somit auf die Ermittlung der Schranke reduzieren. Da die Berechnung des Graphabstands NP-vollständig ist, gilt dies somit auch für die Berechnung der idealen Schranke.

Ein Ansatz für eine solche Schranke war das Ermitteln eines bipartiten Matchings. Die dadurch ermittelte Schranke war jedoch nicht korrekt. Das Approximieren des Graphabstandes wäre ein weiterer möglicher Ansatz. Lässt sich zu einer möglichen Lösung sagen, wie weit sie höchstens vom Optimum ist, dann lässt sich daraus auch eine untere bzw. obere Schranke formulieren.

Aufwandbestimmende Eigenschaften

Bei der Untersuchung der Testergebnisse des B&B-Verfahrens zeigte sich, dass die Laufzeit von der Kantendichte abhängig sein kann. Dies führte zwar nicht zu einer unmittelbaren Verbesserung, jedoch begründet es die Vermutung, dass mittels weiterer Eigenschaften sich die Laufzeit besser vorhersagen lässt und man die Suche besser optimieren kann.

Mehr Kenntnisse über die Auswirkung verschiedener Grapheigenschaften auf den Aufwand könnte auch die Organisation der Suche verbessern. Im Rahmen dieser Arbeit gelang eine Verbesserung dadurch, dass die Knoten nach Anzahl ihrer Nachbarn sortiert wurden. Eventuell existieren bessere Sortierungen, welche die Suche ebenfalls beschleunigen.

6.1.5 Evolutionärer Algorithmus

Dieses lediglich aus Neugierde und Experimentierfreude motivierte Verfahren erwies sich als ein hinreichend schnelles Verfahren. Bei kleinen Graphenpaaren bedeutete der Mehraufwand für die Verwaltung der Population und das Durchlaufen der Generationen noch einen zeitlichen Nachteil gegenüber anderen Verfahren. Der Anstieg der Laufzeit ist jedoch deutlich geringer. Zusätzlich zur zufriedenstellenden Laufzeit erwies sich auch die Qualität der Lösung als hoch. Für kleine bis mittlere Graphenpaare besteht sogar eine gute Chance, die beste Lösung zu finden.

Potential für weitere Untersuchungen bieten unter anderem die Wahl der Populationsgröße sowie die Abbruchbedingung. Auch die für Rekombination und Mutation verwendeten Methoden sind nur erste Ansätze.

6.2 Konzept zur praktischen Umsetzung

Das vorgestellt Konzept befasst sich mit der Umsetzung der LV-Form, der Komplexitätsverringerung bei der Suche nach einem ECGM und der Suche nach Mustern.

6.2.1 Erweiterung

Das bestehende Konzept lässt sich erweitern, indem man Knoten und Kanten beschriftet. Mit einer solchen Beschriftung (engl. Label) ließe sich eventuell die Eingabe des Lerners besser nachvollziehen.

So könnte man auf die Vorgabe von Knoten verzichten. Eine solche Vorgabe ist einerseits hilfreich, um die Laufzeit beim Vergleichen der Graphen zu senken (siehe Abschnitt 5.2.1). Andererseits kann sie zur eindeutigen Identifikation von Knoten dienen. So unterscheidet sich ein LV-Graph, in dem Wölfe Schafe jagen, strukturell nicht von einem LV-Graph, in dem die Schafe Jagd auf Wölfe machen.

Statt einer solchen Vorgabe vergleicht man die Beschriftung des Lerners mit der Beschriftung der Musterlösung. Mittels der so genannten *Levenshtein-Distanz* zwischen den Beschriftungen, weist man dann die Knoten der Eingabe Knoten der Musterlösung zu. Die Levenshtein-Distanz lässt sich mit quadratischen Zeitaufwand ermitteln [19].

Eine weitere Möglichkeit wäre es (wie bereits in Abschnitt 2.5 erwähnt), die Kostenfunktion für ein ECGM um die Levenshtein-Distanz zu erweitern. Somit würde die Zuordnung indirekt bei der Suche nach einem ECGM erfolgen. Dabei gilt es zu beachten, dass es (bei entsprechender Wahl der Kosten) billiger sein kann einen Knoten zu löschen und einen neuen einzufügen statt die Beschriftung zu ändern. Satz 3 (Abschnitt 3.2.1) wäre dann nicht mehr gültig und somit kann auch die Laufzeit zur Ermittlung einer Lösung steigen. Es kann aber auch passieren, dass sich bessere Schranken für B&B formulieren lassen. Dies würde dann zu einer besseren Laufzeit führen.

6.2.2 Funktionsfähigkeit in der Praxis

Für alle Teile des Konzepts bleibt jedoch die Frage, ob sie in der Praxis funktionieren. So bietet der Ansatz der Knotenfärbung, also das Einteilen der Knoten in Gruppen, die Möglichkeit die Anzahl der möglichen Permutationen und somit die Laufzeit stark zu reduzieren. Eventuell ist dies aber nicht bei allen Modellen sinnvoll.

Auch die Nützlichkeit einer Mustersuche kann bei konkreten Modellen gering sein. Bereits die Voraussetzung dafür kann unbrauchbar sein. Streng genommen beschreibt der Graphabstand nur den Aufwand für den Umbau eines Graphen. Man könnte es als geringsten Reparaturaufwand bezeichnen. Es bleibt die Frage, ob sich mittels Graphabstand und Mustersuche auch die Fehler des Lerners rekonstruieren lassen. Es gibt einen Unterschied zwischen dem Zeigen, wie man es richtig macht, und

dem Erklären, was man falsch gemacht hat.

6.3 Bewerten ohne Musterlösung

Ein gänzlich anderer Ansatz ist es, keine Musterlösung vorzugeben. Stattdessen wird überprüft, ob die Eingabe des Lerners bestimmte Eigenschaften erfüllt. Dies bietet sich bei Modellen an, bei denen eine kleine strukturelle Änderung zu einer starken semantischen Änderung führt. Ebenfalls denkbar wäre ein solcher Ansatz, wenn eine große Zahl an möglichen korrekten Lösungen existiert.

Der Ansatz kann jedoch deutlich schwieriger umzusetzen sein. In einem hinreichend komplexen Modell können Eigenschaften nur schwer oder gar nicht überprüfbar sein. Ist ein Modell beispielsweise Turing-vollständig, dann lässt sich das Halteproblem für dieses Modell nicht lösen.

Ebenfalls fraglich ist, ob es hilfreich für den Lerner ist. Man kann dem Lerner zwar sagen, was an seiner Lösung fehlt, aber kann man ihm auch sagen, was er falsch gemacht hat?

6.4 Ausblick

Es gibt nun mehrere Punkte an denen man das hier vorgestellte weiterentwickeln kann. Aus algorithmischer Sicht wäre es interessant, bessere Schranken für das B&B-Verfahren zu finden. So werden beispielsweise in [20] eine untere und eine obere Schranke vorgestellt. Ebenfalls sinnvoll wäre eine gute Unterscheidungsmöglichkeit, wann B&B und wann ein evolutionärer Algorithmus verwendet werden soll.

Aus E-Learning-Sicht wären zwei Schritte interessant. Zum einen ist zu überprüfen, ob das in Kapitel 5 vorgestellte Konzept in der Praxis funktioniert. Zum anderen sollten mögliche Muster für die Auswertung der Fehlergraphen gefunden werden. In beiden Fällen können die Ergebnisse stark vom Modell abhängen, das dem Lerner vermittelt werden soll. Eventuell gibt es jedoch für einige Modelle bereits Erfahrungswerte, auf die zurückgegriffen werden kann.

Literaturverzeichnis

[1] Abu-Khzam, Faisal N. ; Samatova, Nagiza F. ; Rizk, Mohamad A. ; Langston, Michael A.: The Maximum Common Subgraph Problem: Faster Solutions via Vertex Cover. In: *Computer Systems and Applications, ACS/IEEE International Conference on* 0 (2007), S. 367–373. ISBN 1–4244–1030–4

[2] Bravo, Crescencio ; Van Joolingen, Wouter R. ; De Jong, Ton: Modeling and Simulation in Inquiry Learning: Checking Solutions and Giving Intelligent Advice. In: *Simulation* 82 (2006), November, S. 769–784. – ISSN 0037–5497

[3] Bunke, H.: On a relation between graph edit distance and maximum common subgraph. In: *Pattern Recogn. Lett.* 18 (1997), August, S. 689–694. – ISSN 0167–8655

[4] Conte, D. ; Guidobaldi, C. ; Sansone, C.: A comparison of three maximum common subgraph algorithms on a large database of labeled graphs. In: *Proceedings of the 4th IAPR international conference on Graph based representations in pattern recognition*. Berlin, Heidelberg : Springer-Verlag, 2003 (GbRPR'03). – ISBN 3–540–40452–X, 130–141

[5] Cook, Stephen A.: The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1971 (STOC '71), 151–158

[6] Frasconi, Paolo (Hrsg.) ; Kersting, Kristian (Hrsg.) ; Tsuda, Koji (Hrsg.): *Mining and Learning with Graphs, MLG 2007, Firence, Italy, August 1-3, 2007, Proceedings*. 2007

[7] Garey, Michael R. ; Johnson, David S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1990. – ISBN 0716710455

[8] Hart, Peter E. ; Nilsson, Nils J. ; Raphael, Bertram: A formal basis for the heuristic determination of minimum cost paths. In: *IEEE Transactions on Systems, Science, and Cybernetics* SSC-4 (1968), Nr. 2, S. 100–107

[9] Hart, Peter E. ; Nilsson, Nils J. ; Raphael, Bertram: Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *SIGART Bull.* (1972), December, S. 28–29. – ISSN 0163–5719

[10] Land, A. H. ; Doig, A. G.: An Automatic Method of Solving Discrete Programming Problems. In: *Econometrica* 28 (1960), Nr. 3, S. 497–520

[11] Munkres, J.: Algorithms for the Assignment and Transportation Problems. In: *Journal of the Society of Industrial and Applied Mathematics* 5 (1957), Nr. 1, S. 32–38

[12] Riesen, Kaspar: *Classification and Clustering of Vector Space Embedded Graphs*. Bern, Switzerland, Institut für Informatik und angewandte Mathematik, Universität Bern, Diss., 2009

[13] Riesen, Kaspar ; Fankhauser, Stefan ; Bunke, Horst: Speeding Up Graph Edit Distance Computation with a Bipartite Heuristic. In: **[6]**

[14] Schöning, Uwe: Graph isomorphism is in the low hierarchy. In: *J. Comput. Syst. Sci.* 37 (1988), December, S. 312–323. – ISSN 0022–0000

[15] Suters, W. H. ; Abu-khzam, Faisal N. ; Zhang, Yun ; Symons, Christopher T. ; Samatova, Nagiza F. ; Langston, Michael A.: *A New Approach and Faster Exact Methods for the Maximum Common Subgraph Problem*

[16] Wikipedia: *Isomorphie von Graphen — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Isomorphie_von_Graphen&oldid=77741052`. Version: 2010. – [Online; Stand 31. Januar 2011]

[17] Wikipedia: *Lotka-Volterra-Regeln — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Lotka-Volterra-Regeln&oldid=82636686`. Version: 2010. – [Online; Stand 12. März 2011]

[18] Wikipedia: *Graph isomorphism problem — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Graph_isomorphism_problem&oldid=407719426`. Version: 2011. – [Online; accessed 31-January-2011]

[19] Wikipedia: *Levenshtein-Distanz — Wikipedia, Die freie Enzyklopädie*. `http://de.wikipedia.org/w/index.php?title=Levenshtein-Distanz&oldid=86671458`. Version: 2011. – [Online; Stand 22. März 2011]

[20] Zeng, Zhiping ; Tung, Anthony K. H. ; Wang, Jianyong ; Feng, Jianhua ; Zhou, Lizhu: Comparing stars: on approximating graph edit distance. In: *Proc. VLDB Endow.* 2 (2009), August, S. 25–36. – ISSN 2150–8097

[21] Zhang, Kaizhong ; Wang, Jason ; Shasha, Dennis ; Jiang, Communicated T.: *On the Editing Distance between Undirected Acyclic Graphs*. 1995

Definitionsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

B&B	Branch and Bound
ECGM	error correcting graph matching
GI	Graphisomorphie
giTG	gemeinsamer induzierter Teilgraph
gTG	gemeinsamer Teilgraph
LV-Form	Lotka-Volterra-Form
LVG	Lotka-Volterra-Graph
LV-Regeln	Lotka-Volterra-Regeln
MCS	größter gemeinsamer induzierter Teilgraph (engl: maximum common subgraph)
MIS	maximum independent set
TGI	Teilgraphisomorphie

Änderungen

Diese Version der Arbeit enthält gegenüber der abgegeben Originalfassung vom 29. Juli 2011 einige Änderungen. Die meisten sind jedoch lediglich kosmetischer Natur.

Die nachfolgende Tabelle listet die Änderungen auf. Die dabei angegebenen Seitenzahlen beziehen sich auf die Seite in der abgegeben Originalfassung.

Seite		Änderung
i		generisches Maskulinum
		Position des Unilogos (Schrift auf gleicher höhe mit rechtem Text)
7	Abb. 2.2b	fehlerhafte Kanten entfernt
8	Abb. 2.3b	fehlerhafte Kanten entfernt
11	Abb. 2.4b	Knotenbeschriftung korrigiert
14	Abb. 2.6	Kanten sauber mit Knoten verbunden
		Abb. (a) und (b) getauscht
20	Abb. 3.2b	grüne Kante von $2b$ zu $3d$ gestrichelt statt durchgezogen
37	Absatz 2	korrektes $\mathcal{O}$-Zeichen für „$O(n^3)$“
53	Formeln ganz unten	Betragsstriche leicht vergrößert
54	erste Formel	„*sonst*“ in `\text{}`-Befehl gesetzt
72	gesamte Seite	Erklärung entfernt (nur für Abgabe erforderlich und ohne Unterschrift bedeutungslos)

verschiedene Seiten	Zeilenumbrüche vor oder nach Zahlen entfernt (z. B.: „Abbildung 4.4“)
	Schriebfehler korrigiert