Universität Bremen
Fachbereich 3, Mathematik und Informatik

Bachelorarbeit

Ein Algorithmus zum Finden von perfekten Matchings in bipartiten und allgemeinen Graphen

An algorithm for finding perfect matchings in bipartite and arbitrary graphs

Corinna Koch

1. Gutachterin: Dr. Sabine Kuske
2. Gutachter: Prof. Dr. Martin Gogolla

6. November 2018

Diese Erklärungen sind in jedes Exemplar der Abschlussarbeit mit einzubinden.

Name:_____ Matrikel-Nr.: _____

Urheberrechtliche Erklärung

Erklärung gem. § 10 (10) Allgemeiner Teil der BPO vom 27.10.2010

Hiermit versichere ich, dass ich meine Bachelorarbeit ohne fremde Hilfe angefertigt habe, und dass ich keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Die Bachelorarbeit darf nach Abgabe nicht mehr verändert werden.

Datum: _____ Unterschrift: _____

Erklärung zur Veröffentlichung von Abschlussarbeiten

Bitte auswählen und ankreuzen:

- ○ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ○ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs.2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ○ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum: _____ Unterschrift: _____

Inhaltsverzeichnis

1 Einleitung

Matchings werden in der Graphentheorie dazu genutzt, um Paare zu finden. Oft verwendet man sie in bipartiten Graphen, um bestimmte Probleme zu lösen.

Ein bekanntes Problem ist beispielsweise das Heiratsproblem, das mit dem Heiratssatz von Philip Hall gelöst werden kann [Bri05]. In dem Problem stehen sich je eine Gruppe von heiratswilligen Frauen und Männern gegenüber, wobei jede Frau nur bestimmte Männer heiraten will. Dies lässt sich nun einfach in einem bipartiten Graphen darstellen. Dabei wird zwischen einer Frau und einem Mann genau dann eine Kante gezogen, wenn die Frau sich vorstellen kann, diesen Mann zu heiraten. Die Frage des Heiratsproblems ist jetzt, ob jede Frau einen Mann findet (das wäre dann bei gleicher Anzahl von Frauen und Männeren ein perfektes Matching) oder ob wenigstens maximal viele Frauen einen Mann finden (natürlich lässt sich das Problem auch mit getauschten Geschlechtern aufstellen).

Wie im gerade vorgestellten Problem ist man also oft auf der Suche nach möglichst großen, wenn nicht gar perfekten Matchings. Für bipartite Graphen lässt sich hier schnell ein Algorithmus finden, der auch mit polynomiellem Aufwand noch eine Lösung findet. Gängige Algorithmen sind zum Beispiel der sogenannte *ungarische Algorithmus* [Bri05] und [Die10] oder der *Algorithmus von Hopcroft und Karp* [HK71]. Beide Algorithmen finden maximale Matchings in bipartiten Graphen, können also auch zur Suche nach perfekten Matchings genutzt werden.

Aber was, wenn man nun keinen bipartiten Graphen hat, in dem perfekte Matchings gefunden werden sollen? Ein weiteres Problem könnte beispielsweise sein, dass man in einer Gruppe von Personen Paare bilden will. Jede Person soll hinterher einen Partner haben. Die einzig weitere Bedingung, die gestellt wird, ist, dass die Personen sich kennen müssen. Ob die Personen eines Paares nun dem gleichen oder einem anderem Geschlecht angehören, ist egal. Auch müssen die Personen nicht zwei Gruppen zugeordnet werden können.

Die Bekanntheit zwischen den Personen lässt sich zwar als Graph darstellen, ein bipartiter Graph kommt nicht mehr in Frage. Nun sollen Paare gebildet werden. Jetzt lässt sich wieder die Frage stellen: Kann ein perfektes Matching gefunden werden? Kann also jede Person eine Partnerin bzw. einen Partner finden?

Diese Frage ist schwieriger zu lösen als das Heiratsproblem. In ihrem Buch „Graphentheoretische Konzepte und Algorithmen“ [KN12] bieten Sven Oliver Krumke und Hartmut Noltemeier einen Algorithmus an, der perfekte Matchings auch in nicht-bipartiten Graphen, im folgenden auch allgemeine Graphen genannt, findet. Dieser Algorithmus soll in dieser Arbeit vorgestellt werden.

Da der Algortihmus nicht ganz trivial ist, erläutern Krumke und Noltemeier ihn zunächst auf Basis von bipartiten Graphen und bauen ihn später zu einem Algorithmus für allgemeine Graphen aus.

Um das Verständnis des Algorihmus zu erleichern, wurde außerdem der Algorithmus für allgemeine Graphen implementiert und visualisiert. Eine Inspiration für die Darstellung des Algorithmus bot hier die Website zu Graphenalgorithmen der TU München [Mün].

Weiterführende Literatur zu Matchings, deren Anwendungen, sowie andere Algorithmen, die Matchings suchen, findet sich unter anderem bei [Gib85], [ADH98] und [Jun13].

1.1 Aufbau der Arbeit

Zunächst werden in Kapitel 2 alle für diese Arbeit wichtigen Definitionen und Konzepte zu Graphen vorgestellt.

In Kapitel 3 folgen dann alle benötigten Definitionen und Erläuterungen zum Thema Matchings.

Das Kapitel 4 handelt von den beiden eingangs erwähnten Algorithmen. Hier wird zunächst der Ablauf des Algorithmus *Bipartite-Perfect-Matching* für bipartite Graphen beschrieben und an einem Beispiel verdeutlicht. Im zweiten Teil des Kapitels schließt sich der Algorithmus *Perfect-Matching* an, welcher allgemeine Graphen als Eingabe akzeptiert. Auch dieser wird Schritt für Schritt erklärt und mit einem Beispiel verständlicher gemacht.

Die Implementierung und Visualisierung des Algorithmus *Perfect-Matching* wird in Kapitel 5 beschrieben.

Abschließend wird in Kapitel 6 nach einer kurzen Zusammenfassung ein Ausblick auf weitere Aspekte von Matchings gegeben und auf Verbesserungs- und Evolutionsmöglichkeiten der Implementierung eingegangen.

2 Graphen

In diesem Kapitel werden die Grundlagen für die in Kapitel 4 folgenden Algorithmen eingeführt. Zunächst werden dafür Graphen definiert. Darauf folgen wichtige Definitionen, wie beispielsweise die von Wegen und Kreisen, Zusammenhang, bipartiten Graphen und Bäumen. Am Ende des Kapitels wird das Konzept der Kreiskontraktion beschrieben. Literarische Basis für dieses Kapitel bilden [KN12] und [Kus18].

Ein *Graph* G ist ein Tupel $G = (V, E)$ mit einer nicht leeren Knotenmenge V und einer Kantenmenge E mit $E \subseteq \binom{V}{2}$. Jede Kante $e \in E$ ist also eine zweielementige Teilmenge aus Knoten: $e = \{v_1, v_2\}$ mit $v_1, v_2 \in V, v_1 \neq v_2$.

Die Knotenmenge eines Graphen wird auch mit $V(G)$ bezeichnet, die Kantenmenge entsprechend mit $E(G)$.

Visualisieren kann man Graphen, indem man die Knoten als Kreise darstellt und die Kanten als Verbindungslinien zwischen den jeweiligen Knoten zeichnet.

Sei zum Beispiel ein Graph $G = (V, E)$ gegeben mit:

$$
\begin{aligned}
V &= \{v_1, \cdots, v_{10}\} \\
E &= \{e_1, \cdots, e_{11}\} \\
&\text{mit } e_1 = \{v_1, v_2\}, e_2 = \{v_1, v_3\}, e_3 = \{v_3, v_7\}, e_4 = \{v_3, v_8\}, \\
&\quad e_5 = \{v_4, v_8\}, e_6 = \{v_4, v_5\}, e_7 = \{v_5, v_{10}\}, e_8 = \{v_6, v_7\}, \\
&\quad e_9 = \{v_7, v_8\}, e_{10} = \{v_8, v_9\}, e_{11} = \{v_9, v_{10}\}
\end{aligned}
$$

In Abbildung 2.1 ist dieser Graph visualisiert.

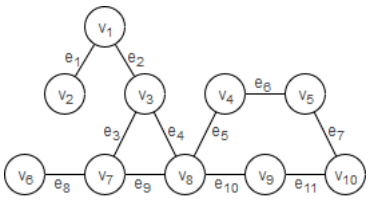


Abbildung 2.1: Visualisierung des Beispielgraphen G

Knoten- und Kantennamen können bei der Visualisierung auch weggelassen werden, wie es in Abbildung 2.2 zu sehen ist.

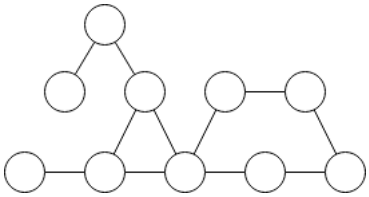


Abbildung 2.2: Visualisierung des Beispielgraphen G ohne Knoten- und Kantennamen

Dabei müssen Knoten in einem Graph nicht zwingend eine Kante haben oder komplett verbunden sein. Auch die Abbildung 2.3 stellt einen Graphen dar.

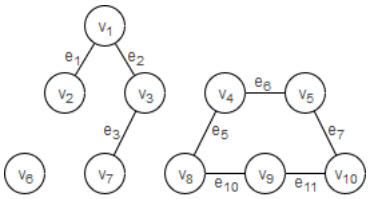


Abbildung 2.3: Auch ein Graph

Die in dieser Arbeit betrachteten Graphen sind nach Definition *einfach* und ohne Schlingen. In Abbildung 2.4 sind zwei Graphen zu erkennen, wie sie in dieser Arbeit betrachtet werden können. Nicht erlaubte Graphen sind dagegen in Abbildung 2.5 dargestellt.

Abbildung 2.4: Beispiel für in dieser Arbeit betrachtete Graphen

(a) Graph mit Parallelkante

(b) Graph mit Schlinge

Abbildung 2.5: Beispiel für in dieser Arbeit **nicht** betrachtete Graphen

Bei der Durchführung des Algorithmus auf allgemeinen Graphen kann es zeitweise zu Parallelkanten kommen (mehr dazu in Abschnitt 2.7). Diese werden im Laufe des Algorithmus auch wieder aufgelöst und sind daher nur aus technischer Sicht wichtig, weswegen sie hier auch nicht formal definiert werden.

2.1 Teilgraphen

Sei $G = (V, E)$ ein Graph. Dann ist ein Graph $G' = (V', E')$ ein *Teilgraph* von G (geschrieben $G' \subseteq G$), wenn $V' \subseteq V$ und $E' \subseteq E$. Der Graph aus Abbildung 2.3 ist beispielsweise ein Teilgraph des Graphen G aus Abbildung 2.1.

2.2 Beziehungen zwischen Knoten und Kanten

Knoten und Kanten eines Graphen $G = (V, E)$ können in verschiedenen Beziehungen zueinander stehen.

Ein Knoten $v \in V$ und eine Kante $e \in E$ *inzidieren*, wenn v einer der Knoten von e ist:

$$v \text{ und } e \text{ sind inzident } \Leftrightarrow v \in e$$

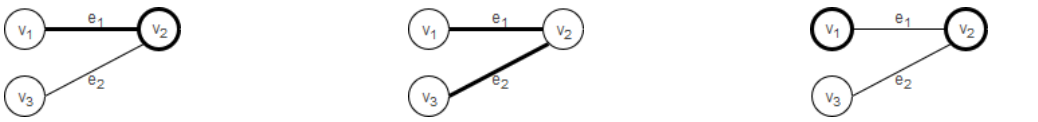
Zwei Kanten $e_1, e_2 \in E$ heißen *inzident*, wenn es einen Knoten v gibt, der mit beiden Kanten inzidiert. Es gilt also:

$$e_1 \text{ und } e_2 \text{ sind inzident } \Leftrightarrow \exists v \in V \text{ mit } v \in e_1 \text{ und } v \in e_2$$

Zwei Knoten $v_1, v_2 \in V$ heißen *adjazent* oder *benachbart*, wenn eine Kante e existiert, die zu v_1 und zu v_2 inzident ist, die also die Knoten verbindet:

$$v_1 \text{ und } v_2 \text{ sind adjazent } \Leftrightarrow \exists e \in E \text{ mit } e = \{v_1, v_2\}$$

Die Beziehungen zwischen Knoten und Kanten sind in der Abbildung 2.6 dargestellt.



(a) v_2 und e_1 sind inzident (b) e_1 und e_2 sind inzident (c) v_1 und v_2 sind adjazent

Abbildung 2.6: Beziehungen zwischen Knoten und Kanten

2.3 Wege und Kreise

Ein *Weg* P ist eine endliche Folge von Knoten und Kanten:

$$P = v_0 e_1 v_1 e_2 v_2 \cdots e_k v_k \text{ mit } v_0, \ldots, v_k \in V, e_1, \ldots, e_k \in E \\ \text{und } e_i = \{v_{i-1}, v_i\} \text{ für } i = 1, \ldots, k$$

Die *Länge* eines Weges $|P|$ ist definiert als die Anzahl der besuchten Kanten. Für $P = v_0 e_1 v_1 e_2 v_2 \cdots e_k v_k$ gilt also $|P| = k$.

Der erste Knoten eines Weges wird *Startknoten*, der letzte Knoten eines Weges *Endknoten* genannt.

Wie auch beim Graphen bezeichnet man mit $V(P)$ die Knotenmenge und mit $E(P)$ die Kantenmenge des Weges.

Ein *Kreis* C ist ein Weg, bei dem Anfangs- und Endknoten gleich sind und bei dem gilt:

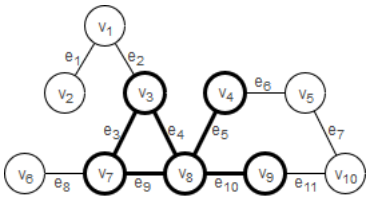
- $e_i \neq e_{i+1}$ für $i = 1, \ldots, k-1$,
- $e_k \neq e_1$,
- $k > 2$

Aufeinander folgende Kanten eines Kreises dürfen also nicht identisch sein und die Länge eines Kreises muss größer sein als 2. Die beiden in Abbildung 2.7 dargestellten Graphen haben also keine Kreise.

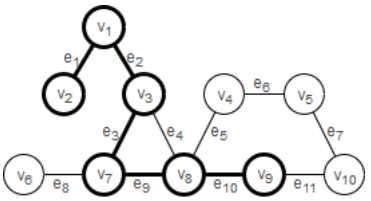
Abbildung 2.7: Beide Graphen enthalten keine Kreise

Wenn jeder Knoten in einem Weg höchstens einmal durchlaufen wird, spricht man von einem *einfachen Weg*. Sind alle Knoten bis auf Anfangs- und Endknoten paarweise verschieden, spricht man von einem *einfachen Kreis*.

In Abbildung 2.8a ist ein Weg P mit $P = v_4e_5\boldsymbol{v_8}e_4v_3e_3v_7e_9\boldsymbol{v_8}e_{10}v_9$ dargestellt. Dieser Weg ist nicht einfach, da der Knoten v_8 zweimal durchlaufen wird. Der Weg P' in Abbildung 2.8b mit $P' = v_2e_1v_1e_2v_3e_3v_7e_9v_8e_{10}v_9$ dagegen ist ein einfacher Weg.



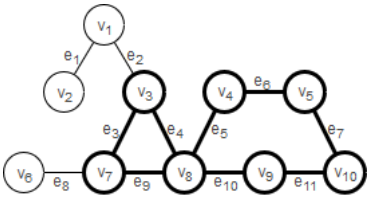
(a) Weg P mit der Länge $|P| = 5$



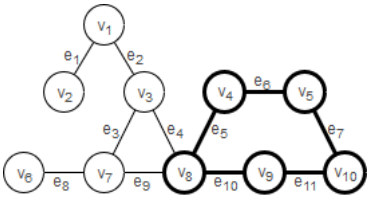
(b) einfacher Weg P' mit der Länge $|P'| = 5$

Abbildung 2.8: Beispiel für einen Weg und einen einfachen Weg

Der in Abbildung 2.9a dargestellte Kreis $C = v_4e_5\boldsymbol{v_8}e_4v_3e_3v_7e_9\boldsymbol{v_8}e_{10}v_9e_{11}v_{10}e_7v_5e_6v_4$ durchquert den Knoten v_8 zweimal, ist also nicht einfach. Der Kreis $C' = v_4e_5v_8e_{10}v_9e_{11}v_{10}e_7v_5e_6v_4$ aus Abbildung 2.9b dagegen ist ein einfacher Kreis.



(a) Kreis C



(b) einfacher Kreis C'

Abbildung 2.9: Beispiel für einen Kreis und einen einfachen Kreis

2.4 Zusammenhang

Ein Graph $G = (V, E)$ heißt *zusammenhängend*, wenn es zwischen allen Knotenpaaren v und v' mit $v, v' \in V$ einen Weg von v nach v' gibt. Der Graph in Abbildung 2.1 ist also zusammenhängend, der Graph in Abbildung 2.3 nicht.

Seien G_1 und G_2 Teilgraphen von G. G_2 ist ein *größerer Teilgraph* als G_1 (geschrieben $G_1 \subsetneq G_2$), wenn gilt:

$$G_1 \subseteq G_2 \text{ und } G_1 \neq G_2$$

Ein Teilgraph $G_1 \subseteq G$ ist eine *Zusammenhangskomponente* des Graphen G, wenn G_1 zusammenhängend ist und für alle größeren Teilgraphen $G_2 \subseteq G$ mit $G_1 \subsetneq G_2$ gilt, dass G_2 nicht zusammenhängend ist.

Der Graph in Abbildung 2.3 besteht also aus drei Zusammenhangskomponenten. Ein zusammenhängender Graph besteht dagegen nur aus einer Zusammenhangskomponente, nämlich sich selbst.

2.5 Bipartite Graphen

Ein *bipartiter Graph* ist ein Graph, bei dem sich die Knotenmenge in zwei Gruppen oder Partitionen aufteilen lässt. Knoten innerhalb derselben Gruppe dürfen nicht untereinander verbunden sein. Für die Knotenmenge V und die Kantenmenge E eines Graphen $G = (V, E)$ gilt deshalb:

$$V = A \cup B \text{ mit } A \cap B = \varnothing \text{ und } A, B \subseteq V$$
$$e \cap X \neq \varnothing \text{ für alle } e \in E \text{ und alle } X \in \{A, B\}$$

Bipartite Graphen werden oft dafür genutzt, um mögliche Beziehungen zwischen zwei Gruppen darzustellen. So könnte die eine Gruppe aus Frauen, die andere aus Männern bestehen. Mit einem bipartiten Graphen kann man dann beispielsweise darstellen, welche Frauen welche Männer heiraten wollen, wie im eingangs erwähnten Hochzeitsproblem. Solch ein Graph ist in Abbildung 2.10 dargestellt.

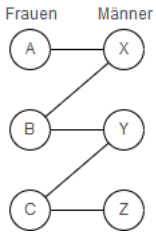


Abbildung 2.10: Bipartiter Graph zur Darstellung der Bekanntheit zwischen Frauen und Männern

Nicht bipartite Graphen werden in dieser Arbeit auch als *allgemeine Graphen* bezeichnet, um hervorzuheben, dass hier keine zwei Partitionen gefunden werden können.

2.6 Bäume

Ein kreisfreier, zusammenhängender Graph, wie in Abbildung 2.11 zu sehen ist, wird auch *Baum* genannt.

Innerhalb eines Baumes T kann ein beliebiger Knoten $s \in V(T)$ als *Wurzel* festgelegt werden. Jeder andere Knoten v des Baumes kann mit einem einfachen Weg von s aus

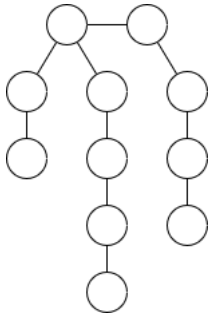


Abbildung 2.11: Ein Baum

erreicht werden. Ein Knoten v', welcher adjazent zu einem Knoten v ist und für den gilt, dass die Länge des Weges von v' zur Wurzel s (um genau 1) länger ist als der Weg für v zu s, heißt *Nachfolger von* v.

Aus jedem Baum T kann ein bipartiter Graph erstellt werden, indem man sich zunächst einen Knoten $s \in V(T)$ als Wurzel definiert. Diesen ordnet man der ersten Gruppe A zu. Alle Nachfolger von v werden der zweiten Gruppe B zugeordnet. Die Nachfolger dieser Knoten werden wiederum A zugeordnet und so weiter. Da ein Baum kreisfrei ist, kann so problemlos ein bipartiter Graph erstellt werden (siehe Abbildung 2.12).

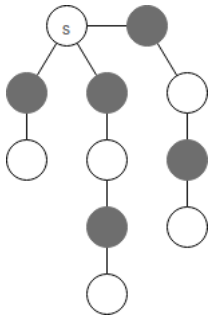


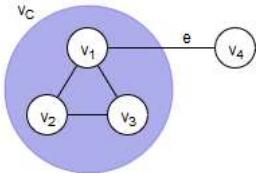
Abbildung 2.12: Bipartiter Graph, Knoten einer Gruppe haben die gleiche Farbe

2.7 Kreiskontraktion

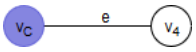
Eine spezielle Konstruktion ist die Kontraktion von Kreisen zu sogenannten Superknoten, welche für den Algorithmus für allgemeine Graphen benötigt wird.

Ein *Superknoten* v_C ist ein Knoten, der einen Kreis C enthält. Die im Kreis enthaltenen Knoten können wiederum Superknoten sein. Kanten können als Endknoten zwar Superknoten haben, sind aber eigentlich mit einem Knoten innerhalb des Superknotens verbunden, wie in Abbildung 2.13 zu sehen ist. Der Knoten v_4 ist dort eigentlich mit dem Knoten v_1 durch die Kante e verbunden. Während ein Superknoten besteht, werden diese Kanten allerdings so behandelt, als wären sie mit dem Superknoten verbunden. Die Kante e wird also als $e = \{v_C, v_4\}$ angesehen.

Superknoten entstehen durch die Kontraktion eines Kreises (mit ungerader Länge) in einem Graphen:



(a) Eigentliche Sicht auf den Superknoten



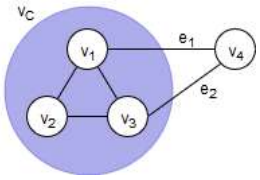
(b) Komprimierte Sicht auf den Superknoten

Abbildung 2.13: Darstellung von Superknoten, diese werden durch blaue Kreise dargestellt.

Sei $G = (V, E)$ ein Graph und $C = v_1\{v_1, v_2\}v_2 \cdots v_k\{v_k, v_1\}v_1$ ein Kreis mit $k = 2n + 1$ für $k, n \in \mathbb{N}$. Bei der Kontraktion werden die Knoten $v_1, \ldots, v_k \in V(C)$ und die Kanten in $E(C)$ zu einem Superknoten v_C zusammengefasst. Bei Kanten, die nur einen Endknoten im Kreis C haben, wird der entsprechende Endknoten $v \in V(C)$ durch den Superknoten v_C ersetzt. Man schreibt für die Kontraktion eines Kreises C im Graphen G auch G/C.

Dadurch kann es auch zu Parallelkanten kommen, wie in Abbildung 2.14 zu sehen ist. Hier wird der Kreis C bestehend aus den Knoten v_1, v_2 und v_3 zu einem Superknoten v_C kontrahiert. Die Kanten $e_1 = \{v_1, v_4\}$ und $e_2 = \{v_3, v_4\}$ werden dadurch zu den Kanten $e_1 = \{v_C, v_4\}$ und $e_2 = \{v_C, v_4\}$, sind jetzt also parallel. Diese Parallelität ist aber nicht weiter von Bedeutung, da sie im Algorithmus später wieder aufgelöst wird.

(a) Der Kreis aus den Knoten v_1, v_2 und v_3 wird zum Superknoten v_C kontrahiert



(b) Die Kanten e_1 und e_2 sind nicht wirklich parallel.

Abbildung 2.14: Kontraktion kann zu Parallelkanten führen

Ein Hinweis am Rande: Die Kontraktion könnte genauso gut mit geraden Kreisen durchgeführt werden, im Algorithmus wird sie aber explizit nur für ungerade Kreise benötigt und ist daher hier auch so definiert.

3 Matchings

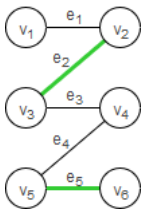
Dieses Kapitel behandelt Grundlagen zu Matchings, sowie Konzepte zur Matching-Erweiterung. Am Anfang des Kapitels werden Matchings definiert. Um Matchings zu finden und zu erweitern, werden alternierende und augmentierende Wege eingeführt. Die alternierenden Wege werden im letzten Teil des Kapitels zu alternierenden Bäumen fortgesetzt. Hier wird nicht nur definiert, was ein alternierender Baum ist, sondern auch beschrieben, wie solch ein Baum wächst, wie man mit Hilfe des Baumes ein Matching erweitern kann und was ein verkümmerter Baum ist.

3.1 Matchings

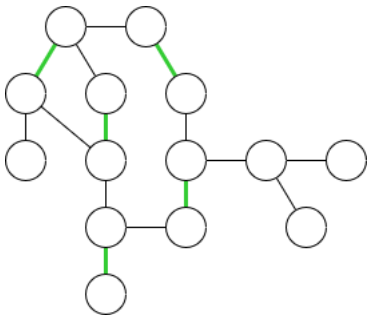
Ein *Matching* M in einem Graphen $G = (V, E)$ ist eine Teilmenge der Kantenmenge E, in welcher keine Kante mit einer anderen inzident ist.

In Abbildung 3.1a ist ein Matching in einem bipartiten Graphen zu sehen. Dieses besteht aus den Kanten e_2 und e_5. Die anderen Kanten e_1, e_3 und e_4 können nicht hinzugefügt werden, da sie mit den Kanten aus dem Matching inzidieren würden.

Ein weiteres Matching, dieses Mal in einem allgemeinen Graphen, ist in Abbildung 3.1b dargestellt.



(a) Matching M, markiert durch dicke grüne Linien



(b) Ein Matching in einem allgemeinen Graphen

Abbildung 3.1: Beispiele für Matchings

Die Knoten in einem Graphen $G = (V, E)$ mit einem Matching M sind entweder M-frei oder M-überdeckt. Dabei ist ein Knoten $v \in V$ *M-frei*, wenn keine Kante $e \in M$ existiert, mit der v inzidiert. Umgekehrt ist ein Knoten v *M-überdeckt*, wenn es eine Kante $e \in M$ gibt, sodass v und e inzident sind.

Die Knoten v_1 und v_4 im Graphen aus Abbildung 3.1a sind beispielsweise M-frei, während die Knoten v_2, v_3, v_5 und v_6 M-überdeckt sind.

Ein *perfektes Matching* M in einem Graphen $G = (V, E)$ ist ein Matching, das alle Knoten V überdeckt. Das Matching M in Abbildung 3.1a ist daher kein perfektes Matching. Das Matching $M' = \{e_1, e_3, e_5\}$ dagegen ist perfekt (siehe auch Abbildung 3.3).

3.2 Alternierende und augmentierende Wege

Zum Finden und Vergrößern von Matchings haben sich alternierende und augmentierende Wege als nützlich erwiesen.

Gegeben sei ein Graph $G = (V, E)$ und ein Matching M in G. Ein *M-alternierender Weg* P ist ein einfacher Weg, bei dem die Kanten in P abwechselnd im Matching und nicht im Matching sind (siehe Abbildung 3.2). Wenn beide Endknoten des Weges P nicht vom Matching überdeckt werden, nennt man den Weg *augmentierenden Weg*. Der Weg $P = v_1e_1v_2e_2v_3e_3v_4$ in Abbildung 3.2 ist also ein augmentierender Weg.

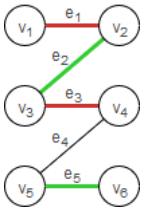


Abbildung 3.2: ein möglicher M-alternierender Weg P zum Matching M aus Abbildung 3.1a, die Kanten, welche nicht im Matching sind, sind rot markiert

Anhand eines M-augmentierenden Weges P lässt sich das Matching M verändern. Dazu „tauscht“ man die Kanten aus: man nimmt alle Kanten aus dem Weg, die vorher nicht im Matching waren, zum Matching dazu und entfernt alle Kanten aus dem Matching, die vorher im Matching waren und gleichzeitig auf dem Weg P liegen.

Für eine Menge M' gilt also:

$$M' = \Big(M \cup E(P)\Big) \setminus \Big(M \cap E(P)\Big)$$

Aufgrund der Eigenschaften des augmentierenden Weges P, das beide Endknoten von P M-frei sind, ist M' wieder ein Matching, welches außerdem größer ist als das vorherige Matching M.

Daher kann man mit dem in Abbildung 3.2 gefundenen augmentierenden Weg das Matching vergrößern (siehe Abbildung 3.3).

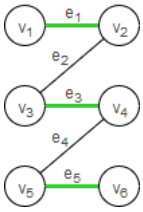


Abbildung 3.3: Vergrößertes Matching

3.3 Alternierende Bäume

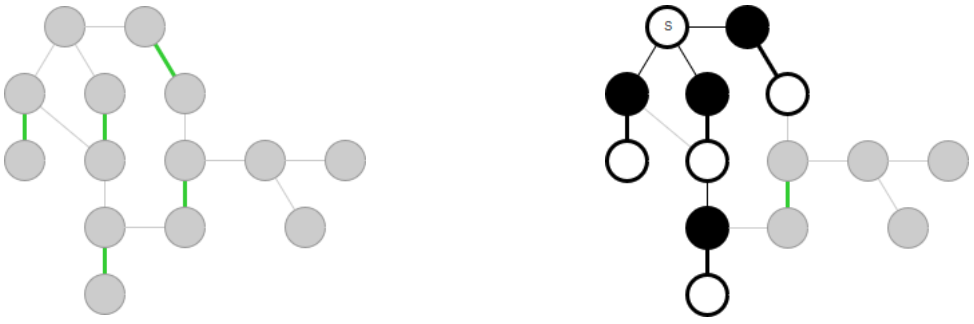
Zusätzlich zu alternierenden Wegen kann man auch alternierende Bäume definieren.

Gegeben seien ein Graph $G = (V, E)$ mit einem Matching M in G. Dann ist ein Baum T mit $T \subseteq G$ ein *alternierender Baum*, wenn folgendes gilt:

- Es gibt genau einen Knoten $s \in V(T)$, der M-frei ist. Dieser Knoten wird *Wurzel* von T genannt.
- Jeder Weg in T von der Wurzel s zu einem Knoten $v \in V(T)$ ist ein M-alternierender Weg.

Die Knoten eines alternierenden Baumes werden in zwei Knotenmengen partitioniert, welche im Folgendem mit $even(T)$ und $odd(T)$ bezeichnet werden. Die Entfernung eines Knotens $v \in V(T)$ von der Wurzel $s \in V(T)$ bestimmt dabei, in welcher Knotenmenge ein Knoten liegt. Knoten, die eine gerade Anzahl an Kanten zwischen sich und der Wurzel aufweisen, liegen in $even(T)$, die anderen in $odd(T)$. Die Wurzel s hat selbstverständlich einen Abstand von 0 zu sich selbst und liegt somit in $even(T)$. Knoten in $even(T)$ bzw. $odd(T)$ werden umgangssprachlich auch als „Knoten innerhalb der geraden bzw. ungeraden Knoten von T“ bezeichnet.

Bei der Visualisierung von alternierenden Bäumen werden die Knoten je nach Zugehörigkeit zu $even(T)$ bzw. $odd(T)$ unterschiedlich dargestellt und die Kanten des Matchings hervorgehoben. Im Graphen G, welcher in Abbildung 3.4a dargestellt ist, können alternierende Bäume gefunden werden, zum Beispiel der in Abbildung 3.4b dargestellte Baum T. Bei diesem sind alle Knoten aus $even(T)$ weiß, die Knoten aus $odd(T)$ dagegen schwarz. Die Kanten des Matchings M, welche im Baum liegen, sind mit dicken, schwarzen Linien gekennzeichnet, während die Kanten des Matchings außerhalb des Baumes weiterhin in grün gezeichnet werden. Alle anderen nicht zum Baum gehörigen Knoten und Kanten sind grau dargestellt.



(a) Graph G mit Matching M

(b) Ein M-alternierender Baum T mit Wurzel s

Abbildung 3.4: Ein M-alternierender Baum im Graphen G

3.3.1 Wachsen eines alternierenden Baumes

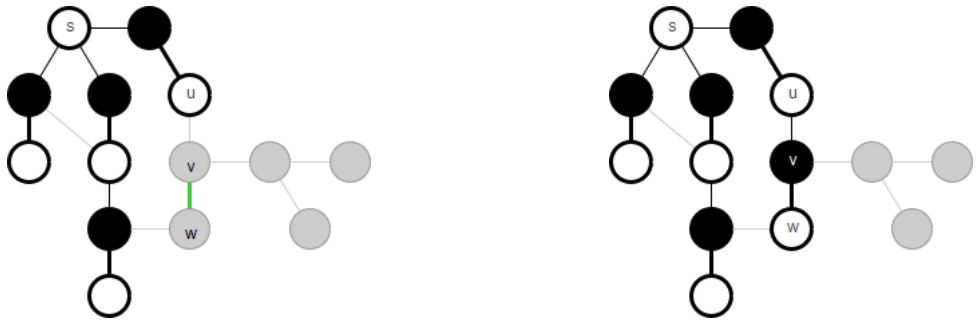
Wachsen kann ein alternierender Baum $T = (V_T, E_T)$ innerhalb eines Graphen $G = (V, E)$ mit $T \subseteq G$, wenn ein Knoten $v \in V \setminus V_T$ existiert, welcher mit einem Knoten $u \in even(T)$ adjazent ist und welcher von einer Kante $\{v, w\}$ in dem zum Graphen G gehörenden

Matching M überdeckt ist. Die Knoten v und w und die Kanten $\{u, v\}$, $\{v, w\}$ werden dann zu dem Baum hinzugefügt. Für den gewachsenen Baum T' gilt dann also:

$$\begin{aligned} T' &= (V_{T'}, E_{T'}) \\ \text{mit } V_{T'} &= V_T \cup \{v, w\} \\ \text{und } E_{T'} &= E_T \cup \Big\{\{u, v\}, \{v, w\}\Big\} \end{aligned}$$

Innerhalb eines alternierenden Baumes T haben Knoten in $odd(T)$ immer genau einen Nachfolger. Dies liegt daran, dass ein Baum nur an den Knoten aus $even(T)$ erweitert werden kann und immer um zwei Knoten längs eines Matchings wächst. Alles andere würde der Definition eines alternierenden Baumes widersprechen (siehe 3.3). Der Knoten v liegt also nach dem Wachsen in $odd(T)$, während der Knoten w in $even(T)$ zu finden ist.

Das Wachsen eines Baumes ist auch in Abbildung 3.5 zu erkennen. Der Graph aus Abbildung 3.5a enthält die Knoten u, v und w, die die oben genannten Bedingungen erfüllen. Der gewachsene Baum ist in Abbildung 3.5b zu erkennen.



(a) Graph mit erweiterbarem Baum

(b) Gewachsener Baum

Abbildung 3.5: Beispiel für das Wachsen eines Baumes

3.3.2 Erweiterung von Matchings

Mit Hilfe von alternierenden Bäumen können Matchings erweitert werden.

Sei ein Graph $G = (V, E)$, ein Matching $M \subseteq E$ und ein alternierender Baum $T = (V_T, E_T)$ mit $T \subseteq G$ gegeben. Sei $u \in even(T)$ und sei P der Weg in T von der Wurzel s zu u. Wenn ein M-freier Knoten $v \in V \setminus V_T$ existiert, der mit u adjazent ist, ist der Weg $P' = P\{u, v\}v$ ein augmentierender Weg in G, da sowohl s als auch v M-frei sind. Das Matching kann jetzt längs dieses augmentierenden Weges erweitert werden. Dazu werden, wie bereits beschrieben, alle Kanten, die im Matching M und auf dem Weg P' liegen, aus dem Matching entfernt. Stattdessen werden die Kanten von P', die nicht in M liegen, zum Matching hinzugefügt. Dies ist auch in Abbildung 3.6 zu erkennen.

Da der Baum über das Matching definiert war, kann er durch das geänderte Matching nicht mehr bestehen.

(a) Alternierender Baum (b) Erweitertes Matching

Abbildung 3.6: Beispiel für das Erweitern eines Matching mit Hilfe eines alternierenden Baumes

3.3.3 Verkümmerter Baum

Ein alternierender Baum T in $G = (V, E)$ ist ein *verkümmerter Baum*, wenn für jeden Knoten $v \in even(T)$ alle zu v adjazenten Knoten in $odd(T)$ liegen. Der Baum T kann dann nicht mehr weiter wachsen und auch das zu T gehörende Matching kann nicht mehr erweitert werden.

Für alle anderen Knoten $v \in V \setminus V(T)$ gilt, dass diese entweder gar nicht mit Knoten aus dem Baum adjazent sind oder nur mit einem ungeraden Knoten.

Der in Abbildung 3.7 dargestellte Baum ist zum Beispiel verkümmert.

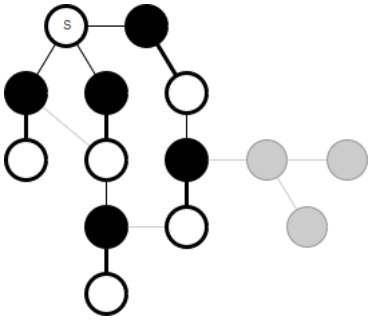


Abbildung 3.7: Verkümmerter Baum T

Ein verkümmerter Baum, kann kein perfektes Matching haben [KN12].

4 Algorithmen für perfekte Matchings

Dieses Kapitel stellt zwei Algorithmen zum Finden von perfekten Matchings vor. Zunächst wird in Kapitel 4.1 ein Algorithmus für bipartite Graphen erläutert. Dieser bietet einen guten Einstieg in das Thema und soll helfen, den in Kapitel 4.2 vorgestellten komplizierteren Algorithmus für allgemeine Graphen zu verstehen. Zunächst wird in den Unterkapiteln Schritt für Schritt der jeweilige Algorithmus erklärt. Darauf folgt der jeweilige Algorithmus in Pseudocode, sowie ein Beispiel.

Die Idee für das Finden von perfekten Matchings in Graphen besteht in den Algorithmen aus der Erstellung und dem Wachsen von alternierenden Bäumen oder dem Auffinden von verkümmerten Bäumen im Eingabegraphen. Mithilfe dieser Bäume kann ein anfangs noch leeres Matching nach und nach erweitert werden. Dies ist in bipartiten Graphen einfach umzusetzen, da hier keine Kreise ungerader Länge entstehen können. Für allgemeine Graphen muss dagegen noch die Kreis-Kontraktion in den Algorithmus eingebaut werden. Diese basiert auf einer Idee von Jack Edmonds [Edm65].

Beide Algorithmen akzeptieren als Eingabe nur einfache, zusammenhängende Graphen ohne Superknoten. Möchte man perfekte Matchings in einfachen, aber nicht-zusammenhängenden Graphen finden, so muss man den jeweiligen Algorithmus für jede Zusammenhangskomponente des fraglichen Graphen ausführen. Wenn alle Zusammenhangskomponenten ein perfektes Matching haben, so bildet die Vereinigung dieser Matchings ein perfektes Matching für den Gesamtgraphen. Leere Graphen sind dabei auch erlaubt.

Der Nachweis der Korrektheit dieses Algorithmus wird in [KN12] bewiesen und steht nicht im Fokus dieser Bachelorarbeit.

4.1 Algorithmus für perfekte Matchings in bipartiten Graphen

Im Folgenden wird Schritt für Schritt der Algorithmus *Bipartite-Perfect-Matching* erklärt, welcher innerhalb eines bipartiten Graphen nach einem perfektem Matching sucht. Anschließend folgt der Algorithmus in Pseudo-Code und ein Beispiel zum besseren Verständnis.

Ein- und Ausgabe Als Eingabe bekommt der Algorithmus einen einfachen, zusammenhängenden bipartiten Graphen $G = (V, E)$.

Ausgegeben wird entweder ein perfektes Matching M oder, wenn ein solches nicht existiert, die Meldung „G besitzt kein perfektes Matching“.

Variableninitialisierung Zunächst wird das Matching M mit der leeren Menge initialisiert. Dieses wird innerhalb des Algorithmus erweitert werden.

Prüfe, ob das aktuelle Matching perfekt ist Im nächsten Schritt wird nun geprüft, ob es sich bei dem aktuellem Matching M um ein perfektes Matching im Graphen G handelt. Dazu muss jeder Knoten des Graphen G M-überdeckt sein.

Handelt es sich bei M um ein perfektes Matching, so gibt der Algorithmus dieses Matching aus und terminiert. Wenn M kein perfektes Matching ist, muss geprüft werden, ob das Matching erweitert oder angepasst werden kann. Dazu erstellt der Algorithmus einen alternierenden Baum und versucht, den Baum oder das Matching wachsen zu lassen.

Initialisierung des alternierenden Baumes Um einen alternierenden Baum zu initialisieren, wird zunächst ein Knoten benötigt, der als Wurzel dienen kann. Hierzu wird ein M-freier Knoten s gewählt, um anschließend den alternierenden Baum T mit $T = (\{s\}, \varnothing)$ zu initialisieren. $even(T)$ besteht dann aus dem Knoten s und $odd(T)$ ist die leere Menge.

Suche eine Kante, mit der der Baum wachsen oder das Matching erweitert werden kann Nun muss eine Kante gesucht werden, mit der der Baum oder das Matching vergrößert werden kann. Findet sich keine solche Kante, folgt daraus, dass der Baum verkümmert ist. Wie bereits in Kapitel 3 beschrieben, müssen für die gesuchte Kante $\{u, v\} \in E$ die folgenden Bedingungen gelten: Der Knoten u muss sich in den geraden Knoten des Baumes befinden und der Knoten v darf nicht in den Knoten des Baumes $V(T)$ sein. Für $\{u, v\}$ muss also gelten:

$$\{u, v\} \in E \text{ mit } u \in even(T) \text{ und } v \notin V(T)$$

Ist der Baum verkümmert, terminiert der Algorithmus an dieser Stelle mit der Nachricht, dass der Eingabegraph G kein perfektes Matching besitzt. Ein bipartiter zusammenhängender Graph mit einem verkümmerten Baum kann, wie in Kapitel 3.3.3 beschrieben, kein perfektes Matching haben.

Ist der Baum dagegen nicht verkümmert, existiert also eine Kante $\{u, v\}$ mit den oben genannten Bedingungen, läuft der Algorithmus weiter. Nun folgt eine Fallunterscheidung, in der überprüft wird, ob der Baum oder das Matching erweitert werden kann.

Fall 1: v ist M-frei Wenn v M-frei ist, kann das Matching erweitert werden. In diesem Fall wird das Matching längs des alternierenden Weges P erweitert, welcher von v bis zur Wurzel s von T geht, wie es unter Abschnitt 3.3.2 beschrieben ist.

Da der alternierende Baum T über das Matching definiert ist (siehe 3.3), welches nun geändert wurde, ist er nicht mehr gültig und wird auf einen leeren Graphen zurückgesetzt. Es gilt jetzt also $T = (\varnothing, \varnothing)$.

Anschließend springt der Algorithmus zu dem Schritt **Prüfe, ob das aktuelle Matching perfekt ist**.

Fall 2: v ist M-überdeckt Ist v von einer Kante $\{v, w\} \in M$ überdeckt, für die gilt, dass der Knoten w nicht im Baum liegt, kann der Baum wachsen. Er wird dann um die Knoten v, w und die Kanten $\{u, v\}, \{v, w\}$ erweitert. Sei T' der vorherige Baum. Wie in Kapitel 3.3.1 definiert, besteht der gewachsene Baum T dann aus den Knoten $V(T) = V(T') \cup \{v, w\}$ und den Kanten $E(T) = E(T') \cup \big\{\{u, v\}, \{v, w\}\big\}$. Die geraden Knoten gewachsenen Baumes sind dann $even(T) = even(T') \cup \{w\}$ und die ungeraden $odd(T) = odd(T') \cup \{v\}$.

Nach dem Erweitern des Baumes muss wieder geprüft werden, ob der Baum verkümmert ist oder der Algorithmus fortgesetzt werden kann. Der Algorithmus geht also zu dem Schritt **Suche eine Kante, mit der der Baum wachsen oder das Matching erweitert werden kann**.

Algorithmus in Pseudo-Code Der Algorithmus wurde in dem Buch von Krumke und Noltemeier durch einen dem folgenden ähnlichem Pseudo-Code zur Verfügung gestellt.

```
BIPARTITE-PERFECT-MATCHING(G, M)
1  Setze M := ∅
2  if M ist ein perfektes Matching then
3       return M
4  Wähle einen M-freien Knoten s und setze T := ({s}, ∅)
5  if es gibt eine Kante {u, v} ∈ E mit u ∈ even(T) und v ∉ V(T) then
6       if v ist M-frei then
7            Vergrößere das Matching M längs
             des gefundenen augmentierenden Weges.
8            goto 2
9       if v ist von {v, w} ∈ M überdeckt mit w ∉ V(T) then
10           Füge u, v und die Kanten {u, v}, {v, w} zu T hinzu.
11           goto 5
12 Der Baum T ist verkümmert.
13 return "G besitzt kein perfektes Matching".
```

Beispiel Zum besseren Verständnis folgt nun ein beispielhafter Durchlauf des Algorithmus. Als Eingabe soll der Graph $G = (V, E)$ aus Abbildung 4.1 dienen. Es ist leicht zu sehen, dass das perfekte Matching am Ende der Menge $\big\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5, v_6\}\big\}$ entsprechen muss.

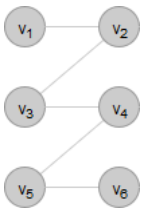


Abbildung 4.1: Bipartiter Beispielgraph G

Im ersten Schritt des Algorithmus wird ein Matching M mit der leeren Menge initialisiert. Daraufhin erfolgt die Überprüfung, ob dieses Matching perfekt ist. Da M aber ja noch leer ist, schlägt die Überprüfung fehl und ein alternierender Baum T wird initialisiert.

In diesem Beispiel wird hierzu zunächst der Knoten v_2 als Wurzel s gewählt. Der dadurch entstehende Baum $T = (\{v_2\}, \varnothing)$ mit $even(T) = \{v_2\}$ und $odd(T) = \varnothing$ ist in Abbildung 4.2 zu sehen.

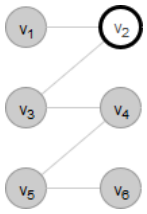


Abbildung 4.2: Alternierender Baum T mit $s = v_2$

Nun muss eine Kante $\{u, v\} \in E$ gesucht werden, wobei u in den geraden Knoten des Baumes liegen muss und v nicht in den Knoten des Baumes sein darf. Zur Verfügung stehen in diesem Beispiel die Kanten $\{v_2, v_1\}$ und $\{v_2, v_3\}$. Sei nun $\{v_2, v_3\}$ die gewählte Kante. Damit ist $v = v_3$ und muss untersucht werden. Dieser Knoten ist M-frei, somit kann das Matching erweitert werden. Der gefundene augmentierende Weg vom gewählten Knoten v zur Wurzel s besteht nur aus v_3, $\{v_2, v_3\}$ und v_2. Somit ist das erweiterte Matching $\big\{\{v_2, v_3\}\big\}$. Dies ist auch in Abbildung 4.3 zu sehen.

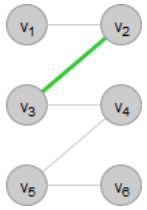


Abbildung 4.3: Matching M wurde um $\{v_2, v_3\}$ erweitert

Der Algorithmus kehrt nun zur Überprüfung, ob das Matching perfekt ist, zurück. Da M noch nicht alle Knoten in G überdeckt und es damit nicht perfekt ist, wird der alternierende Baum T neu initialisiert. Diesmal wird aus den M-freien Knoten der Knoten v_4 für die Wurzel s gewählt. Für T gilt nun:

$$T = (\{v_4\}, \varnothing), even(T) = \{v_4\}, odd(T) = \varnothing$$

Dies ist auch in Abbildung 4.4 zu sehen.

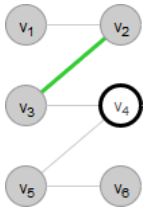


Abbildung 4.4: Der alternierende Baum T wurde neu initialisiert mit $s = v_4$

Nun muss wieder nach einer Kante $\{u, v\}$ mit den obigen Eigenschaften geguckt werden. Die Kanten $\{v_3, v_4\}$ und $\{v_4, v_5\}$ erfüllen diesmal die Bedingungen, gewählt wird für dieses Beispiel die Kante $\{v_4, v_5\}$. Damit ist $v = v_5$. Dieser Knoten ist M-frei, also kann das Matching M wieder erweitert werden. Das erweiterte Matching $M = \big\{\{v_2, v_3\}, \{v_4, v_5\}\big\}$ ist in Abbildung 4.5 zu sehen.

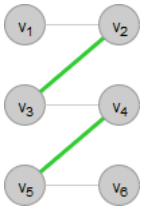


Abbildung 4.5: Matching M wurde um $\{v_4, v_5\}$ erweitert

Wieder springt der Algorithmus jetzt zu Schritt 2 zurück und testet, ob das Matching perfekt ist. Mit v_1 und v_6 gibt es aber noch zwei M-freie Knoten, also wird der Algorithmus fortgesetzt.

Sei v_1 der nächste M-freie Knoten, der für die Wurzel s des neu zu erstellenden Baumes T genommen wird. Der so entstandene Baum ist in Abbildung 4.6 zu sehen.

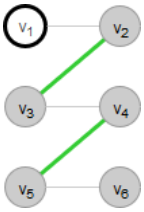


Abbildung 4.6: Der alternierende Baum T wurde neu initialisiert mit $s = v_1$

Nun ist die einzige Kante, die für die Kante $\{u, v\}$ in Frage kommt, $\{v_1, v_2\}$ und somit gilt $v = v_2$. Der Knoten v_2 ist nicht M-frei, sondern wird von $\{v_2, v_3\} \in M$ überdeckt. Der Baum T wird daher nun vergrößert um die Knoten v_2 und v_3 und die Kanten $\{v_1, v_2\}$ und $\{v_2, v_3\}$. Dies ist auch in Abbildung 4.7 zu sehen.

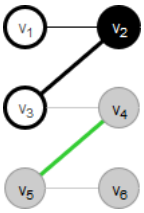


Abbildung 4.7: Baum T ist um die Knoten v_2 und v_3 und die Kanten $\{v_1, v_2\}$ und $\{v_2, v_3\}$ gewachsen

Für den Baum T gilt jetzt:

$$T = (\{v_1, v_2, v_3\}, \big\{\{v_1, v_2\}, \{v_2, v_3\}\big\}), even(T) = \{v_1, v_3\}, odd(T) = \{v_2\}$$

Der Algorithmus geht zu dem Schritt **Suche eine Kante, mit der der Baum wachsen oder das Matching erweitert werden kann** zurück und sucht nun erneut eine passende Kante $\{u, v\}$. Diesmal kann nur $\{v_3, v_4\}$ gewählt werden. Damit ist $v = v_4$, welcher wie zuvor nicht M-frei ist, sondern von einer Kante des Matchings überdeckt wird. Dies ist die Kante $\{v_4, v_5\}$. Somit kann T wachsen und sieht nach diesem Schritt wie folgend aus (wie auch in Abbildung 4.8 zu sehen ist):

$$\begin{aligned} T &= (\{v_1, v_2, v_3, v_4, v_5\}, \big\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}\big\}), \\ even(T) &= \{v_1, v_3, v_5\}, \\ odd(T) &= \{v_2, v_4\} \end{aligned}$$

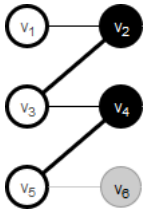


Abbildung 4.8: Baum T ist um die Knoten v_4 und v_5 und die Kanten $\{v_3, v_4\}$ und $\{v_4, v_5\}$ gewachsen

Nach dem Wachsen des Baumes geht der Algorithmus wieder zur Suche nach einer passenden Kante $\{u, v\}$ zurück. Nur $\{v_5, v_6\}$ erfüllt die Bedingung, dass $u = v_5$ in den geraden Knoten des Baumes liegt und $v = v_6$ nicht im Baum liegt. v_6 ist M-frei und erfüllt somit die Bedingung des ersten Falls. Jetzt kann M längs des gefundene augmentierenden Weges $P = v_6\{v_5, v_6\}v_5\{v_4, v_5\}v_4\{v_3, v_4\}v_3\{v_2, v_3\}v_2\{v_1, v_2\}v_1$ vergrößert werden. Sei M' das alte Matching. Für das erweiterte Matching M gilt dann, wie auch in Kapitel 3.3.2 beschrieben:

$$M = \Big(M' \cup E(P)\Big) \setminus \Big(M' \cap E(P)\Big)$$

wobei $E(P)$ die Menge der Kanten des Weges P ist.

Mit $M' = \Big\{\{v_2, v_3\}, \{v_4, v_5\}\Big\}$ und $E(P) = \Big\{\{v_5, v_6\}, \{v_4, v_5\}, \{v_3, v_4\}, \{v_2, v_3\}, \{v_1, v_2\}\Big\}$ gilt dann:

$$M = \Big\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5, v_6\}\Big\}$$

Dies ist in Abbildung 4.9 dargestellt.

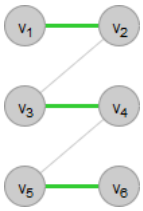


Abbildung 4.9: Das erweiterte und nun perfekte Matching M

Nun springt der Algorithmus zur Überprüfung, ob das Matching perfekt ist. Da alle Knoten von Kanten aus M überdeckt werden, ist dies der Fall, das Matching M wird ausgegeben und der Algorithmus terminiert.

Das gefundene Matching entspricht dem erwarteten Ergebnis, welches am Anfang des Beispieles beschrieben wurde.

4.2 Algorithmus für perfekte Matchings in allgemeinen Graphen

Des Weiteren sollen auch in allgemeinen Graphen perfekte Matchings gefunden werden. Ein allgemeiner Graph kann im Gegensatz zu bipartiten Graphen Kreise ungerader Länge enthalten. Dies führt zu einem weiteren Fall in der Fallunterscheidung und dazu, dass auch der Eingabegraph zwischendurch geändert werden muss. Krumke und Noltemeier schlagen hier den Algorithmus *Perfect-Matching* vor. Wie auch schon in Kapitel 4.1 wird

nun das Vorgehen des Algorithmus erklärt, wobei besonders auf die Unterschiede zu dem Algorithmus *Bipartite-Perfect-Matching* eingegangen wird. Danach folgt wiederum der Pseudo-Code des Algorithmus und ein Beispiel.

Ein- und Ausgabe Die Eingabe dieses Algorithmus besteht aus einem einfachen, zusammenhängendem Graphen $G = (V, E)$ und einem Matching M in G. Innerhalb dieser Arbeit wird immer davon ausgegangen, dass der Algorithmus mit einem leeren Matching aufgerufen wird, da dies nichts am eigentlichen Algorithmus ändert und somit Beispiele und die spätere Implementierung leichter fallen.

Die Ausgabe entspricht der Ausgabe des Algorithmus in bipartiten Graphen: Wird ein perfektes Matching gefunden, wird dieses ausgegeben, andernfalls erfolgt die Information, dass der Graph kein perfektes Matching enthält.

Variableninitialisierung Zunächst werden zwei Variablen G' und M' initialisiert, in welchen der aktuelle Graph und das aktuelle Matching gespeichert werden. Dabei wird die Variable G' mit dem Eingabegraphen G belegt und das Matching M' mit dem Eingabematching M.

Prüfe, ob das aktuelle Matching perfekt ist Nun folgt wie auch im Algorithmus für bipartite Graphen die Überprüfung, ob M' ein perfektes Matching in G' ist. Ist dies der Fall, terminiert der Algorithmus und gibt das Matching M' aus, andernfalls läuft er weiter.

Initialisierung des alternierenden Baumes Auch die Initialisierung des alternierenden Baumes T erfolgt wie im Algorithmus für bipartite Graphen. Ein Knoten s, welcher M'-frei ist, wird gewählt. Es gilt dann:

$$T = (\{s\}, \varnothing), even(T) = \{s\}, odd(T) = \varnothing$$

Suche eine Kante, mit der der Baum wachsen oder das Matching erweitert werden kann Nun folgt die Überprüfung, ob eine Kante existiert, mit welcher entweder der Baum wachsen oder das Matching erweitert werden kann. Dazu wird wie schon im Algorithmus für bipartite Graphen eine Kante $\{u, v\} \in E$ gesucht, allerdings darf in diesem Algorithmus der Knoten v innerhalb der geraden Knoten des Baumes liegen.

Für die Kante $\{u, v\}$ muss also gelten:

$$\{u, v\} \in E \text{ mit } u \in even(T) \text{ und } v \notin odd(T)$$

Wird keine solche Kante gefunden, ist der Baum verkümmert. Somit kann weder das Matching erweitert werden, noch der Baum wachsen. Der Algorithmus gibt in diesem Fall die Nachricht „G besitzt kein perfektes Matching“ aus und terminiert.

Wird dagegen eine solche Kante gefunden, können drei Fälle auftreten.

Fall 1: v **ist** M'**-frei und** $v \notin V(T)$ In diesem Fall kann das Matching erweitert werden. Dies ist einfach, wenn noch kein Kreis gefunden wurde. In diesem Fall wird nämlich wie im bipartiten Algorithmus nur das Matching längs des alternierenden Weges P erweitert und der Baum auf einen leeren Graphen zurückgesetzt. Der kompliziertere Fall, in dem der Algorithmus vorher einen Kreis ungerader Länge gefunden hat, wird in Abschnitt **Kreis extrahieren** beschrieben.

Der Algorithmus wird anschließend ab dem Schritt **Prüfe, ob das aktuelle Matching perfekt ist** fortgesetzt.

Fall 2: v **ist** M'**-überdeckt und** $v \notin V(T)$ Wenn v von einer Kante $\{v, w\} \in M'$ mit $w \notin V(T)$ überdeckt wird, kann der Baum wachsen. Dies geschieht wie im bipartiten Fall. Sei $T' = (V_{T'}, E_{T'})$ der Baum vor dem Wachsen. Für den gewachsenen Baum $T = (V_T, E_T)$ gilt nun also:

$$\begin{aligned} T = (V_T, E_T) \text{ mit } V_T &= V_{T'} \cup \{v, w\} \\ \text{und } E_T &= E_{T'} \cup \Big\{\{u, v\}, \{v, w\}\Big\} \\ even(T) &= even(T') \cup \{w\} \\ odd(T) &= odd(T') \cup \{v\} \end{aligned}$$

Anschließend springt der Algorithmus zu dem Schritt **Suche eine Kante, mit der der Baum wachsen oder das Matching erweitert werden kann**.

Fall 3: $v \in even(T)$ Wenn v in den geraden Knoten des Baumes liegt, enthält G den Kreis ungerader Länge $P\{u, v\}v$, wobei P der Weg von v nach u in T ist. Der Algorithmus kontrahiert nun den gefundenen Kreis zu einem Superknoten und aktualisiert das Matching M' und den Baum T. Dieser Vorgang wird mit *Shrink-and-Update* bezeichnet.

Dass der Kreis in diesem Fall eine ungerade Länge hat, kann man leicht sehen: da u und v in $even(T)$ liegen, haben sie beide nach Definition eines alternierenden Baumes immer zwei Kanten zwischen sich und dem nächsten geraden Knoten (Knoten in $odd(T)$ haben ja nur einen Nachfolger). Dadurch ist die Länge des Weges von u nach v auf jeden Fall gerade. Fügt man nun die Kante $\{u, v\}$ zu dem Weg von u nach v hinzu, erhält man einen Kreis ungerader Länge.

Shrink-and-Update Zunächst werden die Knoten und Kanten, die bei Hinzunahme von $\{u, v\}$ zu T einen Kreis ungerader Länge C in T bilden würden, zu einem Superknoten v_C zusammengefasst. Es ist klar, dass u und v zwei Knoten des Superknotens sind. Um die anderen Knoten und Kanten zu finden, muss man lediglich von den Knoten u und v den jeweiligen Weg zur Wurzel s finden und alle Knoten und Kanten, die auf nur einem der beiden Wege liegen, dem Superknoten hinzufügen. Der Knoten z, an dem die beiden Wege sich treffen, ist dann der letzte Knoten, der zum Superknoten hinzugefügt wird. Ist einer der beiden Knoten v oder u die Wurzel oder liegen beide Knoten auf einem Weg zur Wurzel ohne Abzweigung, so besteht der Kreis einfach aus dem Weg innerhalb des

Baumes von u zu v und der Kante $\{u, v\}$. In diesem Fall kann z gleich einem der Knoten u bzw. v gesetzt werden.

Alle Kanten, die vorher einen Knoten im Kreis mit einem Knoten außerhalb verbunden haben, verbinden jetzt den Knoten außerhalb mit dem Superknoten. Wie bereits erwähnt kann es an dieser Stelle zu Parallelkanten kommen, welche aber beim späteren Extrahieren des Superknotens wieder aufgelöst werden.

Der Graph G und der Baum T werden dann wie folgend angepasst: Alle Knoten und Kanten innerhalb des Superknotens v_C werden aus den jeweiligen Knoten- und Kantenmengen entfernt und der Knoten z wird in der jeweiligen Knotenmenge durch den Knoten v_C ersetzt. v_C gehört dann zu den geraden Knoten im Baum T, da auch z in $even(T)$ lag: entweder wurde z mit einem der Knoten u oder v gleichgesetzt, oder z ist der Knoten, welcher die Abzweigung darstellt, an der sich die Wege von u bzw. z zur Wurzel treffen. Da nur Knoten in der geraden Knotenmenge von T mehr als einen Nachfolger haben können und z in diesem Fall mindestens zwei hat, liegt z in $even(T)$.

Auch das Matching M' muss angepasst werden. Dazu werden aus dem Matching die Kanten des Kreises C entfernt.

Nun muss erneut geprüft werden, ob der Baum verkümmert ist oder der Algorithmus fortgesetzt werden kann. Dazu springt der Algorithmus wieder zu dem Schritt **Suche eine Kante, mit der der Baum wachsen oder das Matching erweitert werden kann** und fährt von dort aus fort.

Kreis extrahieren Ist Fall 1 eingetreten (v ist M'-frei und $v \notin V(T)$) und vorher wurde ein Kreis kontrahiert, muss dieser nun wieder extrahiert werden.

Zunächst wird jedoch das Matching längs des augmentierenden Weges erweitert, wie es auch ohne Superknoten geschehen würde.

Danach werden die Knoten und Kanten des Superknoten wieder dem Graphen hinzugefügt und der Superknoten selbst wird entfernt. Kanten, welche den Superknoten als einen ihrer Endknoten hatten, bekommen nun an dessen Stelle wieder ihren ursprünglichen Knoten. Der dadurch entstandene Graph wird G genannt.

Jetzt muss das Matching noch um die entsprechenden Kanten des Kreises erweitert werden. Dazu muss ein alternierender Weg im Kreis gefunden werden. Um dies zu tun, wird zunächst ein Knoten aus dem Kreis ausgewählt. War der Superknoten M'-frei, so kann der Knoten beliebig gewählt werden, andernfalls wird der Knoten genommen, der durch M' überdeckt wird. Entfernt man diesen Knoten und seine Kanten aus dem Kreis, entsteht ein Weg $P_C = v_1 e_1 v_2 \dots e_k v_k$ ungerader Länge, aus dem ein Matching M_C gebildet werden kann: angefangen mit der ersten Kante des Weges P_C wird jede zweite zu M_C hinzugefügt. Es gilt also:

$$M_C = \{e_i | e_i \in E(P_C) \text{ und } i \text{ ist ungerade}\}$$

Für das neue Matching M gilt dann:

$$M = M' \cup M_C$$

Im letzten Schritt von **Kreis extrahieren** werden die vom Algorithmus gespeicherten Variablen G' und M' auf den entstandenen Graphen G und das neue Matching M gesetzt.

Algorithmus in Pseudo-Code Wie auch schon in Kapitel 4.1 folgt hier nun der von Krumke und Noltemeier bereitgestellte Algorithmus in leicht abgeändertem Pseudo-Code. Anhand dieses Pseudo-Codes erfolgte auch die spätere Implementierung.

```
SHRINK-AND-UPDATE(M', T, {u,v})
1  Sei C der einfache Kreis ungerader Länge,
   den die Hinzunahme von {u, v} zu T erzeugt
2  Setze G' := G'/C und M' := M' \ E(C)
3  Ersetze T durch den Baum im aktualisierten Graphen G'
   mit Kantenmenge E(T) \ E(C)
```

```
PERFECT-MATCHING(G, M)
1  Setze G' := G
2  Setze M' := M
3  if M' ist ein perfektes Matching in G' then
4       return M'
5  Wähle einen M'-freien Knoten s und setzte T := ({s}, ∅)
6  if es gibt eine Kante {u, v} ∈ E(G')
   mit u ∈ even(T) und v ∉ odd(T) then
7       if v ist M'-frei und v ∉ V(T) then
8            vergrößere das Matching M'
             längs des gefundenen augmentierenden Weges.
9            if G' enthält einen Superknoten then
                  Setze G := G'.
                  Extrahiere für jeden Superknoten v_C in G den im
                  Superknoten enthaltenen Kreis C.
                  Erstelle ein Matching M bestehend aus M' und dem
                  Matching, welches in C gefunden werden kann.
                  Setze G' := G, M' := M.
10           goto 3
11      if v ist von {v, w} ∈ M' überdeckt mit v, w ∉ V(T) then
12           Füge v, w und die Kanten {u, v}, {v, w} zu T hinzu.
13           goto 6
14      if v ∈ even(T) then
15           Kontrahiere den gefundenen einfachen Kreis C
             ungerader Länge und aktualisiere M' und T
             mittels SHRINK-AND-UPDATE(M', T, {u,v}), goto 6
16 T ist ein verkümmerter Baum in G'.
17 return "G besitzt kein perfektes Matching"
```

Beispiel Um den Algorithmus besser zu verstehen, ist es praktisch, sich dessen Durchlauf an einem Beispiel anzusehen.

Sei der Graph aus Abbildung 4.10 ein Graph G, in dem wir ein perfektes Matching suchen. Man sieht hier leicht, dass es nur ein perfektes Matching geben kann, nämlich das Matching $\big\{\{v_1, v_4\}, \{v_2, v_3\}\big\}$, da die Kante $\{v_1, v_4\}$ die einzige ist, die den Knoten v_4

überdeckt.

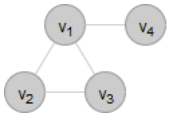


Abbildung 4.10: Beispielgraph G

Der Algorithmus wird mit dem Graphen G und einem leeren Matching aufgerufen. Im Algorithmus werden zunächst G' und M' initialisiert:

$$G' := G \text{ mit } G = (\{v_1, v_2, v_3, v_4\}, \big\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{v_1, v_4\}\big\})$$
$$M' := \varnothing$$

Die darauf folgende Überprüfung, ob das Matching M' perfekt ist, schlägt selbstverständlich fehl, da das leere Matching gar keine Knoten überdeckt.

Nun muss ein M'-freier Knoten gewählt werden. Da jeder Knoten $v \in V(G)$ M'-frei ist, ist es egal, welcher ausgesucht wird. Für dieses Beispiel wird der Knoten v_1 gewählt. Mit diesem Knoten wird ein alternierender Baum $T := (\{v_1\}, \varnothing)$ initialisiert. Dies ist auch in Abbildung 4.11 zu sehen.

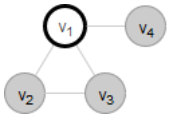


Abbildung 4.11: Baum $T := (\{v_1\}, \varnothing)$ wurde initialisiert

Als nächstes muss eine Kante $\{u, v\}$ gesucht werden, bei der der eine Knoten in den geraden Knoten und der andere nicht in den ungeraden Knoten des Baumes T liegt. Da $even(T)$ momentan nur aus v_1 besteht und $odd(T)$ noch leer ist, kommen hier die Kanten $\{v_1, v_2\}$, $\{v_1, v_3\}$ und $\{v_1, v_4\}$ in Frage. Es wird zum Beispiel die Kante $\{v_1, v_2\}$ gewählt (siehe auch Abbildung 4.12).

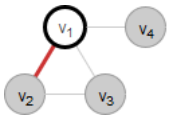


Abbildung 4.12: Für $\{u, v\}$ wurde die Kante $\{v_1, v_2\}$ gewählt

Nun muss der Knoten v aus der gewählten Kante $\{u, v\}$ betrachtet werden. Dies ist momentan v_2. v_2 ist M'-frei, daher kann das Matching M' längs des gefundenen augmentierenden Weges (dies ist hier v_1 $\{v_1, v_2\}$ v_2) vergrößert werden:

$$M' = \big\{\{v_1, v_2\}\big\}$$

Das vergrößerte Matching ist in Abbildung 4.13 zu erkennen.

Bisher wurde kein Kreis kontrahiert. Aus diesem Grund können die Schritte 9 und 10 übersprungen werden und der Algorithmus kann direkt zu Schritt 3 wechseln. Es wird nun also erneut überprüft, ob das Matching M' perfekt ist. Es werden aber immer noch nicht

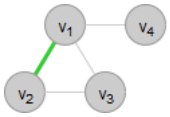


Abbildung 4.13: Matching $M' = \{\{v_1, v_2\}\}$

alle Knoten von Kanten aus M' überdeckt, deswegen muss der Algorithmus fortgesetzt werden.

Erneut muss ein M'-freier Knoten gewählt werden. Zur Verfügung stehen v_3 und v_4. In diesem Beispiel wird als nächstes v_3 gewählt und damit der Baum T neu initialisiert, wie in Abbildung 4.14 zu sehen ist:

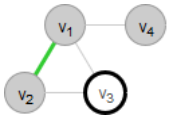


Abbildung 4.14: Baum $T := (\{v_3\}, \varnothing)$ wurde initialisiert

Im nächsten Schritt muss wieder die Kante $\{u, v\}$ gewählt werden. Dieses Mal erfüllen die Kanten $\{v_3, v_1\}$ und $\{v_3, v_2\}$ die Bedingung, da $even(T) = \{v_3\}$ und $odd(T)$ leer ist.

Gewählt wird in diesem Beispiel nun die Kante $\{v_3, v_1\}$ mit $v = v_1$. Jetzt muss also der Knoten v_1 genauer betrachtet werden. v_1 ist nicht M'-frei, sondern wird von der Kante $\{v_1, v_2\} \in M'$ überdeckt. Desshalb wird als nächstes der Baum erweitert, indem die Knoten v_1 und v_2 und die entsprechenden Kanten $\{v_3, v_1\}$ und $\{v_1, v_2\}$ zum Baum hinzugefügt werden. Jetzt gilt:

$$even(T) = \{v_3, v_2\}$$
$$odd(T) = \{v_1\}$$

Der aktuelle Zustand des Algorithmus ist auch in Abbildung 4.15 zu sehen.

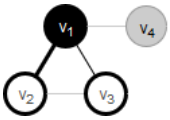


Abbildung 4.15: T wurde erweitert

Der Algorithmus springt nun wieder zur Suche nach einer passenden Kante $\{u, v\}$. Die einzige Möglichkeit für $\{u, v\}$ ist diesmal die Kante $\{v_3, v_2\}$. Da sowohl v_3 als auch v_2 in den geraden Knoten von T liegen, ist es nicht wichtig, welcher Knoten als u und welcher als v gewählt wird. In diesem Beispiel sei $u = v_3$ und $v = v_2$. v_2 liegt aber bereits im Baum, daher trifft sowohl der 1. als auch der 2. Fall nicht ein. Dafür wird in Schritt 14 festgestellt, dass v_2 in den geraden Knoten des Baumes liegt. Die Zunahme der Kante $\{v_3, v_2\}$ zum Baum würde also zu einem Kreis führen.

Der Algorithmus ruft nun *Shrink-and-Update* mit den Argumenten $M' = \big\{\{v_1, v_2\}\big\}$, $T = (\{v_3, v_1, v_2\}, \big\{\{v_3, v_1\}, \{v_1, v_2\}\big\})$ und $\{u, v\} = \{v_3, v_2\}$ auf.

Hier wird zunächst der Kreis C initialisiert. Dieser besteht in diesem Fall aus den Knoten v_1, v_2 und v_3 und den Kanten $\{v_1, v_2\}$, $\{v_2, v3\}$ und $\{v_3, v_1\}$.

Dann wird der Kreis kontrahiert: alle Knoten und Kanten von C werden aus dem Graph entfernt und durch einen neuen Superknoten v_C ersetzt. Außerdem werden aus dem Matching alle Kanten des Kreises C entfernt. Da M' momentan nur aus der Kante $\{v_1, v_2\}$ besteht, welche in C liegt, ist das Matching anschließend leer.

Im letzten Schritt von *Shrink-and-Update* wird der Baum angepasst. Dieser besteht nun nur noch aus dem Superknoten v_C.

Der Zustand des Algorithmus nach *Shrink-and-Update* ist in Abbildung 4.16 zu sehen.

Abbildung 4.16: Graph G' nach *Shrink-and-Update*

Erneut wird in Schritt 6 nach einer passenden Kante $\{u, v\}$ gesucht. Dieses Mal kommt nur $\{v_C, v_4\}$ in Frage, v ist also v_4.

v_4 ist M'-frei, also kann das Matching erweitert werden. Der gefundene augmentierende Weg ist nun v_C $\{v_C, v_4\}$ v_4. Das mit diesem Weg aktualisierte Matching M' ist auch in Abbildung 4.17 zu sehen.

Abbildung 4.17: Matching $M' = \{\{v_c, v_4\}\}$

In Schritt 9 wird der vorher kontrahierte Kreis nun wieder extrahiert und das Matching wird angepasst. Die Einzelschritte der Extraktion des Kreises sind auch in Abbildung 4.18 zu sehen. Da v_C von $\{v_C, v_4\} \in M'$ überdeckt wird, ist auch der Knoten v_1 M' durch überdeckt, da er vorher eine Kante zu v_4 hatte. Entfernt man nun v_1 und die dazugehörigen Kanten $\{v_1, v_2\}$ und $\{v_1, v_3\}$ aus dem Kreis C, bleibt ein Weg übrig, welcher in diesem Fall nur aus v_2 $\{v_2, v_3\}$ v_3 besteht. Mit diesem Weg wird nun das Matching erweitert.

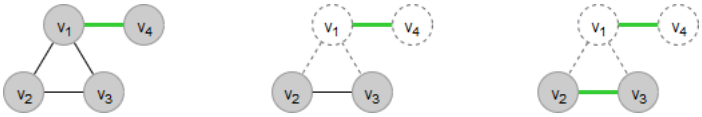


Abbildung 4.18: Schritte der Kreisextraktion

G' besteht nun also wieder aus den Knoten v_1, v_2, v_3 und v_4 und den Kanten $\{v_1, v_2\}$, $\{v_2, v_3\}$, $\{v_1, v_3\}$ und $\{v_1, v_4\}$. Das Matching M' besteht nun aus $\{v_1, v_4\}$ und $\{v_2, v_3\}$. Dies ist auch in Abbildung 4.19 zu erkennen.

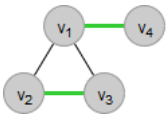


Abbildung 4.19: Graph G mit extrahiertem Kreis und perfektem Matching $M' = \{\{v_1, v_4\}, \{v_2, v_3\}\}$

Der Algorithmus springt daraufhin ein letztes Mal zur Überprüfung, ob das Matching M' perfekt ist. Da alle Knoten in G von Kanten aus M' überdeckt sind, ist dies der Fall, der Algorithmus gibt M' aus und terminiert.

Das vom Algorithmus gefundene Matching stimmt mir den Überlegungen vom Anfang des Beispieles überein.

5 Implementierung

In diesem Kapitel wird die Implementierung erläutert. Dabei wird zunächst die verwendete Technologie erklärt, anschließend folgt die Beschreibung der Architektur. Auf eine Beschreibung des gesamten Source-Codes wird verzichtet, da die Implementierung nicht Hauptfokus dieser Arbeit ist. Stattdessen wird in der Ausführungssicht beschrieben, wie die einzelnen Komponenten der Implementierung zusammenarbeiten und es wird ein Überblick über die Implementierung des eigentlichen Algorithmus gegeben. Im nächsten Teil dieses Kapitels wird auf alle erfolgten Tests eingegangen und schlussendlich wird die Verwendung der Applikation inklusive der Visualisierung vorgestellt .

Hauptziel der Implementierung war es, eine Applikation zu erstellen, in der der Algorithmus für einfache, zusammenhängende und allgemeine Graphen Schritt für Schritt durchgegangen werden kann und die den jeweiligen Zustand des Graphen inklusive aller dabei entstehenden alternierenden Bäume, Matchings und Superknoten visualisiert. Um Graphen für den Algorithmus eingeben zu können, wurde ein simples Graph-Erstellungs-Feature entwickelt, indem die Graphen interaktiv erstellt, bearbeitet, gespeichert und geladen werden können. Eine weitere Möglichkeit der Grapheneingabe ist die manuelle Erstellung von sogenannten .json-Dateien und das anschließende Laden innerhalb der Applikation.

5.1 Verwendete Technologien und Formate

Das Backend, welches unter anderem auch die Implementierung des Algorithmus beinhaltet, wurde in Python 3.6 implementiert. Python ist eine interpretierte Skriptsprache, in welcher einfach lesbarer Code verfasst werden kann. Auch kann sie für eine objektorientierte Implementierung genutzt werden, ist jedoch nicht darauf beschränkt. Für die Verwaltung von Pythonpackages wie z.B. dem verwendeten Package „websockets“ wurde das Tool pip genutzt.

Für das Frontend wurde zunächst ein Prototyp in PyQt5 entwickelt, dieser wurde jedoch verworfen, da die Technologie heutzutage zu veraltet ist. Stattdessen wurde ein Web-Frontend entwickelt, welches das Javascript-Framework React in der Version 16.5.2 nutzt. Zum ersten Erstellen des Frontends wurde das Tool Create React App genutzt, da so eine erste Applikation ohne weitere Konfiguration erstellt werden konnte. Dies hat den Umstieg auf React noch einmal erheblich vereinfacht. Um die Graphen zu zeichnen wurde zusätzlich die Bibliothek Konva (Version 2.4) zusammen mit react-konva (Version 1.7.15) genutzt. Alle weiteren genutzten Javascript-Bibliotheken können der auf der CD befindlichen `frontend\package.json` Datei entnommen werden.

Außerdem wurde CSS verwendet, um die Oberfläche der Applikation ansprechend zu gestalten sowie npm, um Javascript Bibliotheken zu verwalten und den Frontendcode zu

bauen.

Die gespeicherten Graphen werden in .json-Dateien abgelegt. JSON steht für „JavaScript Object Notation“ und bietet eine gute Möglichkeit, um Daten in lesbarer Form zu speichern und zu laden.

Für den Implementierungsprozess wurde außerdem das Versionskontrollsystem GIT eingesetzt.

Zum Erstellen der ausführbaren Programme auf Windows und Linux wurde das Tool pyinstaller genutzt.

5.2 Architektur

Die Architektur dieser Implementierung orientiert sich an dem Konzept „Model-View-Controller“ [Ree79]. Dieses Konzept sieht vor, dass Module in drei Schichten eingeordnet werden:

- Die Schicht *Model*, welche die einfachen Datenklassen beinhaltet, die in der Oberfläche angezeigt werden,
- die Schicht *View*, in welcher aller Code für die Benutzeroberfläche liegt und
- die Schicht *Controller*, in welcher die eigentliche Programmlogik implementiert ist.

Durch dieses Prinzip können die einzelnen Schichten ausgetauscht werden, ohne die anderen groß anpassen zu müssen. Nur die jeweiligen Schnittstellen müssen eventuell angepasst werden. Dadurch war der Umstieg von PyQt5 auf Javascript simpel.

Manche Module lassen sich allerdings nicht eindeutig einer dieser Schichten zuordnen. Diese wurden in einer zusätzlichen Schicht *Utils* gesammelt.

Die Umsetzung dieses Konzeptes ist in Abbildung 5.1 zu sehen.

Im Backend Server liegen die Schichten *Model*, *Controller* und *Utils*, während die Schicht *View* einen Frontend Server bereitstellt, welcher Anzeigedaten für den Browser Client generiert. Für die Kommunikation zwischen Front- und Backend ist im *Controller* eine Schnittstelle implementiert, welche die Nutzereingaben empfängt und auswertet, während die Schicht *View* eine Schnittstelle für alle Graphdaten bereitstellt.

5.2.1 Model

In Abbildung 5.2 sind die Klassen innerhalb der Schicht *Model* in einem Klassendiagramm dargestellt. Die Assoziationen werden hier als unidirektionale bzw. bidirektionale Assozationen angegeben, da sowohl die Klasse **`Graph`** als auch die Klasse **`AlternatingTree`** ihre assozierten Klassen **`Vertex`**, **`SuperVertex`** und **`Edge`** kennen, andersherum **`Vertex`**, **`SuperVertex`** und **`Edge`** aber keinen Zugriff auf die Klassen **`Graph`** und **`AlternatingTree`** haben.

Knoten sind in der Klasse **`Vertex`** abgebildet. Sie haben eine eindeutige Id, x- und y-Koordinaten, einen Text, welcher in der Oberfläche angezeigt wird und eine Liste an

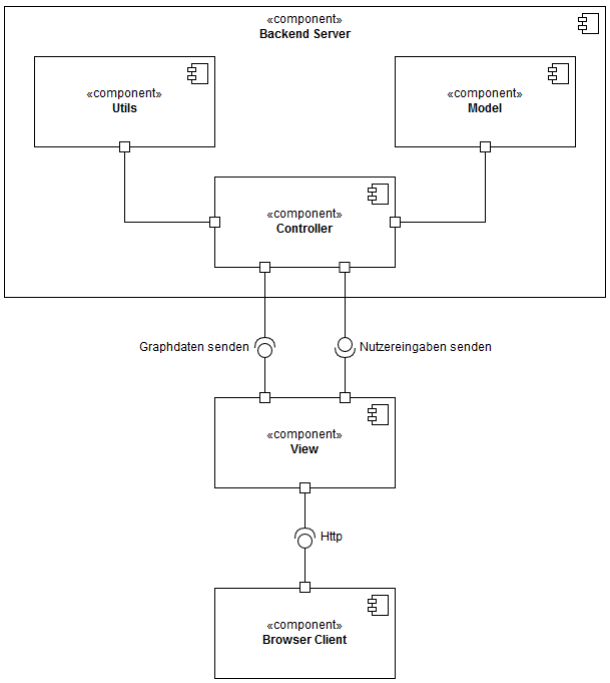


Abbildung 5.1: Komponentendiagramm

Kantenobjekten. Außerdem haben sie ein Attribut **`mark`**, mit dem sie als „ungerade“, „gerade“ oder „verbunden“ markiert werden können und ein Attribut **`distanceToTreeRoot`**, in welchem die Distanz zur Wurzel eines Baumes gespeichert werden kann. Initial besteht der Text eines Konten aus dem Buchstaben „v“ und der Id, die Markierung ist leer und die Distanz liegt bei -1. Die Klasse bietet neben dem Konstruktor **`__init__()`** und einer Funktion zur Textrepräsentation **`__str__()`** die Möglichkeit, den Text und die Koordinaten neu zu setzen und Kanten hinzuzufügen bzw. zu entfernen sowie eine Funktion, die zu einem anderen Knoten eine Kante sucht, falls zwischen diesem und dem anderen Knoten eine Kante existiert. Dem Konstruktor müssen die Koordinaten übergeben werden, optional kann auch direkt ein Text an den Konstruktor gegeben werden

Die Klasse **`SuperVertex`** erbt von der Klasse **`Vertex`** und bildet einen Superknoten ab. Sie überschreibt den Konstruktor **`__init__()`** ihrer Elternklasse. Dieser bekommt eine Liste an Knoten übergeben. Die Koordinaten eines Superknotens werden aus einem der übergebenen Knoten übernommen und mit diesen Koordinaten wird der Konstruktor der Elternklasse aufgerufen. Der Superknoten besitzt also immer die Koordinaten von einem in ihm gespeicherten Knoten. Zusätzlich wird das Textattribut überschrieben. Es wird auf den Buchstaben „C“ und die Id gesetzt. Kanten sind in dieser Klasse nicht zu finden, da die Knoten innerhalb einer Superkante ihre Kanten, welche nur innerhalb des kontrahierten Kreises liegen, noch kennen.

Kanten sind in der Klasse **`Edge`** zu finden. Wie auch Knoten haben sie eine eindeutige Id und einen in der Oberfläche anzeigbaren Text. Außerdem kennen sie ihre zwei Knoten und haben ein Attribut, in welchem gespeichert wird, ob die Kante in einem Baum liegt oder nicht. An Funktionen hat die Klasse **`Edge`** den Konstruktor **`__init__()`**, die Textrepräsentation **`__str__()`**, eine Funktion zum Setzen des Textattributes sowie eine Funktion, welche einen Knoten übergeben bekommt und, falls dieser Knoten zur Kante gehört, den anderen Knoten der Kante zurück gibt.

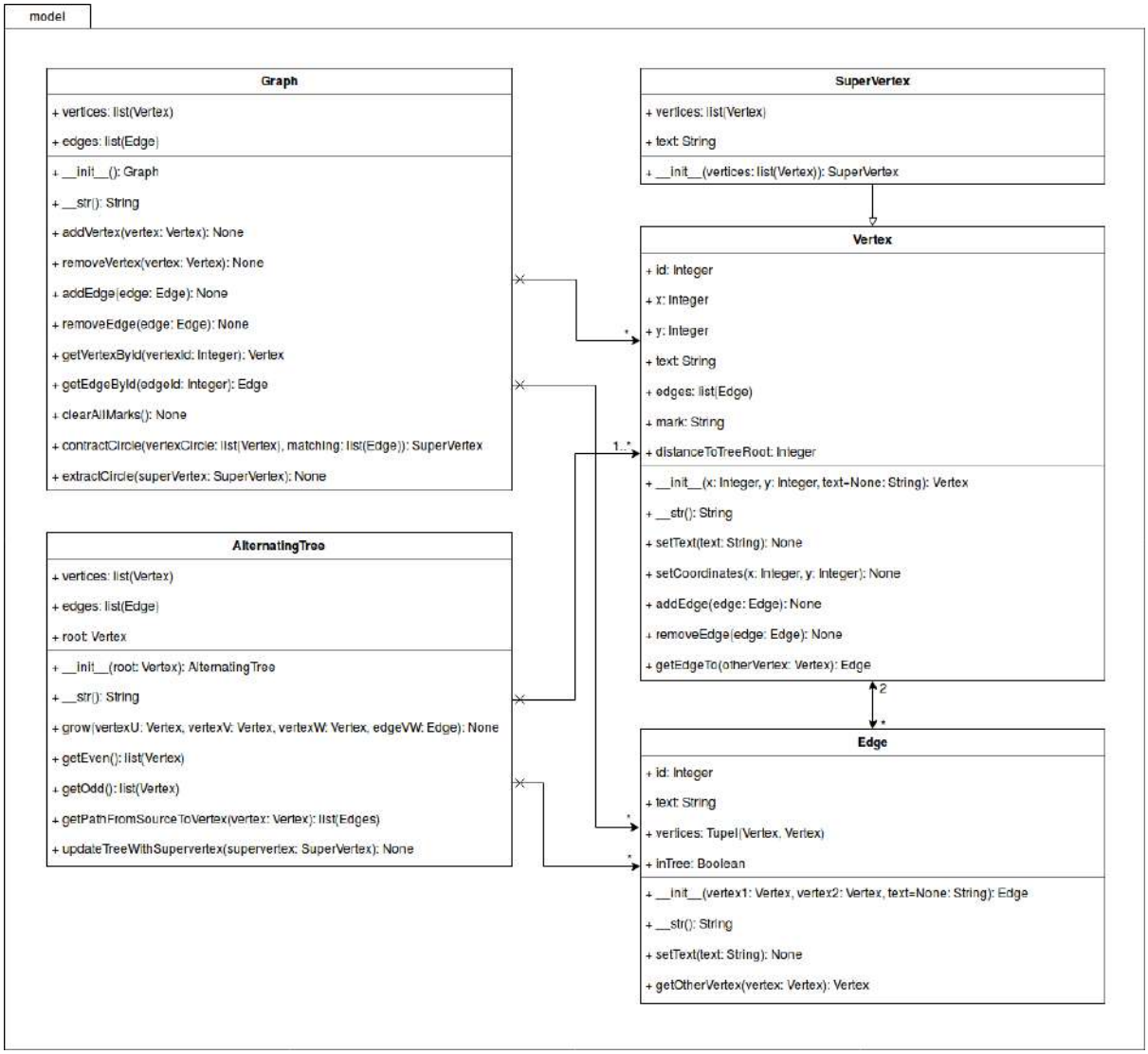


Abbildung 5.2: Klassendiagramm der Schicht *Model*

Ein Objekt der Klasse **`Graph`** hat je eine Liste von Objekten der Klassen **`Vertex`** und **`Edge`**. In den Funktionen der Klasse findet sich der Konstruktor **`__init__()`**, eine Funktion zur Textrepräsentation **`__str__()`** sowie Methoden, um Knoten und Kanten hinzuzufügen, entfernen oder im Graphen zu finden. Außerdem gibt es hier eine Funktion, um alle Markierungen, die während des Durchlaufs des Algorithmus bei Knoten und Kanten gesetzt werden, wieder auf ihren Initialwert zu setzen. Des Weiteren bietet der Graph die Möglichkeit, einen Kreis zu kontraktieren bzw. einen Superknoten zu extrahieren. Diese beiden Funktionen wurden vom Modul **`arbitraryalgorithm`** der Schicht *Controller* in die Klasse **`Graph`** verschoben, da so der Implementierungsaufwand geringer war. Semantisch gehören sie aber eigentlich zu den Funktionen des Algorithmus, welche im eben genannten Modul **`arbitraryalgorithm`** zu finden sind.

Die Klasse **`AlternatingTree`** besitzt mit je einer Liste von Knoten und Kanten einen ähnlichen Aufbau wie die Klasse **`Graph`**, erbt jedoch nicht von **`Graph`**, obwohl ein alternierender Baum selbstverständlich auf theoretischer Ebene ein Graph ist. Für die Implementierung ist es jedoch simpler, zwei eigenständige Klassen zu erstellen, da sie verschiedene Ansprüche haben. Einem Graphen müssen zum Beispiel Kanten und Knoten einfach hinzugefügt

werden können, während ein alternierender Baum immer nur um zwei Kanten und zwei Knoten wachsen kann (siehe Kapitel 3.3.1). Zusätzlich zu den Knoten- und Kantenlisten hat ein Objekt der Klasse **`AlternatingTree`** auch noch ein Attribut **`root`** vom Typ **`Vertex`**, welches die Wurzel des alternierenden Baumes darstellt. Dieses Attribut muss dem Konstruktor **`__init__()`** übergeben werden, weil ein alternierender Baum nicht ohne Wurzel bestehen kann. Außer dem Konstruktor besitzt auch diese Klasse eine Funktion zur Textrepräsentation **`__str__()`**. Des Weiteren gibt es eine Funktion zum Wachsen des Baumes. Außerdem können bei der Klasse alle geraden bzw. ungeraden Knoten angefordert werden und ein Weg zwischen der Wurzel und einem gegebenen Knoten berechnet werden. Zu beachten ist hierbei, dass der Einfachheit halber innerhalb der Implementierung Wege nur aus Kanten bestehen. Die letzte Funktion der Klasse **`AlternatingTree`** kann den Baum aktualisieren, wenn ein Superknoten kontrahiert wurde.

5.2.2 View

In dieser Schicht ist die Nutzeroberfläche und eine Schnittstelle zum Backend implementiert. Die genaue Beschreibung dieser Oberfläche ist in Kapitel 5.6.3 beschrieben.

Besonders erwähnt sei hier nur die Datei **`constants.js`**, da hier die wichtigsten Konstanten wie der Radius der Knoten und die Liniendicke festgelegt werden. Außerdem sind hier auch alle verwendeten Farben definiert.

5.2.3 Controller

Die Schicht *Controller* beinhaltet die Klasse **`controller`**, die Module **`arbitraryalgorithm`**, **`web`**, **`encodedecodegraph`** und **`history`** sowie das Package **`commands`**.

Die Klasse **`Controller`** bildet die Schnittstelle zwischen der Modulschicht und dem Modul **`web`**, welches wiederum die Schnittstelle zum Frontend darstellt.

Das Modul **`arbitraryalgorithm`** beinhaltet den Algorithmus und alle dafür benötigten Funktionen sowie die Klasse **`AlgorithmData`**, in welcher ein Graph, ein Matching und eine Kante gespeichert werden können. Die Kante stellt die im Algorithmus immer wieder gesuchte Kante $\{u, v\}$ dar. Die genauere Implementierung ist im Kapitel 5.4 beschrieben.

Das Modul **`encodedecodegraph`** bietet Funktionen, um Objekte vom Typ **`Graph`** und **`AlgorithmData`** in Strings im JSON-Format umzuwandeln, welche dann an das Frontend geschickt werden können oder in .json-Dateien gespeichert werden können. Außerdem können hier auch .json-Dateien eingelesen werden, um daraus einen Graphen zu laden.

Um bei der Erstellung eines Graphen im Frontend eine Rückgängig-Funktion anbieten zu können, wurde das Entwurfsmuster *Command* implementiert [Sie14]. Jede Aktion in der Oberfläche, die rückgänig gemacht werden kann, muss dazu als ein Kommando in einer **`Command`**-Klasse implementiert sein. Ein **`Command`** kann dabei ausgeführt werden oder rückgänig gemacht werden. Diese Kommandos werden in einer Klasse **`History`** gesammelt. Sie beinhaltet eine Liste aller bereits ausgeführten Kommandos. Wird in der Oberfläche die Aktion „Rückgänig“ ausgewählt, entfernt die Klasse **`History`** das neuese Element ihrer Kommandoliste und führt dessen **`undo()`** Funktion aus. Alle Kommandos, die abstrakte

Oberklasse **`Command`** und die Klasse **`History`** sind im Klassendiagramm in Abbildung 5.3 zu sehen.

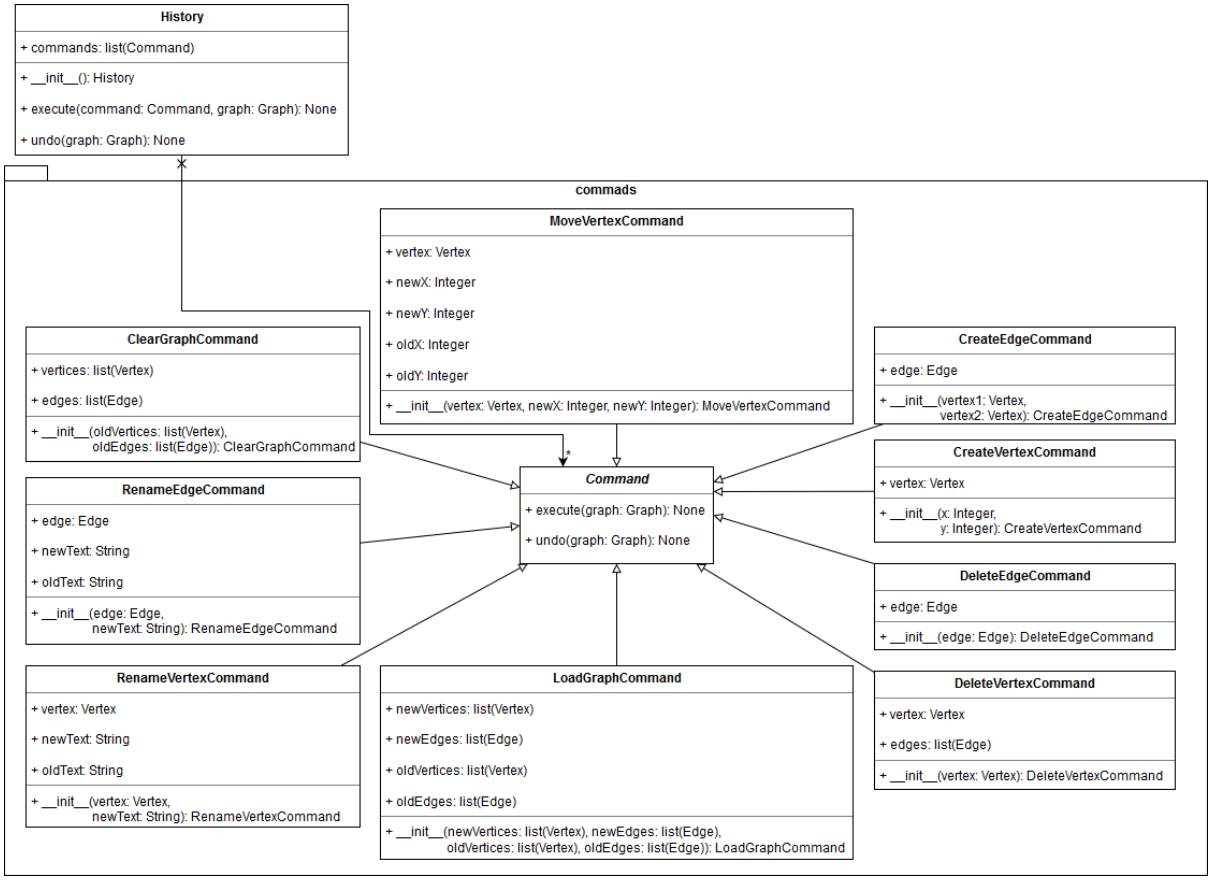


Abbildung 5.3: Klassendiagramm zum Entwurfsmuster *Command*

5.2.4 Utils

Innerhalb der Schicht *Utils* finden sich die Klasse **`Result`**, in welcher Fehlermeldungen gesammelt werden können, das Modul **`idgenerator`** mit Generatoren für Knoten- und Kantenids, ein Modul zum Sammeln von Konstanten, welche in der restlichen Implementierung benötigt werden, und das Modul **`fileutils`**. Letzteres beinhaltet zwei Methoden, um den Pfad zum Speicherordner zu finden und dort alle .json Dateien zu sammeln. Dieser Pfad liegt bei Windows unter `%appdata%\..\local\SAVEDATA_DIR` und bei Linux unter `~/.SAVEDATA_DIR`, wobei **`SAVEDATA_DIR`** standardmäßig gleich **`perfect_matchings`** ist.

5.3 Ausführungssicht

Die Sequenzdiagramme in Abbildung 5.4 und 5.5 zeigen, wie das Zusammenspiel von Front- und Backend Server während der Programmausführung funktioniert. Für diese Erklärung wurden die Abläufe zweier Usecases aufgezeichnet.

In Abbildung 5.4 ist ein Sequenzdiagramm zum Usecase „Laden eines Graphen“ zu sehen. Voraussetzung für diesen Usecase ist, dass bereits gespeicherte Graphen vorhanden

sind. Gestartet wird der Usecase, indem der Nutzer den Button „Laden“ betätigt. Der Browser sendet dann an den Frontend Server ein Klick-Event. Dieser wiederum sendet einen JSON-String an die Web-Schnittstelle **`web.py`** des Backend Servers. Im Diagramm ist dieser Vorgang verkürzt dargestellt, das Codieren bzw. Decodieren der gesendeten JSON-Nachrichten wurde weggelassen. Die Web-Schnittstelle holt sich nun die verfügbaren Dateien im Speicherordner und sendet diese als JSON-Nachricht zurück an den Frontend Server. Dieser rendert nun einen Ladedialog. Im Ladedialog wählt der Nutzer einen Dateinamen aus. Das daraus resultierende Klick-Event wird wieder vom Browser an den Frontend Server weitergegeben, welcher aus dem gewählten Dateinamen eine JSON-Nachricht generiert und diese an das Backend weiter gibt. Im Backend wird nun die Lade-Funktion des Controllers aufgerufen. Innerhalb dieser Funktion erstellt der Controller ein entsprechendes Kommando, fügt es der History hinzu und führt das Kommando schließlich aus. Dabei wird der aktuelle Graph des Controllers auf den neu gewählten Graphen gesetzt. Der Controller gibt der Web-Schnittstelle ein Objekt vom Typ **`Result`** zurück, in welchem etwaige Fehlermeldungen gespeichert sein können. Wenn kein Fehler aufgetreten ist, werden die neuen Graphendaten wieder zurück an den Frontendserver gereicht, wo diese anschließend angezeigt werden.

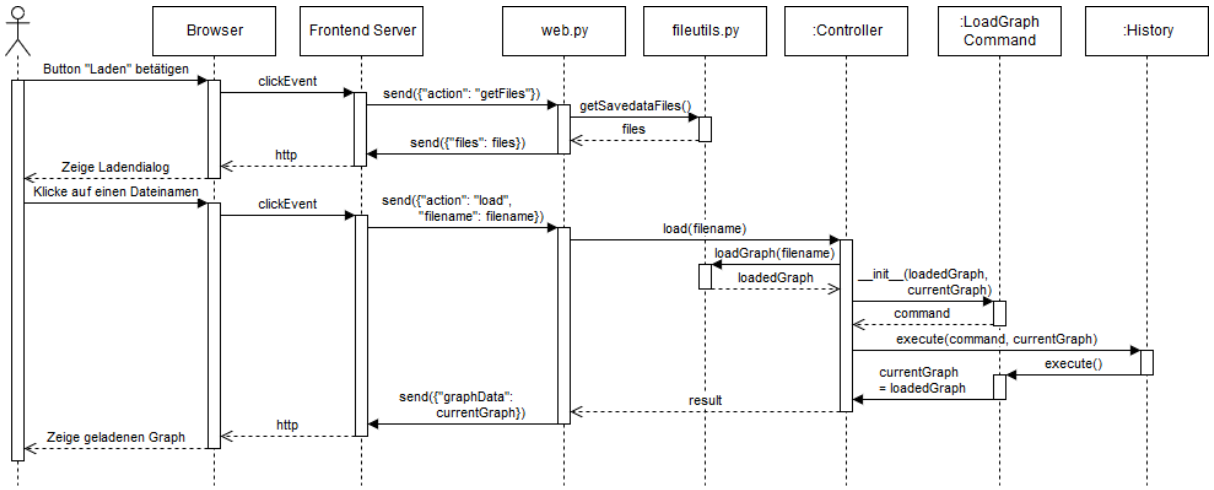


Abbildung 5.4: Sequenzdiagramm zum Usecase „Laden eines Graphen“

Der zweite Usecase, zu sehen in Abbildung 5.5, beschreibt das Starten und Ausführen des Algorithmus. Damit dieser Usecase wie im folgenden beschrieben ausgeführt werden kann, muss ein zusammenhängender Graph geladen bzw. erstellt worden sein oder es muss ein leerer Graph vorliegen. Der Nutzer startet den Usecase in der Graphbearbeitungssicht mit Klick auf den Button „Zum Algorithmus wechseln“. Durch das Klick-Event wird im Frontend Server das Senden einer Information an den Backend Server ausgelöst. Diese Information beinhaltet lediglich das Stichwort „algorithm_mode“ und drückt aus, dass nun der Algorithmus gestartet werden soll. Dies wird an den Controller weiter gegeben, welcher daraufhin zunächst ein Objekt der Klasse **`AlgorithmData`** mit dem aktuellen Graphen als Parameter erstellt. Danach wird der Algorithmus selbst initialisiert mit dem eben erstellten **`AlgorithmData`**-Objekt als Eingabe. Das **`AlgorithmData`**-Objekt und ein initialer Erklärungstext werden anschließend an das Frontend geleitet. Im Browser wird daraufhin zur Algorithmus-Sicht gewechselt.

Der Nutzer kann nun den Button „Nächster Schritt“ betätigen. Das Frontend teilt dem Backend mit, dass der nächste Schritt des Algorithmus ausgeführt werden soll. Dies

wird vom Controller erledigt. Die geänderten Werte im **`AlgorithmData`**-Objekt und Erklärungstext werden wieder ans Frontend zurückgeschickt und im Browser wird die Anzeige aktualisiert. Diesen Vorgang kann der Nutzer nun beliebig oft wiederholen. Ist der Algorithmus beendet, werden allerdings keine neuen Daten mehr angezeigt, sondern die letzte Ansicht auf den Algorithmus bleibt bestehen. In der aktuellen Version der Applikation bekommt der Nutzer außer dem angezeigten Erklärungstext kein Feedback, ob der Algorithmus fertig ist oder nicht. Dieser Punkt wurde in die Liste der Verbesserungen aufgenommen, welche in Kapitel 6.2 zu finden ist.

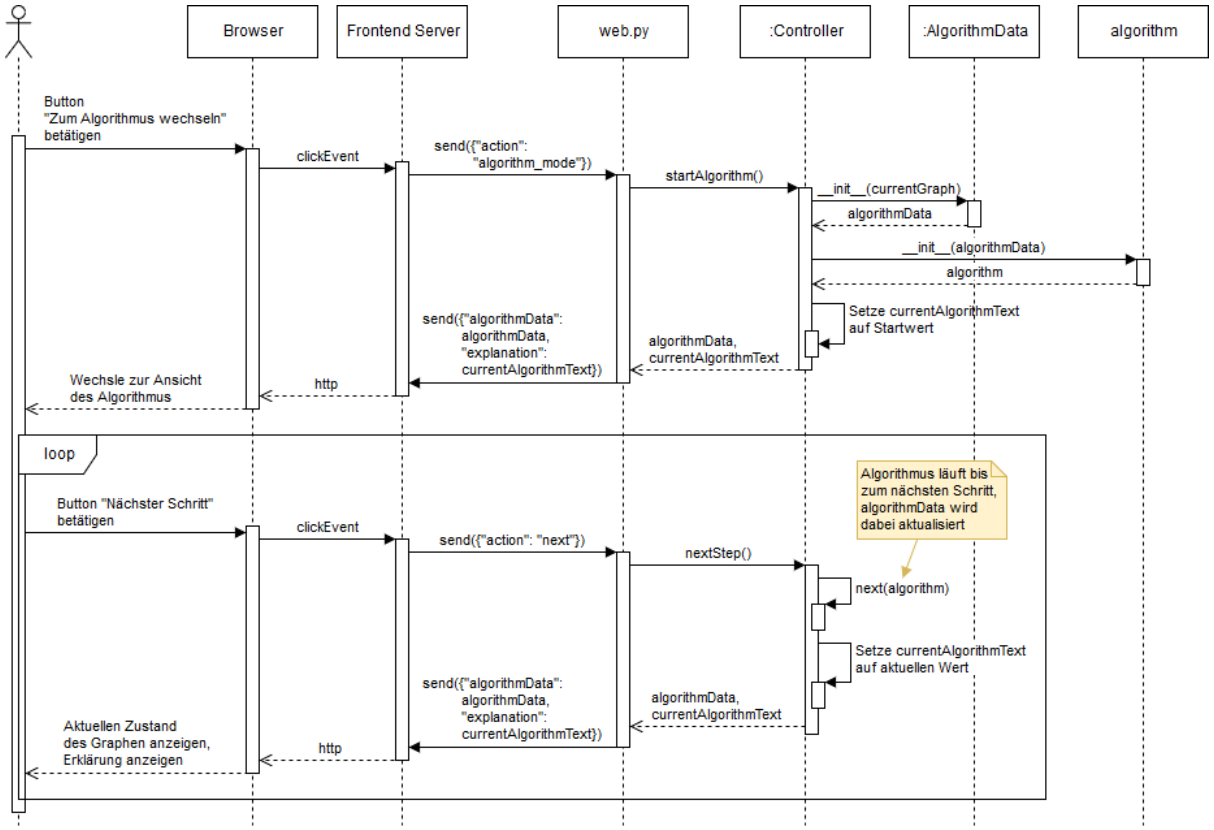


Abbildung 5.5: Sequenzdiagramm zum Usecase „Algorithmus starten und ausführen“

5.4 Implementierung des Algorithmus

Der Algorithmus ist im Modul `src\controller\arbitraryalgorithm.py` implementiert. Da die Basis für die Implementierung der Pseudo-Code aus Kapitel 4.2 ist, wird der Algorithmus hier nicht mehr Schritt für Schritt erklärt. Stattdessen wird die Umsetzung in einem Aktivitätsdiagramm in Abbildung 5.6 dargestellt. Zu sehen ist dort eine vereinfachte Darstellung der Funktion **`algorithm()`**. Um das Diagramm übersichtlicher zu gestalten, wurden für das Verständnis unwichtige Textausgaben, Variableninitialisierungen und Parameter bei einigen Funktionsaufrufen weggelassen. Im Folgenden werden die wichtigsten Herangehensweisen der Implementierung des Algorithmus beschrieben. Alle hier beschriebenen Funktionen finden sich im oben genannten Modul.

Um den Algorithmus Schritt für Schritt durchgehen zu können, wurde die Funktion **`algorithm()`** als sogenannter Generator implementiert. Generatoren sind in Python

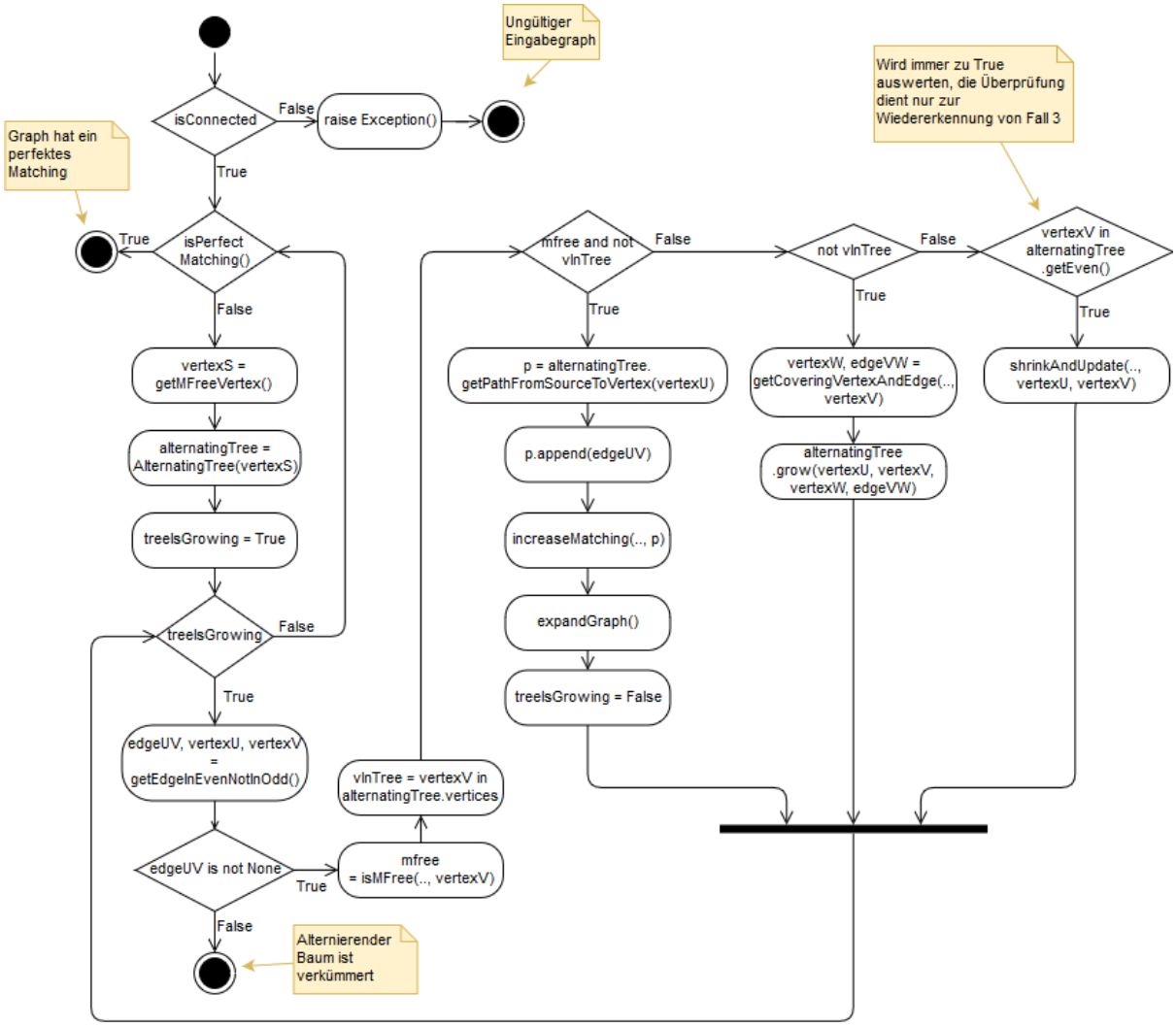


Abbildung 5.6: Aktivitätsdiagramm zum implementierten Algorithmus

Iteratoren, welche nicht direkt durchgeführt werden, sondern immer nur bis zu dem Schlüsselwort **`yield`** laufen. **`yield`** funktioniert dann wie das Schlüsselwort **`return`**, es kann also Werte zurückgeben. Bei erneutem Aufruf der Funktion wird dann von dem zuletzt ereichten **`yield`** die Funktion fortgesetzt. Bei der Erstellung des Generators wird ein Objekt vom Typ **`AlgorithmData`** übergeben. Das Attribut **`graph`** des **`AlgorithmData`**-Objektes bildet den Eingabegraphen für den Algorithmus.

Um die Eingabe auf Zusammenhang zu überprüfen, wird der Graph in der Funktion **`isConnected`** rekursiv per Tiefensuche traversiert und jeder erreichte Knoten markiert. Sind nach diesem Durchgang alle Knoten markiert, so ist der Graph zusammenhängend. Anschließend werden die Markierungen wieder gelöscht.

Die Überprüfung, ob das aktuelle Matching perfekt ist, wird über einfache Mengenoperationen innerhalb der Funktion **`isPerfectMatching`** realisiert. Dazu werden alle Knoten, die vom Matching überdeckt werden, in einer Menge **`matchingVertices`** gesammelt und alle Knoten des Graphen in einer zweiten Menge **`graphVertices`** gespeichert. Danach wird die Differenz der Mengen berechnet (also **`graphVertices`** *setminus* **`matchingVertices`**). Ist die so entstandene Menge leer, so ist das Matching perfekt.

Die Funktion **getMFreeVertex** liefert einen Matching-freien Knoten zurück, indem zunächst alle vom Matching überdeckten Knoten gesammelt werden und dann in einer Schleife die Knoten des Graphen durchgegangen werden. Der erste Knoten, der nicht in der Menge der vom Matching überdeckten Knoten liegt, wird zurückgegeben.

Um die Kante $\{u, v\}$ zu finden, wurde die Funktion **getEdgeInEvenNotInOdd** implementiert. Hier werden die geraden Knoten des alternierenden Baumes in einer Schleife durchgegangen. Für jeden Knoten u wird überprüft, ob sie einen adjazenten Knoten v haben, welcher nicht in den ungeraden Knoten ist. Wird ein solcher Knoten gefunden, werden die Knoten u und v und die entsprechende Kante $\{u, v\}$ zurückgegeben, die Schleife wird dadurch beendet.

Eine wichtige Überprüfung ist in der Funktion **isMfree** implementiert. Hier wird für einen übergebenen Knoten überprüft, ob dieser von einer Kante aus einem übergebenen Matching überdeckt ist. Dazu werden die Kanten des übergebenen Knotenobjektes durchgelaufen. Ist eine dieser Kanten in dem Matching, welches als Liste von Kanten implementiert ist, wird der Boolean-Wert **False** zurückgegeben. Wird keine Kante des Knotens im Matching gefunden, wird stattdessen der Boolean-Wert **True** zurückgegeben.

Die Funktion **getCoveringVertexAndEdge** sucht für einen vom Matching überdeckten Knoten die entsprechende inzidente Kante im Matching und den dazugehörigen adjazenten Knoten. Dazu werden die Kanten des Matchings nach entsprechenden Kanten durchsucht.

Um das Matching zu vergrößern, wurde die Funktion **increaseMatching** implementiert. Sie bekommt ein Matching und einen augmentierenden Pfad je als Liste von Kanten übergeben. Diese werden in Mengen umgewandelt und ein neues Matching wird per Mengenoperationen wie in der Definition im Kapitel 3.3.2 berechnet.

Das Extrahieren von eventuell vorhandenen Superknoten ist in der Funktion **expandGraph** umgesetzt. Zunächst wird überprüft, ob überhaupt ein Superknoten vorhanden ist. Dazu werden die Knoten des Graphen durchlaufen. Sobald ein Superknoten gefunden wurde, wird die Suche abgebrochen und dieser Superknoten extrahiert. Nach dem Extrahieren wird erneut überprüft, ob noch Superknoten vorhanden sind. Dies wird solange wiederholt, bis der Graph keine Superknoten mehr besitzt. Das Extrahieren selbst startet mit dem Anpassen des Matchings. Wie in Kapitel 4.2 beschrieben muss hierzu ein Knoten aus dem Kreis des Superknotens entfernt werden, damit man einen augmentierenen Weg zum Erweitern des Matchings hat. Dazu wird eine Kante aus dem Matching gesucht, die den Superknoten überdeckt. Existiert eine solche Kante, dann ist der zu entfernende Knoten gleich dem Knoten der Kante, welcher innerhalb des Kreises liegt. Findet man dagegen keine solche Kante, so ist der Superknoten Matching-frei und es wird für den zu entfernenden Knoten einfach der erste Knoten aus der internen Liste des Superknotenobjektes genommen. Um den augmentierenden Weg zu finden wird nun eine sortierte Liste der Kanten erstellt, welche innerhalb des Kreises im Superknoten liegen. Sortiert wird die Liste so, dass immer zwei inzidente Kanten nebeneinander liegen. Die erste und die letzte Kante sind die beiden zu dem zu entfernenden Knoten inzidenten Kanten. Beide werden nach dem Sortieren aus der Liste entfernt. Die daraus resultierende Liste bildet den augmentierenden Weg. Das Matching wird nun erweitert, indem alle Listenelemente mit einem ungeraden Listenindex dem Matching hinzugefügt werden. Zum Schluss wird noch die im Graphen implementierte Funktion **extractCircle** mit dem Superknoten als Parameter aufgerufen, welche den Superknoten aus dem Graphen entfernt und alle Knoten und Kanten des Superknotens

wieder hinzufügt. Hier werden auch alle Kanten, die zwar mit dem Superknoten inzident waren, aber nicht im Kreis lagen, wieder mit ihren alten Knoten verbunden.

In der Funktion `getCircle` wird der Kreis zwischen zwei Knoten u und v, welche beide in einem alternierenden Baum liegen, in Form einer Liste von Knoten zusammengestellt. Bei Aufruf dieser Funktion ist bereits bekannt, das ein solcher Kreis existieren muss. Für beide Knoten wird der Weg von sich selbst zur Wurzel des Baumes gesucht (dieser könnte auch für einen der Knoten leer sein, wenn der Knoten selbst die Wurzel ist). Die beiden Wege werden nun in Mengen von Kanten umgewandelt. Aus diesen beiden Mengen wird die symmetrische Differenz gebildet, es werden also nur Kanten gesucht, die nur in einer der Mengen zu finden sind. Aus der so entstandenen Kantenmenge werden alle zu diesen Kanten inzidenten Knoten in einer Knotenmenge gespeichert. Diese Knotenmenge bildet dann den Kreis und wird als Liste zurückgegeben.

Die Funktion `shrinkAndUpdate` orientiert sich wie der Algorithmus am Pseudocode des Kapitels 4.2 und soll daher hier nicht genauer erklärt werden.

5.5 Tests

Zum Testen der Applikation wurden Integrationstests durchgeführt und Unittests erstellt. Getestet wurde auf den Betriebssystemen Windows 10 und Ubuntu 16.04 jeweils mit dem Browser Firefox in der Version 63.0. Auf Windows wurde außerdem mit dem Browser Chrome in der Version 70.0.3538.77 getestet, während auf Ubuntu der Chromium-Browser in der Version 70.0.3538.67 genutzt wurde.

5.5.1 Integrationstests

Für die Integrationstests wurde zunächst ein Testprotokoll erstellt, in welchem zu allen Usecases die Voraussetzungen, Schritte der Durchführung und das erwartete Ergebnis aufgeschrieben wurde. Danach wurden die Schritte der Usecases ausgeführt und das Verhalten wurde im Testprotokoll notiert. Wurden alle Erwartungen erfüllt, steht in der entsprechenden Zelle der Text „Ok“, andernfalls wurde eine Beschreibung des tatsächlichen Verhaltens aufgeschrieben. In der Tabelle 5.1 ist der Test zum Usecase „Speichern“ aufgeführt. Das komplette Testprotokoll befindet sich auf der beiliegenden CD.

Bis auf einen Test konnten alle Usecases erfolgreich durchgeführt werden. Lediglich beim Usecase „Algorithmus mit Testgraphen durchlaufen“ kam es bei einer Situation zu einem Anzeigefehler. Wenn der Algorithmus mit einem nicht zusammenhängenden Graphen gestartet wird, so wird beim ersten Klick auf den Button „Nächster Schritt“ erwartungskonform eine Fehlermeldung angezeigt, die den Nutzer darauf hinweist, dass der Graph nicht zusammenhängend ist. Die Fehlermeldung wird allerdings bei erneutem Buttondruck unerwünschter Weise wieder ausgeblendet. Da der Algorithmus aber trotzdem nicht durchläuft, wurde dieser Anzeigefehler als nicht kritisch eingestuft und wegen Zeitmangels nicht gefixt.

Usecase	Voraussetzungen	Erwartungen	Schritte	Ergebnis
Speichern				
a) Speichern				
	Graphbearbeitungssicht ist aktiv	Im Speicherordner liegt eine .json-Datei mit dem eingegebenen Dateinamen	Button „Speichern“ klicken	ok
			Speicherdialog öffnet sich	ok
			Dateinamen eingeben	ok
			Button „Speichern“ klicken	ok
b) Speichern abbrechen				
	Speicherdialog ist offen	Im Speicherordner liegt keine neue oder gerade geänderte Datei	Button „Abbrechen“ klicken	ok

Tabelle 5.1: Usecase „Speichern“ aus dem Testprotokoll

5.5.2 Unittests

Die Integrationstests decken vor allem die Tests der Benutzeroberfläche ab. Dadurch wurden auch Funktionen, wie zum Beispiel das Knotenerstellen, hinreichend getestet. Dies ist der Grund dafür, dass nur Unittests für Module erstellt wurden, die entweder als kritisch angesehen, oder deren Funktion nicht genügend durch die Integrationstests getestet wurden. Die vollständigen Unittests sind in der beigelegten CD zu finden.

So wurde beispielsweise für die Klasse `Result` überprüft, ob die Textrepräsentation richtig funktioniert. Außerdem gibt es Unittests zu den Funktionen `load`, `clearGraph` und `deleteVertex` der Klasse `Controller`. Bei all diesen Funktionen wurde auch das Rückgängig-Machen der jeweiligen Aktion getestet.

Das größte Testmodul ist das Modul `arbitraryalgorithmtest.py`, welches Tests zum Algorithmus bereitstellt. Hier wird anhand von verschiedenen Testgraphen, die jeweils extra für die Tests erstellt wurden, geprüft, ob der Algorithmus das richtige Ergebnis zurückgibt. Auch wurden für einzelne Funktionen des Algorithmus, beispielsweise für die Funktion `isConnected()`, eigene Testklassen implementiert.

Alle erstellten Unittests können erfolgreich ausgeführt werden.

5.6 Verwendung der Applikation

Wie bereits erwähnt, hat die Applikation zwei Hauptfunktionen. Zum Einen ist dies die manuelle Erstellung von Graphen, zum Anderen die Ausführung und Visualisierung des vorgestellten Algorithmus für allgemeine Graphen. In diesem Abschnitt soll das Starten, Beenden und die Verwendung dieser Funktionen erklärt werden.

5.6.1 Starten und Beenden der Applikation

In der beiliegenden CD ist sowohl der Quellcode als auch die ausführbare Applikation vorhanden. Dort findet sich neben der Beschreibung, wie der Inhalt der CD aufgebaut ist auch eine Beschreibung, wie der Quellcode gebaut werden kann.

Die ausführbare Applikation befindet sich für das Betriebssystem Windows im Ordner `perfectmatchings-windows` und für Unix im Ordner `perfectmatchings-unix`. Während unter Windows einfach die Datei **`perfectmatchings.exe`** durch einen Doppelklich ausgeführt werden kann, muss unter Unix die Datei **`perfectmatchings`** aus einer Konsole aus aufgerufen werden.

Nach Ausführen der jeweiligen Datei öffnet sich im Standardbrowser des Nutzers die Applikation und es können sofort Graphen bearbeitet und der Algorithmus ausgeführt werden. Weitere Installationsschritte sind nicht nötig.

Beim ersten Starten der Applikation wird der in Kapitel 5.2.4 erwähnte Speicherordner angelegt. Auf der CD befinden sich im Ordner `Testgraphen` einige .json-Dateien, welche nach dem ersten Programmstart in den entsprechenden Ordner kopiert werden können. Im Anschluss stehen sie zum Laden bereit, die Applikation muss dafür nicht neu gestartet werden.

Durch das Ausführen der Applikation wird ein Konsolenfenster geöffnet bzw. das Konsolenfenster weiter genutzt, in welchem die Applikation eventuell gestartet wurde. Um das Programm zu beenden, muss in der Konsole nur die Enter-Taste betätigt werden.

5.6.2 Erstellung von Graphen

Beim Starten der Applikation wird im Browser die Graphbearbeitungssicht geöffnet. Hier können Graphen erstellt und bearbeitet werden. Wie die Ansicht aussieht, ist in Abbildung 5.7 zu erkennen.

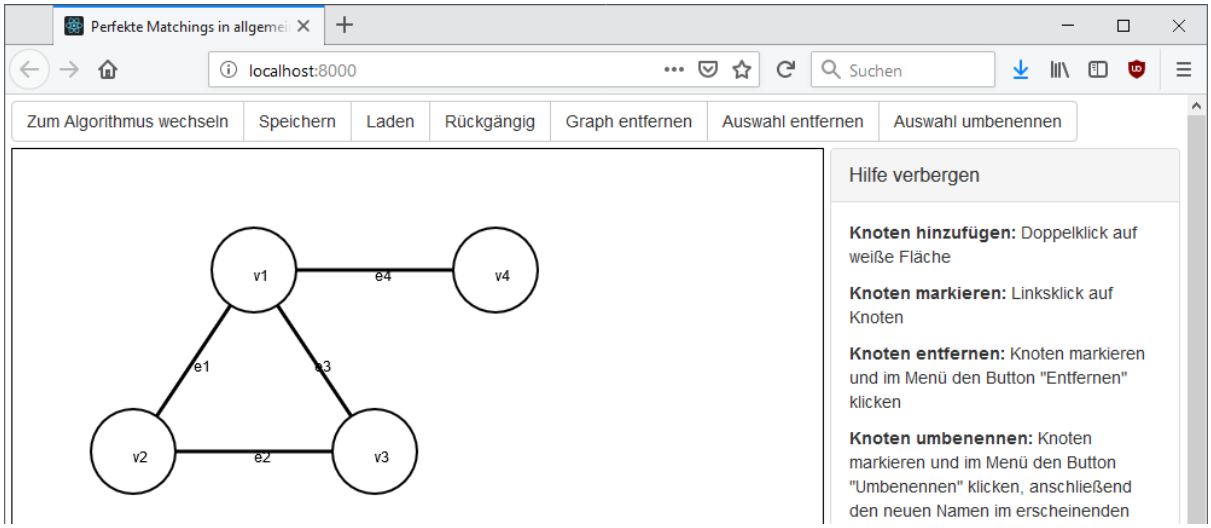


Abbildung 5.7: Graphbearbeitungssicht

Am oberen Rand der Applikation befindet sich eine Menüleiste mit verschiedenen Buttons. Darunter ist eine Zeichenfläche zum Graphen bearbeiten und ein Hilfetext, welcher alle Funktionen erläutert. Initial ist die Zeichenfläche leer und der Hilfetext eingeklappt, letzterer kann mit einem Klick auf den Text „Hilfe anzeigen“ ausgeklappt und genauso auch wieder eingeklappt werden.

Knoten können per Doppelklick auf die Zeichenfläche erstellt und per Drag&Drop verschoben werden. Ein einzelner Klick markiert einen Knoten. Soll eine Kante erstellt werden,

müssen hintereinander zwei Knoten markiert werden.

Mit dem Button „Zum Algorithmus wechseln“ kann der Algorithmus gestartet werden. Näheres findet sich dazu im Abschnitt 5.6.3.

Soll der erstellte Graph gespeichert werden, muss der Button „Speichern“ betätigt werden. Der sich daraufhin öffnende Speicherdialog ist in Abbildung 5.8 zu sehen. Hier kann ein Dateiname eingegeben werden. Dabei ist es egal, ob die Dateiendung .json mit angegeben wird oder nicht. Alle im Speicherordner schon vorhandenen Dateien werden in einer Liste angezeigt.

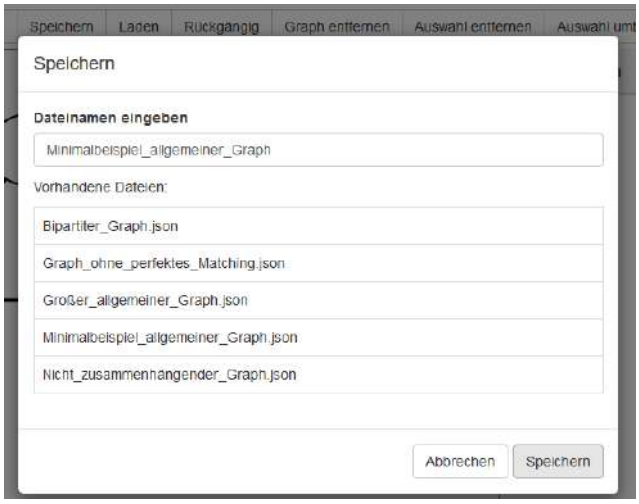


Abbildung 5.8: Speicherdialog

Analog zum Speichern öffnet sich beim Drücken des Buttons „Laden“ ein Ladedialog. Hier werden in einer Liste die vorhandenen Dateien angezeigt. Ein Klick auf eines der Listenelemente lädt die entsprechende Datei. Der Dialog wird in Abbildung 5.9 gezeigt.

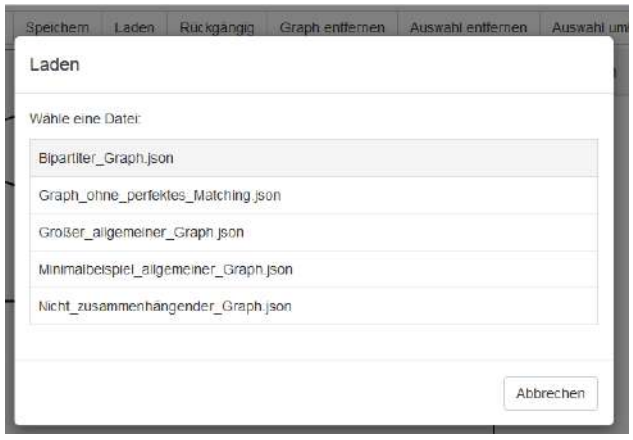


Abbildung 5.9: Ladedialog

Der Button „Rückgängig“ macht, wie der Name schon sagt, die letzte durchgeführte Aktion rückgängig. Zum Beispiel können dadurch einfach falsch erstellte Kanten wieder entfernt werden. Bis auf das Speichern und das Bewegen von Knoten können alle Funktionen innerhalb der Graphbearbeitungssicht rückgängig gemacht werden.

Um die komplette Zeichenfläche zu leeren kann der Button „Graph entfernen“ genutzt werden.

Der Button „Auswahl entfernen“ entfernt dagegen nur den aktuell ausgewählten Knoten oder die aktuell ausgewählte Kante.

Soll ein Knoten oder eine Kante umbenannt werden, muss das Element markiert werden. Anschließend kann der Button „Auswahl umbenennen“ betätigt werden. Wie in Abbildung 5.10 zu sehen, öffnet sich dadurch ein Dialog, in dem ein neuer Dateiname eingegeben werden kann. Der Button „Umbenennen“ kann erst betätigt werden, wenn mindestens ein Zeichen im Textfeld steht. Ist dies der Fall, kann mit Klick auf diesen Button das Umbenennen durchgeführt werden.

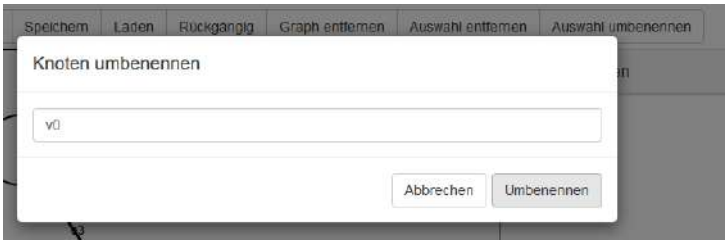


Abbildung 5.10: Umbenennendialog

Sowohl der Speicher- und Ladevorgang als auch das Umbenennen können abgebrochen werden.

5.6.3 Visualisierung des Algorithmus

Die Algorithmus-Sicht ist ähnlich aufgebaut wie die Graphbearbeitungssicht. Oben befinden sich Buttons für Aktionen, darunter wird der Graph angezeigt und links neben dem Graphen ist ein Bereich für den Erklärungstext. Wie die Ansicht aussieht, wenn mit einem vorher geladenen Graphen zu ihr gewechselt wurde, ist in Abbildung 5.11 zu sehen. Anders als in der Graphbearbeitungssicht kann hier der Graph allerdings in keiner Art und Weise bearbeitet werden. Dazu muss zunächst wieder die Sicht gewechselt werden. Dies geschieht mit dem Betätigen des Buttons „Zur Graphbearbeitung wechseln“. Wird dieser ausgeführt, wird in der Graphbearbeitungssicht wieder der Eingabegraph angezeigt.

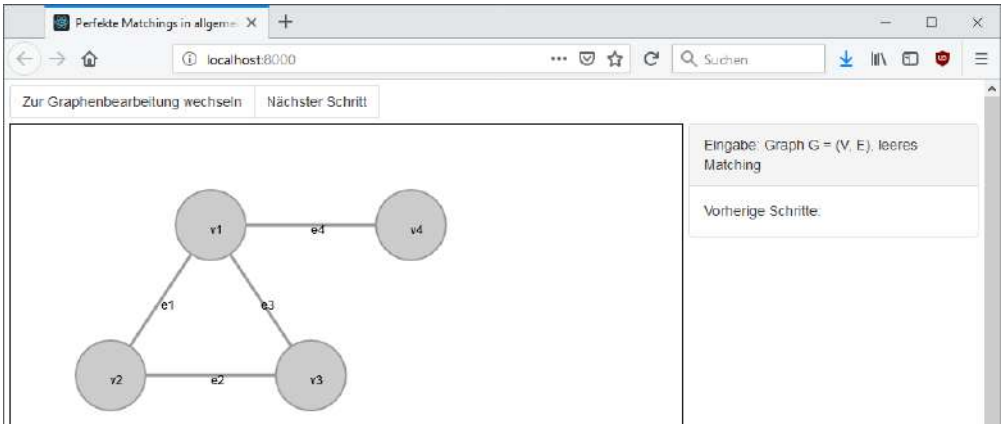


Abbildung 5.11: Anzeige beim Wechsel zur Algorithmus-Sicht

Mit dem Button „Nächster Schritt“ kann ein Schritt des Algorithmus ausgeführt werden. Wie ein Graph nach mehreren Schritten aussieht, wird in Abbildung 5.12 gezeigt.

Der Erklärungstext ist in zwei Abschnitte unterteilt. Im vorgehobenen grauen Bereich oben ist die Beschreibung des aktuellen Schrittes zu finden. Darunter, im zweiten Bereich, sind die vorherigen Schritte aufgelistet. Diese sind chronologisch sortiert.

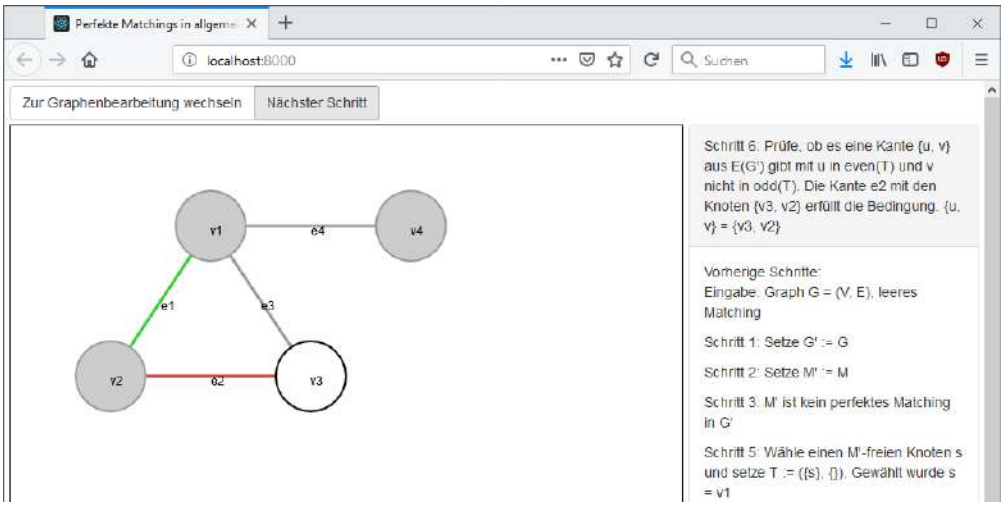


Abbildung 5.12: Anzeige der Algorithmus-Sicht während des Algorithmus-Durchlaufs

Die Farben der Knoten und Kanten halten sich dabei an die in den vorherigen Kapiteln beschriebenen Darstellungen. Eine komplette Übersicht über das Farbschema kann in Abbildung 5.13 gefunden werden.

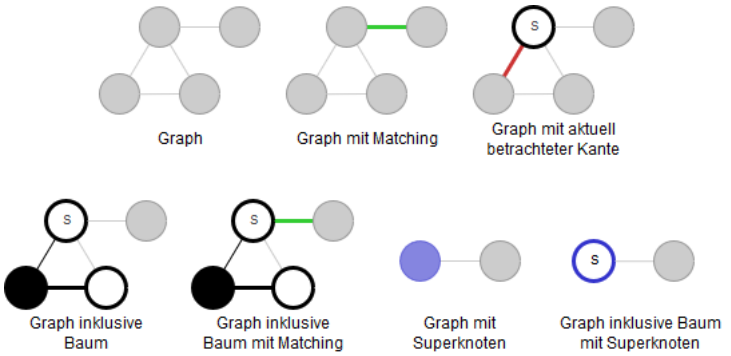


Abbildung 5.13: Farbschema

Allgemein werden Knoten und Kanten grau dargestellt. Kanten, die nur in einem Matching sind, werden grün gezeichnet. Die aktuell betrachtete Kante $\{u, v\}$ wird rot markiert. Existiert ein alternierender Baum, so sind die geraden Knoten weiß ausgefüllt, während die ungeraden schwarz gefüllt sind. Kanten in einem Baum sind schwarz gemalt, wenn sie zusätzlich zum Matching gehören bestehen sie außerdem aus dickeren Linien. Superknoten werden blau ausgefüllt, gehören sie zu einem Baum, dann ist die Umrandung blau. Da ein Superknoten nicht in den ungeraden Knoten eines Baumes liegen kann, ist dafür keine extra Darstellung nötig.

6 Fazit

Dieses Kapitel stellt eine Zusammenfassung der Arbeit bereit und gibt einen Ausblick auf weiterführende Themen und mögliche Verbesserungen der Implementierung.

6.1 Zusammenfassung

Ziel der Arbeit war es, einen Algorithmus zu erklären, um perfekte Matchings nicht nur in bipartiten, sondern auch allgemeinen Graphen zu finden. Dazu wurden zunächst Grundlagen der Graphentheorie erklärt und wichtige Konstruktionen wie die Kreiskontraktion eingeführt. Danach wurden Matchings definiert und Wege aufgezeigt, wie diese Matchings vergrößert werden können. Hierfür wurden alternierende und augmentierende Wege definiert, zusätzlich wurden alternierende Bäume vorgestellt. Im Hauptteil der Arbeit wurde zuerst ein Algorithmus für bipartite Graphen erklärt, welcher auf den gleichen Ideen basiert, wie der im nächsten Abschnitt vorgestellte Algorithmus für allgemeine Graphen. Um die Funktionsweise der Algorithmen deutlich zu machen, wurden sie jeweils Schritt für Schritt beschrieben, als Pseudocode bereitgestellt und in einem Beispiel durchgegangen. Für den Algorithmus für allgemeine Graphen wurde dann in einem nächsten Schritt eine Implementierung vorgestellt. Hier wurde insbesondere auf die verwendete Architektur, auf die Zusammenarbeit der einzelnen Komponenten und auf die eigentliche Implementierung des Algorithmus eingegangen. Des Weiteren wurde die Verwendung der entstandenen Applikation vorgestellt. An dieser Stelle ist auch zu sehen, wie der Algorithmus für perfekte Matchings schlussendlich visualisiert wurde.

6.2 Ausblick

Außer den perfekten Matchings gibt es noch weitere Arten von Matchings, die betrachtet werden können. So kann es beispielsweise in manchen Graphen mehrere perfekte Matchings geben. Wenn nun der Graph gewichtete Kanten hat, könnte man nach einem minimalen oder maximalen perfekten Matching suchen.

Der vorgestellte Algorithmus *Perfect-Matching* kann außerdem so erweitert werden, dass er in allgemeinen Graphen auch Matchings mit maximaler Größe findet. Dass heißt, selbst wenn ein Graph kein perfektes Matching besitzt, kann trotzdem das größtmögliche Matching gefunden werden. Wie diese Erweiterung funktioniert, kann in [KN12] nachgelesen werden.

Die Implementierung bietet in ihrer jetzigen Version die Möglichkeit, Graphen zu erstellen, zu laden und zu speichern und den Algorithmus für allgemeine Graphen Schritt für Schritt

auszuführen. Allerdings ist die Graphbearbeitungssicht sehr rudimentär. Beide Ansichten bieten Platz für Verbesserungsmöglichkeiten und neue Features.

Momentan kann die Bewegung eines Knotens nicht rückgängig gemacht werden. Zwar liegt die Implementierung eines Kommandos vor, welches in die Klasse `History` eingefügt werden könnte und somit auch rückgängig gemacht werden könnte, allerdings würde dann jede kleinste Bewegung während des Ausführens von Drag&Drop eingespeichert werden. Statt die Bewegung zu verfolgen, müsste also eigentlich beim Loslassen des Knotens ein Kommando erstellt werden. Der Grund für die aktuelle Implementierung liegt darin, dass Kanten anhand der Knotenkoordinaten gezeichnet werden und diese daher immer aktuell gehalten werden müssen.

Ein zusätzlicher offener Punkt ist die fehlende Skalierung der Zeichenfläche. Der angezeigte Graph wird nicht angepasst, wenn das Fenster verkleinert wird. Die einfachste Lösungsmöglichkeit wäre das Anbieten von Scrollbars.

Auch der Speicherdialog bietet Erweiterungspotential. So sollte eventuell eine Warnung angezeigt werden, wenn der Nutzer dabei ist, eine Datei zu überschreiben. Zusätzlich wäre es praktisch, wenn ein Klick auf die Elemente der Liste der vorhandenen Dateien den entsprechenden Namen in das Textfeld einfügen würde.

Der Visualisierung des Algorithmus fehlt, wie im letzten Kapitel bereits erwähnt, das Nutzerfeedback darüber, ob der Algorithmus bereits terminiert ist oder nicht. Hierfür fehlt das entsprechende Signal aus dem Backend.

Ein interessantes zu implementierendes Feature wäre zum Beispiel ein Button, welcher bei Betätigung den Algorithmus komplett durchlaufen lässt, sowie die Möglichkeit, bei Klick auf eine Zeile des Erklärungstextes den Zustand des Graphen an diesem Zeitpunkt anzuzeigen.

Beim Anzeigen eines Superknotens könnte man dem Nutzer die Möglichkeit anbieten, sich eine Ansicht des kontrahierten Kreises anzeigen zu lassen. Dazu könnte man beispielsweise beim Hovern über den entsprechenden Knoten den Kreis zeichnen.

In beiden Ansichten wäre außerdem ein Button denkbar, der die Knoten- und Kantentexte im gezeichneten Graphen ein- und ausblendet.

Ebenso wäre ein Einstellungsdialog innerhalb der Applikation ein gutes Feature. In diesem könnten dann zum Beispiel die Größe der Knoten, Dicke der Linien oder verwendete Farben eingestellt werden. Eine weitere denkbare Einstellung wäre die Möglichkeit, Standardtexte für Knoten, Superknoten und Kanten anhand von regulären Ausdrücken setzen zu können.

Um weitere Verbesserungsmöglichkeiten und Features zu finden, könnten Tests mit den Nutzern durchgeführt werden. Darüber hinaus wäre eine Studie interessant, die versucht herauszufinden, ob Studierende durch die Visualisierung das Thema und insbesondere den Algorithmus besser verstehen.

Literatur

[ADH98] Armen S. Asratian, Tristan M. J. Denley und Roland Häggkvist. *Bipartite graphs and their applications*. Cambridge tracts in mathematics ; 131. Cambridge [u.a.]: Cambridge Univ. Press, 1998.

[Bri05] Manfred Brill. *Mathematik für Informatiker : Einführung an praktischen Beispielen aus der Welt der Computer*. 2., völlig neu bearb. Aufl. München [u.a.]: Hanser, 2005.

[Die10] Reinhard Diestel. *Graphentheorie*. 4. Aufl. Heidelberg [u.a.]: Springer, 2010.

[Edm65] Jack Edmonds. „Paths, trees and flowers“. In: *Canadian Journal of Mathematics* 17 (1965), S. 449–467.

[Gib85] Alan M. Gibbons. *Algorithmic graph theory*. Cambridge [u.a.]: Cambridge Univ. Press, 1985.

[HK71] John E. Hopcroft und Richard M. Karp. „A n5/2 algorithm for maximum matchings in bipartite“. In: *Series-parallel irreducibility: Machine oriented definitions and proofs* (1971). DOI: http://dx.doi.org/10.1109/SWAT.1971.1.

[Jun13] Dieter Jungnickel. *Graphs, networks and algorithms*. 4th ed. Algorithms and computation in mathematics ; 5. Berlin [u.a.]: Springer, 2013.

[KN12] Sven O. Krumke und Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. 3. Aufl. Leitfäden der Informatik. Wiesbaden: Teubner, 2012.

[Kus18] Sabine Kuske. *Algorithmen auf Graphen*. Vorlesungsskript. Universität Bremen, 2018.

[Mün] TU München. *TUM - Mathematik - M9 - Graphalgorithmen*. URL: https://www-m9.ma.tum.de/Allgemeines/GraphAlgorithmen (besucht am 13. 10. 2018).

[Ree79] Trygve Mikjel H. Reenskaug. *The original MVC reports*. Oslo: The University of Oslo, 1979.

[Sie14] Florian Siebler. *Design Patterns mit Java : eine Einführung in Entwurfsmuster*. 1. Aufl. München: Hanser, 2014.

Abbildungsverzeichnis

Tabellenverzeichnis