technische universität
dortmund

Bachelorarbeit

Matchings in bipartiten Graphen im Semi-Streaming Modell

Jonas Ellert

25. August 2016

Betreuer:
Dr. Nils Kriege
Prof. Dr. Petra Mutzel

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Durch Graphen lassen sich diverse Daten darstellen, insbesondere wenn diese die Relationen zwischen Entitäten beschreiben. Beispielsweise lassen sich Freundschaftsbeziehungen in sozialen Netzwerken oder auch die Links zwischen Internetseiten als Graph modellieren, wobei die anfallenden Datenmengen oft deutlich schneller wachsen als die Kapazitäten der zur Auswertung verfügbaren Hardware. So besteht ein 2012 erfasster Webgraph bereits aus 3.5 Mrd. Internetseiten und über 128 Mrd. Links zwischen den Seiten.[1] Dadurch sind traditionelle Algorithmen, welche *Random-Access* auf alle Kanten erfordern, oftmals nicht oder nur kostspielig verwendbar, da der verfügbare Arbeitsspeicher zu klein ist, um den gesamten Graphen zu beinhalten. Aus diesem Grund wurden in den letzten Jahren verschiedene Streaming Modelle entwickelt [12][9][1], die dabei helfen sollen, große Graphen trotz mitunter stark begrenzter Ressourcen effizient zu verarbeiten. An großer Beliebtheit erfreut sich unter anderem das *Semi-Streaming Modell* [12], welches in dieser Arbeit eine zentrale Rolle spielt. Beim Streaming werden die Kanten des Graphen in der Regel sequentiell direkt von der Festplatte oder einem vergleichbaren Massenspeichermedium gelesen und unmittelbar verarbeitet, wobei häufig mehrere *Passes* über diesen Kanten-Stream erforderlich sind. Insgesamt darf dabei nur eine von der Knotenzahl abhängige Menge an Arbeitsspeicher belegt werden. Mit dieser Methode ist man also nicht mehr nur von der Prozessor- und Arbeitsspeicherleistung abhängig, sondern auch stark von der Lesegeschwindigkeit der Festplatte. Umso wichtiger ist es daher, dass man die Kanten möglichst selten lesen muss, weswegen es sich selbst für sonst in Polynomialzeit lösbare Probleme lohnen kann, sich mit einer approximativen Lösung zufrieden zu stellen.
Ein Beispiel dafür ist das *Maximum-Matching*-Problem in ungewichteten bipartiten Graphen, dessen Ziel das Finden eines *kardinalitätsmaximalen Matchings* ist. Obwohl schon seit den 1970er Jahren effiziente und exakte Lösungen dafür vorhanden sind [14], gibt es Anwendungsfälle auf großen Graphen, in denen eine Approximation an ein Maximum-Matching genügt. Dazu zählt zum Beispiel das Bestimmen der Kontrollierbarkeit von

[1] http://webdatacommons.org/hyperlinkgraph/index.html

Netzwerken, welche zwar an für sich keine bipartiten Graphen sind, aber auf solche abgebildet werden können [18]. Die Anzahl der sogenannten Treiberknoten, die unmittelbar von der Größe des kardinalitätsmaximalen Matchings abhängt, ist dabei das Maß, welches die Kontrollierbarkeit angibt. Betrachtet man etwa Genregulationsnetzwerke, in welchen DNS-Segmente miteinander interagieren, so sind viele Treiberknoten vorhanden. Im Gegensatz dazu zeigen die wenigen Treiberknoten, die in verschiedenen sozialen Netzwerken vorhanden sind, dass bereits eine kleine Teilmenge der Nutzer das Netzwerk vollständig kontrollieren kann [18]. Das Maximum-Matching Problem ist also nicht nur ein klassisches Problem der theoretischen Informatik, sondern erfreut sich auch heutzutage noch an einer hohen Relevanz.
Deswegen ist es Ziel dieser Arbeit, einen effizienten Streaming-Algorithmus zu entwickeln, der Maximum-Matchings approximiert. Dabei sollen die experimentellen Testergebnisse, die von einem bestehenden Algorithmus erreicht wurden [15], verbessert werden.

1.1 Aufbau der Arbeit

Zuallererst werden wir uns mit diversen formalen Grundlagen vertraut machen (Kapitel 2), wobei wir mit bipartiten Graphen und Matchings beginnen und anschließend alternierende Pfade und Bäume kennenlernen, die ein beliebtes Mittel zur Suche von großen Matchings sind. Danach wird das Semi-Streaming Modell erläutert, welches die Rahmenbedingungen der hier präsentierten Algorithmen festlegt.
In Kapitel 3 befassen wir uns mit bereits publizierten Arbeiten, die mit den Schwerpunkten Matchings und Streaming in Verbindung stehen. Bereits recht detailliert gehen wir auf zwei sogenannte *DAP-Algorithmen* ein, welche Maximum-Matchings in einem Streaming Modell approximieren. Mit dem sogenannten *Strong-Spanning-Tree-Algorithmus* lernen wir außerdem eine alternative Nicht-Streaming-Methode zur Konstruktion von Matchings kennen.
Kapitel 4 zeigt einen Versuch, den Strong-Spanning-Tree-Algorithmus ins Semi-Streaming Modell zu adaptieren. Obwohl die Voraussetzungen für eine solche Adaption gegeben sind, werden wir sehen, dass es einige grundlegende Probleme gibt, die eine effiziente Umsetzung verhindern.
Als wesentlich effektiver hat sich die in Kapitel 5 vorgestellte Abänderung des baumbasierten DAP-Algorithmus erwiesen. Es wird nicht nur detailliert auf Funktionsweise und Korrektheit eingegangen, sondern auch eine genaue Abgrenzung zum bestehenden DAP-Verfahren gegeben. Ebenso wird auf theoretische Schranken bezüglich Speicherplatz, Laufzeit und Anzahl der Passes eingegangen.
Zu guter Letzt sehen wir in einer ausführlichen Evaluierung, wie sich der neue Algorithmus in der Praxis schlägt und enden mit einem kurzen Fazit und Ausblick.

Kapitel 2

Grundlagen

2.1 Bipartite Graphen

Zunächst wollen wir die formalen Grundlagen dieser Arbeit genau festlegen und beginnen dabei mit der Definition bipartiter Graphen. Allgemein ist ein ungewichteter ungerichteter Graph ein Tupel $G = (V, E)$, das aus der Menge V seiner *Knoten* und der Menge $E \subseteq \{\{v_1, v_2\} \mid v_1 \neq v_2 \wedge v_1, v_2 \in V\}$ seiner *Kanten* besteht, die zwischen den Knoten verlaufen. Wir definieren:

2.1.1 Definition (Bipartiter Graph). Ein ungewichteter, ungerichteter Graph $G = (V, E)$ heißt *bipartit* und wir schreiben $G = (A, B, E)$, wenn sich V wie folgt in die *Partitionszellen* A und B partitionieren lässt:

$$\exists A, B \subseteq V : A \cap B = \varnothing \ \wedge \ A \cup B = V \wedge E \subseteq \{\{a, b\} \mid a \in A \wedge b \in B\}$$

Die Knoten aus A heißen *Zeilen*, während die Knoten aus B *Spalten* sind. ◁

Für bipartite Graphen definieren wir die Menge aller theoretisch möglichen Kanten $E_{max} := \{\{a, b\} \mid a \in A \wedge b \in B\}$. Daraus kann die *Dichte* $D := \frac{|E|}{|E_{max}|} = \frac{|E|}{|A| \cdot |B|}$ von G errechnet werden, die angibt, welcher Anteil aller theoretisch möglichen Kanten tatsächlich in E liegt. Da gewichtete und gerichtete Graphen in dieser Arbeit eine untergeordnete Rolle spielen, wird fortan immer vereinfacht *bipartiter Graph* anstelle von ungewichteter ungerichteter bipartiter Graph geschrieben. Häufig interessiert uns die Mächtigkeit der Mengen des Graphen, welche wir fortan mit den Werten $n := |A| + |B| = |V|$ und $m := |E|$ beschreiben.

Weniger formal ausgedrückt ist ein Graph genau dann bipartit, wenn man seine Knoten in zwei Zellen aufteilen kann, sodass für keine Kante beide Knoten in derselben Zelle liegen. Betrachten wir etwa den Graph aller Studenten und Vorlesungen, der zwischen Student und Vorlesung genau dann eine Kante hat, wenn der Student die Vorlesung besucht. Dieser Graph ist offenbar bipartit, da es weder eine Kante zwischen zwei Studenten, noch

zwischen zwei Vorlesungen gibt. Abbildung 2.1a zeigt einen kleinen bipartiten Graphen. Im Folgenden werden wir immer wieder Strukturen sehen, die auf Kantenmengen beruhen, für welche wir festlegen:

2.1.2 Definition (Knotendisjunktheit). Sei E eine beliebige Menge von Kanten, so ist die Menge aller in E enthaltener Knoten definiert als $V(E) := \{v_1 \mid \{v_1, v_2\} \in E\}$. Wir nennen die Kanten in E *(paarweise) knotendisjunkt*, wenn $|V(E)| = 2 \cdot |E|$ gilt. Zwei Kantenmengen E_1 und E_2 heißen *knotendisjunkt*, wenn $V(E_1) \cap V(E_2) = \varnothing$ gilt. Dementsprechend ist eine Menge E^* von Kantenmengen *(paarweise) knotendisjunkt*, wenn alle möglichen Kombinationen $E_1, E_2 \in E^*$ knotendisjunkt sind. ◁

2.2 Matchings

Ein klassisches Problem der Graphentheorie ist das Finden von *Maximum-Matchings*. Dabei wird nach der größten paarweise knotendisjunkten Teilmenge der Kanten eines Graphen gesucht. Da so jeder Knoten nur zu genau einer Kante dieser Teilmenge inzident sein darf, handelt es sich beim Matching-Problem um ein Zuordnungsproblem: Wie viele Knoten können wir jeweils mit genau einem anderen Knoten paaren, sodass kein Knoten in zwei Paarungen vorkommt?

2.2.1 Definition (Matching). Sei $G = (V, E)$ ein ungewichteter ungerichteter Graph. Die Menge $M \subseteq E$ ist ein *Matching*, wenn $\forall m_1, m_2 \in M : m_1 \cap m_2 = \varnothing$ gilt. ◁

Alle in M liegenden Kanten heißen *Matching-Kanten* und alle Knoten in $V(M)$ heißen *gematcht* oder auch *besetzt*. Im Gegensatz dazu nennen wir nicht gematchte Knoten und Kanten *frei*. Sei v ein beliebiger gematchter Knoten, so definieren wir den *Matching-Partner* $m(v)$ von v, für welchen $m(v) = v' \Leftrightarrow \{v, v'\} \in M$ gilt. Für freie Knoten definieren wir hingegen $m(v) := \texttt{null}$. Abhängig von ihrer Größe können Matchings besondere Kriterien erfüllen:

Maximal-Matchings / inklusionsmaximale Matchings mit:

$$\nexists e \in E \setminus M : e \cap V(M) = \varnothing$$

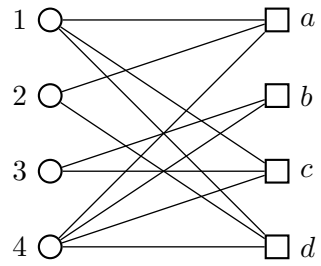
Für diese Matchings ist es nicht möglich weitere Knoten zuzuordnen, ohne bereits erfolgte Zuordnungen rückgängig zu machen. Deswegen werden sie manchmal auch als nicht erweiterbar bezeichnet. Damit dies gilt, darf keine Kante des Graphen zwei freie Knoten beinhalten. Ein Beispiel dafür wird von Abbildung 2.1b gezeigt: Es gibt keine Kante, die man zu M hinzufügen könnte, ohne die Matching-Eigenschaft zu verletzen.
Maximal-Matchings sind in Zeit $\mathcal{O}(m)$ leicht zu finden: Man betrachtet nacheinander alle Kanten des Graphen und fügt jede Kante, die zwei freie Knoten beinhaltet, zum Matching hinzu. Diesen simplen Algorithmus nennen wir Greedy .

Maximum-Matchings / kardinalitätsmaximale Matchings mit:

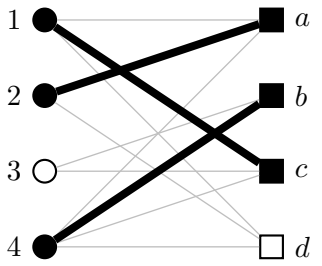
$$\forall M' \subseteq E : |M'| > |M| \implies \exists m_1, m_2 \in M' : m_1 \cap m_2 \neq \varnothing$$

Diese Matchings sind größtmöglich. Es gibt kein weiteres Matching im Graphen, das größer ist als M. Es wird also die maximale Zahl von Knoten zugeordnet. Offenbar folgt aus der Kardinalitätsmaximalität auch die Inklusionsmaximalität. Wenn M alle Knoten zuordnet (es gilt $V(M) = V$), nennen wir es *perfektes Matching* oder sogar *einzigartiges perfektes Matching*, wenn es kein weiteres perfektes Matching gibt. Das in Abbildung 2.1d gezeigte Matching ist perfekt.

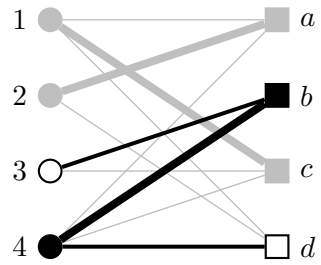
Wollen wir eine approximative Lösung für das Maximum-Matching-Problem berechnen, ist M für $p \in [0, 1]$ genau dann eine *p-Approximation (eines Maximum-Matchings)*, wenn es ein Maximum-Matching M^* gibt, sodass $|M| \geq p \cdot |M^*|$ gilt. Wir sagen dann auch, dass M die *Approximationsgüte* p erreicht. Es ist eine bekannte Tatsache, dass jedes inklusionsmaximale Matching (also auch das von Greedy berechnete) mindestens die Approximationsgüte $\frac{1}{2}$ erreicht.



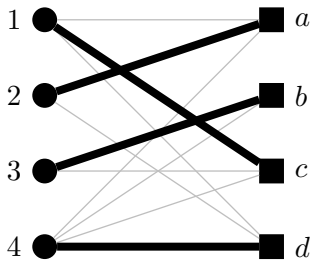
(a) Ein bipartiter Graph $G = (A, B, E)$ mit Zellen $A = \{1, 2, 3, 4\}$ und $B = \{a, b, c, d\}$.



(b) Ein inklusions-maximales Matching $M = \{\{1, c\}, \{2, a\}, \{4, b\}\}$. Die Knoten 3 und d sind frei, alle anderen sind gematcht.



(c) Ein augmentierender Pfad $P = (3, b, 4, d)$.



(d) Nachdem M augmentiert wurde, ist das Matching kardinalitätsmaximal.

Abbildung 2.1: Matchings in einem bipartiten Graphen. Gematchte Knoten sind gefüllt, freie nicht. Matching-Kanten sind dick gedruckt, freie Kanten nicht. Kreise repräsentieren Zeilen und Quadrate repräsentieren Spalten.

2.3 Alternierende Pfade

Viele Matching-Algorithmen beginnen mit einem leeren oder inklusionsmaximalen Matching und versuchen dieses mittels *augmentierender Pfade* zu vergrößern. Dafür definieren wir:

2.3.1 Definition (Weg, Zyklus, Pfad). Sei E eine beliebige Menge von Kanten. Ein *Weg* P der *Länge* $l > 0$ ist definiert als:

$$P := \{\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{l-2}, v_{l-1}\}, \{v_{l-1}, v_l\}\} \subseteq E$$

Wir schreiben verkürzt $P = (v_0, v_1, \ldots, v_l)$ und es gilt offenbar $V(P) = \{v_i \mid 0 \leq i \leq l\}$. Wenn *Start-* und *Endknoten* v_0 und v_l gleich sind, nennen wir P *Zyklus*. Kommt hingegen jeder Knoten aus $V(P)$ nur einmal in P vor, nennen wir P *Pfad*. ◁

Mit einer Sequenz von Knoten meinen wir also immer einen Weg. Dementsprechend verwenden wir auch den Existenzquantor: „Es gibt in E einen Weg zwischen v_1 und v_2“ ist äquivalent zu $\exists(v_1, \ldots, v_2) \subseteq E$. Außerdem können wir durch diese Schreibweise Eigenschaften von Wegen spezifizieren. So ist mit $(v_1, \ldots, v_2, v_3)$ zum Beispiel ein beliebiger Weg gemeint, der bei v_1 beginnt und mit der Kante zwischen v_2 und v_3 endet.

2.3.2 Definition (Kreisfreiheit, Zusammenhang). Sei E eine beliebige Menge von Kanten. Damit E *kreisfrei* ist, darf es keinen Zyklus in E geben: $\forall v \in V(E) : \nexists(v, \ldots, v) \subseteq E$. Damit E *zusammenhängend* ist, muss jeder Knoten jeden anderen Knoten über einen Weg erreichen können: $\forall v_1, v_2 \in V(E) : v_1 \neq v_2 \implies \exists(v_1, \ldots, v_2) \subseteq E$. ◁

2.3.3 Definition (Alternierender / Augmentierender Pfad). Sei $G = (A, B, E)$ ein bipartiter Graph, M ein Matching in G und $P = (v_0, \ldots, v_l)$ ein Pfad in G. Wir nennen P genau dann *M-alternierender Pfad*, wenn für $0 \leq i < l - 1$ gilt: $\{v_i, v_{i+1}\} \in M \Leftrightarrow \{v_{i+1}, v_{i+2}\} \notin M$. Wenn zusätzlich $v_0 \notin V(M) \wedge v_l \notin V(M)$ gilt, bezeichnen wir P als *M-augmentierend* oder auch *M-verbessernd*. Wenn aus dem Kontext ersichtlich wird, welches Matching gemeint ist, schreiben wir kurz *alternierender* bzw. *augmentierender Pfad*. ◁

Demnach ist ein alternierender Pfad eine zusammenhängende Abfolge von Kanten, in der sich Matching- und Nicht-Matching-Kanten abwechseln. Für augmentierende Pfade muss dabei sowohl der Start- als auch der Endknoten frei sein, sodass ein M-augmentierender Pfad P immer eine ungerade Länge $2k+1$ hat und aus genau $|M \cap P| = k$ Matching- und $|P \setminus M| = k+1$ Nicht-Matching-Kanten besteht. Offenbar sind dabei sowohl $M \cap P$ als auch $P \setminus M$ jeweils paarweise knotendisjunkt. Dank dieser Tatsache erlaubt P die Vergrößerung von M. Zunächst entfernt man die k in P liegenden Matching-Kanten aus M, wodurch alle Knoten in $V(P)$ frei werden. Danach können die $k + 1$ anderen Kanten aus P zu M hinzugefügt werden. Abbildung 2.1 zeigt einen beispielhaften Augmentiervorgang. Wir übernehmen:

2.3.4 Lemma (Hopcroft und Karp, [14]). *Sei M ein Matching und P ein M-augmentierender Pfad. $M \triangle P := (M \setminus P) \cup (P \setminus M)$ ist ein Matching mit $|M \triangle P| = |M| + 1$.*

Augmentierende Pfade sind für Matching-Algorithmen ein gutes Hilfsmittel, da ihre Existenz unmittelbar mit der Kardinalitätsmaximalität von Matchings zusammenhängt. So hat Berge 1957 sein berühmtes Lemma bewiesen, welches aussagt:

2.3.5 Lemma (Berge, [5]). *Ein Matching M ist genau dann kardinalitätsmaximal, wenn es keine M-augmentierenden Pfade gibt.*

Es bietet sich bei der Suche nach Maximum-Matchings also an, mit einem leeren oder inklusionsmaximalen Matching zu beginnen und dieses dann schrittweise mit augmentierenden Pfaden zu verbessern. Sobald keine solche Pfade mehr existieren, weiß man, dass das Matching kardinalitätsmaximal ist. Zur Suche nach augmentierenden Pfaden können zum Beispiel Breiten- und Tiefensuche verwendet werden, wodurch sogar mehrere knotendisjunkte Pfade gleichzeitig gefunden werden können. Andere Algorithmen, zum Beispiel Streaming-Algorithmen, bauen schrittweise neue Datenstrukturen auf, bis sich daraus augmentierende Pfade ergeben [15][11]. Dabei werden beispielsweise Bäume benutzt [15], die auch in dieser Arbeit eine zentrale Rolle spielen.

2.4 Alternierende Bäume

2.4.1 Definition (Baum / Wald). Sei $G = (V, E)$ ein ungewichteter ungerichteter Graph. Ein *Baum* $T = (r_T, E_T)$ in G ist definiert durch die kreisfrei zusammenhängenden Kanten $E_T \subseteq E$ sowie die *Wurzel* $r_T \in V(E_T)$. Die Menge der Knoten von T ist durch $V(T) := V(E_T)$ gegeben. Befinden sich im Baum keine Kanten ($|E_T| = 0$), darf die Wurzel ein beliebiger Knoten aus V sein und es gilt $V(T) := \{r_T\}$. Für einen beliebigen Knoten $v \in V(T)$ definieren wir:

1. den *Vorgänger* $p(v) \in V(T)$ mit $\exists(r_T, \ldots, p(v), v) \subseteq E_T$ bzw. $p(r_T) := \texttt{null}$.
2. die *indirekten Vorgänger* $p'(v) \subseteq V(T)$ mit $\forall p' \in p'(v) : \exists(r_T, \ldots, p', \ldots v) \subseteq E_T$.
3. die *Nachfolger* $S(v) \subseteq V(T)$ mit $\forall s \in S : p(s) = v$. Gilt $|S(v)| = 0$, heißt v *Blatt*.
4. die *indirekten Nachfolger* $S'(v) \subseteq V(T)$ mit $\forall s \in S' : v \in p'(s)$.
5. den *in v wurzelnden Teilbaum* $T(v) := (v, E_v)$ mit
   $E_v := \{\{v_1, v_2\} \mid \{v_1, v_2\} \in E_T \wedge v_1, v_2 \in S'(v) \cup \{v\}\} \subseteq E_T$.
6. die *Tiefe* $l(v)$ mit $l(r_T) = 0$ und $l(v) = l(p(v)) + 1$ für $v \neq r_T$.

Für Knoten außerhalb des Baums sind diese Werte hingegen nicht definiert. Seien T_1 und T_2 beliebige Bäume in G, so gilt außerdem:

- Die *Tiefe von* T_1 ist definiert als $l(T_1) := \max(\{l(v) \mid v \in V(T_1)\})$.
- Der Baum T_1 mit $|V(T_1)| = n$ heißt *n-elementig*.
- Die Bäume T_1 und T_2 mit $V(T_1) \cap V(T_2) = \varnothing$ heißen *knotendisjunkt*.

Eine Menge F von Bäumen, die im selben Graphen liegen, heißt genau dann *Wald*, wenn alle beliebigen $T_1, T_2 \in F$ mit $T_1 \neq T_2$ knotendisjunkt sind. Die Knotenmenge von F ist definiert als $V(F) := \{v \mid v \in V \wedge \exists T \in F : v \in V(T)\}$. ◁

In gängiger Literatur werden Bäume und Wälder oft als Graphen oder Subgraphen definiert (siehe z.B. [10]). Wir weichen hier bewusst von dieser Methode ab und wählen eine etwas kompliziertere Definition. Diese wird es uns dafür später erleichtern, Bäume in Wäldern eindeutig zu identifizieren.

2.4.2 Definition (Alternierender Baum / Wald). Sei $G = (A, B, E)$ ein bipartiter Graph, $T = (r_T, E_T)$ ein Baum in G und M ein Matching in G. Wir nennen T einen *M-alternierenden Baum*, wenn folgende Bedingungen gelten:

1. $\forall v \in V_T \setminus \{r_T\} : (r_T, \ldots, v) \subseteq E_T$ ist ein M-alternierender Pfad
2. $\forall v \in V_T : (S(v) = \varnothing) \implies (p(v) = m(v))$
3. $m(r_T) \neq \texttt{null} \implies (p(m(r_T)) = r_T)$

Ein beliebiger Knoten $v \in V_T$ heißt *Gabelknoten* oder auch *Gabel*, wenn $p(v) = m(v)$ gilt. Wir nennen T und die darin enthaltenen Gabelknoten genau dann *gültig*, wenn r_T ein freier Knoten ist. Liegt r_T in A, so sagen wir, dass T in Zelle A wurzelt (analog für B). Ein Wald heißt *M-alternierend*, wenn er ausschließlich aus M-alternierenden Bäumen besteht. Wenn aus dem Kontext ersichtlich wird, auf welches Matching sich ein Baum oder Wald bezieht, schreiben wir verkürzt *alternierender Baum* bzw. *alternierender Wald*. ◁

In einem M-alternierenden Baum ist also jeder Knoten durch einen M-alternierenden Pfad mit der Wurzel verbunden. Dieser Pfad muss bei Blättern mit einer Matching-Kante enden. Wenn die Wurzel von M gematcht ist, dann muss sie im Baum mit ihrem Matching-Partner verbunden sein. Wir stellen fest, dass Nichtgabelknoten in alternierenden Bäumen immer nur genau einen Nachfolger haben dürfen, und zwar ihren Matching-Partner. Ansonsten würde Bedingung 1. in Definition 2.4.2 verletzt. Diese Restriktion gilt nicht für Gabelknoten, was ihnen ihren Namen verleiht: Sie dürfen beliebig viele Nachfolger im Baum haben. Dadurch ergibt sich eine charakteristische Struktur, in der sich „Ebenen“ von Matching- und Nicht-Matching-Kanten abwechseln. Ein Beispiel dafür ist in Abbildung 2.2 gegeben. Gültige alternierende Bäume haben weitere besondere Eigenschaften. Weil die Wurzel ein Gabelknoten ist, muss in ihnen die Tiefe aller Gabelknoten gerade sein. Außerdem beginnt in ihnen jeder Pfad zwischen Wurzel und Gabelknoten an der Wurzel mit einer

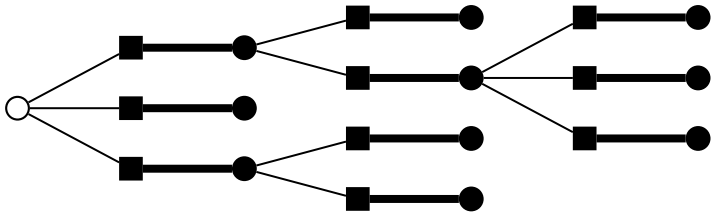


Abbildung 2.2: Ein gültiger alternierender Baum. Matching-Kanten sind dick gedruckt, gematchte Knoten sind gefüllt. Zeilen sind Kreise, Spalten sind Quadrate. Nur die Wurzel ist frei.

Nicht-Matching-Kante und endet am Gabelknoten mit einer Matching-Kante. Deswegen ergibt sich ein augmentierender Pfad, wenn es eine zusätzliche Kante zwischen einem Gabelknoten und einem freien Knoten gibt. Jack Edmonds zeigte bereits 1965:

2.4.3 Lemma (Edmonds, [10]). *Sei $T = (r, E)$ ein gültiger M-alternierender Baum, a ein Gabelknoten in $V(T)$ und b ein freier Knoten. Der Pfad $(r, \ldots, a, b) \subseteq E \cup \{\{a, b\}\}$ ist M-augmentierend.*

Des weiteren bildet jeder freie Knoten v einen 1-elementigen und gültigen alternierenden Baum $(v, \varnothing)$: Da es keine anderen Knoten im Baum gibt, muss Bedingung 1. aus Definition 2.4.2 gelten. Außerdem ist $m(v) = p(v) =$ `null`, weswegen auch 2. und 3. erfüllt sein müssen. Möchte man also schrittweise alternierende Bäume aufbauen, kann einfach mit diesen 1-elementigen Bäumen begonnen werden.

2.5 Streaming Modelle

Traditionelle Algorithmen erwarten, dass alle benötigten Daten unmittelbar verfügbar sind, sodass sie im Arbeitsspeicher liegen müssen, um Random-Access zu gewähren [2]. Streaming-Algorithmen stellen keine solche Forderungen: Die Daten werden direkt beim Lesen vom Speichermedium (Festplatte, Flash-Speicher, optische Speichermedien etc.) verarbeitet, wodurch sich zwei große Vorteile ergeben. Einerseits kann so auch mit Datenmengen umgegangen werden, die viel zu groß für den Arbeitsspeicher sind, andererseits kann sich auch für kleiner Datensätze eine direkte Auswertung lohnen. Warum sollte man schließlich erst alle Informationen in den Arbeitsspeicher laden und anschließend die eigentliche Berechnung ausführen, wenn das Lesen vom Speichermedium und die Berechnung auch parallel stattfinden können. Dieses Vorgehen ist auch für Graphen möglich, indem der Graph als Stream betrachtet wird:

2.5.1 Definition (Kanten-Stream). Sei $G = (V, E)$ ein beliebiger Graph. Die Sequenz $e_1, e_2, \ldots, e_m$ mit $E = \{e_i \mid 1 \leq i \leq m\}$ ist ein Kanten-Stream von G. ◁

In der Realität ist die Reihenfolge der Kanten im Stream durch die vorliegenden Daten festgelegt. Diese müssen als sequentieller Stream gelesen werden, also in der Reihenfolge, in

der sie physisch auf dem Speichermedium hinterlegt sind. Dadurch kann die bestmögliche Leistung von Festplatten erreicht werden, da diese beim sequentiellen Lesen am schnellsten sind[1]. In manchen Streaming Modellen wird angenommen, dass alle Kanten, die zu einem Knoten adjazent sind, direkt nacheinander im Stream vorkommen [20]. Wir gehen hier nicht von dieser Annahme aus, obwohl sie für echte Graphen häufig von Natur aus gilt, etwa wenn der Graph auf dem Speichermedium in Adjazenzlistendarstellung hinterlegt ist.

2.5.1 Das Semi-Streaming Modell

Es gibt verschiedene Modelle, die beschreiben wie ein Kanten-Stream verarbeitet werden kann. Dazu gehört auch das 2004 von Feigenbaum et. al. formalisierte Semi-Streaming Modell [12]:

2.5.2 Definition (Semi-Streaming-Graphalgorithmus). Ein *Semi-Streaming-Graphalgorithmus* rechnet auf einem Graphen mit $\mathcal{O}(n \cdot \text{polylog}(n))$ Bits im Arbeitsspeicher. Dazu wird der Graph als sequentieller Kanten-Stream betrachtet, welcher wenn nötig mehrfach gelesen wird. Wir bezeichnen das einmalige Lesen des gesamten Streams als *Pass* und die Anzahl aller Passes vor der Terminierung des Algorithmus als *Pass-Count*. ◁

Diese Speicherplatzrestriktion erlaubt es uns nicht, den gesamten Graphen im Arbeitsspeicher zu halten. Wir können uns zwar problemlos alle Knoten und möglicherweise vorhandene Label merken, allerdings nur einen kleinen, von n abhängigen Teil aller Kanten. Deswegen sind vor allem Graphen von hoher Dichte eine Herausforderung für Semi-Streaming-Algorithmen. Außerdem verhindert der geringe Speicherplatz die effiziente Umsetzung einiger grundlegender Algorithmen, die außerhalb von Streaming Modellen als Hilfsmittel zur Lösung vieler Probleme verwendet werden. Zum Beispiel konnte gezeigt werden, dass Breitensuche im Semi-Streaming Modell praktisch nicht zu realisieren ist: Zum konstruieren der ersten l Ebenen eines Breitensuchbaums sind entweder $\Omega(l)$ Passes oder $\Omega(n^{1+\frac{1}{l}})$ Bits Speicherplatz erforderlich [20]. Letzteres widerspricht den Rahmenbedingungen des Semi-Streaming Modells, und auch $\Omega(l)$ Passes pro Breitensuche sind in der Regel nicht akzeptabel. Wenn beispielsweise eine auf Breitensuche basierende Implementierung von HopcroftKarp ausgeführt werden soll, müssen $\mathcal{O}(\sqrt{n})$ Breitensuchbäume konstruiert werden [14]. Deswegen sind zum effizienten Streaming neue Ansätze nötig.

2.5.2 Weitere Streaming Modelle

Es seien kurz zwei weitere Streaming Modelle erwähnt, welche ebenfalls eine Speicherbeschränkung haben und den Daten-Stream sequentiell lesen. Eine Zusammenfassung verschiedener Modelle und Konzepte ist in [20] zu finden.

[1] http://www.storagereview.com/seagate_enterprise_capacity_6tb_35_hdd_review_v4

Das *W-Stream Modell* [9] erlaubt mehrere Passes über den Stream und gewährt Algorithmen das Schreiben von zusätzlichen Informationen in den Stream. Die Informationen tauchen in späteren Passes wieder auf und können dann verwendet werden. Durch diese Schreibmöglichkeit konnte zum Beispiel ein erster Streaming-Algorithmus entwickelt werden, der kürzeste Pfade in gerichteten Graphen findet. In manchen Fällen wird durch das W-Stream Modell außerdem die Simulation paralleler Algorithmen möglich [8].

Das *Stream-Sort Modell* [1] erlaubt es, sogenannte *Sorting-Passes* zwischen den regulären Passes auszuführen, wodurch die Reihenfolge der Kanten im Stream verändert werden kann. Zum Sortieren können zusätzliche Informationen in den Stream geschrieben werden. Tatsächlich ist das W-Stream Modell eine Modifikation des Stream-Sort Modells, bei der nur die Schreiboperationen, nicht jedoch das Sortieren übernommen wurde.

Wir befassen uns in dieser Arbeit ausschließlich mit dem Semi-Streaming Modell, welches durch das Fehlen von Schreiboperationen offenbar wesentlich restriktiver ist als die W-Stream und Stream-Sort Modelle. Dadurch ist es in vielen Szenarien anwendbar und es wurden bereits zahlreiche Semi-Streaming-Algorithmen entwickelt. Eine gute Übersicht über bisherige Forschungsergebnisse wird in [21] gegeben.

Kapitel 3

Verwandte Arbeiten

Da wir nun mit den formalen Grundlagen vertraut sind, wollen wir uns mit einigen Publikationen befassen, die Matchings sowohl innerhalb als auch außerhalb von Streaming Modellen thematisieren. Im letzten Kapitel haben wir bereits Berges Lemma [5] kennengelernt, welches die Basis aller Matching-Algorithmen bildet, die auf augmentierenden Pfaden beruhen. Ein ebenso revolutionäres Werk von Jack Edmonds wurde 1965 veröffentlicht [10]. In diesem werden erstmals alternierende Bäume und Blüten definiert. Letztere sind Kreise ungerader Länge, welche die Berechnung von Maximum-Matchings in nicht bipartiten Graphen erschweren. Edmonds Idee, Blüten durch Kontraktion zu Knoten zu machen, führte zu einem Matching-Algorithmus für allgemeine Graphen, der mit einer Laufzeit von $\mathcal{O}(n^4)$ als erster seiner Art in Polynomialzeit terminiert.
Für bipartite Graphen sind zahlreiche Matching-Algorithmen bekannt. Ist der vorliegende Graph von hoher Dichte, erzielt der Algorithmus von Alt et al. mit $\mathcal{O}(n^{1.5} \cdot \sqrt{\frac{m}{\log n}})$ die beste bekannte Laufzeitschranke [3]. Auf dünneren bipartiten Graphen hat hingegen der Algorithmus von Hopcroft und Karp mit $\mathcal{O}(\sqrt{n} \cdot m)$ die beste Worst-Case-Laufzeit. Dessen Grundgerüst ist in Algorithmus 1 zu sehen. In ihrer 1973 erschienenen Publikation konnten Hopcroft und Karp zeigen, dass die Schleife höchstens $\mathcal{O}(\sqrt{n})$ mal durchlaufen wird, wobei bekannt ist, dass ein Runde für bipartite Graphen durch Breiten- und Tiefensuche in Laufzeit $\mathcal{O}(m)$ realisierbar ist. Später haben Micali und Vazirani eine Möglichkeit gefunden, selbst für nicht bipartite Graphen die Rundenzeit auf $\mathcal{O}(m)$ zu minimieren [22], sodass die schnellste bekannte Lösung für das Maximum-Matching-Problem in allgemeinen Graphen ebenfalls eine Worst-Case-Laufzeit von $\mathcal{O}(\sqrt{n} \cdot m)$ hat.

Für das Semi-Streaming Modell oder sogar Streaming allgemein sind Matching-Algorithmen weniger gut erforscht. Immerhin können wir in ungewichteten Graphen problemlos eine $\frac{1}{2}$-Approximation an ein Maximum-Matching erreichen. Denn da der Greedy Algorithmus genau einmal über alle Kanten iteriert und dabei nur $\mathcal{O}(n)$ Bits Speicherplatz belegt, kann er in einem einzelnen Pass im Semi-Streaming Modell ausgeführt werden. Soll eine bessere Approximationsgüte erreicht werden, kann ein randomisierter Algorith-

Algorithmus 1 HopcroftKarp [14]

Eingabe: Allgemeiner Graph $G = (V, E)$
Ausgabe: Maximum-Matching M

1: $M \leftarrow \varnothing$
2: **repeat**
3: $P \leftarrow \{P_1, \cdots, P_k\}$, sodass P eine inklusionsmaximale Menge kürzester knotendisjunkter augmentierender Pfade ist
4: $M \leftarrow M \triangle (P_1 \cup \cdots \cup P_k)$
5: **until** $P = \varnothing$
6: **return** M

mus von McGregor [19] verwendet werden. Dieser berechnet eine $\frac{1}{1+k}$-Approximation in einer von k abhängigen aber konstanten Anzahl an Passes. In [11] wird gezeigt, dass es bipartite Graphen gibt, für welche McGregors Algorithmus $\Omega(\frac{1}{k})^{\Omega(\frac{1}{k})}$ Passes benötigt.

3.1 DAP-Algorithmen

Pfadbasierte DAP-Approximation [11]

Ein erster deterministischer, auf augmentierenden Pfaden beruhender Approximationsalgorithmus für Matchings in bipartiten Graphen im Semi-Streaming Modell wurde 2011 publiziert [11]. Dieser kommt mit $\mathcal{O}(\frac{1}{k^5})$ Passes über den Stream aus, wenn die Approximationsgüte $\frac{k}{k+1}$ erreicht werden soll. Die Vorgehensweise ähnelt traditionellen Nicht-Streaming-Algorithmen, in welchen wiederholt eine Menge von paarweise knotendisjunkten augmentierenden Pfaden gesucht wird, um das Matching zu verbessern. Sei $\mathcal{D}$ eine solche Menge, so nennen wir sie *(λ_1, λ_2) DAP-Menge* [15], wenn folgende Bedingungen gelten:

1. Alle Pfade in $\mathcal{D}$ sind M-augmentierende Pfade mit höchstens $2\lambda_2 + 1$ Kanten.
2. Alle Pfade in $\mathcal{D}$ sind paarweise knotendisjunkt.
3. Es gibt keinen M-augmentierenden Pfad mit höchstens $2\lambda_1 + 1$ Kanten, den man zu $\mathcal{D}$ hinzufügen könnte, ohne 2. zu verletzen.

Daraus ergibt sich der in Algorithmus 2 gegebene Rahmen für DAP-Algorithmen, der offenbar große Ähnlichkeit mit HopcroftKarp hat. Dabei werden λ_1 und λ_2 in Abhängigkeit von der gewünschten Approximationsgüte gewählt und δ wird aus ihnen berechnet [15]. Um die DAP-Menge aufzubauen (Algorithmus 2, Z.4), werden Passes über den Stream der Kanten ausgeführt.

Algorithmus 2 DapApprox [15]

Eingabe: Allgemeiner Graph $G = (V, E)$
Ausgabe: Maximum-Matching M

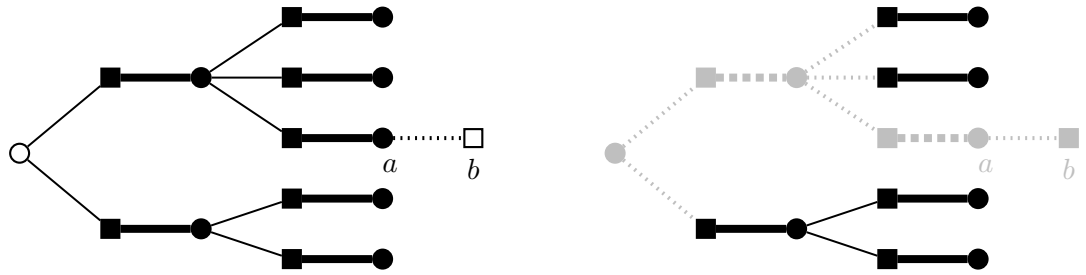
1: $M \leftarrow$ beliebiges Maximal-Matching
2: **repeat**
3: $\quad c \leftarrow |M|$
4: $\quad D \leftarrow \{P_1, \cdots, P_k\}$, sodass D eine (λ_1, λ_2)-DAP-Menge ist
5: $\quad M \leftarrow M \triangle (P_1 \cup \cdots \cup P_k)$
6: **until** $|D| \leq \delta \cdot c$
7: **return** M

In der pfadbasierten Variante gilt $\lambda_1 = \lambda_2 = k$ und zum Finden der DAP-Menge $\mathcal{D}$ werden knotendisjunkte alternierende Pfade konstruiert [11][15]. Für jede gelesene Stream-Kante wird überprüft, ob sie an einen bestehenden alternierenden Pfad angehängt werden kann, ohne dass dieser die Länge $2k + 1$ überschreitet. Ergibt sich dadurch ein augmentierender Pfad, wird er zu $\mathcal{D}$ hinzugefügt und alle in ihm enthaltenen Knoten dürfen solange nicht mehr zu anderen Pfaden hinzugefügt werden, bis augmentiert wurde. Durch geschicktes Backtracking wird verhindert, dass alternierende Pfade nach dem Erreichen der Maximallänge feststecken, wenn sie nicht zu einem augmentierenden Pfad vervollständigt werden können. Das Matching wird mit allen in $\mathcal{D}$ enthaltenen Pfaden augmentiert, sobald $\mathcal{D}$ die DAP-Eigenschaften erfüllt (Algorithmus 2, Z.5). Abhängig davon, wie viele augmentierenden Pfade in $\mathcal{D}$ enthalten waren, wird entweder erneut nach einer DAP-Menge gesucht, oder (wenn nur noch sehr wenige Pfade gefunden wurden) der Algorithmus terminiert und das Matching wird zurückgegeben (Algorithmus 2, Z.6–7).
Obwohl die theoretische Pass-Schranke von $\mathcal{O}(\frac{1}{k^5})$ gegenüber vorherigen Algorithmen (z.B. [19]) überlegen ist, kam es in einer experimentellen Evaluierung zu sehr hohen Pass-Counts [15]. Selbst bei einer Approximationsgüte von 90% lag die durchschnittliche Passzahl für die meisten der untersuchten Graphklassen im dreistelligen Bereich, wobei es einige Instanzen gab, die sogar mehr als 30 000 Passes benötigten.

Baumbasierte DAP-Approximation [15]

Auf der Suche nach einer praxistauglicheren Variante konnte Kliemann in [15] den pfadbasierten Approximationsalgorithmus deutlich verbessern. Gleich geblieben ist in der neuen Version der äußere Rahmen des Algorithmus: Es wird nach wie vor DapApprox verwendet, wobei immer noch $\lambda_1 = k$ gilt, durch $\lambda_2 \in \{k, \ldots, 2k - 1\}$ jedoch auch längere augmentierende Pfade in $\mathcal{D}$ erlaubt werden können.
Neu ist, dass nicht mehr alternierende Pfade aufgebaut werden, um augmentierende Pfade

zu finden. Stattdessen wird ein Wald aus alternierenden Bäumen konstruiert, weswegen wir den Algorithmus im Folgenden als DapTrees bezeichnen. Für $G = (A, B, E)$ ist der initiale Wald durch $\{(a, \varnothing) \mid a \in A\}$ gegeben. Nun betrachten wir jede Kante im Stream und entscheiden, ob eine Operation möglich ist. Ein augmentierender Pfad P wird genau dann gefunden und zu $\mathcal{D}$ hinzugefügt, wenn eine Kante $\{a, b\}$ gelesen wird, für welche a ein Gabelknoten in einem gültigen Baum und b ein freier Knoten ist (Lemma 2.4.3). Ist dies der Fall, werden alle Knoten des Pfades aus dem alternierenden Baum entfernt und dürfen erst wieder zu anderen Bäumen hinzugefügt werden, nachdem augmentiert wurde. Dadurch entstehen ungültige alternierende Bäume, die später wieder an gültige Bäume angehängt werden können (Abbildung 3.1).

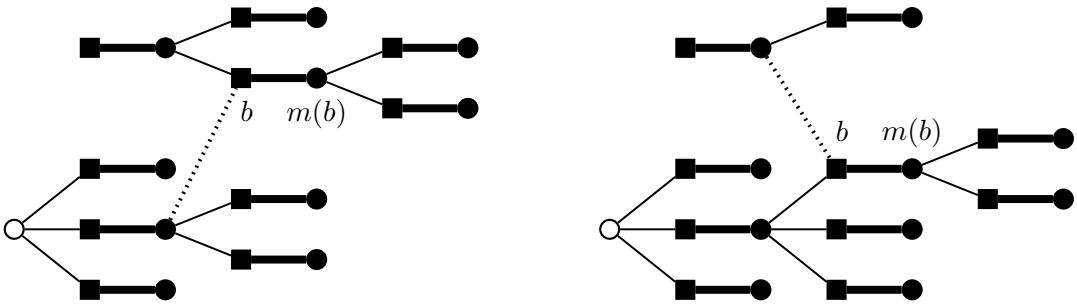


(a) vorher **(b)** nachher

Abbildung 3.1: Ein augmentierender Pfad wird durch einen alternierenden Baum entdeckt. Durch das Augmentieren des Matchings zerfällt der Baum in einen Wald aus drei Bäumen. Matching-Kanten sind dick gedruckt, gematchte Knoten sind gefüllt. Durchgezogene Kanten gehören zu einem alternierenden Baum, gepunktete Kanten nicht. Zeilen sind Kreise, Spalten sind Quadrate.

Insbesondere wird im ersten Pass jede Kante zwischen einer gültigen Wurzel aus A und einem freien Knoten aus B als augmentierender Pfad erkannt, sodass die erste gefundene DAP-Menge bereits ein inklusionsmaximales Matching und somit eine $\frac{1}{2}$-Approximation darstellt.

Erlaubt eine Stream-Kante $\{a, b\}$ keinen Augmentiervorgang und ist a ein Gabelknoten, ist oft eine Erweiterung des a enthaltenden Baums T_a möglich. Ist b in keinem Baum enthalten, wird T_a um die beiden Kanten $\{a, b\}$ und $\{b, m(b)\}$ erweitert. Ist b in einem nicht gültigen Baum enthalten, oder ist die Tiefe von b in seinem gültigen Baum höher als die Tiefe von a in T_a, wird der gesamte in b wurzelnde Teilbaum $T(b)$ vom b enthaltenden Baum abgetrennt und an a angehängt (siehe Abbildung 3.2). Nachdem $\mathcal{D}$ die Kriterien einer DAP-Menge erfüllt und das Matching augmentiert wird, bleibt der alternierende Wald bestehen. Er kann für die Konstruktion der nächsten DAP-Menge wiederverwendet werden.



(a) vorher **(b)** nachher

Abbildung 3.2: Ein gültiger alternierender Baum wird mit einem Teilbaum eines nicht gültigen alternierenden Baums erweitert. Matching-Kanten sind dick gedruckt, gematchte Knoten sind gefüllt. Durchgezogene Kanten gehören zu einem alternierenden Baum, gepunktete Kanten nicht. Zeilen sind Kreise, Spalten sind Quadrate.

Im Vergleich zur pfadbasierten DAP-Approximation ist DapTrees wesentlich effizienter. In einer experimentellen Auswertung [15] wurden nie mehr als 100 Passes benötigt, um eine 90%-Approximation eines Maximum-Matchings zu berechnen.

3.2 Strong-Spanning-Tree-Algorithmus

Mit den DAP-Algorithmen haben wir einen Ansatz kennengelernt, Matchings im Semi-Streaming Modell schrittweise zu approximieren. Ein nicht weniger interessanter Nicht-Streaming-Algorithmus zur exakten Berechnung von Maximum-Matchings in bipartiten Graphen wurde von Balinski und Gonzalez entwickelt [4]. Bis dato unterscheidet dieser sich von einem Großteil aller anderen Lösungen. Das besondere ist, dass vollständig auf alternierende Strukturen verzichtet wird und stattdessen ein einzelner *Strong-Spanning-Tree* Verwendung findet. Dieser ist definiert als:

3.2.1 Definition (Strong-Spanning-Tree). Sei $G = (A, B, V)$ ein bipartiter Graph. Ein Baum $T = (r_T, E_T)$ in G mit $r_T \in A$ heißt *Strong-Spanning-Tree (SST)*, wenn gilt:

1. $V(T) = A \cup B$
2. $\forall b \in B : S(b) = \varnothing \implies p(b) = r_T$ ◁

Ein SST muss also immer aus allen Knoten des zugehörigen Graphen bestehen und Spalten, die Blätter sind, müssen direkt an der Wurzel hängen. Es wird nun o.B.d.A angenommen, dass immer $|A| > |B|$ gilt und dass G zusammenhängend ist. Für nicht zusammenhängende Graphen könnte der später folgende Algorithmus einfach auf jeder Zusammenhangskomponente separat ausgeführt werden. Um einen SST in einem beliebigen Graphen G zu konstruieren, sind noch ein paar kleinere Modifikationen nötig, denn sonst wäre es nicht

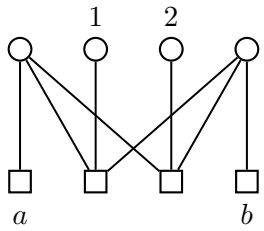
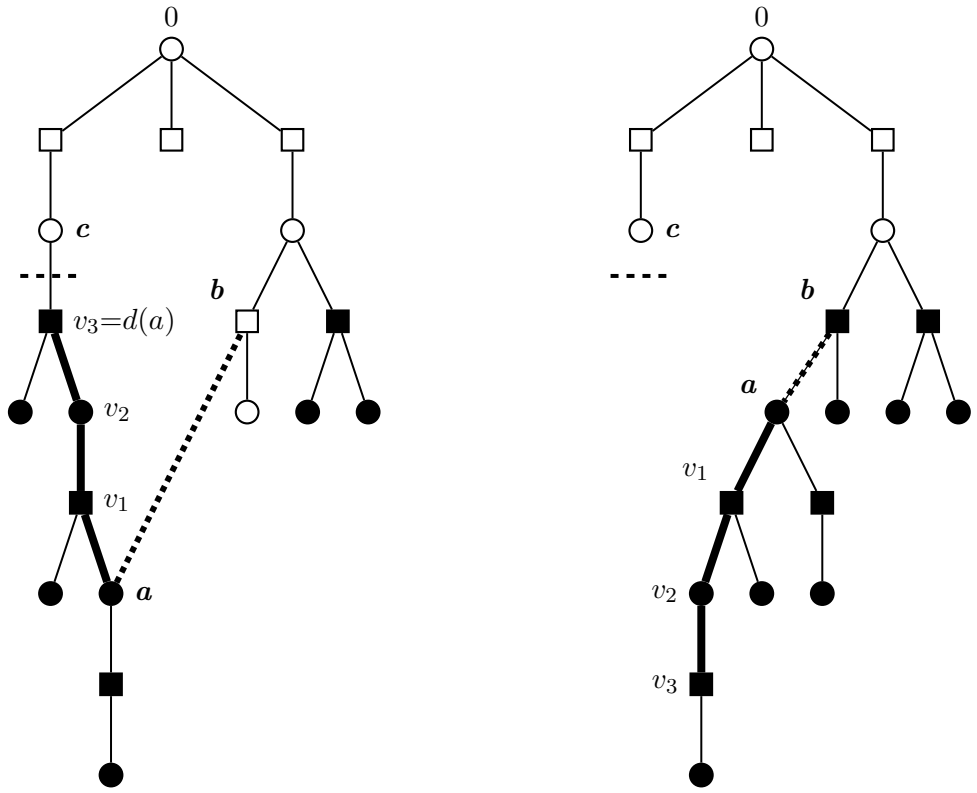


Abbildung 3.3: Ein Graph, für den es keinen SST gibt. Zeilen sind Kreise, Spalten sind Quadrate.

gewährt, dass es überhaupt einen passenden SST gibt. Betrachten wir etwa den in Abbildung 3.3 gegebenen Graphen, so kann kein SST gefunden werden. Da die Spalten a und b jeweils nur eine einzige adjazente Kante haben und damit an verschiedene Zeilen angebunden sind, kann keine Zeile als Wurzel eines SSTs dienen. Denn dann wären sowohl a als auch b Blatt, und mindestens einer der beiden Knoten würde nicht an der Wurzel hängen. Auf die gleiche Weise kann keine Spalte als Wurzel gewählt werden, denn dann wären die Knoten 1 und 2 Blätter, von denen mindestens eines nicht direkt an der Wurzel hängt. Aus diesem Grund wird für den Strong-Spanning-Tree-Algorithmus niemals der eigentliche Graph $G = (A, B, E)$ betrachtet, sondert immer ein Hilfsgraph $G^h = (A^h, B, E^h)$ mit $A^h = A \cup \{0\}$ und $E^h = E \cup \{\{0, b\} \mid b \in B\}$. Mit dem neuen Hilfsknoten 0 kommt die ideale Wurzel hinzu, da im SST jede Spalte direkt an ihr angehängt werden kann. Da G^h zusammenhängend ist, kann nun auch jede Zeile mit einer beliebigen zu ihr adjazenten Kante angehängt werden, sodass man einen SST in G^h erhält.
Jeder SST in G^h induziert mindestens ein Matching in G, da jede Spalte mit einem beliebigen Nachfolger gematcht werden kann. Deswegen nennen wir alle Spalten, die Blätter sind, *frei*, da sie von keinem vom SST induzierten Matching besetzt werden. Spalten können außerdem weitere besondere Stellungen im Baum haben, die von ihrer Nachfolgerzahl abhängig sind. Sei T ein beliebiger in 0 wurzelnder SST in G^h. Hat eine Spalte $b \in B$ mehr als einen Nachfolger in T, heißt sie *Kandidat* oder sogar *Dominante*, falls es auf dem Pfad zwischen b und 0 in T keinen weiteren Kandidaten gibt. Ein beliebiger Knoten $v \in A \cup B$ wird genau dann *dominiert*, wenn sich eine Dominante $d(v)$ auf dem in T liegenden Pfad zwischen v und der Wurzel 0 befindet. Insbesondere dominiert jede Dominante auch sich selbst [4].

Beim Vergrößern des von T induzierten Matchings spielen dominierte Zeilen die wichtigste Rolle. Sie erlauben die sogenannte *Pivot*-Operation, bei welcher ein Teilbaum von T gelöst und an einer anderen Stelle wieder an T angehängt wird. Eine *Pivot-Operation an der Kante* $\{a, b\}$ kann ausgeführt werden, wenn a eine dominierte Zeile ist und b nicht ebenfalls von $d(a)$ dominiert wird. In diesem Fall wird der gesamte Teilbaum, der in $d(a)$ wurzelt, aus T entfernt und mit der Kante $\{a, b\}$ wieder angefügt. Dabei wird der Pfad zwischen



(a) vorher **(b)** nachher

Abbildung 3.4: Ein Pivot an der Kante $\{a, b\}$ wird ausgeführt. Die Reihenfolge der Knoten im dick gedruckten Pfad zwischen a und $v_3 = d(a)$ wird dadurch umgekehrt. Gefüllte Knoten werden dominiert. Zeilen sind Kreise, Spalten sind Quadrate.

a und $d(a)$ sozusagen umgedreht, sodass für darauf liegende Knoten die Vorgänger zu Nachfolgern werden und die Nachfolger, die ebenfalls auf dem Pfad liegen, zu Vorgängern werden. Abbildung 3.4 veranschaulicht die Prozedur. Bezeichne $p(v)$ den Vorgänger von v und $S(v)$ die Menge aller Nachfolger von v im SST (siehe Definition 2.4.1). Für die in der Abbildung gezeigte Pivot-Operation müssen folgende Zuweisungen ausgeführt werden:

$$
\begin{array}{lll}
p(a) \leftarrow b; & \text{———} & S(b) \leftarrow S(b) \cup \{a\}; \\
p(v_1) \leftarrow a; & S(v_1) \leftarrow S(v_1) \setminus \{a\}; & S(a) \leftarrow S(a) \cup \{v_1\}; \\
p(v_2) \leftarrow v_1; & S(v_2) \leftarrow S(v_2) \setminus \{v_1\}; & S(v_1) \leftarrow S(v_1) \cup \{v_2\}; \\
p(v_3) \leftarrow v_2; & S(v_3) \leftarrow S(v_3) \setminus \{v_2\}; & S(v_2) \leftarrow S(v_2) \cup \{v_3\}; \\
\text{———} & S(c) \leftarrow S(c) \setminus \{v_3\}; & \text{———}
\end{array}
$$

Alle Kanten, die nicht auf dem Pfad zwischen a und der Dominante $d(a) = v_3$ liegen, bleiben unverändert. Nur die Verbindung zwischen c und $d(a)$ wird getrennt. Dabei geht $d(a)$ ein Nachfolger verloren, da v_2 der neue Vorgänger wird. Da wir aber wissen, dass je-

de Dominante mindestens zwei Nachfolger hat, kann $d(a)$ auch nach der Pivot-Operation nicht zu einem Blatt werden und die Eigenschaften des SST bleiben unverletzt.
Auf der Suche nach einem Maximum-Matching ist eine Pivot-Operation an $\{a, b\}$ nützlich, wenn b nicht dominiert wird: Ist b kein Blatt, so wird b nach der Pivot-Operation automatisch zur Dominante, da a als neuer Nachfolger angehängt wird. Dadurch wird auch der zuvor einzige Nachfolger von b dominiert (siehe Abbildung 3.4) und sukzessiv alle seine indirekten Nachfolger, wodurch sich potentiell neue Pivotmöglichkeiten ergeben. Ist b hingegen ein Blatt, erhält es seinen ersten Nachfolger, wodurch die Größe des vom SST induzierten Matchings steigt.

3.2.1 Algorithmus von Balinski & Gonzalez

Balinski und Gonzalez konstruieren in ihrem Strong-Spanning-Tree-Algorithmus einen SST im Hilfsgraphen und verwenden die zuvor beschriebene Pivot-Operation, um Maxium-Matchings zu berechnen [4]. Dabei werden alle Dominanten des SSTs nacheinander abgearbeitet, wobei Knoten als bearbeitet markiert werden können. Im Wesentlichen besteht der Algorithmus aus folgenden Schritten:

1. **Konstruktion des initialen SSTs.** Hänge alle Spalten an die Wurzel 0 und wähle für jede Zeile eine beliebige adjazente Kante, welche sie mit dem SST verbindet.

2. **Überprüfen der Optimalität.** Gib ein beliebiges vom SST induziertes Matching als Ergebnis zurück, wenn es keine freien Spalten mehr gibt oder alle Dominanten als bearbeitet markiert wurden. Ansonsten wähle eine beliebige noch nicht bearbeitete Dominante d und fahre mit 3. fort.

3. **Pivot-Operation für Dominante *d*.** Suche eine Kante $\{a, b\}$ mit folgenden Eigenschaften: Zeile a ist noch nicht als bearbeitet markiert und wird von d dominiert; Spalte b ist noch nicht als bearbeitet markiert und wird nicht von d dominiert. Falls es eine solche Kante gibt, dann führe eine Pivot-Operation an ihr durch. War b zuvor frei, so fahre mit 2. fort. Ansonsten fahre mit 3. für $d \leftarrow b$ fort.
   Ist keine Pivot-Operation möglich, markiere d und alle seine indirekten Nachfolger als bearbeitet und fahre mit 2. fort.

Es konnte gezeigt werden, dass der vom Algorithmus konstruierte SST nach folgender Definition *abgeschlossen* ist [4]:

3.2.2 Definition. Ein SST heißt *abgeschlossen*, wenn er keine Dominanten hat, oder wenn er keine freien Spalten hat, oder wenn es keine Kante $\{a, b\} \in E$ gibt, für die nur a, nicht aber b dominiert wird. ◁

Deswegen ist das als Ergebnis zurückgegebene Matching kardinalitätsmaximal:

3.2.3 Theorem (Balinski & Gonzalez, [4]). *Es sei T ein abgeschlossener SST in G^h. Jedes von T induzierte Matching ist kardinalitätsmaximal.*

Offenbar verzichtet der Algorithmus von Balinski und Gonzalez auf die gängigen Methoden, die in den meisten Arbeiten zum Thema Matchings verwendet werden. Auch ohne die direkte Verwendung von alternierenden und augmentierenden Pfaden wird eine akzeptable theoretische Laufzeitschranke erreicht:

3.2.4 Theorem (Balinski & Gonzalez, [4]). *Der Strong-Spanning-Tree-Algorithmus berechnet ein Maximum-Matching in Zeit $\mathcal{O}(n \cdot m)$.*

Später konnte gezeigt werden, dass augmentierende Pfade und SSTs gewissermaßen miteinander in Verbindung stehen [13]. In diesem Rahmen wurde auch eine Modifikation des Algorithmus von Balinski und Gonzalez vorgestellt, die dank einer Laufzeit von $\mathcal{O}(\sqrt{n} \cdot m)$ mit dem Algorithmus von Hopcroft und Karp konkurrieren kann. Im nächsten Kapitel versuchen wir, eine Streaming-Adaption des Strong-Spanning-Tree-Algorithmus zu entwickeln.

Kapitel 4

Streaming-Ansatz mit Strong-Spanning-Trees

Um einen für das Semi-Streaming Modell geeigneten Matching-Algorithmus zu entwickeln, liegt die Überlegung nahe, einen traditionellen Algorithmus so umzurüsten, dass er die Anforderungen des Modells erfüllt. Dafür gibt es nur eine spärliche Auswahl an Kandidaten, da bereits alle Algorithmen ausscheiden, die sich auf Breiten- oder Tiefensuche verlassen. Außerdem sind auch rundenbasierte Verfahren nicht unbedingt gut geeignet, da sich die Runden nur schwer oder gar nicht auf Stream-Passes abbilden lassen. Der in Abschnitt 3.2 vorgestellte Algorithmus von Balinski und Gonzales erfüllt alle diese Kriterien. Da für den gesamten Graphen lediglich ein einziger Spannbaum aufrecht erhalten werden muss, kann auch die Speicherplatzbeschränkung des Semi-Streaming Modells von $\mathcal{O}(n \cdot \text{polylog}(n))$ eingehalten werden.

Ein Aspekt des Originalalgorithmus, der sehr leicht übernommen werden kann, ist die Konstruktion des initialen SST. Nach wie vor können einfach alle Spalten an den als Wurzel fungierenden Hilfsknoten 0 gehängt werden. Im ersten Stream-Pass sind dann alle Zeilen jeweils durch die erstbeste Kante zum Baum hinzuzufügen. Auch die Pivot-Operation bleibt zum Nicht-Streaming-Algorithmus identisch. Allerdings müssen vor allem bezüglich der Ausführungsreihenfolge Kompromisse gemacht werden. Ohne Random-Access auf den Graphen können wir nicht einfach nacheinander alle Dominanten abarbeiten. Um herauszufinden, ob für eine Dominante eine Pivot-Operation möglich ist, brauchen wir im schlimmsten Fall schließlich einen ganzen Pass, da wir zu jeder von ihr dominierten Zeile alle adjazenten Kanten überprüfen müssen. Deswegen scheint es wesentlich sinnvoller zu sein, für jede einzelne Stream-Kante zu kontrollieren, ob eine Pivot-Operation an ihr ausgeführt werden kann und (wenn das der Fall ist) diese direkt auszuführen.

Algorithmus 3 SstStreaming

Eingabe: Bipartiter Graph $G = (A, B, E)$

Ausgabe: Maximum-Matching M

1: $T \leftarrow (0, \{\{0, b\} \mid b \in B\})$

2: **for** $\{a, b\} \in E$ **where** $a \in A \wedge b \in B$ **do**
3: **if** $p(a) =$ `null` **then**
4: $p(a) \leftarrow b$
5: $S(b) \leftarrow S(b) \cup a$
6: **end if**
7: **end for**

8: $changed \leftarrow \top$
9: **while** $changed = \top$ **do**
10: $changed \leftarrow \bot$
11: **for** $\{a, b\} \in E$ **where** $a \in A \wedge b \in B$ **do**

12: **if** a wird in T von $d(a)$ dominiert $\wedge$
b wird in T nicht von $d(a)$ dominiert **then**

13: $changed \leftarrow \top$
14: Führe Pivot in T an $\{a, b\}$ aus.
15: **end if**

16: **end for**
17: **end while**

18: $M \leftarrow \varnothing$
19: **for** $b \in B$ **do**
20: **if** $\exists a \in S(b)$ **then**
21: $M \leftarrow M \cup \{\{a, b\}\}$
22: **end if**
23: **end for**
24: **return** M

4.1 Ein neuer SST-Algorithmus für Graph-Streams

Eine mögliche Realisierung der vorhergegangenen Überlegungen ist in SstStreaming (Algorithmus 3) zu sehen. Anfangs wird ein Pass dafür verwendet, den initialen SST im Hilfsgraphen G^h zu konstruieren (Z.1–7). Anschließend wird der Stream so oft gelesen, bis einen ganzen Pass lang für keine Kante eine Pivot-Operation möglich ist (Z.8–17). Wenn dies der Fall ist, wird ein beliebiges vom SST induziertes Matching zurückgegeben (Z.18–24).
Es lässt sich zeigen, dass der Algorithmus mit Sicherheit terminiert, da die Anzahl der möglichen Pivot-Operationen begrenzt ist. Wir teilen nun die Ausführung des Algorithmus in sogenannte *Phasen* ein, wobei jede Phase beginnt, nachdem eine freie Spalte einen Nachfolger erhalten hat, und endet, nachdem wieder eine freie Spalte einen Nachfolger erhalten hat. Die erste Phase beginnt am Anfang des ersten Stream-Passes und die letzte Phase endet, wenn keine Pivot-Operation mehr möglich ist. Damit ist die Anzahl der Phasen durch $\mathcal{O}(n)$ beschränkt. Innerhalb jeder Phase gibt es zwei unterschiedliche Arten von Pivot-Operationen, die ausgeführt werden können:

- Eine Kante $\{a, b\}$ trifft ein, für die a von $d(a)$ dominiert wird und b nicht dominiert wird. Wenn b frei ist, wird die Phase nach Ausführung der Pivot-Operation beendet. Andernfalls hat b genau einen Nachfolger und wird durch die Pivot-Operation automatisch zu einer Dominante, die mehr Knoten dominiert, als zuvor von $d(a)$ dominiert wurden. Da von einer Dominante höchstens $\mathcal{O}(n)$ Knoten dominiert werden können, ist die Anzahl der Pivot-Operationen dieser Art pro Phase auf $\mathcal{O}(n)$ beschränkt.

- Eine Kante $\{a, b\}$ trifft ein, für die a von $d(a)$ dominiert wird und b von $d(b) \neq d(a)$ dominiert wird. Nach der Pivot-Operation werden alle zuvor von $d(a)$ dominierten Knoten von $d(b)$ dominiert und die Anzahl der Dominanten im SST ist um 1 gefallen. Da die Anzahl der Dominanten im SST höchstens $\mathcal{O}(n)$ beträgt, ist die Anzahl der Pivot-Operationen dieser Art pro Phase auf $\mathcal{O}(n)$ beschränkt.

Insgesamt werden also höchstens $\mathcal{O}(n^2)$ Pivot-Operationen ausgeführt. Danach wird die Abbruchbedingung der While-Schleife in Algorithmus 3 auf jeden Fall wahr. Außerdem gilt:

4.1.1 Theorem. *Sei T der SST in* SstStreaming *nach Abschluss des letzten Passes, so ist T abgeschlossen. Deswegen ist das von* SstStreaming *berechnete Matching kardinalitätsmaximal.*

Beweis. Wir wissen, dass im letzten Pass keine Pivot-Operation ausgeführt worden ist. Wegen Algorithmus 3, Z.12 muss für alle $\{a, b\} \in E$ gelten:

$$\begin{aligned} & \neg((a \text{ wird von } d(a) \text{ dominiert}) \ \wedge \ (b \text{ wird nicht von } d(a) \text{ dominiert})) \\ \equiv \ & \neg(a \text{ wird von } d(a) \text{ dominiert}) \ \vee \ \neg(b \text{ wird nicht von } d(a) \text{ dominiert}) \\ \equiv \ & \neg(a \text{ wird von } d(a) \text{ dominiert}) \ \vee \ (b \text{ wird von } d(a) \text{ dominiert}) \\ \equiv \ & (a \text{ wird von } d(a) \text{ dominiert}) \implies (b \text{ wird von } d(a) \text{ dominiert}) \end{aligned}$$

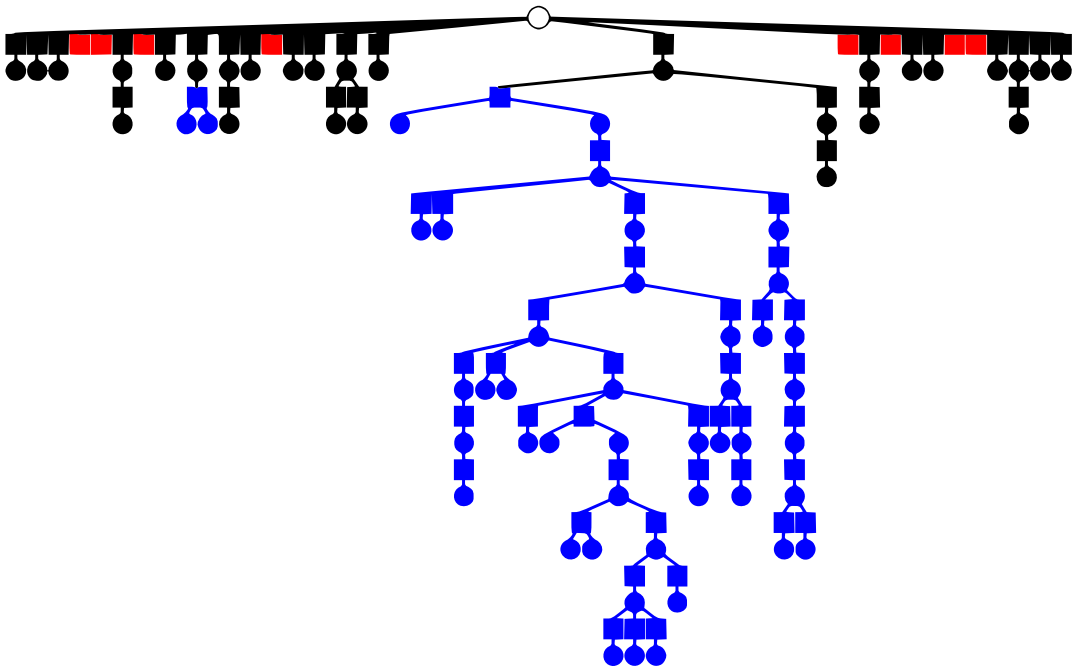
Es gibt also keine Kante $\{a, b\} \in E$, für die nur a, nicht aber b dominiert wird. Nach Definition 3.2.2 ist T abgeschlossen und induziert aufgrund von Theorem 3.2.3 somit ausschließlich Maximum-Matchings. Da SstStreaming ein von T induziertes Matching zurückgibt, muss dieses kardinalitätsmaximal sein. □

4.2 Hindernisse

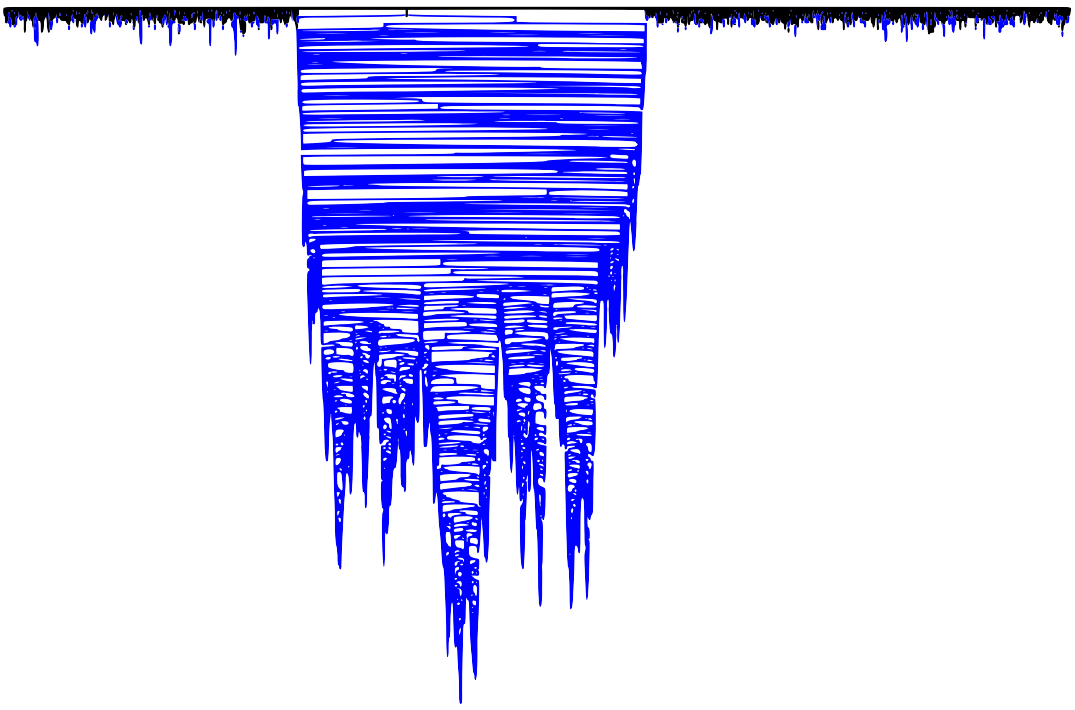
Bei ersten Tests einer Implementierung von Algorithmus 3 mit kleinen Graphinstanzen ($|E| \leq 10^6$) konnten vielversprechende Pass-Counts beobachtet werden. Größere Instanzen haben allerdings grundlegende Probleme mit SSTs in Streaming Modellen offengelegt. Obwohl es sich bei der in SstStreaming gezeigten Realisierung um eine sehr simple Variante handelt, für die es zahlreiche Verbesserungsmöglichkeiten gibt, teilen vermutliche alle auf SSTs beruhenden Streaming-Algorithmen eine wichtige Eigenschaft: Wenn die Kante $\{a, b\}$ eintrifft, muss herausgefunden werden, ob und von welchen Knoten a und b dominiert werden (Algorithmus 3, Z.12). Ansonsten kann nicht entschieden werden, ob eine Pivot-Operation möglich ist. Dabei gibt es prinzipiell zwei verschiedene Vorgehensweisen:

[I]: Jeder Knoten kennt seine Dominante:
Alle Zeilen und Spalten $v \in A \cup B$ erhalten ein Label $d(v)$, welches die aktuelle Dominante beinhaltet. Als Folge dessen ist es nach jeder Pivot-Operation nötig, die Label aller Knoten im umgehängten Teilbaum zu aktualisieren.

[II]: Kein Knoten kennt seine Dominante:
Ist dies der Fall, müssen für jede eintreffende Kante $\{a, b\}$ die Pfade zwischen a bzw. b und der Wurzel des SSTs vollständig nach Kandidaten durchsucht werden. Dabei muss zwangsläufig immer bis zur Wurzel gegangen werden, da nur der am nächsten an der Wurzel liegende Kandidat eine Dominante ist.



(a) $n = 150$



(b) $n = 150\,000$

Abbildung 4.1: Strong-Spanning-Trees in Hilfsgraphen von zufälligen bipartiten Graphen. Kreise repräsentieren Zeilen, Quadrate repräsentieren Spalten. Rote Spalten sind frei. Alle dominierten Knoten sind blau gefärbt. Es ist klar zu erkennen, dass sich in beiden Beispielen eine große Traube unter jeweils einer einzelnen Dominante gebildet hat.

Werfen wir nun einen Blick auf Abbildung 4.1, welche SSTs in Hilfsgraphen zeigt, deren ursprüngliche Graphen aus 150 bzw. 150 000 Knoten bestehen. Tests mit bis zu 500 000 Knoten und verschiedenen Dichten konnten bestätigen, dass die Struktur der in der Abbildung zu sehenden Bäume repräsentativ für die meisten SSTs ist, die im Verlauf des Streaming-Algorithmus entstehen. Zu beobachten ist, dass ein großer Teil aller Knoten von nur wenigen Spalten dominiert wird. In den gezeigten Beispielen hat sich jeweils unter einer Dominante eine große „Traube“ gebildet, in welcher die meisten Knoten nur einen geringen Grad haben, sodass die Tiefe des SSTs beachtlich ist. Dadurch ist Vorgehensweise [I] kaum praktikabel. Wenn jeder Knoten seine Dominante kennen soll, muss nach den meisten Pivot-Operationen über die gesamte Traube iteriert werden, um die $d(v)$ Werte zu aktualisieren. Vor allem in späteren Passes hat sich gezeigt, dass dabei mitunter $\Theta(n)$ Knoten abgearbeitet werden müssen, was deutlich zu lange dauert. Verwendet man hingegen Vorgehensweise [II], muss man bei jeder eingehenden Kante $\{a, b\}$ über alle Knoten auf dem Pfad zwischen der Wurzel und a iterieren (und dasselbe noch einmal für b, falls a dominiert wird). Dank der großen Tiefe, die sich durch die Traube ergibt, ist der Rechenaufwand dafür im Durchschnitt deutlich zu hoch und auf jeden Fall in irgendeiner Form von n abhängig. Unabhängig vom Pass-Count gibt es in SstStreaming also ein großes Problem mit der Rechenzeit pro Kante.

4.2.1 Implementierungsversuche

Bei dem Versuch, trotz der Laufzeitprobleme eine effiziente Lösung zu implementieren, wurden [I] und [II] so kombiniert, dass es zwar Label $d(v)$ gibt, es nach Pivot-Operationen jedoch nicht nötig ist, alle $d(v)$ Label im umgehängten Teilbaum zu aktualisieren. Wir legen fest, dass $d(v)$ immer die letzte bekannte Dominante von v enthalten soll. Für nicht dominierte Knoten darf $d(v) = \texttt{null}$ gesetzt werden. Jeder Knoten erhält außerdem eine Versionsnummer $z(v)$, wobei auch $z(\texttt{null})$ definiert ist und zunächst alle Nummern mit 0 initialisiert werden. Darüber hinaus kennt jeder Knoten die Versionsnummer $z_d(v)$, welche der Versionsnummer $z(d(v))$ zum Zeitpunkt der Zuweisung von $d(v)$ entspricht. Diese Werte werden mit -1 initialisiert. Für die Wurzel 0 wird sichergestellt, dass jederzeit $d(0) = \texttt{null}$ und $z_d(0) = z(d(0))$ gelten.
Algorithmus 4 zeigt, wie nun herausgefunden werden kann, ob und von welcher Dominante ein Knoten v dominiert wird. Hat sich die Versionsnummer von $d(v)$ nicht verändert (siehe Z.1, es gilt dann $z_d(v) = z(d(v))$), ist $d(v)$ tatsächlich die aktuelle Dominante von v. Ist dies jedoch nicht der Fall, gehen wir rekursiv über den Pfad zwischen v und der Wurzel des SSTs und suchen nach der Dominante (Z.4–7). Ist diese gefunden, wird $d(v)$ entsprechend aktualisiert und als Ergebnis zurückgegeben. Zuvor stellt $z_d(v) \leftarrow z(d(v))$ sicher, dass beim nächsten Aufruf von GetDominator(v) nicht wieder über den Pfad iteriert werden muss, sofern sich die Versionsnummer der Dominante bis dahin nicht verändert hat (Z.8).

Algorithmus 4 SstStreaming (GetDominator)

Eingabe: Knoten v

Ausgabe: Dominator von v oder $\texttt{null}$, falls dieser nicht vorhanden ist.

1: **if** $z(d(v)) = z_d(v)$ **then**
2: **return** $d(v)$
3: **end if**

4: $d(v) \leftarrow GetDominator(p(v))$
5: **if** $d(v) = \texttt{null} \wedge (v \text{ ist Spalte}) \wedge |S(v)| > 1$ **then**
6: $d(v) = v$
7: **end if**

8: $z_d(v) \leftarrow z(d(v))$
9: **return** $d(v)$

Dieser Mechanismus ermöglicht es nach Pivot-Operationen, die Dominanten aller im umgehängten Teilbaum befindlichen Knoten in konstanter Zeit ungültig zu machen. Soll etwa eine Pivot-Operation an der Kante $\{a, b\}$ erfolgen und sei d_a die Dominante von a, so muss lediglich eine Zuweisung $z(d_a) \leftarrow z(d_a) + 1$ erfolgen. Wird b durch die Pivot-Operation zur Dominante, ist außerdem die Versionsnummer von $\texttt{null}$ um 1 zu erhöhen.

Obwohl durch dieses Konzept bereits deutlich verbesserte Laufzeiten gemessen werden konnten, hat es längst nicht zum Erreichen der Praxistauglichkeit gereicht. Bei Tests mit zufällig generierten Graphen von 10^6 Knoten pro Partitionszelle und 10^8 Kanten konnten nur sehr langsame Geschwindigkeiten beobachtet werden. Selbst mit allen Kanten im Arbeitsspeicher und unter Verwendung einer modernen CPU (4GHz) können zu Beginn der Ausführung für lange Zeit nur unter 10 000 Kanten pro Sekunde verarbeitet werden. Der überwiegende Teil der Rechenzeit wird dabei nicht mit dem Bestimmen der Dominanten verbracht, sondern mit den eigentlichen Pivot-Operationen. Später können deutlich mehr Kanten pro Sekunde verarbeitet werden, da die Zahl der möglichen Pivot-Operationen sinkt. Dennoch hat die Auswertung für diese eher kleinen Graphen bereits etwa 45 Minuten gedauert, wobei echte Graphinstanzen ohne Weiteres um ein Hundertfaches größer sein können[1]. Aus diesem Grund sind Strong-Spanning-Trees im Semi-Streaming Modell wenn überhaupt nur schwer zu realisieren. Als wesentlich besser geeignet haben sich die klassischen alternierenden Bäume erwiesen, die im nächsten Kapitel verwendet werden, um einen Approximationsalgorithmus für Maximum-Matchings im Semi-Streaming Modell zu entwickeln.

[1] http://law.di.unimi.it/datasets.php

Kapitel 5

Streaming-Algorithmus mit alternierenden Bäumen

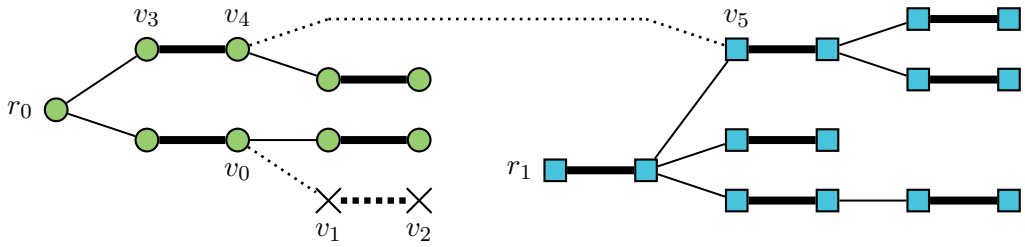
Als Basis für den in diesem Kapitel vorgestellten Algorithmus dienen ähnlich wie in Kliemanns DapTrees [15] knotendisjunkte alternierende Bäume, welche genutzt werden um augmentierende Pfade zu finden. Vor allem im Aufbau der Bäume ähneln sich die beiden Verfahren stark. Eine genaue Abgrenzung erfolgt an späterer Stelle.

5.1 Baum- und Waldoperatoren

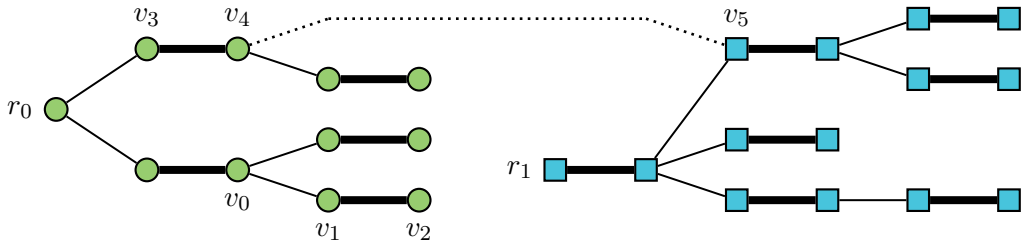
Zunächst führen wir eine Reihe neuer Operatoren für alternierende Bäume und Wälder ein.

5.1.1 Definition (Löschoperator $\ominus$). Es sei $G = (V, E)$ ein beliebiger Graph, $T = (r_T, E_T)$ ein Baum in G und F ein Wald in G. Der Löschoperator $\ominus$ kann Knoten aus T bzw. F entfernen. Die Knoten $\overline{v} \in V \setminus V(T)$, $v \in V(T) \setminus \{r_T\}$, sowie $v_0, \ldots, v_q \in V$ seien beliebig und $T(v) = (v, E_v)$ sei der in v wurzelnde Teilbaum von T. Wir definieren:

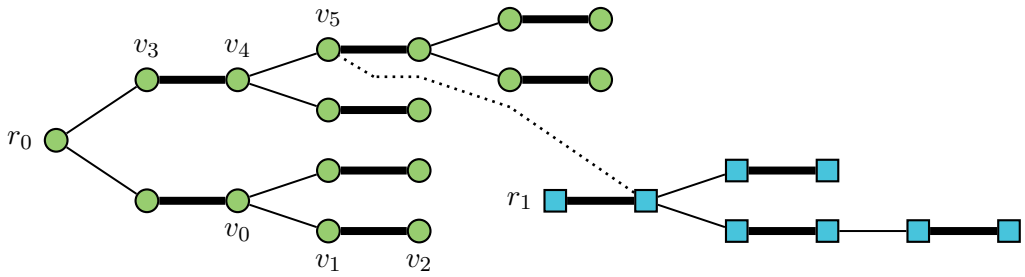
1. $T \ominus \{\overline{v}\} := \{T\}$
2. $T \ominus \{r_T\} := \{T(s) \mid s \in S(r_T)\}$
3. $T \ominus \{v\} := \{T(s) \mid s \in S(v)\} \cup \{(r_T, E_T \setminus E_v)\}$
4. $F \ominus \{v_0\} := \{T_i \ominus \{v_0\} \mid T_i \in F\}$
5. $F \ominus \{v_0, v_1, \ldots, v_q\} := (F \ominus \{v_0\}) \ominus \{v_1, \ldots, v_q\}$
6. $T \ominus \{v_0, v_1, \ldots, v_q\} := (T \ominus \{v_0\}) \ominus \{v_1, \ldots, v_q\}$ ◁



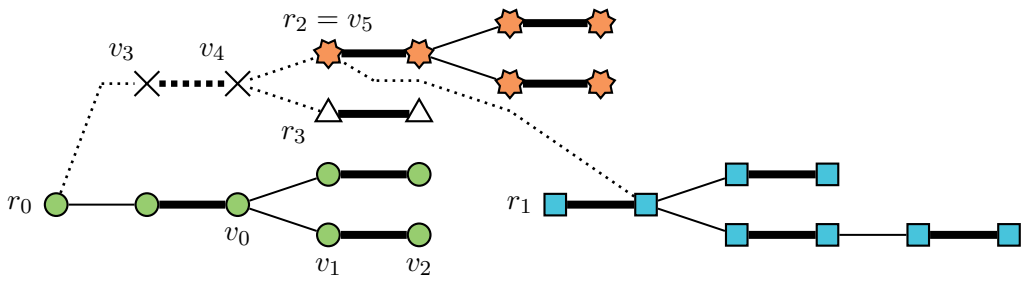
(a) Alternierender Wald F_0



(b) Alternierender Wald $F_1 = (F_0 \oplus \{v_0, v_1\}) \oplus \{v_1, v_2\}$



(c) Alternierender Wald $F_2 = F_1 \oslash \{v_4, v_5\}$



(d) Alternierender Wald $F_3 = (F_2 \ominus \{v_3\}) \ominus \{v_4\} = F_2 \ominus \{v_3, v_4\}$

Abbildung 5.1: Operationen in einem alternierenden Wald. Durchgezogene Kanten liegen im Wald, gepunktete Kanten nicht. Knoten, die zum selben Baum gehören, haben dieselbe Form und Farbe. Knoten, die nicht im Wald liegen, sind Kreuze. Wurzeln folgen dem Namensschema r_i.

Wenn wir mit dem Löschoperator einen Knoten v aus einem Baum T entfernen, zerfällt dieser also in einen Wald F. Dabei besteht F aus allen Teilbäumen von T, die in einem Nachfolger von v wurzeln, sowie dem Baum, den man durch das Entfernen aller Kanten des in v wurzelnden Teilbaums aus T erhält. In Abbildung 5.1c–5.1d ist zu sehen, wie die Knoten v_3 und v_4 aus einem alternierenden Wald entfernt werden. Als Resultat dessen verliert der in r_0 wurzelnde Baum einige Kanten und zwei neue, in r_2 und r_3 wurzelnde Bäume kommen zum Wald hinzu.
Wir zeigen nun, dass der Wald, der durch das Entfernen einer gültigen Wurzel aus einem alternierenden Wald entsteht, ebenfalls alternierend ist:

5.1.2 Lemma. *Sei $T = (r_T, E_T)$ ein gültiger alternierender Baum im alternierenden Wald F. Der Wald $F \ominus \{r_T\}$ ist alternierend.*

Beweis. Laut Definition 5.1.1 zerfällt der Baum T durch das Entfernen der Wurzel in die Bäume $\{T(s) \mid s \in S(r_T)\}$, welche genau die Teilbäume sind, die in T in einem Nachfolger von r_T wurzeln.
Es ist also zu zeigen, dass jeder Baum $T' \in \{T(s) \mid s \in S(r_T)\}$ alternierend ist. Wir wissen, dass alle Blätter von T' zuvor Blätter im alternierenden Baum T waren. Deswegen kann Bedingung 2. aus Definition 2.4.2 von T' nicht verletzt werden. Es sei $T' = (r_{T'}, E_{T'})$. Da es zuvor einen alternierenden Pfad $(r_T, r_{T'}, m(r_{T'}), \ldots, v') \subseteq E_T$ zu jedem Knoten $v' \in V(T') \setminus \{r_{T'}\}$ gab, muss es einen alternierenden Pfad $(r_{T'}, m(r_{T'}), \ldots, v') \subseteq E_{T'}$ geben und auch Bedingung 1. ist erfüllt. Weil $m(r_{T'})$ in diesem Pfad direkt hinter $r_{T'}$ liegt, muss außerdem $p(m(r_{T'})) = r_{T'}$ gelten und somit Bedingung 3. erfüllt sein. □

Außerdem ergibt sich ein alternierender Wald, wenn man aus einem alternierenden Baum die Knoten einer beliebigen Matching-Kante entfernt:

5.1.3 Lemma. *Sei $T = (r_T, E_T)$ ein alternierender Baum und v ein beliebiger gematchter Knoten in T. Der Wald $T \ominus \{v, m(v)\}$ ist alternierend.*

Beweis. Wir nehmen o.B.d.A an, dass v ein Nichtgabelknoten ist (wenn v ein Gabelknoten ist, erfolgt der Beweis analog mit v und $m(v)$ vertauscht). Es sei $T(v) = (v, E_v)$ der in v wurzelnde Teilbaum von T.
Laut Definition 5.1.1 zerfällt der Baum T durch das Entfernen der beiden Knoten in mehrere Bäume. Zum einen entsteht der Baum T', den man erhält, wenn man aus T den gesamten in v wurzelnden Teilbaum entfernt (vorausgesetzt v ist nicht die Wurzel von T). Zum andern entstehen die Bäume $\{T(s) \mid s \in S(m(v))\}$, welche genau die Teilbäume sind, die in T in einem Nachfolger von $m(v)$ wurzeln.
Beginnen wir also mit dem Baum T'. Dieser muss alternierend sein: Es wurde lediglich der Teilbaum $T(v)$ abgetrennt, wobei der Vorgänger $p(v)$ einen Nachfolger verloren hat. Da es sich bei $p(v)$ um einen Gabelknoten handeln muss, kann Bedingung 2. aus Definition

2.4.2 nicht verletzt worden sein. Außerdem befinden sich in T' nur Knoten, die zuvor in T waren und offenbar sind diese Knoten durch dieselben Pfade mit der Wurzel verbunden, wie zuvor. Deswegen muss Bedingung 1. erfüllt sein. Da sich r_T nicht verändert hat, kann auch Bedingung 3. nicht von T' verletzt werden.
Es bleibt zu zeigen, dass auch jeder Baum $T'' \in \{T(s) \mid s \in S(m(v))\}$ alternierend ist. Wir wissen, dass alle Blätter von T'' zuvor Blätter im alternierenden Baum T waren. Deswegen kann Bedingung 2. aus Definition 2.4.2 von T'' nicht verletzt werden. Es sei $T'' = (r_{T''}, E_{T''})$. Da es zuvor einen alternierenden Pfad $(r_T, \ldots, v, m(v), r_{T''}, m(r_{T''}), \ldots, v'') \subseteq E_T$ zu jedem Knoten $v'' \in V(T'') \setminus \{r_{T''}\}$ gab, muss es einen alternierenden Pfad $(r_{T''}, m(r_{T''}), \ldots, v'') \subseteq E_{T''}$ geben und auch Bedingung 1. ist erfüllt. Weil $m(r_{T''})$ in diesem Pfad direkt hinter $r_{T''}$ liegt, muss außerdem $p(m(r_{T''})) = r_{T''}$ gelten und somit Bedingung 3. erfüllt sein. □

Aus Lemma 5.1.3 folgt unmittelbar:

5.1.4 Korollar. *Sei F ein alternierender Wald und v ein beliebiger gematchter Knoten in F. Der Wald $F \ominus \{v, m(v)\}$ ist alternierend.*

5.1.5 Definition (Erweiterungsoperator $\oplus$). Sei $G = (V, E)$ ein beliebiger Graph, F ein Wald in G und $T = (r_T, E_T)$ ein Baum in F. Der Erweiterungsoperator $\oplus$ kann Kanten zu T bzw. F hinzufügen. Es sei $\{a, b\}$ eine beliebige Kante zwischen $a \in V(T)$ und $b \notin V(F)$. Wir definieren:

1. $T \oplus \{a, b\} := (r_T, E_T \cup \{\{a, b\}\})$
2. $F \oplus \{a, b\} := (F \setminus T) \cup \{T \oplus \{a, b\}\}$ ◁

Der Erweiterungsoperator kann also eine Kante an jeden im Wald enthaltenen Knoten anhängen. Abbildung 5.1a–5.1b zeigt, wie zwei Kanten an einen Wald angehängt werden. Für die Erweiterung alternierender Bäume gilt:

5.1.6 Lemma. *Sei $T = (r_T, E_T)$ ein alternierender Baum, v_1 ein beliebiger Gabelknoten in T und $v_2 \notin V(T)$ ein beliebiger aber gematchter Knoten. Der Baum $T' = T \oplus \{v_1, v_2\} \oplus \{v_2, m(v_2)\}$ ist alternierend.*

Beweis. Da zu T lediglich Kanten hinzukommen, kann sich an Eigenschaft 3. aus Definition 2.4.2 nichts verändert haben. Es sei $T' = (r_{T'}, E_{T'})$. Weil v_1 Gabelknoten ist, gab es vor der Erweiterung einen alternierenden Pfad $(r_T, \ldots, m(v_1), v_1) \subseteq E_T$ in T (oder v_1 ist die Wurzel von T). Daraus folgt, dass es in T' einen alternierenden Pfad $(r_T, \ldots, m(v_1), v_1, v_2, m(v_2)) \subseteq E_{T'}$ gibt (oder v_1 ist die Wurzel von T und es gibt einen Pfad $(v_1, v_2, m(v_2)) \subseteq E_{T'}$), sodass Eigenschaft 1. nicht verletzt wird. Von den beiden neu hinzugekommenen Knoten hat nur $m(v_2)$ eine leere Nachfolgermenge $S(m(v_2))$ und es gilt offenbar $p(m(v_2)) = m(m(v_2)) = v_2$. Deswegen ist Eigenschaft 2. ebenfalls erfüllt. □

Aus Lemma 5.1.6 folgt unmittelbar:

5.1.7 Korollar. *Sei F ein alternierender Wald, v_1 ein beliebiger Gabelknoten in F und $v_2 \notin V(F)$ ein beliebiger aber gematchter Knoten. Der Wald $F' = F \oplus \{v_1, v_2\} \oplus \{v_2, m(v_2)\}$ ist alternierend.*

5.1.8 Definition (Umhängeoperator $\oslash$). Sei $G = (V, E)$ ein beliebiger Graph, F ein Wald in G und $T_1 = (r_1, E_1) \in F$ sowie $T_2 = (r_2, E_2) \in F$ beliebige Bäume in F (es darf $T_1 = T_2$ gelten). Der Umhängeoperator $\oslash$ kann Teilbäume innerhalb von F umhängen. Es sei $\{a, b\}$ eine beliebige Kante zwischen $a \in V(T_1)$ und $b \in V(T_2) \setminus \{r_2\}$. Der in b wurzelnde Teilbaum sei $T(b) = (b, E_b)$. Für $T_1 = T_2$ muss $a \notin S'(b)$ gelten. Wir definieren:

$$F \oslash \{a, b\} := ((F \ominus V(T(b))) \setminus \{T_1\}) \cup \{(r_1, E_1 \cup E_b \cup \{\{a, b\}\})\}$$

Gibt es eine Kante $\{a, r_2\}$, definieren wir außerdem:

$$F \oslash \{a, r_2\} := (F \setminus \{T_1, T_2\}) \cup \{(r_1, E_1 \cup E_2 \cup \{\{a, r_2\}\})\}$$

◁

Obwohl der Umhängeoperator formal zunächst einen komplizierten Eindruck macht, handelt es sich um einen sehr simplen Vorgang: Ein Teilbaum wird von einem Baum im Wald abgetrennt und an einem anderen Baum wieder angehängt (oder an demselben Baum an anderer Stelle). Wir sehen in Abbildung 5.1b–5.1c, wie der in v_5 wurzelnde Teilbaum umgehängt wird.

5.1.9 Lemma. *Sei F ein alternierender Wald, $T_1, T_2 \in F$ Bäume in F, a ein Gabelknoten in T_1 und b ein Nichtgabelknoten in T_2. Gibt es eine Kante $\{a, b\}$ im F enthaltenden Graphen, so ist $F' = F \oslash \{a, b\}$ ein alternierender Wald.*

Beweis. Es sei $T(b) = (b, E_b)$ der in b wurzelnde Teilbaum von T_2. Wir betrachten zunächst den Baum $T_2' = T_2 \ominus V(T(b))$, welcher sich in F' ergibt, wenn b nicht die Wurzel von T_2 ist. Da b ein Nichtgabelknoten ist, beginnen alle Pfade zwischen b und beliebigen Knoten in $V(T(b))$ mit einer Matching-Kante. Alle Pfade zu Blättern müssen ebenfalls mit einer Matching-Kante enden (weil T_2 ein alternierender Baum ist und in ihm Definition 2.4.2, Eigenschaft 2. erfüllt ist) und enthalten deswegen nur gematchte Knoten. Außerdem liegt offenbar jeder Knoten $v \in V(T(b))$ auf einem solchen Pfad, weswegen es in $T(b)$ keine freien Knoten geben kann. Deswegen können wir $T_2' = T_2 \ominus V(T(b)) = T_2 \ominus \{v, m(v) \mid v \in V(T(b))\}$ schreiben, woraus zusammen mit Korollar 5.1.4 folgt, dass T_2' alternierend ist. Außerdem folgt unmittelbar, dass auch $T(b)$ ein alternierender Baum ist.
Es bleibt zu zeigen, dass $T_1' = (r_1, E_1 \cup E_b \cup \{a, b\})$ für $T_1 = (r_1, E_1)$ ein alternierender Baum ist. Da zu T_1 lediglich Kanten hinzukommen, kann Eigenschaft 3. aus Definition 2.4.2 in T_1' nicht verletzt sein. Außerdem wissen wir bereits, dass $T(v)$ ein alternierender

Baum ist. Weil alle Blätter in T_1' entweder Blätter in T_1 oder in $T(v)$ waren, wird auch Eigenschaft 2. erfüllt. Wir wissen, dass T_1 und $T(b)$ alternierend sind, weswegen es alternierende Pfad $(r_1, \ldots, m(a), a) \in E_1$ und $(b, m(b), \ldots, v) \in E_b$ für $v \in V(T(b)) \setminus \{b\}$ gibt (oder a ist die Wurzel von T_1). Als Folge dessen muss es in T_1' einen alternierenden Pfad $(r_1, \ldots, m(a), a, b, m(b), \ldots v) \in E_1 \cup E_b \cup \{a, b\}$ geben (oder a ist die Wurzel von T_1 und es gibt einen Pfad $(a, b, m(b), \ldots v) \in E_1 \cup E_b \cup \{a, b\}$). Somit ist auch Eigenschaft 1. in T_1' erfüllt.
Wir haben gezeigt, dass T_1' und T_2' alternierende Bäume sind. Für F gilt laut Definition 5.1.8:

$$\begin{aligned} F \oslash \{a,b\} &= ((F \ominus V(T(b))) \setminus \{T_1\}) \cup \{(r_1, E_1 \cup E_b \cup \{\{a,b\}\})\} \\ &= ((F \ominus V(T(b))) \setminus \{T_1\}) \cup \{T_1'\} \\ &= (F \setminus \{T_1, T_2\}) \cup \{T_1', T_2 \ominus V(T(b))\} \\ &= (F \setminus \{T_1, T_2\}) \cup \{T_1', T_2'\} \end{aligned}$$

Also ist auch $F' = F \oslash \{a, b\}$ alternierend. □

5.2 Algorithmisches Konzept

Die eingeführten Operatoren sind die Basis für den neuen Algorithmus AugTrees und erlauben es uns, einen alternierenden Wald aufzubauen und aufrecht zu erhalten, während wir den Stream der Kanten lesen. Alle Zeilenangaben beziehen sich auf Algorithmus 5. Zu Beginn ist das Matching M leer und der alternierende Wald F enthält für jeden Knoten im Graphen einen 1-elementigen gültigen alternierenden Baum (Z.1–2).
Nun lesen wir den Kanten-Stream so oft, bis die Abbruchbedingung erfüllt ist, auf welche wir später genauer eingehen (Z.4–29). Dabei überprüfen wir für jede gelesene Kante $\{a, b\}$, ob sie zwei Gabelknoten von gültigen Bäumen verbindet (Z.7), denn genau dann haben wir einen augmentierenden Pfad P gefunden. Liegen a und b in den alternierenden Bäumen $T_a = (r_a, E_a)$ und $T_b = (r_b, E_b)$, verläuft dieser Pfad zwischen den Wurzeln r_a und r_b (Z.9–11). Bevor wir mithilfe von P das Matching augmentieren (Z.13), werden alle in $V(P)$ enthaltenen Knoten aus dem alternierenden Wald entfernt (Z.12). Dadurch zerfallen T_a und T_b jeweils in ungültige alternierende Bäume (analog zu Abbildung 3.1). Im ersten Pass wird besonders oft augmentiert, da alle Kanten zwischen zwei gültigen Wurzeln in einem augmentierenden Pfad der Länge 1 resultieren.
Verbindet $\{a, b\}$ jedoch keine zwei Gabelknoten, wird überprüft ob zumindest nur a oder nur b Gabelknoten in einem gültigen alternierenden Baum ist (Z.14 bzw. Z.25). Ist dies für a der Fall (analog für b), muss zunächst sichergestellt werden, dass a nicht zu tief in seinem alternierenden Baum liegt (Z.15). Andernfalls wären später gefundene augmentierende Pfade häufig sehr lang und somit für das Erreichen der gewünschten Approximationsgüte unnötig. Wir werden in Abschnitt 5.3 sehen, wie die Länge der Pfade sich auf

Algorithmus 5 AugTrees

Eingabe: Bipartiter Graph $G = (A, B, E)$, Approximationsparameter k
Ausgabe: Matching M mit Approximationsgüte $\frac{k}{k+1}$

1: $M \leftarrow \varnothing$
2: $F \leftarrow \{(v, \varnothing) \mid v \in A \cup B\}$
3: $changed \leftarrow \{A, B\}$
4: **while** $changed = \{A, B\}$ **do**
5: $changed \leftarrow \varnothing$
6: **for** $\{a, b\} \in E$ **where** $a \in A \wedge b \in B$ **do**
7: **if** a ist gültige Gabel in $T_a = (r_a, E_a) \in F \wedge$ b ist gültige Gabel in $T_b = (r_b, E_b) \in F$ **then**
8: $changed \leftarrow \{A, B\}$
9: $P_a \leftarrow (r_a, \ldots, a) \subseteq E_a$
10: $P_b \leftarrow (r_b, \ldots, b) \subseteq E_b$
11: $P \leftarrow P_a \cup \{\{a, b\}\} \cup P_b$
12: $F \leftarrow F \ominus V(P)$
13: $M \leftarrow M \triangle P$
14: **else if** a ist gültige Gabel in F **then**
15: **if** $l(a) + 2 < 2k - 1$ **then**
16: **if** b ist Gabel in $F \vee b$ liegt nicht in F **then**
17: $changed \leftarrow changed \cup \{A\}$
18: $F \leftarrow F \ominus \{b, m(b)\}$
19: $F \leftarrow F \oplus \{a, b\} \oplus \{b, m(b)\}$
20: **else if** $T_b \in F$ mit $b \in V(T_b)$ ist ungültig $\vee$ $(l(a) + 1 < l(b))$ **then**
21: $changed \leftarrow changed \cup \{A\}$
22: $F \leftarrow F \oslash \{a, b\}$
23: **end if**
24: **end if**
25: **else if** b ist gültige Gabel in $T_b \in F$ **then**
26: $[\ldots]$
27: **end if**
28: **end for**
29: **end while**
30: **return** M

die Approximation auswirkt. Bei ausreichend geringer Tiefe von a ist unter Umständen die Erweiterung des a enthaltenden Baums T_a möglich. Wir wissen bereits, dass b keine Gabel in einem gültigen Baum ist (sonst hätten wir schließlich einen augmentierenden Pfad gefunden). Ist b nun Gabel in einem ungültigen alternierenden Baum, können wir b und seinen Matching-Partner $m(b)$ aus diesem entfernen (Z.18) und anschließend sowohl $\{a, b\}$ als auch $\{b, m(b)\}$ zu T_a (bzw. zu F) hinzufügen (Z.19). Liegt b hingegen in gar keinem Baum in F, bleibt Zeile 18 ohne Effekt und die beiden eben genannten Kanten kommen nach wie vor zu F hinzu. Durch das Erweitern der alternierenden Bäume kommen neue Gabelknoten hinzu, die für zukünftige Stream-Kanten und -Passes potentiell augmentierende Pfade zur Folge haben.
Kommt eine Erweiterung nicht in Frage, bietet sich möglicherweise eine Umhängeoperation an (Z.20–23). Bei dieser wird der ganze in b wurzelnde Teilbaum von dem b enthaltenden alternierenden Baum getrennt und mit der Kante $\{a, b\}$ an den a enthaltenden Baum angefügt (analog zu Abbildung 3.2). Damit diese Operation erlaubt ist, muss eine von zwei Bedingungen zutreffen (Z.20): Entweder muss b in einem ungültigen alternierenden Baum liegen, oder die Tiefe von b muss nach dem Umhängen kleiner sein, als zuvor. In jedem Fall ist b ein Nichtgabelknoten, da sonst der Block in Z.16–19 ausgeführt worden wäre. Durch Umhängeoperationen werden ungültige Teilbäume wieder in den Wald integriert, wodurch sich möglicherweise neue augmentierende Pfade ergeben. Außerdem wird die Tiefe von Gabelknoten gesenkt, wodurch sich auch zusätzliche Erweiterungs- und Umhängeoperationen ergeben können.
Zu jeder der beiden Partitionszellen merken wir uns, ob sich im aktuellen Pass bereits ein darin wurzelnder alternierender Baum verändert hat. Das ist genau dann der Fall, wenn ein solcher Baum dazu geführt hat, dass ein augmentierender Pfad gefunden wurde, oder wenn der Baum erweitert wurde, oder wenn ein Teilbaum an ihn umgehängt wurde. Erst wenn sich einen ganzen Pass lang nur für höchsten eine Partitionszelle Bäume verändern, ist die Abbruchbedingung erfüllt und das aktuelle Matching wird als Ergebnis zurückgegeben.

5.3 Korrektheit und Approximationsgüte

Wir zeigen zunächst, dass es nach der Ausführung von AugTrees keine augmentierenden Pfade der Länge $2k + 1$ oder kürzer mehr gibt. Zusammen mit dem darauf folgenden Abschnitt zur Approximationsgüte ergibt sich die Korrektheit. Eingangs wollen wir ein paar allgemeine Aussagen zur Gültigkeit des Algorithmus darlegen:

5.3.1 Theorem. *In Algorithmus 5 ist F vor und nach jeder Iteration der For-Schleife ein alternierender Wald.*

Beweis. Zu Beginn ist F offenbar alternierend: Es befinden sich nur 1-elementige alternierende Bäume in F, welche jeweils aus einem freien Knoten bestehen. Danach wird F an drei Stellen in Algorithmus 5 verändert:

- **Z.18–19**. Liegt b (und somit auch $m(b)$) nicht in F, ist Z.18 ohne Effekt. Andernfalls bleibt F alternierend, da Korollar 5.1.4 angewendet werden kann. Wegen Z.14 ist a gültige Gabel in F, während b vor Ausführung von Z.19 mit Sicherheit aus F entfernt wurde (Z.18). Deswegen lässt sich Korollar 5.1.6 auf Z.19 anwenden und F ist nach der Ausführung von Z.18–19 nach wie vor alternierend.

- **Z.22**. Wegen Z.14 ist a gültige Gabel in F, wegen Z.16 ist b Nichtgabelknoten in F. Dadurch lässt sich Lemma 5.1.9 anwenden und die alternierende Eigenschaft von F bleibt erhalten.

- **Z.12**. Es werden die beiden Wurzeln r_a und r_b sowie alle Knoten auf dem augmentierenden Pfad zwischen den Wurzeln entfernt. Da im Pfad zu jedem Knoten (außer zu r_a und r_b) auch sein Matching-Partner enthalten ist, lässt sich Korollar 5.1.4 anwenden. Auch das Entfernen der beiden gültigen Wurzeln hinterlässt nur alternierende Bäume (Lemma 5.1.2).

Es wird also niemals die alternierende Eigenschaft von F verletzt. □

5.3.2 Theorem. *In Algorithmus 5 ist M vor und nach jeder Iteration der For-Schleife ein Matching.*

Beweis. Zu Beginn ist M leer und somit ein Matching. Für die beiden gültigen Gabeln $a \in V((r_a, E_a))$ und $b \in V((r_b, E_b))$ sind die Pfade $P_a = (r_a, \ldots, a) \subseteq E_a$ und $P_b = (r_b, \ldots, b) \subseteq E_b$ alternierend und beginnen bei den beiden Wurzeln r_a und r_b jeweils mit einer freien Kante. Außerdem enden sie in a bzw. b mit einer Matching-Kante. Deswegen muss $P = P_a \cup \{\{a, b\}\} \cup P_b$ augmentierend sein (Z.11) und aus Lemma 2.3.4 folgt, dass $M \triangle P$ ein Matching ist. □

Nun wollen wir per Widerspruch zeigen, dass es keine augmentierenden Pfade der Länge $2k-1$ oder kürzer mehr gibt, wenn Algorithmus 5 terminiert. Wir betrachten dazu einen beliebigen bipartiten Graphen $G = (A, B, E)$ und ein zugehöriges Matching M, das von Algorithmus 5 mit Parameter k berechnet wurde. Es sei $P = (v_0, v_1, \ldots, v_{2q-1})$ ein M-augmentierender Pfad mit $q \leq k$ und $v_0 \in A$ (für $v_0 \in B$ kann analog argumentiert werden). Weil AugTrees terminiert ist, kann sich im letzten Pass für mindestens eine der beiden Partitionszellen kein darin wurzelnder Baum verändert haben. Wir nehmen nun an, dass dies für A der Fall ist. Es sei $v_i \in A \cup V(P)$ ein beliebiger gültiger Gabelknoten mit $0 \leq i < 2q-2$. Offenbar ist i gerade und v_i liegt in einem in A wurzelnden alternierenden Baum. Es gilt mindestens eine der folgende Aussagen:

[I]: Es gilt $l(v_i)+2 \geq 2k-1$. Dadurch wird jede Erweiterungs- oder Umhängeoperation an den v_i enthaltenden Baum verhindert, was nötig ist, damit sich kein in A wurzelnder alternierender Baum verändert (Algorithmus 5, Z.15).

[II]: Wenn [I] nicht zutrifft, muss v_{i+1} Nichtgabelknoten in einem gültigen alternierenden Baum sein, sodass $l(v_i)+1 \geq l(v_{i+1})$ gilt. Da v_{i+1} mit v_{i+2} gematcht ist, muss auch $p(v_{i+2}) = m(v_{i+2}) = v_{i+1}$ und somit $l(v_{i+2}) = l(v_{i+1})+1 \leq l(v_i)+2$ gelten und v_{i+2} ist ein gültiger Gabelknoten.

Begründung: Wenn v_{i+1} Gabelknoten wäre, oder in gar keinem Baum liegen würde, hätte eine Erweiterung stattgefunden und AugTrees wäre nicht terminiert (Algorithmus 5, Z.16–19). Wenn v_{i+1} Nichtgabelknoten in einem ungültigen alternierenden Baum wäre oder $l(v_i)+1 < l(v_{i+1})$ gelten würde, hätte eine Umhängeoperation stattgefunden und ein in A wurzelnder alternierender Baum hätte sich verändert (Algorithmus 5, Z.20–22).

Wir zeigen nun induktiv für alle $v_i \in A \cup V(P)$ mit $0 \leq i < 2q-1$:

Induktionsvoraussetzung (I.V.):

Bei v_i handelt es sich um einen gültigen Gabelknoten mit $l(v_i) \leq i$. Gilt $i < 2q-2$, so trifft für v_i Aussage [I] nicht zu und es muss [II] gelten.

Induktionsanfang (I.A.): $i \leftarrow 0$

Die gültige Wurzel v_0 ist offenbar gültiger Gabelknoten mit $l(v_0) = 0$.

Induktionsschritt (I.S.): $i \leftarrow i+2$

Wegen I.V. ist $l(v_i) \leq i$ und Aussage [II] gilt für v_i. Deswegen ist auch v_{i+2} gültiger Gabelknoten und es gilt $l(v_{i+2}) \underset{\text{[II]}}{\leq} l(v_i)+2 \underset{\text{I.V.}}{\leq} i+2$.

Wenn sich im letzten Pass also kein in A wurzelnder alternierender Baum verändert hat, ist insbesondere v_{2q-2} ein gültiger Gabelknoten. Da allerdings v_{2q-1} als Wurzel eines gültigen alternierenden Baums ebenfalls ein gültiger Gabelknoten sein muss, wäre in AugTrees bei eintreffender Stream-Kante $\{v_{2q-2}, v_{2q-1}\}$ ein alternierender Pfad gefunden worden und es wäre mindestens noch ein weiterer Pass ausgeführt worden. Aufgrund dieses Widerspruchs muss sich im letzten Pass ein in A wurzelnder alternierender Baum verändert haben, wenn es einen augmentierenden Pfad der Länge $2q-1$ oder kürzer gibt. Genau dasselbe können wir auch für B zeigen. Dazu kehren wir die Reihenfolge der Kanten in P einfach um, sodass [I], [II] und die Induktion nicht mehr für $v_i \in A \cup V(P)$ und A gelten, sondern für $v_i \in B \cup V(P)$ und B. Somit ändern sich in jedem Pass für beide Partitionen darin wurzelnde gültige Bäume, solange es für $q \leq k$ noch augmentierende Pfade der Länge $2q-1$ gibt. Weil augmentierende Pfade immer von ungerader Länge sind, ergibt sich:

5.3.3 Theorem. *Sei M das von* AugTrees *zurückgegebene Matching. Der kürzeste M-augmentierende Pfad hat mindestens Länge $2k+1$.*

5.3.1 Approximationsgüte

Im ersten Pass scheint AugTrees die Funktionalität von Greedy zu beinhalten: Paare von freien Knoten werden bei entsprechender Stream-Kante direkt gematcht. Tatsächlich können wir leicht zeigen:

5.3.4 Theorem. *Das Matching M, welches in* AugTrees *nach Beendigung des ersten Passes vorliegt, ist mindestens eine $\frac{1}{2}$-Approximation eines Maximum-Matchings.*

Beweis. Es sei $G = (A, B, E)$ der Graph, auf welchem AugTrees ausgeführt wird. Nach dem ersten Pass seien die Knoten $a \in A$ und $b \in B$ frei und es gebe eine Kante $\{a, b\} \in E$. Da a und b frei sind, müssen sie Wurzeln und somit Gabeln von gültigen Bäumen sein. Beim Lesen der Kante $\{a, b\}$ im ersten Pass gelten also beide Bedingungen in Z.7, wodurch a und b miteinander gematcht werden. Es kommt zum Widerspruch, weswegen a und b nach dem ersten Pass nicht beide frei sein können. Demnach ist M inklusionsmaximal und somit eine $\frac{1}{2}$-Approximation an ein Maximum-Matching. □

Wir haben bereits gesehen, dass es keine augmentierenden Pfade der Länge $2k - 1$ oder kürzer mehr gibt, wenn AugTrees terminiert. Über diese Pfadlänge werden wir nun die Approximationsgüte beweisen und berufen uns dabei auf:

5.3.5 Theorem (Hopcroft und Karp, [14]). *Seien M und M' mit $|M'| > |M|$ Matchings im selben Graphen, so enthält $M \triangle M'$ mindestens $|M'| - |M|$ knotendisjunkte M-augmentierende Pfade.*

5.3.6 Korollar. *Sei M ein Matching, sodass der kürzeste M-augmentierenden Pfad die Länge $2k + 1$ hat. Dann ist M eine $(\frac{k}{k+1})$-Approximation eines Maximum-Matchings.*

Beweis. Sei $G = (A, B, E)$ ein beliebiger Graph, M ein Matching in G und M^* ein Maximum-Matching in G. Wir nehmen an, dass $|M| < |M^*|$ gilt (andernfalls ist M bereits kardinalitätsmaximal und somit auch eine $(\frac{k}{k+1})$-Approximation). Wegen $M \triangle M^* \subseteq E$ und Theorem 5.3.5 gibt es in G mindestens $|M^*| - |M|$ knotendisjunkte M-augmentierende Pfade. Der kürzeste M-augmentierende Pfad habe die Länge $2k+1$ und somit k Matching-Kanten. Da es mindestens $|M^*| - |M|$ knotendisjunkte M-augmentierende Pfade mit gleicher oder größerer Länge gibt, von denen jeder mindestens k Matching-Kanten beinhaltet, muss es auch mindestens $k \cdot (|M^*| - |M|)$ Matching-Kanten in M geben:

$$\begin{aligned} & k \cdot (|M^*| - |M|) && \leq && |M| \\ \iff \quad & k \cdot |M^*| - k \cdot |M| && \leq && |M| \\ \iff \quad & k \cdot |M^*| && \leq && |M| \cdot (k+1) \\ \iff \quad & k/(k+1) \cdot |M^*| && \leq && |M| \end{aligned}$$

Demnach ist M eine $\frac{k}{k+1}$-Approximation an ein Maximum-Matching. □

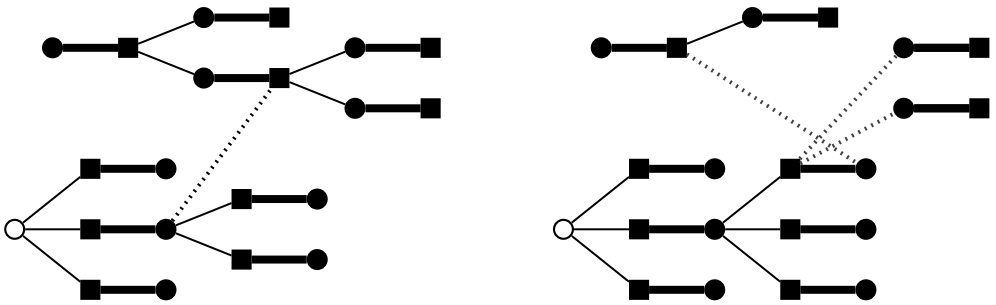
AugTrees hält laut Theorem 5.3.3 erst dann an, wenn der kürzeste augmentierende Pfad mindestens $2k + 1$ Kanten hat, und es gilt somit:

5.3.7 Korollar. *Das von* AugTrees *berechnete Matching ist mindestens eine* $\frac{k}{k+1}$*-Approximation an ein Maximum-Matching.*

5.4 Abgrenzung

Wie zu Beginn des Kapitels erwähnt, ähneln sich AugTrees und DapTrees sehr. Lediglich zwei wichtige Schlüsselaspekte wurden mit Bedacht verändert, um die Anzahl der benötigten Passes sowohl im Worst- als auch im Average-Case zu verringern.

In beiden Algorithmen werden alternierende Bäume aufgebaut, um augmentierende Pfade zu finden. Wir betrachten zunächst Kliemanns Vorgehen, bei welchem alle Bäume in der selben Partitionszelle wurzeln, sodass ein augmentierender Pfad genau dann gefunden wird, wenn eine Kante von einer Gabel zu einem freien Knoten der anderen Partitionszelle auftaucht (es wird sozusagen nur *von einer Seite aus* gesucht). Sei $(v_1, v_2, \ldots, v_q)$ mit $\frac{q}{2} \in \mathbb{N}$ ein beliebiger augmentierender Pfad der Länge $q - 1$. Wir nehmen an, dass keiner der involvierten Knoten in einem anderen augmentierenden Pfad verwendet werden kann und kein anderer augmentierender Pfad v_1 und v_q matchen kann. Da nur von einer Seite aus gesucht wird, werden im schlimmsten Fall $\frac{q}{2}$ Passes benötigt um den Pfad zu finden, denn wenn die Kanten im Stream in umgekehrter Reihenfolge (also $\{v_{q-1}, v_q\}$, $\{v_{q-2}, v_{q-1}\}, \ldots, \{v_1, v_2\}$) eintreffen, kann pro Pass nur eine Kante sowie die dazugehörige Matching-Kante zum in v_1 wurzelnden alternierenden Baum hinzugefügt werden.
Sucht man hingegen wie in AugTrees von beiden Seiten aus nach Pfaden, kann man die Anzahl der Passes, die im Worst-Case für einen einzelnen Pfad benötigt werden, halbieren. Selbst wenn die Kanten im Stream in Reihenfolge $\{v_{\frac{q}{2}-1}, v_{\frac{q}{2}}\}$, $\{v_{\frac{q}{2}-2}, v_{\frac{q}{2}-1}\}, \ldots, \{v_1, v_2\}$, $\{v_{\frac{q}{2}+1}, v_{\frac{q}{2}+2}\}$, $\{v_{\frac{q}{2}+2}, v_{\frac{q}{2}+3}\}, \ldots, \{v_{q-1}, v_q\}$ eintreffen, können pro Pass immer noch zwei Kanten sowie die dazugehörigen Matching-Kanten zu den in v_1 und v_q wurzelnden alternierenden Bäumen hinzugefügt werden. Daraus ergeben sich maximal $\frac{q}{4}$ Passes, um einen augmentierenden Pfad der Länge q zu finden.
Obwohl die Verbesserung für Worst-Case-Instanzen vielversprechend aussieht, ist durch beidseitiges Suchen durchaus auch eine Verschlechterung im Average-Case denkbar. Da die alternierenden Bäume im Durschnitt eine geringere Tiefe erreichen, sind auch die ungültigen Teilbäume kleiner, die beim Augmentieren zum Wald hinzukommen. Deswegen sind sie auch weniger wertvoll, wenn sie später an andere Bäume angehängt werden. Ein ungültiger Teilbaum, der in Zelle A wurzelt, wird außerdem zerstückelt, wenn Knoten aus ihm an einen ebenfalls in Zelle A wurzelnden Baum angehängt werden (Abbildung 5.2). Dadurch sinkt die Wiederverwertbarkeit alter Bäume noch weiter.



(a) vorher **(b)** nachher

Abbildung 5.2: Ein gültiger alternierender Baum wird mit einer Matching-Kante eines ungültigen alternierenden Baums erweitert. Dieser zerfällt dadurch in mehrere Bäume. Matching-Kanten sind dick gedruckt, gematchte Knoten sind gefüllt. Durchgezogene Kanten gehören zu einem alternierenden Baum, gepunktete Kanten nicht. Zeilen sind Kreise, Spalten sind Quadrate.

Der zweite wesentliche Unterschied zu DapTrees liegt im Zeitpunkt der Augmentiervorgänge. In Abschnitt 3.1 haben wir bereits gesehen, dass Kliemanns Algorithmus immer erst eine Menge $\mathcal{D}$ von knotendisjunkten augmentierenden Pfaden aufbaut, bevor das Matching tatsächlich verbessert wird. Während diese Methode für den Nicht-Streaming-Fall bestens geeignet ist, da durch den vorhandenen Random-Access schnell viele Pfade gleichzeitig gefunden werden können, kommt es im Semi-Streaming Modell zu zwei grundlegenden Probleme:

1. Das Aufbauen der Pfadmenge kann mehrere Passes andauern und da die augmentierenden Pfade knotendisjunkt sein müssen, können darin enthaltene Knoten zunächst nicht erneut an alternierende Bäume angehängt werden. Wird ein augmentierender Pfad zu $\mathcal{D}$ hinzugefügt, bleibt jeder darin enthaltene Knoten solange blockiert, bis $\mathcal{D}$ eine DAP-Menge ist und das Matching augmentiert wird. Dabei ist zu bedenken, dass die erwartete Anzahl der blockierten Knoten mitunter sehr hoch sein kann, vor allem wenn es noch viele augmentierende Pfade gibt.

2. Wenn $\mathcal{D}$ eine DAP-Menge wird (also wenn der letzte dafür benötigte Pfad hinzugefügt wird), ist gegebenenfalls ein zusätzlicher Pass nötig, um zu wissen, dass kein weiterer Pfad hinzugefügt werden kann. Das liegt daran, dass (sofern nicht nur noch wenige freie Knoten vorhanden sind) eine Abbruchbedingung erfüllt sein muss, die der aus Algorithmus 5 ähnelt. Erst wenn einen gesamten Pass lang nicht nur kein augmentierender Pfad gefunden wird, sondern sich auch kein alternierender Baum verändert, weiß man sicher, dass $\mathcal{D}$ eine DAP-Menge ist [15].

Diese Aspekte legen es nahe, dass die Anzahl der Stream-Passes deutlich verringert werden kann, wenn nicht erst auf eine Menge augmentierender Pfade gewartet werden muss, sondern jeder gefundene Pfad sofort zum Verbessern des Matchings verwendet wird. Dadurch

können sogar im ersten Pass bereits Pfade verschiedener Länge gefunden werden. Genau das wurde in AugTrees realisiert.

Während in DapTrees nur augmentierende Pfade mit höchstens $2\lambda_2 + 1$ Kanten betrachtet werden, verzichtet AugTrees auf eine obere Beschränkung der Pfadlänge. Das ist für die Korrektheit des Algorithmus zwingend erforderlich, da mit einer Beschränkung unter Umständen augmentierende Pfade, die zum Erreichen der Approximationsgüte nötig sind, nicht gefunden werden. Ursache dessen ist die Tatsache, dass alternierende Bäume trotz der Distanzüberprüfung in Algorithmus 5, Z.15 eine beliebige (also weit über $2k-1$ hinausgehende) Tiefe erreichen können. Schließlich werden ungültige (Teil)Bäume vollständig an gültige Bäume angehängt, welche nach dem Augmentieren neue ungültige Teilbäume von potentiell größerer Tiefe hinterlassen. Liegen in einem augmentierenden Pfad der Länge $2k-1$ zwei oder mehr Matching-Kanten jeweils zu tief in einem alternierenden Baum, kann es zum Deadlock kommen, wie hier für $k = 3$ illustriert wird:

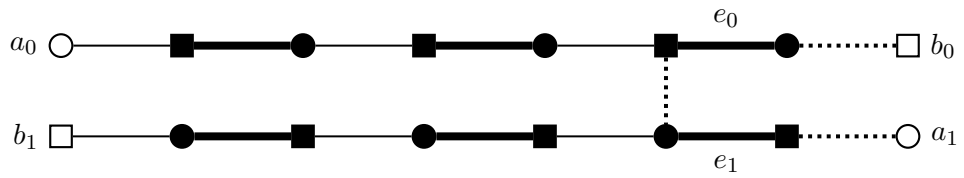


Abbildung 5.3: Deadlock für AugTrees mit Pfadlängenlimit. Der kurze augmentierende Pfad zwischen b_0 und a_1 kann nicht gefunden werden, weil e_0 und e_1 in alternierenden Bäumen blockiert sind. Matching-Kanten sind dick gedruckt, gematchte Knoten sind gefüllt. Baumkanten sind durchgezogen. Zeilen sind Kreise, Spalten sind Quadrate.

Lassen wir nur augmentierende Pfade der Höchstlänge $2k - 1 = 5$ zu, können die Pfade zwischen a_0 und b_0 sowie zwischen a_1 und b_1 nicht verwendet werden, weil sie zu lang sind. Dadurch bleiben die Kanten e_0 und e_1 innerhalb ihrer gültigen Bäume blockiert, sodass der kurze augmentierende Pfad zwischen a_1 und b_0 nicht gefunden werden kann. Aus diesem Grund gibt es in AugTrees für augmentierende Pfade kein Limit der Länge, wodurch blockierende Szenarios wie das oben beschriebene nicht auftreten können.

5.5 Details zur Implementierung

Eines der wichtigsten Ziele des Semi-Streaming Modells ist das effiziente Arbeiten mit möglichst wenig Arbeitsspeicher. Bei der Implementierung von AugTrees wurde dementsprechend besonders auf den Speicherplatzbedarf geachtet. Knoten werden nicht durch Objekte repräsentiert, sondern durch 32-bit Integer-Werte zwischen 0 und $n-1$, wodurch bis zu etwa 2^{32} Knoten eindeutig adressiert werden können. Jeweils in Integer-Arrays der Größe n speichern wir zu jedem Knoten v seinen Matching-Partner $m(v)$, seinen Vorgänger $p(v)$,

die Wurzel $r(v)$ des gültigen alternierenden Baums, welcher v enthält, sowie die Tiefe $l(v)$. Ist v nicht in einem gültigen alternierenden Baum enthalten, enthält $r(v)$ die Wurzel des letzten gültigen Baums, in welchem v enthalten war. Ein Array von Integer-Listen wird verwendet, um die Nachfolgermengen $S(v)$ im Speicher zu halten. Um den Wert `null` zu repräsentieren, wird eine Integer-Konstante mit Wert n eingeführt.

5.6 Theoretische Schranken

5.6.1 Passes

In Abschnitt 5.4 haben wir gesehen, dass der erwartete Pass-Count für AugTrees geringer ist, als für den ähnlichen DAP-Algorithmus. Die in [15] für den Worst-Case angegebenen $\mathcal{O}(kn)$ Passes können wir dennoch nicht unterschreiten. Wir werden sogar Graphen sehen, für die bei ungünstiger Kantenreihenfolge im Stream $\Omega(n)$ Passes benötigt werden.
Dass Algorithmus 5 höchstens $\mathcal{O}(kn)$ mal über den Stream läuft, wird schnell ersichtlich. Wie bereits in Abschnitt 5.4 beobachtet werden zum Finden eines augmentierenden Pfades der Länge $q \leq k$ höchstens $\frac{q}{4} = \mathcal{O}(k)$ Passes benötigt. Somit ergibt sich bereits die Pass-Schranke von $\mathcal{O}(kn)$, um die theoretisch möglichen $\max(|A|, |B|) = \mathcal{O}(n)$ Pfade zu finden. Es werden genau dann viele Passes benötigt, wenn zu jedem Zeitpunkt für jede Partition nur wenige alternierende Bäume aufgebaut werden und diese, nachdem durch sie ein augmentierender Pfad gefunden wird, keine wiederverwertbaren Teilbäume hinterlassen. Beides ist in diesem Beispielgraphen der Fall:

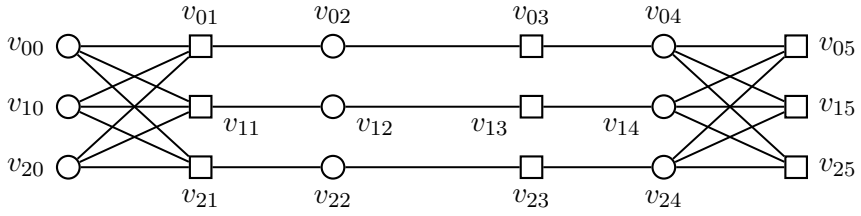


Abbildung 5.4: Ein Worst-Case-Graph für AugTrees (1/3). Zeilen sind Kreise, Spalten sind Quadrate.

Um es dem Algorithmus möglichst schwer zu machen, wählen wir als Kantenreihenfolge im Stream:

$$
\begin{array}{lll}
\{v_{01}, v_{02}\}, & \{v_{11}, v_{12}\}, & \{v_{21}, v_{22}\}, \\
\{v_{03}, v_{04}\}, & \{v_{13}, v_{14}\}, & \{v_{23}, v_{24}\}, \\
\{v_{02}, v_{03}\}, & \{v_{12}, v_{13}\}, & \{v_{22}, v_{23}\}, \\
\\
\{v_{00}, v_{01}\}, & \{v_{00}, v_{11}\}, & \{v_{00}, v_{21}\}, \\
\{v_{10}, v_{01}\}, & \{v_{10}, v_{11}\}, & \{v_{10}, v_{21}\}, \\
\{v_{20}, v_{01}\}, & \{v_{20}, v_{11}\}, & \{v_{20}, v_{21}\}, \\
\\
\{v_{04}, v_{05}\}, & \{v_{14}, v_{05}\}, & \{v_{24}, v_{05}\}, \\
\{v_{04}, v_{15}\}, & \{v_{14}, v_{15}\}, & \{v_{24}, v_{15}\}, \\
\{v_{04}, v_{25}\}, & \{v_{14}, v_{25}\}, & \{v_{24}, v_{25}\}
\end{array}
$$

Durch die ersten neun Kanten werden zu Beginn des ersten Passes sechs Kanten direkt gematcht. Anschließend tauchen zuerst alle zu v_{00} adjazenten Kanten auf, sodass der in v_{00} wurzelnde alternierende Baum allen anderen Wurzeln aus der selben Partitionszelle alle Erweiterungsmöglichkeiten abschneidet. Auf dieselbe Weise erscheinen auch die zu v_{05} adjazenten Kanten früh genug, sodass alle in der anderen Partitionszelle wurzelnden Bäume nicht mehr erweitert werden können. Nach dem ersten Pass ergibt sich folgende Situation:

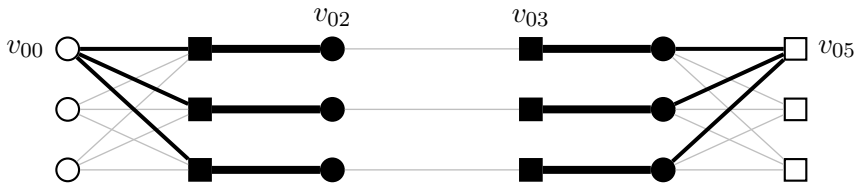


Abbildung 5.5: Ein Worst-Case-Graph für AugTrees (2/3). Schwarze Kanten gehören zu alternierenden Bäumen, dick gedruckte Kanten sind Matching-Kanten. Zeilen sind Kreise, Spalten sind Quadrate.

Im nächsten Pass findet die erste Operation beim Lesen der Kante $\{v_{02}, v_{03}\}$ statt, welche einen Augmentiervorgang erlaubt. Danach werden durch v_{10} und v_{15} wieder alle anderen Wurzeln von allen Erweiterungsmöglichkeiten abgeschnitten:

Abbildung 5.6: Ein Worst-Case-Graph für AugTrees (3/3). Schwarze Kanten gehören zu alternierenden Bäumen, dick gedruckte Kanten sind Matching-Kanten. Zeilen sind Kreise, Spalten sind Quadrate.

Selbst für diese sehr kleine Beispielinstanz werden bereits drei Passes zur Berechnung des Matchings (sowie ein Pass zum Erfüllen der Abbruchbedingung) benötigt, um für $k \geq 3$ ein Ergebnis zu erhalten. Dabei lässt sich das Beispiel auf beliebige Größe erweitern: Die äußeren Knoten jeder „Zeile“ des Graphen können erst dann gematcht werden, wenn alle vorherigen Zeilen bereits vollständig gematcht wurden. Dazu wird pro Zeile ein Pass benötigt und es können unbegrenzt viele Zeilen hinzugefügt werden. Um einen Graphen mit n Knoten (und somit $\frac{n}{6}$ Zeilen) zu verarbeiten, muss der Stream demnach $\frac{n}{6}+1 = \Omega(n)$-mal gelesen werden. Auf ähnliche Weise wird in [16] gezeigt, dass auch der DAP-Algorithmus mit einseitiger Suche auf einigen Graphen $\Omega(n)$ Passes benötigt.
Damit hat AugTrees einen schlechteren Worst-Case-Pass-Count, als die in [19] und [11] gezeigten Algorithmen, deren Pass-Count völlig unabhängig von n ist. Die experimentellen Ergebnisse, die sich im Rahmen der Evaluierung ergeben haben, sprechen jedoch für sich und können mit anderen Algorithmen problemlos konkurrieren [15]. Da wir allerdings gesehen haben, dass es durchaus Worst-Case-Instanzen gibt, für welche sehr viele Passes benötigt werden, sei an dieser Stelle auf eine in [16] gezeigte Methode hingewiesen. Anstatt nur einen Algorithmus auszuführen, kann parallel ein zweiter verwendet werden, welcher im Durchschnitt zwar mehr, dafür aber mit Sicherheit konstant viele Passes braucht (z.B. [11] oder [19]). Sobald einer der beiden Algorithmen terminiert, hat man das gewünschte Ergebnis. Auf diese Weise kann der gute Average-Case-Pass-Count von AugTrees mit dem guten Worst-Case-Pass-Count anderer Algorithmen kombiniert werden.

5.6.2 Laufzeit

Obwohl diese Arbeit sich hauptsächlich mit der Anzahl der benötigten Passes beschäftigt, werfen wir an dieser Stelle auch einen Blick auf die theoretische Laufzeit von Algorithmus 5, auf welchen sich im Folgenden alle Zeilenangaben beziehen. Genauer gesagt betrachten wir hier die Laufzeit pro eintreffender Stream-Kante, da alle Anweisungen, die außerhalb der For-Schleife in Z.6–28 aufgerufen werden, entweder konstante Zeit benötigen oder linear von der Anzahl der Passes abhängen. Je nach Kante wird folgende Laufzeit benötigt:

$\mathcal{O}(1)$, falls keine Operation möglich ist.
Es werden höchsten Z.7, 14 und 25 ausgeführt. Um herauszufinden, ob ein Knoten v gültige Gabel ist, müssen lediglich $m(v) \neq \texttt{null}$ und $p(v) = m(v)$ überprüft werden. Dafür wird konstante Zeit benötigt.

$\mathcal{O}(l)$, falls ein augmentierender Pfad P der Länge l gefunden wird.
Das Matching mit P zu augmentieren dauert $\mathcal{O}(l)$, da insgesamt für $l+1$ Knoten der Wert $m(v)$ geändert werden muss (Z.13). Um diese Knoten jedoch aus ihrem Baum zu entfernen (Z.12), müssen alle Nachfolger von ihnen getrennt werden. Theoretisch müssten dazu die Vorgänger aller dieser Nachfolger auf `null` gesetzt werden. Bei einer wie in Abschnitt 5.5 beschriebenen Implementierung ist dies allerdings

nicht nötig: Da jeder Knoten v die Wurzel $r(v)$ seines letzten gültigen alternierenden Baums kennt, kann auch ohne den Vorgängerwert $p(v)$ herausgefunden werden, dass v kein gültiger Gabelknoten ist. Dazu muss lediglich überprüft werden, ob $r(v)$ gültig ist. Deswegen müssen die Vorgängerwerte nur in gültigen Bäumen korrekt sein, nicht aber in abgetrennten ungültigen Bäumen. Somit genügt es beim Entfernen der Knoten $V(P)$ jeweils die Nachfolgermenge zu leeren, was pro Knoten konstante Zeit (insgesamt also $\mathcal{O}(l)$) benötigt.

Es ließe sich sogar argumentieren, dass die amortisierte Zeit für einen Augmentiervorgang bei $\mathcal{O}(1)$ liegt, da vor dem Augmentieren eines Pfades der Länge l immer erst $\Omega(l)$ Erweiterungsoperationen stattgefunden haben müssen, auf welche man die Laufzeit von $\mathcal{O}(l)$ verteilen könnte. Da wir aber in der Geschwindigkeit, mit der wir Kanten verarbeiten, stark von der Geschwindigkeit des Speichermediums abhängen, können wir die Verarbeitungsdauer von zeitaufwändigen Kanten nur dann amortisieren, wenn eine geeignete Pufferlösung das kontinuierliche Lesen vom Speichermedium sicherstellt.

$\mathcal{O}(1)$, falls eine Erweiterungsoperation möglich ist.

Das Überprüfen der Bedingungen in Z.7 und 14–16 benötigt konstante Zeit. Auch die Dauer für das unter Umständen nötige Entfernen von b und $m(b)$ aus ihrem Baum ist konstant (Z.18, gleiche Begründung wie im vorherigen Punkt). Für das Hinzufügen von Kanten zu Bäumen (Z.19) müssen immer lediglich zwei Nachfolgermengen um einen Knoten erweitert und zwei Vorgänger neu gesetzt werden. Bei Verwendung einer geeigneten Datenstruktur für Nachfolgermengen ist die Zeit dafür konstant.

$\mathcal{O}(|A \cup B|)$, falls eine Umhängeoperation möglich ist.

Das Überprüfen der Bedingungen in Z.7, 14–16 und 20 benötigt konstante Zeit. Um einen Teilbaum abzutrennen und wieder anzufügen, müssen nur zwei Nachfolgermengen und Vorgängerwerte verändert werden. Anschließend ist allerdings sowohl die Tiefe $l(v)$ also auch der Wurzelwert $r(v)$ für alle umgehängten Knoten falsch. Um dies zu korrigieren, muss der gesamte Teilbaum traversiert werden, und die Laufzeit ist linear von der Größe des Teilbaums abhängig, welche durch $O(|A \cup B|)$ beschränkt ist. Vermutlich könnte die Laufzeit durch Versionsnummern (vergleiche mit Abschnitt 4.2.1) verbessert werden.

Die Worst-Case-Laufzeit ist mit $\mathcal{O}(|A \cup B|)$ pro Kante also vergleichsweise hoch. Es ist allerdings zu bedenken, dass man in der Realität mit deutlich besseren Werten rechnen kann. Zum einen macht die Menge aller Kanten, bei denen tatsächlich eine Umhängeoperation ausgeführt wird, nur einen Bruchteil der Gesamtmenge aus. Das liegt an den zahlreichen Bedingungen, die vor dem Aufruf erfüllt sein müssen. Zum anderen ist es sehr

unwahrscheinlich, dass ein um- oder anzuhängender Teilbaum tatsächlich eine mit $|A \cup B|$ vergleichbare Größe erreicht. Wir gehen an dieser Stelle nicht weiter ins Detail, werden in der Evaluierung jedoch sehen, wie schnell AugTrees Kanten verarbeiten kann.

5.6.3 Speicherplatzbedarf

Wir gehen von einer wie in Abschnitt 5.5 beschriebenen Implementierung aus, legen uns jedoch nicht auf 32bit Integer-Werte fest. Alle Knoten werden nummeriert, sodass sie eindeutig identifiziert werden können. Dazu fallen pro Knoten $\lceil \log_2(n) \rceil = \mathcal{O}(\log(n))$ Bits an. Hinzu kommen (ebenfalls pro Knoten):

1. $\mathcal{O}(\log(n))$ Bits, um den Matching-Partner $m(v)$ zu speichern
2. $\mathcal{O}(\log(n))$ Bits, um den Vorgänger $p(v)$ zu speichern
3. $\mathcal{O}(\log(n))$ Bits, um in der Nachfolgermenge $S(p(v))$ des Vorgängers gespeichert zu werden
4. $\mathcal{O}(\log(n))$ Bits, um die Wurzel $r(v)$ des alternierenden Baums zu speichern, in welchem sich der Knoten befindet
5. $\mathcal{O}(\log(n))$ Bits, um die Tiefe $l(v)$ zu speichern, welche durch $\mathcal{O}(n)$ beschränkt ist
6. $\mathcal{O}(1)$ Bits, um zu speichern, ob der Knoten eine gültige Wurzel ist

Dadurch reichen pro Knoten $\mathcal{O}(\log(n))$ Bits aus und der gesamte Speicherplatzbedarf liegt mit $\mathcal{O}(n \cdot \log(n)) \subseteq \mathcal{O}(n \cdot \text{polylog}(n))$ innerhalb des vom Semi-Streaming Modell vorgeschriebenen Rahmens.

Kapitel 6

Evaluierung

Eine ausführliche Evaluierung soll die Praxistauglichkeit von AugTrees zeigen. Dazu werden verschiedene Testszenarien betrachtet.

6.1 Vergleich der baumbasierten Algorithmen

Da AugTrees und DapTrees sich sehr ähneln, steht ein direkter Vergleich der beiden Algorithmen im Vordergrund der Evaluierung. Dabei ist vor allem der Pass-Count von Interesse, also die Anzahl der Male, die der Stream gelesen werden muss, bis der jeweilige Algorithmus terminiert. Wir rechnen also insbesondere immer alle Passes mit ein, die nach dem Erreichen der gewünschten Approximationsgüte benötigt werden, damit die Abbruchbedingung des Algorithmus wahr wird.

6.1.1 Graphklassen

Wir betrachten verschiedene Graphklassen, für die jeweils ein entsprechender Graphgenerator implementiert wurde. Die Klassen sind die gleichen wie in [15].

rand [15]

Parameter: $n \in \mathbb{N}$, $p \in [0, 1]$. Instanzen vom Typ ***rand*** sind komplett zufällige bipartite Graphen. Zunächst wird festgelegt, wie viele Knoten in A bzw. B liegen. Dazu werden n zufällige Wahrheitswerte gezogen, wobei für jedes *true* ein Knoten zu A hinzugefügt wird, für jedes *false* entsprechend zu B. Anschließend kommt jede mögliche Kante $\{a, b\} \in \{\{a, b\} \mid a \in A \wedge b \in B\}$ mit Wahrscheinlichkeit p zu E hinzu. In der Praxis dauert es zu lange, über alle theoretisch möglichen Kanten zu iterieren, dabei jedes mal eine Zufallszahl zu ziehen und anhand dieser zu entscheiden, ob die Kante zum Graphen hinzugefügt wird. Stattdessen verwenden wir folgende deutlich effizientere Methode:
Es sei $A = \{a_1, \ldots, a_k\}$ und zu Beginn $d(i) = 0$ für $1 \leq i \leq k$. Nun werden $m = \lceil |A| \cdot |B| \cdot p \rceil$ Zufallszahlen zwischen 1 und k gezogen, wobei das Auftreten der Zufallszahl

i den Wert $d(a_i)$ um 1 erhöht. Dabei darf $d(a_i)$ nicht größer werden als $|B|$. Anschließend ziehen wir für jedes $a_i \in A$ insgesamt $d(a_i)$ zufällige Knoten aus B und fügen für jeden gezogenen Knoten b die Kante $\{a_i, b\}$ zum Graphen hinzu. Der so entstehende Graph hat genau m Kanten.

degm [15]

Parameter: $n \in \mathbb{N}$, $p \in [0, 1]$. Für *degm* wird genau wie bei *rand* bestimmt, wie viele Knoten in A und in B liegen sollen. Der erwartete Grad der Knoten in Zelle $A = \{a_1, \ldots, a_k\}$ ist linear vom Knotenindex abhängig. Um das zu erreichen, wird je nach gewünschter Kantenzahl ein Parameter $p \in [0, 1]$ gewählt. Daraus ergibt sich der Grad $d(a_i) = \left\lceil \frac{|B|}{k} \cdot p \cdot i \right\rceil$, anhand dessen genau wie in *rand* zufällige Kanten gezogen werden, die zum Graphen hinzukommen.

rope [7]

Parameter: $n \in \mathbb{N}$, $l \in \mathbb{N}$ mit $\frac{n}{l} \in \mathbb{N}$, $p \in [0, 1]$. Beide Partitionszellen sind gleich groß und werden jeweils in l Gruppen der Größe k aufgeteilt: $|A| = |B| = \frac{n}{2} = lk$. Für $1 \leq i \leq l$ sei $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,k}\}$ die i-te Gruppe in A (analog für B). Für alle A_i und B_i werden die Kanten $\{\{a_{i,j}, b_{i,j}\} \mid 1 \leq j \leq k\}$ zum Graphen hinzugefügt. Der Teilgraph zwischen A_i und B_i ist also ein perfektes Matching. Für $1 < i \leq l$ wird außerdem jede mögliche Kante zwischen A_i und B_{i-1} mit Wahrscheinlichkeit p zum Graphen hinzugefügt. Der Teilgraph zwischen A_i und B_{i-1} ist also ein zufälliger bipartiter Graph, der mit dem *rand*-Generator erzeugt werden kann. Somit ist jede *rope*-Instanz eine Abfolge von Gruppen, die immer abwechselnd mit einem perfekten Matching und einem zufälligen Graphen untereinander verbunden sind. Dadurch gibt es immer ein einzigartiges perfektes Matching, welches im folgenden Beispiel hervorgehoben ist.

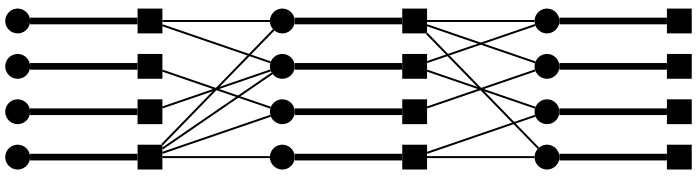


Abbildung 6.1: Eine *rope*-Instanz mit Eingabeparametern $n = 24$, $l = 3$ und $p = 0.5$. Zeilen sind Kreise, Spalten sind Quadrate. Matching-Kanten des einzigartigen perfekten Matchings sind dick gedruckt.

rgb [7][15]

Parameter: $n \in \mathbb{N}$, $l \in \mathbb{N}$ mit $\frac{n}{l} \in \mathbb{N}$, $p \in [0, 1]$. Auch bekannt als *fewg* mit $l = 32$ und *manyg* mit $l = 256$. Die Partitionszellen werden jeweils in l Gruppen der Größe k geteilt, wobei A_i und B_i genauso definiert werden wie für *rope*-Instanzen. Jede mögliche Kante

zwischen der Gruppe A_i und den Gruppen B_i, $B_{((i-1) \mod l)}$ und $B_{((i+1) \mod l)}$ wird mit Wahrscheinlichkeit p zum Graphen hinzugefügt. Das bedeutet, dass in `rgb`-Instanzen jede Gruppe aus A durch zufällige bipartite Graphen mit den drei „nächstgelegenen“ Gruppen aus B verbunden ist. Diese Graphen können z.B. mithilfe des `rand`-Generators erzeugt werden.

`hi-lo` [7]

Parameter: $n \in \mathbb{N}, l \in \mathbb{N}$ mit $\frac{n}{l} \in \mathbb{N}, p \in [0, 1]$. Die Partitionszellen werden jeweils in l Gruppen der Größe k geteilt, wobei A_i und B_i genauso definiert werden wie für `rope`-Instanzen. Wir legen fest, dass $d := \lceil max(1, p \cdot k) \rceil$) gilt. Für $1 \leq i \leq l$ sei $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,k}\}$ die i-te Gruppe in A (analog für B). Wir fügen alle Kanten $\{a_{i,j}, b_{i,q}\}$ mit $1 \leq j \leq k$ und $\max(0, j - d) < q \leq i$ zum Graphen hinzu. Für $i < l$ fügen wir außerdem alle Kanten $\{a_{i,j}, b_{i+1,q}\}$ mit $1 \leq j \leq k$ und $\max(0, j - d) < q \leq i$ hinzu. In der so entstehenden `hi-lo`-Instanz ist also jeder Knoten $a_{i,j}$ aus A_i mit den bis zu d Knoten aus B_i bzw. B_{i+1} verbunden, deren Index innerhalb ihrer Gruppe kleiner gleich j, aber größer als $j - d$ ist. Somit enthalten `hi-lo`-Instanzen kein Zufallselement, sondern sind in Abhängigkeit von den Eingabeparametern genau definiert. Des weiteren enthalten sie immer ein einzigartiges perfektes Matching [7], welches im folgenden Beispiel hervorgehoben ist.

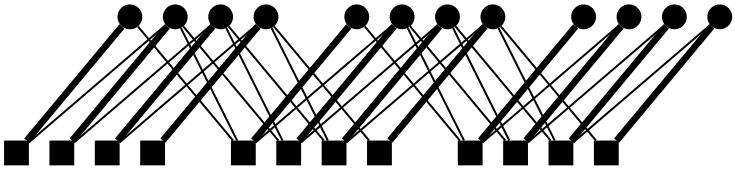


Abbildung 6.2: Eine `hi-lo`-Instanz mit Eingabeparametern $n = 24$, $l = 3$ und $p = 0.5$, woraus $d = 2$ folgt. Zeilen sind Kreise, Spalten sind Quadrate. Matching-Kanten des einzigartigen perfekten Matchings sind dick gedruckt.

Obwohl die gruppenbasierten Klassen `hi-lo`, `rgb` und `rope` spezielle Sonderfälle sind, die sich in echten Datensätzen wenn überhaupt nur selten ergeben, spiegeln sie dennoch die Verhältnisse in vielen realen Netzwerken wieder, vor allem da Kanten nur zwischen „nahe zusammenliegenden“ Gruppen bestehen. Betrachten wir etwa soziale Netzwerke, so sind die meisten Nutzer überwiegend mit Menschen aus ihrer unmittelbaren geographischen Umgebung befreundet. Deswegen sind diese Klassen in der Evaluierung trotz ihrer strengen Kriterien durchaus von Interesse. Darüber hinaus sind Matchings in `rope` und `hi-lo` für sowohl AugTrees als auch DapTrees besonders schwer zu finden. Zwar werden in diesen Instanzen bereits durch Greedy (und somit auch durch den ersten Pass der beiden Streaming-Algorithmen) viele Knoten gematcht (etwa 90% für `rope` mit $l = \frac{n}{6}$ [7]), soll jedoch eine höhere Approximationsgüte erreicht werden, sind oftmals sehr lange augmentierende Pfade erforderlich. Das liegt daran, dass nur sehr wenige der zum einzig-

artigen perfekten Matching gehörenden Kanten von GREEDY gematcht werden, wodurch insbesondere viele Knoten der ersten Gruppe in A und der letzten Gruppe in B frei bleiben. Jeder augmentierende Pfad zwischen zwei dieser Knoten muss alle anderen Gruppen durchlaufen und hat somit die Länge $\Omega(l)$. Dementsprechend ist zu erwarten, dass der maximale Pass-Count für *rope* und *hi-lo* besonders hoch ausfällt.

6.1.2 Testaufbau

Nach Rücksprache mit dem Autor von [15] wurde der Aufbau des folgenden Testszenarios so konzipiert, dass ein möglichst genauer Vergleich mit den dort präsentierten Evaluierungsergebnissen möglich ist. Des Weiteren soll ersichtlich werden, ob die Anzahl der benötigten Passes von n abhängt. Für alle Instanzen wird ein oberes Limit von $E_{max} = 10^9$ Kanten sowie eine Maximaldichte von $D_{max} = 0.1$ festgelegt. Die geringste erlaubte Dichte ist $D_{min} = 10^{-6}$.
Es sei n die gewünschte Anzahl von Knoten für die zu generierenden Graphen. Für die drei gruppenbasierten Graphklassen (*hi-lo*, *rgb*, *rope*) werden jeweils 64 zufällige l-Parameter gezogen. In Abhängigkeit von n wird zu jeder Kombination aus Klasse und l-Parameter p_{max} berechnet, welches der größtmögliche Wert für p ist, durch welchen generierte Graphen weder D_{max} noch E_{max} überschreiten. Analog dazu ist p_{min} der kleinste Wert für p, durch welchen D_{min} nicht unterschritten wird. Es werden jeweils 4 verschiedene Parameterwerte für p gleichmäßig aus dem Intervall $[p_{min}, p_{max}]$ gewählt. Anhand von n, l und p werden pro Graphklasse jeweils $64 \cdot 4 = 256$ Instanzen generiert.
Für *rand* und *degm* werden p_{max} und p_{min} in Abhängigkeit von n bestimmt. Insgesamt 16 mögliche Werte für den Parameter p werden gleichmäßig aus dem Intervall $[p_{min}, p_{max}]$ gewählt. Pro p-Wert und Graphklasse werden 16 zufällige Instanzen generiert, sodass sich pro Klasse und n insgesamt $16 \cdot 16 = 256$ Instanzen ergeben. In der ursprünglichen Evaluierung von DAPTREES war die Anzahl der Instanzen doppelt so hoch, da jede Instanz zusätzlich mit vertauschten Partitionszellen getestet wurde [15]. Dies ist für AUGTREES nicht nötig, da beidseitig gesucht wird (siehe Abschnitt 5.4) und das Vertauschen der Partitionszellen ohne Effekt bleiben würde.
Wir betrachten 10 verschiedene Werte für n mit jeweils 256 Graphinstanzen pro Graphklasse, was insgesamt 12 800 Instanten ergibt. Für die Tests werden zunächst alle Kanten im Arbeitsspeicher gehalten und das Streamingszenario nur simuliert. Die Reihenfolge der Kanten im Stream ist pro Instanz zufällig gewählt, bleibt jedoch für alle Passes gleich.

6.1.3 Ergebnisse

Tabelle 6.1 zeigt die durschnittlichen Pass-Counts, die von AUGTREES und DAPTREES benötigt wurden. Der neue Algorithmus schneidet in fast allen Fällen besser ab und braucht nur für *rgb* und $n = 6 \cdot 10^5$ und $n = 8 \cdot 10^5$ minimal mehr Passes. Besonders nennenswert

Tabelle 6.1: Durchschittliche Pass-Counts, die AugTrees benötigt hat, um eine 0.9-Approximation zu berechnen. Die Pass-Counts von DapTrees sind in Klammern angegeben. Pro Wert von n wurden 256 Graphinstanzen mit $D_{max} = 10^{-1}$, $D_{min} = 10^{-6}$ und $E_{max} = 10^{9}$ ausgewertet, die wie in Abschnitt 6.1.3 beschrieben generiert wurden.

	Durchschnittlicher Pass-Count									
n	*rand*		*degm*		*hi-lo*		*rgb*		*rope*	
100 000	2.0	(2.5)	2.0	(3.2)	9.6	(35.0)	3.0	(5.1)	6.6	(19.8)
200 000	2.1	(2.5)	2.0	(2.8)	10.1	(37.6)	3.2	(4.7)	6.6	(19.1)
300 000	2.1	(2.5)	2.1	(2.9)	10.3	(38.6)	3.2	(3.9)	7.0	(18.2)
400 000	2.1	(2.5)	2.1	(2.9)	10.2	(36.3)	3.3	(5.3)	7.0	(15.6)
500 000	2.1	(2.5)	2.1	(3.0)	10.1	(36.7)	3.3	(4.4)	7.1	(19.4)
600 000	2.2	(2.5)	2.1	(3.5)	10.3	(38.4)	3.5	(3.3)	7.2	(18.1)
700 000	2.2	(2.5)	2.1	(3.6)	10.0	(37.4)	3.5	(3.9)	7.4	(18.5)
800 000	2.2	(2.5)	2.1	(3.5)	10.2	(37.9)	3.4	(3.1)	7.2	(16.2)
900 000	2.2	(2.6)	2.1	(3.3)	10.2	(37.0)	3.4	(3.7)	7.6	(14.5)
1 000 000	2.2	(2.5)	2.2	(3.1)	10.2	(33.4)	3.7	(4.6)	7.3	(18.2)
∅	2.2	(2.5)	2.1	(3.2)	10.1	(36.8)	3.4	(4.2)	7.1	(17.8)
	88.0%		65.6%		27.4%		81.0%		39.9%	

sind die Ergebnisse auf *hi-lo*, wo der Pass-Count von AugTrees für alle Instanzen im Durchschnitt nur etwas über einem Viertel und für jedes n unter einem Drittel der von DapTrees benötigten Passes liegt. Auch auf *rope* konnten die Ergebnisse im Schnitt mehr als halbiert werden.
Ein Blick auf Tabelle 6.2 verrät, dass AugTrees auch bezüglich des maximalen Pass-Counts klar überlegen ist. Lediglich für *rand* und 3 verschiedene n-Werte konnte DapTrees nicht geschlagen werden, wobei für alle anderen Klassen im Durchschnitt nur 18.4% bis 35% der von DapTrees benötigten Passes angefallen sind. Die größten Verbesserungen sind in den *rope*-Instanzen zu finden, wo die Werte aus [15] für jedes n mindestens gefünftelt werden.

Leider ist der Pass-Count nicht völlig unabhängig von n, wie die steigende Tendenz der Werte in jeder Spalte von Tabelle 6.1 zeigt. Es scheint sich aber um eine sehr geringe Abhängigkeit zu handeln, da für die meisten Klassen das Verzehnfachen von n von 10^5 auf 10^6 zu einem lediglich 10% höheren Pass-Count führt. In den maximal benötigten Pass-Counts (Tabelle 6.2) tritt diese Tendenz insgesamt weniger eindeutig auf, ist dafür aber für *rand* umso klarer zu erkennen. Die beobachtete Abhängigkeit stimmt mit den theoretischen Ergebnissen aus Kapitel 5 überein, in welchem wir bereits gesehen haben, dass AugTrees auf einigen Graphinstanzen $\Omega(n)$ Passes benötigt.

Tabelle 6.2: Maximale Pass-Counts, die AugTrees benötigt hat, um eine 0.9-Approximation zu berechnen. Die Pass-Counts von DapTrees sind in Klammern angegeben. Pro Wert von n wurden 256 Graphinstanzen mit $D_{max} = 10^{-1}$, $D_{min} = 10^{-6}$ und $E_{max} = 10^9$ ausgewertet, die wie in Abschnitt 6.1.3 beschrieben generiert wurden.

	Maximaler Pass-Count									
n	*rand*		*degm*		*hi-lo*		*rgb*		*rope*	
100 000	2	(3)	2	(8)	13	(53)	10	(30)	12	(62)
200 000	3	(3)	2	(7)	14	(56)	9	(31)	13	(63)
300 000	3	(3)	3	(7)	14	(55)	8	(29)	12	(64)
400 000	3	(3)	3	(8)	14	(56)	9	(33)	10	(63)
500 000	4	(3)	3	(7)	14	(58)	9	(34)	11	(64)
600 000	4	(3)	3	(9)	14	(58)	9	(30)	12	(64)
700 000	4	(6)	3	(9)	15	(56)	9	(35)	10	(62)
800 000	5	(3)	3	(8)	14	(58)	9	(31)	11	(63)
900 000	5	(7)	3	(8)	14	(61)	9	(32)	12	(62)
1 000 000	5	(6)	3	(9)	14	(60)	10	(34)	13	(65)
∅	3.8	(4.0)	2.8	(8.0)	14.0	(57.1)	9.1	(31.9)	11.6	(63.2)
	95.0%		35.0%		24.5%		28.5%		18.4%	

6.1.4 Tests in dünneren Graphen

In [15] befinden sich zusätzlich Testergebnisse für Instanzen von geringerer Dichte. Obwohl DapTrees dort für einige Graphklassen deutlich mehr Passes benötigt (*rand* und *degm* mit bis zu über 40 Passes im Maximum), verzichten wir an dieser Stelle auf vergleichbare Tests. Grund dafür ist, dass Graphen mit einer geringen Dichte nicht im Semi-Streaming Modell ausgewertet werden sollten, da der für die Bäume benötigte Platz im Arbeitsspeicher nicht wesentlich geringer ist, als der für den gesamten Graphen benötigte. Wenn die Dichte eingangs nicht bekannt ist, kann nach folgendem Schema verfahren werden:
Im ersten Pass wird der Streaming-Algorithmus ausgeführt, wobei parallel dazu die Anzahl aller eintreffenden Kanten gezählt wird. Ergibt sich am Ende des Passes, dass nicht $|E| = \mathcal{O}(n)$ gilt, wird unverändert mit dem Streaming-Algorithmus fortgefahren. Der Rechenaufwand für das Zählen der Kanten ist dann praktisch zu vernachlässigen und die Gesamtleistung ist nicht schlechter als sonst. Ergibt sich am Ende des ersten Passes jedoch, dass $|E| = \mathcal{O}(n)$ gilt, werden alle bisher vom Algorithmus belegten Ressourcen freigegeben. Jeder Knoten erhält eine Adjazenzliste im Arbeitsspeicher, welche im zweiten Pass nach und nach gefüllt wird. Ist der Pass vollendet, befindet sich der gesamte Graph im Speicher und ein beliebiger Nicht-Streaming-Algorithmus kann verwendet werden, um das gewünschte Matching zu berechnen. Einzige Voraussetzung für diesen Algorithmus

ist, dass er mit Adjazenzlisten funktioniert. Die durch die beiden Passes entstehenden Kosten sind dabei vergleichsweise gering, vor allem wenn sonst ein hoher Pass-Count nötig gewesen wäre.

6.2 Weitere Tests

6.2.1 Verschiedene Approximationsgüten

Nachdem sich bereits eine leichte Abhängigkeit des Pass-Counts von n gezeigt hat, wollen wir auch die Abhängigkeit von k betrachten. Je nach gewähltem k erhält man die Approximationsgüte $f(k) = \frac{k}{k+1}$. Soll eine x-Approximation erreicht werden, können wir zur Wahl von k die Umkehrfunktion von f verwenden und erhalten $k = \lceil f^{-1}(x) \rceil = \left\lceil \frac{x}{1-x} \right\rceil$. Es ist zu erwarten, dass die Pass-Counts von AugTrees linear von k abhängen.
Der erste Test findet auf *hi-lo*-Instanzen statt, da diese Graphklasse für AugTrees in den vorherigen Tests am schwierigsten zu verarbeiten war. Es ist $n = 5 \cdot 10^5$ und für l werden 11 verschiedene Teiler von n gewählt (4, 10, 25, 80, 200, 500, 1250, 3125, 10000, 25000 und 62500). Genau wie in Abschnitt 6.1.3 sind $D_{min} = 10^{-6}$, $D_{max} = 10^{-1}$ und $E_{max} = 10^9$ gegeben und es wird pro Wert von l jeweils p_{min} und p_{max} bestimmt, sodass 4 mögliche Werte für p gleichmäßig aus dem Intervall $[p_{min}, p_{max}]$ gewählt werden können. Das ergibt $p \cdot l = 44$ Graphinstanzen mit unterschiedlichen Gruppengrößen und Dichten.

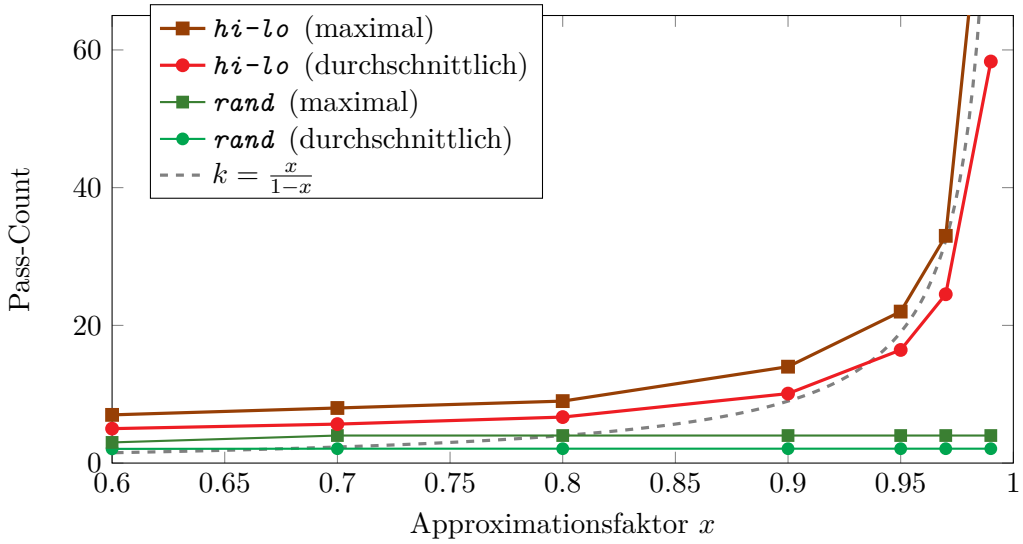


Abbildung 6.3: Abhängigkeit zwischen den von AugTrees benötigten Pass-Counts und der Approximationsgüte. Es werden 44 verschiedene *hi-lo*-Instanzen und 64 verschiedene *rand*-Instanzen mit $n = 5 \cdot 10^5$ und bis zu 10^9 Kanten verwendet.

Von AugTrees lassen wir in jeder dieser Instanzen 7 Matchings unterschiedlicher Approximationsgüten zwischen $x = 0.6$ und $x = 0.99$ berechnen. Die in Abbildung 6.3 zu

sehenden Ergebnisse decken sich mit den theoretischen Vorüberlegungen. Wie erwartet ist der Zusammenhang zwischen k und sowohl den durchschnittlichen als auch den maximalen Pass-Counts stark. Ab einer Approximationsgüte von 95% wächst k und somit auch die Anzahl der benötigten Passes extrem schnell. Während für $x = 0.95$ der Stream im Durchschnitt nur knapp 16.5 Mal gelesen werden muss, sind für $x = 0.99$ fast vier Mal so viele Passes nötig. Positiv fällt auf, dass die Lücke zwischen durchschnittlichen und maximalen Werten unabhängig von der Approximationsgüte eher schmal ist. Dabei liegt das Maximum immer nur etwa 50% über dem Durchschnitt.
Der zweite Test mit *rand*-Instanzen zeigt ein deutlich anderes Bild. Zu 16 verschiedenen Dichten werden jeweils 4 zufällige Instanzen betrachtet. Für $x \geq 0.7$ benötigt AugTrees unabhängig von der Approximationsgüte durchschnittlich 2 und maximal 4 Passes.

6.2.2 Twitter-Graph

Zu guter Letzt testen wir AugTrees mit einem echten Graphen, der einen Bruchteil des sozialen Netzwerks Twitter[1] widerspiegelt. Dabei handelt es sich um einen in [17] präsentierten gerichteten Graphen, wobei die Knoten den Twitter-Nutzern entsprechen und eine Kante von a zu b signalisiert, dass a ein *Follower* von b ist. Die ursprünglichen Daten liegen in komprimierter Form vor und müssen deswegen zunächst mithilfe des WebGraph Frameworks [6] dekomprimiert werden. Außerdem muss der gerichtete Graph wie in [18] beschrieben in einen ungerichteten bipartiten Graph überführt werden, wodurch sich die Zahl der Knoten von $4 \cdot 10^7$ auf $8 \cdot 10^7$ verdoppelt. Zwischen diesen Knoten gibt es knapp $1.5 \cdot 10^9$ Kanten.

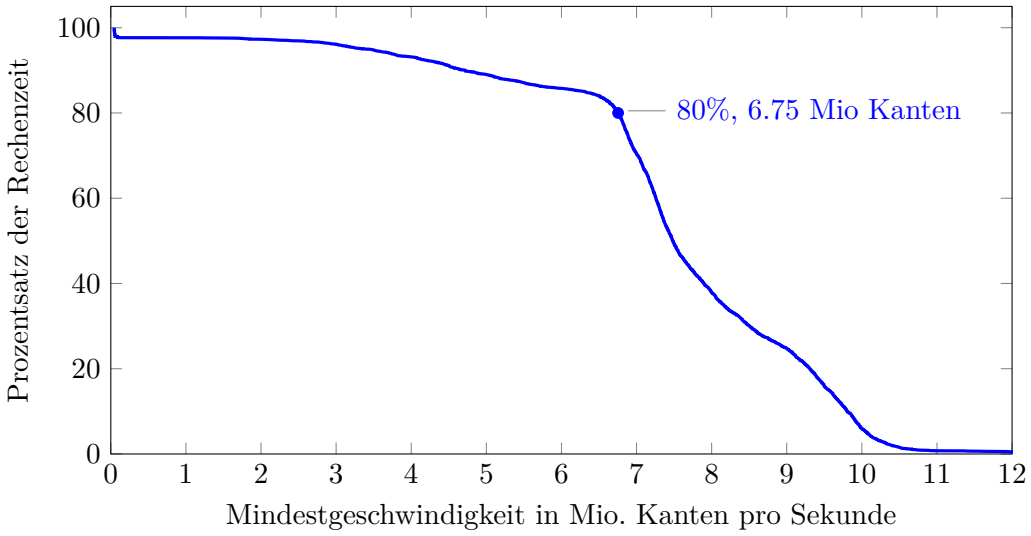


Abbildung 6.4: Mindestgeschwindigkeit, mit der AugTrees Kanten des Twitter-Graphen [17] liest, wenn eine 0.9-Approximation berechnet werden soll.

[1] https://twitter.com/

Für den Test befindet sich der Graph als Liste von Kanten auf einer Festplatte und im Gegensatz zu den vorherigen Tests wird tatsächlich gestreamt (keine Simulation im RAM). Als Prozessor wird ein Intel Core i5 4670k mit 4GHz verwendet.
Zum Berechnen der gewünschten 0.9-Approximation benötigt AugTrees trotz der hohen Knotenzahl insgesamt nur 19 Passes über den Stream. Abbildung 6.4 zeigt die Geschwindigkeit, mit der dabei Kanten gelesen werden. Für 80% der gesamten Rechenzeit können pro Sekunde mehr als 6.75 Mio. Kanten verarbeitet werden. Liegt der Graph als kompakte Liste aller Kanten vor (Binärdatei mit 64bit pro Kante), entspricht dies etwa 54MB/s. Ist der Graph hingegen als Kantenliste oder Adjazenzliste in einer Textdatei gegeben, ergeben sich ungefähr 120MB/s (Kantenliste) bzw. 60MB/s (Adjazenzliste). Damit können moderne Festplatten zwar nicht voll ausgelastet werden, allerdings wäre eine wesentlich schnellere Variante von AugTrees durchaus denkbar.
Mit einem Mehrkernprozessor und einer auf Multithreading ausgelegten Modifikation des Algorithmus könnte die Verarbeitungsgeschwindigkeit für Kanten wahrscheinlich deutlich erhöht werden. Schließlich können problemlos mehrere Kanten gleichzeitig von mehreren Kernen verarbeitet werden, solange auf jedem Baum des alternierenden Waldes immer nur eine Operation gleichzeitig ausgeführt sind. Das kann zum Beispiel durch einen Mutex pro Wurzel effizient realisiert werden.

Kapitel 7

Fazit und Ausblick

Wir haben uns umfassend mit dem Maximum-Matching-Problem im Semi-Streaming Modell auseinander gesetzt. Obwohl Strong-Spanning-Trees eine interessante Alternative zu traditionellen Methoden der Matching-Konstruktion sind, konnte mit ihnen leider kein effizienter Streaming-Algorithmus realisiert werden. Der neue Algorithmus AugTrees, der Kliemanns DapTrees um beidseitiges Suchen und sofortige Augmentiervorgänge erweitert, konnte hingegen praxistauglich implementiert werden. Um Maximum-Matchings in bipartiten Graphen mit bis zu 10^6 Knoten zu approximieren, wurden in allen Testinstanzen nur Pass-Counts im ein- bis zweistelligen Bereich benötigt, wenn eine 90%-Approximation erreicht werden sollte. Dabei ist der Pass-Count hauptsächlich vom Approximationsparameter k, und nur sehr leicht von n abhängig. Der vom Semi-Streaming Modell vorgeschriebene Speicherplatz von $\mathcal{O}(n \cdot \text{polylog}(n))$ wird von AugTrees niemals überschritten. Trotz der theoretisch hohen Laufzeitschranke von $\mathcal{O}(n)$ pro Kante konnte auf einem echten Graphen mit $8 \cdot 10^7$ Knoten eine gute Verarbeitungsgeschwindigkeit beobachtet werden.

Offen geblieben ist die Frage, ob es sogar effiziente exakte Lösungen für das Maximum-Matching-Problem im Semi-Streaming Modell gibt. Es wäre bereits ein großer Schritt, einen Approximationsalgorithmus zu finden, dessen Pass-Count linear von der Approximationsgüte abhängt. Für die beiden baumbasierten Algorithmen AugTrees und DapTrees könnte untersucht werden, ob es Modifikationen gibt, durch welche die Worst-Case-Pass-Schranke von $\mathcal{O}(kn)$ so verändert werden kann, dass keine Abhängigkeit von n mehr vorliegt. Auch die bereits in Kapitel 6 angedachte Umsetzung von Multithreading zur schnelleren Kantenverarbeitung ist ein interessantes Thema, das genauer untersucht werden sollte. Auf jeden Fall gibt es für neue Matching-Algorithmen zahlreiche Anwendungsmöglichkeiten und die immer größer werdenden Graphen, die in der heutigen Zeit entstehen, werden die Relevanz von Streaming Modellen mit Sicherheit nicht senken.

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Aggarwal, G., M. Datar, S. Rajagopalan und M. Ruhl: *On the Streaming Model Augmented with a Sorting Primitive.* In: *45th Annual IEEE Symposium on Foundations of Computer Science.* Institute of Electrical & Electronics Engineers (IEEE).

[2] Aho, Alfred V. und John E. Hopcroft: *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing, erste Auflage, 1974.

[3] Alt, H., N. Blum, K. Mehlhorn und M. Paul: *Computing a maximum cardinality matching in a bipartite graph in time* $\mathcal{O}(n^{1.5} \cdot \sqrt{\frac{m}{\log n}})$. Information Processing Letters, 37(4):237–240, 1991.

[4] Balinski, M. L. und J. Gonzalez: *Maximum matchings in bipartite graphs via strong spanning trees.* Networks, 21(2):165–179, 1991.

[5] Berge, C.: *TWO THEOREMS IN GRAPH THEORY.* Proceedings of the National Academy of Sciences, 43(9):842–844, 1957.

[6] Boldi, P. und S. Vigna: *The webgraph framework I.* In: *Proceedings of the 13th conference on World Wide Web - WWW '04.* Association for Computing Machinery (ACM), 2004.

[7] Cherkassky, Boris V., Andrew V. Goldberg, Paul Martin, Joao C. Setubal und Jorge Stolfi: *Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms.* J. Exp. Algorithmics, 3:8–es, 1998.

[8] Demetrescu, Camil, Bruno Escoffier, Gabriel Moruz und Andrea Ribichini: *Adapting parallel algorithms to the W-Stream model, with applications to graph problems.* Theoretical Computer Science, 411(44-46):3994–4004, 2010.

[9] Demetrescu, Camil, Irene Finocchi und Andrea Ribichini: *Trading off space for passes in graph streaming problems.* ACM Transactions on Algorithms, 6(1):1–17, 2009.

[10] Edmonds, Jack: *Paths, trees, and flowers.* Journal canadien de mathématiques, 17(0):449–467, 1965.

[11] Eggert, Sebastian, Lasse Kliemann, Peter Munstermann und Anand Srivastav: *Bipartite Matching in the Semi-streaming Model.* Algorithmica, 63(1-2):490, 2012.

[12] Feigenbaum, Joan, Sampath Kannan, Andrew McGregor, Siddharth Suri und Jian Zhang: *On Graph Problems in a Semi-streaming Model.* In: *Automata, Languages and Programming*, Seiten 531–543. Springer Science + Business Media, 2004.

[13] González, Jaime und Osvaldo Landaeta: *A Competitive Strong Spanning Tree Algorithm for the Maximum Bipartite Matching Problem.* SIAM J. Discrete Math., 8(2):186–195, 1995.

[14] Hopcroft, John E. und Richard M. Karp: *An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs.* SIAM J. Comput., 2(4):225–231, dec 1973.

[15] Kliemann, Lasse: *Matching in Bipartite Graph Streams in a Small Number of Passes.* In: *Experimental Algorithms*, Seiten 254–266. Springer Science + Business Media, 2011.

[16] Kliemann, Lasse: *Engineering a Bipartite Matching Algorithm in the Semi-Streaming Model.* In: *Algorithm Engineering: Selected Results and Surveys.* Springer, erscheint 2016.

[17] Kwak, Haewoon, Changhyun Lee, Hosung Park und Sue Moon: *What is Twitter, a social network or a news media?* In: *Proceedings of the 19th international conference on World wide web - WWW '10.* Association for Computing Machinery (ACM), 2010.

[18] Liu, Yang-Yu, Jean-Jacques Slotine und Albert-László Barabási: *Controllability of complex networks.* Nature, 473(7346):167–173, 2011.

[19] McGregor, Andrew: *Finding Graph Matchings in Data Streams.* In: *Proceedings of the 8th International Workshop on Approximation, Randomization and Combinatorial Optimization Problems, and Proceedings of the 9th International Conference on Randamization and Computation: Algorithms and Techniques*, APPROX'05/RANDOM'05, Seiten 170–181. Springer-Verlag, 2005.

[20] McGregor, Andrew: *Graph Mining on Streams.* In: *Encyclopedia of Database Systems.* Springer US, 2009.

[21] McGregor, Andrew: *Graph stream algorithms: a survey.* ACM SIGMOD Record, 43(1):9–20, 2014.

[22] Micali, Silvio und Vijay V. Vazirani: *An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algoithm for finding maximum matching in general graphs.* In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. Institute of Electrical & Electronics Engineers (IEEE), 1980.