Levin von Hollen
Tilman Beck

Implementierung der Cut & Count-Technik für das Steiner Tree-Problem

24. November 2016

Inhaltsverzeichnis

Kapitel 1
Einleitung

In dieser Arbeit wird die Cut & Count-Technik aus [CNP+] behandelt und eine konkrete Implementierung für das Steiner-Tree-Problem dargestellt. In Sektion 1.1 werden wir die Notation einführen, welche für das Verständnis der folgenden Kapitel wichtig ist. Die Cut & Count-Technik benutzt eine angepasste Form einer Nice Tree Decomposition aus [Klo94], welche wir in Kapitel 2 definieren und veranschaulichen. In Kapitel 3 wird die Funktionsweise der Cut & Count-Technik allgemein erklärt. Anschließend wird in Kapitel 4 die Technik angewendet auf das Steiner-Tree-Problem erläutert. Kapitel 5 umfasst unsere Implementierung zum Steiner-Tree-Problem, eine kurze Evaluation zu verschiedenen Eingabegrößen und einen Ausblick. Im letzten Kapitel 6 wird der Inhalt dieser Arbeit zusammengefasst.

1.1 Notation

Für den Rest der Arbeit bedienen wir uns der Notation aus der Arbeit [CNP+]. Die Bezeichnung $G = (V, E)$ beschreibt einen ungerichteten Graphen. Entsprechend beschreiben $V(G)$ und $E(G)$ die Menge der Knoten bzw. Kanten des Graphen G. Die Bezeichung $G[X]$ einer Knotenmenge $X \subseteq V(G)$ steht für den Subgraphen, der von X erzeugt wird. Für eine Menge an Kanten $X \subseteq E$ beschreibt $V(X)$ die Menge der Endknoten der Kanten aus X und $G[X]$ den Subgraphen (V, X). Die Knotenmenge für eine Menge von Kanten X im Graphen $G[X]$ ist diesselbe wie im Graphen G.

Mit einem „Schnitt“einer Menge $X \subseteq V$ ist das Paar (X_1, X_2) mit den Eigenschaften $X_1 \cap X_2 = \emptyset, X_1 \cup X_2 = X$. X_1 und X_2 werden als linke und rechte „Seiten“des Schnittes bezeichnet.

Die Zahl $cc(G)$ eines Graphen G beschreibt die Anzahl der zusammenhängenden Komponenten („**c**onnected **c**omponent“).

Für zwei Bags x, y eines Wurzelbaums gilt, dass y ein Nachkomme von x ist, falls es möglich ist ausgehend von y einen Weg zu x zu finden, der im Baum nur in Richtung des Wurzelknotens verläuft. Insbesondere ist x sein eigener Nachkomme.

Für zwei Integer a, b sagt die Gleichung $a \equiv b$ aus, dass a genau dann gerade ist, wenn auch b gerade ist. Zudem wird Iverson's Klammernotation verwendet. Falls p ein Prädikat ist, dann sei $[p]$ 1 (0) falls p wahr (unwahr) ist. Falls $\omega : U \to 1, \ldots, N$, so bezeichnet $\omega(S) = \sum_{e \in S} \omega(e)$ für $S \subseteq U$.

Für eine Funktion s mit $s[v \to \alpha]$ schreiben wir die Funktion $s\ (v, s(v)) \cup (v, \alpha)$. Diese Definition funktioniert unabhängig davon, ob $s(v)$ bereits definiert wurde oder nicht.

Kapitel 2
(Nice) Tree Decomposition

2.1 Tree Decomposition

Definition 2.1. (Tree decomposition, [RS84]). Eine Tree Decomposition *eines (ungerichteten oder gerichteten) Graphen G ist ein Baum* $\mathbb{T}$ *in dem jedem Knoten* $x \in \mathbb{T}$ *eine Menge von Knoten* $B_x \subseteq V$ *(genannt „Bag“) zugeordnet ist, so dass*

- *für jede Kante* $uv \in E$ *existiert ein* $x \in \mathbb{T}$*, so dass* $u, v \in B_x$
- *falls* $v \in B_x$ *und* $v \in B_y$*, dann* $v \in B_z$ *für alle z auf dem Pfad von x nach y in* $\mathbb{T}$

Das Konzept der Tree Decomposition wurde 1976 von Rudolf Halin[Hal76] eingeführt. Sie dient dazu die Baumweite zu definieren und Berechnungsprobleme auf Graphen schneller zu lösen.

Die Baumweite ist eine Zahl und beschreibt die „Baum-Ähnlichkeit“eines Graphen. Die Baumweite $tw(\mathbb{T})$ einer Tree Decomposition $\mathbb{T}$ ist die Größe des größten Bags minus eins. Die Baumweite eines Graphen G ist die minimale Baumweite aller möglichen Tree Decompositions von G.

Ein Beispiel für eine Tree Decomposition ist in Abbildung 2.1 gegeben. Der Ursprungsgraph ist in Abbildung 2.1.

2.2 Nice Tree Decomposition

Kloks[Klo94] führte die sogenannte Nice Tree Decomposition ein, welche oft für Algorithmen mit dynamischen Programmen genutzt werden. Da in Sektion 2.3 weitere Modifikationen auf Basis der Nice Tree Decomposition eingeführt werden, wird die hier vorgestellte Nice Tree Decomposition als *standardmäßige Nice Tree Decomposition* bezeichnet.

Definition 2.2. Eine standardmäßige Nice Tree Decomposition ist eine *Tree Decomposition, für die gilt:*

- *jeder Bag besitzt höchstens zwei Kind-Knoten*
- *falls ein Bag zwei Kind-Knoten* l, r *besitzt, dann gilt* $B_x = B_l = B_r$
- *falls ein Bag x einen Kind-Knoten besitzt, dann gilt entweder* $|B_x| = |B_y| + 1$ *und* $B_y \subseteq B_x$ *oder* $|B_x| + 1 = |B_y|$ *und* $B_x \subseteq B_y$

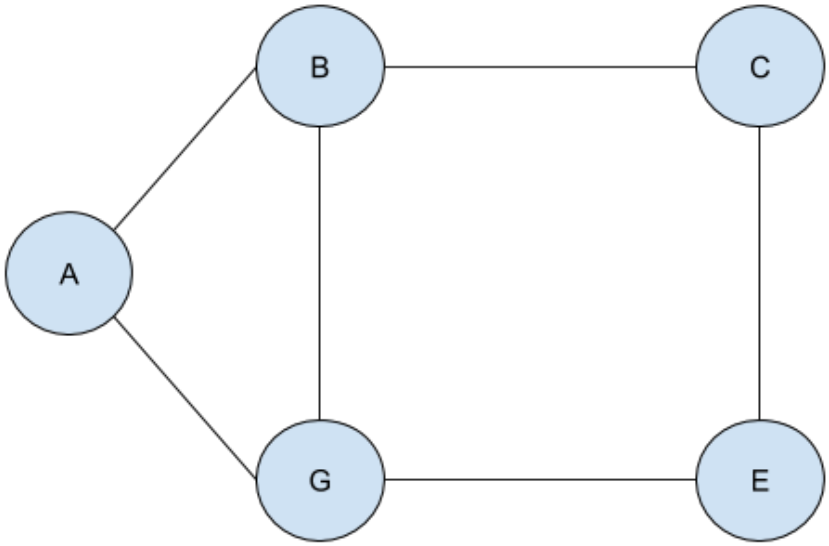

Abb. 2.1 Ursprungsgraph für eine Tree Decomposition

2.3 Weitere Modifikationen

Für den von [CNP+] beschriebenen Algorithmus wird die standardmäßige Nice Tree Decomposition zusätzlich noch auf folgende Weise modifiziert:

Definition 2.3. (Nice Tree Decomposition). Eine Nice Tree Decomposition *ist eine Tree Decomposition mit einem speziellen Bag z (Wurzel) mit* $B_z = \emptyset$ *und in der jeder Bag einer der folgenden Arten entspricht:*

- ***Leaf Bag****: ein Blatt x aus* $\mathbb{T}$ *mit* $B_x = \emptyset$.
- ***Introduce Vertex Bag****: ein innerhalb von* $\mathbb{T}$ *liegender Knoten x mit einem Kind-Knoten y für den gilt* $B_x = B_y \cup \{v\}$ *für einen* $v \notin B_y$*. Dieser Bag führt den Knoten v ein.*
- ***Introduce Edge Bag****: ein innerhalb von* $\mathbb{T}$ *liegender Knoten x, der mit der Kante* $uv \in E$ *gekennzeichnet ist und einen Kind-Knoten y mit* $u, v \in B_x = B_y$ *besitzt. Dieser Knoten führt die Kante uv ein.*
- ***Forget Bag****: ein innerhalb von* $\mathbb{T}$ *liegender Knoten x mit einem Kind-Knoten y, für den gilt* $B_x = B_y \setminus \{v\}$ *für ein* $v \in B_y$*. Dieser Bag vergisst den Knoten v.*
- ***Join Bag****: ein innerhalb von* $\mathbb{T}$ *liegender Knoten x mit zwei Kind-Knoten* l, r*, für die* $B_x = B_r = B_l$ *gilt.*

Zusätzlich wird gefordert, dass jede Kante aus E genau einmal eingeführt wird.

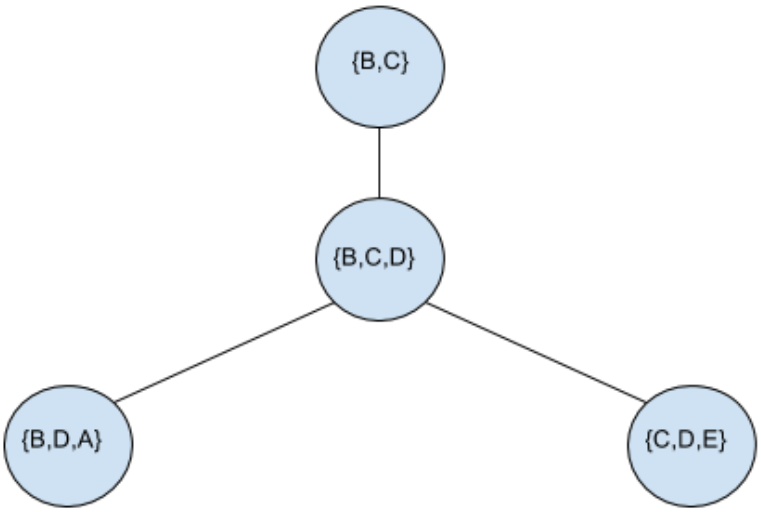

Abb. 2.2 Eine Tree Decomposition für den in Abbildung 2.1 gegebenen Ursprungsgraphen

Ein Beispiel für eine Nice Tree Decomposition ist in Abbildung 2.3.

Sei eine Tree Decomposition gegeben, kann eine standardmäßige Nice Tree Decomposition von gleicher Breite in polynomieller Zeit gefunden werden [Klo94]. In der selber Laufzeit kann der Algorithmus für eine standardmäßige Nice Tree Decomposition modifizert werden, so dass das Ergebnis zusätzlich die o.g. Kriterien erfüllt.

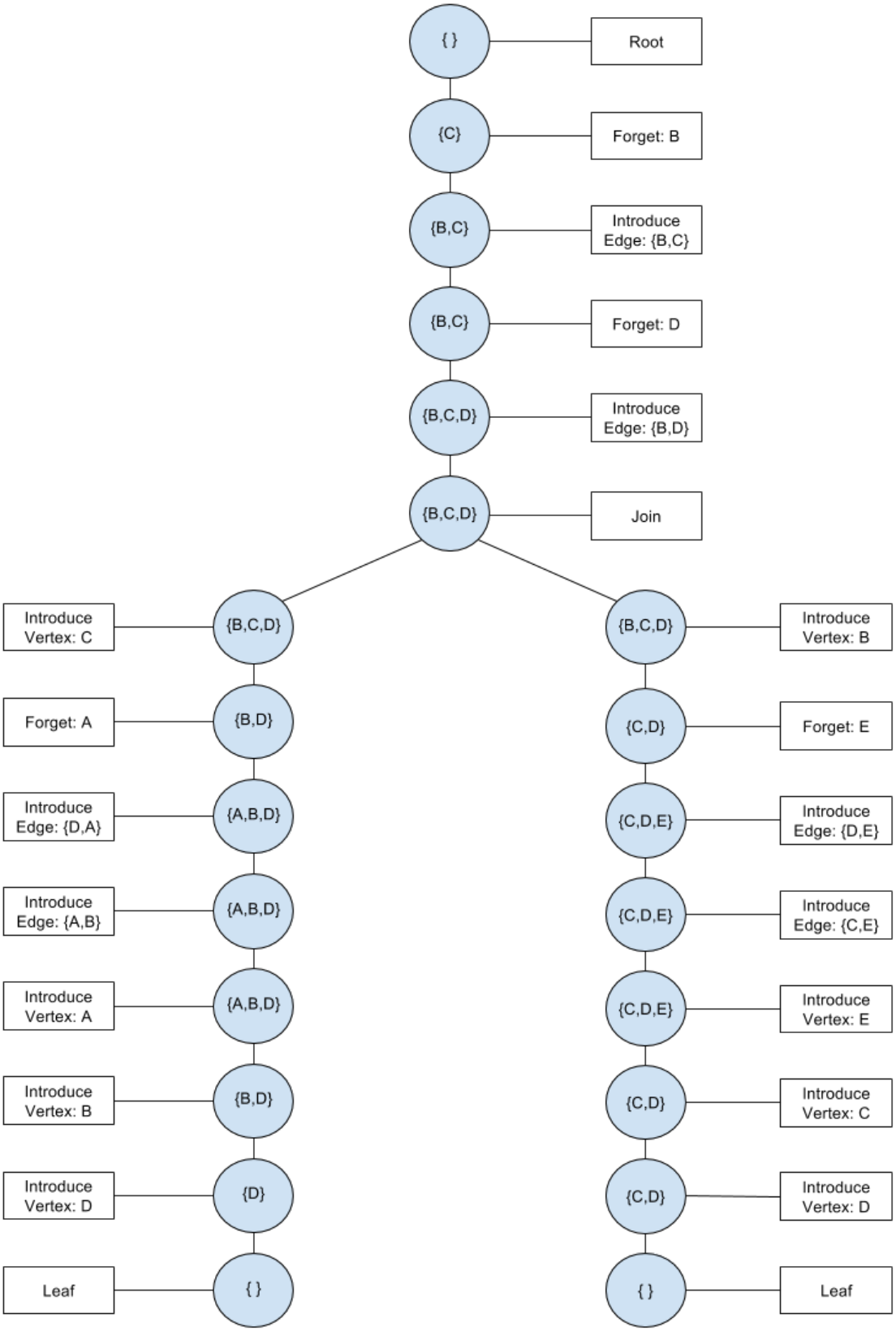

Abb. 2.3 Eine Nice Tree Decomposition für den Ursprungsgraphen in Abbildung 2.1

Kapitel 3
Cut & Count-Technik

3.1 Einführendes

Die Cut & Count-Technik wird benutzt um connectivity-type Probleme mithilfe von Randomisierung zu lösen. Dabei handelt es sich um Graphen-Probleme, bei denen zusammenhängende Submengen der Knoten gefunden werden müssen, die problemspezifische Eigenschaften erfüllen müssen. Beispiele hierfür sind Longest Path, Steiner Tree, Feedback Vertex Set, uvm. Bei Verwendung der Cut & Count-Technik wird jedem Element des Problems eine zufällig und unabhängig gewählte Eigenschaft zugeteilt. Dies folg daher, dass die Technik nur korrekt funktioniert, wenn es eine oder keine Lösung gibt. Beim Lösen des Problems muss diese neue Eigenschaft zusätzlich eine Anforderung erfüllen, damit die Lösungsmenge gültig ist. Diese Isolation der Lösungsmenge wird durch das Isolations-Lemma beschrieben.

3.2 Monte-Carlo-Algorithmus

Verwendet man die Cut & Count-Technik um ein Algorithmus zur Lösung der connectivity-type Probleme zu entwerfen, enthält man einen Monte-Carlo-Algorithmus. Dies bedeutet, dass der hervorgegangene Algorithmus niemals ein false-positiv [1] ausgeben kann aber zu einer gewissen Wahrscheinlichkeit ein false-negativ [2]. Die Wahrscheinlichkeit eines false-negativ wird durch das Isolations-Lemma bestimmt. Da eine Ausgabe eine false-negativ einer bestimmten Wahrscheinlichkeit unterliegt ist es ratsam Monte-Carlo-Algorithmen mehrmals hintereinander auszuführen. Es begünstigt die Wahrscheinlichkeit eine Lösungsmenge zu finden.

Der hervorgegangene Monte-Carlo-Algorithmus kann bei gegebener Tree-Decomposition mit Braumbreite t eines ungerichteten Graphens Lösungen bestimmen bei folgenden Probleme und Laufzeiten:

- Steiner Tree in $3^t|V|^{O(1)}$
- FeedBack Vertex Set in $3^t|V|^{O(1)}$
- ...

[1] Der Algorithmus gibt eine positive Antwort auf das Problem, obwohl keine Lösung existiert

[2] Der Algorithmus gibt eine negative Antwort auf das Problem, obwohl eine Lösung existiert.

3.3 Isolation Lemma

Wie zuvor erwähnt ist die Cut & Count-Technik nur dann korrekt, wenn eine oder keine Lösung existiert. Da es bei vielen der connectivity-type Probleme sehr wahrscheinlich ist, dass mehrere Lösungen existieren, muss die Lösungsmenge reduziert werden. Dafür wird auf das Isolations-Lemma zurück gegriffen. Es ist wie folgt definiert:

A function $\omega : U \to \mathbb{Z}$ isolates a set family $\mathscr{F} \subseteq 2^U$ if there is a unique $S' \in \mathscr{F}$ with $\omega(S') = min_{S \in \mathscr{F}} \omega(S)$

Im Paper wird es im Lemma 2.5 verwendet, um etwas über die Wahrscheinlichkeit der Isolation auszusagen:

Let $\mathscr{F} \subseteq 2^U$ be a set family over a universe U with $|\mathscr{F}| > 0$. For each $u \in U$, choose a weight $\omega(u) \in 1, 2, ..., N$ uniformly and independently at random. Then

$$prob[\omega \text{ isolates } \mathscr{F}] \geq 1 - \frac{|U|}{N}$$

Durch die Wahl eines großen N kann somit die Wahrscheinlichkeit eines false-negativ stark reduziert werden aber es wird zugleich auch die Laufzeit des Algorithmus erhöht.

Kapitel 4
Cut & Count für Steiner Tree

4.1 Steiner Tree

Definition 4.1. Steiner Tree
Input: An undirected graph $G = (V,E)$, a set of terminals $T \subseteq V$ and an integer k.
Question: Is there a set $X \subseteq V$ of cardinality k such that $T \subseteq X$ and $G[X]$ is connected?

Bei dem Steiner-Tree Problem handelt es sich um ein connectivity-Typ Problem. In einen gegebenen Graphen sind einige Knoten als Terminale makiert. Nun wird nach einem Subgraphen innerhalb des Ursprungsgraphen gesucht, der alle Terminale enthält und aus k vielen Knoten insgesamt besteht. Desweiteren müssen alle Knoten innerhalb des Subgraphen über Kanten untereinander erreichbar sein.

4.2 Cut

Zu beginn des Cut-Part wird eine zufällige Gewichtsfunktion $\omega : V \to \{1,\ldots,N\}$ definiert. Diese wird für die Isolation der Lösungsmenge verwendet.
Desweiteren wird im Cut-Part die Menge $\mathscr{R}_W$ definiert. Sie ist die Menge aller Teilmengen von X aus V mit $T \subseteq X$, $\omega(X) = W$ und $|X| = k$. Somit stellt die Menge $\mathscr{R}_W$ die Menge aller Lösungskandidaten dar.
Eine weitere zu definierende Menge ist $\mathscr{S}_W = \{X \in \mathscr{R}_W | G[X] \text{ ist zusammenhängend}\}$. Somit bildet die Menge $\mathscr{S}_W$ Lösungen für ein gegebenes bestimmtes Gewicht W.
$\cup_W \mathscr{S}_W$ bildet so die Lösungsmenge. Gibt es nun ein Gewicht W für das die Menge nicht leer ist, so gibt der Algorithmus eine positive Antwort.
Von der Menge der Terminalknoten wird ein Terminal als v_1-Terminal festgelegt. Dieses dient dazu, dass bei der Bildung von konsistenten Cuts keine Cuts doppelt gezählt werden.
Die konsistenten Cuts werden in der Menge $\mathscr{C}_W$ beschrieben. Diese Bilden die Menge aller Subgraphen, die einen konsistenten Cut $(X,(X_1,X_2))$ bilden, wobei $X \in \mathscr{R}_W$ und $v_1 \in X_1$.

Die Anzahl der konstistenten Cuts sind im Lemma 3.3 des Papers beschrieben:
Let $G = (V,E)$ be a graph and let X be a subset of vertices such that $v_1 \in X \subseteq V$. The number of consistently cut subgraphs $(X,(X_1,X_2))$ such that $v_1 \in X_1$ is equal to $2^{cc(G[X])-1}$.

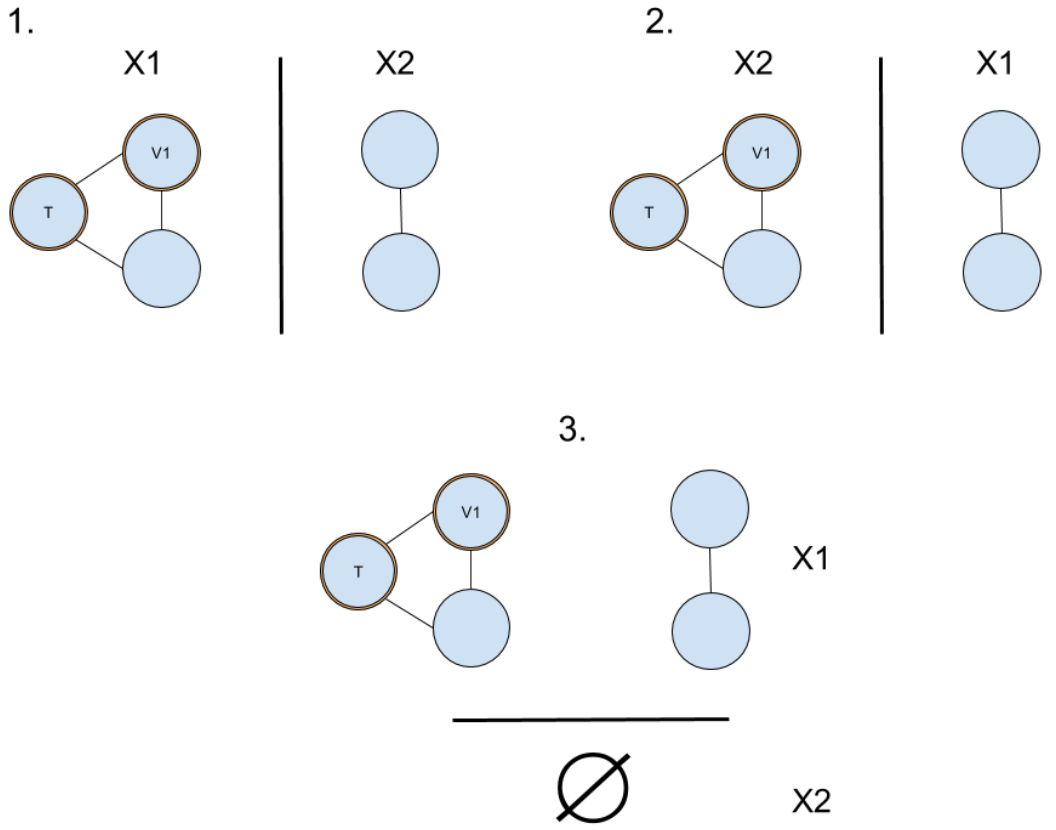

Abb. 4.1 Konsistente Cuts ohne Beschränkung $v_1 \in X_1$. Man sieht, dass die konsistenten Cuts 1 und 2 identisch sind.

Beweis: Per Definition ist bekannt, dass für jeden konsistenten Cut $(X,(X_1,X_2))$ und Connected Component C aus $G[X]$ C entweder in X_1 oder in X_2 enthalten sein muss. Für die Connected Component, die v_1 enthält ist die Wahl der Submenge fix. Für alle anderen Connected Components kann die Zugehörigkeit von Submengen frei gewählt werden. Daher erhalten wir $2^{cc(G[X])-1}$ verschiedene konsistente Cuts.

4.3 Count

Aus Lemma 3.3 ist bekannt: $|\mathscr{C}| = \sum_{X \in \mathscr{R}} 2^{cc(G[X])-1}$. Wir legen W fest und ignorieren die Indices: $|\mathscr{C}| \equiv |\{X \in \mathscr{R} | cc(G[X]) = 1\}| = |\mathscr{S}|$. In Worte gefasst bedeutet dies, dass die Anzahl der konsistenten Cuts

eines Graphen modulo zwei gleich die Anzahl der Lösungen ist. Im Lemma 3.4 trifft das Paper dazu eine Aussage: Let G, ω, $\mathscr{C}_W$ and $\mathscr{S}_W$ be as defined above. Then for every W, $|\mathscr{S}_W| \equiv |\mathscr{C}_W|$.

4.4 Dynamisches Programm

$|\mathscr{C}_W|$ modulo 2 kann mit dynamischen Programm auf der NTD $\mathbb{T}$ berechnet werden. für jeden Bag $x \in \mathbb{T}$, integers $0 \leq i \leq k, 0 \leq w \leq kN$ und Färbung $s \in \{0, 1_1, 1_2\}^{B_x}$ definiere:

- $\mathscr{R}_x(i,w) = \{X \subseteq V_x | (T \cap V_x) \subseteq X \wedge |X| = i \wedge \omega(X) = w\}$
- $\mathscr{C}_x(i,w) = \{(X,(X_1,X_2)) | X \in \mathscr{R}_x(i,w) \wedge (X,(X_1,X_2))$ is a consistently cut subgraph of $G_x \wedge (v_1 \in V_x \Rightarrow v_1 \in X_1\}$
- $\mathscr{A}_x(i,w,s) = |\{(X,(X_1,X_2)) \in \mathscr{C}_x(i,w) | (s(v) = 1_j \Rightarrow v \in X_j) \wedge (s(v) = 0 \Rightarrow v \notin X\}|$

Die Färbungen der Knoten gibt an ob sie zu einem konsistenten Cut gehören und wenn ja zu welcher der beiden Teilmengen des konsistenten Cuts sie gehören. Färbung $s \in \{0, 1_1, 1_2\}^{B_x}$ der Knoten aus B_x bzgl. der Menge C_x

- $s[v] = 0 \Rightarrow v \notin X$
- $s[v] = 1_1 \Rightarrow v \in X_1$
- $s[v] = 1_2 \Rightarrow v \in X_2$

$A_x(i,w,s)$ zählt alle möglichkeiten die Knoten gemäß der Definition zu färben. Ob der Algorithmus eine Lösung gefunden hat kann im Wurzel-Knoten eingesehen werden.
Im dynamischen Programm werden die folgenden Berechnungsregeln für jede $A_x(i,w,s)$ Matrix eines Bags angewandt. Zur Vereinfachung der Notation beschreibt im folgenden v den neu eingeführten Vertex. y und z stehen für das linke bzw. das rechte Kind:

- **Leaf bag**:
  - $A_x = (0,0,\emptyset) = 1$
    Leaf bags enhalten keine Knoten, daher werden sie mit einem Initialwert gefüllt.
- **Introduce vertex v**:
  - $A_x = (i,w,s[v \to 0]) = [v \notin T]A_y(i,w,s)$
    Ist der eingeführte Vertex kein Terminal, so wird der Wert aus dem Bag des Kindes übernommen.
  - $A_x = (i,w,s[v \to 1_1]) = A_y(i-1, w-w(v), s)$
    Reduziere beim Zugriff auf den Bag des Kindes i um 1 und ziehen das Gewicht des eingeführten Knoten ab. Übernehme den Wert.
  - $A_x = (i,w,s[v \to 1_2]) = [v \neq v_1]A_y(i-1, w-w(v), s)$
    Ist der eingeführte Vertex nicht das speziell gewählte Terminal so verfahre wie bei 1_1.
- **Introduce edge uv**
  - $A_x(i,w,s) = [s(u) = 0 \vee s(v) = 0 \vee s(u) = s(v)]A_y(i,w,s)$
    Ist einer der Verticies 0 gefärbt oder sind beide gleich gefärbt, so wird der Wert aus dem Bag des Kindes übernommen.
- **Forget vertex v**

- $A_x(i,w,s) = \sum_{\alpha \in 0,1_1,1_2} A_x(i,w,s[v \to \alpha])$
  Es wird die Summe gebildet über alle Färbungen des vergessenen Vertex im Bag des Kindes.

- **Join bag**
  - $A_x(i,w,s) = \sum_{i_1+i_2=i+\lvert s^{-1}(1_1,1_2)\rvert} \sum_{w_1+w_2=w+w(s^{-1}(1_1,1_2))} A_y(i_1,w_1,s)A_z(i_2,w_2,s)$
    In der inneren Summe wird über die Gewichte innerhalb beider Bags der Kinder iteriert. Ist deren Summe gleich der Summe von w und der Summe der Gewichte von Knoten mit der Färbung 1_1 und 1_2, so werden sie akkumuliert.
    In der äußeren Summe wird über den Parameter i der Bags der Kinder iteriert. Ist die Summe gleich der Summe von i und der Anzahl der Knoten die 1_1 und 1_2 gefärbt sind, so werden sie akkumuliert.

Da alle Berechnungen der Bags jeweils nur von den Werten des Kindes abhängig sind, kann das Ergebnis des Algorithmus im Wurzelknoten ausgelesen werden. Die Matrix des Wurzelknotens $A_r(k,W,\emptyset)$ enthält an den Stellen k und W eine 1, zu denen es eine Lösung der Größe k mit Gesamtgewicht W existiert. Kleinere Kardinalitäten können hierbei auch direkt überprüft werden.

4.5 Monte-Carlo Algorithmus und Laufzeit

Im Theorem 3.6 des Papers wird zusammenfassend erwähnt:

Theorem 4.1. *There exists a Monte-Carlo algorithm that given a tree decomposition of width t solves STEINER TREE in $3^t|V|^{\mathcal{O}(1)}$ time. The algorithm cannot give false positives and may give false negatives with probability at most 1/2.*

Die Laufzeit setzt sich wie folgt zusammen:

- 3^t:
  Da alle Knoten drei verschiedene Färbungen annehmen können und über all diese iteriert werden muss, erhalten wir 3^t verschiedene Färbungen. Hierbei ist die *treewidth* der Falschenhals. Das Dynamische Programm berechnet nur die Färbungen für alle im Bag enthaltenen Verticies.
- $|V|^{\mathcal{O}(1)}$:
  Kommt durch die beiden Input-Parameter k und N.
- Die Wahrscheinlichkeit von 1/2 für falsch-negativ entsteht durch die Gewichtsfunktion $\omega : V \to \{1,\ldots,N\}$ und durch das Isolations-Lemma.

Kapitel 5
Implementierung

5.1 Einführung

In unserer Implementierung liegt der Fokus auf der Implementierung des dynamischen Programms der Cut & Count-Technik. Wir nehmen an, dass für den Ursprungsgraphen G bereits eine standardmäßige Nice Tree Decomposition gemäß Sektion 2.2 vorliegt. In Sektion 5.2 erklären wir, wie die standardmäßige Nice Tree Decomposition zu einer Nice Tree Decomposition (siehe Sektion 2.3) erweitert wird. Anschließend erläutern wir in Sektion 5.3 die Implementierung des dynamischen Programms und gehen auf implementierungsspezifische Details ein. In Sektion 5.4 zeigen und diskutieren wir unsere Ergebnisse der Laufzeit mit verschiedenen Eingabegröße. Sektion 5.5 gibt einen Ausblick für mögliche Optimierungen und Erweiterungen unserer Implementierung.

5.2 Nice Tree Decomposition

Es wurde ein Algorithmus entwickelt, der als Eingabe eine standardmäßige Nice Tree Decomposition $\mathbb{T}$ eines Graphen G erhält und eine Nice Tree Decomposition (siehe Kapitel 2.2) ausgibt. Da der Fokus dieser Arbeit auf der Implementierung des dynamischen Programms des Cut & Count-Algorithmus liegt, wurde der Algorithmus nicht hinsichtlich der in [Klo94] beschriebenen polynomiellen Laufzeit optimiert.

Der Algorithmus iteriert mehrmals in symmetrischer Reihenfolge (ausgehend von der Wurzel linksseitig absteigend) über $\mathbb{T}$ und fügt dabei die fehlenden Knoten ein. Hierbei sollte erwähnt werden, dass in unserer Implementierung mit Ausnahme des „Join“-Bags neue Knoten stets linksseitig an den Elternknoten angehängt werden. Dies ist für die rekursive Iteration in Sektion 5.3 wichtig.

Zu Beginn werden am bisherigen Wurzelknoten so lange „Forget“-Knoten angehängt bis noch ein Knoten des Ursprungsgraphen im Bag verbleibt. Anschließend wird ein letzter Knoten mit leerem Bag als neuer Wurzelknoten hinzugefügt. Ähnlich wird hinsichtlich der Blattknoten verfahren. Entsprechend der Differenz eines leeren Bags und der Bags der bisherigen Blattknoten werden am Ende jedes Pfades neue „Introduce-Vertex“-Knoten und ein Knoten mit leerem Bag als neuer Blattknoten angehängt. Für bestehende „Join“-Knoten werden die Bags der beiden Kindknoten verglichen. Sofern sie nicht denselben Bag wie der „Join“-Knoten haben, werden neue Knoten („Forget“, „Introduce Vertex“) eingefügt, bis die Bags

identisch mit dem Elternknoten sind. In der nächsten Iteration werden die Differenzen der Bags von Kind- und Elternknoten verglichen. Falls diese größer eins ist, werden entsprechend viele neue Knoten („Forget“, „Introduce Vertex“) eingeführt. Zuletzt wird für jede Kante e des Ursprungsgraphen G über den Graphen iteriert. Beim ersten gemeinsamen Auftreten der Knoten der Kante e innerhalb eines Bags, wird oberhalb des Knoten dieses Bags ein neuer „Introduce Edge“-Knoten eingeführt und mit den Knoten der Kante e gekennzeichnet.

Nach diesen Modifikationen liegt der Graph in der Form einer Nice Tree Decomposition wie in Sektion 2.2 beschrieben vor und kann zur Berechnung des dynamischen Programms verwendet werden.

5.3 Dynamisches Programm

Die Berechnungsvorschrift des dynamischen Programms ist in Sektion 4 erläutert. Für die Implementierung wird vom Wurzelknoten ausgehend in symmetrischer Reihenfolge über die Nice Tree Decomposition $\mathbb{T}$ iteriert und für jeden Bag eine $k \times kN \times 3^{|B_x|}$ - Matrix berechnet, wobei die Größe der Lösung k und N als Eingabeparameter übergeben werden. Die erste Dimension k beschreibt die Größe (Anzahl Knoten) der Lösungsmenge. Die zweite Dimension kN steht für die Summe der Gewichte der Lösungsmenge. Obwohl die Gewichte zufällig einheitlich verteilt werden, kann im schlechtesten Fall jedem Knoten das Gewicht N zugewiesen werden. Daher kann die Gesamtsumme der Gewichte W gleich kN sein. $3^{|B_x|}$ beschreibt die Anzahl der möglichen Färbungen innerhalb eines Bags. Während k, N festgelegt sind, kann die Länge der Farb-Dimension $3^{|B_x|}$ von Bag zu Bag variieren.

5.3.1 Implementierung der CountC-Prozedur

Da das dynamische Programm des Cut & Count-Algorithmus die Berechnungsvorschrift für einen Bag rekursiv über den Bag des jeweiligen Kind-Knotens definiert, steigt der Algorithmus zu Beginn in $\mathbb{T}$ rekursiv ab bis er an einem Blattknoten angekommen ist. Für diesen gibt es keine Farb-Dimension (der Bag ist leer) und die $k \times kN$-Matrix wird initialisiert. Anschließend wird beim rekursiven Aufstieg für jeden Bag die entsprechende Berechnungsvorschrift angewendet und eine neue Datenmatrix berechnet.

Im Falle eines „Introduce Vertex“-Bag werden für jede Färbung des Bags des Kindknotens drei neue Färbungen hinzugefügt. Für einen „Forget“-Bag werden jeweils drei Färbungen des Bags des Kindknotens zu einer Färbung zusammengeführt. Für alle anderen Bag-Typen bleibt die Länge der Farb-Dimension gleich, es werden jedoch nur Werte übernommen, welche die in Sektion 4.4 definierten Bedingungen erfüllen. Der „Join“-Bag nimmt eine Sonderstellung ein, da er als einziger zwei Kindknoten besitzt und im Algorithmus auf die Daten beider Bags zugreift. Die rekursive Berechnung in symmetrischer Reihenfolge gewährleistet, dass beide Kindknoten eines „Join“-Bags vorher berechnet werden. Der letzte Berechnungsschritt für den Wurzelknoten entspricht der Berechnung eines „Forget“-Knoten und führt die letzten drei Färbungen zusammen, so dass der Wurzelknoten (mit leerem Bag) eine $k \times kN$-Datenmatrix enthält. Diese kann für die Abfrage der Lösungen $A_r(k, W, \emptyset)$ genutzt werden.

Der Programmablauf ist im Algorithmus 5.1 dargestellt.

```
v1 = terminals[0]
def countC(node, indices, child_data, k, N, terminals, weights):
    if node.linker_Knoten is not None:
        # steige linksseitig ab
        child_data = countC(node.linker_Knoten, indices, child_data, k, N,
            terminals, weights)
    if node.rechter_Knoten is not None:
        # steige rechtsseitig ab
        child_data_right = countC(node.rechter_Knoten, indices, child_data, k,
             N, terminals, weights)
    if node.Bag_Typ == Leaf:
        # erzeuge und initialisiere k x kN Datenmatrix
        dm = new data[k, k*N]
        dm[0,0] =  1
        return dm
    elif node.Bag_Typ == Root:
        # v wurde aus Bag entfernt
        data = new data[3^node.bag.length, k, k*N]
        for i in range(0,k):
                for w in range(0,kN):
                        data[s,i,w] = child_data[s=0,i,w] + child_data[s=1,i,w
                            ] + child_data[s=2,i,w]
        return data
    elif node.Bag_Typ == Introduce_Edge:
        # im 'Introduce Edge'-Bag (u,v sind die Knoten der Kante e)
        # s ist die Faerbung
                data = data = new data[3^node.bag.length, k, k*N]
                for s in len(data):
                        if( s(u) == 0 | s(v) == 0 | s(u) == s(v) ):
                                for i in range(0,k):
                                        for w in range(0,kN):
                                                data[s,i,w] = child_data[s,i,w
                                                    ]
        return data
    elif node.Bag_Typ == Introduce_Vertex:
        # erzeuge neue Datenmatrix (groesser als Kind, da pro Knoten im Bag
            drei neue Faerbungen)
        # v ist der 'introduced vertex'
        data = new data[3^(child_data.length + 1), k, k * N]

        for s in child_colorings:
                # berechne Indices fuer neue Faerbungen
                indices = calculate_new_indices
            for i in range(0, k):
                for w in range(0, k * N):
                    if not terminals.contains(v):
                        data[indices[0], i, w] = child_data[s, i, w]
                    data[indices[1], i, w] = child_data[s, i - 1, w - weights(
                        v)]
                    if v != v1:
```

```
                    data[indices[2], i, w] = data[s, i - 1, w - weights.
                        get(introduced_vertex)]
        return data
    elif node.Bag_Typ == Forget:
        # v ist der 'forget vertex'
        data = new data[3^(child_data.length - 1), k, k * N]

        for s in colorings:
            for i in range(0, k):
                for w in range(0, k * N):
                    # berechne die Indices der drei Faerbungen, die
                        aufsummiert werden
                    indices = calculate_indices()
                    data[s, i, w] = child_data[indices[0], i, w] +
                                    child_data[indices[1], i, w] +
                                    child_data[indices[2], i, w]
        return data
    elif node.Bag_Typ == Join:
        data = new data[node.bag.length, k, k * N]
        for s in colorings:
            for i in range(0, k):
                for w in range(0, k * N):
                    colored_nodes = all_nodes_by_coloring(1,2)
                    bound_i = i + colored_nodes.length
                    bound_w = w + get_sum_of_weights(colored_nodes)
                    for i1 in range(0, bound_i):
                        for w1 in range(0, bound_w):
                            i2 = acc_bound_1 - i1
                            w2 = acc_bound_2 - w1
                            if w1 >= (k * N) or w2 >= (k * N) or i1 >= k or i2
                                >= k:
                                value += 0
                            else:
                                value += child_data[s, i1, w1] *
                                         child_data_right[s, i2, w2])
                    data[s, i, w] = value
        return data
    return data
```

Algorithmus 5.1 Pseudocode für das dynamische Programm

5.3.2 Berechnung der Färbungen

Für die Kodierung der Färbungsdimension muss eine geeignete Zuordnung von Färbungen zu Indices der Datenmatrix eingeführt werden. Dafür benutzen wir eine ternäre Kodierung mit folgender Zuordnung:

$$0 \to s = 0$$
$$1 \to s = 1_1$$

$$2 \to s = 1_2$$

Die ternäre Darstellung beschreibt die Färbung einer Menge von Knoten, wobei jede Ziffer einen gefärbten Knoten markiert. Die entsprechende Dezimaldarstellung steht für den Index der Färbungsdimension. Der Zugriff auf diesen Index liefert die $k \times kN$-Datenmatrix für die entsprechende Färbung. Für die Reduzierung bzw. Erweiterung der Färbungsdimension werden Funktionen benötigt, welche die richtige Zuordnung zwischen Färbung und Kodierung gewährleisten. Dazu wurde die Funktion `calculate_indices` implementiert. Diese bekommt als Parameter die Färbung als geordnete Liste (z. B. `[0,2,1]` für 7 als Index), wobei jede Zahl dieser Liste für einen entsprechend der o.g. Zuordnung gefärbten Knoten steht. Zusätzlich wird eine Liste von Indices übergeben, welche die „freien“ Stellen der Färbungsliste markieren. Die „freien“ Stellen sind die Knoten, für die alle Färbungsmöglichkeiten mit den anderen Stellen der Liste kombiniert werden. Die Funktion berechnet die entsprechenden Möglichkeiten mittels Horner-Schema und gibt sie als null-basierte Dezimalzahlen zurück. Diese dienen im dynamischen Programm als Zugriffsindices für die Färbungsdimension der Datenmatrizen. Als Beispiel wird der Funktion `([0,2], [0])` als Parameter übergeben. Dies bedeutet, dass alle Färbungsmöglichkeiten der ersten Stelle (null-basiert) mit der zweiten Stelle (mit der Färbung $2 \to 1_2$) kombiniert werden. Als Ergebnis gibt die Funktion in diesem Fall die Liste `[2,5,8]` als Ergebnis aus, wobei `[2,5,8]` für die Färbungen `[0,2],[1,2],[2,2]` stehen.

Um auf Färbungen einzelner Knoten zurückgreifen zu können (z. B. für die Berechnung des „Introduce Edge“-Bags) wird eine Funktion `get_index_as_list` benötigt. Dieser wird eine Färbung als Index (in Dezimaldarstellung) und die Länge des Bags als Parameter übergeben wird. Diese berechnet die Liste der ternär codierten Färbungen mit der Länge des Bags. Der Code dafür ist in Algorithmus 5.2 dargestellt.

```
def get_index_as_list(x, nr_of_vertices):
    if nr_of_vertices == 0:
        return [0]
    number = x
    res = []
    for i in range(nr_of_vertices-1, -1, -1):
        value = m.floor(number/(3**i))
        number -= (value * 3**i)
        res.append(value)

    return res
```

Algorithmus 5.2 Funktion get_index_as_list zur Berechnung der ternären Färbung aus dem Index

5.4 Evaluierung

Die Laufzeit des dynamischen Programms wurde anhand von Beispielen getestet. Zu sehen sind die Eingabeparameter k und N sowie die Anzahl der Knoten des Graphen sowie die in der Nice-Tree-Decomposition erhaltene *treewidth*.

Eingabe	∅ **T in** *s*
(k=2, N= 6, |V|=3, tw(G)=2)	~ 0.002
(k=3, N=14, |V|=7, tw(G)=3)	~ 0.83
(k=4, N=32, |V|=16, tw(G)=3)	~ 14.23

Tabelle 5.1 Laufzeit des dynamischen Programms für verschiedene Eingabegrößen

Es ist ersichtlich, dass der Algorithmus schon bei kleinen Grapherweiterungen merkbare Laufzeitverlängerungen aufweist. Es ist anzunehmen, dass der Algorithmus in der Praxis nur eingeschränkt nutzbar ist. Praxisrelevante Probleme werden sich wahrscheinlich in Graphen repräsentieren, die weit aus komplexer sind, als die von uns getesteten Beispiele. Wir vermuten die Laufzeit wäre dort nichtmehr tragbar oder zumindest nicht effizient genug. Desweiteren ist der Speicherbedarf der Cut & Count-Technik nicht außer acht zu lassen. Jeder Bag benötigt eine $3^t \times k \times kN$-Matrix. Bei größeren Graphen und hohem gewählten N, zur Verringerung des Falsch-Negativ Ergebnisses, kann diese schnell sehr groß werden.

5.5 Ausblick

Kapitel 6
Zusammenfassung

Verwendet man die Cut & Count-Technik, um einen Algorithmus für Graphenprobleme zu implementieren, so erhält man einen Monte-Carlo-Algorithmus. Mittels Randomisierung und dem Isolations-Lemma3.3 erhalten wir niemals eine falsch-positive Antwort aber zu einer, abhängig von der Wahl eines Parameters, eine falsch-negativ Antwort.

In dieser Arbeit haben wir uns damit beschäftigt diese Technik für das Steiner-Tree Problem 4.1 zu implementieren. Zusätzlich, da die Cut & Count-Technik eine spezifische Nice-Tree-Decomposition 2 benötigt, beschäftigten wir uns ebenfalls mit einer Implementation die Nice-Tree-Decompositions in die Cut & Count spezifische überführt.

Aufgrund der von der Technik resultierenden Laufzeit $3^t|V|^{\mathcal{O}(1)}$ sollte bei Verwendung darauf geachtet werden, dass die *treewidth* der Nice-Tree-Decomposition des Problemgraphen gering ausfällt.

Da für jeden Bag innerhalb der Treedecomosition eine Datenmatrix angelegt werden muss, ist die Speicherbelastung recht hoch.

Die Laufzeit zeigt schon bei kleineren Beispielen schnell merkbare Erhöhungen. Da Probleme dieser Art in der realen Welt deutlich größer ausfallen, halten wir diese Technik nur für bedingt praxis tauglich aufgrund ihrer Laufzeit.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[CNP+] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 150–159. IEEE.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1):171–186, 1976.

[Klo94] Ton Kloks. *Treewidth: computations and approximations*, volume 842. Springer Science & Business Media, 1994.

[RS84] Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.