

Levin von Hollen
Tilman Beck

Implementierung der Cut & Count- Technik für das Steinerbaum-Problem

24. November 2016

Inhaltsverzeichnis

1	Einleitung	5
1.1	Notation	5
2	(Nice) Tree Decomposition	6
2.1	Tree Decomposition	6
2.2	Nice Tree Decomposition	6
2.3	Weitere Modifikationen	7
3	Cut & Count-Technik	10
3.1	Einführendes	10
3.2	Cut & Count	10
3.2.1	Cut	10
3.2.2	Count	11
3.3	Monte-Carlo-Algorithmus	11
3.4	Isolation Lemma	11
4	Cut & Count für das Steinerbaum-Problem	13
4.1	Steinerbaum	13
4.2	Cut	13
4.3	Count	15
4.4	Dynamisches Programm	15
4.5	Monte-Carlo Algorithmus und Laufzeit	16
5	Implementierung	18
5.1	Einführung	18
5.2	Nice Tree Decomposition	18
5.3	Dynamisches Programm	19
5.3.1	Implementierung der CountC-Prozedur	19
5.3.2	Berechnung der Färbungen	21
5.4	Evaluierung	23
5.5	Ausblick	23

6 Zusammenfassung	24
Literaturverzeichnis	26

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 Bezeichnung $G[X]$ 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 dieselbe 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 („connected component“).

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 $\omega : U \rightarrow 1, \dots, N$, so bezeichnet $\omega(S) = \sum_{e \in S} \omega(e)$ für $S \subseteq U$.

Für eine Funktion s mit $s[v \rightarrow \alpha]$ 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 $u \in B_x$ 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 $B_x \subseteq B_y$

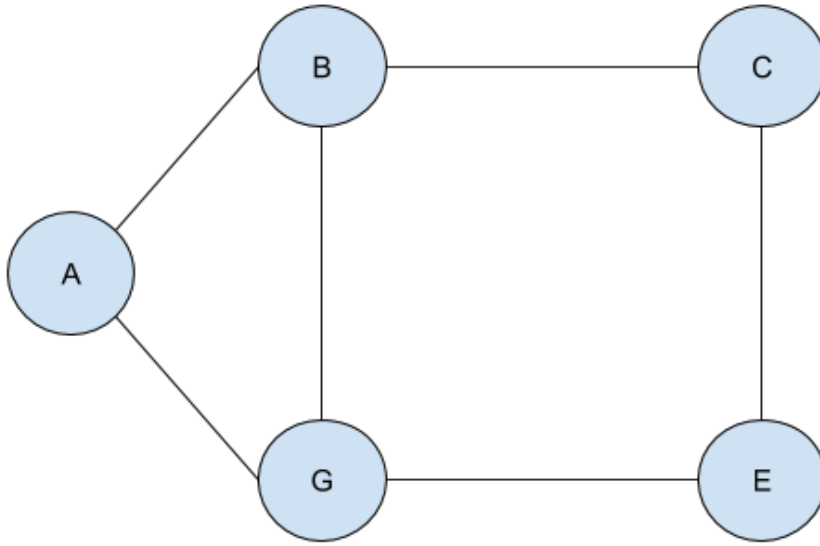


Abb. 2.1 Ursprungsgraph für eine Tree Decomposition

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.

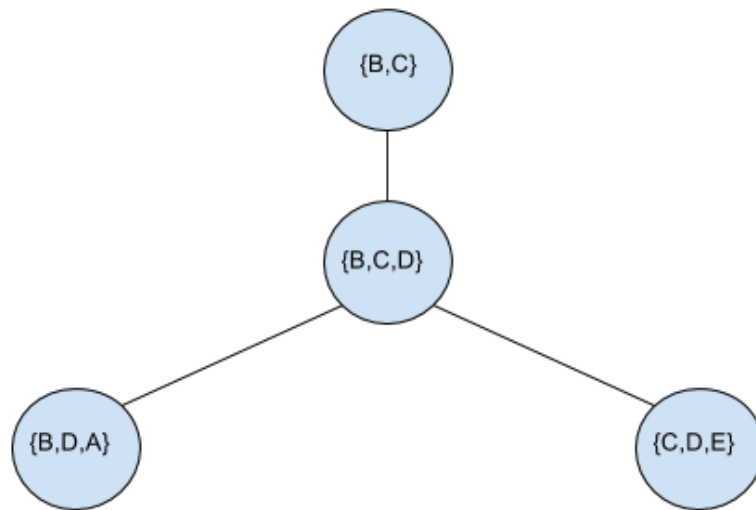


Abb. 2.2 Eine Tree Decomposition für den in Abbildung 2.1 gegebenen Ursprungsgraphen

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 modifiziert werden, so dass das Ergebnis zusätzlich die o.g. Kriterien erfüllt.

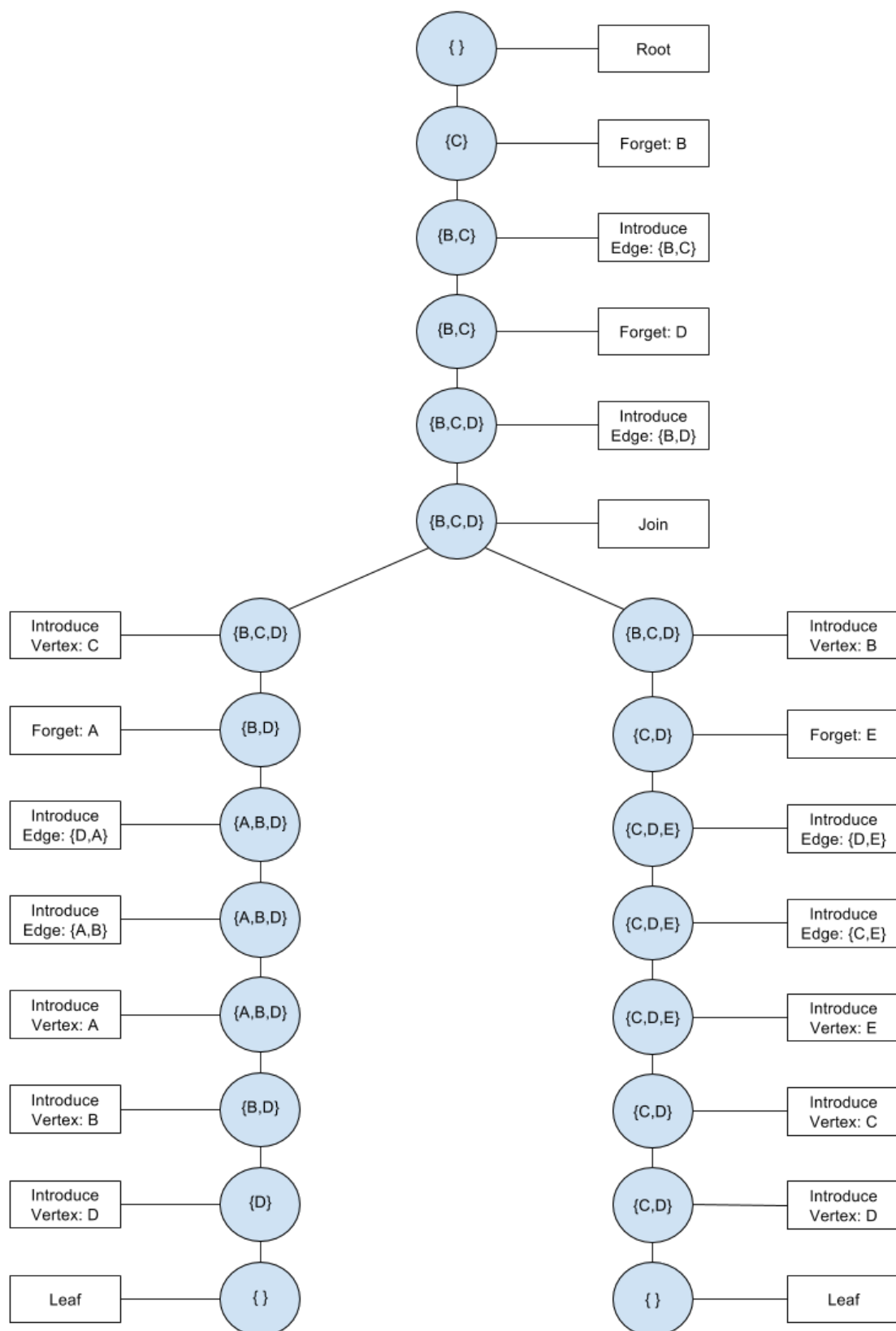


Abb. 2.3 Eine Nice Tree Decomposition für den Ursprungsgraphen in Abbildung 2.1

Kapitel 3

Cut & Count-Technik

3.1 Einführendes

Die Cut & Count-Technik aus [CNP⁺] wird benutzt um Zusammenhangs-Probleme von Graphen zu lösen. Dabei handelt es sich um Graphen-Probleme, bei denen zusammenhängende Submengen der Knoten gefunden werden müssen, die problemspezifische Eigenschaften erfüllen müssen. Die Technik setzt sich aus den beiden Teilen „Cut“, „Count“ zusammen (siehe 3.2) und kann auf mehrere Probleme angewendet werden, u.A. Längster Weg, Steinerbaum oder Feedback Vertex Set.

Das Ergebnis ist ein Monte-Carlo-Algorithmus, welcher eine Aussage über die Existenz einer (oder mehrerer) Lösungen gibt. Dies ist in Section 3.3 erläutert. Mithilfe von Randomisierung beschränkt die Technik die Anzahl der Lösungsmengen mit einer Wahrscheinlichkeit, die durch das Isolation-Lemma begrenzt werden kann. Dies wird in Section 3.4 näher erklärt.

3.2 Cut & Count

Sei \mathcal{S} die Menge der Lösungen für das Graphenproblem. Dann entscheidet Cut & Count ob diese Menge leer ist. Dazu kann Cut & Count folgendermaßen aufgeteilt werden:

3.2.1 Cut

Die Zusammenhangs-Bedingung aus dem Graphenproblem wird abgeschwächt, indem eine Menge von Lösungskandidaten $\mathcal{R} \supseteq \mathcal{S}$ betrachtet wird. Zusätzlich wird die Menge \mathcal{C} von Paaren (X, C) betrachtet, wobei $X \in \mathcal{R}$ und C einen konsistenten Schnitt beschreibt.

3.2.2 Count

Berechne $|\mathcal{C}|$ modulo 2 mithilfe der Prozedur `countC` (siehe Sektion 5.3.1). Dadurch werden alle unzusammenhängenden Lösungskandidaten $X \in \mathcal{R} \setminus \mathcal{S}$ ignoriert, da die Anzahl derer mit einer geraden Anzahl von konsistenten Schnitten konsistent sind. Es bleiben die zusammenhängenden Lösungskandidaten $x \in \mathcal{S}$ übrig.

Der Algorithmus „zählt“ nicht die Lösungen, sondern die Anzahl der Lösungskandidaten, die einen konsistenten Schnitt bilden.

3.3 Monte-Carlo-Algorithmus

Ein Monte-Carlo-Algorithmus ist ein randomisierter Algorithmus, welcher mit einer nach oben beschränkten Wahrscheinlichkeit ein falsches Ergebnis liefern darf. Für die Cut & Count-Technik bedeutet dies, dass der hervorgegangene Algorithmus niemals ein falsch-positives Ergebnis¹ ausgeben kann, aber zu einer gewissen Wahrscheinlichkeit ein falsch-negatives Ergebnis². Die Wahrscheinlichkeit eines falsch-negativ wird durch das Isolations-Lemma bestimmt. Die Ausgabe eines falsch-negativ unterliegt einer bestimmten Wahrscheinlichkeit. Daher ist sinnvoll den Algorithmus in diesem Fall mehrmals hintereinander ausführen. Es begünstigt die Wahrscheinlichkeit eine Lösungsmenge zu finden, falls diese existiert.

Der Monte-Carlo-Algorithmus erreicht folgende Laufzeiten für einige bekannte Zusammenhangs-Probleme bei Graphen, wobei t die Braumbreite der Tree Decomposition beschreibt:

- Steinerbaum in $3^t |V|^{O(1)}$
- Feedback Vertex Set in $3^t |V|^{O(1)}$
- ...

3.4 Isolation Lemma

Da im Count-Teil die Menge mithilfe $|\mathcal{C}|$ modulo 2 berechnet wird, muss sichergestellt werden, dass die Menge der Lösungen begrenzt wird. Für eine gerade Anzahl von Lösungen würde der Algorithmus sonst ein falsch-negatives Ergebnis liefern, da jede gerade Anzahl durch die Modulo-Operation nicht erkannt wird. Daher muss die Lösungsmenge reduziert werden. Dafür wird auf das Isolations-Lemma zurückgegriffen. Dieses ist wie folgt definiert:

Definition 3.1. Isolation Lemma

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

In der Arbeit [CNP⁺] wird das Isolations-Lemma in Lemma 2.5 verwendet, um folgende Aussage über die Wahrscheinlichkeit der Isolation auszusagen:

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.

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

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

Durch die Wahl eines großen N kann die Wahrscheinlichkeit eines falsch-negativ reduziert werden. Dies geht mit einer Erhöhung der Laufzeit des Algorithmus einher.

Kapitel 4

Cut & Count für das Steinerbaum-Problem

4.1 Steinerbaum

Definition 4.1. Steinerbaum-Problem

Eingabe: Sei $G = (V, E)$ ein ungerichteter Graph, $T \subseteq V$ eine Menge von Terminalknoten und k eine positive Ganzzahl.

Problemstellung: Gibt es eine Menge $X \subseteq V$ der Kardinalität k , so dass $T \subseteq X$ und $G[X]$ zusammenhängend ist?

In einem gegebenen Graphen ist eine Menge $T \subseteq V$ als Terminale markiert. Gesucht ist ein Subgraph innerhalb des Ursprungsgraphen, der alle Terminale enthält und genau k Knoten enthält. Zudem muss der Subgraph zusammenhängend sein, es muss also jeder Knoten von jedem anderen Knoten des Subgraphen über einen Pfad innerhalb des Subgraphen erreichbar sein.

4.2 Cut

Zu Beginn des Cut-Teils wird eine zufällige Gewichtsfunktion $\omega : V \rightarrow \{1, \dots, N\}$ definiert. Diese wird für die Isolation der Lösungsmenge verwendet. ω weist jedem Knoten zufällig ein Gewicht zu. Wird ω auf eine Menge von Knoten angewendet, ist das Ergebnis die Summe der einzelnen Knotengewichte.

Anschließend definieren wir die Menge \mathcal{R} , welche die Zusammenhangs-Bedingung abschwächt. Somit ist \mathcal{R}_W die Menge aller Teilmengen von X aus V mit $T \subseteq X$, $\omega(X) = W$ und $|X| = k$. Die Menge \mathcal{R}_W beschreibt alle Lösungskandidaten.

Die Menge $\mathcal{S}_W = \{X \in \mathcal{R}_W \mid G[X] \text{ ist zusammenhängend}\}$ umfasst die Lösungsmenge für eine Menge \mathcal{R}_W . $\cup_W \mathcal{S}_W$ bildet so die gesamte Lösungsmenge. Gibt es ein Gewicht W für das die Menge nicht leer ist, so gibt der Algorithmus eine positive Antwort.

Von der Menge der Terminalknoten wird ein zufälliges Terminal als v_1 -Terminal festgelegt. Dieses dient dazu, dass bei der Bildung von konsistenten Schnitten kein Schnitt doppelt gezählt wird.

Ein Schnitt (V_1, V_2) eines ungerichteten Graphen $G = (V, E)$ ist konsistent, falls $u \in V_1$ und $v \in V_2$ impliziert, dass $uv \notin E$. Ein Subgraph, der einen konsistenten Schnitt aus G bildet, ist ein Paar $(X, (X_1, X_2))$, so dass

(X_1, X_2) ein konsistenter Schnitt von $G[X]$ ist. \mathcal{C}_W sei die Menge aller Subgraphen, die einen konsistenten Schnitt $(X, (X_1, X_2))$ bilden, wobei $X \in \mathcal{R}_W$ und $v_1 \in X_1$.

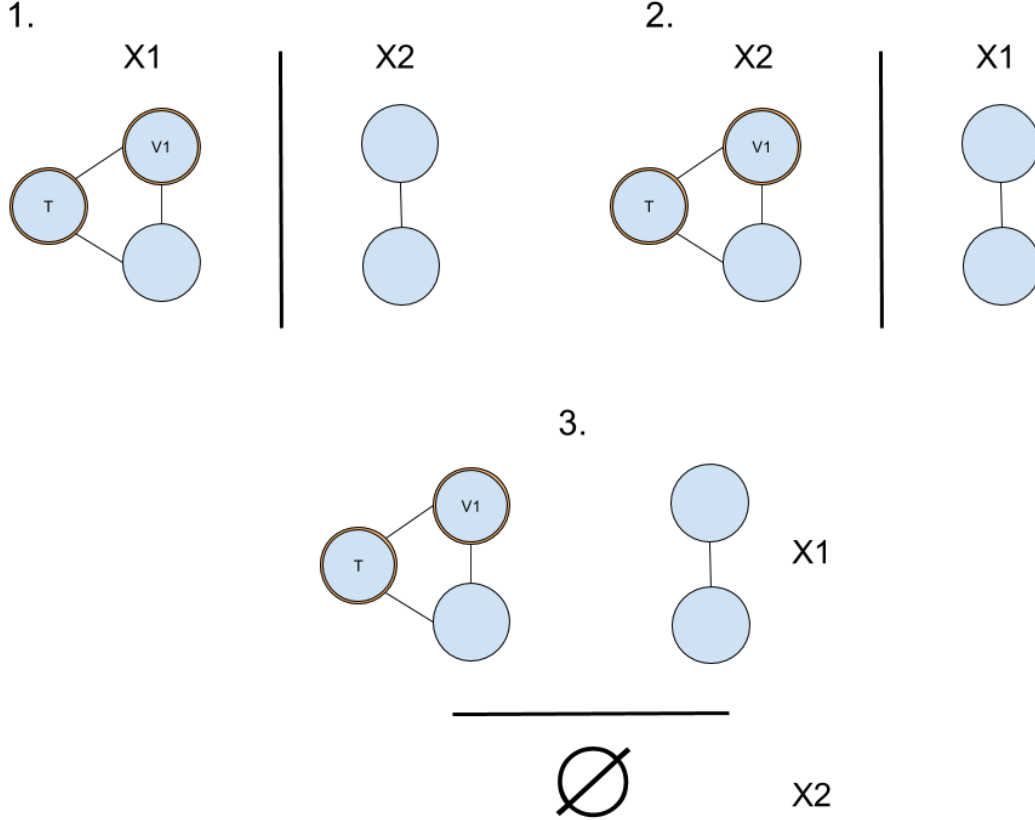


Abb. 4.1 Konsistente Schnitte ohne Beschränkung $v_1 \in X_1$.

Die Anzahl der Subgraphen, welche einen konsistenten Schnitt bilden, sind in Lemma 3.3 in $[CNP^+]$ definiert:

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 $2^{cc(G[X])-1}$.

Ausgehend von der Definition \mathcal{C}_W ist für jeden Subgraphen, der einen konsistenten Schnitt $(X, (X_1, X_2))$ bildet, und für jede Zusammenhangskomponente C aus $G[X]$ bekannt, dass C entweder in X_1 oder in X_2 enthalten sein muss. Für die Zusammenhangskomponente, die v_1 enthält, ist die Seite des Schnitts fest. Für alle anderen Zusammenhangskomponenten kann die Seiten der Mengen X_1, X_2 frei gewählt werden. Daher erhalten wir $2^{cc(G[X])-1}$ verschiedene konsistente Schnitte.

In Abbildung 3 sind die verschiedenen Schnittmöglichkeiten eines Graphen ohne die Beschränkung des v_1 -Terminals visualisiert. Um nicht beide identischen Schnitte (in der Abbildung 1. und 2. Schnitt) zu zählen, wird $v_1 \in X_1$ festgelegt.

4.3 Count

Aus Lemma 3.3 ist bekannt: $|\mathcal{C}| = \sum_{X \in \mathcal{R}} 2^{cc(G[X])-1}$.

Durch die Randomisierung mithilfe der Knotengewichte wird die Wahrscheinlichkeit, dass die Menge der Lösungskandidaten \mathcal{R} mehrere Lösungen enthält, reduziert. Für eine Lösung $\mathcal{S} \in \mathcal{R}$ gilt, dass $G[X]$ zusammenhängend ist. Durch die Festlegung des v_1 -Terminals wird zudem die Möglichkeit des Schnitts für eine Lösung auf eins reduziert (Schnitt mit der leeren Menge). Daher gilt $|\mathcal{S}| = |\{X \in \mathcal{R}\}|cc(G[X] = 1)$. Also können wir schreiben: $|\mathcal{C}_W| \equiv |\mathcal{S}_W|$.

Diese Äquivalenz erlaubt es ein dynamisches Programm zu beschreiben, dass nicht die Menge der Lösungen \mathcal{S} , sondern die Anzahl der Subgraphen $|\mathcal{C}_W|$, die einen konsistenten Schnitt bilden, zu berechnen. Dieses Programm ist in folgender Sektion näher erläutert.

4.4 Dynamisches Programm

Für das dynamische Programm werden für jeden Bag $x \in \mathbb{T}$, die Zahlen $0 \leq i \leq k, 0 \leq w \leq kN$ und die Färbung $s \in \{0, 1_1, 1_2\}^{B_x}$ folgende Mengen definiert:

- $\mathcal{R}_x(i, w) = \{X \subseteq V_x | (T \cap V_x) \subseteq X \wedge |X| = i \wedge \omega(X) = w\}$
beschreibt die Menge der Lösungskandidaten
- $\mathcal{C}_x(i, w) = \{(X, (X_1, X_2)) | X \in \mathcal{R}_x(i, w) \wedge (X, (X_1, X_2)) \text{ ist ein Subgraph, der einen konsistenten Schnitt von } G_x \text{ bildet} \wedge (v_1 \in V_x \Rightarrow v_1 \in X_1)\}$
- $\mathcal{A}_x(i, w, s) = |\{(X, (X_1, X_2)) \in \mathcal{C}_x(i, w) | (s(v) = 1_j \Rightarrow v \in X_j) \wedge (s(v) = 0 \Rightarrow v \notin X)\}|$
beschreibt die Anzahl der Mengen $\mathcal{C}_x(i, w)$, wobei die in X enthaltenen Knoten mit $s \in \{1_1, 1_2\}$ gefärbt sind und die restlichen Knoten mit $s = 0$

Die Färbung gibt an, ob und zu welcher Seite eines konsistenten Schnitts ein Knoten gehört. Die Zuordnung der Färbung $s \in \{0, 1_1, 1_2\}^{B_x}$ der Knoten aus Bag B_x bzgl. der Menge C_x ist folgendermaßen definiert:

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

$A_x(i, w, s)$ zählt alle Möglichkeiten die Knoten eines Bags x gemäß der Definition zu färben. Es gibt für einen Bag 3^{B_x} Möglichkeiten die Knoten zu färben.

Im dynamischen Programm werden die folgenden Berechnungsregeln für jede $A_x(i, w, s)$ Matrix eines Bags x 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 enthalten keine Knoten, daher werden sie mit einem Initialwert gefüllt.
 - **Introduce vertex v:**
 - $A_x = (i, w, s[v \rightarrow 0]) = [v \notin T]A_y(i, w, s)$
Ist der eingeführte Vertex kein Terminal, so wird der Wert aus dem Bag des Kindes übernommen.
 - $A_x = (i, w, s[v \rightarrow 1_1]) = A_y(i-1, w-w(v), s)$
Reduziere beim Zugriff auf den Bag des Kindes i um 1 und ziehe das Gewicht des eingeführten Knoten ab. Übernehme den Wert.
 - $A_x = (i, w, s[v \rightarrow 1_2]) = [v \neq v_1]A_y(i-1, w-w(v), s)$
Ist der eingeführte Vertex nicht das speziell gewählte Terminal so verfähre 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 Vertices 0 gefärbt oder sind beide gleich gefärbt, so wird der Wert aus dem Bag des Kindes übernommen.
 - **Forget vertex v**
 - $A_x(i, w, s) = \sum_{\alpha \in \{0, 1_1, 1_2\}} A_x(i, w, s[v \rightarrow \alpha])$
Es wird die Summe gebildet über alle Färbungen des vergessenen Vertex im Bag des Kindes.
 - **Join bag**
 - $A_x(i, w, s) = \sum_{i_1+i_2=i+|s^{-1}(1_1, 1_2)|} \sum_{w_1+w_2=w+w(s^{-1}(1_1, 1_2))} A_y(i_1, w_1, s)A_z(i_2, w_2, s)$
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.
- Ob der Algorithmus eine Lösung gefunden hat, kann aus der $k \times kN$ -Datenmatrix des Wurzel-Knotens $A_r(i, w, \emptyset)$ ausgelesen werden. Dieser Zugriff ist konstant in $\mathcal{O}(1)$.
Falls eine Lösung existiert und diese gefunden wurde, dann existiert ein $0 \leq W \leq kN$ für das $A_r(k, W, \emptyset) \equiv 2 = 1$.

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, wobei t für die Baumweite der NTD steht.

- 3^t :
Für jeden Bag muss über alle Farbkombinationen iteriert werden. Die obere Grenze hierbei ist die Anzahl der Knoten im größten Bag. Dieser besitzt 3^t verschiedene Färbungen.
- $|V|^{\mathcal{O}(1)}$:
obere Grenze der Eingabe-Parameter k und N .
- Die Wahrscheinlichkeit von $1/2$ für falsch-negativ entsteht durch die Gewichtsfunktion $\omega : V \rightarrow \{1, \dots, N\}$ und durch das Isolations-Lemma, sofern $N = 2|V|$ gesetzt wird.

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.

Der Algorithmus wurde in der Programmiersprache Python¹ in der Version 3.5 entwickelt. Als Entwicklungsumgebung wurde PyCharm[®] von JetBrains² genutzt.

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

¹ <https://www.python.org/>

² <https://www.jetbrains.com/pycharm/>

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 es einen Fall geben in dem jedem Knoten das Gewicht N zugewiesen wird. 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 zu-

sammen, 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 = 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 Färbungen, 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:

$$\begin{aligned} 0 &\rightarrow s = 0 \\ 1 &\rightarrow s = 1_1 \\ 2 &\rightarrow s = 1_2 \end{aligned}$$

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 Zugriffssindices 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 $2 \rightarrow 1_2$) 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 mit verschiedenen Eingabegrößen getestet. Tabelle 5.4 zeigt die Eingabeparameter k und N , die Anzahl der Knoten des Graphen ($|V|$), die Baumweite der NTD sowie die Ergebnisse der Laufzeit in Sekunden.

Eingabe	$\varnothing T$ in s
($k=2$, $N=6$, $ V =3$, $\text{tw}(G)=2$)	~ 0.002
($k=3$, $N=14$, $ V =7$, $\text{tw}(G)=3$)	~ 0.83
($k=4$, $N=32$, $ V =16$, $\text{tw}(G)=3$)	~ 14.23

Tabelle 5.1 Laufzeit des dynamischen Programms für verschiedene Eingabegrößen

Die Tabelle zeigt, dass die Laufzeit mit der Eingabegröße stark ansteigt. Dies lässt die Frage offen, inwiefern der Algorithmus eine Lösung für praxisrelevante Probleme darstellt, da die Graphen sehr viel größer sind als die hier getesteten Beispiel. Die Berechnung über das dynamische Programm bietet dennoch den Vorteil Lösungen x mit $x < k$ direkt aus dem Ergebnis abzulesen.

5.5 Diskussion und Ausblick

Unsere Ergebnisse in Sektion 5.4 eröffnen die Fragestellung inwiefern die Cut Count-Technik das Steinerbaumproblem hinsichtlich Laufzeit und Speicherbedarf effizient löst. Hier bedarf es weiterer Vergleiche mit anderen Ansätzen, wie z. B. einer Brute-Force-Suche, welche für kleinere Graphen womöglich effizienter Lösungen berechnet.

In der Arbeit [CNP⁺] wird zudem nicht auf den Speicherbedarf eingegangen. Dieser ist abhängig von der Baumweite als auch von der Beschaffenheit der Nice Tree Decomposition, da für jeden Bag eine $3^l \times k \times kN$ -Datenmatrix berechnet wird. Für speicherkritische Anwendungen stellt die Cut & Count-Technik keine zufriedenstellende Lösung dar. Eine Optimierung des Speicherbedarfs des dynamischen Programms wurde von uns nicht untersucht.

Die Praxisrelevanz des Algorithmus wird auch durch die Art des Ergebnis verringert. Der Cut & Count-Algorithmus liefert keine Lösung, sondern beantwortet nur die Frage ob eine solche existiert. Für Probleme, die eine Lösungsmenge benötigen, ist er somit nicht geeignet. Der Aufbau des dynamischen Programms lässt auch nicht ohne Weiteres Rückschlüsse auf die Lösungsmenge zu.

Weiterhin ist vermutlich die Wahrscheinlichkeit von $\frac{1}{2}$ für ein falsch-negativ-Ergebnis für die meisten praxisrelevanten Probleme nicht ausreichend. Diese Wahrscheinlichkeit lässt sich durch das Isolations-Lemma zwar weiter verringern, dies wirkt sich jedoch stark auf den Speicherbedarf der Berechnung aus.

Kapitel 6

Zusammenfassung

Das Ergebnis der Anwendung der Cut & Count-Technik auf ein Graphproblem ist ein Monte-Carlo-Algorithmus, der mit einer nach oben beschränkten Wahrscheinlichkeit Aussage über die Existenz eines Ergebnisses liefert. Mittels Randomisierung und dem Isolations-Lemma 3.4 erhalten wir niemals eine falsch-positive Antwort aber zu einer, abhängig von der Wahl eines Parameters, eine falsch-negativ Antwort.

In dieser Arbeit wurde diese Technik für das Steinerbaum-Problem implementiert. Zusätzlich, da die Cut & Count-Technik eine angepasste Nice-Tree-Decomposition 2 benötigt, wurde eine Implementierung entwickelt, welche eine standardmäßige Nice Tree Decomposition in die von der Cut & Count-Technik benötigte Form überführt.

Die Evaluation verschiedener Eingabegrößen warf die Frage auf inwiefern die Count & Count-Technik praxisrelevante Probleme hinsichtlich der Laufzeit und des Speicherbedarfs effizient lösen kann. Für die Technik ist eine effiziente Vorverarbeitung der Nice Tree Decomposition wichtig, da die deren Baumweite und Anzahl der Bags Auswirkungen auf die Laufzeit und den Speicherbedarf hat. Hierbei bedarf es weiterer Forschung in Vergleich mit anderen Ansätzen, wie z. B. einer Brute-Force-Berechnung.

Abbildungsverzeichnis

2.1	Ursprungsgraph für eine Tree Decomposition	7
2.2	Eine Tree Decomposition für den in Abbildung 2.1 gegebenen Ursprungsgraphen	8
2.3	Eine Nice Tree Decomposition für den Ursprungsgraphen in Abbildung 2.1	9
4.1	Konsistente Schnitte ohne Beschränkung $v_1 \in X_1$	14

Tabellenverzeichnis

5.1	Laufzeit des dynamischen Programms für verschiedene Eingabegrößen	23
-----	---	----

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.