UNIVERSITÄT
DUISBURG
ESSEN

Offen im Denken

Bachelorarbeit

zum Thema

Effiziente Graphalgorithmen basierend auf Baumweite und Baumzerlegungen

Mohamad Alkaraazeh
Matrikelnummer: [redacted]

Erstgutachterin:	Prof. Dr. rer. nat. Barbara König
Zweitgutachter:	Prof. Dr. Janis Voigtländer
Betreuerin:	Lara Stoltenow
Hauptfach:	Angewandte Informatik
Semester:	Sommersemester 2023
Abgabedatum:	11.10.2023

Inhaltsverzeichnis

1 Einleitung

Graphen haben in verschiedenen Bereichen Anwendungen wie z.B in der Informatik, Netzwerken und Mathematik. In vielen Fällen können Probleme mittels Graphen erarbeitet werden wie zum Beispiel das Färben von Graphen oder das Finden von Teilmengen mit bestimmten Eigenschaften. Bedauerlicherweise sind viele Entscheidungsprobleme auf Graphen NP-vollständig. Nach unserem Wissen existieren keine Algorithmen, da bis heute nicht bekannt ist, ob P = NP gilt, die solche Probleme in polynomieller Zeit lösen können. Manchmal sind Probleme, die im Allgemeinen NP-schwer sind, lösbar in polynomialer oder sogar linearer Zeit, wenn sie auf Bäume eingeschränkt werden. Es wurde allerdings nachgewiesen, dass viele NP-schwere Probleme wie Hamiltonian Circuit, Dominating Set, die Chromatische Zahl oder der maximalen unabhängigen Menge [AP89, BK08] in linearer oder polynomieller Zeit auf Graphen mit beschränkter Baumweite effizient gelöst werden können. Jedoch erfordert dies in der Regel eine Baumzerlegung als Eingabe. Es ist wichtig zu beachten, dass die Laufzeit dieser Algorithmen oft exponentiell von der Baumweite abhängt, was bedeutet, dass sie für Graphen mit einer großen Baumweite ineffizient sein können. Aus diesem Grund wurde in den letzten 30 Jahren das Konzept der Baumweite entwickelt, um zu beschreiben, wie baumähnlich ein Graph ist. Hierbei wird der Graph in eine Baumstruktur zerlegt, die als Baumzerlegung bezeichnet wird.

Der Begriff *Baumzerlegung* wurde 1986 [RS86] von Neil Robertson und Paul D. Seymour in einer Reihe von Veröffentlichungen zur Graph-Minor-Theorie eingeführt. Eine Baumzerlegung eines Graphen ist ein Baum, bei dem die Knoten den sogenannten Taschen entsprechen, die die Knoten des Graphen beinhalten. Die Struktur des Baums ist so beschaffen, dass sie bestimmte Verbindungseigenschaften des Graphen beibehält [Bod05]. Es ist zwar möglich, eine Baumzerlegung eines Graphen so zu erstellen, bei der alle Knoten in einer einzigen Tasche enthalten sind, jedoch bietet diese Art von Baumzerlegung keine Einblicke in die Struktur des Graphen und ist daher wenig hilfreich. Zur Beurteilung der Effektivität einer Baumzerlegung wurden unterschiedliche Methoden eingeführt. Diese Methoden verfolgen ein gemeinsames Ziel: Die Identifizierung von Baumzerlegungen, die höchsten Anforderungen erfüllen. Anders ausgedrückt, sie streben danach, optimale Baumzerlegungen zu finden.

In den parameterisierten Algorithmen wurden mehrere Algorithmen vorgeschlagen, die sich in der Praxis als effizient erwiesen haben. Eine Gruppe dieser Algorithmen sind Heuristiken zur Approximation der Baumweite eines Graphen, zu denen auch die Min-Fill-In Heuristik gehört.

Die theoretische Grundlage für diese Heuristiken beruht auf der Idee, einen minimalen chordalen Supergraph zu finden, indem so wenige Kanten wie möglich hinzugefügt werden, um die Baumweite zu minimieren. Chordale Graphen sind solche Graphen, die eine perfekte Eliminationsreihenfolge besitzen. Die hinzugefügten Kanten, um einen Graph chordal zu machen, ergibt das sogenannte Fill-In eines Kno-

tens. Die Fill-In-Kanten sind ein charakteristisches Merkmal der Min-Fill-In-Heuristik [BK10].

In dieser Arbeit haben wir darauf verzichtet, einen umfassenden Überblick über alle Algorithmen zur Berechnung von Baumzerlegungen zu geben. Es gibt einfach zu viele verschiedene Methoden, was erschwert, alle möglichen Verfahren zu berücksichtigen und zu erfassen. Unser Hauptziel besteht darin, eine effiziente Methode zur Baumzerlegung zu finden und sie als Werkzeug zur Lösung von Graphproblemen einzusetzen.

Nachdem wir grundlegende Kenntnisse über Graphen und Bäume in Kapitel 2 erworben haben, die aus der Graphentheorie bekannt und wesentlich zur Beschreibung von Baumzerlegungen sind, widmen wir uns im Kapitel 3 den vorgestellten Verfahren (Min-Fill-In) [BK10] zur Berechnung von Baumzerlegungen und anschließend der Baumweite. In diesem Kapitel werden auch chordale Graphen [Sat14] und Eliminierungsreihenfolge [Bod05] behandelt. Auf Grundlage dieser Berechnungen sind wir in der Lage, die schöne Baumzerlegung in linearer Zeit zu bestimmen [Klo94]. Die Idee hinter der Verwendung der schönen Baumzerlegung zur Lösung von Graphproblemen besteht darin, dass sie die Entwicklung von Algorithmen zur Lösung solcher Probleme erleichtert [Bod97a]. Dies geschieht, indem wir die schöne Baumzerlegung verwenden und darauf aufbauend dynamische Algorithmen entwickeln [Bod97a], die diese Probleme in polynomialer Zeit lösen können, vorausgesetzt, dass die Baumweite klein ist. In den letzten beiden Abschnitten dieses Kapitels werden zwei Graphprobleme nämlich das Färbbarkeitsproblem und die maximale unabhängige Menge mit dieser Technik gelöst, wobei auch zwei alternative Verfahren vorgestellt werden, um den Unterschied zwischen den dynamischen Algorithmen und anderen Ansätzen zu verdeutlichen. Kapitel 6 beschäftigt sich mit der Implementierung. Anhand von Beispielen wird im letzten Abschnitt des Kapitels das Laufzeitverhalten der Algorithmen evaluiert.

2 Einführung in die Graphentheorie

Im ersten Abschnitt dieses Kapitel setzen wir den Fokus auf die Definitionen grundlegender Konzepte der Graphentheorie, die dazu beitragen können, ein tieferes Verständnis unseres Themas zu erlangen. Im zweiten Abschnitt beschäftigen wir uns mit Bäumen und ihren Eigenschaften. Die Definitionen in diesem Kapitel orientieren sich an. [KBVH01, Sud16, Wal20]

Definition 2.1 (Graph)
Graphen sind allgemeine Strukturen, die aus einer Menge von Knoten V und einer Menge von Kanten E bestehen, wobei $E \rightarrow V \times V$ eine Abbildung ist. Es gibt verschiedene Varianten von Graphen, wie endliche oder unendliche, gerichtete oder ungerichtete, und Graphen mit oder ohne Mehrfachkanten.

Für die Baumweite und diese Arbeit sind insbesondere folgende Varianten relevant:
Ungerichteter Graph: *In ungerichteten Graphen verläuft der Weg zwischen zwei Knoten in beide Richtungen. Das heißt, der Weg zwischen den beiden Knoten ist bidirektional, und es ist nicht festgelegt, welcher Knoten der Ausgangs- und welcher der Zielknoten ist.*
Endlicher Graph: *Ein Graph ist endlich, wenn sowohl die Knotenmenge V als auch die Kantenmenge E endlich sind, d.h. ($|V| < \infty$ und $|E| < \infty$), was bedeutet, dass es nur eine endliche Anzahl von Knoten und Kanten im Graph gibt.*

Beispiel 2.1

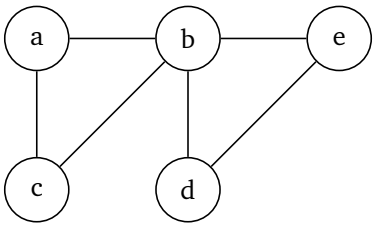


Abbildung 1: Ungerichteter Graph G mit 5 Knoten und 6 Kanten

Definition 2.2 (Pfad)
Ein Pfad in einem Graph G ist eine sequentielle Abfolge von Knoten $(v_0, v_1, v_2, \ldots, v_n)$ sowie eine sequentielle Abfolge von Kanten, so dass $\{v_i, v_{i+1}\} \in E$ für alle $i \in \{0, 1, \ldots, n-1\}$. Ein Pfad wird als Weg bezeichnet, wenn alle Knoten v_i in der Abfolge eindeutig sind.

Definition 2.3 (Nachbarschaft und Grad eines Knotens)
Zwei Knoten u und v sind benachbart, falls $\{u, v\} \in E$. Die Menge der Knoten, die zu Knoten v in Graph benachbart sind, bilden die Nachbarschaft von v. Der Grad eines Knotens ist die Anzahl der Knoten in seiner Nachbarschaft.

Definition 2.4 (K-Clique)
Eine k-Clique in einem Graph besteht aus einer Gruppe von k Knoten, bei der jeder Knoten mit jedem anderen Knoten in der Teilmenge benachbart ist. Beispielsweise bildet eine 2-Clique einfach ein Paar von Knoten, die durch eine Kante verbunden sind. Eine 3-Clique besteht aus drei Knoten, die ein Dreieck bilden.

Definition 2.5 (Baum)
Ein Baum ist ein zusammenhängender ungerichteter Graph, bei dem jedes Paar von Knoten durch genau einen einfachen Pfad verbunden werden kann. [Kor20]

Ein Baum ist tatsächlich eine spezielle Art von Graphen. Alle Bäume sind Graphen, aber nicht alle Graphen sind Bäume. Was Bäume von anderen Graphen unterscheidet, ist ihre spezifische Struktur.

Bäume haben folgende charakteristische Eigenschaften:

1. Richtung: Bäume fließen nur in eine Richtung - von einem Wurzelknoten zu den Leaf-Knoten oder Kindknoten. Es gibt keine Rückverbindungen von den Leaf-Knoten zu einem vorherigen Knoten.

2. Einwegverbindungen: In einem Baum kann ein Kindknoten nur einen Elternknoten haben. Es gibt keine Verzweigungen, bei denen ein Kindknoten mehrere Elternknoten hat.

3. Keine Zyklen: Ein Baum enthält keine Schleifen oder zyklischen Verbindungen. Das bedeutet, dass es keinen Weg gibt, der es erlaubt, zu einem vorherigen Knoten im Baum zurückzukehren.

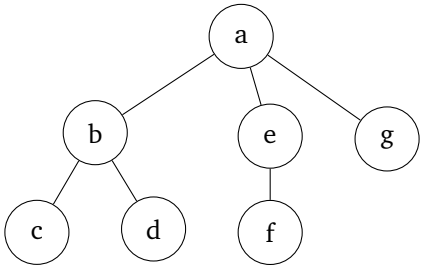


Abbildung 2: Beispielbaum

3 Baumzerlegung und Baumweite

Der Begriff Baumzerlegung kann auf verschiedene Weisen definiert werden. Die traditionelle Definition stammt aus der Struktur der Baumzerlegung. Bei einer Baumzerlegung werden die Knoten eines Graphen in Taschen zusammengefügt, um letztendlich einen Baum zu erhalten. Es existieren nicht nur eine einzelne mögliche Zerlegung, sondern mehrere. Das Ziel dieser Arbeit besteht darin, eine gute Baumzerlegung zu finden.

Die Zerlegung von verschiedenen Arten von Graphen ist möglich, jedoch werden in dieser Arbeit ausschließlich einfache Graphen $G = (V, E)$ betrachtet, **d.h. endliche, ungerichtete Graphen ohne Schleifen oder parallele Kanten**.

In diesem Kapitel werden die Begriffe Baumzerlegung und Baumweite formal definiert und ihre wesentlichen Eigenschaften erläutert. Darüber hinaus wird ein Algorithmus zur Berechnung einer Baumzerlegung ausführlich erklärt.

3.1 Baumzerlegung

Die Idee der Baumzerlegung (Tree Decomposition) besteht darin, das Konzept von Bäumen auf Graphen zu verallgemeinern und sie in eine baumartige Struktur zu überführen. Bei der Baumzerlegung werden die Knoten in Gruppen, sogenannte Taschen I, organisiert und auf eine bestimmte Weise, nämlich in Form einer Baumstruktur, miteinander verbunden. Die Baumweite hingegen beschreibt, wie baumähnlich ein Graph ist.

Formell ist die Baumzerlegung wie folgt definiert [BK10]:

Definition 3.1 (Baumzerlegung)
Gegeben sei ein Graph $G = (V, E)$. Die Baumzerlegung eines Graphen G ist das Paar (X, T), wobei $T = (I, F)$ ein Baum ist, dessen Menge von Baumknoten I hier als Taschen bezeichnet werden, und der Kantenmenge F. Die Taschen definieren eine Sammlung $X = \{X_i \subseteq V \mid i \in I\}$ von Teilmengen der Knotenmenge V von G, sodass folgende Bedingung gilt:

1. $\bigcup_{i \in I} X_i = V$ *(Jeder Knoten taucht in mindestens einer Tasche auf).*
2. *Für alle $\{v, w\} \in E$, es existiert ein $i \in I$ mit $v, w \in X_i$ (Es gibt also für jede Kante eine Menge X_i, so dass die Knoten, die über die Kante verbunden, in einer Tasche liegen.)*
3. *Für alle $v \in V$, $T_v = \{i \in I \mid v \in X_i\}$ bildet einen zusammenhängenden Teilbaum von T.*

Die dritte Bedingung der Baumzerlegung besagt:

- Für alle $i_0, i_1, i_2 \in I$: Wenn i_1 auf dem Pfad von i_0 nach i_2 in T liegt, dann gilt $X_{i_0} \cap X_{i_2} \subseteq X_{i_1}$.

Um eine gültige Zerlegung zu erhalten und den gegebenen Graph in eine Baumstruktur darzustellen, müssen alle drei Bedingungen erfüllt sein.

Beispiel 3.1
Wir betrachten nun den folgenden Graph, der in Abbildung 5 dargestellt ist. Unser Ziel ist, eine gültige Baumzerlegung für den gegebenen Graph zu erstellen und anschließend zu überprüfen, ob diese Baumzerlegung die drei vorgeschriebenen Bedingungen erfüllt. Im nächsten Abschnitt werden wir detailliert erläutern, wie genau diese Baumzerlegung konstruiert wird.

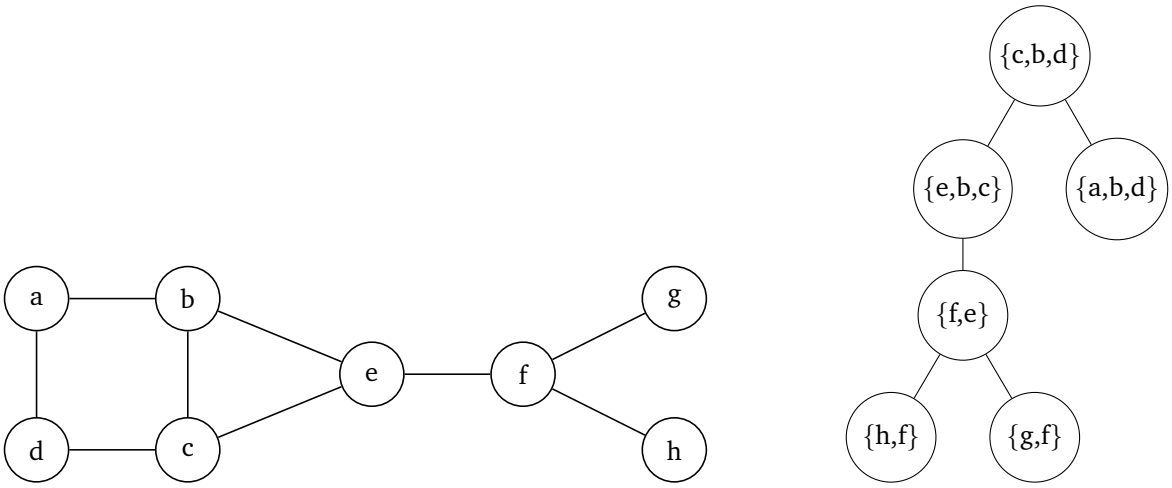


Abbildung 3: Graph G mit entsprechender Baumzerlegung

Auf der linken Seite ist der Graph G abgebildet, während auf der rechten Seite eine gültige Baumzerlegung dargestellt ist. Die Taschen in der Baumzerlegung (X, T) werden durch $X = \{\{c, b, d\}, \{a, b, d\}, \{e, b, c\}, \{f, e\}, \{h, f\}, \{g, f\}\}$ repräsentiert.
Dadurch wird die erste Bedingung der Baumzerlegung erfüllt.
Jede Kante des gegebenen Graphen taucht mindestens in einer Tasche auf, folglich ist die zweite Bedingung erfüllt.
Für die dritte Bedingung betrachten wir einen Knoten im Graph G, beispielsweise den Knoten f. Dann stellen wir fest, dass alle Taschen, die den Knoten f enthalten, miteinander verbunden sind (siehe Abbildung 4). Das gilt natürlich für alle anderen Knoten in Graph G. Daraus folgt, dass die Baumzerlegung korrekt ist.

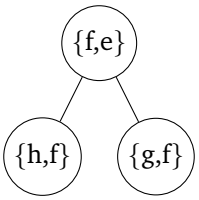


Abbildung 4: Zusammenhängender Teilbaum, der f enthält

3.2 Baumweite

Die Baumweite (Treewidth) eines Graphen ist ein Maß, das zeigt, wie *ähnlich* der Graph zu einem Baum ist.
Durch die Baumzerlegung können wir die Baumweite definieren. Die Baumzerlegung (X, T) teilt den Graph in eine Baumstruktur auf, wodurch die Knoten in spezielle Gruppen (Taschen I) organisiert werden.

Definition 3.2 (Baumweite)

- *Die Weite einer Baumzerlegung (X, T) ist definiert als $max_{\ i \in I} |X_i| - 1$, also die Größe der größten Tasche minus eins.*

- *Die Baumweite eines Graphen G, bezeichnet man mit $tw(G)$, ist die minimale Weite einer Baumzerlegung über alle möglichen Baumzerlegungen von G.*

Bemerkung 3.1
Die in Abbildung 3 dargestellte Baumzerlegung besitzt eine Baumweite von 2. Dies ergibt sich, indem wir die größte Tasche, die 3 Knoten enthält, um eines reduzieren, also 3 - 1 = 2. Dadurch erreichen wir eine Baumweite von 2.

Bemerkung 3.2
Bäume und Wälder haben eine Baumweite von höchstens 1. [Sch89]

Lemma 3.3
Sei (X, T) eine Baumzerlegung von G. Dann gibt es für jede Clique $K \subseteq G$ mindestens eine Tasche $i \in I$ mit $V(K) \subseteq X_i$. [Sch89]

Beweis:
Um die Aussage zu beweisen, wird eine Induktion über die Größe der Cliquen verwendet:

1. *Für $k = 1$ folgt die Aussage direkt aus der ersten Bedingung der Baumzerlegung, da eine einzelne Clique (bestehend aus einem einzigen Knoten) nur eine Tasche sein kann, und somit ist $V(K)$ in dieser Tasche enthalten.*

2. *Für $k = 2$ ergibt sich die Aussage ebenfalls aus der Definition der Baumzerlegung, genauere aus der zweiten Bedingung der Baumzerlegung. Für eine Clique mit zwei Knoten ist es erforderlich, dass beide Knoten in derselben Tasche der Baumzerlegung enthalten sind, wie es in der Bedingung 2 der Baumzerlegung 3.1 vorgeschrieben ist.*

3. *Induktionsannahme: Angenommen, die Aussage ist für alle Cliquen mit $k \geq 2$ richtig.*

4. *Induktionsschritt: Sei nun $|V(K)| = k > 2$, $v \in V(K)$, und $K' = K \setminus \{v\}$. Gemäß der Induktionsannahme gibt es eine Tasche i', für die gilt $V(K') \subseteq X_{i'}$.*
   Ist nun $v \notin X_{i'}$, so gibt es aufgrund der dritten Bedingung der Baumzerlegung einen eindeutig bestimmten verbundenen Teilbaum, der alle Taschen mit $v \in X_s$ enthält. Sei i die erste Tasche auf dem Weg von i', deren zugeordnete Menge X_i den Knoten v enthält. Aufgrund der zweiten und dritten Bedingung der Baumzerlegung folgt dann, dass $V(K) \subseteq X_i$.

Damit ist die Aussage für den Induktionsschritt bewiesen, und somit gilt die Aussage für alle Cliquen K im Graph.

Beispiel 3.2
Sei G ein Graph (Abbildung 5 links) mit 4 Knoten $\{a, b, c, d\}$. Drei dieser Knoten bilden eine 3-Clique (vollständig verbundener Teilgraph mit drei Knoten, $\{a, b, c\}$). Nun möchten wir eine Baumzerlegung für diesen Graph erstellen und dabei auf die drei Bedingungen der Baumzerlegung achten.
Wenn wir eine Baumzerlegung für den gegebenen Graph erstellen, die alle drei Bedingungen erfüllt, sollten wir auch Lemma 3.3 in Betracht ziehen. Dieses Lemma besagt,

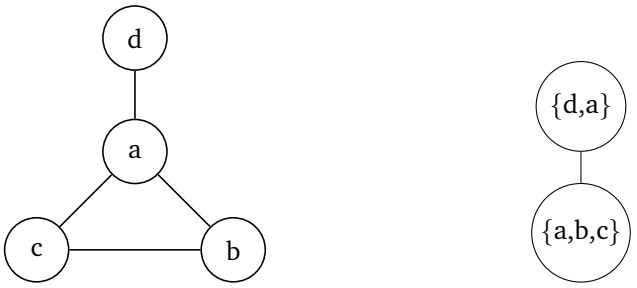


Abbildung 5: Graph G, der eine Clique hat, und eine gültige Baumzerlegung

dass alle Knoten, die eine Clique bilden, derselben Tasche angehören. Anders ausgedrückt liegen die Knoten $\{a, b, c\}$, die eine Clique bilden, in derselben Tasche. Die von uns erstellte Baumzerlegung in Abbildung 5 rechts erfüllt ebenfalls sämtliche Anforderungen der Baumzerlegung.

Lemma 3.4
Wenn H ein Teilgraph von G ist, dann hat H eine Baumweite kleiner oder gleich der Baumweite von G, d.h. $tw(H) \leq tw(G)$. [Sat14]

Beweis:
Sei(X, T) eine Baumzerlegung der Graph G mit Baumweite $tw(G)$. Von der Baumzerlegung für G können wir die Baumzerlegung für H konstruieren (Y, T), wobei die Teilmengen Y_t wie folgt definiert sind:

$$Y_t = X_t \cap V(H).$$

Offensichtlich hat (Y, T) eine Baumweite von höchstens $tw(G)$.

Bemerkung 3.5
Jeder Graph mit einer Baumweite von k enthält einen Knoten, der einen Grad von höchstens k hat. [Sat14]

3.3 Algorithmus zur Berechnung von Baumzerlegung

Die genaue Bestimmung der Baumweite eines Graphen und die Erstellung einer Baumzerlegung minimaler Breite sind äußerst schwierige Probleme, die zur Klasse der NP-schweren Probleme gehören [BB05].

Die Wahl zwischen exakten und heuristischen Algorithmen ist eine zentrale Frage bei der Lösung komplexer Probleme in der Graphentheorie. Während exakte Algorithmen garantierte optimale Lösungen bieten, sind sie in der Praxis oft sehr ineffizient, insbesondere bei größeren Graphen. Heuristische Algorithmen hingegen sind in der Lage, schnell zu arbeiten und praktische Ergebnisse zu liefern, ohne die Optimalität zu garantieren. [BFK+06]

Aus diesem Grund greift man in der Realität häufig auf heuristische Algorithmen zurück, die obere Schranken für die Baumweite liefern und somit Baumzerlegungen

mit beschränkter Baumweite generieren. Diese heuristischen Ansätze bieten zwar keine exakte Lösung, liefern jedoch oft schnellere Ergebnisse mit akzeptabler Genauigkeit. Dieser Ansatz liefert häufig gute Schätzungen für die Baumweite. Obwohl die genaue Bestimmung der Baumweite eine anspruchsvolle Aufgabe bleibt, ermöglichen diese heuristischen Algorithmen praktische und effiziente Lösungen für verschiedene Probleme in der Graphentheorie.

In diesem Kapitel betrachten wir den sogenannten *Minimum Fill-in Algorithmus*, der auf der Beziehung zwischen der Baumweite und den Triangulationen von Graphen basiert, um die Baumweite des gegebenen Graphen zu berechnen.
Bevor wir jedoch mit dem Algorithmus beginnen, führen einige Begriffe ein. [BK10, Sat14]

Definition 3.3 (Chordal Graph)
Ein Graph $G = (V, E)$ wird als chordal bezeichnet, wenn jeder Zyklus Z in G mit einer Länge von mindestens vier Kanten mindestens eine Sehne (Chord) enthält. Eine Sehne *ist eine Kante, die zwei Knoten in Z miteinander verbindet, die bisher entlang des Zyklus Z nicht direkt benachbart waren.*

Bemerkung 3.6
Jeder induzierte Teilgraph $G[W]$, $W \in V$, eines chordalen Graphen G, der einen Kreis bildet, ist ein Dreieck (Triangle). Das bedeutet, dass jeder Zyklus in einem chordalen Graph entweder aus drei Knoten besteht und in Form eines Dreiecks vorliegt oder länger sein kann und mindestens eine Sehne aufweist, die als zusätzliche Kante zwischen zwei nicht benachbarten Knoten hinzugefügt wird.
Chordale Graphen werden auch als triangulierte Graphen bezeichnet.

Beispiel 3.3
Wir richten unsere Aufmerksamkeit auf den folgenden Graph in Abbildung 6. Links ist der Graph G dargestellt, der jedoch nicht als chordaler Graph klassifiziert werden kann. Der Grund dafür liegt darin, dass er zyklische Struktur ohne zusätzliche Verbindungen aufweist, wie den zyklischen Pfad $\{a, b, c, d, e\}$. Solche zyklischen Strukturen, die keine zusätzlichen Kanten aufweisen, werden als sehnenlose (chordless) Zyklen bezeichnet.
Hingegen wird auf der rechten Seite der Abbildung 6 ein anderer Graph H gezeigt, der von G abgeleitet werden kann durch Einfügen zusätzliche Kanten. Eine Möglichkeit wäre, eine Kante zwischen $\{b, c\}$ und eine weitere zwischen $\{b, d\}$ hinzuzufügen. Mit diesen neuen Kanten ist der Graph jetzt chordal.

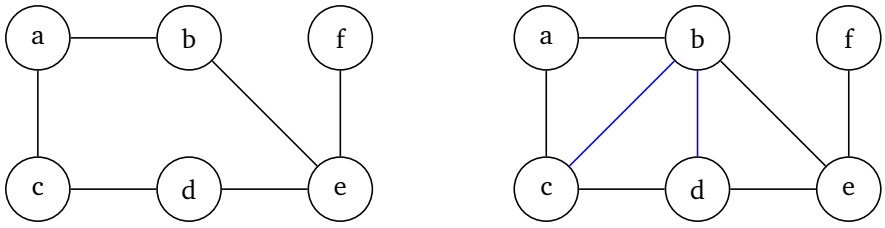


Abbildung 6: Chordaler Beispielgraph

Definition 3.4 (Perfekte Eliminierungsreihenfolge)
Eine Eliminierungsreihenfolge (Elimination ordering) eines Graphen ist eine Bijektion $\pi : V \to \{1, ..., |V|\}$. *Sie wird als perfekt bezeichnet, wenn für jeden Knoten* $v \in V$ *die Menge der höher nummerierten Nachbarn* $N_G^{\pi}(v) := \{w \in N(v) | \pi(w) > \pi(v)\}$ *eine Clique im Graph* G *bilden.*

Theorem 3.7
Sei $G = (V, E)$ *ein Graph. Die folgenden Aussagen sind äquivalent:*

1. *G ist ein chordaler Graph.*
2. *G hat eine perfekte Eliminierungsreihenfolge, d.h. eine Permutation der Knoten, so dass für jeden Knoten seine höher nummerierten Nachbarn eine Clique bilden.*

wir werden hier keinen Beweis für das Theorem vorlegen, allerdings kann er in [Gol80, Theorem 4.1] gefunden werden.

Beispiel 3.4
Wir betrachten den folgenden Chordalen Graph G, der in Abbildung 7 dargestellt ist.

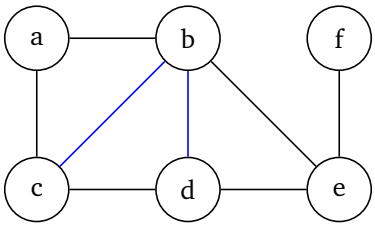


Abbildung 7: Chordaler Graph

- *Der Graph ist Chordal und gemäß Theorem 3.7 besitzt er eine perfekte Eliminierungsreihenfolge.*
- *Eine perfekte Eliminierungsreihenfolge sei* $\pi = \{f, e, d, b, c, a\}$. *Wenn wir den Knoten e in Betracht ziehen, wird offensichtlich, dass wenn e aus* π *eliminiert würde, bilden seine nachfolgenden Nachbaren in* π *eine Clique. Genauer, bilden die Knoten* $\{b, d\}$ *eine 2-Clique. Das gilt auch für die Nachbarn des Knotens d, nämlich die Knoten* $\{b, c\}$, *die in* π *nach d folgen. Diese Eigenschaft zieht sich auf alle anderen Knoten in* π *und daher ist* π *perfekte Eliminierungsreihenfolge für den gegebenen Graph.*

3.3.1 Obere Schrankenheuristiken

Es gibt mehrere obere Schranken für die Baumweite eines Graphen, die aus vielen verschiedenen Charakterisierungen der Baumweite resultieren. Die meisten dieser Charakterisierungen werden als Optimierungsprobleme definiert. Um die Baumweite von allgemeinen Graphen zu charakterisieren, nutzen wir die Tatsache, dass alle Graphen zu einem erweiterten chordalen Graphen gemacht werden können. Dies wird erreicht, indem ein Graph trianguliert wird. Formell ist eine Triangulierung

eines Graphen $G = (V, E)$ ein chordaler Graph $G^{\triangle} = (V, E^{\triangle})$ mit $E \subseteq E^{\triangle}$. Zusätzlich wird sie als minimal bezeichnet, wenn sie keine Triangulierung von G mit weniger Kanten enthält.

Die Triangulierung eines Graphen G erfolgt durch das Hinzufügen von Kanten zu G, um ihn chordal zu machen. Dabei werden Kanten zu Zyklen der Länge vier oder mehr hinzugefügt. Eine Kante wird zu einem Graph hinzugefügt, wenn sie Teil eines Dreiecks (Zyklus der Länge 3) in G ist. Dies bedeutet, dass die hinzugefügte Kante zusammen mit den bereits vorhandenen Kanten ein Dreieck bildet. Dieser Prozess wird solange wiederholt, bis der gesamte Graph G zu einem chordalen Graph $G^{\triangle}$ umgewandelt ist. Dieser Prozess wird als *Fill in* bezeichnet. Ihr Ziel ist es, eine Triangulierung von G zu finden, deren Cliquenzahl so klein wie möglich ist, um somit eine gute obere Schranke für die Baumweite $tw(G)$ zu finden. Bei größeren Cliquen werden auch die kleineren berücksichtigt (Zum Beispiel: ein 4-Clique-Graph enthält unter anderem auch vier 3- Cliquen an derselben Stelle).
Die Heuristiken im Algorithmus 1 nutzen dabei denselben Ansatz wie bei [BK10, Algorithmus 3].

Algorithmus 1 Eliminierungsreihenfolge

Eingabe: Graph $G = (V, E)$
Ausgabe: Eliminierungsreihenfolge π
  for $i = 1$ to $|V|$ **do**
    Wähle einen Knoten v aus H gemäß einem Kriterium χ
    Setze v an die i-te Position in der Eliminierungsreihenfolge π
    Verbinde jeweils jeden nicht benachbarten Nachbarn von v durch eine Kante
    (mach die Nachbarschaft von v zu einer Clique)
    Entferne v und die dazugehörigen Kanten aus H
  end for
  return π.

Unterschiedliche Kriterien χ in diesem Verfahren führen zu unterschiedlichen Baumweiten Heuristiken [Bod05]. Diese Bezeichnung leitet sich von der Tatsache ab, dass diese Heuristik den Prozess des *Fill-in* nutzt und Kriterien verwendet, um die Knoten so auszuwählen, dass die minimale Anzahl von zusätzlichen Kanten benötigt wird, um ihre Nachbarknoten zu verbinden und somit eine Clique zu bilden. Dieses Verfahen wird als *Minimum Fill-In* bezeichnet.

Um die Heuristik deterministisch [1] zu machen, werden die Knoten zuerst in eine aufsteigender Reihenfolge sortiert, bevor das Min-Fill-In Verfahren gestartet wird. Dies hilft dabei, eventuelle Gleichstände aufzulösen und die Heuristik eindeutig anzuwenden. Im Algorithmus 2 zeigen wir die Version des Min-Fill-in-Verfahrens, die sowohl die Baumweite als auch die Baumzerlegung zurückgibt.

[1] Um sicherzustellen, dass bei gleicher Eingabe immer das gleiche Ergebnis zurückkommt

Algorithmus 2 Min-Fill-in Heuristik

Eingabe: Graph $G = (V, E)$.
Ausgabe: Baumweite tw und die entsprechende Baumzerlegung.

```
G^△ ← G
tw ← 0
if |V| = 1 then
    return Baumzerlegung mit einer Tasche X_{v_1} = {v_1}
end if
for i ≥ 2 to |V| do
    N ← Sortierte Knotenmenge nach ihrem Grad in aufsteigende Reihenfolge.
    v ← Knoten mit optimale Priorität in G^△ gemäß χ
    Tasche konstruieren X_v = N_G(v)
    G^△ ← (G^△ − v) + clique(N_{G^△}(v))          //Eliminierung von v
    tw ← max{|X_v| − 1, tw}
end for
Taschen verbinden
return Baumweite tw und dazugehörige Baumzerlegung
```

Der Min-Fill-In Algorithmus besteht aus folgenden Schritte:

1. Knoten in einem gegebenen Graph nach ihrem Grad in aufsteigender Reihenfolge sortieren.

2. Knoten auswählen:
   Wähle einen Knoten aus dem Graph, der die minimale Anzahl von Kanten benötigt (optimale Priorität), um seine Nachbarn zu verbinden. Der Grad dieses Knotens sollte minimal sein. Dies wird erreicht, indem wir den ersten Knoten in der sortierten Reihenfolge auswählen und die Anzahl der benötigten Kanten berechnen. Falls die Anzahl für alle Knoten gleich ist, wählen wir den ersten Knoten in dieser Reihenfolge aus.

3. Tasche erstellen:
   Füge den ausgewählten Knoten und seine direkten Nachbarn in eine Tasche ein, ohne dabei die Bedingung 2 der Baumzerlegung 3.1 zu verletzen.

4. Entferne den ausgewählten Knoten aus dem Graph, und alle Kanten, die mit diesem Knoten verbunden sind.

5. Verbinde die Nachbarn der ausgewählten Knoten miteinander, falls sie noch nicht miteinander verbunden sind, um die Nachbarn dieses Knotens zu einer Clique zu machen.

6. Baumweite Berechnen
   Die Baumweite wird berechnet, indem 1 von der Größe der Tasche aus Schritt 3 abgezogen wird, und der größere Wert zwischen dem aktuellen und dem vorherigen Wert der Baumweite ausgewählt wird.

7. Wiederhole die Schritte 1 bis 6.

8. Taschen verbinden
   Am Ende erfolgt die Verbindung der Taschen miteinander, wobei die letzte

Tasche als Wurzelknoten gewählt wird. Anschließend werden die verbleibenden Taschen mit dem Wurzelknoten verbunden. Dies geschieht, indem die Nachbarschaft der eliminierten Knoten in einer Tasche eine Teilmenge der eliminierten Knoten in der Wurzeltasche ist. Wenn keine Taschen mehr mit dem Wurzelknoten verbunden werden können, werden die verbleibenden Taschen mit den Kindknoten des Wurzelknotens verbunden. Dieser Vorgang wird wiederholt, bis keine Taschen mehr übrig sind.
Das Beispiel 3.5 verdeutlicht die Vorgehensweise beim Verbinden von Taschen.

Nachfolgend erläutern wir die Schritte des Min-Fill-in Algorithmus anhand eines Graphen, der in Abbildung 8 (a) dargestellt ist.

Beispiel 3.5
Wie bereits angekündigt, beginnen wir damit, die Knoten in G nach ihrem Grad in aufsteigender Reihenfolge zu sortieren $N = \{g, h, a, d, f, e, b, c\}$. Anschließend wird ein Knoten ausgewählt, der die optimale Priorität gemäß Min-Fill-In Kriterien aufweist.

In diesem Fall nehmen wir den ersten Knoten aus der sortierten Reihenfolge, den Knoten g, und berechnen wie viele Kanten benötigt werden, um seine Nachbarschaft zu verbinden. Wir stellen fest, dass keine Kante benötigt wird. Das bedeutet, dass Knoten g hier die optimale Priorität hat. Knoten g wird ausgewählt.
Die Knoten Anordnung hilft uns hier die Knoten mit optimaler Priorität schnell zu finden. Falls Knoten g in diesem Fall Kanten benötigt würde, um die Nachbarn zu verbinden, würden wir weiter in der Knotenreihenfolge suchen, bis wir einen Knoten finden, der weniger Kanten benötigt. Sollten alle Knoten gleich viele oder mehr Kanten benötigen, würden wir den ersten Knoten in der Reihenfolge auswählen.
Wir fügen den Knoten g und seine Nachbarknoten in einer Tasche zusammen. Die Menge aller Taschen ist nun $X = \{\{g, f\}\}$ ist. Anschließend eliminieren wir den Knoten g und dazugehörigen Kanten aus dem ursprünglichen Graph G.
Die Differenz zwischen der Größe der Tasche und 1 ergibt 2 - 1 = 1. Da der ursprüngliche Wert der Baumbreite null ist, wählen wir den größeren Wert von beiden, in diesem Fall 1, und speichern diesen Wert für den nächsten Durchlauf.
Die Abbildung 8 (b) zeigt den Graph G nach dem Entfernen von Knoten g.

Nun wiederholen wir die Schritte von vorne und sortieren die verbleibende Knoten $N = \{h, f, a, d, b, c, e\}$. Wie zuvor wählen Wir den ersten Knoten aus N aus und berechnen, wie viel Kanten benötigt werden, um die Nachbarschaft von h zu verbinden. Auch in diesem Fall werden keine Kanten benötigt, und daher wird h ausgewählt. Wir fügen h und seine Nachbarn in einer Tasche hinzu und aktualisieren die Menge$X = \{\{g, f\}, \{h, f\}\}$. Anschließend eliminieren wir den Knoten h und die zugehörigen Kanten aus dem Graph G. Die Baumweite bleibt 1, da die Tasche $\{h, f\}$ eine Größe von 2 hat, was eine Baumweite von 1 ergibt, und der vorherige Wert ebenfalls 1 war. Abbildung 8 (c) zeigt den Graph G nach dem Eliminieren von Knoten h.

Das gleiche gilt für den Knoten f und seiner Nachbarknoten e. Wir aktualisieren die Menge $X = \{\{f, g\}, \{f, h\}, \{f, e\}\}$. Für den knoten e gilt dasselbe wie zuvor. Die sortierte Knotenmenge ist $N = \{e, a, b, c, d\}$. Wir fügen e und seine Nachbarn zur Menge $X = \{\{f, g\}, \{f, h\}, \{e, f\}, \{e, b, c\}\}$ hinzu und eliminieren e sowie die dazugehörigen Kanten aus dem Graph G. Die Tasche $\{e, b, c\}$ hat eine Größe von 3, was zu einer Baumweite von 2 führt. Der vorherige Wert der Baumweite war 1, daher aktualisieren wir ihn auf 2. Abbildung 8 (d) (e) zeigen den Graph G nach dem Entfernen der Knoten f und e.

Wie Sie vielleicht bemerkt haben, mussten wir bei vorherigen Knoten die Nachbarschaft nicht verbinden, aber dies ist nicht der Fall beim nächsten Knoten.

Jetzt hat der Graph vier Knoten $\{a, b, c, d\}$, wie in Abbildung 8 (e) gezeigt. Wir sortieren die verbleibenden Knoten in aufsteigender Rheinfolge $N = \{a, b, c, d\}$ und wählen zunächst Knoten a aus. Dann berechnen wir, wie viele Kanten benötigt werden, um die Nachbarschaft von a zu verbinden, in diesem Fall wird eine Kante benötigt. Wir wählen den nächsten Knoten in N aus und wiederholen die Berechnung. Bei der Berechnung wird ebenfalls für alle anderen Knoten in N eine benötigt, um ihre Nachbarschaft zu verbinden. Daher wählen wir den Knoten a aus. Wir fügen a sowie seine Nachbaren $\{b, d\}$in eine Taschen hinzu und löschen den Knoten a und verbundenen Kanten aus dem Graph G. Wir verbinden die Nachbarknoten von a, den Knoten $\{b, d\}$, miteinander durch eine Kante. Die Größe der Tasche beträgt 3, was eine Baumweite von 2 ergibt. Da die vorherige Wert der Baumweite bereits 2 war, ist eine Aktualisierung dieses Werts nicht erforderlich.

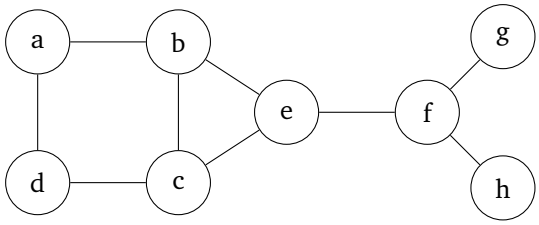
Am Ende bleiben nur noch drei Knoten übrig, die eine Clique bilden. Gemäß Lemma 3.3 fügen wir alle drei Knoten in eine Tasche, $X = \{\{f, g\}, \{f, h\}, \{e, f\}, \{e, b, c\}, \{a, b, d\}, \{b, d, c\}\}$. Die Baumweite bleibt auch 2, da die größte Tasche 3 Knoten hat. Es gibt keine Knoten mehr übrig, das bedeutet, dass wir mit der Zerlegung fertig sind.

Die Aufgabe besteht nun darin, die Taschen miteinander zu verbinden und dabei einen Baum zu erstellen. Eine Tasche wird als Wurzel ausgewählt. In unserem Fall wird immer die letzte Tasche als Wurzel festgelegt, d.h. die Tasche $\{b, d, c\}$. Anschließend suchen wir in den anderen Taschen nach Knoten, deren Nachbarschaft - die Knoten, die in der selben Tasche enthalten sind - eine Teilmenge der Knoten in der Wurzelknoten ist.
Um dies zu verdeutlichen, nehmen wir die Tasche $\{a, b, d\}$. Die Tasche wird konstruiert, indem wir den Knoten a und seine Nachbarknoten in einer Tasche zusammenfügen und den Knoten a aus dem Graph G entfernen. Das bedeutet, dass a der eliminierte Knoten ist und die Knoten $\{b, d\}$ seine Nachbarn in dieser Tasche sind. Es kann sein, dass Knoten a im ursprünglichen Graph mehr als zwei Nachbarn hat, aber hier geht es nur um die Knoten, die sich in derselben Tasche befinden. Wir prüfen nun, ob diese Nachbarschaft eine Teilmenge der Knoten in der Wurzelknoten $\{b, d, c\}$ ist. In diesem Fall ist die Nachbarschaft von a in der Tasche $\{a, b, d\}$ tatsächlich eine Teilmenge der Wurzel, also verbinden wir die Wurzelknoten mit der Tasche $\{a, b, d\}$.

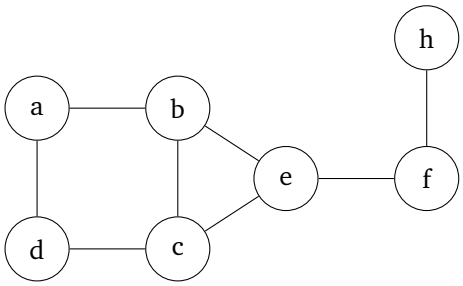
Nun wird die Tasche $\{e, b, c\}$ betrachtet. Wir untersuchen, ob die Nachbarschaft der

eliminierten Knoten e eine Teilmenge der Knoten in der Wurzelknoten ist. Die Nachbarschaft von e in der Tasche sind $\{b, c\}$. Wir stellen fest, dass die Nachbarschaft von e eine Teilmenge der Knoten in der Wurzelknoten ist, und daher wird die Tasche $\{e, b, c\}$ mit der Wurzelknoten verbunden.
Wir wiederholen diesen Vorgang für alle weiteren Taschen in X und stellen fest, dass keine weiteren Taschen mit der Wurzelknoten verbunden werden können.

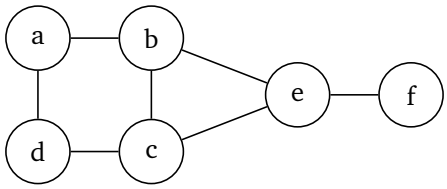
Dann wird die Tasche $\{b, d, c\}$ aus der Menge $X = \{\{g, f\}, \{h, f\}, \{f, e\}, \{e, b, c\}, \{a, b, d\}\}$ entfernt.
Zunächst betrachten wir die letzte Tasche $\{a, b, d\}$ in der Taschenmenge X. Wir prüfen, ob die Nachbarschaft der eliminierten Knoten in der verbleibenden Taschen eine Teilmenge von Knoten in der Tasche $\{a, b, d\}$ ist. Wir stellen fest, dass keine der verbleibenden Taschen in X diese Bedingung erfüllt. Das bedeutet, dass $\{a, b, d\}$ keine Nachfolger hat.
Analog können alle Taschen in X miteinander verbunden werden. Nachdem alle Taschen verbunden sind, überprüfen wir, ob die Taschen, die gemeinsame Knoten haben, einen zusammenhängenden Teilbaum bilden oder nicht. Hierbei erfüllt die Baumstruktur die Bedingungen. Die Baumzerlegung des Graphen G ist in Abbildung 9 dargestellt.



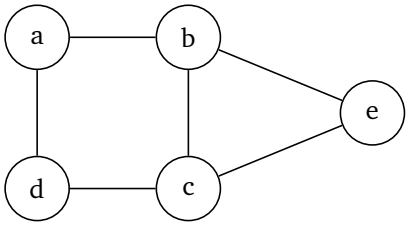
(a) Zu zerlegende Graph G,
$N = \{g, h, a, d, f, e, b, c\}$



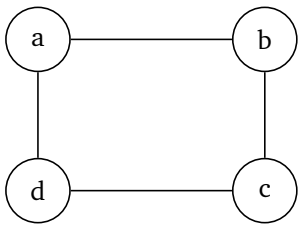
(b) $X = \{\{g, f\}\}, N = \{h, f, a, d, b, c, e\}$



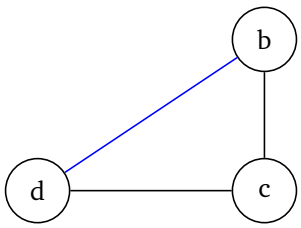
(c)
$X = \{\{g, f\}, \{h, f\}\}, N = \{f, a, d, b, c, e\}$



(d) $X = \{\{g, f\}, \{h, f\}, \{f, e\}\}$,
$N = \{e, a, b, c, d\}$



(e) $X = \{\{g, f\}, \{h, f\},$
$\{f, e\}, \{e, b, c\}\}$,
$N = \{a, b, c, d\}$



(f)
$X = \{\{g, f\}, \{f, h\}, \{e, f\},$
$\{e, b, c\}, \{a, b, d\}, \{b, d, c\}\}$

Abbildung 8: Graph G nach dem Entfernen jedes Knotens

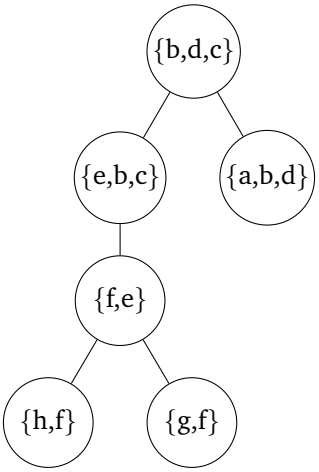


Abbildung 9: Baumzerlegung der Graph G in Abbildung 8

4 Schöne Baumzerlegung

Im Kontext der algorithmischen Nutzung der Baumweite erweist es sich als hilfreich, dass wir uns auf bestimmte Baumzerlegungen, die als *schöne Baumzerlegung (Nice Tree Decomposition)* bezeichnet werden, beschränken können.

Die schöne Baumzerlegung, auch bekannt als *Angenehme* Baumzerlegung, ist eine spezielle Form der Baumzerlegung, die zusätzliche Eigenschaften aufweist.
Die vorliegenden Definitionen und Aussagen in diesem Abschnitt verfolgen den in [Klo94] beschriebenen Ansatz zur Entwicklung des gesuchten Algorithmus. Dieser Algorithmus ermöglicht uns, eine schöne Baumzerlegung in lineare Zeitkomlexität zu generieren. Die Grundidee dieses Algorithmus besteht darin, von einer gegebenen Baumzerlegung auszugehen und daraus eine schöne Baumzerlegung mit einer Baumweite von k zu erstellen. Die schöne Baumzerlegung besitzt die gleiche Baumweite wie bei der Baumzerlegung. Dafür bearbeitet man den gegebene Baum vom Wurzel bis Leaf-Knoten hin ab und bildet für jeden Knoten eine oder mehrere Nachfolger, die die schöne Baumzerlegung repräsentieren.

Im weiteren Verlauf dieser Arbeit nutzen wir die schöne Baumzerlegung, um die Probleme in leichter lösbare Teilprobleme zu zerlegen und daher wenige fälle betrachten, da bei der schönen Baumzerlegung weniger Änderungen zwischen den Knoten auftreten als bei der normalen Zerlegung. Die Verwendung von schönen Baumzerlegungen im Vergleich zu normalen Baumzerlegungen erweitert im Allgemeinen zwar nicht die algorithmischen Möglichkeiten, erleichtert jedoch erheblich die Entwicklung von Algorithmen. In vielen Fällen führt dies zu besseren Konstantenfaktoren in den Laufzeiten der Algorithmen, die schöne Baumzerlegungen verwenden. [Bod97b]

Definition 4.1
Eine Baumzerlegung $(\{ X_i \mid i \in I \}, T = (I, F))$ *wird als schöne Baumzerlegung bezeichnet, wenn die folgenden Bedingungen erfüllt sind:*

1. *Jeder Tasche in T hat höchstens zwei Kinder.*
2. *Wenn eine Tasche* $i \in I$ *zwei Kinder* j *und* k *hat, dann gilt* $X_i = X_j = X_k$.
3. *Wenn eine Tasche* i *ein Kind* j *hat, dann gilt entweder*
   - $|X_i| = |X_j| + 1$ *und* $X_j \subset X_i$ *oder*
   - $|X_i| = |X_j| - 1$ *und* $X_i \subset X_j$.

In einer schönen Baumzerlegung eines Graphen G lassen sich die verschiedenen Knotentypen wie folgt benennen. Die Namen beziehen sich auf das Ablaufen des Baumes von den Leaf-Knoten Richtung Wurzel, was auch zur Bottom-Up Arbeitsweise viele Algorithmen auf Baumzerlegung passt.

- **Leaf-Knoten**: Wenn eine Tasche i kein Kindknoten hat (das heißt, dass er keine Nachfolger hat), dann wird er als Leaf-Knoten bezeichnet $|X_i| = 1$.
- **Forget-Knoten**: hat i nur einen Nachfolger j und gilt $X_i = X_j \setminus \{v\}$ für $v \in X_j$ dann wird i als Forget-Knoten bezeichnet.
- **Introduce-Knoten**: hat i nur einen Nachfolger j und gilt $X_i = X_j \cup \{v\}$ für $v \notin X_j$, dann wird i als Introduce-Knoten benannt.
- **Join-Knoten**: Wenn i zwei Nachfolger j_1 und j_2 hat und gilt $X_i = X_{j_1} = X_{j_2}$, dann nennen wir ihn als Join-Knoten.

Lemma 4.1
Gegeben sei ein Graph G und eine Baumzerlegung von G, die eine Baumweite k und $O(n)$ Knoten hat, wobei n die Anzahl der Knoten von G ist. Dann können wir eine schöne Baumzerlegung von G finden, die ebenfalls eine Baumweite k und $O(n)$ Knoten hat, in linear Zeit O(n).

4.1 Berechnung der schönen Baumzerlegung

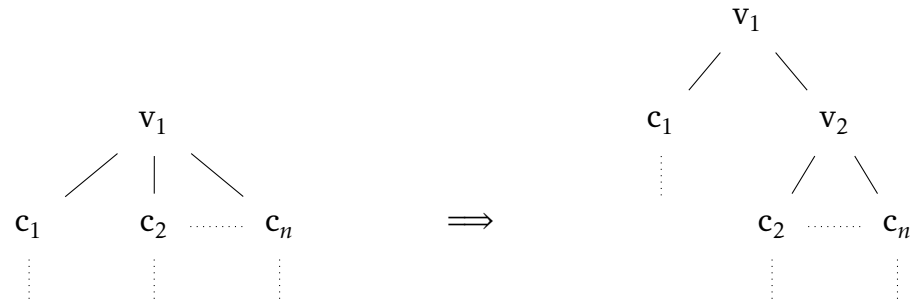
Um die schöne Baumzerlegung in linearer Zeit zu berechnen, nutzen wir den folgenden Algorithmus [Klo94]. Als Eingabe erhält der Algorithmus die Baumzerlegung und gibt als Ergebnis eine schöne Baumzerlegung mit derselben Baumweite zurück.

Algorithmus 3 Schöne Baumzerlegung

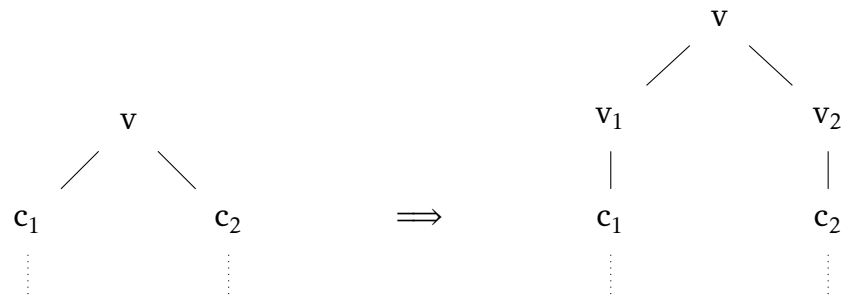
Eingabe: Baumzerlegung mit der Breite k.
Ausgabe: Schöne Baumzerlegung der Breite k.

1. Wähle einen beliebigen Knoten als *Wurzel*.
2. Wenn Knoten v_1 mehr als 2 Kinder hat $\{c_1,...,c_n\}$:
   (a) Kopie v_2 von v_1 erstellen.
   (b) Setze v_2 als Kind von v_1 zusammen mit c_1.
   (c) Setze $\{c_2, ..., c_n\}$ als Kinder von v_2.
   (d) Wiederhole , bis keine Knoten, in der gesamten Zerlegung, mehr als zwei Nachfolger hat.
3. Für Knoten v mit 2 Kinder $\{c_1, c_2\}$:
   (a) Kopien erstellen von v: v_1 und v_2.
   (b) Setze v_1, v_2 als Kinder von v.
   (c) Setze c_x als Kind von v_x.
4. Für Knoten mit einem Kind, erstelle eine Reihe von Introduce und Forget-Knoten zwischen ihnen und ihrem Kind.
5. Für Leaf-Knoten, erstelle eine Reihe von Introduce-Knoten, so dass der Leaf-Knoten eine Größe von 1 hat.

Abbildung 10 veranschaulicht die Schritte 2 und 3 des Algorithmus.



(a) Schritt 2



(b) Schritt 3

Abbildung 10: Veranschaulichung von Schritt 2 und 3 im Algorithmus 3

Beispiel 4.1
Wir betrachen den Graph G aus der folgenden Darstellung.

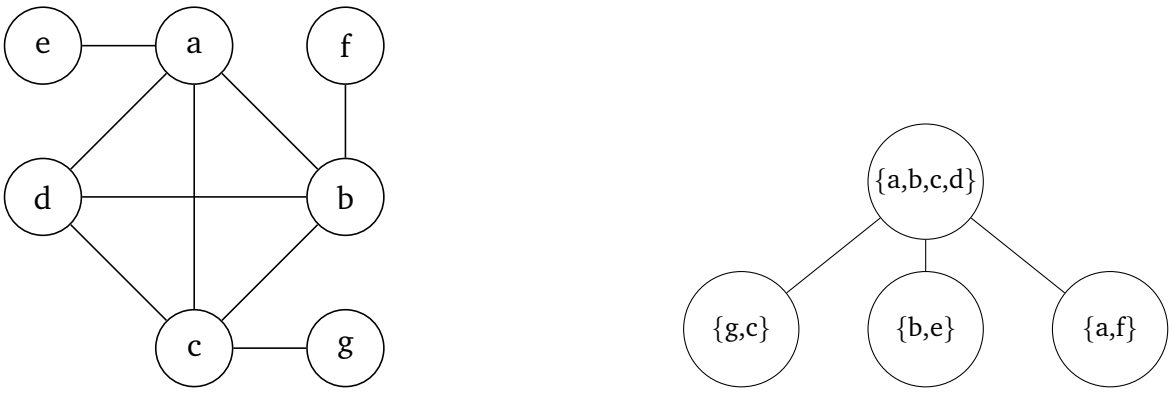


Abbildung 11: Graph G und seiner Baumzerlegung

Um den gegebenen Graph in eine baumartige Struktur umzuwandeln, verwenden wir die Schritte, die im Algorithmus 2 beschrieben sind. Zunächst sortieren wir die Knoten in G nach ihrem Grad, d.h. $N = \{e, f, g, d, a, b, c\}$. Offensichtlich haben die Knoten $\{e, f, g\}$ hier die höchste Priorität, da sie keine zusätzlichen Kanten benötigen, um ihre Nachbarn zu verbinden. Wir wählen daher zuerst den Knoten e aus, fügen ihn und seine Nachbarschaft zu einer Tasche hinzu und entfernen dann e und die entsprechenden Kanten aus G. Die gleichen Schritte werden für die Knoten f und g durchgeführt.

Für den Knoten d stellen wir fest, dass alle Nachbarn bereits miteinander verbunden sind und einen vollständigen Graphen bilden. Daher fügen wir alle diese Knoten zu einer Tasche hinzu, und somit sind wir fertig. Die Baumzerlegung ist in Abbildung 11 rechts gegeben.

Die Konstruktion einer schönen Baumzerlegung ist recht einfach und intuitiv. Alles, was wir tun müssen, ist, den Schritten zu folgen, die im Algorithmus 3 beschrieben sind.

Zu Beginn wählen wir eine Wurzel für den Baum. In diesem speziellen Fall, um sämtliche Fälle des Algorithmus abzudecken, entscheiden wir uns für Tasche $\{a, b, c, d\}$ als Wurzel.

Anschließend betrachten wir die Anzahl der Nachfolger des Wurzelknotens. Wir stellen fest, dass es mehr als zwei Nachfolger gibt, dann wenden wir den zweiten Schritt an: Wir erstellen eine Kopie v_2 des Wurzelknotens $\{a, b, c, d\}$ und setzen v_2 als Kindknoten des Wurzelknotens ein. Die ursprüngliche linke Tasche $\{g, c\}$ bleibt als Kind des Wurzelknotens erhalten, während die beiden anderen Knoten $\{b, e\}$ und $\{a, f\}$ nun Kinder der Kopie v_2 sind.

Da der Wurzelknoten $\{a, b, c, d\}$ jetzt nur noch zwei Nachfolger hat, wenden wir den dritten Schritt des Algorithmus an. Allerdings ist einer der beiden Nachfolger bereits eine Kopie der Wurzel. Daher erstellen wir nur eine Kopie und setzen den Knoten $\{g, c\}$als Nachfolger dieser neuen Kopie ein.

Im linken Teil des Baums hat die Kopie $\{a, b, c, d\}$ nun einen Nachfolger $\{g, c\}$. Dies führt uns zum vierten Schritt des Algorithmus. Hierbei fügen wir eine Serie von Introduce-Knoten und Forget-Knoten zwischen der Kopie und ihrem Nachfolger ein.
Nun hat der Nachfolger der Kopie im linken Teil keine Kindknoten mehr. Daher wenden wir den fünften Schritt des Algorithmus an und führen eine Reihe von Introduce-Knoten ein, bis die Größe der Knoten 1 ist.

Im rechten Teil des Baums hat die Kopie $\{a, b, c, d\}$ zwei Nachfolger. Gemäß dem dritten Schritt erstellen wir zwei Kopien von $\{a, b, c, d\}$ und setzen diese Kopien als Kindknoten für die entsprechenden Nachfolger ein. Jetzt hat jede Kopie eine einzige Nachfolgerin. Daher wenden wir ebenfalls den vierten Schritt des Algorithmus an. Sobald die Nachfolgerinnen keine Kindknoten mehr haben, setzen wir den fünften Schritt fort, bis die Größe der Knoten 1 erreicht.

Die schöne Baumzerlegung wird in Abbildung 12 gezeigt.

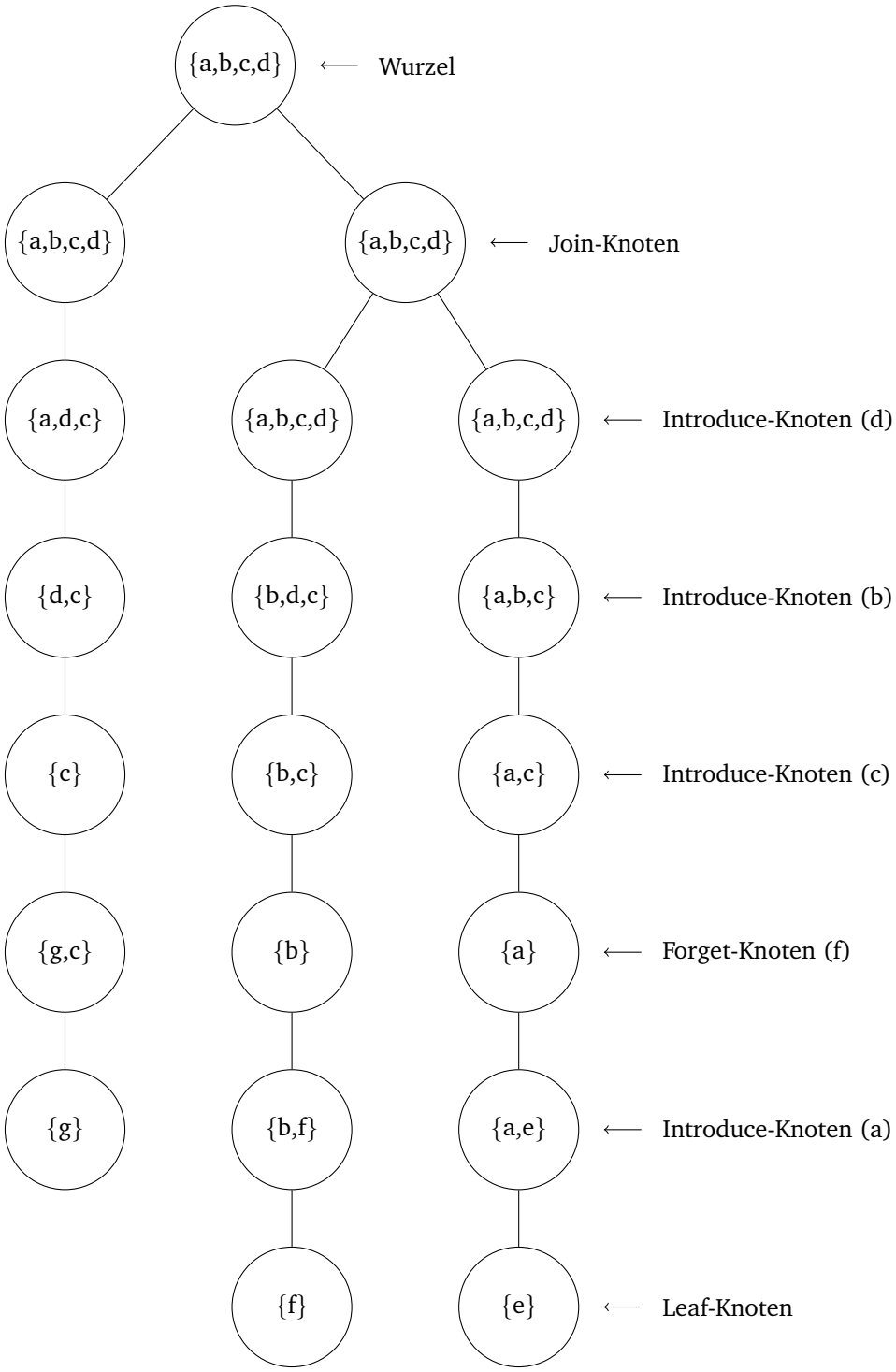


Abbildung 12: Schöne Baumzerlegung für Graph G in Abbildung 11

5 Graphenprobleme auf Bäumen mit beschränkter Baumweite

Nachdem wir in den Kapiteln 3 und 4 die erforderlichen Grundlagen eingeführt haben, sind wir nun in der Lage, Probleme auf Graphen mit beschränkter Baumweite zu lösen.
Für eine größere Anzahl von Graphenprobleme wurde gezeigt, dass sie in linearer oder polynomialer Zeit gelöst werden können, vorausgesetzt, die Eingabe besteht aus Graphen mit einer Baumweite von höchstens k. [BK08].
In der Welt der Komplexitätstheorie gibt es zwei wichtige Klassen von Problemen: Die P-Klasse, die Probleme umfasst, die von deterministischen Turing-Maschinen in polynomialer Zeit gelöst werden können, und die NP-Klasse, die Probleme enthält, bei denen Lösungen in polynomialer Zeit verifiziert werden können. Das bedeutet, wenn uns eine mögliche Lösung gegeben ist, können wir in polynomialer Zeit überprüfen, ob sie korrekt ist. Ein charakteristisches Merkmal von NP ist, dass es keine bekannten Algorithmen gibt, die in polynomialer Zeit alle Instanzen dieser Probleme lösen können. Dies ist der Kern des P vs. NP-Problems.
Viele Graphenprobleme, wie die Färbung von Graphen und viele andere, gehören zur Klasse der NP-vollständigen Probleme, was bedeutet, dass sie in NP liegen und mindestens so schwierig als jedes andere NP-Problem. Wenn man annimmt, dass P ungleich NP ist, dann sind diese Probleme nicht in P lösbar, und es fehlen geeignete Algorithmen, die für größere Eingabegrößen effizient arbeiten können. Allerdings trifft dies nicht zu, wenn der Graph eine beschränkte Baumweite aufweist. In diesem Fall können diese Probleme in polynomialer Zeit gelöst werden.
Der Zusammenhang zwischen der P vs. NP-Komplexität und der Baumzerlegung liegt darin, dass einige NP-vollständige und NP-schwere Probleme, wie das Färbbarkeitsproblem und maximale unabhängige Menge, in Form von Graphenproblemen formuliert werden können. Wenn es möglich ist, die Baumweite solcher Graphen zu beschränken, indem effiziente Baumzerlegung gefunden werden, kann dies dazu beitragen, diese NP Probleme auf spezielle Fälle zu begrenzen. Dies eröffnet die Möglichkeit, diese Probleme für bestimmten Baumweiten in polynomieller Zeit zu lösen.

In diesem Abschnitt werden wir die Baumzerlegung nutzen, um zwei Graphprobleme effizient zu lösen, nämlich die Färbbarkeit und das Problem der maximalen unabhängige Menge. Bevor wir jedoch dazu kommen, werden wir ein wichtiges Konzept vorstellen, das auf viele Graphprobleme angewendet werden kann.

5.1 Dynamische Programmierung

Wir möchten nun eine Methode zur Lösung eines Problems P auf einem Graph G mit beschränkter Baumweite betrachten. Wenn G eine relativ kleine Baumweite k hat, hilft diese Methode, P effizient zu lösen.
Das Konzept der Dynamische Programmierung ist, Probleme zu lösen, indem Lösungen aus geschachtelten Teilaufgaben auf eine Weise kombiniert werden, die si-

cherstellt, dass keine Teilaufgabe doppelt bearbeitet wird. Es ist jedoch wichtig zu beachten, dass nicht alle NP-Probleme auf diese Weise gelöst werden können.

Eine häufig anwendbare Technik für die Lösung von Problemen auf Graphen mit begrenzter Baumweite besteht darin, *Tabellen zur Charakterisierung von Teillösungen* in einer *Bottom-up* Reihenfolge, für jede Tasche $i \in I$ in der Zerlegung, zu berechnen. Bei diesem Prinzip der Dynammischen Programmierung haben die Algorithmen folgende Struktur [Bod97b]:

Algorithmus 4 Dynamische Programmierung

1. Finde eine Baumzerlegung von G mit einer Breite von höchstens k.
2. Diese Baumzerlegung wird verwendet, um eine schöne Baumzerlegung zu konstruieren, deren Breite ebenfalls durch k begrenzt ist.
3. T wird in Bottom-up Reihenfolge durchlaufen, d.h. von den Blättern bis zu einer festgelegten Wurzel. Berechne für jeden Knoten(Tasche) $i \in I$ eine bestimmte Tabelle mit Teillösungen für die akuelle Tasche. Um eine Tabelle für einen Knoten i zu erstellen, werden ausschließlich die bereits für die Kinder von i berechneten Tabellen, und die Informationen über G verwendet, die auf X_i beschränkt sind. Abhängig von der Art des Knotens i wird ein bestimmter Berechnungsverfahren verwendet.
4. Die für die Wurzel berechnete Teillösung liefert dann eine Antwort auf das übergeordnete Problem, indem man die Tabelle der Wurzelknoten untersucht.

Wir werden den Algorithmus nutzen, um zwei Graphprobleme, nämlich die *Färbbarkeit und die maximale unabhängige Menge*, exemplarisch zu untersuchen und zu verdeutlichen, wie sich eine gegebene Baumzerlegung in diesem Zusammenhang einsetzen lässt.

5.2 Färbbarkeitsproblem

Definition 5.1
Eine r-Färbung eines Graphen ist eine Zuordnung von Knoten zu r Farben, so dass keine zwei benachbarte Knoten gleichfarbig sind. Genauer gesagt, sei $G = (V, E)$ ein Graph. Eine r-Färbung ist eine Funktion $f : V \to \{1, \ldots, r\}$, so dass $f(u) \neq f(v)$ für jede Kante $(u, v) \in E$.
Ein Graph G heißt r-färbbar, wenn es eine r-Färbung für G gibt.
Für $r = 2$ gibt es einen Linearen Algorithmus, der das Problem löst. Das r-Färbbarkeitsproblem gehört zu den NP-vollständigen Problemen, wenn $r \geq 3$. [Mou14]

5.2.1 Algorithmus zur Lösung des r-Färbbarkeitsproblems

Wir entwickeln nun einen dynamischen Programmieralgorithmus zur Lösung des r-Färbbarkeitsproblem auf Graphen mit beschränkter Baumweite. Die grundlegende Idee des Algorithmus besteht darin, dass wir jeder Knoten i in der schönen Baumzerlegung überprüfen, um herauszufinden, ob es eine Lösung für das Problem gibt, die sich auf G_i beschränkt. G_i ist der Teilgraph von G, der aus den Knoten in der

Tasche X_i der schönen Baumzerlegung besteht.
Gleichzeitig speichern wir ausreichende Informationen über diese Lösungen, um sie später für die übergeordneten Knoten verwenden zu können. Wir iterieren dann von den Kindknoten zu den Elternknoten, um die Lösungen schrittweise zu erweitern.

Wir verwenden den Algorithmus 4 aus dem vorherigen Abschnitt und erläutern, wie jeder Tabelleneintrag für jeden Knoten in der schönen Baumzerlegung berechnet werden kann. Diese Tabelleneinträge repräsentieren Mengen von möglichen Färbungen für die Knoten in der jeweiligen Tasche X_i und werden als
$c : X_i \to \{1, \ldots, r\}$ definiert.

Algorithmus 5 Berechnung der Tabellen für Färbbarkeitsproblem

- Für Leaf-Knoten $i \in I$ gibt es nur einen einzelnen Knoten $X_i = \{v\}$ zu berücksichtigen. Daher erstelle eine Tabelle mit r Einträgen, jeweils für jede Farbe, die wir dem einzigen Knoten in i geben können.
- Für Introduce-Knoten $i \in I$ und seine Kindknoten j bezeichnen wir mit v der neu eingeführten Knoten in i, d.h. $\{v\} = X_i \setminus X_j$. Für den Knoten v wir müssen lediglich alle Lösungen für den Kindknoten j durchgehen und für jede davon überprüfen, welche Erweiterungen gültig sind, indem die Farben der benachbarten Knoten für jede mögliche Erweiterung überprüft werden müssen.
- Für Forget-Knoten $i \in I$ und seine Kindknoten j bezeichnen wir die Forget-Knotensenen Knoten in i als v, d.h. $\{v\} = X_j \setminus X_i$. In diesem Fall lösche den Forget-Knoten aus jedem Eintrag in der Tabelle der Kindknoten.
- Für Join-Knoten $i \in I$ bezeichnen wir mit j und k seine beiden Kindknoten. Wie im Abschnitt zur schönen Baumzerlegung erklärt wurde, sind beide Kindknoten identisch, dann behalte nur die Einträge, die auch in beiden Tabellen der Kindknoten von i enthalten sind, d.h. $f(i) = f(j) \cap f(k)$.

Mithilfe dieses Verfahren kann die Tabelle für den Wurzelknoten in bottom-up Reihenfolge berechnt werden. Dabei benötigt das Verfahren höchstens $O(r^k)$ für Tabellen mit höchstens r^k Einträgen, da jede Tasche, mit maximal k Knoten, in der schönen Baumzerlegung, r mögliche Farben hat, d.h. $r \cdot r \cdot r \cdots r = r^k$. Eine schöne Baumzerlegung hat höchstens $(k \cdot n)$ Taschen. Daher ergibt sich eine Laufzeit von $O(r^k \cdot k \cdot n)$.

Beispiel 5.1
Um die Färbbarkeit anhand einer schönen Baumzerlegung zu illustrieren, betrachten wir den folgenden Graph, der in Abbildung 13 dargestellt ist.

Zuerst benötigen wir eine Baumzerlegung für den Graph G, die in Abbildung 13 rechts dargestellt ist. Anschließend transformieren die Baumzerlegung in eine schöne Baumzerlegung, wie in Abbildung 14.
Die Farbnamen werden abgekürzt: r steht für Rot, b für Blau und g für Grün.

Nun können wir mit der Berechnung der Tabellen beginnen. Wir starten mit den Leaf-Knoten a. Da a Leaf-Knoten ist, kann er eine beliebige Farbe(Rot, Grün oder Blau)

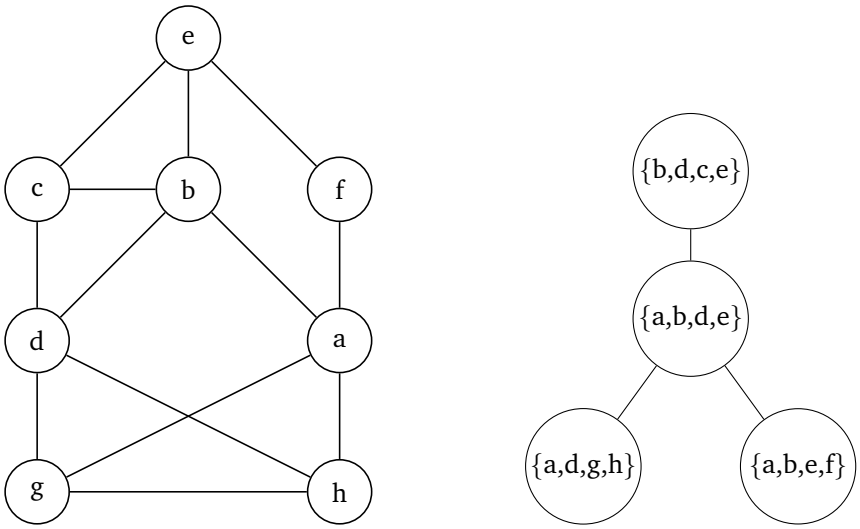


Abbildung 13: Graph G und eine gültige Baumzerlegung

erhalten. Die Tabelle für den Knoten a ist in Tabelle 1 dargestellt.

Weiter oben im Baum erreichen wir den Introduce-Knoten {a, h}. Um die Tabelle für diesen Knoten zu erstellen, müssten wir alle Farbkombinationen für die Kindknoten betrachten, und untersuchen welche Erweiterungen möglich sind. Das bedeutet, wenn der Kindknoten a die Farbe Rot hat, dann sollte h, weil sie in G benachbart sind, eine andere Farbe wie Blau oder Grün haben.
Der Knoten a könnte auch Grün sein. In diesem Fall nimmt h die restlichen Farben an, nämlich Rot oder Blau. Das Gleiche gilt, wenn a Blau ist. Die Tabelle 2 verdeutlicht alle möglichen Farbkombinationen für die Knoten {a, h}.
Die Tabellen für die restlichen Introduce-Knoten können auf die gleiche Weise berechnet werden.

Als nächstes betrachten wir den Forget-Knoten {a, d, g}. In diesem Fall besteht unsere Hauptaufgabe darin, die Farbmöglichkeiten für die Knoten {a, d, g} aus der Tabelle der Kindknoten zu extrahieren. Bei genauer Untersuchung der Farbkombinationen für den Kindknoten {a, d, g, h} in Abbildung 14 stellen wir fest, dass die Knoten {a, d, g} folgende Kombinationen haben kann, (Rot, Rot, Grün), (Rot, Rot, Blau), (Grün,Grün, Blau), (Blau, Blau, Grün) oder (Blau, Blau, Grün).

Knoten	Farbkombination
{a}	Rot, Grün oder Blau

Tabelle 1: Farben für den Leaf-Knoten {a}

Knoten	Farbkombinationen
{a ,h}	(Rot, Grün) (Rot, Blau) (Grün, Blau) (Grün, Rot) (Blau, Rot) (Blau, Grün)

Tabelle 2: Farbkombinationen für die Knoten {a, h}, d ist der Introduce-Knoten

Besonders interessant wird es bei den Join-Knoten, bei denen wir nur die Einträge be-

halten, die in beiden Kindknoten vorkommen. Abbildung 14 zeigt, dass die Join-Knoten $\{a, b, d, e\}$, nur Einträge aufweisen, die in beiden Kindknoten gemeinsam auftreten.

Wenn wir feststellen, dass die Knoten in einer Tasche nicht mit den vorgegebenen Farben eingefärbt werden können, brechen wir den Vorgang ab, da dies darauf hinweist, dass der gesamte Graph mit der gegebenen Farben nicht färbbar ist.

Auf diese Weise können alle Tabellen für die Knoten (Taschen) in der schönen Baumzerlegung berechnet werden. In Abbildung 14 sind alle möglichen Farbkombinationen für jeden Knoten dargestellt.

Wie im Algorithmus 4 beschrieben, wenn die Tabelle für die Wurzelknoten nicht leer ist, wissen wir, dass es mindestens eine gültige r-Färbung der Knoten von G gibt.

Zusätzlich können wir eine Top-Down-Strategie auf unseren Baum in Abbildung 14 anwenden, um mögliche Farben für die Knoten auszuwählen. Der Wurzelknoten bietet verschiedene Möglichkeiten, aus denen wir eine auswählen können, zum Beispiel (gbrb). Dies impliziert, dass $b \rightarrow$ Grün, $d \rightarrow$ Blau, $c \rightarrow$ Rot und $e \rightarrow$ Blau assoziiert sind.
Wir setzen unsere Betrachtung im rechten Teilbaum fort. Die Tasche $\{b, d, e\}$ umfasst drei Knoten, von denen bereits alle gefärbt sind.
In der unteren Ebene des Baumes finden wir der Knoten $\{a, b, d, e\}$. Alle drei Knoten $\{b, d, e\}$ sind bereits in (Grün, Blau, Blau) gefärbt. Dies bedeutet, dass wir in der Tabelle des Knotens $\{a, b, d, e\}$ nach Farbkombinationen suchen, in denen die Knoten $\{b, d, e\}$ die Kombination (gbb) haben. In diesem Fall finden wir nur eine Kombination, nämlich (bgbb), wir wählen sie aus.
Da der Knoten $\{a, b, d, e\}$ Join-Knoten war, wählen wir von seinen beiden Kindknoten dasselbe aus, was wir bei ihm gewählt haben.
Bei der Knoten $\{a, b, d\}$ und $\{a, d\}$ sind bereits alle enthaltenen Knoten gefärbt.
Bei dem Knoten $\{a, d, g\}$ suchen wir nach einer Kombination, bei der sowohl die Knoten a als auch d in Blau gefärbt sind. Es gibt nur eine mögliche Lösung (bbg): In diesem Fall muss g in Grün gefärbt werden.

Auf die gleiche Weise können alle verbleibenden Knoten im Baum eingefärbt werden. Abbildung 15 zeigt den gefärbten Graph.

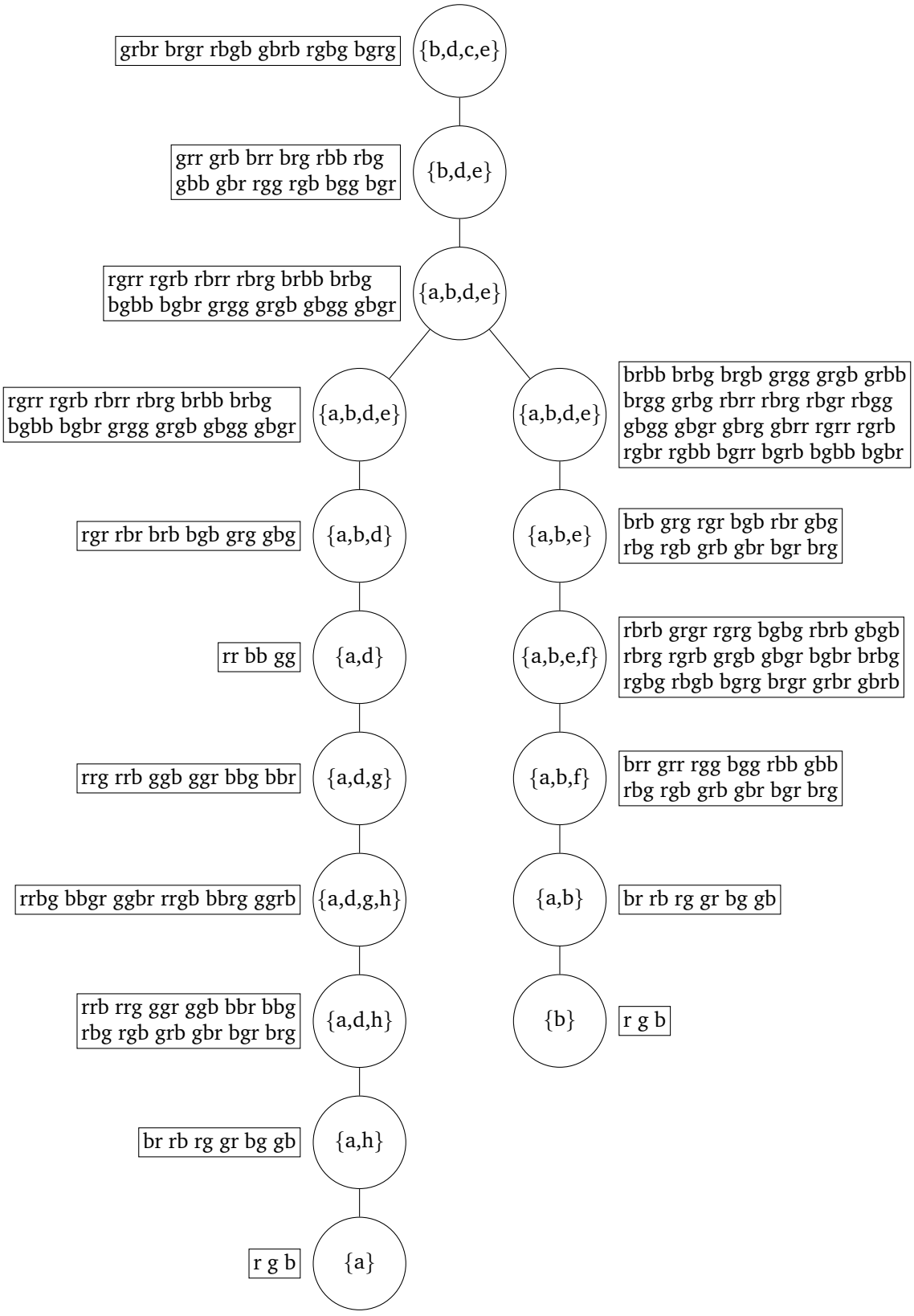


Abbildung 14: Schöne Baumzerlegung für Graph G in Abbildung 13 mit Tabellen der Färbbarkeitsproblem

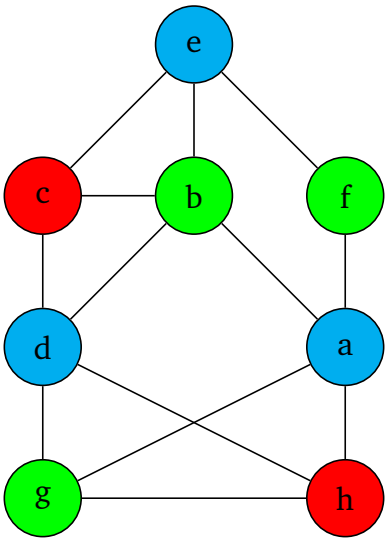


Abbildung 15: Gefärbte Graph G

5.2.2 Alternative Verfahren für das k-Färbbarkeitsproblem

In diesem Abschnitt werden wir uns mit dem Backtracking-Algorithmus und seiner Anwendung zur Lösung des Färbbarkeitsproblems auseinandersetzen.

Definition 5.2
Ein Backtracking-Algorithmus ist eine flexible Methode, welche auf viele Probleme angewandet werden kann. Er funktioniert, indem er nacheinander alle möglichen Lösungen eines Optimierungsproblems erzeugt. Der Algorithmus terminiert, sobald eine Lösung gefunden wurde oder, falls es keine Lösung gibt, nachdem alle Lösungsmöglichkeiten ausprobiert wurden. Wenn eine Lösungsmöglichkeit in eine Sackgasse führt, beispielsweise wenn in einer Teil-Lösung Widersprüche auftreten, wie einen falsch gefärbten Knoten, wird der letzte Schritt rückgängig gemacht (backtrack), und es wird versucht, das Problem mit andere Lösungsmöglichkeiten zu lösen. [HZ]

Grundsätzlich sucht der Backtracking-Algorithmus nach einer Lösung für ein Problem aus allen verfügbaren Optionen. Anfangs beginnt das Backtracking mit einer möglichen Option, und wenn das Problem mit dieser ausgewählten Option gelöst werden kann, wird die Lösung zurückgegeben. Andernfalls geht der Algorithmus zurück und versucht, das Problem mit anderen Optionen zu lösen.

Zusammenfassend lässt sich feststellen, dass *der Begriff Backtracking impliziert, dass bei einer ungeeigneten aktuellen Lösung diese ausgeschlossen und zurückverfolgt wird, um nach anderen Lösungen zu suchen*.

Backtracking und das k-Färbbarkeitsproblem

Eingegeben ist ein Graph $G = (V, E)$ und eine Anzahl von Farben r. Der Algorithmus hat die folgenden Schritte:

Algorithmus 6 Färbung mit Backtracking

1. Numeriere /sortiere die Knoten durch, beginnend mit Knoten 1.
2. In jedem Schritt wird versucht, einen neuen Knoten einzufärben, indem ihm eine Farbe zugewiesen wird, die sich von den Farben seiner Nachbarknoten unterscheidet.
3. Wenn die Knoten $v_1, v_2, \ldots, v_{i-1}$ bereits gefärbt sind, wird Knoten v_i mit der ersten verfügbaren Farbe eingefärbt.
4. Überprüfe, ob diese Färbung von Knoten v_i gültig ist.
5. Wenn die Färbung gültig ist, wird mit dem nächsten Knoten auf dieselbe Weise fortgefahren.
6. Falls nicht, wird Knoten v_i mit der nächsten Farbe eingefärbt, und die Überprüfung wird wiederholt.
7. Wenn keine der r verfügbaren Farben zu einer gültigen Färbung führt, wird ein Rückwärtsschritt durchgeführt, indem die Farbe des vorherigen Knotens v_{i-1} geändert wird. Anschließend wird der Algorithmus fortgesetzt.

Es gibt insgesamt r^n, für $(n = |V|)$, Kombinationen von Farben. Somit ist die Zeitkomplexität des Backtracking $O(r^n)$, wobei r die Anzahl der Farben ist.
Backtracking ist ein mächtiger Werkzeug, um alle möglichen Lösungen oder die optimale Lösung für Probleme mit mehreren Lösungen zu finden. Jedoch aufgrund seiner exponentiellen Zeit Komplexität kann Backtracking in großen Lösungsräumen zeitaufwändig sein, da es eine beträchtliche Anzahl von Möglichkeiten erkunden muss, bevor es eine Lösung findet. Dies kann unpraktisch sein, um große oder komplexe Probleme effizient zu lösen.

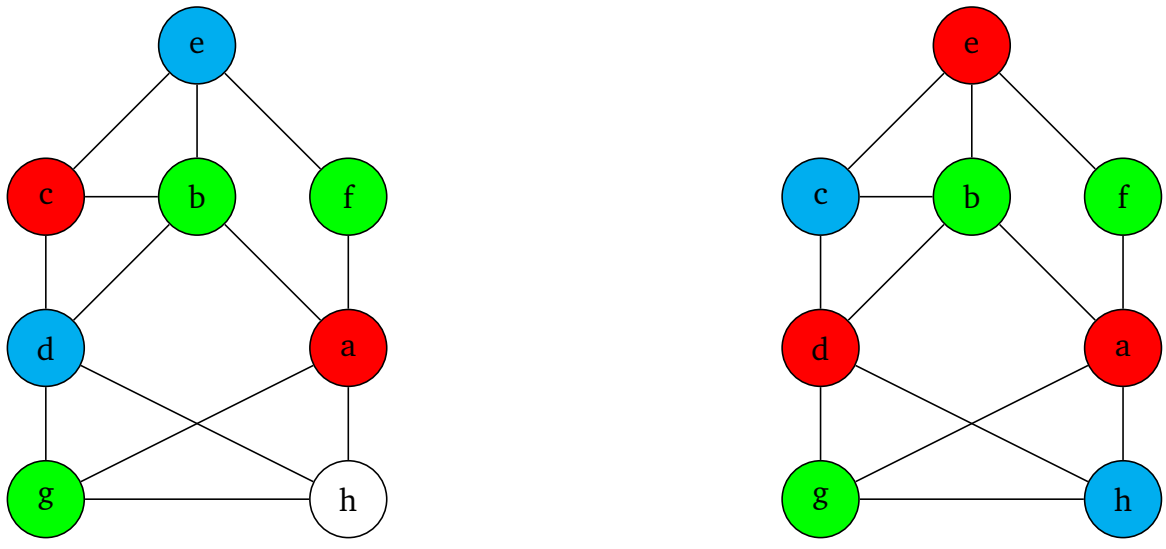
Im Vergleich zum zuvor vorgestellten Verfahren mit einer Zeitkomplexität von $O(r^k \cdot k \cdot n)$, wobei k die Baumweite des Graphen ist, zeigt sich, dass die Komplexität bei geringerer Baumweite in Bezug auf die Größe des Graphen nahezu linear verläuft. Das bedeutet, dass die Zeitkomplexität lediglich in $O(n)$ bleibt, insbesondere wenn die Baumweite k konstant niedrig ist. In solchen Fällen steigt die Anzahl der Berechnungsschritte nur minimal an. Dies stellt einen erheblichen Vorteil dar, insbesondere bei der Anwendung auf Probleme mit geringer Baumweite, da die Laufzeit des Algorithmus in solchen Situationen gut vorhersehbar und effizient ist

Im Gegensatz dazu kann die Zeitkomplexität von Backtracking Algorithmen in vielen Fällen exponentiell mit der Anzahl der Knoten im Graph ansteigen. Dies geschieht insbesondere dann, wenn alle möglichen Kombinationen von Knoten im Graph durchprobiert werden müssen. In solchen Fällen kann die Laufzeit erheblich ansteigen, da die Anzahl der möglichen Kombinationen exponentiell mit der Anzahl der Knoten im Graph wächst.

Beispiel 5.2
Wir betrachten nun den Graph G, der in Abbildung 13 dargestellt ist, und lösen ihn mithilfe des Backtracking-Algorithmus.
Zunächst sortieren wir die Knoten alphabetisch $\{a, b, c, d, e, f, g, h\}$. Wir vergeben an Knoten a die erste verfügbare Farbe, in diesem Fall Rot. Dann gehen wir zum nächsten

Knoten b. Da b mit a benachbart ist, muss er eine neue Farbe erhalten, nämlich Grün. Der Knoten c ist nur mit b in Graph G benachbart, daher erhält c ebenfalls die erste verfügbare Farbe, Rot. Weiter mit dem Knoten d, der mit b und c benachbart ist, erhält die einzige verfügbare Farbe Blau. Analog wird e zunächst Blau, f Grün und g Grün gefärbt. Am Ende, beim letzten Knoten h, gibt es keine Möglichkeit, h mit eine Farbe zu färben, und darum kann die bisherige Teillösung für Knoten $a - g$ nicht zu einer 3-Färbung von Graph G vervollständigt werden.
In solchen Fällen würde der Backtracking-Algorithmus einen oder mehrere Schritte rückgängig machen (Backtrack) und versucht, den Graph mit anderen Farbkombinationen zu färben. Dies verdeutlicht einen der Nachteile des Backtracking Algorithmus, nämlich dass er nicht immer die richtige Lösung effizient findet und alternative Lösungen für die Knoten in Graph G ausprobieren muss.

Nun ist der Algorithmus gezwungen, einen Schritt zurückzugehen. Er versucht, den Knoten g mit einer anderen Farbe zu färben. In diesem Fall ist dies jedoch nicht möglich, da g sowohl mit a als auch mit d benachbart ist und keine andere Farbe vorhanden ist. Ein weiterer Schritt zurück führt zu dem Versuch, die Farbe des Knotens f zu ändern, was ebenfalls scheitert, da die Farben von a und e nicht geändert werden können, da sie vor f in der sortierten Reihenfolge auftreten. Der Algorithmus muss erneut einen Schritt zurückgehen und versucht nun, die Farbe der Knoten e zu ändern, was sich jedoch ebenfalls als nicht möglich erweist. Gleiches gilt für den Knoten d. Schließlich, wenn der Algorithmus zu Knoten c gelangt, kann er die Farbe von c ändern und folglich auch die Farben der benachbarten Knoten. Da c mit b verbunden ist, bleibt die einzige mögliche Farboption für c Blau, während e und d mit Rot gefärbt werden. Analog erhalt f Grün, g auch Grün und schließlich h Blau.



(a) Fehlgeschlagene Farbversuch. **(b)** Erfolgreiche Farbversuch.

Abbildung 16: Färbung mit dem Backtracking-Algorithmus

5.3 Maximale unabhängige Menge

Definition 5.3
Gegeben sei ein ungerichteter Graph $G = (V, E)$. Eine maximale unabhängige Menge (MUM) ist eine Teilmenge von Knoten V in G, bei der keine zwei Knoten benachbart sind und keine weiteren Knoten aus V hinzugefügt werden können, ohne die Unabhängigkeitseigenschaft zu verletzen.

5.3.1 Algorithmus zur Berechnung von maximalen unabhängigen Mengen

Wie beim Färbbarkeitsproblem verwenden wir den dynamischen Programmieralgorithmus 4 zur Lösung des Problems der maximalen unabhängigen Menge. Die Berechnung der Tabellen erfolgt zwar in ähnlicher Weise wie beim Färbbarkeitsproblem, jedoch gibt es Unterschiede im Detail. Die grundlegende Idee bleibt jedoch identisch: Der Algorithmus berechnet für jede Tasche in der schönen Baumzerlegung, und zwar in Bottom-Up Reihenfolge, eine Tabelle und überprüft anschließend, ob die berechneten Tabellen eine Lösung für das gesuchte Problem bieten können

Gegeben sei eine schöne Baumzerlegung $(\{X_i \mid i \in I\}, T = (I, F)))$ mit n Taschen. Für jede Tasche $i \in I$ wird eine Tabelle berechnet, die als C_i bezeichnet wird. C_i enthält einen ganzzahligen Wert für jede Teilmenge $S \subseteq X_i$. Daher enthält jede Tabelle C_i, wenn die Weite der Baumzerlegung höchstens k ist, höchstens 2^{k+1} Werte. Jeder dieser Werte $C_i(S)$ für $S \subseteq X_i$ entspricht dem maximalen unabhängigen Menge in G_i, wobei G_i ist der Teilgraph von G, der aus den Knoten in der Tasche X_i der schönen Baumzerlegung besteht.
Die Tabellen können wie folgt berechnet werden: [BK08]

Algorithmus 7 Berechnung der Tabellen für maximale Unabhängige Menge

- Für Leaf-Knoten i mit $X_i = \{v\}$:
  - $C_1(\emptyset) = 0 \; und \; C_i(\{v\}) = 1$
- Für Forget-Knoten i mit Kindknoten j und $X_i = X_j \setminus \{v\}$ und $S \subseteq X_i$:
  $-C_i(S) = max\{C_j(S), C_j(S \cup \{v\})\}$
- Für Introduce-Knoten mit Kindknoten j und $X_i = X_j \cup \{v\}$ und $S \subseteq X_j$:
  - $C_i(S) = C_j(S)$
  - $$C_i(S \cup \{v\}) = \begin{cases} -\infty, & wenn \; \exists \; w \in S : \{v, w\} \in E, \\ C_j(s) + 1, & sonst \end{cases}$$
- Für Join-Knoten i mit Kindknoten j_1 und j_2 und $X_i = X_{j_1} = X_{j_2}$ und $S \subseteq X_i$:
  - $C_i(S) = C_{j_1}(S) = C_{j_2}(S) - |S|$

Für jeden Knoten i ist die Zeit, die wir für die Berechnung der Tabelle für Knoten i benötigen, durch $O(2^{|X_i|})$ beschränkt. Für die Wurzel verwenden wir zusätzlich eine Zeit von $O(2^{|X_r|})$, um die Antwort auf das Problem zu finden. Da wir annehmen, dass wir eine schöne Baumzerlegung mit einer Weite von höchstens k und mit n Taschen haben, ist die Gesamtzeit durch $O(2^k \cdot n)$ beschränkt.

Beispiel 5.3
Wir betrachten nun den folgenden Graph G, der in Abbildung 17 dargestellt ist. Für den dynamischen Algorithmus 4 benötigen wir zu Beginn eine gültige Baumzerlegung, die rechts in Abbildung 17 gezeigt wird. Aus dieser Baumzerlegung leiten wir dann die schöne Baumzerlegung ab, wie in Abbildung 18 dargestellt.
Um die Knoten voneinander unterscheiden zu können, nummerieren wir sie von der Wurzel bis zum Leaf-Knoten, wie in Abbildung 18 gezeigt. Dabei erhält der Wurzelknoten die Nummer 0, gefolgt von den Knoten im linken und rechten Teil des Baums.

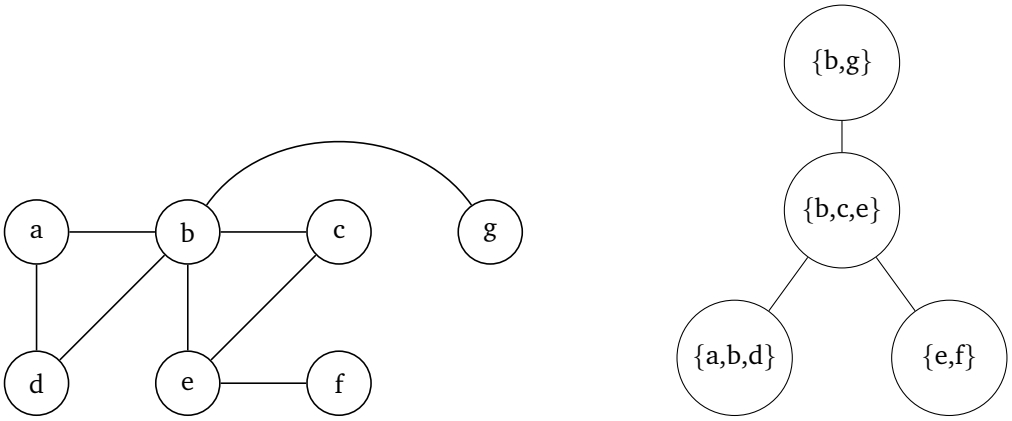


Abbildung 17: Graph G und eine gültigen Baumzerlegung

Wir beginenn mit der Berechnung. Für den Leaf-Knoten $\{d\}$ gilt folgendes:

- $C_{10}(\{\emptyset\}) = 0$
- $C_{10}(\{d\}) = 1$
- *Der Knoten d hat den Wert 1, was bedeutet, dass er der einzige Knoten in der maximalen unabhängigen Menge im Graph G_i ist. Wenn der Knoten d aus dieser Menge ausgeschlossen wird, führt dies dazu, dass die maximale Menge leer ist, und daher wird der Wert für die leere Menge als 0 festgelegt.*

Der nächsten ist der Introduce-Knoten $\{b, d\}$, dann gilt:

- *$S \subseteq X_j$, d.h. S ist eine Teilmenge der Kindknoten $\{d\}$, dann ist $S = \{\emptyset, d\}$, die leere Menge $\emptyset$ ist Teilmenge jede andere Menge. $v = \{b\}$.*
- $C_9(\emptyset) = 0$
- $C_9(\{d\}) = 1$
- $C_9(\{\emptyset \cup \{b\}\}) = C_9(\{b\}) = C_{10}(\emptyset) + C_{10}(b) = 0 + 1 = 1$
- $C_9(\{\{d\} \cup \{b\}\}) = C_9(\{d, b\}) = -\infty$, *da b und d in Graph benachbart sind.*

Weiter oben im Baum treffen wir den Introduce-Knoten $\{a, b, d\}$. Die Berechnung erfolgt ähnlich wie bei den vorherigen Knoten:

- *Die Teilmenge* $S = \{b, d, \emptyset\}$ *und* $v = \{a\}$
- $C_8(\emptyset) = 0$
- $C_8(\{b\}) = 1$
- $C_8(\{d\}) = 1$
- $C_8(\{a\}) = 1$
- $C_8(\{b, d\}) = -\infty$
- $C_8(\{a, b\}) = -\infty$
- $C_8(\{a, d\}) = -\infty$
- $C_8(\{a, b, d\}) = -\infty$

Der nächste Knoten in der Rheinfolge ist der Forget-Knoten $\{a, b\}$*, dann gilt:*

- S *ist eine Teilmenge von derselben Knoten, das heist* $S = \{a, b, \emptyset\}$ *und* $v = \{d\}$
- $C_7(\emptyset) = max\{C_8(\emptyset), C_8(\emptyset \cup \{d\})\} = max\{0, 1\} = 1$
- $C_7(\{a\}) = max\{C_8(a), C_8(\{a\} \cup \{d\})\} = max\{1, -\infty\} = 1$
- $C_7(\{b\}) = max\{C_8(b), C_8(\{b\} \cup \{d\})\} = max\{1, -\infty\} = 1$
- $C_7(\{a, b\}) = max\{C_8(\{a, b\}), C_8(\{a, b\} \cup \{d\})\} = max\{C_8(\{ab\}), C_8(\{a, b, d\})\} = max\{-\infty, -\infty\} = -\infty$

Alle anderen Tabellen können auf ähnliche Weise berechnet werden. Die Abbildung 18 zeigt für jeden Knoten, in der schönen Baumzerlegung, seine Tabelle.

Für den Join-Knoten $\{b, c, e\}$ *gilt folgendes:*

- S *ist eine Teilmenge von derselben Knoten, d.h* $S = \{b, c, e, \emptyset\}$
- $C_3(\emptyset) = C_4(\emptyset) + C_{11}(\emptyset) - 0 = 1 + 1 - 0 = 2$
- $C_3(\{b\}) = C_4(\{b\}) + C_{11}(b) - 1 = 1 + 2 - 1 = 2$
- $C_3(\{c\}) = C_3(\{c\}) + C_{11}(\{c\}) - 1 = 2 + 2 - 1 = 3$
- $C_3(\{e\}) = C_4(\{e\}) + C_{11}(\{e\}) - 1 = 1 + 1 - 1 = 1$
- $C_3(\{b, c\}) = C_3(\{b, c\}) + C_{11}(b, c) - 2 = -\infty + (-\infty) - 2 = -\infty$
- $C_3(\{b, e\}) = C_4(\{b, e\}) + C_{11}(\{b, e\}) - 2 = -\infty + (-\infty) - 2 = -\infty$
- $C_3(\{e, c\}) = C_4(\{e, c\}) + C_{11}(\{e, c\}) - 2 = -\infty + (-\infty) - 2 = -\infty$
- $C_3(\{b, c, e\}) = C_3(\{b, c, e\}) + C_{11}(\{b, c, e\}) - 3 = -\infty + (-\infty) - 3 = -\infty$

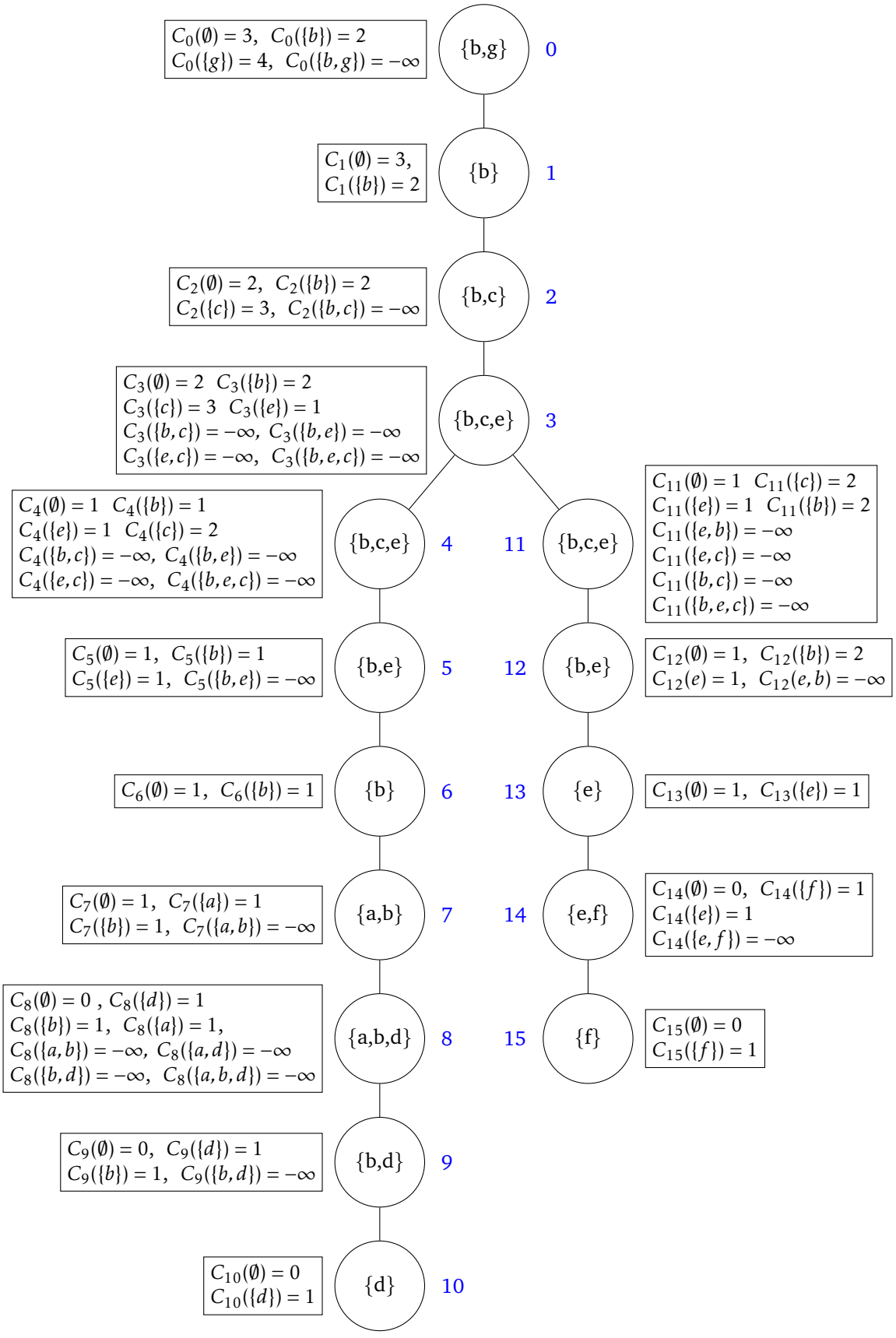


Abbildung 18: Schöne Baumzerlegung für Graph G in Abbildung 17 mit Tabellen der Max.Unabhängige Menge

Wir werden eine Top-Down-Strategie anwenden [BK08], um die Knoten der maximalen unabhängigen Menge zu bestimmen.

- *Zuerst bestimmt der Wurzelknoten gemäß Algorithmus 4 die Größe der maximalen unabhängigen Menge (MUM). Dann überprüfen wir, welcher Knoten den größten Wert hat, und stellen fest, dass Knoten g den höchsten Wert von 4 aufweist. Dies bedeutet, dass g in der MUM enthalten sein muss.*

  Anschließend prüfen wir, ob g in anderen Kombinationen enthalten ist und ob diese Kombination zur MUM hinzugefügt werden kann. Die Kombination $\{b, g\}$ hat einen Wert von $-\infty$, was bedeutet, dass Knoten b und jede andere Kombination, die b enthält, nicht zur MUM hinzugefügt werden können, solange g darin enthalten ist. Diese Information speichern wir für den nächsten Schritt. Die maximale unabhängige Menge enthält nun den Knoten $MUM = \{g\}$.

- *Weiter unten im Baum kommen wir zum Knoten b. Da wir bereits wissen, dass dieser Knoten nicht Teil der MUM sein kann, müssen wir die Tabelle für diesen Knoten nicht überprüfen.*

- *Der Knoten $\{b, c\}$ enthält die beiden Knoten b und c. In diesem Fall müssen wir nur den Knoten c überprüfen. Der Wert des Knotens c beträgt 3, was größer ist als der Wert der leeren Menge. Daher fügen wir c zur MUM hinzu. Beachten Sie, dass ein Knoten einen Wert haben muss, der mindestens so groß wie der Wert der leeren Menge ist, da er sonst nicht Teil der MUM sein kann. Wir suchen nach Kombinationen, die c enthalten, und finden die Kombination $\{b, c\}$. Da jedoch b und jede Kombination, die b enthält, nicht zur MUM hinzugefügt werden kann, müssen wir diese Kombination nicht weiter überprüfen. Wir aktualisieren die MUM auf $MUM = \{g, c\}$.*

- *Für den Join-Knoten $\{b, c, e\}$ müssen wir nur den Knoten e überprüfen, um festzustellen, ob er zur MUM hinzugefügt werden kann oder nicht. Der Knoten e hat hier den Wert 1, der kleine ist als der Wert der leeren Menge. Das bedeutet, dass e nicht Teil der maximalen unabhängigen Menge ist. Daher ist er und jede andere Kombination, die e enthält, von der maximalen unabhängigen Menge ausgeschlossen.*

- *Der Join-Knoten $\{b, c, e\}$ hat zwei identische Kindknoten, daher wählen wir denselben Knoten aus, den wir bereits beim Join-Knoten ausgewählt haben.*

- *Wir gehen nach unten in den rechten Zweig des Baums und finden den Knoten $\{b, e\}$. Wir wissen, dass weder der Knoten b noch der Knoten e zur MUM hinzugefügt werden können.*

- *Weiter unten finden wir den Knoten $\{b\}$, und wie bereits bekannt ist, kann er nicht in die MUM aufgenommen werden.*

- *Der Knoten $\{a, b\}$ wird überprüft, wobei wir nur den Knoten a betrachten. Wir stellen fest, dass dieser Knoten den gleichen Wert wie die leere Menge hat. Anschließend suchen wir nach Kombinationen, die den Knoten a und einen Knoten enthalten, der bereits in der MUM enthalten ist. Wir finden keine solche*

Kombination, daher fügen wir a zur MUM hinzu. Die aktualisierte MUM lautet: $MUM = \{g, c, a\}$.

- *Für den Knoten $\{a, b, d\}$ überprüfen wir nur den Knoten d und suchen nach Kombinationen, die d und einen anderen Knoten enthalten, der bereits in der MUM ist. Wir stellen fest, dass d einen Wert von 1 hat, der größer ist als der Wert der leeren Menge, was in Ordnung ist. Wir finden jedoch, dass die Kombination $\{a, d\}$ einen Wert von $-\infty$ hat, was bedeutet, dass wenn a bereits in der MUM enthalten ist, d nicht in die MUM eingefügt werden kann.*
- *Für die weiteren Knoten $\{b, d\}$ und $\{d\}$ ist keine weitere Untersuchung erforderlich.*
- *Die gleiche Überprüfung führen wir im rechten Zweig des Baums durch, und stellen fest, dass f zur MUM gehört.*
- *Am Ende ist $MUM = \{g, c, a, f\}$*

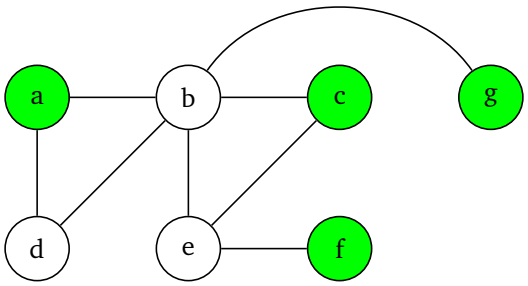


Abbildung 19: Maximale unabhängige Menge im Graph G unter Verwendung der schönen Baumzerlegung. Die Knoten in Grün repräsentieren die maximale unabhängige Menge

5.3.2 Alternative Verfahren zur Berechnung der maximalen unabhängige - Menge

In diesem Abschnitt werden wir eine alternative Methode zur Berechnung der maximalen unabhängigen Menge in einem Graphen vorstellen. Hierbei setzen wir das Backtracking Prinzip ein, um dieses Problem zu lösen.

Backtracking und die maximale Unabhängige Menge

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Unsere Aufgabe besteht darin, mithilfe des Backtracking Algorithmus die maximale unabhängige Menge zu finden. Ähnlich wie beim Färbbarkeitsproblem wenden wir das Backtracking Prinzip an, um systematisch alle möglichen unabhängigen Mengen im Graph zu berechnen und schließlich die größtmögliche davon zu ermitteln. [TT77]

Algorithmus 8 Maximale unabhängige Menge mit Backtracking

1. Starte mit zwei leeren Mengen, M und S. M enthält die maximale unabhängige Menge, die im Laufe des Algorithmus aufgebaut wird, während S eine temporäre ist, die während der Exploration von Möglichkeiten verwendet wird.
2. Wähle einen Startknoten aus dem Graph aus und füge den ausgewählten Knoten zur aktuellen Menge S hinzu.
3. Rekursiver Schritt: Suche nach einem weiteren Knoten, der zur aktuellen S hinzugefügt werden kann, ohne Konflikte zu verursachen. Dieser Schritt wird rekursiv durchgeführt, um alle möglichen Kombinationen dieser Knoten zu erkunden.
4. Backtracking: Wenn kein weiterer Knoten gibt, der zur S hinzugefügt werden kann, wird S zu M hinzugefügt, sofern die Größe von S größer größer ist als die zuvor hinzugefügten Menge in M. Anschließend geht der Algorithmus einen Schritt zurück (Backtrack), entfernt den zuletzt hinzugefügten Knoten aus S und sucht nach einer anderen Teillösung. Wenn S leer wird, geht der Algorithmus zum nächsten Schritt 5.
5. Wiederhole die Schritte 2 bis 4, bis alle möglichen Kombinationen von Knoten untersucht werden. Der Algorithmus wird somit alle Knoten im Graph überprüfen, um die maximale unabhängige Menge zu finden.
6. Der Algorithmus endet, nachdem alle möglichen Kombinationen untersucht wurden. Die in M gespeicherte Menge repräsentiert die maximale unabhängige Menge eines Graphen.

Beispiel 5.4
Wir betrachten den folgenden Graph G, der in Abbildung 20 dargestellt ist. Im vorherigen Abschnitt haben wir bereits die maximale unabhängige Menge für diesen Graph mithilfe einer schönen Baumzerlegung berechnet. Jetzt werden wir die maximale unabhängige Menge unter Verwendung des Backtracking Algorithmus berechnen.

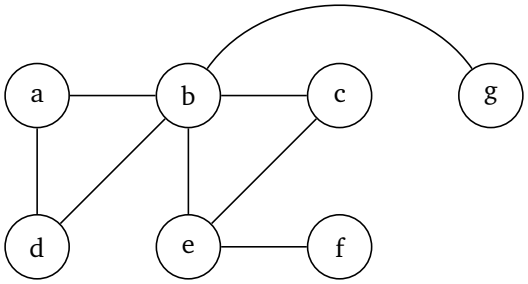


Abbildung 20: Graph G zur Berechnung maximaler unabhängiger Menge mit Backracking Algorithmus

Am Anfang sind beide Mengen M und S leer. Wir wählen einen Knoten aus dem Graph aus und fügen ihn zu S hinzu. Angenommen, der ausgewählte Knoten ist b. Da S zu Beginn leer ist, ist die Überprüfung auf Konflikte nicht erforderlich. Daher wird der ausgewählte Knoten b unmittelbar zu S hinzugefügt. Nun gilt: $S = \{b\}$.
Dann werden wir rekursiv für alle Knoten überprüfen, ob sie zu S eingefügt werden können, wenn b bereits ein Teil von S ist . In diesem Fall kann nur der Knoten f zu S hinzugefügt werden, da b mit allen anderen Knoten in Konflikt steht. Die Menge $S = \{b, f\}$.

Es gibt keine weiteren Knoten, die zu S hinzugefügt werden können. Dann fügen wir die Menge S zu M hinzu, was zu $M = \{\{b, f\}\}$ führt. Anschließend geht der Algorithmus einen Schritt zurück und entfernt den zuletzt hinzugefügten Knoten aus S. In diesem Fall wird der Knoten f aus S entfernt, und es wird überprüft, ob es weitere Möglichkeiten gibt, S zu erweitern. Da keine weiteren Möglichkeiten vorhanden sind, geht der Algorithmus einen weiteren Schritt zurück und entfernt in diesem Fall den Knoten b aus S. Dadurch wird $S = \{\}$. S ist nun leer, und wir gehen zum nächsten Schritt (5) im Algorithmus 8.

Wir wählen einen Knoten aus dem Graph G, nämlich den Knoten c, und fügen ihn zur Menge S hinzu. Da S leer ist, gibt es keine Konflikte mit anderen Knoten. Anschließend werden wir rekursiv durchlaufen, um alle möglichen Knoten zu S hinzuzufügen. Die Knoten $\{a, f, g\}$ können zu S hinzugefügt werden, so dass $S = \{c, a, f, g\}$ ist. Nachdem alle Möglichkeiten untersucht wurden, wird die Menge S mit der zuvor berechneten Menge in M verglichen. Es wird festgestellt, dass die neu berechnete Menge S größer (hat mehr Knoten) ist als die in M enthaltene Menge. Daher wird die Menge M aktualisiert, die zuvor berechnete Menge in M wird gelöscht und die neue Menge $\{c, a, f, g\}$ zu M hinzugefügt. Nun ist $M = \{\{c, a, f, g\}\}$

Dann geht der Algorithmus einen Schritt zurück und entfernt den Knoten g aus S, um andere Teilmengen zu entdecken. Nachdem g entfernt wird, besteht auch keine Möglichkeit, die Teillösung weiter auszudehnen. Ein weiterer Schritt zurück führt dazu, dass der Knoten f aus S entfernt wird. Die Menge S lautet $S = \{c, a\}$. Es ist nun möglich, die Teillösung zu erweitern,indem der Knoten g zu S hinzugefügt wird. Daraufhin wird $S = \{c, a, g\}$, und schließlich kann auch der Knoten f zu S hinzugefügt werden, so dass $S = \{c, a, g, f\}$. Keine weiteren Knoten können zu S hinzugefügt werden. Dann wird, wie zuvor beschrieben, die Menge S mit der Menge, die in M enthalten ist, verglichen. In diesem Fall enthält die neu berechnete Menge S vier Knoten, und die Menge, die in M enthalten ist, ebenfalls vier Knoten. Daher wird M nicht aktualisiert, und der Algorithmus geht einen Schritt zurück, um diese Teillösung weiter zu erweitern. Die Knoten g und f werden entfernt, und dann wird auch der Knoten a entfernt.

Die Menge $S = \{c\}$ hat nur einen Knoten. In diesem Fall kann der Knoten d zu S hinzugefügt werden, d.h. $S = \{c, d\}$. Eine weitere Möglichkeit besteht darin, die Knoten g und f hinzuzufügen, so dass $S = \{c, d, g, f\}$. Keine weiteren Knoten können zu S hinzugefügt werden. In diesem Fall wird S mit der Menge, die in M enthalten ist, verglichen. Es wird festgestellt, dass beide Mengen gleich groß sind, und daher ist keine Aktualisierung von M erforderlich.
Der Algorithmus geht einen Schritt zurück und entfernt f aus S, was zu keiner weiteren Erweiterung führt. Dann geht er noch einen weiteren Schritt zurück und entfernt g aus S, so dass $S = \{c, d\}$. In diesem Fall könnte f zu S hinzugefügt werden, so dass S zu $\{c, d, f\}$ erweitert wird. Der Knoten g kann ebenfalls zu S hinzugefügt werden, $S = \{c, d, f, g\}$. Keine weiteren Knoten können zu S hinzugefügt werden. Dann wird S erneut mit der Menge, die in M enthalten ist, verglichen, und es wird festgestellt, dass beide Mengen gleich groß sind. Daher ist auch keine Aktualisierung von M erforderlich.

Die Knoten werden schrittweise aus S entfernt, um andere Teilmengen zu finden. $\{g, f, d\}$ und c werden aus S entfernt, da keine Erweiterung mehr möglich ist. Da S leer geworden ist, kehrt der Algorithmus zum Anfang zurück und wählt einen neuen Knoten aus.

Nachdem wir alle möglichen maximalen unabhängigen Mengen in G betrachtet haben, stellen wir fest, dass $\{c, a, f, g\}$ eine maximale unabhängige Menge ist. Es ist jedoch wichtig zu beachten, dass es in Graph G auch andere maximale unabhängige Mengen gibt, wie $\{a, c, f, g\}$. Die Auswahl der Knotenfolge kann hierbei eine Rolle spielen.
Abbildung 21 zeigt die maximale unabhängige Menge der Graph G, Knoten in Grün repräsentieren die Knoten der maximalen unabhängigen Menge.

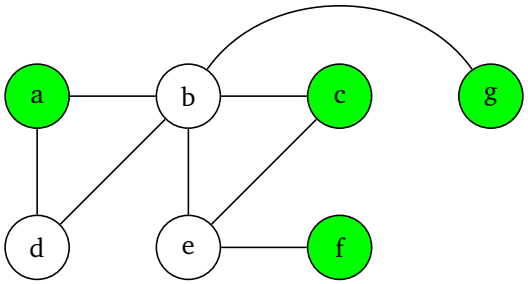


Abbildung 21: Maximale unabhängige Menge des Graphen G, die mit dem Backtracking Algorithmus berechnet wurde

6 Implementierung

Um die zuvor vorgestellten Algorithmen in praktischer Hinsicht miteinander vergleichen zu können, wurde im Rahmen dieser Arbeit ein Programm entwickelt. Dieses Programm ist in der Lage, die Baumzerlegung für vom Nutzer eingegebene Graph zu berechnen und weitere Algorithmen zu berechnen, die von dieser Baumzerlegung abhängen. Ein Beispiel hierfür ist die schöne Baumzerlegung, die zur effizienten Lösung spezifischer Graphenprobleme eingesetzt werden kann. Die Implementierung des Programms erfolgte in der Programmiersprache Python[2], die sich aufgrund ihrer benutzerfreundlichen Syntax besonders gut für eine verständliche Implementierung der Algorithmen eignet.
Als Entwicklungsumgebung wurde PyCharm[3] von JetBrains ausgewählt, da PyCharm zahlreiche Funktionen bereitstellt, darunter Debugger, effizientes Projektmanagement und umfangreiche Unterstützung für verschiedene Python-Frameworks.
Im folgenden Abschnitt werden wir die Implementierung des Programms und deren Anwendung im Detail beschreiben. Abschließend erfolgt ein Vergleich der beiden Lösungsansätze für Graphenprobleme anhand ihrer Laufzeiten.

6.1 Struktur des Programms

Das Programm ist grundsätzlich in zwölf Dateien aufgeteilt: *algorithm.py, dec1.py, Alt-max-ind-set.py, G-coloring-BT.py, und validation.py*, diese bilden die Hauptkomponenten der Arbeit.

In *algorithm.py* wird die Implementierung des Min-Fill-In-Algorithmus sowie die Strukturierung und Verknüpfung von Taschen, die aus der Zerlegung resultieren, beschrieben. Die Datei validation.py dient der Validierung der berechneten Baumzerlegung, um festzustellen, ob sie gültig ist und alle Bedingungen einer Baumzerlegung erfüllt.

In der Datei *dec1.py* befinden sich die wesentlichen Teile des Projekts. Hier wurden Funktionen zur Berechnung der schönen Baumzerlegung, der Färbbarkeit und der maximal abhängigen Mengen implementiert.

In den Dateien *Alt-max-ind-set.py* und *G-coloring-BT.py* wurden alternative Verfahren für die maximale Unabhängige Menge und die Färbbarkeit mit dem Backtracking-Algorithmus implementiert.

Zusätzlich zu diesen Hauptdateien dienen die Dateien *Window01.py, Graphdraw.py, decomposition.py, nice.py, coloring.py* und *max-set-gui.py* zur Visualisierung der Ergebnisse, dabei wird die Bibliothek Graphviz [4] verwendet, über eine benutzerfreundliche grafische Benutzeroberfläche.

[2] `https://www.python.org/`
[3] `https://www.jetbrains.com/pycharm/`
[4] `https://graphviz.org/`

Window01.py stellt das anfänglich angezeigte Fenster dar und ermöglicht die Eingabe von Nutzer sowie die graphische Darstellung des Graphen. *Graphdraw.py* ist für die Verarbeitung der Benutzereingaben und die graphische Darstellung des Graphen verantwortlich. In dieser Datei ist die Implementierung von zwei Fenstern enthalten: eines für die maximale unabhängige Menge und eines für die Färbbarkeit mit dem Backtracking-Algorithmus.
decomposition.py präsentiert die Ergebnisse der Zerlegung, während *nice.py* die Ergebnisse der schönen Baumzerlegung anzeigt. Abschließend bieten die Dateien *coloring.py* und *max-set-gui.py* die Implementierung von zwei Fenstern zur Darstellung der Ergebnisse der Färbbarkeit und maximale Unabhängige Menge unter Verwendung der schönen Baumzerlegung.
In den weiteren Abschnitten dieser Arbeit werden die Funktionalität jeder Datei beschrieben.

6.2 Verwendung des Programmes

Das Programm bietet der Nutzer die Möglichkeit, Graphen einzugeben und anzuzeigen. Zunächst sollte der Datei *Window.01.py* ausgeführt werden, alle Benutzerschnittstellen in diesem Programm werden mithilfe der Bibliothek PyQt5[5] erstellt. Die Eingabe erfolgt durch das Eintragen der Kanten in das Textfeld, wobei jede Kante als ein Tupel repräsentiert wird und die Tupel durch Semikolon voneinander getrennt sind. Ein Beispiel für eine solche Eingabe wird in Abbildung 22 gezeigt.

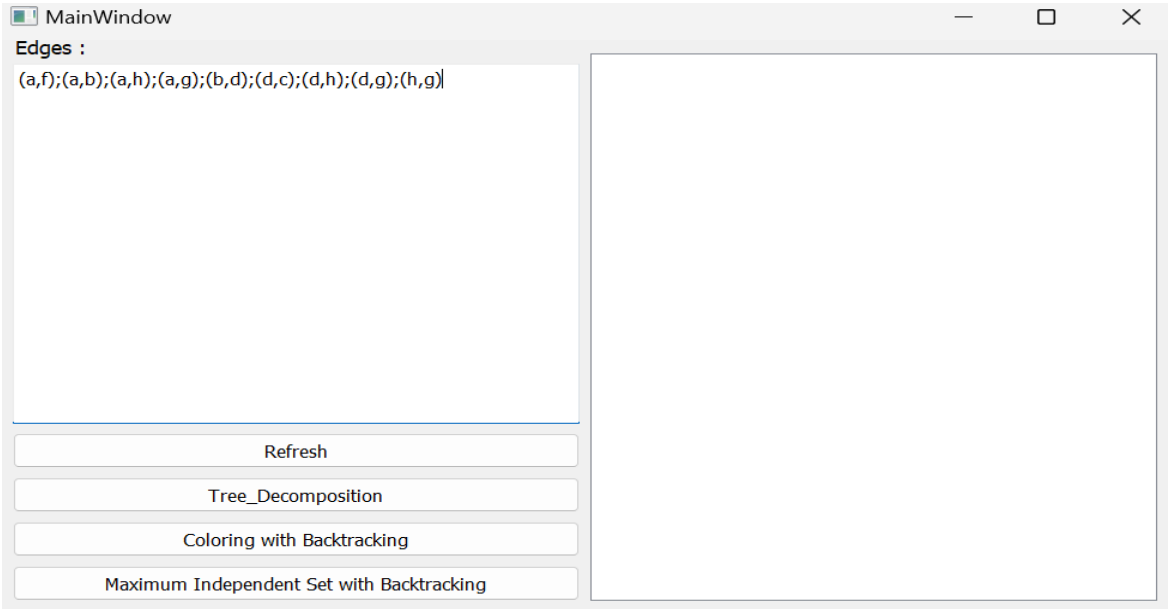


Abbildung 22: Kanteneingabefenster

Nach der Eingabe kann der Nutzer auf die Schaltfläche *Refresh* klicken, um den Graph anzuzeigen. Die Abbildung 23 zeigt, wie das Fenster nach der Eingabe der Kanten und dem Klicken auf die *Refresh*-Schaltfläche aussieht.

[5] `https://pypi.org/project/PyQt5/`

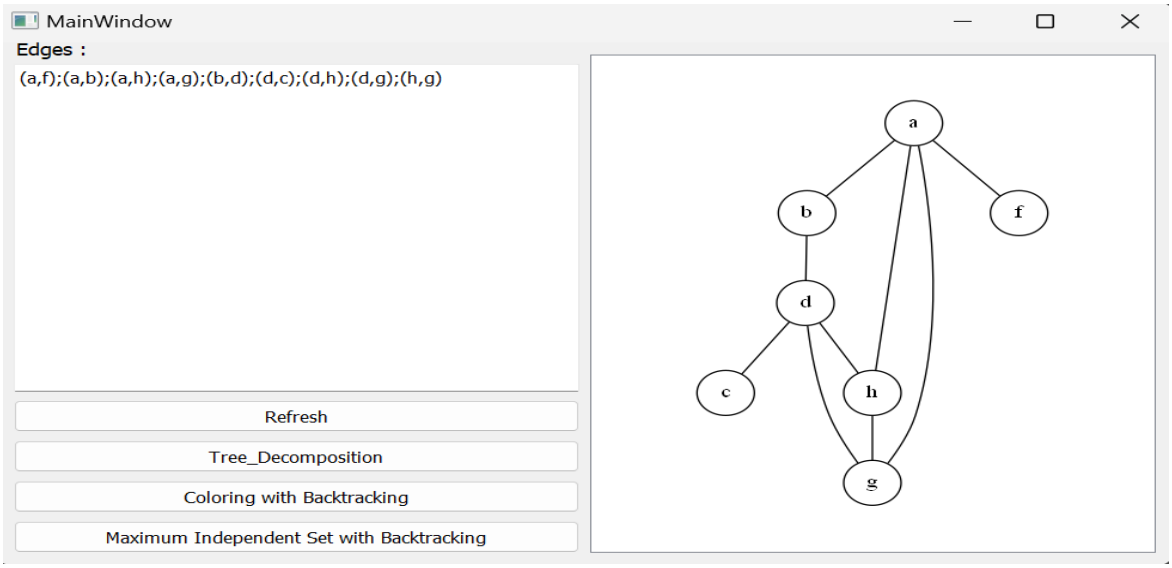


Abbildung 23: Fenster der Graphdarstellung

Falls der Nutzer die Eingabe ändern oder erweitern möchte, kann er einfach die neuen Kanten in das Textfeld eingeben und erneut auf die Schaltfläche *Refresh* klicken. Dem Nutzer stehen nun mehrere Optionen zur Verfügung. Beginnen wir mit den alternativen Verfahren. Wenn der Nutzer auf die Schaltfläche *Coloring With Backtracking* drückt, wird im Hintergrund der Backtracking-Algorithmus ausgeführt, und die resultierenden Ergebnisse wird in den entsprechenden Fenstern dargestellt. Wie in Abbildung 24. Die *Exit*-Schaltfläche in allen nachfolgenden Fenstern dient dazu, die geöffneten Fenster zu schließen.

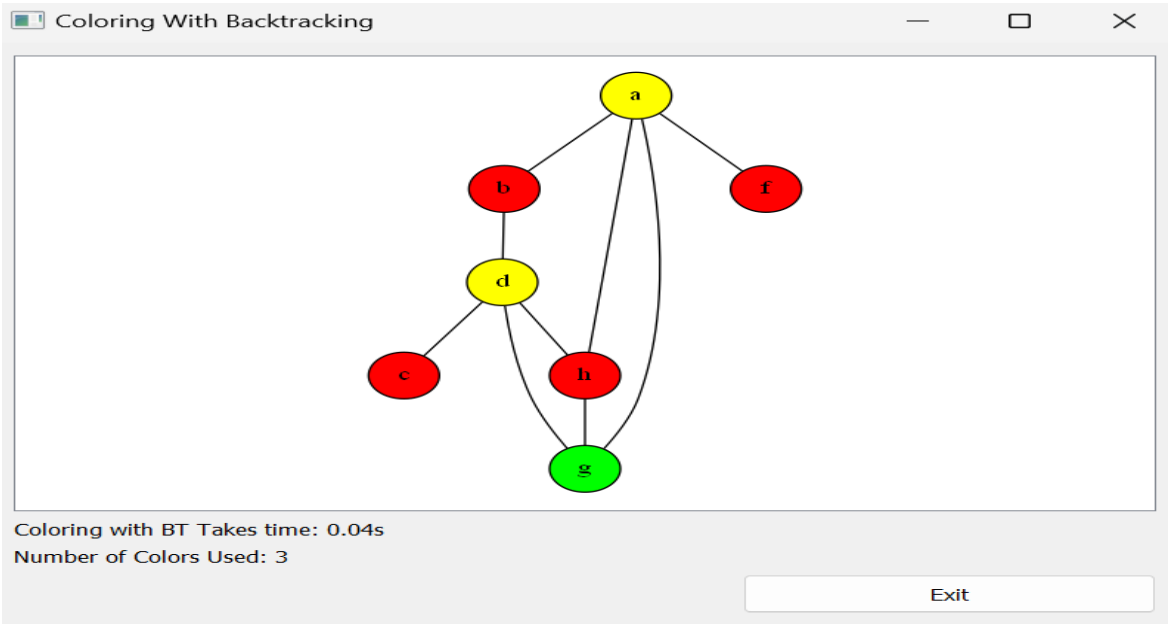


Abbildung 24: Färbungsfenster mit Backtracking Algorithmus

Das Klicken auf die Schaltfläche *Maximale Ind Set with Backtracking* öffnet ein neues

Fenster mit der Ergebnisse der maximalen unabhängigen Menge unter der Verwendung des Backtracking-Algorithmus.

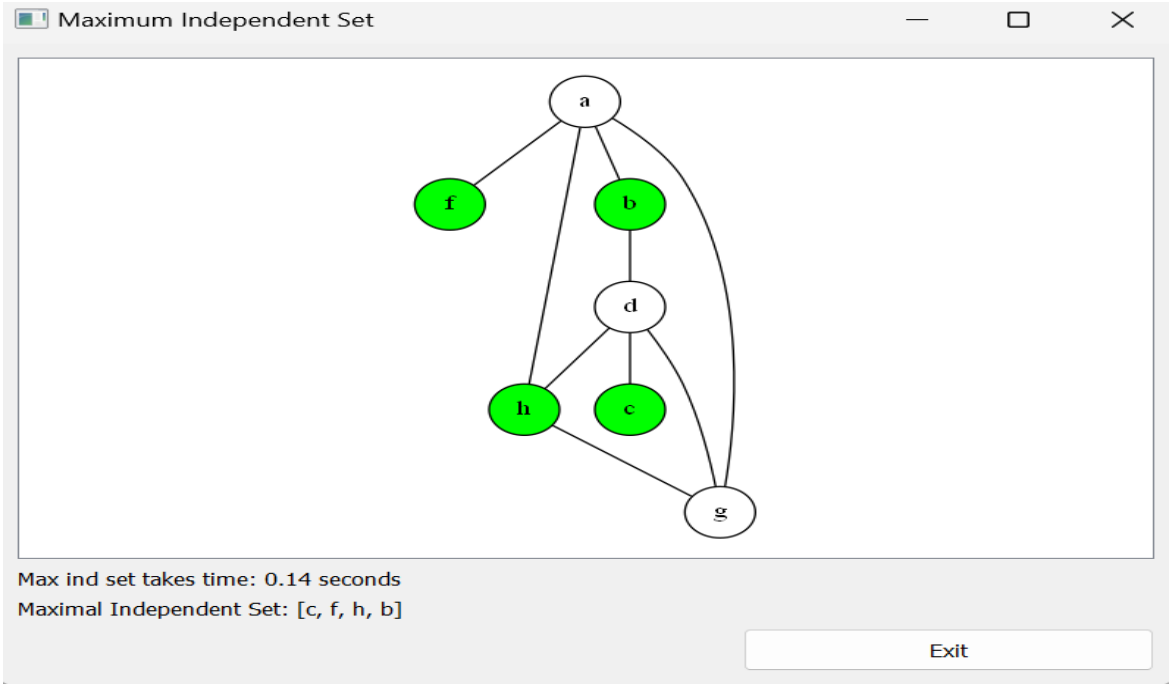


Abbildung 25: Fenster der maximalen unabhängigen Menge mit Backtracking

Der Klick auf die Schaltfläche *Tree Decomposition* bewirkt, dass das Programm die Ergebnisse der Baumzerlegung in einem neuen Fenster anzeigt. In Abbildung 26 ist die Oberfläche zu sehen, nachdem die Baumzerlegung erfolgreich abgeschlossen wurde und die Ergebnisse präsentiert wird.

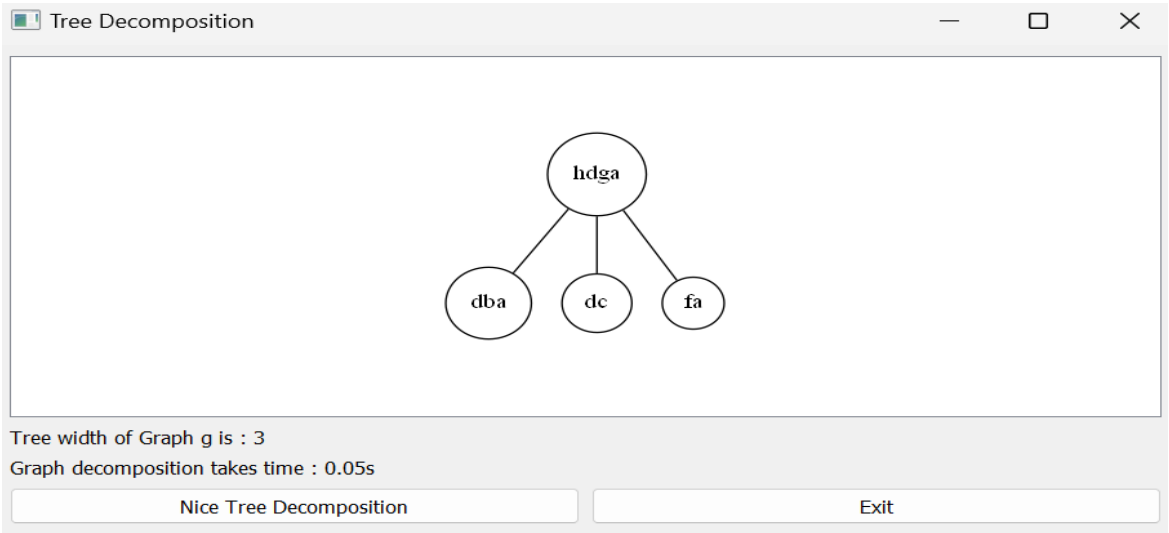


Abbildung 26: Fenster der Baumzerlegung

Wie in Abbildung 26 ersichtlich ist, initiiert der Nutzer die Berechnung der schönen Baumzerlegung, indem er auf die Schaltfläche *Nice* klickt. Infolgedessen öffnet sich

ein neues Fenster, das die ermittelten Ergebnisse der schönen Baumzerlegung präsentiert. Abbildung 27.

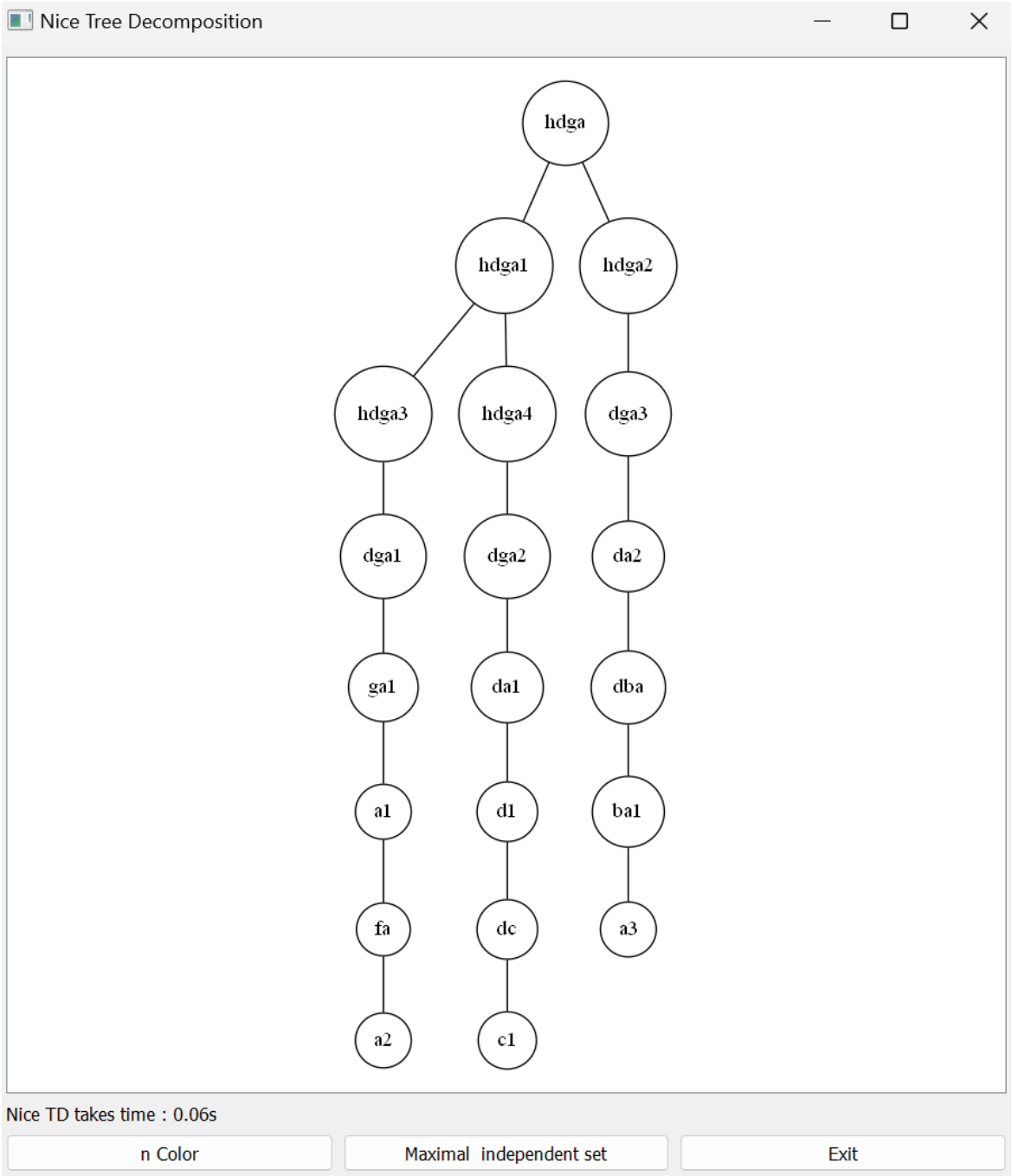


Abbildung 27: Schöne Baumzerlegung

Die in den Taschen der schönen Baumzerlegung enthaltenen Zahlen haben keine Relevanz mit dem Algorithmus selbst. Sie wurden aus Implementierungsgründen hinzugefügt, um eine eindeutige Unterscheidung zwischen den verschiedenen Kopien der Join-Knoten zu ermöglichen.
In der Benutzeroberfläche der schönen Baumzerlegung stehen dem Benutzer zwei

Möglichkeiten zur Verfügung: *n Color* und *Maximal independent set*. Wenn der Nutzer auf die Schaltfläche *n Color* klickt, öffnet sich ein neues Fenster, das ihn nach der gewünschten Anzahl von Farben für die Färbung des Graph fragt. Wenn die angegebene Anzahl nicht ausreicht, um den Graph zu färben, wird ihm eine Nachricht im Fenster angezeigt, die besagt, dass der Graph mit dieser Anzahl nicht färbbar ist, wie in Abbildung 28 gezeigt. In diesem Fall kann der Nutzer eine neue Anzahl von Farben eingeben und schließlich auf schaltfläche *Show* klicken. Wenn die gewählte Anzahl von Farben ausreicht, wird der gefärbte Graph angezeigt, Abbildung 29.

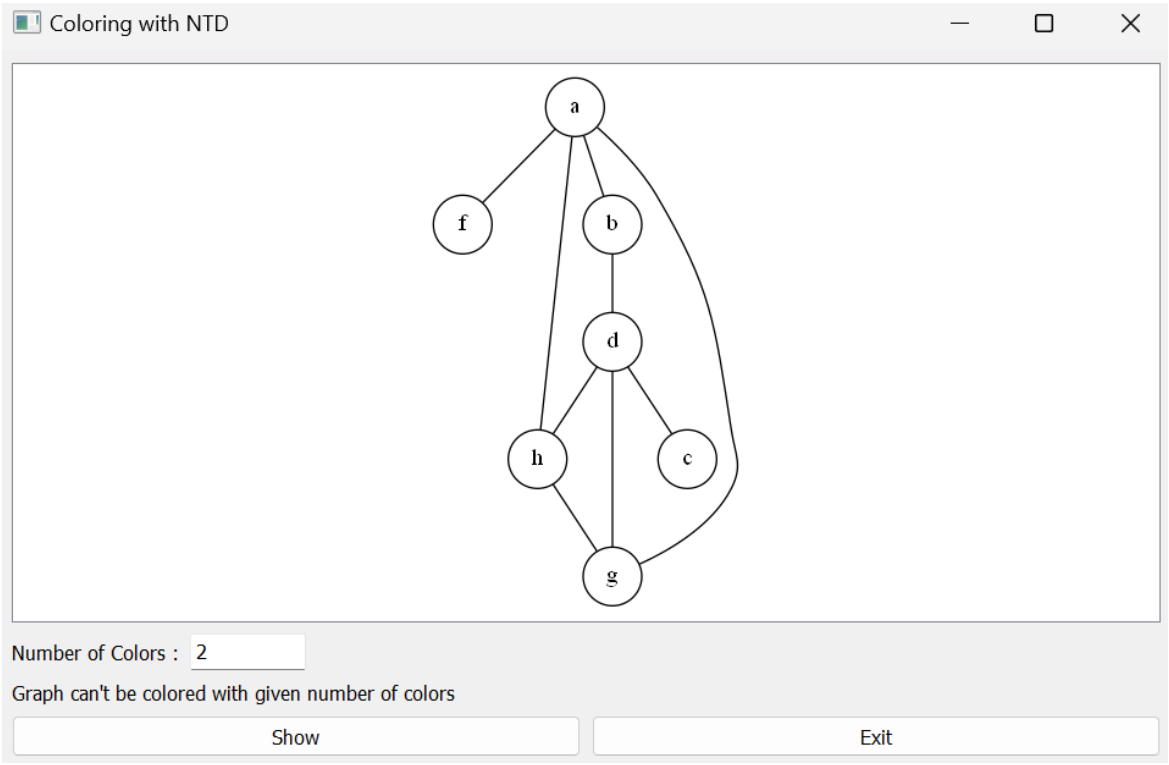


Abbildung 28: 2-Färbungsergebnisse

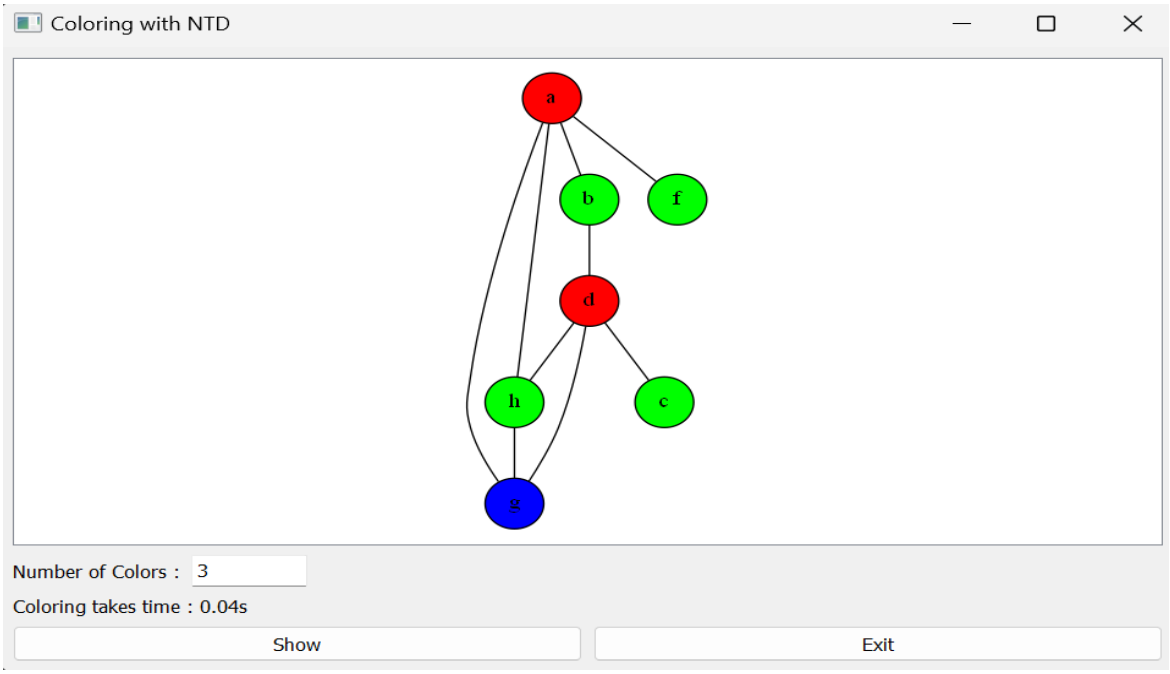


Abbildung 29: Graphenfärbung mit schöner Baumzerlegung

Die zweite Option besteht darin, die maximale unabhängige Menge berechnen zu lassen, indem der Nutzer auf *Maximal independent set* klickt. Die Ergebnisse wird in einem neuen Fenster angezeigt, wie in Abbildung 30.

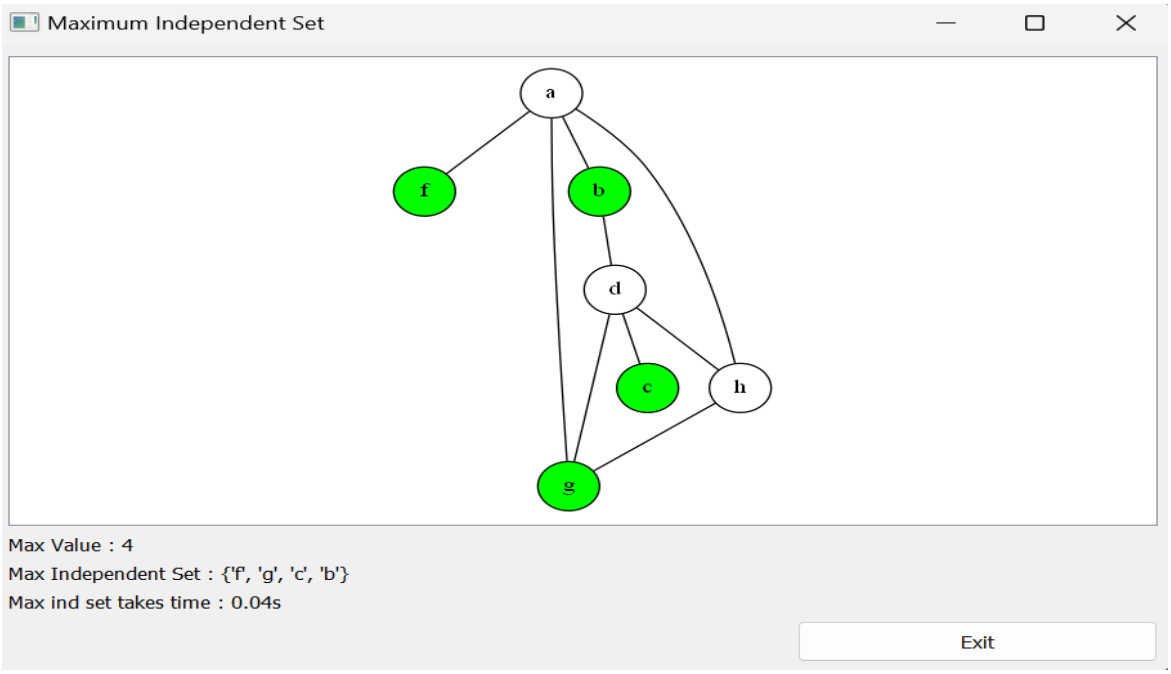


Abbildung 30: Maximale unabhängige Menge mit schöner Baumzerlegung

6.3 Implementierungsdetails des Projekt

Während der Projektimplementierung lag der Schwerpunkt auf der Umsetzung der zuvor erwähnten Algorithmen. Dies erforderte die Entwicklung eines Programms, das die spezifischen Anforderungen dieser Algorithmen erfüllt. Zur Gewährleistung der erforderlichen Funktionalität wurden verschiedene Bibliotheken und Tools verwendet.
Die Implementierung erfolgte in mehreren Schritten, wobei der Algorithmus zur Berechnung der Baumzerlegung als Ausgangspunkt diente. Diese Wahl war von entscheidender Bedeutung, da eine gute Baumzerlegung die Grundlage für die Leistung anderer Algorithmen bildete. Daher wurde besonders Augenmerk auf die Entwicklung dieses Algorithmus gelegt.
In diesem Abschnitt werden wir erläutern, wie die Algorithmen entwickelt wurden und welche Besonderheiten bei der Implementierung auftraten.

6.3.1 Implementierung und Komplexität der Algorithmen

Wie im vorherigen Abschnitt *Struktur des Programms* 6.1 beschrieben, enthält die Datei algorithm.py die eigentliche Implementierung des Min-Fill-in-Algorithmus, der die Berechnung der Baumzerlegung eines gegebenen Graphen ermöglicht. Der Algorithmus besteht aus zwei Funktionen: Die erste Funktion identifiziert den Knoten im Graphen, der den geringsten Anzahl von Kanten erfordert, um seine Nachbarschaft zu verbinden. Die zweite Funktion verwendet die ausgewählten Knoten, um die endgültige Baumzerlegung des Graphen zu erstellen. Diese Baumzerlegung wird durch das Zusammenfassen von Knoten in Taschen realisiert, und anschließend werden sie miteinander in einem Baumstruktur verbunden wird ein Baum erstellt, dessen Baumweite berechnet und zurückgegeben wird.

Zur Implementierung des Algorithmus wird die Bibliothek Networkx[6] verwendet, die eine umfangreiche Sammlung von Graphenoperationen und Algorithmen, was die Entwicklung von Algorithmen in Python vereinfacht.
Die Laufzeitkomplexität des Min-Fill-In-Algorithmus beträgt $O(n^3)$, wobei n die Anzahl der Knoten im Graph repräsentiert. Diese Laufzeitkomplexität ergibt sich aus den Iterationen und Schleifen über Knoten und deren Nachbarschaften im Algorithmus. Es ist jedoch zu beachten, dass die tatsächliche Laufzeit von verschiedenen Faktoren abhängt, einschließlich der spezifischen Struktur des Eingabe-Graphen.

Um sicherzustellen, dass die resultierende Baumzerlegung in einem bestimmten Format vorliegt und keine doppelten Taschen enthält, haben wir zusätzlich eine Funktion entwickelt, die die Baumzerlegung als Eingabe nimmt und sie überprüft, um sicherzustellen, dass sie den entsprechenden Qualitäts und Strukturkriterien entspricht.
Basierend auf der Ergebnisse der Baumzerlegung wird eine Funktion entwickelt, die die schöne Baumzerlegung berechnet. Obwohl sie den algorithmischen Umfang

[6] `https://networkx.org/`

nicht erweitert, kann sie dennoch zu einer verbesserten Laufzeit führen (Siehe - Kapitel 4). Die Funktion *nice-tree* in der Datei *dec1.py* nimmt eine List von Kanten in einem Baum als Eingabe und führt eine Optimierung auf dieser Baumstruktur durch. Die Optimierung erfolgt durch die Anwendung spezifischer Regeln und Bedingungen, um die Baumstruktur zu verbessern. In diesem Fall wird die Baumstruktur schrittweise durchlaufen, und je nach Anzahl der Nachfolger eines Knotens werden unterschiedliche Optimierungen vorgenommen. Das Ergebnis dieser Funktion ist eine optimierte Baumstruktur, die in Form des Wurzelknotens und einer Liste von Baumkanten zurückgegeben wird. Die Zeitkomplexität dieser Funktion hängt von der Größe und Struktur der Eingabe ab, und kann im Allgemeinen als $O(n)$ eingeschätzt werden.

Die Funktion *coloring* dient dazu, die Färbung von Graphen zu ermöglichen. Sie erwartet zwei Listen von Kanten als Eingabe: Eine die ursprünglichen Kanten des Graphen und eine der schönen Baumzerlegung, sowie einen Startknoten und die Anzahl der gewünschten Farben. Die Funktion erstellt einen DFS-Baum (Depth-First Search) aus dem Eingabebaum und durchläuft diesen Baum in *Postorder-Reihenfolge*. Dies bedeutet, dass zuerst der linke Teil des Baumes besucht wird, dann der rechte Teil und schließlich die Wurzel des Baumes. Dabei werden die Knoten basierend auf ihren Farben kombiniert, und die Färbungsinformationen werden ausgegeben. Wenn eine gültige Färbung gefunden wird, erfolgt die Auswahl der Knoten in einer *Preorder-Reihenfolge*. Hierbei wird zuerst die Wurzel des Baumes betrachtet, anschließend der linke Teil und schließlich der rechte Teil des Baumes, um eine mögliche Färbung der Knoten auszuwählen. Andernfalls wird *None* zurückgegeben.
Die Zeitkomplexität der Funktion beträgt $O(r^k \cdot k \cdot n)$, wobei k die Baumweite des Graphen, r die Anzahl der Farben und n Anzahl der Knoten in Graph ist. Da jeder Knoten und jede Kante höchstens einmal besucht wird, bleibt die Komplexität linear $O(n)$ in Bezug auf die Größe des Graphen, wenn die Baumweite des Graphen k klein ist. Es ist jedoch wichtig zu beachten, dass die Laufzeit exponentiell wachsen kann, wenn die Baumweite k des Graphen groß ist, da dies die Anzahl der möglichen Farbkombinationen erhöht und somit die Ausführungszeit der Funktion aufwändiger macht.

Die letzte Methode in *dec1.py* zielt darauf ab, die maximale unabhängige Menge in einem Graph zu berechnen. Dieser Algorithmus unterteilt die Berechnung in zwei Hauptphasen, ähnlich wie bei der Färbung: Zunächst werden Kandidaten für jede Tasche im Baum berechnet, und anschließend wird die tatsächliche maximale unabhängige Menge ermittelt. Dies erfolgt durch eine Kombination von *Postorder- und Preorder-Reihenfolge* des Baumes, wobei verschiedene Berechnungen und Überlegungen zur Unabhängigkeit der Knoten berücksichtigt werden. Es ist jedoch wichtig zu beachten, dass diese Methode aufgrund der Berechnung von Kandidaten für jede Teilmenge der Knoten in den Taschen der schönen Baumzerlegung eine exponentielle Laufzeitkomplexität aufweisen kann, wenn die Baumweite des Graphen groß ist $O(2^k \cdot n)$. Wenn die Baumweite hingegen klein ist, kann die Komplexität linear in Bezug auf die Größe des Graphen sein.

Die beiden alternativen Verfahren zur Graphenfärbung und zur Suche nach maximalen unabhängigen Mengen sind in den Dateien *G-coloring-BT.py* und *Alt-max-ind-set.py* implementiert. Beide Methoden nutzen den Backtracking-Ansatz, um den Graph zu färben bzw. die maximale unabhängige Menge zu finden. Die Vorgehensweise des Backtracking Algorithmus wird in den Abschnitten 5.2.2 und 5.3.2 beschrieben.
Die Zeitkomplexität für die Färbung des Graphen mit Backtracking beträgt $O(r^n)$, wobei n die Anzahl der Knoten im Graph und r die Anzahl der Farben ist. Dies führt zu einer exponentiellen Laufzeit in Bezug auf die Anzahl der Knoten im Graphen.
Auf der anderen Seite beträgt die Zeitkomplexität für die Suche nach der maximalen unabhängigen Menge mit Backtracking $O(2^n)$, wobei n die Anzahl der Knoten im Graph ist. Auch hier handelt es sich um eine exponentielle Laufzeit in Bezug auf die Anzahl der Knoten im Graph.

6.3.2 Validation

Da die Baumzerlegung der Ausgangspunkt für die Entwicklung anderer Algorithmen ist, wurde eine Funktion in der Datei *validation.py* implementiert, die die Bedingungen einer gültigen Baumzerlegung überprüft und *True* zurückgibt, wenn alle drei Bedingungen erfüllt sind, andernfalls *False*.

Um die Korrektheit der Algorithmen zu beweisen, führen wir Tests anhand von zufällig generierten Graphen durch. Die Art und Weise, wie wir diese zufälligen Graph erstellen, hat einen erheblichen Einfluss auf die Leistung der Algorithmen. Dies liegt daran, dass die Qualität der Algorithmen stark von der Anzahl der Kanten im Graph abhängt.

Um die Verbundenheit eines zufällig generierten Graphen sicherzustellen, wurde in der Datei *Evaluation.py* eine spezielle funktion verwendet. Dabei wird der Graph schrittweise aufgebaut, indem zuerst eine Anfangskante erstellt wird. Anschließend werden zufällige Gruppen von Knoten gebildet, zwischen denen Kanten mit einer Wahrscheinlichkeit von 50% hinzugefügt werden. Weitere zufällige Kanten werden hinzugefügt, bis der Graph vollständig zusammenhängend ist. Diese funktion erzeugt somit einen zufälligen Graph mit gewährleisteter Verbundenheit.

Vom Beginn an wurde nur mit Zeichenketten (Character) gearbeitet, während die von NetworkX generierten zufälligen Graphen in der Regel mit ganzen Zahlen (Integer) beschriftet sind. Aus diesem Grund wurde eine Funktion implementiert, die verschiedene Alphabete wie Chinesisch, Englisch, Deutsch, Spanisch, Persisch und Russisch zusammenführt. Eine weitere Funktion wurde verwendet, um die Knotenbeschriftungen von ganzen Zahlen in Zeichenketten umzuwandeln.

6.3.3 Laufzeit der Algorithmen

In diesem Abschnitt präsentieren und diskutieren wir die Laufzeitergebnisse der Algorithmen.

Das Programm erfasst mithilfe der Time[7] Bibliothek die Zeit, die für die Ausführung der Algorithmen benötigt wird, und erstellt Diagramme mithilfe der Matplotlib[8] Bibliothek in Python. Diese Zeitmessung ermöglicht uns, die Leistung der Algorithmen zu analysieren und zu optimieren.

Im Folgenden werden die Algorithmen für die Baumzerlegung (BZ) und die schöne Baumzerlegung (S.BZ) anhand von zufälligen Graphen getestet.

Knoten	Kanten	Baumweite	Zeit der BZ	Zeit der S.BZ
8	12	2	0.000	0.000
102	151	10	0.004	0.010
205	351	28	0.039	0.066
303	500	49	0.090	0.151
401	667	59	0.139	0.224
508	1081	111	0.489	0.710
577	856	65	0.205	0.300
705	1323	130	0.794	1.128
798	1729	179	1.623	2.273

Tabelle 3: Testergebnisse für Baumzerlegung und schöne Baumzerlegung mit zufälligen Graphen

In der Tabelle werden die ersten beiden Spalten verwendet, um die Anzahl der Knoten und Kanten im zufälligen Graphen darzustellen. In der nächsten Spalte wird die Baumweite des Graphen angegeben. Die letzten beiden Spalten zeigen die Ausführungszeiten beider Methoden in Sekunden, wobei die Werte auf drei Dezimalstellen gerundet sind.

In Abbildung 31 ist das Zeitdiagramm dargestellt, welches die Beziehung zwischen der Anzahl der Knoten und der benötigten Zeit in Sekunden zeigt. Jeder Punkt im Diagramm entspricht einer bestimmten Anzahl von Knoten und der dazugehörigen Zeit. Beim Betrachten des Diagramms wird ersichtlich, dass mit zunehmender Anzahl von Knoten und Kanten die benötigte Ausführungszeit der Algorithmen ansteigt.

Bei der Betrachtung von Tabelle 3 und Abbildung 31 wird deutlich, dass die schöne Baumzerlegung mehr Zeit in Anspruch nimmt als die normale Baumzerlegung. Dies liegt daran, dass die Berechnung der schönen Baumzerlegung die Berechnung der

[7] `https://docs.python.org/3/library/time.html`
[8] `https://matplotlib.org/`

normalen Baumzerlegung einschließt. Das bedeutet, dass die Zeit der normale Baumzerlegung in die Zeit der schönen Baumzerlegung einfließt.

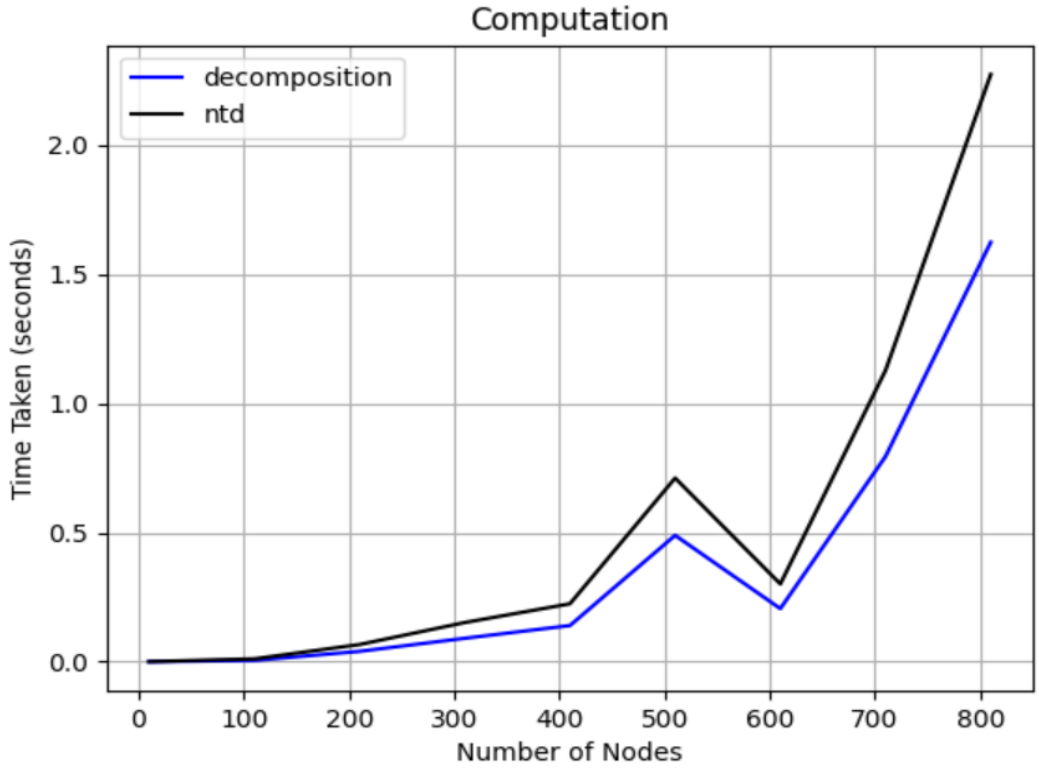


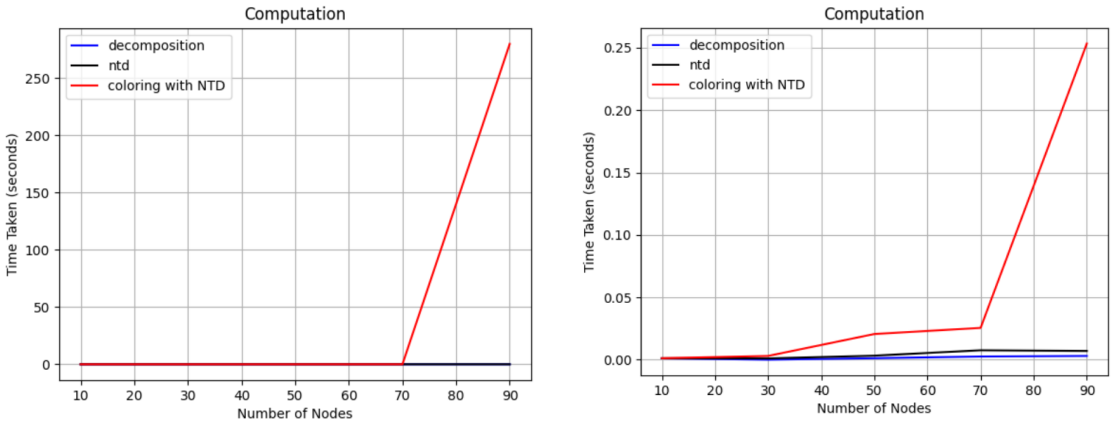
Abbildung 31: Graphische Darstellung der Testergebnisse

Die Ergebnisse der Baumzerlegung und schönen Baumzerlegung können dazu beitragen, verschiedene Graphenprobleme effizient zu lösen, insbesondere wenn die Baumweite klein ist. Bei einer größeren Baumweite steigt die Laufzeit exponentiell an.
Nachfolgend werden wir mithilfe der schönen Baumzerlegung das Färbbarkeitsproblem lösen. Dieses Problem befasst sich damit, eine gültige Färbung für einen gegebenen Graph in polynomialer Zeit zu finden.
Die Tabelle 4 zeigt die für die Färbung verschiedener zufälliger Graphen benötigte Zeit mithilfe der schönen Baumzerlegung.

Knoten	Kanten	Baumweite	Zeit der Färbung mit S.BZ
9	9	2	0.001
26	35	4	0.004
45	55	4	0.013
66	76	4	0.014
82	124	10	279.864

Tabelle 4: Laufzeiten des Färbungsverfahrens unter schöner Baumzerlegung

Die Abbildung 32 zeigt die grafische Darstellung der Graphen, die in der Tabelle 4 gegeben sind. Auf der rechten Seite 32b ist ein Graph mit 83 Knoten dargestellt, bei dem die Baumweite 6 beträgt und die Ausführungszeit deutlich besser ist als bei dem, was auf der linken Seite 32a in Abbildung dargestellt ist.



(a) Die Baumweite für Graph mit 82 Knoten ist 10 **(b)** Die Baumweite für Graph mit 90 Knoten ist 6

Abbildung 32: Visualisierung der Testergebnisse für Färbung

Die Zeit für die Färbung erfasst sowohl die Zeit für die Baumzerlegung als auch für die schöne Baumzerlegung.

Die Färbung von Graphen mit dem Backtracking-Algorithmus kann effizient sein für sehr kleine Graphen, bei denen die Anzahl der möglichen Teillösungen nicht groß ist. Dies trifft jedoch nicht zu, wenn der Graph groß ist. Die Abbildung veranschaulicht die Laufzeit des Backtracking-Verfahrens für Graphen bis 30 Knoten. Es sollte beachtet werden, dass die Anzahl der Knoten beim Backtracking Algorithmus nicht bedeutet, dass er mit einer größeren Anzahl von Knoten nicht funktioniert, sondern dass er bei einer größeren Anzahl von Knoten erheblich mehr Zeit benötigt.

Knoten	Kanten	Zeit der Färbung mit S.BZ	Zeit der Färbung mit Backtracking
13	13	0.001	0.014
19	21	0.001	0.385
25	40	0.000	0,377
30	44	0.009	7.680
35	50	0.010	57.452

Tabelle 5: Vergleich der Laufzeiten beider Färbungsverfahren

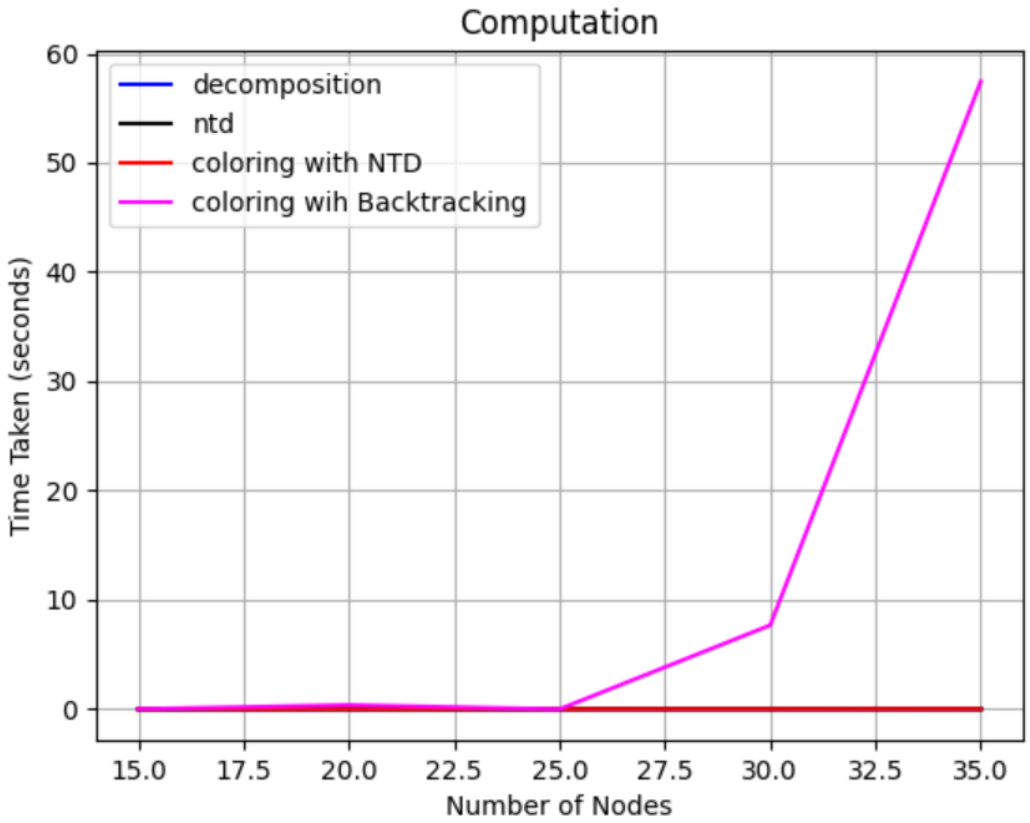


Abbildung 33: Visualisierung der Laufzeitunterschiede zwischen den beiden Färbungsverfahren

Bei der Färbung mit schöner Baumzerlegung kann die Dauer der Färbung eines Graphen mit 100 Knoten sowohl schnell als auch langsam sein, abhängig von der Baumweite des Graphen. Wie in Abbildung 32 dargestellt, hat die Baumweite einen erheblichen Einfluss auf die Laufzeit. Im Gegensatz dazu steigt die Laufzeit beim Backtracking Algorithmus schneller (exponentiell) mit der Größe des Graphen an.

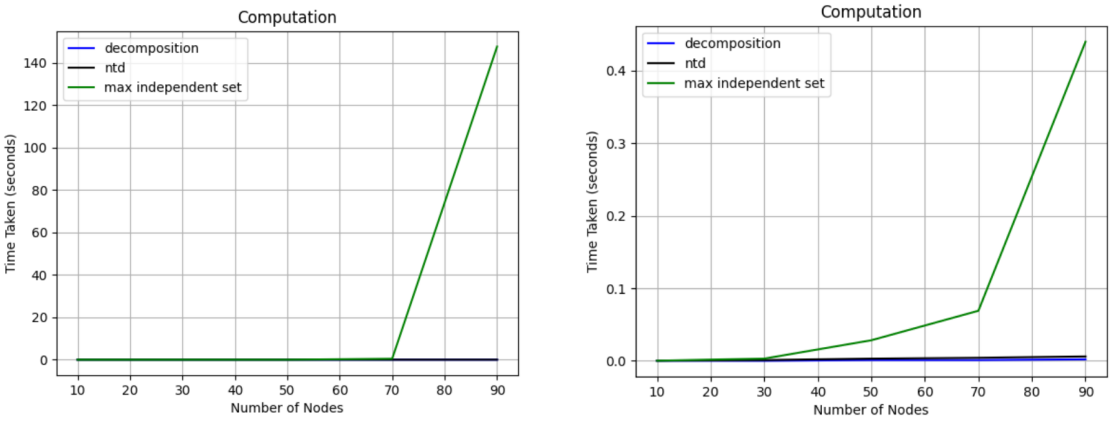
Ein weiteres Graphenproblem, dem wir uns widmen möchten, ist das Finden der maximalen unabhängigen Menge (MUM) in einem Graph. Ähnlich wie bei dem zuvor vorgestellten Verfahren verwenden wir hierfür zwei verschiedene Methoden: Die schöne Baumzerlegung und den Backtracking Algorithmus. Im Rahmen dieser Untersuchung werden wir auch die Laufzeiten beider Methoden betrachten.

Wenn wir die Tabelle 6 betrachten, fällt auf, dass der Algorithmus langsamer wird, wenn die Baumweite größer als 10 ist. In Abbildung 34 auf der rechten Seite, insbesondere in 34a, wird die Laufzeit des Algorithmus für die in Tabelle 6 aufgeführten Graphen veranschaulicht. Ebenso zeigt die gleiche Abbildung in 34b, dass der Algorithmus schneller arbeitet, wenn der Graph 90 Knoten und eine Baumweite von 7 hat. Dies legt nahe, dass der Algorithmus bei einer Baumweite von weniger als 10

Knoten	Kanten	Baumweite	Zeit der MUM Algorithmus mit S.BZ
8	7	1	0.001
29	37	4	0.009
45	59	4	0,011
67	94	7	0.473
88	144	12	147

Tabelle 6: Laufzeiten des MUM-Verfahrens mit schöner Baumzerlegung

in linearer Zeit in Bezug auf die Größe des Graphen läuft. Andererseits steigt die Laufzeit exponentiell mit der Größe des Graphen, wenn die Baumweite größer als 10 ist.



(a) Die Baumweite für Graph mit 88 Knoten ist 12 **(b)** Die Baumweite für Graph mit 90 Knoten ist 7

Abbildung 34: Graphische Darstellung der Testergebnisse für maximale unabhängige Menge

Die Zeit für MUM erfasst die Zeit für die Baumzerlegung und schöne Baumzerlegung.

Im Folgenden stellen wir den Unterschied zwischen den beiden MUM-Verfahren anhand ihrer Laufzeiten dar. Das zweite Verfahren zur Berechnung der maximalen unabhängigen Menge im Graph basiert auf dem Backtracking Algorithmus, der alle möglichen unabhängigen Mengen im Graph berechnet und der maximalen unabhängigen Mengen zurückgibt, was seine exponentielle Laufzeit erklärt.

Die Tabelle 7 zeigt die benötigte Zeit für die jeweiligen Verfahren.
Abbildung 35 zeigt das zeitliche Diagramm beider Verfahren.

Knoten	Kanten	Zeit der MUM mit S.BZ	Zeit der MUM mit Backtracking
7	6	0.000	0.001
10	9	0.000	0.003
19	19	0.001	0.061
22	24	0.001	0.245
30	33	0.002	1.079
34	42	0.006	30.615
39	48	0.009	129.586

Tabelle 7: Vergleich der Laufzeiten beider MUM Verfahren

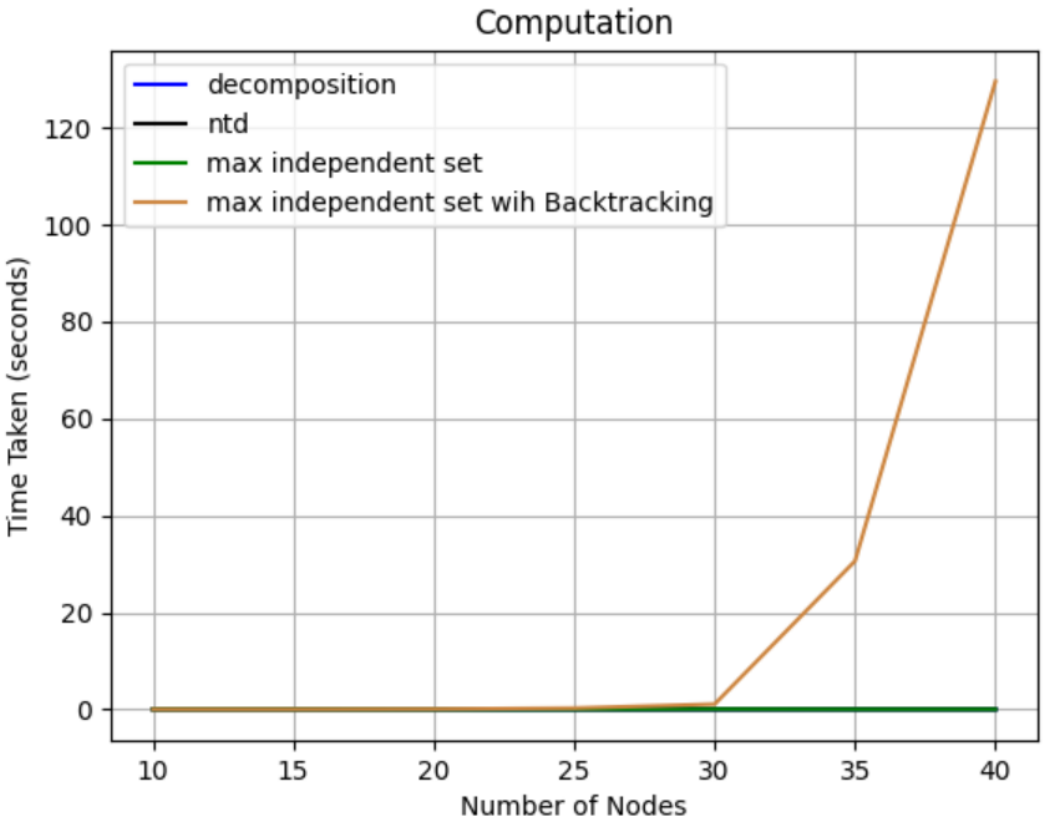


Abbildung 35: Graphische Darstellung der Testergebnisse für beide Verfahren der MUM

7 Fazit und zukünftige Arbeit

Zusammenfassend haben wir sechs Funktionen entwickelt für die Lösung von zwei verschiedenen Graphproblemen, nämlich dem Färbbarkeitsproblem und der Berechnung der maximalen unabhängigen Menge. Die ersten beiden Methoden bilden die Grundlage für die folgenden zwei Verfahren. Die Baumzerlegung wird zuerst berechnet und dann die schöne Baumzerlegung. Die Einzelheiten dieser Schritte wurden in den Kapiteln 3 und 4 ausführlich erörtert.

Bei der Lösung von Graphproblemen war es wichtig, auf die erhebliche Auswirkung der Baumweite zu achten. Zudem wurde erklärt, wie sich die Laufzeit der beiden Verfahren bei verschiedenen Baumweiten verhält. Außerdem wird zwecks Vergleich der Methoden zwei weitere Verfahren vorgestellt, die nach dem Backtracking Prinzip arbeiten.

In der Implementierung haben wir erfolgreich alle Konzepte aus dem theoretischen Teil umgesetzt. Um die vorgestellten Algorithmen und Verfahren anschaulicher zu gestalten, wurde ein Programm entwickelt, mit dem Benutzer Graphen eingeben und visualisieren können. Darüber hinaus ermöglicht das Programm dem Benutzer, den Graphen auf zwei verschiedene Arten zu färben oder die maximale unabhängige Menge zu finden.

Eine besondere Herausforderung in der Implementierung trat im Evaluationsabschnitt auf. Anfangs arbeiteten wir mit Zeichen, aber dann sollte der Graph automatisch generiert werden. Die Idee war, verschiedene Alphabete zusammenzuführen. Ein weiterer Fall betraf die Labeling der Knoten mit ganzen Zahlen. Eine mögliche Verbesserung der Laufzeit könnte darin bestehen, anstelle von Zeichen mit Zahlen zu arbeiten. Dies würde Zeit der beiden Schritten sparen. In der Implementierung der maximalen unabhängigen Menge mit Verwendung der schönen Baumzerlegung könnten wir anstatt der Berechnung in Bottom-Up und dann in Top-Down-Reihenfolge zu arbeiten, die Knoten mit dem maximalen Wert von den Blättern zur Wurzel schrittweise auswählen, was die Laufzeit ebenfalls verbessern würde. Als zukünftige Arbeit könnte die Lösung weiterer Graphprobleme in Betracht gezogen werden, wie das Domaintag Set Problem und andere Probleme. Alles in allem zeigt die Arbeit, wie unterschiedliche Methoden zur Lösung von Graphproblemen eingesetzt werden können und welche Auswirkungen die Wahl der Methode und bestimmte Implementierungsdetails auf die Laufzeit haben können.

Schließlich lässt sich feststellen, dass die Umsetzung und Implementierung von Graphalgorithmen eine anspruchsvolle Herausforderung darstellt, die ein tiefes Verständnis der theoretischen Grundlagen erfordert.

Literatur

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete applied mathematics*, 23(1):11–24, 1989.

[BB05] Emgad H Bachoore and Hans L Bodlaender. New upper bound heuristics for treewidth. In *International Workshop on Experimental and Efficient Algorithms*, pages 216–227. Springer, 2005.

[BFK+06] Hans L Bodlaender, Fedor V Fomin, Arie MCA Koster, Dieter Kratsch, and Dimitrios M Thilikos. On exact algorithms for treewidth. In *European Symposium on Algorithms*, pages 672–683. Springer, 2006.

[BK08] Hans L Bodlaender and Arie MCA Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[BK10] Hans L Bodlaender and Arie MCA Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[Bod97a] Hans L Bodlaender. Treewidth: Algorithmic techniques and results. In *International Symposium on Mathematical Foundations of Computer Science*, pages 19–36. Springer, 1997.

[Bod97b] Hans L Bodlaender. Treewidth: Algorithmic techniques and results. In *International Symposium on Mathematical Foundations of Computer Science*, pages 19–36. Springer, 1997.

[Bod05] Hans L Bodlaender. Discovering treewidth. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 1–16. Springer, 2005.

[Gol80] Martin Charles Golumbic. Algorithmic graph theory and perfect graphs, courant institute of mathematical science, new york university. *Academic Press*, 1:980, 1980.

[HZ] Martin Hebenstreit and Dipl-Ing Dr Volker Ziegler. Färbung von graphen grundlagen, algorithmen und anwendungen.

[KBVH01] Arie MCA Koster, Hans L Bodlaender, and Stan PM Van Hoesel. Treewidth: computational experiments. *Electronic Notes in Discrete Mathematics*, 8:54–57, 2001.

[Klo94] Ton Kloks. *Treewidth: computations and approximations*. Springer, 1994.

[Kor20] Tuuka Korhonen. Finding optimal tree decompositions, 2020.

[Mou14] Lalla Mouatadid. Introduction to complexity theory: 3-colouring is np-complete. 2014.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[Sat14] Christina Satzinger. On tree-decompositions and their algorithmic implications for bounded-treewidth graphs. Diplomarbeit, 2014.

[Sch89] Petra Scheffler. Die baumweite von graphen als ein mass für die kompliziertheit algorithmischer probleme. 1989.

[Sud16] Benny Sudakov. Graph theory, 2016.

[TT77] Robert Endre Tarjan and Anthony E Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[Wal20] Anton Wallgren. Treewidth of graphs and algorithmic implications, 2020.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit bzw. im Fall einer Gruppenarbeit den von mir entsprechend gekennzeichneten Anteil an der Arbeit selbständig verfasst habe. Ich habe keine unzulässige Hilfe Dritter in Anspruch genommen. Zudem habe ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Ausführungen, die anderen Quellen wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht. Ich versichere, dass die von mir in elektronischer Form eingereichte Version dieser Arbeit mit den eingereichten gedruckten Exemplaren übereinstimmt. Mir ist bekannt, dass im Falle eines Täuschungsversuches die betreffende Leistung als mit “nicht ausreichend” (5,0) bewertet gilt. Zudem kann ein Täuschungsversuch als Ordnungswidrigkeit mit einer Geldbuße von bis zu 50.000 Euro geahndet werden. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuchs kann ich zudem exmatrikuliert werden. Mir ist bekannt, dass sich die Prüferin oder der Prüfer bzw. der Prüfungsausschuss zur Feststellung der Täuschung des Einsatzes einer entsprechenden Software oder sonstiger elektronischer Hilfsmittel bedienen kann.

Duisburg, 11.10.2023

Ort, Datum Unterschrift