FernUniversität in Hagen
Fachbereich Mathematik

Diplomarbeit

Erkennen von Graphenklassen mittels lexikographischer Breitensuche

von
Mathias Biermann

Aufgabenstellung und Betreuung:
Prof. Dr. W. Hochstättler

Hannover, den 31. Juli 2007

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Hannover, den 31. Juli 2007 Mathias Biermann

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

1 Einleitung

Graphen sind in der Modellierung vieler bekannter Optimierungsprobleme von herausragender Bedeutung. Einige wichtige Probleme der Graphentheorie, gemessen an der Zahl ihrer praktischen Anwendungen, sind für allgemeine Graphen nur mit erheblicher Rechenzeit zu lösen. Für bestimmte Graphenklassen, wie zum Beispiel chordale Graphen, haben viele dieser Probleme deutlich günstigere Laufzeiten. Die Zugehörigkeit einzelner Graphen zu bestimmten Graphenklassen zu überprüfen, kann also eine lohnende Aufgabe sein. Dabei hat sich die lexikographische Breitensuche LexBFS (= engl. Lexicographic Breadth First Search) als sehr erfolgreiches Konzept erwiesen. Die Anwendung von LexBFS zur Erkennung bestimmter Graphenklassen steht im Mittelpunkt dieser Arbeit. Ursprünglich war LexBFS entwickelt worden, um chordale Graphen in Linearzeit zu erkennen. Mittlerweile gibt es auf LexBFS basierende Algorithmen, die in Linearzeit Cographen, distance-hereditary Graphen, P_4-reduzierbare Graphen, P_4-spärliche Graphen u. a. erkennen. Für einige Klassen erreichten die LexBFS-basierten Algorithmen erstmalig Linearzeit. Für andere Klassen sind sie deutlich einfacher zu implementieren als andere Algorithmen. Weitere Anwendungen von LexBFS sind u. a. die exakte oder näherungsweise Durchmesserbestimmung für Graphen einzelner Graphenklassen in Linearzeit.

Zentraler Punkt dieser Arbeit sind die konkreten Algorithmen und vor allem ihre Implementierung. Es wurde ein Java-Programm erstellt, das einige Graphenklassen mittels LexBFS und seiner Varianten erkennen kann. Das Programm wird in Kapitel 5 vorgestellt und unter
www.rwev.de/web-content/LexBFSFrames.html
besteht die Möglichkeit zum Download.
Im Kapitel 2 werden zunächst die Grundlagen der weiteren Ausführungen gelegt. Dort werden Notationen, Begriffe aus der Graphentheorie und die betrachteten Graphenklassen vorgestellt. In Kapitel 3 wird detailliert auf LexBFS eingegangen. Der genaue Ablauf des Algorithmus, Implementierungsdetails und die daraus resultierende, lineare Laufzeit werden erläutert. Zusätzlich werden noch Varianten von LexBFS vorgestellt, die in den später vorgestellten Algorithmen zum Einsatz kommen. Außerdem wird eine Einordnung von LexBFS in eine größere Klasse von Graphsuchalgorithmen vorgenommen.
In Kapitel 4 werden die Anwendungen von LexBFS, die Erkennung bestimmter Graphenklassen und weitere Anwendungen beschrieben. Dabei werden die jeweiligen Algorithmen unter Berücksichtigung von Korrektheitsbeweisen (teils ausgeführt, teils auf die entsprechende Literatur hingewiesen), Laufzeiten und Implementierungsdetails betrachtet. Aufgrund des begrenzten Umfangs dieser Arbeit, wird der ausführliche Beweis der Korrektheit nur für chordale Graphen, Cographen und echte Intervall Graphen ausgeführt. Für die anderen Graphenklassen werden Skizzen der Beweise erläutert und auf die entsprechende Literatur verwiesen. Als Abschluss von Kapitel 4 werden noch die weiteren, über die Erkennung von Graphenklassen hinausgehende Anwendungen von LexBFS erläutert.

Kapitel 5 behandelt die, in Java erstellte, Implementierung der betrachteten Algorithmen. Es werden die verwendeten Datenstrukturen erläutert. Die Graphische Benutzeroberfläche (engl. Graphic User Interface = GUI) und der Funktionsumfang des Programms werden in einer Art Bedienungsanleitung erklärt.
Als Abschluss werden in Kapitel 6 die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf weitere Möglichkeiten der Suchalgorithmen in Graphen gegeben.
Da es fast ausschließlich englischsprachige Referenzliteratur gibt, existieren es für viele der verwendeten Begriffe keine, in der Graphentheorie übliche, deutsche Übersetzung. Zudem sind die englischen Begriffe oft sehr prägnant und besitzen nur unbefriedigende Übersetzungen ins Deutsche. Als Beispiel sei nur „tie-breaker“ erwähnt, dass sich mit „Unentschieden-Brecher“ sicherlich nicht gut übersetzen lässt. Daher werden meistens die englischen Originalbegriffe verwendet, auch wenn dies den Lesefluss etwas stört.
Der große Umfang dieser Arbeit, resultiert aus dem Bestreben die Algorithmen und Graphklassen anhand von Beispielen und Abbildungen zu verdeutlichen. Eine Verlagerung von diesen Teilen in den Anhang hätte das Lesen erschwert.

2 Vorüberlegungen

2.1 Notationen, Bezeichnungen und grundlegende Definitionen

Es werden ausschließlich ungerichtete Graphen betrachtet. Der Graph $G = (V, E)$ besteht aus der Menge seiner Knoten V und der Menge seiner Kanten E. Jede Kante $e \in E$ verbindet zwei verschiedene Knoten $v, w \in V$. Sind v und w durch eine Kante verbunden, also adjazent $v \in Adj(w)$ $w \in Adj(v)$, wird diese Schreibweise oftmals durch $vw \in E$ ersetzt. Es gibt keine Schleifen in G. Zwei Kanten $e, f \in E$ unterscheiden sich stets mindestens in einem ihrer Knoten, d.h. zwischen zwei Knoten gibt es maximal eine Kante. Die Anzahl der Knoten eines Graphen G wird stets mit n bezeichnet $n = |V|$. Die Anzahl der Kanten eines Graphen G wird stets mit m bezeichnet $m = |E|$.
Wird im weiteren Verlauf der Arbeit von linearer Laufzeit der Algorithmen gesprochen, so ist damit $\mathcal{O}(|V| + |E|) = \mathcal{O}(n + m)$ gemeint. Auf Ausnahmen wird ausdrücklich hingewiesen.

Definition 2.1 (Nummerierung der Knoten)
Eine Nummerierung oder auch Sortierung der Knoten ist eine Bijektion:

$$\sigma : 1, \ldots, n \to V = \{v_1, \ldots, v_n\}$$

und

$$\sigma^{-1}(V = \{v_1, \ldots, v_n\}) \to 1, \ldots, n$$

In den später betrachteten Algorithmen ist die Nummerierung ein Prozess, d. h. es wird sukzessiv nummeriert. Daher macht es Sinn, je nach der Richtung der Nummerierung, von vorwärts $1, \ldots, n$ oder rückwärts $n, \ldots, 1$ zu sprechen. Bezüglich der Knotenmenge werden die Begriffe „Nummerierung“, „Sortierung“ und Ordnung der Knoten im Text synonym verwandt. Ebenso werden die Begriffe „Suche“ bzw. „Sortieren der Knoten“ synonym verwandt. Der Durchlauf durch alle Knoten eines Graphen mit gleichzeitiger Nummerierung wird im weiteren Verlauf als Sweep bezeichnet. Um die Darstellung zu vereinfachen, wird oftmals für zwei Knoten $v, w \in V$ die Schreibweise $\sigma^{-1}(v) < \sigma^{-1}(w)$ durch $v <_\sigma w$ ersetzt. Wenn klar ist, welches σ gemeint ist, wird es auch einfach durch $v < w$ ersetzt. Unter dem ***Komplementärgraphen*** $\bar{G} = V, \bar{E}$ eines Graphen $G = (V, E)$ versteht man den Graphen, der genau die Kanten enthält, die in G nicht enthalten sind. $\forall x, y \in V : xy \in \bar{E} \Leftrightarrow xy \notin E$. Die Menge aller zum Knoten v adjazenten Knoten wird mit $N(v)$ bezeichnet. N für Nachbarschaft bzw. Neighbourhood. Man unterscheidet die ***offene Nachbarschaft*** $N(v)$ und die ***geschlossene Nachbarschaft*** $N[v] = N(v) \cup \{v\}$. Wenn nicht anders erwähnt, ist im Folgenden stets die offene Nachbarschaft gemeint. Im Hinblick auf die Nummerierung σ eines Graphen, gibt es die beiden wichtigen Teilmengen von $N(v)$: $N^+(v)$ und $N^-(v)$ mit
$N^+(v) = \{w \in N(v) \wedge \sigma^{-1}(w) > \sigma^{-1}(v)\}$
$N^-(v) = \{w \in N(v) \wedge \sigma^{-1}(w) < \sigma^{-1}(v)\}$

Je nach Richtung der Nummerierung ist demnach entweder $N^+(v)$ oder $N^-(v)$ die, zum Zeitpunkt der Nummerierung von v, ***nummerierte Nachbarschaft*** von v.
Ein ***induzierter Teilgraph*** $H = (V_H, E_H)$ eines Graphen $G = (V, E)$ ist ein Graph, der eine Teilmenge V_H der Knoten V als Knotenmenge besitzt und genau die Kanten aus G, deren beiden Endpunkte in V_H liegen. Es ist stets ein induzierter Teilgraph gemeint, wenn von einem Teilgraphen gesprochen wird. Auf Ausnahmen wird ausdrücklich hingewiesen. Von besonderer Bedeutung sind (induzierte) Teilgraphen, die sich aus der Nummerierung der Knoten ergeben.

Definition 2.2 (Durch Nummerierung festgelegte Teilgraphen)
Der durch die Nummerierung σ festgelegte (induzierte) Teilgraph $G_i = (V_H, E_H)$, ist der Teilgraph des Graphen G, der nur die Knoten $v \in V$ mit $\sigma^{-1}(v) \geq i$ enthält. Also: $V_H = \{v \in V : \sigma^{-1}(v) \geq i\}$.
Umgekehrt definiert man den Graphen $G_{\bar{i}} = (V_H, E_H)$ als den (induzierten) Teilgraphen von G, mit $V_H = \{v \in V : \sigma^{-1}(v) < i\}$.

Die Teilgraphen G_i bilden die Grundlage für so genannte ***Eliminationsordnungen*** $=$ ***EOs***, siehe Abschnitt 2.3.2. Die Teilgraphen $G_{\bar{i}}$ sind u. a. bei der Erkennung von Cographen von Bedeutung.
Eine ***Clique*** der Größe l, bezeichnet stets mit K_l, ist ein vollständiger Graph mit l Knoten und dementsprechend $\frac{l^2-l}{2}$ Kanten. Ein ***Weg(Way)*** der Länge n, bezeichnet stets mit W_n, ist eine Knotenfolge n verschiedener Knoten $\{v_1, \ldots, v_n\}$, mit paarweise verbunden Knotenpaaren, d.h. $\forall v_i, v_j \; i, j = 1, \ldots, n \wedge |i - j| = 1 : v_i \in N(v_j)$.
Ein ***Pfad (Path)*** der Länge n, bezeichnet stets mit P_n , ist ein Weg der Länge n, für den gilt: $\forall v_i, v_j \; i, j = 1, \ldots, n \wedge |i - j| \neq 1 : v_i \notin N(v_j)$. Ein Pfad ist ein Weg ohne Abkürzung. Ein ***Kreis (Circle)*** der Länge n, ist ein geschlossener Weg, d.h. die Anfangs- und Endknoten sind adjazent $v_1 v_n \in E$. Von Bedeutung sind vor allem sehnenlose Kreise, bezeichnet stets mit C_n, für diese gilt $v_1 v_n \in E$ und sonst gilt:

$$\forall i = 1, \ldots, n : \begin{cases} x_i x_j \in E & |i - j| = 1 \\ x_i x_j \notin E & |i - j| \neq 1 \end{cases}$$

Die ***Entfernung (Distance)*** zweier Knoten $v, w \in V$ in $G = (V, E)$, bezeichnet mit $d_G(v, w)$, ist die Länge des kürzesten Pfades zwischen v und w. Sollte dieser (bei nicht zusammenhängenden Graphen) nicht existieren, gilt: $d_G(v, w) = \infty$. Sollte aus dem Zusammenhang klar sein, welcher Graph der Entfernung zugrunde liegt, wird der Index G auch weggelassen. Die ***Exzentrizität*** $ecc(v)$ eines Knoten v ist die maximale Entfernung eines Knoten u von v in G, also: $ecc(v) = \underset{u \in V}{Max}(d(v, u))$.
Der ***Durchmesser(Diameter)*** eines Graphen G, bezeichnet mit D_G, bezeichnet den größten Abstand zweier Knoten $v, w \in V$ innerhalb des Graphen G, oder auch: $D_G = \underset{v \in V}{Max}(ecc(v))$. Bezeichnet man den zuerst nummerierten Knoten innerhalb einer Nummerierung σ mit s, dann wird die Teilmenge $U \subseteq V$: $\forall u \in U : d(u, s) = k; k = 1, \ldots, ecc(s)$ als ***k-ter Layer*** von G bezeichnet.
Mittels der Entfernungen innerhalb von G werden die ***Potenzen*** G^k mit $k \in \mathbb{N}$

von G definiert. Die Potenzen G^k eines Graphen G besitzen die Knotenmenge V und zwischen allen Knoten, deren Entfernung in G maximal k beträgt, existiert eine Kante in G^k. $G^k = (V, E_k)$ mit $E_k = \{vw \in V | d_G(v, w) \leq k\}$.
Ein ***Modul*** M eines Graphen bezeichnet eine Teilmenge $M \subseteq V$ der Knotenmenge, für die gilt, dass jeder Knoten außerhalb von M entweder zu jedem Knoten aus M oder zu keinem Knoten aus M adjazent ist. $\forall v, w \in M, \forall x \in V \setminus M : v \in N(x) \Leftrightarrow w \in N(x)$. Gilt: $M = V$, $M = \emptyset$ oder $|M| = 1$ spricht man von ***trivialen Modulen***, sonst von ***echten Modulen***. Man bezeichnet einen Graphen G als ***Prim-Graphen (prime Graph)***, wenn er nur triviale Module als Teilgraphen enthält.
Einen Knoten $v \in V$ mit $N(v) = 1$ nennt man ***hängenden Knoten***. Module, die zwei Knoten enthalten, nennt man ***Zwillinge***. Sind die beiden Knoten benachbart, ***echte Zwillinge***, ansonsten ***unechte Zwillinge***.
Eine Teilmenge S der Knotenmenge V eines zusammenhängenden Graphen $G = (V, E)$ bezeichnet man als ***Separator***, wenn der induzierte Teilgraph $H = (V_H, E_H)$ mit $V_H = \{V \setminus S\}$ nicht zusammenhängend ist.

Innerhalb eines Graphen spricht man von einem ***„asteroidal triple“***, wenn es drei Knoten $u, v, w \in V$ gibt, für die gilt: Es gibt zwischen allen drei Knoten paarweise Pfade, die die Nachbarschaft des dritten Knoten nicht benutzen. Gibt es innerhalb eines Graphen kein **asteroidal triple**, so spricht man von einem ***AT-freien Graphen***.
Ein Graph $G = (V, E)$ ist ein ***bipartiter Graph*** oder auch zweifärbbarer Graph, wenn sich die Knotenmenge V so in zwei disjunkte Teilmengen S und T aufteilen lässt, dass für die von S und T induzierten Teilgraphen G_S, G_T gilt: $G_S = (S, \emptyset), G_T = (T, \emptyset)$.
Einige Graphenklassen lassen sich mittels ihrer ***Dekomposition*** charakterisieren, daher werden nachfolgend zwei Arten von Dekompositionen vorgestellt.

2.2 Dekompositionsarten von Graphen

2.2.1 Modulare Dekomposition

Die modulare Dekomposition zerlegt den Graphen in seine Module. Dabei entsteht eine Baumdarstellung des Graphen, der der Satz 2.1 zugrunde liegt:

Satz 2.1 (Modulare Dekomposition) *zitiert nach [8]*
Gegeben sei ein Graph $G = (V, E)$ mit $|V| \geq 2$, dann gilt genau eine der folgenden Bedingungen:

1. *Der Graph G ist nicht zusammenhängend und kann in seine zusammenhängenden Komponenten zerlegt werden.* ***Parallele Zerlegung***

2. *Der Komplementärgraph $\bar{G}$ ist nicht zusammenhängend und G kann in die zusammenhängenden Komponenten von $\bar{G}$ zerlegt werden.* ***Serielle Zerlegung***

3. *G und $\bar{G}$ sind zusammenhängend. Dann gibt es mehrere Teilmengen $U \subseteq V$ und eine Partition P von V, so dass gilt:*
   a) *Jede Teilmenge U hat mindestens drei Knoten. $|U| > 3$.*
   b) *Die durch die Teilmengen U induzierten Teilgraphen $G(U)$ sind maximale prime Teilgraphen von G.*
   c) *Für jede Klasse S der Partition P gilt: S ist ein Modul und $|S \cap U| = 1$*

Jeder Knoten $v \in V$ bildet ein Blatt des Dekompositions-Baumes T von G. Für jedes Modul M von G wird ein Modul-Knoten k eingefügt. Die Blätter, die Nachfolger von k sind, sind genau die Knoten des Graphen aus M. Es gibt drei Arten von Modul-Knoten gemäß der drei Fälle aus Satz 2.1. Parallele Modul-Knoten werden mit einer 1 gekennzeichnet, Serielle Knoten mit einer 0 und Prim Knoten mit einem P. In Abb. 3 ist ein Beispielgraph und in Abb. 4 seine modulare Zerlegung dargestellt. Treffen sich die Pfade zweier Knoten v, w zur Wurzel des Baumes in einem mit einer 1 gekennzeichnetem parallelen Knoten, ist $vw \in E$. Treffen sich die Pfade in einem mit einer 0 gekennzeichneten seriellen Knoten, ist $vw \notin E$. Die Nachfolger eines Prim-Knoten sind gerade die Knoten eines maximalen primen Teilgraphen von G. Wenn ein Dekompositions-Baum eines Graphen G nur parallele und serielle Knoten enthält, so handelt es sich bei G um einen Cographen. Der modulare Dekompositionsbaum eines Cographen wird auch als Cotree bezeichnet, siehe 2.3.4.
Bisher gibt es einige Algorithmen zur modularen Dekomposition. Habib, de Montgolfier und Paul [45] haben 2004 einen Algorithmus mit linearer Zeitkomplexität zur modularen Dekomposition allgemeiner Graphen entwickelt. Corneil, Bretscher, Habib und Paul [9] vermuten, dass LexBFS auch auf diesem Gebiet von Nutzen sein kann.
Eine weitere Art der Dekomposition von Graphen, die von Cunningham [28] als eine Verallgemeinerung der modularen Dekomposition für gerichtete Graphen entwickelt wurde, ist die Split-Dekomposition.

2.2.2 Split-Dekomposition

Diese Art der Dekomposition ist für Graphen mit erblicher Distanz (engl. distance hereditary Graphs DH-Graphen), siehe Abschnitt 2.3.5, von Bedeutung. Sie wird durch so genannte „Splits“ definiert.

Definition 2.3 (Split)
Ein „Split“ eines Graphen $G = (V, E)$ ist eine disjunkte Aufteilung der Knotenmenge $V = \{V_1 \cup V_2\}$, für die gilt:

1. *$|V_1|, |V_2| \geq 2$*
2. *Bildet man für jeden einzelnen Knoten v aus V_i, die Schnittmenge $N^*(v)$ seiner Nachbarschaft mit der anderen Teilmenge V_j $N^*(v) = N(v) \cap V_j$. Dann bildet die Vereinigung sämtlicher Schnittmengen $\bigcup_{v \in V} N^*(v)$ einen vollständigen bipartiten Teilgraphen.*

Gibt es für einen Graphen einen solchen Split, nennt man ihn ***dekomponierbar bezüglich der Split-Dekomposition***, andernfalls diesbezüglich ***prim***. ***Triviale Splits*** sind solche, bei denen $|V_1| = 1 \vee |V_2| = 1$ gilt, dann ist G eine Clique oder ein ***Stern***. Ein Stern G_S ist ein Graph, der einen, zu allen anderen adjazenten Knoten besitzt. Alle übrigen Knoten sind hängende Knoten. Jeder Split für einen, durch einen Split entstandenen, Teilgraphen von G ist auch ein Split für G. So ergibt sich eine rekursive Dekomposition mittels Splits. An deren Ende stehen prime Teilgraphen oder solche, die nur noch triviale Splits zulassen. Ein Graph G ist vollständig dekomponierbar, wenn er keine primen Teilgraphen enthält, sondern nur Cliquen und Sterne. Diese Graphen sind gerade die Graphen mit erblicher Distanz aus Abschnitt 2.3.5. Die Baumstruktur, die durch die Split-Dekomposition definiert wird, erreicht man, indem man bei jedem Split für jede Teilmenge $V_i, i = 1, 2$ einen künstlichen (Dummy-) Knoten einfügt. Dieser besitzt zu allen Knoten aus $V_j, j = 1, 2 \neq i$ eine Kante. Diese beiden Dummy-Knoten werden ebenfalls verbunden.
Cunningham [28] hat gezeigt, dass jeder zusammenhängende Graph eine eindeutige Split-Darstellung, mit minimaler Anzahl an Split-Komponenten besitzt. Die Anzahl der möglichen Splits ist durch die Anzahl der Knoten nach oben begrenzt [28]. Dahlhaus [30] hat gezeigt, dass sich Split-Dekompositionen für allgemeine Graphen in Linearzeit berechnen lassen.

2.3 Graphenklassen

2.3.1 Perfekte Graphen

Eine Obermenge aller nachfolgend betrachteten Graphenklassen stellen die perfekten Graphen dar. Einen sehr guten Überblick zu diesem Thema liefert Golumbic [44]. Perfekte Graphen lassen sich u. a. folgendermaßen definieren.

Definition 2.4 (Perfekte Graphen)
Ein Graph $G = (V, E)$ wird als perfekt bezeichnet, wenn für alle Teilgraphen H von G gilt: die chromatische Zahl $\chi(H)$ ist gleich der Knotenanzahl der größten Clique $\omega(H)$ von H.

Perfekte Graphen lassen sich in polynomialer Zeit färben. Für allgemeine Graphen gilt das nicht. Die chromatische Zahlen chordaler Graphen, eine wichtige Teilmenge der perfekten Graphen, lassen sich sogar in linearer Zeit berechnen [44].

2.3.2 Chordale Graphen

Chordale Graphen oder auch triangulierte Graphen waren eine der ersten Graphenklassen, deren Zugehörigkeit zu den perfekten Graphen nachgewiesen wurde [44].

Definition 2.5 (Chordale Graphen)
Ein Graph G ist genau dann chordal, wenn er keinen sehnenlosen (chordless) Kreis

C_n mit $n \geq 4$ als Teilgraphen enthält. Eine Sehne innerhalb eines Kreises C_n bezeichnet eine Kante $uw \in E$ zwischen zwei, im Kreis nicht benachbarter Knoten u, w.

Definition 2.6 (simplizialer Knoten)
Ein Knoten v, dessen Nachbarschaft $N(v)$ eine Clique bildet also $N(v) = K_k, k \in \mathbb{N}$, nennt man einen simplizialen Knoten.

Simpliziale Knoten sind die Grundlage von perfekten Eliminationsordnungen.

Definition 2.7 (Perfekte Eliminationsordnung)
Ein PEO ist eine Nummerierung $\sigma^{-1}(V = \{v_1, \ldots, v_n\}) \rightarrow 1, \ldots, n$ der Knoten für die gilt:
$\forall v_i, i = 1, \ldots, n : N^+(v_i) = \{v_j \in N(v_i) | \sigma^{-1}(v_j) > \sigma^{-1}(v_i)\}$ bildet eine Clique. Das heißt, dass jeder Knoten v_i im Teilgraph G_i ein simplizialer Knoten ist.

Es gilt:

Satz 2.2 (Eigenschaften chordaler Graphen) *nach [41]*
Folgende Aussagen sind äquivalent:

1. *Ein Graph G ist chordal.*
2. *Ein Graph G besitzt eine „perfekte Eliminationsordnung“ (Perfect Elimination Order = PEO).*
3. *Jeder minimale, d.h. inklusionsfreie, Knoten Separator aus G ist ein vollständiger Teilgraph von G.*

Satz 2.3 (Anzahl simplizialer Knoten) *[35]*
Jede chordale Graph $G = (V, E)$ hat einen simplizialen Knoten. Wenn G kein vollständiger Graph ist, hat G mindestens zwei nicht adjazente simpliziale Knoten.

Der Graph aus Abb. 1 ist ein chordaler Graph. Ohne die Kante v_3v_1 wäre er nicht chordal; denn v_1, v_2, v_3, v_5 würden einen C_4 bilden. Chordalität ist eine erbliche Eigenschaft, d.h. ist G chordal, so sind es auch alle seine (induzierten) Teilgraphen. Der Begriff „perfektes Eliminationsschema“ bezieht sich auf den Zusammenhang zwischen Graphen und dem Gaußschen Eliminationsschema für lineare Gleichungssysteme. Anwendungen von chordalen Graphen liegen daher u. a. im Lösen von schwach besetzten, linearen Gleichungssystemen [43]. Zu den Potenzen von chordalen Graphen lassen sich folgende allgemeine Aussagen treffen:

Satz 2.4 (Potenzen von chordalen Graphen) *[1]*
Ist ein Graph G^k chordal, dann ist auch G^{k+2} chordal. Jede ungerade Potenz G^k mit $k = 2n-1 : n \in \mathbb{N}$ eines chordalen Graphen ist chordal, das gilt im Allgemeinen nicht für gerade Potenzen.

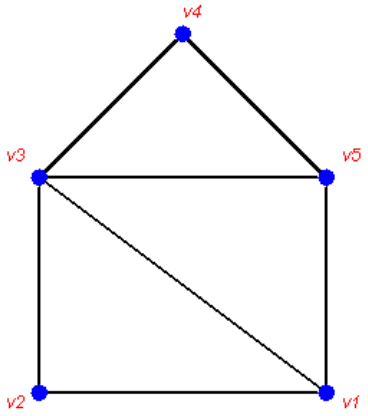

Abbildung 1: Ein chordaler Graph

2.3.3 Stark chordale Graphen

Eine wichtige Unterklasse der chordalen Graphen stellen die stark chordalen Graphen dar. Sie wurden erstmals von Farber [39] untersucht. Einige Probleme lassen sich auf stark chordalen Graphen in polynomialer Zeit lösen, die auf chordalen Graphen zu den NP-schweren Problemen zählen [31].

Definition 2.8 (stark chordale Graphen)
Ein Graph $G = (V, E)$ ist stark chordal, genau dann wenn er chordal ist und jeder Kreis gerader Länge von mindestens 6, eine „ungerade" Sehne besitzt. Eine „ungerade" Sehne ist eine Kante zwischen zwei Knoten $v, w \in C_m$, deren Abstand in C_m ungerade ist $d_{C_m}(v, w) = 2j + 1, j \in \mathbb{N}$.

Stark chordale Graphen lassen sich auch mit Hilfe von Eliminationsordnungen charakterisieren:

Definition 2.9 (Starke Eliminationsordnung = SEO)
Eine Knoten Nummerierung σ bezeichnet man als starke Eliminationsordnung (SEO = strong elimination ordering), wenn gilt:

1. *σ ist eine PEO*
2. *$\forall i, j, k, l \in V$ $i <_\sigma j <_\sigma k <_\sigma l$: $ik \in E \wedge il \in E \wedge jk \in E \Rightarrow jl \in E$*

Die zweite Eigenschaft wird auch als Γ-Freiheit der Adjazenzmatrix bezeichnet.
Es gilt folgender Satz:

Satz 2.5 *[39]*
Ein Graph ist stark chordal, wenn er eine starke Eliminationsordnung (SEO = strong elimination ordering) besitzt.

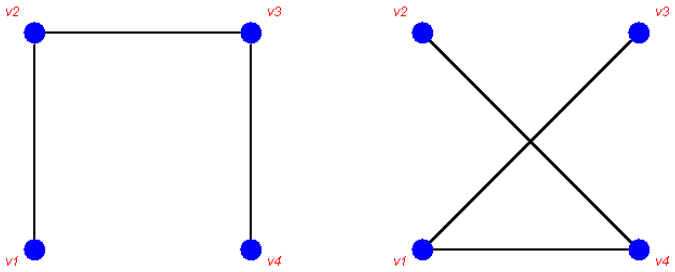

Abbildung 2: P_4 in G und in $\bar{G}$

2.3.4 Cographen

Komplement reduzierbare Graphen, üblicherweise nur noch Cographen genannt, wurden von mehreren Autoren u. a. [15] unabhängig voneinander in den 1970ern und frühen 1980ern entdeckt. Sie zeichnen sich durch mehrere Definitionen bzw. Charakterisierungen aus, deren Äquivalenz nicht unbedingt offensichtlich ist.

Definition 2.10 (Cographen)
Ein Graph G ist genau dann ein Cograph, wenn bei allen seinen induzierten Teilgraphen H, die mindestens zwei Knoten enthalten, entweder H oder $\bar{H}$ unzusammenhängend sind.

Die gängigste Charakterisierung von Cographen erfolgt, wie auch bei den chordalen Graphen, über einen verbotenen Teilgraphen.

Satz 2.6 (P_4 Freiheit von Cographen) *[67]*
Ein Graph G ist genau dann ein Cograph, wenn er keinen induzierten Pfad der Länge 4 oder größer enthält.

Daher bezeichnet man Cographen oft auch als P_4 - freie Graphen. Wie man anhand von Abb. 2 erkennt, ist ein induzierter $P_4 = \{v_1, v_2, v_3, v_4\}$ auch ein induzierter $P_4 = \{v_2, v_4, v_1, v_3\}$ im komplementären Graphen $\bar{G}$ mit vertauschten Mittel- und Endpunkten.
Ein P_4 ist gerade der kleinstmögliche, nicht triviale, prime Teilgraph (bezogen auf modulare Dekomposition) eines Graphen. Er lässt sich also nicht nur mittels paralleler und serieller Knoten darstellen.
Es gibt eine konstruktive Charakterisierung von Cographen:

Satz 2.7 (Konstruktion von Cographen) *[15]*
Jeder Cograph lässt sich mit Hilfe der der folgenden Punkte konstruieren:

1. *Jeder einzelne Knoten v bildet einen Cographen $G = (v, \oslash)$.*
2. *Wenn G ein Cograph ist, dann ist auch sein Komplement $\bar{G}$ ein Cograph.*

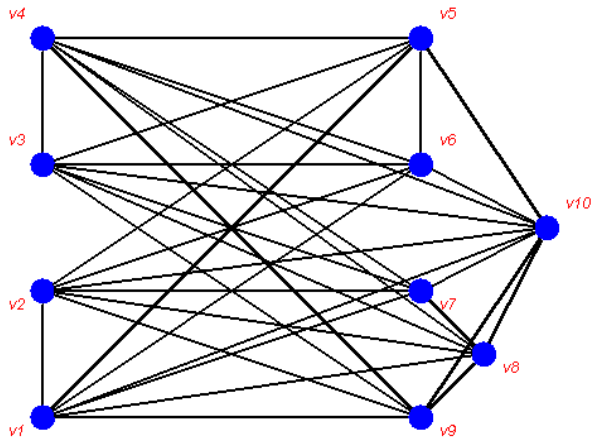

Abbildung 3: Ein Cograph

3. *Wenn* $G = (V_G, E_G)$ *und* $H = (V_H, E_H)$ *knotendisjunkte Cographen sind, dann ist auch ihre Vereinigung ein Cograph.*
   $G \cup H = (V_G \cup V_H, E_G \cup E_H \cup E_{G \cup H})$ *ein Cograph. Mit* $E_{G \cup H} = xy : x \in V_G, y \in V_H$

4. *Es gibt keine weiteren Cographen.*

In Abb. 3 ist ein Cograph dargestellt.
Der modulare Dekompositionsbaum eines Cographen wird als Cotree bezeichnet. Aufgrund der Konstruktionsvorschrift für Cographen aus 2.7 ist ersichtlich, dass Cotrees ausschließlich serielle und parallele Knoten enthalten können, vgl. Abb. 4.
Cotrees enthalten einen Wurzelknoten R und der Pfad zwischen einem Knoten x und der Wurzel R wird mit P_R^x bezeichnet. Die internen, sich abwechselnden Knoten auf diesem Pfad werden mit $0_1^x, 1_1^x, 0_2^x, 1_2^x$ bezeichnet. Der Laufindex erhöht sich in Richtung der Wurzel R. Jeder dieser inneren Knoten stellt wiederum die Wurzel eines Teilbaumes dar. Diese werden nach ihrer Wurzel mit $T_{01}^x, T_{11}^x, T_{02}^x, \ldots$ bezeichnet, vgl. Abb. 4.
$T_{01}^{v_1} = \{v_3, v_4\}$, $T_{11}^{v_1} = \{v_2\}$, $T_{12}^{v_1} = \{v_5, v_6, v_7, v_8, v_9, v_{10}\}$.
In späteren Ausführungen wird auch die Menge der äußeren Knoten aus den Teilbäumen T_{0i}^x oder auch T_{1j}^x mit T_{0i}^x bzw. T_{1j}^x bezeichnet. Aus dem Zusammenhang ergibt sich, ob der Teilbaum oder seine Knotenmenge gemeint ist.
Im Zusammenhang mit dem Algorithmus aus Abschnitt 4.2 werden noch folgende drei Eigenschaften von Cographen benutzt, zitiert nach [9]:

Satz 2.8 (Cotree komplementär)
Der Cotree $T_{\bar{G}}$ *des komplementären Graphen* $\bar{G}$ *ist genau der Cotree* T_G *von* G *mit vertauschten 0 und 1-Knoten.*

Satz 2.9 (Cograph Teilgraph)
Jeder induzierte Teilgraph eines Cographen ist wieder ein Cograph.

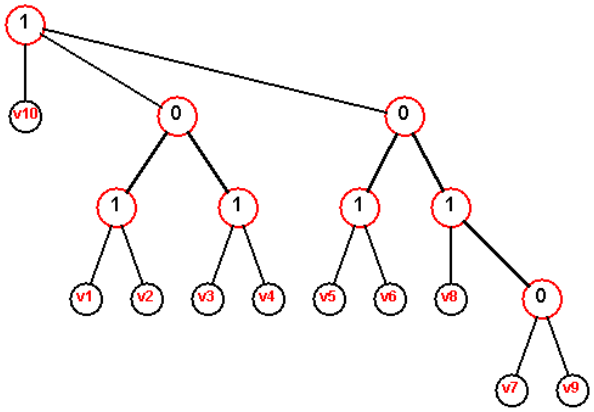

Abbildung 4: Der Cotree zum Cographen aus Abb. 3

Satz 2.10 (Cograph Teilbaum)
Jeder mit einer Wurzel ausgestattete Teilbaum $\acute{T}$ des Cotrees T_G stellt den Cotree, des durch die äußeren Knoten von $\acute{T}$ induzierten Teilgraphen, dar.

Die Darstellung eines Cographen als Cotree ist, bei abwechselnden inneren Knoten, bis auf Isomorphien eindeutig [15].

2.3.5 Graphen mit erblicher Distanz

Graphen mit erblicher Distanz (engl. distance hereditary Graphs), im weiteren stets mit DH-Graphen bezeichnet, sind eine weitere Teilmenge der perfekten Graphen. Sie lassen sich folgendermaßen definieren:

Definition 2.11 (DH-Graphen)
Ein Graph $G = (V, E)$ ist genau dann ein DH-Graph, wenn für alle Knoten $v, w \in V$ gilt: Sind v, w in einen induzierten Teilgraphen H von G in der gleichen Komponente, dann ist die Distanz zwischen v und w in H genauso groß wie in G, also: $d_H(v, w) = d_G(v, w)$.

Ebenso wie die bisher betrachteten Graphenklassen, lassen sich auch die DH-Graphen mittels verbotener Teilgraphen charakterisieren.

Satz 2.11 (Verbotene Teilgraphen von DH-Graphen) *nach [3]*
Ein Graph $G = (V, E)$ ist genau dann ein DH-Graph, wenn er keine der Graphen aus Abb. 5 als induzierte Teilgraphen enthält.

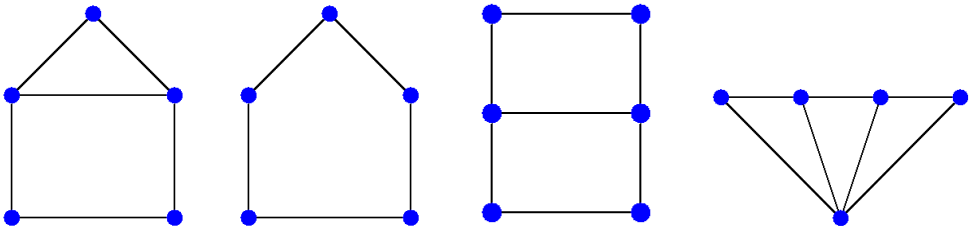

Abbildung 5: Haus, Loch, Domino und Edelstein

Diese induzierten Teilgraphen bezeichnet man (von links nach rechts) als Haus, Loch, Domino und Edelstein. Der abgebildete Graph „Loch“ steht stellvertretend für alle induzierten Teilgraphen, die einen Kreis ungerader Länge darstellen.
Auch DH-Graphen lassen sich mittels einer Konstruktionsvorschrift charakterisieren.

Satz 2.12 (Konstruktion von DH-Graphen) *nach [3]*
Ein Graph G ist genau dann ein DH-Graph, wenn er ausgehend von zwei adjazenten Knoten erzeugt werden kann, indem man sukzessiv einfache Knoten anfügt. Diese Knoten müssen vom Grad 1 (hängenden Knoten) sein oder nach Einfügung in einem echten oder unechten Zwillingspaar enthalten sein.

Es gelten folgende Zusammenhänge zwischen Cographen und DH-Graphen:

Satz 2.13 (Zusammenhang Cograph DH-Graph) *nach [3]*
Cographen stellen eine Untermenge der Graphen mit erblicher Distanz dar.

Beweis: Da jeder der Graphen aus Abb. 5 einen P_4 enthält, kann kein Cograph einen der Graphen als Teilgraphen enthalten und ist somit ein DH-Graph.

Satz 2.14 (Zusammenhang Cograph DH-Graph II) *nach [3]*
Sei G ein DH-Graph dann gilt: Für jeden Knoten $v \in V$ und $k \in \mathbb{N}$ gilt: Die Teilmenge U seiner Knotenmenge V mit $\{U \subset V | u \in U : d_G(u,v) = k\}$ induziert einen Cographen.

Definition 2.12 (2-simplizialer Knoten)
In einem Graphen G bezeichnet man einen Knoten $v \in V$ als 2-simplizialen Knoten, wenn die Teilmenge $\{U \subset V | u \in U : d_G(u,v) \leq 2\}$ einen Cographen induziert.

Dann kann, analog zu einer PEO für chordale Graphen, eine Eliminationsordnung definiert werden:

Definition 2.13 (2-simpliziale EO)
Eine 2-simpliziale EO ist eine Nummerierung
$\sigma^{-1}(V = \{v_1, \ldots, v_n\}) \rightarrow 1, \ldots, n$ der Knoten, für die gilt: Jeder Knoten v_i ist im Teilgraph G_i ein 2-simplizialer Knoten.

Es gilt nach [40]:

Satz 2.15 (2-simpliziale EO DH-Graph)
Ein Graph G ist genau dann ein DH-Graph, wenn er eine 2-simpliziale Eliminationsordnung besitzt.

2.3.6 Intervall-Graphen

Eine weitere Teilmenge der perfekten Graphen stellen die Intervall-Graphen dar. Anschaulich lassen sich diese folgendermaßen definieren:

Definition 2.14 (Intervall-Graphen)
Ein Graph $G = (V, E)$ ist genau dann ein Intervall-Graph, wenn man ihm folgendes Modell zugrunde legen kann:
Jedem Knoten v_i $i = 1, \ldots, n \in V$ entspricht ein Intervall I_i $i = 1, \ldots, n$ auf der reellen Zahlengerade.
Zwei Knoten $v_i v_j$ sind genau dann adjazent, wenn $I_i \cap I_j \neq \oslash$

Es besteht zwischen chordalen Graphen und Intervall-Graphen folgender Zusammenhang:

Satz 2.16 (Zusammenhang Intervall-Graphen chordale Graphen) *[60]*
Ein Graph ist genau dann ein Intervall-Graph, wenn er chordal ist und AT-frei ist.

Weiterhin gilt:

Satz 2.17 („Regenschirmfreiheit“) *[64]*
Ein Graph $G = (V, E)$ ist genau dann ein Intervall-Graph, wenn es eine lineare Ordnung $\prec$ auf der Menge der Knoten V gibt, so dass $\forall v, w, u \in V$ gilt:

$$u \prec v \wedge v \prec w \wedge uw \in E \Rightarrow uv \in E$$

Diese Bedingung wird in [24] als „Regenschirmfreiheit“ bezeichnet. Im Zusammenhang mit Cographen 4.2 wird auch von „Regenschirmfreiheit“ gesprochen, damit ist dort aber ein etwas anderer Sachverhalt gemeint. Daher wird im weiteren Verlauf, die Bedingung aus Satz 2.17 als ***IG-Regenschirmfreiheit*** bezeichnet.
Um im späteren Algorithmus testen zu können, ob eine Nummerierung Satz 2.17 erfüllt, benutzt man Satz 2.18.

Satz 2.18 (rechtseitige Nachbarschaftsbedingung) *nach [24]*
Eine Nummerierung σ der Knoten V eines Graphen erfüllt genau dann die Bedingung 2.17, wenn die rechtsseitigen Nachbarschaften $N_+(v)$ der Knoten $v \in V$ fortlaufend gemäß σ nummeriert sind.

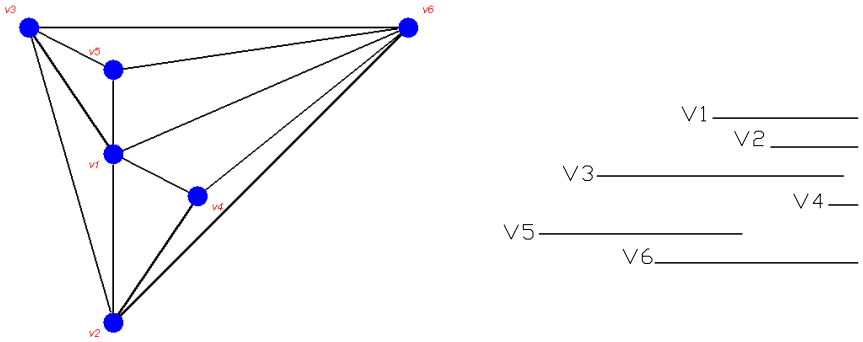

Abbildung 6: Ein Intervall-Graph und seine Intervall-Darstellung

Beweis: Angenommen es gibt drei Knoten $u, v, w \in V$ mit $u <_\sigma v <_\sigma w \wedge uw \in E \wedge uv \notin E$, dann ist $N_+(u)$ nicht fortlaufend nummeriert in σ. Andererseits sei $N_+(u)$ nicht fortlaufend nummeriert und x sei der am weitesten rechts stehende Nachbar von u, dann muss es einen Knoten t geben mit: $u <_\sigma t <_\sigma x \wedge ux \in E \wedge ut \notin E$ und u, t, x bilden einen IG-Regenschirm. □
Wenn eine Nummerierung σ Satz 2.17 erfüllt und damit gemäß Satz 2.18 die rechten Nachbarschaften fortlaufend sind, gilt folgender Satz:

Satz 2.19 (Intervall-Darstellung) *zitiert nach [24]*
Sei G ein Intervall-Graph und σ eine Satz 2.17 erfüllende Nummerierung, dann ist folgende Darstellung $I_G = \{I_1, \ldots, I_n\}$ eine gültige Intervall-Darstellung von G $\forall v \in V, \ldots, n : I_v = [\sigma^{-1}(v), \sigma^{-1}(N_r(v))]$ ($N_r(v)$ ist der rechteste Knoten von $N[v]$ in σ).

Beweis: Man betrachtet zwei Knoten $v_i, v_j \in V$ für die o. B. d. A. gilt: $v_i <_\sigma v_j$. Angenommen $v_i, v_j \in E$, aber es gilt $I_{v_i} \cap I_{v_j} = \emptyset$. Da aber auch $\sigma^{-1}(v_i) < \sigma^{-1}(v_j) \leq \sigma^{-1}(N_r(v_i))$ gilt. Widerspruch! Angenommen $v_i, v_j \notin E$, aber $I_{v_i} \cap I_{v_j} \neq \emptyset$. Dann muss $\sigma^{-1}(v_i) < \sigma^{-1}(v_j) \leq \sigma^{-1}(N_r(v_i))$ gelten und die rechtsseitige Nachbarschaft von v_i ist nicht fortlaufend nummeriert. Also erfüllt σ nicht Satz 2.17. Widerspruch! □
Abb.6 zeigt einen Intervall-Graphen und die dazugehörige Intervall-Darstellung. Die Intervalle wurden leicht ausgedehnt, um eine Überschneidung zu erhalten. In Tab. 1 ist eine Satz 2.17 erfüllende Nummerierung, nebst der sich daraus gemäß Satz 2.19 ergebenen Intervallgrenzen.

2.3.7 Echte Intervall-Graphen

Eine wichtige Teilmenge der Intervall-Graphen stellen die echten Intervall-Graphen (= Proper Interval Graph PIG) dar. Sie sind folgendermaßen definiert:

Definition 2.15 (Echter Intervall-Graph)
Ein Graph G ist genau dann ein echter Intervall-Graph, wenn er ein Intervall-Graph

i	$\sigma(i)$	**linke Int.-Grenze**	**rechte Int.-Grenze**	**linke PIG Int.-Grenze**	**rechte PIG Int.-Grenze**
1	v_5	1	4	1	4
2	v_3	2	5	2	$5,5$
3	v_6	3	6	3	$6,\bar{6}$
4	v_1	4	6	4	$6,75$
5	v_2	5	6	5	$6,8$
6	v_4	6	6	6	$6,8\bar{3}$

Tabelle 1: Intervall-Darstellung des Graphen aus Abb. 6

ist und es eine wie in 2.14 beschriebene Intervall-Darstellung $I = \{I_1, \ldots, I_n\}$ gibt, die inklusionsfrei ist, d.h. $\forall i, j\ I_i \not\subseteq I_j$

Einheits-Intervall-Graphen (= engl. Unit Interval Graph UIG) stellen eine weitere Teilmenge der Intervall-Graphen dar.

Definition 2.16 (Einheits-Intervall-Graph)
Ein Graph G ist genau dann ein Einheits-Intervall-Graph, wenn er ein Intervall-Graph ist und es eine wie in 2.14 beschriebene Intervall-Darstellung $I = \{I_1, \ldots, I_n\}$ gibt, in der alle I_i die gleiche Länge besitzen.

Die beiden Klassen der echten Intervall-Graphen und der Einheits-Intervall-Graphen sind gleich. D.h. ein Graph G ist genau dann ein PIG, wenn er ein UIG ist. Zur Äquivalenz gibt es einen kurzen Beweis von Bogart und West [5].
Für PIGs bzw. UIGs gibt es weitere Charakterisierungen.

Satz 2.20 (3-Knoten Bedingung) *nach [61]*
Ein Graph G ist genau dann ein UIG, wenn es eine lineare Ordnung $\prec$ auf der Menge der Knoten V gibt, so dass $\forall v, w, u \in V$ gilt: $u \prec v \wedge v \prec w \wedge uw \in E \Rightarrow uv \in E \wedge vw \in E$.

Satz 2.21 (Nachbarschaftsbedingung) *nach [65]*
Ein Graph ist genau dann ein UIG, wenn es eine lineare Ordnung $\prec$ auf V gibt, so dass gilt: $\forall v \in V : N(v)$ ist fortlaufend. d.h.: Seien $u = \underset{\prec}{Min}\ N(v)$ und $w = \underset{\prec}{Max}\ N(v)$, dann gilt: $\nexists z \in V : u \prec z \prec w \wedge z \notin N(v)$.

Es gilt:

Satz 2.22 *[65] [61]*
Die folgenden Aussagen sind für einen Graphen $G = (V, E)$ äquivalent:

1. *G ist ein PIG.*
2. *G ist ein UIG.*
3. *G erfüllt die Nachbarschaftsbedingung 2.21.*

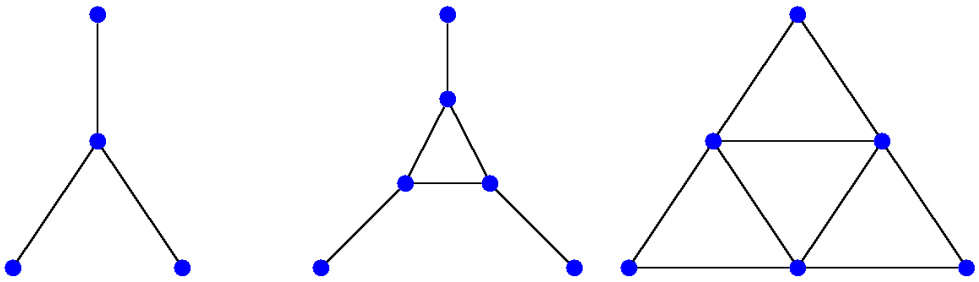

Abbildung 7: Klaue, Netz und Zelt

4. G erfüllt die 3-Knoten Bedingung 2.20.

Analog zu den anderen bereits erwähnten Graphenklassen, gibt es auch für PIGs eine Charakterisierung mittels verbotener Teilgraphen nach [75].

Satz 2.23 (verbotene Teilgraphen PIG) *[75]*
Ein Graph ist genau dann ein PIG, wenn er weder einen C_n $n \geq 4$ noch einen, der in Abb. 7 aufgeführten, Graphen als Teilgraphen enthält.

Wie in [34] gezeigt wurde, gibt es folgende Darstellungsart eines PIG:

Satz 2.24 (Intervall-Darstellung PIG) *[34]*
Sei ein Graph ein echter Intervall-Graph und seine Nummerierung σ erfüllt Satz 2.20, dann ist: I_i $i = 1, \ldots, n$ $I_i = [i, U(i) + 1 - \frac{1}{i}]$ mit $U(i) = \underset{w \in N[\sigma(i)]}{Max}\ \sigma^{-1}(w)$ eine Repräsentation des PIG.

Der Intervall-Graph aus Abb. 6 ist auch ein PIG. Die in Tab. 1 aufgeführte Nummerierung erfüllt auch Satz 2.20 und in Tab. 1 ist auch eine, mittels Satz 2.24 berechnete, Intervall-Darstellung aufgeführt.
Diese lässt sich, mittels eines von Gardi [42] vorgeschlagenen Algorithmus, in Linearzeit auch in eine UIG Darstellung transformieren.

2.3.8 P_4-reduzierbare Graphen

P_4-reduzierbare Graphen stellen eine Obermenge der Cographen dar. Sie lassen sich mittels verbotener Teilgraphen definieren.

Definition 2.17 (P_4-reduzierbare Graphen)
Ein Graph $G = (V, E)$ ist ein P_4-reduzierbarer Graph, wenn jeder Knoten $v \in V$ zu höchstens einem P_4 gehört.

Eine alternative Charakterisierung ergibt sich aus folgendem Satz:

Satz 2.25 *[51]*
Ein Graph G ist ein P_4-reduzierbarer Graph, wenn für jeden induzierten Teilgraphen $H = (V_H, E_H)$ genau eine der folgenden Bedingungen gilt:

1. *H ist nicht zusammenhängend.*
2. *$\bar{H}$ ist nicht zusammenhängend.*
3. *H und $\bar{H}$ sind zusammenhängend und es gibt einen einzigen P_4 bezeichnet mit $P = abcd$ in H, so dass gilt: $\forall v \in V_H \wedge v \notin P : vb \in E_H \wedge vc \in E_H \wedge va \notin E_H \wedge vd \notin E_H$.*

P_4-reduzierbare Graphen haben aufgrund von Satz 2.25 eine ähnliche Baumdarstellung wie Cotrees für Cographen [51].

2.3.9 P_4-spärliche Graphen

Eine Obermenge der P_4-reduzierbaren Graphen bilden die so genannten P_4-spärlichen Graphen (engl. P_4-sparse Graphs).

Definition 2.18 (P_4-spärliche Graphen)
Ein Graph $G = (V, E)$ ist ein P_4-spärlicher Graph (P_4 - sparse graph), wenn es keine induzierten Teilgraphen $H = (V_H, E_H)$ mit $|V_H| = 5$ von G gibt, die mindestens zwei verschiedene P_4 in G induzieren.

Auch für P_4-spärliche Graphen gibt es eine alternative Charakterisierung:

Satz 2.26 *nach [53]*
Ein Graph G ist ein P_4 -spärlicher Graph, wenn für jeden induzierten Teilgraphen $H = (V_H, E_H)$ genau eine der folgenden Bedingungen gilt:

1. *H ist nicht zusammenhängend.*
2. *$\bar{H}$ ist nicht zusammenhängend.*
3. *H und $\bar{H}$ sind zusammenhängend und es gibt eine disjunkte Aufteilung der Knoten von H in eine Clique K, eine unabhängige Menge I und die Restmenge R, so dass gilt:*
   a) *$|K| = |I| = n \geq 2$*
   b) *Es gibt eine Bijektion $f : K \rightarrow I$ so dass eine der beiden folgenden Bedingungen $\forall x \in I$ gilt:*
      i. *$N(x) = \{f(x)\}$ „dünne Spinne“*
      ii. *$N(x) = K \setminus \{f(x)\}$ „dicke Spinne“*
   c) *$\forall y \in R, z \in K : yz \in E_H$*

Auch P_4-spärliche Graphen haben aufgrund von 2.26 eine ähnliche Baumdarstellung wie Cotrees für Cographen [53].

2.3.10 Permutationsgraphen

Ähnlich den Intervall-Graphen 2.3.6 lassen sich auch Permutationsgraphen anschaulich definieren.

Definition 2.19 (Permutationsgraph)
Ein Graph $G = (V, E)$ ist ein Permutationsgraph, wenn man ihm folgendes Modell zugrunde legen kann: Seien L_1, L_2 zwei parallele Linien in einer Ebene. Legt man auf beiden Linien jeweils n verschiedene Punkte $P_{11}, \ldots, P_{1n}$ und $P_{21}, \ldots, P_{2n}$ beliebig fest und verbindet P_{1i} mit P_{2i} für $i = 1, \ldots, n$. Jede dieser Linien $P_{1i}P_{2i}$ wird einem Knoten v_i zugeordnet. Zwei Knoten v_i, v_j sind genau dann adjazent, wenn sich die Linien $P_{1i}P_{2i}$ und $P_{1j}P_{2j}$ schneiden.

Ohne diese geometrische Veranschaulichung kann man auch feststellen: Ein Graph $G = (V, E)$ ist ein Permutationsgraph, wenn es zwei Nummerierungen σ und τ gibt, so dass für zwei Knoten $v, w \in V$ gilt: v und w sind genau dann adjazent in G, wenn es innerhalb von σ und τ einen Fehlstand bezüglich v und w gibt.
Daher auch die Bezeichnung Permutationsgraph, denn τ ist eine Permutation der Nummerierung σ.

2.3.11 Bipartite Permutationsgraphen

Bipartite Permutationsgraphen bilden ebenfalls eine Teilmenge der perfekten Graphen. Ihre Menge ergibt sich als Schnittmenge zweier ebenfalls perfekter Graphenklassen, der bipartiten Graphen mit den Permutationsgraphen. Zu der Klasse der bipartiten Permutationsgraphen gibt es mehrere äquivalente Graphenklassen. U. a. ***echte Intervall-BiGraphen = UIBGs*** und ***bipartite AT-freie Graphen*** [48].

Definition 2.20 (echte Intervall-BiGraphen)
Ein Graph $G = (V, E)$ ist ein echter Intervall-BiGraph, wenn er eine bipartite Aufteilung $V = \{V_1, V_2\}$ seiner Knotenmenge besitzt und es außerdem eine Intervallfamilie I_i $i = 1, \ldots, n$ mit einander nicht enthaltenen Intervallen gibt, so dass gilt: Zwei Knoten $v_1 \in V_1$ und $v_2 \in V_2$ sind genau dann adjazent, wenn sich ihre Intervalle I_{v_1} und I_{v_2} überschneiden. Man beachte: Über die Überschneidung von Intervallen zweier Knoten $v_i v_j \in V_1$ oder V_2 wird damit nichts ausgesagt.

Aufgrund der Äquivalenz zu den echten Intervall-Bigraphen, gibt es für bipartite Permutationsgraphen die folgende, alternative Charakterisierung.

Satz 2.27 (verbotene Teilgraphen UIBG) *[49]*
Ein Graph G ist ein bipartiter Permutationsgraph, wenn er keinen der folgenden Graphen als Teilgraphen enthält:

1. *Einen sehnenlosen Kreis mit einer Länge von mindestens 6: C_n $n \geq 6$.*
2. *Einen Graph aus Abb. 8.*

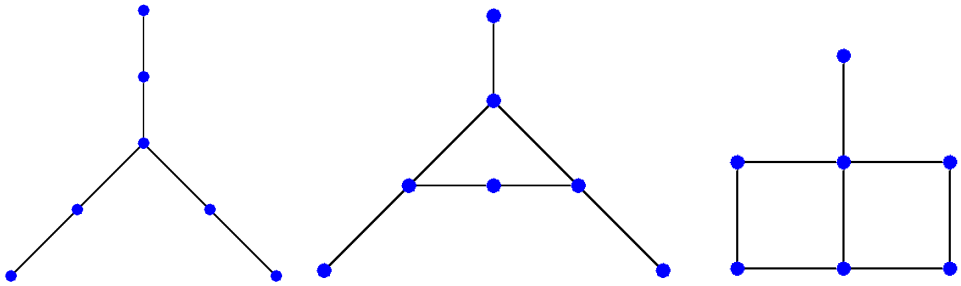

Abbildung 8: Bipartite(s) Klaue, Netz und Zelt

Ähnlich der 3-Knoten Bedingung 2.20 für UIGs gilt für UIBGs folgender Satz:

Satz 2.28 (schwache 3 Knoten Bedingung) *[48]*
Ein bipartiter Graph G ist ein UIBG, wenn es eine Nummerierung σ seiner Knoten gibt, so dass gilt: $\forall x, y, z \in V$ mit $x <_\sigma y <_\sigma z$: $xz \in E \Rightarrow xy \in E \vee yz \in E$

Mittels einer, diese schwache 3-Knoten Bedingung erfüllenden Nummerierung σ, lässt sich eine Intervall-Darstellung des UIBG erzeugen.

Satz 2.29 (Intervall-Darstellung echter Intervall-Bigraph) *[49]*
Sei G ein bipartiter Graph und σ seine Knotennummerierung, die Satz 2.28 erfüllt. Ordnet man jedem Knoten $v \in V$ das Intervall

$$I_i i = \sigma^{-1}(v) I_i = [i, R(i) + 1 - \frac{1}{i}]$$

zu, bildet die Familie $I_v \forall v \in V$ dieser Intervalle eine echte Intervall-Bigraph- Darstellung von G. Mit:

$$U(i) = \underset{w \in N[\sigma(i)]}{Max} \sigma(v)$$

$$\acute{U}(i) = \underset{w \in N(\sigma(i))}{Max} \sigma^{-1}(w)$$

$$R(i) = \begin{cases} U(\acute{U}(i)) \; wenn \; U(i) = i \wedge U(\acute{U}(i) > i) \\ U(i) \; sonst \end{cases}$$

Für bipartite Permutationsgraphen lassen sich einige Probleme in polynomialer Zeit lösen. Für deren Lösung sowohl für allgemeine Permutationsgraphen als auch für allgemeine bipartite Graphen die Komplexität unbekannt oder aber NP-vollständig ist [71].

3 Lexikographische Breitensuche (LexBFS) und andere Suchstrategien

3.1 Graph-Suchstrategien

Das Durchsuchen von Graphen, also das Finden bzw. Besuchen von sämtlichen Knoten und Kanten eines Graphen, ist von grundlegender Bedeutung. Üblicherweise wird eine Suche in einem Graphen durch die Reihenfolge der besuchten Knoten charakterisiert. Für einen Graphen mit n Knoten gibt es $n!$ Möglichkeiten die Knoten aufzulisten bzw. zu nummerieren. Unter diesen $n!$ Möglichkeiten stellen die Sortierungen, die durch eine Suchstrategie gefunden wurden, eine besondere Teilmenge dar. Eine Suche zeichnet sich durch einen bestimmten, aber zunächst willkürlichen Startknoten $s \in V$ aus. Von s aus werden alle weiteren Knoten besucht. Ein Knoten kann nur „gefunden“ werden, wenn bereits einer seiner Nachbarknoten besucht bzw. nummeriert wurde. Die Menge der besuchbaren, aber noch nicht besuchten Knoten wird im Folgenden mit S bezeichnet. Sollte der Graph nicht zusammenhängend sein, bedeutet dies, dass zunächst eine Komponente vollständig besucht wird. Anschließend wird aus den anderen Komponenten wiederum jeweils ein willkürlicher Startknoten ausgewählt. Allgemein lassen sich alle bekannten Graph-Suchstrategien als Spezialfälle des Algorithmus 1 darstellen.

Algorithmus 1 : Allgemeine Graphsuche

Eingabe : ein Graph $G = (V, E)$ und einen Startknoten $s \in V$
Ausgabe : eine Sortierung σ von V

```
1 {s} → S
2 für i = 1 bis n tue
3 |   nimm einen unnummerierten Knoten aus S
4 |   σ(i) = v
  |   /* Ordne v Nummer i zu.                    */
5 |   für alle unnummerierten Knoten aus N(v) tue
6 |   |   {w} → S
  |   |   /* Füge w in S ein                     */
```

Traditionell gibt es zwei Suchstrategien:

Breitensuche = BFS (Breadth First Search) und Tiefensuche = DFS (Depth First Search).
Breitensuche bedeutet, dass zunächst sämtliche Nachbarn $x \in N(v)$ eines Knoten v besucht werden, bevor weitere zu den Nachbarn benachbarte Knoten besucht werden. Anwendungen sind vor allem: kürzeste Wege-Probleme, Netzwerk-Flüsse und die Erkennung verschiedener Graphenklassen [22]. Tiefensuche bedeutet, dass ausgehend vom Startknoten stets ein unbesuchter Nachbar des zuletzt besuchten Knoten aufgesucht wird. Sollte es keinen unbesuchten Nachbarn mehr geben, wird so weit

in der Reihenfolge der besuchten Knoten zurückgesprungen, bis es wieder einen unbesuchten Nachbarn gibt. Anwendungen sind vor allem Planarität, topologische Sortierungen von Graphen und der Zusammenhang von gerichteten Graphen [22]. Im Hinblick auf den Algorithmus 1, heißt das, dass bei BFS die Menge S als Warteschlange (queue) realisiert werden kann, First In - First Out - Prinzip. Wohingegen bei DFS S als Stapel (stack) realisiert werden kann, Last In - First Out - Prinzip. Neben BFS und DFS wurden andere Suchstrategien entwickelt. Diese sind meistens Varianten von BFS und DFS. Alle Strategien unterscheiden sich in der mit (*) gekennzeichneten Zeile aus 1, also in der Auswahl des als nächstes aus S zu entnehmenden Knoten.

3.2 LexBFS

Lexikographische Breitensuche (engl. Lexicographic Breadth First Search = LexBFS) wurde erstmals von Rose, Tarjan und Lueker [66] vorgestellt. Sie haben LexBFS für die Erkennung chordaler Graphen „maßgeschneidert“. Bei BFS kann S aus Algorithmus 1 als einfache Warteschlange realisiert werden. Es wird stets der Knoten als nächstes nummeriert, der den frühesten nummerierten Nachbarn hat. Die neu gefundenen Knoten werden in willkürlicher Reihenfolge am Ende von S eingefügt und stets am Anfang von S wird der nächste zu nummerierende Knoten entnommen. Sind Knoten erstmal in die Warteschlange eingefügt, wird ihre Reihenfolge bei BFS nicht mehr verändert. Man kann sich BFS auch folgendermaßen realisiert vorstellen: Am Anfang bevor der erste Knoten nummeriert wird, erhalten alle Knoten $x \in V$ ein leeres Label $\forall x \in V : L(x) = \emptyset$. In diese Labels wird im Laufe der Nummerierung, die Nummer des am frühesten nummerierten Nachbarn eingetragen. Wenn erstmal eine Zahl eingetragen wurde in $L(x)$, ändert sich diese nicht mehr im Laufe der Nummerierung. Es wird stets einer, der noch nicht nummerierten Knoten als nächstes nummeriert, der das niedrigste Label besitzt. Sollte das Label bei mehreren Knoten gleich sein, wird willkürlich ein Knoten mit dem niedrigstem Label genommen.

Bei LexBFS werden nicht nur die frühest nummerierten Nachbarn betrachtet, sondern, sobald dieser bei zwei oder mehr noch nicht nummerierten Knoten gleich ist, wird der zweitfrüheste nummerierte benachbarte Knoten betrachtet usw.. Daher stammt auch der Name lexikographische Breitensuche. Man ordnet jedem Knoten $x \in V$ auch ein Label $L(x)$ zu, dieses erhält im Laufe der Nummerierung, solange x selber noch nicht nummeriert wurde, die Nummern aller nummerierten Nachbarn von x. Neue Nummern werden hinten im Label angefügt. In der ursprünglichen Form [66], wurden die Knoten von n bis 1 rückwärts nummeriert. Die Labels $L(x)$ $\forall x \in V$ sind absteigend geordnete Teilmengen von $\{1, 2, \ldots, n\}$ Beispielsweise: $\{6;5;4\}$,$\{9;8;7\}$ etc.. Lexikographische Ordnung bedeutet dabei, dass wie in einem Lexikon geordnet wird: z. B. gilt: $\{6;5;4\} <_{LEX} \{7;5;4\} <_{LEX} \{8;2\} <_{LEX} \{10\}$. Die Knoten werden geordnet, indem man Knoten mit gleichem Label in so genannten Sets zusammenfasst und diese Sets nach den Labels ihrer Knoten sortiert anordnet.

Einzelheiten siehe Algorithmus 2. Haben zwei oder mehr Knoten das gleiche Label, wird willkürlich zwischen ihnen entschieden. An dieser Stelle setzen Verfeinerungen von LexBFS an, siehe Abschnitt 3.4, die im Laufe der Jahre entwickelt wurden, um LexBFS auch für andere Anwendungen nutzbar zu machen.
Neben dieser „labelorientierten“ Herangehensweise, verwenden viele Autoren eine „labelfreie “ Betrachtung von LexBFS. Anfangs haben alle Knoten $v \in V$ ein leeres Label und sind daher in einem Set S. Beginnend mit dem Startknoten s verwendet man den gerade zu nummerierenden Knoten x als Pivot-Element. Man entfernt aus allen vorhandenen Sets S_i die Knoten, die zu x adjazent sind, und fügt sie in ein neues Set S_i^* ein. Das Set S_i^* wird direkt vor S_i in die Liste der Sets einsortiert. Um zu wissen, welches Set das höchste Label besitzt, ist es nicht notwendig die Labels explizit zu berechnen. Das Set mit dem höchsten Label steht vorne in der Liste der Sets.
Wie im nächsten Abschnitt gezeigt wird, lässt sich LexBFS einfach implementieren. Dabei hat es die gleiche, bestmögliche Laufzeit für das Sortieren von Graphen von $(O)(|V| + |E|)$ wie BFS.

3.2.1 LexBFS: der Algorithmus

Algorithmus 2 : Lexikographische Breitensuche

Eingabe : ein Graph $G = (V, E)$ und einen Startknoten $s \in V$
Ausgabe : eine LexBFS Sortierung σ von V

1 **für alle** $v \in V$ **tue**
2 $\quad L_v = \emptyset$
`/* Ordne zunächst allen Knoten ein leeres Label zu. */`
3 $\sigma(s) \rightarrow n$
`/* Ordne dem Startknoten s die Nummer n zu. */`
4 $\{s\} \rightarrow S$
5 **für alle** $v \in N(s)$ **tue**
6 $\quad L_v = \{s\}$
`/* Füge den Labels aller Nachbarn von s die Zahl n hinzu. */`
7 **für** $i = 1$ **bis** n **tue**
8 $\quad \sigma(w) \rightarrow n - i : L_w \geq L_x \forall x \in V \sigma(x) = \emptyset$
`/* Nimm einen beliebigen, noch nicht nummerierten Knoten` w `mit dem höchsten Label und ordne ihm die Zahl n-i zu. */`
9 $\quad \forall u \in N(w) : \sigma(u) = \emptyset \; L_u = L_u \cup \{n - i\}$
`/* Füge allen Labels, der zu w benachbarten und noch nicht nummerierten Knoten u die Zahl n-i hinzu. */`

Für die Labels $L(x)$ der Knoten innerhalb einer LexBFS- Nummerierung gelten Gesetzmäßigkeiten, die für die Implementierung von erheblicher Bedeutung sind.

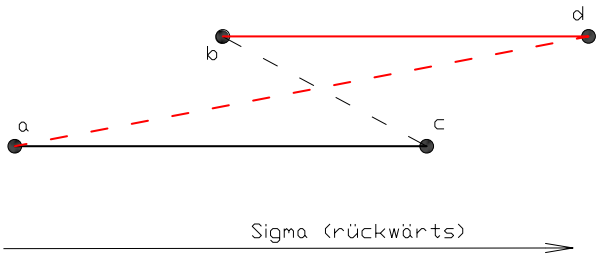

Abbildung 9: Bedingung 3 für LexBFS Nummerierungen

Satz 3.1 (Label Gesetzmäßigkeiten bei LexBFS) *nach [44]*
Sei $L_i(x)$ das Label des Knoten $x \in V$ nach der i-ten Nummerierung, also wenn ein Knoten $v \in V$ mit der Nummerierung i versehen wurde, gelten folgende Zusammenhänge (Rückwärtsnummerierung beachten):

1. $L_i(x) \geq L_j(x)\ j < i$ *Das Label eines Knotens wird nicht kleiner, neue Zahlen werden hinten angehängt.*

2. $L_i(x) > L_i(y) \Rightarrow L_j(x) > L_j(y)\ j < i x, y \in V$ *Liegt ein Knoten vor einem anderen (nach lexikographischen Ordnung), so ändert sich das nicht mehr im weiteren Verlauf des Algorithmus.*

3. $a, b, c \in V : \sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c) \wedge ac \in E \wedge bc \notin E$
   $\Rightarrow \exists d \in V : \sigma^{-1}(c) < \sigma^{-1}(d) \wedge db \in E \wedge da \notin E$
   Diese Bedingung ist in Abb. 9 veranschaulicht. Gäbe es Knoten d nicht, gilt stets $L(a) \geq L(b)$. Wenn c nummeriert wird, sogar $L(a) > L(b)$. Ein Widerspruch zu $\sigma^{-1}(a) < \sigma^{-1}(b)$.

3.2.2 Implementierungsdetails und Laufzeiten

Der Algorithmus 2 lässt sich in linearer Laufzeit realisieren. Das ist auf den ersten Blick nicht offensichtlich. Daher müssen Einzelheiten der Implementierung näher betrachtet werden. Dabei wird sich an Golumbic [44] angelehnt.

Datenstrukturen Man speichert Knoten mit gleichem Label in Sets, die als doppelt verkettete Listen implementiert werden. Zusätzlich erhalten die Sets noch einen Parameter, an dem abgelesen werden kann, ob schon ein unmittelbares Nachfolger-Set eingefügt wurde. Die Liste aller Sets wird ebenfalls als doppelt verkettete Liste implementiert. Die Nummerierung σ wird als Array gespeichert. Zwei weitere Arrays

speichern Pointer auf den genauen Ort jedes Knoten zum Zwecke des Zugriffs, des Löschens und des Einfügens. Ein Pointer zeigt auf den Knoten im Set, ein weiterer Pointer auf das jeweilige Set, in dem der Knoten aktuell enthalten ist. Zusätzlich wird noch eine doppelt verkettete Liste *FixList* zur Löschung leerer Sets benötigt. Aufgrund der unter Satz 3.1 über die Labels getroffenen Aussagen, kann auf die Berechnung und Speicherung der einzelnen Labels verzichtet werden. Es ist lediglich von Bedeutung welche Knoten das gleiche Label haben. Außerdem ist innerhalb der Liste der Sets eine aufsteigende Ordnung bezüglich der Labels einzuhalten. Nach dem „updaten" der Labels im Anschluss an die Nummerierung eines Knotens können Knoten nur in das unmittelbar nachfolgende Set aufsteigen. Gem. 3.1 Satz 2 ist ein „Überholen" von Knoten bzw. Sets nicht möglich. Man spricht zwar vom „updaten" der Labels, genau genommen wird aber nur die Liste der Sets „upgedatet" . Die Knoten müssen im richtigen Set und die Sets lexikographisch geordnet bleiben. In Pseudocode hat der Algorithmus LexBFS das in Algorithmus 3 dargestellte Aussehen.

Algorithmus 3 : Implementierung LexBFS

Eingabe : ein Graph $G = (V, E)$ und einen Startknoten $s \in V$
Ausgabe : eine LexBFS Sortierung σ von V

```
1 für i = 1 bis n tue
2 |   label(v_i) = nil
3 |   pos(v_i) = 0
  /* Initialisierung                                          */
4 für i = n bis 1 tue
5 |   select v_max with pos(v_max) = 0
  |   /* v_max = Knoten mit höchstem Label                    */
6 |   σ(v_max) = i
  |   /* Knoten wird Nummer i zugeordnet                      */
7 |   für j = 1 bis n tue
8 |   |   wenn pos(v_j) = 0 und v_max ∈ N(v_j) dann
9 |   |   |   update label(v_j)
```

Die Prozedur **update** ist in Algorithmus 4 aufgeführt.
Es ergibt sich für die Zeitkomplexität:
Das Auswählen eines Knoten v_{max} mit höchstem Label kann in konstanter Zeit $\mathcal{O}(1)$ erfolgen. Die erste for-Schleife aus **update** lässt sich für jeden Schleifendurchlauf in konstanter Zeit $\mathcal{O}(1)$ ausführen. Sie wird über den gesamten Algorithmus LexBFS $\sum_{v \in V} |N(v)|$-mal aufgerufen. Durch das Anhängen der neuen Sets, direkt nach den entsprechenden alten Sets, bleiben die Knoten in entsprechender Reihenfolge (aufsteigende Sortierung der Sets). Dabei werden die Bedingungen 1 und 2 aus Satz 3.1 benutzt und ein Sortieren der Sets entfällt. Die zweite for-Schleife aus **update** kann

Algorithmus 4 : Update LexBFS

Eingabe : Die Liste der Sets Q der gerade nummerierte Knoten v
Ausgabe : Die geänderte Liste Q der Sets

für alle $w \in (v)$ *und* $pos(w) = 0$ **tue**
  wenn $Flag(Set(w)) = 0$ **dann**
    /* In diesem Update Durchgang wurde noch kein direktes
       Nachfolge Set eingefügt */
    new (Set S^*)
    next(Set(w)) = S^* (*)
    /* mit Umhängen des vorher nachfolgenden Sets */
    $Flag(Set(w)) = 1$
    $Flag(S^*) = 0$
    /* Markierungen setzen */
    Setze Zeiger(Pointer) auf $Set(w)$ auf FixList
  entferne w aus Set(w)
  füge w S^* hinzu
  /* da $Flag(S) = 1$, existiert unmittelbar nachfolgendes Set schon */
  für alle *Sets S auf FixList* **tue**
    $Flag(S) = 0$
    wenn $S = \emptyset$ **dann**
      entferne S aus Q

ebenfalls in konstanter Zeit ausgeführt werden und wird auch maximal $\sum_{v \in V} |N(v)|$-mal aufgerufen. Das Einrichten der Datenstrukturen lässt sich in $\mathcal{O}(|V|)$ realisieren. Weiterhin gilt: $\mathcal{O}(\sum_{v \in V} N(v)|) = \mathcal{O}(|E|)$ da $\sum_{v \in V} |N(v)| = 2E$
So dass für LexBFS gilt:

Satz 3.2 (Komplexität LexBFS) *nach [44]*
LexBFS kann in Zeitkomplexität und mit Speicherplatz $\mathcal{O}(|V|+|E|)$ ausgeführt werden.

Das Entfernen leerer Sets am Schluss von **update** mittels **FixList** erfolgt aus Laufzeit- und Speicherplatzgründen.

3.3 Maximale Nachbarschaftssuche

LexBFS nummeriert immer einen Knoten v aus der Menge der nicht nummerierten Knoten V^*, dessen nummerierte Nachbarschaft $N^+(v)$ nicht echte Teilmenge einer nummerierten Nachbarschaft $N^+(w)$ eines anderen Knoten $w \in V^*$ ist.

$$\forall w \in V^* : N^+(v) \not\subset N^+(w) \tag{1}$$

LexBFS ist damit ein Spezialfall der Maximalen Nachbarschaftssuche (MNS). Zwei weitere Arten von MNS wurden bisher näher untersucht. „Maximum Cardinality Search“ (MCS) von Tarjan und Yannakakis [74] und lexikographische Tiefensuche (LexDFS) von Krueger und Corneil [22].

3.3.1 Maximum Cardinality Search

Bei der von Tarjan vorgestellten MCS wird der Knoten v als nächstes nummeriert, dessen nummerierte Nachbarschaft maximal bezüglich der Menge ist:

$$\forall w \in V^* : |N^+(w)| \leq |N^+(v)| \tag{2}$$

MCS lässt sich ebenfalls mit linearem Platz und Zeitbedarf $\mathcal{O}(|V|+|E|)$ implementieren. Die Labels der Knoten müssen bei MCS nur die Anzahl der nummerierten Nachbarn enthalten. Damit gelten die Bedingungen aus Satz 3.1 in abgewandelter Form.

Satz 3.3 (Bedingungen der MCS Labels)
Sei $L_i^{mcs}(x)$ das Label des Knotens $x \in V$ nach der $n-i$-ten Nummerierung, also wenn ein Knoten $v \in V$ mit der Nummer i versehen wurde. Es gelten folgende Zusammenhänge (Rückwärtsnummerierung beachten):

1. $\forall x \in V \wedge \ j < i : L_i^{mcs}(x) \geq L_j^{mcs}(x)$ *Das Label eines Knotens wird nicht kleiner, das Label wird nur erhöht.*

2. $\forall x, y \in V \wedge \ j = i-1 : L_i^{mcs}(x) > L_i^{mcs}(y) \Rightarrow L_j^{mcs}(x) \geq L_j^{mcs}(y)$ *Liegt ein Knoten vor einem anderen (nach MCS Ordnung), so kann ein Knoten in einem Schritt nur eingeholt (nicht überholt) werden.*

3. $\forall a, b, c \in V : \sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c) \wedge ac \in E \wedge bc \notin E$
   $\Rightarrow \exists d \in V : \sigma^{-1}(b) < \sigma^{-1}(d) \wedge db \in E \wedge da \notin E$ *Gäbe es diesen Knoten d nicht, so wäre das Label von a nach Nummerierung von c in jedem Fall höher als das Label von b.*

Auch MCS implementiert man „labelfrei“. Gegenüber der Implementierung von LexBFS muss lediglich der **update** Algorithmus verändert werden. Er hat folgendes Aussehen:

Algorithmus 5 : Update MCS

Eingabe : Die Liste der Sets Q der gerade nummerierte Knoten v
Ausgabe : Die geänderte Liste Q der Sets

```
1 für alle w ∈ (v) und pos(w) = 0 tue
2   wenn next(Set(w)) = nil dann
      /* es ist bereits das Set mit dem höchsten Label          */
3     new (Set S*)
4     next(Set(w)) = S*
5   entferne w aus Set(w)
6   füge w zu S* hinzu
  /* jetzt existiert unmittelbar nachfolgendes Set in jedem Fall
     schon                                                      */
```

Aus den Überlegungen zu LexBFS ergibt sich, dass auch MCS in linearer Zeit ausgeführt werden kann. Das Löschen der leeren Sets am Schluss von **update** entfällt. Die leeren Sets werden als Platzhalter benötigt. Außerdem können nur maximal $n - 1$ verschiedene Sets entstehen. In Tabelle 3.4.8 ist für Graphen verschiedener Größen die Laufzeit von LexBFS, MCS und LexDFS aufgeführt. Es hat sich gezeigt, dass trotz der gleichen Laufzeit von MCS und LexBFS in der O-Notation, sich die Laufzeiten doch erheblich unterscheiden.

Satz 3.4 (Komplexität MCS) *[73]*
MCS kann in Zeitkomplexität und mit Speicherplatz $\mathcal{O}(|V|+|E|)$ ausgeführt werden.

Eine Variante von MCS könnte in einer Verschmelzung von MCS und LexBFS liegen. Bei Gleichstand der MCS-Labels, könnte man das LexBFS Kriterium als „tiebreaker“ einsetzen. Diese Idee soll aber hier nicht weiter betrachtet werden.

3.3.2 Lexikographische Tiefensuche

LexBFS stellt einen Spezialfall der Breitensuche dar. Es liegt nahe zu fragen, ob es einen solchen Spezialfall auch für die Tiefensuche gibt. Corneil und Krüger haben diese Lexikographische Tiefensuche LexDFS 2005 vorgestellt [22]. Bei BFS werden die Knoten als nächstes nummeriert, die die frühest besuchten Nachbarn besitzen.

Bei DFS werden die Knoten als nächstes besucht, die die am spätest besuchten Knoten als Nachbarn haben. DFS ließe sich also mittels Labels folgendermaßen erklären: Jedem unnummerierten Knoten $x \in V$ wird in seinem Label $L(x)$ die Nummer, seines zuletzt nummerierten Nachbarn, z. B. $w \in N(x)$ zugeordnet $L(x) = \sigma^{-1}(w)$. Es wird stets der Knoten als nächstes nummeriert, der das niedrigste Label (bei Rückwärts-Nummerierung) oder das höchste (bei Vorwärts-Nummerierung) aufweist. Bei Gleichstand würde analog zu BFS willkürlich entschieden. Bei LexDFS wird das Label, wie bei LexBFS, aus den Nummern aller bereits besuchten Nachbarn gebildet. Bei Vorwärts-Nummerierung wird die Nummer des gerade nummerierten Nachbarn vorne ans Label angefügt. Die Nummern innerhalb der Labels sind also absteigend sortiert. Es wird stets der unnummerierte Knoten ausgewählt, der das höchste Label besitzt. Die Richtung der Nummerierung, vorwärts oder rückwärts, spielt nur im Zusammenhang mit der Ordnung der Labels eine Rolle. Durch Spiegelung der Nummerierung, d.h. $\forall v \in V : \sigma_{for}(v) = i \Leftrightarrow \sigma_{back}(v) = n - i$, erhält man auch durch LexDFS eine Rückwärts Sortierung, umgekehrt gilt das auch für LexBFS und MCS. Der Algorithmus LexDFS hat prinzipiell das gleiche Aussehen wie Algorithmus 2.
Bei der Implementierung liegt der entscheidende Unterschied zwischen LexBFS und LexDFS auch im Algorithmus **update**. Der Rumpfalgorithmus aus Algorithmus 3 kann verwendet werden, mit dem Unterschied der umgekehrten Nummerierung. Allerdings ist bei LexDFS die Auswahl eines Knoten mit höchstem Label v_{max} nicht in Linearzeit möglich, bzw. ist bisher keine solche Implementierung bekannt. Das Problem liegt in der zweiten Bedingung für die Labels aus Satz 3.1. Diese Bedingung gilt nicht für LexDFS. Ein Knoten kann mit einem gerade nummerierten Nachbarn alle anderen Knoten überholen.
Die Einordnung, der zu dem gerade nummerierten Knoten benachbarten, unnummerierten Knoten, ist für die Laufzeit von $update_{LexDFS}$ von entscheidender Bedeutung.
Bei LexBFS und MCS wurden die benachbarten Knoten stets in das nächste, evtl. neu zu schaffende, Set S^* eingefügt. Bei LexDFS rückt jeder Nachbar eines gerade nummerierten Knotens ganz nach vorn in der Setliste. Sollte nur ein noch nicht nummerierter Knoten zum gerade nummerierten Knoten adjazent sein oder alle diese Knoten zum gleichen Set gehören, so ist das neue Set vorne einzufügen. Sollten aber die innerhalb eines **update** Aufrufs umzusetzenden Knoten zu verschiedenen Sets gehören, lässt sich nicht einfach bestimmen, in welcher Reihenfolge die neu entstehenden Sets vorne in die Setliste einzufügen sind. Die zu dem gerade nummerierten Knoten v adjazenten Knoten liegen nicht in nach Labels sortierter Reihenfolge vor und müssten erst sortiert werden. Das kann beispielsweise dadurch erreicht werden, dass man wie bei LexBFS jedes neu entstandene Set S^* direkt nach dessen Ursprungsset S einfügt. Kennzeichnet man alle neuen Sets mit einer Markierung (Flag = 2), so können alle neuen Sets in einem Setlisten Durchlauf eingesammelt werden. Das Einsammeln beinhaltet das Löschen aus der ursprünglichen Setliste und das Einfügen in eine neue Setliste. In dieser sind die neu entstandenen Sets in lexikographisch geordneter Reihenfolge. Daher muss diese Setliste nur noch an die

ursprüngliche vorne angefügt werden. Krueger schlägt diese und eine zweite Variante vor [57]. Beide haben aber keine lineare Laufzeit. Diese beschriebene und von Krueger als ***split + collect*** bezeichnete Variante, hat eine quadratische Laufzeit $\mathcal{O}(n^2)$ und wurde im erstellten Programm implementiert. Der Algorithmus $update_{LexDFS}$ ist in Algorithmus 6 aufgeführt. Gegenüber $update_{LexBFS}$ fällt der zusätzliche Scan der Setliste an. Dieser muss in Reihenfolge der Setliste erfolgen, um die richtige Reihenfolge der neuen Sets beizubehalten. Der Scan kann daher nicht durch Pointer auf die einzelnen Sets erfolgen. Da für jeden der n Durchgänge von **update** ein Scan sämtlicher Sets erfolgen muss und die Anzahl der jeweils aktuell vorhandenen Sets eine Größenordnung von $\mathcal{O}(n)$ hat, ist die Komplexität aller Scans $\mathcal{O}(n^2)$. Damit ergibt sich unter Berücksichtigung, der für LexBFS gewonnenen Laufzeiterkenntnisse und $\mathcal{O}(m + n + n^2) = \mathcal{O}(n^2)$.

Satz 3.5 (Laufzeit LexDFS) *nach [57]*
LexDFS kann mit einer Laufzeit von $\mathcal{O}(n^2)$ ausgeführt werden.

Algorithmus 6 : Update LexDFS

Eingabe : Die Liste der Sets Q der gerade nummerierte Knoten v
Ausgabe : Die geänderte Liste Q der Sets

für alle $w \in (v)$ *und* $pos(w) = 0$ **tue**
  wenn $Flag(Set(w)) = 0$ **dann**
    `/* In diesem Update Durchgang wurde noch kein direktes`
    `   Nachfolge Set eingefügt */`
    $new(SetS^*)$
    $next(Set(w)) = S^*$
    `/* mit Umhängen des vorher nachfolgenden Sets */`
    $Flag(S^*) = 2$
    $Flag(Set(w)) = 1$
    `/* Markierungen setzen */`
  entferne w aus $Set(w)$
  füge w S^* hinzu
`/* da` $Flag(S) = 1$`, existiert unmittelbar nachfolgendes Set schon */`
für alle *Sets S mit* $Flag(S) = 2$ **tue**
  füge S in NewList ein
  entferne S aus Q
  $Flag(S) = 0$
für alle *Sets S aus Q* **tue**
  wenn $S = \emptyset$ **dann**
    entferne S aus Q
füge NewList am Anfang von Q ein

Ein Beispiel mit den entsprechenden Labels ist in Tabellen 4 - 6 für LexBFS, MCS und LexDFS aufgeführt. Im erstellten Programm werden allerdings sämtliche Nummerierungen mit den beschriebenen labelfreien Implementierungen berechnet.
In Tabelle 10 sind beispielhaft für den Graphen aus Abb. 12 die Nummerierungen der verschiedenen Nummerierungsarten aufgeführt.
Wie Corneil und Krueger [22] dargelegt haben, lassen sich alle Sortierungen durch folgende Situation eindeutig charakterisieren, siehe Abb. 10. Sei folgende Konstellation innerhalb einer (Vorwärts-) Sortierung σ von V gegeben:

$$a <_\sigma b <_\sigma c \wedge ab \notin E \wedge ac \in E \tag{3}$$

Dann muss für σ gelten: Bei allgemeiner Graph-Suche AGS:

$$\exists d \in V : d <_\sigma b \wedge db \in E \tag{4}$$

Bei Breitensuche BFS:

$$\exists d \in V : d <_\sigma a \wedge db \in E \tag{5}$$

Bei Tiefensuche DFS:

$$\exists d \in V : a <_\sigma d <_\sigma b \wedge db \in E \tag{6}$$

Bei lexikographischer Breitensuche LexBFS:

$$\exists d \in V : d <_\sigma a \wedge db \in E \wedge dc \notin E \tag{7}$$

Bei lexikographischer Tiefensuche LexDFS:

$$\exists d \in V : a <_\sigma d <_\sigma b \wedge db \in E \wedge dc \notin E \tag{8}$$

Bei Maximum Cardinality Search MCS und allgemeiner Maximum Neighbourhood Search MNS:

$$\exists d \in V : d <_\sigma b \wedge db \in E \wedge dc \notin E \tag{9}$$

damit σ eine, nach der jeweiligen Suchstrategie gültige Sortierung von V ist. Zu den jeweiligen Beweisen vgl. [57]. Sollte σ keine Konstellation 3 aufweisen, ist σ eine nach allen genannten Strategien gültige Nummerierung, z. B. eine an einem der beiden Randknoten beginnenden Nummerierung eines P_n.
Aus (4) - (9) lassen sich folgende Mengenbeziehungen von gültigen Sortierungen folgern: Bezeichne z.B. BFS_G die Menge aller BFS Sortierungen eines Graphen G, MAS_G = Menge aller Sortierungen (d.h. aller Permutationen der Knoten) von G

$$MAS_G \supseteq AGS_G \supseteq BFS_G \supseteq LexBFS_G \tag{10}$$

$$MAS_G \supseteq AGS_G \supseteq DFS_G \supseteq LexDFS_G \tag{11}$$

$$MAS_G \supseteq AGS_G \supseteq MNS_G \supseteq MCS_G \tag{12}$$

$$MNS_G \supseteq LexBFS_G \tag{13}$$

$$MNS_G \supseteq LexDFS_G \tag{14}$$

Zusammenhänge zwischen den einzelnen Suchstrategien sind Abb. 11 zu entnehmen, die aus [22] stammt. LexBFS ist bisher, abgesehen von den traditionellen Strategien BFS und DFS, am genauesten untersucht worden.

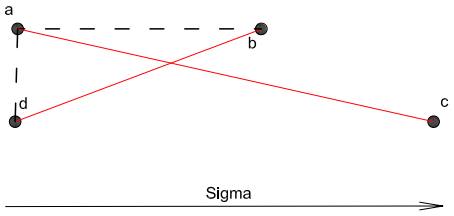

Abbildung 10: Die Situation zur Charakterisierung der Suchstrategien

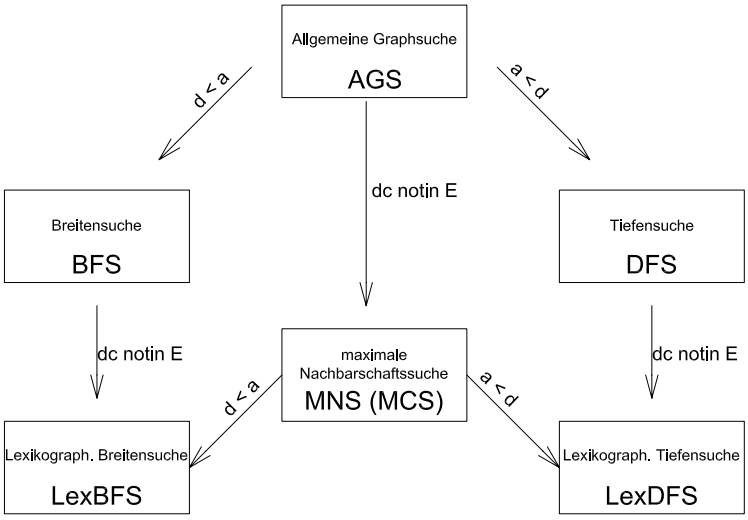

Abbildung 11: Zusammenhänge der verschiedenen Suchstrategien

3.4 Varianten von LexBFS

Zur Erkennung chordaler Graphen wurden die Knoten rückwärts von n bis 1 nummeriert und aus der Menge von Knoten mit dem größten Labels wurde willkürlich ein Knoten ausgewählt. Für andere Anwendungen wurden Varianten dieses ursprünglichen Algorithmus entwickelt.

3.4.1 LexBFS-vorwärts

Für die meisten folgenden Anwendungen ist es nötig oder zumindest günstiger, die Knoten in aufsteigender Folge zu nummerieren. Einem beliebigen Startknoten s die 1 zuzuordnen und mit aufsteigenden Nummern 2 bis n fortzufahren. Auch bei LexBFS-vorwärts werden die Knoten zuerst nummeriert, die die frühest besuchten Nachbarn haben. Da aber die Knoten vorwärts nummeriert werden, sind deren Labels nicht die lexikographisch höchsten. Der Algorithmus ist identisch bis auf die Zuordnung der Nummern und hat dementsprechend auch die gleiche Laufzeitkomplexität von $\mathcal{O}(|V| + |E|)$.

3.4.2 LexBFS$^+$

Bisher wurde ein willkürlich Knoten aus dem Set S_{MAX} mit dem höchsten Label entnommen und nummeriert. Daher kann es für einen Graphen G verschiedene gültige LexBFS Nummerierungen geben. Von Simon [69] stammt die Idee, einen vorgeschalteten LexBFS-vorwärts Sweep als „tie-breaker“ zu verwenden. Bei LexBFS$^+$ wird aus S_{MAX} stets der Knoten ausgewählt, der die höchste Vornummerierung (der zuletzt nummerierte Knoten) in σ hat. Damit ist eine LexBFS$^+$ Nummerierung bei gegebener Vornummerierung eindeutig. Der letzte Knoten aus σ ist stets der erste in $\sigma+$, da am Anfang alle Knoten leere Labels haben. Welche Laufzeit hat LexBFS$^+$? Im LexBFS$^+$ Sweep ist stets der Knoten mit höchster Vorsortierung zu nehmen. Wenn jedes Mal S_{MAX} nach dem Knoten mit höchster Vorsortierung durchsucht werden muss, ist keine lineare Laufzeit möglich. Liegen allerdings die Knoten innerhalb der Sets in geordneter Reihenfolge vor, kann diese Suche in S_{MAX} entfallen. Dazu müssten die Adjazenzlisten aller Knoten vorsortiert, d.h. hier in absteigender Nummerierung, vorliegen. Dann werden stets die Knoten in richtiger Reihenfolge in neu entstehende Sets eingefügt und dementsprechend entnommen. Die Adjazenzlisten jedes Knotens lassen sich gemäß der Nummerierung des ersten LexBFS Sweeps in Linearzeit sortieren. Dazu verwendet man den Algorithmus 7. Die Knoten sind vorher bereits in Linearzeit absteigend sortiert worden. Das ist möglich, da auf sie mittels Pointer in konstanter Zeit zugegriffen werden kann.
Dieses Verfahren aus Algorithmus 7 hat eine Laufzeit von $\mathcal{O}(|V| + |E|)$, da jede Kante zweimal verwendet wird und n Listen neu erzeugt und angehängt werden müssen.
Für spätere Ausführungen ist folgender Satz 3.6 von Bedeutung:

Satz 3.6 (LexBFS$^+$ Satz) *nach [48]*
Sei σ^+ eine mittels LexBFS$^+$ erzeugte Nummerierung eines Graphen $G = (V, E)$,

Algorithmus 7 : Sortierung Linear

Eingabe : ein Graph G mit sortierter Knotenliste, aber unsortierten Adjazenzlisten

Ausgabe : G mit sortierten Adjazenzlisten

```
1 für i = 1 bis n tue
2   newAdjList(v_i) = nil
  /* Initialisierung der neuen Adjazenzlisten                    */
3 für i = n bis 1 tue
4   nimm nächsten Knoten v
    /* Knoten sind bereits sortiert                              */
5   für j = 1 bis |N(v)| tue
6     für alle w ∈ N(v) tue
7       Füge v in NewAdjList(w) ein
8 für i = 1 bis n tue
9   AdjList(v_i) = newAdjList(v_i)
  /* Ersetzen der Adjazenzlisten                                 */
```

dann gilt: Gibt es $v, w \in V$ *mit* $\sigma^{-1}(v) < \sigma^{-1}(w) \wedge \sigma_+^{-1}(v) < \sigma_+^{-1}(w) \Rightarrow \exists u \in V$: $\sigma_+^{-1}(u) < \sigma_+^{-1}(v) \wedge uv \in E \wedge uw \notin E$

Beweis: Gäbe es Knoten u nicht, kann Knoten v kein größeres Label als w haben. Das aber widerspricht der Tatsache, dass v vor w in σ^+ nummeriert wird, obwohl es in bereits in σ vorher nummeriert wurde. □
Folgender Satz 3.7 hat bei der Erkennung von UIGs große Bedeutung:

Satz 3.7 (Startknoten LexBFS$^+$)
Ein Knoten $v \in V$ *kann nur dann Startknoten eines LexBFS*$^+$ *Nummerierung sein, wenn gilt:* $\exists w \in V : ecc(w) = d(v, w)$.

Beweis: Sei u der Startknoten des LexBFS-Sweep, der dem LexBFS$^+$-Sweep zugrunde liegt. $v = \sigma_+(1)$ kann nur gelten, wenn $v = \sigma(n)$ ist und somit $ecc(u) = d(u, v)$. □

3.4.3 LexBFS$^-$

LexBFS$^-$ benutzt ebenso wie LexBFS$^+$ eine Vorsortierung σ durch einen LexBFS-vorwärts Sweep. Der LexBFS$^-$ Sweep wird auf dem Komplementärgraphen $\bar{G}$ von G ausgeführt. Dabei wird bei mehreren Knoten in S_{MAX} der Knoten genommen, der in σ als erstes nummeriert wurde. LexBFS$^-$ wurde zuerst in [11] verwandt und findet Anwendung in der Erkennung von Cographen, DH-Graphen, P_4-reduzierbaren und P_4-spärlichen Graphen. Bei der Laufzeituntersuchung von LexBFS$^-$ ist Folgendes zu beachten: Der LexBFS$^-$ Sweep erfolgt auf $\bar{G} = (V, \bar{E})$. Da aber gilt: $|\bar{E}| =$

		b		
		F	NF	OK
	F	b	b	b
a	NF	a	b	b
	OK	a	a	b

Tabelle 2: Entscheidungsfälle bei LexBFS*

$\frac{|V|^2-|V|}{2} - |E|$ liegt die Zeitkomplexität nicht zwangsläufig in $\mathcal{O}(|V| + |E|)$. Man kann sich aber folgenden Sachverhalt zunutze machen nach [46]:
Für einen Graphen G erzeugt man eine LexBFS auf $\bar{G} = (V, \bar{E})$, indem man die neu erstellten Sets S^* nicht vor den alten Sets einfügt, sondern dahinter. Damit bleiben allen Knoten eines Sets, die in G nicht zum gerade nummerierten Knoten adjazent sind (in $\bar{G}$ sind sie es), direkt ein Set vor den anderen Knoten aus ihrem alten Set S. In **update LexBFS** aus Algorithmus 4 wird Zeile (*) folgendermaßen geändert: previous(SET(w)) = S^* statt next(Set(w)) = S^*. So lässt sich auch LexBFS$^-$ in Linearzeit, also in $\mathcal{O}(|V| + |E|)$, ausführen.

3.4.4 LexBFS-Layer (nach [10])

LexBFS-Layer berücksichtigt bei dem Update der Labels, bzw. dem Versetzen von Knoten in andere Sets nur Adjazenzen innerhalb eines Layers. Wenn zwei Knoten adjazent sind aber nicht im gleichen Layer, wird bei dieser Variante von LexBFS die Adjazenz ignoriert. LexBFS-Layer lässt sich auch in Linearzeit implementieren. Es wird jedem Knoten die Zugehörigkeit zu einem Layer, aus einem vorherigen LexBFS-Sweep, mitgegeben. Im **update**-Teil von LexBFS-Layer wird in $\mathcal{O}(1)$ kontrolliert, ob zwei Knoten in einem Layer liegen. Dementsprechend wird die Adjazenz berücksichtigt oder nicht. LexBFS-Layer kommt im Algorithmus zur Erkennung von DH-Graphen (siehe Abschnitt 4.5) zum Einsatz. Dort wird ein LexBFS$^-$-Sweep „layerweise“ durchgeführt. Die Layer, die diesem Sweep zugrunde gelegt werden, stammen aus G und nicht aus $\bar{G}$.

3.4.5 LexBFS* nach [24]

LexBFS* benutzt zwei vorherige LexBFS$^+$-Sweeps, um zu entscheiden, welcher Knoten als nächstes nummeriert werden soll. Die Nummerierungen seien mit σ^+ und σ^{++} bezeichnet. Gibt es zu einem Zeitpunkt mehrere Knoten im Set S_{MAX}, so wird nach dem, in Tabelle 2 dargestellten, Kriterium zwischen dem Knoten $a = \underset{w \in S_{MAX}}{MAX}\ \sigma^{+^{-1}}(w)$, und $b = \underset{w \in S_{MAX}}{MAX}\ \sigma^{++^{-1}}(w)$ entschieden.
F steht für „a bzw. b fliegt“ , d.h. es gibt hinter S_{MAX} einen Knoten c, mit $c \in N(a)$ bzw. $c \in N(b)$. NF steht für „ein Nachbar von a bzw. b fliegt“ , wenn a bzw. b nicht fliegt aber ein Nachbar in S_{MAX} fliegt. Sollte weder a, noch ein Nachbar von a fliegen, so ist a OK. Wenn b OK ist, wird stets b ausgewählt. Falls aber a OK ist und b NF, dann wird a ausgewählt.

Wie lässt sich LexBFS* in Linearzeit implementieren? Zunächst einmal sind folgende Daten für jeden Knoten v zu speichern:

1. Die höchste Nummer, die ein zu v adjazenter Knoten in σ^+ und σ^{++} hat, mit $i_+(v)$ bzw. $i_{++}(v)$ bezeichnet. Wenn es keinen Knoten gibt, der hinter v steht und adjazent zu v ist, so ist $i_+(v) = \sigma^{+-1}(v)$ bzw. $i_{++}(v) = \sigma^{++-1}(w)$
2. Die Menge der Knoten, die Nachbarn von v sind, vor v nummeriert wurden und einen Nachbarn haben, der nach v nummeriert wurde. Für σ^+ wird die Menge mit $A(v)$ bezeichnet, für σ^{++} mit $B(v)$.
3. Außerdem müssen die Größe der Mengen $A(v)$ und $B(v)$ gespeichert werden.

Diese Daten können während der beiden LexBFS$^+$-Sweeps gespeichert werden. Deren Laufzeit bleibt trotzdem linear. Außerdem müssen die Listen $A(v)$ und $B(v)$ auf dem Laufenden gehalten werden, d.h. sobald ein Knoten w nummeriert wurde, müssen die Einträge von w in den Listen $A(v)$ und $B(v)$ entfernt werden. Wenn diese Einträge mittels einer Liste verbunden sind gelingt eine Entfernung von w aus allen Listen $A(v)$ und $B(v)$ in $\mathcal{O}(deg(w))$.
Um den aktuell zutreffenden Fall aus Tab. 2 zu finden, sind folgende Abfragen durchzuführen. Sobald die erste zu einem Ergebnis kommt, wird eine Entscheidung getroffen.

1. $i_+(a) > \sigma^{+-1}(a)$ (a fliegt) wähle b.
2. $i_{++}(b) > \sigma^{++-1}(b)$ (b fliegt und a fliegt ja nicht) wähle a.
3. $|B(b)| = 0 \vee |A(a)| \neq 0$ wähle b (b ist OK oder a ist nicht OK).
4. $y \in B(b)$ $i_+(y) = \sigma^{+-1}(a)$ wähle b.
5. wähle a.

Im Hinblick auf die lineare Laufzeit ist das Auffinden von Knoten a für jedes Set S_{MAX} von Bedeutung. a ist der Knoten aus S_{MAX}, der als letztes in σ_+ nummeriert wurde. Dieser kann in der S_{MAX}, die nach der Nummerierung σ_{++} sortiert ist, an beliebiger Stelle stehen. In der Implementierung wird der von Spinrad [72] vorgeschlagene Weg gewählt. Man erstellt eine Kopie des Graphen, in der die Adjazenzlisten gemäß σ_+ sortiert sind. Führt man LexBFS* aus, so wählt man zwischen dem Knoten b, der in S_{MAX} im Original Set vorne steht, und dem Knoten a, der in S_{MAX} in der Kopie vorne steht. LexBFS* wurde von Corneil, Olariu und Stewart [24] zur Erkennung von Intervall-Graphen, siehe Abschnitt 4.3, entwickelt.

3.4.6 LexBFS SEO

Bei dieser LexBFS Variante nach [55] werden die Knoten rückwärts nummeriert, mit einem „tie-breaker“, der sich aus den minimalen Knoten Separatoren ergibt. Auch wenn der Algorithmus zur Erkennung stark chordaler Graphen (siehe 4.7.2)

später nur angedeutet wird und auch nicht vollständig implementiert wurde, folgen einige Überlegungen zu LexBFS-SEO (SEO = Strong Elimination Ordering). Beachtenswert an LexBFS-SEO ist die ursprüngliche Idee MCS und LexBFS einzusetzen. Als „tie-breaker“ wird ein Wert $Sep(v)$ verwendet, der über die Anzahl der Knoten Separatoren berechnet wird, in denen v enthalten ist. Zunächst wird jeder Kante $vw = e \in E$ die Anzahl der Separatoren $S(e)$ zugeordnet, in denen ihre beiden Endpunkte enthalten sind. Im eigentlichen LexBFS-SEO Sweep wird zunächst $\forall v \in V : Sep(v) = 0$ gesetzt und mit einem beliebigen Startknoten s angefangen. $\sigma_{LexBFS-SEO}(s) = n$. Der „Sep-Wert“ aller Nachbarn von s wird um den Wert $S(e)$ ihrer gemeinsamen Kante erhöht. Sollte in einem Nummerierungsschritt das Set S_{MAX} mit dem höchsten Label mehr als einen Knoten enthalten, wird der Knoten mit dem höchsten Sep-Wert nummeriert.
Sämtliche minimalen Knoten Separatoren lassen sich mittels MCS in $\mathcal{O}(|V| + |E|)$ berechnen, siehe [58]. Von Chandran und Gandroni [12] wurde eine Möglichkeit vorgestellt, alle minimalen Knoten Separatoren für einen chordalen Graphen in $\mathcal{O}(|V| + |E|)$ mittels einer beliebigen PEO zu bestimmen, so dass anstelle von MCS auch LexBFS eingesetzt werden kann. Die Laufzeit von LexBFS-SEO ist $\mathcal{O}(k^2|V|)$ mit $k =$ maximale Größe eines minimalen Knoten Separators [55]. Der dominierende Faktor der Laufzeit ist die Bestimmung der Anzahl der Separatoren, in der eine Kante enthalten ist. Für die Laufzeit ist es zwar nicht von Bedeutung, aber erwähnenswert ist auch die Tatsache, dass innerhalb der Sets die Knoten nicht sortiert sind. Daher ist stets, wenn $|S_{MAX}| > 1$ gilt, der Knoten mit maximalem Sep-Wert in S_{MAX} explizit zu bestimmen.

3.4.7 LexBFS - min deg

Bei einer weiteren Variante von LexBFS [34] wird als „tie-breaker“ die Anzahl seiner noch nicht nummerierten Nachbarn verwandt. Diese Variante lässt sich in linearer Zeit $\mathcal{O}(|V| + |E|)$ ausführen. Analog zu den linearen Implementierungen von LexBFS$^+$ lassen sich auch für LexBFS-„min deg“ die Knoten Liste und sämtliche Adjazenzlisten nach dem Grad jedes Knoten sortieren. Man verwendet den Knotengrad an Stelle der Nummerierung und sortiert Knotenliste und Adjazenzliste aufsteigend. Diese „Nummerierung “ ist nur eine Relation $\tau^{-1}(V = \{v_1, \ldots, v_n\}) \rightarrow \{1, \ldots, n-\}$. Knoten gleichen Grades können in die Listen in beliebiger Reihenfolge eingefügt werden. Wird ein Knoten v mit i neu nummeriert und dementsprechend die Sets aktualisiert, behalten die unnummerierten Knoten $u, w \in N(v)$ ihre Reihenfolge bei. Somit müssen Sie auch nur in der Reihenfolge bearbeitet werden in der sie in der Adjazenzliste von v hinterlegt sind. D. h. ein Update der Knotengrade ist für die Teilgraphen $G_i, i = 1, \ldots, n$ ebenso wenig notwendig wie das Berechnen der Labels.

3.4.8 Laufzeiten

Die in Tabelle 3.4.8 Laufzeiten wurden auf einem Rechner mit einem Pentium III Prozessor mit 700 MHz Taktung und 512 MB Arbeitsspeicher, 512 MB maximalem

Heap-Speicher für die Java Virtual Machine und SUSE Linux 10.1 als Betriebssystem erreicht.

Knotenanzahl	**Kantenanzahl**	**LexBFS**	**MCS**	**LexDFS**
1000	0,25 Mio.	1,3	0,2	0,8
5000	1,25 Mio.	30	3	18
50000	1,25 Mio.	33	5	441
100000	2,5 Mio.	68	10	1746
200000	1,25 Mio.	43	9	5705
300000	1,25 Mio.	32	17	10934
500000	1,25 Mio.	38	22	23570

Tabelle 3: Laufzeiten von LexBFS, MCS und LexDFS in Sekunden

3.4.9 Ein Beispiel

Gegeben sei der Graph aus Abb. 12, siehe Seite 38. Es ergeben sich die in Tab. 4 - 6 dargestellten Labels bei LexBFS, MCS und LexDFS. Die unterschiedlichen Nummerierungen sind Tabellen 10 und 8 zu entnehmen. Für LexBFS-SEO und LexBFS-rückwärts ergeben sich die gleiche Nummerierung. Die Gewichte der Kanten sind: $v_4v_8 = 12$, $v_8v_{10} = 7$, $v_3v_6 = 6$, $v_6v_{10} = 3$. Bei allen weiteren Kanten gehören nicht beide Knoten zu einem gemeinsamen minimalen Knoten Separator.

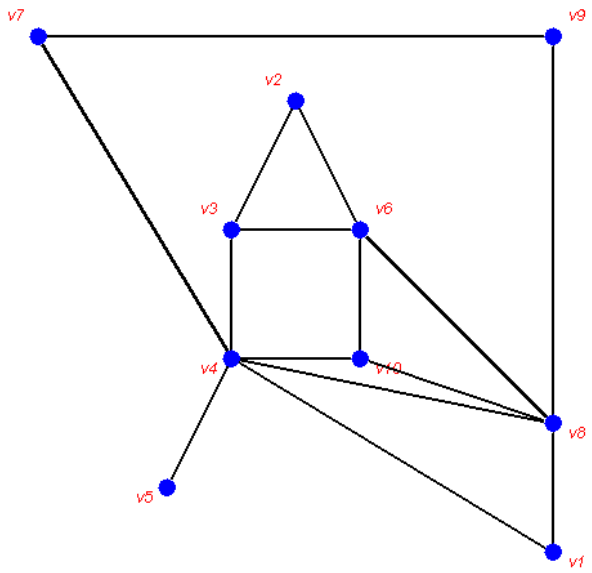

Abbildung 12: Ein Beispielgraph

Knoten	**n. 1.**	**n. 2.**	**n. 3.**	**n. 4.**	**n. 5.**	**n. 6.**	**n. 7.**	**n. 8.**	**n. 9.**
v1	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v2	∅	∅	∅	∅	6	6	6	6/3	6/3
v3	∅	9	9	9	xxx	xxx	xxx	xxx	xxx
v4	10	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v5	∅	9	9	9	9	xxx	xxx	xxx	xxx
v6	∅	∅	8	8/7	8/7/6	8/7/6	8/7/6	xxx	xxx
v7	∅	9	9	9	9	9	xxx	xxx	xxx
v8	10	10/9	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v9	∅	∅	8	8	8	8	8/4	8/4	xxx
v10	∅	9	9/8	xxx	xxx	xxx	xxx	xxx	xxx

Tabelle 4: LexBFS Labels nach der *i-ten* Nummerierung für Graph aus Abb. 12

Knoten	**n. 1.**	**n. 2.**	**n. 3.**	**n. 4.**	**n. 5.**	**n. 6.**	**n. 7.**	**n. 8.**	**n. 9.**
v1	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v2	∅	∅	∅	∅	I	II	xxx	xxx	xxx
v3	∅	I	I	I	II	xxx	xxx	xxx	xxx
v4	I	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v5	∅	I	I	I	I	I	I	xxx	xxx
v6	∅	∅	I	II	xxx	xxx	xxx	xxx	xxx
v7	∅	I	I	I	I	I	I	I	xxx
v8	I	II	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v9	∅	∅	I	I	I	I	I	I	II
v10	∅	I	II	xxx	xxx	xxx	xxx	xxx	xxx

Tabelle 5: MCS Labels nach der *i-ten* Nummerierung für Graph aus Abb. 12

Knoten	**n. 1.**	**n. 2.**	**n. 3.**	**n. 4.**	**n. 5.**	**n. 6.**	**n. 7.**	**n. 8.**	**n. 9.**
v1	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v2	∅	∅	∅	∅	5	6/5	xxx	xxx	xxx
v3	∅	2	2	2	5/2	xxx	xxx	xxx	xxx
v4	1	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v5	∅	2	2	2	2	2	2	2	2
v6	∅	∅	3	4/3	xxx	xxx	xxx	xxx	xxx
v7	∅	2	2	2	2	2	2	8/2	xxx
v8	1	2/1	xxx	xxx	xxx	xxx	xxx	xxx	xxx
v9	∅	∅	3	3	3	3	3	xxx	xxx
v10	∅	2	3/2	xxx	xxx	xxx	xxx	xxx	xxx

Tabelle 6: LexDFS Labels (vorwärts Nummerierung) nach der *i-ten* Nummerierung für Graph aus Abb. 12

Knoten	**LexBFS rückw.**	**LexBFS vorw.**	**LexBFS** $^{+}$	**LexBFS** $^{+}{}_{+}$	**LexBFS** $^{-}$	**LexBFS** *
v1	10	1	7	4	1	7
v2	1	10	1	10	7	1
v3	6	5	3	7	3	3
v4	9	2	6	2	10	6
v5	5	6	10	1	4	10
v6	3	8	2	9	8	2
v7	4	7	9	3	5	9
v8	8	3	5	5	9	5
v9	2	9	8	8	6	8
v10	7	4	4	6	2	4

Tabelle 7: Nummerierungen des Graphen aus Abb. 12

Knoten	**LexBFS minDeg**	**LexBFS Layer**	**MCS**	**LexDFS**
v1	1	1	10	1
v2	10	10	4	7
v3	7	6	5	6
v4	3	2	9	2
v5	9	7	3	10
v6	5	5	6	5
v7	8	8	2	9
v8	2	3	8	3
v9	6	9	1	8
v10	4	4	7	4

Tabelle 8: Nummerierungen des Graphen aus Abb. 12

4 Erkennung von Graphenklassen und weitere Anwendungen von LexBFS

Zur Erkennung der vorgestellten Graphenklassen wurden im Laufe der Zeit unterschiedliche Algorithmen vorgeschlagen. Literaturhinweise dazu erfolgen in den entsprechenden Abschnitten. Was aber zeichnet einen Algorithmus gegenüber einem anderen aus? Was sind die Gütekriterien für einen Algorithmus? Es gibt drei wichtige Kriterien. Die Korrektheit des Algorithmus wird selbstverständlich vorausgesetzt, auch wenn sie im weiteren Verlauf für jeden Algorithmus nachgewiesen bzw. auf Literaturstellen dazu hingewiesen wird.
Kriterien für die Güte eines Algorithmus zur Erkennung von Graphenklassen:

1. Laufzeit und Speicherplatzbedarf.
2. Komplexität des Algorithmus und der verwendeten Datenstrukturen.
3. Möglichkeit, eine Bestätigung für die Zugehörigkeit oder die Nichtzugehörigkeit zur Graphenklasse zu liefern.

1.) Für alle hier betrachteten Graphalgorithmen gilt als untere Grenze der oberen Schranke, eine Laufzeit und ein Speicherplatzbedarf von $\mathcal{O}(|V| + |E|)$. Für eine Überprüfung, ob ein Graph zu einer bestimmten Klasse gehört, ist in der Regel das Betrachten der gesamten Knotenmenge und der gesamten Kantenmenge erforderlich, d. h. es ist mindestens eine Laufzeit und ein Platzbedarf, der linear, proportional zur Anzahl der Knoten und Kanten ansteigt, anzunehmen. Auf LexBFS basierende Algorithmen können für fast alle hier betrachteten Graphenklassen eine lineare Laufzeit garantieren. Sie waren damit aber nicht immer die ersten Algorithmen.
2.) Weiterhin wünschenswert ist ein einfacher Ablauf und die Verwendung einfacher Datenstrukturen, also eine einfache Implementierung. Gerade dabei zeichnen sich die auf LexBFS basierenden Algorithmen aus.
3.) Sinn der Algorithmen ist es zu entscheiden, ob ein Graph zu einer Graphenklasse gehört oder nicht. Oftmals ist man daran interessiert, die für eine Graphenklasse charakteristischen Merkmale, die die Zugehörigkeit oder die Nichtzugehörigkeit zur Klasse bestimmen, zu finden, z. B. bei Cographen den Cotree oder einen P_4. Die auf LexBFS basierenden Algorithmen sind oftmals unter diesem Aspekt anderen Algorithmen vorzuziehen, auch wenn diese gleiche Laufzeit haben.

Die Gewichtung dieser drei Kriterien ist von unterschiedlichen Faktoren abhängig. Teilweise kann es Zielkonflikte geben. Gerade auch der dritte Punkt ist von immenser praktischer Bedeutung, selbst wenn die Bestätigungen für weitere Rechnungen nicht benötigt werden. Besonders wenn man die Möglichkeit einer nicht fehlerfreien Programmierung berücksichtigt [29]. Allen in den nächsten Abschnitten vorgestellten Algorithmen liegt ein gemeinsames Schema zugrunde, abgewandelt nach [32]:

1. Erzeuge eine oder mehrere bestimmte Ordnungen σ_i $i = 1, \ldots, k$ der Knoten mittels eines oder mehrerer Sweeps von LexBFS und oder seiner Varianten.
2. Überprüfe für σ_i $i = 1, \ldots, k$ charakteristische Eigenschaften.
3. Erzeuge unter Zuhilfenahme von σ_i $i = 1, \ldots, k$ definierende Merkmale oder verbotene Teilgraphen dieser Graphenklasse.

4.1 Erkennung von chordalen Graphen

4.1.1 Algorithmus und Laufzeit

Der Algorithmus zur Erkennung chordaler Graphen hat folgendes Aussehen.

Algorithmus 8 : Chordalitätstest mit Bestätigung

Eingabe : ein Graph $G = (V, E)$
Ausgabe : Ein PEO oder einen C_n mit $n \geq 4$

1 Erzeugen einer Ordnung σ auf V mittels eines LexBFS - rückwärts Sweeps.
2 Überprüfen, ob σ ein PEO ist.
3 Ausgabe von σ falls G chordal ist, oder Ausgabe eines sehnenlosen Kreises mit Mindestlänge 4: C_n $n \geq 4$.

Zu Punkt 1.) Es ist bereits oben der Algorithmus angegeben und außerdem die Laufzeit $\mathcal{O}(|V| + |E|)$ hergeleitet worden.
zu 2.) Es wird Satz 2.3 benutzt. Daher kann am Beginn der Nummerierung ein beliebiger Knoten ausgewählt werden. Wenn dieser ein simplizialer Knoten ist, bleibt mindestens ein weiterer, nicht benachbarter, simplizialer Knoten. Und wie Fulkerson und Gross [41] gezeigt haben, gilt:

Satz 4.1 (Startknoten PEO) *[41]*
Jeder simpliziale Knoten v kann Startknoten eines PEO sein.

Wie überprüft man, ob σ ein PEO ist? Naiverweise müsste man für jeden Knoten $v \in V$ überprüfen, ob $N^+(v)$ eine Clique bildet. D.h. für jeden Knoten $x \in N^+(v)$ überprüfen, ob x zu allen anderen Knoten aus $N^+(v)$ adjazent ist. Dieses Vorgehen hat, summiert über alle Knoten $v \in V$, keine lineare Laufzeit in $\mathcal{O}(|V|+|E|)$. Daher geht man einen anderen Weg. Man bildet für jeden Knoten $v \in V$ eine Liste von Knoten $A(v)$, mit denen v benachbart sein muss, damit σ ein PEO ist. Das sind alle Knoten $u \in V$, die mit v einen gemeinsamen Nachbarn w haben, der vor u und v nummeriert wurde.

$$u \in V : \exists w \in V \text{mit } v \in N(w) \wedge u \in N(w) \wedge \ \sigma(w) < \sigma(v) < \sigma(u)$$

Dieser Algorithmus **perfect** ist in Algorithmus 9 in Pseudocode dargestellt:
Die Teilprozedur **differ** ist in Algorithmus 10 dargestellt.

Algorithmus 9 : perfect

Daten : Ein Graph G und eine Sortierung σ
Ergebnis : wahr oder falsch

für alle $v \in V$ **tue**
  $A(v) = \emptyset$
`/* Initialisierung */`
für $i = 1$ **bis** $n - 1$ **tue**
  $v = \sigma(i)$
  $X = N^+(v)$
  wenn $X \neq \emptyset$ **dann**
    $u = \sigma(\underset{x \in X}{Min}\ \sigma^{-1}(x))$
    concat $\{X|u\}\{A(u)\}$
    `/* Anhängen von X ohne u in` $A(u)$ `*/`
    wenn *differ*$(A(v), Adj(v))$ **dann**
      return false
  return true;

Algorithmus 10 : differ

Daten : zwei Knoten Listen $A(v); Adj(v)$
Ergebnis : wahr oder falsch

für alle $w \in Adj(v)$ **tue**
  $Test(w) = 1$
`/* Initialisierung */`
für alle $w \in A(v)$ **tue**
  wenn $Test(w) = 0$ **dann**
    `/*` $w \notin Adj(u)$ `*/`
    return true
für alle $w \in Adj(v)$ **tue**
  $Test(w) = 0$
`/* Zurückstellen */`
return false

Ein Durchlauf von **differ** lässt sich in $\mathcal{O}(|A(v)| + |Adj(v)|)$ Zeit ausführen, d. h. über alle Aufrufe $\mathcal{O}(\sum_{u \in V} |A(u)| + \sum_{v \in V} |Adj(v)|)$. Es gilt $\sum_{u \in V} |A(u)| < \sum_{v \in V} |Adj(v)|$, da alle Einträge in $A(u)$ aus $Adj(v)$ stammen müssen und jeder Eintrag aus einer Adjazenzliste $Adj(v)$ maximal in eine $A(u)$ -Liste übernommen wurde. Zu den Zeiten aus **differ** kommen noch die Zeiten für das Initialisieren der $A(u)$-Listen, das Bestimmen der X-Mengen und das Suchen von $u \in X$. Das Zusammenstellen von X und das Suchen von $u \in X$ hat jeweils die Zeitkomplexität $\mathcal{O}(|Adj(v)|)$. Über alle Aufrufe gilt: $\mathcal{O}(|\underset{v \in V}{Adj(v)}|)$. Für das Initialisieren gilt: $\mathcal{O}(|V|)$, daher gilt für den gesamten Algorithmus **perfect**:

Satz 4.2 (Komplexität „perfect“) *[44]*
*Algorithmus **perfect** testet in $\mathcal{O}(|V|+|E|)$ Zeit, ob ein Schema σ ein PEO ist. Der Platzbedarf ist ebenfalls $\mathcal{O}(|V| + |E|)$.*

4.1.2 Korrektheit des Algorithmus

Um die Korrektheit des Algorithmus nachzuweisen, sind zwei getrennte Beweise zu führen. Zum einen ist Satz 4.3 zu beweisen, zum anderen ist die Korrektheit des Algorithmus **perfect** zu zeigen.

Satz 4.3 (LexBFS PEO) *nach [66]*
Ein endlicher Graph $G = (V, E)$ ist genau dann ein chordaler Graph, wenn die gespiegelte Nummerierung bzw. Sortierung σ der Knoten, die durch den Algorithmus LexBFS(rückwärts) festgelegt wurde, ein perfektes Eliminationsschema darstellt.
G ist chordal $\Leftrightarrow$ σ_{LexBFS} ist perfekt.

Beweis:
„$\Rightarrow$“: Beweis mittels vollständiger Induktion über $|V|$.
Induktionsanfang: Wenn $|V| = 1$ trivial.
Induktionsannahme: Wenn $|V| \leq n$ gilt der Satz 4.3.
Induktionsschluss: Da gemäß Satz 4.1 jeder simpliziale Knoten ein PEO starten kann, reicht es zu zeigen, dass $x = \sigma(1)$ ein simplizialer Knoten von G ist. Die Nummerierung der übrigen Knoten $V \setminus x$ in σ ist unabhängig von x. G_1 sei der induzierte Teilgraph von G ohne x. Da Chordalität an induzierte Teilgraphen vererbt wird, ist auch G_i chordal. Somit stellt gemäß Induktionsannahme die Reihenfolge der Knoten gemäß σ für G_1 ein PEO dar. Also ist σ ein PEO für G, wenn $x = \sigma(1)$ simplizial ist.
Ist Knoten $x = \sigma(1)$ simplizial? Angenommen, x wäre nicht simplizial, dann gibt es mindestens zwei Knoten $v_1, v_2 \in V$, die zwar zu x adjazent sind, aber nicht untereinander. Es sei o. B. d. A. $v_2 >_\sigma v_1$ und das Paar $v_2 v_1$ sei das lexikographisch größte Paar gemäß σ, d.h. $\{\sigma^{-1}(v_2), \sigma^{-1}(v_1)\} >_{LEX} \{\sigma^{-1}(v_i), \sigma^{-1}(v_j)\}\ \forall i, j \ \wedge$ $\sigma^{-1}(v_i) > \sigma^{-1}(v_j) : v_i v_j \notin E \wedge v_i, v_j \in N(x)$. Da sowohl v_1 als auch v_2 vor x nummeriert werden, muss es einen Pfad $P^{\setminus x}_{v_1, v_2}$ ohne x geben. Wenn es einen solchen sehnenlosen Pfad ohne einen Knoten $v_i \in N(x)$ gibt, wäre G nicht chordal, siehe

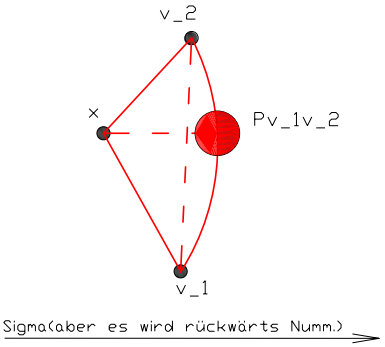

Abbildung 13: Sehnenloser Kreis mit v_1, v_2

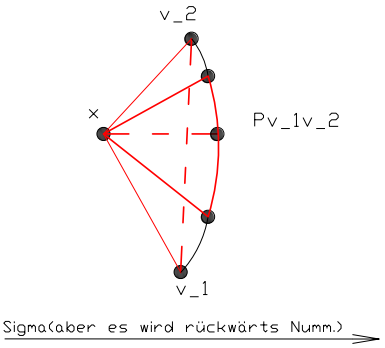

Abbildung 14: Sehnenloser Kreis ohne v_1, v_2

Abb. 13. Auf $P^{\setminus x}_{v_1,v_2}$ können v_1 und v_2 nur jeweils zu einem Knoten benachbart sein, sonst ist $P^{\setminus x}_{v_1,v_2}$ nicht sehnenlos. $P^{\setminus x}_{v_1,v_2}$ kann nur Knoten aus $N(x)$ enthalten sonst ergibt sich ein C_n $n \geq 4$, siehe Abb. 14. Ebenfalls o. B. d. A. kann man annehmen, dass $P^{\setminus x}_{v_1,v_2}$ nur Knoten v enthält, die vor v_1 nummeriert wurden. Dann muss aber v_2 zu allen $v \in P^{\setminus x}_{v_1,v_2}$ adjazent sein, sonst wäre $\{\sigma^{-1}(v_2), \sigma^{-1}(v_1)\}$ nicht lexikographisch am größten. Folglich kann es zwischen v_2 und v_1 nur einen sehnenlosen Pfad der Länge 3 geben, der als Mittelknoten einen Knoten v_N aus $N(x)$ enthält, siehe Abb. 15).

Es muss einen Knoten v_3 geben mit $v_3 > v_2 \wedge v_3v_1 \in E \wedge v_3x \notin E$ gemäß Bedingung 3 aus Satz 3.1. Der Knoten $v_3 = \sigma(Max(\sigma^{-1}(v \in V | vv_1 \in E \wedge vx \notin E)))$ sei der am frühesten nummerierten Knoten aus σ, der zu v_1 adjazent ist und zu x nicht. $v_3v_2 \notin E$ sonst wäre $v_3v_2xv_1$ ein sehnenloser Kreis. Daher muss es aber wegen Bedingung 3 aus Satz 3.1 auch einen Knoten v_4 geben, der nicht zu v_1 adjazent ist, aber zu v_2. Sei v_4 auch von diesen Knoten, der frühest nummerierte in σ. $v_4 = \sigma(Max(\sigma^{-1}(v \in V | vv_1 \notin E \wedge vv_2 \in E)))$. Der Knoten v_4 kann auch nicht zu x adjazent sein, sonst bildeten v_4v_1 ein lexikographisch höheres Knotenpaar, dessen Knoten nicht untereinander adjazent aber beide zu x adjazent sind. Es ergibt sich die

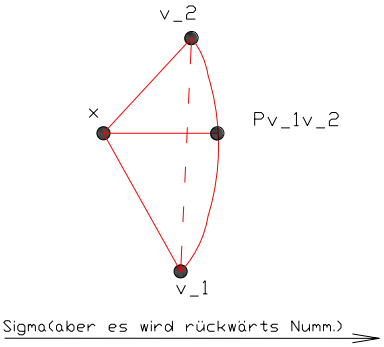

Abbildung 15: Der einzig mögliche Pfad zwischen v_1 und v_2

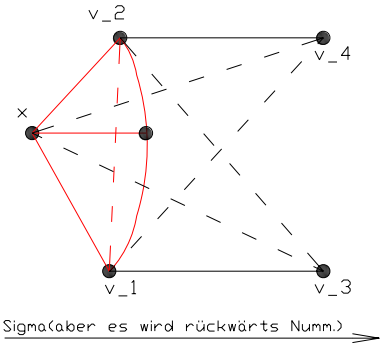

Abbildung 16: Ausgangssituation mit v_3 und v_4

in Abb. 16 dargestellte Situation. Wenn es zwischen v_3 und v_4 einen (sehnenlosen) Pfad $P_{v_3v_4}^{\setminus\{x\cup N(x)\}}$ gibt, der weder x noch Knoten aus $N(x)$ enthält, ergibt sich ein sehnenloser Kreis $xv_1v_3P_{v_3v_4}^{\setminus\{x\cup N(x)\}}v_4v_2$. Betrachtet man die Knoten aus $N(x)$, die vor v_1 und v_2 nummeriert wurden und bezeichnet sie mit $N^*(x)$, so müssen alle $y \in N^*(v)$ mit v_1 und v_2 adjazent sein, aufgrund der Maximalität von v_2v_1. Daher müssen v_3 und v_4 auch zu allen diesen Knoten aus $N^*(x)$ adjazent sein oder aber einen Nachbarn vor $N^*(x)$ haben. Wenn aber v_3 und v_4 nur Nachbarn in $N^*(x)$ hätten, würde v_2 vor v_3 nummeriert, da $v_4v_2 \in E \wedge v_3v_4 \notin E$. Also hat v_3 Nachbarn vor $N(x)$ bzw. $N^*(x)$, somit muss aber auch v_4 einen solchen Nachbarn haben und der Pfad $P_{v_3v_4}^{\setminus\{x\cup N(x)\}}$ existiert. Damit ist G nicht chordal, siehe Abb. 17. Widerspruch zur Annahme und $x = \sigma(1)$ ist ein simplizialer Knoten und „=>“ ist gezeigt.
„<= “ ist klar: Wenn es ein PEO gibt, dann ist G ein chordaler Graph. □

Einen anderen Beweis dafür, dass der zuletzt nummerierte Knoten $x = \sigma(1)$ ein simplizialer Knoten sein muss, liefert Simon [70]. Er benutzt die Eigenschaft, dass jeder minimale Separator ein vollständiger Teilgraph ist gemäß Satz 2.2. Es wird folgender Satz hergeleitet:

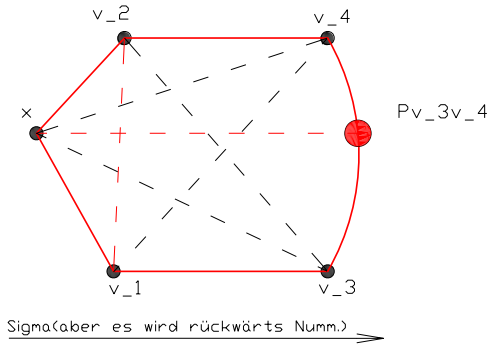

Abbildung 17: Sehnenloser Kreis mit v_3 und v_4

Satz 4.4 (Zusammenhang LexBFS Separator) *[70]*
Sei $G = (V, E)$ ein chordaler Graph und σ eine LexBFS Nummerierung. Ferner seien $x, y \in V xy \notin E$. Sei S der minimale $x - y$ Separator. Dann gilt: $\sigma^{-1}(x) < \sigma^{-1}(y) \Rightarrow \forall v \in S : \sigma^{-1}(x) < \sigma^{-1}(v)$

Die Knoten jedes minimalen Separators S werden in einer LexBFS Nummerierung σ vor dem letzten der beiden durch S getrennten Knoten nummeriert. Wäre $x = \sigma(1)$ kein simplizialer Knoten, gibt es zwei Knoten $v_1, v_2 \in N(x) : v_1 v_2 \notin E \wedge v_2 >_\sigma v_1$. Mindestens ein minimaler v_1, v_2-Separator enthält x. Gemäß Satz 4.4 gilt: $\sigma^{-1}(v_1) < \sigma^{-1}(x)$ Widerspruch! $\square$
Simon nahm an, dass der Satz 4.4 nur für LexBFS PEOs gelte. Diese Annahme haben Chandran, Ruskey, Ibara und Sawada [13] aber widerlegt und gezeigt, dass Satz 4.4 für sämtliche PEOs gilt und damit der vermutete Grund für die häufigere Verwendung von LexBFS gegenüber MCS nicht gegeben ist.

Es bleibt noch zu zeigen, dass der Algorithmus **perfect** korrekt ist. Es gibt zwei mögliche Fehler:

1. σ ist perfekt, aber **perfect** gibt „falsch“ zurück.

2. σ ist nicht perfekt, aber **perfect** gibt „richtig“ zurück.

Zu 1) Wenn „falsch“ zurückgegeben wird, muss sich in $A(v)$ ein Knoten w befinden, der nicht in $Adj(v)$ enthalten ist. Also muss gelten $\sigma^{-1}(w) > \sigma^{-1}(v)$. Ferner muss es einen Knoten u geben mit $\sigma^{-1}(v) > \sigma^{-1}(u)$ und $v, w \in N(u)$, damit w in der $\sigma^{-1}(u)$-ten Iteration in $A(v)$ aufgenommen werden konnte. Da aber $v \notin N(w)$, wäre σ kein PEO.
Zu 2) Wenn „richtig“ zurückgegeben wurde, ist in keinem $A(u)$ ein Knoten enthalten, der nicht auch in $Adj(u)$ vorkommt. Sei aber v der Knoten mit der höchsten Nummer $\sigma^{-1}(v)$, für den gilt: $N^+(v) = \{x \in N(v) | \sigma^{-1}(x) > \sigma^{-1}(v)\}$ ist keine Clique.
Sei $\sigma^{-1}(u)$ das Minimum der Knoten aus $N^+(v)$ bez. σ. Da „richtig“ zurückgegeben wird, gilt $\forall x \in N^+(v) \setminus \{u\} : x \in Adj(u)$. Es gilt $\sigma^{-1}(u) > \sigma^{-1}(v)$

Alle Paare $x, y \in N^+(v) \setminus \{u\}$ sind adjazent. x, y gehören zu $Z = z \in N(u) | \sigma^{-1}(z) > \sigma^{-1}(u)$ und Z ist gemäß der Annahme, dass v maximal ist, eine Clique. Dann wäre aber auch $N^+(v)$ komplett. Widerspruch! Damit ist **perfect** korrekt. □
Insgesamt gilt:

Satz 4.5 *[66]*
Chordale Graphen können in linearer Zeit erkannt werden.

Aus folgenden drei Tatsachen kann die von Corneil [17] als „$P3$-Regel" bezeichnete Eigenschaft von LexBFS Nummerierungen auf chordalen Graphen gewonnen werden:

1. Jede LexBFS Ordnung stellt für einen chordalen Graphen ein PEO dar.
2. Jeder induzierte Teilgraph eines chordalen Graphen ist chordal.
3. Die Nummerierung eines Knoten mittels LexBFS ist unabhängig vom zu diesem Zeitpunkt unnummerierten Teil des Graphen.

Satz 4.6 ($P3$-Regel) *[17]*
Sei G ein chordaler Graph und σ eine LexBFS Nummerierung von G. Dann gelte für $u, v, w \in V$ mit $u <_\sigma v <_\sigma w$:

$$uv \in E \wedge uw \in E \Rightarrow vw \in E$$

Beweis: Alle Knoten, die vor u nummeriert wurden, einschließlich u, induzieren einen chordalen Graphen G_u. Die LexBFS Nummerierung für G stellt beschränkt auf die Knoten von G_u auch eine LexBFS Nummerierung für G_u dar. Dann ist u der Startknoten eines PEO für G_u und demnach simplizial. Demnach muss $vw \in E$ gelten. □
Wie Tarjan und Yannakakis [73] gezeigt haben, gilt für alle Sortierungen, die eine MNS Sortierung darstellen, dass sie in umgekehrter Reihenfolge für einen chordalen Graphen ein PEO darstellen. Somit lässt sich der Algorithmus u. a. sowohl mit MCS als auch mit LexDFS Nummerierung ausführen. Bei LexDFS allerdings bisher nicht mit linearer Laufzeit.

4.1.3 Erweiterung des Algorithmus

Tarjan und Yannakakis [74] haben ihren Algorithmus zur Erkennung von chordalen Graphen um eine Bestätigung für Nichtchordalität erweitert. Im Falle eines nicht chordalen Graphen wird einen sehnenlosen Kreis mit Mindestlänge 4, also ein verbotener Teilgraph zurückgegeben. Diese Erweiterung basiert auf MCS Nummerierungen und kann für den Fall, dass eine LexBFS Nummerierung vorliegt, leicht abgewandelt und vereinfacht werden. Bisher wurde in **perfect** nur ein $w, v \in V$ Knotenpaar bestimmt, das zu einem Tripel $u, v, w \in V$ gehört, für das gilt: $u <_\sigma v <_\sigma w$ und $uv \in E \wedge uw \in E \wedge vw \notin E$. Für ein solches Tripel gilt, wenn v, w das lexikographisch höchste Paar zu u adjazenter aber nicht untereinander adjazenter Knoten ist, folgender Satz:

Satz 4.7 (Pfad) *nach [74]*
Sei $u, v, w \in V$ ein Tripel, für das gilt: $u <_\sigma v <_\sigma w$ und $uv \in E \wedge uw \in E \wedge vw \notin E$ dann gilt, wenn v, w das lexikographisch maximale Paar bezüglich σ mit diesen Eigenschaften ist:
Es gibt einen Pfad P zwischen v und w, der weder u noch einen Knoten $x \in N(u)$ enthält.

Beweis: Der Beweis ergibt sich aus der Umkehrung des Beweises von Satz 4.3. Sei σ eine LexBFS Nummerierung. Falls G chordal wäre, ist u in dem Teilgraphen $G_{\sigma(u)}$, der durch alle vor u nummerierten Knoten induziert wird, ein simplizialer Knoten. Da u nicht simplizial ist, muss mindestens ein C_n $n \geq 4$ in $G_{\sigma(u)}$ vorliegen. Gemäß des Beweises von Satz 4.3 muss es mindestens einen Pfad $P_{v,w}^{\setminus u \cup N(u)}$ geben. Der kürzeste dieser Pfade zwischen v und w muss sehnenlos sein. Zusammen mit u, v, w und diesem minimalen Pfad P_{vw} ergibt sich ein sehnenloser Kreis mit einer Länge von mindestens 4. □

Die Laufzeit für diese Erweiterung des Algorithmus ist ebenfalls linear, wie sich anhand folgender Überlegungen zeigen lässt. Um aus den mittels „perfect“ gefundenem Knotenpaar v, w ein Tripel zu machen, müssen beide Adjazenzlisten auf einen gemeinsamen Nachbarn u hin untersucht werden. Ein gemeinsamer Nachbar kann in Linearzeit gefunden werden. Aber wie kann ein Tripel gefunden werden, das die geforderte Maximalität von v und w bezüglich u garantiert? Man lässt **perfect** vor der Ausführung von **differ** für alle Knoten durchlaufen. Also berechnet man alle $A(v)$ Listen. Anschließend bestimmt man, rückwärts vom Knoten $y = \sigma(n)$ aus startend, das erste verletzende Paar w^*, v^*. Wenn außerdem noch die jeweiligen Listen $A(v)$ und $Adj(v)$ sortiert sind, bestimmt man damit automatisch das lexikographisch größte verletzende Tripel u^*, v^*, w^*. Abschließend ist nur noch der kürzeste Pfad zwischen den nicht benachbarten Knoten v^* und w^* des Tripels zu finden, der weder u^* noch Nachbarn von u^* enthält. Dazu lässt sich ein spezieller LexBFS-Sweep verwenden. Die zu vermeidenden Knoten $N[u] \setminus \{v, w\}$ setzt man in ein Set hinter dem eigentlichen Startset, dadurch werden sie erst am Schluss nummeriert. Den Knoten w setzt man an den Anfang des ersten Sets, damit dieser zuerst nummeriert wird. Man merkt sich im Zuge dieses Sweep die Entfernung aller Knoten vom Startknoten w. Die Entfernung der zu vermeidenden Knoten wird dabei überschätzt. Ausgehend vom Zielknoten v kann stets ein Knoten mit einer um 1 niedrigeren Entfernung gefunden werden, bis wieder der Startknoten w erreicht ist. Dazu muss lediglich maximal einmal die gesamte Knotenmenge durchlaufen werden. Man sortiert, wie bereits in Algorithmus 7 beschrieben, die Adjazenzlisten aller Knoten aufsteigend gemäß σ. Dann muss der nächste Knoten auf dem Pfad P_{vw} stets der erste Knoten, der Adjazenzliste des gerade nummerierten Knoten, sein. Dieses ist der gesuchte Pfad P_{vw}, der zusammen mit u, v, w den gesuchten, verbotenen Kreis bildet.

4.1.4 Weitere Zusammenhänge zwischen LexBFS und chordalen Graphen

Die ungeraden Potenzen eines chordalen Graphen G sind wiederum chordal. Laskar und Shier [59] haben gezeigt, dass dies auch für die geraden Potenzen eines chordalen Graphen G gilt, wenn G bestimmte Teilgraphen nicht enthält. Es stellt sich aber die Frage, ob ein PEO für einen Graphen G auch stets ein PEO für seine Potenzen darstellt. Sind G und G^2 beide chordal, dann haben sie auch ein gemeinsames PEO [7]. Brandstädt, Dragan, Chepoi und Voloshin haben gezeigt, dass es für alle chordalen Potenzen eines Graphen G ein gemeinsames PEO gibt [2]. Dieses lässt sich mit einer verallgemeinerten Version von MCS erzeugen. Interessanterweise ist jede LexBFS Sortierung eines chordalen Graphen G auch ein PEO für sämtliche ungeraden Potenzen, die ja bekanntermaßen ebenfalls chordal sind. Für einen Großteil der chordalen Graphen sind auch die LexBFS Nummerierungen PEOs für Potenzen mit geraden Exponenten, sofern diese chordal sind. Verbotene Teilgraphen, die Ausnahmen von dieser Regel charakterisieren, wurden ebenfalls von Brandstädt, Dragan und Nicolai [1] bestimmt. Für MCS und LexDFS Nummerierungen lassen sich nicht so allgemeine Aussagen treffen.

4.1.5 PEOs und chordale Graphen

Von den $720 = 6!$ Möglichkeiten den Graphen aus Abb. 18 zu nummerieren, stellen 68 Sortierungen ein PEO dar.

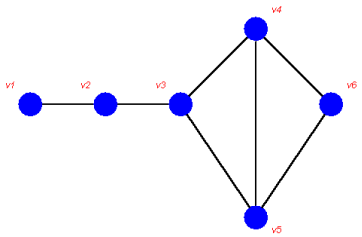

Abbildung 18: Beispielgraph

Sortierung	**LexBFS**	**MCS**	**LexDFS**	**MNS**	**PEO** in umgek. Richt.
$v_1, v_2, v_3, v_4, v_5, v_6$	+	+	+	+	+
$v_3, v_2, v_4, v_5, v_6, v_1$	+	+	+	+	+
$v_3, v_2, v_1, v_4, v_5, v_6$	-	+	+	+	+
$v_3, v_4, v_5, v_6, v_2, v_1$	-	+	+	+	+
$v_2, v_3, v_1, v_4, v_5, v_6$	+	+	-	+	+
$v_3, v_5, v_2, v_4, v_1, v_6$	-	-	-	-	+

Tabelle 9: Sortierungen des Graphen aus Abb. 18

Aus Tab. 9 ist zu ersehen, dass es PEOs gibt, die in umgekehrter Reihenfolge weder LexBFS oder MCS oder LexDFS oder andere MNS Nummerierungen darstellen. Shier [68] hat eine Verallgemeinerung von MCS vorgeschlagen, mit der sich sämtliche PEOs eines Graphen G bestimmen lassen.

4.2 Erkennung von Cographen

Zur Erkennung von Cographen wurden unterschiedliche Algorithmen vorgeschlagen. Erster mit linearer Laufzeit war der von Corneil, Perl und Stewart 1985 vorgeschlagene [27]. Dieser konstruierte den Cotree durch sukzessives Einfügen der Knoten. Ein weiterer Algorithmus mit linearer Laufzeit stammt von Habib und Paul [47]. Die erste Variante Cographen mittels LexBFS, zu erkennen stammt von Bretscher, Corneil, Bretscher, Habib und Paul [11], siehe auch [10]. Dort werden drei LexBFS-Sweeps benutzt. In [9] wurde dieser Algorithmus dahingehend verbessert, dass er nur noch zwei LexBFS-Sweeps benötigt. Dieser Algorithmus wird in diesem Abschnitt vorgestellt.

4.2.1 Der Algorithmus

Der Algorithmus aus [9] hat folgendes Aussehen:

Algorithmus 11 : Cograph 2-SweepTest

Eingabe : ein Graph $G = (V, E)$
Ausgabe : Ein P_4 wenn G kein Cograph, sonst ein Cotree T_G von G

1 Erzeuge eine Ordnung σ der Knoten mittels eines LexBFS-vorwärts Sweep.
2 Erzeuge unter Berücksichtigung von σ mittels eines LexBFS$^-$-Sweep auf $\bar{G}$ eine Ordnung $\bar{\sigma}$.
3 Überprüfe für σ und $\bar{\sigma}^-$, ob sie die Nachbarschafts-Untermengen-Bedingung (Neighbourhood Subset Property = NSP) erfüllen.
4 **wenn** *ja* **dann**
5 | Konstruiere den Cotree T_G.
6 **sonst**
7 | Finde einen P_4.

4.2.2 Beispiel: LexBFS auf Cographen

Wenn man den LexBFS-Algorithmus „labelfrei“ interpretiert, wird der gerade zu nummerierende Knoten x als Pivot-Element aufgefasst. Alle zu diesem Zeitpunkt vorhanden Sets S_i werden in einen zu x adjazenten Teil S_i^{Ax} und einen zu x nicht-adjazenten Teil S_i^{Nx} aufgeteilt. Diese Partitions-Vorstellung wird im Algorithmus 11 häufig verwendet und soll daher anhand des Cographen aus Abb. 19 und eines leicht modifizierten „Nicht-Cographen“ ($v_2v_7 \notin E$), siehe Abb. 20, erläutert werden. Beispielsweise ist:

$$\sigma = v_1, v_2, v_5, v_6, v_7, v_8, v_9, v_3, v_4$$

eine gültige LexBFS-Ordnung des Cographen aus Abb. 19. Daraus ergibt sich die folgende $\bar{\sigma}^-$ Nummerierung:

$$\bar{\sigma}^- = v_1, v_3, v_4, v_2, v_5, v_7, v_9, v_8, v_6$$

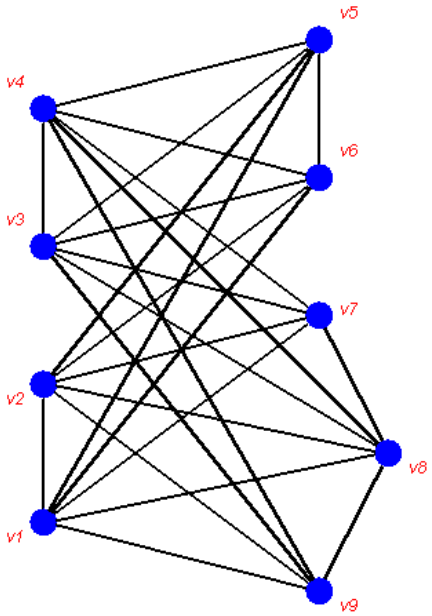

Abbildung 19: Ein Cograph

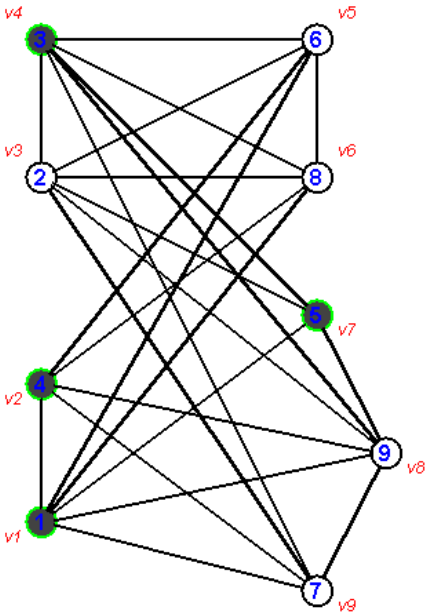

Abbildung 20: Ein „NichtCograph“

Für den Graphen aus Abb. 20 ist:

$$\sigma = v_1, v_2, v_5, v_6, v_8, v_9, v_7, v_3, v_4$$

eine gültige LexBFS Nummerierung und daraus ergibt sich:

$$\bar{\sigma}^- = v_1, v_3, v_4, v_2, v_7, v_5, v_9, v_6, v_8$$

Die Sets, die sich während der Nummerierung σ ergeben sind in Tab. 10 aufgelistet. Die Sets, deren Knoten zum aktuell nummerierten Knoten adjazent sind, sind mit [grün] und die nichtadjazenten mit [rot] hinterlegt.
Die Sets, die sich während der Nummerierung $\bar{\sigma}^-$ ergeben, sind in Tab. 11 aufgelistet. Die Sets deren Knoten zum gerade nummerierten Knoten in $\bar{G}$ adjazent sind, sind mit [grün] und die nichtadjazenten mit [rot] hinterlegt.
Die Sets, die sich für den Graphen aus Abb. 20 bei σ und $\bar{\sigma}^-$ ergeben sind in Tab. 12 und Tab. 13 zu entnehmen.
Um den Zusammenhang zwischen den LexBFS Nummerierungen σ und $\bar{\sigma}^-$ und dem Cotree T_G des Cographen G deutlich zu machen, sind in den Cotrees der Abbildung 21 jeweils die Sets mit dem höchsten Label rot eingefärbt.
Zwischen den Sets mit dem höchsten Label S_{MAX} und dem Cotree und seinen Teilbäumen scheint ein Zusammenhang zu bestehen, siehe Abbildung 21. Diesem wird gleich weiter nachgegangen, doch vorher soll der Begriff der ***Scheibe (= engl. Slice)*** erläutert werden.

$\sigma^{-1}(v_i)$	v_i	**Set mit höchstem Label**	**übrige Sets**
0	xxx	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$	xxx
1	v_1	$\{v_2, v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$
2	v_2	$\{v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$
3	v_5	$\{v_6\}$	$\{v_7, v_8, v_9\}$ $\{v_3, v_4\}$
4	v_6	$\{v_7, v_8, v_9\}$	$\{v_3, v_4\}$
5	v_7	$\{v_8\}$	$\{v_9\}$ $\{v_3, v_4\}$
6	v_8	$\{v_9\}$	$\{v_3, v_4\}$
7	v_9	$\{v_3, v_4\}$	xxx
8	v_3	$\{v_4\}$	xxx
9	v_4	xxx	xxx

Tabelle 10: Sets nach der i.ten σ Nummerierung für Graph aus Abb. 19

$\bar{\sigma}^{-1}(v_i)$	v_i	**Set mit höchstem Label**	**übrige Sets**
0	xxx	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$	xxx
1	v_1	$\{v_3, v_4\}$	$\{v_2, v_5, v_6, v_7, v_8, v_9\}$
2	v_3	$\{v_4\}$	$\{v_2\}$ $\{v_5, v_6, v_7, v_8, v_9\}$
3	v_4	$\{v_2\}$	$\{v_5, v_6, v_7, v_8, v_9\}$
4	v_2	$\{v_5, v_6, v_7, v_8, v_9\}$	xxx
5	v_5	$\{v_7, v_8, v_9\}$	$\{v_6\}$
6	v_7	$\{v_9\}$	$\{v_8\}$ $\{v_6\}$
7	v_9	$\{v_8\}$	$\{v_6\}$
8	v_8	$\{v_6\}$	xxx
9	v_6	xxx	xxx

Tabelle 11: Sets nach der i.ten $\bar{\sigma}$ Nummerierung für Graph aus Abb. 19

$\sigma^{-1}(v_i)$	v_i	**Set mit höchstem Label**	**übrige Sets**
0	xxx	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$	xxx
1	v_1	$\{v_2, v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$
2	v_2	$\{v_5, v_6, v_8, v_9\}$	$\{v_7\}$ $\{v_3, v_4\}$
3	v_5	$\{v_6\}$	$\{v_8, v_9\}$ $\{v_7\}$ $\{v_3, v_4\}$
4	v_6	$\{v_8, v_9\}$	$\{v_7\}$ $\{v_3, v_4\}$
5	v_8	$\{v_9\}$	$\{v_7\}$ $\{v_3, v_4\}$
6	v_9	$\{v_7\}$	$\{v_3, v_4\}$
7	v_7	$\{v_3, v_4\}$	xxx
8	v_3	$\{v_4\}$	xxx
9	v_4	xxx	xxx

Tabelle 12: Sets nach der i.ten σ Nummerierung für Graph aus Abb. 20

$\bar{\sigma}^{-1}(v_i)$	v_i	**Set mit höchstem Label**	**übrige Sets**
0	xxx	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$	xxx
1	v_1	$\{v_3, v_4\}$	$\{v_2, v_5, v_6, v_7, v_8, v_9\}$
2	v_3	$\{v_4\}$	$\{v_2\}$ $\{v_5, v_6, v_7, v_8, v_9\}$
3	v_4	$\{v_2\}$	$\{v_5, v_6, v_7, v_8, v_9\}$
4	v_2	$\{v_7\}$	$\{v_5, v_6, v_8, v_9\}$
5	v_7	$\{v_5, v_6, , v_9\}$	$\{v_8\}$
6	v_5	$\{v_9\}$	$\{v_6\}$ $\{v_8\}$
7	v_9	$\{v_6\}$	$\{v_8\}$
8	v_6	$\{v_8\}$	xxx
9	v_8	xxx	xxx

Tabelle 13: Sets nach der i.ten $\bar{\sigma}$ Nummerierung für Graph aus Abb. 20

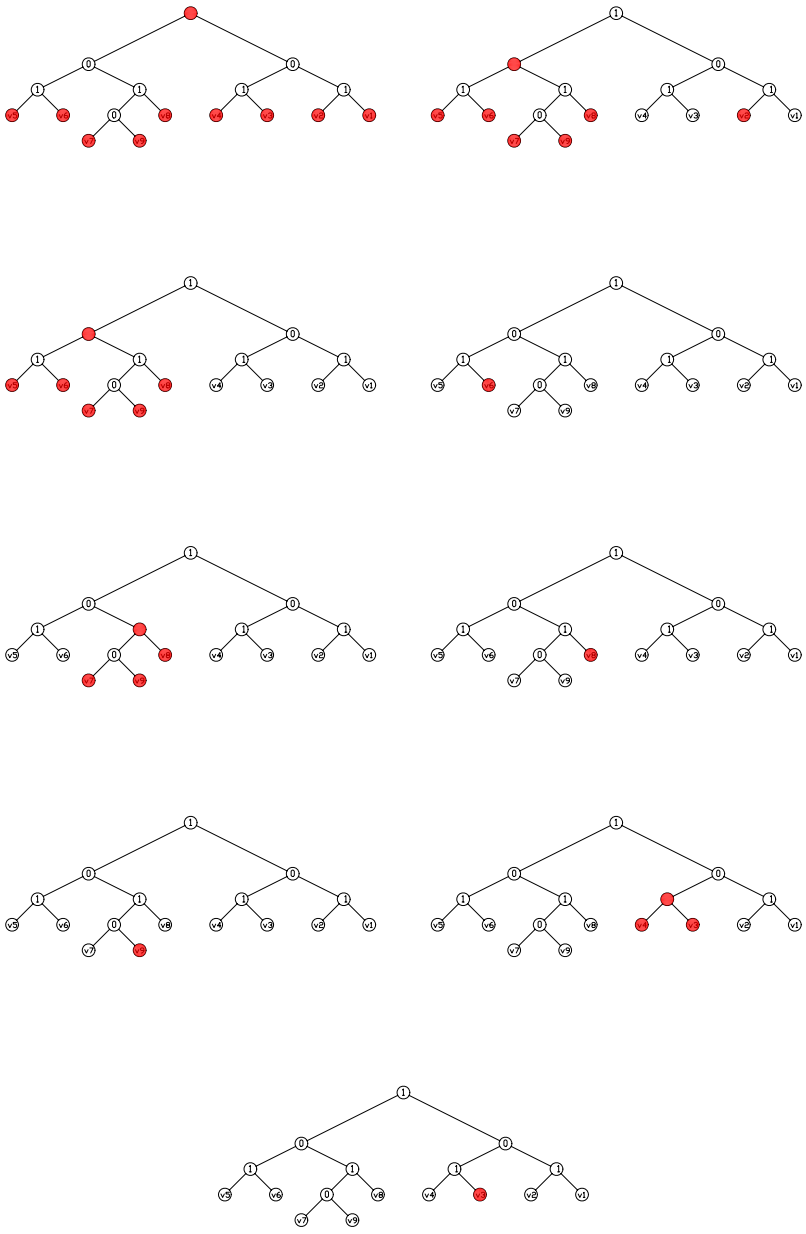

Abbildung 21: S_{MAX} $i = 1, \ldots, 9$ für den Graphen aus Abb. 19

4.2.3 Slices

Bei LexBFS-Nummerierungen wird stets ein Knoten als nächstes nummeriert, der aus dem Set mit dem aktuell höchsten Label entnommen wird. Diese Sets werden im Zusammenhang mit Cographen, speziell in den o. g. Arbeiten [10], [11] und [9], als ***Slices*** bezeichnet. Für das Verständnis des Algorithmus bzw. dessen Korrektheitsbeweis ist der Begriff und vor allem die Eigenschaften von Slices von zentraler Bedeutung.

Definition 4.1 (Slice)
Eine Menge von Knoten, die innerhalb einer LexBFS Nummerierung aufeinander folgen, wird als Slice S bezeichnet, wenn alle Knoten aus S zum Zeitpunkt der Nummerierung des ersten Knoten x aus S die gleiche bereits nummerierte Nachbarschaft haben. Also seien: $i = \min_{v \in S}(\sigma(v))$ $j = \max_{v \in S}(\sigma(v))$ *dann gilt:* $\forall v, w \in S : N_i(v) = N_i(w) \ \wedge \ \forall t : \sigma(t) > j \ N_i(t) \neq N_i(v)$

Da für eine LexBFS Nummerierung Bedingung 2 von Satz 3.1 gilt, ist offensichtlich, dass die Knoten aus dem Set mit dem höchsten Label (wie auch aus allen anderen Sets) aufeinander folgend nummeriert werden. Es gibt im Verlauf des gesamten LexBFS-Sweep $|V| = n$ verschiedene Slices. So stellt die komplette Knotenmenge V am Anfang der Nummerierung ein Slice dar. Slices werden mit $S(x)$ bezeichnet, wobei x, der erste von $S(x)$ nummerierte Knoten ist. Sobald x aus $S(x)$ nummeriert wurde, teilt sich $S(x)$ in 3 Teile auf:

1. x.
2. $S^A(x)$, die zu x adjazenten Knoten aus $S(x)$.
3. $S^N(x)$, die zu x nicht adjazenten Knoten aus $S(x)$.

$S^A(x)$ und $S^N(x)$ stellen nach der Nummerierung von x neue Sets dar. $S^A(x)$ ist nach der Nummerierung von x das Set mit dem höchsten Label. Das Set $S^N(x)$ teilt sich durch die weitere Nummerierung, der Knoten aus $S^A(x)$ in kleinere Sets auf. Diese Sets $S_i(x)$ $i = 1, \ldots, k$ werden im weiteren Verlauf als ***Zellen*** bezeichnet. Nachdem der letzte Knoten z aus $S^A(x)$ mit $\sigma^{-1}(z) = |S^A(x)| + 1 = l$ nummeriert wurde, ist $S^N(x)$ in seine Zellen $S_i(x)$ aufgeteilt worden. Betrachtet man den Graphen aus Abb. 19, so teilt sich $S(v_1)$ in $\{v_1\}$, $S^A(v_1) = \{v_2, v_5, v_6, v_7, v_8, v_9\}$ und $S^N(v_1) = S_1(v_1) = \{v_3, v_4\}$ auf. $S^N(v_1)$ hat nur eine Zelle, da alle v_3, v_4 die gleichen Nachbarn in $S^A(v_1)$ haben. Bezeichnet man mit A_v^x die Knoten aus $S^A(x)$ zu denen der Knoten v adjazent ist, so gilt:

$$v, w \in S^N(x) \wedge v \in S_i(x) \wedge w \in S_j(x) \begin{cases} A_v^x = A_w^x & \text{wenn } i = j \\ A_v^x \neq A_w^x & \text{wenn } i \neq j \end{cases}$$

Exemplarisch ist ein Slice in Abb. 22 dargestellt.
Analog dazu werden im Fall des LexBFS$^-$-Sweep, also auf dem komplementären Graphen $\bar{G}$, die Slices mit $\bar{S}^A(v_i)$ und mit $\bar{S}^N(v_i) = \bar{S}_1(v_i) \cup \ldots \cup \bar{S}_l(v_i)$ bezeichnet.

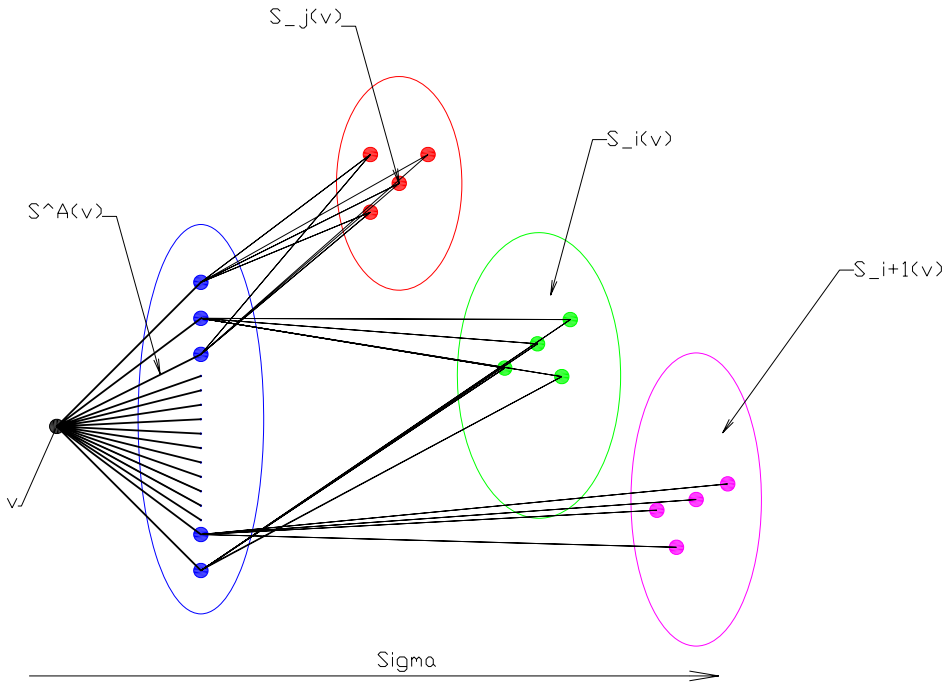

Abbildung 22: Ein Slice $S(v)$

Die während der Nummerierung $\bar{\sigma}^-$ für den Graphen aus Abb. 3 entstehenden Sets und damit auch Slices sind in Tabelle 11 und Tabelle 15 aufgeführt.
Für die Zellen gilt folgender Satz 4.8:

Satz 4.8 (Zellen nicht adjazent) *[9]*
Sei $G = (V, E)$ ein Cograph. Und $S_i(v)$ und $S_j(v)$ verschiedene Zellen von $S^N(v)$. Sei $v_i \in S_i(v)$ und $v_j \in S_j(v)$ dann gilt: $v_i v_j \notin E$. Zwei Knoten v_i, v_j aus unterschiedliche Zellen $S_i(v)$, $S_j(v)$ eines Slices $S(v)$ sind nicht adjazent.

Beweis: Angenommen, o. B. d. A. $j < l$. Sei $v_j v_l \in E$. Es gilt $v_j v \notin E \wedge v_l v \notin E$. Ferner muss es ein $x \in S^A(v)$ geben, mit $x \in A^x_{v_j} \wedge x \notin A^x_{v_l}$. Denn Zelle $S_j(v) <_\sigma S_l(v)$. Die Knoten (v, x, v_j, v_l) würden einen P_4 bilden. Siehe Abb. 23.□
Aus den gemachten Aussagen folgt für die Teilmengen $S^A(x)$ und $S_i(x)$ für $i = 1, \ldots, k$ folgender Satz 4.9:

Satz 4.9 (Subslices sind auch Slices) *nach [10]*
Wenn G ein Cograph ist, stellen folgende Teilmengen eines Slices wieder ein Slice dar:

1. $S^A(x)$

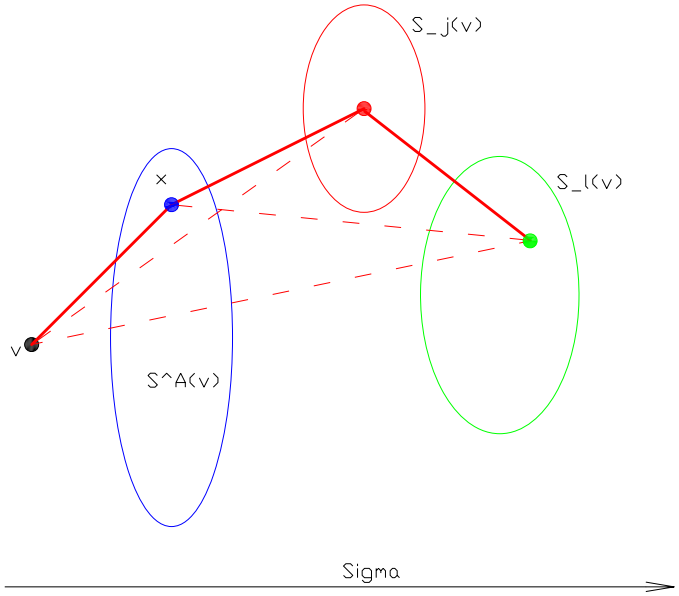

Abbildung 23: Adjazenzen innerhalb der Subsets $S_j(v_i)$ und $S_l(v_i)$

2. $S_i(x)$ *für* $i = 1, \ldots, k$

Man bezeichnet Sie auch als ***Subslices****.*

Beweis: zu 1.) Jeder Knoten $v \in S(x)$ hat die gleiche nummerierte Nachbarschaft $N_i(x)$ mit $\sigma^{-1}(x) = i$. Sei y der erste Knoten aus $S^A(x)$. Jeder Knoten $z \in S^A(x)$, der vor y nummeriert wird, hat die gleiche nummerierte Nachbarschaft $N_i(x) \cup x$. $S^A(x)$ ist nach $S(x)$ als nächstes das Set mit dem höchsten Label und daher ebenfalls ein Slice.
zu 2.) Gemäß Satz 4.8 haben die Zellen $S_i(x)$ keine Adjazenzen untereinander. Daher können die $S_i(x)$ zum Zeitpunkt der Nummerierung ihres jeweils ersten Knotens y_i, nur adjazent sein zu Knoten aus $N_i(x) \cup A^x_{y_i}$. Diese Nachbarschaft muss für alle Knoten $z \in S_i(x)$ gleich sein. □
Aus den Sets aus Tab. 10 für den Graphen aus Abb. 19 ergeben sich die Slices und Subslices gemäß Tab. 14.
Sollte G kein Cograph sein, können Adjazenzen zwischen den Zellen $S_i(v)$ und $S_j(v)$ existieren. Nicht jede Zelle ist in diesem Falle ein Slice. Im Graph aus Abb. 20 ist das Set $\bar{S}_2(v_1) = \{v_5, v_6, v_7, v_8, v_9\}$ ein Teilset, vgl. Tab.: 17 das bei der Nummerierung $\bar{\sigma}^-$ entstanden ist, nachdem alle Knoten $v_i \in \bar{S}^A(v_1)$ nummeriert wurden. Aber $\bar{S}_2(v_1)$ ist kein Slice, sondern eine Vereinigung der beiden Slices $\{v_7\}$ und $\{v_5, v_6, v_8, v_9\}$.
Die Knoten jedes Slices sind in einer LexBFS Nummerierung fortlaufend nummeriert. Es gilt Hilfssatz 4.2.1

v_i	$S^A(v_i)$	$S_1(v_i), \ldots, S_k(v_i)$	$N^l(S_i(v_i))$
v_1	$\{v_2, v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$	$\{v_5, v_6, v_7, v_8, v_9\}$
v_2	$\{v_5, v_6, v_7, v_8, v_9\}$	$\emptyset$	
v_5	$\{v_6\}$	$\{v_7, v_8, v_9\}$	$\emptyset$
v_6	$\emptyset$	$\emptyset$	
v_7	$\{v_8\}$	$\{v_9\}$	$\{v_8\}$
v_8	$\emptyset$	$\emptyset$	
v_9	$\emptyset$	$\emptyset$	
v_3	$\{v_4\}$	$\emptyset$	
v_4	$\emptyset$	$\emptyset$	

Tabelle 14: Slices und Subslices aus σ für den Graphen aus Abb. 19

v_i	$\bar{S}^A(v_i)$	$\bar{S}_1(v_i), \ldots, \bar{S}_m(v_i)$	$\bar{N}^l(\bar{S}_j(v_i))$
v_1	$\{v_3, v_4\}$	$\{v_2\}\{v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$ $\emptyset$
v_3	$\emptyset$	$\{v_4\}$	$\emptyset$
v_4	$\emptyset$	$\emptyset$	
v_2	$\emptyset$	$\emptyset$	
v_5	$\{v_7, v_8, v_9\}$	$\{v_6\}$	$\{v_7, v_8, v_9\}$
v_7	$\{v_9\}$	$\{v_8\}$	
v_9	$\emptyset$	$\emptyset$	
v_8	$\emptyset$	$\emptyset$	
v_6	$\emptyset$	$\emptyset$	

Tabelle 15: Slices und Subslices aus $\bar{\sigma}^-$ für den Graphen aus Abb. 19

v_i	$S^A(v_i)$	$S_1(v_i), \ldots, S_k(v_i)$	$N^l(S_i(v_i))$
v_1	$\{v_2, v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$	$\{v_5, v_6, v_7, v_8, v_9\}$
v_2	$\{v_5, v_6, v_8, v_9\}$	$\{v_7\}$	$\{v_8, v_9\}$
v_5	$\{v_6\}$	$\{v_8, v_9\}$	$\emptyset$
v_6	$\emptyset$	$\emptyset$	
v_8	$\{v_9\}$	$\emptyset$	
v_9	$\emptyset$	$\emptyset$	
v_7	$\emptyset$	$\emptyset$	
v_3	$\{v_4\}$	$\emptyset$	
v_4	$\emptyset$	$\emptyset$	

Tabelle 16: Slices und Subsets aus σ für den Graphen aus Abb. 20

v_i	$\bar{S}^A(v_i)$	$\bar{S}_1(v_i), \ldots, \bar{S}_m(v_i)$	$\bar{N}^l(\bar{S}_j(v_i))$
v_1	$\{v_3, v_4\}$	$\{v_2\}\{v_5, v_6, v_7, v_8, v_9\}$	$\{v_3, v_4\}$ $\{v_2\}$
v_3	$\emptyset$	$\{v_4\}$	$\emptyset$
v_4	$\emptyset$	$\emptyset$	
v_2	$\emptyset$	$\emptyset$	
v_7	$\emptyset$	$\emptyset$	
v_5	$\{v_9\}$	$\{v_6\}$	$\{v_9\}$
v_9	$\emptyset$	$\emptyset$	
v_6	$\emptyset$	$\emptyset$	
v_8	$\emptyset$	$\emptyset$	

Tabelle 17: Slices und Subsets aus $\bar{\sigma}^-$ für den Graphen aus Abb. 20

Hilfssatz 4.2.1 *nach [9] Sei σ eine LexBFS Nummerierung eines Graphen $G = (V, E)$. Sei σ_S der Ausschnitt von σ, der nur die Nummerierungen der Knoten von S enthält. Dann ist σ_S eine LexBFS Ordnung für den durch die Knoten $v \in S$ induzierten Teilgraphen.*

Beweis: Klar, da die Nummerierung innerhalb eines Slices S nur durch die Adjazenzen innerhalb des Slices S festgelegt wird. Demnach kann die Ordnung $<_{LexBFS}$ des Slices aber auch unabhängig von den Knoten $w \in V \setminus S$ betrachtet werden. □
Im weiteren Verlauf spielen die Nachbarschaften der Subslices $S_i(x)$ und $\bar{S}_j(x)$ eine große Rolle, dazu bedient man sich einer weiteren Nachbarschaftsdefinition. Da alle Knoten eines Slices $S(v)$ in dem Teil der Knotenmenge V, der vor dem Slice nummeriert wurde $\{w \in V | w <_\sigma v\}$, definitionsgemäß die gleiche Nachbarschaft haben.

Definition 4.2 (lokale nummerierte Nachbarschaft)
Man bezeichnet als lokale nummerierte Nachbarschaft $N^l(S_i(v))$ eines Slices $S_i(v)$, die nummerierten Knoten, die zu den Knoten aus $S_i(v)$ benachbart sind und zu $S(v)$ gehören. Nummeriert bedeutet dabei, zum Zeitpunkt des Entstehens dieses Slices bzw. Subslices. Also:

$$N^l(S_i(v)) = \{u \in S(v) | \forall w \in S_i(v) : uw \in E$$

Auf dem komplementären Graphen wird die lokale Nachbarschaft analog definiert. Allerdings betrachtet man auch hier sowohl die Nachbarschaften auf G als auch auf $\bar{G}$.

$$N^l(\bar{S}_i(v)) = \{u \in \bar{S}(v) | \forall w \in \bar{S}_i(v) : uw \in E$$

oder auf $\bar{G}$

$$\bar{N}^l(\bar{S}_i(v)) = \{u \in \bar{S}(v) | \forall w \in \bar{S}_i(v) : uw \in \bar{E}.$$

Da im weiteren Verlauf nur der bereits nummerierte Teil der Nachbarschaften interessant ist, wird auf den Zusatz „nummeriert“ meist verzichtet. Für Cographen setzen sich die lokalen Nachbarschaften aller $\forall i = 1, \ldots, k; \forall v \in V$ $S_i(v)$ ausschließlich aus Knoten aus $S^A(v)$ zusammen. Gemäß Satz 4.8 können Knoten aus unterschiedlich Subslices $S_i(v), S_j(v)$ nicht adjazent sein.
Um für diese lokalen Nachbarschaften einen entscheidenden Satz beweisen zu können, bedarf es des folgenden Hilfssatzes 4.2.2:

Hilfssatz 4.2.2 („Regenschirmfreiheit“) *[9]*
Jede LexBFS Nummerierung σ eines Cographen G ist „Regenschirm frei“ (umbrella free), d.h. es gibt keine

$$u, v, w \in V \; u <_\sigma v <_\sigma w \; \wedge \; uw \in E \; \wedge \; uv \notin E \; \wedge \; vw \notin E$$

Beweis: (vgl. auch Abb. 24) Gäbe es eine solche Konstellation und betrachtet man die am weitesten links stehende innerhalb von σ_{LexBFS}, gibt es gemäß 3.1 ein $x \in$

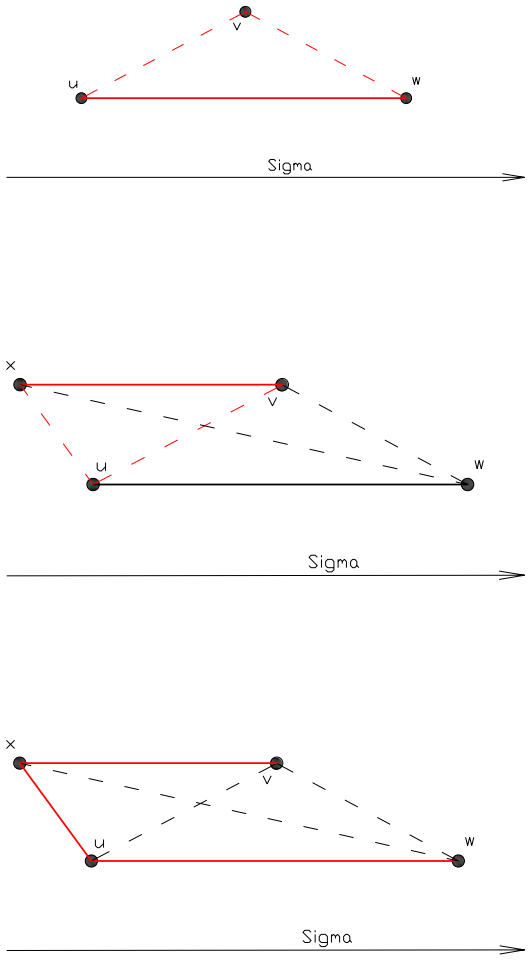

Abbildung 24: Zum Beweis der „Regenschirm-Freiheit“ von Cographen

$V : x <_\sigma u \ \wedge \ xv \in E \ \wedge \ xw \notin E$. Wenn $xu \in E$, dann ist v, x, u, w ein P_4. Wenn aber $xu \notin E$, bildet x, u, v einen „Regenschirm“ , der weiter links in σ steht. (Widerspruch zur Annahme)$\square$
Nun kann für die lokalen Nachbarschaften der folgende Satz 4.10 bewiesen werden:

Satz 4.10 (Neighbourhood Subset Property) *[10]*
Sei $G = (V, E)$ ein Cograph und σ eine LexBFS Nummerierung von G, dann gilt für alle $v \in V$ und $\forall i < j$:

$$N^l(S_i(v)) \supset N^l(S_j(v))$$

Beweis: Sei $y \in N^l(S_j(v))$, $x_j \in S_j(v)$ und $x_i \in S_i(v)$ $i < j$, dann muss, da $x_i x_j \notin E$ (gemäß4.8), $y \in N^l(S_i(v))$ sein, denn sonst bildeten $y <_\sigma < x_i <_\sigma x_j$ einen „Regenschirm“ . Da $S_i(v) <_\sigma S_j(v)$ gilt, muss es mindestens ein $x \in V$ mit $x \in N^l(S_i(v)) \ \wedge \ x \notin N^l(S_j(v))$ geben. Somit gilt in 4.10 die echten Teilmengen Beziehung. Vgl. dazu Abb. 25. Die Adjazenzen zwischen den Nachbarschaften $N^l(S_m(v))$ zu den Slices $S_m(v)$ sind aus Übersichtsgründen nicht dargestellt. □

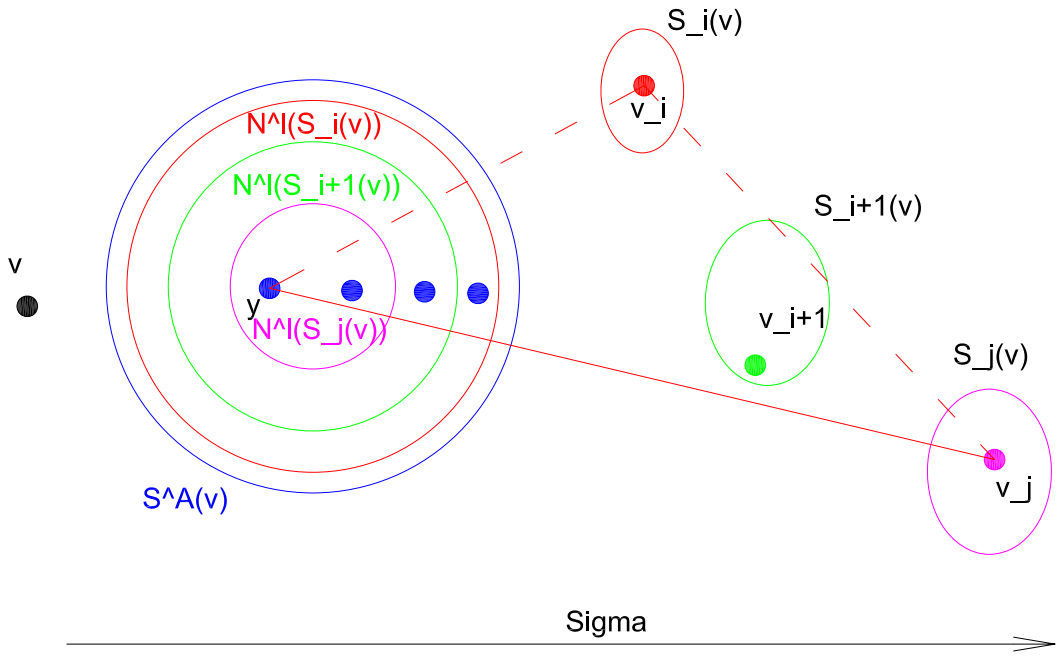

Abbildung 25: Neighbour Subset Theorem, rot eingefärbte Kanten ergäben einen „Regenschirm“

Analog gilt Satz 4.10 auch auf dem komplementären Graphen $\bar{G}$ und den komplementären Slices $\bar{S}_j(v)$. Da für den späteren Algorithmus eine Überprüfung der NSP auf G und $\bar{G}$ durchgeführt wird, andererseits aber eine Zeitkomplexität $\mathcal{O}(|V|+|E|)$ angestrebt wird, ist die Überprüfung der NSP auf $\bar{G}$ durch eine Überprüfung auf G zu modifizieren. Dazu benötigt man folgende Definition:

Definition 4.3 (Complement Neighbourhood Subset Property)
Sei $G = (V, E)$ ein Graph und $\bar{\sigma}^-$ eine LexBFS Nummerierung von $\bar{G}$, dann erfüllt $\bar{\sigma}^-$ die CNSP (Complement Neighbourhood Subset Property), wenn $\forall v \in V$ und $i = 1, \ldots, l-1$ gilt:

$$N^l(\bar{S}_i(v)) \cup \bar{S}_i(v) \subset N^l(\bar{S}_{i+1}(v))$$

Es gilt folgender Zusammenhang:

Satz 4.11 (CNSP mittels NSP) *[10]*
Wenn eine $\bar{\sigma}^-$ LexBFS Nummerierung von $\bar{G}$ die CNSP erfüllt ,dann erfüllt sie auch die NSP für $\bar{\sigma}^-$: $\forall v \in V \wedge \forall i < j$

$$\bar{N}^l(\bar{S}_i(v)) \supset \bar{N}^l(\bar{S}_j(v))$$

Also:

$$\forall i < j \; N^l(\bar{S}_i(v)) \cup \bar{S}_i(v) \subset N^l(\bar{S}_{i+1}(v)) \Rightarrow \bar{N}^l(\bar{S}_i(v)) \supset \bar{N}^l(\bar{S}_j(v))$$

Beweis: Der gesamte Slice, der vor einem Subslice $S_i(v)$ nummeriert wird, setzt sich aus der nummerierten Nachbarschaft $N^l(S_i(v))$ und der nummerierten „Nicht-Nachbarschaft“ $\bar{N}^l(S_i(v))$ zusammen. Außerdem sind alle vor $\bar{S}_{i+1}(v)$ nummerierten Knoten aus $S(v)$ vor $\bar{S}_i(v)$ nummeriert worden oder stammen aus $\bar{S}_i(v)$. Daher gilt:

$$N^l(\bar{S}_i(v)) \cup \bar{S}_i(v) \cup \bar{N}^l(\bar{S}_i(v)) = N^l(\bar{S}_{i+1}(v)) \cup \bar{N}^l(\bar{S}_{i+1}(v)) \tag{15}$$

Bezeichnen $A_i = N^l(\bar{S}_i(v))$ und $B_i = \bar{N}^l(\bar{S}_i(v))$, so gilt gemäß 15:

$$A_i \cup \bar{S}_i(v) \cup B_i = A_{i+1} \cup B_{i+1} \tag{16}$$

Wenn

$$\exists y : y \in A_{i+1} \wedge y \notin \{A_i \cup \bar{S}_i(v)\}$$

und

$$\nexists z : z \notin A_{i+1} \wedge z \in \{A_i \cup \bar{S}_i(v)\}$$

gilt, muss aufgrund von 15 gelten:

$$\exists y^* : y^* \in B_i \wedge y^* \notin B_{i+1}$$

und

$$\nexists z^* : z^* \notin B_i \wedge z^* \in B_{i+1}$$

also gilt:

$$\forall i : i = 1, \ldots, l-1 : \; \{A_i \cup \bar{S}_i(v)\} \subset A_{i+1} \Rightarrow B_i \supset B_{i+1} \tag{17}$$

Damit ist folgende Implikationskette offensichtlich:

$$\forall i \in 1, \ldots, l-1 : \; A_i \cup \bar{S}_i(v) \subset A_{i+1} \Leftrightarrow B_i \supset B_{i+1} \Rightarrow B_i \supset B_j (j > i) \tag{18}$$

□

4.2.4 Slices und der Cotree

Zwischen den Subslices $S_i(x)$,$\bar{S}_j(x)$ und den Teilbäumen des Cotrees T_G besteht folgender wichtiger Zusammenhang, der später bei der Konstruktion von T_G benötigt wird:

Satz 4.12 (Subslice to Subtree Theorem) *[10]*
Sei $G = (V, E)$ ein Cograph, T_G sein Cotree und σ eine LexBFS Nummerierung von G, und betrachtet man den Pfad P_x^R des ersten Knoten $x : \sigma(x) = 1$ zur Wurzel, so gilt: $S_i(x)$ enthält genau die Blätter (Knoten) des Teilbaumes T_{0i}^x.

Beweis: Seien alle Knoten $y \in S^A(x)$ bereits nummeriert, d.h. der Slice $S^N(x)$ ist bereits komplett in seine Subslices aufgeteilt. gemäß Satz 4.10 ist jeder Knoten aus $N^l(S_j(x))$ auch in $N^l(S_i(x))$ $j > i$ enthalten. Dementsprechend ist $N^l(S_1(x)) \supset N^l(S_i(x))$ $i > 1$. Der Teilbaum T_{01}^x enthält gerade die Knoten, die innerhalb der zu x nichtadjazenten Knoten die größte Nachbarschaft innerhalb von $N(x)$ haben. Daher muss den Blättern von T_{01}^x der Subslice $S_1(x)$ entsprechen. Folgerichtig entsprechen die Knoten von T_{02}^x den Blättern von $S_2(x)$ usw. $\forall T_{0i}^x : i = 1, \ldots, k$. □
Für den komplementären Graph $\bar{G}$ gilt:

Satz 4.13 (Subslice to Subset Theorem on complement Graph) *[10]*
Sei G ein Cograph und $\bar{\sigma}^-$ eine LexBFS Nummerierung von $\bar{G}$, und betrachtet man den Pfad P_x^R des ersten Knoten $x = \bar{\sigma}(1)$ zur Wurzel R, so gilt: $\bar{S}_j(x)$ enthält genau die Blätter (Knoten) des Teilbaumes T_{1j}^x.

Beweis: Folgt aus der Tatsache, dass der Cotree $T_{\bar{G}}$ (oder auch $\bar{T}$) genau der Cotree T_G (oder auch T) mit vertauschten 0 und 1 Knoten ist. Somit gilt $T_{0i}^x = \bar{T}_{1i}^x$ und $T_{1j}^x = \bar{T}_{0j}^x$ und daher auch, dass alle Knoten aus $\bar{S}_i(x)$ gerade die Blätter von $\bar{T}_{0i}^x = T_{1i}^x$ sind. □
Um diese Sachverhalte zu verdeutlichen, sind in den Abb. 26 - 28 der Cotree des Graphen aus Abb. 3 und seine weiteren Teilbäume dargestellt.

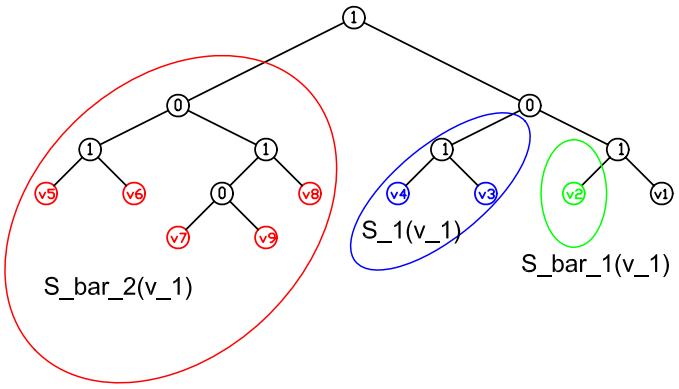

Abbildung 26: Die Subslices $S_1(v_1)$ und $\bar{S}_{1,2}(v_1)$

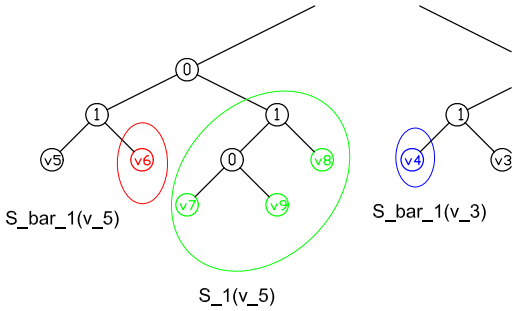

Abbildung 27: Die Subslices $S_1(v_5)$, $\bar{S}_1(v_5)$ und $\bar{S}_1(v_3)$

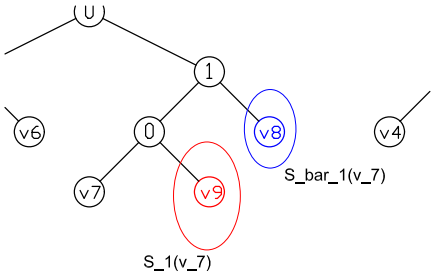

Abbildung 28: Die Subslices $S_1(v_7)$ und $\bar{S}_1(v_7)$

4.2.5 Korrektheit des Algorithmus zur Erkennung von Cographen

Zunächst soll die Korrektheit des Algorithmus 11 gezeigt werden. Dazu bedarf es des Beweises des folgenden Satzes 4.14

Satz 4.14 (Zusammenhang CoGraph NSP Bedingung) *[10]*
Sei $G = (V, E)$ ein Graph und σ eine LexBFS-Nummerierung von G und $\bar{\sigma}^-$ eine LexBFS$^-$-Nummerierung von (G, σ), dann ist G genau dann ein Cograph, wenn σ und $\bar{\sigma}^-$ die NSP Bedingung erfüllen.

Beweis: (nach [9]) „⇒“: Wenn G ein Cograph ist, dann erfüllt σ die NSP gemäß Satz 4.10. Für den Cographen $\bar{G}$ und $\bar{\sigma}^-$ gilt dies entsprechend.
„⇐“: Angenommen G sei kein Cograph, dann existiert mindestens ein P_4 in G. Ein solcher P_4 sei $p = \{a, b, c, d\}$ und p enthalte den Knoten w in G, der die niedrigste Nummerierung, eines in einem P_4 enthaltenen Knoten, in σ besitzt. Diese vier Knoten bilden auch in $\bar{G}$ einen P_4 ($\bar{p} = \{c, a, d, b\}$) mit getauschten Mittel- und Endpunkten, siehe Abschnitt 2.3.4. Es gibt somit verschiedene Fälle zu unterscheiden, je nach der Lage von p. $S(v)$ bezeichne den minimalen Slice in σ, der p enthält und $\bar{S}(u)$ den minimalen Slice in $\bar{\sigma}^-$, der $\bar{p}$ enthält.

1. Fall: Sei $a = v$, siehe Abb. 29 Bild 1, also der erste Knoten des minimalen p enthaltenen Slices ist ein Endpunkt von p. Dann gilt: $b \in S^A(a)$, $c \in S_i(a)$, $d \in S_j(a)$ $i \neq j$, da $bc \in E \wedge bd \notin E$. Da $cd \in E$, ist es egal ob $i < j$ oder $j < i$. In jedem Fall ist ein Element aus $S_i(x) \in N^l(S_j(x))$. Also eine Verletzung der NSP.

2. Fall: Sei $v \notin p$, dann gibt es drei Möglichkeiten:
   a) $\nexists y \in p : yv \in E$, dann ist $p \in S^N(v)$ und a, b; b, c, d sowie c, d müssten jeweils in einer Zelle $S_i(v)$ sein, sonst wäre die NSP verletzt. Dann müsste aber p komplett in einer Zelle $S_i(v)$ sein, dieses aber widerspräche der Minimalität von $S(v)$.
   b) $\forall y \in p : yv \in E$, dann wäre $p \in S^A(v)$, also $S(v)$ nicht minimal.
   c) $p^* \subset p \wedge p^* \in S^A(v) \wedge p^* \neq \{b, c\}$, dann ist v aber Teil eines P_4, wie man Abb. 30 entnehmen kann. Das widerspräche der Annahme, dass w minimaler Knoten $\in P_4$ ist. Ist aber $p^* = \{b, c\}$, siehe Abb. 29 Bild 3, dann gehören a und d zwei unterschiedlichen Zellen $S_i(v)$ und $S_j(v)$ an und $S_i(v)$, $S_j(v)$ verletzen die NSP.

3. Fall: Sei $p \in S(b)$, siehe Abb. 29, Bild 2, also der erste Knoten von $S(v)$ ist ein Mittelpunkt von p, dann gibt es drei Möglichkeiten:
   a) $b = u$, dann ist b Endpunkt von $\bar{p}$ und auf $\bar{G}$ gilt der erste Fall und die NSP auf $\bar{G}$ wäre verletzt.
   b) $b \neq u$, dann gibt es wieder vier Fälle:
      i. $\forall y \in \bar{p} : yu \in \bar{E}$, dann wäre $\bar{S}(u)$ nicht minimal.

ii. $\nexists y \in \bar{p} : yu \in \bar{E}$, dann gelten analoge Überlegungen zum obigen Fall 2a.

iii. $\bar{p}^* \subset \bar{p} \wedge \bar{p}^* \in \bar{S}^A(u)$, dann wäre aber u in einem P_4 gemäß analoger Überlegungen zum obigen Fall 2c, u und p haben, wenn u in $\bar{\sigma}^-$ nummeriert wird, die gleiche nummerierte Nachbarschaft und u wird nur vor p ausgewählt, da $u <_\sigma p$. Dann darf aber annahmegemäß u in keinem P_4 enthalten sein. Wenn $\bar{p}^* = \{a, d\}$, wäre analog zum Fall 2 $p^* = \{b, c\}$ die NSP verletzt.

iv. $u \in p \wedge u \neq b$, dann würde sowohl $S(v)$ mit einem Knoten aus p beginnen, wie auch $\bar{S}(u)$. Alle Knoten in $S(v)$ haben die gleiche nummerierte Nachbarschaft, wenn v nummeriert wird, das gilt auch für $\bar{S}(u)$ und u. u wird aber als erstes von $\bar{S}(u)$ nummeriert, da u vor allen aus $\bar{S}(u)$ in σ nummeriert wurde. Daher muss gelten: Wenn $p \in S(v) \wedge p \in \bar{S}(u) \wedge v \in p \wedge u \in p \Rightarrow u = v$, ist b entweder in p oder in $\bar{p}$ Endpunkt und somit Fall 1 anwendbar.

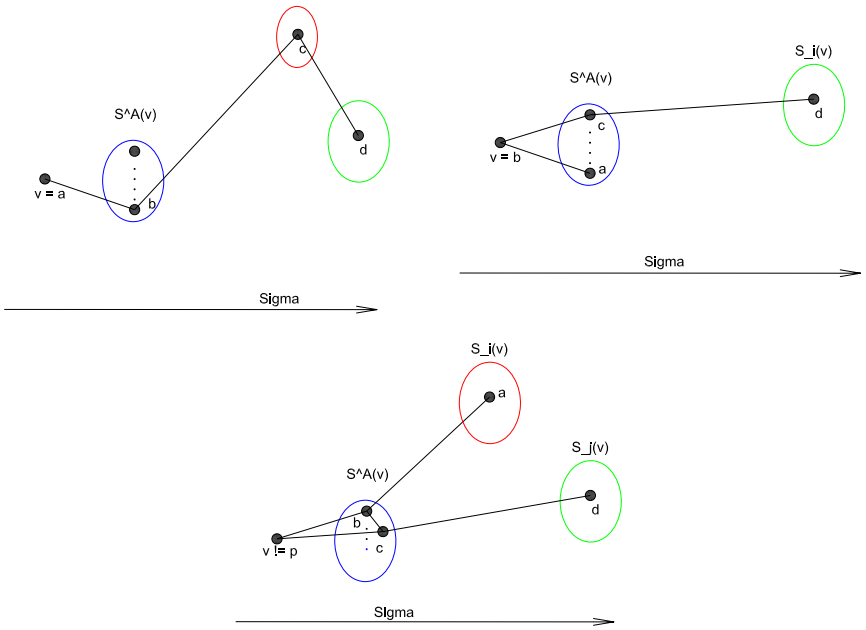

Abbildung 29: Die verschiedenen Möglichkeiten: P_4 $p = \{a, b, c, d\} \in S(v)$ die eine Verletzung der NSP bedeuten

Damit ist gezeigt, dass wenn G kein Cograph ist, zumindest die NSP für σ oder für $\bar{\sigma}^-$ nicht erfüllt ist. Damit ist Satz 4.14 bewiesen. □

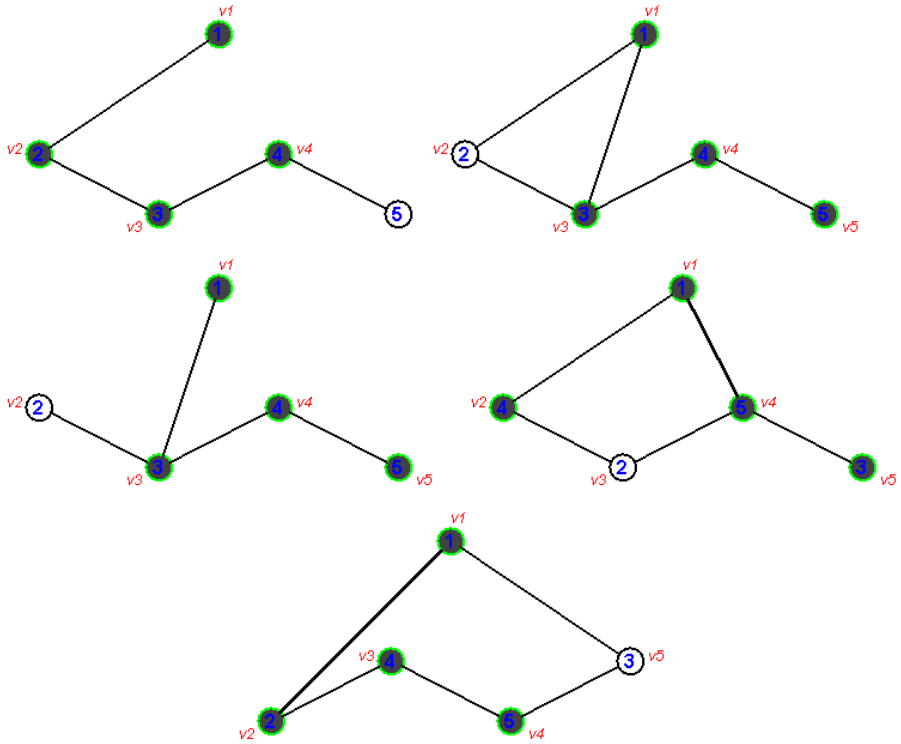

Abbildung 30: 5 Fälle $p^* \subset p \wedge p^* \in S^A(v)$ $v_1 = v, v_2 = a, \ldots, v_5 = d$

4.2.6 Ausgabe eines P_4

Nachdem zwei Subslices $S_i(v)$,$S_j(v)$ bzw. $\bar{S}_i(v)\bar{S}_j(v)$ in σ oder $\bar{\sigma}^-$ in G oder $\bar{G}$ gefunden wurden, die die NSP nicht erfüllen, soll ein P_4 in G bzw. $\bar{G}$ gefunden werden.
Es gilt folgender Satz:

Satz 4.15 (Lage des P_4) *(mit leichten Abwandlungen nach [10])*
Sei σ eine LexBFS Nummerierung von G und mindestens zwei Zellen aus $S^N(v)$ verletzen die NSP. Bezeichnen weiterhin $S_i(v)S_{i+1}(v)$ die Zellen mit niedrigsten Indices, die die NSP verletzen, dann gibt es einen der folgenden induzierten P_4 in G:

$$(vwyv_{i+1}) \vee (v_iwyv_{i+1}) \vee (yvwv_i)$$

mit v_i ist erster Knoten gemäß σ aus $S_i(v)$,
$W = \{t \in V : t \in N^l(S_i(v)) \ \wedge \ t \notin N^l(S_{i+1}(v))\}$
$Y = \{s \in V : s \notin N^l(S_i(v)) \ \wedge \ s \in N^l(S_{i+1}(v))\}$
$w \in W$, $y \in Y \wedge \forall s \in Y : s <_\sigma y$

Beweis: $W \neq \emptyset$, sonst wäre nicht $S_i(v) <_\sigma S_{i+1}(v)$. $Y \neq \emptyset$, da die NSP verletzt wurde. Betrachtet man den Ausschnitt des Graphen für die „verdächtigen“ Knoten, so sind alle offensichtlichen Kanten bzw. Nichtkanten in Abb. 31 zu ersehen.

Es gilt $wv \in E$, da i der kleinste Index ist für den die NSP in $S^N(v)$ nicht erfüllt ist, kann also w nur aus $S^A(v)$ stammen, siehe Abb. 31.

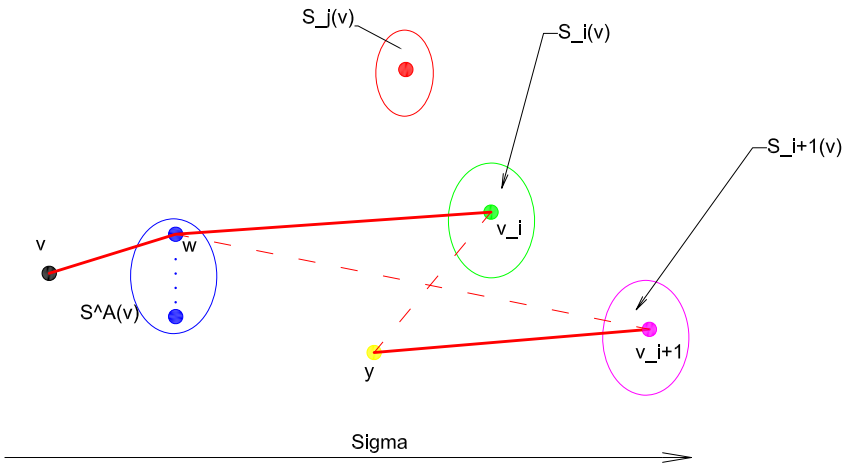

Abbildung 31: Report P_4 mit definitionsgemäßen Kanten und $wv \in E$

Es sind folgende Fälle zu unterscheiden:
Fall 1: $yv \notin E$, dann muss y aus einem SubSlice $S_j(v)$ vor $S_i(v)$ $j < i$ stammen. Da aber i minimal bezüglich der Verletzung der NSP ist und $wv_i \in E$, muss $yw \in E$ also $w \in N^l(S_j(v))$ sein. Dann bildet v, w, y, v_{i+1} einen P_4, siehe Abb. 32).
Fall 2: $yv \in E$ also $y \in S^A(v)$. Da y der am weitesten rechts stehende Knoten in σ ist, für den gilt: $y \notin N^l(S_i(v)) \ \wedge \ y \in N^l(S_{i+1}(v))$, folgt daraus, dass $v_i v_{i+1} \notin E$. Wäre $v_i v_{i+1} \in E$, dann wäre $v_i \notin N^l(S_i(v)) \ \wedge \ v_i \in N^l(S_{i+1}(v))$, dann aber widersprächen sich die Maximalität von y in Y und die Zugehörigkeit von y zu $S^A(v)$.
Gilt $wy \in E$, dann gibt es die Konstellation aus Abb. 33 und $\{v_i, w, y, v_{i+1}\} = P_4$.
Gilt $wy \notin E$, dann stellt u. a. $\{y, v, w, v_i\} = P_4$ einen P_4 dar, siehe Abb. 34.
Damit lässt sich ein einfacher Algorithmus zur Ausgabe eines P_4 konstruieren, siehe Algorithmus 12.
Sollte innerhalb von komplementären Subslices $\bar{S}_j(x)$ und $\bar{S}_{j+1}$ die NSP Verletzung bzw. CNSP Verletzung gefunden werden, dann gilt in Abwandlung von Satz 4.15 der folgende Satz:

Satz 4.16 (Lage des P_4 in $\bar{G}$) *(leicht modifiziert nach [10])*
Sei $\bar{\sigma}^-$ eine LexBFS$^-$ Nummerierung von $\bar{G}$ und mindestens 2 Subslices von $\bar{S}^N(v)$ verletzen die CNSP. Bezeichnen weiterhin $\bar{S}_j(v)\bar{S}_{j+1}(v)$ die Subslices mit niedrigsten Indices, die die CNSP 4.3 verletzen, dann gibt es einen der folgenden induzierten

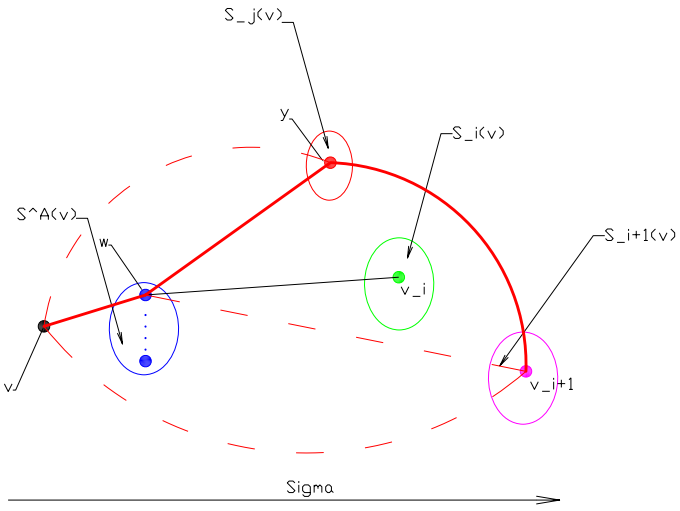

Abbildung 32: Fall 1: $yv \notin E \Rightarrow yw \in E \Rightarrow P_4 = \{v, w, y, v_{i+1}\}$

P_4 in $\bar{G}$ und damit auch in G:

$$(yw\bar{v}_{j+1}v) \vee (y\bar{v}_j\bar{v}_{j+1}w) \vee (yw\bar{v}_{j+1}v)$$

mit $\bar{v}_j$ ist erster Knoten gemäß $\bar{\sigma}^-$ aus $\bar{S}_j(v)$,
$W = \{t \in V : t \notin N^l(\bar{S}_j(v)) \cup \bar{S}_j(v) \ \wedge \ t \in N^l(\bar{S}_{j+1}(v))\}$
$Y = \{s \in V : s \notin N^l(\bar{S}_{j+1}(v)) \ \wedge \ s \in N^l(\bar{S}_j(v)) \cup \bar{S}_j(v)$
$w \in W$, $y \in Y \wedge \forall s \in Y : s <_{\bar{\sigma}^-} y$

Beweis: Läuft analog zum Beweis von Satz 4.15. □
Der Algorithmus für den komplementären Fall hat ein ähnliches Aussehen, wie Algorithmus 12. Bei der Implementierung muss vorher nur an **Report-P_4** mit übergeben werden, ob eine NSP oder CNSP Verletzung vorliegt.
Ein Beispiel: Wenn man den „Nicht-Cographen“ aus Abb. 20 betrachtet, bilden dort v_2, v_1, v_7, v_4 einen P_4. Wie wird der P_4 wirklich gefunden?
In Tabelle 16 und 17 sind die Slices für den Graphen aus Abb. 20 aufgeführt. Bei der Überprüfung der NSP kann man sich natürlich auf die Knoten v_j beschränken, bei denen sich der Slice $S^N(v_j)$ in mehr als eine Zelle aufteilt. Das ist bei dem „Nicht-Cographen“ aus Abb. 20 nur bei v_1 im komplementären Graphen der Fall. Überprüft werden muss also konkret:

$$N^l(\bar{S}_1(v_1)) \cup \{v_2\} \subset N^l(\bar{S}_2(v_1))$$

Es gilt: $N^l(\bar{S}_2(v_1)) = \{v_1, v_3, v_4\}$ und $N^l(\bar{S}_1(v_1)) = \{v_1\}$. Es ist die CNSP nicht erfüllt, da $\{v_1\} \cup \{v_2\} \not\subset \{v_1, v_3, v_4\}$. Es ergibt sich gemäß **Report-P_4**. $y = v_2; w =$

Algorithmus 12 : Report P_4

Eingabe : Subslices $S_i(v), S_{i+1}(v)$ mit verletzter NSP
Ausgabe : Eine Folge von 4 Knoten die einen P_4 bilden

```
1 Finde w, y
2 wenn yv ∉ E dann
3 |  return {vwyv_{i+1}}
4 sonst
5 |  wenn yw ∈ E dann
6 |  |  return {v_i wy v_{i+1}}
7 |  sonst
8 |  |  return {yvwv_i}
```

Algorithmus 13 : Report P_4 Complement

Eingabe : Subslices $S_i(v), S_{i+1}(v)$ mit verletzter CNSP
Ausgabe : Eine Folge von 4 Knoten die einen P_4 bilden

```
1 Finde w, y
2 wenn yv ∈ E dann
3 |  return yv v̄_{i+1} w}
4 sonst
5 |  wenn yw ∈ E dann
6 |  |  return yw v̄_i v}
7 |  sonst
8 |  |  return {y v̄_i v̄_{i+1} v}
```

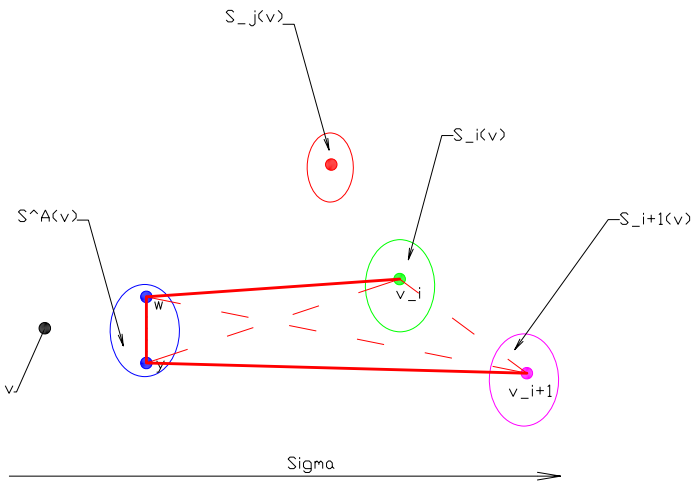

Abbildung 33: Fall 2a: $wy \in E \Rightarrow \{v_i, w, y, v_{i+1}\} = P_4$

$v_4; v = v_1; \bar{v}_i = v_2; \bar{v}_{i+1} = v_7$ Dann ergibt sich $P_4 = \{v_2 v_1 v_7 v_4\}$, da $yv \in E$. An diesem Beispiel ist beachtenswert, dass wirklich der erste Knoten jedes Slices betrachtet werden muss. Denn nur mit der lokalen Nachbarschaft von v_7 wird die Verletzung der CNSP bemerkt. Aufgrund der Tatsache, dass $v_2 v_7 \notin E$ ist, ist v_7 der erste Knoten von $\bar{S}_2(v_1)$.

4.2.7 Konstruieren des Cotree

Ausgehend davon, dass σ und $\bar{\sigma}^-$ die NSP für G und $\bar{G}$ erfüllen, es sich also bei G um einen Cographen handelt, soll der Cotree konstruiert werden. Es ist gemäß des Subslice Teilbaum Satzes (4.12) bekannt, dass für einen Cographen G und eine LexBFS Nummerierung σ von G für den Pfad des Knoten $x \in V : x = \sigma(1)$ zur Wurzel R gilt:
$\forall i = 1, \ldots, k$ $S_i(x)$ enthält genau die Blätter (Knoten) des Teilbaumes T^x_{0i}.
Analog gilt für den für den Pfad P^R_x des ersten Knotens $x = \bar{\sigma}(1)$ zur Wurzel R:
$\forall j = 1, \ldots, l$ $\bar{S}_j(x)$ enthält genau die Blätter (Knoten) des Teilbaumes T^x_{1j}.
Das Aussehen des Pfades P^x_R, also ob $P^x_R = x{-}0{-}1, \ldots, R$ oder $P^x_R = x{-}1{-}0, \ldots, R$, hängt von der Adjazenz zwischen $S_1(x)$ und $\bar{S}_1(x)$ ab. Sind die Knoten der beiden Subslices adjazent, dann müssen sich die Wege von T^x_{01} und T^x_{11} in einem 1-Knoten treffen, andernfalls in einem 0-Knoten. Sind die beiden adjazent, muss also der Pfad $P^x_R = x - 0 - 1, \ldots, R$ lauten. Der Knoten R ist ein 0-Knoten oder ein 1-Knoten, je nach der Anzahl der Subslices $S_i(x)$ und $\bar{S}_j(x)$. Wie sieht es überhaupt mit der Anzahl der Subslices von x aus? Dazu gilt folgender Satz.

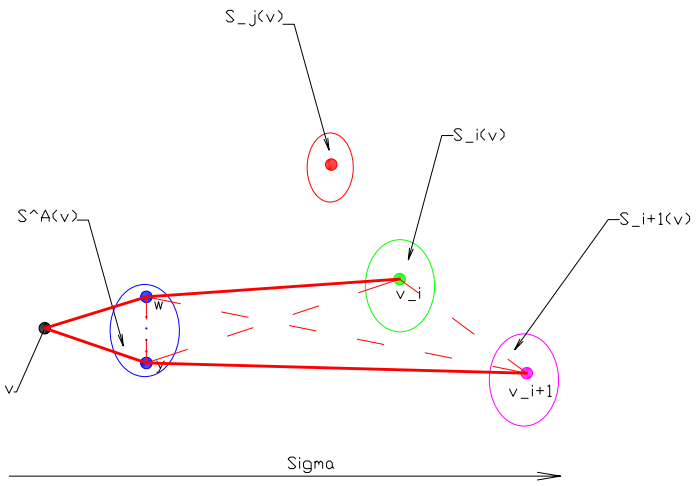

Abbildung 34: Fall 2b: $wy \notin E \Rightarrow P_4 = \{yvwv_i\}$

Hilfssatz 4.2.3 (Anzahl der Subslices von $x = \sigma(1)$)
Sei x der erste nummerierte Knoten eines Cographen G, dann unterscheiden sich die Anzahl, der Subslices $S_i(x)$ und $\bar{S}_j(x)$ um maximal 1.

Beweis: Bezeichnet man mit $S_i^A(x)$ und $\bar{S}_j^A(x)$ die Teilmengen von $S^A(x)$ bzw. $\bar{S}^A(x)$, die adjazent zu den Slices $S_i(x)$ bzw. $\bar{S}_j(x)$ sind, so gilt gemäß Satz 4.10: $S_i^A(x) \supset S_{i+1}^A(x)$ und die Relation $S_i^A(x) \rightarrow S_i(x)$ ist eine Bijektion. Bezeichnet man wiederum mit $\hat{S}_i^A(x) = S_i^A(x) \setminus S_{i+1}^A(x)$, dann ist auch die folgende Relation eine Bijektion: $\hat{S}_i^A(x) \rightarrow \bar{S}_i(x)$. Dann gilt $l = k$. Allerdings kann es auch zwei „Sonderfälle“ geben:

1. **eine** Teilmenge von $S^A(x)$ existiert, deren Knoten zu keinem Knoten aus $S^N(x)$ adjazent ist, diese Knoten bilden dann $\bar{S}_1(x)$. Dann ist $l = k + 1$.

2. Andererseits kann auch der letzte Subslice $S_k(x)$ zu keinem Knoten aus $S^A(x)$ und damit zu keinem Knoten außerhalb von $S_k(x)$ adjazent sein. Dann wäre $l = k - 1$.

Wenn es beide Sonderfälle gibt, gilt wiederum $k = l$. Zur Erläuterung von Hilfssatz 4.2.3 dienen die beiden Abb. 35 und 36.

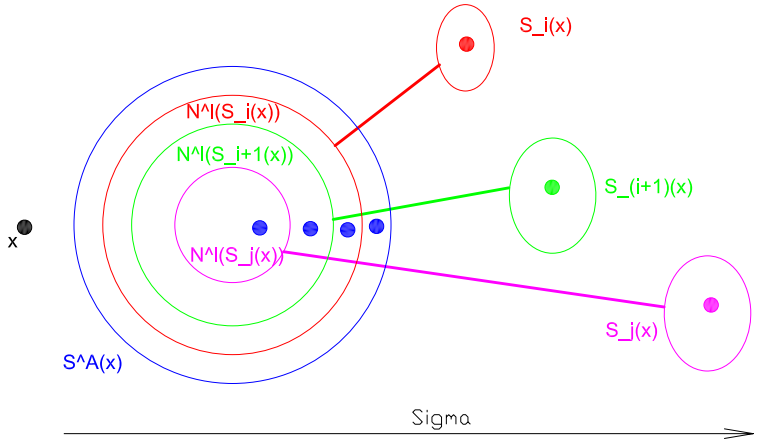

Abbildung 35: Zusammenhang zwischen $S_i(x)$ und $S^A(x)$

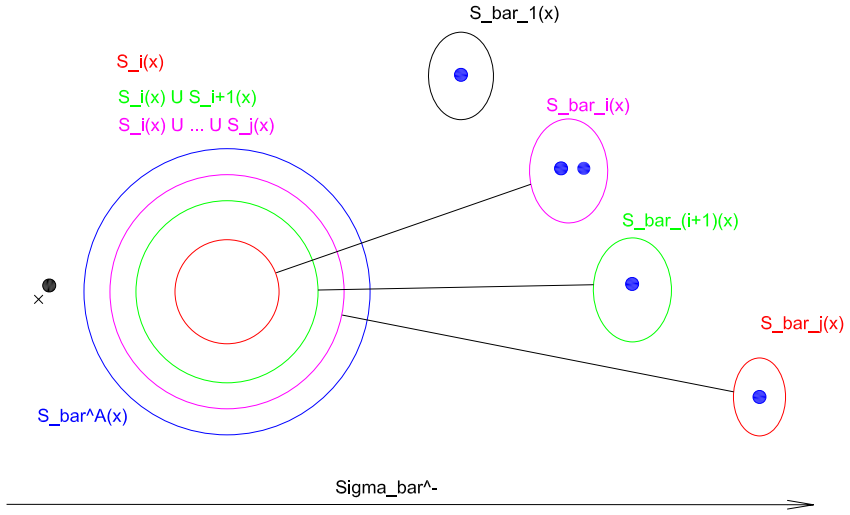

Abbildung 36: Zusammenhang zwischen $\bar{S}_j(x)$ und $\bar{S}^A(x)$. Adjazenzen aus G

Die durch die Knoten eines Subslices $S_i(x)$ und $\bar{S}_j(x)$ induzierten Teilgraphen, stellen gemäß Satz 2.9, wiederum Cographen dar. Wie kann für diese Teil-Cographen der jeweilige Cotree berechnet werden? Zunächst einmal dadurch, dass man für jeden durch einen Subslice $S_i(x)$ und $\bar{S}_j(x)$ induzierten Teil-Cographen rekursiv den Cotree bestimmt. Dazu müsste für die Knotenmenge jedes Teilbaums ein LexBFS und ein LexBFS$^-$-Sweep durchgeführt werden. Dann hätte man jeweils den Pfad des Startknotens zur Wurzel und damit ließe sich rekursiv der Cotree konstruieren, vgl. dazu Abb. 38. Dazu müssten alle Teil-Cotrees, beginnend mit den einzelnen Knoten, zum Gesamtbaum zusammengefügt werden. Wie aber gleich gezeigt wird, gibt es mittels der Subslices, die durch σ und $\bar{\sigma}^-$ für G und $\bar{G}$ berechnet wurden, eine einfache Möglichkeit den Cotree T_G zu konstruieren. Auf dem Weg vom Startknoten x zur Wurzel R des Gesamtbaumes T_G sind die durch die Knoten, der sich abwechselnden Subslices $S_i(x)$ und $\bar{S}_j(x)$, induzierten Teilbäume T^x_{0i} und T^x_{1j} angehängt. Die Knoten dieser Teilbäume bilden jeweils wieder einen Cographen. Innerhalb dieser Teilgraphen gibt es wieder einen Startknoten x^*, den mit der niedrigsten Nummerierung in σ. Die Teilbäume des Graphen aus Abb. 19 finden sich in Abb. 26 - 28. Die entscheidende Frage lautet: Enthalten die Subslices $S_i(y)$ $\bar{S}_j(y)$ mit $S(y) = S_i(x)$ und $\bar{S}(y) = \bar{S}_j(x)$ exakt die Knoten der Teilbäume T^y_{0j} und T^y_{1j}? Es sind vier Fälle zu unterscheiden:

1. Der Subslice $S_i(y)$ und ein Slice $S(y) = S_i(x) = T^x_{0i}$
2. Der Subslice $\bar{S}_j(y)$ und ein Slice $\bar{S}(y) = \bar{S}_j(x) = T^x_{1j}$
3. Der Subslice $S_i(y)$ und ein Slice $\bar{S}(y) = \bar{S}_j(x) = T^x_{1j}$
4. Der Subslice $\bar{S}_j(y)$ und ein Slice $S(y) = S_i(x) = T^x_{01}$

zu 1.) Der Subslice $S_i(y)$ ist ein Subslice des Slices $S(y) = S_i(x)$ und stellt wieder ein Modul und vor allem einen Cographen dar. Damit enthält, aufgrund von Satz 4.12 der Subslice $S_i(y)$ genau die Knoten des Teilbaumes T^y_{0i}.
zu 2.) Analog zu 1.) gilt, dass der Subslice $\bar{S}_j(y)$ ein Subslice des Slices $\bar{S}(y) = \bar{S}_j(x)$ ist und ein Modul und einen Cographen darstellt. Damit enthält aufgrund des Satzes 4.13 der Subslice $\bar{S}_j(y)$ genau die Knoten des Teilbaumes T^y_{1j}.
zu 3.) und 4.) Vergleicht man im angegebenen Beispiel aus Abb. 19 die Slices und Subslices mit dem Cotree aus Abb. 4 und den in den Teilbäumen enthaltenen Knoten, so gilt auch hier bei den „Kreuz"-Subslices das „Subset-Subtree Theorem" ,z. B. ist $S_1(v_5) = \{v_7, v_8, v_9\} = T^{v_5}_{01}$ mit $\bar{S}(v_5) = T^{v_1}_{12}$. Problematisch könnte an den Fällen 3 und 4 aber die Tatsache sein, dass die Slices $\bar{S}_j(y)$ und $S(y) = S_i(x)$ nicht in der gleichen Nummerierung entstanden sind. Damit muss der Slice $\bar{S}_j(y)$ nicht zwangsläufig eine Teilmenge des Slices $S(y) = S_i(x)$ sein. Ein Gegenbeispiel wäre z. B. der Graph G aus Abb. 37. G ist ein Cograph und $\sigma = \{v_1, v_2, v_3, v_4, v_5\}$ ist eine gültige LexBFS Nummerierung. Dann ist $\bar{\sigma} = \{v_1, v_4, v_5, v_2, v_3\}$, die sich ergebende LexBFS$^-$ Nummerierung. Und es gilt: $\bar{S}(v_4) = \{v_4, v_5\} \not\subseteq S(v_4) = \{v_4\}$.
Die Frage ist, ob allgemein gilt:

$$S(y) = S_i(x) \Rightarrow \bar{S}_j(y) = T^y_{1j} \text{ und } \bar{S}(y) = \bar{S}_j(x) \Rightarrow S_i(y) = T^y_{0i}.$$

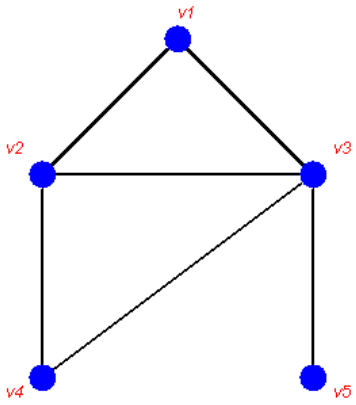

Abbildung 37: Beispiel für $\bar{S}(v) \not\subseteq S(v)$

Um dieses zu zeigen, sind einige Vorüberlegungen nötig. Es ist klar, dass die Knoten von $S_i(x)$ und $\bar{S}_j(x)$, wenn x der Startknoten ist, jeweils ein Modul bilden. Sie entsprechen den Knoten der Teilbäume T^x_{0i} und T^x_{1j} und definitionsgemäß bildet **jeder** Teilbaum ein Modul. Es lassen sich Zusammenhänge zwischen Modulen und Slices zeigen.

Hilfssatz 4.2.4 *[9] Seien σ und $\bar{\sigma}^-$ von G gegeben und ein M ein Modul von G. Sei $S(v)$ der kleinste Slice in σ, das M enthält, dann ist $v \in M$ und $M \subseteq \bar{S}(v)$. Genauso gilt: Sei $\bar{S}(y)$ das kleinste Set in $\bar{\sigma}^-$, das M enthält, dann ist $y \in M$ und $M \subseteq S(y)$.*

Beweis: Ein Modul M kann durch Aufteilung in Slices während des LexBFS-Sweep nicht geteilt werden, bevor ein Knoten $z \in M$ selbst nummeriert wird. Denn alle Knoten $z \notin M$ sind zu allen Knoten aus M adjazent oder zu keinem. Damit gilt also $v \in M$. Da v in σ als erstes von allen $z \in M$ nummeriert wird, muss das auch für $\bar{\sigma}^-$ gelten und somit auch $M \subseteq \bar{S}(v)$. □
Der Beweis für den komplementären Fall ist analog zu führen.
Die Übertragung des Ergebnisses aus 4.2.4 auf Teilbäume gelingt mit 4.2.5.

Hilfssatz 4.2.5 *[9] Seien σ und $\bar{\sigma}^-$ von G gegeben und x der erste nummerierte Knoten in T^y_{0i}, dann gilt: $T^y_{0i} \subseteq S(x)$ und $T^y_{0i} \subseteq \bar{S}(x)$. Ebenso gilt: $T^y_{1i} \subseteq S(x)$ und $T^y_{1i} \subseteq \bar{S}(x)$.*

Beweis: Klar mit 4.2.4, da die Blätter jedes Teilbaumes ein Modul induzieren. □
Mit dem nächsten Satz wird eine entscheidende Aussage über die Knoten getroffen, die zu $\bar{S}(y)$ gehören aber nicht zu $S_i(v)$.

Hilfssatz 4.2.6 *[9] Seien σ und $\bar{\sigma}^-$ von G gegeben und $S(v)$ sei ein Modul in G. Und jedes Subslice $S_i(v)$ enthalte die Knoten T^v_{0i}. Sei $\bar{S}(u)$ der minimale Slice, der $S_i(v)$ enthält. Dann ist $\bar{S}(u) - S_i(v)$ eine Untermenge von $\bar{S}^A(u)$ und $\forall z \in \bar{S}(u) - S_i(v)$ ist z unabhängig von den Knoten in $S_i(v)$.*

Beweis: Zunächst einmal gilt, da $S(v)$ ein Modul ist gemäß 4.2.4, dass $S(v) \subseteq \bar{S}(v)$. Da $S_i(v) \subseteq \bar{S}^A(v)$,vgl. Abb. 35 und 36, und $\bar{S}^A(v)$ ein Slice ist gilt:

$$S_i(v) \subseteq \bar{S}(u) \subseteq \bar{S}^A(v) \subset \bar{S}(v)$$

Da aufgrund der Minimalität von $\bar{S}(u)$ und $\bar{S}(u) \subseteq \bar{S}^A(v)$ gilt: $\forall t \in \bar{S}(u) : tv \notin E$. Betrachtet man die Menge $Q = \bar{S}(u) - S_i(v)$, so gilt für jeden Knoten $q \in Q$: $qv \notin E$. Alle Knoten aus Q müssen in Teilbäumen liegen, die Nachfolger eines 0-Knoten auf dem Pfad P_R^v sind. Allerdings nicht im Baum T_{0i}^v, denn $q \notin S_i(v)$. Damit treffen alle Knoten aus Q in einem 0-Knoten auf die Knoten aus $S_i(v)$. Also folgt: $\forall q \in Q \wedge \forall s_i \in S_i(x)$ gilt: $qs_i \notin E$. Gemäß 4.2.4 ist $u \in S_i(v)$ und alle Knoten aus Q sind daher nicht adjazent zu u, daher gilt $Q \subset \bar{S}^A(u)$. □
Für den komplementären Fall gilt:

Hilfssatz 4.2.7 *[9] Sei $S(x)$ der minimale Slice, der $\bar{S}_i(y)$ enthält. Dann ist $S(x) - \bar{S}_i(y)$ eine Untermenge von $S^A(x)$ und $\forall z \in S(x) - \bar{S}_i(y)$ ist z adjazent zu allen Knoten in $\bar{S}_i(y)$.*

Beweis: Läuft analog zu dem Beweis von 4.2.6. □
Damit lässt sich der folgende Satz 4.17 beweisen, der auch die Fälle 3 und 4 einschließt.

Satz 4.17 (Zusammenhang Teilbäume und Subslices) *[10]*
Seien σ und $\bar{\sigma}^-$ von G gegeben, dann gilt für alle Knoten $v \in V$ $S_i(v)$ enthält genau die Blätter aus T_{0i}^v und $\bar{S}_j(v)$ genau die Blätter aus T_{1j}^v .

Beweis: Die Subslices $S_i(x)$ und $S_j(x)$ induzieren Module in G, wenn x der erste Knoten in σ und damit auch in $\bar{\sigma}^-$ ist. Das ist der Induktionsanfang. Die Induktionsannahme ist: Für beliebige Knoten w innerhalb der Sweeps σ und $\bar{\sigma}^-$ gilt: $S(w)$ ist ein Modul und $S_i(w)$ enthält genau die Blätter aus T_{0i}^w und $\bar{S}_j(w)$ genau die Blätter aus T_{1j}^w. Sei v der erste Knoten aus $S_i(w)$, mit $S(v) = S_i(w)$. Gemäß Fall 1 enthält jedes Subslice $S_l(v)$ genau die Knoten der Teilbäume T_{0l}^v von T_{0i}^w . Betrachtet man zunächst die Teilbäume von T_{0i}^w, die eine 1 als Wurzel haben. Gemäß 4.2.5: gilt $S_i(w) \subseteq \bar{S}(v)$, da v der erste Knoten von $S_i(w)$ ist. Sei $X = \{x \in V | x \in \bar{S}(v) \wedge x \notin S_i(w)\}$, dann gilt gemäß 4.2.6: $X \in \bar{S}^A(v)$. Und außerdem gilt: $\forall z \in X \wedge \forall t \in \bar{S}(v) \cap S_i(w) : zt \notin E$. Da die Knoten innerhalb von X unabhängig sind von den Knoten aus $\bar{S}(v)$, erfolgt die Nummerierung $\bar{\sigma}^-$ innerhalb der Knoten des Teilbaums T_{0i}^v nur unter Berücksichtigung der Adjazenzen innerhalb von T_{0i}^v und damit lässt sich der Satz 4.13 anwenden. Daher enthält jedes Subslice $\bar{S}_j(v)$ genau die Blätter der in einem 1-Knoten wurzelnden Teilbäume T_{1j}^v von T_{0i}^w. Betrachtet man die Teilbäume von T_{0i}^w, die eine 1 als Wurzel haben. So gilt gemäß der Umkehrung von 4.13: Jeder Subslice $\bar{S}_l(v)$ enthält genau die Knoten der Teilbäume T_{1j}^v von T_{0i}^w. 4.2.5 liefert $\bar{S}_j(w) = T_{1j}^w$ und zusammen mit der Tatsache, dass v der erste Knoten von $\bar{S}_j(w)$ ist, gilt $\bar{S}_j(w) \subseteq S(v)$. Alle Knoten aus $S(v) - \bar{S}_j(w)$ gehören gemäß 4.2.7 zu $S^A(v)$ und es gilt: $\forall z \in S(v) - \bar{S}_j(w) \wedge \forall t \in \bar{S}_j(w) : zt \in E$.

Daher enthält jeder Subslice $S_i(v)$ genau die Blätter der in einem 0 Knoten wurzelnden Teilbäume T^v_{0i} von T^w_{1j} $\square$
Also lassen sich alle Teilbäume T^v_{0i} und T^v_{1j} aus den Slices und Subslices, die während σ und $\bar{\sigma}^-$ bestimmt wurden, berechnen. Dieses kann dazu verwendet werden, den Cotree rekursiv komplett zu berechnen. Damit folgt der entscheidende Satz:

Satz 4.18 (Berechnung des Cotree) *[9]*
Mit gegebenen σ und $\bar{\sigma}^-$ kann der Cotree konstruiert werden.

Beweis: Der Beweis erfolgt induktiv. Zunächst einmal ist der Satz für einen, aus einem einzigen Knoten bestehenden Cographen offensichtlich. Beginnt man mit dem ersten Knoten v aus σ und $\bar{\sigma}^-$, erhält man $S(v)$ und seine Subslice $S_i(v)$, diese enthalten genau die Blätter von T^v_{0i}. Sei w der erste Knoten von $S_i(v)$, jedes $S_j(w)$ enthält genau die Blätter von T^w_{0j} und jedes $\bar{S}_l(v)$ die Blätter von T^w_{1l}. So kann der Pfad von w zur Wurzel von T^v_{0i} konstruiert werden. Für den komplementären Fall ist die Begründung identisch: $\bar{S}_j(v)$ enthalten genau die Blätter von T^v_{1i}. Sei w der erste Knoten von $\bar{S}_j(v)$, dann enthält jedes $S_j(w)$ genau die Blätter von T^w_{0j} und jedes $\bar{S}_l(v)$ die Blätter von T^w_{1l}. So kann der Pfad von w zur Wurzel von T^v_{1j} konstruiert werden und somit induktiv alle anderen Teilbäume und damit der ganze Cotree.$\square$
Es ergibt sich folgender, rekursiver Algorithmus zur Konstruktion des Cotree T_G von G, siehe Algorithmus 14. Dieser benutzt ausschließlich die Nummerierungen σ und $\bar{\sigma}^-$ und die jeweils ersten Knoten der Slices und Subslices. Der erste Knoten des Subslices $S_i(v)$ ist mit $v[i]$ und der erste Knoten des Subslices $\bar{S}_j(v)$ mit $\bar{v}[j]$ bezeichnet.

Algorithmus 14 : CoTreeLinear nach [9]

Eingabe : Knoten v, Modul M eines Cographen G
Ausgabe : der Cotree T_G von M

wenn $v[1] = \emptyset \ \wedge \ \bar{v}[1] = \emptyset$ **dann**
  return $\{v\}$
`/* Modul hat nur einen Knoten */`
wenn $v[1] = \emptyset$ **dann**
  return $\{v; 1; CotreeLinear(\bar{v}[1], G)\}$
wenn $\bar{v}[1] = \emptyset$ **dann**
  return $\{v; 0; CotreeLinear(v[1], G\}$
wenn $v[1]\bar{v}[1] \in E$ **dann**
  return $\{v; 0; CotreeLinear(v[1], S_1(v)); 1; CotreeLinear(\bar{v}[1], \bar{S}_1(v));$
  $0; CotreeLinear(v[2]S_2(v)), \ldots)\}$
sonst
  return $\{v; 1; CotreeLinear(\bar{v}[1], \bar{S}_1(v)); 0; CotreeLinear(v[1], S_1(v)); 1;$
  $CotreeLinear(\bar{v}[2], \bar{S}_2(v)), \ldots)\}$

Die rekursive Struktur dieses Algorithmus 14 verdeutlicht nochmals Abb. 38. Die Pfade der Startknoten v_i zur jeweiligen Wurzel des Teilbaumes T^{v_i} sind nach der Rekursionstiefe farblich gekennzeichnet.

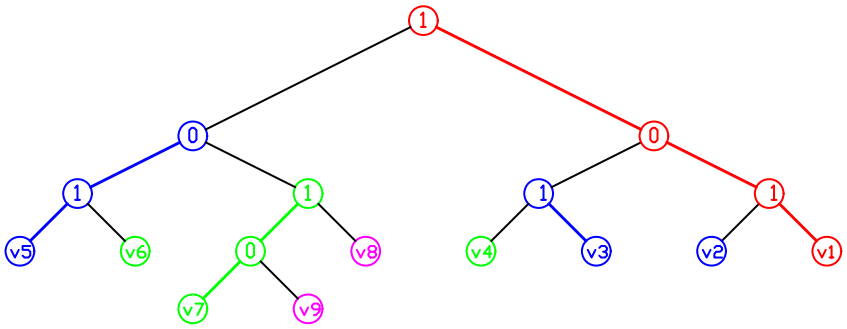

Abbildung 38: Die Pfade der Startknoten der Teilbäume zur jeweiligen Wurzel, farblich gekennzeichnet nach Rekursionstiefe.

Dieser Algorithmus aus 14 muss noch um eine Zeile erweitert werden, um die möglichen Ketten von gleichen inneren Knoten (0-Knoten oder 1-Knoten) zu verhindern. Ließe man in dem Beispielgraphen aus Abb. 19 die Kanten v_1v_2 und v_3v_4 weg, würde sich mit dem Algorithmus 14 der Cotree aus Abb. 39 ergeben. Da aber die Folgen innerer Knoten alternierend sein sollten, ist diese Korrektur angebracht. Daher wird abschließend für jeden inneren Knoten überprüft, ob er Nachfolger des gleichen Knotentyps hat. Wenn ja, werden diese Knoten zu einem Knoten zusammengefasst, der alle Nachfahren (d.h. alle äußeren Knoten und evtl. innere Knoten anderen Typs) der ursprünglichen Knoten besitzt.

4.2.8 Laufzeit des Algorithmus und Implementierungsdetails

Nachdem der Algorithmus 11 vorgestellt wurde, bedarf es einiger Überlegungen, um dessen lineare Laufzeit herzuleiten, diese wird für die einzelnen Teile von Algorithmus 11 gezeigt.
LexBFS-vorwärts und LexBFS$^-$ haben lineare Laufzeiten. Aber wie sieht es mit den anderen Schritten aus? Benötigt werden zunächst einmal alle Subslices $\forall v \in V\ \forall i : S_iv \wedge \forall j : \bar{S}_j(v)$ und deren Nachbarschaften $N^l(S_i(v))$ bzw. $N^l(\bar{S}_j(v))$. Wie können diese während LexBFS und LexBFS$^-$ gewonnen werden? Dazu betrachtet

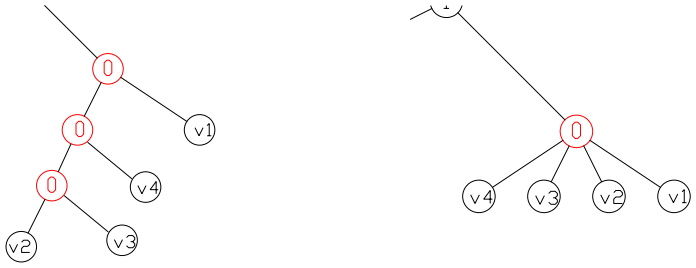

Abbildung 39: Unkorrigierter und korrigierter Teilbaum $S_1(v_1)$

man nochmals als Beispiel die entstanden Sets aus den Tabellen 10 und 11 und die daraus entstandenen Slices aus den Tabellen 14 und 15.
Man kann bei der Entstehung eines Sets verschiedene Fälle unterscheiden.
x bezeichne den Pivot-Knoten, S_A das alte Set, bevor x zum Pivot-Knoten wurde, und S^* das neue Set, das aus S_A entstanden ist, nachdem x nummeriert wurde.

1. S^* entsteht aus den Knoten aus S_A, die adjazent zu x sind, und $x \in S_A$. Das neue Set ist $S^* = S^A(x)$. Diesem Set wird als Nachbarschaft $\{x\}$ zugeordnet. Beispiel: $\{v_2, v_5, v_6, v_7, v_8, v_9\}$ bei der Nummerierung von v_1 in σ.

2. S^* entsteht aus den Knoten aus S_A die adjazent zu x sind, aber $x \notin S_A$. Das Set S^* behält den Namen von S_A und der Nachbarschaft wird $\{x\}$ angehängt. Beispiel: $\{v_2\}$ bei der Nummerierung von v_3 in $\bar{\sigma}^-$.

3. S^* entsteht aus den Knoten aus S_A, die nicht adjazent zu x sind, und $x \in S_A$. Das neue Set ist $S^* \in S^N(x)$ Diesem Set wird die leere Nachbarschaft zugeordnet. Beispiel: $\{v_3, v_4\}$ bei der Nummerierung v_1 in σ.

4. S^* entsteht aus den Knoten aus S_A die nicht adjazent zu x sind und $x \notin S_A$, Das Set S^* behält den Namen von S_A. Die Nachbarschaft bleibt unverändert. Beispiel: $\{v_3, v_4\}$ bei der Nummerierung von v_6 in σ.

Wenn ein Knoten v nummeriert wird, erhält er den Namen des Sets aus dem er entnommen wurde. Dieses wird für jeden Knoten in einem zweier Array gespeichert. In nachfolgender Tabelle 18 ist das für die Nummerierungen σ und $\bar{\sigma}^-$ und die entsprechenden Nachbarschaften aufgelistet.
Aus diesen Setnamen können die ersten Knoten jedes Subslices $S_i(v)$ und $\bar{S}_j(v)$ bestimmt werden. Dazu sammelt man hinter jedem Knoten v_i die Knoten v_j, dessen Setname $S^N(v_i)$ lautet. Diese müssen die ersten Knoten jedes Subslices $S_i(v)$ und $\bar{S}_j(v)$ sein, vgl. Tab. 14 und 15.
Um für die Subslices die NSP und CNSP zu überprüfen, müssen lediglich die lokalen Nachbarschaften der ersten Knoten jedes Subslices überprüft werden (die lokalen Nachbarschaften aller Knoten eines Slices müssen gleich sein). Die Subslices sollten

SetName	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
σ	xxx	$S^A(v_1)$	$S^N(v_1$	$S^A(v_3)$	$S^A(v_2)$	$S^A(v_6)$	$S^N(v_5)$	$S^A(v_7)$	$S^A(v_8)$
$\bar{\sigma}^-$	xxx	$\bar{S}^N(v_1)$	$\bar{S}^A(v_1)$	$\bar{S}^N(v_3)$	$\bar{S}^N(v_1)$	$\bar{S}^N(v_5)$	$\bar{S}^A(v_5)$	$\bar{S}^N(v_7)$	$\bar{S}^A(v_7)$
N^l von σ	$\emptyset$	$\{v_1\}$	$\{v_5, v_6, v_7$ $, v_8, v_9\}$	$\{v_3\}$	$\{v_2\}$	$\{v_5\}$	$\emptyset$	$\{v_7\}$	$\{v_8\}$
N^l von $\bar{\sigma}^-$	$\emptyset$	$\{v_1\}$	$\emptyset$	$\{v_3\}$	$\{v_1, v_2,$ $v_3, v_4\}$	$\{v_5\}$	$\emptyset$	$\{v_7, v_9\}$	$\emptyset$

Tabelle 18: Setnamen für σ und $\bar{\sigma}^-$ für den Graphen aus Abb. 19

natürlich in aufsteigender Reihenfolge vorliegen, damit die Nachbarschaften jedes Subslice $S_i(v)$ mit $S_{i+1}(v)$ verglichen werden können. Liegen die Knoten in sortierter Reihenfolge vor, bevor die Knoten der Subslices gesammelt werden, sind die Subslices für jeden Knoten automatisch geordnet. Die lokalen Nachbarschaften jedes Knoten liegen ebenfalls geordnet vor, da sie in der Reihenfolge der Nummerierung erstellt wurden. Die zwei lokalen Nachbarschaften $N^l(S_i(v))$ und $N^l(S_{i+1}(v))$ können durch einfaches, paralleles Durchlaufen verglichen werden. Das benötigt eine Laufzeit von $\mathcal{O}(Max(|N^l(S_i(v))|, |N^l(S_{i+1}(v)|)))$. Sobald ein Element aus $N^l(S_{i+1}(v))$ nicht in $N^l(S_i(v))$ vorkommt oder das Ende von $N^l(S_i(v))$ erreicht wurde und in $N^l(S_{i+1}(v))$ noch Elemente vorhanden sind, ist die NSP verletzt. Das Überprüfen der CNSP läuft analog ab. Die Nachbarschaften und die Subslices $\bar{S}_j(v)$ können während des LexBFS$^-$-Sweep bestimmt werden.
Sind die Anzahlen der Slices und deren Nachbarschaften begrenzt? Die Gesamtanzahl der Slices ist durch $|V|$ sowohl für σ als auch für $\bar{\sigma}^-$ nach oben begrenzt. Die lokalen Nachbarschaften $N^l(S_i(v))$ und $N^l(\bar{S}_j(v))$ enthalten maximal die Labels der den Slices zugrunde liegenden Sets. Daher gilt der folgende Satz 4.19. (Λ = Label eines Slices. λ = Label eines Knotens)

Satz 4.19 (Größe der Labels) *[10]*
$$\sum_{\forall i \in (N), x \in V} |\Lambda(S_i(x))| \leq \sum_{\forall x \in V} |\lambda(x)| \leq |E|$$

Beweis: Jede Kante $e = vw$ in G kann nur zu einem Eintrag im Label eines Sets S führen. Andererseits muss es zu jedem Element von S eine Kante des gerade nummerierten Knotens v geben, wenn v im Label von S enthalten ist. Damit ist die Summe aller Labels durch $|E|$ begrenzt. □
Entsprechend gilt das auch für die Labels der Slices im komplementären Graphen. Dort werden für die Überprüfung der CNSP auch die Labels aus G und nicht aus $\bar{G}$ verwendet. Da jede Nachbarschaft nur maximal zweimal verwendet wird: $N^l(S_i(v))$ wird mit (soweit vorhanden) $N^l(S_{i+1}(v))$ und $N^l(S_{i-1}(v))$ verglichen. Daher ist auch die Gesamtanzahl aller Vergleiche von Knoten aus lokalen Nachbarschaften durch $\mathcal{O}(|E|)$ begrenzt. Ein P_4 wird ebenfalls in linearer Zeit gefunden. Nachdem zwei Subslices $S_i(x)$ und $S_{i+1}(x)$ gefunden wurden, die die NSP verletzen, lassen sich in linearer Zeit die benötigten Knoten aus den nummerierten Nachbarschaften der Slices finden und damit der Algorithmus **report-P_4** ausführen. Sollte die Verletzung in den komplementären Slices $\bar{S}_j(x)$ und $\bar{S}_{j+1}(x)$ auftreten, ändert das an

den Laufzeitüberlegungen natürlich nichts. Wurde keine Verletzung der NSP oder CNSP gefunden und G ist demnach ein Cograph, lässt sich mittels Algorithmus 14 in linearer Zeit $\mathcal{O}(|V| + |E|)$ der Cotree konstruieren. Die ersten Knoten der Subslices $v[i]$ und $\bar{v}[j]$ $\forall v \in V$ sind bereits berechnet. Lediglich die Überprüfung von $v[1]\bar{v}[1] \in E$ muss noch erfolgen. Dieses lässt sich summiert über alle Überprüfungen in $\mathcal{O}(E)$ durchführen. Der abschließende Scan auf hintereinander folgende innere Knoten gleichen Typs kann ebenfalls in linearer Zeit erfolgen.
Somit ergibt sich:

Satz 4.20 (Komplexität Cograph Erkennung) *[11]*
Cographen können mittels LexBFS in linearer Zeit erkannt werden.

Ein fast identischer Algorithmus, der einen LexBFS-Sweep mehr benötigt, wurde u. a. von Bretscher [10] genauer beschrieben. In [10] wurde der Algorithmus für DH-Graphen, P_4-reduzierbare und P_4-sparse Graphen erweitert. Corneil, Bretscher, Habib und Paul [9] vermuten, dass man den hier vorgestellten Algorithmus mit leichten Veränderungen und Erweiterungen auch für die anderen Graphen der P_4-Hierarchie, siehe Abschnitt 4.6, verwenden kann. Außerdem nehmen sie an, dass sich dieser Algorithmus, zu einem einfachen Algorithmus zur modularen Dekomposition für allgemeine Graphen in Linearzeit erweitern lässt.

4.3 Erkennung von Intervall-Graphen

Intervall-Graphen stellen eine Teilmenge der chordalen Graphen und damit auch der perfekten Graphen dar. Für Ihre Erkennung wurden bereits einige Algorithmen vorgeschlagen. Der erste mit linearer Laufzeit stammt von Booth und Lueker [6]. Dieser Algorithmus ist aufgrund der Datenstrukturen, für die dort verwendeten PQ-Bäume, kompliziert zu implementieren. Vereinfacht wurde dieser Algorithmus von Korte und Möhring [56]. Sie benutzten einen einfachen LexBFS-Sweep und MPQ-Bäume, eine Abwandlung der PQ-Bäume. Weitere Vorschläge, LexBFS zur Erkennung von Intervall-Graphen einzusetzen, stammen von Hsu und Ma [50], die LexBFS und modulare Dekomposition benutzen, von Simon der vier LexBFS$^+$-Sweeps benutzt [69] und von Habib, McConnell, Paul und Viennot [46], die ebenfalls LexBFS und eine Verfeinerung der Partitionen benutzen. Der von Simon vorgeschlagene Algorithmus wurde mittlerweile mit Gegenbeispielen widerlegt [24]. Vorgestellt wird hier der von Corneil, Olariu und Stewart [24] vorgeschlagene Algorithmus und zwar in der Version von 2006 [26] mit **sechs** LexBFS-Sweeps. In der ursprünglichen Version aus [24] hatte der Algorithmus **vier** LexBFS-Sweeps. Später wurde der Algorithmus um einen LexBFS$^+$-Sweep erweitert [16], siehe Algorithmus 15. In [26] ist ein weiterer LexBFS$^+$-Sweep hinzugefügt worden. Corneil ist aber der Meinung [18], dass auch der LexBFS Algorithmus mit **fünf** Sweeps korrekt ist. Der Beweis liegt aber bisher nur für den Algorithmus mit **sechs** Sweeps vor.
Diese Algorithmen aus [24], [16] und [26] kommen ohne zusätzliche Datenstrukturen aus und sind daher vor dem Hintergrund der Implementierung u. a. dem von Korte und Möhring vorgeschlagenen Algorithmus vorzuziehen.

4.3.1 Der Algorithmus

Der Algorithmus aus [26] hat im Einzelnen folgendes Aussehen:

Algorithmus 15 : Intervall-Graph 6-Sweep Test

Eingabe : ein Graph $G = (V, E)$
Ausgabe : Rückgabe wahr oder falsch

1 Führe einen LexBFS vorwärts Sweep durch. Ergibt: π.
2 Führe einen LexBFS$^+$-Sweep basierend auf π durch. Ergibt $\acute{\pi}$. (Auf diesen Schritt wird in [24] verzichtet.)
3 Führe einen LexBFS$^+$-Sweep basierend auf $\acute{\pi}$ durch. Ergibt σ. (Auf diesen Schritt wird in [24] und [16] verzichtet.)
4 Führe einen LexBFS$^+$-Sweep basierend auf σ durch. Ergibt σ^+.
5 Führe einen LexBFS$^+$-Sweep basierend auf σ^+ durch. Ergibt σ^{++}.
6 Führe einen LexBFS*-Sweep basierend auf σ^+ und σ^{++} aus. Ergibt σ^*.
7 Überprüfe, ob σ^* die Bedingung aus 2.17 erfüllt. Wenn ja ist G ein IG, andernfalls nicht.

4.3.2 Laufzeit und Korrektheitsüberlegungen

Die ersten fünf Schritte haben, wie bereits gezeigt wurde, lineare Laufzeit $\mathcal{O}(|V| + |E|)$. Die lineare Laufzeit von LexBFS* ist nicht so offensichtlich. Oben wurde schon kurz angedeutet, wie sich LexBFS* in Linearzeit implementieren lässt. Eine abschließende Analyse zur Laufzeit ist in [26] veröffentlicht. Die Überprüfung der „IG-Regenschirmfreiheit" aus Satz 2.17 lässt sich, aufgrund des Zusammenhanges zu den rechtsseitigen Nachbarschaften aus Satz 2.18, in Linearzeit durchführen.
Wenn nach dem LexBFS*-Sweep die Nachbarschaften und die Liste der Knoten gemäß σ^* sortiert wurden, kann die Überprüfung, ob die rechtseitigen Nachbarschaften fortlaufend sind in $\mathcal{O}(|V| + |E|)$ erfolgen.
Zum Korrektheitsbeweis dieses Algorithmus und vor allem zum Beweis von Satz 4.21 sei an dieser Stelle auf [26] verwiesen. Der Hinweis aus [24], dass der Beweis für die Korrektheit des Algorithmus lang und kompliziert sei, hat sich mit [26] bestätigt.

Satz 4.21 (Korrektheit IG-Graph Algorithmus) *[26]*
Die letzte Nummerierung σ^* eines Graphen G erfüllt genau dann Satz 2.17, wenn G ein Intervall-Graph ist.

Eine Bestätigung dafür, dass G ein Intervall-Graph ist, in Form einer Intervall-Darstellung von G gewinnt man mittels Satz 2.19.
Mittlerweile gibt es einen von Kratsch, McConnell, Mehlhorn und Spinrad [29] entworfenen Algorithmus, der in Linearzeit Intervall-Graphen erkennt und sowohl positive als auch negative Bestätigungen liefert. Die Bestätigung dafür, dass G kein Intervall-Graph ist, wird mittels Satz 2.16 erzeugt. Man sucht also entweder einen C_n mit n ≥ 4 mittels des unter 4.1 beschriebenen Algorithmus oder ein asteroidal-triple (AT). Zum Auffinden eines AT, benutzen sie den von Korte und Möhring [56] vorgeschlagenen Algorithmus.
Inwieweit sich aus dem hier vorgestellten Algorithmus eine negative Bestätigung herleiten lässt, bleibt abzuwarten.

4.4 Erkennung von echten Intervall-Graphen

Nachdem bereits einige andere Linearzeit-Algorithmen zur Erkennung von echten Intervall-Graphen vorgeschlagen worden waren [21] [34], wurde die Möglichkeit entdeckt, LexBFS einzusetzen. Dafür wurden unterschiedliche Ansätze vorgeschlagen. Herrera, de Figueiredo, Meidanis und de Mello [33] verwenden einen einfachen LexBFS-Sweep. Dessen Ergebnis wird mittels eines inkrementellen Ansatzes überprüft. In dieser Arbeit wird der von Corneil [17] vorgeschlagene Algorithmus vorgestellt. Es sind zwar drei LexBFS-Sweeps nötig, aber die Überprüfung der Nummerierung ist erheblich einfacher als in [33]. Weiterhin ist vor dem Hintergrund der Implementierung und der bereits u. a. für die Intervall-Graphen Erkennung implementierten Varianten von LexBFS dieser Algorithmus vorzuziehen. Der Bestätigungsteil des Algorithmus stammt von Hell und Huang [48].

4.4.1 Algorithmus und Laufzeit

Der Algorithmus hat folgendes Aussehen:

Algorithmus 16 : Echter Intervall-Graph 3-Sweep Test

Eingabe : ein zusammenhängender Graph $G = (V, E)$
Ausgabe : eine Intervall-Darstellung I_G des Graphen oder ein verbotener Teilgraph H_G

Führe einen LexBFS-vorwärts Sweep durch. Ergibt: σ
Überprüfe, ob G ein chordaler Graph ist, mittels Algorithmus 8.
wenn G *chordal* **dann**
  Führe einen LexBFS$^+$-Sweep basierend auf σ durch. Ergibt σ_+.
  Teste auf Vorkommen von Fall A oder B_2. (siehe Abb. 49). **wenn** *Fall* A *oder* B_2 *auftritt* **dann**
    gib verbotenen Teilgraph aus.
  sonst
    Führe einen LexBFS$^+$-Sweep basierend auf σ_+ durch. Ergibt σ_{++}.
  wenn σ_{++} *die Nachbarschaftsbedingung 2.21 erfüllt.* **dann**
    Erzeuge eine Intervall-Darstellung I_G.
  sonst
    finde Fall A, B_1 oder B_2 und erzeuge einen verbotenen Teilgraphen.
sonst
  Erzeuge einen C_n mit $n \geq 4$ mit Algorithmus 8

Zunächst wird die Korrektheit des „Erkennungs-Algorithmus“ gezeigt. Dieser Teil besteht aus den drei Sweeps und der anschließenden Überprüfung von Bedingung 2.21.

4.4.2 Korrektheit und Laufzeit des Algorithmus

Um die Korrektheit des Algorithmus zu beweisen, sind einige Zwischenergebnisse nötig, die in den folgenden Hilfssätzen 4.4.1 - 4.4.5 festgehalten werden.
Für 4.4.1 - 4.4.5 wird vorausgesetzt, dass der Graph G ein UIG ist.

Hilfssatz 4.4.1 *nach [17]*
Wenn σ_+ mit Knoten u startet und mit Knoten v endet, dann startet σ_{++} mit v und endet mit u.

Beweis: σ_{++} startet mit v, da σ_+ mit v endet. Aber angenommen, es gäbe einen Knoten w in σ_{++} der hinter u kommt. Da $w >_{\sigma_+} u$, gilt für u, w Satz 3.6. Also es existiert ein $x \in V : xu \in E \wedge xw \notin E \wedge x <_{\sigma_{++}} u$. Wäre $v = x$, müssen alle Knoten adjazent zu u sein, da v in σ_+ die maximale Entfernung von u hat. Wenn aber alle Knoten $y \in V$ adjazent zu u sind, kann u nur letzter Knoten in σ sein, wenn G ein vollständiger Graph ist. In diesem Fall muss σ_{++} eine Spiegelung von σ_+ sein. Also

gilt: $v \neq x$. Über die Reihenfolge der Knoten u, v, w, x kann in den einzelnen Sweeps folgendes festgestellt werden.

$$v <_\sigma u \wedge w <_\sigma u \wedge x <_\sigma u$$

$$u <_{\sigma_+} w <_{\sigma_+} v \wedge u <_{\sigma_+} x <_{\sigma_+} v$$

$$v <_{\sigma_{++}} x <_{\sigma_{++}} u <_{\sigma_{++}} w$$

Aufgrund der Tatsache, dass G ein UIG ist und damit chordal, muss u in G ein simplizialer Knoten sein. Da $xu \in E \wedge xw \notin E$, kann nur $uw \notin E$. Damit ist für σ_+ die Reihenfolge für u, w, v, x komplett festgelegt:

$$u <_{\sigma_+} x <_{\sigma_+} w <_{\sigma_+} v$$

Nun gibt es zwei Möglichkeiten bezüglich der Zugehörigkeit von u und w zu verschiedenen Layern in σ_{++}. Sei $L^{++}(u) = d(v, u) = k$.

1. Sei $L^{++}(w) = k + 1$ und betrachtet man den ersten LexBFS-Sweep, von dem man weiß, dass u als letztes nummeriert wurde, gibt es folgende Möglichkeiten der Anordnung:
   a) $v <_\sigma w <_\sigma x <_\sigma u \Rightarrow d(v, w) \leq d(v, u)$ Widerspruch!
   b) $v <_\sigma x <_\sigma w <_\sigma u \; d(v, w) \leq d(v, u)$ Widerspruch!
   c) $w <_\sigma v <_\sigma x <_\sigma u \Rightarrow d(w, v) \leq d(w, u)$, da aber aus σ_+ folgt: $d(u, w) \leq d(u, v)$ Widerspruch zu $d(v, w) > d(v, u)$
   d) $w <_\sigma x <_\sigma v <_\sigma u \; d(w, v) \leq d(w, u) \leq d(u, v) > d(v, w)$ Widerspruch
   e) $x <_\sigma w <_\sigma v <_\sigma u \Rightarrow d(x, w) \leq d(x, u)$ Widerspruch!
   f) $x <_\sigma v <_\sigma w <_\sigma u \; d(x, w) \leq d(x, u)$ Widerspruch!

   Also kann w nicht in L^{++}_{k+1} liegen.

2. Sei $L^{++}(w) = k$, dann liegen w und u in L^{++}_k und sind nicht adjazent. Die Ausgangssituation ist in Abb. 40 dargestellt. Zunächst sei $k \geq 3$. Es sind wieder verschiedene Fälle zu unterscheiden.
   a) Die Knoten u und w haben einen gemeinsamen Nachbar y in L^{++}_{k-1}, dann bildet y zusammen mit seinem Nachbarn t aus L^{++}_{k-2} eine Klaue y, t, u, w, siehe Abb. 41.
   b) u und w haben keinen gemeinsamen Nachbar in L^{++}_{k-1}. Sei je ein Nachbar $x \in N(u)$ und $y \in N(w)$ aus L^{++}_{k-1} adjazent. Wenn x und y in L^{++}_{k-2} einen gemeinsamen Nachbar z hätten, bildet dessen Nachbar $r \in L^{++}_{k-3}$ (bei $k = 3$ $r = v$) zusammen mit r, x, y, u, w ein Netz, siehe Abb. 42. x und y keine gemeinsamen Nachbarn in L^{++}_{k-2}. Es entsteht in jedem Fall mit x, y und ihren Nachbarn in L^{++}_{k-2} und einem Pfad zwischen diesen beiden Nachbarn, der ausschließlich in Layern L^{++}_l mit $l < k - 1$ liegen

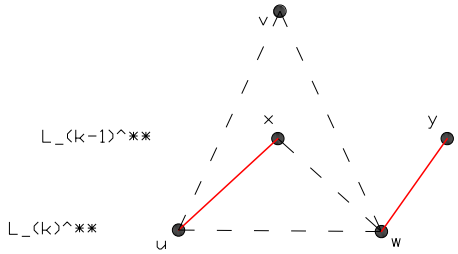

Abbildung 40: Ausgangssituation

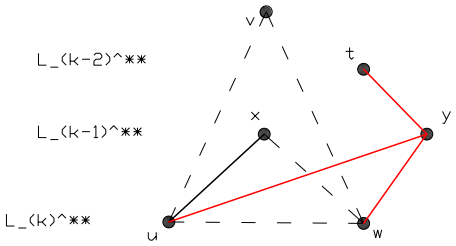

Abbildung 41: Eine Klaue

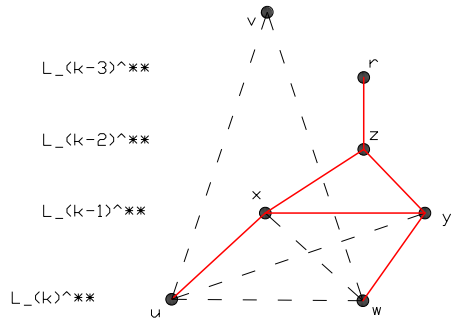

Abbildung 42: Ein Netz

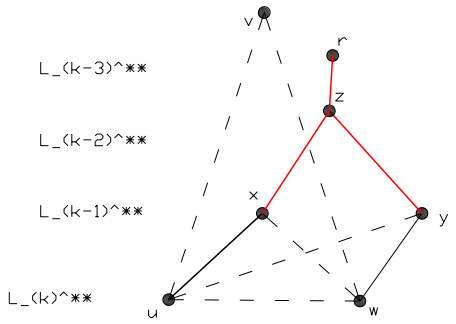

Abbildung 43: Eine Klaue, einen Layer höher

muss, ein C_n $n > 4$, siehe Abb.44. Sind aber alle Nachbarn von u und w in L_{k-1}^{++} paarweise nicht adjazent, können die Überlegungen mit den zwei nicht adjazenten Knoten x und y aus L_{k-1}^{++} durchgeführt werden, siehe Abb. 43). Man stößt stets entweder auf einen der genannten Fälle, oder aber erst der Knoten v ist ein gemeinsamer Vorfahr, siehe Abb. 45. In diesem Fall kann v aufgrund von Satz 3.7 nicht der Startknoten von σ_{++} sein.

Wenn $k = 2$ und u und w einen gemeinsamen Nachbar in L_1^{++} haben, bildet sich ein Klaue, siehe Abb. 46. Wenn es keinen gemeinsamen Nachbar in L_1^{++} gibt, ist entweder v kein möglicher Startknoten oder aber es gibt einen Knoten $z \in L_2^{++} : zu \in E \wedge zw \in E$ (siehe Abb. 47). Der Graph enthielte ein Zelt oder einen C_n mit $n \geq 4$, Siehe Abb. 48. Wenn $k = 1$ könnte v kein Startknoten sein.

Somit kann es in σ_{++} keinen Knoten w hinter u geben und der Beweis ist komplett. □

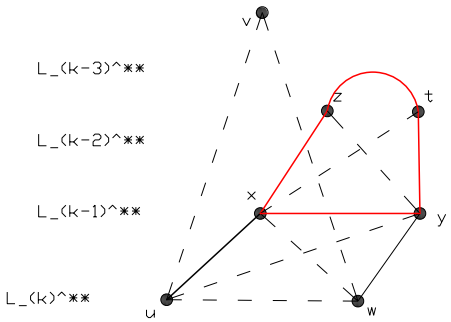

Abbildung 44: Ein Kreis

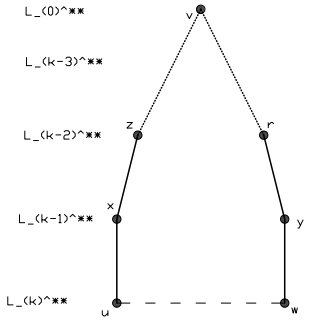

Abbildung 45: v kann nicht Startknoten sein

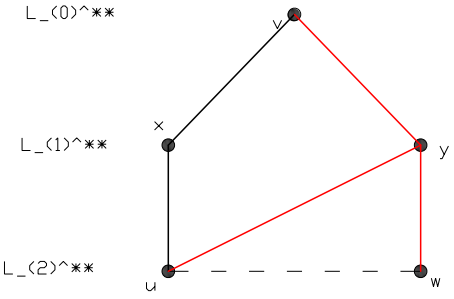

Abbildung 46: $k = 2$ eine Klaue

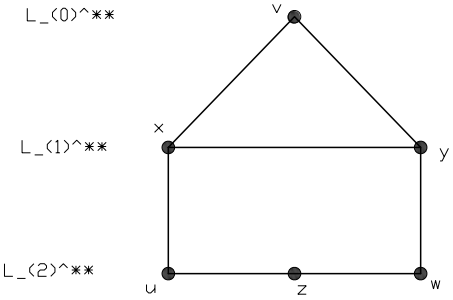

Abbildung 47: $k = 2$ ein Kreis

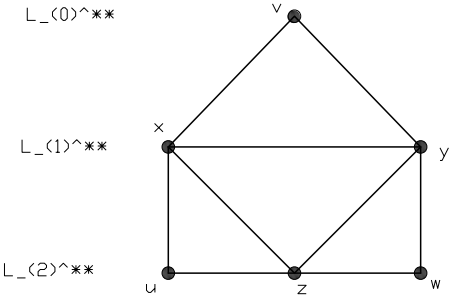

Abbildung 48: $k = 2$ ein Zelt

Nachfolgende Bezeichnungen gelten für die Hilfssätze 4.4.2 - 4.4.5. $\sigma_+(1) = u$; $\sigma_+(n) = v$; $\sigma_{++}(1) = v$; $\sigma_{++}(n) = u$. P bezeichnet einen kürzesten Pfad zwischen u und v. Es gilt: $dist(u, v) = k$.

Hilfssatz 4.4.2 *nach [21]*
Seien L_i^+ und L_i^{++} für $1 \leq i \leq k$ die Layer der beiden LexBFS$^+$-Sweeps. Dass der k-te Layer für beide LexBFS$^+$-Sweeps der letzte Layer ist, folgt aus 4.4.1. Dann gilt: Jeder L_i^+ und L_i^{++} für $1 \leq i \leq k$ bildet eine Clique.

Beweis: Man benutzt die Aussagen aus dem Beweis von 4.4.1. Dort wird gezeigt, dass zwei nicht adjazente Knoten nicht in einem Layer liegen können. Damit bildet im Umkehrschluss jeder Layer eine Clique. In diesem Teil des Beweises wird nur die Tatsache verwendet, dass σ ein LexBFS$^+$-Sweep ist. Daher gelten die dort gemachten Aussagen für σ_+ und σ_{++}. □

Hilfssatz 4.4.3 *[24]*

$$\textit{Sei der Knoten } x \in L_i^{++}, 0 \leq i \leq k \Rightarrow x \in \begin{cases} L_{k-i}^+ & \textit{wenn } x \in P \\ L_{k-i+1}^+ & \textit{wenn } x \notin P \end{cases}$$

Beweis: Wenn $x \in P$, dann gilt: $d(x, u) + d(x, v) = d(u, v) = k$. $x \in L_i^+ \Leftrightarrow d(u, x) = 1 \Leftrightarrow d(v, x) = k - i \Leftrightarrow x \in L_k - i^{++}$. Wenn $x \notin P$, gibt es ein $y \in L_i^+ : y \in P$. Somit ist gemäß Satz 4.4.2 $xy \in E$ und demnach $d(x, u) = d(y, u) \wedge d(x, v) = (y, v) + 1$ und somit $x \in L_{k-i+1}^{++}$. □

Hilfssatz 4.4.4 *[24] Seien die Knoten $x, y \in L_i^{++}$, $0 < i < k$ dann gilt:*

1. $(L_{i-1}^{++} \cap N(x)) \subseteq (L_{i-1}^{++} \cap N(y))$ *oder* $(L_{i-1}^{++} \cap N(y)) \subseteq (L_{i-1}^{++} \cap N(x))$
2. $(L_{i+1}^{++} \cap N(x)) \subseteq (L_{i+1}^{++} \cap N(y))$ *oder* $(L_{i+1}^{++} \cap N(y)) \subseteq (L_{i+1}^{++} \cap N(x))$
3. $(L_{i-1}^{++} \cap N(x)) \subset (L_{i-1}^{++} \cap N(y)) \Rightarrow (L_{i+1}^{++} \cap N(x)) \supseteq (L_{i+1}^{++} \cap N(y))$
4. $(L_{i+1}^{++} \cap N(x)) \subset (L_{i+1}^{++} \cap N(y)) \Rightarrow (L_{i-1}^{++} \cap N(x)) \supseteq (L_{i-1}^{++} \cap N(y))$

Beweis: Angenommen 1 sei nicht erfüllt, dann muss es sowohl mindestens einen Nachbar x_N von x aus L_{i-1}^{++} geben, mit $x_N \notin N(y)$, als auch einen Nachbar y_N von y mit $y_N \notin N(x)$. x_N, x, y, y_N bildeten einen C_4. Der zweite Teilsatz lässt sich analog beweisen. Zu 3 und 4: Angenommen es gäbe sowohl in L_{i-1}^{++} einen Knoten s als auch in L_{i+1}^{++} einen Knoten t mit $s, t \in N(x) \wedge s, t \notin N(y)$. In diesem Fall bildeten x, y, s, t eine Klaue.□

Hilfssatz 4.4.5 *[24] Wenn $x, y \in S \subset V$ und S sei ein Slice mit $1 < |S| < n$, dann gilt: $x <_{++} y \Leftrightarrow x >_+ y$*

Beweis: Alle Knoten eines Slices S haben die gleiche nummerierte Nachbarschaft und demnach auch die gleiche Entfernung zum Startknoten. Daher ist ein Slice S eine Teilmenge eines Layers L_i^{++} und alle Knoten des Slices sind gemäß 4.4.2 adjazent. Damit wird S beim Nummerieren seiner Knoten nicht geteilt. Es wird stets der Knoten aus S entnommen, der in σ_+ am weitesten hinten steht. Damit wird in σ_{++} die Reihenfolge von S gespiegelt. □
Nun kann der folgende Satz bewiesen werden:

Satz 4.22 *[24]*
Ein Graph G ist ein UIG genau dann, wenn σ_{++} die „Drei-Knoten Bedingung" aus 2.20 erfüllt.

Beweis: Wenn σ_{++} 2.20 erfüllt, ist klar, dass G ein UIG ist. Nun sei angenommen, G wäre ein UIG und σ_{++} erfüllt nicht 2.20. Wenn σ_{++} die „Drei-Knoten Bedingung" verletzt, dann gibt es ein Knoten Tripel x, y, z mit $x <_{++} y <_{++} z$ und $xz \in E \wedge (xy \notin E \vee yz \notin E)$. Angenommen, o. B. d. A. ist z der Knoten mit der höchsten Nummerierung in σ_{++}, die zu einem 2.20 verletzenden Tripel mit x und y gehört und es sei $x \in L_i^{++}$. Es muss, da LexBFS$^+$ auch eine BFS Nummerierung ist, gelten: $d(v,x) \leq d(v,y) \leq d(v,z)$. Wenn $d(v,x) = d(v,z)$ liegen x, y, z in einem Layer L_i^{++} und wären adjazent(4.4.2). Also muss $z \in L_{i+1}^{++}$ sein. Nun gibt es zwei Möglichkeiten für y.

1. $y \in L_{i+1}^{++} \Rightarrow yz \in E$ und auch $xy \in E$, wenn G chordal ist, gemäß 4.6. Dann bilden die Knoten x, y, z aber kein verletzendes Tripel.

2. $y \in L_i^{++}$, also $xy \in E$ $xz \in E$ $yz \notin E$.

Also ist gemäß 4.4.4 $(L_{i+1}^{++} \cap N(x)) \supset (L_{i+1}^{++} \cap N(y)) \Rightarrow (L_{i-1}^{++} \cap N(x)) \subseteq (L_{i-1}^{++} \cap N(y))$. Da aber x in σ_{++} vor y nummeriert wird, gilt $(L_{i-1}^{++} \cap N(x)) \supseteq (L_{i-1}^{++} \cap N(y))$. Also gilt $(L_{i-1}^{++} \cap N(x)) = (L_{i-1}^{++} \cap N(y))$. Da außerdem alle Elemente eines Layers adjazent sind, gilt: Es gibt einen Slice $S \in L_i^{++}$ der x und y enthält. Gemäß Satz 4.4.5 ist $y <_+ x$ und damit auch $x <_+ z$, da G chordal sein soll. Betrachtet man die Entfernungen von x und z zu u und v so gilt: $d(x,v) = i$ $d(z,v) = i+1$ $d(x,u) \leq d(z,u)$ Andererseits gilt aber: $d(x,v) + d(x,u) - d(z,v) - d(z,u) \leq 1$, wegen Satz 4.4.3. Also $x, z \in L_{k-i}^{+}$ und $x \in P \wedge z \notin P$. Es muss ein z^* geben mit $z^* \in \{L_{i+1}^{++} \cap P\} \wedge z^*x \in E$. Ebenso kann y nicht adjazent sein zu z^*, da $y \in L_i^{++}$ und demnach auch auf P liegt und damit in L_{k-i}^{+} enthalten ist, wie x und z. Dann bilden aber x, y, z eine Clique, was ein Widerspruch zu $yz \notin E$ ist. Wo aber liegt z^* in σ_{++}? Da $z \notin P \wedge z^* \in P$, muss die Nachbarschaft in L_{i+2}^{++} von z^* einen Knoten enthalten, der auf P liegt und nicht in $N(z)$ liegt. Sonst liegt z auch auf P. Also gilt: $(L_{i+2}^{++} \cap N(z)) \subset (L_{i+2}^{++} \cap N(z^*))$ daraus folgt gemäß 4.4.4

1. $(L_i^{++} \cap N(z)) \supset (L_i^{++} \cap N(z^*))$, und es muss $z <_{++} z^*$ gelten oder

2. $(L_i^{++} \cap N(z)) = (L_i^{++} \cap N(z^*))$, und z und z^* sind in einem Slice bezüglich σ_{++}. Andererseits liegt z^* in σ_+ in einem früheren Layer als z, da es im

Gegensatz zu z auf P liegt. Damit aber ist $z^* <_+ z$ und gemäß Satz 4.4.5 $z <_{++} z^*$.

Wenn aber $z <_{++} z^*$, ist das ein Widerspruch zur Maximalität von z bezüglich σ_{++}. Damit ist gezeigt, dass wenn G ein UIG ist, σ_{++} die „Drei-Knoten Bedingung“ erfüllt. □

Der Ablauf der LexBFS-Sweeps ist klar, aber wie überprüft man die Nachbarschaftsbedingung? Dazu muss für jeden Knoten $v \in V$ die Nummer des am weitesten links stehenden Nachbar $N_{wl}(v)$ und die Nummer des am weitesten rechts stehenden Nachbar $N_{wr}(v)$ in der Nummerierung σ_{++} gespeichert werden. Gibt es keinen linken bzw. rechten Nachbar, so wird $\sigma_{++}^{-1}(v)$ verwendet. Also ${\sigma_{++}}^{-1}(N_{wl}(v)) = {\sigma_{++}}^{-1}(v)$ oder ${\sigma_{++}}^{-1}(N_{wr}(v)) = {\sigma_{++}}^{-1}(v)$. Anschließend muss für jeden Knoten überprüft werden ob gilt:

$$|N[v]| = {\sigma_{++}}^{-1}(N_{wr}(v)) - {\sigma_{++}}^{-1}(N_{wl}(v)) \tag{19}$$

d. h. ob in der σ_{++} Nummerierung kein Knoten zwischen dem weitesten linken Nachbarn und dem weitesten rechten Nachbarn, der nicht in der geschlossenen Nachbarschaft $N[v]$ liegt, enthalten ist. Dann ist die Nachbarschaft $N[v]$ fortlaufend nummeriert. Die Laufzeit der drei LexBFS-Sweeps ist linear und die Überprüfung von Gleichung 19 lässt sich für jeden Knoten in konstanter Zeit ausführen. Die Initialisierung von zwei Arrays für $N_{wl}(v)$ und $N_{wr}(v)$ von allen Knoten lässt sich in $(O)(|V|)$ durchführen. Nach Abschluss des dritten Sweeps können die Nachbarschaften wieder mit Algorithmus 7 sortiert werden. Dann können $N_{wl}(v)$ und $N_{wr}(v)$ einfach entnommen und mit v bezüglich σ_{++} verglichen werden.

Der von Hell und Huang beschriebene Bestätigungsteil [48], wird in den o. g. Algorithmus eingebaut. Angenommen, a, b, c bilden ein Satz 2.20 verletzendes Tripel. In diesem Fall gibt es drei Möglichkeiten der Adjazenzen innerhalb von a, b, c. Diese drei Fälle sind in Abb. 49 dargestellt. Alle Fälle lassen sich durch die Überprüfung der schon verwendeten Bedingung: $|N[v]| \neq \sigma_{++}^{-1}(N_{wr}(v)) - {\sigma_{++}}^{-1}(N_{wl}(v))$ finden. Es wird innerhalb der Knotenmenge deren Nummerierung zwischen $\sigma_{++}^{-1}(N_{wr}(v))$ und $\sigma_{++}^{-1}(N_{wl}(v)$ liegt, ein Knoten w gesucht mit $vw \notin E$. Sollte $\sigma_{++}^{-1}(w) > \sigma_{++}^{-1}$sein, bilden $v, w, N_{wr}(v)$ ein verletzendes Tripel. Sollte $\sigma_{++}^{-1}(w) < \sigma_{++}^{-1}$ sein, ist $N_{wl}(v), w, v$ ein solches.

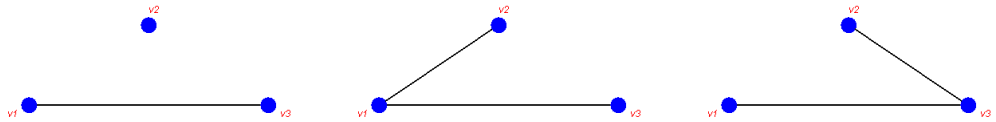

Abbildung 49: Fall A, Fall B, Fall C

Für alle drei Fälle gilt:

$$L^{++}(a) \leq L^{++}(b) \leq L^{++}(c) \ \wedge \ L^{++}(c) \leq L^{++}(a) + 1$$

Der Fall C kann in den folgenden Betrachtungen vernachlässigt werden, denn ein Auftreten von C würde bedeuten, dass G nicht chordal ist, gemäß Satz 4.6. In einer

LexBFS-Nummerierung τ kann für einen chordalen Graphen nicht Fall C auftreten, denn die gespiegelte Nummerierung von τ müsste ein PEO sein. Sei $\tau^{-1}(c) = i$, dann ist c in $G_i = (V_i, E_i)$ kein simplizialer Knoten, da $a, b \in V_i$. Fall C würde nur auftreten, wenn G nicht chordal ist. Die Nichtchordalität kann aber auch schon in σ mittels des Tests aus Abschnitt 4.1 erkannt werden. Als Bestätigung von Fall C wird ein C_n mit $n \geq 4$ ausgegeben.
Schon im Beweis der Korrektheit von Satz 4.22 und den zuvor bewiesenen Hilfssätzen ist der Hilfssatz 4.4.2 von entscheidender Bedeutung. Im Beweis von Hilfssatz 4.4.1 wird gezeigt, wie sich aus zwei nichtadjazenten Knoten eines Layers die verbotenen Teilgraphen konstruieren lassen. Dieses wird im Bestätigungsteil des Algorithmus benutzt. Für den Fall A aus Abb. 49 müssen zwei Knoten eines Layers nicht adjazent sein (a b oder b c). Aus diesen beiden Knoten kann mittels der im Beweis von 4.4.1 und 4.4.2 gezeigten Zusammenhänge eine Klaue, ein Netz oder ein Zelt konstruiert werden. Für den Fall B müssen zwei Unterfälle unterschieden werden:

1. Fall B_1: Alle Knoten eines Layers sind adjazent.

2. Fall B_2: Es gibt zwei nicht adjazente Knoten eines Layers.

Fall B_2 lässt sich genauso, wie Fall A behandeln. Aber wie ist es mit Fall B_1? Dazu sind einige Hilfssätze erforderlich.

Hilfssatz 4.4.6 *[48] In einer (Lex)BFS Nummerierung σ gibt es die Fälle A und B_2 nur genau dann, wenn es zwei nicht adjazente Knoten in einem Layer gibt.*

Beweis: Für Fall A sind entweder b und c oder a und b im gleichen Layer, da sich die Layer von a und c nur maximal um eins unterscheiden. Für Fall B_2 sind b und c im gleichen Layer. Umgekehrt gilt: gibt es zwei Knoten a und b mit $L(a) = L(b) = k \wedge \sigma^{-1}(b) > \sigma^{-1}(a) \wedge ab \notin E$, dann gibt es einen Knoten c mit $bc \in E \wedge L(c) = k-1$ und a, b, c bilden entweder Fall A oder B_2. □

Hilfssatz 4.4.7 *[48] Wenn σ_+ keinen Fall A oder B_2 enthält und $L^+(a) < L^+(b)$, dann gilt: $a >_{++} b$.*

Beweis: Angenommen es würde $b >_{++} a$ gelten. Knoten a sei aus allen Knoten w mit $L^+(w) < L^+(b)$, der Knoten mit der niedrigsten Nummerierung in σ_{++}. Da a nicht letzter Knoten in σ_+ war, kann er nicht der erste Knoten in σ_{++} sein. Gemäß 3.6 gibt es, wenn $b >_{++} a$ und $b >_+ a$ gilt, einen Knoten c mit $a >_{++} c \wedge ca \in E \wedge bc \notin E$. Da a aber der erste Knoten in σ_{++} ist, der einem früheren Layer L_+ ist als b, gilt $L_{\sigma_+}(c) \geq L_{\sigma_+}(b) > L_{\sigma_+}(a)$. Da $ac \in E$ gilt $L_{\sigma_+}(c) = L_{\sigma_+}(b)$. Außerdem ist $bc \notin E$, so dass gemäß 4.4.6 Fall A oder B_2 in σ_+ auftreten müssen. Widerspruch! □.

Hilfssatz 4.4.8 *[48] Sei G ein chordaler Graph und σ_{++} enthalte Fall B_1, wenn in σ_+ kein Fall A oder B_2 auftritt, dann enthält G eine Klaue, die a, b, c enthält.*

Beweis: Es gilt: $a <_{++} b <_{++} c$ und da Fall B_1 vorliegt auch $L^{++}(a) = L^{++}(b) = L^{++}(c-1)$. Da $bc \notin E$ gilt und außerdem in σ_+ kein Fall A oder B_2 auftritt, gilt $L^+(b) \neq L^+(c)$, so folgt aus der Umkehrung von 4.4.7, die Beziehung $L^+(b) > L^+(c)$ und damit auch $c <_+ b$. Fall C (hier zum Beispiel $c <_+ b <_+ a$) darf nicht auftreten, also gilt $a <_+ b$. Also liegt a vor b in beiden LexBFS$^+$-Sweeps und damit gibt es, gemäß 3.6, ein d mit $da \in E \wedge db \notin E \wedge d <_{++} a$. $dc \notin E$, da sonst d, b, c einen Fall C bilden würden. Dann aber bilden a, b, c, d eine Klaue. □
Mit diesen Hilfssätzen ist klar, wie man mit dem Einbau von Kontrollen nach dem ersten, zweiten und dritten LexBFS-Sweep, falls G kein UIG ist, einen verbotenen Teilgraph findet.

4.5 Erkennung von DH-Graphen

Es bestehen enge Beziehungen zwischen DH-Graphen und den chordalen Graphen einerseits und zu den Cographen andererseits. Aufgrund dieser Zusammenhänge gibt es zwei Möglichkeiten DH-Graphen mittels LexBFS zu erkennen.

1. Eine Nummerierung σ mittels eines einfachen LexBFS-rückwärts Sweep erzeugen und überprüfen, inwieweit σ eine zwei-simpliziale EO darstellt.

2. Einen dreifachen LexBFS-Sweeps auszuführen und dabei überprüfen, inwieweit diese Nummerierungen bestimmten Anforderungen genügen. Danach kann aus den gefundenen Nummerierungen und den Ergebnissen der Überprüfung je nach deren Ergebnis eine verbotener induzierter Teilgraph ausgegeben oder ein Splitbaum konstruiert werden

4.5.1 Erkennung von DH-Graphen mittels eines einfachen LexBFS-Sweep

Der Algorithmus hat folgenden Aufbau:

Algorithmus 17 : Einfacher DH-Graph LexBFS Test

Eingabe : ein Graph $G = (V, E)$
Ausgabe : wahr oder falsch

1 Erzeuge eine Nummerierung σ auf V mittels eines LexBFS - rückwärts Sweep.
2 Überprüfe für σ, ob es sich dabei um eine zwei-simpliziale EO handelt.

Dieser Algorithmus weist gegenüber dem unter 4.5.2 dargestellten folgende Vorteile auf:

1. Leichte Implementierbarkeit, wenn der Cograph Algorithmus aus 4.2 bereits implementiert ist.

2. Einfacher Beweis der Korrektheit.

Hat aber leider zwei entscheidende Nachteile:

1. quadratische Laufzeit, d.h. $\mathcal{O}(|V|^2)$

2. keine Rückgabe von Beweisen, d.h. es wird kein verbotener Teilgraph oder eine Baumdarstellung ausgegeben.

Die quadratische Laufzeit von $\mathcal{O}(|V|^2)$ ergibt sich aus den folgenden Überlegungen: Der LexBFS-Sweep hat lineare Laufzeit und das Überprüfen der einzelnen Teilgraphen von G mit Radius zwei: $D_2^G(v) = (V_v, E_v)$ hat für jeden Knoten v ebenfalls lineare Laufzeit $(|V_v| + |E_v|)$. Z. B. mittels des dargestellten Cograph Algorithmus aus 4.2. Aber für allgemeine Graphen gilt: $\sum_{v \in V} |V_v| = \mathcal{O}(|V|^2)$. Zudem liegt die Zeitkomplexität zur Bestimmung sämtlicher $D_2^G(v)$ nicht in $(|V| + |E|)$.

Im erstellten Programm wird zunächst der quadrierte Graph G^2 berechnet und aus G^2 der zu betrachtende Teilgraph G_v berechnet. Die G_vs werden mittels des 2-Sweep LexBFS Cograph-Tests untersucht. Sollte in einem der $D_2^G(v) = (V_v, E_v)$ Graphen ein P_4 gefunden werden, ist daraus auch ein verbotener Teilgraph konstruierbar.

4.5.2 Erkennung von DH-Graphen mittels eines 3-Sweeps LexBFS Tests

Dieser Algorithmus wurde von Bretscher [10] vorgeschlagen. Wichtig für sein Verständnis ist folgende Definition:

Definition 4.4 (Primary Slice)
Sei $x \in V$ der Startknoten einer LexBFS Nummerierung σ, dann werden die Slices $S^A(x) = P_0(x)$ und $S_i(x) = P_i(x)$ $i = 1, \ldots, k$ als Primary Slices P_j bezeichnet.

Der Algorithmus läuft folgendermaßen ab:

Algorithmus 18 : DH-Graph 3-Sweep Test

Eingabe : ein Graph $G = (V, E)$
Ausgabe : ein verbotener Teilgraph oder eine Splitbaum-Darstellung von G

1 Erzeuge eine Nummerierung σ mittels eines LexBFS (vorwärts) Sweep.
2 Erzeuge eine Nummerierung $\bar{\sigma}_L^-$ mittels eines LexBFS$^-$-Sweep layerweise auf dem komplementären Graphen.
3 Erzeuge eine Nummerierung σ^- mittels eines LexBFS$^-$-Sweep auf $\bar{G}$ basierend auf $\bar{\sigma}_L^-$.
4 Überprüfe mittels der gefundenen Nummerierungen, ob es sich bei den Primary Slices $P_0(x), \ldots, P_k(x)$ um Cographen handelt. **und** untersuche ob die Nummerierungen aus 2.) und 3.) die so genannte DH-Nachbarschaftsbedingung (DHN) erfüllen.
5 **wenn** *ja* **dann**
6 | Konstruiere den Splitbaum T_G^S
7 **sonst**
8 | Finde einen verbotenen Teilgraphen.

Wie ist die DHN Bedingung definiert?

Definition 4.5 (DHN-Bedingung)
Sei ein Graph G und seine Nummerierungen $\bar{\sigma}_L^-$ und σ^- gegeben, dann erfüllt G die DHN-Bedingung, wenn für jeden $P_i \in L_j$ gilt:

1. *Es gibt einen induzierten Wurzel-Teilbaum $\{t_i\}$ eines Cotrees, der einen induzierten Teilgraphen von L_{j-1} darstellt und für den gilt: $N(P) \subset \{t_i\}$.*
2. *Für alle Primary Slices $\acute{P} <_{\sigma^-} P$ gilt: $N(P) \cap N(\acute{P}) = \emptyset \vee N(P) \subseteq N(\acute{P})$.*

Es gilt folgender Satz:

Satz 4.23 (Zusammenhang DHN-Bedingung DH-Graph) *[10]*
Sei ein Graph G und seine Nummerierungen $\bar{\sigma}_L^-$ und σ^- gegeben. Genau dann ist G ein DH-Graph, wenn alle seine primary Slices P die DHN-Bedingung erfüllen und einen Cographen induzieren.

Laufzeit und Korrektheitsüberlegungen Die Laufzeit des gesamten Algorithmus 18 liegt in $\mathcal{O}(|V|+|E|)$ [10]. Die LexBFS-Sweeps haben lineare Laufzeit. Die Überprüfung, ob jeder primary Slice einen Cographen induziert, ist ebenfalls mittels Algorithmus 11 in Linearzeit möglich. Jeder Knoten kann sowohl für $\bar{\sigma}_L^-$ als auch für σ^- nur zu einem Primary Slice gehören. Überlegungen bezüglich der Laufzeit zur Überprüfung der Bedingung $N(P) \cap N(\acute{P}) = \emptyset \vee N(P) \subseteq N(\acute{P})$ lassen sich ähnlich zu den Überprüfungen der NSP und CNSP aus Abschnitt 4.2 anstellen. Die Laufzeit für das Auffinden des Teilbaumes $\{t_i\}$ ist ebenfalls linear, aber die Herleitung genauso wie der Korrektheitsbeweis des Algorithmus ist sehr umfangreich. Daher wird diesbezüglich auf [10] verwiesen. Die Konstruktion des Splitbaumes und das Auffinden einer negativen Bestätigung (House, Hole,...) laufen ähnlich wie im Algorithmus zur Erkennung von Cographen.
Dieser Algorithmus 18 ist zurzeit in der erstellten Implementierung noch nicht fertig.

4.5.3 LexBFS und Potenzen von DH-Graphen

Die LexBFS Nummerierungen von Potenzen von DH-Graphen haben interessante Eigenschaften.

Satz 4.24 (LexBFS von Potenzen von DH-Graphen) *nach [40]*
Sei G ein DH-Graph und σ eine LexBFS-(rückwärts) Sortierung von G, dann gelten folgende Zusammenhänge:

1. *σ ist ein PEO von G^{2k} $k \in \mathbb{N}$*
2. *σ ist ein PEO von G^{2k-1} $k \in \mathbb{N}$ mit der Ausnahme eines verbotenen Teilgraphen für G*

Der umgekehrte Zusammenhang gilt allgemein nicht. Nicht jeder Graph G, dessen Quadrat G^2 chordal ist, ist auch ein DH-Graph.

4.6 Weitere Graphenklassen der P_4-Hierarchie

Neben den bereits vorne beschriebenen Graphenklassen, der P_4–reduzierbaren Graphen und der P_4-spärlichen Graphen, gibt es weitere Graphenklassen, die sich ausschließlich über die Häufigkeit des Auftretens von P_4s charakterisieren lassen. Dazu gehören u. a. P_4-lite, P_4-extendible und P_4-tidy Graphen. Zu genauen Definitionen siehe [8] und die dort angegebene Literatur. Die Autoren von [9] vermuten, dass sich auf dem Cograph Algorithmus basierende Algorithmen zur Erkennung weiterer Graphenklassen aus der P_4-Hierarchie finden lassen.
Für die folgenden Algorithmen zur Erkennung der P_4-reduzierbaren und der P_4-spärlichen Graphen, gab es vor dem Einsatz von LexBFS bereits Linearzeit - Algorithmen, vgl. [52] und [54].

4.6.1 P_4–reduzierbare Graphen

Der im Folgenden skizzierte Algorithmus stammt ebenfalls von Bretscher [10].

Algorithmus 19 : P_4-reduzierbarer Graph 4-Sweep Test

Eingabe : ein Graph $G = (V, E)$
Ausgabe : Zwei P_4 mit einem gemeinsamen Knoten v oder eine Baum-Darstellung von G

1 Erzeuge eine Nummerierung σ mittels eines LexBFS vorwärts Sweep.
2 Erzeuge eine Nummerierung σ^+ mittels eines LexBFS$^+(G, \sigma)$ Sweep.
3 Erzeuge eine Nummerierung $\bar{\sigma}^-$ mittels eines LexBFS$^-(G, \sigma^+)$ Sweep.
4 Erzeuge eine Nummerierung σ^- mittels eines LexBFS$^-(G, \bar{\sigma}^-)$ Sweep.
5 Untersuche, ob $x = \sigma^-(1)$ die so genannte P_4-reduzierbare Nachbarschaftsbedingung (engl. P_4-reducible neighbourhood property = P_4-RNP) erfüllt.
6 **wenn** *ja* **dann**
7 | Konstruiere den P_4R-Baum
8 **sonst**
9 | Gib zwei P_4 mit einem gemeinsamen Knoten v zurück.

Wie ist die P_4-RNP definiert?

Definition 4.6 (P_4-reduzierbare Nachbarschaftsbedingung)

Sind die Nummerierungen σ^+, $\bar{\sigma}^-$ und σ^- gegeben, dann erfüllt ein Knoten x die P_4-RNP, wenn eine der folgenden Bedingungen gilt:

1. *$S(x)$ und $\bar{S}(x)$ erfüllen die Cograph - NSP und alle $S_i(x)$ und $\bar{S}_j(x)$ erfüllen die P_4-RNP.*

2. *a) $S(x)$ erfüllt die Cograph - NSP mit Ausnahme von $N^l(S_2(x)) \not\subset N^l(S_1(x))$.*

b) Es gibt Knoten $a, b, c \in V$ mit: $a \in S^A(x)$, $b \in S_1(x)$, $c \in S_2(x)$ so dass gilt:

i. $N^l(S_1(x)) = S^A(x) = a \cup S^{A(a)} \wedge\ N^l(S_2(x)) = S^A(x) - \{a\} \cup \{b\}$.

ii. $S_2(x) = c \wedge\ c \cup S_1(x) - b \subseteq N(b)$.

c) $\bar{S}(x)$ erfüllt die Cograph - NSP.

d) Für alle i, j erfüllen die ersten Knoten von $S_i(x)$ und $\bar{S}_j(x)$ die P_4-RNP.

Satz 4.25 (Zusammenhang $P_4 - RNP$ P_4-reduzierbare Graphen) *nach [10] Genau dann, wenn der erste Knoten von σ^- die P_4-RNP erfüllt, ist G ein P_4-reduzierbarer Graph.*

Laufzeit und Korrektheitsüberlegungen Für den Korrektheitsbeweis sei auf [10] verwiesen. Die Laufzeit dieses Algorithmus ist $\mathcal{O}(|V| + |E|)$. Die LexBFS-Sweeps haben lineare Laufzeit und auch die Überprüfung der P_4-RNP lässt sich in Linearzeit durchführen. Gleiches gilt für die Konstruktion des P_4R-Baumes und die Rückgabe von zwei P_4 mit einem gemeinsamen Knoten v.
Implementiert wurde dieser Algorithmus ohne Bestätigungsteil.

4.6.2 P_4-spärliche Graphen

Auch der folgende Algorithmus, der dem unter 4.6.1 vorgestelltem sehr ähnlich ist, stammt von Bretscher [10]. Er benutzt die alternative, rekursive Charakterisierung von P_4-spärlichen Graphen gemäß Satz 2.26.
Der Algorithmus hat folgendes Aussehen:

Algorithmus 20 : P_4-spärlicher Graph 4-Sweep Test

Eingabe : ein Graph $G = (V, E)$
Ausgabe : ein P_4SBaum oder ein Teilgraph H mit fünf Knoten, der mindestens zwei P_4s enthält

```
1 Erzeuge eine Nummerierung σ mittels eines LexBFS vorwärts Sweep.
2 Erzeuge eine Nummerierung σ⁺ mittels eines LexBFS⁺(G, σ) Sweep.
3 Erzeuge eine Nummerierung σ̄⁻ mittels eines LexBFS⁻(G, σ⁺) Sweep.
4 Erzeuge eine Nummerierung σ⁻ mittels eines LexBFS⁻(G, σ̄⁻) Sweep.
5 Untersuche, ob x = σ⁻(1) die so genannte P4-sparse Nachbarschaftsbedingung
  (engl. P4-sparse neighbourhood property = P4-SNP) erfüllt.
6 wenn ja dann
7 |  Konstruiere den P4S-Baum
8 sonst
9 |  gib einen induzierten Teilgraphen H von G aus H = (V_H, E_H) ∧ |V_H| = 5,
  |  der mindestens 2 verschiedene P4s enthält.
```

Wie ist die P_4-spärliche Nachbarschaftsbedingung (P_4-SNP) definiert?

Definition 4.7 (P_4-SNP)
*Ein Knoten v erfüllt die P_4-SNP, wenn er entweder die **Thin Spider Property (TnSP)** oder die **Thick Spider Property (TkSP)** erfüllt.*

Definition 4.8 (Thin Spider Property (TnSP))
Sei G ein Graph die Nummerierungen aus Algorithmus 20 gegeben, dann erfüllt ein Knoten x die TnSP, wenn eine der folgenden Bedingungen gilt:

1. *$S(x)$ und $\bar{S}(x)$ erfüllen die Cograph-NSP und die ersten Knoten aller $S_i(x)$ und $\bar{S}_j(x)$ erfüllen die P_4-SNP.*
2. *oder alle folgenden Bedingungen:*
   a) *$S(x)$ erfüllt die Cograph-NSP mit Ausnahme von allen*
   $$i, 2 \leq i \leq k, N^l(S_i(x)) \not\subset N^l(S_{i-1}(x))$$
   b) *Es gibt Knoten $x_1, \ldots, x_k \in S(x)$ und $y \in S^A(x), y_1, \ldots, y_k \in S_1(x)$, so dass für $i \geq 1$.*
      i. *$S_i(x) = x_i$ und $N^l(S_i(x)) = S^A(x) - y \cup y_i$.*
      ii. *$N^l(S_i(x)) = S^A(x) = y \cup S^A(y)$ und $S_1(x) - y \subset N(y)$.*
   c) *$\bar{S}(x)$ erfüllt die Cograph-NSP.*
   d) *Seien $S^*(x)$ und $\bar{S}^*(x)$ die Slices, die durch Entfernen von x_i und y_i für $i = 1, \ldots, k$ entstanden sind. Für $j \geq 1$ erfüllt der erste Knoten von jedem Subslice $S^*_j(x)$ von $S^*(x)$ und $\bar{S^*}_j(x)$ von $\bar{S}^*(x)$ die P_4-SNP.*

Definition 4.9 (Thick Spider Property (TkSP))
Sei G ein Graph die Nummerierungen aus Algorithmus 20 gegeben, dann erfüllt ein Knoten x die TnSP, wenn eine der folgenden Bedingungen gilt:

1. *$S(x)$ erfüllt die Cograph-NSP und die ersten Knoten aller $S_i(x)$ und $\bar{S}_j(x)$ erfüllen die P_4-SNP*
2. *oder alle folgenden Bedingungen:*
   a) *$S(x)$ erfüllt die Cograph-NSP mit Ausnahme von allen*
   $$i, 1 \leq i \leq k, N^l(S_i(x)) \not\subset N^l(S_{i-1}(x))$$
   b) *Es gibt Knoten $x_1, \ldots, x_k \in S(x)$ und $y \in S^A(x), y_1, \ldots, y_k \in S_1(x)$, so dass für $i \geq 1$*
      i. *$S_i(x) = x_i$ und $N^l(S_i(x)) = S^A(x) - y \cup y_i$ mit $y_i = \bar{S}_{k-i+1}(x)$.*
      ii. *$N^l(S_i(x)) = S^A(x)$ und $S^A(x) \cup S_1(x) - y_i \subset N(y_i)$.*
   c) *$\bar{S}(x)$ erfüllt die Cograph-NSP mit Ausnahme von $1 \leq i \leq k$ $N^l(\bar{S}_{k-i}(x)) \cup \bar{S}_{k-i}(x) - x_i \cup x_{i+1} = N^l(\bar{S}_{k-i+1}(x))$*

d) Seien $S^*(x)$ und $\bar{S}^*(x)$ die Slices, die durch Entfernen von $x_i = S_i(x)$ und $y_i = \bar{S}_{k-i+1}(x)$ für $i = 2, \ldots, k$ entstanden sind. Dann erfüllt der erste Knoten von jedem Subslice $S_i^*(x)$ von $S^*(x)$ und $\bar{S}^*{}_j(x)$ von $\bar{S}^*(x)$ die P_4-SNP.

Laufzeit und Korrektheitsüberlegungen Für den Korrektheitsbeweis sei auf [10] verwiesen. Die Laufzeit dieses Algorithmus ist $\mathcal{O}(|V| + |E|)$. Die LexBFS-Sweeps haben lineare Laufzeit und auch die Überprüfung der P_4-SNP lässt sich in Linearzeit durchführen. Gleiches gilt für die Konstruktion des P_4S-Baumes und die Rückgabe des induzierten Teilgraphen mit zwei verschiedenen P_4s.
Die Implementierung dieses Algorithmus ist im erstellten Programm noch nicht fertig.

4.7 Erkennung weiterer Graphenklassen

4.7.1 Bipartite Permutationsgraphen

Der im Anschluss vorgestellte Algorithmus stammt von Hell und Huang [48] und ist dem Algorithmus aus Abschnitt 4.4 sehr ähnlich.

Algorithmus 21 : Unit Interval BiGraph 3-Sweep Test

Eingabe : ein zusammenhängender Graph $G = (V, E)$
Ausgabe : eine Intervall-Darstellung I_G des Graphen oder ein verbotener Teilgraph H_G

Führe einen LexBFS-vorwärts Sweep durch. Ergibt: σ
Führe einen LexBFS$^+$-Sweep basierend auf σ durch, ergibt σ_+
Teste auf Vorkommen von Fall D oder E_1 (siehe Abb. 50 - 52), also Verletzungen der Bedingung aus 2.28
wenn *Fall D oder E_1 auftritt* **dann**
  Gib verbotenen Teilgraph aus
sonst
  Führe einen LexBFS$^+$-Sweep basierend auf σ_+ durch, ergibt σ_{++}.
wenn *Fall D oder E_1 auftritt* **dann**
  Gib verbotenen Teilgraph aus
sonst
  erzeuge eine Intervall-Darstellung I_G.

Laufzeit und Korrektheitsüberlegungen Dieser Algorithmus ist dem Algorithmus 16 zur Erkennung von UIGs sehr ähnlich. Er hat auch eine Laufzeit von $\mathcal{O}(|V|+|E|)$. Die LexBFS-Sweeps haben lineare Laufzeit und auch das Auffinden der Fälle E und D lässt sich in Linearzeit durchführen. Dabei ist wichtig, dass der Algorithmus nur auf bipartite Graphen anwendbar ist. Also muss vorher bekannt sein, ob G bipartit ist oder es muss während des Algorithmus überprüft werden. Das lässt sich während des ersten LexBFS-Sweep feststellen, indem man die Entfernung vom Startknoten, jedem Knoten $v \in V$ als Parameter $L(v)$ mitgibt. Anschließend wird überprüft, ob für jede Kante $vw \in E$ gilt: $|L(v)-L(w)| = 1$. Sollte G nicht zusammenhängend sein (für Algorithmus 22 muss G zusammenhängend sein), lässt sich diese Überprüfung für jede Komponente von G einzeln durchführen. Die Überprüfung, ob G bipartit ist, hat auch lineare Laufzeit und daher bleibt die Zeitkomplexität des gesamten Algorithmus in $\mathcal{O}(|V| + |E|)$. Die Überlegungen zur Korrektheit des Algorithmus sind ähnlich zum Fall der einheitlichen Intervall-Graphen (vgl. 4.4).

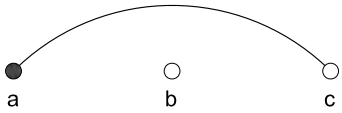

Abbildung 50: Fall D

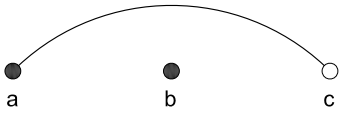

Abbildung 51: Fall E

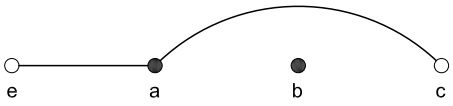

Abbildung 52: Fall E_1

4.7.2 Erkennung von stark chordalen Graphen

Auch zur Erkennung von stark chordalen Graphen wurden bereits mehrere Algorithmen vorgeschlagen, Siehe [55] für Details. Die bis dato beste Zeitkomplexität lag bei $\mathcal{O}(|E|log(|V|))$ für allgemeine Graphen. Prasad und Kumar verwenden als erste LexBFS zur Erkennung. Deren Algorithmus hat folgendes Aussehen:

Algorithmus 22 : Strongly Chordal Graph 2-Sweep Test

Eingabe : Ein Graph $G = (V, E)$
Ausgabe : eine SEO des Graphen oder $false$

Führe einen LexBFS-rückwärts Sweep durch. Ergibt: σ_{LexBFS}.
`/* (Alternativ: Führe einen MCS-Sweep durch. Ergibt:` σ_{MCS}`. */`
Überprüfe ob G chordal ist
wenn G *ist chordal* **dann**
  Berechne die Anzahl der Separatoren zu jeder Kante.
  Führe einen LexBFS-SEO Sweep durch. Ergibt σ_{SEO}.
  Überprüfe ob σ_{SEO} eine SEO darstellt (*).
  wenn σ *ist SEO* **dann**
    return σ
  sonst
    return falsch
sonst
  return falsch

Laufzeit und Korrektheitsüberlegungen Die Laufzeit des gesamten Algorithmus wird durch die Laufzeit von $\mathcal{O}(k^2n)$ des LexBFS-SEO Sweep dominiert. Die Überprüfung (*), ob σ_{SEO} eine SEO darstellt, kann mittels des von Lubiw [62] vorgeschlagenen Algorithmus in Linearzeit $\mathcal{O}(|V|+|E|)$ durchgeführt werden. Zu Korrektheitsüberlegungen sei auf [55] und [62] verwiesen. In [55] wird gezeigt, dass folgender Satz gilt:

Satz 4.26 *[55]*
Ein Graph G ist genau dann ein stark chordaler Graph, wenn die mittels LexBFS-SEO gewonnen Nummerierung σ_{SEO} ein SEO darstellt.

4.7.3 Schnittmengen von Graphenklassen und weitere „verdächtige“ Graphenklassen

Aus den bereits erläuterten Algorithmen lassen sich durch Kombination, natürlich auch die Schnittmengen von Graphenklassen erkennen. Z. B. P_4-freie chordale Graphen können mittels einer Kombination aus den erwähnten Algorithmen für Cographen und chordale Graphen erkannt werden. Eine positive oder negative Bestätigung ihrer Zugehörigkeit könnte ebenfalls geliefert werden.

Graphenklasse	Algorithmus	Genauigkeit	Referenz
DH-Graph	zweifach	$= D_G$	[37]
chordaler Graph	einfach	$\geq D_G - 1$	[36] [38]
AT-freier Graph	einfach	$\geq D_G - 1$	[36] [38]
Intervall-Graph	einfach	$= D_G$	[38]
allgemeine Graphen	einfach	$\geq \frac{D_G}{2}$	

Tabelle 19: Durchmesserbestimmung bestimmter Graphenklassen.

Neben denen bereits erwähnten Graphenklassen, gibt es weitere, von denen angenommen wird, dass sie mittels LexBFS und seiner Varianten erkennbar sind.
Dazu gehören u. a. die cocomparability Graphen [25] und weitere Graphenklassen der P_4-Hierarchie [9].

4.8 Andere Anwendungen von LexBFS

4.8.1 Bestimmen des Durchmessers eines Graphen

Zu den grundlegendsten Graphproblemen gehört das Bestimmen des Durchmessers eines Graphen. Für dieses Problem gibt es bisher keine schnellere und für alle Graphen anwendbare Methode, als die Bestimmung der Distanzmatrix und die Bestimmung des höchsten Eintrags innerhalb dieser Matrix [19]. Die Distanzmatrix lässt sich zum Beispiel mittels eines BFS-Sweeps von jedem Knoten aus bestimmen. Jeder BFS-Sweep bestimmt eine Zeile der Distanzmatrix, so dass eine Zeitkomplexität von $\mathcal{O}(|V|\cdot(|V|+|E|))$ bzw. $\mathcal{O}(|V|\cdot|E|)$ erhalten. Für schwach besetzte Graphen wurden Algorithmen mit günstigerer Zeitkomplexität vorgeschlagen, die aber auch wenig praxistauglich sind [20]. Für einen Überblick über das Problem der Durchmesserbestimmung von Graphen vgl. den Aufsatz von Zwick [76]. Hat man einen Knoten v mit maximaler Exzentrizität innerhalb eines Graphen bestimmt oder ist dieser Knoten bekannt, kann mittels eines BFS-Sweep $ecc(v)$ und damit $D(G)$ bestimmt werden. An dieser Stelle setzt LexBFS an. Für jeden Graphen liefert ein LexBFS-Sweep einen letztnummerierten Knoten mit hoher Exzentrizität. Für bestimmte Graphenklassen ließ sich mittels eines einfachen LexBFS-Sweep eine sehr gute Abschätzung des Durchmessers erhalten, für andere Graphenklassen bedarf es eines zweifachen Sweeps, wobei der zweite mit dem letzten Knoten des ersten Sweep gestartet wird. Es wird anhand der Exzentrizität des letzten Knotens eine Abschätzung für den Durchmessers des Gesamtgraphen vorgenommen. In Tabelle 19 sind für einzelne Graphenklassen die mittels LexBFS möglichen Abschätzungen des Durchmessers aufgelistet. Es sind die Algorithmen, ob einfacher oder doppelter Sweep, und die erstmalige Erwähnung dieser Abschätzung aufgeführt.
Die Genauigkeit bezieht sich auf das Verhältnis der Exzentrizität des letzt besuchten Knoten des letzten LexBFS-Sweep im Vergleich zum tatsächlichen Durchmesser $D(G)$ des Graphen G.

Für AT-freie Graphen ist zwar die Abschätzung des Durchmessers mittels LexBFS in Linearzeit möglich, da aber bisher die beste Laufzeit zur Erkennung von AT-freien Graphen von $\mathcal{O}(n^3)$ [23] hat, kann die Zeitkomplexität der Durchmesserberechnung nur verbessert werden, wenn bereits bekannt ist das G AT-frei ist. Die Schranke kann für bestimmte chordale Graphen noch verbessert werden, siehe [19].
Für einzelne Graphenklassen haben Corneil, Dragan und Köhler [20] nachgewiesen, dass mittels BFS-Sweeps anstelle von LexBFS-Sweeps, Ergebnisse mit gleicher bzw. nur etwas geringerer Genauigkeit erzielen lassen. Ferner wurde von Corneil, Dragan, Habib und Paul [19] gezeigt, dass auch mittels Erhöhung der Anzahl einfacher LexBFS-Sweep keine Steigerung der Genauigkeit bei bestimmten Graphenklassen erreicht werden kann. Inwieweit sich mittels Kombinationen, z. B. Hintereinanderschaltung, unterschiedlicher LexBFS Varianten, wie auch schon bei der Graphenklassenerkennung auch für die Durchmesserbestimmung neue Erkenntnisse gewinnen lassen, bleibt eine offenes Feld für weitere Forschungen.

4.8.2 Weitere Anwendungen von LexBFS

Es sind bisher einige weitere Anwendungen von LexBFS untersucht bzw. gefunden worden. Dazu gehört u. a. die Bestimmung von Dominating Pairs in AT-freien Graphen [14]. Diese lassen sich mittels eines zweifachen LexBFS-Sweep ähnlich der Durchmesserbestimmung ermitteln. Der zweite Sweep wird auch mit dem letztnummerierten Knoten des ersten Sweep begonnen. Das dominierende Paar besteht aus dem jeweils letztnummerierten Knoten der beiden Sweeps.
Berry und Bordat [4] haben gezeigt, dass es auch interessante Eigenschaften von LexBFS Sortierungen, speziell der letzt nummerierten Knoten, in allgemeinen Graphen gibt.

5 Die Implementierung

Die erstellte Implementierung LexBFS besteht aus Java Klassen, die als jar-Datei ausführbar sind. Alle verwendeten Java Klassen und Funktionen sind unter Java 1.4.x lauffähig. Der Quellcode wurde mittels Eclipse 3.1 bzw. 3.2 erstellt. Alle Einzelheiten der Implementierung sind sicherlich nicht von Relevanz und selbst eine Auswahl aller wichtigen Punkte würde den Rahmen dieser Arbeit sprengen. Daher kann nur auf die zentralen Punkte eingegangen werden. Der Download der jar-Datei kann von **www.rwev.de/web-content/LexBFSFrames.html** erfolgen.
Für alle Algorithmen wurde die gleiche Datenstruktur für die Graphdarstellung gewählt, das ist für einzelne Algorithmen unter dem Aspekt der Laufzeit nicht die optimale Lösung, siehe unter Punkt 5.5 Ausbaumöglichkeiten.
Es gibt immer einen aktuellen Graphen $G_{akt} = (V_{akt}, E_{akt})$, dieser wird zunächst mittels **Neuen Graph eingeben** festgelegt. Bevor ein anderer Graph betrachtet werden kann, muss stets mit **Laufende Berechnung stoppen** G_{akt} gelöscht werden.
Graphen deren Knotenanzahl, die den in der Main-Class ***LEXBFSAppl*** festgelegten Parameterwert ***MaxVerticesToDisplay*** nicht überschreiten, werden auf der Graphischen Benutzeroberfläche (GUI = Graphic User Interface) dargestellt. Standardmäßig ist dieser Parameter auf 50 eingestellt. Die Grenze der maximalen Anzahl an Knoten eines Graphen ist mit dem Parameter ***MaxVerticesToCalculate*** festgelegt. Versuche mit Speichergrößen von 1,2 GB als maximalen Heap-Speicher für die Java Virtual Machine haben ergeben, dass sich Graphen mit 2 Mio. Knoten und 12,8 Mio. Kanten noch bearbeitet werden können. Diese Graphen waren mittels der weiter unten beschriebenen Funktion **zufälliger Graph erzeugen** erstellt worden. Evtl. lassen sich größere Graphen verarbeiten, wenn diese nur eingelesen und nicht erst erzeugt werden müssen.

5.1 Der Funktionsumfang

Es war Anspruch der Implementierung **LEXBFS** viele, der in dieser Arbeit vorgestellten Algorithmen und Anwendungsmöglichkeiten von LexBFS zu integrieren. Dieses ist sicherlich erreicht worden, wenn auch kleinere Ergänzungen noch fehlen. Auf der GUI lassen sich mittels „drop-down“ Menüs die nachfolgend beschriebenen Funktionen ausführen.

5.1.1 Graph-Menü

Dient zum Import und Export der Graphen und hat folgende Menüpunkte:

Neuen Graph eingeben Der Aufruf dieses Punktes steht am Anfang jeder Berechnung. Es bestehen verschieden Möglichkeiten einen Graphen einzugeben.

1. **Grapheingabe durch Zeichnen** Es wird die Anzahl der Knoten festgelegt. Kanten werden per Maus erzeugt. Anzahl der Knoten muss kleiner als ***MaxVerticesToDisplay*** sein

2. **vollständiger Graph** Erzeugt einen vollständigen Graphen mit festzulegender Anzahl an Knoten.

3. **Turangraph** Erzeugt einen Turangraphen, d. h. einen k-partiten Graphen mit maximaler Kantenzahl. Die Knoten werden möglichst gleichmäßig auf die Partitionen verteilt. Anzahl der Partitionen muss neben der Knotenanzahl festgelegt werden.

4. **zufälliger Graph 1** Erzeugt einen Graphen mit festzulegender Knotenanzahl und festzulegender Wahrscheinlichkeit für das Auftreten einer bestimmten Kante.

5. **zufälliger Graph 2** Erzeugt ebenfalls eine zufälligen Graphen mit etwas anderer Zufallsauswahl. Einzelheiten sind dem Quellcode zu entnehmen.

6. **Eingabe mittels String Datei** Öffnet eine Datei-Auswahl-Fenster, mit Hilfe dessen eine Text-Datei gewählt werden kann. Aus dieser liest das LexBFS-Programm sämtlichen Informationen aus. Die Text-Datei muss den **LEDA Native Graph File Format** Konventionen genügen. Einzelheiten siehe unter 5.3.

Export des Graphen Exportiert den Graphen mittels eines Datei-Auswahl-Fensters in eine Text-Datei. Einzelheiten siehe unter 5.3.

Export des Graphen mit Nummerierung Exportiert den Graphen mittels eines Datei-Auswahl-Fenster in eine Text-Datei und speichert die aktuelle Nummerierung.

Grafikexport des Graphen Exportiert den Graphen in seiner aktuellen Form mittels eines Datei-Auswahl-Fenster in eine png-Grafikdatei. Ist nur möglich, wenn der aktuelle Graph maximal ***MaxVerticesToDisplay*** Knoten besitzt.

5.1.2 Nummerierung

Erzeugt jeweils die entsprechenden Nummerierungen:

LexBFS vorwärts Nummeriert den Graphen mittels LexBFS vorwärts gemäß 3.4.1.

LexBFS rückwärts Nummeriert den Graphen mittels LexBFS rückwärts gemäß Algorithmus 2.

LexBFS vorwärts manuell Nummeriert schrittweise den Graphen mittels LexBFS vorwärts, in jedem Schritt werden die möglichen nächsten Knoten farbig unterlegt. Der nächste Knoten ist per Mausklick wählbar.

LexBFS vorwärts auf $\bar{G}$ Nummeriert den komplementären Graphen mittels LexBFS vorwärts, wie unter 3.4.3 beschrieben.

LexBFS plus Zunächst wird der Graph mittels LexBFS vorwärts nummeriert, diese Nummerierung σ wird zur Erzeugung einer LexBFS$^+$ Nummerierung gemäß 3.4.2 benutzt.

LexBFS minus Zunächst wird der Graph mittels LexBFS vorwärts nummeriert, diese Nummerierung σ wird zur Erzeugung einer LexBFS$^-$ Nummerierung gemäß 3.4.3 benutzt.

MCS Nummeriert den Graphen mittels MCS gemäß 3.3.1.

MCS manuell Nummeriert schrittweise den Graphen mittels MCS, in jedem Schritt werden die möglichen nächsten Knoten farbig unterlegt. Der nächste Knoten ist per Mausklick wählbar.

LexDFS Nummeriert den Graphen mittels LexDFS gemäß 3.3.2.

LexDFS manuell Nummeriert schrittweise den Graphen mittels LexDFS, in jedem Schritt werden die möglichen nächsten Knoten farbig unterlegt. Der nächste Knoten ist per Mausklick wählbar.

LexBFS layerweise Nummeriert den Graphen mittels LexBFS layerweise gemäß 3.4.4.

5.1.3 Graphenklassen

Bei Aufruf der nachfolgend genannten Menüpunkte wird der entsprechende Test auf G_{akt} ausgeführt. Falls es die Anzahl der Knoten und die Art der Bestätigung zulassen, wird die Bestätigung in der GUI dargestellt. Falls der Algorithmus mit Bestätigung vorliegt und auch als solcher implementiert ist, wird die Bestätigung für die Zugehörigkeit zu einer Graphenklasse zusammen mit dem Graphen in einer Text-Datei gespeichert.

Chordaler Graph Führt einen Test auf Chordalität, wie unter 4.1 beschrieben, aus. Gibt das Ergebnis (wahr oder falsch) des Tests auf dem Bildschirm aus.

Chordaler Graph mit Bestätigung Gibt zusätzlich zum Test auf Chordalität eine PEO oder einen C_n mit $n \geq 4$ aus.

Cographtest Führt den, in [10] beschriebenen, 3-Sweeps Cographentest aus. Gibt zusätzlich eine positive Bestätigung in Form eines Cotree oder eine negative Bestätigung in Form eines P_4 aus. Der Cotree wird ebenfalls in der GUI angezeigt und lässt sich als Grafik Datei (png-Format) exportieren, falls
$|V| \leq \boldsymbol{MaxVerticesToDisplay}$.

Cographtest 2-Sweep Führt den, in 4.2 beschriebenen, 2-Sweeps Cographentest aus. Gibt zusätzlich eine positive Bestätigung in Form eines Cotree oder eine negative Bestätigung in Form eines P_4 aus. Der Cotree wird ebenfalls in der GUI angezeigt und lässt sich als Grafik Datei (png-Format) exportieren, falls
$|V| \leq \boldsymbol{MaxVerticesToDisplay}$.

Cographtest 2-Sweep ohne Cotree Führt den, in 4.2 beschriebenen, 2-Sweeps Cographentest aus. Gibt keine positive Bestätigung in Form eines Cotree aus. Gibt lediglich eine negative Bestätigung in Form eines P_4 aus. Ist auch für große Graphen geeignet.

Intervall-Graph 4-Sweep Führt den, in [24] beschriebenen, 4-Sweeps Intervall-Graphen Test aus.

Intervall-Graph 5-Sweep Führt den, in [16] beschriebenen, 5-Sweeps Intervall-Graphen Test aus.

Intervall-Graph 6-Sweep Führt den, in [26] beschriebenen, 6-Sweeps Intervall-Graphen Test aus. Speichert zusätzlich eine positive Bestätigung in Form einer Intervall-Darstellung.

Echter Intervall-Graph Test Führt den, in [17] beschriebenen, echten Intervall-Graphen Test aus. Gibt das Ergebnis (wahr oder falsch) des Tests auf dem Bildschirm aus.

Echter Intervall-Graph Test mit Bestätigung Gibt zusätzlich zum vorherigen Test eine positive oder negative Bestätigung aus (siehe 4.4).

DH-Graph mittels G^2 Führt den unter 4.5.1 erläuterten Algorithmus aus und gibt das Ergebnis (wahr oder falsch) des Tests auf dem Bildschirm aus.

DH-Graph mittels 3-Sweep Führt den unter 4.5.2 erläuterten Algorithmus aus und gibt das Ergebnis (wahr oder falsch) des Tests auf dem Bildschirm aus. (Ist leider noch nicht implementiert.)

P_4-reduzierbarer Graph Führt den unter 2.17 erläuterten Algorithmus aus und gibt das Ergebnis (wahr oder falsch) des Tests auf dem Bildschirm aus.

P_4-spärlicher Graph Führt den unter 4.6.2 erläuterten Algorithmus aus und gibt das Ergebnis (wahr oder falsch) des Tests auf dem Bildschirm aus.(Ist leider noch nicht implementiert.)

5.1.4 Graphansicht

Dient zum Verbessern der Darstellung des Graphen auf der GUI und zum optisch besseren Export der Graphen als png-Datei. Dieser Menüpunkt ist nur bei Graphen mit $|V| \leq \boldsymbol{MaxVerticesToDisplay}$ vorhanden.

Graphradius Verändert den Radius des Kreises auf dem die Knoten bei Erstellung des Graphen liegen.

Knotendurchmesser Verändert den Durchmesser der dargestellten Knoten.

Grafik Modus ein-/ausschalten Hinterlegt den Graphen mit einem Raster, an dem die Knoten ausgerichtet werden können.

Knoten einfärben Entfernt die Nummerierung der Knoten aus der Darstellung und färbt die Knoten blau oder macht diesen Schritt rückgängig.

Knoten Namen entfernen Entfernt die Namen der Knoten aus der Darstellung oder fügt sie wieder hinzu.

5.1.5 Graph verändern

Mit den unter diesem Menüpunkt zusammengefassten Funktionen, lässt sich der aktuelle Graph verändern.

Kanten hinzufügen oder entfernen Schaltet vom Modus Kanten hinzufügen zum Modus Kanten entfernen um und umgekehrt. Kanten lassen sich hinzufügen oder entfernen durch das Anklicken zweier Knoten.

Komplementärgraph Erzeugt den komplementären Graphen $\bar{G}_{akt}$.

Quadratur Quadriert G_{akt}.

Graph-Potenzen Potenziert G_{akt}. Der Exponent muss eingeben werden.

Entfernen von Knoten Entfernt einen festzulegenden Knoten aus G_{akt}. Danach ist der G_{akt} evtl. nur noch eine Isomorphie, des um den Knoten reduzierten vorherigen Graphen.

Graph mit Mittelpunkt und Radius Erzeugt eine Isomorphie, des durch Mittelpunkt und Radius festgelegten induzierten Teilgraphen von G_{akt}.

5.1.6 Extras

Liefert verschiedene Informationen zu G_{akt}.

Adjazenzlisten Gibt die Adjazenzlisten im Meldungsfenster aus.

Adjazenzmatrix Gibt die Adjazenzmatrix im Meldungsfenster aus.

Distanzmatrix Gibt die Distanzmatrix im Meldungsfenster aus.

Färben nach Entfernung Färbt die Knoten des G_{akt} nach der Entfernung zu einem festzulegenden Knoten. Dieser Menüpunkt ist nur bei $|V_{akt}| \leq$ ***MaxVerticesToDisplay*** vorhanden.

5.2 Die Datenstrukturen

Die verwendeten Datenstrukturen ergeben sich aus oben beschriebenen Details. Graphen sind als eine doppelt verkettete Liste der Knoten implementiert. Jeder dieser Knoten hat wiederum eine Adjazenzliste, die wiederum durch eine doppelt verkettete Liste realisiert ist. Die beschriebenen Sets sind auch als solche Listen realisiert. Um die lineare Laufzeit bei den LexBFS-Sweeps zu gewährleisten, wird mit Pointern auf die jeweiligen Sets und die Position innerhalb dieser Sets gearbeitet. Alle weiteren Details sind den Quellcodes zu entnehmen. Es werden also keine „komplizierten“ Datenstrukturen verwendet, was ja auch als Vorteil von LexBFS in vielen der zitierten Arbeiten angepriesen wird.
Exemplarisch ist in Abb. 53 vereinfacht der Aufbau von Objekten der Klasse **NumberedGraph** dargestellt. Diese Klasse ist die zentrale Klasse der gesamten Implementierung. Grundlegend ist dabei die Aufteilung in den eigentlichen Graphen „myGraph“ ein Objekt der Klasse Graph, die Nummerierung „mySorting“ ein Array von integer Zahlen, und die beiden PointerArrays, in denen die Position der Knoten und die Position der Sets für die einzelnen Knoten abgelegt werden. Die Objekte der Klasse **Graph** bestehen wiederum jeweils aus einem Objekt der Klasse **ListOfVertices** als Liste der Knoten.

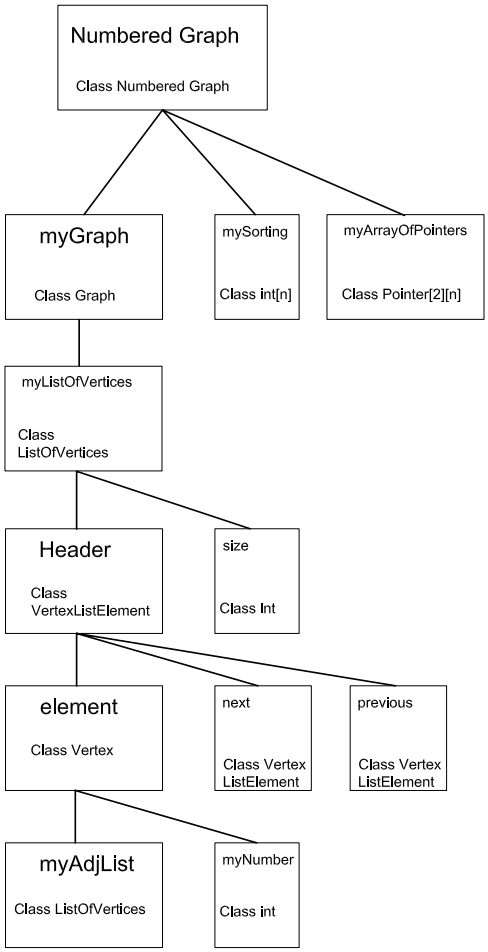

Abbildung 53: Vereinfachter Aufbau der Klasse NumberedGraph

5.3 Import und Export Schnittstellen

Als Graphformat wird zum Import und Export der Graphen das ***native LEDA Graph File Format*** verwendet [63]. Exemplarisch ist für den Graphen aus Abb. 19 die zu importierende bzw. zu exportierende Text-Datei nachfolgend aufgelistet.

```
#header section
LEDA.GRAPH
string
int
-1
#nodes section
9
|{v1}|
|{v2}|
|{v3}|
|{v4}|
|{v5}|
|{v6}|
|{v7}|
|{v8}|
|{v9}|

#edges section
25
1 7 0 |{0}|
1 9 0 |{1}|
1 8 0 |{2}|
1 6 0 |{3}|
1 5 0 |{4}|
1 2 0 |{5}|
2 9 0 |{6}|
2 8 0 |{7}|
2 7 0 |{8}|
2 5 0 |{9}|
2 6 0 |{10}|
3 5 0 |{11}|
3 6 0 |{12}|
3 7 0 |{13}|
3 8 0 |{14}|
3 9 0 |{15}|
3 4 0 |{16}|
4 5 0 |{17}|
4 6 0 |{18}|
4 7 0 |{19}|
4 8 0 |{20}|
4 9 0 |{21}|
5 6 0 |{22}|
7 8 0 |{23}|
8 9 0 |{24}|
```

5.4 Die Benutzeroberfläche = GUI

Die Knoten des Graphen werden zunächst stets im Kreis angeordnet. Sie lassen sich aber per Maus (drag and drop) verschieben. Das kann gerade für den Grafik Export eines Graphen als png - Datei von Nutzen sein. Die im Text verwendeten Graphenabbildungen sind teilweise auch mit **LEXBFS** erzeugt worden. Jedem Graphen, der auf der GUI dargestellt wird, lassen sich per Mausklick Kanten hinzufügen oder entfernen.

5.5 Ausbaumöglichkeiten

Sollten sich in Zukunft weitere, bisher nicht bekannte Anwendungen der angesprochenen Suchalgorithmen LexBFS, MCS und LexDFS ergeben, so lassen sich diese in **LEXBFS** integrieren. Einschränkend muss angemerkt werden, dass schon bei allen bisher implementierten Algorithmen zur Graphenklassenerkennung, der eigentliche LexBFS Anteil an der Implementierung von untergeordneter Bedeutung war. Gerade die Überprüfung der Graphenklassen spezifischen Bedingungen z.B. der „Neighbourhood Subset Property“ beim Cographen Test, hat den Quellcode und auch die zu interne Darstellung der Graphen aufgebläht. Es mussten für jeden Knoten Nachbarschaften, Zugehörigkeiten zu Slices etc. „mitgeschleppt“ werden. Daher ist bei zusätzlichen Anwendungen der Aufwand der zu überprüfenden Bedingungen zu berücksichtigen. Evtl. kann ein vorher begrenzter Funktionsumfang die Laufzeiten einzelner Algorithmen erheblich verbessern. Es macht z. B. keinen Sinn für Chordalitätsprüfungen die nummerierten Nachbarschaften und die Slice-Arrays als Parameter bereitzustellen, selbst wenn diese nicht berechnet werden.

6 Ergebnisse und Ausblick

Wie sich im Laufe der Ausführungen gezeigt hat, sind die Anwendungen von LexBFS vielfältig und dürften noch nicht erschöpfend entdeckt worden sein. Gerade vor dem Hintergrund, dass LexBFS zwar schon vor 30 Jahren von Rose, Tarjan und Lueker entwickelt wurde, aber doch erst seit etwa 10 Jahren verstärkt Ergebnisse für andere Anwendungen, als die Erkennung chordaler Graphen, gewonnen wurden, ist von weiteren, noch unentdeckten Anwendungen auszugehen. Insbesondere aus der Kombination verschiedener Sortierungen von LexBFS, LexDFS und MCS bzw. MNS lassen sich vermutlich noch viele Anwendungen von LexBFS entwickeln. Selbst im Laufe des Jahres, in dem diese Arbeit entstanden ist, sind erst einige der Anwendungen von LexBFS entdeckt oder zumindest stark verbessert worden.

Literatur

[1] Falk Nicolai Andreas Brandstädt, Feodor F. Dragan. *LexBFS-orderings and powers of chordal graphs*, volume 287 of *Schriftenreihe des Fachbereichs Mathematik / Universität Duisburg*. Universität Duisburg, 1995.

[2] Feodor F. Dragan Andreas Brandstädt, Victor D. Chepoi. Note on perfect elimination orderings of chordal powers of graphs. *Discrete Mathematics*, 158:273–278, 1996.

[3] Hans-Jürgen Bandelt and Henry Martyn Mulder. Distance-hereditary graphs. *J. Comb. Theory, Ser. B*, 41(2):182–208, 1986.

[4] Anne Berry and Jean paul Bordat. Local lexBFS properties in an arbitrary graph (extended abstract), February 10 2000.

[5] Bogart and West. A short proof that 'proper = unit'. *DMATH: Discrete Mathematics*, 201, 1999.

[6] Kellogg S. Booth and George S. Lueker. Linear algorithms to recognize interval graphs and test for the consecutive ones property. In *STOC*, pages 255–265, 1975.

[7] Andreas Brandstädt, Feodor F. Dragan, Victor Chepoi, and Vitaly I. Voloshin. Dually chordal graphs. *SIAM J. Discrete Math.*, 11(3):437–455, 1998.

[8] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: A survey*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[9] A. Bretscher, D.G. Corneil, M. Habib, and C. Paul. A simple linear time lexbfs cograph recognition algorithm, 2006. "www.liafa.jussieu.fr/ habib/Documents/cograph.ps".

[10] Anna Bretscher. *LexBfs based recognition algorithms for Cographs and related families*. PhD thesis, Department of Computer Science, University of Toronto, 2005.

[11] Anna Bretscher, Derek G. Corneil, Michel Habib, and Christophe Paul. A simple linear time lexbfs cograph recognition algorithm. In *WG*, pages 119–130, 2003.

[12] L. Sunil Chandran and Fabrizio Grandoni. A linear time algorithm to list the minimal separators of chordal graphs. *Discrete Mathematics*, 306(3):351–358, 2006.

[13] L. Sunil Chandran, Louis Ibarra, Frank Ruskey, and Joe Sawada. Generating and characterizing the perfect elimination orderings of a chordal graph. *Theor. Comput. Sci.*, 307(2):303–317, 2003.

[14] Corneil, Olariu, and Stewart. Linear time algorithms for dominating pairs in asteroidal triple-free graphs. *SICOMP: SIAM Journal on Computing*, 28, 1999.

[15] D. G. Corneil, H. Lerchs, and L. Stewart Burlingham. Complement reducible graphs. *Discrete Appl. Math.*, 3:163–174, 1981.

[16] Derek G. Corneil. Lexicographic breadth first search - a survey. In *WG*, pages 1–19, 2004.

[17] Derek G. Corneil. A simple 3-sweep lbfs algorithm for the recognition of unit interval graphs. *Discrete Applied Mathematics*, 138(3):371–379, 2004.

[18] Derek G. Corneil. Private communication, 2007.

[19] Derek G. Corneil, Feodor F. Dragan, Michel Habib, and Christophe Paul. Diameter determination on restricted graph families. *Discrete Applied Mathematics*, 113(2-3):143–166, 2001.

[20] Derek G. Corneil, Feodor F. Dragan, and Ekkehard Köhler. On the power of bfs to determine a graph's diameter. *Networks*, 42(4):209–222, 2003.

[21] Derek G. Corneil, Hiryoung Kim, Sridhar Natarajan, Stephan Olariu, and Alan P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55(2):99–104, 1995.

[22] Derek G. Corneil and Richard Krueger. A unified view of graph searching, to appear. "http://www.cs.toronto.edu/ krueger/papers/unified.ps",.

[23] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. A linear time algorithm to compute a dominating path in an at-free graph. *Inf. Process. Lett.*, 54(5):253–257, 1995.

[24] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. The ultimate interval graph recognition algorithm? (extended abstract). In *SODA*, pages 175–180, 1998.

[25] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. LBFS orderings and cocomparability graphs. In *SODA*, pages 883–884, 1999.

[26] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. The lbfs structure and recognition of interval graphs. *submitted for publication*, 2006.

[27] Derek G. Corneil, Yehoshua Perl, and L. K. Stewart. A linear recognition algorithm for cographs. *SIAM J. Comput.*, 14(4):926–934, 1985.

[28] William H. Cunningham. Decomposition of directed graphs. *SIAM Journal on Algebraic and Discrete Methods*, 3(2):214–228, 1982.

[29] K. Mehlhorn D. Kratsch, R. McConnell and J. Spinrad. Certifying algorithms for recognizing interval and permutation graphs. *SIAM Journal on Computing*, 36:326–353, 2006.

[30] Elias Dahlhaus. Efficient parallel and linear time sequential split decomposition (extended abstract). In *FSTTCS*, pages 171–180, 1994.

[31] Elias Dahlhaus, Paul D. Manuel, and Mirka Miller. A characterization of strongly chordal graphs. Technical Report 96-27, Callaghan 2308, Australia, 1996.

[32] Guillaume Damiand, Michel Habib, and Christophe Paul. A simple paradigm for graph recognition: application to cographs and distance hereditary graphs. *Theor. Comput. Sci.*, 263(1-2):99–111, 2001.

[33] Celina M. Herrera de Figueiredo, João Meidanis, and Célia Picinin de Mello. A linear-time algorithm for proper interval graph recognition. *Inf. Process. Lett.*, 56(3):179–184, 1995.

[34] Xiaotie Deng, Pavol Hell, and Jing Huang. Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. *SIAM J. Comput.*, 25(2):390–403, 1996.

[35] G. A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg 25*, 25:71–76, 1961.

[36] Feodor F. Dragan. Almost diameter of a house-hole-free graph in linear time via lexbfs. *Discrete Applied Mathematics*, 95(1-3):223–239, 1999.

[37] Feodor F. Dragan and Falk Nicolai. Lexbfs-orderings of distance-hereditary graphs with application to the diametral pair problem. *Discrete Applied Mathematics*, 98(3):191–207, 2000.

[38] Feodor F. Dragan, Falk Nicolai, and Andreas Brandstädt. Lexbfs-orderings and power of graphs. In *WG*, pages 166–180, 1996.

[39] Martin Farber. Characterization of strongly chordal graphs. *Discrete Mathematics*, 43:173–189, 1983.

[40] Falk Nicolai Feodor F. Dragan. *LexBFS-Orderings of distance-hereditary graphs*, volume 303 of *Schriftenreihe des Fachbereichs Mathematik / Universität Duisburg*. Universität Duisburg, 1995.

[41] D. R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15(3):835–855, 1965.

[42] Frederic Gardi. A note on the roberts characterization of proper and unit interval graphs. *submitted to Discrete Mathematics*, 2005.

[43] Martin C. Golumbic. *Algorithmic Graph Theory and Its Applications*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 41–62. Martin C. Golumbic, 2004.

[44] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57 of *Annals of Discrete Mathematics*. Elsevier Science B.V., Amsterdam-Boston-Heidelberg-London-New York-Oxford-Paris-San Diego-San Francisco-Singapore-Sydney-Tokyo, 2004. Second Edition.

[45] Michel Habib, Fabien de Montgolfier, and Christophe Paul. A simple linear-time modular decomposition algorithm for graphs, using order extension. In *SWAT: Scandinavian Workshop on Algorithm Theory*, pages 187–198, 2004.

[46] Michel Habib, Ross M. McConnell, Christophe Paul, and Laurent Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theor. Comput. Sci.*, 234(1-2):59–84, 2000.

[47] Michel Habib and Christophe Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 145(2):183–197, 2005.

[48] Pavol Hell and Jing Huang. Certifying lexbfs recognition algorithms for proper interval graphs and proper interval bigraphs. *SIAM J. Discrete Math.*, 18(3):554–570, 2004.

[49] Pavol Hell and Jing Huang. Interval bigraphs and circular arc graphs. *Journal of Graph Theory*, 46(4):313–327, 2004.

[50] Wen-Lian Hsu and Tze-Heng Ma. Fast and simple algorithms for recognizing chordal comparability graphs and interval graphs. *SIAM J. Comput.*, 28(3):1004–1020, 1999.

[51] Beverly Jamison and Stephan Olariu. P4- reducible graphs, a class of uniquely tree representable graphs. *Studies in Applied Mathematics*, 81:79–87, 1989.

[52] Beverly Jamison and Stephan Olariu. Recognizing p_4 sparse graphs in linear time. *SIAM J. Comput.*, 21:381–406, 1992.

[53] Beverly Jamison and Stephan Olariu. A unique tree representation for p_4 - sparse graphs. *Discrete Applied Math.*, 35:115–129, 1992.

[54] Beverly Jamison and Stephan Olariu. A linear-time recognition algorithm for p4-reducible graphs. *Theor. Comput. Sci.*, 145(1&2):329–344, 1995.

[55] N. Kalyana Rama Prasad and P. Sreenivasa Kumar. On generating strong elimination orderings of strongly chordal graphs. *Lecture Notes in Computer Science*, 1530:221–232, 1998.

[56] Norbert Korte and Rolf H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.*, 18(1):68–81, 1989.

[57] Richard Krueger. *Graph Searching.* PhD thesis, Department of Computer Science, University of Toronto, 2005. "http://www.cs.toronto.edu/ krueger/-papers/thesis.ps".

[58] P. Sreenivasa Kumar and C. E. Veni Madhavan. Minimal vertex separators of chordal graphs. *Discrete Applied Mathematics*, 89(1-3):155–168, 1998.

[59] R. Laskar and D.R. Shier. On powers and centers of chordal graphs. *Discrete Applied Mathematics*, (6):139–147, 1983.

[60] C. Lekkerkerker and D. Boland. Representation of finite graphs by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[61] P.J. Looges and S. Olariu. *Optimal greedy recognition and coloring for indifference graphs*, volume 34, pages 127–137. Pergamon Press, 1992.

[62] Anna Lubiw. Doubly lexical orderings of matrices. *SIAM J. Comput.*, 16(5):854–879, 1987.

[63] Kurt Mehlhorn and Stefan Näher. *LEDA: a platform for combinatorial and geometric computing.* Cambridge University Press, New York, NY, USA, 1999.

[64] Stephan Olariu. An optimal greedy heuristic to color interval graphs. *Inf. Process. Lett.*, 37(1):21–25, 1991.

[65] F. S. Roberts. *Indifference graphs*, pages 139–146. Academic Press, New York, 1969.

[66] Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5(2):266–283, 1976.

[67] D. Seinsche. On a property of the class of n-colorable graphs. *J. Combinatorial Theory Ser.*, B(16):191–193, 1974.

[68] D. R. Shier. Some aspects of perfect elimination orderings in chordal graphs. *Discrete Appl. Math.*, 7:325–331, 1984.

[69] Klaus Simon. A new simple linear algorithm to recognize interval graphs. In *Workshop on Computational Geometry*, pages 289–308, 1991.

[70] Klaus Simon. A note on lexicographic breadt first search for chordal graphs. *Inf. Process. Lett.*, 54(5):249–251, 1995.

[71] Jeremy Spinrad, Andreas Brandstädt, and Lorna Stewart. Bipartite permutation graphs. *Discrete Appl. Math.*, 18:279–292, 1987.

[72] Jeremy P. Spinrad. *Efficient Graph Representations*, volume 19 of *Field Institute Monographs*. American Mathematical Society, 2003.

[73] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[74] Robert Endre Tarjan and Mihalis Yannakakis. Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 14(1):254–255, 1985.

[75] G. Wegner. *Eigenschaften der Nerven homologische-einfactor Familien in* R^n. PhD thesis, Universität Goettigen, Goettingen, 1967.

[76] Uri Zwick. Exact and approximate distances in graphs - a survey. In *ESA*, pages 33–48, 2001.