Platzeffiziente Erkennung von chordalen Graphen

Seminar „Algorithmen und Datenstrukturen“

Konstantin Fickel

Sommersemester 2018
Universität Augsburg

Sei G ein ungerichteter Graph. G ist chordal, wenn jeder Kreis mit mindestens vier Knoten in G eine Sehne besitzt. In dieser Seminararbeit, die hauptsächlich auf der Arbeit „Space-Efficient Algorithms for Maximum Cardinality Search, Stack BFS, Queue BFS and Applications“ von Sankardeep Chakraborty und Srinivasa Rao Satti ([CS17]) sowie auf Kapitel 4 des Buches „Algorithmic graph theory and perfect graphs“ von Martin Charles Golumbic ([Gol04]) basiert, wird zuerst eine Möglichkeit, chordale Graphen zu erkennen, vorgestellt. Anschließend werden für einen Teilschritt davon verschiedene Zeit/Platz-Tradeoffs erklärt.

In dieser Seminarbeit soll eine spezielle Untermenge der Graphen untersucht werden, die sogenannten chordalen Graphen.

Chordale Graphen sind vor allem theoretisch interessant, da einige Probleme, die auf der Menge aller Graphen NP-vollständig sind, auf chordalen Graphen nur lineare Zeit benötigen. Dazu zählt beispielsweise das Clique-Problem (Entscheide für einen Graphen $G = (V, E)$ und eine Zahl k, ob es eine Knotenmenge $X \subseteq V$ gibt mit $|X| = k$, deren induzierter Teilgraph $G[X]$ vollständig ist).[1]

Definition 1 – Chordaler Graph: *Sei $G = (V, E)$ ein ungerichteter Graph. G ist chordal, wenn jeder Kreis mit mindestens 4 Knoten eine Sehne (englisch: chord, also eine Kante zwischen zwei auf dem Kreis nicht-benachbarten Knoten) besitzt.*[2]

Ein Beispiel für einen chordalen Graphen ist in Abbildung 1 (ohne die gestrichelte Kante) zu sehen. Würde man die gestrichelte Kante hinzunehmen, wäre $(v_0, v_1, v_2, v_5, v_0)$ ein Kreis mit vier Knoten ohne Sehne – und somit wäre der Graph nicht mehr chordal.

[1] siehe [Gol04, Satz 4.17]
[2] siehe [Gol04, Kapitel 4.1]

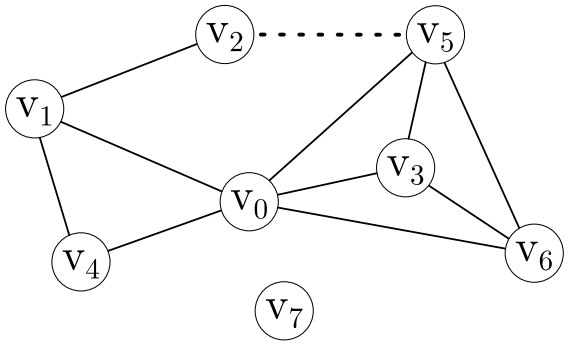

Abbildung 1: Der hier abgebildete Graph G (ohne die gestrichelte Kante) ist chordal.

1 Perfekte Eliminations-Reihenfolgen

Um einen Ansatz zur Prüfung der Chordalität eines Graphen zu finden, benötigt man eine äquivalente Aussage, die in diesem Abschnitt hergeleitet werden soll.

Definition 2 – Vollständiger Graph: *Ein Graph $G = (V, E)$ ist vollständig, wenn jedes Paar $v, w \in V$ von Knoten durch eine Kante $\{v, w\} \in E$ verbunden ist.*[3]

Die *Adjazenzmenge* eines Knoten $x \in V$ bezüglich eines ungerichteten Graphen $G = (V, E)$ sei die Menge der Knoten, zu denen x im Graphen G eine Verbindung besitzt: $\mathrm{Adj}(x) = \{y \in V \mid \{x, y\} \in E\}$. Für eine Menge von Knoten $X \subseteq V$ ist diese $\mathrm{Adj}(X) = \bigcup_{x \in X} \mathrm{Adj}(x)$. Der durch die Teilmenge $X \subseteq V$ induzierte *Teilgraph* $G[X]$ sei definiert als $G[X] = (X, \{\{v, w\} \in E \mid v, w \in X\})$.

Definition 3 – Simplizialer Knoten: *Ein simplizialer Knoten v in einem Graphen G ist ein Knoten, dessen Nachbarn einen vollständigen Teilgraphen $G[\mathrm{Adj}(v)]$ induzieren.*[4]

Für den Beweis der nächsten Lemmata benötigen wir folgendes Hilfskonstrukt:

Definition 4 – Minimaler Knotentrenner: *Ein Knotentrenner S der Knoten $a, b \in V$ ist eine Teilmenge $S \subseteq V$ mit $a, b \notin S$, bei dem sich a und b in verschiedenen Zusammenhangskomponenten von $G[V \setminus S]$ befinden.*

S ist ein minimaler Knotentrenner, falls diese Aussage für kein $T \subsetneq S$ gilt.[5]

Das folgende Lemma stellt eine bestimmte Eigenschaft von chordalen Graphen dar, die wir in den darauf folgenden Beweisen benötigen:

[3] siehe [Gol04, Kapitel 1.1]
[4] siehe [Gol04, Kapitel 4.2]
[5] siehe [Gol04, Kapitel 4.2]

Lemma 1. *Sei $G = (V, E)$ ein chordaler Graph. Dann ist der induzierte Teilgraph $G[S]$ zu jedem minimalen Knotentrenner $S \subseteq V$ vollständig.*[6]

Beweis. Sei S ein minimaler Knotentrenner für die Knoten a und b. $G[A] = (A, E_A)$ und $G[B] = (B, E_B)$ seien die Zusammenhangskomponenten von $G_{V \setminus S}$, die jeweils a und b enthalten.

Da S minimal ist, hat jeder Knoten $v \in S$ jeweils einen Nachbarn in A und B. Deshalb existiert zu jedem Knotenpaar $x, y \in S$ ein minimaler Weg (bezüglich Kantenanzahl) $(x, a_1, a_2, \dots a_r, y)$ mit $a_i \in A$ und $(y, b_1, b_2, \dots b_t, x)$ mit $b_i \in B$.

Der Kreis $(x, a_1, a_2, \dots, a_r, y, b_1, b_2, \dots, b_t, x)$ besitzt mindestens die Länge 4 und hat somit, da G chordal ist, eine Sehne. Da die Wege minimal gewählt sind, kann diese jedoch nicht zwei Knoten aus demselben Weg verbinden, womit $\{a_i, a_j\} \notin E$ für $i, j \in \{1, \dots r\}, |i - j| > 1$ und $\{b_i, b_j\} \notin E$ für $i, j \in \{1, \dots, t\}, |i - j| > 1$ gilt. Da alle a_i und b_i durch S getrennt sind, ist außerdem $\{a_i, b_j\} \notin E$ für $i \in \{1, \dots, r\}$ und $j \in \{1, \dots, t\}$. Damit müssen die beiden Knoten x und y aus dem Knotentrenner verbunden sein. □

Mithilfe von Lemma 1 können wir nun auch folgende Aussage treffen:
Lemma 2. *Jeder chordale Graph $G = (V, E)$ besitzt einen simplizialen Knoten. Falls G nicht vollständig ist, dann besitzt dieser zwei nicht-benachbarte simpliziale Knoten.*[7]

Beweis. Induktionsvoraussetzung: Für jeden Graphen $G = (V, E)$ mit $|V| \leq t$ gelte, dass G entweder vollständig ist oder zwei nicht-benachbarte simpliziale Knoten besitzt.

$t = 1$ Da ein Graph bestehend aus nur einem Knoten vollständig ist, gilt der Induktionsanfang.

$t \Rightarrow t + 1$ Sei $G = (V, E)$ ein Graph mit $|V| = t$ Kanten. Falls G vollständig ist, ist die Aussage trivial.

Habe G also nun zwei nicht-benachbarte Knoten a und b und die Aussage sei für alle Graphen mit weniger Knoten als G gültig. Wähle als S einen minimalen Knotentrenner von a und b; $G[A]$ und $G[B]$ seien wie in Lemma 1 definiert.

Wegen der Induktionsvoraussetzung gilt, dass entweder der Teilgraph $G[A \cup S]$ zwei nicht-benachbarte simpliziale Knoten besitzt, von dem einer in $G[A]$ liegen muss (da S nach Satz 1 vollständig ist) oder $G[A \cup S]$ vollständig ist.

[6] siehe [Gol04, Satz 4.1 (i) ⇒ (iii)]
[7] siehe [Gol04, Lemma 4.2]

Da $\mathrm{Adj}(A) \subseteq A \cup S$, ist jeder simpliziale Knoten in $G[A]$ ebenfalls simplizial in gesamt G. Mit einer ähnlichen Argumentation lässt sich zeigen, dass auch $G[B]$ einen simplizialen Knoten besitzt.

□

Um den Graphen auf Chordalität zu überprüfen, werden wir eine sogenannte perfekte Eliminations-Reihenfolge verwenden, die genau dann existiert, wenn G chordal ist. Eine *Reihenfolge* ist dabei eine Bijektion $\sigma : \{1, \ldots, |V|\} \mapsto V$.

Definition 5 – Perfekte Eliminations Reihenfolge: *Eine Reihenfolge σ auf $G = (V, E)$ ist eine perfekte Eliminations-Reihenfolge, wenn für jedes $i \in \{1, \ldots, |V|\}$ der Knoten $\sigma(i)$ simplizial im Teilgraph $G[\{\sigma(i), \ldots, \sigma(|V|)\}]$ ist.*[8]

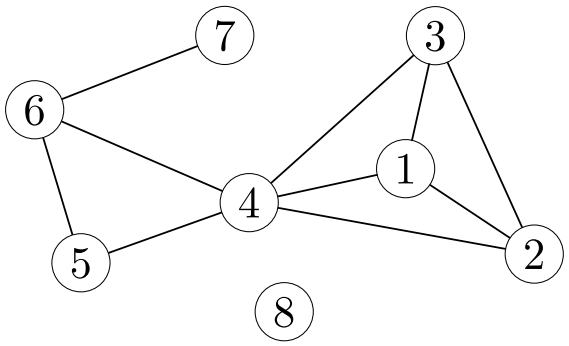

Abbildung 2: Die durch die Beschriftung der Knoten dargestellte Reihenfolge σ ist eine perfekte Eliminations-Reihenfolge.

Nun ist es uns möglich, die gesuchte Äquivalenz zu beweisen und damit die Hauptaussage des Abschnitts zu zeigen:

Satz 1. *Sei G ein ungerichteter Graph. G ist chordal genau dann, wenn für G eine perfekte Eliminations-Reihenfolge σ existiert.*[9]

Beweis. $\Rightarrow$ Nach Satz 2 besitzt ein chordaler Graph G einen simplizialen Knoten x. Da auch $G[V \setminus \{x\}]$ chordal ist und weniger Knoten als G besitzt, lässt sich induktiv eine perfekte Eliminations-Reihenfolge erstellen, wenn σ als Reihenfolge der in jedem Schritt entfernten x gebildet wird.

$\Leftarrow$ Sei C ein Kreis in G mit mindestens vier Knoten und x der kleinste Knoten darin bezüglich $\sigma^{-1}(x)$.

[8] siehe [Gol04, Kapitel 4.2]
[9] siehe [Gol04, Satz 4.1 (i) $\Leftrightarrow$ (ii)]

Durch die Wahl von x liegt C vollständig in $G\left[\{\sigma\left(i\right) \mid i \geq \sigma^{-1}\left(x\right)\}\right]$. Da der Knoten x in diesem Teilgraphen simplizial ist muss zwischen den beiden Nachbarknoten v, w von x im Kreis C eine Sehne existieren.

□

2 Kardinalitätssuche

Nachdem wir im vorherigen Kapitel gezeigt haben, dass G genau dann chordal ist, wenn eine perfekte Eliminations-Reihenfolge σ für G existiert, werden wir uns in diesem Abschnitt damit beschäftigen, wie sich eine solche perfekte Eliminations-Reihenfolge algorithmisch bestimmen lässt. Dazu werden wir die sogenannte Kardinalitätssuche verwenden, die in der englischen Literatur *Maximum Cardinality Search (kurz MCS)* genannt wird.

Diese wird „von hinten“ gebildet, indem in jedem Schritt $\sigma\left(i\right)$ mit $i \in \{|V|, \ldots, 1\}$ jeweils der nicht-nummerierte (also in σ noch nicht zugeordnete) Knoten ist, der am meisten Nachbarn besitzt, die bereits nummeriert worden sind.[10]

Die perfekte Eliminations-Reihenfolge aus Abbildung 2 hätte dabei durch die Kardinalitätssuche generiert werden können. Diese ist jedoch nicht eindeutig, da in vielen Schritten mehrere Knoten mit größter Zahl an nummerierten Nachbarn verfügbar sind.

Standard-Implementierung Eine einfache Implementierung basiert darauf, die Knoten nach der Anzahl der benachbarten, nummerierten Knoten in Mengen sortiert zu halten:

Als Datenstruktur wird ein Array `S` von $n = |V|$ Mengen verwendet, in dem alle nicht-nummerierten Knoten gespeichert sind. Dabei sind jeweils in $\texttt{S}\left[i\right]$ die Knoten mit i nummerierten Nachbarn zu finden ($i \in \{0, \ldots, n-1\}$). Um in konstanter Zeit auf Knoten in `S` zugreifen zu können und den Index der Menge zu finden, benötigen wir zusätzlich dazu noch zwei Listen von Zahlen: Während in `M` für jeden Knoten $v \in V$ die Position in `S` gespeichert wird, sind es in `N` jeweils die Anzahl der nummerierten Nachbarn und somit auch der Index der Menge in `S`.[11]

Da in `M` und `N` für alle n Knoten die in $\mathcal{O}\left(\lg\left(n\right)\right)$ bits darstellbare Position in `S` gespeichert ist, genügen hier jeweils $\mathcal{O}\left(n \cdot \lg\left(n\right)\right)$ Speicher. `S` lässt sich dabei durch einen Array von n

[10] siehe [Gol04, Kapitel 4.3] und [CS17, 2.1]

[11] siehe [TY84, Abschnitt 2, Algorithmus Maximum Cardinality Search], nur, dass in `M` nun die genaue Position in `S` gespeichert wird, vergleichbar mit [CS17, Abschnitt 2]

doppelt-verketteten Listen darstellen, deren einzelne Einträge sich mit jeweils $\mathcal{O}(\lg(n))$ bits speichern lassen, womit insgesamt hier ebenfalls $\mathcal{O}(n \cdot \lg(n))$ bits ausreichen.[12]

Zu Beginn der Ausführung werden alle Knoten von G in $\mathtt{S}[0]$ eingetragen und in $\mathtt{M}$ und $\mathtt{N}$ entsprechend gegenreferenziert. In jedem Schritt $i \in \{n, \ldots, 1\}$ wird dann das Element $\sigma(i)$ bestimmt: Dieses ist ein beliebiger Knoten $v \in \mathtt{S}[j]$, wobei j der größte Index einer Menge in $\mathtt{S}$ ist, für den $\mathtt{S}[j] \neq \emptyset$. Anschließend wird jeder nicht-nummerierte Nachbarknoten w von v aus der Menge $\mathtt{S}[\mathtt{N}[w]]$ entfernt und in $\mathtt{S}[\mathtt{N}[w]+1]$ eingefügt und dann auch jeweils die Gegenreferenzen in $\mathtt{N}$ und $\mathtt{M}$ aktualisiert.

Data: Graph $G = (V, E)$
for $i \in \{0, \ldots, n-1\}$ **do**
  $\mathtt{S}[i] \longleftarrow \emptyset$
for $v \in V$ **do**
  füge v in $\mathtt{S}[0]$ ein;
  $\mathtt{M}[v] \longleftarrow$ Position von v in $\mathtt{S}[0]$;
  $\mathtt{N}[v] \longleftarrow 0$;
$j \longleftarrow 0$;
for $i \in \{n, \ldots, 1\}$ **do**
  $v \longleftarrow$ entferne einen Knoten aus $\mathtt{S}[j]$;
  $\sigma[i] \longleftarrow v$;
  $j \longleftarrow j + 1$;
  $\mathtt{N}[v] \longleftarrow -1$;
  for $w \in \mathrm{Adj}(v)$ *mit* $\mathtt{N}[w] \geq 0$ **do**
    $k \longleftarrow N[w]$;
    lösche w aus $\mathtt{S}$ an Position $\mathtt{M}[w]$ aus der Menge $\mathtt{S}[k]$;
    füge w in $\mathtt{S}[k+1]$ ein;
    $\mathtt{N}[w] \longleftarrow k + 1$;
    $\mathtt{M}[w] \longleftarrow$ neue Postion von v in $\mathtt{S}[k]$;
  while $j \geq 0 \wedge \mathtt{S}[j] = \emptyset$ **do**
    $j \longleftarrow j - 1$;
Result: Reihenfolge $\sigma : \{1, \ldots, n\} \mapsto V$ der Knoten in V

Algorithmus 1: Kardinalitätssuche nach [CS17, 2.1]

Eine Implementierung in Pseudo-Code ist in Algorithmus 1 zu finden. Dabei ist vor allem die Verwaltung des Index der größten Menge j interessant. Am Anfang wird dieser auf 0 gesetzt, da sich alle Elemente in $\mathtt{S}[0]$ befinden. Da der Index der Menge in $\mathtt{S}$ eines Knotens in jedem Schritt um maximal 1 erhöht werden kann, wird j um 1 inkrementiert, um sicherzustellen, dass $j \geq \mathsf{max}\{j \in \{1, \ldots, n\} \mid \mathtt{S} \neq \emptyset\}$. Mithilfe der **while**-Schleife wird diese Ungleichung dann wieder zur Gleichung.

[12] siehe [CS17]

Bei der Ausführung werden dann zuerst alle Knoten in $\mathtt{S}[0]$ eingefügt und entsprechend in $\mathtt{M}$ gegenreferenziert, was $\mathcal{O}(n)$ Zeit benötigt. Anschließend werden für jeden der n Knoten jeweils die Adjazenzliste durchlaufen; dies kann in $\mathcal{O}\left(\sum_{v \in V}(1 + |\mathrm{Adj}(v)|)\right) = \mathcal{O}(n+m)$ erreicht werden. Da j in jedem der n Durchläufe um jeweils 1 inkrementiert wird, und nie $j < -1$ gelten kann, wir der Rumpf der **while**-Schleife höchstens $n+1$ mal durchlaufen, sodass alle Operationen auf j höchstens $\mathcal{O}(n)$ Zeit benötigen. Damit ist die gesamte Laufzeit des Algorithmus' $\mathcal{O}(n+m)$.

Um nun zu beweisen, dass die Kardinalitätssuche wirklich – falls dies möglich ist, also G chordal ist – eine perfekte Eliminations-Reihenfolge bestimmt, beweisen wir zunächst, dass jede von der Kardinalitätssuche generierte Reihenfolge folgende Eigenschaft besitzt:

Definition 6 – Reihenfolgeneigenschaft P: *Sei σ eine Reihenfolge der Knoten des Graphen $G = (V, E)$.*

σ besitzt die Eigenschaft P, falls für alle Knoten $a, b, c \in V$ mit $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$ und $c \in \mathrm{Adj}(a) \setminus \mathrm{Adj}(b)$ ein weiterer Knoten $x \in \mathrm{Adj}(b) \setminus \mathrm{Adj}(a)$ mit $\sigma^{-1}(b) < \sigma^{-1}(x)$ existiert.[13]

Satz 2. *Jede von der Kardinalitätssuche generierte Reihenfolge σ besitzt die Eigenschaft P.*[14]

Beweis. Sei σ eine Reihenfolge der Knoten des Graphen $G = (V, E)$, die durch die Kardinalitätssuche berechnet wurde.

Seien $a, b, c \in V$ mit den Eigenschaften $\sigma^{-1}(a) < \sigma^{-1}(b) < \sigma^{-1}(c)$, $\{a, c\} \in E$ und $\{c, b\} \notin E$.

In dem Moment, in dem bei der Ausführung der Kardinalitätssuche b nummeriert wird, muss b zu mindestens genauso vielen bereits nummerierten Knoten benachbart sein wie a. Folglich, da a (aber nicht b) mit c verbunden ist, ist b benachbart zu einem anderen zuvor nummerierten Knoten x, der nicht mit a verbunden ist. Da dies für beliebige a, b und c gilt, besitzt die gesamte Sortierung σ die Eigenschaft P. □

Um bewiesen zu haben, dass mit der Kardinalitätssuche wirklich eine perfekte Eliminations-Reihenfolge berechnet wird, fehlt somit noch folgende Aussage:

Satz 3. *Sei G ein chordaler Graph.*
Falls die Reihenfolge σ von den Knoten von G die Eigenschaft P erfüllt, so ist σ eine perfekte Eliminations-Reihenfolge.[15]

[13] siehe [TY84, Lemma 4]
[14] siehe [TY84, Satz 2]
[15] siehe [TY84, Lemma 4] mit für perfekte Eliminationsreihenfolgen abgewandeltem Beweisende

Beweis. Sei σ eine Reihenfolge der Knoten des chordalen Graphen $G = (V, E)$, die die Eigenschaft P aufweist.

Ein Weg $\pi = (v_0, \ldots, v_k)$ in G mit $k \geq 2$ besitzt die Eigenschaft Q genau dann, wenn folgende Aussagen erfüllt sind:

- π besitzt keine Zwischenverbindungen (für beliebige $i, j \in \{1, \ldots k\}, |i - j| \neq 1$ gelte $\{v_i, v_j\} \notin E$).
- Für ein $i \in \{1, 2, \ldots, k-1\}$ sei $\sigma^{-1}(v_0) > \sigma^{-1}(v_k) > \sigma^{-1}(v_1) > \sigma^{-1}(v_2) > \ldots > \sigma^{-1}(v_i)$ und $\sigma^{-1}(v_i) < \sigma^{-1}(v_{i+1}) < \ldots < \sigma^{-1}(v_k)$.

Unser Ziel ist nun zu zeigen, dass ein solcher Weg mit Eigenschaft Q nicht existieren kann: Dazu nehmen wir an, das π so gewählt ist, dass es keinen Pfad mit größerem $\sigma^{-1}(v_k)$ gibt, der Q erfüllt.

Da π keine Zwischenverbindung besitzt, sind zwar v_0 und v_1 benachbart, aber v_0 und v_k nicht. Da durch Q die Ungleichung $\sigma^{-1}(v_1) < \sigma^{-1}(v_k) < \sigma^{-1}(v_0)$ gilt, existiert nach P ein Knoten x mit $\sigma^{-1}(v_k) < \sigma^{-1}(x)$, der zu v_k, aber nicht zu v_1, benachbart ist.

Sei $j > 0$ minimal mit $v_j \in \mathrm{Adj}(x)$. x ist nicht zu v_0 benachbart, da sonst $(x, v_0, v_1, \ldots, v_j, x)$ ein Kreis ohne Sehne wäre, was der Chordalität von G widersprechen würde. Falls nun $\sigma^{-1}(v_0) > \sigma^{-1}(x)$ gelten würde, besäße $(v_0, v_1, \ldots, v_j, x)$ die Eigenschaft Q; im anderen Fall $\sigma^{-1}(v_0) < \sigma^{-1}(x)$ wäre es $(x, v_j, v_{j-1}, \ldots, v_0)$.

Dies steht im Widerspruch dazu, dass $\sigma^{-1}(v_k)$ maximal gewählt ist, womit kein Pfad die Eigenschaft Q besitzen kann.

Betrachte nun mit $i \in \{1, \ldots, |V|\}$ beliebig den Knoten $\sigma(i)$. Um zu beweisen, dass σ eine perfekte Eliminations-Reihenfolge ist, muss hier somit gezeigt werden, dass $\sigma(i)$ in dem Teilgraphen $G[\{\sigma(i), \ldots, \sigma(|V|)\}]$ simplizial ist:

Seien also $v, w \in \mathrm{Adj}(\sigma(i)), v \neq w$ mit $i < \sigma^{-1}(v)$ und $i < \sigma^{-1}(w)$. v und w sind somit benachbart, da sonst entweder die Wege $(v, \sigma(i), w)$ oder $(w, \sigma(i), v)$ die Eigenschaft Q erfüllen würden. □

Korollar 1. *Ein Graph $G = (V, E)$ ist genau dann chordal, wenn die von der Kardinalitätssuche auf G generierte Ordnung σ eine perfekte Eliminations-Reihenfolge ist.*

Beweis. ⇒ Für eine von der Kardinalitätssuche auf einem chordalen Graphen G generierte Reihenfolge σ wurde in Satz 2 gezeigt, dass diese P erfüllt. Daraus folgt nach Satz 3, dass es sich bei σ um eine perfekte Eliminations-Reihenfolge handelt.

$\Leftarrow$ Wenn ein von der Kardinalitätssuche auf G generiertes σ eine perfekte Eliminationsreihenfolge ist, folgt mithilfe der Äquivalenz aus Satz 1, dass G chordal ist.

□

Um einen Graphen G auf Chordalität zu überprüfen, können wir somit zuerst mithilfe der Kardinalitätssuche die Reihenfolge σ berechnen, und müssen anschließend noch überprüfen, ob es sich bei σ um eine perfekte Eliminations-Reihenfolge handelt. Genauer beschrieben wird dieser letzte Schritt beispielsweise in [Gol04, Algorithmus 4.2], der mit $\mathcal{O}(|V| + |E|)$ Zeit auskommt.

3 Zeit- / Platz-Tradeoffs für die Kardinalitätssuche

Der in Abschnitt 2 vorgestellte Algorithmus besitzt zwar eine lineare Laufzeit, verwendet aber mit $\mathcal{O}(n \cdot \lg(n))$ bits relativ viel Speicher. Dies kann vor allem dann störend werden, wenn Zeit keine Priorität ist und große Datenmengen mit einem kleinen beschreibbaren Arbeitsspeicher verarbeitet werden sollen (beispielsweise auf eingebetteten Systemen). Deshalb werden in [CS17] platzeffizientere Varianten der Kardinalitätssuche vorgestellt, die dafür eine längere Laufzeit besitzen:

Im Folgenden wird nun das Rechnermodell formal definiert, das auch schon implizit in Abschnitt 2 verwendet wurde:

Rechnermodell und Datenformat Zur Ausführung und Analyse der Algorithmen in dieser Arbeit wird das *Random Access Machine*-Modell verwendet, wobei für die Ausführung ein beliebig les- und beschreibbarer Speicher mit einer bestimmten Größenordnung zur Verfügung steht. Die für Adressierung eines Knotens benötigte Wortgröße liegt dabei in $\mathcal{O}(\lg(n))$ Bits.

Der Eingabegraph G mit seinen n Knoten und m Kanten ist in einem nur lesbaren separaten Speicher zu finden. Die Kanten sind als Adjazenzlisten $\mathrm{Adj}(x)$ verfügbar, womit ein beliebiger Nachbar eines Knotens $x \in V$ in $\mathcal{O}(1)$ Zeit ausgewählt werden kann.

Für die Ausgabe wird ein separater Speicher verwendet, aus dem bereits geschriebene Ausgaben weder gelesen noch verändert werden können.[16] In dieser Arbeit wird der Ausgabespeicher beginnend mit dem letzten Element von σ beschrieben. Bei einem nur sequentiell beschreibbaren Ausgabespeicher würden damit bei der Verwendung dieser Algorithmen

[16] Rechnermodell von [CS17], genauer spezifiziert in [AIK+14]

eine umgekehrte perfekte Eliminations-Reihenfolge berechnet werden können und damit zusätzliche Zeit und Kosten für das Umkehren der Reihenfolge anfallen.

Data: Graph $G = (V, E)$
for $v \in V$ **do**
  $\texttt{B}[v] \longleftarrow \texttt{false}$;
for $i \in \{n, \ldots, 1\}$ **do**
  $u \longleftarrow$ wähle beliebigen Knoten $x \in V$;
  $c \longleftarrow 0$;
  for $v \in V$ *mit* $\texttt{B}[v] = \texttt{false}$ **do**
    $k \longleftarrow 0$;
    for $w \in \mathrm{Adj}(v)$ *mit* $\texttt{B}[w] = \texttt{true}$ **do**
      $k \longleftarrow k + 1$;
    if $k \geq c$ **then**
      $u \longleftarrow v$;
      $c \longleftarrow k$;
  $\texttt{B}[u] = \texttt{true}$;
  $\sigma[i] \longleftarrow u$;
Result: Reihenfolge $\sigma : V \mapsto \{1, \ldots, n\}$ der Knoten in V
Algorithmus 2: Kardinalitätssuche mit $n + \mathcal{O}(\lg(n))$ bits Speicher nach [CS17, 3.1]

$n + \mathcal{O}(\lg(n))$ bits Bei dieser sehr speicher-fokussierten Realisierung der Kardinalitätssuche verwenden wir nur ein Bit-Array $\texttt{B} : V \mapsto \{\texttt{true}, \texttt{false}\}$ und Zählvariablen zum Iterieren der Knoten. Somit werden n bits für die Knoten und jeweils $\lg(n)$ bits für die Zählvariablen benötigt, also insgesamt $n + \mathcal{O}(\lg(n))$ bits.

Zu Beginn wird der gesamte Array $\texttt{B}$ mit $\texttt{false}$ (für nicht markiert) initialisiert.

Anschließend wird σ mit dem letzten Element beginnend schrittweise berechnet: In jedem Durchlauf $i \in \{n, \ldots, 1\}$ wird für jedes $v \in V$ geprüft, ob es noch nicht nummeriert wurde, also ob $\texttt{B}[v] = \texttt{false}$. Falls ja, wird die Adjazenzliste $\mathrm{Adj}(v)$ durchlaufen und die Anzahl der Elemente darin gezählt, die bereits nummeriert sind (also $\texttt{B} = \texttt{true}$). Dabei wird der Knoten u mit der in dem jeweiligen Schritt größten Anzahl an nummerierten Nachbarn bestimmt und anschließend so in die Position von $\sigma(i)$ des Ausgabespeichers geschrieben – womit er am Anfang der bereits berechneten Einträge steht. In Algorithmus 2 ist dieses Vorgehen als Pseudo-Code zusammengefasst.

Da für jeden der n Knoten in jedem Schritt einmal alle Adjazenzlisten durchlaufen werden (was $O(m)$ Zeit benötigt), hat diese Variante eine Laufzeit von $\mathcal{O}(n \cdot m)$.[17]

[17] siehe [CS17, Abschnitt 3.1]

$\mathcal{O}(n)$ **bits** Die folgende Version der Kardinalitätssuche verwendet deutlich mehr Speicher ($\mathcal{O}(n)$ statt n bits):

Neben dem Bit-Vektor `B` werden hier weitere Datenstrukturen zum Speichern der Knoten verwendet: In der doppelt verketteten Liste `L` besteht jedes Element aus einer Zahl `labeledNeighbors` für die Anzahl der nummerierten Nachbarn und einer weiteren verketteten Liste von Knoten, deren Einträge jeweils `labeledNeighbors` nummerierte Nachbarn besitzen. Um Platz zu sparen, werden immer nur $s \in \Theta\left(\frac{n}{\lg(n)}\right)$ Elemente in `S` und `T` gehalten. Die äußere Liste ist nach `labeledNeighbors` sortiert; auf das jeweils erste und letzte Element kann in $\mathcal{O}(1)$ zugegriffen werden. Um auf einen Knoten innerhalb der geschachtelten Liste innerhalb von $\mathcal{O}(\lg(n))$ zugreifen zu können, wird außerdem ein Suchbaum `T` angelegt, in dem alle Knoten zu finden sein sollen, die auch in `L` sind. Bei den Knoten in `L` und `T` sind Zeiger auf diese in der jeweils anderen Struktur eingetragen.

Da jeder Knoten mit $\mathcal{O}(\lg(n))$ bits in den Datenstrukturen `L` und `T` gespeichert werden kann und maximal $\Theta\left(\frac{n}{\lg(n)}\right)$ Knoten auf einmal geladen sind, wird für beide Strukturen $\mathcal{O}(n)$ Bits Speicher benötigt. `B` hat mit n Bits einen geringeren Speicherbedarf - womit hier insgesamt $\mathcal{O}(n)$ bits ausreichen.

Bei der Ausführung des Algorithmus' werden zuerst s Knoten in `L` und `T` geladen, bei `L` kommen dabei alle Knoten in die Liste mit `labeledNeighbors` $= 0$. `B` wird wieder komplett mit `false` initialisiert.

Im i-ten Schritt wird zuerst ein Knoten v aus `L` aus der Unterliste mit dem größten `labeledNeighbors` genommen. Dieser wird dann in σ in die Position $\sigma(n - i + 1)$ angefügt, in `B` als nummeriert markiert und anschließend aus `L` und `T` entfernt. Falls dadurch die entsprechende Liste, aus der der Knoten entfernt wurde, leer wird, wird das entsprechende Element aus `L` gelöscht.

Für jeden Nachbarn von $w \in \mathrm{Adj}(v)$ wird anschließend zuerst geprüft, ob sich dieser bereits in `T` (und somit auch `L`) befindet. Falls ja, wird dieser in das (eventuell neu erstellte) Listenelement von `L` mit einem um eins höheren `labeledNeighbors` eingefügt und aus seiner alten Unterliste entfernt. Zusätzlich wird die Referenz in `T` aktualisiert. Falls dieser noch nicht in `T` vorhanden ist, wird zuerst mithilfe von `B` die Anzahl der nummerierten Nachbarn bestimmt und in einer Variable k gespeichert. Falls diese Zahl größer ist als das kleinste `labeledNeighbors` in `L`, wird der Knoten in die entsprechende (eventuell neu erstellte) Unterliste mit `labeledNeighbors` $= k$ von `L` eingefügt, und - falls sonst die Maximalanzahl von s Knoten überschritten werden würden - der erste Knoten aus der Liste mit dem kleinsten `labeledNeighbors` in `L` (und `T`) entfernt.

Wenn während der Ausführung die in `L` und `T` zwischengespeicherten Knoten ausgehen,

werden wieder s neue, nicht-nummerierte Knoten nachgeladen (soweit vorhanden). Das Einfügen der ersten s Knoten erfolgt für jeden der Knoten v, indem die Nachbarn $w \in \mathrm{Adj}(v)$ mit $\mathtt{B}[w] = \mathtt{true}$ gezählt werden (diese Anzahl sei k) und anschließend v in die (eventuell neu erstellte Liste) mit $\mathtt{labeledNeighbors} = k$ eingefügt wird. Bei den restlichen Knoten wird genauso die Anzahl k der benachbarten, nummerierten Knoten berechnet, und falls diese größer ist als das kleinste vorhandene `labeledNeighbors` bei einem Element von `L`, wird ein Knoten aus der Liste mit dem kleinsten `labeledNeighbors` durch diesen wie bereits beschrieben ersetzt.

Bestimmen wir im Folgenden die Laufzeit dieser Variante der Kardinalitätssuche:

Bei einem Nachfüllen von `L` wird das Hinzufügen von einem Knoten durch die Zeit zum Durchlaufen der Nachbarn dominiert - und lässt sich somit mit $\mathcal{O}(m)$ abschätzen. Da das Nachfüllen der Liste höchstens $\lg(n)$ mal passiert, hat dies insgesamt eine Laufzeit von $\mathcal{O}(m \cdot \lg(n))$.

Das darauf folgende Einfügen der $s \in \Theta\left(\frac{\lg(n)}{n}\right)$ Knoten in den balancierten Baum `T` kostet jeweils $\mathcal{O}(\lg(n))$ Zeit, somit werden dafür insgesamt $\mathcal{O}(n)$ Rechenschritte benötigt.

Da bei jedem Verschieben von Knoten zwischen Listen ein solcher zuerst mit $\mathcal{O}(\lg(n))$ Aufwand in `T` gesucht werden muss, ist die Laufzeit für das Verschieben von Knoten $\mathcal{O}(\lg(n))$. Ein Knoten muss maximal $|\mathrm{Adj}(v)|$ mal verschoben werden (jedes Mal, wenn ein Nachbar nummeriert wird) - dies passiert maximal $\mathcal{O}\left(\sum_{w \in V} |\mathrm{Adj}(w)|\right) = \mathcal{O}(m)$ mal. Damit betragen die Laufzeitkosten dafür insgesamt $\mathcal{O}(m \cdot \lg(n))$.

Beim Berechnen der Anzahl der nummerierten Nachbarn eines Knoten v muss jeweils die Adjazenzliste durchlaufen werden ($\mathcal{O}(|\mathrm{Adj}(v)|)$ Zeit) - und da dies bei der Nummerierung jedes Nachbarn passieren kann, ergibt sich dafür insgesamt eine Laufzeit von $\mathcal{O}\left(\sum_{w \in V} |\mathrm{Adj}(w)|^2\right)$ Rechenschritten.

Die Autoren von [CS17] haben dies mit $\mathcal{O}\left(\sum_{w \in V} |\mathrm{Adj}(w)|^2\right) = \mathcal{O}\left(\frac{m^2}{n}\right)$ abgeschätzt. Dass diese Abschätzung nicht funktioniert, kann man an folgendem Beispiel erkennen:

Sei $G = (V, E)$ ein sternförmiger Graph, also mit $V = \{v_1, \ldots, v_n\}$ und

$$E = \{\{v_1, v_2\}, \ldots, \{v_1, v_n\}\}.$$

Dann ergibt sich für die linke Seite $\sum_{v \in V} |\mathrm{Adj}(v)^2 = 1 \cdot (n-1)^2 + (n-1) \cdot 1^2 \in \mathcal{O}(n^2)$ und für die rechte Seite $\frac{(n-1)^2}{n} \in \mathcal{O}(n)$, womit eine Gleichheit der Mengen nicht gegeben sein kann.

Eine gültige Abschätzung wäre

$$\mathcal{O}\left(\sum_{w\in V}|\mathrm{Adj}\,(w)|^2\right)\subseteq\mathcal{O}\left(\sum_{v\in V}\sum_{w\in V}|\mathrm{Adj}\,(v)|\cdot|\mathrm{Adj}\,(w)|\right)=\mathcal{O}\left(m^2\right),$$

die im Falle des beschriebenen sternförmigen Graphen eine Gleichheit wäre.

Die daraus resultierende Gesamtlaufzeit von $\mathcal{O}\left(m^2+m\cdot\lg\left(n\right)\right)$ (statt $\mathcal{O}\left(\frac{m^2}{n}+m\cdot\lg\left(n\right)\right)$)[18] besitzt jedoch schon bei Graphen mit $m\in\mathcal{O}\left(n\right)$ mit dieser Abschätzung eine schlechtere Laufzeit als die platzsparendere erste Version.

Zusammengefasst ergeben sich durch die beschriebenen Algorithmen folgende Zeit- / Speicherplatz-Tradeoffs:
Satz 4. *Es ist möglich, für einen Graphen $G=(V,E)$ mit $n=|V|$ Knoten und $m=|E|$ Kanten eine perfekte Eliminations-Reihenfolge σ zu erhalten in*[19]

a) $\mathcal{O}\left(n\cdot m\right)$ *Zeit mit* $n+\mathcal{O}\left(\lg\left(n\right)\right)$ *bits*

b) $\mathcal{O}\left(m^2+m\cdot\lg\left(n\right)\right)$ *Zeit mit* $\mathcal{O}\left(n\right)$ *bits*

c) $\mathcal{O}\left(n+m\right)$ *Zeit mit* $\mathcal{O}\left(n\cdot\lg\left(n\right)\right)$ *bits*

4 Anwendung beim Gauß-Algorithmus

Eine praktische Anwendung von chordalen Graphen und perfekten Eliminations-Reihenfolgen ist eine Möglichkeit zum Beschleunigen des dem Leser bereits bekannten Gauß-Algorithmus[20] auf dünnbesetzten Matrizen.

Eine *dünnbesetzte* Matrix ist eine Matrix, die deutlich mehr 0-Einträge als Einträge mit anderen Zahlen besitzt. Um sich Rechenoperationen bei der Ausführung des Gauß-Algorithmus zu sparen, ist es dabei von Interesse, diese 0-Einträge zu erhalten, da diese Einträge bei den Zeilenadditionen - falls man nur die Nicht-0-Einträge zusammen mit ihren Positionen speichert - ausgelassen werden können, und damit zusätzlich weniger Speicherplatz für die Matrix benötigt wird.

Für das folgende Beispiel werden wir diese symmetrische 8×8-Matrix betrachten, deren Diagonale vollständig besetzt ist[21]:

[18] siehe [CS17, Abschnitt 3.2]
[19] siehe [CS17, Satz 1] (Option c wurde hier ersetzt durch den Standard-Algorithmus)
[20] beispielsweise erklärt in [Sty16, Abschnitt 3.3]
[21] A muss symmetrisch sein, damit der später daraus generierte Graph ungerichtet ist. Die Diagonalen müssen $\neq 0$, damit der Gauß-Algorithmus ohne Zeilenvertauschungen auskommt.

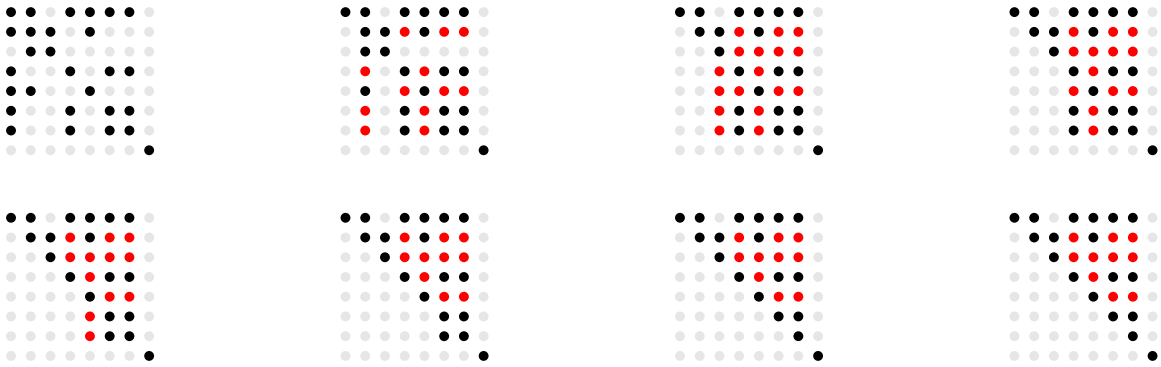

Abbildung 3: Schritte des Gauß-Algorithmus (zeilenweise) auf der Matrix A

$$A = \begin{bmatrix} 9 & 2 & 0 & 2 & 1 & 1 & 2 & 0 \\ 2 & 8 & 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 7 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 8 & 0 & 1 & 3 & 0 \\ 1 & 1 & 0 & 0 & 9 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 7 & 2 & 0 \\ 2 & 0 & 0 & 3 & 0 & 2 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Die genauen Einträge von A sind hier so gewählt, dass bei der Ausführung des Gauß-Algorithmus keine 0 in einer Spalte entsteht, die nicht gerade eliminiert wird. Dies stellt sicher, dass immer ein Pivotelement ohne Vertauschung gefunden werden kann.

Wenn wir darauf den Gauß-Algorithmus ausführen (Abbildung 3), werden im Laufe der Berechnung zusätzlich zu den bereits in der Originalmatrix vorhandenen Nicht-0-Einträgen ● einige weitere entstehen ●.

Dies kann jedoch mithilfe einer Vertauschung der Zeilen und Spalten der Matrix verhindert werden. Eine perfekte Eliminations-Reihenfolge ist eine solche Permutation, die uns zusätzliche Einträge erspart.

Dafür erstellen wir zuerst den *Eliminationsgraphen* G, der für jede Zeile/Spalte i von $A \in \mathbb{R}^{n \times n}$ einen Knoten v_i, $i \in \{0, \ldots, n-1\}$ enthält. Eine Kante von v_i zu v_j mit $i \neq j$ ist genau dann gegeben, wenn $A_{i,j} \neq 0$. G ist ungerichtet, da A symmetrisch ist. Der entstehende Graph ist der Beispielgraph aus Abbildung 1.[22]

Wenn wir nun die perfekte Eliminations-Reihenfolge $(4, 6, 7, 1, 5, 3, 2, 8)$ aus Abbildung 2 verwenden, um die Zeilen und Spalten jeweils danach zu sortieren, erhalten wir folgende

[22] siehe Beschreibung aus [RTL76]

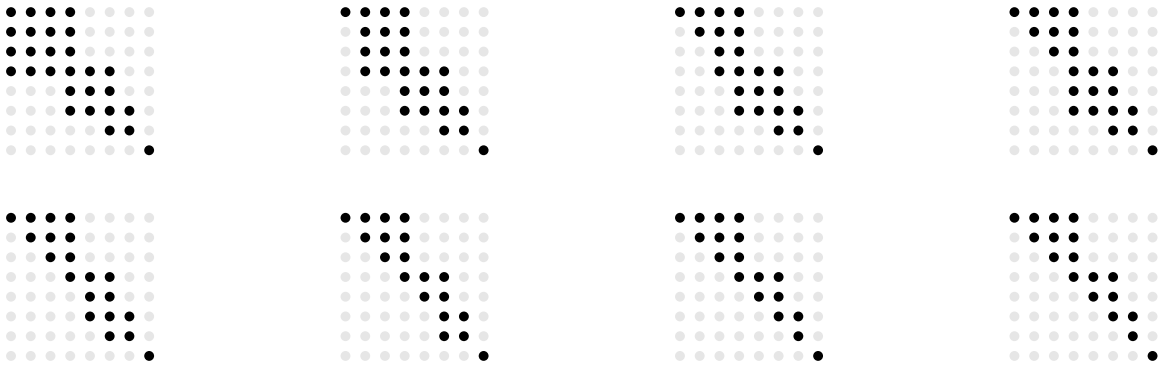

Abbildung 4: Schritte des Gauß-Algorithmus (zeilenweise) auf der permutierten Matrix A^*

Matrix:

$$P^{\mathrm{T}} \cdot A \cdot P = \begin{bmatrix} 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix} \cdot \begin{bmatrix} 9&2&0&2&1&1&2&0 \\ 2&8&2&0&1&0&0&0 \\ 0&2&7&0&0&0&0&0 \\ 2&0&0&8&0&1&3&0 \\ 1&1&0&0&9&0&0&0 \\ 1&0&0&1&0&7&2&0 \\ 2&0&0&3&0&2&9&0 \\ 0&0&0&0&0&0&0&8 \end{bmatrix} \cdot \begin{bmatrix} 0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix} = \begin{bmatrix} 8&3&1&2&0&0&0&0 \\ 3&9&2&2&0&0&0&0 \\ 1&2&7&1&0&0&0&0 \\ 2&2&1&9&1&2&0&0 \\ 0&0&0&1&9&1&0&0 \\ 0&0&0&2&1&8&2&0 \\ 0&0&0&0&0&2&7&0 \\ 0&0&0&0&0&0&0&8 \end{bmatrix} = A^*$$

Der Gauß-Algorithmus auf A^* erzeugt keine zusätzlichen Nicht-0-Einträge – wie auch in Abbildung 4 zu erkennen ist.

Ein Beweis dafür, dass dies kein glücklicher Zufall ist, lässt sich wie folgt skizzieren:

Beweis. Um sicherzustellen, dass immer das Element $A_{1,1} \neq 0$ ist und somit als Pivotelement gewählt werden kann, wird davon ausgegangen, dass nie ein Element außerhalb der Spalte, die eliminiert werden soll, zufällig 0 wird.

Induktionsvoraussetzung: Sei A eine $t \times t$-Matrix, bei der die Nulleinträge symmetrisch sind, also für $i, j \in \{1, \ldots, t\}$ gilt die Aussage $A_{i,j} = 0 \Leftrightarrow A_{j,i} = 0$. Außerdem sei in A die Diagonale vollständig besetzt und die Zeilen nach einer perfekten Eliminations-Reihenfolge sortiert. Dann lässt sich der Gauß-Algorithmus auf A durchführen, ohne dabei zusätzliche Nicht-0-Einträge zu erhalten.

$t = 1$ Da bei einer 1×1-Matrix kein Eliminationsschritt durchgeführt werden muss, entstehen hier auch keine neuen Einträge, die nicht Null sind.

$t \Rightarrow t + 1$ Sei $J = \{j \in \{2, \ldots, t\} \mid A_{j,1} \neq 0\}$ die Menge der Indices der Zeilen von A, bei denen der Eintrag in der ersten Spalte eliminiert werden muss – und somit die erste Zeile auf diese addiert werden soll.

Durch die Einträge der ersten Spalte von A sind in dem zugehörigen Eliminationsgraphen die Knoten, die zu den Zeilen von J gehören, mit dem Knoten der zur ersten Zeile gehört verbunden. Da die Zeilen von A nach einer perfekten Eliminations-Reihenfolge sortiert worden sind und diese mit dem simplizialen Knoten an erster Stelle in σ verbunden sind, sind alle Knoten, die zu den Zeilen mit Index J gehören, auch untereinander verbunden.

Dies bedeutet, dass für $i, j \in J$ der Eintrag $A_{i,j}$ nicht 0 sein kann. (Im Falle von $i = j$ folgt dies aus der voll besetzten Diagonalen.) Da die erste Zeile wegen der Symmetrie von A neben dem ersten Eintrag nur in Elementen mit einem Index aus J Nicht-0-Einträge besitzt, werden diese bei der Elimination (also der skalierten Addition auf die Zeilen mit Index in J) nie auf 0-Einträge der entsprechenden Zeile addiert, womit kein zusätzlicher Nicht-0-Eintrag entstehen kann.

Sei A' die Matrix nach dem Eliminationschritt. Die Teilmatrix $A'_{\{2,\dots,t\},\{2,\dots,t\}}$, die man erhält, wenn man von A' die erste Zeile und Spalte entfernt, erfüllt wieder die Eigenschaften der Induktionsvoraussetzung, wodurch auch in späteren Schritten keine zusätzlichen Nicht-0-Einträge hinzukommen.

□

5 Fazit

Im ersten Teil der Arbeit wurde mit der Existenz einer perfekten Eliminations-Reihenfolge eine äquivalente Aussage zu der Chordalität des Graphen dargestellt und bewiesen. Anschließend haben wir mit der Kardinalitätssuche einen Algorithmus gezeigt, der dazu verwendet werden kann, eine perfekte Eliminations-Reihenfolge – falls möglich – zu finden und damit chordale Graphen zu erkennen. Daraufhin wurden, neben der Standard-Implementierung, zwei platzeffizientere Varianten vorgestellt. Zum Abschluss haben wir außerdem mit der Erhaltung von 0-Einträgen in dünnbesetzten Matrizen mithilfe von perfekten Elimination-Reihenfolgen beim Gauß-Algorithmus eine praktische Anwendung der vorgestellten theoretischen Grundlagen präsentiert.

6 Literaturverzeichnis

[AIK+14] Asano, Tetsuo ; Izumi, Taisuke ; Kiyomi, Masashi ; Konagaya, Matsuo ; Ono, Hirotaka ; Otachi, Yota ; Schweitzer, Pascal ; Tarui, Jun ; Uehara,

Ryuhei: Depth-First Search Using $O(n)$ bits. In: *International Symposium on Algorithms and Computation* Springer, 2014, S. 553–564

[CS17] Chakraborty, Sankar D. ; Satti, Srinivasa R.: Space-Efficient Algorithms for Maximum Cardinality Search, Stack BFS, Queue BFS and Applications. In: *Computing and Combinatorics - 23rd International Conference, COCOON 2017, Hong Kong, China, August 3-5, 2017, Proceedings*, 2017, S. 87–98

[Gol04] Golumbic, Martin C.: *Annals of Discrete Mathematics*. Bd. 57: *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 2004

[RTL76] Rose, Donald J. ; Tarjan, R E. ; Lueker, George S.: Algorithmic aspects of vertex elimination on graphs. In: *SIAM Journal on computing* 5 (1976), Nr. 2, S. 266–283

[Sty16] Stykel, Tatjana: *Vorlesungsskript: Lineare Algebra 1*. Institut für Mathematik – Universität Augsburg, 2016

[TY84] Tarjan, Robert E. ; Yannakakis, Mihalis: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. In: *SIAM J. Comput.* 13 (1984), Juli, Nr. 3, 566–579. http://dx.doi.org/10.1137/0213035. – DOI 10.1137/0213035. – ISSN 0097–5397