Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Experimentelle Bestimmung vergleichsoptimaler Sortieralgorithmen

Julian Obst

Studiengang: Informatik

Prüfer/in: Prof. Dr. Volker Diekert

Betreuer/in: Dr. Armin Weiß

Beginn am: 5. Juni 2019

Beendet am: 5. Dezember 2019

Kurzfassung

Die gängigsten Sortierverfahren sortieren Mengen, indem sie zwei Elemente miteinander vergleichen und damit Schritt für Schritt eine Ordnung aufbauen. Es stellt sich die Frage, was die minimale Anzahl an Vergleichen ist, die benötigt wird, um eine beliebige Permutation einer Menge zu sortieren. Das zentrale Ziel dieser Arbeit ist es, die theoretischen Grundlagen zur Bestimmung vergleichsoptimaler Sortieralgorithmen zu erläutern und die Anzahl der Vergleiche von solchen Algorithmen zu bestimmen. Dabei werden auch die praktischen Ansätze besprochen, die benutzt wurden, um einen Algorithmus zu erstellen, der dieses Problem lösen soll. In diesem Zusammenhang werden die Aspekte der Implementierung dieses Algorithmus beleuchtet. Vor allem werden in diesem Punkt aber auch Optimierungen beschrieben, die helfen sollen, die Laufzeit zu verkürzen. Für $n = 12, 13$ konnten bekannte Ergebnisse bestätigt werden.

Inhaltsverzeichnis

1 Einleitung **13**

2 Theoretische Aspekte **15**
2.1 Theoretische Grundlagen 15
2.1.1 Informationstheoretische untere Schranke 15
2.1.2 Efficiency 16
2.2 Algorithmus 17
2.2.1 Forwardsteps 17
2.2.2 Backwardsteps 19
2.3 Lineare Erweiterungen zählen 21

3 Implementierung in der Praxis **25**
3.1 Grundlagen 25
3.1.1 Die Klasse PosetObj 25
3.2 Algorithmen 26
3.2.1 Lineare Erweiterungen 26
3.2.2 Forwardsteps 29
3.2.3 Kongruenz-Check 29
3.2.4 Transitive Reduktion 31
3.2.5 Backwardsteps 34
3.2.6 Sortable 35
3.3 Parallelität 36
3.4 Ergebnisse 38

4 Zusammenfassung und Ausblick **41**

Literaturverzeichnis **43**

Abbildungsverzeichnis

2.1 Beispiel Ordnung [Pec11, Abb. 6] 21
2.2 $\mathcal{G}$ zum Beispiel Graph 2.1 [Pec11, Abb. 6] 22
2.3 $\mathcal{G}$ Die Beschriftung eines Knoten D steht für $[d(D), u(D)]$ [Pec11, Abb. 7] 23
2.4 Visualisierung der Mengen V und $\overline{W}$. 24

3.1 Visuelle Darstellung der 4 Fälle in Tabelle t 29
3.2 Veranschaulichung der Transitiven Reduktion. 34
3.3 Die Mengen $\mathcal{S}$ und $\mathcal{S}^*$ für $n = 4$, $C = 5$ 38

Tabellenverzeichnis

2.1 Tabelle $t[j,k]$ zur Ordnung 2.1 [Pec11, Tab. 2] 24

3.1 Detaillierte Auflistung der Anzahl der partiellen Ordnung in $\mathcal{S}$, $\mathcal{S}^*$ für $n = 12$ (l.) und $n = 13$ (r.) 39

Verzeichnis der Algorithmen

2.1 Forwardsteps zur Berechnung von $\mathcal{S}$ 19
2.2 Backwardsteps zur Berechnung von $\mathcal{S}^*$ 20
2.3 Sortable Funktion 21
3.1 Transitive Reduktion 33

1 Einleitung

Die asymptotische Komplexität n Elemente zu sortieren liegt in $O(n \log n)$. Der Versuch dieser Arbeit ist, diese Schranke in konkrete Zahlen zu fassen. Zentrales Ziel ist es, für eine Zahl n eine pessimistische Untere Schranke für die Anzahl der Vergleiche anzugeben, sodass diese Anzahl ausreicht, um n Elemente zu sortieren. Zwar gibt es Sortierverfahren wie Radixsort, die ohne Vergleiche sortieren können, hier sind allerdings nur Sortieralgorithmen relevant, die mit Vergleichen arbeiten. Ebenso wird davon ausgegangen, dass die n Elemente paarweise verschieden sind, es gibt also keine Duplikate. Knuth [Knu98] beschäftigte sich bereits im Jahre 1968 mit diesem Problem. Damals waren die Werte für $n = 1, \ldots, 12$ bekannt. Mittlerweile sind dank Arbeiten von Peczarski [Pec02] auch die Werte für $n = 13, 14, 15$ bekannt. Bislang noch unbekannt ist der Wert für $n = 16$. Tatsächlich gibt es einen Sortieralgorithmus, der für die bekannten n genau diese Anzahl an Vergleichen benötigt. Dies ist der *Ford-Johnson Algorithmus*, auch bekannt als *Merge-Insertion Sort*. Allerdings ist dieser Algorithmus nicht immer optimal, denn für $n = 47$ ist ein Algorithmus bekannt, der weniger Vergleiche als Merge-Inserstion Sort benötigt [Pec07].

Notation. Die Elemente, die sortiert werden sollen, stammen aus der Menge $U = \{u_0, u_1, \ldots, u_{n-1}\}$ Die Anzahl der Elemente von U wird mit $n = |U|$ bezeichnet. Bei der Untersuchung der Verfahren bezeichnet C die Anzahl der Vergleiche, die wir maximal machen. Bei der Untersuchung, ob 5 Elemente in 7 Vergleichen sortiert werden können, ist also $n = 5$ und $C = 7$. Der aktuelle Vergleich wird mit c bezeichnet.

Die einfachste Art und Weise unser Problem zu lösen, ist, alle verschiedene Sortierverfahren auszuprobieren und bei jedem Verfahren zu schauen, ob am Ende eine lineare Ordung hergestellt wurde. Die Schwierigkeit liegt allerdings in der Größe des Problems. Da für einen Vergleich immer genau zwei Elemente ausgewählt werden, gibt es bei n Elementen genau $\binom{n}{2}$ Paare, die verglichen werden können. Hat der Algorithmus nach höchstens C Vergleichen alle Elemente sortiert, so hat er C verschiedene Paare ausgewählt und verglichen. Da kein Vergleich zweimal gemacht wird, kann im ersten Schritt aus allen Paaren gewählt werden, im zweiten aus allen verbliebenen usw. Im Gesamten ergibt das $\binom{n}{2} \cdot (\binom{n}{2} - 1) \cdot \ldots \cdot (\binom{n}{2} - C)$ verschiedene Möglichkeiten Paare zu vergleichen. Auf eine Formel gebracht:

$$\frac{\binom{n}{2}!}{(\binom{n}{2} - C)!}$$

Für 4 Elemente in 5 Vergleichen gibt es 720 Verfahren. Bei 5 Elementen in 7 Vergleichen sind es bereits 604 800 Verfahren. Spätestens bei 7 Elementen müssten ca. $1.2 \cdot 10^{15}$ Verfahren überprüft werden, was außerhalb des sinnvoll Lösbaren liegt.

2 Theoretische Aspekte

Um in die Thematik einzuführen und die Grundlagen zu verstehen, werden die theoretischen Aspekte der Arbeit an dieser Stelle dargelegt. Besonderes Augenmerk liegt auf dem Algorithmus, der in Kapitel 3 seine praktische Anwendung findet.

2.1 Theoretische Grundlagen

2.1.1 Informationstheoretische untere Schranke

Egal wie gut ein Sortieralgorithmus ist, der unseren Anforderungen genügt, er wird eine Mindestanzahl an Vergleichen nicht unterbieten können. Genauer: Um n Elemente zu sortieren, sind mindestens $\lceil \log_2 n! \rceil$ Vergleiche nötig. Diese Zahl bezeichnet man als *Informationstheoretische untere Schranke*, im Folgenden mit $C(n)$ für $n \in \mathbb{N}$ abgekürzt. Die tatsächliche Anzahl an Vergleichen, die ein Sortieralgorithmus benötigt, wird mit $S(n)$ bezeichnet. Wir können einen Sortieralgorithmus als binären Entscheidungsbaum auffassen, dessen innere Knoten den Vergleich von zwei Elementen u_j und u_k darstellt. Der linke Teilbaum eines Knotens beschreibt, dass das Ergebnis des Vergleichs $u_j < u_k$ war, entsprechend der rechte $u_j > u_k$. Überflüssige Vergleiche sind hier möglich, bspw. für die Ergebnisse $u_j < u_k$, $u_k < u_l$ wäre der Vergleich $u_j : u_l$ überflüssig. Da die Vergleiche minimiert werden sollen, gehen wir davon aus, dass im Algorithmus keine überflüssigen Vergleiche gemacht werden. Ein Blatt repräsentiert eine Permutation der n Elemente. Da auf jeder Ebene des Baumes der gleiche Vergleich gemacht wird, sofern dieser nicht überflüssig ist, erhalten wir an einem Blatt eine eindeutige Permutation. Der Algorithmus muss mit jeder Permutation umgehen können, also ist jede Permutation als Blatt möglich. Damit gibt es genau $n!$ viele Blätter in einem binären Entscheidungsbaum, der keine überflüssigen Vergleiche anstellt. Es sei nun der Baum zu einem Sortieralgorithmus gegeben, der maximal $S(n)$ Vergleiche benötigt. Damit sind alle Blätter auf einer Ebene $< S(n)$ (Wurzel ist auf Ebene 0). Daraus folgt, dass der Baum höchstens $2^{S(n)}$ Blätter hat. Aufgrund der Konstruktion des Baums wissen wir, dass er $n!$ viele Blätter besitzt und somit gilt:

$$2^{S(n)} \geq n!$$

Da die Anzahl der Vergleiche eine ganze Zahl sein muss, lässt sich die Ungleichung umstellen nach

$$S(n) \geq \lceil \log_2 n! \rceil.$$

Die folgende Tabelle zeigt für verschiedene n die Informationstheoretische untere Schranke und die bekannten Werte für $S(n)$.

n	=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\lceil \log_2 n! \rceil$	=	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45
$S(n)$	=	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	?

Definition 1 (Partielle Ordnungen)**.** Um ein optimales Sortierverfahren zu finden, betrachten wir Partielle Ordnungen P über einer Menge U. Der etwas knappere englische Begriff lautet „Poset“ (partially ordered set). Für unser Vorhaben sei $U = \{u_0, u_1, \ldots, u_{n-1}\}$ eine n-elementige Menge. Damit ist $P = (U, R)$, wobei $R = \{(u, v) : u, v \in U\}$ eine partielle Ordnungsrelation ist. Durch einen weiteren Vergleich fügen wir eine neue Relation hinzu, dabei bezeichnet $P + u_j u_k$ die transitive Hülle von $R \cup \{(u_j, u_k)\}$. P lässt sich als Graph interpretieren, dessen Knotenmenge U und Kantenmenge R ist. Eine Kante (u_j, u_k) lässt sich dann als $u_j < u_k$ verstehen. Die partielle Ordnung $P^* = (U, R^*)$ ist die duale Ordnung von $P = (U, R)$, mit $R^* = \{(v, u) : (u, v) \in R\}$. Graphisch bedeutet das, dass in P^* alle Kanten umgedreht sind.

Definition 2 (Lineare Erweiterung)**.** Sei P eine partielle Ordnung und L eine lineare Ordnung und es gilt $P \subseteq L$. Dann wird L *lineare Erweiterung* von P genannt [Pec02]. Die Anzahl der linearen Erweiterungen einer partiellen Ordnung P wird mit $e(P)$ bezeichnet.

Eine lineare Erweiterung ist also eine Permutation, die konsistent mit P ist, also keine der Relationen aus R verletzt. $e(P)$ ist deshalb gleich der Anzahl an Varianten, den Knoten die Zahlen aus $\{1, \ldots, n\}$ zuzuweisen, sodass die Zahl von Knoten u_j kleiner als die Zahl von Knoten u_k ist, für jede Kante (u_j, u_k). Die Zahl, die Knoten u_j zugewiesen wird, wird mit $L(u_j)$ bezeichnet. Interpretiert man die Relationen aus R nicht als $u_j < u_k$, sondern als u_j muss vor u_k in der linearen Ordnung stehen, ist leicht zu erkennen, dass $e(P)$ gleich der Anzahl an verschiedenen Möglichkeiten ist den Graphen von P topologisch zu sortieren. Eine lineare Ordnung hat offensichtlich genau eine Permutation, die mit ihr konsistent ist ($e(P) = 1$). Für eine partielle Ordnung, bei der alle Elemente unabhängig sind, ist jede Permutation konsistent, also hat diese Ordnung $n!$ lineare Erweiterungen.

2.1.2 Efficiency

Für einen Graphen P mit n Knoten, der nach c Vergleichen entstanden ist, bezeichnet

$$E(P) = \frac{n!}{2^c e(P)}$$

die *Efficiency* des Graphen [Knu98]. Die Efficiency wird benutzt, um eine Aussage über die Güte der partiellen Ordnung zu treffen.

Sei P eine partielle Ordnung, die nach c Vergleichen entstanden und nicht vollständig sortiert ist. Ein weiterer Vergleich von zwei unabhängigen Elementen hat zwei weitere partielle Ordnungen zur Folge. Nämlich $P_1 = P + u_j u_k$ und $P_2 = P + u_k u_j$. Es ist leicht zu sehen, dass gilt

$$e(P) = e(P_1) + e(P_2).$$

Jede lineare Erweiterung von P hat entweder $u_j < u_k$ oder $u_j > u_k$. Entsprechend ist genau dann eine Erweiterung von P konsistent mit P_1, wenn $u_j < u_k$, mit P_2 genau dann, wenn $u_j > u_k$ gilt. Sei nun o.B.d.A. $e(P_1) \geq e(P_2)$, es folgt

$$e(P) \leq 2e(P_1), \tag{2.1}$$

$$E(P_1) = \frac{n!}{2^{c+1} e(P_1)} = \frac{n!}{2^c \cdot 2e(P_1)} \cdot \frac{e(P)}{e(P)} = \frac{n! e(P)}{2^c e(P) 2e(P_1)} = \frac{E(P)e(P)}{2e(P_1)} \overset{(2.1)}{\leq} E(P) \tag{2.2}$$

Daraus lässt sich ablesen, dass jeder Vergleich eine partielle Ordnung von gleicher oder geringerer Efficiency als Ergebnis hat. Die Efficiency lässt sich durch Vergleiche also nicht verbessern. Bei einer partiellen Ordnung ohne Relationen wurden noch keine Vergleiche angestellt, $c = 0$ und wie oben erwähnt gilt $e(P) = n!$. Die initiale Efficiency beträgt also 1. Das Ziel ist eine lineare Ordnung P_l, für sie gilt $e(P_l) = 1$. Auf der Suche nach einem Sortierverfahren von bspw. 5 Elementen, die mittels 7 Vergleichen oder weniger sortiert werden sollen, muss am Ende eine lineare Ordnung stehen mit Efficiency $\frac{5!}{2^7 \cdot 1} = \frac{120}{128} = \frac{15}{16}$. Jede partielle Ordnung des Sortierverfahrens muss eine Efficiency $\geq \frac{15}{16}$ haben. Denn sollte eine partielle Ordnung eine geringere Efficiency haben, so hat auch mindestens ein Nachfolger eine geringere Efficiency. Schlussendlich erreicht man eine lineare Ordnung mit Efficiency $< \frac{15}{16}$.

Dieses Beispiel lässt sich verallgemeinern und umformen. Wie oben erwähnt, hat eine lineare Ordnung P_l genau eine lineare Erweiterung (sie selbst). Die Efficiency der linearen Ordnung ist also $\frac{n!}{2^C}$. Für eine beliebige partielle Ordnung P, die nach c Vergleichen entstanden ist, muss Folgendes gelten:

$$\begin{aligned} E(P) = \frac{n!}{2^c e(P)} \geq \frac{n!}{2^C} &= E(P_l) \\ 2^c e(P) &\leq 2^C \\ e(P) &\leq 2^{C-c} \end{aligned}$$

Liegt also nach c Vergleichen eine partielle Ordnung vor, die mehr als 2^{C-c} lineare Erweiterungen besitzt, kann diese nicht in den verbliebenen $C - c$ Vergleichen sortiert werden. Entsprechend sollte der letzte Vergleich nicht in Schritt c gemacht werden.

2.2 Algorithmus

Mit diesem Wissen lässt sich ein besserer Algorithmus als der aus Kapitel 1 formulieren. Der Algorithmus gibt uns die Antwort auf die Frage, ob eine gegebene partielle Ordnung in den verbliebenen $C - c$ Vergleichen sortiert werden kann. Der Algorithmus kann dann auf die partielle Ordnung ohne Relationen aufgerufen werden, mit der Frage, ob diese in C Vergleichen sortierbar ist. Der Algorithmus geht auf Wells [Wel71] zurück und wurde von Peczarski [Pec02; Pec04; Pec11] verbessert.

Der Algorithmus besteht aus den sogenannten *Forwardsteps* und *Backwardssteps*. In den Forwardsteps generieren wir die partiellen Ordnungen, die die Anforderung der Efficiency nicht verletzen. In den Backwardssteps untersuchen wir diese Ordnungen darauf, ob sie wirklich in den verbliebenen Vergleichen sortierbar sind. Die Forwardsteps sind quasi ein Filter, der alle irrelevanten Ordnungen aussortiert, die eigentliche Prüfung sind die Backwardsteps.

2.2.1 Forwardsteps

Die Forwardsteps erstellen Schritt für Schritt eine Folge von Mengen, $\mathcal{S}$. $\mathcal{S}_c$ ist eine Menge von partiellen Ordnungen. Zur Erstellung von $\mathcal{S}_c$ werden partiellen Ordnungen aus $\mathcal{S}_{c-1}$ um eine Relation/Vergleich erweitert. c gibt die Anzahl der Vergleiche an, die für diese Ordnung bisher

gebraucht wurden, das heißt allerdings nicht, dass diese partielle Ordnung garantiert nach c Vergleichen entsteht, siehe Abschnitt 2.2.2. Wie genau entsteht also die Menge $\mathcal{S}_c$? Den Anfang bildet $\mathcal{S}_0$. Diese Menge enthält nur die partielle Ordnung P_0, dessen Ordnungsrelation lediglich aus $R = \{(u,u) : u \in U\}$ besteht. P_0 ist damit komplett unsortiert, und keine zwei verschiedene Elemente stehen in Relation. In Schritt $c > 0$ werden nun alle Paare $u_j u_k$ betrachtet, die nicht bereits in Relation zueinander stehen. Ein Vergleich dieser hat die partiellen Ordnungen P_1 und P_2 als Nachfolger. P_1 repräsentiert den Fall, dass $u_j < u_k$ und damit ist $P_1 = P + u_j u_k$. Analog gilt $P_2 = P + u_k u_j$. An diesem Punkt wird die Eigenschaft der Efficiency genutzt, d.h. es werden die Anzahl der Linearen Erweiterungen von P_1 und P_2 gezählt. Überschreitet einer der beiden Werte 2^{C-c}, so kann P nicht in den verbliebenen $C - c$ Vergleichen sortiert werden. In Schritt c sollten bei dieser partiellen Ordnung also u_j und u_k nicht miteinander verglichen werden. Sind beide Werte kleiner gleich der Grenze 2^{C-c}, so wird die partielle Ordnung mit mehr linearen Erweiterungen zu $\mathcal{S}_c$ hinzugefügt, da bei mehr Erweiterungen wahrscheinlicher die Grenze im späteren Verlauf überschritten wird. Für die Korrektheit des Ergebnisses ist es prinzipiell egal, welche der beiden Ordnungen hinzugefügt wird, da in den Backwardsteps sowohl P_1 als auch P_2 auf Sortierbarkeit überprüft werden.

Definition 3. Zwei partielle Ordnungen P_1, P_2 werden als *kongruent* bezeichnet, falls P_1 und P_2 oder P_1 und P_2^* isomorphe Graphen sind.

Lemma 1. *Kongruente Ordnungen benötigen die gleiche Anzahl an Vergleichen, um sortiert zu werden.*

Beweis. Angenommen, es gibt einen Algorithmus, der P_1 in c Vergleichen sortiert. Offensichtlich gilt das Lemma für isomorphe Ordnungen, ein Vergleich von zwei Elementen von P_1 wird einfach durch einen Vergleich der isomorphen Elemente aus P_2 angewendet. Für den Fall, dass P_1 und P_2^* isomorph sind, wird P_2^* nach dem Algorithmus sortiert und anschließend die entstandene lineare Ordnung umgedreht.

□

Beim Hinzufügen wird deshalb überprüft, ob nicht bereits eine kongruente Ordnung abgespeichert wurde. In diesem Fall wird nichts hinzugefügt. Es wird sowohl für P_1 als auch für P_2 nach kongruenten Ordnungen gesucht. Denn wie oben bereits erwähnt, wird die Ordnung, die nicht hinzugefügt wurde, in den Backwardsteps überprüft. Es geht also keine Ordnung verloren, wenn deshalb die Ordnung nicht zu $\mathcal{S}_c$ hinzufügt wird. Anstatt die Isomorphie auf der Transitiven Hülle der Graphen zu überprüfen, kann man dies auch auf der Transitiven Reduktion der Graphen, siehe Satz 2. Diese Prozedur wird mit jeder Ordnung aus $\mathcal{S}_{c-1}$ durchlaufen. Mit der Ergebnis Menge $\mathcal{S}_c$ wird der gleiche Algorithmus ausgeführt. Erreicht man schließlich C, gibt es zwei Möglichkeiten: Die Menge $\mathcal{S}_C$ enthält die lineare Ordnung mit n Elementen oder $\mathcal{S}_C$ ist leer. Ersteres bedeutet, dass die n Elemente in C Vergleichen sortiert werden könnten. Eine Garantie ist hier allerdings noch nicht gegeben, was der Fall $n = 13, C = 33$ zeigt [Pec02]. Grund dafür ist, dass bei den Forwardsteps lediglich eine der zwei partiellen Ordnungen in $\mathcal{S}_{c+1}$ aufgenommen wird, die durch einen Vergleich entstehen (nämlich $P + u_j u_k$ oder $P + u_k u_j$). Falls also $\mathcal{S}_C$ leer ist, lässt sich schließen, dass eine Sortierung in C Vergleichen nicht möglich ist. Dies ist bei $n = 12, C = 29$ der Fall.

Algorithmus 2.1 Forwardsteps zur Berechnung von $\mathcal{S}$

```
procedure FORWARDSTEPS(P = (U, R), C)          // C die verbliebene Anzahl an Vergleichen
    S_0 ← {P}
    for c ← 1, ..., C do
        for all P' ∈ S_{c-1} do
            for each unrelated Pair u_j, u_k do
                e(P_1) ← t[j, k]                                  // Berechnung von t nach 2.3
                e(P_2) ← t[k, j]                                  // P_1, P_2 werden nicht explizit erstellt
                if e(P_1) > 2^{C-c} or e(P_2) > 2^{C-c} then
                    continue        // Sortieren in C − c Vergleichen nicht möglich nach 2.1.2
                else                                              // Transitive Reduktion aus 3.1
                    P_1 ← TRANSITIVEREDUCTION(P' + u_j u_k)
                    P_2 ← TRANSITIVEREDUCTION(P' + u_k u_j)
                    if ISOMORPHIC(P_1, S_c) or ISOMORPHIC(P_2, S_c) then
                        continue // ISOMORPHIC ist true, wenn S_c kongruente Ordnung enthält
                    else
                        if e(P_1) ≥ e(P_2) then
                            S_c ← S_c ∪ {P_1}
                        else
                            S_c ← S_c ∪ {P_2}
                        end if
                    end if
                end if
            end for
        end for
    end for
    return S
end procedure
```

2.2.2 Backwardsteps

Wenn also die Forwardssteps erfolgreich ausgeführt werden, also $\mathcal{S}_C \neq \emptyset$, wird mit den Backwardsteps weitergemacht. Das Ergebnis ist die Folge $\mathcal{S}^*$. $\mathcal{S}^*_c$ enthält jene partielle Ordnungen, die tatsächlich in $C-c$ Vergleichen sortierbar sind. Dabei wird $\mathcal{S}$ von hinten durchgegangen. Den Anfang bildet $\mathcal{S}_C$, da die einzige Ordnung hier die lineare Ordnung ist und diese in $C - C = 0$ Vergleichen sortierbar ist, wird sie zu $\mathcal{S}^*_C$ hinzugefügt. In den folgenden Schritten wird für $c = C - 1, \ldots, 0$ jede Ordnung aus $\mathcal{S}_c$ betrachtet. Falls es für eine Ordnung P ein nicht verglichenes Paar (u_j, u_k) gibt, sodass $P_1 = P + u_j u_k$ und $P_2 = P + u_k u_j$ in $C - c - 1$ vergleichen sortierbar sind, fügen wir P zu $\mathcal{S}^*_c$ hinzu. Damit Sortierbarkeit überhaupt möglich ist, muss sich entweder P_1 oder P_2 bereits in $\mathcal{S}^*_{c+1}$ befinden. Falls sich keine der Ordnungen P_1, P_2 in $\mathcal{S}^*_{c+1}$ befindet, wurden diese entweder bereits in den Forwardsteps ausgeschlossen oder in den Backwardssteps wurde festgestellt, dass diese in den verbliebenen Vergleichen nicht sortierbar sind. In beiden Fällen sind P_1 und P_2 nicht sortierbar. Befindet sich also kein Nachfolger von P in $\mathcal{S}^*_{c+1}$, d.h. es wurden alle Paare (u_j, u_k) überprüft und bei keinem stellte sich heraus, dass die folgenden Ordnungen beide sortierbar sind, dann kann P auch nicht sortierbar sein. Folglich wird P nicht zu $\mathcal{S}^*_c$ hinzugefügt. Befindet sich allerdings mindestens

einer der Nachfolger in $\mathcal{S}^*_{c+1}$, muss deren Sortierbarkeit überprüft werden. Der einfachste Fall ist, dass $P_1 \in \mathcal{S}^*_{c+1}$ als auch $P_2 \in \mathcal{S}^*_{c+1}$. Damit steht fest, dass beide Ordnungen in $C-c-1$ Vergleichen sortierbar sind und damit ist P in $C-c$ Vergleichen sortierbar. Falls allerdings $P_1 \in \mathcal{S}^*_c$ und $P_2 \notin \mathcal{S}^*_c$, muss separat geprüft werden, ob P_2 in $C-c-1$ Vergleichen sortierbar ist. Die Prüfung erfolgt durch rekursiven Aufruf der Forward- und Backwardsteps. Hier können aber Informationen genutzt werden, die wir bereits berechnet haben, Genaueres in 3.1. Analog die Prüfung für $P_1 \notin \mathcal{S}^*_c$ und $P_2 \in \mathcal{S}^*_c$. Da aus $\mathcal{S}^*_c$ die partiellen Ordnungen ablesbar sind, die in $C-c$ Vergleichen sortierbar sind, lässt sich aus $\mathcal{S}^*_0$ ablesen, ob n Elemente in C Vergleichen sortierbar sind: Enthält $\mathcal{S}^*_0$ die Ordnung ohne Relationen (abgesehen von $(u,u) \in U \times U$), so sind n Elemente mit C Vergleichen sortierbar. Ist $\mathcal{S}^*_0$ leer, dann gibt es keine Möglichkeit, n Elemente in C Vergleichen zu sortieren.

Algorithmus 2.2 Backwardsteps zur Berechnung von $\mathcal{S}^*$

```
 1: procedure BACKWARDSTEPS(S, C)                    // C die verbliebene Anzahl an Vergleichen
 2:     S*_C ← S_C                                   // S_C enthält nur lineare Ordnung
 3:     for c ← C − 1, ..., 0 do
 4:         for all P ∈ S_c do
 5:             P_sortable ← false
 6:             for each unrelated Pair u_j, u_k do
 7:                 e(P_1) ← t[j, k]                 // Tabelle t von P
 8:                 e(P_2) ← t[k, j]                 // P_1, P_2 werden nicht explizit erstellt
 9:                 if e(P_1) > 2^(C−c−1) or e(P_2) > 2^(C−c−1) then
10:                     continue   // Sortieren in C − c − 1 Vergleichen nicht möglich nach 2.1.2
11:                 else                             // Transitive Reduktion aus 3.1
12:                     P_1 ← TRANSITIVEREDUCTION(P + u_j u_k)
13:                     P_2 ← TRANSITIVEREDUCTION(P + u_k u_j)
14:                     if P_1 ∈ S*_(c+1) and P_2 ∈ S*_(c+1) then   // P ∈ S*_(c+1) ist Kongruenzcheck
15:                         P_sortable ← true
16:                     else if P_1 ∈ S*_(c+1) and SORTABLE(P_2, C − c − 1) then
17:                         P_sortable ← true
18:                     else if P_2 ∈ S*_(c+1) and SORTABLE(P_1, C − c − 1) then
19:                         P_sortable ← true
20:                     else
21:                         continue
22:                     end if
23:                     if P_sortable then
24:                         S*_c ← S*_c ∪ {P}
25:                         continue with next P
26:                     end if
27:                 end if
28:             end for
29:         end for
30:     end for
31:     return S*
32: end procedure
```

Algorithmus 2.3 Sortable Funktion

```
procedure SORTABLE(P = (U, R), c)
    if e(P) ≤ 6 or number of unrelated pairs ≤ c then
        return true                                   // Begründung siehe 3.2.6 und [Pec04]
    end if
    S ← FORWARDSTEPS(P, c)
    if S_c = ∅ then
        return false
    end if
    S* ← BACKWARDSTEPS(S, c)
    return S*_0 ≟ ∅
end procedure
```

2.3 Lineare Erweiterungen zählen

Ein wichtiger Teil des Algorithmus ist die Berechnung der Anzahl der linearen Erweiterungen. Peczarski [Pec11] beschreibt dazu einen effizienten Algorithmus, der von [DDD06] inspiriert ist. Dieser berechnet neben der Anzahl der linearen Erweiterungen $e(P)$ auch noch eine Tabelle t, in der sich die Anzahl der linearen Erweiterungen befinden, nachdem der Ordnung eine Relation/Kante hinzugefügt wurde. Genauer gesagt, befindet sich in $t[j,k]$ ein Wert, der der Anzahl der linearen Erweiterungen von $P + u_j u_k$ entspricht, also $t[j,k] = e(P + u_j u_k)$ für $j \neq k$. Diese Tabelle ist für die Berechnung besonders praktisch, da mit ihr viele partielle Ordnungen ausgeschlossen werden können, ohne diese explizit zu erstellen. Der Algorithmus arbeitet mit sogenannten *Downsets*.

Definition 4 (Downset)**.** Für eine partielle Ordnung $P = (U,R)$ wird eine Teilmenge $D \subseteq U$ *Downset* genannt, falls für jedes $x \in D$ auch dessen Vorgänger $y \in U$ in D enthalten ist. Genauer gilt $\forall\, x \in D : (y,x) \in R \Rightarrow y \in D$.

Definition 5 (Upset)**.** Analog zu Definition 4 lassen sich die *Upsets* definieren. Für eine partielle Ordnung $P = (U,R)$ wird eine Teilmenge $A \subseteq U$ *Upset* genannt, falls für jedes $x \in A$ auch dessen Nachfolger $y \in U$ in A enthalten ist. Genauer gilt $\forall\, x \in D : (x,y) \in R \Rightarrow y \in D$.

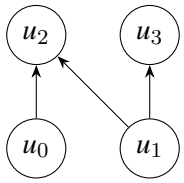

Abbildung 2.1: Beispiel Ordnung [Pec11, Abb. 6]

Die Downsets von 2.1 sind dementsprechend $\{\emptyset, \{u_0\}, \{u_1\}, \{u_0,u_1\}, \{u_1,u_3\}, \{u_0,u_1,u_2\}, \{u_0,u_1,u_3\}, \{u_0,u_1,u_2,u_3\}\}$. Aus den Downsets wird zur Veranschaulichung ein Graph $\mathcal{G}$ erstellt, dessen Knoten die einzelnen Downsets sind. Eine Kante wird von D_1 zu D_2 gezogen, falls $D_1 \subset D_2$ und falls ein $x \in U \setminus D_1$ existiert, sodass $D_1 \cup \{x\} = D_2$. Der Graph $\mathcal{G}$ hat unter Umständen 2^n Knoten, das ist bei der partiellen Ordnung ohne Relationen der Fall. Hier ist jede Teilmenge von U Downset und die Knotenmenge von $\mathcal{G}$ ist die Potenzmenge $\mathcal{P}(U)$.

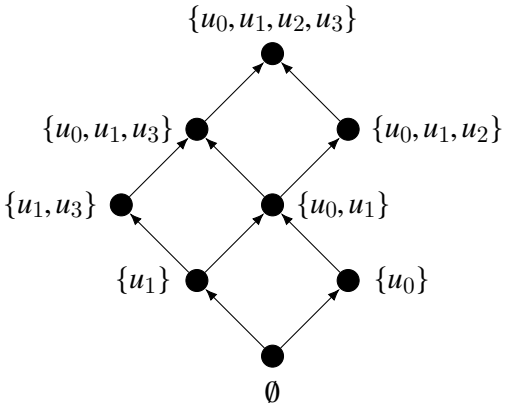

Abbildung 2.2: $\mathcal{G}$ zum Beispiel Graph 2.1 [Pec11, Abb. 6]

Für ein Downset D bezeichne $d(D)$ die Anzahl der linearen Erweiterungen von (D, R). (D, R) wird also interpretiert als die partielle Ordnung P reduziert zu den Elementen von D. Für ein Downset D bezeichne $u(D)$ die Anzahl der linearen Erweiterungen von $(U \setminus D, R)$, also dem Komplement von D. Es wird angenommen $d(\emptyset) = u(U) = 1$. Beachte, das Komplement eines Downsets ist ein Upset, denn U ist ein Upset und für ein Element aus D befinden sich all seine Vorgänger ebenfalls in D, dadurch werden mit $U \setminus D$ immer nur Vorgänger entfernt, die Nachfolger bleiben aber erhalten. Es gilt [DDD06]:

$$d(D) = \sum_{(X,D)} d(X)$$

wobei (X, D) eine Kante des Graphen $\mathcal{G}$ ist. Es wird über die Knoten der eingehenden Kanten summiert.

Beweis. Die Anzahl der linearen Erweiterungen ist gleich die Anzahl der Möglichkeiten D topologisch zu sortieren. Ein Algorithmus, der eine topologische Sortierung erstellt, sieht beispielsweise so aus: Wähle einen Knoten ohne Vorgänger, füge diesen als Teilergebnis der Sortierung hinzu, entferne den Knoten aus dem Graphen. Ein Downset lässt sich deshalb als ein Teilergebnis einer topologischen Sortierung interpretieren. Betrachten wir also Schritt für Schritt die Erstellung einer topologischen Sortierung. Sei M die Menge der Elemente, die bereits in der Sortierung sind, d.h. die Menge erfüllt die Bedingungen einer topologischen Sortierung. Induktion über die Größe der Menge M:

IA: Für leere Menge gilt nach Definition $d(\emptyset) = 1$. Da jede Menge M mit $|M| = 1$ genau auf eine Weise topologisch sortiert werden kann und in $\mathcal{G}$ lediglich eine eingehende Kante $(\emptyset, M)$ besitzt, gilt für solche Mengen: $d(M) = \sum_{(X,M)} d(X) = d(\emptyset) = 1$.

IV: $d(M) = \sum_{(X,M)} d(X)$ gilt für alle M mit $|M| < n, \quad n \in \mathbb{N}$.

IS: Sei nun $|M| = n$. Im beschriebenen Algorithmus kann M aus verschiedenen Mengen entstanden sein $M'_0, M'_1, \ldots, M'_l$ wobei $|M'_i| = n - 1, \quad i \in \{0, \ldots, l\}$ und $l \leq n - 1$. Für alle M'_i ist nach Induktion $d(M'_i)$ korrekt berechnet. Die verschiedenen Möglichkeiten M topologisch zu sortieren, ist gleich der Anzahl der Möglichkeiten im Algorithmus M zu erreichen. Da M aus den

verschiedenen M_i' entstehen kann, muss deren Anzahl, also $d(M_i')$ aufsummiert werden. Ergebnis: $d(M) = \sum_{(M_i',M)} d(M_i')$.

□

Ein Analoges Argument lässt sich für die Gleichung aus [DDD06]

$$u(D) = \sum_{(D,X)} u(X)$$

konstruieren. Hier wird über die Knoten der ausgehenden Kanten (D, X) des Graphen $\mathcal{G}$ summiert. Da U der kompletten partiellen Ordnung entspricht, gilt: $d(U) = u(\emptyset) = e(P)$. Die Werte $d(D)$ und $u(D)$ lassen sich entweder mit zwei Tiefen- oder Breitensuche bestimmen. Entsprechender Anfang der Suche ist der Knoten $(\emptyset)$ bzw. U. Für den Graphen aus 2.2 findet sich die Berechnung in 2.3.

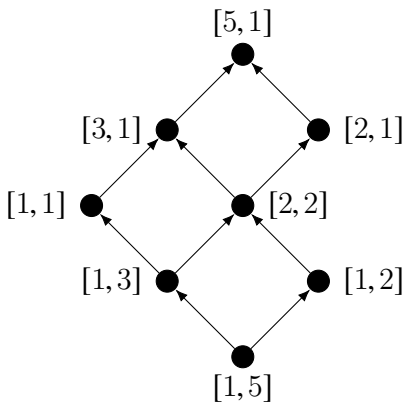

Abbildung 2.3: $\mathcal{G}$ Die Beschriftung eines Knoten D steht für $[d(D), u(D)]$ [Pec11, Abb. 7]

Der für die Berechnung deutlich wichtigere Teil ist allerdings die Tabelle t, die mit der folgenden Formel berechnet werden kann:

$$t[j,k] = \sum_{(V,W)} d(V)u(W)$$

Die Summe wird über alle Kanten (V, W) genommen, wobei $W = V \cup \{u_j\}$ und $u_k \notin W$.

Beweis. Im Folgenden bezeichnet ein Kind einen direkten Vorgängerknoten, Eltern einen direkten Nachfolgerknoten. Sei $P = (U, R)$ eine beliebige partielle Ordnung und W, V Downsets mit $W = V \cup \{u_j\}$. Die Anzahl der linearen Erweiterungen ist wie bereits erwähnt gleich der Anzahl der Varianten eine gegebene partielle Ordnung topologisch zu sortieren. Als Ergebnis soll $e(P + u_j u_k)$ stehen. Es sind also die verschiedenen topologischen Sortierungen gesucht, in denen u_k und seine Nachfolger hinter u_j stehen. Wurden alle Varianten identifiziert, haben wir das gesuchte Ergebnis. Betrachtet wird zunächst eine topologische Sortierung, bei der u_j an einer fixen Stelle steht. Es sind ein paar Dinge festzuhalten: $\overline{W} = \overline{V} \setminus \{u_j\}$; $u_j \notin V$; $u_k \in \overline{W}$ und $u(W) = d(\overline{W})$. Da W Downset ist, entspricht $\overline{W}$ einem Upset. Wegen $V \cup \{u_j\} = W$ also $V \subset W$ sind V und $\overline{W}$ disjunkt. Da V Downset und $\overline{W}$ Upset ist, gilt deshalb für alle Relationen zwischen den Mengen: $\forall u_x \in \overline{W}, u_y \in V : (u_y, u_x) \in R \vee (u_x, u_y) \notin R$. Die Elemente aus $\overline{W}$ sind sozusagen größer gleich derer aus V. Aus diesem Grund lassen sich die Mengen getrennt voneinander topologisch sortieren, lediglich alle

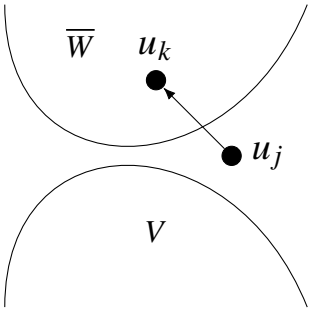

Abbildung 2.4: Visualisierung der Mengen V und $\overline{W}$.

Elemente aus V müssen in der topologischen Sortierunge vor denen aus $\overline{W}$ stehen. Da $W = V \cup \{u_j\}$ und W Downset ist, lässt sich schließen, dass u_j in V keine Eltern hat. Da $\overline{W}$ Upset ist, hat u_j in $\overline{W}$ keine Kinder. u_j befindet sich also „zwischen“ den Mengen (s. 2.4). Durch den Vergleich wird u_j zu u_k in Relation gesetzt: $(u_j, u_k) \in R$, wobei $u_k \in \overline{W}$. Da u_j durch die Konstruktion der Mengen zwischen diesen fixiert ist, ergibt sich durch die Relation eine Weise topologisch zu sortieren. Genauer, in der topologischen Sortierung stehen zuerst alle Elemente aus V, an nächster Stelle u_j und anschließend alle Elemente aus $\overline{W}$. V kann auf $d(V)$ Varianten topologisch sortiert werden und $\overline{W}$ auf $d(\overline{W}) = u(W)$ verschiedene Varianten. Die Komplette Menge U kann deshalb in dieser Reihenfolge $V \rightarrow u_j \rightarrow \overline{W}$ auf $d(V) \cdot u(W)$ verschiedene Arten topologisch sortiert werden. Damit sind aber noch nicht alle möglichen topologischen Sortierungen abgedeckt, die die partielle Ordnung $P + u_j u_k$ haben kann, denn bisher war u_j an eine Stelle gebunden. Außerdem sind alle Elemente aus V vor u_j eingereiht, dies muss aber nicht sein. Beispielsweise kann ein Element, welches keine Eltern oder Kinder hat, sowohl in V als auch in $\overline{W}$ aufgenommen werden, ohne die Down- bzw. Upset Eigenschaft zu verletzen. Aus diesem Grund werden alle Aufteilungen betrachtet, sodass V mit und ohne u_j ein Downset bildet und über diese summiert. Die Größe von V entscheidet also darüber, an welcher Stelle u_j in der topologischen Sortierung steht. Damit werden alle topologischen Sortierungen erreicht und man erhält $t[j, k]$.

□

Die Berechnung der Tabelle erfolgt mit der zweiten Tiefen-/Breitensuche. Dabei wird für einen Knoten D aus $\mathcal{G}$ überprüft, welche Elemente aus U nicht in D sind (dies sind die Elemente u_k) und welches Element durch die ausgehenden Kanten hinzukommt (Element u_j). Das Produkt von $d(V)$ und $u(W)$ wird auf die entsprechenden Stellen der Tabelle addiert. Genaueres findet sich in Abschnitt 3.1. In Tabelle 2.1 finden sich die Werte $t[j, k]$ die zur Beispiel-Ordnung 2.1 gehören.

j \ k	0	1	2	3
0	-	2	5	4
1	3	-	5	5
2	0	0	-	2
3	1	0	3	-

Tabelle 2.1: Tabelle $t[j, k]$ zur Ordnung 2.1 [Pec11, Tab. 2]

3 Implementierung in der Praxis

In diesem Kapitel werden die praktischen Aspekte des Algorithmus näher behandelt. Hier werden ebenso Optimierungen für die einzelnen Teile des Algorithmus erörtert.

3.1 Grundlagen

Der Algorithmus wurde zuerst in Python und anschließend in C++ implementiert. Grund hierfür ist, dass der Code in Python aufgrund der Simplizität der Sprache schneller geschrieben werden konnte. Es konnten deshalb mehrere Ansätze in kürzerer Zeit getestet werden. Ein weiterer Vorteil von Python ist, dass mit dem *networkx* Package [HSS08] bereits eine sehr brauchbare Bibliothek für viele Algorithmen vorhanden war. Unter anderem auch ein Algorithmus, der alle topologischen Sortierungen eines Graphen berechnet, wodurch die Ergebnisse für lineare Erweiterungen verifiziert werden konnten. Zwar muss eine langsamere Implementierung in Python nicht bedeuten, dass die gleiche Implementierung in C++ ebenfalls schlechter läuft, dennoch ist grob zu erkennen, welche Algorithmen sinnvoll sind und welche keinen Zeitgewinn erbringen. Der große Vorteil von C++ ist die Geschwindigkeit, mit der der Algorithmus am Ende läuft. Spätestens mit Optimierungsoptionen des C++-Compilers übertrifft die C++ Implementierung Python um Längen. Für C++ wurde die *Boost Graph Library* [SU00] benutzt. Beide Implementierungen sind im Repository unter [Obs19]. Die Python Implementierung ist allerdings nicht ausgefeilt, da diese mehr zum Testen neuer Algorithmen benutzt wurde.

Eine zentrale Frage war es, wie die partiellen Ordnungen abgespeichert werden. Getestet wurde, den Graph als Adjazenzmatrix und als Adjazenzliste abzuspeichern. Es wurde mehr Wert auf Geschwindigkeit gelegt, die die Datenstruktur bietet, als auf geringen Speicherbedarf. Die Vermutung war deshalb, dass die Adjazenzmatrix die bessere Form ist, da Operationen wie Kanten einfügen und löschen besonders schnell funktionieren. Eine mehrfach gebrauchte Funktionalität ist allerdings das Bestimmen aller eingehenden und ausgehenden Knoten. Für diese Operation bietet sich die Adjazenzliste eher an als die Adjazenzmatrix. Diese Funktionalität ist vermutlich auch der Grund, warum die Implementierung mit Adjazenzliste schneller war als die mit Adjazenzmatrix. Die Boost Graph Libary hat hierfür die passende Datenstruktur: eine bidirektionale `adjacency_list`. Der Vorteil bei dieser Datenstruktur ist, dass sowohl ausgehende als auch eingehende Knoten abgespeichert werden. Der Aufruf eines Iterators über diese ist deshalb in beiden Fällen in konstanter Zeit möglich.

3.1.1 Die Klasse PosetObj

In der C++ Implementierung befindet sich die Klasse `PosetObj`. Sie ist die zentrale Struktur zur Verwaltung der partiellen Ordnungen. Motivation für eine solche Struktur ist, dass auf bereits berechnete Eigenschaften wieder zurückgegriffen werden kann. Die wichtigsten Daten, die ein

`PosetObj` neben seiner graphischen Repräsentaion speichert, sind vor allem die Anzahl der linearen Erweiterungen und die Tabelle t. Daneben werden auch der duale Graph, zuletzt hinzugefügte Kante und die IDs der normalen und dualen Ordnung gespeichert (siehe Abschnitt 3.2.3). Weitere nützliche Eigenschaften des `PosetObj` sind die Anzahl der Singletons und eine obere und untere Schranke für die Anzahl der nötigten Vergleiche, um die Ordnung zu sortieren.

3.2 Algorithmen

3.2.1 Lineare Erweiterungen

Ein wichtiger Teil des Algorithmus ist die effiziente Implementierung zum Zählen der linearen Erweiterungen. Der Überblick über den Algorithmus findet sich in Abschnitt 2.3, hier folgen deshalb einige nützliche Details. Für die Berechnung wird der Graph $\mathcal{G}$ nicht explizit erstellt, sondern es wird eine Tabelle (implementiert mit einem Array) der Größe 2^n erzeugt [Pec11]. Der Index der Tabelle steht dabei für ein Downset. Genauer ist der Index die charakteristische Funktion des Downsets D. Interpretiert man den Index als n-Bitstring, so ist Bit j auf 1 gesetzt, genau dann wenn $u_j \in D$. Für C++ bietet die Boost Library dafür die Klasse `dynamic_bitset`, welche alle nötigen Funktionen bereitstellt. Die Tabelle enthält für jedes Downset D drei Werte: $[d(D), u(D), v(D)]$. Dabei sind $d(D)$, $u(D)$ wie in Abschnitt 2.3 beschrieben. Der dritte Wert $v(D)$ ist ein Zeitstempel, der bei der Traversierung des Graphen $\mathcal{G}$ hilft. Peczarski [Pec11] traversiert $\mathcal{G}$ mit einer Tiefensuche. Die Implentierung hier [Obs19] berechnet die Werte allerdings mit einer Breitensuche auf dem Graphen. Die Breitensuche wirkt plausibler, da so die Werte Ebene für Ebene berechnet werden und ein Knoten so nicht auf die Berechnung anderer Knoten warten muss. Die Tabelle wird an Index 0 mit $[1, 0, 0]$ initialisiert, an Index 2^n mit $[0, 1, 0]$. Die restlichen Einträge sind alle $[0, 0, 0]$, für den globalen Zeitstempel v_t gilt $v_t = 0$.

Generell wird die Tabelle in zwei Breitensuchen gefüllt. Zu Beginn jeder Suche wird der aktuelle Zeitstempel v_t inkrementiert. Die erste der beiden berechnet die Werte $d(D)$. Der Start für diese Suche ist der Knoten, der die leere Menge repäsentiert (Index 0). Dieser wird in eine Queue geschoben. Weiter nimmt der Algorithmus immer einen Knoten D aus der Queue und führt die folgende Berechnung durch. Dies geschieht so lange, wie Knoten in der Queue sind. Von einem Knoten D aus werden alle nachfolgenden Knoten D' identifiziert. Zu den Tabelleneinträgen $d(D')$ wird der Wert $d(D)$ addiert. Gleichzeitig wird überprüft, ob es sich um einen neuen Knoten handelt. Dies ist der Fall, wenn $v(D') < v_t$ ist. Dann wird der Zeitstempel $v(D')$ auf v_t gesetzt und D' (also dessen Index) zur Queue hinzugefügt. Für den Fall, dass $v(D') = v_t$ gilt, wurde D' bereits zur Queue hinzugefügt. Die nachfolgenden Knoten haben immer genau ein Element u_i mehr als D. Ein Knoten D' ist deshalb Nachfolger von D, wenn $D' = D \cup \{u_i\}$ und D' Downset ist. Ob D' Downset ist, wird überprüft, indem für alle direkten Vorgänger von u_i kontrolliert wird, ob diese in D sind. Da D bereits Downset ist, genügt es die direkten Vorgänger zu betrachten.

Die zweite Suche startet mit dem Knoten, der U repräsentiert (Index 2^n). Hier werden die Werte $u(D)$ analog zur ersten Suche berechnet. Allerdings werden hier die Vorgänger $\hat{D}$ eines Knotens D identifiziert. Die vorangegangenen Knoten haben immer genau ein Element u_j weniger als D. Ein Knoten $\hat{D}$ ist deshalb Vorgänger von D wenn $\hat{D} = D \setminus \{u_j\}$ und $\hat{D}$ Downset ist. Da D Downset ist, sind definitiv alle Vorgänger von u_j in $\hat{D}$, die Downset-Eigenschaft kann lediglich durch Nachfolger von u_j verletzt werden. Es wird deshalb für alle direkten Nachfolger von u_j geprüft, ob diese nicht in

D sind. Auch hier genügen wieder nur die direkten Nachfolger. In dieser Suche wird auch gleichzeitig die Tabelle $t[j,k]$ berechnet. Da aus dem aktuellen Knoten D ein u_j entfernt wird, ist D der Knoten W aus Abschnitt 2.3. Vorgängerknoten $\hat{D}$ ist der Knoten V. Für alle u_k, die nicht in $D = W$ sind, wird draufaddiert: $t[j,k] = t[j,k] + d(\hat{D}) \cdot u(D)$.

Soweit der Algorithmus. Es gibt einige Stellen, an denen Optimierungen vorgenommen werden können. In der Berechnung von $t[j,k]$ reicht es nur das obere oder untere Dreieck auszufüllen. Für die Einträge gilt $e(P + u_j u_k) = t[j,k]$ also auch $e(P + u_k u_j) = t[k,j]$. In Abschnitt 2.1.2 wurde bereits erwähnt, dass $e(P) = e(P + u_j u_k) + e(P + u_k u_j)$ gilt. Für die Tabelle t bedeutet das, $t[k,j] = e(P) - t[j,k]$. Die Berechnung eines Dreiecks kann also wegfallen, was die Berechnung deutlich beschleunigt.

Eine weitere Verbesserung wird offensichtlich, wenn man die Struktur der partiellen Ordnung sich genauer anschaut. Aufgrund der exponentiell vielen Downsets kann die Berechnung trotzdem lange dauern. Anfangs ist das der Fall mit der Ordnung P_0, die bis auf die trivialen Relationen keine weiteren hat. Wegen den 2^n Downsets ist die Berechnung recht zeitintensiv. Das muss allerdings nicht sein, denn anfangs ist es bspw. egal, welchen Vergleich wir machen, die nächste partielle Ordnung hat immer die gleiche Struktur: Eine Kante zwischen zwei Elementen und $n - 2$ Elemente ohne eingehende und ausgehende Kanten. Im Weiteren ist mit *Singleton* ein Element ohne eingehende und ausgehende Kanten gemeint. Diese Singletons lassen sich nutzen. Es ist recht offensichtlich, dass egal mit welchem Singleton ein Vergleich angestellt wird, die Struktur der entstandenen Ordnung ist in allen Fällen die gleiche. Folglich sind auch die Anzahl der linearen Erweiterungen und der Eintrag in Tabelle t gleich. Diese Werte lassen sich also einfach kopieren. Es liegt deshalb nahe, die meisten Singletons aus der Ordnung zu entfernen und den Algorithmus auf einer Reduzierten Ordnung durchzuführen. Tatsächlich können wir für die Berechnung alle bis auf ein Singleton entfernen. Für die Berechnung der korrekten Werte helfen die folgenden Lemmas.

Lemma 2. *Für eine partielle Ordnung P der Größe n, die sich in k Singletons und einen Rest $\tilde{P}$ aufteilen lässt, gilt $e(P) = e(\tilde{P}) \cdot \binom{n}{k} \cdot k! = e(\tilde{P}) \cdot \frac{n!}{(n-k)!}$.*

Beweis. Um zu sehen, wie die Anzahl der Singletons $e(P)$ beeinflusst, sei P eine Ordnung mit n Elementen und davon k Singletons. Für eine lineare Erweiterung weisen wir den Elementen nach Definition 2 Zahlen zu. Sei $\tilde{P}$ der Teil von P ohne Singletons. Die Anzahl der linearen Erweiterungen für $\tilde{P}$ ist $e(\tilde{P})$. Um also P die Zahlen zuzuweisen, werden zuerst k verschiedene Zahlen aus $\{1,\ldots,n\}$ ausgewählt und dann auf die Singletons verteilt. Offensichtlich gibt es für die Singletons $k!$ verschiedene konsistente Permutationen, denn jede Permutation ist konsistent. Die restlichen $n-k$ Zahlen können dann auf $e(\tilde{P})$ Arten Verteilt werden. Da die Verteilung unabhängig voneinander geschehen kann, ergeben sich insgesamt $e(\tilde{P}) \cdot \binom{n}{k} \cdot k! = e(\tilde{P}) \cdot \frac{n!}{(n-k)!}$ lineare Erweiterungen. □

Lemma 3. *Der Vergleich von zwei Singletons halbiert die Anzahl der linearen Erweiterungen.*

Beweis. Sei eine P partielle Ordnung mit mindestens zwei Singletons u_j, u_k. Wir betrachten die Menge der linearen Erweiterungen von P. Eine lineare Erweiterung kann nach Definiton 2 als eine Permutation der Zahlen $\{1,\ldots,n\}$ gesehen werden. Den Elementen u werden für eine lineare Erweiterung von P Zahlen zugewiesen, sodass $u_i < u_l$ falls $(u_i, u_l) \in R$. Den Singletons u_j, u_k kann deshalb jede der Zahlen zugewiesen werden. Nun werden die Singletons miteinander verglichen, sei diese Ordnung P_1. O.B.d.A. sei das Ergebnis des Vergleichs $u_j < u_k$. Nun ist es nicht mehr möglich,

den ehemaligen Singletons beliebige Zahlen aus $\{1,\ldots,n\}$ zuzuweisen. Man betrachte deshalb eine lineare Erweiterung L von P. Da alle Zahlen verschieden, sind gilt entweder $L(u_j) < L(u_k)$ oder $L(u_j) > L(u_k)$, wobei $L(u_j)$ die Zahl ist, die u_j zugewiesen wird. Sei nun L' eine lineare Erweiterung von P, sodass gilt $\forall i \in \{0,\ldots,n-1\}, i \neq j,\ i \neq k :\ L'(u_i) = L(u_i)$ und $L'(u_j) = L(u_k)$, $L'(u_k) = L(u_j)$. L' ist also gleich L, lediglich die Zahlen, die u_j und u_k zugewiesen werden, sind vertauscht. L' ist ebenfalls lineare Erweiterung von P, da u_j, u_k jede Zahl zugeweisen werden kann. Allerdings ist entweder L oder L' lineare Erweiterung von P_1, nämlich die, für die gilt $l(u_j) < l(u_k)$, aber nicht beide. Da es für jede lineare Erweiterung L von P genau eine der Form L' gibt und nur eine der beiden konsistent mit P_1 ist, muss die Zahl der linearen Erweiterungen von P_1 halb so groß sein wie die von $P \Rightarrow e(P_1) = \frac{e(P)}{2}$.

Ein anderer, etwas eleganterer Beweis des Lemmas: Es ist leicht zu sehen, das die Partiellen Ordnungen P_1, P_2 isomorph sind. Isomorphismus $f : \forall i \in \{0,\ldots,n-1\}, i \neq j, i \neq k : f(u_i) \mapsto u_i$ und $f(u_j) = u_k$, $f(u_k) = u_j$. Die Anzahl der linearen Erweiterungen kongruenter Ordnungen ist gleich: Lineare Erweiterung L_1 ist Erweiterung von P_1, genau dann wenn L_2 lineare Erweiterung von P_2 ist, mit $\forall i \in \{0,\ldots,n-1\} : L_1(u_i) = L_2(f(u_i))$. Für P_1 isomorph zu P_2^* ist eine Permutation konsistent mit P_1, genau dann wenn die umgekehrte Permutation konsistent mit P_2^* ist. Damit gilt $e(P_1) = e(P_2)$ und wegen Abschnitt 2.1.2 gilt: $e(P_1) + e(P_2) = e(P) \Rightarrow e(P_1) = e(P_2) = \frac{e(P)}{2}$.

□

Mit diesem Wissen kann die Tabelle t mit einer Partiellen Ordnung berechnet werden, aus der alle bis auf ein Singleton entfernt wurden. Ein Singleton bleibt in der Berechnung, da der Wert nach einem Vergleich von Singleton mit Nicht-Singleton nicht so trivial ist. In der Tabelle t hat dieser den Index $n-k$. Nach dem Entfernen der $k-1$ Singletons wird die Tabelle $\tilde{t}$ und der Wert $e(\tilde{P})$ normal berechnet. Die Erweiterung von $e(\tilde{P})$ funktioniert nach Lemma 2. Weil $k-1$ Singletons entfernt wurden, gilt $e(P) = e(\tilde{P}) \cdot \frac{n!}{(n-k+1)!}$. Jeder Eintrag der Tabelle $\tilde{t}$ wird zuerst mit $\frac{n!}{(n-k+1)!}$ multipliziert: $\forall i,j \in \{0,\ldots,n-1\} : \tilde{t}[i,j] \leftarrow \tilde{t}[i,j] \cdot \frac{n!}{(n-k+1)!}$. Beachte, für $i = j$ ist $\tilde{t}[i,j] = 0$. Weiter sind für Tabelle t folgende 4 Fälle zu beachten. Es ist davon auszugehen, dass sich alle Singletons in den hinteren Indizes befinden. Das kann erreicht werden, indem ein Vergleich immer nur mit dem ersten Singleton gemacht wird (s. Abschnitt 3.2.2). Zur Erinnerung P hat n Elemente und k Singletons, ein Eintrag $t[i,j]$ wird abhängig davon unterschiedlich berechnet.

1. Fall: $i \leq n-k$, $j \leq n-k$: Der Eintrag aus $\tilde{t}[i,j]$ wird kopiert, $t[i,j] = \tilde{t}[i,j]$
2. Fall: $i < n-k$, $j > n-k$: Der Wert aus Zeile i des Singleton wird kopiert, $t[i,j] = \tilde{t}[i,n-k]$
3. Fall: $i > n-k$, $j < n-k$: Der Wert aus Spalte j des Singleton wird kopiert, $t[i,j] = \tilde{t}[n-k,j]$
4. Fall: $i \geq n-k$, $j \geq n-k$: Vergleich von zwei Singletons, nach Lemma 3 gilt $t[i,j] = \frac{e(P)}{2}$, Sonderfall $i = j$: $t[i,j] = 0$

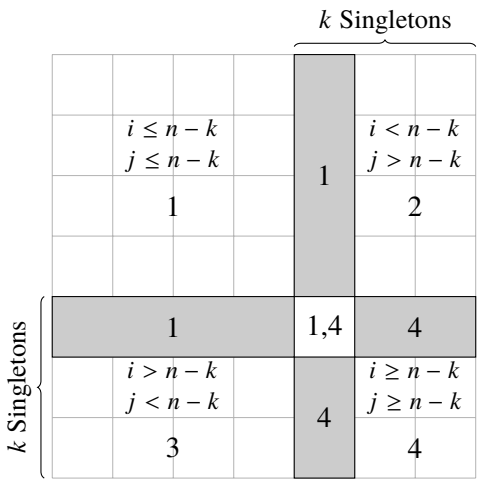

Abbildung 3.1: Visuelle Darstellung der 4 Fälle in Tabelle t

Diese Verbesserung sorgt dafür, dass die Berechnung, vor allem zu Beginn der Forwardsteps, maßgeblich beschleunigt wird. Wie bereits erwähnt, liegt das an dem deutlich reduzierten Graph $\mathcal{G}$.

3.2.2 Forwardsteps

Analog zu den Verbesserungen bei Berechnung der linearen Erweiterungen kann man sich einige Vergleiche in den Forwardsteps sparen. Hier wird ebenfalls ausgenutzt, dass Vergleiche mit Singletons isomorphe Ordnungen zur Folge haben. Nicht-Singletons werden deshalb immer nur mit dem Ersten Singleton verglichen, also mit dem Singleton u_i, für den i minimal ist. Der Singleton Vergleich erfolgt dann zwischen u_i und u_{i+1}. Das ist auch wichtig, da sonst die Optimierung beim Zählen der linearen Erweiterungen nicht funktioniert (Singletons müssen immer in den hinteren Indizes zu finden sein). Diese Heuristik findet sich auch in [Pec02].

3.2.3 Kongruenz-Check

Um die Anzahl der partiellen Ordnung gering zu halten, ist der Kongruenz-Check unerlässlich. Die Kongruenzprüfung wird mit zwei Isomorphie-Checks umgesetzt, einmal mit der Graphrepräsentation von P und der Graphrepräsentation von P^*. Für den Isomorphie-Test ist in der Boost Graph Libary eine Implementierung des VF2-Algorithmus vorhanden. In Schritt c der Forwardsteps soll nun eine Ordnung der Menge $\mathcal{S}_c$ hinzugefügt werden. Es wäre sehr ineffizient, alle Ordnungen, die sich bereits in $\mathcal{S}_c$ befinden, auf Kongruenz zu prüfen. Es wird deshalb eine Vorauswahl möglicher Kandidaten getroffen. Es ist leicht zu sehen, dass Kongruente Ordnungen dieselbe Anzahl an linearen Erweiterungen besitzen. Für eine lineare Erweiterung L wird bei isomorphen Ordnungen mit Isomorphismus f der Wert $L(u_k)$ dem Knoten $f(u_k)$ zugewiesen. Bei dualen Ordnungen ist L eine lineare Erweiterung von der einen Ordnung genau dann, wenn die invertierte Erweiterung L^* eine lineare Erweiterung von der dualen Ordnung ist.

Ist die Anzahl verschieden, lässt sich die Kongruenz ausschließen. Zusätzlich wird die Eingangs- und Ausgangsgradsequenz der Graphen berechnet, um weitere Isomorphien-Kandidaten auszuschließen. Dieses Prinzip ist in einer Hashmap realisiert. Diese enthält Listen von Ordnungen, die diese Eigenschaften teilen. Der Key für eine solche Liste ist ein konkatenierter String von Ausgangsgradsequenz, Eingangsgradsequenz und lineare Erweiterungen in dieser Reihenfolge. Die Gradsequenz ist leicht zu berechnen und wurde deshalb in der ID verwendet. Da die Anzahl der linearen Erweiterungen zu dieser Zeit bereits berechnet ist und in Lemma 3 u.a. gezeigt wurde, dass diese Anzahl für isomorphe partielle Ordnungen gleich ist, wird sie auch in die ID mit aufgenommen.

Neben diesen Graphinvarianten wurde auch noch eine weitere getestet, die Graphfärbung nach Weisfeiler-Lehman. In Python wurde dafür eine Implementierung des *Color refinement* Algorithmus [GKMS17], auch bekannt als *1-dimensionaler Weisfeiler-Lehman*, getestet. Der Algorithmus aus [GKMS17] wurde dabei auf gerichtete Graphen erweitert, indem die Farben von eingehenden und ausgehenden Knoten separat betrachtet wurden. Pseudocode für den ungerichteten Fall findet sich in [GKMS17, Abb. 2]. Der Algorithmus hatte definitv den gewünschten Effekt, die Anzahl der Kandidaten für den Kongruenz-Check zu verringern. Als Beispiel für $n = 9$ Elemente wurde die maximale Anzahl der Ordnungen mit gleicher ID von 8 auf 3 reduziert. Die durchschnittliche Anzahl der Kandidaten für einen Isomorphie-Check ist ebenfalls von maximal 1.7 auf maximal 1.2 gesunken. Generell muss mit *Color refinement* meist nur ein Kandidat auf Isomorphie geprüft werden. Insgesamt wurden die Methode `is_isomorphic` nur 4086-mal statt 8573-mal aufgerufen. Allerdings betrug die Laufzeit ohne *Color refinement* 10 Sekunden, mit hingegen 19. Die Implementierung war im Detail vermutlich zu ineffizient. Ebenso wurde die Färbung nicht genutzt, um einen Isomorphismus daraus abzuleiten. Es wäre möglich, gleichfarbige Knoten zweier Graphen aufeinander zu mappen und dann zu überprüfen, ob es sich um einen Isomorphismus handelt. Hier gibt es definitiv noch Potenzial, die Geschwindigkeit des *Color refinement* Algorithmus zu erhöhen und damit die des gesamten Programms.

Eine weitere Idee Isomorphie-Checks einzusparen ist zu prüfen, ob die partielle Ordnung selbst dual ist. Damit würde die Überprüfung der dualen Ordnung wegfallen, was besonders dann nützlich ist, wenn es viele Isomorphie-Kandidaten gibt. Es wird zuerst der Isomorphie-Check mit der normalen Ordnung durchgeführt. Wurde keine isomorphe Ordnung gefunden, dann werden die IDs betrachtet. Falls die ID und die ID der dualen Ordnung gleich sind, wird ein Isomorphie-Check zwischen P und P^* gemacht. Ist dieser Test positiv, kann abgebrochen werden, denn es werden die gleichen Kandidaten nochmals überprüft. Diese Heuristik hat sich als sehr schlecht erwiesen, da insgesamt mehr Isomorphie-Checks gemacht werden. Das Problem ist, dass dieser Fall recht selten vorkommt. Getestet wurde für $n = \{1, \ldots, 10\}$: Von den insgesamt ca. 16 000 Aufrufen, die gemacht wurden, um $\mathcal{S}_c$ auf kongruente Ordnungen zu untersuchen, waren die IDs 1005-mal gleich. Selbstdualität kam aber nur in 65 Fällen vor. Ohne den Selbstdualitäts-Check wurde `is_isomorphic` 13597-mal aufgerufen, mit hingegen 14518-mal. Die 65 Fälle sparen also 84 Isomorphie-Checks, die Bilanz ist damit 921 zusätzliche Isomorphie-Checks.

Zusammenfassend lässt sich sagen, dass trotz der Ansätze noch Verbesserungspotenzial besteht. Die Kongruenz-Checks haben in dieser Implementierung einen signifikanten Anteil an der Berechnungszeit, Optimierungen hier könnten also noch deutliche Auswirkungen haben.

3.2.4 Transitive Reduktion

Notation. Im Folgenden bezeichnet $G = (V, E)$ einen Graphen mit Knotenmenge V und Kantenmenge E. Die Operation $(G - \{\alpha\})$, $\alpha \in E$ bezeichnet den fast gleichen Graphen G mit Kantenmenge $E' = E \setminus \{\alpha\}$. Genauso ist $(G_1 \cap G_2)$ der Graph G' mit Knotenmenge $(V_1 \cap V_2)$ und Kanten $(E_1 \cap E_2)$. Da im Folgenden die Knotenmenge der Graphen gleich ist, gilt $(V_1 \cap V_2) = V_1 = V_2$, weshalb es genügt den Schnitt der Kantenmenge zu betrachten. Weiter steht G^T für die transitive Hülle und G^R für die transitive Reduktion des Graphen G.

Satz 1. *Die Transitive Reduktion ist für DAGs (directed acyclic graphs) eindeutig.*

Beweis. Der Beweis stammt aus [AGU72]. Die Eindeutigkeit der transitiven Reduktion von DAGs folgt aus folgenden zwei Lemmas.

Lemma 4. *Seien G_1, G_2 DAGs mit der gleichen Knotenmenge V und es gilt $G_1^T = G_2^T$. Falls eine Kante α existiert, mit $\alpha \in G_1$ und $\alpha \notin G_2$, dann gilt: $(G_1 - \{\alpha\})^T = G_1^T = G_2^T$.*

Beweis. Sei $\alpha = (u, v)$ wie oben beschrieben. Da $G_1^T = G_2^T$ gilt, haben beide Graphen die gleichen Erreichbarkeits-Eigenschaften, d.h. es existiert ein Pfad von a nach b in G_1 genau dann, wenn ein Pfad von a nach b in G_2 existiert. Deswegen hat G_2 einen Pfad $u \to v$ über einen anderen Knoten. Sei dieser Knoten w. Wegen der Erreichbarkeits-Eigenschaft muss G_1 auch einen Pfad $u \to w$ und $w \to v$ haben. Falls in G_1 der Pfad $u \to w$ über die Kante α verläuft, hat G_1 einen Pfad $u \to v \to w$. Da aber auch ein Pfad $w \to v$ in G_1 existiert, bildet $v \to w \to v$ einen Zyklus, was im Widerspruch zur Eigenschaft eines DAGs steht. Falls in G_1 der Pfad $w \to v$ über die Kante α verläuft, hat G_1 einen Pfad $w \to u \to v$. Da aber auch ein Pfad $u \to w$ in G_1 existiert, bildet $w \to u \to w$ einen Zyklus, was ebenfalls Widerspruch zur Eigenschaft eines DAGs ist. G_1 enthält deshalb einen Pfad $u \to w$ und $w \to v$, welcher nicht über α verläuft. Es folgt: $G_1 - \{\alpha\}$ besitzt einen Pfad $u \to v$, $(G_1 - \{\alpha\})^T = G_1^T = G_2^T$.

Als nächstes betrachten wir die Menge $S(G) = \{G_1 \mid G_1^T = G^T\}$, also die Graphen, die die selben Erreichbarkeits-Eigenschaft haben.

Lemma 5. *$S(G)$ ist abgeschlossen unter Schnitt, für $G_1, G_2 \in S(G)$ gilt: $(G_1 \cap G_2)^T = G_1^T = G_2^T = G^T$. Der Schnitt zweier solcher Graphen ändert also nichts an den Erreichbarkeits-Eigenschaften.*

Beweis. Seien $\{\alpha_1, \ldots, \alpha_r\}$ die Kanten, die in G_1 aber nicht in G_2 sind: $G_1 - (G_1 \cap G_2) = \{\alpha_1, \ldots, \alpha_r\}$. Wiederholtes Anwenden von Lemma 4 ergibt:

$$\begin{aligned}
(G_1 - \{\alpha_1\})^T &= G^T \\
(G_1 - \{\alpha_1\} - \{\alpha_2\})^T &= G^T \\
&\vdots \\
(\underbrace{G_1 - \{\alpha_1\} - \cdots - \{\alpha_r\}}_{=(G_1 \cap G_2)})^T &= G^T
\end{aligned}$$

Die letzte Gleichung besagt, $(G_1 \cap G_2)^T = G_1^T = G^T$, also gilt $(G_1 \cap G_2) \in S(G)$.

Satz 1 folgt deshalb nach folgender Überlegung:

Beweis Satz 1. Sei $TR(G) = \{G' \mid G' = G^R\}$ die Menge der (verschiedenen) transitiven Reduktionen von G. Da die transitive Reduktion nach Definition die Erreichbarkeiten beibehält, gilt für alle $G_1, G_2 \in TR(G)$: $G_1^T = G_2^T = G^T$. Die Anzahl der Kanten von G_1, G_2 ist nach Definition minimal. Durch den Schnitt wird die Kantenanzahl kleiner oder bleibt gleich. Wegen Lemma 5 gilt aber $(G_1 \cap G_2) \in S(G)$, weshalb die Anzahl der Kanten von $(G_1 \cap G_2)$ gleich derer von G_1 und G_2 ist. $\Rightarrow G_1 = G_2$.

□

Wie in den Forwardsteps 2.2.1 bereits erwähnt, kann der Kongruenz-Check mit der transitiven Hülle oder der transitiven Reduktion gemacht werden. Die Aussage gilt wegen folgendem Satz:

Satz 2. *Die Transitiven Hüllen zweier partieller Ordnungen P_1, P_2 sind genau dann isomorph, wenn die transitiven Reduktionen ebenfalls isomorph sind.*

Beweis.

„$\Rightarrow$“ Seien die transitiven Hüllen der partiellen Ordnungen P_1, P_2 isomorph. In den Forwardsteps wird eine neue Kante immer nur zwischen Paaren hinzugefügt, die nicht in Relation stehen, also keinen gemeinsamen Pfad haben. Aus diesem Grund ist die partielle Ordnung graphisch betrachtet ein DAG. Wie in Satz 1 gezeigt, ist die transitive Reduktion für DAGs eindeutig. Für die transitive Reduktion von P_1, P_2 werden deshalb beide Male die gleichen Kanten entfernt. Die entstandenen Untergraphen sind isomorph.

„$\Leftarrow$“ Seien nun die transitiven Reduktionen von P_1, P_2 isomorph mit Isomorphismus $f : P_1 \to P_2$. Es gilt deshalb: $\exists$ Pfad $p = v_1 v_2 \dots v_l$ in $P_1 \Leftrightarrow \exists$ Pfad $f(p) = f(v_1) f(v_2) \dots f(v_l)$ in P_2. Die transitive Hülle von P_1, P_2 hat für jeden Pfad, der in v_1 startet und in v_l endet, die Kante (v_1, v_l) bzw. $(f(v_1), f(v_l))$. Die Knotenmenge bleibt für die Ordnungen unverändert, $f : R_1 \to R_2$, $f : (u, v) \mapsto (f(u), f(v))$ beschreibt deshalb für die Relationen einen Isomorphismus.

□

Für die Implementierung wurde die transitive Reduktion der partiellen Ordnungen genutzt. Die transitive Reduktion ist wohl die intuitivere Repräsentation einer partiellen Ordnung. Eine lineare Ordnung ist dabei durch einen Pfadgraph mit n Knoten und $n - 1$ Kanten dargestellt. Die Transitive Hülle ist etwas unübersichtlicher, da jedes Knotenpaar eine Relation beschreibt, sind es hier $\binom{n}{2}$ Kanten.

Algorithmus 3.1 Transitive Reduktion

```
1:  procedure TransitiveReduction(P, α = (u_out, u_in)) // α ist hinzugefügte Relation (u_out, u_in)
2:      U_succOfIn ← Successors(P, u_in)                // berechnet alle Elemente größer als u_in
3:      U_predOfOut ← Predecessors(P, u_out)            // berechnet alle Elemente kleiner als u_out
4:      for all u_k ∈ {u_k | (u_out, u_k) ∈ P} do       // u_k sind direkte Nachfolger von u_out
5:          if u_k ∈ U_succOfIn then
6:              P ← P \ {(u_out, u_k)}                  // entferne Kante (u_out, u_k)
7:          end if
8:      end for
9:      for all u_pred ∈ U_predOfOut do
10:         if (u_pred, u_in) ∈ P then
11:             P ← P \ {(u_pred, u_in)}                // entferne Kante (u_pred, u_in)
12:             continue
13:         end if
14:         for all u_succ ∈ U_succOfIn do
15:             if (u_pred, u_succ) ∈ P then
16:                 P ← P \ {(u_pred, u_succ)}          // entferne Kante (u_pred, u_succ)
17:             end if
18:         end for
19:     end for
20: end procedure
```

Da in den Forwardsteps nach jedem Hinzufügen einer Relation der Algorithmus 3.1 aufgerufen wird (Algorithmus 2.1 Zeile 12, 13), ist davon auszugehen, dass P vor Hinzufügen der Relation $\alpha = (u_{out}, u_{in})$ bereits reduziert war. Dies lässt sich nutzen, um die Reduktion zu beschleunigen. Es muss nämlich nicht der ganze Graph reduziert werden, sondern nur Teile, die durch Hinzufügen von α nun nicht mehr reduziert sind. Genauer betrachtet werden müssen nur der Knoten, von dem die Relation ausgeht u_{out} und dessen Vorgänger, sowie der Knoten u_{in}, in dem die Relation endet und dessen Nachfolger. Die Berechnung der Vorgänger und Nachfolger erfolgt in den Zeilen 2 und 3. Dies kann mit 2 Tiefensuchen erreicht werden. Als erstes werden die direkten Nachfolger von u_{out} betrachtet. Was also, wenn ein solcher direkter Nachfolger u_k in der Menge $U_{succOfIn}$ ist? Das bedeutet, wir haben einen Pfad $u_{out} \rightarrow u_{in} \rightarrow u_k$ und einen Pfad $u_{out} \rightarrow u_k$. Da der Knoten u_k über den Pfad $u_{out} \rightarrow u_{in} \rightarrow u_k$ erreichbar ist, stellt $u_{out} \rightarrow u_k$ eine überflüssige Relation dar. Folglich wird diese in Zeile 6 entfernt.

Die andere Möglichkeit für überflüssige Relationen besteht bei den Vorgängern von u_{out}. Durch die neue Relation α kann ein Vorgängerknoten u_{pred} jetzt einen Pfad zu u_{in} haben, der über u_{out} geht. Falls also u_{pred} davor eine Relation zu u_{in} hatte, ist diese, ähnlich zum vorherigen Fall, überflüssig und kann entfernt werden (Zeile 10, 11). Interessant ist dabei, dass nach diesem Fall keine weitere Relation der Form (u_{pred}, v) entfernt werden muss. Das liegt daran, dass die Ordnung vorher bereits reduziert war. Gäbe es also eine weitere Relation die von u_{pred} zu einem Nachfolger von u_{in} geht (sei dieser Knoten u_{succ}), dann hätte bereits die alte Ordnung die beiden Pfade $u_{pred} \rightarrow u_{in} \rightarrow u_{succ}$ und $u_{pred} \rightarrow u_{succ}$. P war vorher also nicht reduziert. Es ist zu beachten, dass alle neuen Pfade, die in u_{pred} starten, von der Form $u_{pred} \rightarrow u_{out} \rightarrow u_{in} \rightarrow u_{succ}$ sind. Demnach wurden alle möglichen überflüssigen Relationen (u_{pred}, v) überprüft. Falls es aber die Relation (u_{pred}, u_{in}) nicht

gibt, muss überprüft werden, ob nicht doch eine Relation zu einem Nachfolger von u_{in} besteht (Zeile 14). Diese müssen ebenfalls entfernt werden, da sie nun über (u_{out}, u_{in}) erreichbar sind. Wenn alle Vorgänger von u_{out} nach diesem Schema abgearbeitet wurden, ist die partielle Ordnung reduziert.

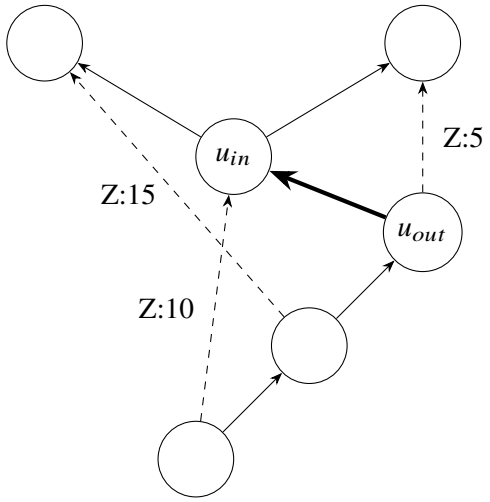

Abbildung 3.2: Veranschaulichung der Transitiven Reduktion.

In Abb. 3.2 findet sich eine Beispiel-Reduktion. Die neue Relation ist die dicke Kante zwischen u_{out} und u_{in}. Die gestrichelten Kanten werden durch Algorithmus 3.1 entfernt. Zusätzlich sind diese Kanten mit „Z:“ annotiert, was für die jeweilige Zeile des Algorithmus steht, wegen der diese Relation entfernt wird.

3.2.5 Backwardsteps

Die Backwardsteps haben von ihrer Struktur starke Ähnlichkeit mit den Forwardsteps. Aus diesem Grund bieten sich hier gleiche Optimierungen an. Das Auslassen der Singleton-Vergleiche funktioniert hier ganz analog. Ein Vorteil ist, dass gegenüber den Forwardsteps die Anzahl der linearen Erweiterungen nicht erneut berechnet werden muss. Diese sind nämlich im `PosetObj` gespeichert. Zur Erinnerung, es werden alle partiellen Ordnungen aus $\mathcal{S}_c$ betrachtet. In den Backwardsteps muss geprüft werden, ob Nachfolger von diesen partiellen Ordnungen in $\mathcal{S}^*_{c+1}$ enthalten sind. $\mathcal{S}^*_{c+1}$ ist hier wieder als Hashmap realisiert, die nach kongruenten Ordnungen durchsucht wird. Hier wird der gleiche Kongruenz-Check benutzt. Da in $\mathcal{S}_c$ alle Ordnungen nicht kongruent zueinander sind, kann der Kongruenz-Check beim Hinzufügen der Ordnung zu $\mathcal{S}^*_c$ wegfallen. Interessanter wird es, wenn für eine partielle Ordnung, die nicht in $\mathcal{S}^*_{c+1}$ ist, geprüft werden muss, ob sie in den verbliebenen Vergleichen sortierbar ist. Bevor die Funktion Sortable aufgerufen wird (2.3), wird zuerst geprüft, ob diese Ordnung bereits bekannt ist.

Das geschieht in den Methoden `isDefinitelySortable` und `isDefinitelyNotSortable`. Prinzipiell können diese Methoden auch Teil von Sortable sein, in dieser Implementierung ist dies nicht der Fall. Wie der Name schon verrät, berechnen die Methoden, ob eine gegebene partielle Ordnung definitiv oder definitiv nicht in den verbliebenen Vergleichen sortiert werden kann. Eine Ordnung ist dabei definitiv sortierbar, wenn sie für die gleiche oder geringe Anzahl an Vergleichen bereits untersucht wurde. Falls nun weniger Vergleiche als bei der vorigen Prüfung zur Verfügung stehen,

kann keine Aussage getroffen werden. Analog ist eine Ordnung definitiv nicht sortierbar, wenn sie für die gleiche oder höhere Anzahl an Vergleichen bereits untersucht wurde. Ebenso wird keine Aussage bei mehr Vergleichen gemacht. Die Methoden durchsuchen dafür zwei Arrays von Hashmaps auf Kongruente Ordnungen. `S_starMap` enthält dabei an Index c eine Hashmap jener Ordnungen, für die festgestellt wurde, dass sie in c Vergleichen sortierbar sind. `S_starMap` ist quasi eine Kopie von $\mathcal{S}^*$. Im Gegensatz zu $\mathcal{S}^*$ ist allerdings der Index invertiert. Elemente aus $\mathcal{S}^*_c$ sind in $C - c$ Vergleichen sortierbar. Ein weiterer Unterschied ist, dass `S_starMap` mehr Ordnungen enthält. Unter anderem kommt es vor, dass eine Ordnung aus $\mathcal{S}^*_c$ auch in $\mathcal{S}^*_{c+1}$ enthalten ist, denn offensichtlich ist eine Ordnung, die in c Vergleichen sortierbar ist, auch in $c + 1$ Vergleichen sortierbar. Analog verhält es sich für `S_compMap`. An Index c stehen Ordnungen, die nicht in c Vergleichen sortierbar sind.

Finden die Methoden kongruente Ordnungen an den entsprechenden Indizes, kann der Aufruf von Sortable wegfallen. Eine kleine Optimierung, die an dieser Stelle gemacht werden kann, sorgt dafür, dass nicht das komplette Array ab dem Index angeschaut werden muss. Für `S_starMap` müssen eine partielle Ordnung P z.B. nur die Indizes bis $\log_2 e(P)$ betrachtet werden. Ein optimales Sortierverfahren würde die Anzahl der linearen Erweiterungen genau halbieren, denn dann ist $e(P_1)$ bzw. $e(P_2)$ im Worst-Case minimal. Es werden deshalb mindestens so viele Vergleiche benötigt. Im Grunde ist dies ein ähnliches Argument wie in Abschnitt 2.1.1.

3.2.6 Sortable

Die Sortable-Funktion bietet auch Möglichkeiten der Optimierung. Eine besonders offensichtliche Verbesserung überprüft, ob die Anzahl der Paare, die nicht in Relation stehen, kleiner gleich der verbliebenen Vergleiche ist. In diesem Fall ist die Ordnung definitiv sortierbar, denn für jedes dieser Paare kann ein Vergleich benutzt werden, um diese zu ordnen. Danach stehen alle Paare in Relation, die partielle Ordnung wurde sortiert. Ebenso lassen sich die rekursiven Aufrufe der Forwardsteps optimieren. Hier kann genutzt werden, dass für einige Ordnungen bereits bekannt ist, dass sie in den verbliebenen Vergleichen nicht sortierbar sind. Falls eine Ordnung nämlich nicht in c Vergleichen sortierbar ist, muss sie in den Forwardsteps nicht in $\mathcal{S}_{C-c}$ aufgenommen werden. Sie wird schließlich so oder so in den Backwardsteps ausgeschlossen und wird nicht zu S^*_{C-c} hinzugefügt. Dafür wird erneut von der Methode `isDefinitelyNotSortable` Gebrauch gemacht, wenn eine neue partielle Ordnung erstellt wird.

Eine interessante Optimierung, die in [Pec04] vorgenommen wurde, geht auf den Satz von Kahn und Saks zurück [KS84]. Dieser besagt, dass für jede partielle Ordnung, die keine lineare Ordnung ist, ein Paar u_j, u_k existiert, sodass $\frac{3}{11} < \frac{e(P+u_ju_k)}{e(P)} < \frac{8}{11}$ gilt. Für die partiellen Ordnungen $P_1 = P + u_ju_k$, $P_2 = P + u_ku_j$ und einem solchen Paar u_ju_k bedeutet das $1 \leq \frac{e(P_1)}{e(P_2)} \leq \frac{8}{3}$. Die Idee besteht nun darin, diese Paare in einem Sortieralgorithmus zu nutzen, ohne die dazugehörigen partiellen Ordnungen explizit erstellen zu müssen. In den Backwardsteps 2.2 wird für eine Ordnung P_1, P_2 in Zeile 9 zuerst überprüft, ob die Anzahl der linearen Erweiterungen kleiner gleich der Grenze ist, die die Efficiency vorgibt. Wenn also Sortable(P_1, c) für eine Ordnung P_1 aufgerufen wird, kann davon ausgegangen werden, dass $e(P_1) \leq 2^c$ gilt. Für den Fall, dass $e(P) \leq 6$ ist, kann deshalb für c ein Mindestwert angenommen werden.

Lemma 6. *Falls $e(P) \leq min\{6, 2^c\}$ gilt, dann ist P in c Vergleichen sortierbar.*

Beweis. Sei also $e(P) \leq \min\{6, 2^c\}$. Falls $e(P) = 1$ ist, ist P lineare Ordnung und damit bereits sortiert. Falls $e(P) = 2$ ist, dann muss wegen Zeile 9 des Algorithmus 2.2 c mindestens den Wert 1 haben. Das bedeutet, es steht mindestens ein Vergleich zur Verfügung. Die Ordnung wird in P_1, P_2 zerlegt, sodass $e(P_1) = e(P_2) = 1$ und damit wieder eine lineare Ordnung vorliegt. Für $e(P) = 3$ muss nach der Vorbedingung $c \geq 2$ sein. Der erste Vergleich teilt P auf, sodass $e(P_1) = 2$ und $e(P_2) = 1$ ist. Der zweite Vegleich wird für P_2 nicht gebraucht und sortiert P_1 wie beschrieben. Für $e(P) = 4$ sind wieder mindestens $c \geq 2$ Vergleiche verfügbar. Hier wird der Satz von Kahn und Saks angewendet, um P mit einem Vergleich in P_1, P_2 aufzuteilen, sodass $e(P_1) = e(P_2) = 2$. Der verbliebene Vergleich kann im nächsten Schritt genutzt werden, um P_1 bzw. P_2 zu sortieren. An dieser Stelle wird der Satz benötigt, da für eine Aufteilung in $e(P_1) = 3, e(P_2) = 1$ mindestens 3 Vergleiche gebraucht werden. Das Verhältnis wäre hierbei $\frac{e(P_1)}{e(P_2)} = \frac{3}{1} > \frac{8}{3}$. Für $e(P) = 5$ ist $c \geq 3$. Eine Aufteilung, mit einem Vergleich nach Satz von Kahn und Saks, in $e(P_1) = 3, e(P_2) = 2$ benötigt nach vorangegangener Argumentation höchstens die 2 verbliebenen Vergleiche. Insgesamt genügen also 3 Vergleiche. Für $e(P) = 6$ ist ebenfalls $c \geq 3$. Nach Satz von Kahn und Saks ist eine Aufteilung in $e(P_1) = 4$, $e(P_2) = 2$ möglich, die insgesamt 3 Vergleiche reichen also auch hier wieder aus.

□

3.3 Parallelität

Nachdem eine Singlethread-Umsetzung implementiert wurde, war das nächste Ziel, den Algorithmus mittels Multithreading zu beschleunigen. Dafür wurde die in C++ standardmäßig vorhandene `thread` Klasse verwendet. Am einfachsten war es, die parallele Berechnung der linearen Erweiterungen umzusetzen. Also hauptsächlich die Berechnung der Tabelle t. Der Vorteil hierbei war, dass wenig Synchronisation der einzelnen Threads nötig war. Jeder Thread hatte dabei sein eigenes `PosetObj` bekommen, um auf diesem die Berechnung auszuführen. Im Gegensatz zur Singlethread-Implementierung werden zuerst die linearen Erweiterungen für alle partiellen Ordnungen berechnet. Erst wenn für jede Ordnung die Tabelle t vorliegt, werden die weiteren Schritte der Forwardsteps gemacht. Die nächste Stufe für mehr Parallelität ist die Multithread-Implementierung für die Schritte der Forwardsteps. Die Struktur für parallele Berechnung ist auf jeden Fall gegeben. Für die Parallelisierung ist zu beachten, dass keine Voraussetzungen verletzt werden, die andere Funktionen treffen. Beispielsweise sollte wie in Abschnitt 3.2.1 darauf geachtet werden, dass neue Relationen nur mit den ersten Singletons eingegangen werden. Es kann pro Thread ein `PosetObj` auf all seine Paare untersucht werden, die nicht in Relation zueinander stehen. Wenn viele neue Relationen zu einer schlechten Ordnung führen, die nicht zu $\mathcal{S}$ hinzugefügt wird, kann dieser Teil durch den Einsatz von Multithreading effektiv beschleunigt werden.

Problematisch wird es allerdings beim Kongruenz-Check. Hier müssen mehrere Threads gleichzeitig auf $\mathcal{S}_c$ zugreifen, um festzustellen, ob die neue Ordnung hinzugefügt werden muss oder bereits eine Ordnung in $\mathcal{S}_c$ ist, die kongruent zu einer der neuen Ordnungen P_1, P_2 ist. Die Container der C++ Standard Template Library erlauben gleichzeitiges Lesen, aber nicht gleichzeitiges Schreiben. Aus diesem Grund muss an dieser Stelle sichergestellt werden, dass nur ein Thread gleichzeitig Schreibzugriff hat. Zur Erinnerung: Die aktuelle Menge $\mathcal{S}_c$, die auf kongruente Ordnungen untersucht werden muss, ist als Hashmap implementiert. In dieser Hashmap befinden sich Listen, die

zu verschiedenen IDs gehören, um die Kandidaten einzugrenzen. Es ist also zulässig, gleichzeitig Kongruenz-Checks anzustellen, da dies nur Lesezugriffe sind, zur selben Zeit in diese Hashmap eine neue Ordnung einzufügen, jedoch nicht.

Um diese Funktionalität umzusetzen, wurde ein `shared_mutex` genutzt. Jeder Thread hat Zugriff auf den gleichen Mutex. Will ein Thread Kongruenz-Checks anstellen, ruft er auf den Mutex die Methode `lock_shared()` auf. Dies verhindert, dass ein anderer Thread zur selben Zeit schreibt. Wurden dabei keine kongruenten Ordnungen gefunden, wird exklusiver Zugriff mittels `lock()` angefordert. Davor muss allerdings der Lesezugriff freigegeben werden. Da zwei Threads zeitgleich eine kongruente Ordnung überprüfen, kann es passieren, dass beide Threads die Liste fertig gelesen haben und ihre Ordnung hinzufügen wollen. Aus diesem Grund wird, nachdem ein Thread exklusiven Zugriff erhalten hat, noch einmal überprüft, ob nicht doch eine kongruente Ordnung bereits in der Hashmap ist. Ein Nachteil dieser Implementierung ist, dass die Kongruenz-Checks zweimal ausgeführt werden. Eine mögliche Verbesserung hierfür ist, einen Versionszähler für die Hashmap einzuführen. Dieser wird immer inkrementiert, wenn ein Thread eine Ordnung hinzufügt. Der Ablauf eines Kongruenz-Checks für einen Thread sieht deshalb so aus:

1. Erwerbe Lesezugriff.
2. Speichere den aktuellen Versionszähler (VZ).
3. Führe den Kongruenz-Check durch.
4. a) Falls kongruente Ordnung gefunden wurde, überprüfe nächste Relation.
   b) Falls keine kongruente Ordnung gefunden wurde, erwerbe Schreibzugriff.
5. a) Falls gespeicherter VZ = aktuellen VZ, füge Ordnung hinzu.
   b) Ansonsten führe erneut Kongruenz-Check durch und füge entsprechend hinzu.

Zum jetzigen Zeitpunkt ist diese Verbesserung noch nicht implementiert.

Das parallele Prüfen der Ordnungen zeigt eine interessante Eigenschaft des Algorithmus. Je nachdem, welche Ordnungen in einem Schritt zuerst hinzugefügt werden, hat dies Einfluss auf die Anzahl der Ordnungen in $\mathcal{S}$ und damit $\mathcal{S}^*$. Als Beispiel hat für $n = 12$, $C = 29$ die Menge $\mathcal{S}_{22}$ in der Singlethread Implementierung 4 partielle Ordnungen. Bei Multithreading sind es allerdings häufig 6. Das folgende Beispiel erklärt diese Tatsache. Die partiellen Ordnungen P und P' haben die Nachfolger P_1, P_2 und P'_1, P'_2. Dabei seien P_2 und P'_1 kongruent. Die anderen Ordnungen haben diese Beziehung nicht. Wenn P_1 zuerst hinzugefügt wird, finden die Nachfolger von P' keine kongruente Ordnung, die bereits hinzugefügt wurde. Folglich wird auch P'_1 hinzugefügt. Für den Fall, dass zuerst P'_1 hinzugefügt wird, finden die Nachfolger von P mit P'_1 eine kongruente Ordnung zu P_2. Folglich wird P_1 nicht hinzugefügt.

In den Backwardsteps lassen sich ähnliche Ideen zur Parallelisierung umsetzen. Denn auch hier werden die Nachfolger einer Ordnung berechnet. Der Kongruenz-Check für $P_1 \in \mathcal{S}^*_{c+1}$ benötigt sogar weniger Synchronisation, da zu diesem Zeitpunkt nicht mehr in $\mathcal{S}^*_{c+1}$ geschrieben wird. Synchronisation ist dafür bei `S_starMap` und `S_compMap` nötig. Auch diese Verbesserung ist zum jetzigen Zeitpunkt noch nicht implementiert.

3.4 Ergebnisse

Für die bekannten Werte von $S(n)$ erhielten frühere Implementierungen gleiche Ergebnisse. Sprich, für $n = 2, \ldots, 11$ und $C = C(n)$ enthält die Menge $\mathcal{S}_0^*$ die Ordnung P_0. Bei $n = 12$ ist die Menge $\mathcal{S}_{29}$ leer. Für $n = 13$ ist die Menge $\mathcal{S}_0^*$ leer. In der aktuellen Version der Implementierung wird bei $n = 11$ ein falsches Ergebnis produziert. Wenn möglich, wird eine verbesserte Version unter [Obs19] verfügbar sein.

Die Berechnungen wurden auf einem Rechner mit 48 parallel laufenden Prozessorkernen durchgeführt. Wie bereits erwähnt, sind die Ergebnisse der Berechnung die Mengen $\mathcal{S}$ und $\mathcal{S}^*$. Als Beispiel sind in Abbildung 3.3 dazu für $n = 4$, $C = 5$ die einzelnen Ordnungen dargestellt. Es ist zu beachten, dass $\mathcal{S}$ und $\mathcal{S}^*$ identisch sind. Die Zahl neben den Ordnungen ist der Index von $\mathcal{S}$ bzw. $\mathcal{S}^*$.

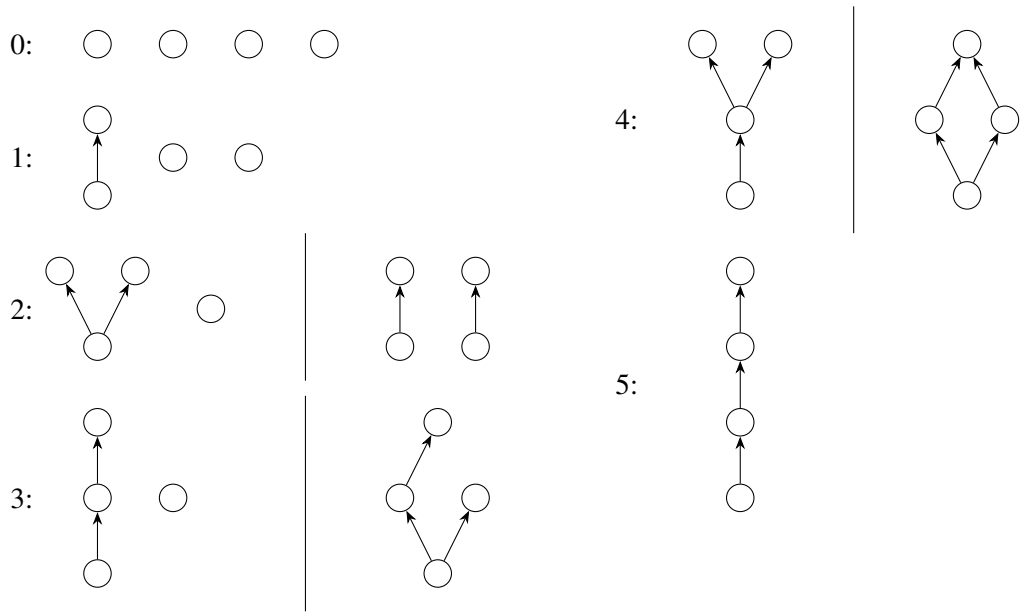

Abbildung 3.3: Die Mengen $\mathcal{S}$ und $\mathcal{S}^*$ für $n = 4$, $C = 5$

Die Anzahl der Partiellen Ordnungen ist in folgenden Tabellen dargestellt. Diese Werte entsprechen den Ergebnissen der Singlethread-Implementierung. An dieser Stelle sei gesagt, dass die Werte von denen aus [Pec04] abweichen. Beispielsweise ist hier erst $\mathcal{S}_{25}$ leer, in [Pec04] war dies bereits für $\mathcal{S}_{23}$ der Fall. Ein Grund könnte die in 3.3 erwähnte Race Condition sein. Andere Gründe sind allerdings nicht auszuschließen.

Die Berechnung für $n = 12$ benötigt ungefähr 350ms, für $n = 13$ waren es 16min 40s. Eine Berechnung für $n = 14$ kam zu keinem Ergebnis. Die Berechnung ist nach Schritt $c = 21$ von insgesamt 37 Schritten in den Forwardsteps abgebrochen. Beim Schreiben der Ordnungen in $\mathcal{S}_{22}$ ist der Speicher vollgelaufen. In diesem Schritt müssen 27 693 672 verschiedene partielle Ordnungen auf Nachfolger überprüft werden. Bis dahin wurden an die 700GB an Daten generiert und es wurde 35h 33min 26s gerechnet.

c	$\lvert\mathcal{S}_c\rvert$	$\lvert\mathcal{S}_c^*\rvert$	c	$\lvert\mathcal{S}_c\rvert$	$\lvert\mathcal{S}_c^*\rvert$
1	1	-	16	161	-
2	1	-	17	90	-
3	2	-	18	35	-
4	3	-	19	25	-
5	4	-	20	20	-
6	8	-	21	10	-
7	14	-	22	4	-
8	35	-	23	2	-
9	69	-	24	1	-
10	110	-	25	0	-
11	176	-	26	0	-
12	254	-	27	0	-
13	337	-	28	0	-
14	319	-	29	0	-
15	262	-			

c	$\lvert\mathcal{S}_c\rvert$	$\lvert\mathcal{S}_c^*\rvert$	c	$\lvert\mathcal{S}_c\rvert$	$\lvert\mathcal{S}_c^*\rvert$
1	1	0	18	398863	6
2	2	0	19	425226	17
3	4	0	20	379864	31
4	8	0	21	281403	57
5	18	0	22	174699	93
6	33	0	23	92581	176
7	68	0	24	42623	312
8	178	0	25	17659	434
9	476	0	26	6731	497
10	1277	0	27	2441	448
11	3445	0	28	867	337
12	9053	0	29	285	185
13	22662	0	30	84	82
14	53099	0	31	34	33
15	110350	0	32	6	6
16	198962	1	33	1	1
17	304836	5			

Tabelle 3.1: Detaillierte Auflistung der Anzahl der partiellen Ordnung in $\mathcal{S}$, $\mathcal{S}^*$ für $n = 12$ (l.) und $n = 13$ (r.)

Im Allgemeinen ist festzuhalten, dass trotz der einfachen Struktur des Algorithmus die Umsetzung Schwierigkeiten mit sich bringt. Das Ziel, die Ergebnisse von Peczarski [Pec07] für $n = 14$ zu reproduzieren, wurde trotz besserer Hardware nicht erreicht. Es ist auf jeden Fall Potenzial zur weiteren Optimierung gegeben.

4 Zusammenfassung und Ausblick

Mit dieser Arbeit ist versucht worden, die theoretischen Grundlagen zur Bestimmung vergleichsoptimaler Sortieralgorithmen darzulegen, um eine praktische Implementierng zu entwerfen und Grenzen experimentell zu finden. Dabei wurde auf der vorangegangenen Arbeit von Peczarski [Pec04] aufgebaut, der die unteren Schranken für 14 und 15 Elemente beweisen konnte. Aus seiner neuesten Arbeit ist der Algorithmus zum Zählen linearer Erweiterungen übernommen worden. Hier wurde auch nochmals die Korrektheit des Algorithmus gezeigt. Weiter wurden andere effektive Heuristiken zum Zählen von linearen Erweiterungen vorgestellt. Als weniger taugliche Methode hat sich in diesem Projekt der Color-Refinement-Algorithmus erwiesen, der sich bei den Versuchen als zu zeitaufwändig dargestellt hat. Trotzdem kann man in den Kongruenz-Checks Verbesserungspotenzial erkennen. Mit der hier vorgestellten Transitiven Reduktion ist eine Methode gefunden worden, die vergleichsweise schneller arbeitet, weil sie mit zwei Tiefensuchen auskommt. In dieser Arbeit sind praktische Details zur Implementierung dargestellt, zum Beispiel, an welchen Stellen sich das Programm parallelisieren lässt. Zum Schluss werden die Ergebnisse, welche die Untersuchungen dieser Arbeit ergeben haben, vorgestellt. Bestätigt werden konnten dabei Ergebnisse bis $n = 13$. Für $n = 14$ ist dies in dieser Versuchsreihe nicht gelungen.

Ausblick

Wie bereits erwähnt, ist das Potenzial der Implementierung noch lange nicht ausgeschöpft. Mit dem aktuellen Hardwarestand sollte es durchaus möglich sein, die Ergebnisse von Peczarski [Pec04] zu reproduzieren. Eine hier nicht behandelte Idee ist es, den Algorithmus auf Grafikkarten zu implementieren, um eine höhere Parallelität zu erreichen. Wann die untere Schranke für 16 Elemente gefunden wird, ist eine Frage von Hardware und wissenschaftlicher Hartnäckigkeit.

Literaturverzeichnis

[AGU72] A. V. Aho, M. R. Garey, J. D. Ullman. „The Transitive Reduction of a Directed Graph“. In: *SIAM Journal on Computing* 1.2 (1972), S. 131–137. DOI: 10.1137/0201008. eprint: https://doi.org/10.1137/0201008. URL: https://doi.org/10.1137/0201008 (zitiert auf S. 31).

[DDD06] K. De Loof, H. De Meyer, B. De Baets. „Exploiting the Lattice of Ideals Representation of a Poset“. In: *Fundam. Inf.* 71.2,3 (Feb. 2006), S. 309–321. ISSN: 0169-2968. URL: http://dl.acm.org/citation.cfm?id=1227505.1227514 (zitiert auf S. 21–23).

[GKMS17] M. Grohe, K. Kersting, M. Mladenov, P. Schweitzer. „Color refinement and its applications“. In: *Van den Broeck, G.; Kersting, K.; Natarajan, S* (2017) (zitiert auf S. 30).

[HSS08] A. A. Hagberg, D. A. Schult, P. J. Swart. „Exploring Network Structure, Dynamics, and Function using NetworkX“. In: *Proceedings of the 7th Python in Science Conference*. Hrsg. von G. Varoquaux, T. Vaught, J. Millman. Pasadena, CA USA, 2008, S. 11–15 (zitiert auf S. 25).

[Knu98] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0 (zitiert auf S. 13, 16).

[KS84] J. Kahn, M. Saks. „Balancing poset extensions“. In: *Order* 1.2 (Juni 1984), S. 113–126. ISSN: 1572-9273. DOI: 10.1007/BF00565647. URL: https://doi.org/10.1007/BF00565647 (zitiert auf S. 35).

[Obs19] J. Obst. *Git Repository*. https://gitlab.com/obstjn/minimum-comparison-sorts. Accessed: 2019-12-05. 2019 (zitiert auf S. 25, 26, 38).

[Pec02] M. Peczarski. „Sorting 13 Elements Requires 34 Comparisons“. In: *Algorithms — ESA 2002*. Hrsg. von R. Möhring Rolfand Raman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 785–794. ISBN: 978-3-540-45749-7. DOI: 10.1007/3-540-45749-6_68 (zitiert auf S. 13, 16–18, 29).

[Pec04] M. Peczarski. „New Results in Minimum-Comparison Sorting“. In: *Algorithmica* 40.2 (Juli 2004), S. 133–145. ISSN: 0178-4617. DOI: 10.1007/s00453-004-1100-7. URL: http://dx.doi.org/10.1007/s00453-004-1100-7 (zitiert auf S. 17, 21, 35, 38, 41).

[Pec07] M. Peczarski. „The Ford–Johnson algorithm still unbeaten for less than 47 elements“. In: *Information Processing Letters* 101.3 (2007), S. 126–128. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2006.09.001. URL: http://www.sciencedirect.com/science/article/pii/S0020019006002742 (zitiert auf S. 13, 39).

[Pec11] M. Peczarski. „Towards Optimal Sorting of 16 Elements“. In: *CoRR* abs/1108.0866 (2011). arXiv: 1108.0866. URL: http://arxiv.org/abs/1108.0866 (zitiert auf S. 17, 21–24, 26).

[SU00] J. Siek, a University of Notre Dame team. *Boost Graph Libary*. https://www.boost.org/doc/libs/1_71_0/libs/graph/doc/index.html. Accessed: 2019-11-30. 2000 (zitiert auf S. 25).

[Wel71] M. B. Wells. *Elements of combinatorial computing*. Oxford, 1971 (zitiert auf S. 17).

Alle URLs wurden zuletzt am 04. 12. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift