Institut für theoretische Informatik

Leibniz Universität Hannover

Bachelorarbeit

Visualisierung von Lindells Isomorphiealgorithmus für Bäume

von Tien Hung Ngo
Matrikelnummer 10006229

Juli 2020

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Maurice Chandoo

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder Ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 9. Juli 2020

Tien Hung Ngo

Inhaltsverzeichnis

1 Einleitung

Eine Isomorphie bezeichnet die strukturelle Gleichheit von zwei Graphen. Beispielsweise sind die folgenden Graphen isomorph, auch wenn sie unterschiedlich aussehen.

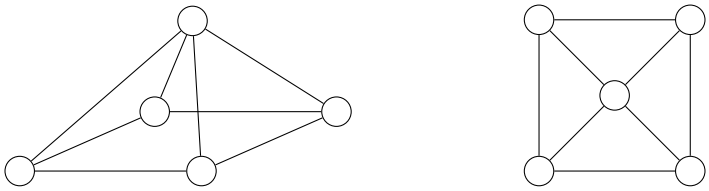


Abbildung 1.1: Beispiel isomorphe Graphen

Es ist kein Algorithmus bekannt, welcher das Problem der Isomorphie von Graphen in polynomieller Laufzeit löst. Graphenisomorphie gehört zu den wenigen Problemen in $\mathcal{NP}$, von denen nicht bekannt ist, ob sie in $\mathcal{P}$ liegen oder $\mathcal{NP}$-vollständig sind. Jedoch gibt es für bestimmte Unterklassen von Graphen, wie z. B. Bäume, Algorithmen, die Graphenisomorphie effizient lösen ([Har12, S.342]). Solche Algorithmen werden beispielsweise in der molekularen Biologie verwendet, um Baumstrukturen wie RNA-Sekundärstrukturen zu vergleichen ([Val02, S.239]). Für große Eingabebäume ist es erforderlich Algorithmen zu haben, die das Isomorphieproblem für Bäume effizient lösen. Lindells Algorithmus ([Lin92]) ist einer davon und überprüft, ob zwei gerichtete Bäume isomorph sind. Dabei benötigt der Algorithmus bei Eingabebäumen mit jeweils n Knoten zur Ausführung nur $\mathcal{O}(\log n)$ Bits Speicherplatz.

Die Implementierung von Lindells Algorithmus erfolgt über eine Methode, die sich *Trace Based Programming* nennt, welche in [Cha18] vorgestellt wurde. Die Grundlage dieser Methode ist eine Abfolge von Ausführungsschritten, welche genau beschreibt wie der Algorithmus zu einer konkreten Eingabe vorgeht. Aus dieser Abfolge, welche als *Execution Trace* (kurz *Trace*) bezeichnet wird, kann das Programm abgeleitet werden. Bei der Trace-Methode wird das Programm als ein gerichteter Graph aufgefasst, welcher den Kontrollfluss repräsentiert. Es wurde ein Framework namens *SP* bereitgestellt, welche die einzelnen Komponenten eines Kontrollflussgraphen erwartet und dessen Logik in Haskellcode übersetzt. Weiterhin bietet *SP* die Möglichkeit Traces zu generieren, sodass anhand der Traces bestimmt werden kann, ob das Programm zu einer konkreten Eingabe sich korrekt verhält.

Die Visualisierung von Lindells Algorithmus basiert auf den Traces. Diese beinhalten alle Ausführunggschritte zu einer konkreten Eingabe und liefern folglich genug Informationen, um den Algorithmus zu visualiseren. In dieser Arbeit soll Lindells Algorithmus auf einem Webbrowser mithilfe von *SP* visualisiert werden.

In Kapitel 2 werden graphentheoretische Grundlagen beschrieben und darauf basierend wird in Kapitel 3 die Funktionsweise von Lindells Algorithmus thematisiert. Infolgedessen wird in Kapitel 4 die Trace-Methode auf Lindells Algorithmus angewandt. In Kapitel 5 wird die Visualisierung des Algorithmus behandelt.

2 Grundlagen

2.1 Isomorphie

Definition 2.1.
Zwei gerichtete Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ sind zueinander **isomorph**, falls es eine bijektive Abildung $\varphi : V_1 \to V_2$ gibt, so dass $\forall u, v \in V_1$ gilt:

$$(u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$$

Anders ausgedrückt: Falls durch Umbenennung der Knoten in G_1 der Graph G_2 konstruiert werden kann, dann sind G_1 und G_2 isomorph [MVS, S. 67].

Beispiel 2.1.

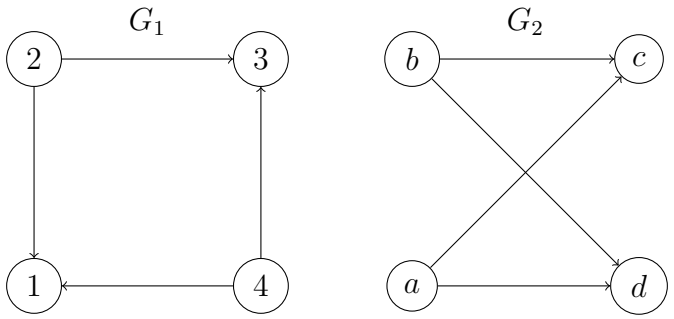


Abbildung 2.1: Beispiel isomorphe Graphen

G_1 und G_2 sind isomorph, wobei $\varphi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ d & b & c & a \end{pmatrix}$.

Definition 2.2.
Ein **gerichteter Baum** ist ein gerichteter, zusammenhängender Graph $G = (V, E)$, wobei der unterliegende ungerichtete Graph azyklisch ist und es gibt einen ausgewählten Knoten r in V, sodass für jeden Knoten $v \in V$ gilt, es gibt einen Pfad in G von r nach v. Der Knoten r heißt Wurzelknoten [Val02, S. 16–17].

Definition 2.3.
Ein gerichteter Graph $G = (V, E)$ ist **zusammenhängend**, falls für alle Paare $u, v \in V$ gilt, es gibt einen Pfad von u nach v, wobei G als ungerichteter Graph aufgefasst wird [Val02, S. 10].

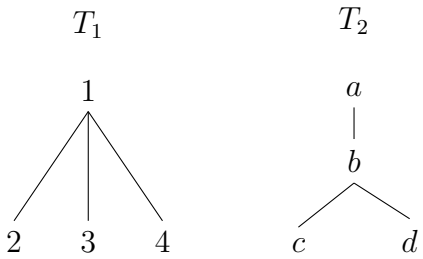


Abbildung 2.2: Beispiel Isomorphie von (un-)gerichteten Bäumen

Bei der Betrachtung der Bäume in in Abbildung 2.2, fällt auf, dass sie nicht isomorph sind, weil die Wurzelknoten $1, a$ unterschiedliche Grade haben. Werden jedoch T_1, T_2 als ungerichtete Bäume aufgefasst, dann existiert ein Isomorphismus mit

$$\varphi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ b & c & a & d \end{pmatrix}.$$

Lindells Algorithmus kann auf ungerichtete Bäume S, T erweitert werden, indem alle möglichen Kombinationen, die S, T als gerichteter Baum annehmen kann, miteinander verglichen werden. Bei einem ungerichteten Baum mit Knotenmenge V kann jeder Knoten als Wurzel ausgewählt werden, sodass es zu einem ungerichteten Baum $|V|$ unterschiedliche gerichtete Bäume gibt. Demnach ist die Anzahl der möglichen Kombinationen quadratisch zur Knotenmenge, da im schlimmsten Fall jeder gerichteter Baum aus S mit jedem gerichteten Baum aus T verglichen wird.

2.2 Notation

Sei T ein gerichteter Baum und v ein Knoten in T. Der Ausdruck $T[v]$ bezeichnet den Teilbaum von T mit Wurzel v und $\#v$ die Anzahl der Kinder von v. $|T|$ kennzeichnet die Anzahl der Knoten in T. Alle Knoten in T werden mit einer id von 1 bis n gekennzeichnet. Weiterhin können folgende Operationen auf T angewandt werden:

- $Root(T)$: Gibt den Wurzelknoten von T zurück.
- $Parent(T, v)$: Gibt zu einem Knoten v im Baum T den Elternknoten von v zurück.
- $FirstChild(T, v)$: Gibt zu einem Knoten v im Baum T das lexikographische erste Kind von v zurück.
- $NextSibling(T, v)$: Gibt zu einem Knoten v im Baum T den lexikographischen nächsten Nachbarn von v zurück.

Falls einer der Operationen fehlschlägt, dann wird ein Knoten N_0 ($id = 0$) zurückgegeben.

2.3 Ordnungsrelation

Lindells Algorithmus stellt die Existenz eines Isomorphismus von zwei Bäumen S, T fest, indem zunächst auf die Ordnungsrelation '$\prec$' geprüft wird. Falls weder $S \prec T$ noch $T \prec S$ gilt, dann sind S und T isomorph ('$\cong$'). In [Cha18] wurde die Relation '$\prec$' in einen nicht-rekursiven Teil '$\prec_1$' und einen rekursiven Teil '$\prec$' aufgeteilt, um den Algorithmus zu vereinfachen. Der nicht-rekursive Teil wird dabei um eine Relation '$<$' auf Multimengen erweitert. Eine Multimenge ist eine Menge, die Duplikate ermöglicht, wie z.B. $\{\{1, 2, 0, 2, 0\}\}$.

Definition 2.4.
Seien A, B Multimengen und $A' = (a_1, \ldots, a_n)$, $B' = (b_1, \ldots, b_n)$ die aufsteigende Sortierung der Elemente von A, B. Sei $<$ eine beliebige Ordnungsrelation. Es gilt $A < B$, falls für das erste $i \in 1 \ldots n$ mit $a_i \neq b_i$, $a_i < b_i$ gilt.

Beispiel 2.2.
Sei $A = \{\{1, 2, 0, 1, 0\}\}$, $B = \{\{2, 1, 0, 2, 0\}\}$ und $A' = (0, 0, 1, 1, 2)$,
$B' = (0, 0, 1, 2, 2)$ die aufsteigende Sortierung von A, B. Sei $<$ die natürliche Ordnung über die Menge $\mathbb{N}$. Es gilt $A < B$, weil das vierte Element von A' kleiner ist als das von B'.

Definition 2.5.
Seien S, T ungerichtete Wurzelbäume mit s, t als Wurzelknoten. Die Knoten $u_1, \ldots, u_k$ und $v_1, \ldots, v_k$ sind Kinder von s und t. Die totale Ordnungsrelation $S \prec_1 T$ gilt, falls:

1. $|S| < |T|$ oder
2. $|S| = |T|$ und $\#s < \#t$ oder
3. $|S| = |T|$ und $\#s = \#t = k$ und $\{\{|S[u_1]|, \ldots, |S[u_k]|\}\} < \{\{|T[v_1]|, \ldots, |T[v_k]|\}\}$

$S \prec T$ gilt, falls:

1. $S \prec_1 T$ oder
2. $S \not\prec_1 T, T \not\prec_1 S$ und $\{\{S[u_1], \ldots, S[u_k]\}\} \prec \{\{T[v_1], \ldots, T[v_k]\}\}$

Die ersten beiden Bedingungen von '$\prec_1$' sind trivial. Falls S weniger Knoten als T oder s weniger Kinder als t hat, dann können S und T nicht isomorph sein. Die Isomorphie ist für 1. nicht mehr gegeben, weil keine bijektive Abbildung nach Definition 2.1 existieren kann (Definitionsmenge ist kleiner als die Zielmenge). Falls die zweite Bedingung gilt, dann sind S, T nicht isomorph, weil s weniger Kinder hat als t. Die dritte Bedingung ist analog zu 1., nur mit dem Unterschied, dass die Größen der von den Kindern induzierten Teilbäumen miteinander verglichen werden. In '$\prec$' wird zunächst überprüft, ob $S \prec_1 T$ oder $T \prec_1 S$ gilt. Falls weder $S \prec_1 T$ noch $T \prec_1 S$ gilt, dann erfolgt ein rekursiver Aufruf von '$\prec$' auf die Kinder von s, t.

Satz 2.1. Zwei Bäume S, T sind **isomorph** gdw. weder $S \prec T$ noch $T \prec S$ gilt.

3 Algorithmus

Lindells Algorithmus prüft zunächst, ob $S[u] \prec T[v]$ oder $T[v] \prec S[u]$ gilt. Falls einer der Aussagen wahr ist, dann sind $S[u]$ und $T[v]$ nicht isomorph. Ansonsten gilt $S[u] \cong T[v]$. Diese Prozedur wird in eine Funktion CMP zusammengefasst.

$$CMP(S[u], T[v]) = \begin{cases} \prec, & \text{falls } S[u] \prec T[v] \\ \succ, & \text{falls } T[v] \prec S[u] \\ \cong, & \text{sonst} \end{cases}$$

Aufgrund der rekursiven Definition von '$\prec$', ist eine einfache Implementierung von CMP in $\mathcal{O}(\log n)$ nicht möglich. Mit jedem Rekursionsaufruf wird die aktuelle Umgebung auf einen Stack gespeichert, sodass bei einer Baumtiefe d, der Stack mindestens d Bits benötigt. Bei einem Baum mit n Knoten, welcher ein Pfad ist, ist die Baumtiefe $d = n - 1$ und daher aufgrund des linearen Wachstums von d nicht mehr in $\mathcal{O}(\log n)$ Speicherplatz. Um dies zu vermeiden, verwendet Lindells Algorithmus eine Form der Rekursion bei der entweder die Aufrufumgebung mit nur sehr wenig zusätzlichen Speicher oder sogar auch ohne zusätzlichen Speicher wiederhergestellt werden kann. Zusätzlich sind die Variablen, die nicht zur Eingabe gehören, auf die Datentypen `Node`, `Int`, `Tupel` und `Stack` beschränkt. Der Wertebereich von einer Variable vom Typ `Int` darf nur einen Wert zwischen 0 und n annehmen, wobei n die Anzahl der Knoten des größeren Baumes ist. Der Stack speichert Tupel bestehend aus einer konstanten Anzahl an Integern. Die ersten drei Datentypen sind nach Definition in $\mathcal{O}(\log n)$. Am Ende dieses Kapitels wird genauer erläutert warum die Verwendung eines Stacks die Speicherbegrenzung von $\mathcal{O}(\log n)$ nicht überschreitet.

3.1 Nicht-rekursive Ordnungsrelation '$\prec_1$'

Seien S, T gerichtete Bäume mit Wurzelknoten u, v. Nach einem Aufruf von CMP wird zunächst überprüft, ob eine Ungleichheit festgestellt werden kann, ohne in eine Rekursion gehen zu müssen. Es wird überprüft, ob $S[u] \prec_1 T[v]$ oder $T[v] \prec_1 S[u]$ gilt. Diese Prozedur wird in eine Funktion CMP_1 zusammengefasst.

$$CMP_1(S[u], T[v]) = \begin{cases} \prec_1, & \text{falls } S[u] \prec_1 T[v] \\ \succ_1, & \text{falls } T[v] \prec_1 S[u] \\ \cong_1, & \text{sonst} \end{cases}$$

Für die Berechnung von CMP_1 müssen folgende Probleme zunächst in $\mathcal{O}(\log n)$ Platz gelöst werden.

Das erste Problem besteht darin, zu einem Knoten v im Baum T die Anzahl der Knoten des Teilbaumes $T[v]$ zu berechnen. Dazu wird jeder Knoten im Teilbaum mittels einer Tiefensuche durchlaufen. Die Tiefensuche benutzt folgende Variablen. Einen zusätzlichen Knoten u für die aktuelle Position im Teilbaum und den letzten ausgeführten Befehl. Folgende Befehle wurden in [Lin92] verwendet:

- *down*: gehe vom Knoten u zum ersten Kind, falls es existiert.
- *over*: gehe vom Knoten u zum nächsten Nachbarn, falls es existiert.
- *up*: gehe vom Knoten u zum Elternknoten, falls es existiert.

Falls v ein Blatt ist, dann gilt $|T[v]| = 1$. Ansonsten muss v mindestens ein Kind haben und es wird *down* ausgeführt. In Abhängigkeit des zuletzt ausgeführten Befehls, werden unterschiedliche Befehlsreihenfolgen ausgeführt. Wenn ein Befehl nicht erfolgreich war, dann wird der nächste Befehl in der Reihenfolge ausgeführt. Falls der letzte Befehl *down* oder *over* war, dann lautet die Befehlsreihenfolge *down*, *over*, *up*. Ansonsten muss der letzte Befehl *up* gewesen sein. Die Befehlsreihenfolge lautet nun *over*, *up*. Falls nach einem *up*-Befehl $u = v$ gilt, dann ist der Tiefendurchlauf von $T[v]$ abgeschlossen. Für die Berechnung von $|T[v]|$ wird ein Zähler (mit 1 initialisiert) inkrementiert, wenn ein neuer Knoten besucht wird. Dies ist der Fall, wann immer der letzte Befehl *down* oder *over* war.

Für das zweite Problem muss die Anzahl der Kinder zu einem Knoten v bestimmt werden. Es wird zunächst *down* ausgeführt. Anschließend wird *over* ausgeführt bis es keinen nächsten Nachbarn mehr gibt. Die Anzahl der erfolgreichen Aufrufe ist die Anzahl der Kinder von v. Folgende Abbildung 3.1 stellt die Berechnung von $|T_3[1]|$ und $\#1$ (T_4) dar.

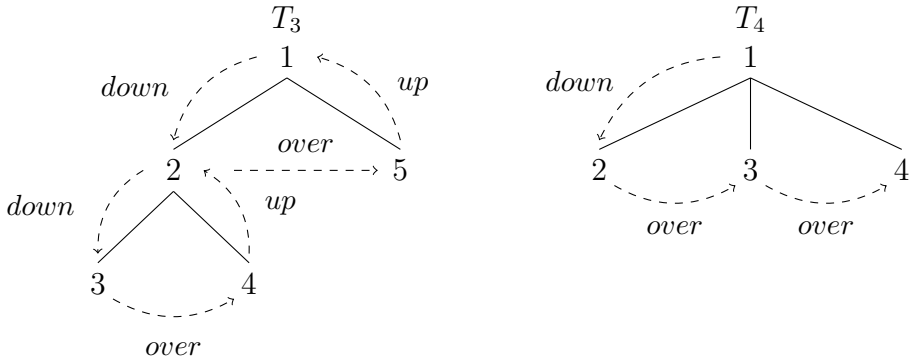


Abbildung 3.1: Logspace-Tiefendurchlauf und $\#v$

Als Nächstes gilt es, die Multimengen zu vergleichen. Dazu werden die Kinder von einem Knoten v in Blöcke unterteilt.

Definition 3.1.
Ein Block $B = \{v_1, \ldots, v_l\}$ ist eine nichtleere Teilmenge von Kindern von v mit

der Eigenschaft, dass die induzierten Teilbäume $T[v_1], \ldots, T[v_l]$ die Größe k haben ($|T[v_1]|, \ldots, |T[v_l]| = k$). Wir schreiben $B(k)$ um auf einen Block mit Größe k und $|B(k)|$ um auf die Kardinalität von $B(k)$ zu verweisen [Cha18, S. 21].

Sei zusätzlich S ein gerichteter Baum und u ein Knoten von S. Um nun eine Ungleichheit von zwei Multimengen festzustellen, werden zunächst die kleinsten Blockgrößen k_S, k_T in u, v berechnet. Falls $k_S < k_T$, dann folgt $S[u] \prec_1 T[v]$. Falls $k_S > k_T$, dann folgt $S[u] \not\prec_1 T[v]$. Ansonsten gilt $k_S = k_T$ und die Kardinalitäten der Blöcke $B_S(k_S)$ und $B_T(k_T)$ müssen miteinander verglichen werden. Falls $|B_S(k_S)| > |B_T(k_T)|$ dann besitzt S mehr minimale Knoten als T, und somit $S \prec_1 T$. Sind die beiden kleinsten Blockgrößen gleich groß, dann wird mit der nächsten Blockgröße fortgeführt. Die Berechnung von $S[u] \prec_1 T[v]$ ist beendet, falls keine weiteren Blockgrößen k mehr existieren oder eine Ungleichheit festgestellt wurde.

Beispiel 3.1.

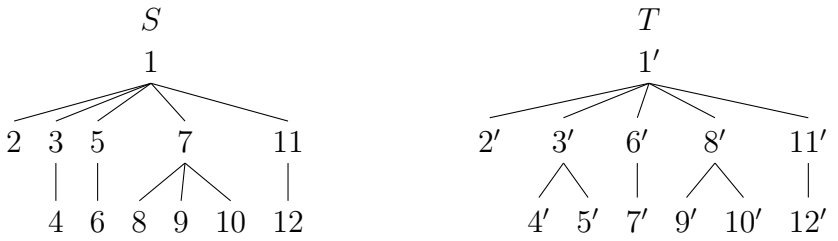


Abbildung 3.2: Beispielbäume für '$\prec_1$'

Es folgt ein Beispiel von '$\prec_1$' im Bezug auf die Bäume S, T in Abbildung 3.2. Offensichtlich gilt $|S| = |T| = 12$ und $\#1 = \#1' = 5$. Die Blockgrößen müssen nun miteinander verglichen werden. Es gilt:

$$\{\{|S[2]|, |S[3]|, |S[5]|, |S[7]|, |S[11]|\}\} = \{\{1, 2, 2, 4, 2\}\}$$

$$\{\{|T[2']|, |T[3']|, |T[6']|, |T[8']|, |T[11']|\}\} = \{\{1, 3, 2, 3, 2\}\}$$

Die kleinste Blockgröße ist $k = 1$. Es gilt $|B_S(1)| = |B_T(1)| = 1$. Der nächstkleinere Block hat die Größe $k = 2$. Es folgt $|B_S(2)| = 3 \neq 2 = |B_T(2)|$. Der Knoten 1 besitzt mehr minimale Knoten als $1'$, d.h. $S \prec_1 T$.

3.2 Rekursiver Vergleich

Es wird angenommen, dass CMP_1 für alle Eingaben das Ergebnis '$\cong_1$' liefert, und somit der Algorithmus in die Rekursion gehen muss. Der Algorithmus simuliert einen Rekursionsaufruf, indem die aktuellen Knoten auf die Kinderknoten gesetzt werden. Die Auswahl der Knoten folgt nach einem bestimmten Prinzip. Dazu wird zunächst die kleinste Blockgröße k bestimmt. Folglich werden die Blöcke $B_S(k)$ und $B_T(k)$ miteinander verglichen.

Definition 3.2.
Die Blöcke B_S, B_T stimmen überein, falls es eine bijektive Abbildung $\gamma : B_S \to B_T$ gibt, sodass $\forall u \in B_S$ gilt: $CMP(S[u], T[\gamma(u)]) = \text{`} \cong \text{'}$ [Cha18, S. 21].

Das Ergebnis von CMP wird in einer weiteren Variable res gespeichert. Falls die Blöcke übereinstimmen und weitere Blöcke existieren, dann wird der kleinste Block ausgewählt, welcher größer als k ist. Angenommen u, v sind Blätter oder stellen $S[u] \ncong T[v]$ fest, dann muss die Aufrufumgebung wiederhergestellt werden. Dazu werden die aktuellen Knoten auf ihre Elternknoten gesetzt. Dies reicht jedoch nicht aus, um mit der rekursiven Prozedur fortzufahren, da es von den Elternknoten aus, nicht möglich ist festzustellen, welcher Block als Nächstes besucht werden soll. Hierzu führt Lindell eine speicherfreie Rekursion und eine Rekursion mit Ordnungsprofilen ein, welche die Aufrufumgebung mit $\mathcal{O}(\log n)$ Speicherplatz wiederherstellen können. Der Algorithmus terminiert, falls von den Wurzelknoten aus alle Blöcke bereits besucht wurden oder eine Ungleichheit festgestellt wurde.

Speicherfreie Rekursion

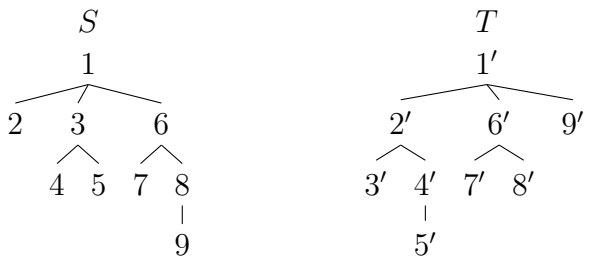


Abbildung 3.3: Beispielbäume für die speicherfreie Rekursion

Die speicherfreie Rekursion tritt ein, falls ein Block nur aus einem Knoten besteht ($l = 1$). Folgende Bäume S, T beziehen sich auf die Bäume in Abbildung 3.3. Der Algorithmus führt zunächst $CMP(S[1], T[1'])$ aus. Offensichtlich liefert $CMP_1(S[1], T[1'])$ das Ergebnis '$\cong_1$', da beide Bäume gleich viele Knoten haben, die Wurzelknoten gleich viele Kinder haben und die Kardinalitäten der Blockgrößen übereinstimmen. Da keine Ungleichheit festgestellt wurde, müssen die Blöcke $B_S(1) = \{2\}$ und $B_T(1) = \{9'\}$ miteinander verglichen werden. Diese Blöcke stimmen nach Definition 3.2 überein, da $CMP(S[2], T[9'])$ das Ergebnis '$\cong$' liefert. Die aktuellen Knoten $2, 9'$ haben keine weiteren Kinder, welches $k = 0$ impliziert. Die Aufrufumgebung muss nun wiederhergestellt werden. Dazu werden die Knoten $u = 2, v = 9'$ auf ihre Elternknoten $1, 1'$ gesetzt und k auf die aktuelle Blockgröße $|S[u]| = 1$. Formal ausgedrückt:

$$(u, v, k) \mapsto (Parent(S, u), Parent(T, v), |S[u]|)$$

In diesem Beispiel wird die Konfiguration $(2, 9', 0)$ auf $(1, 1', 1)$ gesetzt. Anhand dessen kann bestimmt werden, dass der letzter besuchte Block die Größe $k = 1$ hatte und

somit die nächste Blockgröße bestimmt werden kann, welcher in diesem Beispiel $k = 3$ ist. Daher sind die nächsten zu vergleichenden Blöcke $B_S(3) = \{3\}$ und $B_T(3) = \{6'\}$. Nachdem die Blöcke der Größe $k = 3$ und $k = 4$ miteinander verglichen wurden, wird festgestellt, dass es von den Wurzelknoten aus keinen weiteren Block mit $k > 4$ existiert. Der Algorithmus terminiert und es gilt $S \cong T$. Angenommen $CMP(S[3], T[6'])$ liefert das Ergebnis '$\prec$', welches $S \prec T$ impliziert. Der Algorithmus stellt die Aufrufumgebung wiederher und terminiert.

Rekursion mit Ordnungsprofilen

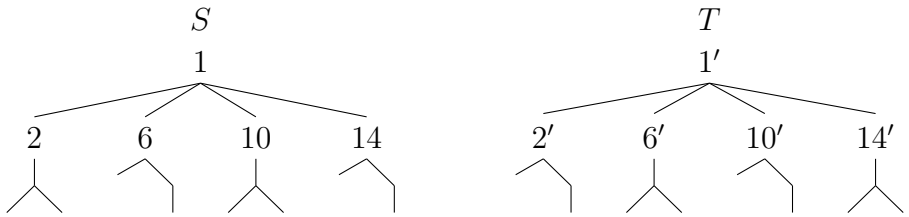


Abbildung 3.4: Beispielbäume für die Rekursion mit Ordnungsprofilen

Die Rekursion mit Ordnungsprofilen tritt ein, falls ein Block aus mehr als einem Knoten besteht ($l > 1$). Folgende Bäume S, T beziehen sich auf die Bäume in Abbildung 3.4. Die vereinfachten Teilbäume '$\curlywedge$' und '$\curlywedge\!\!\!\!\curlywedge$' haben alle die gleiche Größe k, was dazu führt, dass die Wurzelknoten nur einen Block $B(k)$ mit Kardinalität $l = 4$ haben. Weiterhin ist zu beachten, dass '$\curlywedge$' $\prec$ '$\curlywedge\!\!\!\!\curlywedge$' gilt. Die grundlegende Idee hierbei ist es, die unterschiedlichen Teilbäume zu klassifizieren und anschließend zu zählen. In [Cha18] wurden isomorphe Teilbäume zu einem Isomorphietyp zusammengefasst. Bei der Betrachtung der Bäume S, T ist festzustellen, dass die Isomorphietypen '$\curlywedge$' und '$\curlywedge\!\!\!\!\curlywedge$' jeweils zwei Mal in S und T vorkommen und demnach S, T isomorph sind. In Lindells Algorithmus wird dies anhand eines Ordnungsprofils berechnet.

Definition 3.3 ([Cha18, S. 23])**.**
Das Ordnungsprofil von einem Knoten $u \in B_S(k)$ besteht aus zwei Zählern (gt_u, eq_u), wobei

$$gt_u = |\{\ v \mid v \in B_T(k)\ \wedge\ CMP(S[u], T[v]) = \text{`}\succ\text{'}\}|$$
$$eq_u = |\{\ v \mid v \in B_T(k)\ \wedge\ CMP(S[u], T[v]) = \text{`}\cong\text{'}\}|.$$

Beispiel 3.2.
Das Ordnungsprofil von Knoten 2 ist (0,2) und von Knoten 6 ist (2,2), da

$$gt_2 = |\varnothing| = 0, \quad eq_2 = |\{6', 14'\}| = 2$$
$$gt_6 = |\{6', 14'\}| = 2, \quad eq_6 = |\{2', 10'\}| = 2.$$

Lindells Algorithmus beginnt zunächst nach der Suche eines Knotens u in $B_S(k)$, welches kleiner ist als alle anderen Knoten v in $B_T(k)$, also $gt_u = 0$. Nachdem u gefunden wurde, wird als Nächstes nach einem Knoten v in $B_T(k)$ mit $gt_v = 0$ gesucht. In dem Beispiel sind das die Knoten $u = 2$ und $v = 6'$. Da beide Knoten $2, 6'$ den gleichen gt-Wert aufweisen und somit in beiden Blöcken B_S, B_T die kleinsten Knoten sind, muss $S[2] \cong T[6']$ gelten. Nun kann es dazu kommen, dass ein Block mehrere minimale Knoten hat. Dazu werden die eq-Werte miteinander verglichen. In diesem Fall gilt $eq_2 = 2 = eq_{6'}$, d.h. der Teilbaum '$\curlywedge$' gibt es in beiden Blöcken zweimal. Die Ordnungsprofile stimmen überein und führen mit dem nächstkleineren Isomorphietyp fort. Dazu wird ein Knoten mit $gt = 2$ gesucht. Dieser Wert lässt sich aus dem vorherigen Ordnungsprofil ableiten. Da bereits zwei Knoten gefunden wurden, die den kleinsten Isomorphietyp aufweisen, gibt es exakt zwei Knoten eines Isomorphietyps die kleiner sind als der Isomorphietyp des nächsten Knoten. Nachdem ein Isomorphietyp abgearbeitet wurde, wird einen Schwellenwert h gesetzt, welcher der Summe des aktuellen Ordnungsprofils $h = gt_w + eq_w$ entspricht. Resultierend daraus, wird für den nächstkleineren Isomorphietyp Knoten $w + 1$ mit $gt_{w+1} = h$ gesucht. In unserem Beispiel sind die ersten Knoten mit $gt = 2$ die Knoten $u = 6$ und $v = 2'$. Da $eq_6 = 2 = eq_{2'}$, gibt es den Teilbaum '$\curlyvee$' in beiden Blöcken zweimal. Es wird festgestellt, dass die Summe des Ordnungsprofils der Kardinalität des Blocks $l = 4 = gt_6 + eq_6$ entspricht. Es folgt, dass es keinen weiteren Isomorphietyp in den Blöcken gibt. In beiden Blöcken wurden für $gt = 0$ zwei Knoten und für $gt = 2$ zwei Knoten gezählt, d.h. bei einer Kardinalität $l = 4$ wurden alle Knoten einem Isomorphietyp zugeordnet. Demnach wird die Prozedur beendet, falls Folgendes gilt: $l = gt + eq$.

Im Folgenden werden Fälle betrachtet, falls eine Ungleichheit festgestellt wird. Weiterhin ist u ein Knoten in B_S und v ein Knoten in B_T.

Fall 1: $eq_u < eq_v$. In diesem Fall hat der Block B_T mehr minimale Isomorphietypen als B_S. Es folgt $T \prec S$. Analoges gilt für den Fall $eq_u > eq_v$.

Fall 2: Kein Knoten u mit $gt_u = 0$ gefunden. In diesem Fall hat der Block B_T mindestens einen minimalen Isomorphietyp, der kleiner ist als jeder andere Isomorphietyp in B_S. Es folgt $T \prec S$. Analoges gilt für den Fall, falls kein Knoten v mit $gt_v = 0$ gefunden wurde.

Bevor die Erklärung beginnt, wie der Algorithmus für den Fall $l > 1$ die Aufrufumgebung wiederherstellt, wird zunächst zusammengefasst, welche Variablen benötigt werden. Die bereits in der speicherfreien Rekursion deklarierten Variablen u, v, k werden weiterhin verwendet, um $B_S(k)$ und $B_T(k)$ zu bestimmen. Da durch $B_S(k)$ und $B_T(k)$ iteriert wird, muss in den Variablen u_1, v_1 der aktuelle Knoten in den jeweiligen Blöcken gespeichert werden. Das (Zwischen-)Ergebnis eines Ordnungsprofils von u_1 wird in den Variablen sgt, seq und für v_1 in tgt, teq gespeichert. Die Variable f legt fest, in welchem Block (B_S oder B_T) nach einem Knoten mit $gt = h$ gesucht wird. Es ist nicht notwendig die Blockkardinalität l zu speichern, da diese aus den Variablen S, u, k berechnet werden kann ([Cha18, S. 23–24]). Dazu wird ein Zähler inkrementiert, falls ein Kind u_i von u die Bedingung $|S[u_i]| = k$ erfüllt. In diesem Zusammenhang kann die Aufrufumgebung wie folgt wiederhergestellt werden. Da nach einem Aufruf von CMP sichergestellt wird, dass der Algorithmus zum Knoten gelangt, der CMP aufgerufen hat, kann u_1, v_1 auf

u, v gesetzt werden. Anschließend werden u, v wie in dem Fall $l = 1$ auf ihre Elternknoten und k auf die aktuelle Blockgröße gesetzt. Die restlichen Variablen werden als ein Tupel $(h, sgt, seq, tgt, teq, f)$ zusammengefasst. Bevor CMP aufgerufen wird, wird $(h, sgt, seq, tgt, teq, f)$ auf einem Stack gelegt und falls der Algorithmus von der Rekursion zurückkehrt, wird es vom Stack heruntergenommen. Die Verwendung eines Stacks überschreitet dabei nicht die Speicherbegrenzung von $\mathcal{O}(\log n)$.

Beweis ([Cha18, S. 23])**.** Seien S, T gerichtete Bäume der Größe n. Sei $B(k_1)$ ein Block mit Kardinalität l_1. Es gilt nach dem ersten Rekursionsaufruf:

$$l_1 k_1 \leq n \Leftrightarrow l_1 \leq n/k_1$$

Das erste Element im Stack ist ein Tupel bestehend aus Werten x_1 mit $0 \leq x_1 \leq l_1$. Sei nun $B(k_2)$ ein Block mit Kardinalität l_2 im Block $B(k_1)$. Nach dem zweiten Rekursionsaufrufs gilt:

$$l_2 k_2 \leq k_1 \Leftrightarrow l_2 \leq k_1/k_2$$

Das oberste Element ist nun ein Tupel bestehend aus Werten x_2 mit $0 \leq x_2 \leq l_2$. Nach r Rekursionsschritten sei nun $B(k_r)$ ein Block mit Kardinalität l_r. Es folgt:

$$l_r \leq k_{r-1}/k_r$$

Das oberste Element ist nun ein Tupel bestehend aus Werten x_r mit $0 \leq x_r \leq l_r$. Es muss gezeigt werden, dass das Produkt aller maximalen Werte eines Ordnungsprofils nicht größer als n ist.

$$\frac{n}{k_1} \cdot \frac{k_1}{k_2} \cdot \ \ldots \ \cdot \frac{k_{r-1}}{k_r} \leq n \Leftrightarrow 1 \leq k_r$$

Die Ungleichung ist für alle $k_r \geq 1$ erfüllt, es folgt $l_1 l_2 \ldots l_r \leq n$.

Die Abbildung 3.5 bildet den Stack nach r Rekursionsschritten mit den jeweiligen Ordnungsprofilwerten l_i ab.

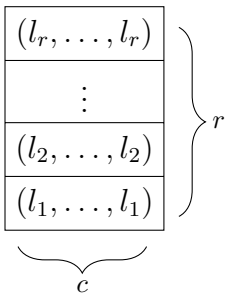


Abbildung 3.5: Stack nach r Rekursionsaufrufen mit c-Tupel, $c \in \mathbb{N}$

Es ist leicht zu sehen, dass der Stack maximal $c \sum_{i=1}^{r} \log l_i$ Bits an Speicher benötigt und somit in $\mathcal{O}(\log n)$ ist, da

$$\sum_{i=1}^{r} \log l_i = \log (l_1 l_2 \ldots l_r) \leq \log (\frac{n}{k_1} \cdot \frac{k_1}{k_2} \cdot \ \ldots \ \cdot \frac{k_{r-1}}{k_r}) \leq \log (n).$$

□

Anhand des Beweises ist zu sehen, dass es Notwendig ist, die Rekursion für $l = 1$ und $l > 1$ aufzuteilen. Angenommen ein Block hat die Kardinalität $l = 1$. Dann gibt es keinen Platz zusätzliche Werte zu speichern, da $\log(1) = 0$.

4 Implementierung

Im Folgenden wird genauer auf die Trace-Methode eingegangen. Wie bereits erwähnt, wird bei der Trace-Methode das Programm als gerichteter Graph aufgefasst, welcher den Kontrollfluss repräsentiert. Die Knoten im Graphen entsprechen einer Operation und jede Kante ist mit einer boole'schen Kombination von Prädikaten beschriftet (wobei ein Prädikat angibt, ob die Variablenwerte eine gewisse Eigenschaft erfüllen). Die boole'sche Kombination kann selbst als ein Prädikat gesehen werden, was als Kantenprädikat bezeichnet wird. Nachdem eine Operation ausgeführt wurde, soll gelten, dass höchstens ein Kantenprädikat der ausgehenden Kanten dieser Operation wahr ist (d.h. das Programm ist deterministisch, [Cha18, S. 4]). Zusätzlich wird eine Halteoperation eingeführt, welche bei Erreichen dieser Operation das Programm beendet. Die Trace-Methode kann in folgende Schritte untergliedert werden:

1. **Trace schreiben**
Dazu werden zunächst die Variablen, die der Algorithmus benötigt, bestimmt. Im Anschluss wird der Algorithmus schrittweise anhand einer konkreten Eingabe ausgeführt. Falls der Algorithmus im Ausführungsschritt i der Variable j einen Wert zuweist, dann wird dieser Wert an Trace$[i][j]$ eingetragen.

2. **Trace verallgemeinern**
In diesem Schritt wird untersucht wie zwei aufeinanderfolgende Zeilen zusammenhängen. Dieser Zusammenhang erfolgt, indem jeder Wert in Zeile $i+1$ durch dem entsprechenden vorherigen Wert in Zeile i dargestellt wird (z.B. $function(x_i) = x_{i+1}$). Daraus resultierend können die Operationen bestimmt werden. Dazu erhält jede Zeile einen Operationsnamen, wobei Zeilen, die die gleichen Ausdrücke beinhalten, zu einer Operation zugeordnet werden können.

3. **Kontrollflussgraph ableiten**
Eine Operation entspricht einem Knoten im Kontrollflussgraphen. Es wird eine Kante von Operation A zu Operation B gezogen, falls in einem Trace Operation B auf A folgt. Eventuell deckt ein Trace nicht alle Fälle des Algorithmus ab. Daher werden die vorherigen Schritte für weitere Eingaben wiederholt.

4. **Kantenprädikate hinzufügen**
Hierbei muss bestimmt werden, unter welchen Bedingungen eine Operation in die nächste Operation übergeht.

In [Cha18] wurde Lindells Algorithmus bereits anhand der Trace-Methode implementiert.

4.1 Unterprogramme

Für die Implementierung von Lindells Algorithmus wurden folgende Unterprogramme geschrieben:

- $SizeLogspace(T, v)$: Berechnet $|T[v]|$.
- $NumberOfChildren(T, v)$: Berechnet $\#v$.
- $BlockCardinality(T, v, k)$: Berechnet $|B_T(k)|$.
- $NonRecursiveCheck(S, u, T, v)$: Berechnet $S[u] \prec_1 T[v]$.
- $NextBlock(T, v, k)$: Gibt in einem Baum T zu einem Knoten v die kleinste Blockgröße zurück, die größer als k ist und 0 sonst.
- $GetChild(T, v, v_1, k)$: Gibt in einem Baum T zu einem Knoten v das lexikographische kleinste Kind zurück, welches lexikographisch größer als Knoten v_1 und im Block $B_T(k)$ ist. Falls solch ein Knoten nicht existiert, dann wird der Knoten N_0 zurückgegeben.

Die Implementierung der ersten vier Unterprogramme wurde bereits im vorherigen Kapitel behandelt. Für die Implementierung von $NextBlock$ und $GetChild$ in $\mathcal{O}(\log n)$ Platz müssen alle Kinder von v besucht werden. Dies erfolgt durch einen Aufruf von $FirstChild$, gefolgt von $NextSibling$-Aufrufen. Währenddessen wird mittels der zuvor geschriebenen Unterprogrammen ein Zähler inkrementiert bzw. der Ergebnisknoten aktualisiert, falls die jeweiligen Bedingungen wahr sind. Das Resultat wird zurückgegeben, nachdem alle Kinder von v besucht wurden.

4.2 Traces schreiben und verallgemeinern

Im vorherigen Kapitel wurden die Variablen, die der Algorithmus benötigt, bereits zusammengefasst. Die Eingabebäume S, T werden vom Algorithmus nicht verändert und können aus den Traces ausgelassen werden. Um die Implementierung übersichtlich zu halten, wird das Schreiben und die Verallgemeinerung eines Traces in einem Schritt zusammengefasst. Weiterhin werden die verallgemeinerten Ausdrücke einer Operation in der Tabelle 5.2 wiedergegeben. Die folgenden Bäume S, T beziehen sich auf die Bäume in Abbildung 5.1 und Zeilenverweise auf die Trace-Tabelle 5.1.

Der Algorithmus beginnt mit der Operation INIT, welche die Knoten u, v auf die entsprechenden Wurzelknoten von S, T setzt und k, stk mit 0 bzw. [ ] initialisiert. Da von Knoten $u = v = 1$ aus keine Ungleichheit festgestellt wurde, muss als Nächstes die kleinste Blockgröße k bestimmt werden. Dies geschieht in der Operation NB (NextBlock). In diesem Beispiel haben die Knoten $u = v = 1$ jeweils vier Kinder, die zum Block $B(4)$ zugeordnet sind, d.h. $|B(4)| = 4 = l$. Da $l > 1$, folgt in Zeile 3 die Operation SETH, welche $h = 0$ initialisiert. Die Operation SETH ist ein Indikator für den Anfang einer Rekursion mit Ordnungsprofilen. In Zeile 4 wird die Operation FINDS erreicht,

welche die Berechnung eines Ordnungsprofils zu einem Knoten im Block B_S startet. Die Operation FINDS setzt u_1, v_1 auf die lexikographischen kleinsten Kinder von $B_{S/T}(4)$ und initialisiert die Ordnungsprofil-Variablen sgt, seq, tgt, teq mit 0, sowie $f = S$. In Zeile 5 wird in die Operation PUSH ausgeführt, welche einen Rekursionsaufruf indiziert und die aktuelle Aufrufumgebung auf dem Stack *stk* speichert. Die Operation PUSH setzt die Knoten u, v auf u_1, v_1, $k = 0$ und legt die Werte von h, sgt, seq, tgt, teq, f als Tupel auf *stk*. Da $CMP(S[2], T[2]) =$ '$\prec$', folgt in Zeile 6 die Operation '$\prec$', welche *res* auf '$\prec$' setzt. Damit ist der Vergleich von $S[2], T[2]$ beendet und die Aufrufumgebung muss wiederhergestellt werden. Dies geschieht in der Operation RET2 in Zeile 7. Die Operation RET2 setzt die Variablen u, v auf ihre Elternknoten, k auf $|S[u]|$ und u_1, v_1 auf u, v. Der Ausdruck $stk_{last}[var]$, $var \in \{h, sgt, seq, tgt, teq, f\}$, kennzeichnet das oberste Element vom Stack *stk* und davon der zugehörige Wert zur Variable *var*. In RET2 wird das oberste Tupel von *stk* in die entsprechenden Variablen extrahiert und anschließend vom Stack entfernt. Basierend auf den folgenden Bedingungen werden die Ordnungsprofil-Variablen aktualisiert.

$$x_{sgt} = \begin{cases} 1, & \text{falls } f = S \ \wedge \ res = \succ \\ 0, & \text{sonst} \end{cases} \qquad x_{seq} = \begin{cases} 1, & \text{falls } f = S \ \wedge \ res = \cong \\ 0, & \text{sonst} \end{cases}$$

$$x_{tgt} = \begin{cases} 1, & \text{falls } f = T \wedge res = \prec \\ 0, & \text{sonst} \end{cases} \qquad x_{teq} = \begin{cases} 1, & \text{falls } f = T \wedge res = \cong \\ 0, & \text{sonst} \end{cases}$$

Für x_{tgt} soll $res =$ '$\prec$' gelten, da immer $S[u]$ mit $T[v]$ verglichen wird. Demnach hat $T[v]$ einen größeren Isomorphietyp als $S[u]$ gdw. $S[u] \prec T[v]$. In Zeile 7 wird keiner dieser Variablen aktualisiert, da keiner der oben aufgelisteten Bedingungen wahr ist. In Zeile 8 folgt die Operation NXTS, welche v_1 auf den lexikographischen nächstkleinsten Knoten aus $B_T(4)$ setzt, d.h. $v_1 = 6$. Beim Vergleichen der Teilbäume $S[2]$ und $T[6]$ wird festgestellt, dass beide Knoten nur ein Kind haben. Es folgt eine speicherfreie Rekursion, welche von der Operataion GC in Zeile 11 eingeleitet wird. Die Operation GC (*GetChild*) setzt u, v auf die lexikographischen kleinsten Knoten in $B_{S/T}(k)$, sowie $k = 0$, also $(u = 3, v = 7, k = 0)$. Die Berechnung von $CMP(S[3], T[7])$ ist in Zeile 31 beendet und es folgt die Operation RET. Diese Operation setzt u, v auf ihre Elternknoten und k auf $|S[u]|$. Da $S[2] \cong T[6]$ und $f = S$ wird *seq* in Zeile 34 inkrementiert. Die Berechnung des Ordnungsprofils von Knoten $u = 2$ ist in Zeile 65 beendet. Da $sgt = 0$, wurde ein Knoten in $B_S(4)$ mit $sgt = h$ gefunden. Als Nächstes muss ein Knoten in $B_T(4)$ mit $tgt = h$ gesucht werden. Folglich wird in Zeile 66 in der Operation FINDT u_1, v_1 auf die lexikographischen kleinsten Knoten aus $B_{S/T}(4)$ und $f = T$ gesetzt. Es wird nun das Ordnungsprofil von Knoten $v = 2$ berechnet. Die Operations NXTT ist analog zu NXTS, mit dem Unterschied, dass u_1 auf den lexikographischen kleinsten Knoten in $B_S(4)$ gesetzt wird. Dadurch wird jeder induzierter Teilbaum aus $B_S(4)$ mit $T[2]$ verglichen. Die Operation NXTT wird in den Zeilen 70,75,79 aufgerufen, wobei $CMP(S[6], T[2])$, $CMP(S[10], T[2])$, $CMP(S[14], T[2])$ berechnet wird. Die Berechnung des Ordnungsprofils vom Knoten $v = 2$ ist in Zeile 83 beendet. Da $tgt = 2 \neq 0$, ist $v = 2$ nicht der gesuchte Knoten und es folgt in Zeile 84 die Operation NCT (*Next Candidate*

in B_T). NCT setzt v_1 auf den lexikographischen nächstkleinsten Knoten in $B_T(4)$. Es wird das Ordnungsprofil vom Knoten $v = 6$ berechnet, welche in Zeile 101 beendet ist. Da $tgt = 0 = h$ und $seq = 2 = teq$, wird h um seq in der Operation INCH in Zeile 102 inkrementiert. Es wird nun ein Knoten mit $sgt = 2$ in $B_S(4)$ gesucht. Die Berechnung des Ordnungsprofils vom Knoten $u = 2$ beginnt in Zeile 103. In Zeile 120 ist die Berechnung des Ordnungsprofils vom Knoten $u = 2$ beendet. Der Knoten $u = 2$ ist nicht der gesuchte Knoten, da $sgt = 0 \neq 2$ und es folgt die Operation NCS, welche analog zu NCT ist, nur dass u_1 auf den lexikographischen nächstkleinsten Knoten in $B_S(4)$ gesetzt wird. In Zeile 122 bis 138 wird das Ordnungsprofil von Knoten $u = 6$ berechnet. Da $sgt = 2 = h$, wird nun ein Knoten in $B_T(4)$ mit $tgt = 2$ gesucht. Das Ordnungsprofil von Knoten $v = 2$ wird berechnet, welche in Zeile 156 beendet ist. Die Ordnungsprofile stimmen überein und die Operation INCH wird ausgeführt. Da $|B_S(4)| = 4 = h$, stimmen die Blöcke $B_S(4)$ und $B_T(4)$ überein. Der Algorithmus stellt fest, dass es keine weitere Blockgröße $k > 4$ gibt und terminiert.

4.3 Kontrollflussgraphen ableiten

Es gibt eine Kante von Operation A zu Opeation B, falls in einem Trace B auf A folgt. Es wird eine Adjazenzmatrix über die Menge der zuvor definierten Operationen angelegt. Die Einträge der Tabelle 5.3 sind Verweise auf eine Zeile der Tabelle 5.1. Beispielsweise gibt es eine Kante von INIT zu NB, weil NB (Zeile 2) auf INIT (Zeile 1) folgt. Zur Vervollständigung des Graphen müssen weitere Kanten gezogen werden, welche in der Adjazenzmatrix mit einem '*' gekennzeichnet sind. Angenommen die Eingabebäume sind unterschiedlich groß, dann muss es einen Übergang von INIT zu '$\prec$' bzw. '$\succ$' geben. Falls die Eingabebäume nur aus einem Knoten bestehen, dann gibt es eine Kante von INIT zu '$\cong$'. Da nach einem Rekursionsaufruf eine Ungleichheit festgestellt werden kann, gibt es eine Kante von GC zu '$\prec$' und '$\succ$'. Es gibt eine Kante von GC zu '$\cong$', falls nach einem Rekursionsaufruf die zu vergleichenden Teilbäume aus nur einem Knoten bestehen. Bei genauerer Betrachtung der Operationen INIT, GC und PUSH ist zu bemerken, dass diese Operationen die gleichen ausgehenden Kanten haben und nach einem INIT, GC oder PUSH die Funktion CMP ausgeführt wird. Zur Vereinfachung des Kontrollflussgraphen wird in [Cha18] eine Operation CMP als NOOP eingeführt, welche von INIT, GC und PUSH ausgehend, jeweils eine eingehende Kante und die ausgehenden Kanten der zuvor genannten Operationen erhält. Die Operation RET besitzt ebenfalls die gleichen ausgehenden Kanten. Jedoch wäre es nicht korrekt CMP unmittelbar nach RET aufzurufen, da nach der Operation RET, die Knoten u, v, nun die Knoten sind, auf denen CMP zuvor ausgeführt wurde. Weiterhin gibt es eine Kante von RET zu '$\prec$' bzw. '$\succ$', falls festgestellt wurde, dass zwei Teilbäume nicht isomorph sind. Aus dem gleichen Grund warum es eine Kante von RET zu NB gibt, muss es auch eine Kante von INCH zu NB geben, sodass nach einer Rekursion mit Ordnungsprofilen weitere Blöcke miteinander verglichen werden können. Im Bezug auf die Bäume in Abbildung 5.1 würde dieser Fall eintreten, falls es zum Beispiel noch einen weiteren Block $B(6)$ geben würde. Falls die Ordnungsprofile nicht übereinstimmen gibt es einen Übergang von RET2 zu '$\prec$' bzw. '$\succ$'.

Es folgt der Kontrollflussgraph in Abbildung 5.2.

4.4 Kantenprädikate

Kantenprädikate zu einer Operation werden in einem Prädikatenbaum zusammengefasst. Ein Prädikatenbaum ist ein Binärbaum, wobei die inneren Knoten Prädikate und die Blätter Operationen entsprechen. Prädikatenbäume zu Operationen mit nur einer ausgehenden Kante $a \to b$ bestehen aus einem Knoten b. Die Operationen INIT, GC, SETH, FINDS, FINDT, NXTS, NXTT, NCS, NCT und PUSH sind solche Operationen. Die Prädikatenbäume für die restliche Operationen werden in den Abbildungen 5.3 und 5.4 zusammengefasst. Die abgebildeten Prädikatenbäume sind so dargestellt, dass die linke Kante genommen wird, falls das Prädikat erfüllt ist.

Im Prädikatenbaum zu CMP wird zunächst überprüft, ob das Prädikat $S[u] \prec T[v]$ gilt. Ist dieses Prädikat erfüllt, dann wird die Kante zu $\prec$ genommen. Gilt dies nicht, dann wird überprüft, ob $T[v] \prec S[u]$ gilt. Ist dieses Prädikat erfüllt, dann wird die Kante zu $\succ$ genommen. Ansonsten wird überprüft, ob das Prädikat $hasNB$ gilt. Dieses Prädikat überprüft, ob es noch einen nächsten Block gibt:

$$hasNb \iff nextBlock(S, u, k) \neq 0$$

Gilt $hasNB$, dann wird die Kante von CMP zu NB genommen. Ansonsten ist die Folgeoperation $\cong$.

Der Prädikatenbaum zu RET ist analog zu CMP, nur mit dem Unterschied, dass in den ersten beiden Prädikaten die Folgeoperation anhand der Variable res bestimmt wird.

Im Prädikatenbaum zu NB wird bestimmt, welche Rekursionsform gewählt werden muss. Falls $|B_S(k)| = 1$, dann erfolgt eine speicherfreie Rekursion (GC) und ansonsten eine Rekursion mit Ordnungsprofilen (SETH).

Der Prädikatenbaum zu INCH bestimmt, ob es in einem Block noch einen nächsten Isomorphietyp gibt. Dies ist der Fall solange $h < |B_S(k)|$ gilt und es folgt die Operation FINDS. Ansonsten wird wie in CMP und RET überprüft, ob es noch einen nächsten Block gibt.

Die Operationen $\prec, \cong, \succ$ erhalten den gleichen Prädikatenbaum. Falls u der Wurzelknoten von S ist, dann folgt die Operation HALT. Ansonsten muss überprüft werden, welche Rekursionsform gewählt wurde. Dies kann bestimmt werden, indem die Blockkardinalität bezüglich des Elternknotens berechnet wird. Es folgt:

$$isRET \iff BlockCardinality(S, Parent(u), |S[u]|) = 1$$

Falls $isRET$ erfüllt ist, dann wird die Kante zu RET genommen, ansonsten die Kante zu RET2.

Es wird zunächst der linke Teilbaum von RET2 betrachtet. Die Operation NXTS folgt auf RET2, falls $f = S$ und v_1 $isLast$ nicht erfüllt ist. Das Prädikat v_1 $isLast$ bestimmt, ob der Knoten v_1 der letzte Knoten in B_T ist. Es folgt:

$$v_1 \; isLast \iff GetChild(T, Parent(T, v_1), v_1, k) = N_0$$

Analoges gilt für u_1 $isLast$. Ist v_1 der letzte Knoten, dann wird überprüft, ob $h = sgt$ gilt. Falls dies gilt, dann ist der Knoten v_1, der Knoten mit dem gesuchten Ordnungsprofil und es folgt die Operation FINDT. Falls $h \neq sgt$ und u_1 $isLast$, dann wird die Kante zu $\succ$ genommen (es gibt mindestens einen Knoten in $B_T(k)$ der einen kleineren Isomorphietyp aufweist als jeder andere Knoten in $B_S(k)$, daher $T[v] \prec S[u]$). Ansonsten wird die Kante zu NCS genommen.

Im Folgenden wird der rechte Teilbaum von RET2 betrachtet. Die Operation NXTT folgt auf RET2, falls $f = T$ und u_1 $isLast$ nicht erfüllt ist. Ist u_1 der letzte Knoten in B_S, dann wird überprüft, ob $h = tgt$ gilt. Gilt dies nicht, dann wird die Kante zu NCT genommen, falls v_1 nicht der letzte Knoten in B_T ist. Ansonsten ist die nächste Operation $\prec$ (analog zu u_1 $isLast \rightarrow \succ$). Falls $tgt = h$ und $seq = teq$, dann stimmen die Ordnungsprofile überein und es folgt die Operation INCH. Falls die Ordnungsprofile nicht übereinstimmen, dann wird überprüft, ob $seq < teq$ erfüllt ist. Ist dies erfüllt, dann ist die nächste Operation $\succ$ und $\prec$ sonst.

5 Visualisierung

Die Grundlage für die Visualisierung von Lindells Algorithms bilden die Trace-Tabellen. Da eine Zeile in einem Trace einen Programmzustand darstellt, müssen die Informationen aus einer Zeile extrahiert, verarbeitet und anschließend visualisiert werden. Daher genügt es bei der Visualisierung der Trace-Tabellen jede Zeile durchzugehen und dann den Programmzustand darzustellen. Das Ziel der Visualisierung wird sein, die zuvor erarbeiteten Komponenten des Algorithmus so gut wie möglich auf einem Webbrowser wiederzugeben. Folgende Abbildung 5.1 stellt die einzelnen Komponenten visuell dar.

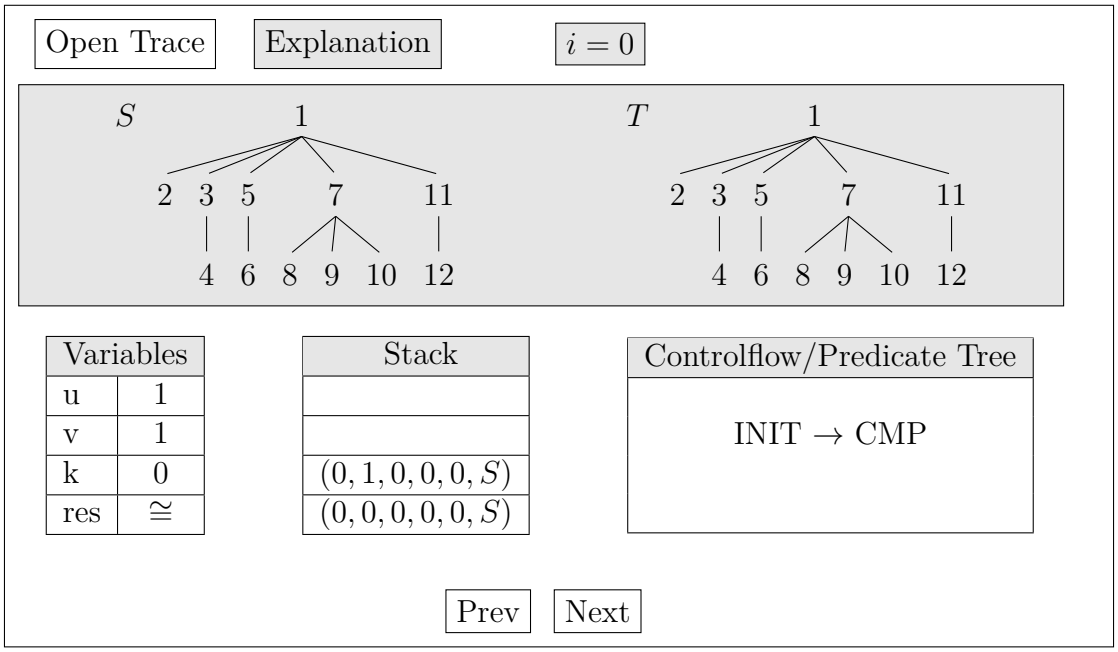


Abbildung 5.1: Mockup

Zur Übersicht werden die komplexen Datentypen von den primitiven Datentypen getrennt. Deshalb erhalten die Bäume und der Stack ihre eigene Darstellung. Die restlichen Variablen werden in der Tabelle *Variables* gespeichert. Aus Platzgründen kann entweder nur der Kontrollflussgraph oder nur der Prädikatenbaum angezeigt werden. Mittels eines Mausklicks kann zwischen diesen beiden Komponenten gewechselt werden. Nach Betätigen eines Buttons *Prev* oder *Next* wird der vorherige bzw. der nächste Zustand angezeigt. Der Zähler i kennzeichnet die Zeilennummer der Trace-Tabelle. Weiterhin gibt es eine kurze Erklärung zum aktuellen Programmzustand. Die vollständige Trace-Tabelle

kann mittels des Buttons *Open Trace* angezeigt werden. Falls dieser Button betätigt wurde, gibt es die Möglichkeit zu einem beliebigen Programmzustand überzugehen, Traces von Unterprogrammen zu öffnen, sowie zum vorherigen Trace zurückzukehren.

Die Visualisierung ist in zwei Teilen aufgeteilt. Der erste Teil behandelt die Generierung der Traces, während der zweite Teil die Visualisierung der Traces thematisiert.

5.1 Traces generieren

Wie zuvor erwähnt, erfolgt die Generierung der Traces mittels des Frameworks SP, welche eine SP-Datei erwartet und dessen Logik in Haskellcode übersetzt. Eine SP-Datei beinhaltet die Komponenten eines Kontrollflussgraphen (Variablen, Prädikate, Operationen, Prädikatenbäume). Da bereits der Kontrollflussgraph erarbeitet wurde, ist die Darstellung vom Lindells Algorithmus als SP-Programm nicht mehr schwierig. Nachdem das SP-Programm übersetzt wurde, wird es anhand einer Eingabe ausgeführt, wobei die Ausgabe in eine TR-Datei umgeleitet wird. Anhand der TR-Datei werden CSV-Dateien erstellt. In diesem Zusammenhang ist zu erwähnen, dass die Generierung der Traces zusätzlich alle Unterprogramme (wie z.B. $NextBlock$, $GetChild$) umfasst, welche in SP geschrieben wurden. Deswegen gibt es zu jedem Programmaufruf einen separaten Trace. Jedoch wurde der Quellcode von SP modifiziert, sodass keine CSV-Dateien mehr erstellt und stattdessen die Inhalte der CSV-Dateien in eine Javascript-Datei geschrieben werden. Dies hat den Vorteil, dass die CSV-Dateien nicht mehr eingelesen werden müssen (war zuvor nur mittels eines HTTP-Servers möglich). Darüber hinaus ist es möglich mittels SP LT-Dateien (*Labeled Tree*) zu generieren, welche ein SP-Programm als Baum darstellt. Diese LT-Dateien werden ebenfalls in eine Javascript-Datei geschrieben und werden verwendet, um bei der Visualisierung den Inhalt einer Operation und den zugehörigen Prädikatenbaum wiederzugeben.

Infolgedessen muss ein Haskell-Programm geschrieben werden, welches zwei Bäume als Textdatei erwartet und anhand dieser Bäume die Traces generiert. Die eingelesenen Bäume bestehen aus aufeinanderfolgenden Tupeln $(Parent, Children)$, wobei der Parent-Eintrag des ersten Tupels, die Wurzel kennzeichnet. Beispielsweise stellt die Tupelfolge "(1,[2,3])(2,[])(3,[])" den Baum mit Knoten 1 als Wurzelknoten und 2 und 3 als Kinderknoten dar. In Haskell ist die gewählte Baumstruktur ein Tupel (r, T), wobei r den Wurzelknoten und T eine Liste bestehend aus $(Parent, Children)$-Einträgen kennzeichnet. Der $Children$-Eintrag ist wiederum eine Liste aus Knoten. Im Folgenden wird der Ablauf in Abbildung 5.2 zum Algorithmus 5.1 betrachtet. In Zeile 2 wird die Funktion `split()` aufgerufen, welche den String s basierend auf dem Trennzeichen ")(" in Teilstrings unterteilt. Das Trennzeichen gibt die Position an, an denen der String aufgeteilt werden soll. Das Ergebnis ist eine Liste aus Strings. Die Funktion *map* ist eine Funktion höherer Ordnung (*Higher Order Function*) und führt eine Funktion auf jedes Listenelement aus. In Zeile 3 werden alle Klammern aus jedem Listeneintrag entfernt. Ein Listeneintrag besteht nun aus durch Komma getrennte Ziffern, wobei die erste Ziffer den Elternknoten und die restlichen Ziffern die Kinderknoten entsprechen; '∅' kennzeichnet den leeren String. Nachdem die Funktion TOTREETUPLE auf jedes Listenelement

ausgeführt wurde, liegt eine Liste aus $(parent, children)$-Einträgen vor. Anhand dieser Liste wird anschließend durch Extrahieren des Wurzelknotens der Baum *tree* konstruiert.

Algorithm 5.1 Parse string to tree

```
1: function ParseTupleTree(s)
2:     treeSplit ← split s ")("
3:     treeFilter ← map (filterParentheses) treeSplit
4:     treeList ← map (toTreeTuple) treeFilter
5:     tree ← Tree((first(treeList[0])), treeList)
6:     return tree
7: function toTreeTuple(s)
8:     list ← split s ", "
9:     parent ← Node(list[0])
10:    if list[1] ≠ ∅ then
11:        children ← [ Node(c) | c ∈ list \ list[0] ]
12:    else
13:        children ← [ ]
14:    return (parent, children)
```

Eingabe s	=	"(1, [2, 3])(2, [])(3, [])"
treeSplit	←	["(1, [2, 3]", "2, []", "3, [])"]
treeFilter	←	["1, 2, 3", "2, ∅", "4, ∅"]
treeTuple	←	[(1, [2, 3]), (2, []), (3, [])]
tree	←	(1, [(1, [2, 3]), (2, []), (3, [])])

Abbildung 5.2: Beispiel zum Algorithmus 5.1

5.2 Traces visualisieren

Die visuelle Darstellung findet in einem Webbrowser statt und erfolgt somit mittels HTML, CSS und Javascript. Da die Traces zunächst als String vorliegen, müssen diese in eine passende Datenstruktur geparst werden.

Algorithm 5.2 Parse trace-string to 2D-Array

```
1: function ParseTrace(csv)
2:     trace ← csv.split(' \ n')
3:     for i = 1 . . . |trace| do
4:         trace[i] ← trace[i].split(';')
5:     return trace
```

Dazu wird der String nach jedem Zeilenumbruch (' \ n') und anschließend aus der resultierenden Liste jedes Listenelement nach einem Semikolon aufgespalten. Die Traces

liegen nun als ein zweidimensionales Array vor. Anhandessen ist es nun möglich Daten zu extrahieren und anschließend diese Daten mittels HTML-Elemente auf der Webseite anzuzeigen. Es musst zunächst die Positonierung der HTML-Elemente spezifiziert werden. Dazu wird anhand des Mockups in Abbildung 5.1 das grundlegende Layout der Webseite erstellt. Die grau makierten Flächen werden als Zielelemente bezeichnet, welche mit einer ID versehen sind. Anhand der ID ist es möglich nach diesen Zielelementen zu suchen und anschließend HTML-Elemente anzuhängen. Es werden folgende Zielelemente verwendet:

- `<table>`: Wird verwendet, um die Variablen und die vollständige Trace-Tabelle auf der Webseite anzuzeigen. Ein `<table>`-Element besteht aus `<tr>`-Elementen (table row), die wiederum aus `<td>`-Elementen (table data) bestehen. Für jede Zeile im Trace wird ein `<tr>`-Element erstellt. Für jeden Tabelleneintrag in dieser Zeile wird ein `<td>`-Element erstellt, welches an dem `<tr>`-Element angehängt wird. Im Anschluss wird das `<tr>`-Element an das Zielelement `<table>` angehängt.

- `<ol>`: Stellt eine geordnete Liste dar und wird verwendet um den Stack darzustellen. Im Trace liegt der Wert von der Variable *stk* zunächst als String vor und muss als Nächstes in eine Liste geparst werden. Dazu wird die Methode `split()` mit einem Komma als Trennzeichen auf dem String ausgeführt. Aus der resultierenden Liste werden `<li>`-Elemente (list item) erstellt, die anschließend an das Zielelement `<ol>` angehängt werden.

- `<p>`: Ermöglicht es Text auf der Webseite anzuzeigen und wird für die Visualisierung des Zählers und der Erklärung verwendet. Dabei genügt es den Wert des `<p>`-Elements zu aktualisieren.

- `<div>`: Dieses HTML-Element hat keine semantische Bedeutung und wird als Platzhalter für die Darstellung der Bäume S, T, Prädikatenbäume und des Kontrollflussgraphen verwendet.

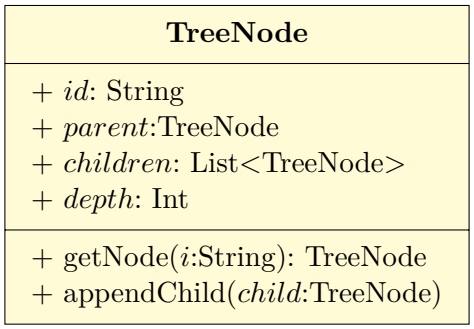


Abbildung 5.3: Klasse TreeNode

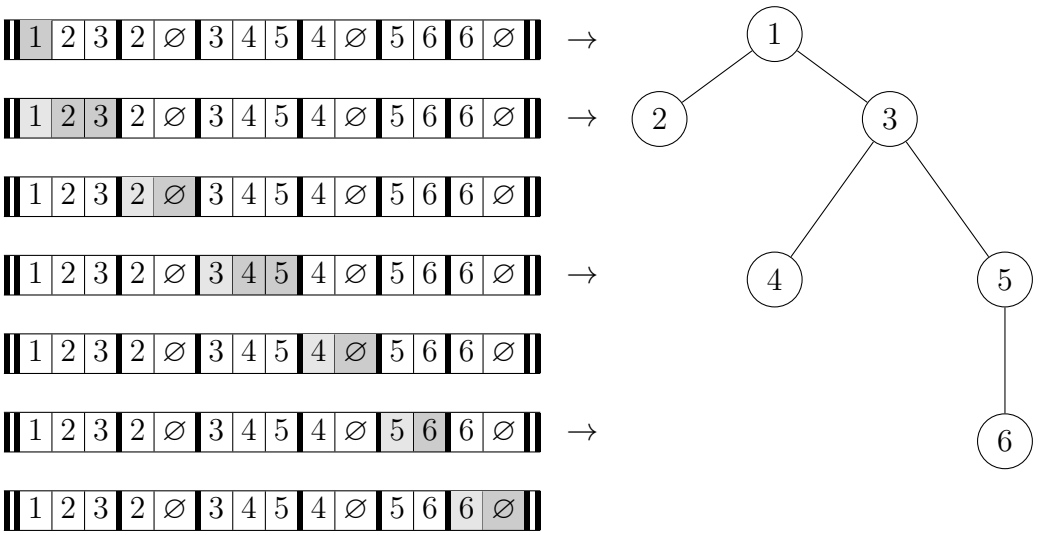
Bevor die Bäume gezeichnet werden können, müssen die Bäume, die zunächst als String vorliegen, in eine Baumstruktur geparst werden. Es wurde die Klasse TreeNode geschrieben. Eine TreeNode hat einen Verweis auf den Eltern- und Kinderknoten, sowie

als Attribut eine *id* und die Tiefe *depth*. Die Methode `getNode(`i`)` gibt den Knoten mit der $id = i$ zurück. Mittels der Methode `appendChild(`*child*`)` werden weitere Knoten zur Liste *children* hinzugefügt. Zunächst werden die Bäume S, T, welche als Tupelfolge "[(parent,childen)]" vorliegen, in eine Baumstruktur geparst.

Algorithm 5.3 Parse TupleTree to TreeNode

```
1:  function ParseTupleTree(s)
2:      s ← splitAndFilter(s)
3:      root ← TreeNode(s[0][0], 0)
4:      for i = 1 ... |s[0]| do
5:          root.appendChild(TreeNode(s[0][i], 1))
6:      for i = 1 ... |s| do
7:          parent ← root.getNode(s[i][0])
8:          for j = 1 ... |s[i]| do
9:              child ← s[i][j]
10:             if child ≠ ∅ then
11:                 parent.appendChild(TreeNode(child, parent.depth + 1))
12:     return root
13: function splitAndFilter(s)
14:     s ← s.split(''), ('')
15:     for i = 1 ... |s| do
16:         s[i] ← filterParentheses(s[i])
17:         s[i] ← s[i].split(', ')
18:     return s
```

Der Eingabestring s wird in der Funktion splitAndFilter in eine Form umgewandelt, sodass die Ziffern in einem zweidimensionalen Array angeorndet sind. In Zeile 3 wird der Wurzelknoten erstellt. Anschließend werden in Zeile 4 und 5 seine Kinderknoten angehängt. Diese Kinderknoten enstsprechen jeweils einem Elterknoten in der darauffolgenden Iteration. Dementsprechend wird in Zeile 7 ein Elternknoten aus dem bisherigen Baum extrahiert und dessen Liste *children* in Zeile 11 um weitere Knoten *child* erweitert, falls der Eintrag $s[i][j]$ nicht leer war. Die Iterationen beginnen mit Index $i = 1$, da jeweils der erste Eintrag mit dem Wurzelknoten bzw. Elternknoten belegt ist. Folgende Abbildung 5.4 stellt den Ablauf vom Algorithmus 5.3 dar.



Eingabe: s = [(1,[2,3]),(2,[ ]),(3,[4,5]),(4,[ ]),(5,[6]),(6,[ ])]

Abbildung 5.4: Beispiel zum Algorithmus 5.3

Der Inhalt einer LT-Datei ist wie folgt strukturiert. Die Knoten sind Zeilenweise angeordnet, wobei die Anzahl der Einrückungen die Tiefe eines Knotens entsprechen. In der Abbildung 5.5 kennzeichnen jeweils '- -' (im Algorithmus '\t') eine Einrückung.

```
1
- - 2
- - - - 3
- - - - 4
- - 5
```

Abbildung 5.5: Struktur einer LT-Datei

Algorithm 5.4 Parse LabeledTree to TreeNode

```
1:  function ParseLabeledTree(s)
2:      s ← s.split(" \ n")
3:      root ← TreeNode(s[0], 0)
4:      prev ← root
5:      for i = 1 . . . |s| do
6:          node ← s[i].split(" \ t"))
7:          depth ← |node| − 1
8:          curr = TreeNode(node[depth], depth)
9:          if curr.depth > prev.depth then
10:             prev.appendChild(curr)
11:         else if curr.depth = prev.depth then
12:             prev.parent.appendChild(curr)
13:         else
14:             BackTrack(prev, curr)
15:         prev ← curr
16:     return root
17: function BackTrack(prev, curr)
18:     if prev.depth = curr.depth then
19:         prev.parent.appendChild(curr)
20:     else
21:         BackTrack(prev.parent, curr)
```

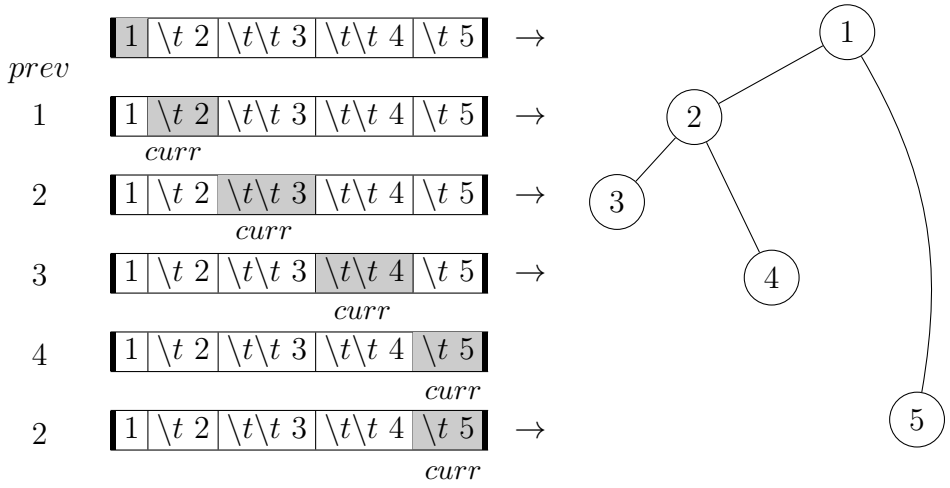


Abbildung 5.6: Beispiel Algorithmus 5.4 zum Baum in Abbildung 5.5

Folgender Abschnitt bezieht sich auf das Beispiel in Abbildung 5.6 zum Algorithmus 5.4. In Zeile 2 wird der String nach jedem Zeilenumbruch aufgespalten. Aus dem ersten Eintrag $s[0]$ der resultierenden Liste wird der Wurzelknoten erstellt. Anschließend wird

in einer Schleife der restliche Baum konstruiert. In Zeile 6 bis 8 wird ein neuer Knoten erstellt, indem der String an $s[i]$ nach jedem Tabulatorzeichen aufgespalten wird. Anhand dessen wird die Tiefe *depth* bestimmt, wobei *depth* sowohl die Indizierung der *id*, als auch die Tiefe des Knotens bestimmt. Für den Knoten *curr* muss nun der richtige Elternknoten bestimmt werden. Dabei wird zwischen drei Fällen unterschieden.

Fall 1: $curr.depth > prev.depth$
Dann ist der Knoten *prev* der Elternknoten von *curr*. Der Knoten *curr* wird an die Liste *children* von *prev* angehängt. In dem Beispiel tritt dieser Fall bei der ersten Iteration ein, da die Tiefe des Wurzelknotens $prev = 1$ kleiner ist als die vom Knoten $curr = 2$

Fall 2: $curr.depth = prev.depth$
Dann sind *prev* und *curr* Nachbarn. D.h. die Liste *children* wird bezüglich des Elterknotens von *prev* um den Knoten *curr* erweitert. Dieser Fall tritt bei der dritten Iteration ein, wobei der Knoten $prev = 3$ mit $curr = 4$ verglichen wird.

Fall 3: $curr.depth < prev.depth$
Dann muss der richtige Elternknoten zum Knoten *curr* bestimmt werden. Dies erfolgt, indem vom Knoten *prev* aus, der Baum bis zu einem Knoten mit $prev.depth = curr.depth$ zurückverfolgt wird. Dies geschieht in der rekursiven Funktion BACKTRACK. Dieser Fall tritt ein, nachdem $prev = 4$ mit $curr = 5$ verglichen wird.

Nachdem der Knoten *curr* an dem Baum angehängt wurde, wird in Zeile 15 *curr* auf *prev* gesetzt.

Alle Bäume liegen nun als Objekt vor und können visualisiert werden. Die visuelle Darstellung der Bäume wurde mithilfe der Bibliothek `d3.js` umgesetzt. Die Basis für das Zeichnen der Bäume bildet das `d3.hierarchy`-Objekt. Dieses Objekt dient als Wrapper-Objekt für hierarchische Datenstrukturen (z.B `TreeNode`), sodass weitere `d3.js`-Funktionen wie z.B `d3.tree()` auf diese Datenstruktur ausgeführt werden können. Die Funktion `d3.tree()` erwartet ein `d3.hierarchy`-Objekt und ordnet das Objekt (basierend auf die Höhe und Breite des Zielelements) in eine Baumstruktur an, indem jeder Knoten eine x- und y-Koordinate erhält. In Folgedessen wird anhand des `d3.hierarchy`-Objekts die Funktion `descendants()` ausgeführt, welche ein Array bestehend aus allen Knoten zurückgibt. Basierend auf der x- und y-Koordinate eines Knotens kann die ID als `<text>` und die Umrandung als `<circle>` in einem `<svg>`-Element (*Scalable Vector Graphics*) gespeichert werden. Für das Zeichnen der Kanten wird die Funktion `links()` ausgeführt. Diese Funktion gibt ein Array zurück, wobei jeder Eintrag aus einer Start- und Zielkoordinate besteht. Anhand dessen können die Kanten anhand dieser Koordinaten als `<line>` auf dem `<svg>`-Element gezeichnet werden.
Die Kontrollflussgraphen wurden mithilfe des Tools *draw.io* [1] erstellt. Dieses Tool ermöglicht es Graphiken zu erstellen, welche als SVG-Dateien exportiert werden. Der Inhalt einer SVG-Datei liegt bei der Visualisierung zunächst als String vor. Es genügt diesen String an einem `<svg>`-Element anzuhängen, um einen Kontrollflussgraphen auf der Webseite anzuzeigen. Anschließend ist ein Zugriff auf die einzelnen Elemente des `<svg>`-

[1] https://app.diagrams.net/

Elements möglich, sodass ein Rechteck im Kontrollflussgraphen je nach der aktuellen Operation farbig makiert werden kann.

Es ist nun möglich ein Trace zu visualisieren und es bleibt zu erklären wie weitere Traces von Unterprogrammen visualisiert werden können. Es wird die Beziehung zwischen den Klassen TraceManager und Trace in Abbildung 5.7 betrachtet.

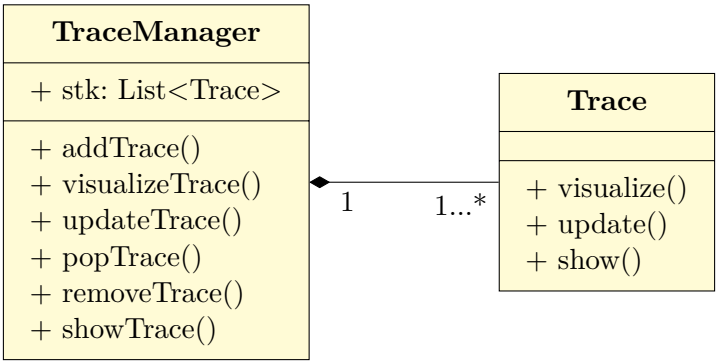


Abbildung 5.7: Veinfachtes Klassendiagramm TraceManager und Trace

Über die Klasse TraceManager werden Objekte vom Typ Trace verwaltet. Die Methode `addTrace()` hängt einen Trace an die Liste *stk* an. Anschließend wird die Methode `visualizeTrace()` aufgerufen, welches auf das letzte Element in *stk* `visualize()` aufruft. Die Methode `visualize()` stellt einen Trace auf der Webseite visuell dar. Die Methode `updateTrace()` korrespondiert zu den Button *prev* und *next* und führt auf das letzte Listenelement `update()` aus. Diese Methode aktualisiert die jeweiligen Werte der zuvor in `visualize()` generierten HTML-Elemente. Des Weiteren entfernt die Methode `popTrace()` die obersten k Listenelemente. Anschließend wird die Funktion `removeTrace()` ausgeführt, welche die Inhalte der Zielelemente entfernt. Dies ist wichtig, da ansonsten die HTML-Elemente vom vorherigen und aktuellen Trace auf der Webseite sich überlappen würden. Nachdem Trace-Objekte aus der Liste entfernt wurden, ist nun das letzte Listenelement ein Trace der bereits visualisiert wurde. D.h. die HTML-Elemente wurden bereits erstellt und müssen anschließend an die entsprechenden Zielelemente angehängt werden; dies erfolgt über die Methode `showTrace()` und `show()`.

Literaturverzeichnis

[Lin92] Steven Lindell. „Logspace algorithm for tree canonization“. In: *Conference Proceedings of the Annual ACM Symposium on Theory of Computing* (Jan. 1992).

[For96] Scott Fortin. „The Graph Isomorphism Problem“. In: 1996. URL: https://pdfs.semanticscholar.org/3da3/ef796c6af4bafe2c0d023fc49f9646f4d949.pdf.

[Val02] Gabriel Valiente. „Tree Isomorphism“. In: *Algorithms on Trees and Graphs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 151–251. ISBN: 978-3-662-04921-1. DOI: 10.1007/978-3-662-04921-1_4. URL: https://doi.org/10.1007/978-3-662-04921-1_4.

[Sma08] Alexander Smal. *Tree isomorphism*. 2008. URL: https://logic.pdmi.ras.ru/~smal/files/smal_jass08_slides.pdf.

[Har12] Sven Oliver Krumke und Hartmut Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Vieweg+Teubner Verlag, Wiesbaden, 2012.

[Cha18] Maurice Chandoo. *A Systematic Approach to Programming*. 2018. arXiv: 1808.08989 [cs.SE].

[Cha19] Maurice Chandoo. *Aufgabenblatt 5 - Lindell*. 2019.

[Coo19] Peter Cook. *Layouts*. 2019. URL: https://www.d3indepth.com/layouts/ (besucht am 17. 06. 2020).

[MDN19] MDN-Mitwirkenden. *Liste der HTML5-Elemente*. 2019. URL: https://developer.mozilla.org/de/docs/Web/HTML/HTML5/HTML5_element_list (besucht am 25. 06. 2020).

[MVS] A. Meier, H. Vollmer und U. Schöning. *Komplexität von Algorithmen*. Mathematik für Anwendungen. Lehmanns Media. ISBN: 9783965431423. URL: https://books.google.de/books?id=9DnvDwAAQBAJ.

Anhang

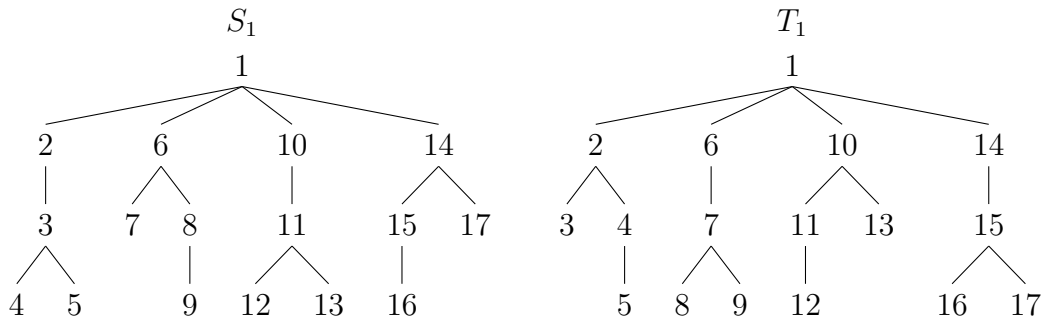


Abbildung 5.1: Beispielbäume Rekursion mit Ordnungsprofilen

Tabelle 5.1: Trace für S_1, T_1 von Abbildung 5.1 (angelehnt an [Cha18, S. 36–40])

	Operation	u	v	k	res	u_1	v_1	h	sgt	seq	tgt	teq	f	stk
1	INIT	1	1	0										[]
2	NB			4										
3	SETH							0						
4	FINDS					2	2		0	0	0	0	S	
5	PUSH	2	2	0										$[(0,0,0,0,0,S)]$
6	$\prec$				$\prec$									
7	RET2	1	1	4		2	2	0	0	0	0	0	S	[]
8	NXTS						6							
9	PUSH	2	6	0										$[(0,0,0,0,0,S)]$
10	NB			3										
11	GC	3	7	0										
12	NB			1										
13	SETH							0						
14	FINDS					4	8		0	0	0	0	S	
15	PUSH	4	8	0										$[(0,0,0,0,0,S),(0,0,0,0,0,S)]$
16	$\cong$				$\cong$									

Tabelle 5.1: Trace für S_1, T_1 von Abbildung 5.1 (angelehnt an [Cha18, S. 36–40])

	Operation	u	v	k	res	u_1	v_1	h	sgt	seq	tgt	teq	f	stk
17	RET2	3	7	1		4	8	0	0	1	0	0	S	$[(0,0,0,0,0,S)]$
18	NXTS						9							
19	PUSH	4	9	0										$[(0,0,0,0,0,S),(0,0,1,0,0,S)]$
20	$\cong$				$\cong$									
21	RET2	3	7	1		4	9	0	0	2	0	0	S	$[(0,0,0,0,0,S)]$
22	FINDT					4	8						T	
23	PUSH	4	8	0										$[(0,0,0,0,0,S),(0,0,2,0,0,T)]$
24	$\cong$				$\cong$									
25	RET2	3	7	1		4	8	0	0	2	0	1	T	$[(0,0,0,0,0,S)]$
26	NXTT					5								
27	PUSH	5	8	0										$[(0,0,0,0,0,S),(0,0,2,0,1,T)]$
28	$\cong$				$\cong$									
29	RET2	3	7	1		5	8	0	0	2	0	2	T	$[(0,0,0,0,0,S)]$
30	INCH							2						
31	$\cong$				$\cong$									
32	RET	2	6	3										
33	$\cong$				$\cong$									
34	RET2	1	1	4		2	6	0	0	1	0	0	S	[]
35	NXTS						10							
36	PUSH	2	10	0										$[(0,0,1,0,0,S)]$
37	$\prec$				$\prec$									
38	RET2	1	1	4		2	10	0	0	1	0	0	S	[]
39	NXTS						14							
40	PUSH	2	14	0										$[(0,0,1,0,0,S)]$
41	NB			3										
42	GC	3	15	0										
43	NB			1										
44	SETH							0						
45	FINDS					4	16		0	0	0	0	S	
46	PUSH	4	16	0										$[(0,0,1,0,0,S),(0,0,0,0,0,S)]$
47	$\cong$				$\cong$									
48	RET2	3	15	1		4	16	0	0	1	0	0	S	$[(0,0,1,0,0,S)]$

Tabelle 5.1: Trace für S_1, T_1 von Abbildung 5.1 (angelehnt an [Cha18, S. 36–40])

	Operation	u	v	k	res	u_1	v_1	h	sgt	seq	tgt	teq	f	stk
49	NXTS						17							
50	PUSH	4	17	0										$[(0,0,1,0,0,S),(0,0,1,0,0,S)]$
51	$\cong$				$\cong$									
52	RET2	3	15	1		4	17	0	0	2	0	0	S	$[(0,0,1,0,0,S)]$
53	FINDT					4	16						T	
54	PUSH	4	16	0										$[(0,0,1,0,0,S),(0,0,2,0,0,T)]$
55	$\cong$				$\cong$									
56	RET2	3	15	1				0	0	2	0	1	T	$[(0,0,1,0,0,S)]$
57	NXTT					5								
58	PUSH	5	16	0										$[(0,0,1,0,0,S),(0,0,2,0,1,T)]$
59	$\cong$				$\cong$									
60	RET2	3	15	1		5	16	0	0	2	0	2	T	$[(0,0,1,0,0,S)]$
61	INCH							2						
62	$\cong$				$\cong$									
63	RET	2	14	3										
64	$\cong$				$\cong$									
65	RET2	1	1	4		2	14	0	0	2	0	0	S	[]
66	FINDT					2	2						T	
67	PUSH	2	2	0										$[(0,0,2,0,0,T)]$
68	$\prec$				$\prec$									
69	RET2	1	1	4		2	2	0	0	2	1	0	T	[]
70	NXTT					6								
71	PUSH	6	2	0										$[(0,0,2,1,0,T)]$
	...													
73	$\cong$				$\cong$									
74	RET2	1	1	4		6	2	0	0	2	1	1	T	[]
75	NXTT					10								
76	PUSH	10	2	0										$[(0,0,2,1,1,T)]$
77	$\prec$				$\prec$									
78	RET2	1	1	4		10	2	0	0	2	2	1	T	[]
79	NXTT					14								
80	PUSH	14	2	0										$[(0,0,2,2,1,T)]$

Tabelle 5.1: Trace für S_1, T_1 von Abbildung 5.1 (angelehnt an [Cha18, S. 36–40])

	Operation	u	v	k	res	u_1	v_1	h	sgt	seq	tgt	teq	f	stk
	...													
82	$\cong$				$\cong$									
83	RET2	1	1	4		14	2	0	0	2	2	2	T	[]
84	NCT					2	6				0	0		
85	PUSH	2	6	0										$[(0,0,2,0,0,T)]$
	...													
87	$\cong$				$\cong$									
88	RET2	1	1	4		2	6	0	0	2	0	1	T	[]
89	NXTT					6								
90	PUSH	6	6	0										$[(0,0,2,0,1,T)]$
91	$\succ$				$\succ$									
92	RET2	1	1	4		6	6	0	0	2	0	1	T	[]
93	NXTT					10								
94	PUSH	10	6	0										$[(0,0,2,0,1,T)]$
	...													
96	$\cong$				$\cong$									
97	RET2	1	1	4		10	6	0	0	2	0	2	T	[]
98	NXTT					14								
99	PUSH	14	6	0										$[(0,0,2,0,2,T)]$
100	$\succ$				$\succ$									
101	RET2	1	1	4		14	6	0	0	2	0	2	T	[]
102	INCH							2						
103	FINDS					2	2		0	0	0	0	S	
104	PUSH	2	2	0										$[(2,0,0,0,0,S)]$
105	$\prec$				$\prec$									
106	RET2	1	1	4		2	2	2	0	0	0	0	S	[]
107	NXTS						6							
108	PUSH	2	6	0										$[(2,0,0,0,0,S)]$
	...													
110	$\cong$				$\cong$									
111	RET2	1	1	4		2	6	2	0	1	0	0	S	[]
112	NXTS						10							

Tabelle 5.1: Trace für S_1, T_1 von Abbildung 5.1 (angelehnt an [Cha18, S. 36–40])

	Operation	u	v	k	res	u_1	v_1	h	sgt	seq	tgt	teq	f	stk
113	PUSH	2	10	0										$[(2,0,1,0,0,S)]$
114	$\prec$				$\prec$									
115	RET2	1	1	4		2	10	2	0	1	0	0	S	[]
116	NXTS						14							
117	PUSH	2	14	0										$[(2,0,1,0,0,S)]$
	...													
119	$\cong$				$\cong$									
120	RET2	1	1	4		2	14	2	0	2	0	0	S	[]
121	NCS					6	2		0	0				
122	PUSH	6	2	0										$[(2,0,0,0,0,S)]$
	...													
124	$\cong$				$\cong$									
125	RET2	1	1	4		6	2	2	0	1	0	0	S	[]
126	NXTS						6							
127	PUSH	6	6	0										$[(2,0,1,0,0,S)]$
128	$\succ$				$\succ$									
129	RET2	1	1	4		6	6	2	1	1	0	0	S	[]
130	NXTS						10							
131	PUSH	6	10	0										$[(2,1,1,0,0,S)]$
	...													
133	$\cong$				$\cong$									
134	RET2	1	1	4		6	10	2	1	2	0	0	S	[]
135	NXTS						14							
136	PUSH	6	14	0										$[(2,1,2,0,0,S)]$
137	$\succ$				$\succ$									
138	RET2	1	1	4		6	14	2	2	2	0	0	S	[]
139	FINDT					2	2				0	0	T	
140	PUSH	2	2	0										$[(2,2,2,0,0,T)]$
141	$\prec$				$\prec$									
142	RET2	1	1	4		2	2	2	2	2	1	0	T	[]
143	NXTT					6								
144	PUSH	6	2	0										$[(2,2,2,1,0,T)]$

Tabelle 5.1: Trace für S_1, T_1 von Abbildung 5.1 (angelehnt an [Cha18, S. 36–40])

	Operation	u	v	k	res	u_1	v_1	h	sgt	seq	tgt	teq	f	stk
	...													
146	$\cong$				$\cong$									
147	RET2	1	1	4		6	2	2	2	2	1	1	T	[]
148	NXTT					10								
149	PUSH	10	2	0										$[(2, 2, 2, 1, 1, T)]$
150	$\prec$				$\prec$									
151	RET2	1	1	4		10	2	2	2	2	2	1	T	[]
152	NXTT					14								
153	PUSH	14	2	0										$[(2, 2, 2, 2, 1, T)]$
	...													
155	$\cong$				$\cong$									
156	RET2	1	1	4		14	2	2	2	2	2	2	T	[]
157	INCH							4						
158	$\cong$				$\cong$									

Tabelle 5.2: Operationen

Operation	Verallgemeinerter Ausdruck
INIT	$u = Root(S)$ $v = Root(T)$ $k = 0$ $stk = []$
NB	$k = NextBlock(S, u, k)$
GC	$u = GetChild(S, u, N_0, k)$ $v = GetChild(T, v, N_0, k)$ $k = 0$
RET	$u = Parent(S, u)$ $v = Parent(T, v)$ $k = \|S[u]\|$
$\cong$	$res = \cong$
$\prec$	$res = \prec$
$\succ$	$res = \succ$
SETH	$h = 0$
INCH	$h = h + seq$
FINDS	$u_1 = GetChild(S, u, N_0, k)$ $v_1 = GetChild(T, v, N_0, k)$ $sgt = 0$ $seq = 0$ $tgt = 0$ $teq = 0$ $f = S$
FINDT	$u_1 = GetChild(S, u, N_0, k)$ $v_1 = GetChild(T, v, N_0, k)$ $f = T$
NXTS	$v_1 = GetChild(T, v, v_1, k)$
NXTT	$u_1 = GetChild(S, u, u_1, k)$
NCS	$u_1 = GetChild(S, u, u_1, k)$ $v_1 = GetChild(T, v, N_0, k)$ $sgt = 0$ $seq = 0$
NCT	$u_1 = GetChild(S, u, N_0, k)$ $v_1 = GetChild(T, v, v_1, k)$ $tgt = 0$ $teq = 0$
PUSH	$u = u_1$ $v = v_1$ $k = 0$ $stk = stk.push((h, sgt, seq, tgt, teq, f))$
RET2	$u = Parent(S, u)$ $v = Parent(T, v)$ $k = \|S[u]\|$ $u_1 = u$ $v_1 = v$ $h = stk_{last}[h]$ $sgt = stk_{last}[sgt] + x_{sgt}$ $seq = stk_{last}[seq] + x_{seq}$ $tgt = stk_{last}[tgt] + x_{tgt}$ $teq = stk_{last}[teq] + x_{teq}$ $f = stk_{last}[f]$ $stk = stk.pop()$

Tabelle 5.3: Adjazenzmatrix (angelehnt an [Cha18, S. 41])

	INIT	GC	PUSH	NB	$\prec$	$\cong$	$\succ$	RET	RET2	SETH	FINDS	FINDT	NXTS	NXTT	NCS	NCT	INCH
INIT				1	*	*	*										
GC				11	*	*	*										
PUSH				9	5	15	90										
NB		10								2							
$\prec$								*	6								
$\cong$								31	16								
$\succ$								*	91								
RET				*	*	32	*										
RET2					*		*					21	7	25	120	83	29
SETH											3						
FINDS			4														
FINDT			22														
NXTS			8														
NXTT			26														
NCS			121														
NCT			84														
INCH				*		30					102						

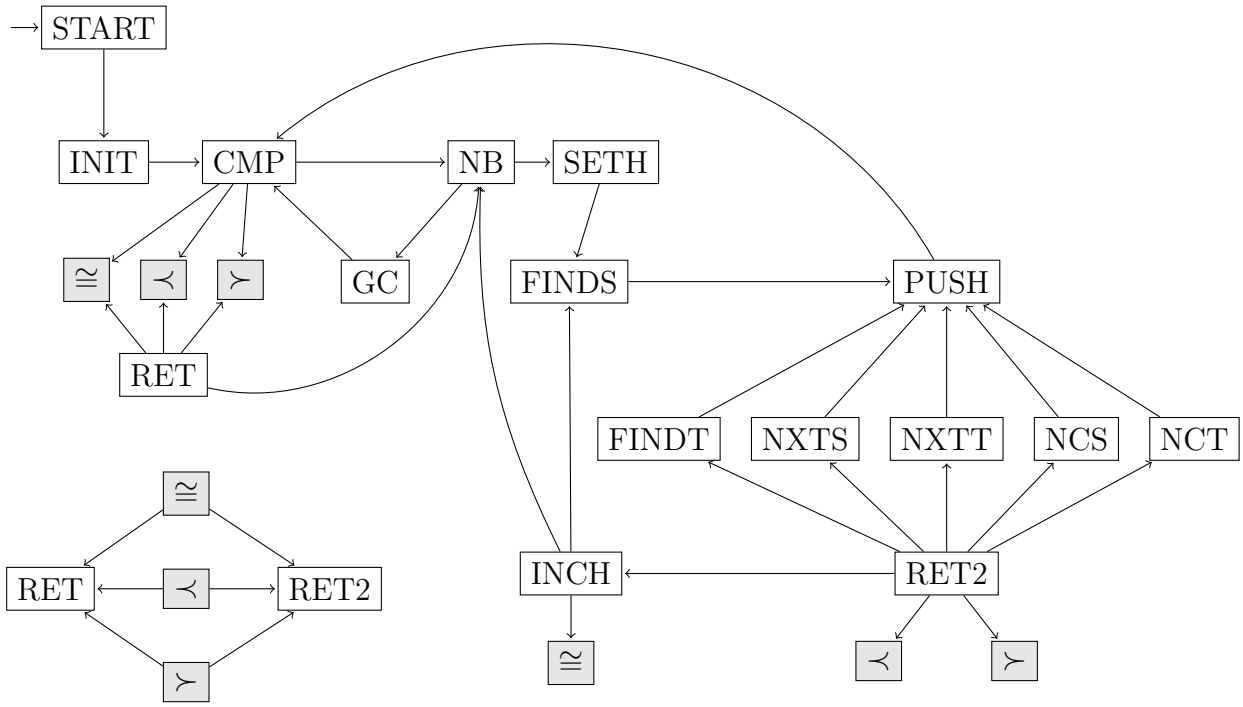


Abbildung 5.2: Kontrollflussgraph [Cha18, S.27]

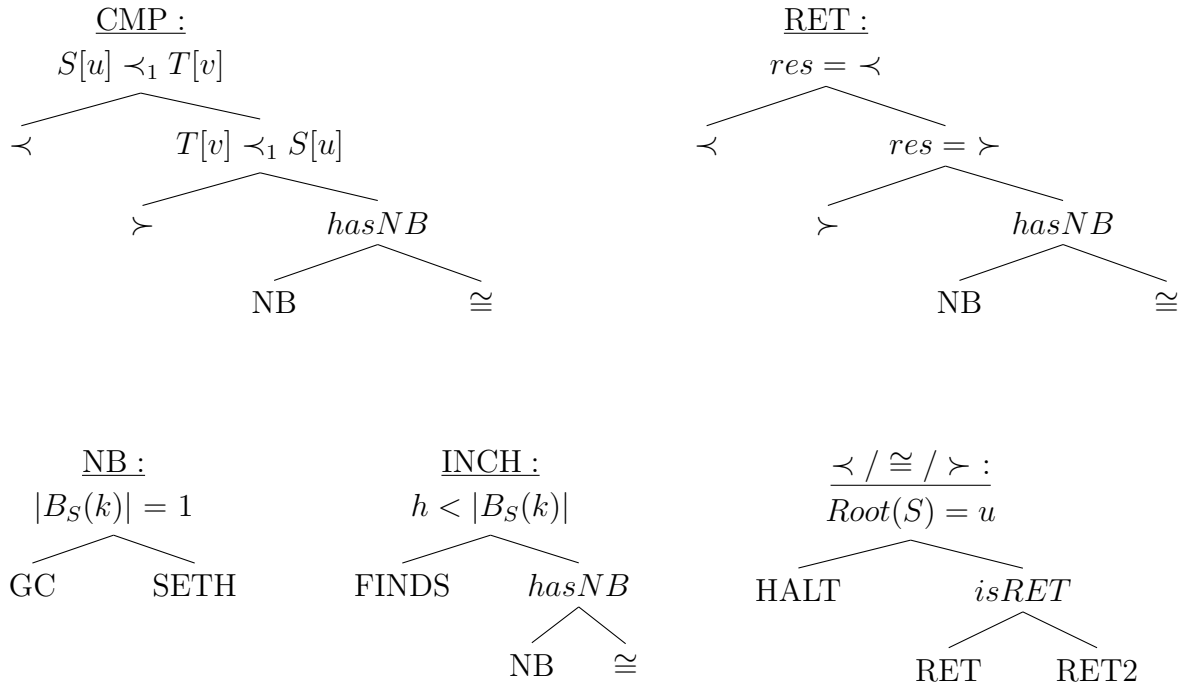


Abbildung 5.3: Prädikatenbäume

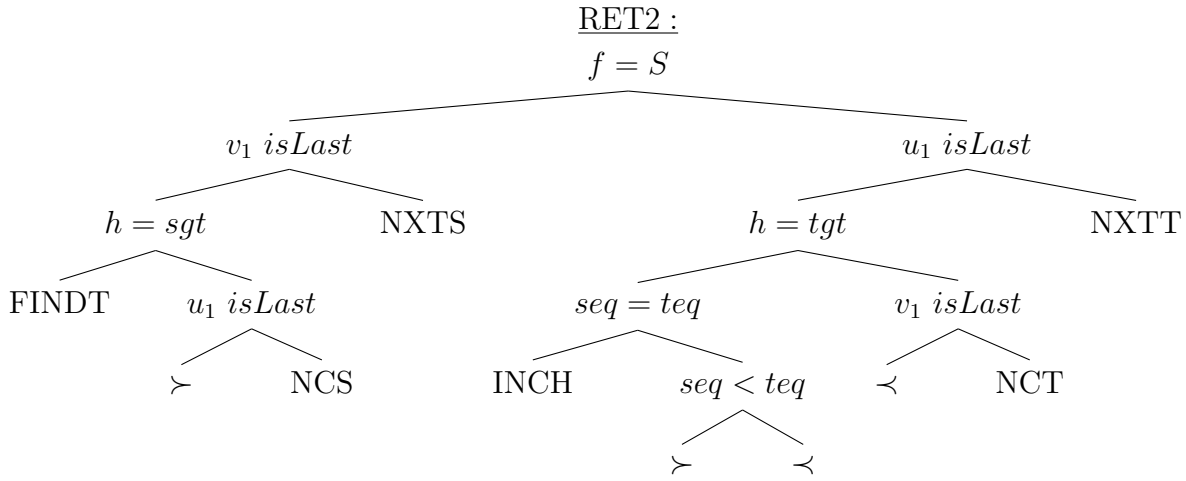


Abbildung 5.4: Prädikatenbaum zu RET2 [Cha18, S.27]