LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

Baumsuche, Partielle Ordnungen und eine neue Familie uninformierter Algorithmen

Simon Matthias Brodt

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	05.2008
Abgabe der Arbeit:	02.2009
Betreuer:	Prof. Dr. François Bry - Dr. Norbert Eisinger

Zusammenfassung

Erstens wird ein Formalismus zur Untersuchung (uninformierter) *Suchverfahren* entwickelt. Dieser stellt einen Zusammenhang zwischen den Suchverfahren und *partiellen Ordnungen* her. Dadurch ermöglicht er die Charakterisierung und einfachere Prüfung der Vollständigkeit eines Suchalgorithmus. Zudem erleichtert er die Formulierung und den Beweis weiterer Eigenschaften eines Verfahrens.

Zweitens wird ein *neuer uninformierter Suchalgorithmus* vorgestellt, der vollständig und speichereffizient ist und ohne Mehrfachexpansionen auskommt. Der Algorithmus wird mit Hilfe des zuvor entwickelten Formalismus analysiert.

Schließlich werden Vorteile des neuen Verfahrens für die *Implementierung logischer Programmiersprachen* angesprochen. Dies betrifft insbesondere die Verarbeitung (nahezu) endrekursiver Programme und die Definition einer möglichst deklarativen Semantik.

Abstract

First a new formalism for the analysis of (uninformed) *search methods* is developed. It connects search methods with *partial orderings*. In this way a characterization of the completeness of a search method and easier completeness checks become possible. Moreover it simplifies the formulation and the proof of further features.

Second a *new uninformed search algorithm* is presented. It is complete and memory efficient and never re-expands nodes. The algorithm is analysed using the previously developed formalism.

Finally advantages of the new method for the *implementation of logic programming languages* are briefly discussed. Especially the efficient processing of (almost) tail recursive progams and the definition of declarative semantics are addressed.

Inhaltsverzeichnis

1. Einleitung **5**
- 1.1. Informierte versus uninformierte Algorithmen . . . 5
  - 1.1.1. Die Unterschiede . . . 5
  - 1.1.2. Der Nutzen uninformierter Algorithmen . . . 6
- 1.2. Bekannte uninformierte Verfahren und ihre Schwächen . . . 6
- 1.3. Uninformierte Baumsuche und Traversierungsalgorithmen . . . 7
- 1.4. Aufbau der Arbeit . . . 7

2. Ein Formalismus zur Untersuchung von Traversierungsalgorithmen **8**
- 2.1. Repräsentation von Bäumen . . . 8
- 2.2. Was ist ein Algorithmus? . . . 10
- 2.3. Traversierungsordnungen . . . 18
- 2.4. Der Speicher . . . 25

3. Bekannte Algorithmen im neuen Formalismus **26**
- 3.1. Tiefendurchlauf . . . 26
  - 3.1.1. Darstellung . . . 26
  - 3.1.2. Unvollständigkeit . . . 26
- 3.2. Breitendurchlauf . . . 27
  - 3.2.1. Darstellung . . . 27
  - 3.2.2. Vollständigkeit . . . 28
- 3.3. Iterativer Tiefendurchlauf . . . 29
  - 3.3.1. Darstellung . . . 29
  - 3.3.2. Vollständigkeit . . . 29

4. Der D&B-Durchlauf und seine Familie **31**
- 4.1. Intuition . . . 31
- 4.2. Die Familie . . . 34
- 4.3. Formell . . . 35
  - 4.3.1. Vollständigkeit . . . 37
  - 4.3.2. Speicherkomplexität . . . 42
  - 4.3.3. Speicherkomplexität der Familie . . . 45

5. Implementierung **46**
- 5.1. Ansatz 1, Tiefenschranken . . . 47
- 5.2. Ansatz 2, Credit-Funktion . . . 49

6. Zusammenfassung, Anwendungen und weitere Ideen **51**
- 6.1. Der D&B-Durchlauf im Überblick . . . 51
- 6.2. Verbesserungen . . . 53
- 6.3. Der D&I-Durchlauf . . . 53
- 6.4. Anwendung in der Logikprogrammierung . . . 54

A. Credit-Funktion **57**

B. Notation **58**

Literatur **59**

1. Einleitung

Baumsuchalgorithmen sind in der Informatik von zentraler Bedeutung und dementsprechend häufig untersucht worden.

Ganz grundlegend lassen diese sich in zwei Kategorien einordnen, die der informierten und die der uninformierten Algorithmen. Erstere sind auf konkrete Problemklassen hin optimiert und damit (im entsprechenden Kontext) effizienter. Aus diesem Grund hat sich die Forschung auf diesen Algorithmustyp konzentriert.

In manchen Anwendungsfällen, wie z.B. der Auswertung logischer Programmiersprachen ist der universelle Charakter der uninformierten Verfahren jedoch unverzichtbar. Die bekannten Algorithmen haben allerdings Probleme mit Unvollständigkeit, hohem Speicherbedarf oder Mehrfachexpansion.

Die vorliegende Arbeit verfolgt deshalb zwei Ziele. Erstens wird ein Formalismus entwickelt, der die Beschreibung und Analyse von uninformierten Baumsuchalgorithmen erleichtert. Zweitens wird eine neue Familie uninformierter Baumsuchalgorithmen vorgestellt, die keine der angeführten Schwächen hat.

1.1. Informierte versus uninformierte Algorithmen

1.1.1. Die Unterschiede

Der Unterschied zwischen informierten und uninformierten Suchalgorithmen besteht, wie der Name schon nahelegt, darin, dass den ersteren zusätzliches Wissen, sogenannte Heuristiken, über den zu untersuchenden Raum zur Verfügung steht, wohingegen zweitere keinerlei Wissen dieser Art besitzen.

Informierte Algorithmen haben durch dieses Wissen die Möglichkeit verschiedene Fortsetzungsmöglichkeiten qualitativ zu bewerten und sich für die vielversprechendste zu entscheiden. Unter Umständen kann ein solcher Algorithmus mit Hilfe dieser Bewertungen sogar große Teile des Suchraumes im Vorhinein ausschließen und damit ununtersucht lassen.

Uninformierte Algorithmen haben keine solche Möglichkeit. Deshalb müssen sie in derselben Situation eine mehr oder weniger willkürliche Entscheidung treffen.

Diese Tatsache wirkt sich natürlich auf die Effizienz aus. Uninformierte Algorithmen treffen häufiger ungünstige bzw. falsche Entscheidungen und gelangen so meist nur über Umwege zum gewünschten Ergebnis.

Warum werden uninformierte Algorithmen dann überhaupt eingesetzt?

1.1.2. Der Nutzen uninformierter Algorithmen

Kurzgesagt ist der größte Nachteil der uninformierten Verfahren gleichzeitig auch ihr Vorteil. Sie besitzen zwar keine Informationen um die Auswertung zu optimieren, andererseits benötigen sie aber auch keine solchen Informationen. Dadurch sind sie universell einsetzbar, auch dann wenn keine entsprechenden Informationen zur Verfügung stehen oder gar nicht existieren.

Nehmen wir das Beispiel der Auswertung einer logischen Programmiersprache. Zwar mag es möglich sein für gewisse Programmklassen passende Heuristiken zu finden, die eine effizientere (weil informierte) Auswertung erlauben. Für allgemeine Programme ist es jedoch leicht einzusehen, dass die stattfindende Lösungssuche nicht im Vorhinein bestimmte Klauseln und damit Teile des Suchraums ignorieren darf, wenn sie alle möglichen Lösungen finden möchte. Man kann natürlich weiterhin versuchen z.B. Basisfälle zu bevorzugen um typische Programme zu beschleunigen. Aber dies bleibt eine willkürliche Entscheidung in dem Sinn, dass es auch „untypische“ Programme gibt, bzw. sogar geben muss, für die diese Entscheidung ungünstig ist.

Außerdem lassen sich uninformierte Verfahren häufig zu informierten Verfahren erweitern. So kann der A^* Algorithmus als eine Erweiterung der Breitensuche und der $IDA*$ Algorithmus als eine Erweiterung der Iterativen-Tiefensuche [Kor85] verstanden werden.

Es macht also Sinn sich mit uninformierten Baumsuchalgorithmen zu beschäftigen.

1.2. Bekannte uninformierte Verfahren und ihre Schwächen

Interessanterweise gibt es im Gegensatz zu den informierten Verfahren, nur vergleichsweise wenige uninformierte Algorithmen. Die bekanntesten uninformierten Verfahren sind wohl *Tiefensuche* und *Breitensuche*. Auch die *Iterative Tiefensuche* [Kor85] dürfte inzwischen allgemein geläufig sein. Weniger bekannt ist der in [GH90] vorgeschlagene *Iterative Broadening* Algorithmus, der im folgenden „*Iterative Ausweitung*“ genannt wird.

Auch *Depth-Bounded Backtrack Search* oder *Credit Search* [Bar04] sind uninformierte Verfahren. Allerdings sind sie nicht einmal auf endlichen Bäumen vollständig sind. Sie werden deshalb hier nicht näher betrachtet.

Die untenstehende Tabelle gibt einen Überblick über einige Eigenschaften[1] der Algorithmen. Unterstrichene Einträge markieren die ungünstigen Eigenschaften der Algorithmen.

Hierbei stehen d für die maximale bisher erreichte Tiefe, n für die Zahl der bis dahin ausgewerteten (unterschiedlichen) Knoten und b für den maximalen Verzweigungsgrad. Im endlichen Fall kann man d auch als maximale Tiefe bzw. n als die Gesamtzahl der Knoten lesen.

[1] Die Auswahl der Eigenschaten ist natürlich eingeschränkt.
Das führt zu einer zugegebenermaßen etwas einseitigen Betrachtungweise.
Z.B. ist der Algorithmus *Iterative Ausweitung* durchaus ein sehr interessanter Ansatz, auch wenn die Tabelle eher das Gegenteil suggeriert.

	Vollständigkeit		Speicherkomplexität		Mehrfachexpansion	Laufzeitkomplexität	
	endl.	unendl.	beste	schlechteste		beste	schlechteste
Tiefendurchlauf	ja	nein	$O(d)$	$O(d)$	nein	$O(n)$	$O(n)$
Iterative Ausweitung	ja	nein	$O(d)$	$O(d)$	ja	$O(n)$	$O(b*n)$
Breitendurchlauf	ja	ja	$O(d)$	$O(2^d)$	nein	$O(n)$	$O(n)$
Iterativer Tiefendurchlauf	ja	ja	$O(d)$	$O(d)$	ja	$O(n)$	$O(n^2)$ *

* Mit einer kleinen Modifikation der klassischen Iterativen Tiefensuche lässt sich hier $O(n \cdot log(n))$ erzielen.

1.3. Uninformierte Baumsuche und Traversierungsalgorithmen

Wie schon erwähnt verfügt ein uninformierter Baumsuchalgorithmus nicht über die Möglichkeit im Voraus Teile des Suchraums auszuschließen. Aus diesem Grund muss er genauso wie ein Baumtraversierungsalgorithmus jeden Knoten des Suchraums besuchen, wenn er gewährleisten soll, dass alle möglichen Lösungen gefunden werden. Da diese Eigenschaft alle Lösungen zu finden (Vollständigkeit genannt [2]) üblicherweise gewünscht ist, macht es Sinn statt uninformierter Suchalgorithmen einfach Traversierungsalgorithmen zu betrachten. Der Vorteil dieses Vorgehens liegt darin, dass man sich bei der Betrachtung nicht mehr mit der Frage beschäftigen muss ob ein bestimmter Knoten nun eine Lösung ist oder nicht. Das vereinfacht die Untersuchung.

1.4. Aufbau der Arbeit

Nach den vorangegangenen einleitenden Worten wird im 2. Kapitel der angekündigte Formalismus eingeführt. Das 3. Kapitel zeigt anhand von bekannten Algorithmen seine Verwendung und demonstriert hierbei bereits seinen Nutzen. Das folgende 4. Kapitel führt schließlich den neuen Algorithmus ein und analysiert ihn. Im Kapitel 5 wird dann genauer auf die Implementierung des neuen Algorithmus eingegangen. Das abschließende 6. Kapitel geht dann noch auf mögliche Anwendungen des Algorithmus, Varianten desselben und auf weitere denkbare Algorithmen ein.

[2] Beachte die Def. 2.2.9 vorangehende Diskussion

2. Ein Formalismus zur Untersuchung von Traversierungsalgorithmen

Zu Beginn dieses Kapitels stellt sich wahrscheinlich die Frage ob es wirklich nötig ist einen neuen Formalismus zur Untersuchung von Traversierungsalgorithmen zu entwickeln. Diese Frage ist natürlich berechtigt, da die momentan bekannten Algorithmen sich ja durchaus auch ohne einen solchen Formalismus analysieren lassen. Allerdings fällt beim Betrachten dieser Analysen auf, dass sie sehr textlastig sind. Nachdem es also bereits bei diesen einfachen Verfahren einigen Aufwands bedarf um sie ohne passenden Formalismus vernünftig zu analysieren, scheint es nicht ratsam zu sein, dies auch für etwas komplexere Verfahren zu versuchen.

Letztlich wird man feststellen, das der vorgeschlagene Formalismus insbesondere in Sachen Vollständigkeit sehr schnelle und knappe Analysen erlaubt. Dies ist unter anderem deshalb möglich, weil sich die Frage der Vollständigkeit in eine äquivalente Fragestellung auf Ordnungen, also auf ein sehr allgemeines und bekanntes Gebiet, überträgt.

Bevor wir jedoch mit der Formalisierung und Analyse von Traversierungsalgorithmen beginnen können müssen wir zunächst unsere Daten, die zu durchlaufenden Bäume, geeignet darstellen.

2.1. Repräsentation von Bäumen

Möchte man Bäume mit endlichem Verzweigungsgrad $b \geq 0$ repräsentieren, so gibt es dafür eine sehr einfache und praktische Möglichkeit. Die Idee ist, dass der Name eines Knoten gerade den Pfad von der Wurzel bis zu sich selbst codiert. Dazu nummeriert man die von einem Knoten ausgehenden Kanten mit $0, \ldots, b-1$ durch. Den Namen eines Knotens erhält man dann einfach dadurch, dass man die Nummern der auf dem Weg von der Wurzel zu diesem Knoten passierten Kanten aneinanderfügt. Der entstehende Name ist dann ein Wort aus $\{0, \ldots, b-1\}^*$.

Im Folgenden sei stets $\Sigma := \{0, \ldots, b-1\}$.

Damit ist Σ^* die Menge aller möglichen Knoten und damit wie man gleich feststellen wird gleichzeitig die Knotenmenge des vollständigen unendlichen Baumes mit Verzweigungsgrad b.

Um beliebige Bäume definieren zu können wählt die anschließende Definition zunächt „sinnvolle“ Knotenmengen $\Omega \subseteq \Sigma^*$ aus.

Definition 2.1.1 (Traversierbarkeit)**:**

Eine Menge $\Omega \subseteq \Sigma^*$ heißt *traversierbar*, falls für alle $uv \in \Omega$ gilt: $u \in \Omega$

Die Menge aller traversierbaren $\Omega \subseteq \Sigma^*$ wird mit $Trav_\Sigma$ bezeichnet.

„Sinnvoll“ ist eine Knotenmenge also dann, wenn zu jedem Knoten auch seine Vorgänger in Ω enthalten sind, und der Knoten somit über diese von der „Wurzel“ ε „erreichbar“ ist.

Jetzt also schließlich die Definition eines Baumes:

Definition 2.1.2 (Bäume)**:**

Sei $E := \{(w, wi) \mid w \in \Sigma^*, i \in \Sigma\}$ dann definiert (Σ^*, E) einen vollständigen unendlichen Baum mit Verzweigungsgrad b.

Einen beliebigen Baum mit maximalem Verzweigungsgrad b erhält man, indem man ein traversierbares $\Omega \subseteq \Sigma^*$ wählt und die Kantenmenge E entsprechend einschränkt. Der resultierende Baum ist dann:

$$(\Omega, E|_\Omega) = (\Omega, \{(w, w') \in E \mid w, w' \in \Omega\})$$

Statt $(\Omega, E|_\Omega)$ werden wir in Zukunft einfach nur Ω schreiben.

Bemerkung 2.1.3:

Ist $\Omega \subseteq \Sigma^*$ traversierbar und nicht leer, dann gilt

1. $\varepsilon \in \Omega$ und ε ist Wurzel des Baumes Ω
2. Die Tiefe d eines Knotens w in Ω entspricht gerade der Länge seines Namens, also $d = |w|$
3. Jedes Präfix w' eines Knotens $w = w'v \in \Omega$ ist ein Vorfahre von w. Insbesondere ist $w' \in \Omega$.

   Falls $|v| = 1$, so ist w' der Vater von w.
4. Für jeden Knoten $w = v_1 v_2 \dots v_k$ in Ω mit $k = |w|$ und $v_i \in \Sigma$ gilt:

   $\{\varepsilon, v_1, v_1 v_2, \dots, v_1 v_2 \dots v_k\}$ ist der Pfad in Ω nach w.

   Dies ist gerade die Menge der Präfixe von w und damit codiert w tatsächlich den Pfad von der Wurzel bis zu sich selbst.

Führen wir jetzt noch ein paar Abkürzungen für die üblichen Relationen auf den Knoten eines Baumes ein.

Notation 2.1.4:

Seien Ω traversierbar und $w \in \Omega$. Dann ist

- $\mathit{sons}_\Omega(w) := \{w' \in \Omega \mid w' = wv \text{ für ein } v \in \Sigma\}$

  die Menge der Söhne von w in Ω
- $\mathit{desc}_\Omega(w) := \{w' \in \Omega \mid w' = wv \text{ für ein } v \in \Sigma^+\}$

  die Menge der Nachfahren von w in Ω
- $\mathit{father}(w) := w'$,mit $w'v = w$ für ein $v \in \Sigma$

  der Vater von w für $w \neq \varepsilon$
- $\mathit{anc}(w) := \{w' \mid w'v = w \text{ für ein } v \in \Sigma^+\}$

  die Menge der Vorfahren von w

Immerwieder möchte man sich ein bestimmtes Level oder eine gewisse Auswahl von Leveln eines Baumes anschauen. Die folgende Schreibweise wird sich dabei als nützlich erweisen.

Notation 2.1.5:

- $\{\geq n\} := \{i \in \mathbb{N} \mid i \geq n\}$
- $\Omega_k := \{w \in \Omega \mid |w| = k\} = \Omega \cap \Sigma^k$
  $\Omega_N := \{w \in \Omega \mid |w| \in N\} = \bigcup_{i \in N} \Omega_i$ für $N \subseteq \mathbb{N}$
  $\Omega_{\geq k} := \Omega_{\{\geq k\}}$

Nachdem uns nun eine praktische Darstellung von Bäumen zur Verfügung steht wenden wir uns den Algorithmen zu.

2.2. Was ist ein Algorithmus?

Zunächst mal folgende wichtige Anmerkung die Notation betreffend:

Notation 2.2.1:

1. Im Folgenden wird immer wieder von einer Ordinalzahl α mit $\alpha \preccurlyeq \omega$ die Rede sein. Um die Ausführungen verstehen zu können sollte es jedoch nicht nötig sein, sich vorher mit Ordinalzahlen beschäftigt zu haben. Ein entsprechendes α steht einfach für eine Menge der Form $\{0, 1, \ldots, n\}$ oder $\emptyset$ im endlichen Fall ($\alpha \prec \omega$), bzw. für $\mathbb{N}$ im unendlichen Fall ($\alpha = \omega$), jeweils zusammen mit der üblichen (Wohl-)Ordnung auf natürlichen Zahlen. Die Notation wird aus zwei Gründen gewählt: Erstens lenkt sie die Aufmerksamkeit auf die „Reihenfolge“ der Zahlen und zweitens muss so bei der Notation nicht dauernd zwischen endlichem und unendlichem Fall unterschieden werden.
2. Werden in Zukunft Ω und α ohne nähere Angaben benutzt, so soll stets gelten:
   - $\Omega \subseteq \Sigma^*$ traversierbar
   - $\alpha \preccurlyeq \omega$ Ordinalzahl

Kommen wir nun zu den Traversierungsalgorithmen. Ein entscheidendes Merkmal eines Traversierungsalgorithmus ist sicherlich die Reihenfolge in der er die Knoten eines Baumes expandiert.

Dieses wird deshalb durch die gleich folgende Definition gefasst. Einige Abkürzungen sind dabei hilfreich.

Notation 2.2.2:

Sei $a : \alpha \to \Omega$ eine endliche bzw. unendliche Folge von Knoten in Ω.

1. Die Stelle des ersten Vorkommens eines Knotens w in a, wird mit $\mathit{first}_a(w)$ bezeichnet und lässt sich definieren als:

   $\mathit{first}_a(w) := \mathit{min}_< \left(a^{-1}[w]\right)$ für $w \in a[\alpha]$

2. Die Stelle des ersten Vorkommens eines Knotens w in a ab einer Stelle $n \in \alpha$, wird mit $\mathit{first}_a(w, n)$ bezeichnet und lässt sich definieren als:

   $\mathit{first}_a(w, n) := \mathit{min}_< \left(a^{-1}[w] \cap \{\geq n\}\right)$ für $w \in a[\alpha \cap \{\geq n\}]$

   (Es gilt: $\mathit{first}_a(w) = \mathit{first}_a(w, 0)$)

Definition 2.2.3 (Traversierungsfolge)**:**

Sei $a : \alpha \to \Omega$ eine endliche bzw. unendliche Folge von Knoten in Ω.

a heißt *Traversierungsfolge*, falls für alle in a vorkommenden w gilt, dass auch sein Vater in a vorkommt und sich das erste Vorkommen seines Vaters vor dem ersten Vorkommen von w befindet.

Formal:

$$\forall w \in a[\alpha] \setminus \{\epsilon\} : \underbrace{\mathit{father}(w) \in a[\alpha]}_{\mathit{father}(w) \text{ kommt in } a \text{ vor}} \wedge \mathit{first}_a(\mathit{father}(w)) < \mathit{first}_a(w)$$

Wie wir später sehen werden genügt obiges Merkmal bereits um über die Vollständigkeit eines Algorithmus entscheiden zu können.

Mit Hilfe von Traversierungsfolgen lässt sich allerdings nur wenig über ein anderes typisches Konzept von Algorithmen aussagen, das des Speichers.

Man kann davon ausgehen, dass ein Traversierungsalgorithmus den Speicher hauptsächlich dazu nutzt sich eine gewisse Menge von Knoten für die spätere Verarbeitung zu merken. Ein Speicherzustand eines Traversierungsalgorithmus lässt sich also durch eine Teilmenge der Knoten des Baumes beschreiben.

Definition 2.2.4 (Traversierungslauf und Traversierungsalgorithmus [3])**:**

- Ein *Traversierungslauf* ist eine Folge von Paaren aus einem Knoten und einer Knotenmenge $A : \alpha \to \Omega \times \mathcal{P}(\Omega)$ für die gilt:
  1. Die erste Knotenmenge enthält genau die Wurzel
  2. In jedem Paar muss der Knoten in der Knotenmenge enthalten sein
  3. In einem Schritt wird der Knoten des Paars expandiert und seine Kinder der Knotenmenge hinzugefügt. Beliebig viele schon vorher enthaltene Knoten können herausfallen.
  4. Ist ein Knoten w in einem bestimmten Schritt in der Knotenmenge enthalten, so war im vorhergehenden Schritt entweder der Knoten in der Knotenmenge enthalten oder sein Vater wurde expandiert.

  Formal:

  Für $\alpha = \emptyset$ ist nichts zu definieren.

  Ansonsten

  1. $A(0) = (\varepsilon, \{\varepsilon\})$
  2. $\forall n \in \alpha : (A)_1(n) \in (A)_2(n)$
  3. $\forall n \in \alpha : \mathit{sons}_\Omega((A)_1(n)) \subseteq (A)_2(n+1)$
  4. $\forall n \in \alpha \, \forall w \in (A)_2(n+1) : w \in (A)_2(n) \vee \mathit{father}(w) = (A)_1(n)$

  Hierbei meint $(A)_i$ die Funktion, die n auf die i-te Komponente von $A(n)$ abbildet.
- Ein *Traversierungsalgorithmus* ist eine Familie $(A_\Omega)_{\Omega \in \mathit{Trav}_\Sigma}$ von Traversierungsläufen. Anders ausgedrückt, ordnet der Algorithmus jedem Baum $\Omega \in \mathit{Trav}_\Sigma$ einen Traversierungslauf über diesen Baum zu.

Notation 2.2.5:

Im Folgenden schreiben wir häufig $A(n)$ statt $(A)_2(n)$, wenn klar ist, dass $A(n)$ im Sinne einer Menge gebraucht wird.

Einem Traversierungslauf lässt sich in natürlicher Weise eine Traversierungsfolge zuordnen.

Definition 2.2.6 (Induzierte Traversierungsfolge)**:**

Sei $A : \alpha \to \Omega \times \mathcal{P}(\Omega)$ ein Traversierungslauf. Die von A *induzierte Traversierungsfolge* $a_A : \alpha \to \Omega$ ist definiert durch:

$$a_A(n) = (A)_1(n)$$

[3] Üblicherweise würde man nur von einem *Algorithmus* sprechen, wenn die Folge der Speicherzustände *berechenbar* ist.
Für die nachfolgenden Betrachtungen ist eine solche Einschränkung jedoch unnötig und höchstens hinderlich.

Umgekehrt wäre es wünschenswert, wenn man auch zu jeder Traversierungsfolge einen passenden Lauf erhalten könnte. Eine sinnvolle (wenn auch nicht die einzige) Möglichkeit liefert diese Definition:

Definition 2.2.7 (Induzierter Traversierungslauf)**:**

Sei $a : \alpha \to \Omega$ eine Traversierungsfolge. Der von a *induzierte Traversierungslauf* $A_a : \alpha \to \Omega \times \mathcal{P}(\Omega)$ ist definiert durch:

$$\begin{aligned} A_a(0) &= (\varepsilon, \{\varepsilon\}) \\ A_a(n+1) &= (a(n+1), (A(n) \setminus D_n) \cup \mathit{sons}_\Omega(a(n))) \text{ falls } n+1 \in \alpha \end{aligned}$$

wobei die Menge D_n leer ist oder aus dem gerade expandierten Knoten besteht, falls dieser nie mehr expandiert wird oder erst wieder nachdem sein Vater ein weiteres Mal expandiert wurde. Also:

$$D_n := \begin{cases} \{a(n)\} & \text{falls} \quad \underbrace{a(n) \notin a[>n]}_{\text{wird nicht mehr expandiert}} \quad \text{oder} \\ & \mathit{father}(a(n)) \in a[>n] \text{ und} \\ & \mathit{first}_a(a(n), n+1) > \mathit{first}_a(\mathit{father}(a(n)), n+1) \\ \emptyset & \text{sonst} \end{cases}$$

Bemerkung 2.2.8:

- Wird ein Traversierungslauf von einer Traversierungsfolge induziert, so ist die von ihm induzierte Traversierungsfolge gleich der ursprünglichen Traversierungsfolge.
- Eine von einem Traversierungslauf induzierte Traversierungsfolge induziert nicht notwendigerweise den ursprünglichen Traversierungslauf.
- Wird ein Traversierungslauf von einer Traversierungsfolge ohne Mehrfachauswertung induziert, so wird in ihm ein Knoten nach seiner (ersten und einzigen) Expansion immer gelöscht. Der induzierte Algorithmus speichert dadurch zu jedem Zeitpunkt nur eine minimale Anzahl von Knoten die noch Vollständigkeit ermöglichen.

Eine Eigenschaft die uns bei einem Traverierungsalgorithmus besonders interessiert ist seine *Vollständigkeit*. Was ist darunter zu verstehen? Der Begriff der Vollständigkeit ist zunächst hauptsächlich im Zusammenhang mit Suchalgorithmen gebräuchlich und wird dort auf zweierlei Weisen gebraucht:

Ein Algorithmus heißt vollständig, wenn er

1. terminiert und eine Lösung liefert, falls eine solche existiert (siehe z.B. [Pea84] S.75)
2. jede Lösung nach endlicher Zeit liefert

Es ist klar, dass die zweite Möglichkeit die erste impliziert. Im endlichen Fall gilt in gewisser Weise auch die Umkehrung. Ist ein Algorithmus vollständig nach 1. so lässt er sich dann nämlich leicht zu einem vollständigen Algorithmus nach 2. umbauen, indem man nach erfolgreichem Finden einer Lösung den Algorithmus auf dem verbliebenen Suchraum erneut startet.

Im ganz allgemeinen Kontext von Traversierungsalgorithmen ist sozusagen jeder Knoten eine Lösung. Offensichtlich lässt sich also die erste Möglichkeit nicht sinnvoll verallgemeinern. Bei der zweiten Möglichkeit ist dies jedoch sehr einfach. Sie bedeutet für Traversierungsalgorithmen, dass jeder Knoten (jede „Lösung“) nach endlicher Zeit besucht wird.[4]

Die folgende Definition fasst den Vollständigkeitsbegriff formal:

Definition 2.2.9 (Vollständigkeit)**:**

- Eine Traversierungsfolge $a : \alpha \to \Omega$ heißt *vollständig*, falls sie surjektiv ist.
- Ein Traversierungslauf $A : \alpha \to \Omega \times \mathcal{P}(\Omega)$ heißt *vollständig*, falls die von ihm induzierte Traversierungsfolge vollständig ist.
- Ein Traversierungsalgorithmus heißt *vollständig* auf $T \subseteq \mathit{Trav}_\Sigma$, wenn seine Läufe A_Ω vollständig sind für alle $\Omega \in T$.
- Ein Traversierungsalgorithmus heißt *endlich vollständig*, falls er auf $\{\Omega \in \mathit{Trav}_\Sigma \mid |\Omega| < \infty\}$ vollständig ist.
- Ein Traversierungsalgorithmus heißt *unendlich vollständig*, falls er auf $\{\Omega \in \mathit{Trav}_\Sigma \mid |\Omega| = \infty\}$ vollständig ist.
- Ein Traversierungsalgorithmus heißt *vollständig*, falls er sowohl endlich als auch unendlich vollständig ist. D.h. er ist vollständig auf Trav_Σ im Sinn des dritten Punkts.

Bemerkung 2.2.10:

- Ein Traversierungslauf expandiert genau dann keinen Knoten mehrfach, wenn die von ihm induzierte Traversierungfolge injektiv ist.
- Ein vollständiger Traversierungslauf ohne Mehrfachauswertung induziert also eine bijektive Traversierungsfolge.
- Ist $A : \alpha \to \Omega \times \mathcal{P}(\Omega)$ ein vollständiger Traversierungslauf ohne Mehrfachauswertung, so kann ein Knoten w, frühestens nach seiner ersten (und einzigen) Expansion gelöscht werden. (Folgt aus der 2. bis 4. Bedingung in Def. 2.2.4).

[4] Da ein uninformierter Suchalgorithmus nicht über die Möglichkeit verfügt im Voraus Teile des Suchraums auszuschließen, fallen für ihn die 2. Definition für Suchalgorithmen und die für Traversierungsalgorithmen zusammen. Schließlich muss auch er jeden Knoten besuchen um Vollständigkeit nach 2. zu gewährleisten, da er ja nicht wissen kann ob ein nicht besuchter Knoten nicht doch eine Lösung ist.

Wie man sieht hängt die Vollständigkeit eines Traversierungsalgorithmus nur von seiner induzierten Traversierungsfolge ab. Wie schon erwähnt wollen wir letztlich ein noch abstrakteres Konzept zur Analyse der Vollständigkeit eines Algorithmus nutzen, das der partiellen Ordnungen.

Beim Herstellen eines Zusammenhangs zwischen Traversierungsfolgen und Ordnungen ist folgende Beobachtung entscheidend:

Man kann sich vorstellen, dass eine Traversierungsfolge eine Folge von Zeitpunkten durchläuft und zu jedem Zeitpunkt einen bestimmten Knoten expandiert. Es bietet sich also an, die Knoten nach dem Zeitpunkt ihrer ersten Expansion zu ordnen.

Dies führt uns zum Begriff einer *Darstellenden Ordnung.*

Definition 2.2.11:

Sei $\lhd$ eine partielle Ordnung von Ω.

1. $comp_{\lhd}(w, w') \;:\Leftrightarrow\; w \lhd w' \vee w = w' \vee w' \lhd w$, testet ob w und w' in $\lhd$ miteinander vergleichbar sind
2. $Comp_{\lhd}(w) := \{w' \in \Omega \mid comp_{\lhd}(w, w')\}$, ist die Menge aller in $\lhd$ mit w vergleichbarer Knoten.
3. Für $M, N \subseteq \Omega$ definiere:

   $M \lhd N \;:\Leftrightarrow\; \forall m \in M, n \in N : m \lhd n$

Definition 2.2.12 (Darstellende Ordnung)**:**

Eine partielle Ordnung $\lhd$ auf Ω heißt darstellende Ordnung einer Traversierungsfolge a, falls folgendes gilt:

- $\forall w, w' \in a\,[\alpha] : w \lhd w' \Leftrightarrow \mathit{first}_a(w) < \mathit{first}_a(w')$

  Alle in a vorkommenden Knoten werden nach dem Zeitpunkt ihrer ersten Expansion geordnet
- $a\,[\alpha] \lhd Comp_{\lhd}(\varepsilon) \setminus a\,[\alpha]$

Ist die darstellende Ordnung eindeutig, so wird sie mit $\lhd_a$ bezeichnet.

Bemerkung 2.2.13:

- Für vollständige Traversierungsfolgen ist die darstellende Ordnung eindeutig und der zweite Punkt der Definition hat keine Bedeutung. In diesem fall ist die darstellende Ordnung zudem total.
- Für unvollständige Traversierungsfolgen ist die darstellende Ordnung nicht eindeutig. Allerdings entsteht diese Nichteindeutigkeit ausschließlich durch unterschiedliche Anordnung der Elemente von $\Omega \setminus a\,[\alpha]$
- Verschiedene Algorithmen können die selbe, bzw. die selben darstellenden Ordnungen haben.
- Haben zwei nicht mehrfach auswertende Traversierungsfolgen die selben (Mengen von) darstellenden Ordnungen, so sind die Traversierungsfolgen gleich.

Letztendlich wollen wir Vollständigkeit ja mit Hilfe von darstellenden Ordnungen analysieren. Wir führen jetzt eine Voraussetzung an Traversierungsfolgen ein, die einerseits notwendig ist, wenn die Traversierungsfolge vollständig sein soll und andererseits für besonders schöne Eigenschaften der darstellenden Ordnungen sorgt. Zunächst scheint dies vielleicht eine vergleichsweise starke Voraussetzung zu seien, wir werden jedoch gleich nach ihrer Definition sehen, dass sich die Voraussetzung in vielen Fällen sehr leicht überprüfen lässt.

Definition 2.2.14 (Schwache Vollständigkeit)**:**

- Eine Traversierungsfolge $a : \alpha \to \Omega$ heißt *schwach vollständig*, falls $|a[\alpha]| = |\Omega|$.
- Ein Traversierungslauf $A : \alpha \to \Omega \times \mathcal{P}(\Omega)$ heißt *schwach vollständig*, falls die von ihm induzierte Traversierungsfolge schwach vollständig ist.
- Ein Traversierungsalgorithmus heißt *schwach vollständig* auf $T \subseteq \mathit{Trav}_\Sigma$, wenn seine Läufe A_Ω schwach vollständig sind für alle $\Omega \in T$.
- Ein Traversierungsalgorithmus heißt *endlich schwach vollständig*, falls er auf $\{\Omega \in \mathit{Trav}_\Sigma \mid |\Omega| < \infty\}$ schwach vollständig ist.
- Ein Traversierungsalgorithmus heißt *unendlich schwach vollständig*, falls er auf $\{\Omega \in \mathit{Trav}_\Sigma \mid |\Omega| = \infty\}$ schwach vollständig ist.
- Ein Traversierungsalgorithmus heißt *schwach vollständig*, falls er sowohl endlich als auch unendlich schwach vollständig ist. D.h. er ist schwach vollständig auf Trav_Σ im Sinn des dritten Punkts.

Lemma 2.2.15:

1. Vollständigkeit impliziert schwache Vollständigkeit
2. Schwache Vollständigkeit impliziert endliche Vollständigkeit
3. Für uninformierte Algorithmen sind schwache und endliche Vollständigkeit äquivalent

Beweis:

zu 1. : klar

zu 2. :

folgt aus dem Schubfachprinzip

zu 3. :

zusätzlich zu 2. ist zu zeigen ist, dass für uninformierte Algorithmen endliche Vollständigkeit schwache Vollständigkeit impliziert.

Dabei ist folgende Überlegung von zentraler Bedeutung: Ein uninformierter Algorithmus kann nach Auswertung endlich vieler Knoten niemals entscheiden, ob ein Baum endlich oder unendlich ist.

Sei also angenommen, dass der endlich vollständige Algorithmus auf einem unendlichen Baum Ω nicht schwach vollständig ist. Dann gilt offenbar für die induzierte Traversierungsfolge a_Ω, dass $|a_\Omega[\alpha]| < \infty$. Sei $m := max_<(\{|w| \mid w \in a_\Omega[\alpha]\})$, also $a_\Omega[\alpha] \subseteq \Omega_{\leq m}$. Da der Algorithmus uninformiert ist, kann er nicht zwischen Ω und $\Omega_{\leq m+1}$ unterscheiden, denn der Algorithmus expandiert keinen der Knoten aus Ω_{m+1} und oberhalb von Ω_{m+1}, also in $\Omega_{<m+1}$, sind die Bäume gleich. Also $a_\Omega[\alpha] = a_{\Omega_{\leq m+1}}[\alpha]$. Allerdings gilt, dass $\Omega_{m+1} \neq \emptyset$, da Ω unendlich aber endlich verzweigend ist. Allerdings expandiert a_Ω keinen der Knoten in Ω_{m+1} und $a_{\Omega_{\leq m+1}}$ damit auch nicht. Es wird also mindestens ein Knoten in $\Omega_{\leq m+1}$ nicht von $a_{\Omega_{\leq m+1}}$ besucht. Anders ausgedrückt gilt, $\left|a_{\Omega_{\leq m+1}}[\alpha]\right| < |\Omega_{m+1}| < \infty$. Damit wäre der Algorithmus aber nicht endlich vollständig. Ein Widerspruch.

□

Es stellt sich natürlich die Frage, ob schwache Vollständigkeit vielleicht schon genauso schwer zu prüfen ist wie Vollständigkeit selbst. Wie bereits angekündigt, ist dies jedoch nicht der Fall. Das anschließende Lemma gibt z.B. ein Kriterium, mit dem sich unter anderem die schwache Vollständigkeit von Tiefen- und Breitensuche unmittelbar zeigen lassen.

Lemma 2.2.16 (Kriterium schwache Vollständigkeit)**:**

Erfüllt ein Traversierungsalgorithmus die Bedingungen

1. Jeder Knoten wird höchstens einmal expandiert
2. Kein Knoten, der noch nicht expandiert wurde, wird gelöscht
3. Der Algorithmus stoppt nicht, solange es noch nicht expandierte Knoten (im Speicher) gibt

so ist er schwach vollständig.

Zudem stellt das Prüfen der schwachen Vollständigkeit höchstens ein Problem dar, wenn man sich von Algorithmen in Richtung Ordnungen bewegt. Häufig wird man jedoch umgekehrt vorgehen, zunächst eine Ordnung definieren, deren Vollständigkeit zeigen und dann zu ihrem induzierten Traversierungsalgorithmus übergehen. Dieser ist stets schwach vollständig (Bem. 2.3.10).

Der nächste Abschnitt zeigt jetzt, was für eine Eigenschaft eine darstellende Ordnung einer Traversierungsfolge haben muss, um deren Vollständigkeit zu garantieren.

2.3. Traversierungsordnungen

Es ist nur natürlich diese Eigenschaft einer Ordnung ebenfalls *Vollständigkeit* zu nennen. Mit Hilfe des Begriffs der *darstellenden Ordnung* und der (später definierten) *induzierten Traversierungsfolge* einer Ordnung wird dann in Satz 2.3.13 die Äquivalenz der beiden Definitionen der Vollständigkeit gezeigt.

Definition 2.3.1 (Vollständigkeit einer Ordnung)**:**

Eine partielle Ordnung $\lhd$ auf $\Omega \subseteq \Sigma^*$ heißt *vollständig*, gdw. $(\Omega, \lhd) \cong \alpha$ für eine Ordinalzahl α, $\alpha \preccurlyeq \omega$.

Die Idee dieser Definition besteht darin, dass falls $(\Omega, \lhd) \cong \alpha$ für eine Ordinalzahl $\alpha \preccurlyeq \omega$ ist, es kein $w \in \Omega$ gibt, das größer (bzgl. $\lhd$) ist als *unendlich* viele andere Elemente von Ω.

Die obige Definition bezieht sich noch auf beliebige partielle Ordnungen. Im Kontext von Traversierungsalgorithmen ist es jedoch sinnvoll die Auswahl der Ordnungen etwas einzuschränken. Analog zu der Einschränkung bei Traversierungsfolgen in Def. 2.2.3 soll ein Vorfahre eines Knotens niemals größer sein als der Knoten selbst.

Zudem, sollen zumindest die mit der Wurzel vergleichbaren Knoten schön, heißt linear, angeordnet sein. Dies ist später für die Definition der induzierten Traversierungsfolge (Def. 2.3.9) wichtig.

Definition 2.3.2 (Traversierungsordnung)**:**

Eine partielle Ordnung $\lhd$ auf traveriserbarem $\Omega \subseteq \Sigma^*$ heißt *Traversierungsordnung*, falls

- Sie mit der Baumstruktur von Ω in dem Sinn verträglich ist, dass für alle $uvw \in \Omega$ gilt:

  $comp_{\lhd}(u, uvw) \;\Rightarrow\; u \trianglelefteq uv \trianglelefteq uvw$
- $Comp_{\lhd}(\varepsilon)$ wird durch $\lhd$ total geordnet
- $\forall w \in Comp_{\lhd}(\varepsilon), w' \in \Omega \backslash Comp_{\lhd}(\varepsilon) : \neg(w' \lhd w)$

Bemerkung 2.3.3:

- Eine totale Traversierungsordnung ist eine topologische Sortierung von Ω
- Die darstellende Ordnung $\lhd_a$ einer vollständigen Traversierungsfolge a ist immer eine Traversierungsordnung. (Folgt umittelbar aus Def. 2.2.3 und Def. 2.2.12)
- Ist eine darstellende Ordnung einer (unvollständigen) Traversierungsfolge $a : \alpha \to \Omega$ keine Traversierungsordnung, so liegt dies ausschließlich an der Anordnung der Elemente von $\Omega \backslash a[\alpha]$.
- Für jede Traversierungsfolge gibt es eine darstellende Traversierungsordnung

Bis jetzt nützt uns der Übergang von Traversierungsfolgen zu Traversierungsordnungen noch nicht viel für die Analyse der Vollständigkeit. Der nächste Satz und das anschließende Korollar werden das ändern. Dafür benötigen wir aber noch die in Lemma 2.3.4 definierte Funktion.

Lemma 2.3.4:

Ist $(\Omega, \lhd)$ eine totale Ordnung, und existiert $f : \mathbb{N} \to \mathbb{N}$ mit $\Omega_k \lhd \Omega_{\geq f(k)}$, dann ist folgende Funktion wohldefiniert:

$g : \alpha \to (\Omega, \lhd)$:

$$\begin{aligned} &g(0) := min_{\lhd}(\Omega) \,(= min_{\lhd}(\Omega \backslash g[\{\emptyset\}])) \\ &g(n+1) := min_{\lhd}(\Omega \backslash g[\{0, \ldots, n\}]) \text{ , falls } n+1 \in \alpha \end{aligned}$$

, wobei $\alpha \preccurlyeq \omega$ und $|\alpha| = |\Omega|$.

Beweis:

Sei $k := min_{<}(\{|w| \mid w \in \Omega \backslash g[\{0, \ldots, n\}]\})$ und $w \in \Omega \backslash g[\{0, \ldots, n\}]$ mit $|w| = k$.

Dann gilt $w \lhd \Omega_{\geq f(k)}$.

Daraus folgt:

$$\begin{aligned} &\exists w \in \Omega \backslash g[\{0, \ldots, n\}] \, \forall v \in \Omega \backslash g[\{0, \ldots, n\}] : w \trianglelefteq v \\ \Leftrightarrow \\ &\exists w \in (\Omega \backslash g[\{0, \ldots, n\}]) \cap \Omega_{<f(k)} \forall v \in \Omega \backslash g[\{0, \ldots, n\}] : w \trianglelefteq v \end{aligned}$$

Aber $(\Omega \backslash g[\{0, \ldots, n\}]) \cap \Omega_{<f(k)}$ ist eine nicht leere endliche Menge, falls $n+1 \in \alpha$ (Schubfachprinzip). Also existiert in ihr ein Minimum und dieses Minimum ist gleich dem von $\Omega \backslash g[\{0, \ldots, n\}]$.

□

Satz 2.3.5:

Eine totale Ordnung $(\Omega, \lhd)$ ist vollständig gdw. $\exists f : \mathbb{N} \to \mathbb{N}$ mit $\Omega_k \lhd \Omega_{\geq f(k)}$.

Beweis:

"$\Rightarrow$":

$(\Omega, \lhd)$ ist vollständig, also existiert ein ordnungserhaltender Isomorphismus $h : (\Omega, \lhd) \to \alpha$ mit $\alpha \preccurlyeq \omega$.

$$max_k := \begin{cases} 0 & \text{falls } \Omega_k = \emptyset \\ max_{<}(h[\Omega_k]) & \text{sonst} \end{cases}$$

ist wohldefiniert, da Ω_k endlich.

Sei nun $f_k := max_< (\{|w| \mid w \in h^{-1}[\{0, \ldots, max_k\}]\}) + 1$.

Dann gilt $\Omega_{\geq f_k} \cap h^{-1}[\{0, \ldots, max_k\}] = \emptyset$ und damit
$min_{f_k} := min_< (h[\Omega_{\geq f_k}]) > max_k$,
denn sonst existiert $w \in \Omega_{\geq f_k}$ mit $h(w) \leq max_k$,
also $w \in h^{-1}[\{0, \ldots, max_k\}]$ und damit $f_k > |w|$, ein Widerspruch.

Da h ordnungserhaltender Isomorphismus gilt weiter
$min_\lhd (\Omega_{\geq f_k}) = h^{-1}(min_{f_k}) \rhd h^{-1}(max_k) = max_\lhd (\Omega_k)$, also $\Omega_k \lhd \Omega_{\geq f_k}$.

Definiere noch $f(k) := f_k$.

"$\Leftarrow$"

Wähle $\alpha \preccurlyeq \omega$ so, dass $|\alpha| = |\Omega|$.

Definiere $g : \alpha \to (\Omega, \lhd)$ wie in Lemma 2.3.4.

Es ist leicht zu sehen, dass g injektiv und ordnungserhaltend ist. Insbesondere gilt $(g[\alpha], \lhd) \cong \alpha$.

Zeige noch g ist surjektiv:
(Dies ist nur für den endlichen Fall trivial. Im unendlichen Fall müssen wir noch ausschließen, dass es ein Element $w \in \Omega$ gibt, das größer als unendlich viele andere Elemente von Ω ist. Das oben konstruierte g würde ein solches w nämlich niemals erreichen)

Sei dazu $w \in \Omega$, $|w| =: k$. Dann gilt:

$$\begin{aligned} & w \in \Omega_k \lhd \Omega_{\geq f(k)} \\ \Rightarrow & \{v \in \Omega \mid v \trianglelefteq w\} \subseteq \Omega_{<f(k)} \subseteq \Sigma^{<f(k)} \\ \Rightarrow & \{v \in \Omega \mid v \trianglelefteq w\} \text{ endliche Menge} \end{aligned}$$

Sei $m := |\{v \in \Omega \mid v \trianglelefteq w\}| \geq 1$.

Zeige per Induktion nach m:

$$\begin{aligned} & g(m-1) = w \\ & \text{und} \\ & g[\{0, \ldots, m-1\}] = \{v \in \Omega \mid v \trianglelefteq w\} \end{aligned}$$

Induktionsanfang $m = 1$:

$$\begin{aligned} & m = 1 \\ \Rightarrow & \{v \in \Omega \mid v \trianglelefteq w\} = \{w\} \\ \Rightarrow & w = min_\lhd (\Omega) \\ \Rightarrow & w = g(0) \text{ und } g[\{0\}] = \{w\} = \{v \in \Omega \mid v \trianglelefteq w\} \end{aligned}$$

Induktionsschritt $m+1$:

Sei $w' := max_{\lhd}(\{v \in \Omega \mid v \lhd w\})$. Dann

$$\begin{aligned} & |\{v \in \Omega \mid v \trianglelefteq w'\}| \\ = & |\{v \in \Omega \mid v \lhd w\}| \\ = & |\{v \in \Omega \mid v \trianglelefteq w\} \setminus \{w\}| \\ = & m \end{aligned}$$

$$\begin{aligned} I.V. &\Rightarrow g[\{0,\ldots,m-1\}] = \{v \in \Omega \mid v \trianglelefteq w'\} = \{v \in \Omega \mid v \lhd w\} \\ &\Rightarrow g(m) = min_{\lhd}(\Omega \backslash g[\{0,\ldots,m-1\}]) = w \\ &\Rightarrow g[\{0,\ldots,m\}] = g[\{0,\ldots,m-1\}] \cup \{g(m)\} = \{v \in \Omega \mid v \trianglelefteq w\} \end{aligned}$$

□

Korollar 2.3.6:

Ist $(\Omega, \lhd)$ eine Traversierungsordnung, dann gilt zusätzlich für die rechte Seite der Äquivalenz:

1. Es genügt, dass $\Omega_k \lhd \Omega_{f(k)}$.
2. o.B.d.A. ist f monoton wachsend

Beweis:

zu 1.

Zeige: Falls $(\Omega, \lhd)$ Traversierungsordnung, dann $\Omega_k \lhd \Omega_{f(k)} \Rightarrow \Omega_k \lhd \Omega_{\geq f(k)}$.

Sei dazu $u \in \Omega_{\geq f(k)}$. Dann $u = u'v$ für ein $u' \in \Omega_{f(k)}$, $v \in \Sigma^*$.
Es gilt für alle $w \in \Omega_k$:

$w \lhd u' \overset{Trav.Ord.}{\trianglelefteq} u'v = u$

zu 2.

Definiere $f^*(l) := max_{0 \leq i \leq l}(f(i))$. Dann gilt $f^*(l) \geq f(l)$ und $\Omega_{\geq f^*(l)} \subseteq \Omega_{\geq f(l)}$ für alle $l \in \mathbb{N}$.

Also folgt aus $\Omega_l \lhd \Omega_{\geq f(l)}$, dass $\Omega_l \lhd \Omega_{\geq f^*(l)}$.

Zudem ist f^* monoton wachsend.

□

Algorithmisch ausgedrückt bedeutet der erste Punkt des Korollars, dass für jedes Level eines Baumes ein tieferes Level existieren muss, sodass kein Knoten dieses tiefern Levels expandiert wird bevor alle Knoten des ursprünglichen Levels expandiert wurden.

Im vorangegangen Satz kann auf die Voraussetzung der Totalität der betrachteten Ordnung nicht verzichtet werden. Ist aber $|\Omega| = \infty$, dann impliziert bereits die Existenz der Funktion f die Totalität.

Korollar 2.3.7:

Ist $(\Omega, \lhd)$ eine Traversierungsordnung, $|\Omega| = \infty$ und existiert $f : \mathbb{N} \to \mathbb{N}$ mit $\Omega_k \lhd \Omega_{\geq f(k)}$, dann ist $\lhd$ total und damit vollständig.

Beweis:

Zeige zunächst durch Widerspruch, dass $|Comp_{\lhd}(\varepsilon)| = \infty$

Sei also $m := max_{<}(\{|w| \mid w \in Comp_{\lhd}(\varepsilon)\})$ und $w \in Comp_{\lhd}(\varepsilon)$ mit $|w| = m$.

Es folgt $\Omega_{\geq f(m)} \cap Comp_{\lhd}(\varepsilon) = \emptyset$.

Aber da $|\Omega| = \infty$ existiert $w' \in \Omega_{f(m)}$ und es gilt wegen $w \in \Omega_m \lhd \Omega_{f(m)}$:

$\varepsilon \lhd w \lhd w'$

Das bedeutet jedoch, dass $w' \in Comp_{\lhd}(\varepsilon)$ ein Widerspruch zur Maximalität von m.

Zeige nun durch Widerspruch, dass $Comp_{\lhd}(\varepsilon) = \Omega$. Daraus folgt dann unmittelbar die Totalität von $\lhd$ auf Ω, da $\lhd$ total ist auf $Comp_{\lhd}(\varepsilon)$.

Seien dazu $w \in \Omega_k$, $w' \in \Omega_{f(k)} \cap Comp_{\lhd}(\varepsilon)$ (verwendet $|Comp_{\lhd}(\varepsilon)| = \infty$) für ein $k \in \mathbb{N}$. Es gilt $w \lhd w'$.

Dann folgt $w \in Comp_{\lhd}(\varepsilon)$, denn falls nicht, gilt $\neg(w \lhd w')$, da $\lhd$ Traversierungsordnung. Ein Widerspruch.

□

Die folgende zweite Charakterisierung der Vollständigkeit einer partiellen Ordnung ist besonders nützlich um Unvollständigkeit zu zeigen.

Satz 2.3.8 (Charakterisierung Vollständigkeit)**:**

Eine partielle Ordnung $(\Omega, \lhd)$ ist vollständig gdw. sie isomorph zu einer endliche Summe von vollständigen Ordinalzahlen ist, wobei höchstens der letzte Summand unendlich (also $= \omega$) ist [5].

Dies ist äquivalent dazu, dass $(\Omega, \lhd)$ ist vollständig gdw. $(\Omega, \lhd)$ isomorph zu einer abzählbar unendlichen Summe von endlichen Ordinalzahlen ist.

Beweis:

Der Beweis ergibt sich unmittelbar aus den Gesetzten der Ordinalzahlarithmetik. Die oben genannten Summen sind immer gleich einem $\alpha \preccurlyeq \omega$.

□

[5] Man beachte, das die Addition der Ordinalzahlarithmetik im allgemeinen nicht kommutativ ist.

Ein Beispiel für die Anwendung dieser Charakterisierung findet sich im Abschnitt 3.1.2 des nächsten Kapitels.

Nachdem wir nun untersucht haben was es bedeutet, dass eine Ordnung vollständig ist, wollen wir uns als nächstes überzeugen, dass der gewählte Vollständigkeitsbegriff für Traversierungsordnungen tatsächlich sinnvoll ist. Dazu soll er mit dem früher definierten Vollständigkeitsbegriff auf Traversierungsfolgen verglichen werden.

Mit den darstellenden Ordnungen besteht schon eine Möglichkeit vom Raum der Traversierungsfolgen in den der Traversierungsordnungen zu gelangen. Den Zusammenhang in der anderen Richtung stellt die Definition der Induzierten Traversierungsfolge her.

Definition 2.3.9 (Induzierte Traversierungsfolge)**:**

Ist $(\Omega, \lhd)$ eine Traversierungsordnung, so gibt es zu ihr eine natürliche Traversierungsfolge. Diese heißt die von $\lhd$ *induzierte Traversierungsfolge* und ist folgendermaßen definiert:

$$a_\lhd(0) := min_\lhd(Comp_\lhd(\varepsilon))$$
$$a_\lhd(n+1) := min_\lhd(Comp_\lhd(\varepsilon) \setminus a_\lhd[\{0,\ldots,n\}]) \text{ , falls } n+1 \in \alpha$$

, wobei $\alpha \preccurlyeq \omega$ und $|\alpha| = |Comp_\lhd(\varepsilon)|$.

Das gerade definierte a ist wohldefiniert.

Zudem ist a injektiv (folgt unmittelbar aus der Definition) und tatsächlich eine Traversierungsfolge, da $\lhd$ eine Traversierungsordnung ist.

Beweis (Wohldefiniertheit)**:**

Zunächst gilt, dass $Comp_\lhd(\varepsilon)$ traversierbar ist, da $\lhd$ eine Traversierungsordnung ist.

Deshalb zerfällt $Comp_\lhd(\varepsilon) \setminus a_\lhd[\{0,\ldots,n\}]$ in endlich viele Bäume $B_1, \ldots, B_l$.

Seien $w_1, \ldots, w_l$ die Wurzeln dieser Bäume, dann gilt weil $\lhd$ Traversierungsordnung ist:

$w_i \trianglelefteq B_i$

Es folgt, dass $min_\lhd(Comp_\lhd(\varepsilon) \setminus a_\lhd[\{0,\ldots,n\}])$ existiert und gleich $min_\lhd(\{w_1,\ldots,w_l\})$ ist.

□

Bemerkung 2.3.10:

Eine von einer totalen Traversierungsordnung induzierte Traversierungsfolge, ist immer schwach vollständig.

Das kommende Lemma beschäftigt sich jetzt noch mit dem Problem, dass es keine eindeutige darstellende Ordnung einer Traversierungsfolge geben muss, und auch nicht alle ihre darstellenden Ordnungen Traversierungsordnungen sein müssen (Vergleiche Bem. 2.3.3).

Lemma 2.3.11:

Sind $\lhd$ und $\lhd'$ zwei darstellende Ordnungen einer Traversierungsfolge a, so gilt:

1. Ist eine der darstellenden Ordnungen unvollständig, dann ist die Traversierungsfolge unvollständig.
2. Ist die Traversierungsfolge schwach vollständig, dann gilt sogar:

   $\lhd$ vollständig $\Leftrightarrow$ $\lhd'$ vollständig

   Insbesondere sind also entweder alle darstellenden Ordnungen einer schwach vollständigen Traversierungsfolge vollständig oder keine.

Bemerkung 2.3.12:

Zur Untersuchung der Vollständigkeit einer schwach vollständigen Traversierungsfolge genügt es also, eine einzige darstellende Ordnung zu betrachten. Aus diesem Grund wird im Folgenden auch im Fall einer unvollständigen (aber schwach vollständigen) Traversierungsfolge a von *der* darstellenden Ordnung $\lhd_a$ gesprochen.

Wie schon erwähnt, bedeuten für uninformierte Traversierungsfolgen schwache und endliche Vollständigkeit dasselbe. Anders ausgedrückt, sind uninformierte Traversierungsfolgen bereits schwach vollständig, wenn sie endlich vollständig sind. Für endlich vollständige, uninformierte Traversierungsfolgen genügt es also immer, eine einzige darstellende Ordnung $\lhd_a$ zu betrachten.

Satz 2.3.13 (Äquivalenz der Vollständigkeitsbegriffe)**:**

1. Eine Traversierungsfolge ist genau dann vollständig im Sinn von Definition 2.2.9, wenn alle ihre darstellenden Ordnungen vollständig sind gemäß Definition 2.3.1.
2. Eine schwach vollständige Traversierungsfolge ist genau dann vollständig im Sinn von Definition 2.2.9, wenn ihre darstellende Ordnung vollständig ist gemäß Definition 2.3.1.

Bemerkung 2.3.14:

- Ist $\lhd$ eine Traversierungsordnung und $a_\lhd$ die induzierte Traversierungsfolge, so ist $\lhd$ eine darstellende Ordnung derselben.
- Ist $\lhd$ vollständig, so ist die darstellende Ordnung der (vollständigen) induzierten Traversierungsfolge $a_\lhd$ gerade wieder $\lhd$ selbst.

Zum Abschluss dieses Abschnitts kann man eine wichtige Tatsache festhalten:

Zur Analyse der Vollständigkeit eines Algorithmus genügt es seine darstellenden Ordnungen zu betrachten. Meistens genügt dabei die Betrachtung einer einzelnen darstellenden Ordnung.

Wie wir im nächsten Kapitel sehen werden, ist es mithilfe des induzierten Algorithmus einer Ordnung häufig sogar möglich den kompletten Algorithmus allein durch seine Ordnung zu spezifizieren.

2.4. Der Speicher

Die Definitionen von Traversierungsläufen und Traversierungsalgorithmen formalisieren ja bereits das Konzept eines Speichers. So ordnet ein Traversierungslauf ja jedem Zeitpunkt, die aktuell gespeicherte Menge von Konten zu. Es wird jedoch noch eine passende Definition der *Speicherkomplexität* benötigt. Dabei werden möglicherweise in der Implementierung vorhandene Verwaltungsinformationen ignoriert und lediglich die Menge der gespeicherten Knoten betrachtet. Zudem muss die Definition auch auf unendlichen Bäumen sinnvoll sein, also auch dann, wenn der Algorithmus letztlich niemals terminiert. Die anschließende Definition leistet dies. Insbesondere hat sie die Eigenschaft, dass sie auf endlichen Bäumen mit der klassischen Definition (Maximum des Speicherbedarfs über alle Zeitpunkte) übereinstimmt.

Definition 2.4.1 (Speicherkomplexität)**:**

1. Die *Speicherkomplexität* $m_A : \mathbb{N} \to \mathbb{N}$ eines Traversierungslaufs A ist definiert als
   $m_A(d) := max_<(\{|A(n)| \mid n \in \alpha, depth_A(n) \leq d\})$ [6]
   mit
   $depth_A(n) := max_<(\{|w| \mid w \in a_A[0, \ldots, n]\})$
2. Die *Speicherkomplexität* $M_T : \mathbb{N} \to \mathbb{N}$ *auf der Menge* $T \subseteq Trav_\Sigma$ eines Traversierungsalgorithmus $(A_\Omega)_{\Omega \in T}$ ist definiert als
   $M_T(d) := max_<(\{m_{A_\Omega}(d) \mid \Omega \in T\})$
3. Die *Speicherkomplexität* $M : \mathbb{N} \to \mathbb{N}$ eines Traversierungsalgorithmus ist definiert als
   $M(d) := M_{Trav}(d)$

[6] siehe Notation 2.2.5 (S.12)

3. Bekannte Algorithmen im neuen Formalismus

Das hier beginnende Kapitel soll mit dem gerade entwickeltem Formalismus vertraut machen und ein erstes Gefühl für seine Vorteile vermitteln. Dazu ist es praktisch sich die bekannten Algorithmen sozusagen in ihrer neuen „Verpackung“ anzuschauen.

Notation 3.0.2:

Seien $(\Omega, \lhd)$ eine partielle Ordnung und $w \in \Omega$.

$\lhd\!\!\!\cdot\,(w) := \{w' \in \Omega \mid w' \lhd w\}$

$\rhd\!\!\!\cdot\,(w) := \{w' \in \Omega \mid w' \rhd w\}$

3.1. Tiefendurchlauf

3.1.1. Darstellung

Der Tiefendurchlauf lässt sich sehr leicht mittels der von ihm auf den Knoten des Baumes definierten Ordnung darstellen. Mit der von uns in Abschnitt 2.1 eingeführten Benennung der Knoten entspricht die Ordnung des Tiefendurchlaufs einfach der lexikografischen Ordnung auf den Namen der Knoten.

Formal:

$\lhd_{depth} = \lhd_{lex}$

bzw.

$\forall w, w' \in \Omega : w \lhd_{depth} w' \Leftrightarrow w \lhd_{lex} w'$

Man kann sich leicht überlegen, dass der von dieser Ordnung induzierte Algorithmus gerade die Knoten im Speicher behält, die auch eine typische Implementierung eines Tiefendurchlaufs speichern würde.

Die von $\lhd_{depth}$ induzierte Traversierungsfolge $a_{\lhd_{depth}}$ soll einfach mit *depth* bezeichnet werden.[7]

3.1.2. Unvollständigkeit

Es ist leicht einzusehen, dass der Tiefendurchlauf auf den meisten unendlichen Bäumen nicht vollständig seien kann. Nachdem zum Beweis der Unvollständigkeit ja die Angabe eines einzigen Gegenbeispiels genügt ist dieser Beweis auch ohne passenden Formalismus natürlich gut machbar. Trotzdem lassen sich an den folgenden Argumenten zwei Dinge erkennen:

[7] Das ist naheliegend, da $\lhd_{depth}$ *die* darstellende Ordnung von $a_{\lhd_{depth}}$ (bzw. *depth*) im Sinne von Bemerkung 2.3.12 ist.

Erstens erlaubt es der Formalismus Aussagen wie „Der Tiefendurchlauf verschwindet im ersten unendlichen Ast“ präzise auszudrücken.

Zweitens kann man in gewisser Weise die Unvollständigkeit „ausrechnen“.

Nun aber zum formalen Beweis der Unvollständigkeit.

Enthält Ω unendlich viele Knoten, so auch einen unendlichen Ast (Königs Lemma). Sei $t \in \Sigma^\omega$ der lexikographisch erste unendliche Ast in Ω. Die Menge der Präfixe $T \subseteq \Omega$ von t entspricht der Menge der Knoten auf t. Sei $S := \{w \in \Omega \mid T \rhd_{depth} w\}$ und $U := \{w \in \Omega \mid T \lhd_{depth} w\}$. Falls $U \neq \emptyset$ gilt, lässt sich die Argumentation auf zweierlei Weise fortsetzen:

Möglichkeit 1 (verwendet Satz 2.3.5):

$\exists w \in \Omega : T \lhd_{depth} w$

und wegen $T \cap \Omega_n \neq \emptyset$ für alle $n \in \mathbb{N}$ damit auch:

$\exists w \in \Omega \forall n \in \mathbb{N} \exists w' \in \Omega_n : w' \lhd_{depth} w$

Es folgt ($k = |w|$):

$\exists k \in \mathbb{N} \forall n \in \mathbb{N} : \Omega_k \not\lhd_{depth} \Omega_n$

Die Unvollständigkeit folgt damit aus Satz 2.3.5.

Möglichkeit 2 (verwendet Satz 2.3.8):

Es gilt: $S \lhd_{depth} T \lhd_{depth} U$

Die kleinste Ordinalzahl β, die diese Bedingung erfüllt[8]ist $\beta = \underbrace{|S|}_{\cong k \prec \omega} + \underbrace{|T|}_{\cong \omega} + \underbrace{|U|}_{\cong \alpha \succ 0} \succ \omega$

Man beachte:

Im Allgemeinen ist die lexikographische Ordnung keine Wohlordnung. Insbesondere gilt im Allgemeinen also nicht $(\Omega, \lhd_{depth}) \cong \beta$. Die obige Überlegung zeigt jedoch, dass selbst wenn $(\Omega, \lhd_{depth})$ wohlgeordnet wäre, es immernoch unvollständig wäre.

3.2. Breitendurchlauf

3.2.1. Darstellung

Auch der Breitendurchlauf lässt sich gut durch die von ihm definierte Ordnung darstellen. Statt nur für den Breitendurchlauf soll jedoch die anschließende Analyse gleich die Verallgemeinerung desselben, die A^*-Suche abdecken [9]. Damit wäre dann auch gezeigt, dass unser Formalismus nicht nur für uninformierte Verfahren nützlich ist.

[8] Eine Ordinalzahl β *erfüllt* die Bedingung $S \lhd_{depth} T \lhd_{depth} U$, wenn es eine isomorphe Wohlordnung $(\Omega, \lhd)$ gibt, die die Bedingung erfüllt.

[9] Mam beachte: Der A^*-Algorithmus schließt keinen Knoten von der Suche aus. Er bevorzugt lediglich vielversprechendere Knoten. Soll er alle Lösungen finden, traversiert er letztlich doch den ganzen Baum.

Charakteristisch für den A^*-Algorithmus ist seine optimistische Kostenschätzfunktion $f(w) = g(w) + h(w)$. Hierbei gibt g die bis zu w bereits entstandenen Kosten an und h die optimistisch geschätzten noch verbleibenden Kosten von w bis zu einem Ziel. Man beachte, dass $g(w) \geq |w|$ ist.

Die Ordnung des A^* ist gegeben durch:

$w \lhd_{A^*} w' \Leftrightarrow (f(w), w) \lhd (f(w'), w') \Leftrightarrow f(w) < f(w')$ oder $f(w) = f(w') \wedge w \lhd_{lex} w'$

Wieder kann man sich leicht davon überzeugen, dass der von obiger Ordnung induzierte Algorithmus sich gerade wie erwartet, also wie eine typischen A^*-Implementierung verhält.

Den Breitendurchlauf erhält man, indem man $g(w) = |w|$ und $h(w) = 1$ setzt.

Es gilt dann:

$\Omega_k \lhd_{breadth} \Omega_{k+1}$

$\forall w, w' \in \Omega_k : w \lhd_{breadth} w' \Leftrightarrow w \lhd_{lex} w'$

3.2.2. Vollständigkeit

Mit Hilfe von Satz 2.3.5 lässt sich die Vollständigkeit des A^*-Algorithmus jetzt schnell zeigen.

Es gilt nämlich

$\Omega_k \lhd_{A^*} \Omega_{(max_<(f[\Omega_k]) + 1)}$,

da für $w \in \Omega_k$ und $w' \in \Omega_{(max_<(f[\Omega_k]) + 1)}$

$f(w') \geq g(w') \geq |w'| = max_<(f[\Omega_k]) + 1 > max_<(f[\Omega_k]) \geq f(w)$

und somit $w \lhd_{A^*} w'$ ist.

Mit der Funktion $k \mapsto max_<(f[\Omega_k]) + 1$ lässt sich also Satz 2.3.5 anwenden und der A^*-Algorithmus ist vollständig.

Speziell für den Breitendurchlauf lässt sich die Vollständigkeit auch sehr schnell mithilfe der Ordinalzahlarithmetik (Satz 2.3.8) beweisen. Dazu überlege man sich, dass folgendes gilt:

$\lhd_{breadth} \cong \sum_{i=0}^{+\infty} |\Omega_i| \preccurlyeq \omega$

Verglichen mit den Beweisen in [Pea84] und [Nil80] ist die obige Argumentation äußerst knapp und präzise. Genaugenommen erfordert sie aufgrund ihres formalen Charakters nicht einmal ein genaueres Verständnis des Ablaufs des A^*-Algorithmus. Es wird somit bereits hier sichtbar, dass es nützlich ist die Algorithmen auf dieser etwas abstrakteren Ebene zu analysieren.

3.3. Iterativer Tiefendurchlauf

3.3.1. Darstellung

Definition 3.3.1:

Seien α_1 und α_2 Ordinalzahlen, Ω_1 und Ω_2 traversierbar und gebe es Abbildungen $a_1 : \alpha_1 \to \Omega_1$ und $a_2 : \alpha_2 \to \Omega_2$.

Die Abbildung $(a_1 \,;\, a_2) : (\alpha_1 + \alpha_2) \to (\Omega_1 \cup \Omega_2)$ ist definiert durch:

falls α_1 endlich

$$(a_1 \,;\, a_2)(n) := \begin{cases} a_1(n) & \text{falls } n < \alpha_1 \\ a_2(n - \alpha_1) & \text{sonst} \end{cases}$$

oder allgemein

$$(a_1 \,;\, a_2)(n) := \begin{cases} a_1(n) & \text{falls } n \in I[\{1\} \times \alpha_1] \\ a_2(n) & \text{falls } n \in I[\{2\} \times \alpha_2] \end{cases}$$

, wobei $I : (\{1\} \times \alpha_1 \cup \{2\} \times \alpha_2) \to \alpha_1 + \alpha_2$ der Isomorphismus mit $I[\{1\} \times \alpha_1] \cong \alpha_1$, $I[\{2\} \times \alpha_2] \cong \alpha_2$ und $I[\{1\} \times \alpha_1] < I[\{2\} \times \alpha_2]$ ist.

Bemerkung 3.3.2:

Sind a_1 und a_2 Traversierungsfolgen, so ist $(a_1 \,;\, a_2)$ ebenfalls eine Traversierungsfolge.

$$idepth := depth_{\leq 0} \,;\, depth_{\leq 1} \,;\, \cdots \,;\, depth_{max(\{|w| \mid w \in \Omega\})} = \mathop{\Large ;}_{i=0}^{max(\{|w| \mid w \in \Omega\})} depth_{\leq i}$$

Hierbei ist mit $depth_{\leq i}$ die Traversierungsfolge des Tiefendurchlaufs auf dem Baum $\Omega_{\leq i}$ gemeint.

Für $|\Omega| = \infty$ sei $max(\{|w| \mid w \in \Omega\}) := \infty$.

3.3.2. Vollständigkeit

Es ist leicht zu sehen, dass $\lhd_{breadth}$ eine darstellende Ordnung von *idepth* ist. Um die Analyse der Vollständigkeit aber auf $\lhd_{breadth}$ beschränken zu können müssen wir noch die schwache Vollständigkeit von *idepth* zeigen. Die folgende Beobachtung ist dabei sehr hilfreich:

Lemma 3.3.3:

Seien a eine Traversierungsfolge, b eine schwach vollständige Traversierungsfolge und gelte, dass b eine Teilfolge von a ist. Dann ist auch a schwach vollständig.

Da $\lhd_{depth}=\lhd_{lex}$ total ist, folgt aus Bemerkung 2.3.10, dass die Traversierungsfolge des Tiefendurchlaufs *depth* schwach vollständig ist. Zudem kann man sich unschwer davon überzeugen, dass *depth* eine Teilfolge von *idepth* ist. Im endlichen Fall z.B. gilt nämlich $depth = depth_{max(\{|w| \mid w \in \Omega\})}$.

Damit ist *idepth* schwach vollständig nach Lemma 3.3.3.

Wir können also (Bemerkung 2.3.12) von der darstellenden Ordnung $\lhd_{idepth}=\lhd_{breadth}$ sprechen.

Bekanntermaßen ist $\lhd_{breadth}$ vollständig, damit auch $\lhd_{idepth}$ und nach Satz 2.3.13 Punkt 2 auch *idepth*.

4. Der D&B-Durchlauf und seine Familie

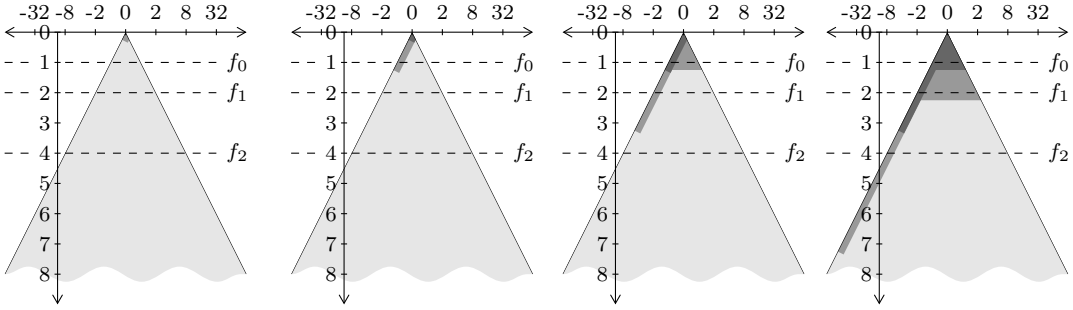
Bereits im einleitenden Kapitel wurde darauf verwiesen, dass die bekannten uninformierten Suchverfahren alle einen der Nachteile, Unvollständigkeit, hoher Speicherbedarf oder Mehrfachexpansion haben.

Dieses Kapitel stellt nun eine Möglichkeit vor, einen Algorithmus zu erhalten, der keine der angeführten Schwächen hat. Weitere Ideen werden in Kapitel 2.3.3 angeschnitten.

4.1. Intuition

Die Grundidee hinter dem D&B-Durchlauf ist denkbar einfach: Man verzahnt einen Tiefendurchlauf mit einem Breitendurchlauf. Dabei soll die Tiefensuche für ein schnelles Vordringen in den Suchraum und die Breitensuche für die Vollständigkeit sorgen. Die eigentlich spannende Frage ist, wie schnell darf bzw. muss die Breitensuche fortschreiten, damit einerseits der Speicher nicht zu stark belastet wird, andererseits aber Vollständigkeit garantiert bleibt.

Die folgende Bilderserie soll ein Gefühl dafür vermitteln.[10]



In den Bildern ist zu erkennen, dass der Tiefendurchlauf sehr viel schneller in die Tiefe vordringt als der Breitendurchlauf. Der entscheidende Punkt ist allerdings, dass einerseits die Tiefe des Tiefendurchlauf durch eine Funktion der Tiefe des Breitendurchlauf beschränkt ist, andererseits aber auch umgekehrt die Tiefe des Breitendurchlaufs durch eine Funktion der Tiefe des Tiefendurchlaufs.[11]Beide Funktionen sollten monoton wachsend sein und idealerweise ist die eine (im wesentlichen) die Umkehrfunktion der anderen. Deshalb taucht im Folgenden auch nur die erste der beiden genannten Funktionen auf, also die die Tiefe des Tiefendurchlaufs beschränkende Funktion $i \mapsto f_i$ der Tiefe des Breitendurchlaufs.

Sei eine solche Funktion $i \mapsto f_i$, die den Voraussetzungen

$$f_i > i$$
$$f_{i+1} > f_i$$

genügt gegeben.

[10] Hierbei sei $f_i := 2^i$.

[11] Man beachte, dass zwar von Tiefen- und Breitendurchlauf die Rede ist, aber trotzdem jeder Knoten nur einmal expandiert wird.

Offensichtlich sind dann die f_i-ten Level Ω_{f_i} von Bedeutung für den Ablauf des Algorithmus. Eine ganz besondere Rolle kommt dabei den lexikographisch kleinsten Elementen dieser f_i-ten Level zu. Sie werden *Pivot-Knoten* genannt, weil sie im Baum Ω die Tiefenschranken repräsentieren. Auf Grundlage dieser Idee, lassen sich zu den Knoten s_i Knotenmengen $S_i \subseteq \Omega$ definieren, die wie vom anschließenden Pseudocode angedeutet nacheinander durchlaufen werden.

Definition 4.1.1:

$i_{max} := max\left(\{i \mid \Omega_{f_i} \neq \emptyset\}\right)$

für $|\Omega| = \infty$ ist $i_{max} := \infty$

$\Omega_{-1} := \emptyset$

$s_i := min_{lex}\left(\Omega_{f_i}\right)$ für $0 \leq i \leq i_{max}$

$$S_i := \left(\underbrace{\lhd_{lex}(s_i)}_{:=D_i} \cup \underbrace{\Omega_{i-1}}_{:=B_i} \right) \setminus \underbrace{\bigcup_{j=0}^{i-1} (S_j \cup \{s_j\})}_{:=K_i} \text{ für } 0 \leq i \leq i_{max}$$

$$R := \Omega \setminus \bigcup_{j=0}^{i_{max}} (S_j \cup \{s_j\})$$

Bemerkung 4.1.2:

- $s_i \in \Omega_{f_i}$, also $|s_i| = f_i$
- Jede der oben definierten Mengen S_i bzw. R ist endlich.
- Für $i_{max} = \infty$ ist $R = \emptyset$.

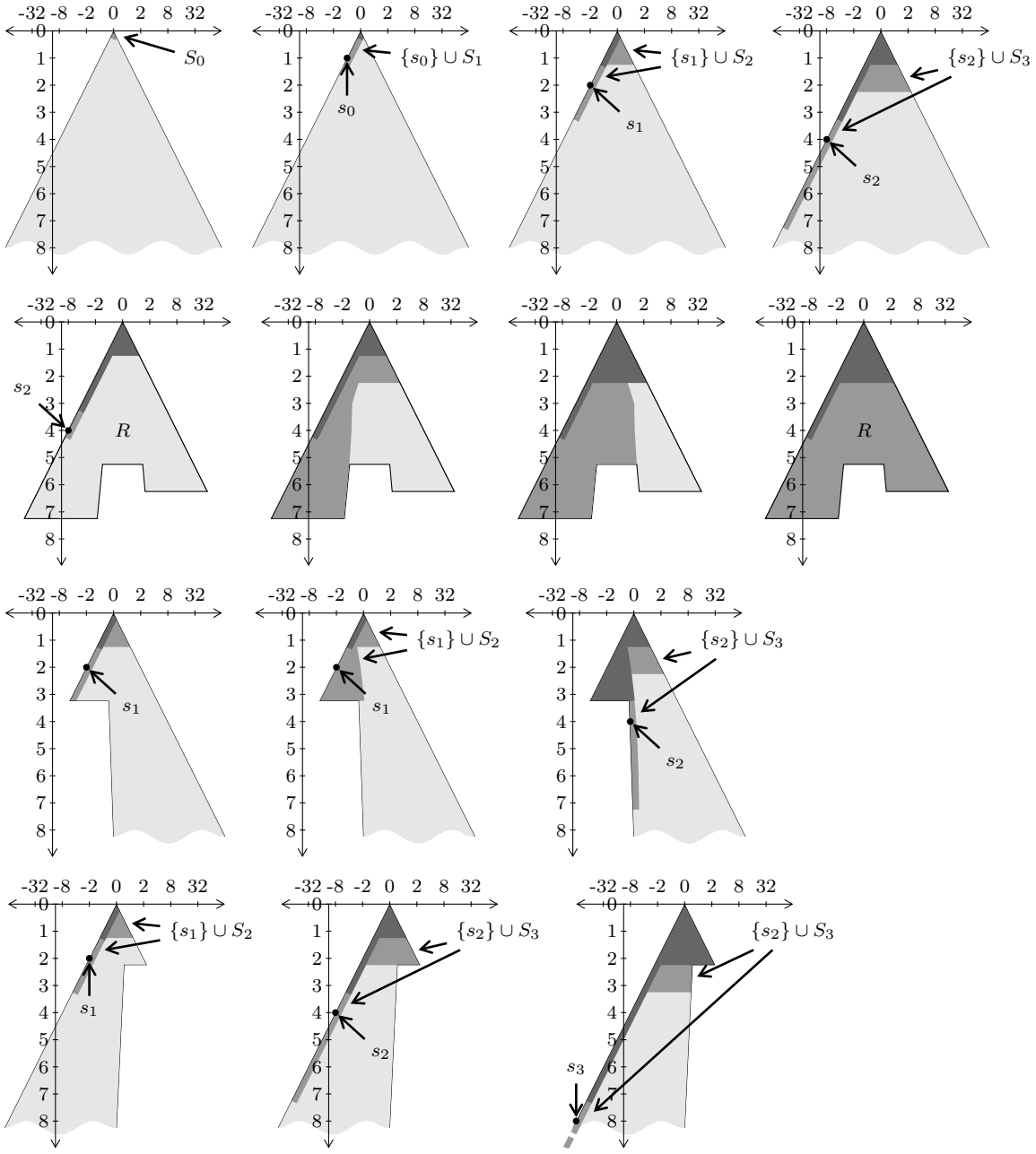
Der D&B-Durchlauf kann jetzt durch folgenden Pseudocode illustriert werden.

```
while i < i_max do
    foreach w ∈ S_i do [12]
        expand(w)
    end
    expand(s_i)
    ++i
end
foreach w ∈ R do [12]
    expand(w)
end
```

[12] Die Reihenfolge in der über die Knoten iteriert wird, ist letztlich nicht völlig frei, wie es der Code suggeriert. Da die genauen Einschränkungen zum jetzigen Zeitpunkt aber nicht zum Verständnis beitragen, wird erst auf Seite 37 darauf eingegangen. In den folgenden Bildern wird immer in *lexikographischer* Reihenfolge über die Mengen iteriert.

Zu obigem Code sei angemerkt, dass er nur dazu dient den beabsichtigten Ablauf bezüglich der Expansionsreihenfolge darzustellen. Er ist keine mögliche Implementierung des Algorithmus. Der Grund ist einfach: Ein uninformierter Algorithmus kennt die Struktur der S_i erst *nachdem* er die Mengen durchlaufen hat. Der obige Code benötigt diese Information jedoch *bevor* die Mengen durchlaufen werden. Kapitel 5 zeigt tatsächlich umsetzbare Implementierungsmöglichkeiten.

Die erste der folgenden Bilderserien entspricht der vorangegangenen, mit dem Unterschied, dass statt der Level f_i die Pivot-Knoten s_i und die Mengen S_i eingetragen sind. Sie zeigt die Auswirkungen der Definition für den unendlichen vollständigen (Binär-)Baum. Die anschließenden drei Bilderserien illustrieren den Ablauf des Algorithmus auf anderen (Binär-)Bäumen. [13]



[13] Die jeweils verwendete Funktion ist, $f_i = 2^i$. Für allgemeine Bäume mit Grad b lassen sich entsprechende Bilder mit Funktionen $f_i = b^i$ erstellen.

Der beschriebene D&B-Durchlauf hat einen interessanten Effekt, der gut zu erkennen ist, wenn man die Bilder des endlichen Baumes mit den Bildern der unendlichen Bäume vergleicht. Da in einem endlichen Baum die Tiefe des Tiefendurchlaufs durch die endliche Tiefe des Baums und außerdem die Tiefe des Breitendurchlauf durch eine Funktion der Tiefe des Tiefendurchlaufs beschränkt ist, bleibt der Breitendurchlauf nach Erreichen dieser maximalen Tiefe stehen. Anders ausgedrückt, verhält sich der beschriebene Algorithmus auf endlichen Bäumen nahezu wie der Tiefendurchlauf. Umgekehrt gilt auf unendlichen Bäumen, dass der Tiefendurchlauf im ersten unendlichen Ast sozusagen „verschwindet“ und der größte Teil des Baumes also vom Breitendurchlauf bearbeitet wird. Das bedeutet, dass sich der Algorithmus auf unendlichen Bäumen im wesentlichen wie ein Breitendurchlauf verhält. Betrachtet man den Tiefendurchlauf als passendes Verfahren für endliche Bäume und den Breitendurchlauf als passendes Verfahren für unendliche Bäume, „wählt“ der Algorithmus also in gewisser Weise das bevorzugte Verfahren für einen bestimmten Baum aus. Für unendliche Bäume würde man vielleicht statt des Breitendurchlaufs lieber den Iterativen Tiefendurchlauf einsetzen. Auf eine entsprechende Kombination von Tiefen- und Iterativem Tiefendurchlauf bei der der gleiche Effekt auftritt, wird in Kapitel 6.3 kurz eingegangen.

4.2. Die Familie

Eine weitere interessante Eigenschaft des Verfahrens ist, dass sich mit passenden Funktionen f_i eine Familie von Algorithmen zu einem Parameter $c \in \mathbb{N} \cup \{\infty\}$ angeben lässt, die folgende Eigenschaften hat:

- Für $c \geq 1$ ist der Algorithmus vollständig.
- Für $1 \leq c < \infty$ besitzt er eine polynomielle Speicherkomplexität von $O(d^c)$.
- Der Algorithmus für $c = 0$ entspricht dem Tiefendurchlauf.
- Der Algorithmus für $c = \infty$ entspricht dem Breitendurchlauf.

Die benötigten Funktionen lassen sich praktischerweise als eine einzige in c parametrisierte Funktion $f_{c,i}$ darstellen. Es ist naheliegend es mit $f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor$ [14] zu versuchen. Diese Funktion hat aber noch nicht die geforderten Eigenschaften (außer für $c = 1$), insbesondere ist sie nicht streng monoton. Die Funktion $f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$ [14] ist jedoch geeignet.

Warum könnte die Parametrisierbarkeit nützlich sein? Mit Hilfe des Parameters c lässt sich sozusagen ausdrücken, wieviel Speicher man in die Vollständigkeit des Algorithmus investieren möchte. Dabei stehen einem jetzt nicht mehr nur die beiden Extreme, der Tiefen- und der Breitendurchlauf, zur Verfügung, die entweder gar nichts oder alles in Vollständigkeit investieren. Man hat jetzt die Möglichkeit nahezu beliebige Abstufungen zwischen diesen Extremen auszuwählen. Später werden wir sehen, dass sich der Parameter c auch als Laufzeitparameter in die Implementierung integrieren lässt. Das bedeutet, dass für alle Algorithmen der Familie nur eine einzige Implementierung benötigt wird. Tatsächlich ist es sogar möglich, den Parameter dynamisch, also während der Traversierung anzupassen.

[14] Hierbei sei $\frac{i}{\infty} = 0$, $\frac{i}{0} = \infty$, $b^\infty = \infty$ und $\Omega_\infty = \emptyset$.

Auf eine entsprechende Implementierung wird in Kapitel 5 eingegangen. Die zentrale Idee für diese Implementierung stammt dabei aus der nun folgenden formalen Analyse der neuen Algorithmen.

4.3. Formell

Zunächst einige Worte zur Struktur der formalen Analyse. Als erstes wird für die mengenbasierte Definition 4.1.1 eine axiomatische Charakterisierung angegeben. Anschließend wird die Äquivalenz der ursprünglichen Definition und der Charakterisierung bewiesen. Die Charakterisierung erlaubt es erstens schnell auf die Vollständigkeit des Algorithmus zu schließen. Zweitens wird sie auch für den Beweis der Speicherkomplexität der Algorithmen nützlich. Schließlich liefert sie uns Invarianten, die wir in Kapitel 5 verwenden werden. Der Beweis der Speicherkomplexität bildet den Abschluss der formalen Analyse.

An dieser Stelle soll an die früher eingeführte Notation 3.0.2 auf Seite 26 erinnert werden. Sie wird nachfolgend ausgiebig benutzt.

Da das neue Verfahren eine Kombination aus Tiefen- und Breitendurchlauf ist, ist es naheliegend zu Beginn nochmal einen Blick auf deren axiomatische Charakterisierung zu werfen (siehe Kapitel 3).

Der Tiefendurchlauf wird durch die Formel

$$\forall w, w' \in \Omega : w \lhd_{depth} w' \Leftrightarrow w \lhd_{lex} w'$$

beschrieben. Die Formel

$$\forall w \in \Omega_k : \lhd_{lex}(w) \lhd_{depth} w \qquad \text{(DEPTH)}$$

enthält die gleiche Aussage nur in anderer Schreibweise.

Der Breitendurchlauf wird durch die beiden Formeln

$$\begin{array}{ll} 1. & \Omega_k \lhd_{breadth} \Omega_{k+1} \\ 2. & \forall w, w' \in \Omega_k : w \lhd_{breadth} w' \Leftrightarrow w \lhd_{lex} w' \end{array} \qquad \text{(BREADTH)}$$

charakterisiert.

Der D&B-Durchlauf lässt sich nun wie folgt charakterisieren:

$$\begin{array}{ll} 1. & \Omega_k \lhd_{d\&b} \Omega_{f_{k+1}} \\ 2. & \forall w, w' \in \Omega_k : w \lhd_{d\&b} w' \Leftrightarrow w \lhd_{lex} w' \\ 3. & \forall w \in \Omega_k : \lhd_{lex}(w) \lhd_{d\&b} w \vee \exists w' \in \Omega_{f_k} : w' \lhd_{d\&b} w \end{array} \qquad \text{(D\&B)}\ ^{15}$$

[15] In Kapitel 5 wird auf die intuitive Bedeutung der Formeln eingegangen.

Dabei fällt auf, dass man die obigen Formeln als Kombination der Formeln für den Breiten- bzw. Tiefendurchlauf auffassen kann. Z.B. ähneln die ersten beiden Formeln sehr denen des Breitendurchlaufs. Einziger Unterschied ist eigentlich, dass die Forderung der ersten Formel etwas schwächer ist (Ermöglicht dem Tiefendurchlauf zu arbeiten). Ebenso ist der erste Teil der dritten Formel unmittelbar vom Tiefendurchlauf übernommen. Auch hier wird lediglich die Forderung durch den zweiten Teil der Disjunktion leicht abgeschwächt (Ermöglicht dem Breitendurchlauf zu arbeiten). Die geforderten $w' \in \Omega_{f_k}$ können übrigens immer als die s_k aus Definition 4.1.1 gewählt werden.

Für die oben vorkommenden Funktionen gelten wie schon früher die Einschränkungen:

$$f_i > i$$
$$f_{i+1} > f_i$$

Als nächstes muss Definition 4.1.1 noch auf das Gebiet der Ordnungen übertragen werden. Das ist nicht allzu schwierig. Man benötigt lediglich eine Ordnung $\lhd$, so dass $S_0 \lhd s_0 \lhd S_1 \lhd s_1 \lhd \cdots \lhd S_{i_{max}} \lhd s_{i_{max}} \lhd R$. Eine solche Ordnung erhält man z.B. per Ordinalzahlarithmetik als $(\Omega, \lhd) = \sum_{j=0}^{i_{max}} ((S_j, \lhd_j) + \{s_j\}) + (R, \lhd_R)$, wobei $\lhd_j$ und $\lhd_R$ lineare Ordnungen auf den Mengen S_j bzw. R sind.

Sowohl Definition 4.1.1 (S.32) als auch der anschließende Pseudocode schränken in keiner Weise die Reihenfolge ein, mit der die Knoten in den Mengen S_i bzw. R durchlaufen werden. Insofern könnten die Ordnungen $\lhd_j$ und $\lhd_R$ auf S_j bzw. R beliebig gewählt werden. Tatsächlich sollen diese Ordnungen aber etwas eingeschränkt werden.[16]

Dabei geht es um Folgendes: Wie sowohl aus Definition 4.1.1, als auch aus den in Abschnitt 4.1 gezeigten Bildern zu erkennen ist, setzten sich die Mengen S_j und R aus zwei Teilen zusammen. Da sind zunächst die verblieben Knoten von Ω_{j-1} bzw. $\Omega_{i_{max}}$, also die Knoten des $(j-1)$-ten bzw. i_{max}-ten Levels. In einem dieser Level sollen die Knoten analog zum Breitendurchlauf in lexikographischer Reihenfolge („von links nach rechts“) expandiert werden. Der nicht auf diesen Leveln liegende, in die Tiefe reichende Teil von S_j bzw. R, $S_j \backslash \Omega_{j-1}$ bzw. $R \backslash \Omega_{i_{max}}$, soll jeweils ebenfalls in lexikographischer Reihenfolge expandiert werden. Das ist ähnlich dem Vorgehen des Tiefendurchlaufs.

Es sei angemerkt, dass dadurch noch nicht die exakte Reihenfolge der Expansion von S_j bzw. R festgelegt ist. Lediglich auf den Mengen $S_j \cap \Omega_{j-1}$ bzw. $R \cap \Omega_{i_{max}}$ und $S_j \backslash \Omega_{j-1}$ bzw. $R \backslash \Omega_{i_{max}}$ ist dies der Fall. Die Verzahnung der Expansion der Mengen $S_j \cap \Omega_{j-1}$ und $S_j \backslash \Omega_{j-1}$ bzw. $R \cap \Omega_{i_{max}}$ und $R \backslash \Omega_{i_{max}}$ bleibt jedoch beliebig.

[16] Bisher wurde auf diese Einschränkung der Reihenfolge, mit der die Knoten in den Mengen S_i bzw. R durchlaufen werden verzichtet, weil sie nicht zum Verständnis der eigentlichen Idee beiträgt und deren Darstellung behindert hätte.

Definition 4.3.1 (Ergänzung zu Definition 4.1.1)**:**

Im Kontext von Definition 4.1.1 sei $\lhd$ eine partielle Ordnung für die gilt:

1. $S_0 \lhd s_0 \lhd S_1 \lhd s_1 \lhd \cdots \lhd S_{i_{max}} \lhd s_{i_{max}} \lhd R$
2. $\forall w, w' \in S_i \cap \Omega_{i-1} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$
3. $\forall w, w' \in S_i \backslash \Omega_{i-1} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$
4. $\forall w, w' \in R \cap \Omega_{i_{max}} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$
5. $\forall w, w' \in R \backslash \Omega_{i_{max}} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$

Bemerkung 4.3.2:

Die Forderungen in Definition 4.3.1 garantieren insbesondere, dass $\lhd$ eine Traversierungsordnung ist.

Hier ein Beispiel für eine Ordnung, die die genannten Bedingungen erfüllt. Diese Ordnung formalisiert die Strategie immer den Tiefendurchlauf so lange arbeiten zu lassen wie es die aktuelle Tiefe des Breitendurchlaufs zulässt und dann den Breitendurchlauf gerade solange fortzusetzen, bis er die nächst größere Tiefe erreicht, die es erlaubt wieder mit dem Tiefendurchlauf weiter zu machen. Sie entspricht der für die Bilder in Abschnitt 4.1 verwendeten Ordnung.

Beispiel 4.3.3:

$$(S_i, \lhd_i) = \left(S_i \cap \lhd_{lex}(s_i), \lhd_{lex}\right) + \left(\left(S_i \backslash \lhd_{lex}(s_i)\right) \cap \Omega_{i-1}, \lhd_{lex}\right)$$
$$(R, \lhd_R) = (R, \lhd_{lex})$$

4.3.1. Vollständigkeit

Wir wollen zeigen, dass Definition 4.1.1 zusammen mit Definition 4.3.1 genau die Modelle von (D&B) beschreibt.

Zunächst zeigen wir, dass jede Ordnung die den Definitionen 4.1.1 und 4.3.1 genügt ein Modell von (D&B) ist.

Lemma 4.3.4:

Genügt $\lhd$ den Definitionen 4.1.1 und 4.3.1, dann gilt

$M := \bigcup_{j=0}^{k} (S_j \cup \{s_j\}) \cup S_{k+1} \ \lhd \ \Omega_{f_{k+1}}$.

Beweis:

Wir zeigen zunächst $M \subseteq \Omega_{<f_{k+1}}$.

Sei $w \in M$.

Fall $w \in \Omega_i$ für $i \leq k$:

$w \in \Omega_i \subseteq \Omega_{<f_{k+1}}$

Fall $w \in \lhd_{lex}(s_i)$ für $i \leq k+1$:

Für alle $w' \in \Omega_{f_i}$ gilt $s_i \trianglelefteq w'$, denn nach Definition ist $s_i \lhd_{lex} w'$, also $w' \notin \lhd_{lex}(s_i)$.

Außerdem ist $w' \notin \Omega_j$ für $j \leq i$, da $i < f_i$. Somit $w' \notin \bigcup_{j=0}^{i-1} (S_j \cup \{s_j\}) \cup S_i$ und damit

$w' \not\lhd s_i$.

$w \in \lhd_{lex}(s_i) \lhd s_i \trianglelefteq \Omega_{f_i} \overset{\lhd \, Trav.Ordnung}{\trianglelefteq} \Omega_{\geq f_{k+1}}$

Somit $w \in \Omega_{<f_{k+1}}$.

Fall $w = s_i$ für $i \leq k$:

$w = s_i \in \Omega_{f_i} \overset{\text{Monotonie von } f}{\subseteq} \Omega_{<f_{k+1}}$

Daraus folgt $M \subseteq \Omega_{<f_{k+1}}$, damit $M \cap \Omega_{f_{k+1}} = \emptyset$ also auch $M \lhd \Omega_{f_{k+1}}$ nach Definition von M und Definition 4.3.1(1).

□

Lemma 4.3.5:

Genügt $\lhd$ den Definitionen 4.1.1 und 4.3.1, dann gilt $\Omega_k \lhd \Omega_{f_{k+1}}$.

Beweis:

Für $k \geq i_{max}$ ist dies klar, denn $\Omega_{f_{k+1}} = \emptyset$.

Für $k < i_{max}$ gilt:

$\Omega_k \subseteq \bigcup_{j=0}^{k} (S_j \cup \{s_j\}) \cup S_{k+1} =:$ M und $M \lhd \Omega_{f_{k+1}}$

□

Lemma 4.3.6:

Genügt $\lhd$ den Definitionen 4.1.1 und 4.3.1, dann gilt $\forall w, w' \in \Omega_k : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$.

Beweis:

Fall $w, w' \in S_{k+1}$: klar, da $w, w' \in S_{k+1} \cap \Omega_k$

Fall $w, w' \in S_i$ für $i \leq k$: klar, da $w, w' \in S_i \backslash \Omega_{i-1}$

Fall $w \in S_i \cup \{s_i\}, w' \in S_j \cup \{s_j\}$ mit $i < j \leq k+1$:

Dann $w \lhd w'$, da $S_i \cup \{s_i\} \lhd S_j \cup \{s_j\}$. Außerdem gilt, $w \lhd_{lex} w'$, denn falls $w' \trianglelefteq_{lex} w$, dann $w' \in \trianglelefteq_{lex}(s_i) \trianglelefteq s_i \lhd S_j \ni w'$ ein Widerspruch.

Fall $w \in S_i \cup \{s_i\}, w' \in R$ mit $i \leq i_{max}$: analog zu vorhergehendem Fall

Fall $w, w' \in R \cap \Omega_{i_{max}}$: klar

Fall $w, w' \in R \backslash \Omega_{i_{max}}$: klar

□

Lemma 4.3.7:

Genügt $\lhd$ den Definitionen 4.1.1 und 4.3.1, dann gilt
$\forall w \in \Omega_k : \trianglelefteq_{lex}(w) \lhd w \vee \exists w' \in \Omega_{f_k} : w' \lhd w$.

Beweis:

Fall $w = s_i$: klar, da $\trianglelefteq_{lex}(s_i) \subseteq \bigcup_{j=0}^{i-1} (S_j \cup \{s_j\}) \cup S_i \lhd s_i$

Fall $w \in S_i \backslash \Omega_{i-1}$:

es gilt $\trianglelefteq_{lex}(w) \lhd w$, da

$\trianglelefteq_{lex}(w) \backslash S_i = \trianglelefteq_{lex}(w) \cap \bigcup_{j=0}^{i-1} (S_j \cup \{s_j\}) \overset{\text{Def. 4.3.1(1)}}{\lhd} w$

und

$\trianglelefteq_{lex}(w) \cap (S_i \backslash \Omega_{i-1}) \lhd w$, da $\trianglelefteq_{lex}(w) \cap (S_i \backslash \Omega_{i-1}) \lhd_{lex} w$ und Def. 4.3.1(3)

und

$\trianglelefteq_{lex}(w) \cap (S_i \cap \Omega_{i-1}) \lhd w$, denn seien $u \in \trianglelefteq_{lex}(w) \cap (S_i \cap \Omega_{i-1})$ und $w' \in \Omega_{i-1} \cap anc(w)$, dann ist $u \lhd_{lex} w$ und deshalb auch $u \lhd_{lex} w'$. Nach Def. 4.3.1(2) folgt, dass $u \lhd w'$ und da $\lhd$ Traversierungsordnung, damit auch $u \lhd w$.

Fall $w \in S_i \cap \Omega_{i-1}$ $(\Rightarrow i \geq 1)$:

Es gilt, dass $s_{i-1} \lhd w$. Desweiteren ist $s_{i-1} \in \Omega_{f_{i-1}}$. Damit gilt der zweite Teil der Disjunktion.

Fall $w \in R \cap \Omega_{i_{max}}$: Analog zu vorangehendem Fall, mit $s_{i_{max}}$ statt s_{i-1}. Wieder gilt der zweite Teil der Disjunktion.

Fall $w \in R \backslash \Omega_{i_{max}}$:

$\lhd_{lex}(w) \backslash R = \lhd_{lex}(w) \cap \bigcup_{j=0}^{i_{max}} (S_j \cup \{s_j\}) \lhd w$

und

$\lhd_{lex}(w) \backslash R \lhd w$, da $\lhd_{lex}(w) \backslash R \lhd_{lex} w$

□

Satz 4.3.8:

Genügt $\lhd$ den Bedingungen der Definitionen 4.1.1 und 4.3.1, so ist $\lhd$ ein Model von (D&B).

Beweis:

Folgt unmittelbar aus den vorangegangenen drei Lemmata.

□

Im zweiten Teil der formalen Analyse wird jetzt gezeigt, dass jedes Modell von (D&B) die in den Definitionen 4.1.1 und 4.3.1 beschriebene Struktur besitzt.

Satz 4.3.9:

Ist $\lhd$ ein Modell von (D&B), dann genügt es den Definitionen 4.1.1 und 4.3.1.

Beweis:

Sei $0 \leq i \leq i_{max}$.

1. Zeige $s_i \lhd \Omega_j$ für $f_i < j$:

   Sei $w \in \Omega_j$. Dann $w \neq s_i$. Angenommen $w \lhd s_i$, dann

   $w \lhd s_i \in \Omega_{f_i} \lhd \Omega_{f_{i+1}}$

   $\overset{\lhd Trav.ordnung}{\underset{f_i+1 \leq j, f_{f_i+1} \leq f_j}{\Rightarrow}} w \lhd \Omega_{f_j}$

   $\Rightarrow \neg\exists w' \in \Omega_{f_j} : w' \lhd w$

   $\overset{(D\&B)(3)}{\Rightarrow} \lhd_{lex}(w) \lhd w$

   Es gilt aber $s_i \lhd_{lex} w$, denn $\trianglelefteq_{lex}(s_i) \subseteq \Omega_{\leq f_i}$ und $w \notin \Omega_{\leq f_i}$

   $\Rightarrow s_i \in \lhd_{lex}(w) \lhd w$ ein Widerspruch.

2. Zeige $s_i \trianglelefteq \Omega_{\geq f_i}$ und $(w \lhd s_i \Rightarrow w \in \Omega_{<f_i})$:

   Aus (D&B)(2) folgt $s_i \trianglelefteq \Omega_{f_i}$, da $s_i \trianglelefteq_{lex} w$ für alle $w \in \Omega_{f_i}$.

   Weil $\lhd$ eine Traversierungsordnung ist, ergibt sich daraus, dass $s_i \trianglelefteq \Omega_{\geq f_i}$.

   Die zweite Aussage folgt jetzt unmittelbar.

3. Zeige $\lhd_{lex}(s_i) \lhd s_i$:

   Aus 2. und der Monotonie von f folgt, dass

   $\neg\exists w' \in \Omega \underbrace{f_{f_i}}_{>f_i} : w' \lhd s_i$

   und wegen (D&B)(3) dann $\lhd_{lex}(s_i) \lhd s_i$.

4. Zeige $\lhd_{lex}(w) \lhd w$ für $w \in \Omega_j$ mit $w \lhd s_i$ und $i \leq j < f_i$:

   Aus 2. und der Monotonie von f folgt, dass

   $\neg\exists w' \in \Omega \underbrace{f_j}_{\geq f_i} : w' \lhd w \quad (w' \lhd w \Rightarrow w' \lhd s_i \Rightarrow w' \in \Omega_{<f_i})$

   $\overset{(D\&B)(3)}{\Rightarrow} \lhd_{lex}(w) \lhd w$

5. Zeige $s_i \lhd \Omega_j \setminus \lhd_{lex}(s_i)$ für $i \leq j < f_i$:

   Sei $w \in \Omega_j \setminus \lhd_{lex}(s_i)$, dann $w \notin \lhd_{lex}(s_i)$ und $w \neq s_i$ also $s_i \lhd_{lex} w$.
   Wäre also $w \lhd s_i$, dann gülte $\lhd_{lex}(w) \lhd w$ nicht, ein Widerspruch zu 4.

6. Zeige $\Omega_{i-1} \lhd s_i$:

   $\Omega_{i-1} \overset{(D\&B)(1)}{\lhd} \Omega_{f_i}$ und $s_i \in \Omega_{f_i}$

7. Zeige 4.3.1(1), also $S_i \lhd s_i \lhd S_{i+1}$ und $S_{i_{max}} \lhd s_{i_{max}} \lhd R$:

   3. und 6. liefern $S_i \subseteq \lhd_{lex}(s_i) \cup \Omega_{i-1} \lhd s_i$.

   Außerdem ist $s_i \lhd S_{i+1}$, da aus $w \in S_{i+1}$ folgt, dass $w \in \Omega_{\geq i}$ und $w \notin \trianglelefteq_{lex}(s_i)$.
   Das bedeutet jedoch, dass $w \in \Omega_k \setminus \trianglelefteq_{lex}(s_i)$ für ein $k \geq i$ und damit nach 5., dass $s_i \lhd w$.

   Mit dem selben Argument gilt $s_{i_{max}} \lhd R$, da aus $w \in R$ folgt, dass $w \in \Omega_{\geq i_{max}}$ und $w \notin \trianglelefteq_{lex}(s_{i_{max}})$.

   Zusammengefasst gilt also: $S_i \lhd s_i \lhd S_{i+1}$ und $s_{i_{max}} \lhd R$.

8. Zeige 4.3.1(2), also $\forall w, w' \in S_i \cap \Omega_{i-1} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$ und 4.3.1(4), also $\forall w, w' \in R \cap \Omega_{i_{max}} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$:

   Klar nach (D&B)(2).

9. Zeige 4.3.1(3), also $\forall w, w' \in S_i \setminus \Omega_{i-1} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$:

   Folgt aus der Tatsache, dass für alle $w \in S_i \setminus \Omega_{i-1}$ gilt, dass $w \in \Omega_{\geq i} \cap \Omega_{<f_i}$ und $w \lhd s_i$, weshalb (wie in 4. gezeigt) $\lhd_{lex}(w) \lhd w$ ist.

10. Zeige 4.3.1(5), also $\forall w, w' \in R \setminus \Omega_{i_{max}} : w \lhd w' \Leftrightarrow w \lhd_{lex} w'$

    Für $w \in R \setminus \Omega_{i_{max}}$ gilt $w \in \Omega_j$ für ein $j > i_{max}$.
    Nach Definition von i_{max} ist $\Omega_{f_j} = \emptyset$ und damit gilt $\neg\exists w' \in \Omega_{=f_j} : w' \lhd w$.

    Also $\lhd_{lex}(w) \lhd w$ nach (D&B)(3) und somit die Behauptung.

□

Satz 4.3.10:

Genügt $\lhd$ den Definitionen 4.1.1 und 4.3.1, oder ist $\lhd$ Modell von (D&B), dann ist $\lhd$ vollständig.

Beweis:

Folgt unmittelbar aus der ersten Formel von (D&B) und den Sätzen 4.3.8 und 2.3.5.

□

4.3.2. Speicherkomplexität

Kommen wir nun zur Analyse des Speicherbedarfs der vorgestellten Algorithmen.

Seien hierzu $\lhd$ ein Modell von (D&B), $a_\lhd$ die zugehörige induzierte Traversierungsfolge und $A_\lhd$ der induzierte Traversierungslauf.

Lemma 4.3.11:

Für $a_\lhd(n) \in S_0$ gilt, $|A_\lhd(n+1)| \leq b \cdot (d+1)$,
wobei $d = max_< (\{|w| \mid w \in a_\lhd[0,\ldots,n]\})$.

Beweis:

Wegen $\Omega_{i-1} = \emptyset$ und dem 3. Punkt von Definition 4.3.1 (S.37) wird S_0 in lexikographischer Reihenfolge durchlaufen. Es gilt also, dass $\trianglelefteq_{lex}(a_\lhd(n)) \in a_\lhd[0,\ldots,n]$.

Eine wichtige Beobachtung ist, dass für alle $v \in \trianglelefteq_{lex}(a_\lhd(n)) \setminus anc(a_\lhd(n))$ gilt, dass $sons(v) \subseteq \trianglelefteq_{lex}(a_\lhd(n))$, also $sons(v) \lhd a_\lhd(n)$. Die Kinder dieser Knoten wurden also bereits expandiert und befinden sich demnach nicht mehr im Speicher.

Insgesamt sind also höchstens die Kinder der Knoten in $anc(a_\lhd(n)) \cup \{a_\lhd(n)\}$ im Speicher, also höchstens $|a_\lhd(n)| + 1$ viele.

□

Lemma 4.3.12:

Für $a_\lhd(n) \in \{s_i\} \cup S_{i+1}$ gilt, $|A_\lhd(n+1)| \leq b \cdot (d+1+b^i)$,
wobei $d = max_< (\{|w| \mid w \in a_\lhd[0,\ldots,n]\})$.

Beweis:

$w_{>i,n} := max_{lex}(\Omega_{>i} \cap a_\lhd[0,\ldots,n])$

$w_{i,n} := max_{lex}(\Omega_i \cap a_\lhd[0,\ldots,n])$

1. Zeige $a_{\lhd}[0,\ldots,n] = \Omega_{<i} \cup \left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right) \cup \left(\Omega_{>i} \cap \trianglelefteq_{lex}(w_{>i,n})\right)$:

   $\trianglelefteq_{lex}(s_i) \subseteq \trianglelefteq_{lex}(w_{>i,n}) \subseteq \Omega_{<i} \cup \left(\Omega_i \cap \trianglelefteq_{lex}(u)\right) \cup \left(\Omega_{>i} \cap \trianglelefteq_{lex}(w_{>i,n})\right)$

   ,wobei $\{u\} = anc(w_{>i,n}) \cap \Omega_i$ und $\Omega_i \cap \trianglelefteq_{lex}(u) \subseteq \Omega_i \cap \trianglelefteq_{lex}(w_{i,n})$.

   Damit gilt $\bigcup_{j=0}^{i} S_j \cup \{s_j\} \subseteq \Omega_{<i} \cup \left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right) \cup \left(\Omega_{>i} \cap \trianglelefteq_{lex}(w_{>i,n})\right)$.

   Für $v \in \Omega_i \cap S_{i+1}$ gilt weiter $v \trianglelefteq w_{i,n} \Rightarrow v \trianglelefteq_{lex} w_{i,n}$. Aus der Maximalität von $w_{i,n}$ folgt dann $\Omega_i \cap a_{\lhd}[0,\ldots,n] \subseteq \left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right)$

   Schließlich gilt für $v \in \Omega_{>i} \cap S_{i+1}$: $v \trianglelefteq w_{>i,n} \Rightarrow v \trianglelefteq_{lex} w_{>i,n}$. Aus der Maximalität von $w_{>i,n}$ folgt dann $\Omega_{>i} \cap a_{\lhd}[0,\ldots,n] \subseteq \left(\Omega_{>i} \cap \trianglelefteq_{lex}(w_{>i,n})\right)$

   Wir haben also $a_{\lhd}[0,\ldots,n] \subseteq \Omega_{<i} \cup \left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right) \cup \left(\Omega_{>i} \cap \trianglelefteq_{lex}(w_{>i,n})\right)$.

   $a_{\lhd}[0,\ldots,n] \supseteq \Omega_{<i} \cup \left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right) \cup \left(\Omega_{>i} \cap \trianglelefteq_{lex}(w_{>i,n})\right)$ folgt unmittelbar daraus, das $w_{>i,n}, w_{i,n} \in a_{\lhd}[0,\ldots,n]$.

2. Schauen wir uns jetzt an welche Knoten sich beim induzierten Traversierungslauf $A_{\lhd}$ zum Zeitpunkt n noch im Speicher befinden.

   Eine erste wichtige Beobachtung ist, dass für alle $v \in \trianglelefteq_{lex}(w_{>i,n}) \setminus anc(w_{>i,n})$ gilt, dass $sons(v) \subseteq \trianglelefteq_{lex}(w_{>i,n})$, also $sons(v) \lhd w_{>i,n}$. Die Kinder dieser Knoten wurden also bereits expandiert und befinden sich demnach nicht mehr im Speicher.

   Für jeden Knoten $v \in \Omega_{<i-1}$ gilt dies offensichtlich ebenfalls.

   Auch die Kinder aller Knoten $v \in \Omega_{i-1} \cap \trianglelefteq_{lex}(father(w_{i,n}))$ wurden bereits expandiert, denn $sons_{\Omega}(v) \lhd_{lex} w_{i,n}$.

   Insgesamt sind also höchstens die Kinder der Knoten in $anc(w_{>i,n}) \cup \{w_{>i,n}\} \cup \left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right) \cup \left(\Omega_{i-1} \cap \trianglerighteq_{lex}(father(w_{i,n}))\right) =: F$ im Speicher.

   Anders ausgedrückt gilt:

   $A_{\lhd}(n+1) \subseteq \bigcup_{v \in F} sons_{\Omega}(v)$

   Nun ist

   $|anc(w_{>i,n}) \cup \{w_{>i,n}\}| = |w_{>i,n}| + 1 \leq d + 1$

   und

   $\left|\left(\Omega_i \cap \trianglelefteq_{lex}(w_{i,n})\right) \cup \left(\Omega_{i-1} \cap \trianglerighteq_{lex}(father(w_{i,n}))\right)\right| \leq b^i$.

   Das bedeutet $|A_{\lhd}(n+1)| \leq b \cdot (d + 1 + b^i)$.

□

Lemma 4.3.13:

Für $a_\lhd(n) \in \{s_{i_{max}}\} \cup R$ gilt, $|A_\lhd(n+1)| \leq b \cdot (d+1+b^{i_{max}})$,
wobei $d = max_< (\{|w| \mid w \in a_\lhd[0,\ldots,n]\})$.

Beweis:

Analog zu vorigem Lemma, mit R statt S_{i+1}

□

Satz 4.3.14:

Sei $d \in \mathbb{N}$.

Dann gilt:

$$m_{A_\lhd}(d) \leq \begin{cases} b \cdot (d+1) & \text{falls } d < f_0 \\ b \cdot (d+1+b^i) & \text{sonst} \end{cases}$$

wobei $i := \arg\max_j \{f_j \mid f_j \leq d\}$, also dass j, für das f_j maximal ist

Beweis:

Für $0 \leq i \leq i_{max}$ sei $n_i := first_{a_\lhd}(s_i)$.

Sei $f_i \leq d < f_{i+1}$ für ein $0 \leq i < i_{max}$.

Dann gilt für $n \in \alpha, depth_{A_\lhd}(n) \leq d$, dass $n \leq n_{i+1}$, da $depth_{A_\lhd}(n_{i+1}) = f_{i+1} > d$.

Also $a_\lhd(n) \in \bigcup_{j=0}^{i} (S_j \cup \{s_j\}) \cup S_{i+1}$ und wegen Lemma 4.3.12

$m_{A_\lhd}(d) = max_< \left(\left\{|A_\lhd(n)| \mid n \in \alpha, depth_{A_\lhd}(n) \leq d\right\}\right) \leq max_{0 \leq j \leq i} (b \cdot (d+1+b^j)) =$
$b \cdot (d+1+b^i)$

Ist $i_{max} < \infty$ und gilt $f_{i_{max}} \leq d$, dann gilt $a_\lhd(n) \in R$ und deshalb analog zu oben nur mit Lemma 4.3.13 statt Lemma 4.3.12

$m_{A_\lhd}(d) \leq b \cdot (d+1+b^{i_{max}})$

Außerdem gilt für $d < f_0$, dass $a_\lhd(n) \in S_0$ und damit wieder analog zu oben mit Lemma 4.3.11

$m_{A_\lhd}(d) \leq b \cdot (d+1)$

□

Korollar 4.3.15:

Die Speicherkomplexität des von $\lhd$ induzierten Algorithmus ist

$$M(d) \leq \begin{cases} b \cdot (d+1) & \text{falls } d < f_0 \\ b \cdot (d+1+b^i) & \text{sonst} \end{cases}$$

wobei $i := argmax_j \{f_j \mid f_j \leq d\}$.

4.3.3. Speicherkomplexität der Familie

In Abschnitt 4.2 auf Seite 34 wurde eine Familie von Algorithmen in einem Paramameter $0 \leq c \leq \infty$ vorgestellt. Dabei wurde für $1 \leq c < \infty$ behauptet, dass der zugehörige Algorithmus die Speicherkomplexität $O(d^c)$ besäße. Dies wird nun bewiesen.

Satz 4.3.16:

Seien $1 \leq c < \infty$, $f_i = f_{c,i} = \lfloor b^{\frac{i}{c}} \rfloor + i$ und $\lhd_c$ die zugehörige Ordnung. Der von $\lhd_c$ induzierte Algorithmus hat die Speicherkomplexität

$M_c(d) \leq b \cdot (d + 1 + d^c)$

Beweis:

Für $d < f_0$ gilt $M_c(d) \overset{\text{Kor. 4.3.15}}{\leq} b \cdot (d+1)$.

Für $f_0 \leq d < f_1$ gilt

$M_c(d) \overset{\text{Kor. 4.3.15}}{\leq} b \cdot (d + 1 + b^0) = b \cdot (d + 1 + 1) \overset{0<f_0 \leq d}{\leq} b \cdot (d + 1 + d) \leq b \cdot (d + 1 + d^c)$

Für $f_1 \leq d$ gilt $f_i \leq d$ für $i := \arg\max_j \{f_j \mid f_j \leq d\}$ mit $i \geq 1$.

Dann $\lfloor b^{\frac{i}{c}} \rfloor + i \leq d$ also da $i \geq 1$ auch $b^{\frac{i}{c}} \leq d$ und damit $b^i \leq d^c$.

Es folgt mit Korollar 4.3.15, dass

$M_c(d) \leq b \cdot (d + 1 + b^i) \leq (d + 1 + d^c)$

□

Im Fall $c = 0$ erhalten wir, wie beim Tiefendurchlauf zu erwarten, dass derselbe lineare Speicherkomplexität in der maximal erreichten Tiefe besitzt.

Satz 4.3.17:

Seien $c = 0$, $f_i = f_{0,i} = \lfloor b^{\frac{i}{0}} \rfloor + i = \infty$ und $\lhd_0$ die zugehörige Ordnung. Der von $\lhd_0$ induzierte Algorithmus hat die Speicherkomplexität

$M_0(d) \leq b \cdot (d + 1)$

Beweis:

Es gilt $f_0 = \infty$, also $d < f_0$ für alle $d \in \mathbb{N}$ und damit

$M_0(d) \overset{\text{Kor. 4.3.15}}{\leq} b \cdot (d+1)$.

□

5. Implementierung

Bis zu diesem Punkt haben wir die Idee des vorgeschlagenen Verfahrens kennengelernt und auch seine Eigenschaften analysiert. Wir müssen allerdings noch zeigen, dass sich das Verfahren auch sinnvoll implementieren lässt. Zwei Möglichkeiten hierfür werden in diesem Kapitel vorgeschlagen.

Ansatz 1 orientiert sich stark an den Formeln von (D&B) auf Seite 35, weshalb deren Gültigkeit leicht zu erkennen ist. Allerdings muss Möglichkeit 1 den maximalen Verzweigungsgrad b kennen.

Ansatz 2 kann selbst auf dieses Wissen verzichten, setzt die Formeln von (D&B) dafür aber nicht mehr strikt um. Man kann sich jedoch überlegen, dass die Abweichungen weder die Vollständigkeit noch die Speicherkomplexität des Verfahrens beeinträchtigen.

Beide Implementierungen werden in ihrem Pseudecode eine Speicherstruktur Ω verwenden, auf deren Inhalt in der Form $\Omega[i]$ für einen Index i zugegriffen wird. Der Name ist bewusst gewählt, sind die durch $\Omega[i]$ referenzierten Knoten doch gerade die Knoten aus Ω_i die sich aktuell im Speicher befinden. Anders ausgedrückt könnte man sagen, dass für einen gewissen Zeitpunkt n gilt, dass $\Omega[i] = \Omega_i \cap A_{d\&b}(n)$. Diese Speicherstruktur lässt sich als Liste von Listen (jeweils doppelt verkettet) implementieren.

Die lexikographische Ordnung innerhalb jeder dieser Listen $\Omega[i]$ wird einfach dadurch gewährleistet, dass neue Knoten gerade in dieser Reihenfolge entstehen und am Ende der Liste eingefügt werden. Dies ist der Fall, da die Methode zum Expandieren eines Knotens immer nur in der Form 'expand(**first of** Ω[i])' aufgerufen wird, also immer nur das lexikographisch kleinste Element einer Liste expandiert wird. Insofern meint '**first of** Ω[i]' zwar einerseits das lexikographisch kleinste Element von Ω[i], andererseits aber schlichtweg den Anfang der Liste. Außerdem hat die zuvor genannte Eigenschaft, dass immer nur das lexikographisch kleinste Element einer Liste expandiert wird, zur Folge, dass die zweite Formel von (D&B) auf jeden Fall erfüllt wird. Schließlich bedeutet die Formel nichts anderes, als dass jedes Level in lexikographischer Reihenfolge durchlaufen wird. Außerdem ist dadurch garantiert, dass für jeden vom Tiefendurchlauf expandierten Knoten der erste Teil der Disjunktion in der dritten Formel von (D&B) erfüllt ist. Dieser besagt, dass ein Knoten expandiert werden darf, wenn alle im Baum links von ihm liegenden (lexikographisch kleineren) Knoten bereits expandiert sind.

Hier der Pseudocode für die Methode zum Expandieren eines Knotens. Sie wird von beiden Implementierungsvarianten benutzt.

```
expand(w)
    Ω[|w|] := Ω[|w|] \ {w}
    Ω[|w| + 1] := Ω[|w| + 1] ∪ sons(w)
    unexpandedNodes := unexpandedNodes − 1 + |sons(w)|
end
```

5.1. Ansatz 1, Tiefenschranken

Wie in den einleitenden Worten dieses Kapitels angekündigt wird sich die kommenende Implementierung stark an den Formeln von (D&B) orientieren. Dafür ist es gut sich zunächst klar zu machen, was die Formeln von (D&B) informell ausgedrückt aussagen.

Die erste Formel $\Omega_k \lhd_{d\&b} \Omega_{f_{k+1}}$ bedeutet, dass kein Knoten aus $\Omega_{f_{k+1}}$ expandiert wird bevor alle Knoten von Ω_k expandiert sind. Es handelt sich also um eine Beschränkung der Tiefe des Tiefendurchlaufs.

Die dritte Formel besteht aus einer (quantifizierten) Disjunktion. Der zweite Teil der Disjunktion betrifft den Breitendurchlauf. Er sagt aus, dass ein Knoten expandiert werden darf, wenn zuvor bereits ein Knoten in ausreichend großer Tiefe (vom Tiefendurchlauf) expandiert wurde. Da nun für alle Knoten einer der beiden Teile der Disjunktion erfüllt sein muss, ist dies gleichbedeutend mit einer Beschränkung der Tiefe das Breitendurchlaufs.

Diese Interpretation der Formeln als wechselseitige Tiefenschranken wird von der hier vorgestellten Implementierung umgesetzt. Die Tiefenschranken sind im Code deutlich als b_{depth} und $b_{breadth}$ zu erkennen. Sie werden immer dann vergrößert (der vierte Fall in der while-Schleife), wenn ein sogenannter “Pivot-Knoten” expandiert wird. Diese Pivot-Knoten sind die s_i aus Definition 4.1.1. Sie sind in den Bildern im Abschnitt 4.1 gut erkennbar.

In den ersten drei Fällen der while-Schleife ist die Einhaltung der Tiefenschranken offensichtlich. Im vierten Fall gilt Folgendes: Da $d_{breadth} = b_{breadth}$ hat der Breitendurchlauf gerade die Bearbeitung eines Levels beendet und $d_{breadth}$ um 1 inkrementiert. Die strenge Monotonie von f hat zur Folge, dass $\mathrm{f}(d_{breadth}) > b_{depth}$. Der Tiefendurchlauf darf also mindestens einen Schritt in die Tiefe vordringen. Damit werden beim Aufruf von depthStep() die Tiefenschranken eingehalten.

Die nachstehende Implementierung hat allerdings einen Nachteil. Die von ihr benutzte Funktion f hängt vom maximalen Verzweigungsgrad b des Baumes ab. Ist dieser unbekannt ist die hier vorgestellte Implementierungsvariante nur eingeschränkt geeignet. Zwar kann man sie dadurch reparieren, dass man der Funktion f einfach stets den größten bisher aufgetretenen Verzweigungsgrad zugrunde legt. Trotzdem wäre eine Implementierung wünschenswert, die sich überhaupt nicht mit dem Verzweigungsgrad des Baumes befassen muss. Eine solche Implementierung wird im folgenden Abschnitt vorgestellt.

```
Ω[0] := {ε}
unexpandedNodes := 1

d_depth   := 0      (*current depth of depth-first-traversal*)
d_breadth := 0      (*current depth of breadth-first-traversal*)
b_depth   := f(0)   (*current depth bound of depth-first-traversal*)
b_breadth := 0      (*current depth bound of breadth-first-traversal*)

depth_breadth_traversal()
    while unexpandedNodes > 0 do
        if d_depth < b_depth and d_breadth < b_breadth then do
            one of depthStep() or breadthStep()
        else if d_depth < b_depth then do
            depthStep()
        else if d_breadth < b_breadth then do
            breadthStep()
        else do (*d_depth = b_depth and d_breadth = b_breadth*)
            depthStep()
            if d_depth > b_depth do (*pivotvertex has been expanded*)
                ++ b_breadth
                b_depth := f(d_breadth)
            end
        end
    end
end

depthStep()
    if Ω[d_depth] ≠ ∅ then do
        expand(first of Ω[d_depth])
        ++ d_depth
    else do
        -- d_depth (*backtrack*)
    end
end

breadthStep()
    if Ω[d_breadth] ≠ ∅ then do
        expand(first of Ω[d_breadth])
        d_depth := max(d_depth, d_breadth + 1)
    else do
        ++ d_breadth
    end
end
```

5.2. Ansatz 2, Credit-Funktion

Ziel der nun folgenden Implementierung ist es, das Verhalten der vorangegangenen Implementierung möglichst genau nachzubilden, ohne dabei jedoch den maximalen Verzweigungsgrad b zu verwenden.

Um dieses Ziel zu erreichen sind zwei Beobachtungen von Bedeutung:

Erstens, ist für den von (D&B) beschriebenen Algorithmus nicht nur die Zahl der im Speicher befindlichen Knoten polynomiell beschränkt in der maximalen bisher erreichten Tiefe des Tiefendurchlaufs, sondern auch die Zahl der vom Breitendurchlauf bis dahin expandierten Knoten (Konkret ist die zweite Schranke höchstens doppelt so groß wie die erste und zwar unabhängig von b).

Zweitens bedeutet eine polynomielle Beschränkung der Zahl der vom Breitendurchlauf expandierten Knoten unmittelbar auch eine polynomielle Beschränkung der Zahl der im Speicher befindlichen Knoten.

Die naheliegende Idee ist, die polynomielle Beschränkung des Speicherbedarfs in der maximal erreichten Tiefe nicht durch eine Beschränkung der *Tiefe* des Breitendurchlaufs, sondern durch eine Beschränkung der Zahl der durch ihn *expandierten Knoten* zu gewährleisten.

Auch die Beschränkung der Tiefe des Tiefendurchlaufs durch die Tiefe des Breitendurchlaufs lässt sich ersetzen. Erzwingt man nämlich, dass die Zahl der vom Breitendurchlauf expandierten Knoten nicht nur nach oben, sondern auch nach unten polynomiell durch die maximale bisherige Tiefe des Tiefendurchlaufs beschränkt ist, so ergibt sich daraus auch eine passende Beschränkung der Tiefe des Tiefendurchlaufs.

Im nachstehenden Pseudocode ist $c_{breadth}$ die Zahl der Knoten die der Breitendurchlauf noch expandieren darf. $c_{breadth}$ wird immer dann um $\mathrm{credit}\,(\mathrm{max_}d_{depth})$ erhöht, wenn der Tiefendurchlauf eine neue maximale Tiefe erreicht. Ist $c_{breadth}$ groß genug, dass der Breitendurchlauf sein aktuelles Level vollständig bearbeiten kann, gilt also $c_{breadth} \geq |\Omega\,[d_{breadth}]|$, dann wird der Tiefendurchlauf angehalten, bis das geschehen ist, was die Tiefe des Tiefendurchlaufs beschränkt.

Hierbei muss $\mathrm{credit}\,(\mathrm{d})$ so gewählt sein, dass $\sum_{\mathrm{i}=0}^{\mathrm{d}} \mathrm{credit}\,(\mathrm{i}\,) = O(\mathrm{d}^c)$.

Am einfachsten lässt sich das natürlich erreichen, indem man $\mathrm{credit}\,(0) = 0$ und $\mathrm{credit}\,(\mathrm{d}) = \mathrm{d}^c - (\mathrm{d}-1)^c$ setzt für $\mathrm{d} > 0$. Allerdings ist diese Möglichkeit nur eingeschränkt effizient implementierbar.

Die Definition $\mathrm{credit}\,(\mathrm{d}) = (2^c - 1) \cdot 2^{(c-1)\lfloor \log_2(\mathrm{d}) \rfloor}$ ist durch die Verwendung von Shift-Operationen sehr effizient implementierbar und erfüllt ebenfalls die Voraussetzungen (Siehe Anhang A).

Das Verhalten dieser Implementierung entspricht auf vollständigen Bäumen nahezu dem der vorangegangen Implementierung. Auf weniger dichten Bäumen schreitet der Breitendurchlauf jedoch schneller fort als zuvor. Neben der Tatsache, dass man sich nicht mehr mit dem maximalen Verzweigungsgrad des Baumes befassen muss, ist dieser Nebeneffekt ein weiterer Vorteil der hier vorgestellten Implementierung.

```
Ω[0] := {ε}
unexpandedNodes := 1

d_depth      := 0      (*current depth of depth-first-traversal*)
d_breadth    := 0      (*current depth of breadth-first-traversal*)
max_d_depth  := 0      (*maximum depth reached*)
c_breadth    := 0      (*credit of breadth-first-traversal*)

depth_breadth_traversal()
    while unexpandedNodes > 0 do
        if c_breadth < |Ω[d_breadth]| and c_breadth > 0 then do
            one of depthStep() or breadthStep()
        else if c_breadth < |Ω[d_breadth]| then do
            depthStep()
        else if c_breadth > 0 then
            breadthStep()
        else do
            (*impossible case*)
        end
    end
end

depthStep()
    if Ω[d_depth] ≠ ∅ then do
        expand(first of Ω[d_depth])
        if d_depth = max_d_depth then do
            ++ max_d_depth
            c_breadth := c_breadth + credit(max_d_depth)
        end
        ++ d_depth
    else do
        -- d_depth (*backtrack*)
    end
end

breadthStep()
    if Ω[d_breadth] ≠ ∅ then do
        expand(first of Ω[d_breadth])
        -- c_breadth
        d_depth := max(d_depth, d_breadth + 1)
    else do
        ++ d_breadth
        if(d_breadth = max_d_depth) then do
            c_breadth := 0
        end
    end
end
```

6. Zusammenfassung, Anwendungen und weitere Ideen

In Kapitel 2 wurde ein Formalismus zur Beschreibung und Analyse von uninformierten Suchalgorithmen bzw. Traversierungsalgorithmen vorgestellt. Dieser Formalismus birgt eine Reihe Vorteile in sich.

Erstens ermöglicht er die übersichtliche Formulierung komplexer Aussagen über die Algorithmen. Dies ist zwingend erforderlich, wenn man Algorithmen analysieren möchte, die einen komplizierteren Aufbau haben als die einfachen Standardverfahren. Möchte man exakte Beweise führen, stößt man ohne einen Formalismus bereits bei diesen einfachen Algorithmen an Grenzen.

Z.B. ist die Unvollständigkeit des Tiefendurchlaufs auf (den meisten) unendlichen Bäumen intuitiv sehr einsichtig. Versucht man die Aussage jedoch ohne Verwendung eines Formalismus exakt zu beweisen, stellt sich dies schnell als äußerst mühsam heraus. Allein zu formulieren, auf welchen Bäumen der Tiefendurchlauf unvollständig ist erweist sich als schwierig und bei der Begründung warum die Tiefensuche auf einem dieser Bäume tatsächlich unvollständig ist, kann man es ab einem gewissen Punkt nicht mehr vermeiden doch an das intuitive Algorithmusverständnis des Lesers zu apellieren.

Speziell in Abschnitt 4.3 haben wir ausgiebig die genannte Fähigkeit des Formalismus genutzt.

Zweitens erleichtert der Formalismus die Analyse der Vollständigkeit. Dies hängt mit dem ersten Punkt zusammen, ermöglicht dieser doch die Formulierung verschiedener Charakterisierungen der Vollständigkeit. Mithilfe dieser Charakterisierungen lassen sich Vollständigkeits- bzw. Unvollständigkeitsbeweise dann deutlich beschleunigen.

Drittens verschlankt der Formalismus auch dadurch die Formulierung und den Beweis von Aussagen über Traversierungsalgorithmen, dass er alle nicht unbedingt nötigen Details ausblendet. Selbst auf der „ausführlichsten“ Beschreibungsebene, den Traversierungsalgorithmen beschäftigt man sich an keiner Stelle mit der Implementierung gewisser Speicherstrukturen und auch Fragen zur Berechenbarkeit werden ignoriert. Diese Fokusierung hilft dabei sich auf die für einen Traversierungsalgorithmus zunächst entscheidenderen Fragen der Vollständigkeit und der Speicherkomplexität zu konzentrieren und vermeidet unnützen syntaktischen Overhead in den entsprechenden Beweisen.

6.1. Der D&B-Durchlauf im Überblick

Der in Kapitel 4 vorgestellte D&B-Durchlauf bzw. seine Familie haben trotz der vergleichsweise einfachen Idee auf der sie basieren einige sehr interessante Eigenschaften die hier nochmal zusammengefasst werden sollen. Wie sich diese Eigenschaften konkret nutzen lassen, darauf geht dann der Abschnitt 6.4 ein.

Grundidee des D&B-Durchlaufs ist ja die Verzahnung eines Tiefen- und eines Breitendurchlaufs. Bereits auf Seite 34 wurde auf einen Effekt eingegangen, der in Verbindung damit entsteht. Es ist nämlich so, dass sich der D&B-Durchlauf auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf verhält. Dieses Verhalten beruht allerdings nicht auf einer bewussten

Unterscheidung endlicher oder unendlicher Bäume, die häufig gar nicht möglich ist. Trotzdem erhält man den Effekt, dass man ein auf endlichen Bäumen „günstiges“ Verfahren, den Tiefendurchlauf, und ein auf unendlichen Bäumen „günstiges“ Verfahren, den Breitendurchlauf, so miteinander kombinieren kann, dass sich der entstehende Algorithmus auf den entsprechenden Bäumen im wesentlichen wie das als günstiger erachtete Verfahren verhält.

Natürlich kann man darüber steiten ob der Breitendurchlauf wirklich „günstig“ für unendliche Bäume ist. Das dahinter steckende Grundprinzip ist jedoch sehr attraktiv und deshalb wird in Abschnitt 6.3 eine ähnliche Kombination angeregt, bei der statt des Breitendurchlaufs der Iterative Tiefendurchlauf zum Einsatz kommt.

Eine weitere Eigenschaft des D&B-Durchlaufs ist seine Familie bzw. die Tatsache, dass man die Menge des für die Vollständigkeit aufgewandten Speichers durch einen einfachen Parameter kontrollieren kann (siehe Abschnitt 4.2 auf Seite 34). Dass hierbei der Tiefen- und der Breitendurchlauf die beiden Extreme der Familie darstellen ist natürlich besonders aus theoretischer Sicht interessant.

Außerdem ist der D&B-Durchlauf vollständig, speichereffizient und kommt ohne Mehrfachexpansionen aus. Nachstehend wird der D&B-Durchlauf bezüglich dieser Punkte in die bereits von Seite 7 bekannte Tabelle eingeordnet.

	Vollständigkeit		Speicherkomplexität		Mehrfachexpansion	Laufzeitkomplexität	
	endl.	unendl.	beste	schlechteste		beste	schlechteste
Tiefendurchlauf	ja	nein	$O(d)$	$O(d)$	nein	$O(n)$	$O(n)$
Iterative Ausweitung	ja	nein	$O(d)$	$O(d)$	ja	$O(n)$	$O(b*n)$
Breitendurchlauf	ja	ja	$O(d)$	$O(2^d)$	nein	$O(n)$	$O(n)$
Iterativer Tiefendurchlauf	ja	ja	$O(d)$	$O(d)$	ja	$O(n)$	$O(n^2)$
D&B-durchlauf $1 \leq c < \infty$	ja	ja	$O(d)$	$O(d^c)$	nein	$O(n)$	$O(n)$

Schließlich wurde in Kapitel 5 gezeigt, dass sich der D&B-Durchlauf und seine Familie mit vertretbarem Aufwand implemtieren lassen. Natürlich ist die Implementierung und insbesondere die benötigte Speicherstruktur komplizierter als die für z.B. den Tiefen- oder den Breitendurchlauf. Insgesamt sind die benötigten Strukturen und Berechnungen aber weiterhin einfach.

Dies lässt erwarten, dass der D&B-Durchlauf auch was die tatsächliche Laufzeit betrifft ähnlich schnell arbeitet, wie Tiefen- oder Breitendurchlauf. Dies müsste allerdings noch experimentell bestätigt werden.

6.2. Verbesserungen

Im Kapitel 4 werden die bei der Definition des D&B-Durchlaufs benutzten Funktionen f_i folgendermaßen beschränkt (siehe S.31 u. S. 36):

$$f_i > i$$
$$f_{i+1} > f_i$$

Eigentlich würde für das Funktionieren des Algorithmus auch $f_{i+1} \geq f_i$ ausreichen. Da dies allerdings die Definitionen und Beweise weiter verkomplizieren würde, wurde darauf verzichtet. Die Änderung wäre jedoch wünschenswert. Die Bedingung $f_{i+1} > f_i$, also die Forderung *strenger* Monotonie führt nämlich dazu, dass man an sich praktische Funktionen, die aber lediglich monoton aber nicht streng monoton sind, künstlich streng monoton macht. Der entstehende Effekt ist, dass für manche i gilt, dass $f_{i+1} = f_i + 1$. Dies ist ungünstig, weil daraus eine starke Synchronisation von Tiefen- und Breitendurchlauf folgt.

Man beachte, dass die in Abschnitt 5.2 auf Seite 49 vorgeschlagene Implementierungsvariante nicht von diesem Effekt betroffen ist, da sie auf eine direkte Verwendung der Tiefenschranken verzichtet

6.3. Der D&I-Durchlauf

Wie in den Abschnitten 4.1 und 6.1 angesprochen besitzt der D&B-Durchlauf die Eigenschaft in gewisser Weise zwischen Tiefen- und Breitendurchlauf zu „wählen“, abhängig davon ob der Baum endlich oder unendlich ist.

Da die Breitensuche wenig speichereffizient ist und sich deshalb auf nahezu vollständigen Bäumen praktisch keine großen Tiefen erreichen lassen, ist es üblich wenn Vollständigkeit auf unendlichen Bäumen benötigt wird, die Iterative Tiefensuche einzusetzten. Auf endlichen Bäumen führt dies insbesondere auf fast linearen Bäumen (siehe auch 6.4 Effizienz) zu deutlich höheren Laufzeiten für die gesamte Suche.

Es ist also eine naheliegende Idee, die Iterative Tiefensuche nur dann einzusetzen, wenn sie auch tatsächlich benötigt wird, also nur auf unendlichen Bäumen. Dies lässt sich in dieser Form aber nicht umsetzten. Kombiniert man jedoch Tiefensuche und Iterative Tiefensuche nach dem gleichen Prinzip wie in der D&B-Suche (bzw. D&B-Durchlauf) Tiefen- und Breitensuche, dann erhält man einen Algorithmus, der sich auf endlichen Bäumen nahezu wie eine Tiefensuche und auf unendlichen Bäumen nahezu wie ein Iterativer Tiefendurchlauf verhält. Dieser Algorithmus soll die D&I-Suche (bzw. D&I-Durchlauf) heißen.

Das oben beschriebene Verhalten ist tatsächlich wieder durch die Verwendung (der gleichen) wechselseitigen Tiefenschranken oder auch durch (um einen Faktor modifizierte) Credit-Funktionen erreichbar. Soll die Iterative Tiefensuche keine Knoten expandieren, die bereits von der Tiefensuche expandiert wurden[17], so erfordert dies eine leichte

[17] Beachte: Die Iterative Tiefensuche benutzt Mehrfachexpansionen

Modifikation des Algorithmus der Iterativen Tiefensuche und eine höchstens geringfügig komplexere Speicherstruktur.

Die Modifikation der Iterativen Tiefensuche besteht darin, dass sie ihren Suchlauf mit Tiefenschranke i nicht immer nur an einem Knoten, der Wurzel, beginnt. Stattdessen beginnt sie an jedem im Speicher befindlichen, höchstens i tiefen Knoten, der noch nicht von der Tiefensuche bearbeitet wurden einen entsprechenden Suchlauf bis zur Tiefe i.

6.4. Anwendung in der Logikprogrammierung

Der hier beginnende Abschnitt soll aufzeigen, dass die D&B-Suche (bzw. D&B-Durchlauf) bzw. der D&I-Suche (bzw. D&I-Durchlauf) nützlich sind um einige klassische Probleme der Logikprogrammierung zu lösen oder zumindest zu mildern.

Der Abschnitt ist folgendermaßen aufgebaut:
Zuerst wird an Hand eines Prolog-Programms das Problem erläutert.
Anschließend wird darauf eingegangen wie die neuen Algorithmen zu einer Lösung beitragen.

Die ersten beiden Punkte beschäftigen sich damit, weshalb der Einsatz eines vollständigen Suchverfahrens wünschenswert ist. Der dritte Punkt geht darauf ein weshalb gerade die D&B-Suche bzw. die D&I-Suche eine besonders attraktive Wahl dafür sind.

Problem: Unvollständige Aufzählungen - Nie gefundene Lösungen

Im folgenden Programm seien die natürlichen Zahlen durch eine beliebige Termdarstellung repräsentiert, zero sei die Termdarstellung der 0 und pred(X,Y) liefere zur Termdarstellung X einer natürlichen Zahl die Termdarstellung P ihres Vorgängers, falls die Zahl einen solchen hat.

Programm:

```
nat(zero).
nat(X) :- pred(X,P), nat(P).
nat2(X,Y) :- nat(X), nat(Y).
```

Anfrage:

```
nat2(X,Y). ?
```

In obigem Programm liefert die Anfrage nat(X) eine Aufzählung der natürlichen Zahlen $\mathbb{N}$. Es wäre zu erwarten, dass nat2(X,Y) eine Aufzählung von $\mathbb{N} \times \mathbb{N}$ liefert. Tatsächlich, wird aber bloß $\{0\} \times \mathbb{N}$ aufgezählt.

Ursache ist die Unvollständigkeit der Tiefensuche. Der Einsatz eines vollständigen Suchverfahrens würde zum erwarteten Ergebnis, nämlich einer Aufzählung von $\mathbb{N} \times \mathbb{N}$, führen.

In diesem Beispiel mag das Problem noch nicht allzu gravierend erscheinen. Für Semientscheidungsverfahren ist die Eigenschaft, dass jede existierende Lösung auch gefunden wird, jedoch unverzichtbar.

Problem: „UND“ bzw. „ODER“ sind nicht kommutativ

Die untenstehenden Anfragen beziehen sich auf das Programm aus dem vorangegangenen Beispiel. Hierbei realisiere das Prädikat greater die Größer-Beziehung auf den natürlichen Zahlen. Das ! am Ende der Anfrage bewirkt jeweis, dass die Lösungsuche nach dem Finden der ersten Lösung abgebrochen wird. Es wird verwendet weil man in diesem Beispiel nicht an der Lösungsmenge, sondern nur an der Existenz einer Lösung interessiert ist.

Anfrage:

```
nat2(X,_), greater(X,zero), !. ?
```

Anfrage:

```
greater(X,zero), nat2(X,_), !. ?
```

Logische Programmiersprachen streben gewöhnlich eine möglicht deklarative Semantik an. Ziel ist es, dass sich der Programmierer beim Schreiben eines Programms (zumindest zunächst) keine Gedanken um den für die Auswertung benutzten Algorithmus machen muss.

Tatsache ist aber, dass die erste Anfrage weder eine Lösung liefert, noch terminiert, die zweite Anfrage jedoch (eine vernünftige Implementierung von greater vorausgesetzt) mit der Antwort ja terminiert.

Dieser Effekt lässt sich in keinster Weise deklarativ sondern nur prozedural begründen. Insbesondere träte er bei Verwendung eines vollständigen Algorithmus gar nicht auf, da beide Anfragen mit Antwort ja terminieren würden. Die Reihenfolge der Literale hätte höchstens noch Auswirkungen auf die Effizienz der Auswertung aber nicht mehr auf ihr Ergebnis.

Zumindest für Programme ohne Negation ließe sich durch Verwendung eines vollständigen Suchalgorithmus für die Auswertung, Kommutativität von „UND“ und „ODER“ erreichen.

Problem: Effizienz

Bis jetzt haben wir nur begründet warum die Verwendung eines vollständigen Suchverfahrens bei der Auswertung sinnvoll ist. Jetzt soll gezeigt werden, dass die D&B-Suche bzw. die D&I-Suche besonders gut dafür geeignet ist.

Im Folgenden meint constant ein Prädikat, dass die Variable an genau eine fixierte aber beliebige Zahl (bzw. deren Termdarstellung) bindet.

```
even(zero).
even(X) :- pred(X,P), odd(P).
odd(X) :- pred(X,P), even(P).

constant(X), odd(X). ?
```

Das obige Programm ist nicht endrekursiv. Aber es ist in dem Sinne nahezu endrekursiv, als dass eine Anfrage der Form odd(t) für jeden Term t einen ähnlich strukturierten, nämlich fast linearen, Suchbaum erzeugt.

Endrekursionsoptimierungen basieren auf dem syntaktischen Erkennen entsprechender Programme und wählen dann eine spezielle Auswertungsmethode. Das angeführte

Programm würde von einer typischen Endrekursionsoptimierung nicht erkannt, also mit der Standard-Auswertungsmethode behandelt.

Ein auf der Tiefensuche basierendes Auswertungsverfahren kann das obige Programm in linearer Zeit in der Größe der Zahl, an die X gebunden wird, auswerten.

Ein auf der Iterativen Tiefensuche basierendes Auswertungsverfahren würde jedoch quadratische Zeit benötigen. Dies lässt sich durch gewisse Optimierungen zwar verbessern, aber das Grundproblem, dass Programme, die eigentlich besonders schnell auswertbar sein sollten, tatsächlich nur vergleichsweise langsam laufen bleibt bestehen. Das ist für die Konkurrenzfähigkeit der Programmiersprache natürlich ungünstig.

Die Alternative ist die Verwendung der D&B-Suche bzw. der D&I-Suche. Erstere verzichtet komplett auf Mehrfachauswertungen und kann das Programm dadurch genauso schnell wie die Tiefensuche ausführen. Zweitere schließt Mehrfachauswertungen nicht aus, garantiert aber, dass deren Zahl so gering ist, dass die Auswertung nur wenig verlangsamt wird und insbesondere linear bleibt. Optimierungen könnten bei der D&I-Suche weitere Verbesserungen bringen, ähnlich wie das bei der Iterativen Tiefensuche der Fall ist.

Abschließend bleibt festzuhalten, dass die D&B-Suche bzw. die D&I-Suche es voraussichtlich ermöglicht logische Programmiersprachen zu implementieren, die für die Auswertung ein vollständiges Suchverfahren mit den daraus resultierenden Vorteilen benutzen und die trotzdem noch konkurrenzfähige Laufzeiten bieten. Diese Behauptung muss sich natürlich noch in der Praxis bestätigen. Eine entsprechende Untersuchung wäre in Anbetracht der möglichen Vorteile sicherlich sinnvoll.

A. Credit-Funktion

Satz A.0.1:

Seien g und h monotone Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ und $k_1 \leq k_2 \leq k_3 \leq \ldots$ eine aufsteigende Folge in $\mathbb{N}$.

Ist außerdem $g(k_i) \leq h(k_i)$ für alle $i \in \mathbb{N}$ und existiert ein $c \in \mathbb{R}$, sodass $h(k_{i+1}) \leq c \cdot h(k_i)$ für alle $i \in \mathbb{N}$, dann gilt:

$g = O(h)$

genauer:

$g \leq c' \cdot h$ für ein $c' \leq c$

Beweis:

Zeige: $g(n) \leq c \cdot h(n)$ für alle $n \in \mathbb{N}$

Sei $k_i \leq n < k_{i+1}$. Dann gilt:

$$g(n) \overset{\text{Monotonie von } g}{\leq} g(k_{i+1}) \overset{Vor.}{\leq} h(k_{i+1}) \overset{Vor.}{\leq} c \cdot h(k_i) \overset{\text{Monotonie von } h}{\leq} c \cdot h(n)$$

□

Im Folgenden ist $\text{credit}(0) = 0$ und $\text{credit}(\mathrm{d}) = \mathrm{d}^c - (\mathrm{d}-1)^c$ für $\mathrm{d} > 0$.

Lemma A.0.2:

$$\sum_{\mathrm{i}=0}^{2^{l+1}} \text{credit}(\mathrm{i}) < \left(1 + 2^{c-(l+1)}\right) \cdot \left(2^{l+1}\right)^c$$

Beweis:

$$\sum_{i=1}^{2^{l+1}-1} (2^c - 1) \cdot 2^{(c-1)\lfloor \log_2(i) \rfloor} = \sum_{i=0}^{l} \sum_{j=1}^{2^i} (2^c - 1) \cdot 2^{(c-1)\cdot i}$$

$$= \sum_{i=0}^{l} 2^i \cdot (2^c - 1) \cdot 2^{(c-1)\cdot i} = (2^c - 1) \cdot \sum_{i=0}^{l} (2^c)^i$$

$$= (2^c - 1) \cdot \frac{(2^c)^{l+1} - 1}{2^c - 1} = \left(2^{l+1}\right)^c - 1$$

$$\sum_{i=1}^{2^{l+1}} (2^c - 1) \cdot 2^{(c-1)\lfloor \log_2(i) \rfloor} = \left(2^{l+1}\right)^c - 1 + (2^c - 1) \cdot 2^{(c-1)\cdot(l+1)}$$

$$< \left(2^{l+1}\right)^c - 1 + 2^c \cdot \left(2^{l+1}\right)^{(c-1)} = \left(1 + 2^{c-(l+1)}\right) \cdot \left(2^{l+1}\right)^c - 1$$

□

Satz A.0.3:

$\sum_{\mathrm{i}=0}^{\mathrm{d}} \mathrm{credit}\,(\mathrm{i}\,) = O(d^c)$

Beweis:

Seien $g\,(d) := \sum_{\mathrm{i}=0}^{\mathrm{d}} \mathrm{credit}\,(\mathrm{i}\,)$ und $h\,(d) := (1 + 2^{c-\lfloor \log_2(d) \rfloor})d^c$. Außerdem sei $k_i := 2^{i+1}$.

Dann gilt $g\,(k_i) \leq h\,(k_i)$ nach Lemma A.0.2.

Außerdem ist $h\,(k_{i+1}) = \left(1 + 2^{c-(i+2)}\right) \cdot \left(2^{i+2}\right)^c \leq \left(1 + 2^{c-(i+1)}\right) \cdot \left(2 \cdot 2^{i+1}\right)^c = 2^c \cdot h\,(k_i)$

Also nach Satz A.0.1 $g\,(d) = O(h) = O(d^c)$

□

B. Notation

$\Sigma^0 = \{\varepsilon\}$

$\Sigma^1 = \Sigma$

$\Sigma^n = \{uv \mid u \in \Sigma^{n-1},\ v \in \Sigma\}$

$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$

$\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n = \Sigma^* \setminus \{\varepsilon\}$

Literatur

[Bar04] R. Barták. Incomplete depth-first search techniques: A short survey. In *Proceedings of 6th Workshop on Constraint Programming for Decision and Control (CPDC2004)*, pages 7–14, 2004.

[GH90] Matthew L. Ginsberg and William D. Harvey. Iterative broadening. In *AAAI*, pages 216–220, 1990.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[Nil80] N. J. Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1980.

[Pea84] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.