Baumsuche, Partielle Ordnungen und eine neue Familie uninformierter Algorithmen

Simon Matthias Brodt

23.4.2009

Übersicht

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
3. Vorteile und mögliche Anwendungen

Übersicht - Verfahren - Eigenschaften

- Übersicht - Verfahren - Eigenschaften

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
3. Vorteile und mögliche Anwendungen

	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	$O(n)$	$O(n)$

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.

Formalismus

1 Formalismus

2 Der D&B-Durchlauf und seine Familie

3 Vorteile und mögliche Anwendungen

Repräsentation von Bäumen

1 Formalismus
- Repräsentation von Bäumen
- Traversierungsalgorithmen
  - Traversierungsfolge
  - Traversierungslauf
  - Traversierungsalgorithmus
  - Vollständigkeit
- Traversierungsordnungen
  - Traversierungsordnung
  - Charakterisierung Vollständigkeit
- Anwendung - Tiefendurchlauf

Definition (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.

Definition (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:

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

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

Notation

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

- $sons_\Omega(w) := \{w' \in \Omega \mid w' = wv \text{ für ein } v \in \Sigma\}$
  die Menge der Söhne von w in Ω
- $desc_\Omega(w) := \{w' \in \Omega \mid w' = wv \text{ für ein } v \in \Sigma^+\}$
  die Menge der Nachfahren von w in Ω
- $father(w) := w'$ *,mit* $w'v = w$ *für ein* $v \in \Sigma$
  der Vater von w für $w \neq \varepsilon$
- $anc(w) := \{w' \mid w'v = w \text{ für ein } v \in \Sigma^+\}$
  die Menge der Vorfahren von w

Traversierungsalgorithmen

1 Formalismus
- Repräsentation von Bäumen
- Traversierungsalgorithmen
  - Traversierungsfolge
  - Traversierungslauf
  - Traversierungsalgorithmus
  - Vollständigkeit
- Traversierungsordnungen
  - Traversierungsordnung
  - Charakterisierung Vollständigkeit
- Anwendung - Tiefendurchlauf

Definition (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:

$$\forall w \in a[\alpha] \setminus \{\varepsilon\} : \underbrace{father(w) \in a[\alpha]}_{father(w) \text{ kommt in } a \text{ vor}} \wedge first_a(father(w)) < first_a(w)$$

Definition (Traversierungslauf informal)

Ein Traversierungslauf ist eine Folge von Paaren aus einem Knoten und einer Knotenmenge $A : \alpha \rightarrow \Omega \times \mathscr{P}(\Omega)$ *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.*

Definition (Traversierungslauf informal)

Ein Traversierungslauf ist eine Folge von Paaren aus einem Knoten und einer Knotenmenge $A : \alpha \to \Omega \times \mathscr{P}(\Omega)$ *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.*

Definition (Traversierungslauf formal)

Ein Traversierungslauf ist eine Folge von Paaren aus einem Knoten und einer Knotenmenge $A : \alpha \to \Omega \times \mathscr{P}(\Omega)$ *für die gilt:*
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. $\forall n \in \alpha : sons_\Omega((A)_1(n)) \subseteq (A)_2(n+1)$
4. $\forall n \in \alpha \forall w \in (A)_2(n+1) : w \in (A)_2(n) \vee father(w) = (A)_1(n)$

Hierbei meint $(A)_i$ *die Funktion, die n auf die i-te Komponente von* $A(n)$ *abbildet.*

Definition (Traversierungsalgorithmus)

Ein Traversierungsalgorithmus ist eine Familie $(A_\Omega)_{\Omega \in Trav_\Sigma}$ *von Traversierungsläufen.*
Anders ausgedrückt, ordnet der Algorithmus jedem Baum $\Omega \in Trav_\Sigma$ *einen Traversierungslauf über diesen Baum zu.*

Definition (Vollständigkeit)

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

Definition (Vollständigkeit)

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

Traversierungsordnungen

1. Formalismus
   - Repräsentation von Bäumen
   - Traversierungsalgorithmen
     - Traversierungsfolge
     - Traversierungslauf
     - Traversierungsalgorithmus
     - Vollständigkeit
   - Traversierungsordnungen
     - Traversierungsordnung
     - Charakterisierung Vollständigkeit
   - Anwendung - Tiefendurchlauf

Definition (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*
- *$\forall w \in Comp_{\lhd}(\varepsilon), w' \in \Omega \setminus Comp_{\lhd}(\varepsilon) : \neg(w' \lhd w)$*

Definition (Vollständigkeit einer Ordnung)

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

Definition (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*
- $\forall w \in Comp_{\lhd}(\varepsilon), w' \in \Omega \setminus Comp_{\lhd}(\varepsilon) : \neg(w' \lhd w)$

Definition (Vollständigkeit einer Ordnung)

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

Definition (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*
- $\forall w \in Comp_{\lhd}(\varepsilon), w' \in \Omega \setminus Comp_{\lhd}(\varepsilon) : \neg(w' \lhd w)$

Definition (Vollständigkeit einer Ordnung)

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

Satz (Charakterisierung Vollständigkeit)

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

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

Satz (Charakterisierung Vollständigkeit)

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

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

Anwendung - Tiefendurchlauf

1. Formalismus
   - Repräsentation von Bäumen
   - Traversierungsalgorithmen
     - Traversierungsfolge
     - Traversierungslauf
     - Traversierungsalgorithmus
     - Vollständigkeit
   - Traversierungsordnungen
     - Traversierungsordnung
     - Charakterisierung Vollständigkeit
   - Anwendung - Tiefendurchlauf

Unvollständigkeit des Tiefendurchlaufs

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

$\exists w \in \Omega : T \lhd_{depth} w$
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 der 1. Charakterisierung der Vollständigkeit.

Möglichkeit 2

Es gilt: $S \lhd_{depth} T \lhd_{depth} U$
Die kleinste Ordinalzahl β, die diese Bedingung erfüllt ist

$$\beta = \underbrace{|S|}_{\cong k \prec \omega} + \underbrace{|T|}_{\cong \omega} + \underbrace{|U|}_{\cong \alpha \succ 0} \succ \omega$$

Also unvollständig nach der 2. Charakterisierung der Vollständigkeit.

Möglichkeit 1

$\exists w \in \Omega : T \lhd_{depth} w$
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 der 1. Charakterisierung der Vollständigkeit.

Möglichkeit 2

Es gilt: $S \lhd_{depth} T \lhd_{depth} U$
Die kleinste Ordinalzahl β, die diese Bedingung erfüllt ist

$$\beta = \underbrace{|S|}_{\cong k \prec \omega} + \underbrace{|T|}_{\cong \omega} + \underbrace{|U|}_{\cong \alpha \succ 0} \succ \omega$$

Also unvollständig nach der 2. Charakterisierung der Vollständigkeit.

Der D&B-Durchlauf und seine Familie

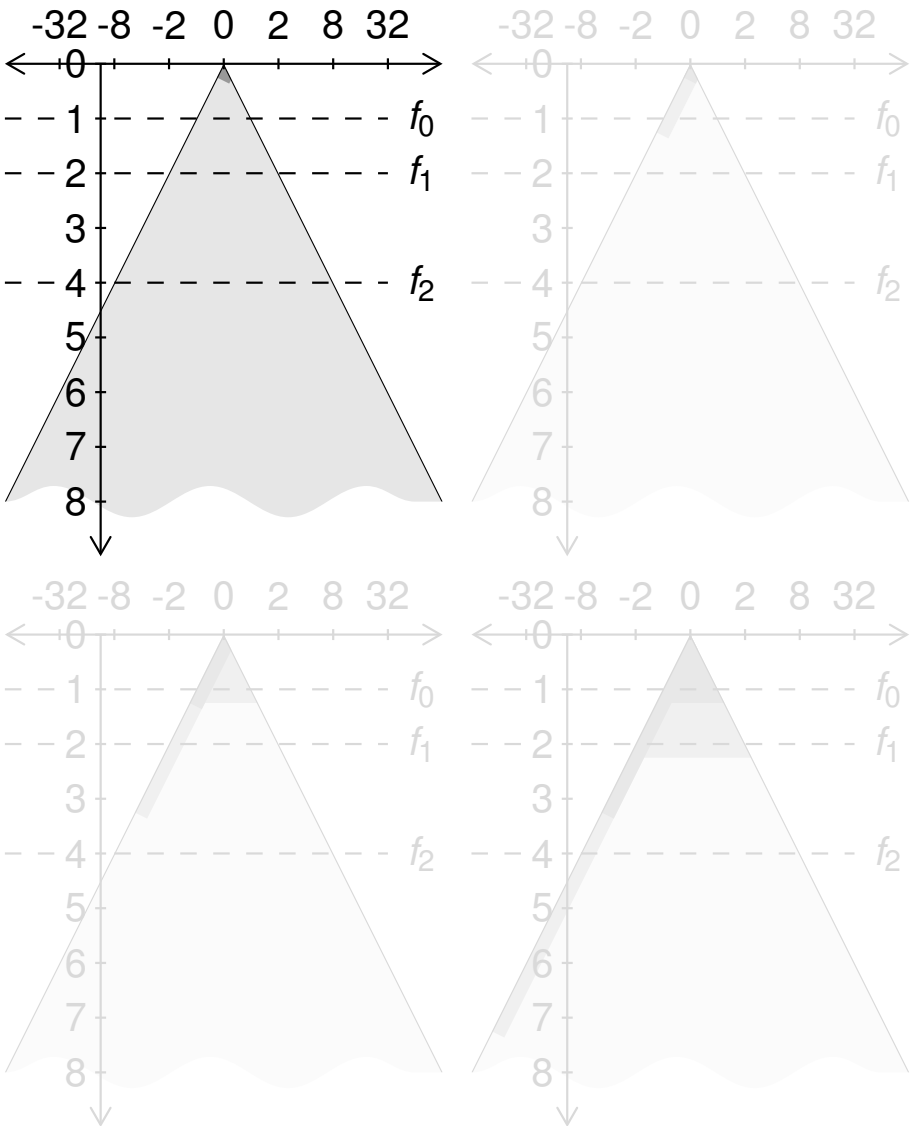
1 Formalismus

2 Der D&B-Durchlauf und seine Familie

3 Vorteile und mögliche Anwendungen

D&B-Durchlauf

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
   - D&B-Durchlauf
   - Die Familie
3. Vorteile und mögliche Anwendungen



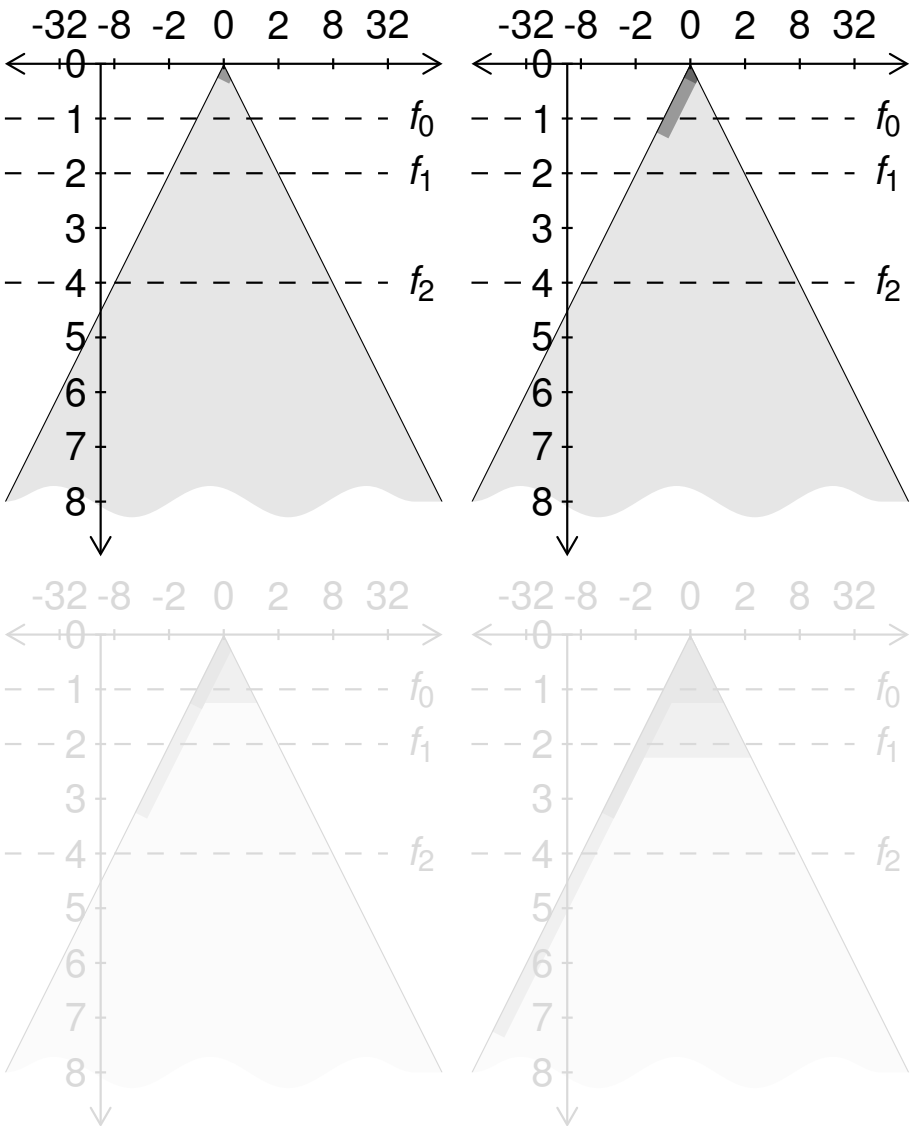
Allgemein:

$$f_i > i$$

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

Konkret im Beispiel:

$$f_i = 2^i$$



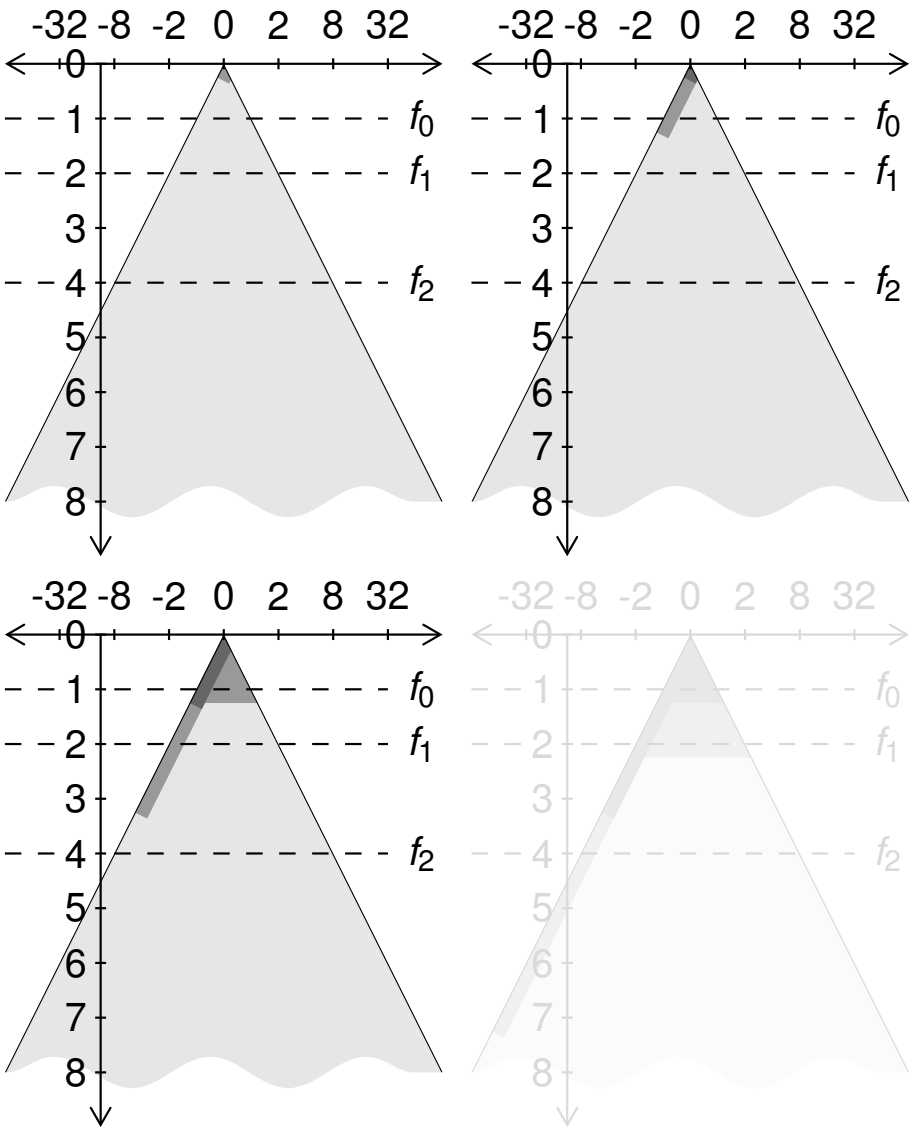
Allgemein:

$$f_i > i$$

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

Konkret im Beispiel:

$$f_i = 2^i$$



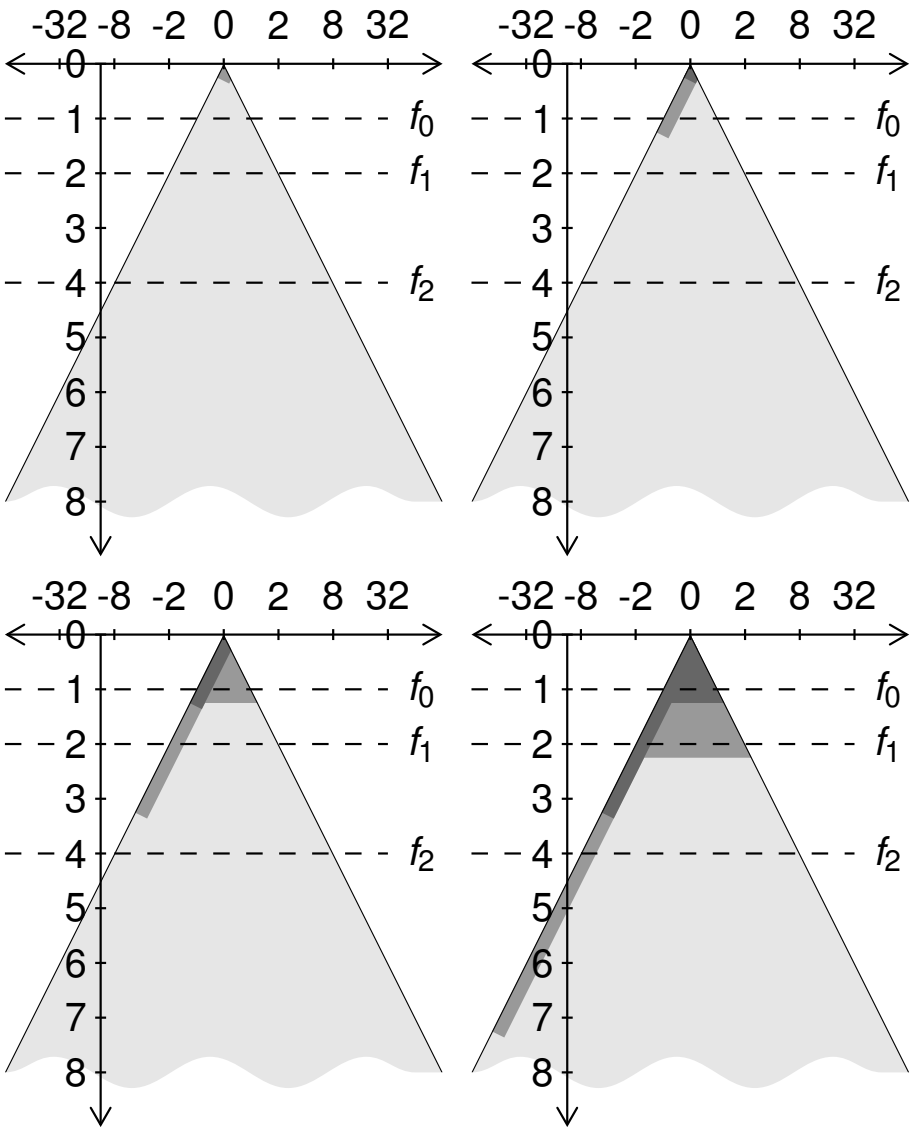
Allgemein:

$$f_i > i$$

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

Konkret im Beispiel:

$$f_i = 2^i$$



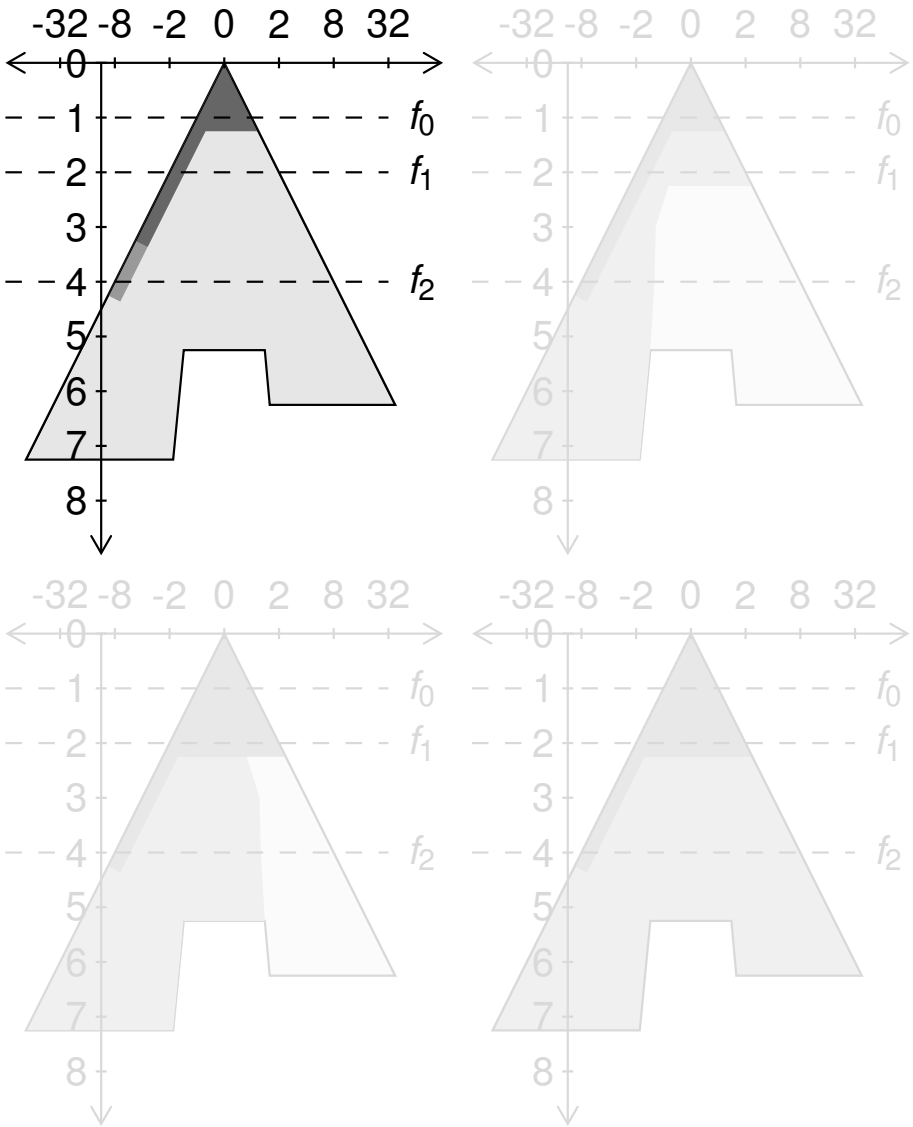
Allgemein:

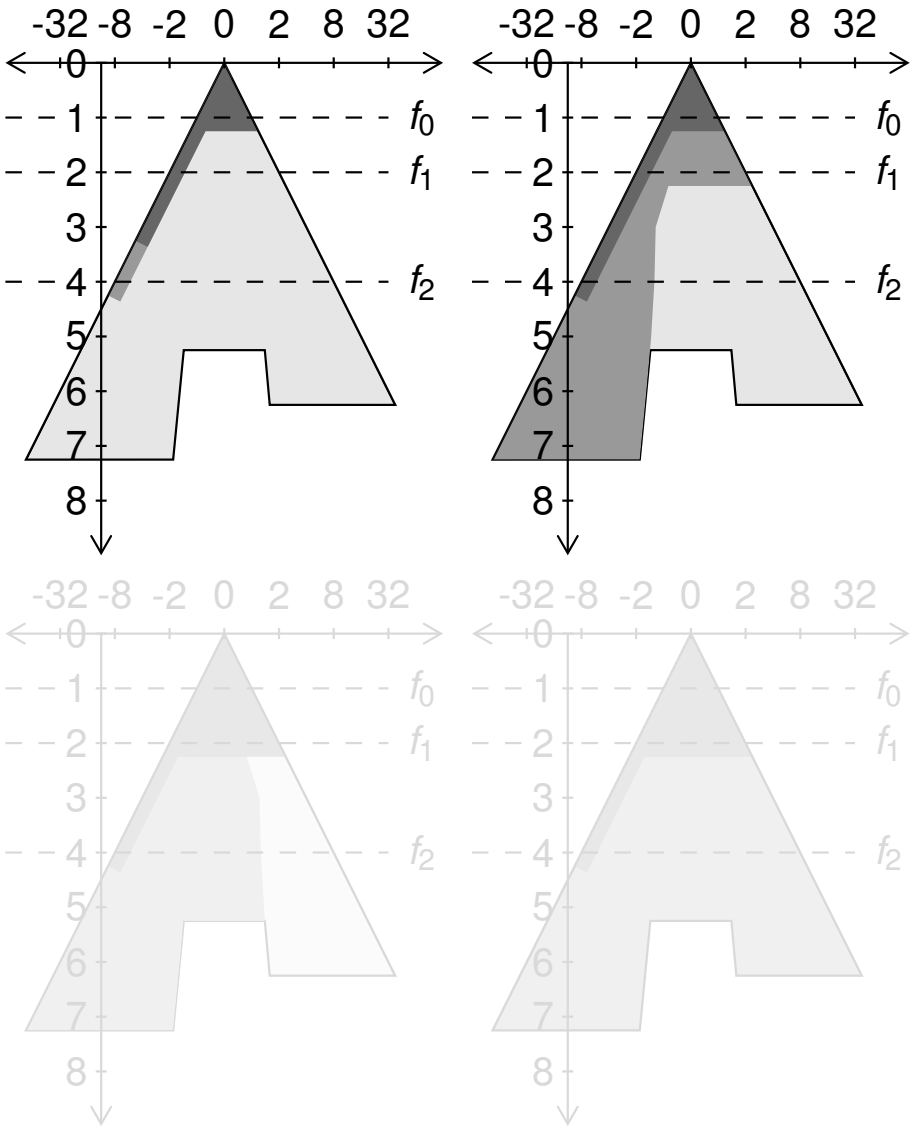
$$f_i > i$$

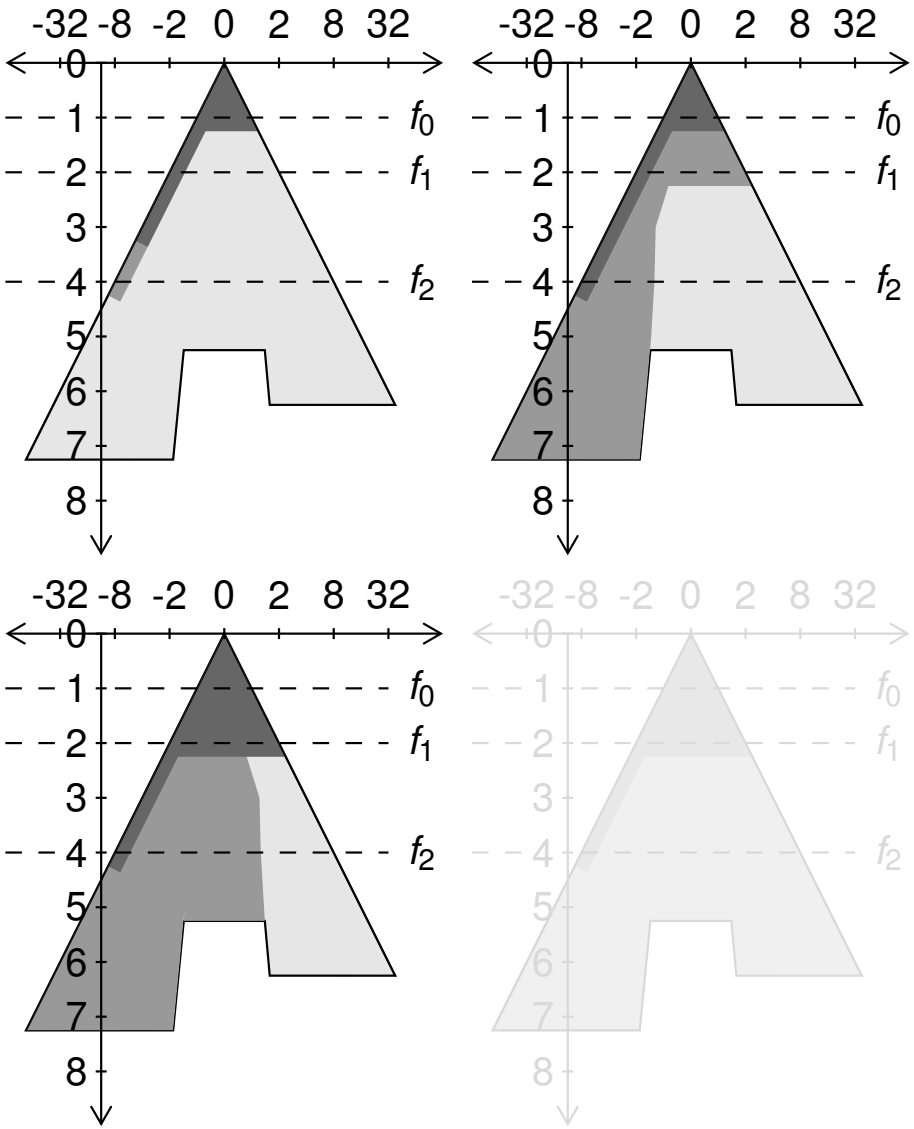
$$f_{i+1} > f_i$$

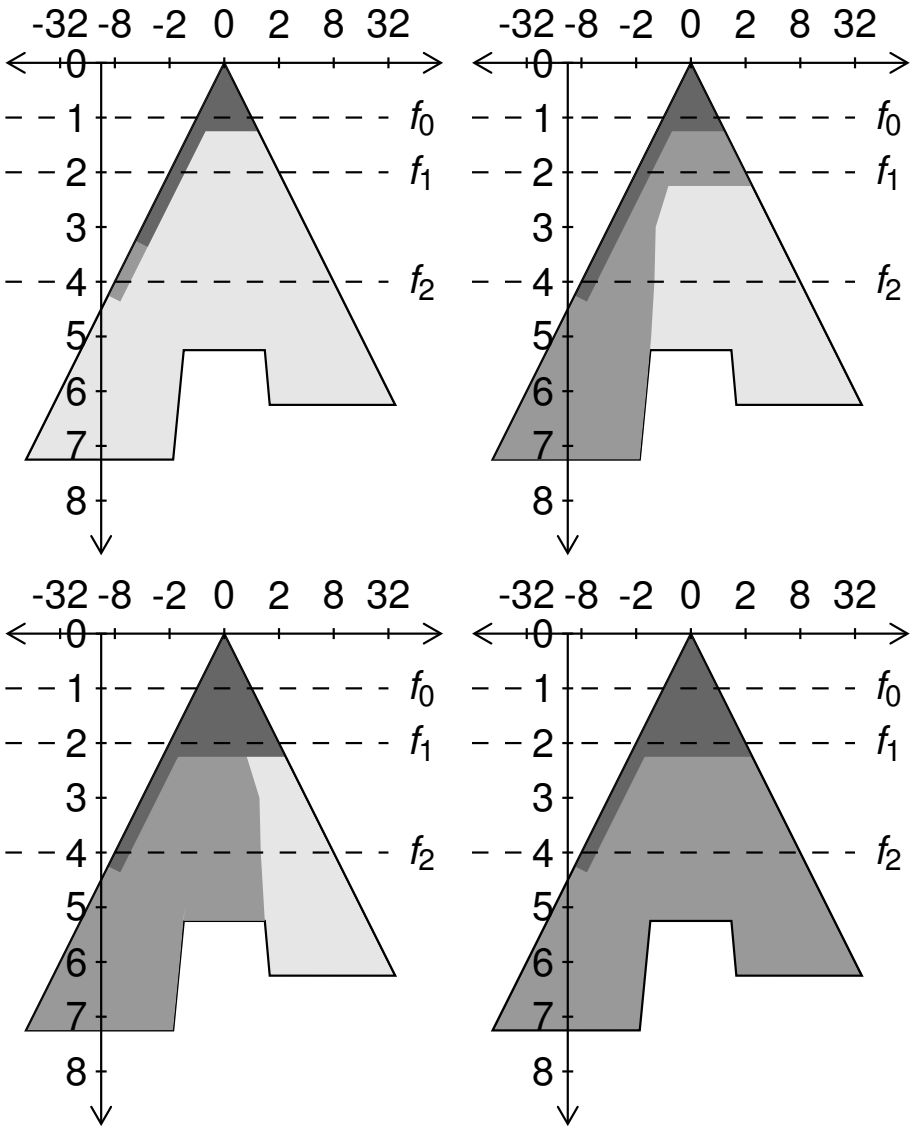
Konkret im Beispiel:

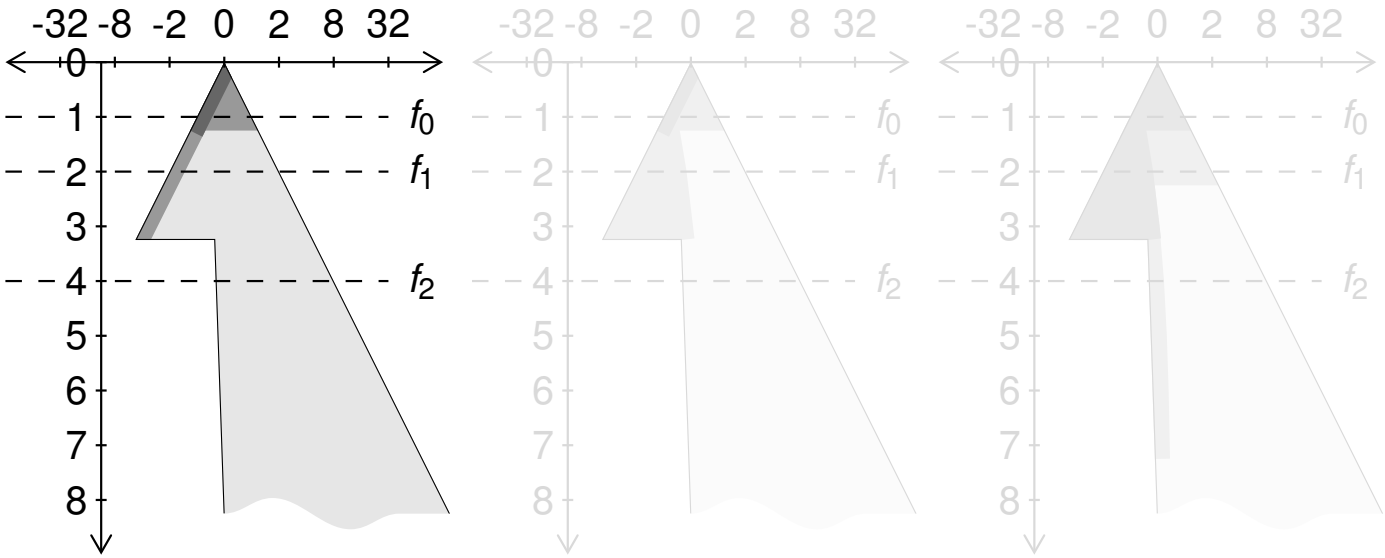
$$f_i = 2^i$$

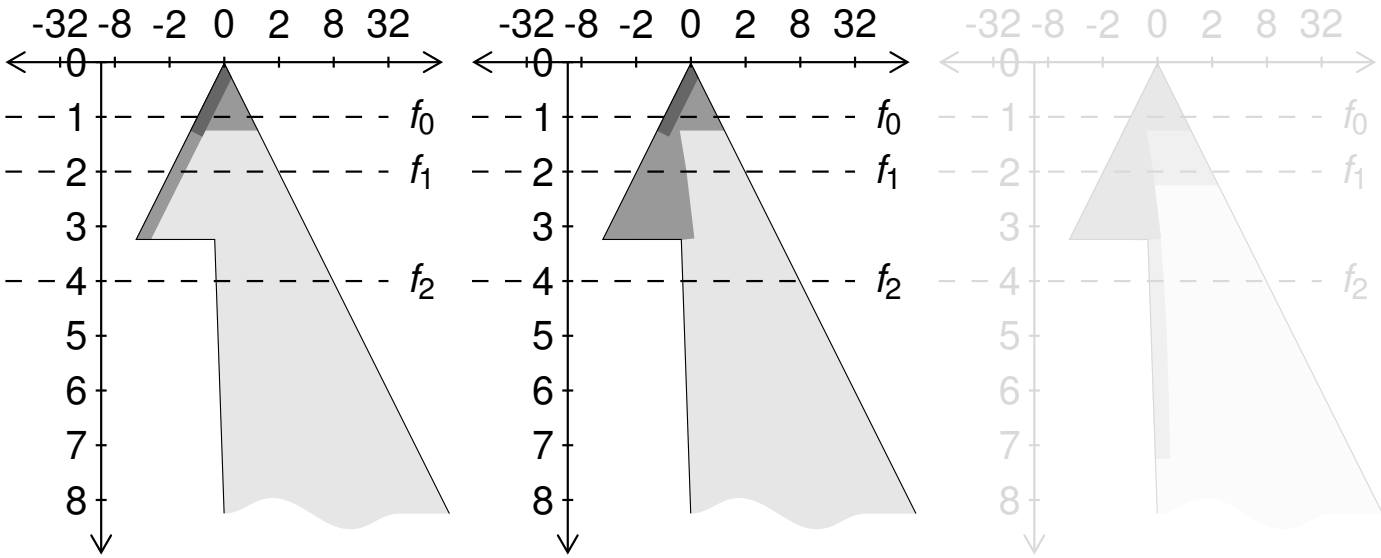

-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2

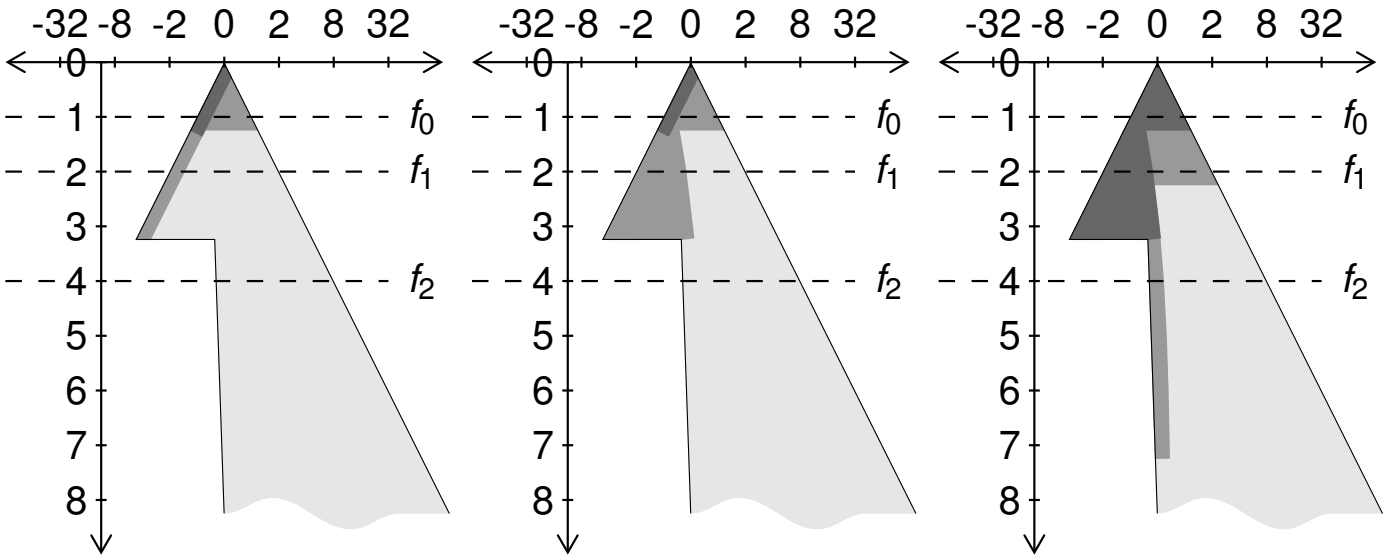

-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2

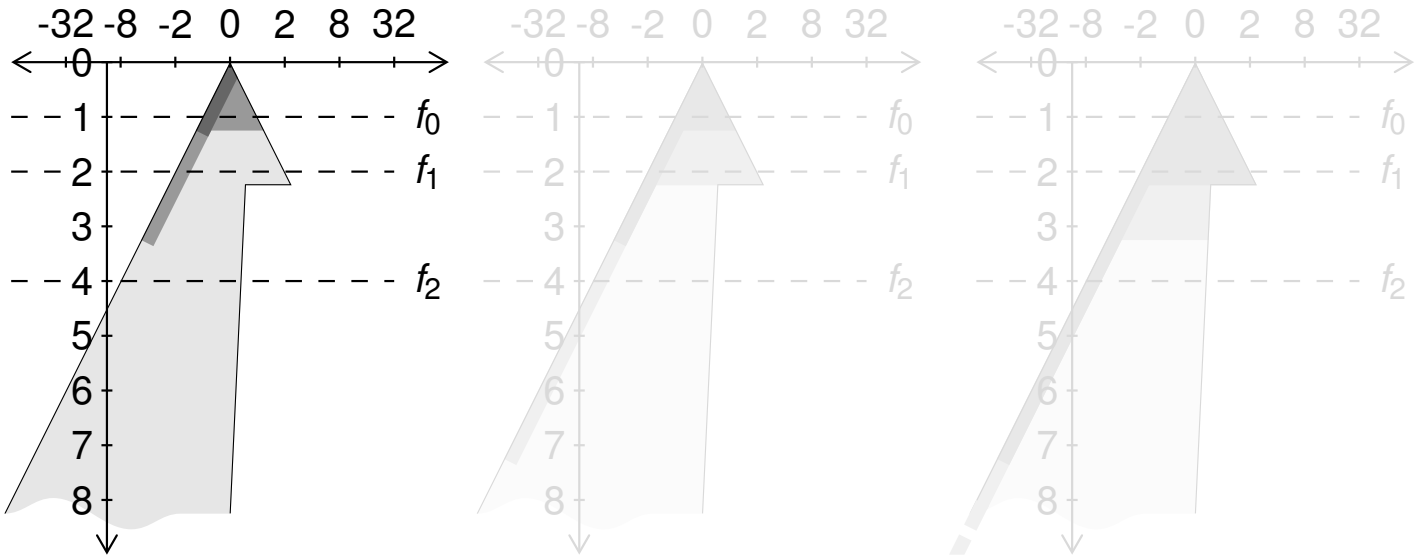

-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2

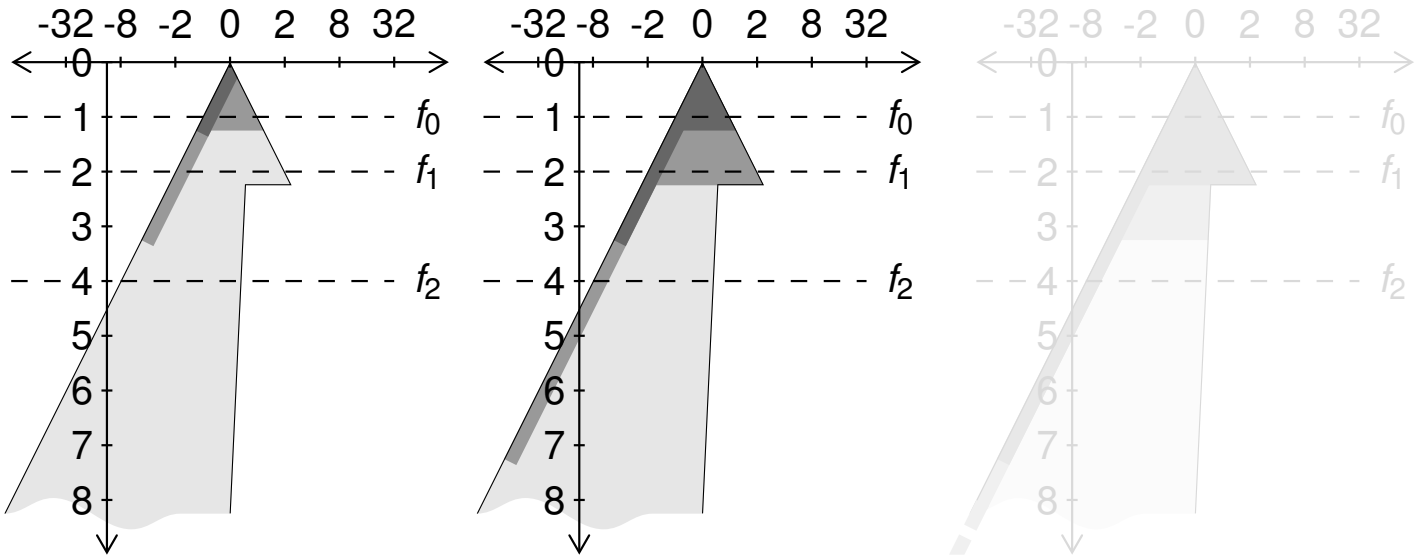

-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2

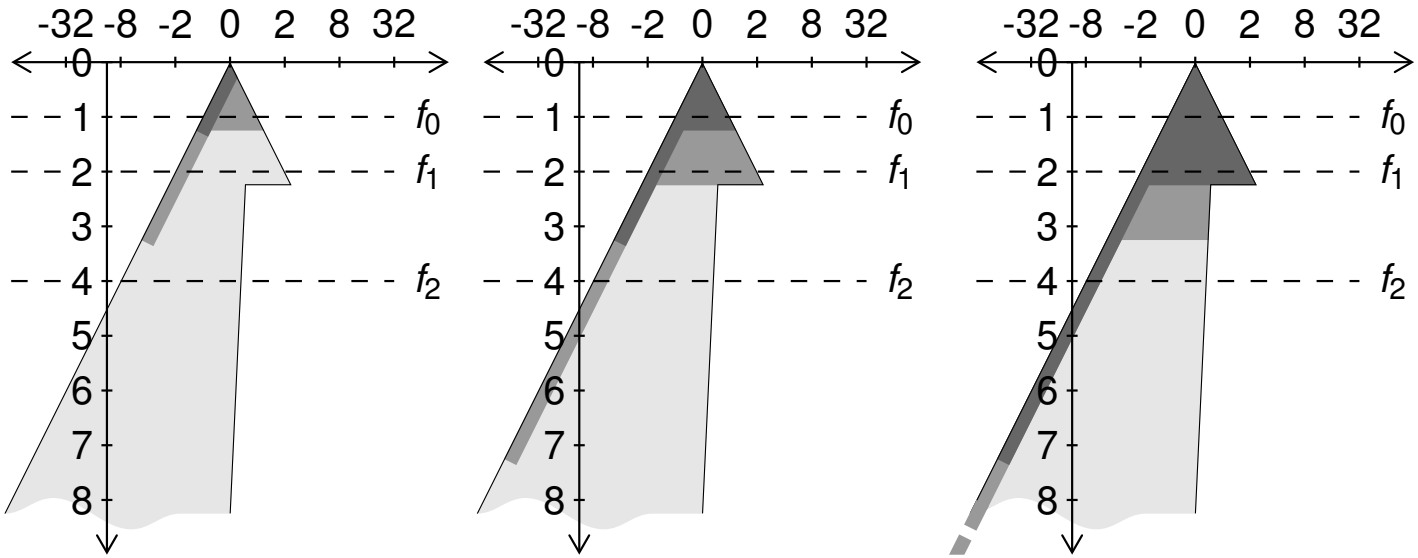

-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2


-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2


-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2


-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2


-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2


-32 -8 -2 0 2 8 32
0
1
2
3
4
5
6
7
8
f_0
f_1
f_2

Die Familie

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
   - D&B-Durchlauf
   - Die Familie
3. Vorteile und mögliche Anwendungen

Eigenschaften

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

Passende Funktion

$$f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$$

Eigenschaften

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

Passende Funktion

$f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$

Eigenschaften

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

Passende Funktion

$$f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$$

Eigenschaften

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

Passende Funktion

$$f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$$

Eigenschaften

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

Passende Funktion

$f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$

Eigenschaften

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

Passende Funktion

$f_{c,i} := \lfloor b^{\frac{i}{c}} \rfloor + i$

Nutzen

- Mit Hilfe des Parameters *c* lässt sich sozusagen ausdrücken, wieviel Speicher man in die Vollständigkeit des Algorithmus investieren möchte.
- Nahezu beliebige Abstufungen zwischen den beiden Extreme, Tiefen- und Breitendurchlauf, die entweder gar nichts oder alles in Vollständigkeit investieren.
- Für alle Algorithmen der Familie wird nur eine einzige Implementierung benötigt, da sich der Parameter *c* sich als Laufzeitparameter in die Implementierung integrieren lässt. Es ist sogar möglich, den Parameter dynamisch, also während der Traversierung anzupassen.

Nutzen

- Mit Hilfe des Parameters *c* lässt sich sozusagen ausdrücken, wieviel Speicher man in die Vollständigkeit des Algorithmus investieren möchte.
- Nahezu beliebige Abstufungen zwischen den beiden Extreme, Tiefen- und Breitendurchlauf, die entweder gar nichts oder alles in Vollständigkeit investieren.
- Für alle Algorithmen der Familie wird nur eine einzige Implementierung benötigt, da sich der Parameter *c* sich als Laufzeitparameter in die Implementierung integrieren lässt. Es ist sogar möglich, den Parameter dynamisch, also während der Traversierung anzupassen.

Nutzen

- Mit Hilfe des Parameters *c* lässt sich sozusagen ausdrücken, wieviel Speicher man in die Vollständigkeit des Algorithmus investieren möchte.
- Nahezu beliebige Abstufungen zwischen den beiden Extreme, Tiefen- und Breitendurchlauf, die entweder gar nichts oder alles in Vollständigkeit investieren.
- Für alle Algorithmen der Familie wird nur eine einzige Implementierung benötigt, da sich der Parameter *c* sich als Laufzeitparameter in die Implementierung integrieren lässt. Es ist sogar möglich, den Parameter dynamisch, also während der Traversierung anzupassen.

Vorteile und mögliche Anwendungen

1 Formalismus

2 Der D&B-Durchlauf und seine Familie

3 Vorteile und mögliche Anwendungen

Allgemein

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
3. Vorteile und mögliche Anwendungen
   - Allgemein
   - D&I-Durchlauf
   - Logikprogrammierung
     - Problem: Unvollständige Aufzählungen - Nie gefundene Lösungen
     - Problem: „UND“ bzw. „ODER“ sind nicht kommutativ
     - Problem: Effizienz

- Der D&B-Durchlauf verhält sich auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf.
- „Wählt“ sozusagen, dass jeweils „günstigere“ Verfahren.
- Die Menge des für die Vollständigkeit aufgewandten Speichers kann durch einen einfachen Parameter kontrollieren werden. (Familie)
- Tiefen- und der Breitendurchlauf die beiden Extreme der Familie.
- Mit vertretbarem Aufwand implemtierbar. Die benötigten Strukturen und Berechnungen sind weiterhin einfach.
- Erwartung: Der D&B-Durchlauf arbeitet auch was die tatsächliche Laufzeit betrifft ähnlich schnell wie Tiefen- oder Breitendurchlauf.

- Der D&B-Durchlauf verhält sich auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf.
- „Wählt“ sozusagen, dass jeweils „günstigere“ Verfahren.
- Die Menge des für die Vollständigkeit aufgewandten Speichers kann durch einen einfachen Parameter kontrollieren werden. (Familie)
- Tiefen- und der Breitendurchlauf die beiden Extreme der Familie.
- Mit vertretbarem Aufwand implemtierbar. Die benötigten Strukturen und Berechnungen sind weiterhin einfach.
- Erwartung: Der D&B-Durchlauf arbeitet auch was die tatsächliche Laufzeit betrifft ähnlich schnell wie Tiefen- oder Breitendurchlauf.

- Der D&B-Durchlauf verhält sich auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf.
- „Wählt“ sozusagen, dass jeweils „günstigere“ Verfahren.
- Die Menge des für die Vollständigkeit aufgewandten Speichers kann durch einen einfachen Parameter kontrollieren werden. (Familie)
- Tiefen- und der Breitendurchlauf die beiden Extreme der Familie.
- Mit vertretbarem Aufwand implemtierbar. Die benötigten Strukturen und Berechnungen sind weiterhin einfach.
- Erwartung: Der D&B-Durchlauf arbeitet auch was die tatsächliche Laufzeit betrifft ähnlich schnell wie Tiefen- oder Breitendurchlauf.

- Der D&B-Durchlauf verhält sich auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf.
- „Wählt“ sozusagen, dass jeweils „günstigere“ Verfahren.
- Die Menge des für die Vollständigkeit aufgewandten Speichers kann durch einen einfachen Parameter kontrollieren werden. (Familie)
- Tiefen- und der Breitendurchlauf die beiden Extreme der Familie.
- Mit vertretbarem Aufwand implemtierbar. Die benötigten Strukturen und Berechnungen sind weiterhin einfach.
- Erwartung: Der D&B-Durchlauf arbeitet auch was die tatsächliche Laufzeit betrifft ähnlich schnell wie Tiefen- oder Breitendurchlauf.

- Der D&B-Durchlauf verhält sich auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf.
- „Wählt“ sozusagen, dass jeweils „günstigere“ Verfahren.
- Die Menge des für die Vollständigkeit aufgewandten Speichers kann durch einen einfachen Parameter kontrollieren werden. (Familie)
- Tiefen- und der Breitendurchlauf die beiden Extreme der Familie.
- Mit vertretbarem Aufwand implemtierbar. Die benötigten Strukturen und Berechnungen sind weiterhin einfach.
- Erwartung: Der D&B-Durchlauf arbeitet auch was die tatsächliche Laufzeit betrifft ähnlich schnell wie Tiefen- oder Breitendurchlauf.

- Der D&B-Durchlauf verhält sich auf endlichen Bäumen nahezu wie ein Tiefendurchlauf, auf unendlichen jedoch fast wie ein Breitendurchlauf.
- „Wählt“ sozusagen, dass jeweils „günstigere“ Verfahren.
- Die Menge des für die Vollständigkeit aufgewandten Speichers kann durch einen einfachen Parameter kontrollieren werden. (Familie)
- Tiefen- und der Breitendurchlauf die beiden Extreme der Familie.
- Mit vertretbarem Aufwand implemtierbar. Die benötigten Strukturen und Berechnungen sind weiterhin einfach.
- Erwartung: Der D&B-Durchlauf arbeitet auch was die tatsächliche Laufzeit betrifft ähnlich schnell wie Tiefen- oder Breitendurchlauf.

D&I-Durchlauf

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
3. Vorteile und mögliche Anwendungen
   - Allgemein
   - D&I-Durchlauf
   - Logikprogrammierung
     - Problem: Unvollständige Aufzählungen - Nie gefundene Lösungen
     - Problem: „UND“ bzw. „ODER“ sind nicht kommutativ
     - Problem: Effizienz

Motivation

- Der D&B-Durchlauf die Eigenschaft in gewisser Weise zwischen Tiefen- und Breitendurchlauf zu „wählen“, abhängig davon ob der Baum endlich oder unendlich ist.
- Die Breitensuche ist wenig speichereffizient ist und kann deshalb auf nahezu vollständigen Bäumen praktisch keine großen Tiefen erreichen
- Wenn Vollständigkeit auf unendlichen Bäumen benötigt wird, die Iterative Tiefensuche eingesetzt.
- Iterative Tiefensuche führt auf endlichen Bäumen, insbesondere auf fast linearen Bäumen, zu deutlich höheren Laufzeiten für die gesamte Suche.

Motivation

- Der D&B-Durchlauf die Eigenschaft in gewisser Weise zwischen Tiefen- und Breitendurchlauf zu „wählen“, abhängig davon ob der Baum endlich oder unendlich ist.
- Die Breitensuche ist wenig speichereffizient ist und kann deshalb auf nahezu vollständigen Bäumen praktisch keine großen Tiefen erreichen
- Wenn Vollständigkeit auf unendlichen Bäumen benötigt wird, die Iterative Tiefensuche eingesetzt.
- Iterative Tiefensuche führt auf endlichen Bäumen, insbesondere auf fast linearen Bäumen, zu deutlich höheren Laufzeiten für die gesamte Suche.

Motivation

- Der D&B-Durchlauf die Eigenschaft in gewisser Weise zwischen Tiefen- und Breitendurchlauf zu „wählen“, abhängig davon ob der Baum endlich oder unendlich ist.
- Die Breitensuche ist wenig speichereffizient ist und kann deshalb auf nahezu vollständigen Bäumen praktisch keine großen Tiefen erreichen
- Wenn Vollständigkeit auf unendlichen Bäumen benötigt wird, die Iterative Tiefensuche eingesetzt.
- Iterative Tiefensuche führt auf endlichen Bäumen, insbesondere auf fast linearen Bäumen, zu deutlich höheren Laufzeiten für die gesamte Suche.

Motivation

- Der D&B-Durchlauf die Eigenschaft in gewisser Weise zwischen Tiefen- und Breitendurchlauf zu „wählen“, abhängig davon ob der Baum endlich oder unendlich ist.
- Die Breitensuche ist wenig speichereffizient ist und kann deshalb auf nahezu vollständigen Bäumen praktisch keine großen Tiefen erreichen
- Wenn Vollständigkeit auf unendlichen Bäumen benötigt wird, die Iterative Tiefensuche eingesetzt.
- Iterative Tiefensuche führt auf endlichen Bäumen, insbesondere auf fast linearen Bäumen, zu deutlich höheren Laufzeiten für die gesamte Suche.

Idee - D&I-Suche

- Iterative Tiefensuche nur dann einzusetzen, wenn sie auch tatsächlich benötigt wird, also nur auf unendlichen Bäumen.
- Kombination von Tiefensuche und Iterativer Tiefensuche nach dem gleichen Prinzip wie Tiefen- und Breitensuche in der D&B-Suche (bzw. D&B-Durchlauf)

Umsetzung

- Durch die Verwendung (der gleichen) wechselseitigen Tiefenschranken oder durch (um einen Faktor modifizierte) Credit-Funktionen erreichbar.
- Leichte Modifikation des Algorithmus der Iterativen Tiefensuche und eine höchstens geringfügig komplexere Speicherstruktur, damit die Iterative Tiefensuche keine Knoten expandiert, die bereits von der Tiefensuche expandiert wurden.

Idee - D&I-Suche

- Iterative Tiefensuche nur dann einzusetzen, wenn sie auch tatsächlich benötigt wird, also nur auf unendlichen Bäumen.
- Kombination von Tiefensuche und Iterativer Tiefensuche nach dem gleichen Prinzip wie Tiefen- und Breitensuche in der D&B-Suche (bzw. D&B-Durchlauf)

Umsetzung

- Durch die Verwendung (der gleichen) wechselseitigen Tiefenschranken oder durch (um einen Faktor modifizierte) Credit-Funktionen erreichbar.
- Leichte Modifikation des Algorithmus der Iterativen Tiefensuche und eine höchstens geringfügig komplexere Speicherstruktur, damit die Iterative Tiefensuche keine Knoten expandiert, die bereits von der Tiefensuche expandiert wurden.

Idee - D&I-Suche

- Iterative Tiefensuche nur dann einzusetzen, wenn sie auch tatsächlich benötigt wird, also nur auf unendlichen Bäumen.
- Kombination von Tiefensuche und Iterativer Tiefensuche nach dem gleichen Prinzip wie Tiefen- und Breitensuche in der D&B-Suche (bzw. D&B-Durchlauf)

Umsetzung

- Durch die Verwendung (der gleichen) wechselseitigen Tiefenschranken oder durch (um einen Faktor modifizierte) Credit-Funktionen erreichbar.
- Leichte Modifikation des Algorithmus der Iterativen Tiefensuche und eine höchstens geringfügig komplexere Speicherstruktur, damit die Iterative Tiefensuche keine Knoten expandiert, die bereits von der Tiefensuche expandiert wurden.

Idee - D&I-Suche

- Iterative Tiefensuche nur dann einzusetzen, wenn sie auch tatsächlich benötigt wird, also nur auf unendlichen Bäumen.
- Kombination von Tiefensuche und Iterativer Tiefensuche nach dem gleichen Prinzip wie Tiefen- und Breitensuche in der D&B-Suche (bzw. D&B-Durchlauf)

Umsetzung

- Durch die Verwendung (der gleichen) wechselseitigen Tiefenschranken oder durch (um einen Faktor modifizierte) Credit-Funktionen erreichbar.
- Leichte Modifikation des Algorithmus der Iterativen Tiefensuche und eine höchstens geringfügig komplexere Speicherstruktur, damit die Iterative Tiefensuche keine Knoten expandiert, die bereits von der Tiefensuche expandiert wurden.

Logikprogrammierung

1. Formalismus
2. Der D&B-Durchlauf und seine Familie
3. Vorteile und mögliche Anwendungen
   - Allgemein
   - D&I-Durchlauf
   - Logikprogrammierung
     - Problem: Unvollständige Aufzählungen - Nie gefundene Lösungen
     - Problem: „UND“ bzw. „ODER“ sind nicht kommutativ
     - Problem: Effizienz

Programm:

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

Anfrage:

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

- nat(X) liefert eine Aufzählung der natürlichen Zahlen $\mathbb{N}$
- Erwartung: nat2(X,Y) liefert eine Aufzählung von $\mathbb{N} \times \mathbb{N}$
- Tatsächlich: nat2(X,Y) liefert nur eine Aufzählung von $\{0\} \times \mathbb{N}$

Programm:

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

Anfrage:

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

- nat(X) liefert eine Aufzählung der natürlichen Zahlen $\mathbb{N}$
- Erwartung: nat2(X,Y) liefert eine Aufzählung von $\mathbb{N} \times \mathbb{N}$
- Tatsächlich: nat2(X,Y) liefert nur eine Aufzählung von $\{0\} \times \mathbb{N}$

Programm:

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

Anfrage:

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

- nat(X) liefert eine Aufzählung der natürlichen Zahlen $\mathbb{N}$
- Erwartung: nat2(X,Y) liefert eine Aufzählung von $\mathbb{N} \times \mathbb{N}$
- Tatsächlich: nat2(X,Y) liefert nur eine Aufzählung von $\{0\} \times \mathbb{N}$

Programm:

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

Anfrage:

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

- nat(X) liefert eine Aufzählung der natürlichen Zahlen $\mathbb{N}$
- Erwartung: nat2(X,Y) liefert eine Aufzählung von $\mathbb{N} \times \mathbb{N}$
- Tatsächlich: nat2(X,Y) liefert nur eine Aufzählung von $\{0\} \times \mathbb{N}$

Anfrage:

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

Anfrage:

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

- Ziel: Deklarative Semantik. Der Programmierer soll sich beim Schreiben eines Programms (zumindest zunächst) keine Gedanken über den zur Auswertung benutzten Algorithmus machen müssen.
- Erwartung: Beide Programme liefern das gleiche Ergebniss
- Tatsächlich: Die erste Anfrage liefert weder eine Lösung, noch terminiert sie. Die zweite Anfrage (eine vernünftige Implementierung von greater vorausgesetzt) terminiert mit der Antwort ja.

Anfrage:

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

Anfrage:

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

- Ziel: Deklarative Semantik. Der Programmierer soll sich beim Schreiben eines Programms (zumindest zunächst) keine Gedanken über den zur Auswertung benutzten Algorithmus machen müssen.
- Erwartung: Beide Programme liefern das gleiche Ergebniss
- Tatsächlich: Die erste Anfrage liefert weder eine Lösung, noch terminiert sie. Die zweite Anfrage (eine vernünftige Implementierung von greater vorausgesetzt) terminiert mit der Antwort ja.

Anfrage:

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

Anfrage:

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

- Ziel: Deklarative Semantik. Der Programmierer soll sich beim Schreiben eines Programms (zumindest zunächst) keine Gedanken über den zur Auswertung benutzten Algorithmus machen müssen.
- Erwartung: Beide Programme liefern das gleiche Ergebniss
- Tatsächlich: Die erste Anfrage liefert weder eine Lösung, noch terminiert sie. Die zweite Anfrage (eine vernünftige Implementierung von greater vorausgesetzt) terminiert mit der Antwort ja.

Anfrage:

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

Anfrage:

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

- Ziel: Deklarative Semantik. Der Programmierer soll sich beim Schreiben eines Programms (zumindest zunächst) keine Gedanken über den zur Auswertung benutzten Algorithmus machen müssen.
- Erwartung: Beide Programme liefern das gleiche Ergebniss
- Tatsächlich: Die erste Anfrage liefert weder eine Lösung, noch terminiert sie. Die zweite Anfrage (eine vernünftige Implementierung von greater vorausgesetzt) terminiert mit der Antwort ja.

Programm:

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

Anfrage:

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

Programm:

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

Anfrage:

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

- Das obige Programm ist „nahezu“ endrekursiv in dem Sinn, als dass die Anfrage einen ähnlich strukturierten, nämlich fast linearen, Suchbaum erzeugt.

Programm:

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

Anfrage:

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

- Das obige Programm ist „nahezu“ endrekursiv in dem Sinn, als dass die Anfrage 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.

Programm:

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

Anfrage:

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

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

Programm:

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

Anfrage:

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

- 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 quadratische Zeit benötigen.
  Das ist für die Konkurrenzfähigkeit der Programmiersprache natürlich ungünstig.

Programm:

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

Anfrage:

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

- Die Alternative ist die Verwendung der D&B-Suche bzw. der D&I-Suche

Programm:

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

Anfrage:

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

- Die D&B-Suche verzichtet komplett auf Mehrfachauswertungen und kann das Programm dadurch genauso schnell wie die Tiefensuche ausführen.

Programm:

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

Anfrage:

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

- Die D&B-Suche verzichtet komplett auf Mehrfachauswertungen und kann das Programm dadurch genauso schnell wie die Tiefensuche ausführen.
- Die D&I-Suche schließt Mehrfachauswertungen nicht aus, garantiert aber, dass deren Zahl so gering ist, dass die Auswertung nur wenig verlangsamt wird und insbesondere linear bleibt.

Die D&B-Suche bzw. die D&I-Suche ermöglichen es voraussichtlich, logische Programmiersprachen zu implementieren, die

- Ein vollständiges Suchverfahren für die Auswertung benutzen, mit den daraus resultierenden Vorteilen.
- Konkurrenzfähige Laufzeiten bieten.

Die D&B-Suche bzw. die D&I-Suche ermöglichen es voraussichtlich, logische Programmiersprachen zu implementieren, die

- Ein vollständiges Suchverfahren für die Auswertung benutzen, mit den daraus resultierenden Vorteilen.
- Konkurrenzfähige Laufzeiten bieten.