Großer Beleg

Calculation of Inside and Outside Weights of Weighted Hypergraphs by Newton's Method

(revised version)

Tobias Denkinger
tobias.denkinger@mailbox.tu-dresden.de

5. Dezember 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl für Grundlagen der Programmierung

Betreuer: Dipl.-Inf. Toni Dietze
Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Heiko Vogler

Aufgabenstellung für den Großen Beleg

„Newton-Verfahren zur Berechnung von Inside- und Outside-Gewichten in gewichteten Hypergraphen“

Technische Universität Dresden
Fakultät Informatik

Student: Tobias Denkinger
Studiengang: Diplom Informatik (Prüfungsordnung 2004)
Matrikelnummer: 3478448

Inside- und Outside-Gewichte spielen beispielsweise beim unüberwachten Training probabilistischer kontextfreier Grammatiken mittels des Expectation-Maximization-Algorithmus eine wichtige Rolle. Sie können leicht mit Hilfe einer Fixpunktiteration angenähert werden, jedoch konvergiert dieses Verfahren im Allgemeinen relativ langsam. Eine andere Möglichkeit zur Bestimmung der Gewichte ist das Newton-Verfahren, welches eine bessere Näherung in weniger Iterationsschritten verspricht.

Im Rahmen der Belegarbeit soll die Berechnung von Inside- und Outside-Gewichten für gewichtete Hypergraphen mittels Newton-Verfahren implementiert werden. Die Implementierung soll möglichst zeit- und speicherplatzeffizient in Haskell umgesetzt werden und in das System *Vanda* einbettbar sein. Die formalen Grundlagen sollen im schriftlichen Teil der Arbeit dokumentiert werden. Dazu zählen

- die Definition der Inside- und Outside-Gewichte [NS08, MS99, Abschnitt 11.3] für gewichtete Hypergraphen,
- die Herleitung der Fixpunktiteration und die Instanziierung des Newton-Verfahrens und
- das Zeigen der Korrektheit der Verfahren [EKL08].

Anhand geeigneter Beispiele soll die Berechnungszeit und die Geschwindigkeit der Konvergenz des implementierten Newton-Verfahrens untersucht und dokumentiert werden. Die Ergebnisse sollen mit einer beim Lehrstuhl vorhandenen Implementierung der Fixpunktiteration verglichen werden.

Optional kann untersucht werden, ob sich eine Zerlegung des Hypergraphen in *strongly connected components* und eine anschließende komponentenweise Berechnung der Gewichte praktisch lohnt. Außerdem können weitere Verfahren zur Bestimmung der Gewichte untersucht und implementiert werden.

Die Arbeit muss den üblichen Standards wie folgt genügen. Die Arbeit muss in sich abgeschlossen sein und alle nötigen Definitionen und Referenzen enthalten. Die Struktur der Arbeit muss klar erkenntlich sein, und der Leser soll gut durch die Arbeit geführt werden. Die Darstellung aller Begriffe und Verfahren soll mathematisch formal fundiert sein. Für jeden wichtigen Begriff sollen Beispiele angegeben werden, ebenso für die Abläufe der beschriebenen Verfahren. Wo es angemessen ist, sollten Illustrationen die Darstellung vervollständigen. Für die Implementierung soll eine ausführliche Dokumentation erfolgen, die sich angemessen auf den Quelltext und die schriftliche Ausarbeitung verteilt. Dabei muss die Funktionsfähigkeit des Programms glaubhaft gemacht werden.

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer: Toni Dietze
Beginn am: 10. April 2012
Einzureichen 10. Oktober 2012

Dresden, 04. April 2012

Unterschrift von Heiko Vogler Unterschrift von Tobias Denkinger

Literatur

[EKL08] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Convergence thresholds of Newton's method for monotone polynomial equations. In Pascal Weil and Susanne Albers, editors, *Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 289–300, Bordeaux, France, 2008. `http://arxiv.org/abs/0802.2856`.

[MS99] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, Cambridge, MA, USA, 1999. `http://dl.acm.org/citation.cfm?id=311445`.

[NS08] Mark-Jan Nederhof and Giorgio Satta. Computing partition functions of pcfgs. *Research on Language and Computation*, 6:139–162, 2008. 10.1007/s11168-008-9052-8, `http://dx.doi.org/10.1007/s11168-008-9052-8`.

Contents

1 Introduction

Inside and outside weights are the basis for many algorithms in *statistical machine translation* (SMT). *Vanda*, developed by the Chair of Foundations of Programming at TU Dresden, is an SMT toolkit. It utilizes the *expectation maximization algorithm* in order to train weights for language and translation models. This algorithm requires the computation of inside and outside weights. But often, these weights can not be computed analytically, therefore we need to develop efficient algorithms to approximate them.

Vanda already implements the *fixed-point method* for that purpose. This work will explore two additional methods, Newton's method and decomposed Newton's method, which are known to have a faster convergence.

2 Preliminaries

In this section, we introduce basic notations and formalisms used in this work. The *natural numbers* are elements of the set $\mathbb{N} = \{0, 1, 2, 3, ...\}$. Let $a \in \mathbb{N}$, then $[a]$ is the set $[a] = \{n \in \mathbb{N} \mid n \leq a\}$.

For any set A, we define the set of all *finite sequences* of A, denoted by A^*. Let $\underline{a} = \langle a_0, a_1, ..., a_n \rangle \in A^*$ be a finite sequence, we define the *length of* $\underline{a}$ as $|\underline{a}| = n + 1$, the *contains relation* $\in$ by $a \in \underline{a} \iff \exists i \in [n] \colon a_i = a$ and the *element at index* i *in* $\underline{a}$ by $\underline{a}.i = a_i$. Let $B \subseteq A$, we define the *removal of all elements of* B *from* $\underline{a}$ as $\underline{a} \setminus B = \langle a_i \in \underline{a} \mid a_i \notin B \rangle$.

Let A, B be arbitrary sets, we denote $\{m \mid m \colon A \to B\}$ by B^A. If A is totally ordered and finite, we also call such a mapping $m \in B^A$ a B*-vector indexed by* A and write it down as a simple line vector over B.

Let Σ be a *ranked alphabet*, i.e. there is a mapping $\mathrm{rk} \colon \Sigma \to \mathbb{N}$ that assigns a *rank* to every symbol. The *set of trees over* Σ, denoted by T_Σ, is the smallest set T such that $\{\sigma \in \Sigma \mid \mathrm{rk}(\sigma) = 0\} \subseteq T$ and $\{\sigma(t_1, ..., t_n) \mid \sigma \in \mathrm{rk}^{-1}(n), t_1, ..., t_n \in T\} \subseteq T$. For every tree $t = \sigma(t_1, ..., t_n) \in T_\Sigma$ we define the mappings $\mathrm{root}(t) = \sigma$ and $\mathrm{children}(t) = \langle t_1, ..., t_n \rangle$. The mapping $\mathrm{yield}(t)$ is the sequence of leaves of t read from left to right. The set of all *subtrees of* t is the set $\mathrm{Sub}(\sigma(t_1, ...t_n)) = \{\sigma(t_1, ..., t_n)\} \cup \bigcup_{i \in [n]} \mathrm{Sub}(t_i)$.

Let Σ and V be a ranked alphabets with $\mathrm{rk} \colon \Sigma \to \mathbb{N}$ and $\mathrm{rk}_V \colon V \to \{0\}$. We define the V*-contexts over* Σ as the set $C_{\Sigma,V} \subseteq T_{\Sigma \cup V}$ such that for every $c \in C_{\Sigma,V} \colon |\mathrm{yield}(c) \setminus \Sigma| = 1$.

2.1 Directed graphs

A *directed graph* $G = (V, E)$ consists of a set V of *vertices* and a set $E \subseteq V \times V$ of *directed edges*. We call an edge $e = (v, v') \in E$ the *edge from* v *to* v' *in* G. We define the projections $\mathrm{head}(e) = v'$ and $\mathrm{tail}(e) = v$.

Let $v, v' \in V$. For some $n \in \mathbb{N}$, a *path* from v to v' with length n is the sequence of edges $\langle e_0, ..., e_n \rangle$ such that $\mathrm{tail}(e_0) = v, \mathrm{head}(e_n) = v'$ and for every $i \in [n-1] \colon \mathrm{head}(e_i) = \mathrm{tail}(e_{i+1})$. If there exists a path from v to v' and one from v' to v, we call v and v' *strongly connected*. A *strongly connected component* is a maximal subset of V such that every pair of vertices in this subset is strongly connected. All strongly connected components of G can be calculated by Tarjan's algorithm [Tar72].

2.2 Hypergraphs

Hypergraphs are a generalization of *directed graphs*. In hypergraphs, edges contain a sequence of tail vertices (which can be empty) and a label.

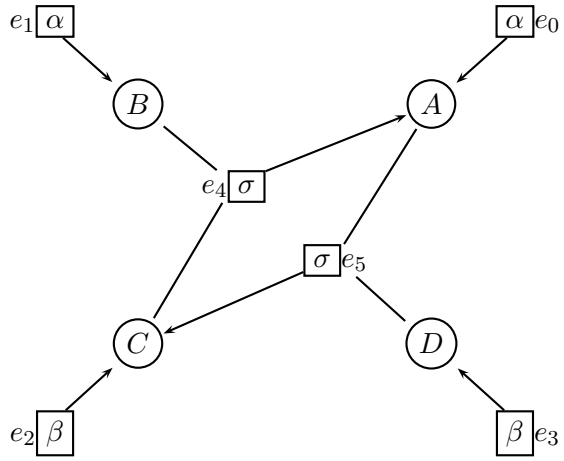

Figure 1: example hypergraph

Definition 1 (weighted hypergraph) Let Σ be an alphabet. A *weighted Σ-hypergraph* is a tuple $H = (V, E, \mu)$ where

- V is a finite set (of vertices),
- $E \subseteq V^* \times \Sigma \times V$ is a finite set (of edges) and
- $\mu \colon E \to \mathbb{R}_{\geq 0}$ assigns a weight to every edge. □

Let $e = (t, \sigma, h) \in E$, we define the projections $\text{tail}(e) = t, \text{label}(e) = \sigma$ and $\text{head}(e) = h$.

Example 1 (hypergraph) Let $\Sigma = \{\alpha, \beta, \sigma\}$ be an alphabet and $H = (V, E, \mu)$ be a weighted Σ-hypergraph with

$$
\begin{aligned}
V &= \{A, B, C, D\}, & E &= \{e_0, ..., e_5\},\\
\mu &\colon E \to \mathbb{R}_{\geq 0},\\
e_0 &= (\langle\rangle, \alpha, A), & \mu(e_0) &= 0.25,\\
e_1 &= (\langle\rangle, \alpha, B), & \mu(e_1) &= 1.0,\\
e_2 &= (\langle\rangle, \beta, C), & \mu(e_2) &= 0.25,\\
e_3 &= (\langle\rangle, \beta, D), & \mu(e_3) &= 1.0,\\
e_4 &= (\langle B, C\rangle, \sigma, A), & \mu(e_4) &= 0.25,\\
e_5 &= (\langle D, A\rangle, \sigma, C), & \mu(e_5) &= 0.25
\end{aligned}
$$

We will use the weighted hypergraph H as a running example. A graphical representation of H is shown in Figure 1. Every vertex is represented by a circle and hyperedges are represented by boxes. Connections from vertices to edges are drawn as line and connections from edges to vertices are drawn as arrow. □

Definition 2 (path of a weighted hypergraph) Let $H = (V, E, \mu)$ be a weighted hypergraph. The *direct path relation* is the relation $\hookrightarrow_H \subseteq V \times V, \hookrightarrow_H = \{(v, \mathit{head}(e)) \mid e \in E, v \in \text{tail}(e)\}$. The *path relation* $\hookrightarrow_H^* \subseteq V \times V$ is the reflexive and transitive closure on $\hookrightarrow_H$. We call v' *reachable from* v if $v \hookrightarrow_H^* v'$. □

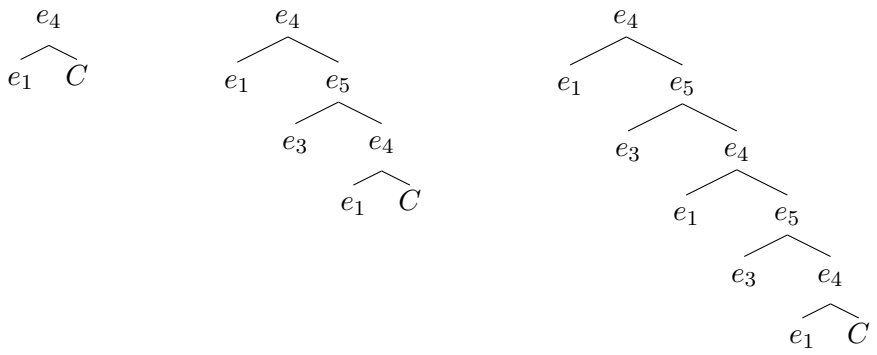

Figure 2: some A-rooted C-contexts of H

Definition 3 (hyperpaths) Let $H = (V, E, \mu)$ be a weighted hypergraph. Then E is a ranked alphabet where for every $e \in E$: $\mathrm{rk}(e) = |\mathrm{tail}(e)|$. The *hyperpaths of* H are elements of the set $\mathrm{HP}(H) = \{t \in T_E \mid \forall e(t_1, ..., t_n) \in \mathrm{Sub}(t), \forall i \in [n]\colon \mathrm{tail}(e).i = \mathrm{head}(\mathrm{root}(t_i))\}$ □

Definition 4 (fragment of hyperpaths) Let $H = (V, E, \mu)$ be a weighted hypergraph and $v \in V$. The *v-fragment of* $\mathrm{HP}(H)$ is the set $\mathrm{HP}(H, v) = \{t \in \mathrm{HP}(H) \mid \mathrm{head}(\mathrm{root}(t)) = v\}$. □

Definition 5 (weight of a hyperpath) Let $H = (V, E, \mu)$ be a weighted hypergraph and $t \in \mathrm{HP}(H)$ a hyperpath in H. The *weight of t in H* is defined by

$$\mathrm{wt}(t) = \mu(\mathrm{root}(t)) \cdot \prod_{t' \in \mathrm{children}(t)} \mathrm{wt}(t').$$

□

Definition 6 (v-rooted v'-context of a hypergraph) Let $H = (V, E, \mu)$ be a weighted hypergraph and $v, v' \in V$. The set E is a ranked alphabet where for every $e \in E$: $\mathrm{rk}(e) = |\mathrm{tail}(e)|$. The set of *$v$-rooted v'-contexts in H* is the set

$$C_H^v(v') = \Bigg\{ c \in C_{E,\{v'\}} \mid \mathrm{head}(\mathrm{root}(c)) = v, \forall e(t_1, ..., t_n) \in \mathrm{Sub}(c) \\ , i \in [n]\colon \mathrm{tail}(e).i = \begin{cases} t_i & \text{if } t_i \in V \\ \mathrm{head}(\mathrm{root}(t_i)) & \text{otherwise} \end{cases} \Bigg\}.$$

Example 2 (context) Let H be the hypergraph from Example 1. Some A-rooted C-contexts in H are shown in Figure 2. □

Definition 7 (weight of a context) Let $H = (V, E, \mu)$ be a weighted hypergraph and $c \in C_{E,V}$ a context in H. The *weight of c in H* is defined by

$$\mathrm{wt}(c) = \begin{cases} 1 & \text{if } c \in V \\ \mu(\mathrm{root}(c)) \cdot \prod_{c' \in \mathrm{children}(c)} \mathrm{wt}(c') & \text{otherwise.} \end{cases}$$

□

2.3 Inside and outside weights

Inside and outside weights are properties of a vertex v in a weighted hypergraph H. The inside weight is the sum of the weights of all hyperpaths in H leading to v and the outside weight is the sum of the weights of all v-contexts in H that have the starting symbol as head of their root.

Definition 8 (inside weights of a hypergraph) Let $H = (V, E, \mu)$ be a weighted hypergraph. Then the *inside weights of* H are defined by

$$\text{inside}(A) = \sum_{t \in \text{HP}(H,A)} \text{wt}(t).$$

□

Since the set $\text{HP}(H, A)$ is potentially not finite, we may not be able to compute the inside weights directly. Therefore we define a function *inner* that calculates the value of inside recursively.

Definition 9 (recursive system for inside weights) Let $H = (V, E, \mu)$ be a weighted hypergraph. The *recursive system for the inside weights of* H is

$$\forall v \in V\colon \text{inner}(v) = \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \prod_{i \in [|w|-1]} \text{inner}(w_i).$$

□

Lemma 1 (inside = inner) *Let $H = (V, E, \mu)$ be a weighted hypergraph, $v, v' \in V$ and*

$$s\colon V \to \{0, 1\}, s(v') = \begin{cases} 1 & \text{if } A = v \\ 0 & \text{otherwise,} \end{cases}$$

then $\text{inside}(v) = \text{inner}(v)$.

Proof

$$\begin{aligned}
&\text{inside}(v) \\
&= \sum_{t \in \text{HP}(v)} \text{wt}(t) \\
&= \sum_{\substack{t \in \text{HP}(v) \\ t=e(t_0,\ldots,t_n)}} \mu(e) \cdot \prod_{i \in [n]} \text{wt}(t_i) \\
&= \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \sum_{t_0 \in \text{HP}(w_0)} \ldots \sum_{t_n \in \text{HP}(w_n)} \prod_{i \in [n]} \text{wt}(t_i) \\
&= \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \sum_{t_0 \in \text{HP}(w_0)} \ldots \sum_{t_n \in \text{HP}(w_n)} \left(\prod_{i \in [n-1]} \text{wt}(t_i) \right) \cdot \text{wt}(t_n) \\
&= \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \left(\sum_{t_0 \in \text{HP}(w_0)} \ldots \sum_{t_{n-1} \in \text{HP}(w_{n-1})} \prod_{i \in [n-1]} \text{wt}(t_i) \right) \cdot \sum_{t_n \in \text{HP}(w_n)} \text{wt}(t_n) \\
&= \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \left(\sum_{t_0 \in \text{HP}(w_0)} \text{wt}(t_0) \right) \cdot \ldots \cdot \left(\sum_{t_n \in \text{HP}(w_n)} \text{wt}(t_n) \right)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \prod_{i \in [|w|]} \sum_{t' \in \mathrm{HP}(w_i)} \mathrm{wt}(t') \\
&= \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \prod_{i \in [|w|]} \mathrm{inside}(w_i) \\
&= \mathrm{inner}(v)
\end{aligned}
$$

■

Definition 10 (outside weights of a hypergraph) Let $H = (V, E, \mu)$ be a weighted hypergraph and $v \in V$. Then the *v-rooted outside weights of H* are defined by

$$\forall v' \in V \colon \mathrm{outside}^v(v') = \sum_{c \in C_H^v(v')} \mathrm{wt}(c).$$

□

We can define a function *outer* that calculates the value of outside recursively.

Definition 11 (recursive system for the outside weights) Let $H = (V, E, \mu)$ be a weighted hypergraph, $v \in V$ and

$$s \colon V \to \{0, 1\}, s(v') = \begin{cases} 1 & \text{if } v' = v \\ 0 & \text{otherwise,} \end{cases}$$

then the *recursive system for v-rooted outside weights of H* is defined by

$$\mathrm{outer}^v(v') = s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,\hat{v}) \\ \exists i \colon w_i = A}} \mathrm{outer}^v(\hat{v}) \cdot \mu(e) \cdot \prod_{\substack{j \in [|w|] \\ j \neq i}} \mathrm{inner}(w_j)$$

□

Lemma 2 (outside = outer) *Let $H = (V, E, \mu)$ be a weighted hypergraph, $v, v' \in V$ and*

$$s \colon V \to \{0, 1\}, s(v') = \begin{cases} 1 & \textit{if } v' = v \\ 0 & \textit{otherwise,} \end{cases}$$

then $\mathrm{outside}^v(v') = \mathrm{outer}^v(v')$.

Proof

$$
\begin{aligned}
&\mathrm{outside}^v(v') \\
=& \sum_{c \in C_H^v(v')} \mathrm{wt}(c) \\
=& \mathrm{wt}(v) \cdot s(v') + \sum_{\substack{c \in C_H^v(v') \\ e \in E \\ c=(e(t_0,\ldots,t_n))}} \mathrm{wt}(c) \\
=& s(v') + \sum_{\substack{c \in C_H^v(v') \\ e \in E \\ c=(e(t_0,\ldots,t_n))}} \mathrm{wt}(c) \\
=& s(v') + \sum_{\substack{c \in C_H^v(v') \\ e \in E \\ c=(e(t_0,\ldots,t_n))}} \mu(e) \cdot \prod_{j \in [n]} \mathrm{wt}(t_j)
\end{aligned}
$$

$$
\begin{aligned}
=& s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \sum_{t_0 \in \mathrm{HP}(w_0)} \dots \sum_{t_{i-1} \in \mathrm{HP}(w_{i-1})} \\
& \sum_{t_i \in C_H^v(v')} \sum_{t_{i+1} \in \mathrm{HP}(w_{i+1})} \dots \sum_{t_n \in \mathrm{HP}(w_n)} \prod_{j \in [n]} \mathrm{wt}(t_j) \\
=& s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \sum_{t_i \in C_H^{w_i}(v')} \sum_{t_0 \in \mathrm{HP}(w_0)} \dots \sum_{t_{i-1} \in \mathrm{HP}(w_{i-1})} \\
& \sum_{t_{i+1} \in \mathrm{HP}(w_{i+1})} \dots \sum_{t_n \in \mathrm{HP}(w_n)} \prod_{j \in [n]} \mathrm{wt}(t_j) \\
=& s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \left(\sum_{t_i \in C_H^{w_i}(v')} \mathrm{wt}(t_i) \right) \left(\sum_{t_0 \in \mathrm{HP}(A_0)} \mathrm{wt}(t_1) \right) \cdot \dots \\
& \cdot \left(\sum_{t_{i-1} \in \mathrm{HP}(w_{i-1})} \mathrm{wt}(t_{i-1}) \right) \cdot \left(\sum_{t_{i+1} \in \mathrm{HP}(w_{i+1})} \mathrm{wt}(t_{i+1}) \right) \dots \left(\sum_{t_n \in \mathrm{HP}(w_n)} \mathrm{wt}(t_n) \right) \\
=& s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \left(\sum_{t_i \in C_H^{w_i}(v')} \mathrm{wt}(t_i) \right) \prod_{\substack{j \in [n] \\ i \neq j}} \sum_{t \in \mathrm{HP}(w_j)} \mathrm{wt}(t) \\
=& s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,v)}} \mu(e) \cdot \left(\sum_{t_i \in C_H^{w_i}(v')} \mathrm{wt}(t_i) \right) \prod_{\substack{j \in [n] \\ i \neq j}} \mathrm{inner}(w_j) \\
=& s(v') + \sum_{\substack{e \in E \\ e=(w,\sigma,v) \\ w_i = A}} \mu(e) \cdot \mathrm{outside}^{w_i}(v') \cdot \prod_{\substack{j \in [n] \\ i \neq j}} \mathrm{inner}(w_j) \\
=& \, \mathrm{outer}^v(v')
\end{aligned}
$$

■

3 Calculating Inside and Outside Weights

In the following, we introduce several methods to obtain the inside and outside weights of a weighted hypergraph. Due to the fact that the inside weights are hard to obtain analytically, we will use iterative algorithms to approximate them.

3.1 Fixed-point method

Since inside and outside weights can be defined recursively, it is natural to consider fixed-point method for their calculation.

3.1.1 Calculating the fixed point

Because of the facts, that we start with the zero vector and that iterations of the recursive functions inner and outer are monotonically increasing, we will compute the least fixed point of these functions [EKL08].

Definition 12 (least fixed point) Let $f\colon A \to A$ be a function and $\leq \subseteq A \times A$ a total order on A. A value $a \in A$ is a *fixed point* of f, if $f(a) = a$ holds. The value $a \in A$ is the *least fixed point of f with respect to* $\leq$ if there is *no* $a' \in A$ such that a' is a fixed point of f and $a' < a$. □

Definition 13 (fixed-point method) Let $f\colon A \to A$ be a function with (at least) one fixed point, $x_0 \in A$ (be a *starting value*) and $c\colon A \times A \to \mathbb{B}$ be a predicate (*breaking condition*). We define the *Kleene sequence* $\langle x_0, x_1, ...\rangle$ where $\forall i \geq 1\colon x_i = f(x_{i-1})$. The *fixed-point method* determines the smallest i such that $c(x_i, x_{i+1})$ holds and returns x_i as *result*. □

3.1.2 Inside weights as fixed-point problem

Given Definition 2.3, we can easily define a function $in\colon \mathbb{R}^V \to \mathbb{R}^V$ such that the least fixed-point of in is the vector of inside weights. Let $V = \{A_0, ..., A_m\}$ and i be the vector of inside weights. Since i is a fixed point of in we have that $in(i) = i$. By Definition 2.3 then we have the fixed point function for inside weights.

Definition 14 (fixed-point function of the inside weights) Let $H = (V, E, \mu)$ be a weighted hypergraph with $V = \{A_0, ..., A_m\}$. Then the *fixed-point function of the inside weights* is

$$in\colon \mathbb{R}^V \to \mathbb{R}^V$$

$$in(i) = \begin{pmatrix} \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_0)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \\ \vdots \\ \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_m)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \end{pmatrix}.$$

We start with the zero vector. Since in only has coefficients greater 0 and we start with the zero vector, every value in the vector grows with every application of in. Therefore we calculate in fact the least fixed point.

Example 3 (fixed-point function of the inside weights) Let H be the hypergraph of Example 1. Then the fixed-point function of its inside weights is

$$in(i) = \begin{pmatrix} \mu(e_0) + \mu(e_4) \cdot i_C \cdot i_B \\ \mu(e_1) \\ \mu(e_2) + \mu(e_5) \cdot i_D \cdot i_A \\ \mu(e_3) \end{pmatrix} = \begin{pmatrix} 0.25 + 0.25 \cdot i_C \cdot i_B \\ 1.0 \\ 0.25 + 0.25 \cdot i_D \cdot i_A \\ 1.0 \end{pmatrix}.$$

We can solve the equation analytically to gather the (least) fixed point i

$$i = \begin{pmatrix} \frac{1}{3} & 1 & \frac{1}{3} & 1 \end{pmatrix}^T.$$

By applying fixed point method, we also compute the (least) fixed point as shown in Table 1. We truncate the values after 2 decimals. Due to the fact that $i_4 = i_5$, we stop the iteration ending up with approximately the same (least) fixed point. □

	i_0	i_1	i_2	i_3	i_4	i_5
A	0.00	0.25	0.31	0.32	0.33	0.33
B	0.00	1.00	1.00	1.00	1.00	1.00
C	0.00	0.25	0.31	0.32	0.33	0.33
D	0.00	1.00	1.00	1.00	1.00	1.00

Table 1: fixed point method for inside weights on example hypergraph

	o_0	o_1	o_2	o_3	o_4
A	0.00	1.00	1.00	1.06	1.06
B	0.00	0.00	0.08	0.08	0.08
C	0.00	0.00	0.25	0.25	0.26
D	0.00	0.00	0.00	0.02	0.02

Table 2: fixed-point method for outside weights on example hypergraph

3.1.3 Outside weights as fixed-point problem

Definition 15 (fixed-point function of the outside weights) Let $H = (V, E, \mu)$ be a weighted hypergraph with $V = \{A_0, ..., A_m\}$, the starting symbol S and s the function from Definition 11. Then the *fixed-point function of the outside weights* is

$$out\colon \mathbb{R}^V \to \mathbb{R}^V$$

$$out(o) = \begin{pmatrix} s(A_0) + \sum\limits_{\substack{e \in E \\ e=(w,\sigma,B) \\ w_i = A_0}} o_B \cdot \mu(e) \cdot \prod\limits_{\substack{j \in [|w|] \\ i \neq j}} \text{inner}(w_j) \\ \vdots \\ s(A_m) + \sum\limits_{\substack{e \in E \\ e=(w,\sigma,B) \\ w_i = A_m}} o_B \cdot \mu(e) \cdot \prod\limits_{\substack{j \in [|w|] \\ i \neq j}} \text{inner}(w_j) \end{pmatrix}.$$

Example 4 (fixed-point function of outside weights) For the hypergraph H from Example 1, the fixed-point function for the outside weights is

$$out(o) = \begin{pmatrix} 1 + o_C \cdot \mu(e_5) \cdot \text{inner}(D) \\ o_A \cdot \mu(e_4) \cdot \text{inner}(C) \\ o_A \cdot \mu(e_4) \cdot \text{inner}(B) \\ o_C \cdot \mu(e_5) \cdot \text{inner}(A) \end{pmatrix} = \begin{pmatrix} 1 + 0.25 \cdot o_c \\ \frac{1}{12} \cdot o_A \\ \frac{1}{4} \cdot o_A \\ \frac{1}{12} \cdot o_C \end{pmatrix}$$

By analytically solving the equation we get the (least) fixed point o

$$o = \begin{pmatrix} \frac{16}{15} & \frac{4}{45} & \frac{4}{15} & \frac{1}{45} \end{pmatrix}^T.$$

The fixed-point method computes approximately the same (least) fixed point, shown in Table 2. We truncated the values after 2 decimals. □

3.2 Newton's Method

Newton's method converges, compared to fixed-point iteration, faster. In our case, we have quadratic convergence [EKL08].

3.2.1 Inside weights as root of a function

We can define a function in' such that the (least) solution of $in'(i) = 0$ is the (least) fixed point of the function in Definition 15. Let $V = \{A_0, ..., A_m\}$ be the set of vertices of H and assume $i \in \mathbb{R}^V$ to be the fixed point of in, then $in(i) = i$. We then have that $0 = -i + in(i)$. More verbose we get

$$0 = -\begin{pmatrix} i_{A_0} \\ \vdots \\ i_{A_m} \end{pmatrix} + \begin{pmatrix} \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_0)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \\ \vdots \\ \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_m)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \end{pmatrix} = \begin{pmatrix} -i_{A_0} + \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_0)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \\ \vdots \\ -i_{A_m} + \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_m)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \end{pmatrix}.$$

Definition 16 Let $H = (V, E, \mu)$ be a weighted hypergraph, $V = \{A_0, ..., A_m\}$. Then in' is the function

$$in' \colon \mathbb{R}^V \to \mathbb{R}^V$$
$$in'(i) = \begin{pmatrix} -i_{A_0} + \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_0)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \\ \vdots \\ -i_{A_m} + \sum\limits_{\substack{e \in E \\ e=(w,\sigma,A_m)}} \mu(e) \cdot \prod\limits_{j \in [|w|]} i_{w_j} \end{pmatrix}.$$

Example 5 (Inside weights as root of a function) For the hypergraph G in Example 1 we get the following function in':

$$in'(i) = \begin{pmatrix} -i_A + \mu(e_0) + \mu(e_4) \cdot i_C \cdot i_B \\ -i_B + \mu(e_1) \\ -i_C + \mu(e_2) + \mu(e_5) \cdot i_D \cdot i_A \\ -i_D + \mu(e_3) \end{pmatrix}.$$

□

3.2.2 Newton's Method

Newton's method has instances for a variety of functions. It is especially used in mathematical analysis to get the root of functions in $\mathbb{R}^{\mathbb{R}}$. But since its definition is more general, we can also use it in our case.

Definition 17 (Newton's method) Let $f \colon \mathbb{R}^V \to \mathbb{R}^V$ be a function with (at least) one root, $x_0 \in \mathbb{R}^V$ (be a *starting value*) and $c \colon \mathbb{R}^V \times \mathbb{R}^V \to \mathbb{B}$ be a predicate (*breaking condition*). We define the sequence $\langle x_0, x_1, ... \rangle$ where $x_{i+1} = x_i - (f'(x_i))^{-1} \cdot f(x_i)$. *Newton's method* determines the smallest i such that $c(x_i, x_{i+1})$ holds and returns x_i as *result*. □

The function, we need to find the root of, has the type $f \colon \mathbb{R}^V \to \mathbb{R}^V$. The derivation for such a function is the *Jacobian matrix*.

Definition 18 (Jacobian matrix) Without loss of generality, let $V = \{v_0, ..., v_k\}$ be the index values, $f \colon \mathbb{R}^V \to \mathbb{R}^V$ a function over real vectors indexed by V with $f =$

k	i_k	$in'(i_k)$	$Jin'(i_k)$	i_{k+1}
0	$\begin{pmatrix}0\\0\\0\\0\end{pmatrix}$	$\begin{pmatrix}0.25\\1\\0.25\\1\end{pmatrix}$	$\begin{pmatrix}-1&0&0&0\\0&-1&0&0\\0&0&-1&0\\0&0&0&-1\end{pmatrix}$	$\begin{pmatrix}0.25\\1\\0.25\\1\end{pmatrix}$
1	$\begin{pmatrix}0.25\\1\\0.25\\1\end{pmatrix}$	$\begin{pmatrix}0.06\\0\\0.06\\0\end{pmatrix}$	$\begin{pmatrix}-1&0.06&0.25&0\\0&-1&0&0\\0.25&0&-1&0.06\\0&0&0&-1\end{pmatrix}$	$\begin{pmatrix}0.33\\1\\0.33\\1\end{pmatrix}$
2	$\begin{pmatrix}0.33\\1\\0.33\\1\end{pmatrix}$	$\begin{pmatrix}0\\0\\0\\0\end{pmatrix}$	$\begin{pmatrix}-1&0.08&0.25&0\\0&-1&0&0\\0.25&0&-1&0.08\\0&0&0&-1\end{pmatrix}$	$\begin{pmatrix}0.33\\1\\0.33\\1\end{pmatrix}$

Table 3: Newton's method for inside weights on example hypergraph

$\begin{pmatrix}f_{v_0} & \dots & f_{v_k}\end{pmatrix}^T$ and $x = \begin{pmatrix}x_{v_0} & \dots & x_{v_k}\end{pmatrix}^T \in \mathbb{R}^V$. The *Jacobian matrix of f in x* is

$$Jf(x) = \begin{pmatrix}\frac{df_{v_0}(x)}{dx_{v_0}} & \dots & \frac{df_{v_0}(x)}{dx_{v_k}}\\ \vdots & \ddots & \vdots\\ \frac{df_{v_k}(x)}{x_{v_0}} & \dots & \frac{df_{v_k}(x)}{x_{v_k}}\end{pmatrix}.$$

□

Example 6 (Newton's method) For the function in' in Example 5 we get the Jacobian matrix

$$Jin'(i) = \begin{pmatrix}-1 & \mu(e_4)\cdot i_C & \mu(e_4)\cdot i_B & 0\\ 0 & -1 & 0 & 0\\ \mu(e_5)\cdot i_D & 0 & -1 & \mu(e_5)\cdot i_A\\ 0 & 0 & 0 & -1\end{pmatrix}.$$

We us the predicate $c\colon \mathbb{R}^V \times \mathbb{R}^V \to \mathbb{B}$ with $c(x, x') = \top \iff x \approx x'$ as breaking condition. The iteration steps of Newton's method are shown in Table 3. We truncate the shown values after 2 decimals for better readability. □

3.3 Decomposed Newton's method

The idea of *decomposed Newton's method* (DNM) is to decompose the hypergraph into *strongly connected components* and then run Newton's method for every component. The components must be in topological order. Hence, for every instance of Newton's method, only the inside weights for the vertices inside the component are unknown while the weights of all incoming vertices outside the component have already been calculated.

3.3.1 Decomposing the hypergraph

Generally speaking, a *strongly connected component* is a set of vertices where every pair of vertices is mutually reachable. We generalize the definition for directed graphs to weighted hypergraphs.

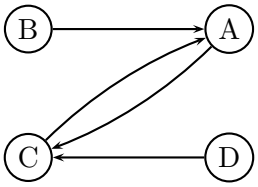

Figure 3: directed graph constructed from the example hypergraph

Definition 19 (strongly connected components) Let $H = (V, E, \mu)$ be a weighted Σ-hypergraph. Then for every $v \in V$ the *strongly connected component associated with* v is $\mathrm{SCC}(H, v) = \{v' \in V \mid v \hookrightarrow_H^* v' \wedge v' \hookrightarrow_H^* v\}$. The *strongly connected components* of H are elements of the set $\mathrm{SCCs}(H) = \{\mathrm{SCC}(H, v) \mid v \in V\}$. □

Example 7 (strongly connected components of a hypergraph) By Definition 19, our example hypergraph has the strongly connected components $\mathrm{SCCs}(H) = \{\{A, C\}, \{B\}, \{D\}\}$. □

In order to calculate the SCCs of a hypergraph H, we will use *Tarjan's algorithm* [Tar72]. Due to the fact that Tarjan's algorithm is designed for directed graphs, we construct a directed graph G such that $\mathrm{SCCs}(H) = \mathrm{SCCs}(G)$.

Construction 1 (weighted hypergraph to directed graph) Let $H = (V, E, \mu)$ be a weighted Σ-hypergraph. We construct the directed graph $G = (V, \hookrightarrow_H)$.

Example 8 (weighted hypergraph to directed graph) By applying Construction 1 to the hypergraph in Example 1, we get a Graph $G = (V, E)$ with $E = \{(B, A), (A, C), (C, A), (D, C)\}$. A graphical representation is shown in Figure 3. □

Theorem 1 *Let $H = (V, E, \mu)$ be a weighted Σ-hypergraph and $G = (V, E')$ the directed graph by construction 1. Then* $\mathrm{SCCs}(H) = \mathrm{SCCs}(G)$.

Proof Given the fact, that $\hookrightarrow_H = \hookrightarrow_G$, $\hookrightarrow_H^* = \hookrightarrow_G^*$ holds. Furthermore $\mathrm{SCCs}(H)$ only depends on V and $\hookrightarrow_H^*$, and $\mathrm{SCCs}(G)$ only depends on V and $\hookrightarrow_G^*$. Hence $\mathrm{SCCs}(H) = \mathrm{SCCs}(G)$. ■

Construction 2 (directed acyclic graph of SCCs) Given a graph $G = (V, E)$ and the set of strongly connected components $\mathrm{SCCs}(G)$, we can associate a directed acyclic graph G' by constructing $G' = (\mathrm{SCCs}(H), E')$ where $E' = \{(\mathrm{SCC}(G, v), \mathrm{SCC}(G, w)) \mid (v, w) \in E, v \notin \mathrm{SCC}(G, w)\}$.

Since all cyclic paths are removed during Construction 2, the generated G' is in fact a directed acyclic graph. Tarjan's algorithm computes $\mathrm{SCCs}(G)$ in a reversed topological order of G'. Therefore we only need to reverse the calculated sequence of SCCs to get a topological order.

Example 9 (directed acyclic graph of SCCs) Given the graph G constructed in Example 8, Tarjan's algorithm calculates the strongly connected components $\mathrm{SCCs}(G) = \{\{A, C\}, \{B\}, \{D\}\}$. First we recognize, they are in fact equal to the strongly connected components of the hypergraph H. By applying Construction 2, we get the directed acyclic graph $G' = (\mathrm{SCCs}(G), E'')$ with $E'' = \{(\{B\}, \{A, C\}), (\{D\}, \{A, C\})\}$, shown in Figure 4. Hence, a topologically sorted sequence of the strongly connected components of H is $\langle\{B\}, \{D\}, \{A, C\}\rangle$. □

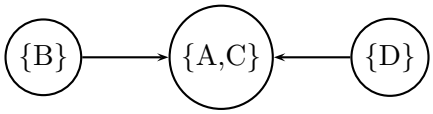

Figure 4: directed acyclic graph by hypergraph

k	i_k	$in'_{\{B\}}(i_k)$	$Jin'_{\{B\}}(i_k)$	i_{k+1}
0	$\begin{pmatrix}0\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}-1\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$
1	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}0\end{pmatrix}$	$\begin{pmatrix}-1\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$

Table 4: Newton's method for $H_{\{B\}}$

3.3.2 Applying Newton's method

Let $H = (V, E, \mu)$ be the given weighted Σ-hypergraph and S a topologically sorted sequence of all SCCs of H. We initialize DNM method with an empty variable assignment $i \colon \mathbb{R}^V_{\geq 0}, i = \emptyset$. Then for every $s \in S$, we intersect H with s leading to H_s.

Definition 20 (intersection of weighted hypergraph and a set of verties) Let $H = (V, E, \mu)$ be a weighted Σ-hypergraph and $S \subseteq V$. The *intersection of H with S* is defined by $H_S = (V', E', \mu')$ where $E' = \{e \in E \mid \text{head}(e) \in S\}, V' = \{v \in V \mid \exists e \in E' \colon v \in \text{tail}(e) \vee v = \text{head}(e)\}$ and for all $e \in E' \colon \mu'(e) = \mu(e)$. □

For H_S we calculate the function in' by Definition 16. Then we use the variable assignment to substitute the variables that are not in S with values and remove the lines corresponding to the substituted values from in'. Now we got a system of dimension $|S| \times |S|$ for which we can use standard Newton's method.

Example 10 (decomposed Newton's method) By intersecting the example hypergraph H with the SCCs from Example 9 $\langle \{B\}, \{D\}, \{A, C\} \rangle$, we get a sequence of hypergraphs $\langle H_{\{B\}}, H_{\{D\}}, H_{\{A,C\}} \rangle$. Then we apply Newton's method to each hypergraph in the sequence in order.

For the hypergraph $H_{\{B\}}$ we get

$$\begin{aligned} H_{\{B\}} &= (\{B\}, \{e_1\}) \\ in'_{\{B\}} &= \begin{pmatrix} -i_B + \mu(e_1) \end{pmatrix} \\ &= \begin{pmatrix} 1 - i_B \end{pmatrix} \\ Jin'_{\{B\}} &= \begin{pmatrix} -1 \end{pmatrix} \\ (Jin'_{\{B\}})^{-1} &= \begin{pmatrix} -1 \end{pmatrix}. \end{aligned}$$

Then by Newton's method in Table 4

$$i_B = 1.$$

k	i_k	$in'_{\{D\}}(i_k)$	$Jin'_{\{D\}}(i_k)$	i_{k+1}
0	$\begin{pmatrix}0\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}-1\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$
1	$\begin{pmatrix}1\end{pmatrix}$	$\begin{pmatrix}0\end{pmatrix}$	$\begin{pmatrix}-1\end{pmatrix}$	$\begin{pmatrix}1\end{pmatrix}$

Table 5: Newton's method for $H_{\{D\}}$

k	i_k	$in'_{\{A,C\}}(i_k)$	$Jin'_{\{A,C\}}(i_k)$	i_{k+1}
0	$\begin{pmatrix}0\\0\end{pmatrix}$	$\begin{pmatrix}0.25\\0.25\end{pmatrix}$	$\begin{pmatrix}-1 & 0.25\\0.25 & -1\end{pmatrix}$	$\begin{pmatrix}0.33\\0.33\end{pmatrix}$
1	$\begin{pmatrix}0.33\\0.33\end{pmatrix}$	$\begin{pmatrix}0\\0\end{pmatrix}$	$\begin{pmatrix}-1 & 0.25\\0.25 & -1\end{pmatrix}$	$\begin{pmatrix}0.33\\0.33\end{pmatrix}$

Table 6: Newton's method for $H_{\{B\}}$

For the hypergraph $H_{\{D\}}$ we get

$$\begin{aligned} H_{\{D\}} &= (\{B\}, \{e_3\}) \\ in'_{\{D\}} &= \begin{pmatrix} -i_D + \mu(e_3) \end{pmatrix} \\ &= \begin{pmatrix} 1 - i_D \end{pmatrix} \\ Jin'_{\{D\}} &= \begin{pmatrix} -1 \end{pmatrix} \\ (Jin'_{\{D\}})^{-1} &= \begin{pmatrix} -1 \end{pmatrix}. \end{aligned}$$

Then by Newton's method in Table 5

$$i_D = 1.$$

For the hypergraph $H_{\{A,C\}}$ we get

$$\begin{aligned} H_{\{A,C\}} &= (\{A, B, C, D\}, \{e_0, e_2, e_4, e_5\}) \\ in'_{\{A,C\}} &= \begin{pmatrix} -i_A + \mu(e_0) + \mu(e_4) \cdot i_C \cdot i_B \\ -i_C + \mu(e_2) + \mu(e_5) \cdot i_D \cdot i_A \end{pmatrix} \\ &= \begin{pmatrix} -i_A + 0.25 + 0.25 \cdot i_C \\ -i_C + 0.25 + 0.25 \cdot i_A \end{pmatrix} \\ Jin'_{\{A,C\}} &= \begin{pmatrix} -1 & 0.25 \\ 0.25 & -1 \end{pmatrix} \\ (Jin'_{\{B\}})^{-1} &= -\frac{4}{15} \begin{pmatrix} 4 & 1 \\ 1 & 4 \end{pmatrix}. \end{aligned}$$

Then by Newton's method in Table 6

$$\begin{aligned} i_A &= \frac{1}{3}, \\ i_C &= \frac{1}{3}. \end{aligned}$$

□

4 Implementation

The following algorithms give a procedural overview of the approaches made, as well as the Haskell functions in charge of each step (denoted in square brackets).

4.1 Convergence

The fixed-point method and Newton's method both require a break condition in the sense of Definition 15 and Definition 17 in order to select the result. For that purpose we define the *correction ratio* which should give us a measure on how much the method improved the value during its last iteration.

Definition 21 (correction ratio) Let $x, x' \in \mathbb{R}_{\geq 0}$ be two values. The *correction ratio* ε *from* x *to* x' is defined as the value $\varepsilon(x, x') = \frac{|x'-x|}{x}$. □

By Definition 21 every step of the iteration is associated with a correction ratio $\varepsilon(x_i, x_{i+1})$. The methods break the iteration if that ratio drops below a threshold ε_0.

```
converged e0 x y = if   x ≥ y
                   then (x - y) / x < e0
                   else (y - x) / x < e0
```

We represent `converged e0` by the relation $\approx \subseteq \mathbb{R} \times \mathbb{R}$. In order to evaluate the convergence of vectors, we need to extend $\approx$ to $\mathbb{R}^V$.

Definition 22 ($\approx$) Let $\approx \subseteq \mathbb{R} \times \mathbb{R}$ be a relation and $x, y \in \mathbb{R}^V$ two vectors, then $\approx \subseteq \mathbb{R}^V \times \mathbb{R}^V$ is a relation where $x \approx y \iff \forall v \in V : x_v \approx y_v$. □

We then use `Data.Foldable.all (converged e0)` on two vectors (of the type `Data.Map`, which is an instance of `Data.Foldable`) to determine if we are done iterating.

4.2 Newton's method

Since Newton's method computes derivations of the multivariate function, the implementation computes a representation of that function and stores it as a List of `Polynomials`. For the implementation of Newton's method, we considers three approaches. The first one implements Newton's method as it is and reuses a symbolic Jacobian matrix, the second one calculates an (evaluated) Jacobian matrix in every iteration and the third one reuses the (evaluated) Jacobian matrix in several consecutive steps.

4.2.1 Approach 1 (symbolic JF)

We calculate a symbolic Jacobian matrix. Then we evaluate and invert it in each step (Figure 5).

`simpleDerive` Calculates the symbolic derivation of a polynomial given a variable to derive by

```
-- | Computes the derivation of a given function
simpleDerive
  :: (Ord v, Num w, Integral i)
  ⇒ Polynomial v w i
  → v
  → Polynomial v w i
simpleDerive pol x
  = let dy = [ (w * (fromIntegral a), list)
```

1. **define** a starting vector i_0 [0-vector]
2. **calculate** the symbolic Jacobian matrix JF [`derive`]
3. **for** $k \in [1, ..]$ **do**
   a) **evaluate** the Jacobian matrix $JF_k = JF(i_{k-1})$ [`evalPolyMatrix`]
   b) **invert** JF_k [`inv`]
   c) **evaluate** the function $F_k = F(i_{k-1})$ [`evalPolyVec`]
   d) **calculate** i_k. $i_k = i_{k-1} - JF_k^{-1} \cdot F_k$
   e) **break** if i_k and i_{k-1} are close enough [`converged`]
4. **output** the last calculated i_k

Figure 5: Newton's method, approach 1

```
9                | (w, oldList) ← pol
10               , let a = M.findWithDefault 0 x
11                       . M.fromList
12                       $ oldList
13               , let list = [ (v, i)
14                            | (v, iOld) ← oldList
15                            , let i = if   (v == x)
16                                      then iOld - 1
17                                      else iOld
18                            , i > 0
19                            ]
20               ]
21     in  dy
```

`derive` Calculates the symbolic Jacobian matrix given vectors of polynomials and variables.

```
1  -- | Computes the Jabobian Matrix of a given Function
2  derive
3    :: (Ord v, Num w, Integral i)
4    ⇒ [Polynomial v w i]
5    → [v]
6    → [[Polynomial v w i]]
7  derive vec vars
8    = [ list
9      | partialFunction ← vec
10     , let list = [ simpleDerive partialFunction x
11                  | x ← vars
12                  ]
13     ]
```

`evalPoly` Evaluates a polynomial given a variable assignment.

```
1  -- | Evaluates a 'Polynomial' by the given 'M.Map'
2  evalPoly
3    :: (Ord v, Num w, Eq w, Integral i)
4    ⇒ M.Map v w
5    → Polynomial v w i
6    → w
7  evalPoly w p
8    = let
9          f k1 []
```

1. **define** a starting vector i_0 [0-vector]
2. **for** $k \in [1, ..]$ **do**
   a) **calculate and evaluate** the Jacobian matrix JF_k [`deriveEval`]
   b) **invert** JF_k [`inv`]
   c) **evaluate** the function $F_k = F(i_{k-1})$ [`evalPolyVec`]
   d) **calculate** i_k. $i_k = i_{k-1} - JF_k^{-1} \cdot F_k$
   e) **break** if i_k and i_{k-1} are close enough [`converged`]
3. **output** the last calculated i_k

Figure 6: Newton's method, approach 2

```
10           = k1
11         f k1 ((w1,l):xs)
12          = g k1 w1 l xs
13         g k1 k2 [] xs
14          = let k' = (k1 + k2)
15             in  seq k' (f k' xs)
16         g k1 k2 ((v,i):ys) xs
17          = let k2' = (k2 * (w M.! v) ^ i)
18             in  seq k2' (g k1 k2' ys xs)
19         r = f 0 p
20     in  r
```

`evalVec, evalMatrix` Extension of `evalPoly` to vectors and matrices of polynomials.

```
1  -- | Evaluates a 'List' of 'Polynomial's
2  evalPolyVec
3    :: (Ord v, Num w, Eq w, Integral i)
4    ⇒ M.Map v w
5    → [Polynomial v w i]
6    → [w]
7  evalPolyVec w = map $ evalPoly w
8
9  -- | Evaluates a Matrix of 'Polynomial's
10 evalPolyMatrix
11   :: (Ord v, Num w, Eq w, Integral i)
12   ⇒ M.Map v w
13   → [[Polynomial v w i]]
14   → [[w]]
15 evalPolyMatrix w = map $ evalPolyVec w
```

`inv` Inverts a Matrix using `Numeric.LinearAlgebra.Algorithms.inv`.

4.2.2 Approach 2 (calculate JF in each iteration)

In each step we calculate the inverted Jacobian matrix from scratch (Figure 6).

`deriveEval` Calculates and evaluates the Jacobian matrix given a function and variable assignment.

```
1  -- | Computes the Jabobian Matrix of a given Function
2  deriveEval
3    :: (Ord v, Num w, Integral i)
```

1. **define** a starting vector i_0 [0-vector]
2. **define** a step size n [$n = 5$]
3. **for** $k \in [1, ..]$ **do**
   a) **if** $k \mod n \equiv 1$
   then calculate the inverted Jacobian matrix JF_k^{-1} [`deriveEval`, `inv`]
   else reuse it $JF_k^{-1} = JF_{k-1}^{-1}$
   b) **evaluate** the function $F_k = F(i_{k-1})$ [`evalPolyVec`]
   c) **balculate** i_k. $i_k = i_{k-1} - JF_k^{-1} \cdot F_k$
   d) **break** if i_k and i_{k-1} are close enough [`converged`]
4. **output** the last calculated i_k

Figure 7: Newton's method, approach 3

```
4    ⇒ M.Map v w
5    → [Polynomial2 v w i]
6    → [v]
7    → [[w]]
8  deriveEval w vec vars
9    = let
10         h k a b = (*) b . flip (^) a . (M.!) w $ k
11         g x p = if   M.notMember x p
12                 then 0
13                 else ((w M.! x) ^ ((p M.! x) - 1))
14                    * (M.foldrWithKey' h 1 . M.delete x $ p)
15                    * (fromIntegral . (M.!) p $ x)
16         i v (x,y) = (*) x . g v $ y
17     in [ val
18        | partialFunction ← vec
19        , let val = [ L.foldl' (+) 0
20                        . map (i x)
21                        $ partialFunction
22                    | x ← vars
23                    ]
24        ]
```

This approach has advantages to the previous one for big matrices or small numbers of iterations. In both cases, the time saved due to reusing the symbolic Jacobian matrix does not justify the overhead of creating and evaluating it.

4.2.3 Approach 3 (reuse JF for several iterations)

Now we only calculate and invert the Jacobian matrix every n steps. The algorithm saves time for big matrices (Figure 7). According to my runs and [NS08], $n = 5$ works. However, we use Approach 2 for Vanda, because it has a highter performance than Approach 1 and Approach 3 is not formally correct.

4.3 Decomposed Newton's Method

As decomposed Newton's method (DNM) is based on splitting the hypergraph into its strongly connected components (SCCs), topologically sorting them and then apply regu-

1. **calculate** SCCs of hypergraph h. $l = sccs(h) \subseteq 2^{2^V}$
2. **sort** SCCs topologically. $l' = topSort(l) \subseteq (2^V)^*$ [`topSortedSCCs`]
3. **set** $w \in \mathbb{R}^V_{\geq 0}$ to $\emptyset$
4. **for** every $s \in l'$ **do**
   a) **intersect** h and s. $h_s = intersect(h, s)$ [`intersectHypergraph`]
   b) **run** regular Newton's method on h_s considering already calculated weights. $w_s = newton_w(h_s)$ [`newton`]
   c) **set** $w := w \cup w_s$
5. **output** w

Figure 8: Decomposed Newton's method

lar Newton's method to the hypergraphs consisting of the SCCs in order of the sorting (Figure 8).

`topSortedSCCs` Converts the hypergraph to a graph, then calculated its SCCs and sorts them topologically using `Data.Graph`

```
-- | Computes a topologically sorted list of SCCs
--   of a 'Hypergraph'
topSortedSCCs
  :: (Ord v)
  ⇒ H.Hypergraph v l w i
  → [[v]]
topSortedSCCs hg
  = let es = map (λ (a,b) → (a, a, S.toList b))
           . M.toList
           . M.map S.fromList
           . M.fromListWith (++)
           . map (λ e → (H.eHead e, H.eTail e))
           . H.edges
           $ hg
        sccs = map G.flattenSCC . G.stronglyConnComp $ es
    in  sccs
```

`intersectHypergraph` Only keeps edges if the head is in the list

```
-- | Intersect a 'Hypergraph' with a 'List' of
--   head vertices
intersectHypergraph
  :: (Ord v)
  ⇒ H.Hypergraph v l w i
  → [v]
  → H.Hypergraph v l w i
intersectHypergraph hg l
  = H.hypergraph . L.foldl' (++) []
      . map ((M.!) (H.edgesM hg)) $ l
```

`newton` Utilizes one of the algorithms described in the previous section.

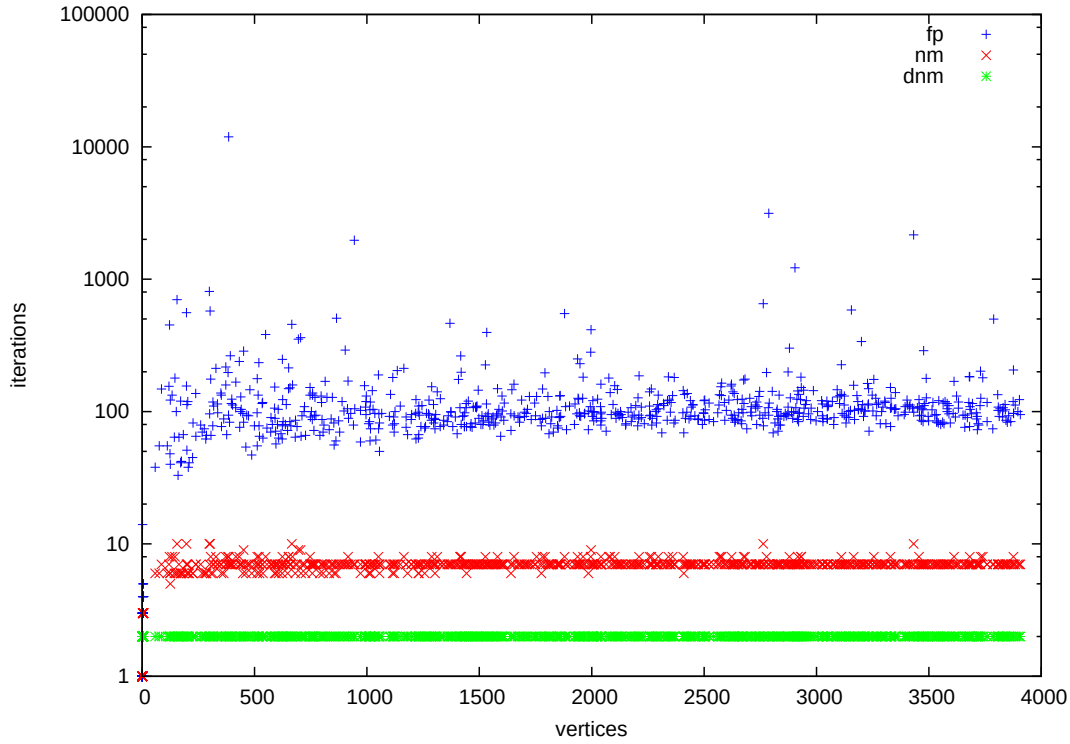

Figure 9: iterations by vertices for random hypergraphs, blue: fixed-point iteration, red: Newton's method, green: decomposed Newton's method

5 Results

The three proposed methods to calculate inside and outside weights in weighted hypergraphs have been implemented in Haskell as part of the natural language processing toolbox *Vanda*, which is being developed at the Chair of Foundations of Programming, Faculty of Computer Science, TU Dresden. They have been tested on 1000 randomly generated hypergraphs with up to 4000 vertices, 2 edges per vertex and, since most hypergraphs in statistical machine translation are binarized, a maximum of 2 tail nodes per edge. The number of vertices of those hypergraphs is equally distributed in $[0, 4000]$. We investigated the number of iterations needed as well as the time needed to compute the weights.

The number of iterations needed for the algorithms to terminate are shown in Figure 9. For decomposed Newton's method, we used the *arithmetic mean* of the iterations of all instantiated Newton's methods instead of the total count.

The fixed-point iteration is significantly more scattered than the the other algorithms, visible even on the used logarithmic scale. With a maximum of 12000 and an average of about 100 iterations, fixed-point method also needs much more steps to converge. Both the other clusters are very dense and only need up to 10 (average 8 for Newton's method and 2 for decomposed Newton's method) iterations. Also for more than 1500 vertices, the average number of iterations needed to compute the weights in any of the algorithms stops to grow. In the matter of steps, Newton's method and decomposed Newton's method clearly converge much faster than fixed-point method.

In Figure 10, we see the time needed by the algorithms to terminate. All clusters are dense. Since Newton's method depends on matrix inversion, which is very expensive, and

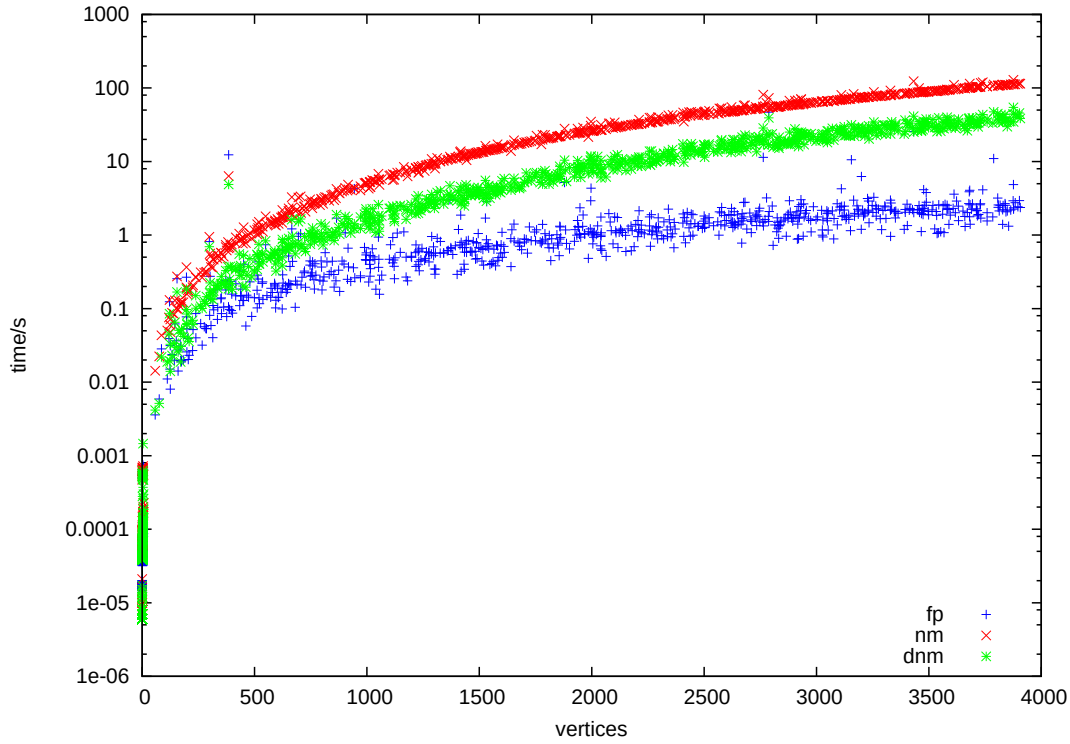

Figure 10: time by vertices for random hypergraphs, blue: fixed-point iteration, red: Newton's method, green: decomposed Newton's method

even the significantly smaller number of iterations can not compensate for that, Newton's method and decomposed Newton's method needs more time to converge than fixed-point method. Decomposed Newton's methods run time additionally depends on the sizes of the strongly connected components. The average ratio between the run time of fixed-point method to Newton's method to decomposed Newton's method is show in Table 7. Depending on the number of vertices, for decomposed Newton's method, the ratio grows linear, for Newton's method it grows super linear.

For randomly generated hypergraphs, fixed-point method proofs to be the best of the three methods. However, for special cases of hypergraphs, e.g. such generated by tools for statistical machine translation, decomposed Newton's method may proof more effective than here.

number of vertices	avg ratio `fp` to `nm`	avg ratio `fp` to `dnm`
$[900, 1100]$	$1 : 8.9$	$1 : 2.9$
$[1900, 2100]$	$1 : 21.7$	$1 : 7.0$
$[2900, 3100]$	$1 : 29.0$	$1 : 10.0$
$[3900, 4000]$	$1 : 44.2$	$1 : 13.2$

Table 7: average ratio of runtime for fixed-point method, Newton's method and decomposed Newton's method

References

[EKL08] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Convergence thresholds of Newton's method for monotone polynomial equations. In Pascal Weil and Susanne Albers, editors, *Proceedings of the 25th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 289–300, Bordeaux, France, 2008. `http://arxiv.org/abs/0802.2856`.

[NS08] Mark-Jan Nederhof and Giorgio Satta. Computing partition functions of pcfgs. *Research on Language and Computation*, 6:139–162, 2008. 10.1007/s11168-008-9052-8, `http://dx.doi.org/10.1007/s11168-008-9052-8`.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4569669`.