Universität Konstanz
FB Informatik und Informationswissenschaft
Bachelor-Studiengang Information Engineering

Bachelorarbeit

Ranking von Partitionen
Ranking of Partitions

zur Erlangung des akademischen Grades eines
Bachelor of Science (B.Sc.)

Studienfach:	Information Engineering
Schwerpunkt:	Informatik
Themengebiet:	Grundlagen der Informatik

von
Matthias Fratz

Erstgutachter:	Dr. Sven Kosub
Zweitgutachter:	Prof. Dr. Ulrik Brandes
Betreuer:	Dr. Sven Kosub
Einreichung:	8. Oktober 2009

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-0-299695

Zusammenfassung

Beim *Ranking von Partitionen* wird versucht, anhand eines Rankings von Individuen, die in mehrere Gruppen partitioniert sind, ein Ranking dieser Gruppen (Partitionen) zu erstellen, welches mit dem Ranking der Individuen konsistent ist. Ein Ansatz dafür ist die Repräsentantenmethode, bei der die Ordnungen, welche durch das Ranking der Individuen auf den Repräsentantensystemen der Menge der Partitionen induziert werden, durch eine geeignete Social Welfare Funktion zu einer Ordnung der Partitionen aggregiert werden.

Aufgrund der potentiell sehr großen Zahl von Repräsentantensystemen ist die tatsächliche Bestimmung eines solchen Partitionsrankings über die Definition praktisch unmöglich. Daher wird ein effizienter Algorithmus vorgestellt, der für eine spezielle Familie von Social Welfare Funktionen, die Rangwertsysteme (eine Generalisierung des Borda-Count), das Partitionsranking für n Individuen und k Partitionen in $O(n \log n \cdot k^3 \log k)$ Zeit bestimmt.

Dieser Algorithmus wurde unter anderem auf ein Ranking von 742 nach Ländern partitionierten Fußballclubs angewandt, um ein Ranking der 53 Länder zu erhalten. Die Auswirkungen verschiedener Implementierungsvarianten auf die Laufzeit des Algorithmus wurden ebenfalls analysiert.

Des Weiteren wird eine axiomatische Charakterisierung der geforderten Konsistenz zwischen den beiden Rankings versucht. Zwei der Axiome stellen Forderungen über das Verhalten bei der Vertauschung zweier benachbarter Elemente im Ranking der Individuen, das dritte fordert die Erhaltung einer Eigenschaft des Rankings der Individuen im Ranking der Partitionen.

Abstract

Given a ranking of individuals partitioned into several groups, *ranking of partitions* tries to create a ranking of those groups (partitions) which is consistent with the ranking of the individuals. One way of doing this is the method of representatives, which uses a social welfare function to aggregate the orders induced on the systems of representatives of the set of partitions by the ranking of individuals, into a ranking of the partitions.

Because of the potentially huge number of systems of representatives, actually determining such a ranking of partitions according to the definition is practically impossible. Therefore an efficient algorithm, capable of computing the ranking of partitions for n individuals, k partitions and a special class of social welfare functions, the positional voting methods (generalizations of the Borda Count), in $O(n \log n \cdot k^3 \log k)$ time is presented.

This algorithm was applied to, among other rankings, one of 742 football (soccer) clubs partitioned by nation, in order to determine a ranking of the 53 nations. The impacts of various implementation variants on the running time of the algorithm were also analyzed.

Furthermore, an axiomatic characterization of the demanded consistency between the two rankings is attempted. Two of these axioms place demands on the behaviour caused by exchanging two adjacent elements of the ranking of individuals, whereas the third one describes a certain property of the ranking of the individuals which must be preserved in the ranking of the partitions.

Inhaltsverzeichnis

1 Einleitende Definitionen

1.1 Das Rankingproblem für Partitionen

Das *Rankingproblem für Partitionen* besteht grob gesagt darin, aus einem Ranking von Individuen ein Ranking von "Gruppen" dieser Individuen zu erstellen, das *konsistent*[1] mit dem Ranking der Individuen ist.

Gegeben ist also ein *Ranking der Individuen*, dargestellt als Folge $(x_i)_{i=1,\dots,n}$, wobei x_1 das beste und x_n das schlechteste Element ist. Diese Folge induziert sowohl eine Menge

$$X := \{x_i : 1 \leq i \leq n\}$$

aller Individuen, als auch eine "besser-als" Ordnung $\prec_X \subseteq X \times X$ definiert durch

$$x_i \prec_X x_j :\Longleftrightarrow i < j$$

Dies ist eine strenge Totalordnung, keine zwei Elemente sind gleich "gut."

Weiterhin gegeben ist eine Menge $P = \{p_1, \dots, p_k\}$ von k Partitionen von X. Diese kann auch dargestellt werden als Funktion $part : X \to P$, die jedem Element $x_i \in X$ ihre Partition $p \in P$ zuweist, so dass also $x_i \in part(x_i)$.

Bestimmt werden soll ein *Ranking der Partitionen*, also eine Ordnung $\prec_P$ auf P, die *konsistent* mit (x_i) (beziehungsweise $\prec_X$) ist.

Im Idealfall ist $\prec_P$ eine strenge Totalordnung (d.h. zwei beliebige Partitionen sind stets vergleichbar und nie gleich gut), denn dann kann das Ranking entsprechend (x_i) als Folge $(g_i)_{i=1,\dots,k}$ dargestellt werden. Allerdings soll hier auch zugelassen werden, dass $\prec_P$ eine strenge Ordnung (d.h. zwei Partitionen können nicht verglichen werden, ohne dabei notwendigerweise gleich gut zu sein) oder eine strenge schwache Ordnung (d.h. zwei Partitionen können gleich gut sein) ist.

Es ist von Vorteil, wenn $\prec_P$ wenigstens eine strenge schwache Ordnung ist, dann kann das Partitionsranking zwar nicht als Folge wie (x_i) dargestellt werden, aber die Darstellung als Liste, in der sich mehrere Einträge eine Position teilen können, ist weiterhin möglich. Diese kann (abgesehen von der im Sportbereich üblichen Konvention, bei Gleichstand Plätze zu überspringen) aufgefasst werden als eine strenge Totalordnung auf den Äquivalenzklassen der Relation $\sim$, definiert durch

$$p \sim q :\Longleftrightarrow p \not\prec_P q \wedge p \not\succ_P q$$

Somit kann das Partitionsranking in diesem Fall als Folge (e_i) von Äquivalenzklassen dargestellt werden.

Bei einer strengen Ordnung ist dies nicht möglich, da mangels negativer Transitivität der Relation $\prec_P$ die abgeleitete Relation $\sim$ nicht transitiv ist und damit keine Äquivalenzklassen besitzen kann.

[1] In Abschnitt 4 wird versucht, die Bedeutung von Konsistenz in diesem Zusammenhang axiomatisch zu charakterisieren.

1.2 Die Repräsentantenmethode

Zur Bestimmung von $\prec_P$ soll die *Repräsentantenmethode* verwendet werden. Dazu sei

$$R := \{r \subseteq X : \forall p \in P : |r \cap p| = 1\}$$

die Menge aller Repräsentantensysteme[2] von P, und $repr_r : P \to X$ so, dass

$$repr_r(p) \in p \cap r$$

für alle $p \in P$ und $r \in R$, das heißt $repr_r(p)$ weist jeder Partition $p \in P$ ihren Repräsentanten im Repräsentantensystem r zu. Für jedes $r \in R$ existiert damit eine strenge Totalordnung $\prec_r \subseteq P \times P$ definiert durch

$$p \prec_r q :\Longleftrightarrow repr_r(p) \prec_X repr_r(q) \tag{1}$$

für alle $p, q \in P$.

Sei nun $R' := \{\prec_r : r \in R\}$ die Multimenge dieser Ordnungen, und $\tilde{R}$ die Menge aller auf P möglichen Ordnungen. Die Elemente von R' werden dann mit Hilfe einer Social Welfare Funktion $F : \tilde{R}^{|R|} \to \tilde{R}$ aggregiert zu $\prec_P \subseteq P \times P$.

Hier soll eine spezielle Klasse von Social Welfare Funktionen verwendet werden, die sogenannten Rangwertverfahren.[3] Dazu liefere die Funktion $pos_r : P \to \{1, \ldots, k\}$ für jedes $r \in \tilde{R}$ und jedes $p \in P$ die Position der Partition p in der Ordnung $\prec_r$; $pos_r(p)$ kann definiert werden als

$$pos_r(p) := 1 + |\{q \in P : q \prec_r p\}|$$

Ein Rangwertsystem wird definiert durch einen k-dimensionalen Vektor $a \in \mathbb{Z}^k$ mit $a_1 \geq a_2 \geq \cdots \geq a_k$.[4] Die Komponenten des Vektors sind die Punktzahlen, die eine Partition in Abhängigkeit von ihrer Position in den möglichen Repräsentantensystemen erhält: Wenn eine Partition p in einem Repräsentantensystem r an erster Stelle ist, dann erhält sie a_1 Punkte. Die Zweite erhält a_2 und so weiter bis zur Letzten, die a_k Punkte erhält. Allgemein erhält p durch r also genau $a_{pos_r(p)}$ Punkte.

Die Punkte werden über alle Repräsentantensysteme aufaddiert, insgesamt erhält eine Partition p also

$$points(p) := \sum_{r \in R} a_{pos_r(p)} \tag{2}$$

Punkte.

[2] englisch *system of representatives*

[3] Rangwertverfahren [8] umfassen einen großen Bereich der sozialen Entscheidungsfunktionen, wie beispielsweise Borda-Count [5] mit $a = (k-1, k-2, \ldots, 1, 0)$, Mehrheitswahl mit $a = (1, 0, \ldots, 0)$, Vetosysteme wie z.B. $a = (1, \ldots, 1, 0)$ usw.

[4] Die Einschränkung auf ganzzahlige Gewichtsvektoren stellt nur dann ein Problem dar, wenn die Gewichte irrationale Zahlen sein sollen. Alle rationalen Gewichte können so skaliert werden, dass der gesamte Vektor ganzzahlig ist. Irrationale Gewichte sind aber generell problematisch, da sie auf einem Computer nicht exakt darstellbar sind.

Je mehr Punkte eine Partition erhält, desto besser ist sie. Das Partitionsranking $\prec_P$ wird folglich definiert als

$$p \prec_P q :\Longleftrightarrow points(p) > points(q) \tag{3}$$

Die Partitionen werden also letztlich anhand ihrer Punktzahlen sortiert, wobei eine Partition mit mehr Punkten weiter vorne einsortiert wird.

Offensichtlich handelt es sich bei $\prec_P$ um eine strenge schwache Ordnung: Wenn zwei Partitionen $p, q \in P$ weder $p \prec_P q$ noch $p \succ_P q$ erfüllen, dann haben sie gleich viele Punkte und sind gleich gut.

2 Bestimmung eines Partitionsrankings

2.1 Grundlegende Überlegungen

Die sofort aus der Definition ersichtliche Methode zur Bestimmung solch eines Partitionsrankings wäre:[5]

Algorithmus 1 : Die offensichtliche Methode

```
points ← [0, ..., 0];      // k Elemente
part ← [1, ..., k];
foreach r ∈ R do
    sortiere part nach ≺_r;
    for i = 1, ..., k do
        j ← part[i];
        points[j] ← points[j] + a_i;
    end
end
```

Diese Lösung ist sehr einfach zu implementieren: Die Repräsentantensysteme können einfach rekursiv aufgezählt werden, und $repr_r(p)$ für die Sortierung nach $\prec_r$ mittels Gleichung (1) zu bestimmen (in $O(1)$), ist ebenfalls kein Problem. Allerdings gibt es potentiell sehr viele Repräsentantensysteme, denn es gilt[6]

$$|R| = \prod_{p \in P} |p|$$

was offensichtlich nicht polynomial ist, da beispielsweise mit $|p| = 2$ für alle $p \in P$:

$$\prod_{p \in P} |p| = 2^{\frac{n}{2}} = (\sqrt{2})^n$$

Mit dem Sortieren und der inneren **for**-Schleife ergibt sich dann eine sehr ungeeignete Gesamtrechenzeit von $O(|R| \cdot k \log k)$.

[5] Aus Gründen der Konsistenz werden auch im Pseudocode 1-basierte Indices verwendet.

[6] Beweis per Induktion über $|P|$

Zur praktischen Anwendung ist ein deutlich schnellerer Algorithmus erforderlich. Dazu müssen X und P durchnummeriert werden, was bei X durch die Definition über die Folge (x_i) ohnehin der Fall ist. $index_X(x_i)$ bezeichne also den Index eines Elements x_i in X, d.h.

$$index_X(x_i) = i$$

für alle $x_i \in X$. Die Partitionen seien ebenfalls (willkürlich) durchnummeriert, so dass entsprechend $index_P(p_j)$ der Index einer Partition p_j in P ist, d.h. $index_P(p_j) = j$ für alle $p_j \in P$.

Die Berechnung wird nun wie folgt in ihre wesentlichen Schritte zerlegt:

(0.) Falls das Individualranking nicht als Folge (x_i) vorliegt, sondern als Menge X mit Ordnung $\prec_X$, dann muss durch Sortieren die Folgen-Darstellung erhalten werden. Dies erfordert beispielsweise mit Mergesort $O(n \log n)$ Rechenzeit und $O(n)$ Speicherplatz.

1. Es wird eine Matrix $V = (v_{bj}) \in \mathbb{N}^{k \times k}$ so berechnet, dass v_{bj} genau die Anzahl der Repräsentantensysteme ist, bei denen die Partition p_b an Position j steht, also

   $$v_{bj} = |\{r \in R : pos_r(p_b) = j\}|$$

   Dazu werden für jedes $x_i \in X$ (aus Partition p_b) folgende 2 Schritte ausgeführt:

   (a) Berechne $\sigma = (\sigma_j) \in \mathbb{N}^k$, so dass

   $$\sigma_j := |\{r \in R : x_i \in r \wedge pos_r(p_b) = j\}| \tag{4}$$

   dass also σ_j die Anzahl der Repräsentantensysteme ist, bei denen x_i in der von $\prec_r$ induzierten Ordnung an Position j ist.

   σ kann berechnet werden, ohne alle $r \in R$ aufzuzählen. Die Vorgehensweise wird in Abschnitt 2.2 gezeigt. Da die direkte Anwendung dieser Methode immer noch eine Komplexität von $O(k \cdot 2^k)$ für ein einzelnes $x_i \in X$ hätte, wird in Abschnitt 2.3 ein Algorithmus vorgestellt, der mit $O(k^3 \log k \log n)$ für jedes einzelne $x_i \in X$ in Polynomialzeit läuft.

   (b) Addiere σ zur b-ten Zeile von V.

   $$(v_{b1}, \ldots, v_{bk}) \leftarrow (v_{b1} + \sigma_1, \ldots, v_{bk} + \sigma_k)$$

   Aufgrund der extrem großen Werte, die in den Komponenten von σ und V auftreten, beträgt die Größe einer Komponente bis zu $O(k \log n)$, und dieser Schritt benötigt pro Komponente bis zu $O(k \log n)$ Rechenzeit, insgesamt also $O(k^2 \log n)$.

Bei Verwendung des polynomialen Algorithmus beträgt die Laufzeit für diesen Schritt $O(n \log n \cdot k^3 \log k)$, und es wird $O(k^3 \log n)$ Speicher verbraucht.

2. $points(p_i)$ wird für alle $p_i \in P$ berechnet als

$$points(p_i) = \sum_{j=1,\ldots,k} a_j \cdot v_{ij}$$

Dieser Schritt erfordert $O(k^3 \log n)$ Rechenzeit (aufgrund der großen Werte) und so viel Speicherplatz, wie für die Ein- und Ausgabedaten erforderlich ist, also ebenfalls $O(k^3 \log n)$.

(3.) Falls das Ranking der Partitionen als Folge (g_i) oder (e_i) gesucht wird, muss P anhand von $\prec_P$ sortiert werden, was $O(k \log k)$ Vergleiche erfordert. Allerdings kann ein einzelner Vergleich der $O(k \log n)$ großen Punktzahlen auch bis zu $O(k \log n)$ Rechenzeit benötigen, so dass insgesamt $O(k^2 \log k \log n)$ Rechenzeit erforderlich sein kann.

Falls zwei verschiedene Partitionen $p_i, p_j \in P$ mit $p_i \neq p_j$ gleich viele Punkte erhalten haben, also falls $points(p_i) = points(p_j)$, dann ist $\prec_P$ nur eine strenge schwache Ordnung. Um $\prec_P$ dennoch als Folge (g_i) von Partitionen angeben zu können, muss zwischen p_i und p_j willkürlich eine Ordnung festgelegt werden.

Falls eine Folge (e_i) von Äquivalenzklassen gesucht wird, kann diese in $O(k)$ Zeit aus der Folge (g_i) bestimmt werden. Die Partitionen, die sich in einer bestimmten Äquivalenzklasse befinden, folgen in (g_i) unmittelbar nacheinander (unabhängig davon, welche willkürliche Ordnung gewählt wurde), so dass ein einzelner Durchlauf ausreicht, um alle Äquivalenzklassen zu erstellen.

Da der Großteil der Komplexität in Schritt 1a liegt, soll dieser Schritt im Folgenden genauer analysiert werden. In Abschnitt 2.4 wird dann Pseudocode für den kompletten ersten und zweiten Schritt vorgestellt. Die Umsetzung der anderen Schritte ist offensichtlich und wird hier nicht weiter betrachtet.

Es sei noch bemerkt, dass die Unterteilung der Schritte 1 und 2 gegenüber Algorithmus 1 den Speicherverbrauch erhöht, da sowohl *part* als auch *points* jeweils aus nur $O(k)$ Komponenten bestehen[7] und der Speicher für die Aufzählung der Repräsentantensysteme ebenfalls in $O(k)$ sind. Algorithmus 1 ist aber aufgrund seiner nicht polynomialen Rechenzeit ohnehin nicht praxisrelevant.

Allerdings wäre es auch beim polynomialen Algorithmus möglich, $points(p)$ direkt zu berechnen und damit mit $O(k)$ Komponenten auszukommen, indem Schritt 2 weggelassen wird und Schritt 1b verändert wird zu

$$points(p_b) \leftarrow points(p_b) \; + \; (a_1 \cdot \sigma_1, \ldots, a_k \cdot \sigma_k)$$

[7] Die k einzelnen Komponenten benötigen in der Regel mehr als eine konstante Menge Speicher, da die Werte extrem groß werden können; jede Komponente von *points* kann bis zu $O(k \log n)$ viele Stellen benötigen. (Dieses Problem wird in Abschnitt 2.3 genauer analysiert.) Allerdings benötigen die k^2 Komponenten der Matrix V jeweils gleich viel.

In der Tat ist die Unterscheidung der beiden Schritte eher formaler Natur. Im Pseudocode in Abschnitt 2.4 wird daher diese Optimierung verwendet, so dass eine quadratische Anzahl Komponenten vermieden werden kann.

2.2 Berechnung von σ_j

Für ein Element $x_i \in X$ und die Nummer $c \in \{1, \ldots, k\}$ einer *anderen* Partition (also $p_c \neq part(x_i)$) seien

$$before_i(c) := |\{y \in p_c : index_X(y) < i\}|$$
$$after_i(c) := |\{y \in p_c : index_X(y) > i\}|$$

also die Anzahl der Elemente der Partition p_c, die im Individualranking vor bzw. nach x_i platziert sind.

Damit kann σ_j für ein Element $x_i \in X$ aus Partition $p_b = part(x_i)$ berechnet werden als[8]

$$\sigma_j = \sum_{M \in \mathcal{P}_{j-1}(\{1,\ldots,k\}\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in \{1,\ldots,k\}\setminus\{b\}\setminus M} after_i(c) \right) \tag{5}$$

Um die Korrektheit zu zeigen, sei

$$[d] := \{1, \ldots, d\}$$

und $R(d)$, entsprechend zu R, die Anzahl der Repräsentantensysteme für die Partitionen $p_1, \ldots, p_d$, also

$$R(d) := \{r \subseteq \bigcup_{m \in [d]} p_m : \forall m \in [d] : |r \cap p_m| = 1\} \tag{6}$$

Offensichtlich gilt $R(k) = R$, es genügt also zu zeigen, dass für alle $j, k \in \mathbb{N}$ gilt

$$\sum_{M \in \mathcal{P}_{j-1}(\{1,\ldots,k\}\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in \{1,\ldots,k\}\setminus\{b\}\setminus M} after_i(c) \right)$$
$$= \sigma_j = |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}|$$

Behauptung: Für jedes $x_i \in X$ aus Partition $p_b = part(x_i)$, jedes $k \in \mathbb{N}$ und jedes $j \in \mathbb{N}$ gilt:

$$\sum_{M \in \mathcal{P}_{j-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right)$$
$$= |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}|$$

[8] $\mathcal{P}_k(A)$ ist, analog zur Potenzmenge $\mathcal{P}(A)$, die Menge der k-elementigen Teilmengen der Menge A, also $\mathcal{P}_k(A) := \{B \in \mathcal{P}(A) : |B| = k\}$. Damit ist $\mathcal{P}_k(A)$ auch für $k > |A|$ definiert: $\mathcal{P}_k(A) = \emptyset$ für $k > |A|$, denn A kann keine Teilmengen haben, die mehr Elemente enthalten als A selbst.

Beweis per Induktion über die Anzahl k der Partitionen. Dazu sei o.B.d.A. p_b die erste Partition laut der Nummerierung von P, also $b = 1$.[9]

Induktionsanfang: $k = 1$, damit ist $[k] \setminus \{b\} = \emptyset$.
Fall 1: $j = 1$, also ist $\mathcal{P}_{j-1}([k] \setminus \{b\}) = \{\emptyset\}$, und Gleichung (6) vereinfacht sich wie folgt:

$$\begin{aligned}
&\sum_{M \in \mathcal{P}_{j-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
&= \sum_{M \in \{\emptyset\}} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in \emptyset\setminus M} after_i(c) \right) \\
&= \prod_{c \in \emptyset} before_i(c) \cdot \prod_{c \in \emptyset} after_i(c) \\
&= 1
\end{aligned}$$

Für Gleichung (4) gilt entsprechend

$$|\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}| = |\{r \in R(1) : x_i \in r \wedge pos_r(p_b) = 1\}| = 1$$

denn alle Repräsentantensysteme $r \in R(1)$ haben genau ein Element, folglich kann die Position ihres einzigen Elements in der Ordnung $\prec_r$ nur 1 sein.
Fall 2: $j \geq 2$, also ist $\mathcal{P}_{j-1}([k] \setminus \{b\}) = \emptyset$, und Gleichung (6) wird sehr einfach.

$$\begin{aligned}
&\sum_{M \in \mathcal{P}_{j-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
&= \sum_{M \in \emptyset} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in \emptyset\setminus M} after_i(c) \right) \\
&= 0
\end{aligned}$$

Für Gleichung (4) gilt entsprechend

$$|\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}| = |\{r \in R(1) : x_i \in r \wedge 1 = j\}| = 0$$

denn wie in Fall 1 haben alle Repräsentantensysteme $r \in R(1)$ genau ein Element, folglich kann die Position der Partition ihres einzigen Elements in der Ordnung $\prec_r$ nicht ≥ 2 sein.

[9] Dies ist selbstverständlich nur dann für alle Elemente x_i möglich, wenn die Nummerierung von P davon anhängt, welches Element gerade bearbeitet wird – was aber auch kein Problem darstellt, da die Nummerierung von P ohnehin willkürlich ist.

Induktionsvoraussetzung: Es existiert ein $k \in \mathbb{N}$, so dass für alle $j \in \mathbb{N}$ gilt

$$\sum_{M \in \mathcal{P}_{j-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) = |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}|$$

Induktionsschritt: $k \rightsquigarrow k+1$

Fall 1: $j = 1$, damit also $\mathcal{P}_{j-1}([k+1] \setminus \{b\}) = \{\emptyset\}$.

$$\begin{aligned}
&\sum_{M \in \mathcal{P}_{1-1}([k+1]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k+1]\setminus\{b\}\setminus M} after_i(c) \right) \\
&= \sum_{M \in \{\emptyset\}} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k+1]\setminus\{b\}\setminus M} after_i(c) \right) \\
&= \prod_{c \in [k+1]\setminus\{b\}} after_i(c) \\
&= after_i(k+1) \cdot \prod_{c \in [k]\setminus\{b\}} after_i(c) \\
&= after_i(k+1) \cdot \sum_{M \in \{\emptyset\}} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
&= after_i(k+1) \cdot \sum_{M \in \mathcal{P}_{1-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
&\overset{IV}{=} after_i(k+1) \cdot |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = 1\}| \\
&= |\{y \in p_{k+1} : index_X(y) > i\}| \cdot |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = 1\}| \\
&= |\{(r, y) \in R(k) \times p_{k+1} : index_X(y) > i \wedge x_i \in r \wedge pos_r(p_b) = 1\}|
\end{aligned}$$

Da hier x_i als Repräsentant für p_b gewählt wird, gilt offensichtlich $pos_{r\cup\{y\}}(p_b) = 1$ genau dann, wenn $index_X(y) > i$, denn genau dann befindet sich die Partition von y (also p_{k+1}) auch in $\prec_{r\cup\{y\}}$ nach der Partition von x_i (also nach p_b), so dass p_b nicht von Position 1 "verdrängt" wird.

$$\begin{aligned}
&= |\{(r, y) \in R(k) \times p_{k+1} : x_i \in r \cup \{y\} \wedge pos_{r\cup\{y\}}(p_b) = 1\}| \\
&= |\{r \in R(k+1) : x_i \in r \wedge pos_r(p_b) = 1\}|
\end{aligned}$$

Fall 2: $j \geq k+2$, damit ist $\mathcal{P}_{j-1}([k+1] \setminus \{b\}) = \emptyset$. Analog zu Fall 2 des Induktionsanfangs:

$$\begin{aligned}
&\sum_{M \in \mathcal{P}_{j-1}([k+1]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
&= \sum_{M \in \emptyset} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in \emptyset\setminus M} after_i(c) \right) \\
&= 0
\end{aligned}$$

Für Gleichung (4) gilt wieder

$$|\{r \in R(k+1) : x_i \in r \wedge pos_r(p_b) = j\}| = 0$$

denn alle Repräsentantensysteme $r \in R(1)$ haben genau $k+1$ Elemente, also kann keine Partition in der Ordnung $\prec_r$ eine Position $\geq k+2$ haben.

Fall 3: $2 \leq j \leq k+1$. Betrachte

$$\sum_{M \in \mathcal{P}_{j-1}([k+1]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k+1]\setminus\{b\}\setminus M} after_i(c) \right)$$

Zuerst wird die Summe zerteilt in die Mengen M, die $k+1$ nicht enthalten, und in diejenigen, die $k+1$ enthalten.

$$\begin{aligned}
&= \sum_{M \in \mathcal{P}_{j-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k+1]\setminus\{b\}\setminus M} after_i(c) \right) \\
&\quad + \sum_{M \in \mathcal{P}_{j-2}([k]\setminus\{b\})} \left(\prod_{c \in M\cup\{k+1\}} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right)
\end{aligned}$$

Dann können die Faktoren für $k+1$ herausgezogen werden, so dass die Induktionsvoraussetzung angewandt werden kann.

$$\begin{aligned}
= \quad & after_i(k+1) \cdot \sum_{M \in \mathcal{P}_{j-1}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
& + before_i(k+1) \cdot \sum_{M \in \mathcal{P}_{j-2}([k]\setminus\{b\})} \left(\prod_{c \in M} before_i(c) \cdot \prod_{c \in [k]\setminus\{b\}\setminus M} after_i(c) \right) \\
\overset{IV}{=} \quad & after_i(k+1) \cdot |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}| \\
& + before_i(k+1) \cdot |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j-1\}|
\end{aligned}$$

Nach Einsetzen der Definition wie in Fall 1 ergibt sich

$$\begin{aligned}
&= |\{y \in p_{k+1} : index_X(y) > i\}| \cdot |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j\}| \\
&+ |\{y \in p_{k+1} : index_X(y) < i\}| \cdot |\{r \in R(k) : x_i \in r \wedge pos_r(p_b) = j-1\}| \\
&= |\{(r,y) \in R(k) \times p_{k+1} : index_X(y) > i \wedge x_i \in r \wedge pos_r(p_b) = j\}| \\
&+ |\{(r,y) \in R(k) \times p_{k+1} : index_X(y) < i \wedge x_i \in r \wedge pos_r(p_b) = j-1\}|
\end{aligned}$$

Die beiden Mengen sind offensichtlich disjunkt (kein Element $y \in p_{k+1}$ kann sowohl $index_X(y) > i$ als auch $index_X(y) < i$ erfüllen), also kann man sie vereinigen.

$$\begin{aligned}
= |\{(r,y) \in R(k) \times p_{k+1} : x_i \in r \wedge \\
((index_X(y) > i \wedge pos_r(p_b) = j) \vee (index_X(y) < i \wedge pos_r(p_b) = j-1))\}|
\end{aligned}$$

Es werden nur diejenigen Repräsentantensysteme r betrachtet, bei denen die Partition p_b durch x_i repräsentiert wird. Wegen $b \neq k+1$ muss ein Repräsentant y der Partition p_{k+1} also entweder $index_X(y) > i$ oder $index_X(y) < i$ erfüllen.

Wenn er $index_X(y) > i$ erfüllt, dann ist $y \succ_X x_i$, also muss die Partition von y, p_{k+1}, in $\prec_{r \cup \{y\}}$ nach p_b kommen ($y \succ_{r \cup \{y\}} p_{k+1}$), und $pos_{r \cup \{y\}}(p_b) = pos_r(p_b) = j$, denn p_b wird nicht von Position j "verdrängt."

Wenn dagegen $index_X(y) < i$ erfüllt ist, dann ist $y \prec_X x_i$, also muss p_{k+1} in $\prec_{r \cup \{y\}}$ vor p_b kommen, p_b wird von Position $j-1$ auf Position j "verdrängt," und es gilt $pos_{r \cup \{y\}}(p_b) = pos_r(p_b) + 1 = j$.

In beiden Fällen ist $pos_{r \cup \{y\}}(p_b) = j$, so dass die Behauptung folgt:

$$\begin{aligned}
&= |\{r \in R(k+1) : x_i \in r \wedge \\
&\qquad\qquad ((pos_{r \cup \{y\}}(p_b) = j) \vee (pos_{r \cup \{y\}}(p_b) = j))\}| \\
&= |\{r \in R(k+1) : x_i \in r \wedge pos_{r \cup \{y\}}(p_b) = j)\}|
\end{aligned}$$

□

2.3 Ein Algorithmus mit polynomialer Laufzeit

Um den vollständigen Vektor σ zu berechnen, muss σ_j mittels Gleichung (5) für alle $j \in \{1, \ldots, k\}$ berechnet werden. Dabei ist es jedoch nicht möglich, alle Summanden aufzuzählen, denn wegen

$$\mathcal{P}(A) = \bigcup_{j \in \{0,\ldots,|A|\}} \mathcal{P}_j(A)$$

(für eine beliebige Menge A) werden dabei alle Elemente der Potenzmenge $\mathcal{P}(\{1, \ldots, k\})$ aufgezählt. Da die Potenzmenge $\mathcal{P}(A)$ einer Menge A genau $2^{|A|}$ Elemente enthält, bräuchte ein Algorithmus, der alle Summanden aufzählt, mindestens $O(k \cdot 2^k)$ Rechenzeit um den Vektor σ zu berechnen (der Faktor k kommt daher, dass für jeden Summand k Multiplikationen durchgeführt werden müssen).

Der Vektor σ kann jedoch auch in polynomialer Zeit berechnet werden, wenn dabei, ähnlich wie im Beweis, Partition für Partition abgearbeitet wird:

Algorithmus 2 : Berechnung von σ in Polynomialzeit

Input : Element $x_i \in X$ aus Partition $p_b \in P$
Output : Vektor $\sigma \in \mathbb{N}^k$, der Gleichung (4) erfüllt
$\sigma \leftarrow (1, 0, \ldots, 0)$;
for $c = 1, \ldots, k$ **do**
  if $c \neq b$ **then**
    $\tau_1 \leftarrow after_i(c) \cdot \sigma_1$;
    for $j = 2, \ldots, k$ **do**
      $\tau_j \leftarrow after_i(c) \cdot \sigma_j + before_i(c) \cdot \sigma_{j-1}$;
    end
    $\sigma \leftarrow \tau$;
  end
end

Bei einfacher Betrachtung scheint dieser Algorithmus eine Laufzeit von $O(k^2)$ für die Berechnung des kompletten Vektors σ für ein $x_i \in X$ zu haben, so dass für die Berechnung der Matrix V insgesamt $O(n \cdot k^2)$ Rechenzeit erforderlich wäre. Für diesen Algorithmus selbst wird nur Speicher für σ und τ benötigt, es wäre also $O(k)$ Speicher erforderlich. (Für die Bestimmung des gesamten Rankings wird eventuell weiterer Speicher benötigt; dies wird in Abschnitt 2.4 genauer analysiert.)

Allerdings trifft hier die übliche vereinfachende Annahme, dass Zahlen eine konstante Größe haben, nicht mehr zu: Im worst case (eine Partition mit nur einem Element) können bestimmte Komponenten von σ und τ den Wert $|R|$ annehmen. Um diesen Wert darzustellen, wird bis zu

$$\log |R| = \log \left(\prod_{c=1}^{k} |p_c| \right) = \sum_{c=1}^{k} \log |p_c| \leq k \cdot \log n$$

Speicherplatz benötigt, eine Addition erfordert damit $O(k \log n)$ Rechenzeit, und σ und τ verbrauchen jeweils $O(k^2 \log n)$ Speicher. Da $after_i(c)$ und $before_i(c)$ jeweils mit $O(\log k)$ Stellen auskommen (ihr maximaler Wert ist k), wird für die auftretenden Multiplikationen jeweils $O(k \log n \log k)$ Rechenzeit benötigt. Daraus ergibt sich für die komplette Matrix V eine Laufzeit von $O(n \log n \cdot k^3 \log k)$ bei einem Speicherverbrauch von $O(k^3 \log n)$, welcher durch die Größe der Matrix dominiert wird.

Die Korrektheit wird mit Hilfe einer Invariante für die äußere **for**-Schleife gezeigt: *Für alle $j \in [k]$ gilt stets, dass*

$$\sigma_j = \sum_{M \in \mathcal{P}_{j-1}([c] \setminus \{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c] \setminus \{b\} \setminus M} after_i(d) \right)$$

Wie in Abschnitt 2.2 gezeigt wurde, gilt damit für $c = k$, also wenn die Invariante nach Beendigung der Schleife erfüllt ist, dass

$$\sigma_j = |\{r \in R : x_i \in r \wedge pos_r(p_b) = j\}|$$

d.h. der Algorithmus hat σ_j für alle $j \in [k]$ korrekt berechnet.

Vor der ersten Iteration ($c = 0$) gilt die Invariante: Für $j = 1$ ist $[c] \setminus \{b\} = \emptyset$, also

$$\sum_{M \in \mathcal{P}_{1-1}(\emptyset)} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in \emptyset \setminus M} after_i(d) \right)$$
$$= \prod_{d \in \emptyset} before_i(d) \cdot \prod_{d \in \emptyset} after_i(d) = 1$$

Für $j \geq 2$ ist $\mathcal{P}_{j-1}([c] \setminus \{b\}) = \emptyset$, und damit

$$\sum_{M \in \emptyset} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c] \setminus \{b\} \setminus M} after_i(d) \right) = 0$$

Genau dies sind die initialen Werte für die jeweiligen Komponenten σ_j des Vektors σ.

Behauptung: Jeder Durchlauf der äußeren **for**-Schleife erhält die Invariante.

Beweis: Vor dem aktuellen Schleifendurchlauf war die Invariante erfüllt, also gilt sie für $c - 1$. Für den ersten Durchlauf mit $c = 1$ ist dies der initiale Zustand mit $c = 0$.

Wenn $c = b$, dann wird der **if**-Block nicht ausgeführt, also werden an σ keine Änderungen vorgenommen. Dadurch ist die Invariante nach der Iteration immer noch wahr, denn in diesem Fall ist

$$[c] \setminus \{b\} = [c - 1] = [c - 1] \setminus \{b\}$$

womit die Gültigkeit der Invariante für c direkt aus ihrer Gültigkeit für $c - 1$ folgt.

Für $b \neq c$ muss die Invariante zum Zeitpunkt der Zuweisung $\sigma \leftarrow \tau$ für τ erfüllt sein, d.h. für alle $j \in [k]$ muss gelten

$$\tau_j = \sum_{M \in \mathcal{P}_{j-1}([c] \setminus \{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c] \setminus \{b\} \setminus M} after_i(d) \right)$$

Die zwei Fälle $j = 1$ und $j \geq 2$ müssen dabei wie im Algorithmus separat behandelt werden.

Fall 1: $j = 1$, damit gilt $\mathcal{P}_{j-1}([c-1] \setminus \{b\}) = \{\emptyset\}$.

$$\begin{aligned}
\tau_1 &= after_i(c) \cdot \sigma_1 \\
&= after_i(c) \cdot \sum_{M \in \mathcal{P}_{1-1}([c-1]\setminus\{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c-1]\setminus\{b\}\setminus M} after_i(d) \right) \\
&= after_i(c) \cdot \prod_{d \in [c-1]\setminus\{b\}\setminus M} after_i(d) \\
&= \prod_{d \in [c]\setminus\{b\}\setminus M} after_i(d) \\
&= \sum_{M \in \mathcal{P}_{1-1}([c]\setminus\{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c]\setminus\{b\}\setminus M} after_i(d) \right)
\end{aligned}$$

Fall 2: $j \geq 2$

$$\begin{aligned}
\tau_j = \quad & after_i(c) \cdot \sigma_j + before_i(c) \cdot \sigma_{j-1} \\
= \quad & after_i(c) \cdot \sum_{M \in \mathcal{P}_{j-1}([c-1]\setminus\{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c-1]\setminus\{b\}\setminus M} after_i(d) \right) \\
+ & before_i(c) \cdot \sum_{M \in \mathcal{P}_{j-2}([c-1]\setminus\{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c-1]\setminus\{b\}\setminus M} after_i(d) \right)
\end{aligned}$$

Die Faktoren können nun in die Summen gezogen werden.

$$\begin{aligned}
= \quad & \sum_{M \in \mathcal{P}_{j-1}([c-1]\setminus\{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in ([c-1]\setminus\{b\}\setminus M)\cup\{c\}} after_i(d) \right) \\
+ \quad & \sum_{M \in \mathcal{P}_{j-2}([c-1]\setminus\{b\})} \left(\prod_{d \in M\cup\{c\}} before_i(d) \cdot \prod_{d \in [c-1]\setminus\{b\}\setminus M} after_i(d) \right)
\end{aligned}$$

Damit enthält die erste Summe genau die $M \in \mathcal{P}_{j-1}([c] \setminus \{b\})$, in denen c nicht enthalten ist, und die zweite Summe diejenigen, die c enthalten.

Durch Zusammenziehen der beiden Summen ergibt sich dann die Invariante.

$$
\begin{aligned}
&= \sum_{M \in \mathcal{P}_{j-1}([c]\setminus\{b\}):c \notin M} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c]\setminus\{b\}\setminus M} after_i(d) \right) \\
&+ \sum_{M \in \mathcal{P}_{j-1}([c]\setminus\{b\}):c \in M} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c]\setminus\{b\}\setminus M} after_i(d) \right) \\
&= \sum_{M \in \mathcal{P}_{j-1}([c]\setminus\{b\})} \left(\prod_{d \in M} before_i(d) \cdot \prod_{d \in [c]\setminus\{b\}\setminus M} after_i(d) \right)
\end{aligned}
$$

□

2.4 Weitere Optimierungen für die Implementation

Für die konkrete Implementierung sollten noch einige weitere Optimierungen vorgenommen werden:

1. $after_i(c)$ und $before_i(c)$ werden nicht bei jeder Verwendung neu berechnet (was jeweils $O(n)$ Rechenzeit erfordern würde), sondern mit den anfangs korrekten Werten

   $$
   \begin{aligned}
   after(c) &= |p_c| \\
   before(c) &= 0
   \end{aligned}
   $$

   initialisiert und laufend aktualisiert: Nach jeder Berechnung von σ für ein Element x_i wird dieses Element von $before(c)$ nach $after(c)$ "verschoben," indem $after(c)$ um 1 erhöht und $before(c)$ um 1 erniedrigt wird.

2. Die innerste **for**-Schleife muss nicht alle k Komponenten von τ berechnen, sondern nur die ersten $\min(c+1, k)$ vielen,[10] denn alle weiteren ab τ_{c+2} sind sicher Null.[11] Dies verändert zwar die asymptotische Laufzeit nicht, reduziert die reale Rechenzeit aber dennoch um etwa 50%.

3. σ wird direkt mit dem Gewichtsvektor a multipliziert und das Ergebnis auf $points(p_b)$ aufaddiert, wie in Abschnitt 2.1 gezeigt. Dies vermeidet den Speicherverbrauch für die Matrix V.

[10] genauer: bis $c+1$ für alle $c < b$ und bis c für $c > b$

[11] Beweis per Induktion über c

Damit sieht der Pseudocode wie folgt aus:

Algorithmus 3 : Bestimmung des Rankings für einen Gewichtsvektor

Input : Folge (x_i) von Elementen, Gewichtsvektor $a = (a_1, \ldots, a_k)$
Output : Array *points* mit Einträgen laut Gleichung (2)

```
for c = 1, ..., k do      // Initialisierung der Arrays
    after(c) ← |p_c|;
    before(c) ← 0;
    points(c) ← 0;
end
for i = 1, ..., n do
    b ← index_P(part(x_i));
    σ ← (1, 0, ..., 0);
    for c = 1, ..., k do
        if c ≠ b then
            τ ← (0, ..., 0);
            τ_1 ← after(c) · σ_1;
            for j = 2, ..., min(c + 1, k) do
                τ_j ← after(c) · σ_j + before(c) · σ_{j−1};
            end
            σ ← τ;
        end
    end
    points(b) ← points(b) + σ · a;      // σ · a ist das Skalarprodukt
    after(b) ← after(b) − 1;
    before(b) ← before(b) + 1;
end
```

Durch die Optimierungen benötigt die Berechnung von σ inklusive der Behandlung von $before$ und $after$ pro Durchlauf weiterhin $O(k^3 \log k \log n)$ Zeit; der Speicherverbrauch beträgt $O(k^2 \log n)$, da die Matrix V nicht berechnet wird.

Allerdings muss die Aktualisierung von *points* separat analysiert werden, denn sie benötigt Rechenzeit und Speicherplatz in Abhängigkeit vom Gewichtsvektor a: Das Skalarprodukt $\sigma \cdot a$ besteht aus k Multiplikationen und $k-1$ Additionen. Für die Multiplikationen fallen insgesamt[12]

$$\sum_{j=1}^{k} \log \sigma_j \cdot \log a_j \leq \sum_{j=1}^{k} k \log n \cdot \log a_j \leq k^2 \log n \log a_1$$

Operationen an, wobei jedes Ergebnis bis zu $O(k \log n + \log a_1)$ Stellen hat. Um anschließend alle k Ergebnisse aufzuaddieren, wird folglich $O(k^2 \log n + k \log a_1)$ Zeit und

[12]Unter Verwendung des naiven Multiplikationsalgorithmus. Ein effizienterer Algorithmus wie der Divide-and-Conquer-Algorithmus von Karatsuba [6] kann hier asymptotisch erst einen Vorteil bringen, wenn extrem große Gewichte verwendet werden (mit mehr als $O(k \log k)$ Stellen), ansonsten dominiert die Berechnung von σ die Laufzeit.

$O(k \log n + \log a_1 + \log k) = O(k \log n + \log a_1)$ Speicherplatz benötigt, womit *points* nach dem Aufsummieren der Ergebnisse für alle n Elemente $O(k \log n + \log a_1 + \log n) = O(k \log n + \log a_1)$ Stellen hat.

Solange also $\log a_1 \in O(k \log k)$, ändert sich asymptotisch weder die Laufzeit noch der Speicherverbrauch. Insgesamt ergibt sich als selbst bei exponentiellen Gewichten eine Laufzeit von $O(n \cdot k^3 \log k \log n)$ bei $O(k^2 \log n)$ Speicherverbrauch.

Es gibt noch drei weitere mögliche Optimierungen, die für eine konkrete Implementierung interessant sein können:

- Falls die Partitionen im Individualranking zusammenhängende Cluster bilden, also falls $part(x_i) = part(x_{i+1})$ für ein oder mehrere i, dann muss für jeden dieser Cluster σ nur einmal berechnet werden. Dazu muss lediglich auf $points(b)$ nicht $\sigma \cdot a$, sondern $m \cdot \sigma \cdot a$ aufaddiert werden, wobei m die Anzahl der Elemente im Cluster ist, und $before(c)$ sowie $after(c)$ müssen nicht um 1, sondern um m aktualisiert werden.

  Wie effektiv diese Optimierung ist, hängt natürlich davon ab, wie stark das Individualranking geclustert ist. Das Potential zur Einsparung von Rechenzeit ist allerdings durchaus vorhanden, denn für jeden Cluster mit mehr als einem Element kann eine Iteration der beiden inneren **for**-Schleifen mit einer Rechenzeit von $O(k^3 \log k \log n)$ gespart werden, auf Kosten von $O(n)$ Rechenzeit für die Bestimmung der Cluster, die einmalig anfällt, und $O((k \log n + \log a_1) \cdot k \log n)$ pro Cluster für die Multiplikation mit m.

- Falls Partitionsrankings für u verschiedene Gewichtsvektoren $\alpha_1, \ldots, \alpha_u$ bestimmt werden sollen, können diese mit einer einzigen Iteration über die Elemente der Folge x_i ermittelt werden, indem einfach anstelle des Schritts

  $$points(b) \leftarrow points(b) + \sigma \cdot a$$

  für jedes $w \in [u]$

  $$points_w(b) \leftarrow points_w(b) + \sigma \cdot \alpha_w$$

  berechnet wird. Dies reduziert die Laufzeit in etwa um Faktor u. Da allerdings die Anzahl der zu verwendenden Gewichtsvektoren im Allgemeinen konstant und relativ klein ist, ändert sich die asymptotische Laufzeit nicht.

- Falls einige Gewichte 0 sind, brauchen die zugehörigen Komponenten von σ nicht berechnet werden. Weil die Gewichte mit steigendem j monoton fallend sind, muss es, wenn überhaupt ein Gewicht 0 ist, ein $z \in [k]$ geben, so dass $a_j = 0$ für alle $j \geq z$. Wichtig ist, das ab σ_z keine der Komponenten von σ mehr benötigt wird, so dass σ_z auch nicht für die Berechnung der darauf folgenden Komponenten von σ erforderlich ist.

  Diese Optimierung spart stets etwas Rechenzeit, denn a_k ist nie von 0 verschieden: Falls $a_k \neq 0$, kann stattdessen ein neuer Gewichtsvektor a' verwendet werden:

  $$a' := (a_1 - a_k, \ldots, a_{k-1} - a_k, 0)$$

Dadurch verliert offensichtlich jede Partition genau $a_k \cdot |R|$ Punkte, so dass das Partitionsranking gleich bleibt. Die asymptotische Laufzeit ändert sich aber im allgemeinen Fall nicht.

Falls mehrere Gewichtsvektoren verwendet werden sollen, muss z selbstverständlich so gewählt werden, dass keiner der Vektoren σ_z oder die darauf folgenden Komponenten benötigt.

Der folgende Pseudocode demonstriert diese drei Optimierungen.

Algorithmus 4 : Ranking mit Clustern und mehreren Gewichtsvektoren

Input : Folge (x_i) von Elementen, Gewichtsvektoren $\alpha_1, \ldots, \alpha_u$
Output : Array *points* mit Einträgen laut Gleichung (2)

$num \leftarrow [1, 0, \ldots, 0]$;
$par \leftarrow [index_P(part(x_1)), 0, \ldots, 0]$; `// je` n `Elemente`
$q \leftarrow 1$;
for $i = 2, \ldots, n$ **do**
  if $par(q) \neq index_P(part(x_i))$ **then**
    $q \leftarrow q + 1$;
    $par(q) \leftarrow index_P(part(x_i))$;
  end
  $num(q) \leftarrow num(q) + 1$;
end

for $w = 1, \ldots, u$ **do** $points_w \leftarrow [0, \ldots, 0]$;
Initialisiere $after$ und $before$;
for $i = 1, \ldots, q$ **do**
  $b \leftarrow par(i)$;
  Berechne die ersten z Komponenten von σ;
  $s \leftarrow num(i) \cdot \sigma$;
  for $w = 1, \ldots, u$ **do**
    $points_w(b) \leftarrow points_w(b) + s \cdot \alpha_w$;
  end
  $after(b) \leftarrow after(b) - num(i)$;
  $before(b) \leftarrow before(b) + num(i)$;
end

3 Anwendung des Verfahrens

3.1 Die Testumgebung

Die beiden beschriebenen Verfahren wurden, in verschiedenen Varianten, in Java implementiert, und auf einige reale und konstruierte Rankings angewandt.

Alle Tests wurden auf einem Notebook mit 2.0 GHz Dual-Core Prozessor und Linux-Betriebssystem durchgeführt. Als Java-VM wurde die 32-bit-Version des offiziellen Sun Java 6 JDK verwendet; der Standard-Heap von 728 MB war ausreichend.

Betriebssystem: Ubuntu 9.04

Prozessor laut `/proc/cpuinfo`:

```
Intel(R) Core(TM)2 Duo CPU     T5750  @ 2.00GHz
```

Kernel laut `/proc/version`:

```
Linux version 2.6.28-15-generic (buildd@palmer) (gcc version 4.3.3
(Ubuntu 4.3.3-5ubuntu4) ) #49-Ubuntu SMP Tue Aug 18 18:40:08 UTC 2009
```

Java-VM laut `java -showversion`:

```
java version "1.6.0_16"
Java(TM) SE Runtime Environment (build 1.6.0_16-b01)
Java HotSpot(TM) Server VM (build 14.2-b01, mixed mode)
```

Java-Heap laut `Runtime.getRuntime().maxMemory()`:
763363328 Bytes (728 MB)

3.2 Beispielrankings

Folgende Beispielrankings wurden verwendet:

`kcr010709` Reproduktion der kicker.de-Computertabelle [2]. Stand: 01.07.2009. 742 Fußball-Clubs aus 53 Verbänden, also ein relativ kleines Ranking, aber dennoch etwa $6.89 \cdot 10^{59}$ mögliche Repräsentantensysteme.

`kcr80` Die ersten 80 Einträge von `kcr010709`. 80 Clubs aus 15 Verbänden. 778377600 mögliche Repräsentantensysteme.

`bundesliga2` Kombinierte Tabelle der 1. + 2. Bundesliga, Stand: 6. Spieltag, 18.09.2009 – 20.09.2009. 36 Fußball-Vereine aus 11 Bundesländern. Quelle der einzelnen Tabellen: kicker.de [3] [4]. 12480 mögliche Repräsentantensysteme.

Die zwei Tabellen wurden einfach aneinandergehängt unter der Annahme, dass die Vereine der 2. Liga schlechter seien als die der 1. Liga, insbesondere dass der beste Verein der 2. Liga unmittelbar nach dem schlechtesten Verein der 1. Liga folgt.

`rc4` Synthetisches Ranking: Die ersten 30000 Bytes des Outputs des RC4-Algorithmus [7], wobei die unteren 7 Bits jedes Bytes als die Nummer einer Partition interpretiert wurden, es gibt also 128 Partitionen.

RC4 hat eine sehr ordentliche zufällige Verteilung. Dadurch kann bei diesem Ranking davon ausgegangen werden, dass jede Partition aus etwa $\frac{n}{k} = \frac{30000}{128} = 234.375$

Elementen besteht, womit es ungefähr $|R| \approx 234.375^{128} \approx 2.23 \cdot 10^{303}$ Repräsentantensysteme geben müsste. Dies ist dicht an $1.7976931348623157 \cdot 10^{308}$, der größten mit einer `double`-Variable repräsentierbaren Zahl. (In der Tat gibt es etwa $1.74 \cdot 10^{303}$ Repräsentantensysteme.)

`wedge` Ein weiteres synthetisches Ranking. 1001 Elemente in 501 Partitionen, wobei für die ersten 501 Elemente gilt, dass das Element x_i jeweils in Partition p_i ist, und für die verbleibenden 500 Elemente x_i in p_{1002-i}. Etwa $8.18 \cdot 10^{152}$ mögliche Repräsentantensysteme.

Die Besonderheit dieses Rankings ist, dass auch bei Verwendung des Borda Count alle Partitionen gleich viele Punkte erhalten.

3.3 Methoden zur Bestimmung des Rankings

Die Implementation wurde so gemacht, dass stets eine Folgendarstellung des Partitionsrankings erstellt wird. Die dazu erforderliche willkürliche Ordnung besteht darin, dass die Sortierung stabil ist, so dass gleich gute Partitionen in der Reihenfolge bleiben, in der sie hinzugefügt wurden. Wenn die Partitionen also erst erstellt werden, wenn sie erforderlich sind, d.h. wenn das erste Element einer Partition im Individualranking auftritt, dann ist diese Reihenfolge des Hinzufügens genau die Reihenfolge der besten Elemente der Partitionen. Bei zwei Partitionen mit gleich vielen Punkten ist also diejenige im Partitionsranking besser, deren bestes Element im Individualranking besser ist.

Jede willkürlich festgelegte Ordnung zwischen eigentlich gleich guten Partitionen kann nur eine Notlösung sein. Allerdings ist die Reihenfolge der besten Elemente sicherlich nicht die schlechteste Wahl; insbesondere ist sie wesentlich besser als die offensichtliche Alternative, die alphabetische Ordnung.

Für die Berechnung des eigentlichen Rankings wurden die folgenden Varianten implementiert:

`repr` Implementation von Algorithmus 1 mit den Klassen `RepresentativesMethod` und `PositionalVoting`. Die rekursive Aufzählung aller Repräsentantensysteme ist in `RepresentativesMethod` umgesetzt; diese Klasse kann eine eine beliebige Social Welfare Funktion zur Aggregation verwenden. `PositionalVoting` ist eine solche Social Welfare Funktion; sie implementiert ein Rangwertsystem, spezifiziert durch einen beliebigen Gewichtsvektor.

Da alle Repräsentantensysteme aufgezählt werden, funktioniert diese Variante nur mit Rankings, die "klein genug" sind, wie `bundesliga2` und `kcr80`.

`long` Die Klasse `PositionalMethod_Long` ist eine direkte Implementierung von Algorithmus 3, wobei für *points*, σ und τ der Datentyp `long` verwendet wurde.

Obwohl diese Implementation erheblich schneller ist als `repr`, kann sie nicht auf wesentlich größere[13] Rankings angewandt werden: Da die Komponenten von σ bis zu $|R|$ groß werden können, läuft eine 64-bit-Variable relativ schnell über.

`bigint` Bei der Klasse `PositionalMethod_BigInteger` wurde für *points*, σ und τ der Datentyp `BigInteger` verwendet. Dadurch ist garantiert, dass keinerlei Rundungsfehler auftreten, und es kein Limit[14] für die maximalen Werte gibt.

`double` Die Klasse `PositionalMethod_Double` verwendet Fließkommazahlen, in diesem Fall `double`, als sehr performante Alternative zur Verwendung "unendlich" großer Ganzzahlen. Die berechneten Punktzahlen sind damit allerdings nur Näherungswerte. Schlimmsten Falls könnten dadurch Partitionen im Partitionsranking vertauscht werden, wenn die Fehler aber "klein genug" bleiben, ist das Partitionsranking trotzdem korrekt. `double` hat außerdem einen begrenzten Wertebereich; das `rc4`-Ranking ist sehr dicht an der Grenze des Machbaren.

Der große Vorteil der Verwendung von `double` ist der, dass alle Operationen zwischen `double`-Zahlen konstante Zeit benötigen, unabhängig vom Wert der Zahl. Dadurch wäre tatsächlich die in Abschnitt 2.3 zuerst genannte Laufzeit von $O(n \cdot k^2)$ möglich.

`bigint/c` Die Klasse `PositionalMethod_BigInteger_Clustered` ist eine Variante von `bigint` mit der Clusterung aus Algorithmus 4.

`double/c` Die Klasse `PositionalMethod_Double_Clustered` ist entsprechend die Variante von `double` mit Clusterung.

3.4 Implementierte Gewichtsvektoren

Diese Gewichtsvektoren wurden, mit einer Ausnahme, ausgewählt als einigermaßen repräsentative Beispiele für sinnvolle Rangwertsysteme.

`1:` $(1, \ldots, 1)$
Da bei diesem Gewichtsvektor alle Partitionen gleich viele Punkte erhalten, resultieren Fehler bei den Punktzahlen sofort in einem letztlich zufälligen Partitionsranking. Korrekt wäre die Beibehaltung der Reihenfolge, in der sie im Individualranking erstmals auftreten.

Dieses Rangwertsystem ist für Tests hilfreich, ansonsten aber völlig nutzlos.

`borda:` $(k-1, k-2, \ldots, 1, 0)$
Borda-Count

`majority:` $(1, 0, \ldots, 0)$
Mehrheitswahl

[13] größer nach Anzahl der möglichen Repräsentantensysteme
[14] abgesehen vom Heap-Space der Java-VM

`veto:` $(1, \ldots, 1, 0)$
: Ein einfaches Vetosystem mit Veto für einen einzelnen Kandidaten.

`mv:` $(2, 1, \ldots, 1, 0)$
: Eine Kombination aus Mehrheitswahl und Veto, mit einem explizit präferierten Kandidaten und einem einzelnen Veto.

`m2:` $(2, 1, 0, \ldots, 0)$
: Modifizierte Mehrheitswahl mit zwei verschieden stark präferierten Kandidaten.

3.5 Analyse der Laufzeiten

Tabelle 1 führt die Laufzeiten der verschiedenen Varianten auf den Rankings auf. Jedes Verfahren wurde mehrfach auf jedem Ranking ausgeführt, wobei die Zeit für die Berechnung des Partitionsrankings bestimmt wurde. Als Gewichtsvektor wurde $(1, \ldots, 1)$ verwendet, da die Ergebnisse nicht von Interesse waren, und jegliche Beeinflussung der Laufzeiten durch die Wahl des Gewichtsvektors ausgeschlossen werden sollte.

Da sich die Laufzeiten erheblich unterscheiden, wurden nicht alle Varianten gleich oft ausgeführt. Varianten mit kurzer Laufzeit wurden sehr oft ausgeführt, um zufällige Störungen besser reduzieren zu können, während bei langer Laufzeit aus praktischen Gründen nur einige wenige Durchläufe möglich waren. Um dennoch eine Vergleichbarkeit der Ergebnisse herzustellen, wird für jede Kombination sowohl die Laufzeit, als auch das 95%-Konfidenzintervall angegeben.

Die mit ∞ markierten Kombinationen würden nicht in vernünftiger Zeit beendet; die zu erwartende Laufzeit bewegt sich jenseits von Jahrhunderten. Werte in Klammern bedeuten, dass die angegebene Kombination zwar in vertretbarer Zeit fertig wird, aber kein korrektes Ergebnis liefert, weil der gewählte Datentyp zu klein ist für die Werte, die in σ, τ und *points* auftreten.

	`bundesliga2`	`kcr80`	`kcr010709`	`wedge`	`rc4`
`repr`	$259 \pm 4.0ms$	$13.5 \pm 2.0min$	∞	∞	∞
`long`	$2.62 \pm 0.056ms$	$3.84 \pm 0.12ms$	$(87.0 \pm 2.6ms)$	$(2.54 \pm 0.057s)$	$(5.22 \pm 0.12s)$
`bigint`	$14.3 \pm 0.32ms$	$90.2 \pm 1.8ms$	$668 \pm 30ms$	$25.4 \pm 0.34s$	$176 \pm 15s$
`bigint/c`	$12.8 \pm 0.27ms$	$81.1 \pm 1.7ms$	$653 \pm 30ms$	$29.1 \pm 0.68s$	$166 \pm 14s$
`double`	$1.98 \pm 0.15ms$	$3.82 \pm 0.24ms$	$80.8 \pm 3.5ms$	$1.57 \pm 0.024s$	$3.15 \pm 0.048s$
`double/c`	$1.85 \pm 0.054ms$	$3.68 \pm 0.45ms$	$85.3 \pm 4.0ms$	$1.50 \pm 0.17s$	$3.13 \pm 0.057s$

Tabelle 1: Laufzeiten der verschiedenen Kombinationen

Die Repräsentantenmethode kann also durchaus praktisch verwendet werden, wenn für die Berechnung der Punkte ein geeigneter Algorithmus eingesetzt wird. Beim Vergleich der Laufzeiten zeigen sich folgende Eigenschaften:

- `repr`, also die Berechnung nach Algorithmus 1, ist für praktische Rankings absolut nicht geeignet.

- `bigint` ist erwartungsgemäß wesentlich langsamer als `double`. Java's `BigInteger` ist für diesen Algorithmus nicht besonders gut geeignet, denn es gibt keine direkte Möglichkeit zur Multiplikation eines `BigInteger` mit einem `int`, was abgesehen von der Addition zweier `BigInteger`s die einzige notwendige Operation wäre. In dieser Hinsicht wäre zum Beispiel die Bibliothek GNU MP [1] wesentlich besser geeignet. Es geht bei dem Overhead, der durch dieses Problem verursacht wird, aber ohnehin lediglich um einen konstanten Faktor; die asymptotische Laufzeit ändert sich nicht,[15] und diese ist schlechter als die einer Fließkomma-basierten Implementation.

  Natürlich kommt bei `double` irgendwann der Punkt, ab dem die Werte in *points* usw. zu groß werden, und sich für jede Partition (selbstverständlich inkorrekter Weise) unendlich viele Punkte ergeben. Bei `bigint` gibt es lediglich das ohnehin vorhandene Limit des verfügbaren Arbeitsspeichers.

  Die Problematik der durch Fließkommazahlen auftretenden Ungenauigkeiten wurde separat analysiert.

- Die Clusterung bringt keine nennenswerte Reduktion der Laufzeit, dazu scheint die Clusterung der Beispielrankings nicht ausgeprägt genug zu sein. Tabelle 2 zeigt die Verteilung der Clustergrößen.

Ranking	Elemente	Anzahl Cluster der Größe								
		1	2	3	4	5	6	7	8	9
`bundesliga2`	36	25	4	1	–	–	–	–	–	–
`kcr80`	80	62	7	–	1	–	–	–	–	–
`kcr010709`	742	601	52	8	1	–	–	–	–	1
`wedge`	1001	1001	–	–	–	–	–	–	–	–
`rc4`	30000	29526	234	2	–	–	–	–	–	–

Tabelle 2: Verteilung der Clustergrößen

Interessant sind hier eigentlich nur die beiden "echten" Rankings `bundesliga2` und `kcr010709`.[16] Bei `bundesliga2` befinden sich etwa 31% der Individuen in einem Cluster, bei `kcr010709` sind es ebenfalls fast 18%. Um einen wirklichen Laufzeitgewinn zu erhalten, müsste das Ranking also vermutlich sehr stark in Cluster unterteilt sein.

Allerdings zeigt sich andererseits auch kein erheblicher Overhead durch die Clusterung.

[15] Selbst unter der Annahme, dass $after$ und $before$ stets eine konstante Größe haben, fällt lediglich der Faktor $\log n$ weg, und das natürlich bei GNU MP genauso wie bei Java's `BigInteger`.

[16] `wedge` hat per Konstruktion keinerlei Cluster, `rc4` zeigt exakt die bei Zufallsverteilung zu erwartende Clustergröße von $\frac{30000}{128^{i-1}}$ für einen Cluster der Größe i, und `kcr80` kann als Teilranking von `kcr010709` nicht unabhängig betrachtet werden.

- `double` ist etwas schneller als `long`. Dies ist jedoch zu erwarten bei einem 32-bit-Prozessor, der nur 32-bit Ganzzahlen, aber 64-bit Fließkommazahlen unterstützt.

3.6 Fehler bei Verwendung von Fließkommazahlen

Für diesen Test wurde mit jedem Ranking und jedem der im Folgenden aufgelisteten Gewichtsvektoren sowohl mit `bigint` als auch mit `double` jeweils ein Partitionsranking berechnet. In jedem Partitionsranking wurde dann für alle Partitionen p die relative Differenz zwischen den korrekten Punktzahlen, berechnet mit `bigint`, und den Punktzahlen von `double` bestimmt, also

$$\Delta(p) = \frac{|points_{\texttt{bigint}}(p) - points_{\texttt{double}}(p)|}{points_{\texttt{bigint}}(p)}$$

In Tabelle 3 ist für jede Kombination aus Ranking und Gewichtsvektor jeweils der maximale relative Fehler angegeben, also

$$\max_{p \in P} \Delta(p)$$

Gewichte	`bundesliga2`	`kcr80`	`kcr010709`	`wedge`	`rc4`
$(1, \ldots, 1)$	0	0	$1.45 \cdot 10^{-15}$	$1.57 \cdot 10^{-15}$	$1.60 \cdot 10^{-14}$
Borda-Count	0	0	$1.52 \cdot 10^{-15}$	$1.92 \cdot 10^{-15}$	$2.18 \cdot 10^{-14}$
$(1, 0, \ldots, 0)$	0	0	$3.18 \cdot 10^{-16}$	$5.48 \cdot 10^{-17}$	$1.07 \cdot 10^{-15}$
$(1, \ldots, 1, 0)$	0	0	$1.45 \cdot 10^{-15}$	$1.57 \cdot 10^{-15}$	$1.62 \cdot 10^{-14}$
$(2, 1, \ldots, 1, 0)$	0	0	$1.42 \cdot 10^{-15}$	$1.57 \cdot 10^{-15}$	$1.73 \cdot 10^{-14}$
$(2, 1, 0, \ldots, 0)$	0	0	$4.22 \cdot 10^{-16}$	$9.71 \cdot 10^{-17}$	$1.99 \cdot 10^{-15}$

Tabelle 3: Relativer Fehler der Punktzahlen

`bundesliga2` und `kcr80` sind klein genug, dass überhaupt keine Fehler auftreten. Aber auch bei den anderen Rankings treten nur extrem kleine Fehler auf, im Bereich der letzten Stellen.[17] Insbesondere ist `rc4` bereits dicht vor dem Punkt, ab dem `double` nicht mehr funktioniert, weil die Werte zu groß werden, aber dennoch bleibt der Fehler im Bereich der letzten drei Dezimalstellen.

Interessanter als die Abweichung der Punktzahlen vom korrekten Wert ist allerdings die Frage, wie stark sich dadurch das Partitionsranking ändert. Dazu wurden die Inversionen gezählt, die bei Verwendung von `double` gegenüber dem korrekten Ergebnis von `bigint` auftreten; das Ergebnis ist in Tabelle 4 aufgeführt. Um die Anzahl der Inversionen zwischen den Rankings vergleichbar zu machen, ist weiterhin angegeben, welcher Anteil der überhaupt möglichen Inversionen (d.h. der Anzahl der Inversionen bei exakt umgekehrt sortierter Liste) auftritt.

[17] `double` hat etwa 16 Dezimalstellen.

Gewichte	`bundesliga2`	`kcr80`	`kcr010709`	`wedge`	`rc4`
$(1,\ldots,1)$	–	–	699 (50.7%)	53323 (42.6%)	3905 (48.0%)
Borda-Count	–	–	–	56069 (44.8%)	–
$(1,0,\ldots,0)$	–	–	–	–	–
$(1,\ldots,1,0)$	–	–	555 (40.3%)	55931 (44.7%)	–
$(2,1,\ldots,1,0)$	–	–	160 (11.6%)	53323 (42.6%)	–
$(2,1,0,\ldots,0)$	–	–	–	–	–

Tabelle 4: Inversionen im Partitionsranking

Natürlich treten weder bei `bundesliga2` noch bei `kcr80` irgendwelche Inversionen auf, schließlich sind die Punktzahlen exakt korrekt. Soweit aus nur 3 Rankings überhaupt Schlüsse gezogen werden dürfen, können folgende Beobachtungen festgehalten werden:

- Wenn es überhaupt zu Inversionen kommt, dann gründlich, im deutlich zweistelligen Prozentbereich. Dies deutet auf ein effektiv zufälliges Partitionsranking in diesen Fällen hin.
- Inversionen scheinen vor allem dann aufzutreten, wenn viele Partitionen gleich viele Punkte erhalten, also wenn viele Komponenten des Gewichtsvektors gleich sind, beispielsweise bei $(1,\ldots,1,0)$ und $(2,1,\ldots,1,0)$.

  Ein Blick in die korrekten Ergebnisse für `kcr010709` und $(2,1,\ldots,1,0)$ zeigt zum Beispiel folgendes Bild: Nach den ersten 20 Partitionen mit paarweise verschiedenen Punktzahlen folgen 27 mit exakt gleich vielen Punkten, die letzten 6 sind wieder verschieden – und die Inversionen treten genau bei den 27 Partitionen mit identischen Punktzahlen auf.

`wedge` ist ein spezieller Fall, denn bei diesem Ranking erhalten alle Partitionen gleich viele Punkte, wenn der Borda-Count verwendet wird. Das korrekte Ranking wäre also, die Reihenfolge der Partitionen aus der Eingabedatei beizubehalten:

```
0000 818347651974035467503297424206899788054160511510766197370822 84...
0001 818347651974035467503297424206899788054160511510766197370822 84...
0002 818347651974035467503297424206899788054160511510766197370822 84...
0003 818347651974035467503297424206899788054160511510766197370822 84...
...
0498 818347651974035467503297424206899788054160511510766197370822 84...
0499 818347651974035467503297424206899788054160511510766197370822 84...
0500 818347651974035467503297424206899788054160511510766197370822 84...
```

Mit `double` ergibt sich dagegen folgendes Ranking:

```
0426 8.18347651974037E152
0412 8.183476519740368E152
```

```
0248 8.183476519740367E152
0330 8.183476519740367E152
...
0403 8.183476519740343E152
0411 8.183476519740343E152
0396 8.183476519740339E152
```

Die Ungenauigkeit betrifft nur die letzten beiden Ziffern eines `double`, aber das Partitionsranking ist komplett falsch. Dies ist ein durchaus repräsentatives Beispiel dafür, wie ein Partitionsranking sich bei Verwendung von `double` verfälschen kann.

Zusammenfassend kann also gesagt werden, dass Fließkommazahlen als Ersatz für die "unendlich" großen Ganzzahlen hier ungeeignet sind. Insbesondere die Tatsache, dass bei Ungenauigkeiten bereits extrem viele der überhaupt möglichen Inversionen auftreten, ist für ein zuverlässigen Ranking sehr problematisch.

4 Ein axiomatischer Ansatz für Konsistenz

Bei der einleitenden Definition des Rankingproblems für Partitionen in Abschnitt 1.1 wurde gefordert, dass das Partitionsranking *konsistent* mit dem Ranking der Individuen sein soll. Was Konsistenz in diesem Fall aber bedeutet, ist nicht unmittelbar offensichtlich. Daher soll hier versucht werden, die Konsistenz eines Partitionsrankings mit einem Individualranking axiomatisch zu charakterisieren, d.h. Axiome zu beschreiben, die eine Methode zur Lösung des Rankingproblems für Partitionsrankings (*Partitionsrankingsmethode*) $\mathcal{M}$ aufweisen sollte.

4.1 Monotonie

Dieses Axiom beschäftigt sich mit der Vertauschung zweier benachbarter Elemente x_λ, $x_{\lambda+1}$ im Individualranking (x_i). Dazu sei (x'_i) das Ranking nach der Vertauschung, also $x'_i = x_i$ für alle $i \in [n] \setminus \{\lambda, \lambda+1\}$, sowie $x'_\lambda = x_{\lambda+1}$ und $x'_{\lambda+1} = x_\lambda$. Die Partition $part(x'_\lambda)$ des Elements x'_λ wird als *Sieger* der Vertauschung bezeichnet, $part(x'_{\lambda+1})$ entsprechend als *Verlierer*. $\prec'_P$ sei das Ranking der Partitionen nach der Vertauschung, d.h. eine Ordnung auf P, die mittels $\mathcal{M}$ aus (x'_i) ermittelt wurde.

Die Monotonie fordert nun, dass der Siegers durch die Vertauschung nicht schlechter werden darf, weder gegenüber dem Verlierer, noch gegenüber der Partition $part(x'_b)$ eines beliebigen anderen Elements x'_b.

Definition: Die Partitionsrankingsmethode $\mathcal{M}$ erfüllt die Bedingung der *Monotonie*, wenn für alle $b \in [n] \setminus \{\lambda\}$ gilt:

(i) $part(x'_\lambda) \prec_P part(x'_b) \Longrightarrow part(x'_\lambda) \prec'_P part(x'_b)$

(ii) $part(x'_\lambda) \not\succ_P part(x'_b) \Longrightarrow part(x'_\lambda) \not\succ'_P part(x'_b)$

Behauptung: Die Kombination aus der Repräsentantenmethode und einem beliebigen Rangwertsystem erfüllt stets die Monotonie.

Beweis: Sei $pos'_r(p)$, entsprechend zu $pos_r(p)$, die Position der Partition p im Repräsentantensystem r laut dem Individualranking (x'_i), und $points'$ entsprechend zu $points$ das Array der Punktzahlen, ebenfalls berechnet aus dem Individualranking (x'_i).

Es genügt zu zeigen, dass $part(x'_\lambda)$ keine Punkte verliert, und $part(x'_b)$ keine dazugewinnt, also dass

(a) $points'(index_P(part(x'_\lambda))) \geq points(index_P(part(x'_\lambda)))$

(b) $points'(index_P(part(x'_b))) \leq points(index_P(part(x'_b)))$

Dann ergeben sich die Bedingungen (i) und (ii) direkt aus Gleichung (3).

Betrachte dazu ein beliebiges Repräsentantensystem r.

Fall 1: $x'_\lambda, x'_{\lambda+1} \notin r$. Damit ist

$$part(x_\beta) \prec'_r part(x_\gamma) \iff part(x_\beta) \prec_r part(x_\gamma)$$

für alle $x_\beta, x_\gamma \in r$, denn die Reihenfolge der Partitionen in r ist definiert über die Reihenfolge ihrer Repräsentanten im Ausgangsranking. In diesem hat sich aber lediglich die Reihenfolge von x'_λ und $x'_{\lambda+1}$ untereinander verändert; insbesondere hat sich die Reihenfolge von x_β und x_γ nicht verändert.

Also ist auch

$$pos'_r(part(x_\beta)) = pos_r(part(x_\beta))$$

für alle $x_\beta \in r$, womit x_β nach der Vertauschung gleich viele Punkte erhält wie davor.

Fall 2: $x'_\lambda \notin r$ oder $x'_{\lambda+1} \notin r$ (d.h. es sind nicht beide in r enthalten). Auch hier gilt für alle $x_\beta, x_\gamma \in r$, dass $part(x_\beta) \prec'_r part(x_\gamma) \iff part(x_\beta) \prec_r part(x_\gamma)$, insbesondere auch für $x_\beta, x_\gamma \in \{x'_\lambda, x'_{\lambda+1}\}$, denn während sich die Reihenfolge von x'_λ und $x'_{\lambda+1}$ untereinander verändert hat, muss die Reihenfolge von x'_λ (bzw. $x'_{\lambda+1}$) und jedem beliebigen anderen Element $x_\beta \in X \setminus \{x'_\lambda, x'_{\lambda+1}\}$ gleich geblieben sein (ansonsten wäre es keine einfache Vertauschung der beiden benachbarten Elemente $x'_\lambda, x'_{\lambda+1}$).

Also gilt hier ebenfalls $pos'_r(part(x_\beta)) = pos_r(part(x_\beta))$ für alle $x_\beta \in r$, und alle Elemente aus r erhalten gleich viele Punkte wie vor der Vertauschung.

Fall 3: $x'_\lambda, x'_{\lambda+1} \in r$. Da x'_λ und $x'_{\lambda+1}$ im Ausgangsranking benachbart waren, sind sie dies auch in $\prec_r$ und $\prec'_r$.

Sei $\pi := pos_r(part(x'_\lambda))$; damit ist $pos'_r(part(x'_\lambda)) = \pi - 1$. Laut Definition des Vektors a gilt, dass $a_{\pi-1} \geq a_\pi$, also kann $part(x'_\lambda)$ durch die Vertauschung nicht weniger Punkte erhalten, nur gleich viele oder mehr.

Sei nun $\rho := pos_r(part(x'_{\lambda+1}))$. Damit ist $pos'_r(part(x'_{\lambda+1})) = \rho + 1$, laut Definition von a gilt, dass $a_{\rho+1} \leq a_\rho$, und $part(x'_{\lambda+1})$ kann nicht mehr Punkte erhalten als vor der Vertauschung.

Für jedes andere Element $x'_\beta \in X \setminus \{x'_\lambda, x'_{\lambda+1}\}$ gilt, weil x'_λ und $x'_{\lambda+1}$ sowohl vor als auch nach der Vertauschung benachbart sind, entweder

$$x'_\beta \prec_r x'_\lambda \ \wedge \ x'_\beta \prec_r x_{\lambda+1}$$

oder entsprechend mit $\succ_r$. Da sich aber die Reihenfolge von x'_β und x'_λ (bzw. $x'_{\lambda+1}$) nicht verändert hat, gilt auch nach der Vertauschung $x'_\beta \prec'_r x'_\lambda \wedge x'_\beta \prec'_r x_{\lambda+1}$ (bzw. entsprechend mit $\succ_r$).

Also ist auch $pos'_r(part(x'_\beta)) = pos_r(part(x'_\beta))$, und die Punktzahl von $part(x'_\beta)$ kann sich nicht verändern. □

4.2 Nichtbeeinflussung Unbeteiligter

Unter den gleichen Voraussetzungen wie bei der Monotonie (Vertauschung zweier benachbarter Elemente $x'_\lambda, x'_{\lambda+1}$) fordert die *Nichtbeeinflussung Unbeteiligter*, dass sich die Reihenfolge zweier beliebiger, nicht an der Vertauschung beteiligter Partitionen durch die Vertauschung nicht verändern darf.

Definition: Die Partitionsrankingsmethode $\mathcal{M}$ erfüllt die *Nichtbeeinflussung Unbeteiligter*, wenn für alle Partitionen $p, q \in P \setminus \{part(\lambda), part(\lambda+1)\}$ gilt:

(i) $p \prec_P q \Longrightarrow p \prec'_P q$

(ii) $p \not\prec_P q \Longrightarrow p \not\prec'_P q$

Behauptung: Die Kombination aus der Repräsentantenmethode und einem beliebigen Rangwertsystem erfüllt stets die Nichtbeeinflussung Unbeteiligter.

Beweis: Seien pos'_r und $points'$ wie beim Beweis der Monotonie. Es genügt zu zeigen, dass sich die Punktzahl von p durch die Vertauschung nicht verändert, also dass

$$points'(index_P(p)) = points(index_P(p))$$

Dadurch folgt automatisch, dass sich auch die Punktzahl von q nicht verändert, und die Bedingungen (i) und (ii) ergeben sich wie bei der Monotonie direkt aus Gleichung (3).

Betrachte ein beliebiges Repräsentantensystem r. Wenn weder x'_λ noch $x'_{\lambda+1}$ in r enthalten ist, oder wenn nur eines der beiden Elemente enthalten ist, dann verändert sich die Reihenfolge der Elemente durch die Vertauschung nicht, also gilt $pos'_r(p) = pos_r(p)$. Wenn beide enthalten sind, dann ändert sich die Reihenfolge von x'_λ und $x'_{\lambda+1}$ in r. Allerdings gilt $pos'_r(x'_{\lambda+1}) = pos_r(x'_\lambda)$ und umgekehrt ($pos'_r(x'_\lambda) = pos_r(x'_{\lambda+1})$), also verschiebt sich der Repräsentant der unbeteiligten Partition p nicht, und es gilt auch hier, dass $pos'_r(p) = pos_r(p)$.

In beiden Fällen gilt damit $points'(index_P(p)) = points(index_P(p))$. □

4.3 Einstimmigkeit

Dieses Axiom bezieht sich nicht auf Veränderungen im Individualranking und deren Auswirkungen im Partitionsranking, sondern stellt Anforderungen an eine Eigenschaft des Individualranking, die ins Partitionsranking übergehen muss: Wenn eine Partition p im Individualranking vollständig vor einer Partition q liegt, muss sie dies auch im Partitionsranking.

Definition: $\mathcal{M}$ erfüllt die *Einstimmigkeit*, wenn für alle Partitionen $p, q \in P$ gilt:

$$(\forall x \in p, y \in q : x \prec_X y) \Longrightarrow p \prec_P q$$

Behauptung: Eine Kombination aus der Repräsentantenmethode und einem Rangwertsystem erfüllt die Einstimmigkeit genau dann, wenn für den Gewichtsvektor des Rangwertsystems gilt

$$a_1 > a_2 > \cdots > a_k$$

Beweis: Es genügt wieder zu zeigen, dass genau in diesem Fall $points(p) > points(q)$ gilt, denn damit folgt die Behauptung direkt aus Gleichung (3).

Betrachte den Fall $a_1 > a_2 > \cdots > a_k$. Für alle $x \in p, y \in q$ gilt $x \prec_X y$, also ist $pos_r(p) < pos_r(q)$ für alle $r \in R$. Somit ist auch $a_{pos_r(p)} > a_{pos_r(q)}$ für alle $r \in R$, woraus folgt:

$$points(p) = \sum_{r \in R} a_{pos_r(p)} > \sum_{r \in R} a_{pos_r(q)} = points(q)$$

Betrachte nun den Fall, in dem für ein $\beta \in [k-1]$ gilt $a_\beta = a_{\beta+1}$, in Verbindung mit dem trivialen Individualranking $(t_i)_{i=1,\ldots,n}$ mit $part(t_i) = \{t_i\}$. Wähle $p := part(t_\beta)$ und $q := part(t_{\beta+1})$. Offensichtlich gibt es genau ein einziges Repräsentantensystem $r = \{t_1, t_2, \ldots, t_n\}$, also ist

$$points(p) = a_{pos_r(p)} = a_\beta = a_{\beta+1} = a_{pos_r(q)} = points(q)$$

und die Partitionsrankingsmethode erfüllt die Einstimmigkeit nicht. □

5 Unabhängigkeit der Axiome

Um die Unabhängigkeit der Axiome zu zeigen, werden im Folgenden drei Partitionsrankingsmethoden vorgestellt, die je zwei der Axiome erfüllen, nicht aber das jeweils dritte.

5.1 Einstimmigkeit

Wie oben gezeigt wurde, erfüllt die Rapräsentantenmethode in Kombination mit jedem beliebigen Rangwertsystem sowohl Monotonie als auch Nichtbeeinflussung Unbeteiligter. Es muss also lediglich ein beliebiges Rangwertsystem gewählt werden, bei dem $a_\beta = a_{\beta+1}$ für ein $\beta \in [k-1]$, zum Beispiel das triviale Rangwertsystem mit $a = (0, \ldots, 0)$. Dieses erfüllt die Einstimmigkeit offensichtlich nicht, da $a_1 \not\succ a_2$.

5.2 Nichtbeeinflussung Unbeteiligter

Das Partitionsranking $\prec'_P$ sei definiert durch die Einstimmigkeit, also für beliebige Partitionen $p, q \in P$

$$p \prec'_P q :\Longleftrightarrow \forall x \in p, y \in q : x \prec_X y$$

$\prec'_P$ erfüllt auch die Monotonie: Betrachte eine beliebige Vertauschung mit Sieger $p := part(x'_\lambda)$ und Verlierer $q := part(x'_{\lambda+1})$ sowie die Partition $\beta := part(x'_b)$ eines beliebigen Elements x'_b. ((x'_i) sei wieder das Individualranking nach und (x_i) das vor der Vertauschung.)

Wenn (vor der Vertauschung) $p \prec_P \beta$ galt, muss p vollständig vor β gelegen haben, also ist $\beta \neq q$, denn mindestens das Element $x'_{\lambda+1} = x_\lambda \in q$ war vor $x'_\lambda = x_{\lambda+1} \in p$, so dass p nicht vollständig vor q gelegen haben kann. Wenn aber $\beta \neq q$, dann kann sich die Reihenfolge von p und β nicht verändert haben und β liegt weiterhin vollständig hinter p, und es gilt immer noch $p \prec'_P \beta$.

Wenn $p \not\succ_P \beta$ galt, kann β nicht vollständig vor p gelegen haben. Wenn $\beta = q$, dann ist auch $p \not\succ'_P \beta$, denn nach der Vertauschung liegt mindestens $x'_\lambda = x_{\lambda+1} \in p$ vor $x'_{\lambda+1} = x_\lambda \in q$. Wenn $\beta \neq q$, dann hat sich die Reihenfolge von β und p nicht verändert, also gilt weiterhin $p \prec'_P \beta$.

Sei weiter $\prec''_P$ das Partitionsranking, das die Partitionen nach ihrem besten Element im Individualranking sortiert. Dazu sei $pos_X(x_\beta)$ die Position des Elements x_β im Individualranking (x_i) (offensichtlich ist $pos_X(x_\beta) = index_X(x_\beta)$) und (für eine Partition $p \in P$)

$$best(p) := \min_{x_\beta \in p} pos_X(x_\beta)$$

Damit kann $\prec''_P$ definiert werden als

$$p \prec''_P q :\Longleftrightarrow best(p) < best(q)$$

Offensichtlich erfüllt $\prec''_P$ die Einstimmigkeit, denn wenn jedes Element von p vor allen Elementen von q liegt, dann liegt insbesondere auch das beste Element von p vor dem besten Element aus q.

Die Monotonie ist ebenfalls erfüllt: Wenn nicht die besten Elemente zweier Partitionen vertauscht werden, kann sich das Partitionsranking ohnehin nicht verändern; wenn die besten Elemente vertauscht werden, dann ist nach der Vertauschung der Sieger echt besser als der Verlierer, womit keine der beiden Bedingungen der Monotonie verletzt sein kann.

Aus diesen beiden Partitionsrankingsmethoden wird nun $\prec_P$ wie folgt aufgebaut: Wenn das beste Element des Individualrankings eine einelementige Partition bildet, dann wird die Einstimmigkeit benutzt, ansonsten das beste Element. $\prec_P$ wird also definiert als:

$$p \prec_P q :\Longleftrightarrow \begin{cases} p \prec'_P q & \text{falls } part(x_1) = \{x_1\} \\ p \prec''_P q & \text{sonst} \end{cases}$$

Diese Methode erfüllt weiterhin sowohl die Einstimmigkeit, als auch die Monotonie. Die Einstimmigkeit ist trivial immer erfüllt, denn beide Verfahren erfüllen sie.

Die Monotonie folgt sofort aus der Monotonie von $\prec'_P$ und $\prec''_P$, außer wenn das einzige Element einer einelementigen Partition an die erste Position im Individualranking wechselt, oder wenn es von dieser weg vertauscht wird.

Betrachte genau diesen Fall, also die Vertauschung mit $x_1 = x'_2$ und $x_2 = x'_1$, wobei entweder $part(x_1) = \{x_1\}$ oder $part(x_2) = \{x_2\}$. Es sei $p := part(x'_1)$ der Sieger und $q := part(x'_2)$ der Verlierer.

Fall 1: $p = \{x_2\} = \{x'_1\}$ und $p = \{x_1\} = \{x'_2\}$. Damit wird sowohl vor als auch nach der Vertauschung $\prec'_P$ verwendet, und die Monotonie folgt sofort aus der Monotonie von $p \prec'_P q$.

Fall 2: $p = \{x_2\} = \{x'_1\}$ und $|q| \geq 2$. (Damit wurde vor der Vertauschung $\prec''_P$ verwendet, und danach wird $\prec'_P$ benutzt.) Wegen der Einstimmigkeit gilt (nach der Vertauschung) $p \prec'_P \beta$ für alle $\beta \in P$, womit wieder keine der beiden Bedingungen der Monotonie verletzt sein kann (weil die Nachbedingung immer wahr ist).

Fall 3: $q = \{x_1\} = \{x'_2\}$ und $|p| \geq 2$. (Damit wurde vor der Vertauschung die Einstimmigkeit verwendet, und danach des beste Element.) Wegen der Einstimmigkeit galt vor der Vertauschung $q \prec''_P \beta$ für alle $\beta \in P$, also insbesondere $p \succ''_P q$, woraus folgt, dass weder $p \prec''_P q$ noch $p \not\prec''_P q$ erfüllt war, d.h. für $\beta = p$ ist die die Vorbedingung beider Bedingungen der Monotonie immer falsch.

Wenn andererseits $\beta \neq p$, dann gilt nach der Vertauschung $q \prec'_P \beta$, denn vor Position 2 ist nur noch eines der Elemente von p. Also kann keine der beiden Bedingungen der Monotonie verletzt sein (weil entweder die Vorbedingung falsch, oder die Nachbedingung wahr ist).

$\prec_P$ erfüllt also sowohl Einstimmigkeit als auch Monotonie, aber offensichtlich nicht die Nichtbeeinflussung Unbeteiligter. Betrachte dazu die folgenden beiden Individualrankings (x_i) und (x'_i) mit ihren zugehörigen Partitionen. Offensichtlich entsteht (x'_i) aus (x_i) durch die Vertauschung von $x_1 = x'_2$ und $x_2 = x'_1$.

i	x_i	$part(x_i)$	$index_P(part(x_i))$	x'_i	$part(x'_i)$	$index_P(part(x'_i))$
1	1	{1,3}	1	2	{2}	2
2	2	{2}	2	1	{1,3}	1
3	3	{1,3}	1	3	{1,3}	1
4	4	{4,6}	3	4	{4,6}	3
5	5	{5}	4	5	{5}	4
6	6	{4,6}	3	6	{4,6}	3

Bei (x_i) wird das beste Element benutzt, somit lautet das Partitionsranking für (x_i):

$$p_1 \prec_P p_2 \prec_P p_3 \prec_P p4$$

Insbesondere ist $p_3 \prec_P p4$.

Bei (x'_i) wird dagegen die Einstimmigkeit benutzt, so dass einige Elemente nicht mehr vergleichbar sind. Es gibt folgende Reihenfolgen im Partitionsranking:

$$p_2 \prec_P p_1 \prec_P p_3 \quad \text{und} \quad p_2 \prec_P p_1 \prec_P p_4$$

Insbesondere gilt aber weder $p_4 \prec_P p_3$ noch $p_3 \prec_P p_4$, wobei letzteres die Nichtbeeinflussung Unbeteiligter verletzt.

5.3 Monotonie

Sei $\prec'_P$ wieder durch die Einstimmigkeit definiert.

$\prec_P$ "erkennt," wenn eine der folgenden zwei Situationen für ein $\beta \in [n]$ gilt:

(a) $part(x_\beta) = part(x_{\beta+1}) = part(x_{\beta+2}) = part(x_{\beta+\mathbf{4}}) = part(x_{\beta+5})$
und $part(x_{\beta+\mathbf{3}}) = part(x_{\beta+6}) = part(x_{\beta+7}) = part(x_{\beta+8})$

(b) $part(x_\beta) = part(x_{\beta+1}) = part(x_{\beta+2}) = part(x_{\beta+\mathbf{3}}) = part(x_{\beta+5})$
und $part(x_{\beta+\mathbf{4}}) = part(x_{\beta+6}) = part(x_{\beta+7}) = part(x_{\beta+8})$

Offensichtlich kann für zwei Partitionen p und q maximal eine Bedingung aus (a), (b) und Einstimmigkeit gelten; jede einzelne schließt die jeweils anderen beiden aus.

$\prec_P$ verhält sich nun wie folgt:

$$p \prec_P q :\Longleftrightarrow \begin{cases} \text{wahr} & \text{falls } \exists \beta \in [n] : x_\beta \in p \wedge \text{(a)} \\ \text{wahr} & \text{falls } \exists \gamma \in [n] : x_\gamma \in p \wedge \text{(b) mit } \beta = \gamma - 8 \\ p \prec'_P q & \text{sonst} \end{cases}$$

D.h. wenn keine der beiden Bedingungen erfüllt ist, und die Reihenfolge von p und q nicht durch die Einstimmigkeit erzwungen ist, besteht keine Reihenfolge zwischen den beiden Partitionen.

Offensichtlich stören die beiden Sonderbedingungen weder bei der Einstimmigkeit (da beide Bedingungen die Einstimmigkeit ausschließen) noch bei der Nichtbeeinflussung Unbeteiligter (denn die Sonderbedingungen betreffen beim Vertauschen zweier Element lediglich die Partitionen der zwei vertauschten Elemente). Allerdings ist die Monotonie nicht erfüllt, betrachte dazu die folgenden beiden Individualrankings (x_i) und (x'_i) mit ihren zugehörigen Partitionen.

i	x_i	$index_P(part(x_i))$	x'_i	$index_P(part(x'_i))$
1	1	1	1	1
2	2	1	2	1
3	3	1	3	1
4	4	1	5	2
5	5	2	4	1
6	6	1	6	1
7	7	2	7	2
8	8	2	8	2
9	9	2	9	2

Offensichtlich entsteht (x'_i) aus (x_i) durch die Vertauschung von $x_4 = x'_5$ und $x_5 = x'_4$; p_2 ist Sieger der Vertauschung, p_1 Verlierer.

Bei (x_i) trifft (b) zu, also ist das Partitionsranking

$$p_2 \prec_P p_1$$

Bei (x'_i) trifft entsprechend (a) zu, also ist

$$p_1 \prec_P p_2$$

Der Sieger p_2 war also vor der Vertauschung besser als der Verlierer p_1; dies kehrt sich mit der Vertauschung um, was die Monotonie verletzt.

6 Weitere mögliche Forderungen

Bei den folgenden Eigenschaften handelt es sich um einige ausgesuchte Forderungen an eine Partitionsrankingsmethode, die zwar nicht als Axiom geeignet sind, aber dennoch nicht einfach ignoriert werden sollten.

6.1 Monotonie des Verlierers

Analog zur Forderung der Monotonie, dass der Sieger nicht schlechter werden darf, kann für den Verlierer gefordert werden, dass er nicht besser werden darf.

Behauptung: Die Monotonie des Verlierers folgt aus der Monotonie.

Beweis: Sei p der Sieger und q der Verlierer der Vertauschung ν mit $x'_\lambda = x_{\lambda+1}$ und $x'_{\lambda+1} = x_\lambda$, die aus (x_i) das Individualranking (x'_i) erstellt. Bei der umgekehrten Vertauschung $\bar{\nu}$ (welche die Vertauschung ν rückgängig macht) ist also q der Sieger und p der Verlierer.

Angenommen, die Vertauschung ν wird auf (x_i) angewandt, und der Verlierer q wird dadurch besser als eine beliebige andere Partition $\beta \in P \setminus \{q\}$, d.h. $q \prec'_P \beta$, obwohl $q \not\prec_P \beta$. Wenn nun die Vertauschung $\bar{\nu}$ auf (x'_i) angewandt wird, darf laut Monotonie der Sieger q nicht schlechter werden, aus $q \prec'_P \beta$ muss folgen $q \prec_P \beta$. Also kann q durch die Vertauschung ν nicht besser geworden sein. □

6.2 Nichtbeeinflussung Übersprungener

Diese Eigenschaft betrachtet den Fall, dass sich die Position eines Elements im Individualranking um mehrere Plätze verbessert, dass es quasi nach vorne springt, wobei mehrere Elemente übersprungen werden. Für diesen Fall könnte nun gefordert werden, dass sich für zwei beliebige übersprungene Elemente x_λ, x_μ die Reihenfolge ihrer Partitionen $\beta := part(x_\lambda)$, $b := part(x_\mu)$ nicht verändern darf.

Allerdings ist diese auf den ersten Blick logisch erscheinende Forderung nicht unbedingt sinnvoll, da es sich im Extremfall bei x_λ um das einzige Element seiner Partition β handeln

könnte, während mit x_μ zum Beispiel nur eines von 100 Elementen der Partition b betroffen ist. Wenn in diesem Extremfall weiterhin β besser ist als b, dann wäre es durchaus nicht unlogisch, dass sich die Reihenfolge der Partitionen ändern sollte – immerhin hat β anteilig sehr viel mehr verloren als b.

6.3 Objektivität

Die Objektivität fordert, dass sich bei der Vertauschung zweier im Ranking der Individuen benachbarter Elemente der gleichen Partition β das Partitionsranking nicht verändern darf.

Behauptung: Jede Partitionsrankingsmethode, welche die Monotonie und die Nichtbeeinflussung Unbeteiligter erfüllt, erfüllt auch die Objektivität.

Beweis: Die Partition β ist sowohl Sieger als auch Verlierer der Vertauschung. Damit darf sie sich gegenüber einer anderen Partition $b \in P \setminus \{\beta\}$ einerseits nicht verschlechtern, und andererseits (wegen der abgeleiteten Monotonie des Verlierers) nicht verbessern. Also kann sich die Reihenfolge zwischen β und b nicht verändern.

Betrachte zwei beliebige Partitionen $p, q \in P \setminus \{\beta\}$. Da weder p noch q an der Vertauschung beteiligt ist, darf sich die Reihenfolge zwischen p und q wegen der Nichtbeeinflussung Unbeteiligter ebenfalls nicht verändern. □

Literatur

[1] GNU Multiple Precision Arithmetic Library. http://gmplib.org/, abgerufen 28. September 2009.

[2] kicker.de Computertabelle. http://www.kicker.de/fussball/intligen/startseite/artikel/350078/.

[3] Tabelle 1. Bundesliga. http://www.kicker.de/news/fussball/bundesliga/spieltag/1-bundesliga/2009-10/6/0/spieltag.html, abgerufen 28. September 2009.

[4] Tabelle 2. Bundesliga. http://www.kicker.de/news/fussball/2bundesliga/spieltag/2-bundesliga/2009-10/6/0/spieltag.html, abgerufen 28. September 2009.

[5] Jean-Charles de Borda. *M*émoire sur les élections au scrutin. Histoire de l'Academie Royale des Sciences, 1781.

[6] A. Karatsuba and Yu Ofman. Multiplication of many-digital numbers by automatic computers. *D*oklady Akad. Nauk SSSR, 145:293–294, 1962. Zitiert in Wolfram MathWorld, http://mathworld.wolfram.com/KaratsubaMultiplication.html.

[7] Bruce Schneier. *A*pplied Cryptography, Second Edition. John Wiley & Sons, 1996.

[8] H. P. Young. Social choice scoring functions. *S*IAM Journal on Applied Mathematics, 28:824–838, 1975. Zitiert in Encyclopaedia of Mathematics, http://eom.springer.de/s/s120170.htm.