Universität des Saarlandes
Max-Planck-Institut für Informatik
AG5

Boolean Tensor Decomposition based on the Walk'n'Merge Algorithm

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Helge Dombrowski

angefertigt unter der Leitung von / supervised by

Dr. Pauli Miettinen

begutachtet von / reviewer

Prof. Dr. Gerhard Weikum

Juni 2016

Hilfsmittelerklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Non-plagiarism Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, den 03. Juli 2012,

(Helge Dombrowski)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlichtwird.

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department at Saarland University.

Saarbrücken, den 03. Juli 2012,

(Helge Dombrowski)

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement in Lieu of an Oath

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken,………………………………..	…………………………………………….
(Datum / Date)	(Unterschrift / Signature)

To whoever it may concern.

- Helge

Abstract

Tensor decomposition is a long standing data mining technique with numerous applications in different fields. It has been used in image processing, signal processing, computer vision, social network analysis and many more. It's a useful tool for understanding high dimensional data, like classic matrix factorization (e.g. SVD) for two dimensional data. This work focuses on the decomposition of 3-way Boolean tensors, and explores new algorithms based on the WALK'N'MERGE algorithm by Miettinen and Erdős.

The main focus is on the by the original paper introduced FIBERGRAPH, which is a data structure of the fibre structure of a tensor. The original Paper explored the FIBERGRAPH using random walks. In this work, it is explored using several different clustering, walking and partitioning strategies. These were evaluated on a new implementation WALK'N'MERGE++ in C++, in contrast to the original WALK'N'MERGE, which was implemented in Python. Finally, I propose the SPLIT'N'EXPAND algorithm, which is a highly parallelizable algorithm, based on a divide and conquer strategy on the FIBERGRAPH.

Contents

Chapter 1

Introduction

1.1 Motivation

Tensor decomposition is a long standing data mining technique with numerous applications in different fields. As matrix factorization (e.g. SVD) is for two dimensional data, it's a useful tool for understanding high dimensional data. It has been used in image processing, signal processing, computer vision, social network analysis and many more [KB09]. In general, it is useful for all kinds of data, that can be interpreted as a ternary (or higher dimensional) relation. For a programmer, the best way to understand a m_1 -by- m_2 -by- $\ldots$ -by- m_n tensor, is as a n-dimensional array with dimension sizes of m_1 to m_n. For the mathematician its an element of $\mathbb{R}^{m_1 \times m_2 \times \cdots \times m_n}$ with $m_i \in \mathbb{N}$ for all i.

Tensor decomposition as a concept was first introduced in 1927 by Hitchcock [Hit27], but was rediscovered on several later occasions, e.g. by Carroll et al. in 1970 [CC70]. While there exist many different variants (e.g. Tucker, CP), the most fundamental decomposition is the tensor rank decomposition (or CPD). For a n_1 -by- n_2 -by- $\ldots$ -by- n_k tensor $\boldsymbol{\mathcal{X}}$, the CPD is defined as:

$$\boldsymbol{\mathcal{X}} = \sum_{i=1}^{r} \vec{\boldsymbol{a}}_{1,i} \boxtimes \vec{\boldsymbol{a}}_{2,i} \boxtimes \ldots \boxtimes \vec{\boldsymbol{a}}_{k,i}$$

Where $\vec{\boldsymbol{a}}_{j,i} \in \mathbb{R}^{n_i}$ and $\boxtimes$ is the outer product. Finding such a decomposition is a NP-hard problem [HL13].

Boolean tensors are a tensors subset of tensors where the entries are Boolean values. An example for a Boolean tensor can be seen in semantic knowledge bases like YAGO ([SKW07]). These can be directly transformed into equivalent Boolean tensors. Typically, the entries of a knowledge base B are tuples (o, p, s) of subject $s \in S$,predicate $p \in P$

and object $o \in O$. A Boolean $|S|$-by-$|P|$-by-$|O|$ tensor $\boldsymbol{\mathcal{X}}$ representing the data can be extracted by bijectively mapping the different subject, predicate and objects to natural numbers ($f_s : S \to \mathbb{N}, f_p : P \to \mathbb{N}$ and $f_o : O \to \mathbb{N}$). An entry (i, j, k) of $\boldsymbol{\mathcal{X}}$ would be 1, when the database contains the entry of the coordinates mapped to its original domain.

$$(\boldsymbol{\mathcal{X}})_{i,j,k} = \begin{cases} 1 & \text{there exist } (s,p,o) \in B \text{ s.t. } f_s(s) = i, f_p(p) = ij \text{ and } f_o(o) = k \\ 0 & \text{otherwise} \end{cases}$$

For a data miner Boolean data can often be easier to interpret, depending on the data. Common techniques to analyse such data are for example association rules. Boolean tensor decomposition gives the data miner yet another tool. How we can extract information using boolean tensor decomposition has been explored by other works [EM13a].

Tensor decomposition is a hard problem, in the sense that most subproblems are NP-hard [HL13]. This naturally requires approximative algorithms. Combine this with the fact that common datasets increase in size in later years. Therefore, the algorithms also need to be highly scalable. By concentrating only on Boolean tensors, we can construct algorithms that outperform generalized tensor decomposition algorithms. Boolean tensor decomposition is still a small field, so there aren't many different dedicated algorithms yet, one is the Walk'n'Merge algorithm on which this work builds on and another one is the SaBoTeur algorithm [MM15].

The Walk'n'Merge algorithm for boolean tensor decomposition introduced in [EM13b] was implemented in Python. One of the goals of this work was to provide an implementation of Walk'n'Merge in a more low level programming language compared to the original. Therefore, the current implementation is in C++. Furthermore, improvements to the algorithm were developed and analysed. These where primarily evaluated in terms of runtime improvements, but also in terms of qualitative improvements when considering the reconstruction error.

One of the most important aspects of the Walk'n'Merge algorithm is the FiberGraph, which allows to find dense structures in a sparse tensor. Most of this work is dedicated to improve how the FiberGraph is analysed by using graph theoretical approaches, e.g. graph clustering. All of these exploration strategies together form the Walk'n'Merge ++ program, which in essence is a new implementation of Walk'n'Merge in C++ with a lot of new options. While it is not optimized for any single strategy it still allows comparing the different strategies.

Finally, I introduce the Split'n'Expand algorithm, based on the results of Walk'n'Merge ++. It uses a divide and conquer strategy to recursively reduce the size of the input tensor into smaller subtensors, that can be handled independently. This leads to a highly scalable algorithm.

1.2 Terminology, Notations and Definitions

1.2.1 Tensors, Matrices and Vectors

The terminology and notation are based on the one used in [EM13b] by Miettinen and Erdős, which introduced the Walk'n'Merge algorithm, and the one used by Kolda and Bader in [KB09]

A *N-way tensor* in context of this work is a simple N-dimensional data object, in a more general sense it is a N-dimensional array or matrix. Since this work mainly deals with 3 dimensional tensors, 3-way tensors are simply referred to as tensor. A Boolean tensor is naturally a tensor filled with the Boolean values, one and zero. Motivated by the idea, that the tensors this work deals with are sparse, I also refer to the ones as (non-zero) entries, while zeros are considered to be empty entries.
Vectors (or 1-way tensors) are denoted by lowercase, italic letters with the typical vector arrow (e.g. $\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}}, \dots$). Matrices (or 2-way tensors) are denoted by uppercase, bold letters (e.g. $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \dots$). Tensors (3-way or higher dimensional) are denoted by boldcase, euler letters (e.g. $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \boldsymbol{\mathcal{C}}, \dots$). Sets of elements are denoted by uppercase letters (e.g. $A, B, C, \dots$). The number of elements in a set X is denoted by $|X|$. When addressing a element from any of these structures, this is denoted by round brackets to not confuse them with indexed sets (e.g. (the i-th element of $\vec{\boldsymbol{a}}$ is given by $(\vec{\boldsymbol{a}})_i$)). When referring to tuples of sets (e.g. blocks) $(X_1, X_2, ..., X_n)$, I refer to the size of it as $|X_1| \times |X_2| \times \dots \times |X_3|$

Given a (3-way) tensor $\boldsymbol{\mathcal{X}}$ with dimensions n-by-m-by-l the element at position (i, j, k) with $i \in \{1, \dots, n\}, j \in \{1, \dots, m\}$ and $l \in \{1, \dots, l\}$ can be addressed as $(\boldsymbol{\mathcal{X}})_{ijk}$. When using : as an index, we refer to all possible values of the respective index. What are the rows and columns in a classical matrix are the *fibers* in a tensor. More concretely, $(\boldsymbol{\mathcal{X}})_{:jk}$ is the *(j, k) mode-1 (column) fiber*, $(\boldsymbol{\mathcal{X}})_{i:k}$ the *(i, k) mode-2 (row) fiber*, and $(\boldsymbol{\mathcal{X}})_{ij:}$ the *(i,j) mode-3 (tube) fiber.* Since the fibers are essentially vectors, they are often denoted as such in other literature ($(\boldsymbol{\mathcal{X}})_{:jk} = (\vec{\boldsymbol{x}})_{:jk}$). While fibers are 1 dimensional substructures, *slices* are two dimensional substructures. For a 3-way tensor we have the $(\boldsymbol{\mathcal{X}})_{i::}$ *horizontal* slices, the $(\boldsymbol{\mathcal{X}})_{:j:}$ lateral slices, and the $(\boldsymbol{\mathcal{X}})_{::l}$ frontal slices. They are often denoted as matrices with $(\boldsymbol{\mathcal{X}})_{i::} = (\boldsymbol{X})_{i::}$. A better visualization is available in figure 1.1 a 10-by-10-by-1 tensor is depicted in figure 1.2, which will serve as a example for the rest of this work.

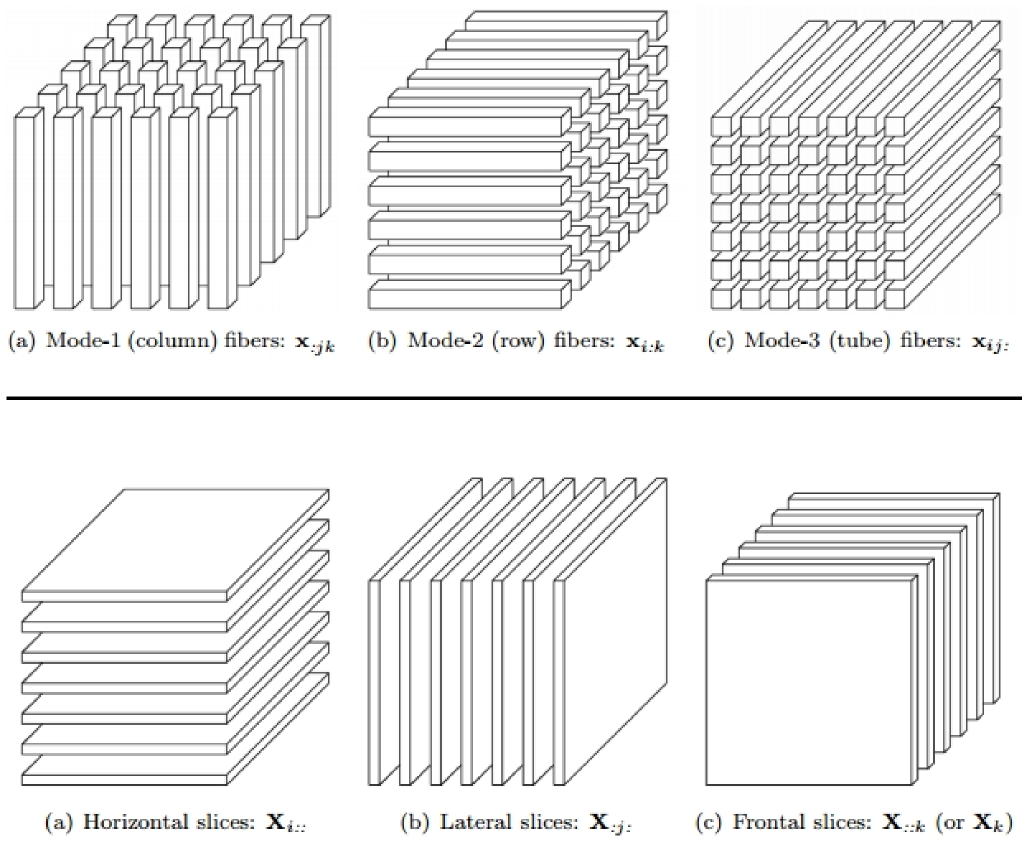

(a) Mode-1 (column) fibers: $\mathbf{x}_{:jk}$
(b) Mode-2 (row) fibers: $\mathbf{x}_{i:k}$
(c) Mode-3 (tube) fibers: $\mathbf{x}_{ij:}$
(a) Horizontal slices: $\mathbf{X}_{i::}$
(b) Lateral slices: $\mathbf{X}_{:j:}$
(c) Frontal slices: $\mathbf{X}_{::k}$ (or $\mathbf{X}_k$)

Figure 1.1: Illustration of fibers and slices from [KB09]

For boolean 3-way n-by-m-by-l tensors $\boldsymbol{\mathcal{X}}$ we define entries($\boldsymbol{\mathcal{X}}$) as the set of all non-zero entries (entries$(\boldsymbol{\mathcal{X}}) = \{(i,j,k) \in (\{1,\ldots,n\} \times \{1,\ldots,m\} \times \{1,\ldots,l\}) | (\boldsymbol{\mathcal{X}})_{ijk} \neq 0\}$)

$|\boldsymbol{\mathcal{X}}|$ denotes the number of non-zero elements in the tensor $\boldsymbol{\mathcal{X}}$ and $\|\boldsymbol{\mathcal{X}}\|$ the Frobenius-norm $(\sum_{ijk}((\boldsymbol{\mathcal{X}})_{ijk})^2)^{\frac{1}{2}}$.

Definition 1.1. *Tensor Operations:* For two same dimensional n-by-m-by-l tensors $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$, the *tensor sum* $\boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{Y}}$ is defined as the element-wise sum:

$$(\boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{Y}})_{ijk} = (\boldsymbol{\mathcal{X}})_{ijk} + (\boldsymbol{\mathcal{Y}})_{ijk}$$

Analogue we have for Boolean tensors the *Boolean tensor sum* defined as:

$$(\boldsymbol{\mathcal{X}} \vee \boldsymbol{\mathcal{Y}})_{ijk} = (\boldsymbol{\mathcal{X}})_{ijk} \vee (\boldsymbol{\mathcal{Y}})_{ijk}$$

Whereby $\vee$ is the standard logical *or*. The same goes for the *exclusive* or ($\oplus$):

$$(\boldsymbol{\mathcal{X}} \oplus \boldsymbol{\mathcal{Y}})_{ijk} = (\boldsymbol{\mathcal{X}})_{ijk} \oplus (\boldsymbol{\mathcal{Y}})_{ijk} = (\boldsymbol{\mathcal{X}})_{ijk} + (\boldsymbol{\mathcal{Y}})_{ijk} \mod 2$$

x	y
10	4
2	6
4	9
10	10
4	3
2	4
10	6
3	6
9	10
9	3
10	1
3	1
10	3
2	1
9	9
3	4
10	9

(a) Tabular representation

$$\begin{bmatrix} 0&0&0&0&0&0&0&0&0&0\\ 1&0&0&1&0&1&0&0&0&0\\ 1&0&0&1&0&1&0&0&0&0\\ 0&0&1&0&0&0&0&0&1&1\\ 0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&1&1\\ 1&0&1&1&0&1&0&0&1&1 \end{bmatrix}$$

(b) Matrix representation

Figure 1.2: Two different Representation of a small 2-way tensor, that is used throughout this work to illustrate the definitions.

Definition 1.2. *Outer Product*: The *outer product* ($\boxtimes$) is a mathematical operation between vectors. For two Boolean vectors $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{b}}$ with dimensions 1-by-n and 1-by-m, the outer product is given by the n-by-m matrix $\boldsymbol{A}$ with:

$$(\boldsymbol{A})_{ij} = (\vec{\boldsymbol{a}} \boxtimes \vec{\boldsymbol{b}})_{ij} = (\vec{\boldsymbol{a}})_i (\vec{\boldsymbol{b}})_j = (\vec{\boldsymbol{a}})_i \wedge (\vec{\boldsymbol{b}})_j$$

Furthermore, the outer product between three vectors results in a tensor. Given three Boolean vectors $\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}$ and $\vec{\boldsymbol{c}}$, with dimensions 1-by-n, 1-by-m and 1-by-l, the outer product is given by the n-by-m-by-l tensor $\boldsymbol{\mathcal{X}}$ with:

$$(\boldsymbol{\mathcal{X}})_{ijk} = (\vec{\boldsymbol{a}} \boxtimes \vec{\boldsymbol{b}} \boxtimes \vec{\boldsymbol{c}})_{ijk} = (\vec{\boldsymbol{a}})_i (\vec{\boldsymbol{b}})_j (\vec{\boldsymbol{c}})_k = (\vec{\boldsymbol{a}})_i \wedge (\vec{\boldsymbol{b}})_j \wedge (\vec{\boldsymbol{c}})_k$$

Definition 1.3. *Rank 1 Tensor:* For vectors $\vec{a}, \vec{b}$ and $\vec{c}$ of length n, m and l the outer product $\vec{a} \boxtimes \vec{b} \boxtimes \vec{c}$ yields a n-by-m-by-l *rank 1 tensor* $\boldsymbol{\mathcal{X}}$.

$$\vec{\boldsymbol{a}} \boxtimes \vec{\boldsymbol{b}} \boxtimes \vec{\boldsymbol{c}} = \boldsymbol{\mathcal{X}}$$

Definition 1.4. *Boolean Tensor Rank:* For a Boolean 3-way tensor $\boldsymbol{\mathcal{X}}$ the **Boolean rank** ($rank_B(\boldsymbol{\mathcal{X}})$) is given by the smallest integer r such that a decomposition into r

rank 1 tensors is possible:

$$\boldsymbol{\mathcal{X}} = \bigvee_{i=1}^{r} \vec{\boldsymbol{a}}_i \boxtimes \vec{\boldsymbol{b}}_i \boxtimes \vec{\boldsymbol{c}}_i$$

1.2.2 Blocks

Definition 1.5. *Blocks*: Let $\boldsymbol{\mathcal{X}}$ be a n-by-m-by-l Boolean tensor. A *block* is a tuple (X, Y, Z) of index sets, whereby $X \subset [n]$, $Y \subset [m]$ and $Z \subset [l]$ ($[i] := 1, ..., i$).

$\boldsymbol{\mathcal{B}}$ is a n-by-m-by-l tensor defined by (X, Y, Z), whereby:

$$(\boldsymbol{\mathcal{B}})_{ijk} = \begin{cases} 1 & i \in X \wedge j \in Y \wedge k \in Z \wedge (\boldsymbol{\mathcal{X}})_{ijk} = 1 \\ 0 & otherwise \end{cases}$$

We denote this by $\boldsymbol{\mathcal{B}} := \text{ten}((X, Y, Z))$

Definition 1.6. *Monochromatic Blocks*: A block (X, Y, Z) of Boolean tensor $\boldsymbol{\mathcal{X}}$ is *monochromatic*, when all its entries defined by the tuple (X, Y, Z) are 1 (i.e. $(\boldsymbol{\mathcal{X}})_{ijk} = 1$ for all $i \in X, j \in Y$, and $k \in Z$).

Definition 1.7. *Dense Blocks*: The *density* of a block (X, Y, Z) of Boolean tensor $\boldsymbol{\mathcal{X}}$ with $\boldsymbol{\mathcal{B}} = \text{ten}((X, Y, Z))$ is defined by the relative number of ones:

$$\text{density}((X, Y, Z)) = \frac{|\boldsymbol{\mathcal{B}}|}{|X||Y||Z|}$$

A block is *dense*, when the density is larger than a given threshold $d \in [0, 1]$.

Definition 1.8. *Convex Hull*: Let $\boldsymbol{\mathcal{X}}$ be a $n \times m \times l$ Boolean tensor and I, J and K be the sets of non-empty slices. That means for all $i \in I$ exist $j \in [m] \wedge k \in [l]$ with $(\boldsymbol{\mathcal{X}})_{ijk} = 1$. The **convex Hull** of $\boldsymbol{\mathcal{X}}$ is given by the tensor $\boldsymbol{\mathcal{Y}}$ with $(\boldsymbol{\mathcal{Y}})_{ijk} = 1$ for all $i \in I$, $j \in J$ and $l \in L$.

We denote this by $\boldsymbol{\mathcal{Y}} = \text{convexHull}(\boldsymbol{\mathcal{X}})$.

Definition 1.9. Operations on Blocks For blocks classical set operations are defined in a element wise manner. Given two blocks (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2), we define:

$$(X_1, Y_1, Z_1) \cup (X_2, Y_2, Z_2) = (X_1 \cup X_2, Y_1 \cup Y_2, Z_1 \cup Z_2)$$

$$(X_1, Y_1, Z_1) \cap (X_2, Y_2, Z_2) = (X_1 \cap X_2, Y_1 \cap Y_2, Z_1 \cap Z_2)$$

Definition 1.10. Two blocks (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) are *connected* if they overlap in two components, e.g. $X_1 \cap X_2 \neq \varnothing$ and $Y_1 \cap Y_2 \neq \varnothing$. This means, that they are connected in the later defined FiberGraph. Two blocks *overlap*, if the components overlap in all three components. This means, that they actually share coordinates.

The sample tensor from figure 1.2 can be completely described using two blocks as show in figure 1.3. One block is represented by the red numbers and the other by blue ones.

$$
\begin{aligned}
&\begin{bmatrix}
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&1&1\\
0&0&1&0&0&0&0&0&1&1
\end{bmatrix} \qquad (= \mathrm{ten}((\{9,10,4\}\{9,10,3\}\{1\})))\\
\oplus &\begin{bmatrix}
0&0&0&0&0&0&0&0&0&0\\
1&0&0&1&0&1&0&0&0&0\\
1&0&0&1&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
1&0&0&1&0&1&0&0&0&0
\end{bmatrix} \qquad (= \mathrm{ten}((\{10,3,2\}\{1,4,6\}\{1\})))\\
= &\begin{bmatrix}
0&0&0&0&0&0&0&0&0&0\\
1&0&0&1&0&1&0&0&0&0\\
1&0&0&1&0&1&0&0&0&0\\
0&0&1&0&0&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&1&1\\
1&0&1&1&0&1&0&0&1&1
\end{bmatrix}
\end{aligned}
$$

Figure 1.3: A decomposition of the demo tensor from figure 1.2 into two tensors defined by the two blocks $(\{9,10,4\}\{9,10,3\}\{1\})$ and $(\{10,3,2\}\{1,4,6\}\{1\})$

Theorem 1.11. *A block (X, Y, Z) of Boolean tensor $\boldsymbol{\mathcal{X}}$ corresponds to a rank 1 subtensor* $\boldsymbol{\mathcal{B}} := \mathrm{convexHull}(\mathrm{ten}((X, Y, Z)))$

Proof. Given a block (X, Y, Z) of n-by-m-by-l Boolean tensor $\boldsymbol{\mathcal{X}}$. We construct the vectors $\vec{\boldsymbol{a}}$ (1-by-n),$\vec{\boldsymbol{b}}$ (1-by-m) and $\vec{\boldsymbol{c}}$ (1-by-l) that make up the subtensor as follows:

$$
\begin{aligned}
X' &= \{i \in X | \exists j \in Y \wedge k \in Z s.t. (\boldsymbol{\mathcal{X}})_{ijk} \neq 0\}\\
Y' &= \{j \in Y | \exists i \in X \wedge k \in Z s.t. (\boldsymbol{\mathcal{X}})_{ijk} \neq 0\}\\
Z' &= \{k \in Z | \exists j \in Y \wedge i \in X s.t. (\boldsymbol{\mathcal{X}})_{ijk} \neq 0\}
\end{aligned}
$$

This is needed to remove the empty slices.

$$(\vec{\boldsymbol{a}})_i = \begin{cases} 1 & i \in X' \\ 0 & otherwise \end{cases} \quad \text{for all } 1 \leq i \leq n$$
$$(\vec{\boldsymbol{b}})_i = \begin{cases} 1 & i \in Y' \\ 0 & otherwise \end{cases} \quad \text{for all } 1 \leq i \leq m$$
$$(\vec{\boldsymbol{c}})_i = \begin{cases} 1 & i \in Z' \\ 0 & otherwise \end{cases} \quad \text{for all } 1 \leq i \leq l$$

Let $\boldsymbol{\mathcal{B}}' = \vec{\boldsymbol{a}} \boxtimes \vec{\boldsymbol{b}} \boxtimes \vec{\boldsymbol{c}}$. It remains to show that $\boldsymbol{\mathcal{B}}' = \boldsymbol{\mathcal{B}}$. I will show that $(\boldsymbol{\mathcal{B}}')_{ijk} = 1 \iff (\boldsymbol{\mathcal{B}})_{ijk} = 1$.

$$\begin{aligned}
& (\boldsymbol{\mathcal{B}}')_{ijk} = 1 \\
\iff & (\vec{\boldsymbol{a}} \boxtimes \vec{\boldsymbol{b}} \boxtimes \vec{\boldsymbol{c}})_{ijk} = 1 && \text{definition of } \boldsymbol{\mathcal{B}}' \\
\iff & (\vec{\boldsymbol{a}})_i = 1 \wedge (\vec{\boldsymbol{b}})_j = 1 \wedge (\vec{\boldsymbol{c}})_k = 1 && \text{definition of } \boxtimes \\
\iff & i \in X' \wedge j \in Y' \wedge k \in Z' && \text{definition of } \vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}} \\
\iff & \text{convexHull}(\text{ten}((X, Y, Z)))_{ijk} = 1 && \text{definition of convex hull} \\
\iff & (\boldsymbol{\mathcal{B}})_{ijk} = 1 && \text{definition of } \boldsymbol{\mathcal{B}}
\end{aligned}$$

□

Theorem 1.12. *Given a block (X, Y, Z) of Boolean 3-way tensor $\boldsymbol{\mathcal{X}}$ with $X = X' \cup \{x'\}$, $X \setminus X' = \{x'\}$,$|X| \geq 2$, $|Y| \geq 1$ and $|Z| \geq 1$ (Meaning X and X' are disjoint). Then holds:*

$$\text{density}(X, Y, Z)\left(1 + \left(\tfrac{1}{|X|-1}\right)\right) \geq \text{density}(X', Y, Z)$$

Proof.

$$\begin{aligned}
&\text{density}((X,Y,Z)) = \frac{\sum\limits_{i\in X, j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk}}{|X||Y||Z|} \\
\iff &\text{density}((X,Y,Z)) = \frac{\sum\limits_{i\in X'\cup\{x\}, j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk}}{|X'\cup\{x\}||Y||Z|} \\
\iff &\text{density}((X,Y,Z)) = \frac{\sum\limits_{i\in X', j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk} + \sum\limits_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk}}{|X'||Y||Z| + |Y||Z|} \\
\iff &\text{density}((X,Y,Z))(|X'||Y||Z| + |Y||Z|) = \sum_{i\in X', j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk} + \sum_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk} \\
\iff &\text{density}((X,Y,Z))(|X'||Y||Z|) + \text{density}((X,Y,Z))(|Y||Z|) \\
&= \sum_{i\in X', j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk} + \sum_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk} \\
\iff &\text{density}((X,Y,Z))(|X'||Y||Z|) + \text{density}((X,Y,Z))(|Y||Z|) - \sum_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk} \\
&= \sum_{i\in X', j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk} \\
\iff &\frac{\text{density}((X,Y,Z))(|X'||Y||Z|)}{|X'||Y||Z|} + \frac{\text{density}((X,Y,Z))(|Y||Z|)}{|X'||Y||Z|} - \frac{\sum\limits_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk}}{|X'||Y||Z|} \\
&= \frac{\sum\limits_{i\in X', j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{ijk}}{|X'||Y||Z|} \\
\iff &\text{density}((X,Y,Z)) + \text{density}((X,Y,Z))\left(\frac{|Y||Z|}{|X'||Y||Z|}\right) - \frac{\sum\limits_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk}}{|X'||Y||Z|} \\
&= \text{density}(X', Y, Z) \\
\iff &\text{density}((X,Y,Z))\left(1 + \left(\frac{|Y||Z|}{|X'||Y||Z|}\right)\right) - \frac{\sum\limits_{j\in Y, z\in Z}(\boldsymbol{\mathcal{X}})_{x'jk}}{|X'||Y||Z|} = \text{density}(X', Y, Z) \\
\implies &\text{density}((X,Y,Z))\left(1 + \left(\frac{|Y||Z|}{|X'||Y||Z|}\right)\right) \geq \text{density}(X', Y, Z) \\
\iff &\text{density}((X,Y,Z))\left(1 + \left(\frac{1}{|X'|}\right)\right) \geq \text{density}(X', Y, Z) \\
\iff &\text{density}((X,Y,Z))\left(1 + \left(\frac{1}{|X|-1}\right)\right) \geq \text{density}(X', Y, Z)
\end{aligned}$$

□

1.2.3 Decompositions

Among the several available decompositions, two are relevant for this work.

- **CP-Decomposition:** Given a n-by-m-by-l (Boolean) tensor $\boldsymbol{\mathcal{X}}$ and an integer r, we search for a n-by-r matrix $\boldsymbol{A}$, a m-by-r matrix $\boldsymbol{B}$ and a l-by-r matrix $\boldsymbol{C}$, we optimize:
$$\min(|\boldsymbol{\mathcal{X}} \oplus \bigvee_{i=1}^{r} (\boldsymbol{A})_{:i} \boxtimes (\boldsymbol{B})_{:i} \boxtimes (\boldsymbol{C})_{:i}|) \tag{1.1}$$
- **Boolean Tucker Decomposition:** [Tuc6c] Given a Boolean n-by-m-by-l tensor $\boldsymbol{\mathcal{X}}$ and an integers p, q, r the minimum-error (p, q, r) Boolean Tucker decomposition of $\boldsymbol{\mathcal{X}}$ is a tuple $(\boldsymbol{\mathcal{G}}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$, whereby $\boldsymbol{\mathcal{G}}$ is a p-by-q-by-r Boolean tensor, $\boldsymbol{A}$ is a n-by-p Boolean (factor) matrix, $\boldsymbol{B}$ a m-by-q one and $\boldsymbol{C}$ a l-by-r one. These minimize
$$\sum_{ijk}((\boldsymbol{\mathcal{X}})_{ijk} \oplus \bigvee_{\alpha=1}^{p} \bigvee_{\beta=1}^{p} \bigvee_{\gamma=1}^{p} (\boldsymbol{\mathcal{G}})_{\alpha\beta\gamma} (\boldsymbol{A})_{i\alpha} \boxtimes (\boldsymbol{B})_{j\beta} \boxtimes (\boldsymbol{C})_{k\gamma}) \tag{1.2}$$

Each of these decompositions of a tensor $\boldsymbol{\mathcal{X}}$ contain a implicit reconstruction $\boldsymbol{\mathcal{X}}'$ (e.g. $\bigvee_{i=1}^{r} (\boldsymbol{A})_{:i} \boxtimes (\boldsymbol{B})_{:i} \boxtimes (\boldsymbol{C})_{:i}$ for the CP-decomposition). The exclusive or of the two $(\boldsymbol{\mathcal{X}} \oplus \boldsymbol{\mathcal{X}}')$ has entries of one, where they differ and consequently the size $(|\boldsymbol{\mathcal{X}} \oplus \boldsymbol{\mathcal{X}}'|)$ of it is the number of errors, which is what both decompositions aim to minimize. This is the so called *reproduction error*, in other fields its equivalent to the accuracy measure.

The CP-decomposition aims to decompose the tensor $\boldsymbol{\mathcal{X}}$ into r rank 1 tensors, whereby the different tensors are made up by the outer products of the columns of the matrices $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{C}$. This is equivalent to selecting r blocks from $\boldsymbol{\mathcal{X}}$, per theorem 1.11. The Tucker decomposition extends this with the core tensor, this allows to select which rows of the matrices are used together.

The Walk'n'Merge algorithm aims to find a CP-decomposition with the slight difference, that the algorithm doesn't aim to find a fixed sized decomposition. A CP-decomposition for a fixed rank can be found by selecting a subsets of the blocks provided by Walk'n'Merge. This is a NP-hard problem and the original paper proposed a greedy approach for this. This approach is shortly described in algorithm 1. The interesting part of this is the choice of the score function. The original paper optimized the encoding length in a MDL inspired approach.

The Tucker decomposition can be constructed in a similar manner. We start with a CP-Decomposition of rank s, with s larger than the parameters p, q, r of the Tucker Decomposition. This leads to the initial Tucker Decomposition with a trivial s-by-s-by-s

```
Input: tensor $\boldsymbol{\mathcal{X}}$, blocks $((X_1, Y_1, Z_1), (X_2, Y_2, Z_2), \ldots, (X_s, Y_s, Z_s))$, integer $r \leq s$
Output: blocks $((X_1, Y_1, Z_1), (X_2, Y_2, Z_2), \ldots, (X_r, Y_r, Z_r))$
1  selectedBlocks = empty set of blocks;
2  forall the $r$ iterations do
3      minScore = $\infty$;
4      minBlock = empty Block ;
5      forall the newBlock $\in$ blocks $\div$ selectedBlocks do
6          if score($\{newBlock\} \cup selectedBlocks$) < minScore then
7              minScore = score($\{newBlock\} \cup selectedBlocks$);
8              minBlock = newBlock
9      selectedBlocks = selectedBlocks $\cup \{minBlock\}$
10 return selectedBlocks;
```

Algorithm 1: Pseudo code for the basic CP selection algorithm

core tensor, that has 1 on its diagonal and 0 otherwise. Then we combine rows of the components matrices in a greedy manner, till we reach the desired dimension.

1.2.4 FiberGraph

Definition 1.13. FiberGraph For a given Boolean 3-way tensor $\boldsymbol{\mathcal{X}}$ the **FiberGraph** $FG(V, E)$ with set of vertices V and edge relation $E \in V \times V$ is defined by:

- **Nodes(V):** for every entry with $(\boldsymbol{\mathcal{X}})_{ijk} = 1$ there is a node (v_{ijk}).
- **Edges(E):** Two nodes (v_{ijk}, v_{pqr}) are connected, when their coordinates differ in exactly one position *(e.g. $i = p, j = q$ and $k \neq r$).*

The FiberGraph has a vertex (or node) for every non-zero entry of the tensor. This is a one-to-one relation, that motivates the following notations. The vertex v corresponds to the non-zero entry at coordinate $(v.x, v.y, v.z)$ and coordinate (i, j, k) corresponds to vertex v_{ijk}. The connections in the graph correspond to non-zero entries, that lie on the same fiber (the row and column equivalents for tensors).

Given a node v of a FiberGraph $FG(V, E)$, we define the set of neighbours of the neighbours of v as $\mathrm{n}(v) = v' \in V | (v, v') \in E$. This is the set of all Nodes that share a fiber with v.

The FiberGraph is the most important component of the Walk'n'Merge algorithm, because it has some useful properties, that help to find blocks in a tensor. Theorem 1.14, shows that a monochromatic block is represented by a subgraph of at most diameter 3. Intuitively, the denser a block is (the more he resembles a monochromatic block), the smaller is the diameter of the corresponding subgraph. Additionally, the denser a block is, the more paths exist between its entries.

One should keep in mind, that the relation between tensor and FiberGraph isn't a one-to-one relation. A tensor has exactly one FiberGraph (modulo bijection), but for a FiberGraph there are multiple pssoible tensors.

Theorem 1.14. *Block Connectivity Given a Boolean 3-way tensor $\boldsymbol{\mathcal{X}}$ and its* FiberGraph *$FG(V, E)$, all entries of a monochomatic block (X, Y, Z) are connected by paths of length 3.*

Proof. Let v_1 and v_2 be entries of the block (X, Y, Z). This means the block contains at least $\{v_1.x\} \cup \{v_2.x\} \subset X$, $\{v_1.y\} \cup \{v_2.y\} \subset Y$ and $\{v_1.z\} \cup \{v_2.z\} \subset Z$. Then the following path exists:

$$(v_1.x, v_1.y, v_1.z) \rightarrow (v_1.x, v_1.y, v_2.z) \rightarrow (v_1.x, v_2.y, v_2.z) \rightarrow (v_2.x, v_2.y, v_2.z)$$

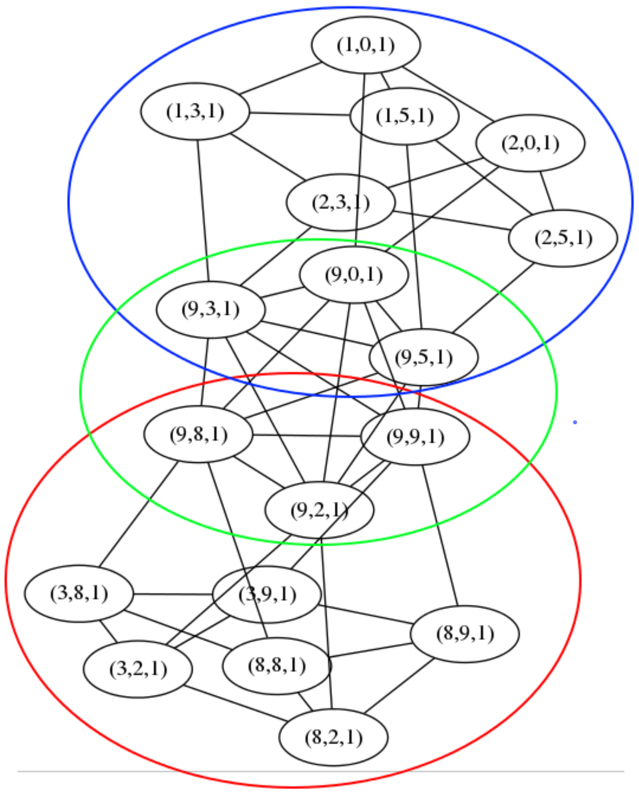


Figure 1.4: The FiberGraph for the sample tensor from figure 1.2

the different nodes, have to be part of the blocks, because of the definition of monochromatic blocks. They are connected in the FiberGraph, because the coordinates differ by exactly one coordinate. However, some of the steps may be none-steps, because some coordinates could be the same. □

The FiberGraph is not a perfect tool. One weakness it, that connected components need not to necessarily correspond to the same block. The most likely biggest challenge of Walk'n'Merge is differentiate between different blocks.

An Example FiberGraph is depicted in figure 1.4. We can see, how the two blocks marked by the blue and red circles are heavily interconnected, but also the green subgraph with clique, that corresponds to a single very dense fiber, where the two blocks share a coordinate.

Chapter 2

Walk'n'Merge

The Walk'n'Merge algorithm consists of 2 phases, which are also the foundation of Walk'n'Merge++. In an abstract way they can be described as follows:

- The first phase tries to find blocks or rank 1 tensors within the original tensor. In the original implementation this is the RandomWalk algorithm of Walk'n'Merge.
- The second phase uses the results of the first phase to construct larger blocks by combining smaller ones. This is the BlockMerge algorithm of Walk'n'Merge.

2.1 RandomWalk

The most fundamental structure for the walk part is the FiberGraph from section 1.2.4, which has a node for every non-zero entry of the tensor and an edge between two nodes, when these share a column, row or tube fiber. Formally, this works as follows:

The basic idea is that a dense structure within the FiberGraph should correspond to a dense structure in the tensor. One reason for this is that a monochromatic block is represented by a subgraph with diameter of at most 3 as per theorem 1.14, because there is a path of length 3 between every two nodes along the columns, rows and tubes. Dense blocks are expected to have on average similar small diameters. More importantly one can show, that a neighbouring node of a node in the FiberGraph is with high probability also part of this block [EM13b]. At least, as long as the tensor is relatively sparse, it is rather unlikely to have a neighbours that was not generated by the same block.

The walk algorithm described by algorithm 2 tries to find these by randomly sampling the graph. Starting from a random node a certain number of random walks are started and

```
   Input: tensor $\boldsymbol{\mathcal{X}}$,density $d$,$walk\_length$,$walk\_num$,$freq$
   Output: blocks $(X_1, Y_1, Z_1)$, $(X_2, Y_2, Z_2)$,...
1  create FiberGraph $FG(V, E)$ from $\boldsymbol{\mathcal{X}}$;
2  while $V$ is not empty do
3      $v$ = random node from $V$;
4      $visitedNodes$.insert($v, count_v = 1$);
5      for $walk\_num$ iterations do
6          $v'$ = random node from $visitedNodes$ ;
7          for $walk\_len$ iterations do
8              $v'$ = random neighbour of $v'$;
9              $visitedNodes$.insert($v', count_{v'} ++$)
10     empty block $(X, Y, Z)$ for $v \in visitedNodes$ do
11         if $count_v > freq$ then
12             $(X, Y, Z)$.add($(v.x, v.y, v.z)$);
13     $V.remove(convexHull(\mathrm{ten}(X, Y, Z)))$;
14     if density of ten$(X, Y, Z)$ in X $> d$ then
15         $blocks$.add($(X, Y, Z)$);
16     return $blocks$;
```

Algorithm 2: Pseudo code for the basic RandomWalk algorithm

whenever a node is encountered a counter is increased. Note, that it is important, that the start node (line 6) isn't chosen uniformly at random from *visitedNodes*, but chosen weighted by the frequency of the node. This means that nodes that where encountered more often have a higher probability to be selected again. When a node is part of a block, it will naturally be encountered more often as the subgraph with the node has a small diameter and so the probability of restarting the walk within increases and the frequency of nodes in the block increases.

The FiberGraph for the sample tensor is depicted in figure 1.4 and a version with sample frequencies can be seen in figure 3.1. We have seen before that the tensor is made up by two blocks, these are marked by the blue and red circles. The dense cluster marked by the green circle are the entries from the last row. This picture illustrates very well the good and bad properties of the FiberGraph. On one hand the entries corresponding to the corresponding to the blocks are densely clustered. However, the green cluster is even denser and although is corresponds to a monochromatic block removing this nodes, will make us unable to find the original two blocks with RandomWalk. The best we can than hope for is to find a decomposition into 3 blocks.

The runtime of this algorithm is mostly depended on how fast the graph is shrinking. The worst case (in terms of runtime) is, that no node has any neighbours. Then the runtime is bound by $O(|V| * walk\_len * walk\_num)$. The original implementation parallelizes this process by doing multiple walks at once with synchronization steps after a fixed number iterations. This results in a not completely semantic identical algorithm, but the differences are negligible.

2.2 BlockMerge

The BlockMerge is important for those decompositions, that rely on a small number of influential (i.e. dense) blocks, because it tries to combine the smaller blocks into larger (i.e. more influential) ones. The ultimate goal is to have dense blocks that are as large as possible. A bit of a challenge is that larger blocks tend to have more false positives. Therefore, decompositions of larger blocks tend to have a worse reconstruction error. A optimal error, can be achieved by having each non zero entry of the tensor as its own block. The BlockMerge algorithm is separated into the MonoMerge and the SubsequentMerge algorithms.

The theoretical worst case runtime bound from the original paper is given by $O(|\mathtt{X}|(b^2 + D^3))$. Whereby D is the original number of blocks and b is the theoretical number of ones in the densest fiber.

Two blocks (X, Y, Z) and (X', Y', Z') are merged, by computing their elementwise unions, meaning:

$$\text{merge}((X, Y, Z), (X', Y', Z')) = (X \cup X', Y \cup Y', Z \cup Z')$$

This corresponds to computing the convex Hull of these two blocks.

2.2.1 MonoMerge

The first step is to find the smaller monochromatic blocks that where not found by the RandomWalk algorithm. This is realized by a index structure and is referred to as the MonoMerge algorithm. The index structure maps x coordinates of non-zero entries to possible y-coordinates and these (x, y)-coordinates again to sets of z coordinates. The algorithm than goes through all possible pairs of x coordinates (x_1, x_2), checks whether they have at least two possible y coordinates in common (y_1, y_2) and lastly checks for all combinations of (x_1, x_2) and (y_1, y_2) pairs, if the set of common z coordinates has at least size 2. This results in monochromatic blocks of size $2 \times 2 \times n$ with $n \geq 2$. Possibly, this can lead to a very large amount of blocks, which later significantly impacts the performance of SubsequentMerge.

2.2.2 SubsequentMerge

The original implementation parallelizes SubsequentMerge by dealing with multiple blocks at once in line 3 of the algorithm again with synchronization steps after a fixed amount of iterations.

The second step actually starts merging the blocks. This is the SubsequentMerge algorithm and described by algorithm 3.

```
Input: tensor $\boldsymbol{\mathcal{X}}$, density $d$, blocks $((X_1, Y_1, Z_1)$, $(X_2, Y_2, Z_2)$,...)
Output: blocks $((X_1, Y_1, Z_1)$, $(X_2, Y_2, Z_2)$,...)
1  find monochromatic blocks of fixed size and add them to blocks;
2  $Q$ = Queue containing all blocks from blocks;
3  while $V$ is not empty do
4      $(X_1, Y_1, Z_1) = Q$.pop();
5      forall the blocks $(X_2, Y_2, Z_2)$ that share coordinates with $(X_1, Y_1, Z_1)$ do
6          D = merge($(X_1, Y_1, Z_1), (X_2, Y_2, Z_2)$);
7          if density(D) $> d$ then
8              $Q$.push(D);
9              replace B and C in blocks with D;
10             break;
11 return blocks;
```

Algorithm 3: Pseudo code for the basic BlockMerge algorithm

Chapter 3

Walk'n'Merge++

This parts focuses on the new C++ implementation of the WALK'N'MERGE algorithm, which mainly means adding several new features to the base algorithm. Section 3.1 shows where the existing algorithm was modified not on a implementation level, but on a logical level. Section 3.1.4 gives some additional technical details of the new implementation. The sections afterwards discusses the different modifications.

3.1 Base Algorithm

3.1.1 RandomWalk Changes

At its core the algorithm stayed the same, with some further options that are discussed in later topics. The merge part on an abstract level stayed the same as the parallel python version. The algorithm is however kept more general, to swap components out. For the RANDOMWALK algorithm the notable changes are the following:

- **at line 4:** The parallelization is realized by using an openMP directive. This is a for-loop that schedules the different iterations in parallel. First a number of possible *startNodes* are selected at random from the nodes in the graph. The walks are then executed in parallel. The blocks construction as well as the node deletion are first done locally and then synchronized over all threads every few steps to keep the internal consistency, because you don't want to delete a node that is currently visited by another thread during the walk.
- **at line 12:** The *selectCandidates* functions generalizes the way that the candidates for the block are generated from the list of visited Nodes and their frequencies. The original algorithm used a fixed threshold and selected all nodes with a frequency

higher than it. The threshold was set to average node frequency of the run. This is still the default behaviour of the implementation. Section 3.2 deals with different selection Functions.

- **at line 14:** The deleteNodes function generalizes the way nodes are deleted. In the original version the convex hull of the visited Nodes is deleted. This topic is dealt with more thoroughly in later sections.

- **at line 17:** The convergence was a necessary innovation, because some combinations of candidate selection functions and node deletions functions, mean that not all nodes want be deleted, so that the algorithm won't terminate. The chosen criteria is, that the number of removed nodes is less or equal to the number of walk iterations during a parallel section which means, that each walk removed only one node ore less.

- **at line 1** There are different versions of the FiberGraph discussed in this work, although they didn't prove to be good changes.

```
Input: tensor X, density d, walk_length, walk_num, function selectCandidate(), function
       deleteNodes()
Output: blocks $\{(X_1, Y_1, Z_1), (X_2, Y_2, Z_2), \dots\}$
1  create FiberGraph $G(V, E)$ from X;
2  while $V$ is not empty do
3      $startNodes$ = random nodes from $V$;
4      for (in parallel) node $v$ in $startNodes$ do
5          for walk_num iterations do
6              $visitedNodes$.insert($v$, $count_v = 1$);
7              $v'$ = random node from $visitedNodes$;
8              for walk_len iterations do
9                  $v'$ = random neighbour of $v'$;
10                 $visitedNodes$.insert($v'$, $count_{v'}++$);
11         empty block $(X, Y, Z)$;
12         $candidates$ = selectCandidates($visitedNodes$);
13         $(X, Y, Z)$.add($candidates$);
14         deleteNodes(X, $visitedNodes$);
15         if $density((X, Y, Z)) > d$ then
16             blocks.add($(X, Y, Z)$);
17     if walk converges then
18         break;
19 return $blocks$;
```

Algorithm 4: Pseudo code for the new implementation of the walk phase

Some of the sections deal with algorithms, that can be used instead of the RandomWalk, most notably are the different clustering algorithm.

3.1.2 BlockMerge Changes

The merge part is a more difficult challenge, because it's one of the major bottlenecks of the whole algorithm. The MonoMerge algorithm tends to find many small blocks, and combining them into larger blocks takes a long time, because many combinations have to be checked.

On a abstract level BlockMerge works the same, but the whole process is parallelized more. In the original implementation the MonoMerge algorithm isn't parallelized. This was changed in Walk'n'Merge++, by having different threads for the candidate pairs of x-coordinates (x_1, x_2), while scanning for possible y and z coordinates.

The SubsequentMerge algorithm is parallelized like in the original version, where one thread gets on block and compares it with all other candidate blocks.

Walk'n'Merge++ implements some techniques to reduce the amount of computations by reducing the the number of considered block merges. These techniques are discussed in Section 3.5.

3.1.3 Preprocessing

The tensor files use a common representation of (sparse Boolean) tensors with some modifications. The first lines contain some meta information, the number of non-zero entries and the size of the tensor. Simply, to save some time not recomputing these ever again and to improve the memory allocation. The entries are represented in a linewise fashion, by the x, y and z coordinates within the tensor. In the first line stand the dimensions of the tensors (i.e. the maximal x,y and z coordinates) and in the second line is the number of entries. The entries are assumed to be unique, although this is not explicitly checked and the algorithm work fine even with non unique entries. Only the results could be a bit strange.

Depending on the chosen processing method, we have several preprocessing steps (For example the construction of the FiberGraph, constructing the index structure for MonoMerge, etc.). These don't influence each other and are therefore paralellized in Walk'n'Merge++.

3.1.4 Technical Details

The program was implemented in C++ using the $C++11$ standard. The code was compiled using the 4.8 gnu c-compiler [SD09] with the *-O3 -march=native* optimization flags. The *-O3* flag enables most debugging options of gcc and the *-march=native* flag makes the comipler compile only for the target hardware. Different parts of the algorithm needed external libraries.

- **OpemMP:** (*(Open Multi-Processing)*)[DM98] OpenMp is a API for parallel code in C, $Fortran$ and $C++$, supporting many different operating systems. It stand out because of its ease of use, through adding only a few keywords (compiler directives) to the code. For example, it allows to simply paralellize a for-loop by only adding the keyword *#pragma omp parallel for*.
- **Armadillo:** [San10] Armadillo is a C++ library that provides access to a large number of linear algebra operations (Also for sparse matrices) through the use of several external libraries. Among them are the well known BLAS library (the OpenBLAS implementation) [BLA02], LAPACK [ABD+90] and SUPERLU [Li05].
- **Gurobi:** [GO15] Gurobi is a efficient mathematical Problem solver tool with support for c++.
- **MCL:** (*Markov Chain Clustering*) MCL is a program for clustering graphs, that is more thoroughly explained in the clustering Section 3.3.

Walk'n'Merge++ has a lot of options, that share code, e.g. there is only on implementation of the FiberGraph. The reason for this is, that it was not in the scope of the work to optimize every different option. Therefore, Walk'n'Merge++ should be seen as a tool to compare different approaches and not a stand alone decomposition tool. A tool that has better optimized performance is discussed in Chapter 5.

3.1.5 Parameters

One of the main problems of the Walk'n'Merge++ program is, that it depends on a lot of parameters and that these parameters have significant impact on the performance of the algorithm. Especially, the experimental evaluation gets difficult, because the time needed to evaluate all combinations increases exponential, with the number of parameters. Therefore we need default parameters that have reasonable good performance for most inputs. This section introduces the most important parameters and motivates their default values.

The default parameters result in a behaviour very similar to the original Walk'n'Merge. All the variants discussed in this sections are not default settings. The default settings are documented in Appendix A.

Walk Parameters

More interesting are the parameters concerning the length of the walk, $walk\_len$ and $walk\_num$. Overall does the algorithm visit $walk\_len$ times $walk\_num$ nodes and the smaller $walk\_len$ is, the more often will the walk be restarted. Some experiments that discuss which parameters to take are described in Section 4.3. There are some parameters, that control the amount of parallelization. These are job_factor and $thread\_num$. These can again be tricky to choose, because a high job_factor means, that more work is done without synchronizing, which could lead to more overlapping blocks and redundant computations.

Block Parameters

The size of blocks is influenced by three parameters. The important is the block density ($block\_min\_density$), that lies in $[0, 1]$. Technically, a value of 0.1 would be a valid parameter, but it would be hard to extract information from such a loose decomposition. The algorithm in general assume, that dense means a value of 0.8 or larger and the algorithm are optimized accordingly. Furthermore, there are the $block\_min\_size$ and the $block\_min\_vol$ parameters, that limit the minimal size of blocks in different ways. Blocks (X, Y, Z) found by Walk'n'Merge++ need to fulfil:

$$|X| > block\_min\_size$$

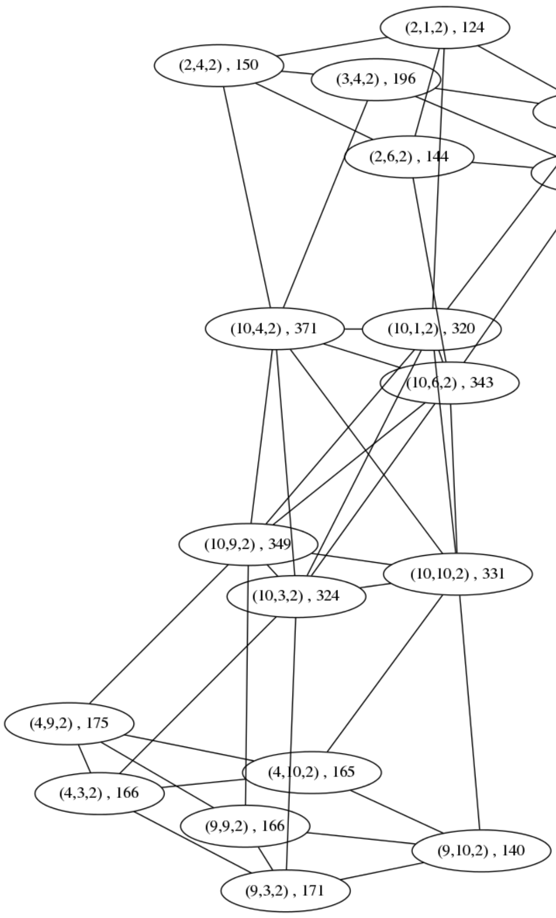
$$|Y| > block\_min\_size$$

$$|Z| > block\_min\_size$$

$$|X_C||Y_C||Z_C| > block\_min\_vol$$

These parameters helps by reducing the number of non-influential blocks in the result. Looking separately at size and volume, helps with irregular shaped blocks, e.g. some of the real world data sets don't contain dense $2 \times 2 \times 2$ blocks, but long $1 \times 1 \times n$ blocks. In that case you would use a small size, but a large volume to remove trivial blocks, while still getting a usable result.

Figure 3.1: The FiberGraph for the sample tensor from figure 1.2 with frequencies from a walk with *walk_length* 10 and *walk_num* 1000, which are the standard parameters



3.2 Variant: Selection Mode

The selection comes into play, when we have the frequencies of the walk and need to select the candidates for creating a block with high enough density. The challenges hereby are to not select too many nodes (e.g. by selecting all visited nodes), because this would reduce the overall density of the block, not to select too few, as we want the blocks be as large as possible, and not to have the runtime of the selection method be too bad, because we have to execute it for every walk once. In some sense this problem is rather similar to the original problem of finding blocks in the tensors, only that we now need to find only one block and have the frequencies to help our selection.
The selection process has the following constraints:

- The selection happens rather frequently (ones per walk iteration), so the *runtime* shouldn't be too high. Most of the proposed methods have linear runtime (or at least linear logarithmic one).

- It should be *dynamic* in the sense that it shouldn't rely on a fixed threshold. Selecting candidates when their frequency is over a fixed threshold doesn't include the size of blocks. When exploring a small block the frequencies are far higher than visiting a large block.

Formally we have a tensor $\boldsymbol{\mathcal{X}}$ with FIBERGRAPH $FG(V, E)$ and a frequency function (freq : $V \to \mathbb{R}_{\geq 0}$), where the frequency is the number of times it was visited during RANDOMWALK. The selection function is a function select : $V \times \mathbb{R}_{\geq 0} \to V \times \mathbb{R}_{\geq 0}$ whose input is a set of n candidates C from $V \times \mathbb{R}_{\geq 0}$ with $C = \{(v_1, f_1), (v_2, f_2), \ldots, (v_n, f_n) | v_i \in V \text{ and} f_i = \text{freq}(v_i) \forall 1 \leq i \leq n\}$ from which it selects a set S of $m \leq n$ candidates.
By $F_C = \{f_i | (v_i, f_i) \in C\}$ we denote the set off a frequencies in C and by $V_C = \{v_i | (v_i, f_i) \in C\}$ the set of all vertices. They are assumed to have the same order as the elements of C.

3.2.1 Selecting by Average

$$\text{select}_{Avg}(C) = \{(v, f) \in C | f \geq \text{mean}(F_C)\}$$

This is the method of the original implementation. It selects all nodes that where more often visited than the average. Formally, the set S of selected entries is made up by $S = \{(v, f) \in C | f \geq \text{mean}(F_C)\}$, whereby $\text{mean}(F_C)$ is the arithmetic mean $(\text{mean}(X) = \frac{1}{|X|} \sum\limits_{x \in X} x)$. It can clearly be computed in linear time and it is dynamic in the sense that it adapts to the size of set, but it has other problems. Assume, that all candidates form one block together, so all elements should be selected by the selection function, the frequencies however won't all be the same, so roughly (since its not the median and the frequencies are not uniformly distributed) half of the elements are discarded.

3.2.2 Selecting by 2Dimk-Means

$$\text{select}_{2Dimk-Means}(C) = \{(v, f) \in C | f \geq t\}$$

With

$$t := \underset{f' \in F_C}{\arg\min}(err_{lower}(F_C, f') + err_{upper}(F_C, f'))$$

and

$$err_{lower}(F_C, f') = \sum_{\{f'' \in (F_C) | f'' < f'\}} |f'' - \text{mean}(\{f'' \in (F_C) | f'' < f'\})|^2$$
$$err_{upper}(F_C, f') = \sum_{\{f'' \in (F_C) | f'' \geq f'\}} |f'' - \text{mean}(\{f'' \in (F_C) | f'' \geq f'\})|^2$$

This selection functions tries to optimally divide the set into 2 partitions by searching for the threshold t that minimizes the mean squared error of the elements in the two partitions ($err_{lower} + err_{upper}$). Which essentially is the classic k-Means problem on a 1 dimensional data set, with $k = 2$.
The naive approach to this would have a runtime of $O(n^2)$. But in case of one dimensional k-Means the algorithms proposed in [?] can solve the problem in $O(kn^2)$ using dynamic programming. However, the ideas from this paper can be used to create a $O(n \log(n))$ algorithm for this special case.
First the data in C is sorted in $O(n \log(n))$. The task changes to finding an index $1 \leq i \leq n$ to split the dataset at. There are $n - 1$ possible splits and we first compute the two means of all possible splits $\{f'' \in F_C | f'' \geq (F_C)_i\}$ and $\{f'' \in F_C | f'' < (F_C)_i\}$. This can be done in linear time using the recursive formula:

$$\mu_i = \frac{f_i + (1 - i)\mu_{i-1}}{i} \tag{3.1}$$

Whereby μ_i is short for the mean of he first i elements. The formula follows directly from the definition of the mean. In the same fashion we can compute mean squared error η_i for the first i elements in linear time.

$$\eta_i = \eta_{i-1} + \frac{i - 1}{i}(f_i - \mu_{i-1})^2 \tag{3.2}$$

Then we have to simply select the split with the smallest summed error (see 5 for pseudo code).

```
**Input**: $C = \{(v_1, f_1), (v_2, f_2), \ldots, (v_n, f_n)\}$
**Output**: $S = \{(v'_1, f'_1), (v'_2, f'_2), \ldots, (v'_m, f'_m)\}$
1 Sort $C$ in non decreasing order;
2 Compute the means $C_{i<t} = \{(v_i, f_i) \in C | i < t\}$ of all thresholds t using (3.1);
3 Compute the means $C_{i \geq t} = \{(v_i, f_i) \in C | i \geq t\}$ of all thresholds t using (3.1) and
  reversed C;
4 Compute the mean squared error $C_{i<t}$ of all thresholds t using (3.2);
5 Compute the mean squared error $C_{i \geq t}$ of all thresholds t using (3.2) and reversed C;
6 **return** $C_{i \geq t}$ for the t that minimizes the summed mean square error;
```

Algorithm 5: computation of select$_{2Dimk-Means}(C)$

3.2.3 Selecting by Graph Neighbourhood

$$\text{select}_{gnn}(C) = \{(v, f) \in C || \text{n}(v) \cap V_C| \geq t\}$$

Initially, the goal was to include an outlier detection algorithm, that works on the graph, but most of the studied algorithms where not applicable on this case or where far too complex for these scenarios. The chosen solution is a relative lightweight algorithm, that considers how many edges connect a node to the subgraph defined by V_C. When the edge count is less than a threshold t it is disregarded. The threshold t is set to the $block\_minsize$ parameter of the algorithm.

Additionally, a method using a top down approach was explored. Instead of disregarding nodes with few nodes it would select k nodes that have many connections first. The process is rarely successful, when selecting on its own, so it wasn't explored on a stand alone basis.

One speciality of this selection methods is that it does not use the frequency, which on hand may look like wasted effort, because we disregard known information, on the other hand does it mean that we can use the function with candidate sets, that come without these informations, for example the partitions (section 3.4) or the clusters (section 3.3).

The intersection operation used by these selection functions, is implemented in such a way that it runs in linear time, by the compiler. This is possible by using sorted lists as representation for sets. Overall this results in a runtime of $O(n^2)$

3.2.4 Selecting by Composition

Given a set of selection functions $\mathfrak{F} = \{\text{select}_1, \text{select}_2, \dots\}$.

$$\text{select}_{comp}(C) = \{\ S |\, \text{select} \in \mathfrak{F} : \text{select}(C) = S\}$$

One of the big advantages of all these selection functions is, that they are fast. Therefore we don't have to limit ourself by using only one and can simply use all of them. The composition algorithm works by computing all candidate sets $S_1, S_2, \dots$using all the introduced selection functions and adds as blocks all those, that form sufficiently large blocks. The advantage of this method is, that the different selection algorithms are good for detecting different kinds of blocks. The graph neighbourhood based selection works good on candidate

Another approach would be to apply selection functions one after another (i.e. $S = \text{select}_1(\text{select}_2(\dots \text{select}_k(C))))$. This is possible, because input data and output data have the same structure. However, this wasn't explored further, because the number of combinations are too large, to explore it in a satisfying manner, given the time constraints of this work. It could still be considered for future work.

3.3 Variant: Clustering

There are two reasons, why clustering was considered during this work. At first, it was considered for partitioning the data, to reduce the overall runtime, since some of the algorithm used have polynomial runtime, but after the initial experiments didn't yield good partitions, this direction wasn't pursued further (see section 3.4). Secondly, it was considered as an alternative to the walk part of the algorithm, because in a abstract way the RANDOMWALK algorithm is nothing, but a partition algorithm.

A clustering algorithms gets a data set as input and its task is to find sets of similar data, whereby the first challenge is to define what similar data means. A geometric interpretation of the a tensor as 3D-object (i.e. non-zero entries are points in a coordinate system) doesn't help in finding blocks, because geometrically close entries, don't have to be part of the same block. As before, our tool to define which nodes are close is the FiberGraph. Graph Clustering is a well researched field and there is a wide variety of possible solutions. In this work we first explored several of the more common clustering algorithms based on k-Means and Spectral Clustering with different Similarity Measures. These didn't produce workable solutions and are only reported for completeness sake. The MCL algorithm was much more successful and in its functionality it is very similar to the walk algorithm.

3.3.1 k-Means based Clustering

Now that we have the FiberGraph $FG(V, E)$ we have to define our similarity measure. From the FiberGraph we get the adjacency matrix A.

$$(A)_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

In this section we explore, how common clustering algorithms perform when clustering based on A.

The k-Means algorithm is one of the oldest and most well known clustering algorithms [Mac67]. The implementation used in this work was based on Lloyd's algorithm [Llo82], which would have been replaced by newer, improved variants, if the initial experiments had been more promising. In essence Lloyd's algorithm works by first assigning each data entry to a random cluster and computing the cluster means. Then iteratively it would reassign all data points to the clusters with the nearest centroid and recompute the centroid afterwards, till the algorithm converges.

The first challenge hereby is to select an appropriate similarity function. One way to define the similarity between two nodes is to take the euclidean distance of the associated rows in A, but since we have binary data there are a lot of possible similarity measures (e.g. the Jaccard similarity or, pearson correlation, cosine similarity,... [Fin05]). It would also be possible to use the length of the shortest path in the graph as similarity measure based on the all-pairs, shortest path problem [Sei95]. However, when using alternative similarity measure, the computation of the centroids may become significantly more complex ([Lei06]). In the future, it could be explored how clustering performs using these different measure, as using the euclidian distance might have been the main flaw during this part of the work.

Beyond the default k-Means algorithm, this work also explored the use of the k-Means++ algorithm [AV07] and the spectral clustering algorithm. More concretely the fast spectral clustering algorithm [YHJ09], which essentially clusters k-Means centroids.

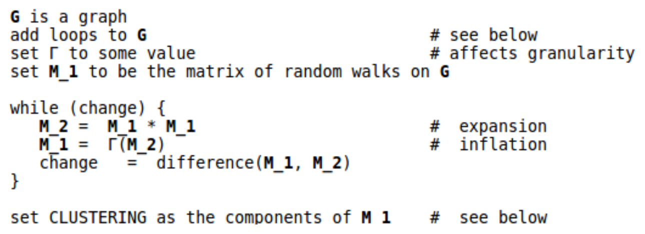
All these approaches had in common, that they weren't able to able to detect dense blocks within the original tensor data. And for partitioning they produced very unbalanced partitions, where one partition contained most of the entries. As the k-Means (which is the basic of all the explored clustering algorithms) algorithm needs a parameter k that determines the number of clusters, there where two kinds of experiments. The first kind was to set k to the rank of the data, which is approximately known for synthetic data (modulo some added noise). The other kind was to set k to a very high value. The results, were so bad, that there isn't really a point to reporting detailed results, as blocks where only rarely found, since the clusters where either too small or not dense enough. As noted before, the approaches discussed here are flawed, but they weren't improved upon, because for both problems there where better alternatives. For clustering this is the MCL algorithm.

3.3.2 Clustering by MCL

The MCL algorithm is a graph clustering algorithm, that works surprisingly similar to the RANDOMWALK algorithm [VD00]. It stands for Markov Cluster Algorithm and is based on the graph flow problem. The official documentation describes the algorithm as seen in Figre 3.2.

The basic idea is the same as with walk algorithm: clusters are subgraphs with many edges between the members and random walks starting within a cluster only seldom leave the cluster. Therefore, the algorithm simulates random walks in the graph based on Markov chains. The matrix M is stochastic Matrix.

Figure 3.2: The pseudo code of MCL as depicted in the official documentation



```
G is a graph
add loops to G                                  # see below
set Γ to some value                             # affects granularity
set M_1 to be the matrix of random walks on G

while (change) {
   M_2 =  M_1 * M_1                             #  expansion
   M_1 =  Γ(M_2)                                #  inflation
   change   =  difference(M_1, M_2)
}

set CLUSTERING as the components of M 1     #  see below
```

Definition 3.1. Stochastic Matrix A n-by-m matrix M is a column stochastic matrix, iff:

- $(M)_{ij} \in [0,1]$ for all $i \in [n]$ and $j \in [m]$
- $\sum(M)_{ij} = 1$ for all columns $k \in [m]$

In short, these matrices can be used to describe a Markov chain, which is a sequence of state changes, where each state only depends on its predecessor. A random walk in a graph $G(V,E)$ can be described by such. The position of the walker is a probability distribution over V, because since the walk is random he can be potentially be at multiple vertices v_i. For a graph with n vertices, this can be represented as a normalized (i.e. length 1) vector $\vec{\boldsymbol{s}}$. We can derive a stochastic matrix $\boldsymbol{M}$ from $G(V,E)$, that describes with what probability a random walk would go from v_i to neighbour v_j, when the direction is decided uniformly at random, given the adjacency matrix $\boldsymbol{A}$ of $G(V,E)$

$$(\boldsymbol{M})_{ij} = \begin{cases} \frac{1}{|(\boldsymbol{A})_{:j}|} & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

This is clearly a stochastic Matrix given the definition 3.1. The step from one state $\vec{(s)}_i$ to another state $\vec{\boldsymbol{s}}_{i+1}$ can be described by the matrix multiplication $\vec{\boldsymbol{s}}_i\boldsymbol{M}$. More extensive foundation can be found in section 3.7.

The algorithm uses two operations on the stochastic Matrix M describing the graph:

- **Expansion:** Corresponds to squaring the matrix $\boldsymbol{M}_1^2 = \boldsymbol{M}_1 * \boldsymbol{M}_1$, with normalization to ensure $\boldsymbol{M}_1^2$ is a stochastic matrix. If $(\boldsymbol{M})_{i,j}$ represents the probability of going from v_j to v_i during a random walk, $(\boldsymbol{M}_1^2)_{ij}$ represents to probability of going from v_j to v_i over another vertex and the more expansions one adds, the longer are the potential paths. Therefore the name expansions. To not have all probabilities get too small disappear, as the walk gets longer the Inflation phase is important.

- **Inflation** The main purpose of the inflation phase is to increase the probability of a random walk staying within a cluster. This happens through the following operation, given a parameter $r > 1$:

$$\text{Inflation}((\boldsymbol{M})_{ij}, r) = \frac{((\boldsymbol{M})_{ij})^2}{\sum\limits_{i' \in [n]} ((\boldsymbol{M})_{i'j})^2}$$

This two operations are applied to the matrix $\boldsymbol{M}$, till they no longer change the matrix. This type of convergence is a common property of Markov chains given certain criteria [MN88]. When it converges vertices in clusters have higher values, than others that are not. These clusters can then be extracted from $\boldsymbol{M}$. For more thorough explanations, see [VD00].

3.4 Variant: Partitioning

One observation of the Walk'n'Merge algorithm is, that most parts scale non linear in terms of runtime with the size of input tensor (mostly the merge part). This leads to a strategy, that reduce the size of the input, to reduce the overall runtime, because a linear decrease in the size of the input leads to a polynomial decrease in the overall runtime (In theory at least).

One way to do this is by preprocessing, the data as described in section 3.1.3. In this step we can eliminate some nodes, that are no possible candidates for blocks. For example because they don't have any neighbours in the FiberGraph. Another strategy is to partition the data into independent segments, whose blocks at best don't overlap, so that the Walk'n'Merge algorithm can run independently on them. This allows for some trivial parallelization of the whole process and since all the segments are significantly smaller (when using good partitioning) this can reduce the overall runtime (A consequence of the binomial formula).

The initial experiments relied on classical clustering algorithms. These ended in general with bad results, mostly because the partitions ended up being heavily imbalanced. This line of thought was discontinued, because a better way was found. One can observe, that most dense blocks have to come from the same connected subgraph of the FiberGraph.

Intuitively, this can be understood as follows. Assume, you have a tensor $\boldsymbol{\mathcal{X}}$ with a FiberGraph $FG(V, E)$ that has two unconnected subgraphs $FG(V_1, E_1)$ with n vertices and $FG(V_2, E_2)$ with m vertices. That means, there exists edges E $(v, v') \in E$ such that $v \in V_1$ and $v \in V_2$ or the other way around. Therefore, there is no vertex in V_1 that is connected by a common fiber with a vertic in V_2. That again means, that we can rearrange the tensor into a block format, such that entries represented by vertices in V_1 are in one corner and the ones presented by V_2 in the other (see 3.3). Now, whenever you have a block from one sub tensor and add a non-zero entry from another tensor, you add a lot of zero entries. In the sketch these are from the red marked entries.

I use a breadth-first-search algorithm to find the connected components ([Ski08]). See algorithm 6 for a pseudo code version. Essentially take a random node from V and add its neighbours and its neighbour's neighbours and so forth to the partition, till we no longer encounter any new nodes. A important aspect is line 14, where we decide, whether we actually add the node set as partition or not. This allows us to use some pruning techniques to remove partitions that cannot contain blocks before wasting computational resources on them. More on that in the next section.

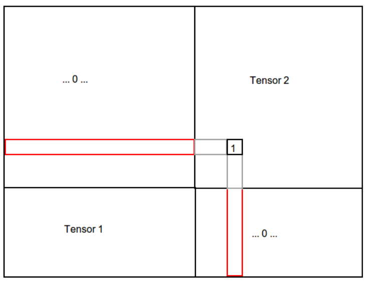


Figure 3.3: A sketch illustrating the density loss, when merging blocks of different partitions using a 2 dimensional tensor as example

```
Input: tensor $\mathcal{X}$
Output: partitions $\{P_1, P_2, \dots\}$
1  create FiberGraph $FG(V, E)$ from $\mathcal{X}$;
2  $partitions$ = an empty set;
3  while $V$ is not empty do
4      $startNode$ = random node from $(V)$;
5      remove $startNode$ from $V$;
6      $closedNodes$ = new Set();
7      $openNodes$ = new Set (unique elements) with $startNode$;
8      for node $v$ in $openNodes$ do
9          insert $v$ into $closedNodes$;
10         for neighbours $v_n$ of $v$ do
11             if $v_n$ not in $closedNodes$ then
12                 insert $v_n$ into $openNodes$;
13         remove $closedNodes$ from $V$;
14     if $closedNodes$ is a Block candidate then
15         Insert $closedNodes$ into $partitions$;
16 return $partitions$;
```

Algorithm 6: Pseudo code for the partitioning algorithm

3.4.1 Pruning

There are three constraints, that blocks found by Walk'n'Merge have to fulfill, based on the input parameters. For a Block $B = (X, Y, Z)$ of tensor $\mathcal{X}$ these are: *block_min_size*, *block_min_vol* and *block_min_density*. (see Section 3.1.5)

The parameters *block_min_size* and *block_min_vol* allow one to control the shape of blocks. With low min size and high volume, the found blocks will be most likely quite lengthy, while for higher min size the volume plays nearly no role. The density has the property, that it allows the user to control the false positive rate of the resulting blocks, since very dense blocks when used for reconstructing the tensor add nearly no new ones to the reconstructed tensor. This comes with the disadvantage, that (large) high density blocks are harder to find, since they are rare in a typical tensor.

The partition algorithm uses a number of pruning criteria. We are given a set of candidate nodes $C = v_1, v_2, \dots, v_w$. $\mathtt{B}_C = (X_C, Y_C, Z_C)$ is the block defined by C, with

$$X_C = \bigcup_{v \in C} v.x$$
$$Y_C = \bigcup_{v \in C} v.y$$
$$Z_C = \bigcup_{v \in C} v.z$$

- **Size criteria**

  $$|X_C| > block\_min\_size \text{ and } |Y_C| > block\_min\_size \text{ and } |Z_C| > block\_min\_size$$

  If the overarching Block isn't large enough there won't exist subsets $X'_C \subset X_C$, $Y'_C \subset Y_C$ and $Z'_C \subset Z_C$ with $C' \subset C$, such that (X'_C, Y'_C, Z'_C) is a dense block.

- **Volume criteria**

  $$|X_C||Y_C||Z_C| > block\_min\_vol$$

  the argumentation is the same as for the size criteria. It should be noted, that a $block\_min\_vol$ value of less than $(block\_min\_size)^3$, makes no sense, as it would be already covered by the size criteria.

- **Block density criteria**

  $$\text{maxDensity}((X_C, Y_C, Z_C)) < block\_min\_density$$

  We can bound the max density of a minimal sized sub-block of a given block using the theorem 1.12 and a recursive calculation. The bound is rather benign, as it goes by the implicit assumption, that removed entries contain nearly no non-zero entries. This could possibly be further improved. As it is, it removes mostly extremely sparse blocks.

Later, in Chapter 5, I will discuss some techniques to further split partitions without breaking apart important structures (too often).

3.5 Variant: Hash Merge

Considering also the huge divergence in terms of runtime between BlockMerge and RandomWalk the focus of this whole work was more on the walk part, because if you could make the walk good enough, that you no longer require the merge part the algorithm would be significantly faster.

Still the merge part is useful, as it is mainly responsible for the good reconstruction errors of the Walk'n'Merge algorithm. Problematic is however, that the blocks found by MonoMerge are all rather small ($2 \times 2 \times n$) and the SubsequentMerge is actually only rarely able to merge blocks. This leads to many small blocks in the result. This means, that the from Walk'n'Merge resulting decomposition has a very high rank, which reduced its overall usefulness.

For the SubsequentMerge we only want to consider blocks, that are similar in some sense, because you want to reduce the amount of possible block pairs as much as possible. Already the original Walk'n'Merge considered only blocks, that share coordinates. Furthermore, I examined the use of Local Sensitive Hashing (LHS) [RU11]. Common (cartographic) hashing schemes try to avoid collisions, while LHS schemes are designed in a way, that similar elements are hashed to the same bucket. What I use is the min-wise independent permutations locality sensitive hashing scheme, short MinHash [Bro97]. Given two sets A and B and a MinHash function h_{min}, the following holds:

$$P(h_{min}(A) = h_{min}(B)) = J(A, B)$$

Whereby P is simply the probability, and J is the Jaccard similarity, which is a common similarity function for sets, defined by:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Given a hash function h, that maps elements of the sets to integers, the corresponding h_{min} is defined as the element which minimizes h.

$$h_{min}(A) = \arg\min_{a \in A} h(a)$$

There exist Variants using multiple hash functions, I however used a variant with just one. Using multiple hash functions could be considered for future works, as it helps approximating the Jaccard similarity better. The SubsequentMerge was adapted, by only considering elements for merges, that where mapped to the same value by the MinHash. The resulting algorithm is called HashMerge.

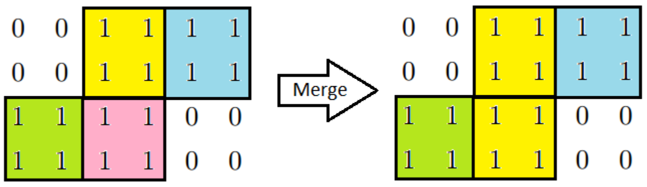


Figure 3.4: The squares mark different blocks in a 2-way tensor, which we would like to merge. Ideally, you would merge yellow with blue and green with pink, but in this case we merged yellow and red first and no further merges are possible (If you want monochromatic blocks). Instead of the optimal 2 blocks you remain with 4 blocks

3.6 Variant: All Merge

One aspect of SubsequentMerge is, that the algorithm does not really consider all possible merges. Potentially, this leads to suboptimal blocks as seen in figure 3.4. The problem is, that considering all possible merges increases the runtime significantly. Nonetheless, I experimented with this idea a bit to see by how much.

The allSubMerge algorithm changes the subsequent merge algorithm by not removing blocks, that where merged or where no merge candidate was found. I call the BlockMerge variant with allSubMerge, simply AllMerge.

3.7 Variant: Simultaneous Walk

In this section I discuss a replacement for the random walk phase of the algorithm that gets very similar results without actually walking. The process is inspired by the PageRank algorithm [PBMW99] [SG12] and also similar the MCL algorithm from Section 3.3.

The original RandomWalk produced a list of frequencies, that indicated how often a node was visited. In SimWalk these are replaced by a probability distribution that indicates, with what probability the walker is at a certain node. For a tensor $\boldsymbol{\mathcal{X}}$ with FiberGraph $FG(V, E)$ we denote this by a vector $\vec{\boldsymbol{s}}$ of length $|V|$. Initially, a start node is selected like in the RandomWalk algorithm. The probability that the walker at the start node is 1, all other probabilities are 0. Given the nodes $V = \{v_1, v_2, \ldots, v_n\}$ and start node v_{start} it the start sate is defined by:

$$(\vec{\boldsymbol{s}}_{start})_i = \begin{cases} 1 & v_i = v_{start} \\ 0 & otherwise \end{cases}$$

RandomWalk jumps back to an already explored state when the walk has reached a length defined by the *wal_len* parameter. Similarly, the page rank gives the random walker the ability to jump to a random node in the graph with a probability ϵ. I use this mechanic to model the backtracking of the original RandomWalk.

The (simplyfied) power method of the PageRank [ANTT02] works as follows. Given a Graph $G(V, E)$, intial state $\vec{\boldsymbol{s}}_{start}$ and jump probability ϵ, we first compute the transition matrix $\boldsymbol{T}$.

$$(\boldsymbol{T})_{ij} = \begin{cases} \frac{1}{L(v_j)} & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

where $L(v_j)$ is the number of edges outgoing from v_j ($|\operatorname{n}(v_j)|$). This matrix contains the probability that a walker goes from neighboring node v_i to v_j at position $\boldsymbol{T})_{ij}$, when the walker decides uniformly at random. Finally, we compute till convergence:

$$\vec{\boldsymbol{s}}_{i+1} = (1 - \epsilon)\vec{\boldsymbol{s}}_i T + \frac{\epsilon}{N}\mathbf{1}^N$$

where N is the number of nodes, $\vec{\boldsymbol{s}}_0 = \vec{\boldsymbol{s}}_{start}$ and $\mathbf{1}^N$ is the vector of ones with length N. This last part simulates the probability, that we jump to a random node, by adding $\frac{1-\epsilon}{N}$ to all states. This is the part we can manipulate to simulate RandomWalk partially. The more often a node was visited during the walk, the higher is the probability of RandomWalk to jump back to it. For SimWalk these probabilities are explicitly computed as the states of the algorithm. This leads to the setp formula:

$$\vec{\boldsymbol{s}}_{i+1} = (1-\epsilon)\vec{\boldsymbol{s}}_i T + \epsilon \vec{\boldsymbol{s}}_i \tag{3.3}$$

The main difference between SimWalk and RandomWalk is, that SimWalk can jump back at any step and RandomWalk only at certain steps. A short pseudo code version of SimWalk in presented as Algorithm 7.

```
Input: tensor $\boldsymbol{\mathcal{X}}$,density $d$,minimum frequency $p$
Output: blocks $(X_1, Y_1, Z_1)$, $(X_2, Y_2, Z_2)$,...
1  create FiberGraph $FG(V, E)$ from $\boldsymbol{\mathcal{X}}$;
2  $(T)$ = Transition Matrix of $FG$;
3  while $V$ is not empty do
4      $v$ = random node from $V$;
5      Compute probability distribution $\vec{\boldsymbol{s}}$ using the power method with equation 3.3;
6      empty block $(X, Y, Z)$ for $v_i \in V$ do
7          if $(\vec{\boldsymbol{s}})_i > p$ then
8              $(X, Y, Z)$.add$((v_i.x, v_i.y, v_i.z))$;
9      $V.remove(convexHull(\mathrm{ten}(X, Y, Z)))$;
10     if density of ten$(X, Y, Z)$ in X $> d$ then
11         blocks.add$((X, Y, Z))$;
12     return blocks;
```

Algorithm 7: Pseudo code for the basic SimWalk algorithm

Not listed in the short explanation are potential normalization steps. As well as pruning steps, where states with very small values are set to 0, because they are mostly irrelevant for the computation. Also the selection function in line 7 can be replaced by the ones discussed in Section 3.2

3.8 Variant: Solving the quasi Clique Problem using LP

Another problem in computer science, that show parallels to the problem of selecting subraghs of a FiberGraph as block candidates, is the quasi clique problem. A clique C of an undirected graph (e.g. FiberGraph) $G(V, E)$ is a set of nodes, where all nodes are connected by edges $C = \{c | c \in V \text{ and for all } c' \in C \text{ with } c \neq c' \text{ exists } (c, c') \in E\}$ (in an undirected graph $(c, c') \in E \implies (c', c) \in E$) The clique problem is finding the largest clique of a graph.

The quasi clique problem is similar, just that it allows for some edges to be missing. A full clique of n nodes has $\binom{n}{2}$ nodes. A γ-quasi clique C has at least γ edges of a full clique. meaning:

$$\frac{|\{e | (c, c') \in E \text{ and } c \in C \text{ and } c' \in C\}|}{\binom{|C|}{2}} \geq \gamma$$

Note, that finding a maximal γ-quasi clique is a NP-hard problem. Finding the largest γ-quasi clique has been studied in the path and [PVBB13] suggests a mixed integer program (MIP) for solving it.

Let $G(V, E)$ be a graph and $\boldsymbol{A}$ its adjacency matrix. Then a MIP for finding the largest γ-quasi clique is given by:

$$\begin{array}{llll}
\text{maximize} & \sum_{i=1}^{n} x_i & & \\
\text{subject to} & \sum_{i=1}^{n} \sum_{j=i+1}^{n} (\gamma - (\boldsymbol{A})_{ij}) w_{ij} & \leq 0, & i = 1, ..., n \\
& w_{ij} \leq x_i, w_{ij} \leq x_j, w_{ij} \geq x_i + x_j - 1 & j > i = 1, ..., n & \\
& w_{ij} \geq 0, x_j \in \{0, 1\}, & j = 1, ..., n &
\end{array}$$

Which has $\frac{n}{(n-1)}$ variables and $\frac{3}{2}n(n-1) + 1$ constraints. There do exist different MIPs in the paper, but I won't present these here. Please, refer to the original paper for more details.

3.8.1 Extended Graph

One has to keep in mind, that subgraphs representing blocks are not cliques or even quasi cliques with large γ values in the FiberGraph, because an entry of the block is just connected with entries with which it shares fibers. In a $n \times m \times l$ monochromatic block (X, Y, Z) those are $n + m + l$ entries (disregarding the fact that there are no self loops). resulting in an edge density of $\frac{n+m+l}{nml}$. The extended graph was designed to give blocks a higher edge density. Thereby I differentiate between two kinds of extended FiberGraph.

The first Variant $FG^2(V, E)$ could be referred to as the SliceGraph and given a FiberGraph $FG(V, E)$ is defined as follows:

Definition 3.2. SliceGraph The first extension of a FiberGraph $FG(V, E)$, the SliceGraph $FG^2(V, E^2)$ has additional edges defined by

$$E^2 = E \cup \{(v_1, v_2) | \text{ the exists } v' \in V \text{ s.t. } (v_1, v') \in E \text{ and } (v', v_2) \in E \text{ with } v_1 \neq v_2\}$$

In short, it connects nodes, that share a slice and have a common neighbour within the slice. The edge density of monochromatic within the SliceGraph is therefore approximately $\frac{nm+ml+nl}{nml}$ with some duplicates. You can go also one step further and connect all edges, that are reachable in the 3 steps in the FiberGraph. I call this the TensorGraph, although the name can be a bit misleading.

3.8.2 Quasi Clique based Block Detection Algorithm

The MIP can be solved using of the shelve solver and in this case I use gurobi. An algorithm, that computes a CP-decomposition of fixed rank r can simply compute the largest quasi clique of the graph and remove it r times.

```
Input: tensor $\boldsymbol{\mathcal{X}}$,density $d$,minimum density $\gamma$, rank $r$
Output: blocks $(X_1, Y_1, Z_1)$, $(X_2, Y_2, Z_2)$,...
1 create FiberGraph $FG(V, E)$ from $\boldsymbol{\mathcal{X}}$ // or an extended variant
2 blocks = empty list;
3 forall the $i \leq r$ do
4     $C$ = largest $\gamma$-quasi clique in $FG(V, E)$;
5     removed nodes in $C$ from $FG(V, E)$;
6     Add block defined by the convex hull of $C$ to blocks;
7 return blocks;
```

Algorithm 8: Pseudo code for the qc_cpDecomp algorithm computing a CP-decomposition based on the quasi clique problem

Chapter 4

Evaluation

In this section I evaluate the different method introduced in chapter 3. First I will describe my experimental framework in section 4.1. After that follow the individual evaluations. I don't report all results exhaustively and concentrate on the interesting ones, e.g. when an experiment only marginally deviates from the baseline, I will say as such and not report the same graph again.

4.1 Experimental Methodology

Most of the strategies were evaluated on synthetic data, because getting large amounts of real world data to get statistical relevant results is very hard. An overview on the tensor generation is in section 4.1.1. Another overview over the different datasets is in section 4.1.2.

For the evaluation I primarily used the *data_scale_size* and *data_scale_rank* datasets, that contain 50 tensors each from 10 different kinds of tensors. To factor in the randomness of the algorithms each experiment was repeated 5 times, so that each graph that is presented in this work is the result of 250 runs, with 25 runs per different kind of tensor. I also report the 99% confidence intervals for each experiment.

The runtime is the most important metric for these experiments, but the reconstruction error is also important. Given a tensor $tenX$ and a reconstructed tensor $\boldsymbol{\mathcal{Y}}$, we have

Table 4.1: Error Matrix

actual / predicted	1	0
1	true positive (tp)	false negative (fn)
0	false positive (fp)	true negative (tn)

several classical error metrics from machine learning and data mining (see table 4.1). The Accuracy or reconstruction error ($= \frac{tp+tn}{tp+tn+fp+fn}$) alone is for the human observer a difficult metric to understand, because the zero entries far outweigh the positive ones. In fact most accuracy measures reach values of 0.9999 or better, even when comparing to a trivial tensor without any non zero entries. The number of true negatives is typically very high and the number of false positives relatively low, because all the algorithms discussed wouldn't add a non-dense tensor and in most cases the density d is kept high (> 0.8) and for the result roughly holds $fp < (1-d)tp$. In the end, I choose the true positive rate($tpr = \frac{tp}{tp+fn}$) also known as Sensitivity as a quality metric, as it measures the critical aspects best.

Another aspect, that I would like to measure is the ability of the results to be used for a CP-decomposition of fixed rank. The best way to do this is to actually construct a CP-decomposition. Since, this is actually a non trivial problem I defined for my cases an algorithm to construct a canonical CP-decomposition from a set of blocks.

```
Input: in_blocks $\{(X_1, Y_1, Z_1), (X_2, Y_2, Z_2), \dots, (X_m, Y_m, Z_m)\}$, rank $r$
Output: out_blocks $\{(X_1, Y_1, Z_1), (X_2, Y_2, Z_2), \dots, (X_r, Y_r, Z_r)\}$
1 if $m > r$ then
2     return in_blocks // No selection needed
3 out_blocks = new collections;
4 forall the $r$ iterations do
5     Add to out_blocks the block with the highest amount of new entries;
6 return out_blocks
```

Algorithm 9: Pseudo code for constructing the canonical CP-decomposition from a set of blocks

A bit complicated is just to calculate this highest amount of new entries. One block (X, Y, Z) alone has $|X||Y||Z|$ new entries. A block that overlaps with an existing block (X', Y', Z') has $(X, Y, Z) - |\text{overlap}((X, Y, Z), (X', Y', Z'))|$ new entries, whereby the overlap of the two blocks can be calculated by $(X \cap X', Y \cap Y', Z \cap Z')$. Calculating the number of new entries gets increasingly complicated, when the number of overlapping blocks increases further, as one has to keep care to only remove entries once from the calculation.

With this I define the top-k true positive rate (tpr_k) as the true positive rate of the rank k canonical CP-decomposition.

All experiments were executed with a relative harsh timeout of 30 minutes, experiments taking longer are interrupted, since the goal of this work is to find fast algorithms and the 30 minutes indicate, that the algorithm won't scale good enough for the given datasets.

4.1.1 Tensor Generation

The tensor generation is very similar to the approach of the original Walk'n'Merge paper. It is based on the hypothesis, that tensors are spanned by a set of core blocks, with some noise. The pseudo code is provided by algorithm 10.

```
Input: $size\_x, size\_y, size\_z, rank, destr\_noise \in [0,1]$, $constr\_noise \in [0,1]$
Output: tensor $\boldsymbol{\mathcal{X}}$
1 $d' = \text{subdensity}(d)$;
2 generate Boolean $size\_x$-by-$rank$ matrix $\boldsymbol{A}$ with density $d'$;
3 generate Boolean $size\_y$-by-$rank$ matrix $\boldsymbol{B}$ with density $d'$;
4 generate Boolean $size\_z$-by-$rank$ matrix $\boldsymbol{C}$ with density $d'$;
5 $\boldsymbol{\mathcal{X}} = \bigvee_{i=1}^{rank} (\boldsymbol{A})_{:i} \boxtimes (\boldsymbol{B})_{:i} \boxtimes (\boldsymbol{C})_{:i}$;
6 apply constructive Noise to $\boldsymbol{\mathcal{X}}$;
7 apply destructive Noise to $\boldsymbol{\mathcal{X}}$;
8 return $\boldsymbol{\mathcal{X}}$;
```

Algorithm 10: Pseudo code for the rank based tensor generation

The input arguments are the 3 sizes, the Boolean rank r and the density of the tensor, as well as the constructive and the destructive noise parameters. The algorithm starts by generating an ideal tensor by first generating the 3 factor matrices with density d' (from equation 8), that define $\boldsymbol{\mathcal{X}}$. In a sense, this is a inverse CP-decomposition. The result is a tensor, that has a perfect decomposition into r rank-1 subtensors. Since, perfect tensors with low rank will be very rare in natural data, I try to emulate real world data, by applying some noise. The two kinds of noise are the *destructive* noise and the *constructive* noise. The destructive noise sets a percentage of non-zero entries to zero, while the constructive noise sets zero entries to one.

The density d' of the three matrices $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{C}$ is estimated by

$$d' = (1 - (1-d)^{\frac{1}{r}})^{\frac{1}{3}} \tag{4.1}$$

Assume the three matrices have a density of d'. That means, that the probability that a entry is 1 is also d'. An entry (i, j, k) of the tensor is one, when there exists a column c, such that $(\boldsymbol{A})_{i,c} = 1$, $(\boldsymbol{B})_{j,c} = 1$ and $(\boldsymbol{C})_{k,c} = 1$. The probability that this is the case for a given c is d'^3, assuming those events are independent. Consequently the probability, that this is not the case is $(1 - d'^3)$. The probability, that atleast one such c for given i, j, k exists is easiest calculated using the probability of the complementary event, resulting in:

$$d = 1 - ((1 - d'^3)^r)$$

which yields the equation 8, when solving the equation for d'.

Table 4.2: Table detailing the tensors used for the synthetic dataset data_scale_size

name	size_x	size_y	size_z	density	constr. noise	destr. noise	rank
gen1	1000	1000	1000	0.000001	0.0000001	0.1	25
gen2	1330	1330	1330	0.000001	0.0000001	0.1	25
gen3	1660	1660	1660	0.000001	0.0000001	0.1	25
gen4	2000	2000	2000	0.000001	0.0000001	0.1	25
gen5	2660	2660	2660	0.000001	0.0000001	0.1	25
gen6	3330	3330	3330	0.000001	0.0000001	0.1	25
gen7	4000	4000	4000	0.000001	0.0000001	0.1	25
gen8	5330	5330	5330	0.000001	0.0000001	0.1	25
gen9	6660	6660	6660	0.000001	0.0000001	0.1	25
gen10	8000	8000	8000	0.000001	0.0000001	0.1	25

4.1.2 Datasets

Synthetic Data

There are two datasets generated for the evaluation. Each of these sets contains 10 kinds of tensors, with 5 versions of each. They are designed in such a way, as to explore how the algorithm scales with increased tensor size and increased tensor rank. The datasets, are kept quite sparse, because most real world data is sparse. Compared to some of the sparser real world datasets, they are still quite dense.

data_scale_size

The generation parameters for the tensors *data_scale_size* are shown in table 4.2. the tensors in this set are of different size, whereby *gen1* tensors are the smallest with roughly 1000 entries and *gen10* tensors the largest with half a million entries. The size increases is cubic. The noise values are kept relatively low, destructive noise is at 10 percent and the constructive noise is chosen so that about 10% of extra ones appear. The rank is fixed at 25, which is rather large for the smaller tensors (blocks a rarely larger than $2 \times 2 \times 2$), but quite small for the larger ones (on average the blocks for *gen10* have a size of $70 \times 70 \times 70$).

data_scale_size

The main difference, to the *data_scale_size* dataset is, that the rank is scaled (linearly) up, such that *gen10* has 10 times he rank of *gen1*, while the size stays the same. In terms of size, they have about as many entries as the *gen8* tensors from the earlier set.

Realworld Data

I had several different real world datasets available and of those I choose a small selection in hope that they best model applications of boolean tensor decomposition in praxis.

Table 4.3: Table detailing the tensors used for the synthetic dataset data_scale_rank

name	size_x	size_y	size_z	density	constr. noise	destr. noise	rank
gen11	3000	3000	3000	0.00001	0.0000001	0.1	20
gen12	3000	3000	3000	0.00001	0.0000001	0.1	40
gen13	3000	3000	3000	0.00001	0.0000001	0.1	60
gen14	3000	3000	3000	0.00001	0.0000001	0.1	80
gen15	3000	3000	3000	0.00001	0.0000001	0.1	100
gen16	3000	3000	3000	0.00001	0.0000001	0.1	120
gen17	3000	3000	3000	0.00001	0.0000001	0.1	140
gen18	3000	3000	3000	0.00001	0.0000001	0.1	160
gen19	3000	3000	3000	0.00001	0.0000001	0.1	180
gen20	3000	3000	3000	0.00001	0.0000001	0.1	200

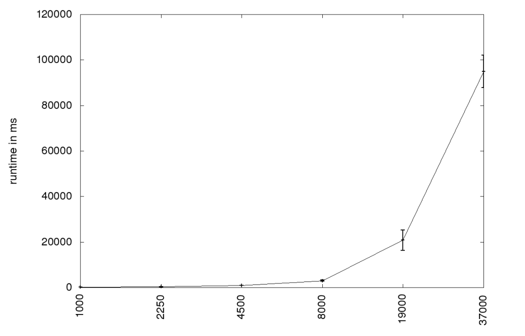
The main factor being that they where the largest of the analysed datasets.

- **TracePort**: anonymized passive traffic traces with fibers: source IP, destination IP, port number
- **Facebook**: who posted on whose wall with fibers: poster, wall owner, week

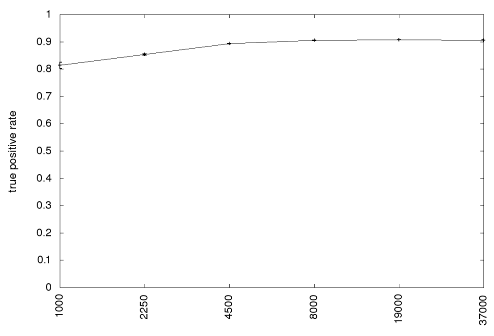
These sets are best characterized, by having a very high rank with rather small dense blocks, which makes them stand apart from the artificial generated datasets.

Hardware

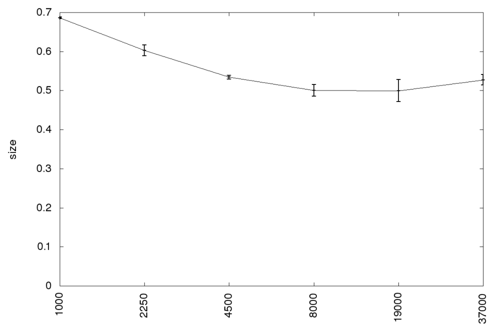
The experiments where run on 64bit linux server, with 16 cores (32 with hyper threading) and Intel(R) Xeon(R) CPU E5530 @ 2.40GHz processors with roughly 50 Gb main memory.



(a) Runtime of Walk'n'Merge in relation to the size of the tensor



(b) Tpr of Walk'n'Merge in relation to the size of the tensor

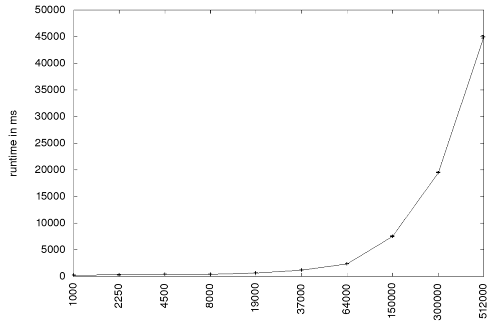


(c) top 20 tpr of Walk'n'Merge in relation to the size of the tensor

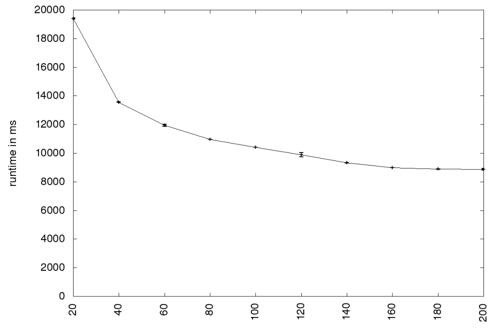
Figure 4.1: The performance of Walk'n'Merge

4.2 Baseline Experiments

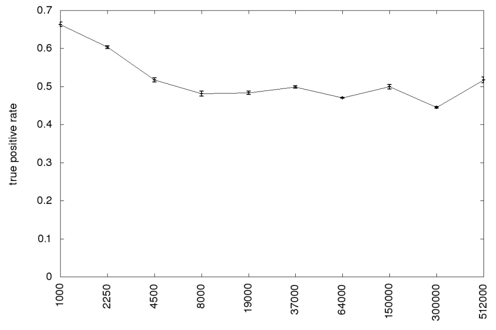
First I will give an overview on the performance of Walk'n'Merge as emulated by Walk'n'Merge++. The algorithm was run using default parameters on the two synthetic datasets from the previous section. The primary results can be seen in figure 4.1. With the timeout of 30minutes a lot of runs where actually terminated early and aren't depicted in the graphs. But for those, that are depicted the most notable thing is true positive rate of nearly 90%, which is about the best one could expect considering, that the tensors contain 10% constructive noise (relative to the number of entries). The top20 tpr is still rather good with slight decline with increasing size of the tensor and of course with increase of the tensor rank.



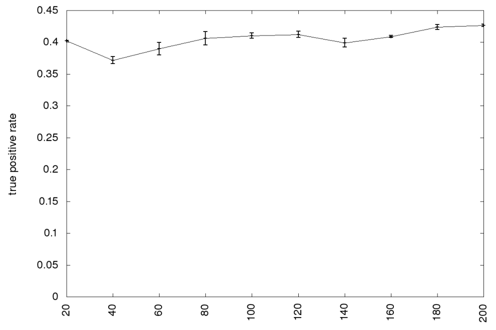
(a) Runtime of RandomWalk in relation to the size of the tensor



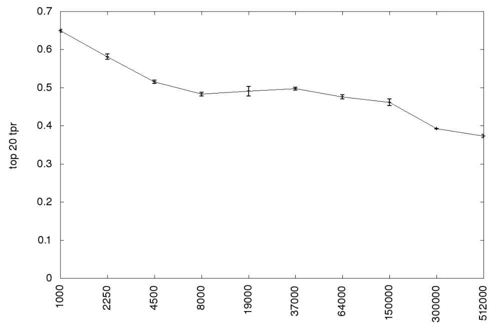
(b) Runtime of RandomWalk in relation to the rank of the tensor



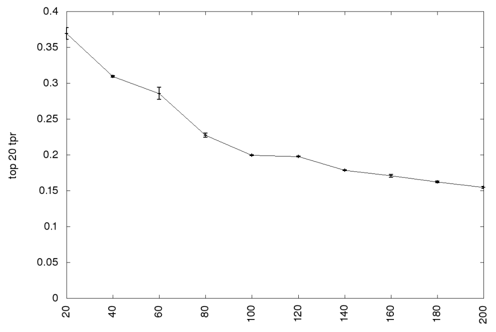
(c) Tpr of RandomWalk in relation to the size of the tensor



(d) Tpr of RandomWalk in relation to the rank of the tensor

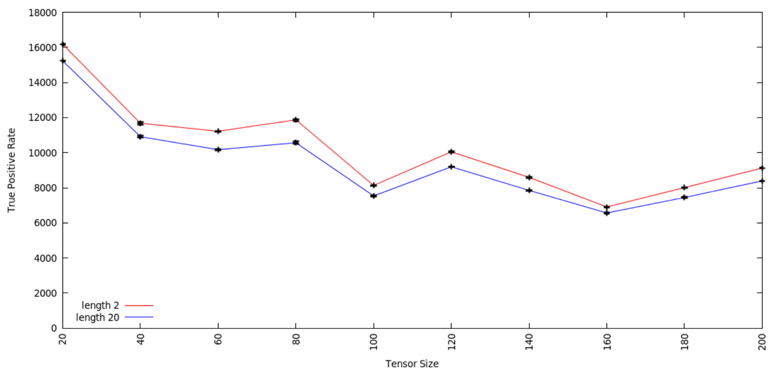


(e) top20-tpr of RandomWalk in relation to the size of the tensor

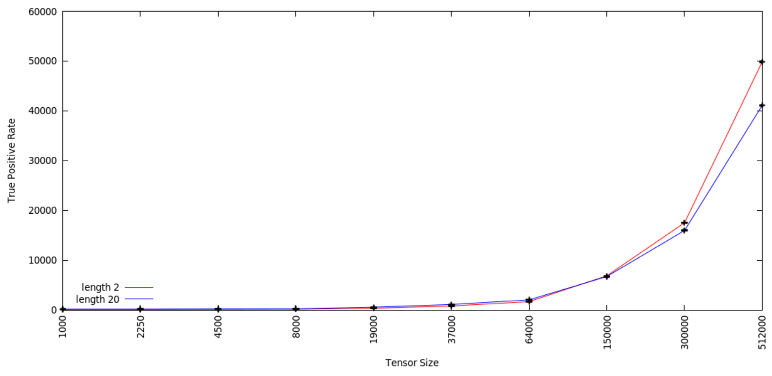


(f) top20-Tpr of RandomWalk in relation to the rank of the tensor

Figure 4.2: The performance of RandomWalk as stand alone



(a) runtime for rank scaling



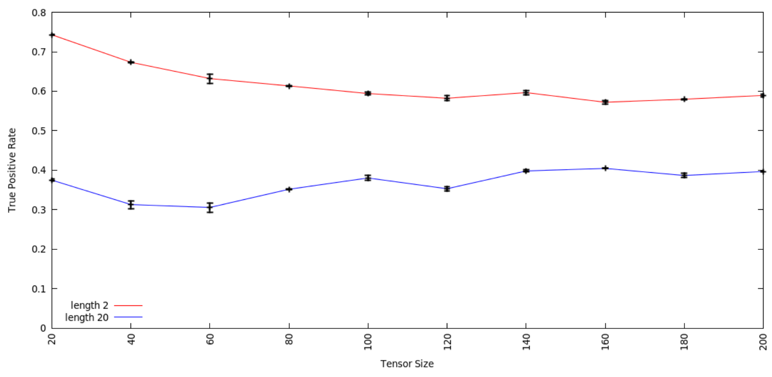
(b) Runtime for size scaling

Figure 4.3: The runtime of RandomWalk for different *walk_len* parameters

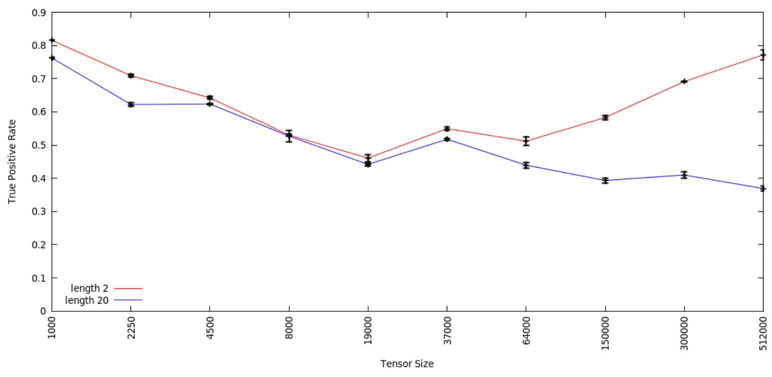
4.3 Experiments on the Walk Length

When choosing the Parameters of the walk (see algorithm 9) the *walk_len* and *walk_num* parameters have the biggest impact. The original paper had some rough runtime estimations for the RandomWalk algorithm, where the runtime was depending on the product of the two. A high *walk_num* results in higher sampling rate and possibly better approximation of the frequencies. The *walk_len* controls how often the walk is restarted, that means for lower values the probability, that the walk reaches far is lower. Another influence is, that shorter walks may lead to less vertices being removed, which results in more iteration and could lead to a higher runtime. At this point it's hard to say what is preferable from a theoretical standpoint, because it is not only quite complex, but also depends on the actual implementation.

I did experiments with different *walk_len* parameters to get an idea on what is happening. These parameters where 2,3,10 and 20. 2 is the lowest possible value, 3 is a quite natural value, because all entries in a (monochomatic) block can be reached in 3 steps. I will only report the results for the extreme values of 2 and 20. The other parameters where kept at the default values.



(a) tpr for rank scaling



(b) tpr for size scaling

Figure 4.4: The true positive rate of RandomWalk for different *walk_len* parameters

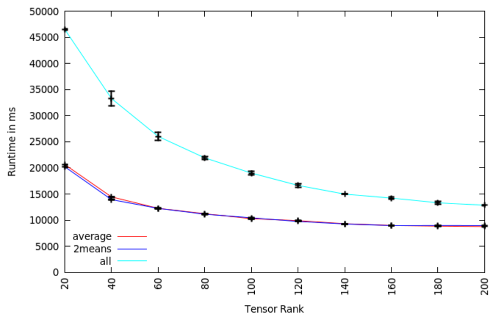
Figure 4.3 shows the impact on the runtime. The distribution of the runtime values stays the same, while the runtime for small *walk_len* values increases. The increase is linear in comparison to the rank of the tensor, but seems to increase more, when the tensors grow larger. In terms of tpr, we can see that it heavily depends on the size of the tensors, but not in a linear fashion. Consider the graphs for size scaling dataset. For a value of 2, we can see that the tpr function first decreases, till it reaches and then it increases. The same observation can be made for higher values, just that the local minimum is later. Still, in most cases a smaller *walk_len* value outperforms the larger ones.

4.4 Evaluation of Selection Algorithms

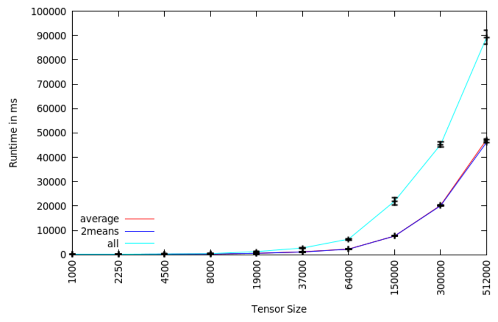
All these methods have a inherit weakness in common. Assuming there is more than 1 block within the visited nodes (as seen in figure 3.1), none would be able to differentiate them, as both would have high frequencies in their respective nodes (If they where encountered roughly at the same time). The result is, that the blocks won't be added as the density of the merged block is most likely too small and both blocks won't be able to be found by the algorithm.

Working around this shouldn't happen within the selection phase, but in the upper level walk procedure. This can be partially controlled by selecting appropriate parameters for the algorithm. For example, the overlaps are commonly relatively small so removing those could be arranged, by selecting the *block_min_size* parameter appropriate. More importantly, in terms of algorithm design (and implementation) it is important to have a clear structure in the algorithm, to keep the implementation maintainable and the algorithm understandable. Should there be multiple blocks within the initial candidate set, this will be viewed as a failing of the higher level algorithms (often it is simply unavoidable). Note, that although $\text{select}_{comp}(C)$ produces multiple blocks they are commonly parts of the same block.

The different selection algorithms were evaluated on the synthetic datasets *scale_size* and *scale_rank* using the RandomWalk of Walk'n'Merge++ with default parameters as specified in section 4.1. Depicted (Figure 4.5 and 4.6) are the results of using different selection functions. For the composite selection function denoted by all I used just about any selection function I could come up with in addition to those mentioned before. The result is, that the runtime explodes. While computing the candidates is not very expensive by design, testing the density of so many blocks actually gets expensive. In terms of runtime select_{avg} and $\text{select}_{2Dimk-Means}$ are very similar. In terms of true positive rate $\text{select}_{2Dimk-Means}$ actually outperforms select_{avg} a bit. The composition of all selection functions is better than the individual selection functions in nearly all cases, which makes sense, since the blocks found by the individual functions are all part of the composite solution. It shows maybe not the best true positive rate we could possibly reach with selection functions, but gives a good idea of the potential of them. In that sense, the results of the two individual selection functions are rather close. In terms of top20-tpr (not depicted), there are no significant differences. One problem with the composite function, that is not shown in the graphs is the fact, that the number of false positives noticeable increases, although all blocks have a density of 0.9. Reason for this, that all the overlapping blocks produce new false positives, but no new true positives. The more overlapping blocks you have the higher is the chance for new false positives, while the number of true positives stays the same. This speaks against the intuition to

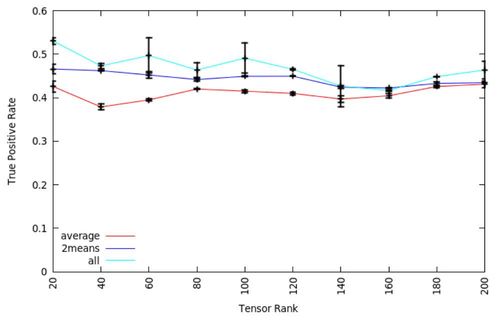


(a) Rank Scaling

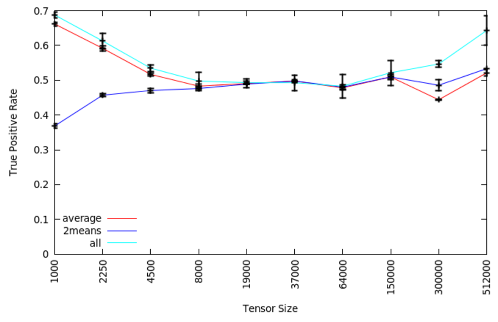


(b) Size Scaling

Figure 4.5: The runtime of the RANDOMWALK algorithm of WALK'N'MERGE++ on the *scale_rank* and the *scale_size* datasets with a block density of 0.9 using select$_{avg}$, select$_{2Dimk-Means}$ and select$_{comp}$ (called all)



(a) Rank Scaling



(b) Size Scaling

Figure 4.6: The true positive rate of the RANDOMWALK algorithm of WALK'N'MERGE++ on the *scale_rank* and the *scale_size* datasets with a block density of 0.9 using select$_{avg}$, select$_{2Dimk-Means}$ and select$_{comp}$ (called all)

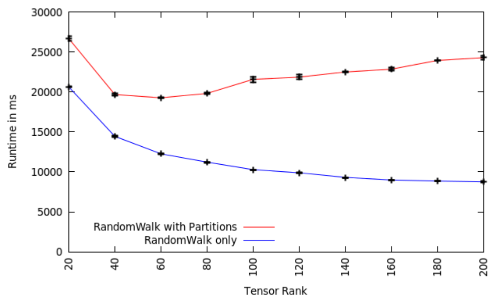
use the true positive measure as sole performance measure. It motivates the composite approach, that only selects the largest block. As a measure of what is the best possible it is still somewhat usable.

4.5 Evaluation of Cluster Algorithms

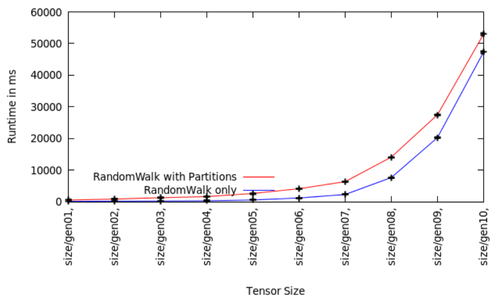
The k-Means, k-Means++, spectral and fast spectral algorithm where all tried with different parameters, but none could show satisfying performance. The reasons for that were already mentioned in Section 3.3.

The MCL algorithm was evaluated in the two synthetic, standard test sets *scale_rank* and *scale_size* and default parameters. The *block_min_size* parameter was varied, to check, whether it only finds the one dimensional clusters (a weakness of many FIBERGRAPH based methods), or if it also finds larger dimensional blocks. For the Selection function the composition function was used.

In terms of runtime it is worse, than the RandomWalk algorithm (without Merge part). It is however, better than the standard Walk'n'Merge procedure, since the BlockMerge algorithm, takes a huge runtime toll. However, since it also only rarely found dense blocks regardless of choice of parameters (besides lowering the *block_density*), the results weren't very satisfying. All in all, I found this result quite surprising, because MCL works on the same general ideas as RandomWalk.



(a) Rank Scaling



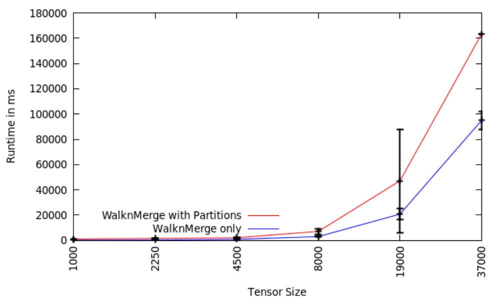
(b) Size Scaling

Figure 4.7: The runtime of the RandomWalk algorithm of Walk'n'Merge++ with partitioning on the *scale_rank* and the *scale_size* datasets using default parameters

4.6 Evaluation of Partitioning

I experimented on both Walk'n'Merge and RandomWalk with partitioning. The experiment setup was to once ran them with partitioning and once without, while keeping all other parameters the same (the default values). In terms of true positive rate and top20-tpr, the results were almost identically, which is not suprising, because with a *block_density* of 0.9 it is highly unlikely, that a dense block spans more than one connected component. In terms of runtime the partitioning did actually perform worse, than the none partitioned variant. For RandomWalk this makes absolutely sense, because walk does not perform different, since non-connected components could never be reached to begin with. What remains is a overhead from the partitioning and maybe from the thread scheduling.

A bit depressing are the results for the Walk'n'Merge experiments (Figure 4.8), because they show worse performance when using partitions. Partially, this could be because of implementation problem, e.g. when scheduling the threads. The MonoMerge isn't affected by that much by the partitioning, because the candidate pairs won't be split. The subsequent merge isn't actually that much of a performance leak. Nonetheless, the partitioning is useful, as we see later with Split'n'Expand.



(a) Rank Scaling

Figure 4.8: The runtime of the WALK'N'MERGE algorithm of WALK'N'MERGE++ with partitioning on the *scale_rank* and the *scale_size* datasets using default parameters

4.7 Evaluation of Simultaneous Walk

The SimWalk was on the same datasets as the RandomWalk with default parameters, without using BlockMerge or a variant. The restart probability of SimWalk was chosen so that the expected walk length was equal to the *walk_len* parameter of RandomWalk. When the algorithm runs through the resulting true positive rates are very similar, however in terms of runtime the SimWalk is significantly worse. Originally, I had hope, that the sparse matrix multiplication provided by Armadillo is optimized enough that, when most entries are zero, its runtime is acceptable. But the experiments show that, when the size of the input tensors gets too large, the runtime is no longer comparable to RandomWalk. Therefore, I did not see a reason to develop this direction further. Of course, an implementation more focused on the SimWalk may have better results.

4.8 Evaluation of Hash Merge

As the HashMerge is an alternative to the BlockMerge algorithm, I simply ran the Walk'n'Merge algorithm and replaced the BlockMerge phase with the new HashMerge. The parameters where kept at their default values and the timeout at 30 minutes. The algorithm didn't have a large impact on the runtime, but it is possible, that it could be more noticeable for those result, where my Experiments timed out. The main reason for this is, that MonoMerge alone has a very significant runtime, such that the subsequent merge is hardly noticeable. The true positive rate was (nearly) not changed and neither was the top-k tpr. The number of actually done merges was smaller, but the number was very small in the original version to begin with.

4.9 Evaluation of All Merge

As the AllMerge is an alternative to the BlockMerge algorithm, I simply ran the Walk'n'Merge algorithm and replaced the BlockMerge phase with the new AllMerge. The parameters where kept at their default values and the timeout at 30 minutes. For those tensors, where the algorithm ran through it had very good results in terms of true positive rate and also top-k true positive rate. Which is not surprising, as its essentially a brute force method for enumerating all possible blocks and this is exactly the problem with this approach. Here is a small though experiment, to give an idea of how many blocks we are talking about.

Assume you have a monochromatic block with d monochromatic subblock, that where found by MonoMerge. First we can merge all those d blocks and have $\binom{d}{2}$ new blocks. those we can merge again with each other and the old d blocks resulting in $\binom{d+\binom{d}{2}}{2}$ merges. This you can continue for quite a time. Of course, this is extremely simplified, but it illustrates well the exploding runtime.

4.10 Evaluation of the Quasi-Clique approach

The paper that proposed the original MIP already stated, that it only really works for small graphs. My experimentation agrees with that. Nonetheless, there where some runs of MIP that I could do with my computational resources. These seem indicate, that the quasi-clique detection on the FiberGraph does not find good block candidates, even when varying the γ parameter. While using it on the extended Graphs let to some encouraging results, where actual dense blocks where found. However, these experiments where too small to be really meaningful and the overall runtime didn't make further examination in this direction feasible without fundamentally changing the detection algorithm.

Chapter 5

Split'n'Expand

5.1 The Algorithm

Based on my results I propose the SPLIT'N'EXPAND algorithm for boolean tensor decomposition. As the name may suggest, the algorithm is based on (recursively) partitioning the tensor. We have seen, that partitioning can be used to reduce the problem size into several smaller parts, at relative cheap cost and that it allows to apply pruning to reduce the problem size further. The partition procedure discussed in this section reduces the size of the tensor even further, than the one discussed earlier in Section 3.4. The SPLIT'N'EXPAND algorithm (algorithm 11) relies on partitioning and splitting to recursively reduce the tensor size, till it either finds trivial blocks, because the whole partition is essentially a dense block, the partition gets pruned away, because it's too small or sparse, or the tensor gets too small to be sensible split. Once this happens, it starts searching for blocks by expanding dense subblocks (line 14).

5.1.1 Partitioning Phase

Using algorithm 6 from WALK'N'MERGE++, partitioning the graph into connected components is possible in linear time. It can additionally be parallelized, as shown in algorithm 12. The algorithm simply tries to compute the partitions in parallel. Sometimes it will create the same partition twice during the same parallel section, when the start nodes are from the same connected component. This is bound to happen not too often, because the entries will be removed after one iteration, so that future iterations won't chose the same component. The probability that the same component is chosen can be estimated by $\min\left(1, \frac{t}{c}\right)$ (t = number of threads, c = number of components), which of course assumes, that the size of the components is uniform. Typically, c is larger

```
Input: tensor $\mathcal{X}$, min_block_size, min_block_vol, min_diameter
Output: blocks $(\{(X_1, Y_1, Z_1), (X_2, Y_2, Z_2), ...\})$
1  create FiberGraph $FG(V, E)$ from $\mathcal{X}$;
2  partitions = empty queue;
   /* The partitioning procedure used in Walk'n'Merge++                    */
3  forall the connected components C $\subset V$ do
4      if C is a Block candidate then
5          partitions.add(C);
   /* paralellized, since the partitions are independent of each other    */
6  while in parallel: partitions is not empty do
7      Candidate C = partitions.pop();
       /* pruning step                                                      */
8      if C is not a block candidate then
9          continue ;                                   // get next candidate
       /* density check                                                     */
10     if C represents a block in $\mathcal{X}$ then
11         blocks.add(toBlock(C))
12     else
           /* reduce to smaller sub problem                                 */
13         if diameter of FiberGraph $FG(C, E_C) <$ min_diameter then
14             Try finding blocks in $FG(V, E)$ by expanding nodes in $C$;
15         else
16             (C$_1$,C$_2$) = split of C;
17             partitions.add(C$_1$);
18             partitions.add(C$_2$);
19 return blocks;
```

Algorithm 11: Pseudo code for the Split'n'Expand algorithm

than t, which further reduces the chance of coalitions. But still, the worst case runtime remains at $O(n)$ and in later phases, when the number of components grows smaller, more collisions happen.

```
Input: tensor $\mathcal{X}$
Output: partitions ({P$_1$, P$_2$,. . . })
1  create FiberGraph $FG(V, E)$ from $\mathcal{X}$;
2  partitions = an empty set;
3  while in parallel: $V$ is not empty do
4      start node $v$ = randomNode($V$);
5      partition P = ConnecteComponents($v$);
6      if P $\notin$partitions then
7          partitions.add(P);
8          $V = V \setminus P$;
9  return partitions;
```

Algorithm 12: Pseudo code for the parallelized partitioning algorithm

5.1.2 Splitting Phase

```
Input: partition $\mathtt{C}$, FiberGraph $FG(\mathtt{C}, E_{\mathtt{C}})$, minimal diameter $dm$
Output: partitions $(\{\mathtt{C}_1, \mathtt{C}_2\})$
1  $dm$ = pseudoDiameter($FG(\mathtt{C}, E_{\mathtt{C}})$);
2  select two nodes $s, t$ from $V$ with shortest path distance $dm$ from each other;
3  $dist_s$ = shortest path distances from $s$ computed using Dijkstra;
4  $dist_t$ = shortest path distances from $t$ computed using Dijkstra;
5  $\mathtt{C}_1$ = partitions with $s$;
6  $\mathtt{C}_2$ = partitions with $t$;
7  forall the $v \in V$ do
       /* add to closestpartition                                    */
8      if $dist_s(v) <= dist_t(v)$ then
9          add $v$ to $\mathtt{C}_1$ ;
10     if $dist_s(v) > dist_t(v)$ then
11         add $v$ to $\mathtt{C}_2$ ;
12 return $(\{\mathtt{C}_1, \mathtt{C}_2\})$;
```

Algorithm 13: Pseudo code for the graph splitting algorithm

Originally, the idea was to use MinCut, based on Karger's algorithm [KS96], but it quickly became obvious, that MinCut is quite a bad splitting procedure for Partitioning FiberGraphs, because they tend to serve small outliers or single nodes from within a block. One has to keep in mind, that the overlapping section of different blocks can actually be quite dense, when considering the edge density. In the end I developed the algorithm displayed in 13.

I start by estimating the diameter of the graph. To do this exactly, you could solve the APSP (All path shortest path) problem, which can be solved in $O(|V|^3)$ using Floyd's algorithm [Flo62] or $O|V|(|V||E|\log(|V|))$ using multiple iterations of Dijkstra's algorithm [MF10]. I use an approximation called the pseudo diameter shown in algorithm 14. It tries to find the longest shortest path of a graph by repeatedly choosing a longest shortest path as calculated by Dijkstra and restarting from one of its endpoints as potential start point for a new longest shortest path [GRD$^+$10].

Having the (pseudo) diameter of the graph, I select two nodes (v and u) with maximal shortest path distance and run Dijkstra's algorithm (worst case performance $O(|E| + |V|log(|V|)))$ from both of them to compute the shortest path distance from all nodes to these two respectively. An important invariant is, that the graphs of the different partitions are connected or else there could be infinite distances. Finally, the graph is divided into two partitions. They are initially made up by the two pivot nodes v and u and all other nodes are assigned to whatever partitions pivot node is nearer. The procedure is motivated by the following idea: When two nodes have a sufficient long shortest path distance we can make some assumptions over the distance to other nodes.

```
**Input**: graph $G(V, E)$
**Output**: pseudo diameter $dm$)
1 $s$ = random node from $V$;
2 $d_{old} = 0$;
3 $d_{new} = 0$;
4 do
5     compute shortest path distances from $v$ using dijkstra's algorithm for all $v' \in V$;
6     $d_{new}$ = max shortest path distance from dijkstra's;
7     $v$ = random node of $V$ with distance $d_{new}$ from the old $v$;
8 while $d_{old} < d_{new}$;
9 return $d_{new}$;
```

Algorithm 14: Pseudo code for the calculation of the pseudo diameter

Let d be the shortest path distance between v and u, let t be another in the graph with distance d_u to u with $d_u \leq d$, then we know that the distance d_v to v is less or equal $d - d_u$ or else the would be a shorter path than measured by the shortest path distance, which is a contradiction. In other words, when the diameter is sufficiently large a node that is close to u is very unlikely to form a dense block together with a node close to v, because the path in the FiberGraph is long.

One always risks splitting important structures in the FiberGraph with the splitting procedure. I did therefore experiment with a few different splitting procedures and among them were some where the partitions actually could overlap (e.g. when all nodes with distance of 3 from the pivot are also added). This is also the reason, why I don't call this process partitioning, because commonly only disjoint sets are referred to as partitions. One problem with having any kind of overlap is, that the rate with which the graph shrinks is reduced, which for all tested variants significantly increased the runtime. In the end, I used the expansion procedure in hope of restoring split structures.

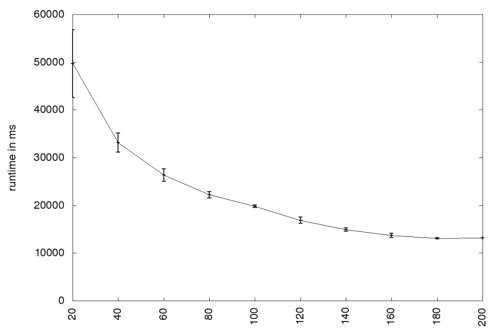
5.1.3 Expansion

The idea of the expansion is to expand a block (X, Y, Z), by adding new entries to the three index sets. The search is guided by the FiberGraph and works in a greedy manner. The pseudo code for this BlockExp algorithm is displayed in Algorithm 15. The algorithm is like the partition procedure motivated by the idea, that not connected components in the FiberGraph are not candidates for dense blocks, so components of dense blocks are connected by edges. The algorithm simply tries to expand with all connected nodes in the order they where encountered. Its not an optimal algorithm, because this would require some backtracking, since the order of nodes tried changes the form of the resulting block.

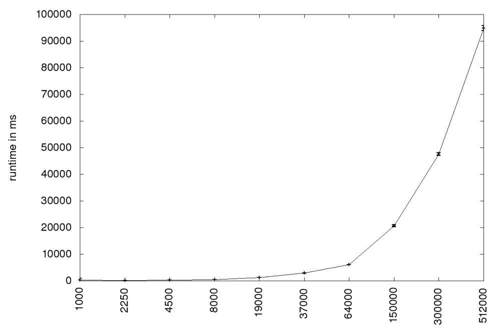
```
Input: dense Block $\mathtt{B} = (X, Y, Z)$, FiberGraph $FG(V, E)$, Tensor $\boldsymbol{\mathcal{X}}$
Output: dense Block $\mathtt{B}' = (X', Y', Z')$ with $X \subset X'$,$Y \subset Y'$ and $Z \subset Z'$
1  Queue openNodes = FG neighbours of non-zero entries of $\boldsymbol{\mathcal{X}}$ in B not in B itself;
2  Set closedNodes = all non-zero entries of $\boldsymbol{\mathcal{X}}$ in B;
3  while openNodes is not empty do
4      node $v$ = openNodes.pop();
5      if $v \notin$ closedNodes then
6          if $(X \cup v.x, Y \cup v.y, Z \cup v.z)$ is a dense Block then
7              $\mathtt{B} = (X \cup v.x, Y \cup v.y, Z \cup v.z)$;
8              Add FG neighbours of $v$ to openNodes if they are not in closedNodes;
9          add $v$ to closedNodes;
10 return (B);
```

Algorithm 15: Block expansion algorithm BlockExp

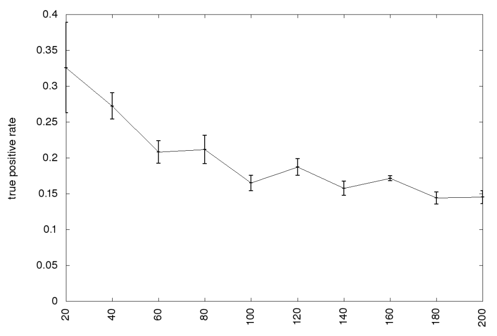
The smallest dense block is the blocks that contains only one element. Therefore, when Split'n'Expand starts expanding in a partition it simply selects some nodes called seeds uniformly at random from the partition and expands them in the original graph using BlockExp (not the graph only containing the partition).



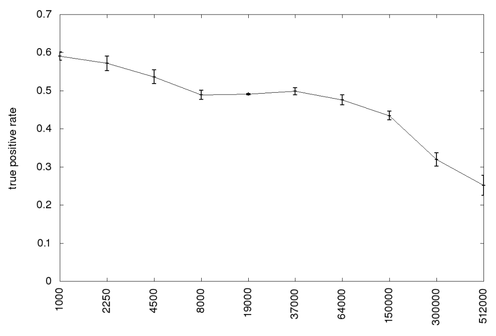
(a) Runtime for the *scale_rank* dataset



(b) Runtime for the *scale_size* dataset



(c) True positive rate for the *scale_rank* dataset



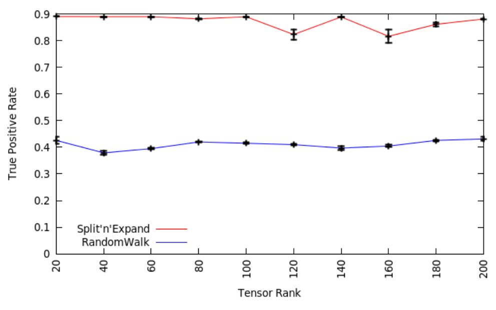
(d) True positive for the *scale_size* dataset

Figure 5.1: Standalone performance of the expansion procedure when used on 100 randomly sample blocks

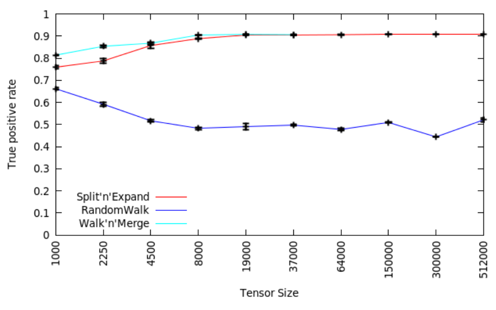
5.2 Evaluation

5.2.1 Evaluation of BlockExp

The expansion algorithm BlockExp is actually potentially useful for any algorithm that finds dense blocks like the Walk'n'Merge algorithm, but it wasn't evaluated in that context, because of time constraints. One could also use the expansion procedure as replacement for the RandomWalk. An experiment that tried this is depicted in Figure 5.1, where I simply sampled 100 non-zero entries from the test tensors and expanded them. Even as a stand alone it doesn't perform half bad. It does not reach the level of RandomWalk or Walk'n'Merge, but for a unguided random sampling algorithm it gets relatively good tpr results. As expected the tpr decreases with increase of the tensor rank, because the number of sampled blocks stays the same, but also with the size of tensor. The later could be an inherit weakness of the greedy approach, that occurs in larger blocks.



(a) Rank Scaling



(b) Size Scaling

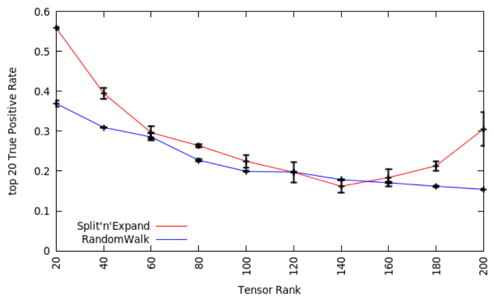
Figure 5.2: The true positive rate of Split'n'Expand on the *scale_rank* and the *scale_size* datasets

5.2.2 Evaluation of Split'n'Expand

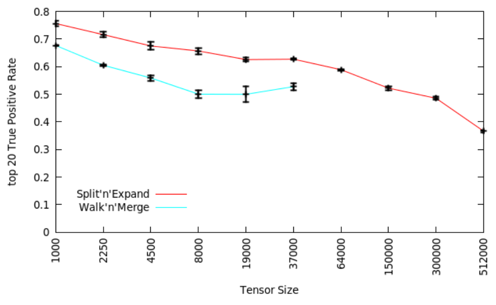
Split'n'Expand was evaluated on the two synthetic data and compared to Walk'n'Merge and RandomWalk, both with default parameters. As discussed before, some parameters can be optimized to increase the performance, but this gives only slight improvements, so we can still compare to the default version and get a good impression of its relative performance. The *min_diameter* was set to 5, because this is a good estimation for the diameter of a somewhat sparse block.

The runtime of Split'n'Expand is significantly better than the one of Walk'n'Merge, but slightly slower than the one of the RandomWalk (Figure 5.4). More importantly, it scales well with increased rank and size. In terms of true positive rate the results of Split'n'Expand are quite convincing. For true positive rate alone it reaches the best possible value of about 0.9 (The remaining 0.1 are lost because of the constructive noise). Additionally, the results of the top 20 true positive rate experiments are good. They are better than the sole RandomWalk, and for those runs where Walk'n'Merge didn't timeout, they also show better performance.

These good results also extend to the real-world datasets, but not with the default parameters. It turns out, that there are actually only few very large, dense blocks in the real-world data. By either changing the minimal block density or the minimal size of the blocks, one gets results, that are comparable to the ones from the synthetic data.

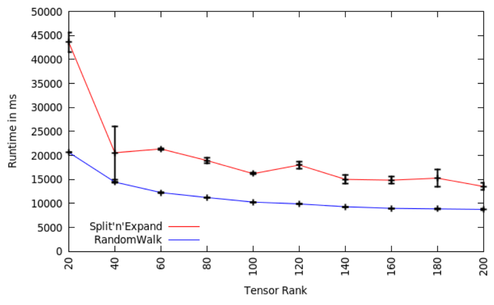


(a) Rank Scaling

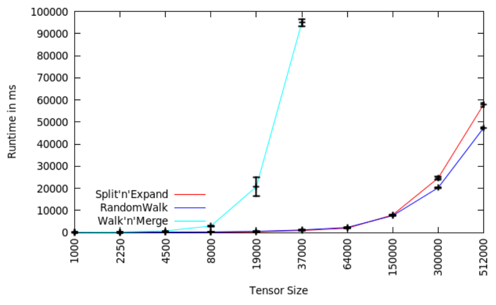


(b) Size Scaling

Figure 5.3: The top 20 true positive rate of SPLIT'N'EXPAND on the *scale_rank* and the *scale_size* datasets



(a) Rank Scaling



(b) Size Scaling

Figure 5.4: The Runtime of SPLIT'N'EXPAND on the *scale_rank* and the *scale_size* datasets

Chapter 6

Conclusion

6.1 Future Work

This work only has the scope of a master thesis, therefore its scope is limited and many questions remain open. I will use this section to list some of the ideas, that I would have liked to explore further.

6.1.1 mode FiberGraph

The clustering approaches could have been explored further. For example by considering the similarity on a fiber level first. Meaning, one could reduce the FIBERGRAPH to only include x-fibers, y-fibers or z-fibers

Definition 6.1. *x-mode* FIBERGRAPH For a given boolean 3-way tensor $\boldsymbol{\mathcal{X}}$ the **x-mode FiberGraph** $FG(V, E)$ with set of vertices V and edge relation $E \in V \times V$ is defined by:

- **Nodes(V):** for every entry with $(\boldsymbol{\mathcal{X}})_{ijk} = 1$ there is a node (v_{ijk}).
- **Edges(E):** Two nodes (v_{ijk}, v_{pqr}) are connected, when their coordinates differ only in their x position *(i.e. $j = q, k = r$ and $i \neq p$).*

y-mode FIBERGRAPH and z-mode FIBERGRAPH are defined analogue. A block (X, Y, Z) consists of elements, that are similar in the x-mode FIBERGRAPH in X, similar in the y-mode FIBERGRAPH in Y and similar in the z-mode FIBERGRAPH in Z. Potentially, this could be used to better differentiate overlapping blocks, in hope, that they don't overlap all dimension the same.

6.1.2 Distance Estimation

Given the FiberGraph, we know that vertices of the same block are connected by a path of length 3. That means, that means we should consider subsets Block as candidates for dense Blocks, when they have a diameter of 3. Overall, I think that many approaches discussed in this work could be improved by using a distance matrix as base ,e.g. the clustering approaches.

Of course, the distance matrix is computational expansive ($O(|V|^2|E|\log(|V|))$ or $O(|V|^3)$) so you would most likely end up using approximative measures.

6.1.3 Improved Splitting

The splitting procedure of Split'n'Expand is still its weakest part and is should be adapted, that its less likely to split connected blocks.

6.1.4 Expansion

The expansion procedure, introduced with Split'n'Expand, could be used to a larger extend, for example, by applying it to all blocks found by the MonoMerge algorithm. Beyond that, the expansion algorithm could be further improved, since its current greedy form is not guaranteed to find good results.

6.1.5 Larger MonoMerge blocks

As of now, MonoMerge is used to find blocks of size 2. You could try to find monochromatic blocks of different size and use them to construct new blocks.

Additionally, you could limit these blocks to size $2 \times 2 \times 2$, and not to those of size $2 \times 2 \times n$ for $n \geq 2$, because the current format limits the ability of blocks to be connected along, the x and y dimensions, e.g. a block of dimensions $20 \times 2 \times 2$ is less likely to be found by subsequent merge as of now, when the are some blocks sharing x and y coordinates. Every block can be approximated by a union of size 2 blocks, so just about every block (not considering the effect of RandomWalk) could be approximated by a sequence of merges.

One could completely skip the RandomWalk and similar procedures and directly start with the MonoMerge and build blocks in a bottom up principle. Of course, this requires a significantly improved merge procedure.

6.1.6 Quasi-Clique approach

The quasi-cliques were only explored in form of a MIP. You could try different other algorithmic approaches and use properties of quasi-cliques, like the quasi-inheritance mentioned in [PVBB13].

6.1.7 Further Develop Split'n'Expand

One could potentially replace the expansion procedure with other functions, e.q. the RandomWalk starting from a node in the partition, because in some sense are the BlockExp and RandomWalk algorithms very similar. The main advantage of the BlockExp algorithm being, that it is guaranteed to find a dense block. The main function of the splitting procedure could then be seen as finding good starting points for the walk.

6.2 Conclusion

This work discussed a range of different modifications for the Walk'n'Merge algorithm and gave some further insights into its performance. The focus of this work was to improve the runtime of the algorithm and thereby it concentrated more on improving the fast RandomWalk, than the significantly slower BlockMerge. Small modifications to the RandomWalk like the new selection functions provided a small performance boost. The modifications aiming at improving the BlockMerge did not provide significant improvements of any kind. The alternative approaches of finding blocks with help of the FiberGraph through clustering or the quasi-clique MIP either didn't have good enough runtime or failed at identifying blocks. Of course, one could still look further into these methods, as their scope definitely wasn't fully exhausted.

The proposed Split'n'Expand algorithm shows very good performance results and makes excellent use of the FiberGraph to explore the tensor. although there are still some aspects, that could be fine tuned further, some of those potential modifications, where discussed in the last section.

All in all, I think that the goal of the work, to improve upon the Walk'n'Merge algorithm was achieved. Although there are still many open questions remaining.

List of Figures

List of Tables

Appendix A

Default Parameter of Walk'n'Merge++

A.1 Default Parameters

This is a list of the parameters used by Walk'n'Merge++. For most of the experiments the settings where kept at their default value.

- The mode of the program: possible values are: WALK, CLUSTER, CLUSTER_N_MERGE, LP (default: WALK_N_MERGE)
- **block_min_density**: The Minimum density of blocks to be generated (default: 0.9)
- **block_min_size**: The Minimum size (in all directions) of blocks to be generated (2)
- **block_min_vol**: The Minimum volume of blocks to be generated (8)
- Extended Graph is not used
- Simultaneous Walk is not used
- HashMerge is not used
- Partitioning is not used
- **cluster_mode:**Cluster mode is either KMEANS, SPECTRAL, FAST_SPECTRAL (KMEANS), EXTERN. EXTERN is used with MCL clustering or other extern clustering programs. Graphs can be exported in (abc)-format using -mode PRINT_GRAPH. It reads the clusters from *clusters* file. Yes, its a ugly hack.

- **cluster_rank:**The approximate Tensor Rank of the data. Cluster algorithms try to find as many blocks. (default: 50)
- **fs_num_cluster:**The number of k-Means Cluster used for the fast Spectral Analysis (default: 25)
- **delete_mode:**Delete_mode is either NO_DELETE, VISITED, CANDIDATES (default: VISITED)
- **walk_num:**The number of walks in the Walk phase of the algorithm. (default: 1000)
- **walk_len:**The length of walks in the Walk phase of the algorithm. (default: 10)
- **walk_min_freq:** The frequency Threshold for node in the Walk phase of the algorithm. (default: 0), used by fixed threshold based selection.
- **graph_reb_freq:**The frequency with which the graph is rebuilt. (default: 4)
- **walk_max_iter:**The Maximum number of Walk Iterations for the NO_DELETE or CANDIDATE delete mode (multiple walks per iteration based on job factor and thread number), later replaced by convergence criteria, but still available as artefact. (default: 10)
- **bloom_cap:**The Capacity of the BLOOM filter (default: 1.000.000)
- **bloom_err:**The Error Rate of the BLOOM filter (default: 0.01)
- **thread_num:**The number of used in the parallel parts (default: 8)
- **job_factor:** Scales the amount of work done in parallel at once (default: 4)

Bibliography

[ABD$^+$90] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[ANTT02] Arvind Arasu, Jasmine Novak, John Tomlin, and John Tomlin. Pagerank computation and the structure of the web: Experiments and algorithms, 2002.

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[BLA02] An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[Bro97] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[CC70] J Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of âĂIJeckart-youngâĂİ decomposition. *Psychometrika*, 35(3):283–319, 1970.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[EM13a] Dora Erdos and Pauli Miettinen. Discovering facts with boolean tensor tucker decomposition. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 1569–1572, New York, NY, USA, 2013. ACM.

[EM13b] Dóra Erdös and Pauli Miettinen. Walk 'n' merge: A scalable algorithm for boolean tensor factorization. In *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pages 1037–1042, 2013.

[Fin05] Holmes Finch. Comparison of distance measures in cluster analysis with dichotomous data. *Journal of Data Science*, 3(1):85–100, 2005.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[GO15] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015.

[GRD$^+$10] Bidyut Gupta, Shahram Rahimi, Narayan C. Debnath, Divya Sree Videm, and Krishnaraj Ethirajan. Pseudo diameter-based pruning - a qos based broadcasting for wide area networks. In Thomas Philips, editor, *CATA*, pages 138–141. ISCA, 2010.

[Hit27] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *J. Math. Phys*, 6(1):164–189, 1927.

[HL13] Christopher J. Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *J. ACM*, 60(6):45:1–45:39, November 2013.

[KB09] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, August 2009.

[KS96] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, July 1996.

[Lei06] Friedrich Leisch. A toolbox for k-centroids cluster analysis. *Comput. Stat. Data Anal.*, 51(2):526–544, November 2006.

[Li05] Xiaoye S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[Llo82] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[Mac67] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[MF10] Thomas J. Misa and Philip L. Frana. An interview with edsger w. dijkstra. *Commun. ACM*, 53(8):41–47, August 2010.

[MM15] Saskia Metzler and Pauli Miettinen. Clustering boolean tensors. *Data Mining and Knowledge Discovery*, 29(5):1343–1373, 2015.

[MN88] A.A. Markov and N.M. Nagorny. *The Theory of Algorithms*. Mathematics and its Applications. Springer Netherlands, 1988.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[PVBB13] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko, and Vladimir Boginski. On the maximum quasi-clique problem. *Discrete Applied Mathematics*, 161(1):244–257, 2013.

[RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[San10] Conrad Sanderson. Armadillo: an open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, September 2010.

[SD09] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[Sei95] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

[SG12] Dafna Shahaf and Carlos Guestrin. Connecting two (or less) dots: Discovering structure in news articles. *ACM Trans. Knowl. Discov. Data*, 5(4):24:1–24:31, February 2012.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[Tuc6c] L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31:279–311, 1966c.

[VD00] Stijn Van Dongen. A cluster algorithm for graphs. *Report-Information systems*, (10):1–40, 2000.

[WS11] Haizhou Wang and Mingzhou Song. Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. *The R Journal*, 3(2):29–33, 2011.

[YHJ09] Donghui Yan, Ling Huang, and Michael I Jordan. Fast approximate spectral clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 907–916. ACM, 2009.