Walk'n'Merge: A Scalable Algorithm for Boolean Tensor Factorization

Dóra Erdős
Boston University
Boston, MA, USA
dora.erdos@bu.edu

Pauli Miettinen
Max-Planck-Institut für Informatik
Saarbrücken, Germany
pauli.miettinen@mpi-inf.mpg.de

***Abstract*—Tensors are becoming increasingly common in data mining, and consequently, tensor factorizations are becoming more important tools for data miners. When the data is binary, it is natural to ask if we can factorize it into binary factors while simultaneously making sure that the reconstructed tensor is still binary. Such factorizations, called Boolean tensor factorizations, can provide improved interpretability and find Boolean structure that is hard to express using normal factorizations. Unfortunately the algorithms for computing Boolean tensor factorizations do not usually scale well. In this paper we present a novel algorithm for finding Boolean CP and Tucker decompositions of large and sparse binary tensors. In our experimental evaluation we show that our algorithm can handle large tensors and accurately reconstructs the latent Boolean structure.**

***Keywords*—Tensor factorizations; Boolean tensors; Random walks; MDL principle**

I. Introduction

Tensors, and their factorizations, are getting increasingly popular in data mining. Many data sets can be interpreted as ternary (or higher arity) relations (e.g. sender, receiver, and date in correspondence). Such relations have a natural representations as 3-way (or higher order) tensors. A data miner who is interested in finding some structure from such a tensor would normally use tensor decomposition methods, commonly either CANDECOMP/PARAFAC (CP) or Tucker decomposition. In both of these methods, the goal is to (approximately) reconstruct the input tensor as a sum of simpler elements (e.g. rank-1 tensors) with the hope that these simpler elements would reveal the latent structure of the data.

The type of these simpler elements plays a crucial role on determining what kind of structure the decomposition will reveal. For example, if the elements contain arbitrary real numbers, we are finding general linear relations; if the numbers are non-negative, we are finding parts-of-whole representations. In this paper, we study yet another type of structure: that of *Boolean tensor factorizations* (BTF). In BTF, we require the data tensor to be binary, and we also require any matrices and tensors that are part of the decomposition to be binary. Further, instead of normal addition, we use the Boolean summation $1+1=1$. The type of structure found under BTF is different to the type of structure found under normal algebra (non-negative or otherwise). Intuitively, if there are multiple "reasons" for a 1 in the data, under normal algebra and non-negative values, for example, we explain this 1 using a sum of smaller values, but under Boolean algebra, any of these reasons alone is sufficient, and there is no penalty for having multiple reasons.

There exists algorithms for BTF (e.g. [1]–[3]), but they do not scale well. Our main contribution in this paper is to present a scalable algorithm for finding Boolean CP and Tucker decompositions. Further, we apply the MDL principle to automatically select the size of the decomposition.

II. Definitions

A. Notation

We present the notation for 3-way tensors, but it can be extended to N-way tensors in a straight forward way. Element (i,j,k) of a 3-way tensor $\boldsymbol{\mathcal{X}}$ is denoted either as x_{ijk} or as $(\boldsymbol{\mathcal{X}})_{ijk}$. A colon in a subscript denotes taking that mode entirely; for a 3-way tensor $\boldsymbol{\mathcal{X}}$, $\boldsymbol{x}_{:jk}$ is the (j,k) *mode-1 (column) fiber*, $\boldsymbol{x}_{i:k}$ the (i,k) *mode-2 (row) fiber*, and $\boldsymbol{x}_{ij:}$ the (i,j) *mode-3 (tube) fiber*. Furthermore, $\boldsymbol{X}_{::k}$ is the kth *frontal slice* of $\boldsymbol{\mathcal{X}}$. We use $\boldsymbol{X}_k$ as a shorthand for the kth frontal slice.

The number of non-zero elements in $\boldsymbol{\mathcal{X}}$ is denoted by $|\boldsymbol{\mathcal{X}}|$. The Frobenius norm $\|\boldsymbol{\mathcal{X}}\|$ is defined as $(\sum_{i,j,k} x_{ijk}^2)^{1/2}$. If $\boldsymbol{\mathcal{X}}$ is binary, i.e. takes values only from $\{0,1\}$, then $|\boldsymbol{\mathcal{X}}| = \|\boldsymbol{\mathcal{X}}\|^2$. The *tensor sum* of two n-by-m-by-l tensors $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ is the element-wise sum, $(\boldsymbol{\mathcal{X}}+\boldsymbol{\mathcal{Y}})_{ijk} = x_{ijk}+y_{ijk}$. For binary $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$, their *Boolean tensor sum* is defined as $(\boldsymbol{\mathcal{X}} \vee \boldsymbol{\mathcal{Y}})_{ijk} = x_{ijk} \vee y_{ijk}$, and their *exclusive or* is $(\boldsymbol{\mathcal{X}} \oplus \boldsymbol{\mathcal{Y}})_{ijk} = x_{ijk}+y_{ijk} \mod 2$. If $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are vectors of length n, m, and l, respectively, then their *outer product*, $\boldsymbol{\mathcal{X}} = \boldsymbol{a} \boxtimes \boldsymbol{b} \boxtimes \boldsymbol{c}$, is an n-by-m-by-l tensor with $x_{ijk} = a_i b_j c_k$. A tensor that is an outer product of three vectors has *tensor rank* 1. Finally, if $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ are binary n-by-m-by-l tensors, we say that $\boldsymbol{\mathcal{Y}}$ *contains* $\boldsymbol{\mathcal{X}}$ if $x_{ijk}=1$ implies $y_{ijk}=1$ for all i,j,k. This relation defines a partial order of n-by-m-by-l binary tensors.

B. Ranks and Factorizations

The Boolean Tensor Rank and CP Decomposition. A binary rank-1 tensor has Boolean rank 1. Higher ranks are defined as Boolean sums of rank-1 tensors:

Definition 2.1 (Boolean tensor rank): The *Boolean rank* of a 3-way binary tensor $\boldsymbol{\mathcal{X}}$, $\mathrm{rank}_B(\boldsymbol{\mathcal{X}})$, is the least integer r such that there exist r rank-1 binary tensors with

$$\boldsymbol{\mathcal{X}} = \bigvee_{i=1}^{r} \boldsymbol{a}_i \boxtimes \boldsymbol{b}_i \boxtimes \boldsymbol{c}_i \ . \quad (1)$$

For (approximate) Boolean CP decomposition, our goal is to find the least-error Boolean rank-k approximation of the input tensor. The error is measured as the number of disagreements.

Problem 2.1 (Boolean CP decomposition): Given a binary tensor $\boldsymbol{\mathcal{X}}$ (n-by-m-by-l) and an integer r, find binary matrices $\boldsymbol{A}$ (n-by-r), $\boldsymbol{B}$ (m-by-r), and $\boldsymbol{C}$ (l-by-r) that minimize

$$\left| \boldsymbol{\mathcal{X}} \oplus \left(\bigvee_{i=1}^{r} \boldsymbol{a}_i \boxtimes \boldsymbol{b}_i \boxtimes \boldsymbol{c}_i \right) \right| . \quad (2)$$

Finding the Boolean rank or a minimum-error rank-r Boolean CP decomposition of a tensor is NP-hard [2].

Boolean Tucker decompositions. The Boolean Tucker decomposition of a 3-way Boolean tensor contains a binary *core tensor* and three binary factor matrices.

Problem 2.2 (Boolean Tucker decomposition): Given an n-by-m-by-l binary tensor $\boldsymbol{\mathcal{X}} = (x_{ijk})$ and three integers p, q, and r, find the minimum-error (p, q, r) *Boolean Tucker decomposition* of $\boldsymbol{\mathcal{X}}$, that is, tuple $(\boldsymbol{\mathcal{G}}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$, where $\boldsymbol{\mathcal{G}}$ is a p-by-q-by-r binary *core tensor* and $\boldsymbol{A}$ (n-by-p), $\boldsymbol{B}$ (m-by-q), and $\boldsymbol{C}$ (l-by-r) are binary *factor matrices*, such that $(\boldsymbol{\mathcal{G}}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ minimizes

$$\sum_{i,j,k} \left(x_{ijk} \oplus \left(\bigvee_{\alpha=1}^{p} \bigvee_{\beta=1}^{q} \bigvee_{\gamma=1}^{r} g_{\alpha\beta\gamma}\, a_{i\alpha} b_{j\beta} c_{k\gamma} \right) \right) . \quad (3)$$

C. Blocks, Convex Hulls, and Factorizations

Let $\boldsymbol{\mathcal{X}}$ be a binary n-by-m-by-l tensor and let $X \subseteq [n]$, $Y \subseteq [m]$, and $Z \subseteq [l]$, where $[x] = \{1, 2, \ldots, x\}$. A *block* of $\boldsymbol{\mathcal{X}}$ is a $|X|$-by-$|Y|$-by-$|Z|$ sub-tensor $\boldsymbol{\mathcal{B}}$ that is formed by taking the rows, columns, and tubes of $\boldsymbol{\mathcal{X}}$ defined by X, Y, and Z, respectively. Block $\boldsymbol{\mathcal{B}}$ is *monochromatic* if all of its values are 1. We can embed $\boldsymbol{\mathcal{B}}$ to n-by-m-by-l tensor by filling the missing values with 0s. Monochromatic $\boldsymbol{\mathcal{B}}$ is (embedded or not) a rank-1 tensor. Almost-monochromatic $\boldsymbol{\mathcal{B}}$ is *dense*.

Let the sets I, J, and K be such that they contain the indices of all the non-zero slices of $\boldsymbol{\mathcal{X}}$. That is, $I = \{i : x_{ijk} = 1 \text{ for some } j, k\}$, $J = \{j : x_{ijk} = 1 \text{ for some } i, k\}$, and $K = \{k : x_{ijk} = 1 \text{ for some } i, j\}$. The *convex hull* of $\boldsymbol{\mathcal{X}}$ is a binary n-by-m-by-l tensor $\boldsymbol{\mathcal{Y}}$ that has 1 in every position defined by the Cartesian product of I, J, and K, $I \times J \times K = \{(i, j, k) : i \in I, j \in J, k \in K\}$. The following lemma will explain the connection between monochromatic blocks (rank-1 tensors) and convex hulls.

Lemma 2.1: Let $\boldsymbol{\mathcal{X}}$ be a binary n-by-m-by-l tensor. Then the convex hull of $\boldsymbol{\mathcal{X}}$ is the smallest n-by-m-by-l rank-1 binary tensor that contains $\boldsymbol{\mathcal{X}}$.

For the proof, see [4]. As a corollary to Lemma 2.1 we get that $\boldsymbol{\mathcal{X}}$ is rank-1 if and only if it is its own convex hull.

III. The Walk'n'Merge Algorithm

In this section we present the main part of our algorithm, Walk'n'Merge, that aims to find the dense blocks from which we build the factorizations. Walk'n'Merge contains two phases: RandomWalk aims at finding and removing the most prominent blocks quickly from the tensor. BlockMerge uses these blocks together with smaller, easier-to-find monochromatic blocks and tries to merge them into bigger blocks.

Algorithm 1 Random walk algorithm to find blocks.

Input: $\boldsymbol{\mathcal{X}}$, d, `walk_length`, `num_walks`, `freq`
Output: $\boldsymbol{\mathcal{B}}_1, \boldsymbol{\mathcal{B}}_2 \ldots \boldsymbol{\mathcal{B}}_k$

```
1:  create graph G(V,E) from X
2:  while V is not empty do
3:      v ← random node from V
4:      visitedNodes ← (v, count_v = 1)
5:      for num_walks number of times do
6:          v_vis ← random node from visitedNodes
7:          for walk_length number of times do
8:              v' ← random neighbor of v_vis
9:              visitedNodes ← (v', count_v' ++)
10:     B ← empty block
11:     for v ∈ visitedNodes do
12:         if count_v > freq then
13:             B ← v
14:     V \ convex_hull(B)
15:     block B is the convex hull of nodes in B
16:     if density of B > d then
17:         add B to blocks
18: return blocks
```

1) Random walk algorithm: In this phase we represent the tensor $\boldsymbol{\mathcal{X}}$ with a graph $G(V, E)$. For every $x_{ijk} = 1$ we have a node $v_{ijk} \in V$. Two nodes v_{ijk} and v_{pqr} are connected by an edge (v_{ijk}, v_{pqr}) if (i, j, k) and (p, q, r) differ in exactly one coordinate. Observe, that a node v_{ijk} is connected to all nodes in V that are in the same fiber as v_{ijk} in any mode of $\boldsymbol{\mathcal{X}}$. Moreover, a monochromatic block in $\boldsymbol{\mathcal{X}}$ corresponds to a subgraph of G with radius at most 3. In case of noisy data, blocks are not perfectly monochromatic and some of the nodes in V might be missing. Still, if the blocks are fairly dense, the radius of the corresponding subgraph is not too big. More precisely, if v_{ijk} is a node that participates in a block of density d, the probability of a random neighbor of v_{ijk} also participating in that block is $\frac{d}{d+d'}$, where d' is the density of the full tensor. Thus, a random neighbor of a node inside a dense block $\boldsymbol{\mathcal{B}}$ is with high probability also in $\boldsymbol{\mathcal{B}}$.

RandomWalk (Algorithm 1) takes as an input the data tensor $\boldsymbol{\mathcal{X}}$, parameters controlling the length and number of the random walks, and the minimum density of the resulting blocks. After creating the graph $G(V, E)$ it finds a block $\boldsymbol{\mathcal{B}}$ in every iteration of the algorithm by means of executing random walks. Nodes that have been assigned to $\boldsymbol{\mathcal{B}}$ are removed from V, resulting in a smaller graph $G'(V - V_{\boldsymbol{\mathcal{B}}}, E')$ on which the subsequent random walks are executed.

The block $\boldsymbol{\mathcal{B}}$ is found by executing a number of random walks on G. The first walk is initiated from a random node in V, but the subsequent walks start from a random already-visited node. This ensures that once we hit a block $\boldsymbol{\mathcal{B}}$ with a walk, the consecutive walks start with higher and higher probability from within that block. The length and number of the walks is given as an input to the algorithm. After executing the set of walks, we create block $\boldsymbol{\mathcal{B}}$ as the convex hull of the nodes that have been visited more than the average number of times in this set. Finally, we accept $\boldsymbol{\mathcal{B}}$ only if it has density above a user-specified threshold d. Before proceeding with the next iteration of RandomWalk we remove all nodes corresponding to $\boldsymbol{\mathcal{B}}$, regardless of whether $\boldsymbol{\mathcal{B}}$ was accepted.

Running time of RANDOMWALK. The crux of this algorithm is that the running time of every iteration of the algorithm is fixed and depends only on the number and length of the walks. How often we have to re-start the walks depends on how quickly we remove the nodes from the graph, but the worst-case running time is bound by $O(|V|) = O(|\boldsymbol{\mathcal{X}}|)$. However, if $\boldsymbol{\mathcal{X}}$ contains several dense blocks, then the running time is significantly less, since all nodes corresponding to cells in the block are removed at the same time. RANDOMWALK is easily paralellizable (see [4] for details).

2) BLOCKMERGE *Algorithm:* The RANDOMWALK algorithm is a fast method, but it is only able to reliably find the most prominent blocks. If a block is too small, the random walks might visit it as a part of a bigger sparse (and hence rejected) block. It can also happen that while most part of a block is found by RANDOMWALK, some of its slices are not discovered. Therefore the second part of our algorithm, BLOCKMERGE, executes two tasks. First it finds smaller monochromatic blocks that for some reason are undiscovered, and adds these to the set of blocks as well. Then the algorithm has a merging phase, where it tries to merge the blocks found so far.

The input for BLOCKMERGE is the same data tensor $\boldsymbol{\mathcal{X}}$ given to the RANDOMWALK algorithm, the blocks already found, and the minimum density d. As its first step, the algorithm will find all *non-trivial* monochromatic blocks of $\boldsymbol{\mathcal{X}}$ that are not yet included in any of the blocks found earlier. A monochromatic block is non-trivial if its volume and dimensions are above some user-defined thresholds (e.g. all modes have at least 2 dimensions). We find these non-trivial blocks in a greedy fashion. We start with *singletons*: elements $x_{ijk} = 1$ that do not belong into any block, and do an exhaustive search of its neighbors (singletons sharing at least one coordinate with it) to find all monochromatic non-trivial blocks containing x_{ijk}. Any singleton not contained in a non-trivial block after this search is regarded as noise and discarded.

The second part of BLOCKMERGE is to try and merge any blocks found so far so that we get larger dense blocks. Each block $\boldsymbol{\mathcal{B}}$ is defined by three sets of indices, I, J, and K, giving the row, column, and tube indices of this block. When we merge two blocks, $\boldsymbol{\mathcal{B}}$ and $\boldsymbol{\mathcal{C}}$, with indices given by $(I_{\boldsymbol{\mathcal{B}}}, J_{\boldsymbol{\mathcal{B}}}, K_{\boldsymbol{\mathcal{B}}})$ and $(I_{\boldsymbol{\mathcal{C}}}, J_{\boldsymbol{\mathcal{C}}}, K_{\boldsymbol{\mathcal{C}}})$, respectively, the resulting block $\boldsymbol{\mathcal{B}} \boxplus \boldsymbol{\mathcal{C}}$ has its indices given by $(I_{\boldsymbol{\mathcal{B}}} \cup I_{\boldsymbol{\mathcal{C}}}, J_{\boldsymbol{\mathcal{B}}} \cup J_{\boldsymbol{\mathcal{C}}}, K_{\boldsymbol{\mathcal{B}}} \cup K_{\boldsymbol{\mathcal{C}}})$. (This is equivalent on taking the convex hull of $\boldsymbol{\mathcal{B}} \vee \boldsymbol{\mathcal{C}}$, ensuring again that the block is rank-1.) The way we merge two blocks means that the resulting block can, and typically will, include elements that were not in either of the merged blocks. Therefore, we will only merge two blocks if the joint density of 1s and elements already included in the other blocks in the area not in either of merged blocks is higher than the density parameter d. (Please see [4] for implementation details.)

Running time of the BLOCKMERGE algorithm. Let the densest fiber in $\boldsymbol{\mathcal{X}}$ have $b = \max\{n, m, l\} \times d$ ones. Observe that any nontrivial monochromatic block is defined exactly by 2 of its cells. Thus for a cell x_{ijk} we can compute all nontrivial monochromatic blocks containing it in b^2 time by checking all blocks defined by pairs of ones in fibers i, j and k. This checking takes constant time. Hence, the first part of the algorithm takes $O(Bb^2)$ time if there are B trivial blocks in the data. In worst case $B = |\boldsymbol{\mathcal{X}}|$. The second part of the algorithm is the actual merging of blocks. If there are D blocks at the begin of this phase, we will try at most $\binom{D}{2}$ merges. The time it takes to check whether to merge depends on the size of the two blocks involved. Executing the merge $\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{B}} \boxplus \boldsymbol{\mathcal{C}}$ takes at most $|\boldsymbol{\mathcal{A}}|$ time. In worst case $|\boldsymbol{\mathcal{A}}| = |\boldsymbol{\mathcal{X}}|$. As a result, a very crude upper bound on the running time can be given as $O(|\boldsymbol{\mathcal{X}}|(b^2 + D^3))$.

Algorithm 2 BLOCKMERGE algorithm for merging blocks.

Input: Data $\boldsymbol{\mathcal{X}}$, threshold d, blocks $B = \{\boldsymbol{\mathcal{B}}_1, \boldsymbol{\mathcal{B}}_2, \ldots, \boldsymbol{\mathcal{B}}_r\}$ from random walk
Output: Final blocks $\boldsymbol{\mathcal{B}}_1, \boldsymbol{\mathcal{B}}_2 \ldots \boldsymbol{\mathcal{B}}_k$

1: find all non-trivial monochromatic blocks $\boldsymbol{\mathcal{B}}$ of size at least 2-by-2-by-2 not included in blocks in B
2: **for** $\boldsymbol{\mathcal{B}}$ is a non-trivial monochromatic block **do**
3: add $\boldsymbol{\mathcal{B}}$ to B
4: let Q be a queue of all the blocks in B
5: **while** Q is not empty **do**
6: $\boldsymbol{\mathcal{B}} \leftarrow Q.\texttt{pop}$
7: **for all** $\boldsymbol{\mathcal{C}}$ that shares co-ordinates with $\boldsymbol{\mathcal{B}}$ in at least one mode **do**
8: compute the density of $\boldsymbol{\mathcal{B}} \boxplus \boldsymbol{\mathcal{C}}$
9: **if** density $> d$ **then**
10: $Q.\texttt{push}(\boldsymbol{\mathcal{B}} \boxplus \boldsymbol{\mathcal{C}})$
11: replace $\boldsymbol{\mathcal{B}}$ and $\boldsymbol{\mathcal{C}}$ in B with $\boldsymbol{\mathcal{B}} \boxplus \boldsymbol{\mathcal{C}}$
12: **break**
13: **return** B

IV. FROM BLOCKS TO FACTORIZATIONS

The WALK'N'MERGE algorithm returns us a set of rank-1 tensors, corresponding to dense blocks in the original tensor. To obtain the final decompositions, we have to do some additional post-processing.

A. Ordering and Selecting the Final Blocks for the CP-decomposition

We can use all the blocks returned by WALK'N'MERGE to obtain a Boolean CP factorization. The rank of this factorization is equal to the number of blocks WALK'N'MERGE returned. Selecting a subset of these blocks that define a CP decomposition that minimizes the error is NP-hard. Since finding the optimal solution is hard, we will use a greedy algorithm proposed in [5]: We will always select the block that has the highest gain given the already-selected blocks. The gain of a block is defined as the number of not-yet-covered 1s of $\boldsymbol{\mathcal{X}}$ minus the number of not-yet-covered 0s of $\boldsymbol{\mathcal{X}}$ covered by this block, and an element x_{ijk} is covered if $b_{ijk} = 1$ for some already-selected block. The greedy algorithm has the benefit that it gives us an ordering of the blocks, so that if the user wants a rank-k decomposition, we can simply return the first k blocks, instead of having to re-compute the ordering.

B. The MDL Principle and Encoding the Data for the CP decomposition

The greedy algorithm returns an ordering of the columns of matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ of the CP-decomposition. A rank r decomposition then corresponds to using the first r columns of each matrix. In order to choose the best rank r for the decomposition we apply the *Minimum Description Length* (MDL) principle [6] to the encoding of the obtained decomposition.

The intuition behind the MDL principle is that the best model is the one that allows us to compress the data best. To compute the encoding length of the data, we use the two-part MDL: if $\boldsymbol{\mathcal{D}}$ is our data (the data tensor) and $\mathcal{M}$ is a model of it, we aim to minimize $L(\mathcal{M}) + L(\boldsymbol{\mathcal{D}} \mid \mathcal{M})$, where $L(\mathcal{M})$ is the number of bits we need to encode $\mathcal{M}$ and $L(\boldsymbol{\mathcal{D}} \mid \mathcal{M})$ is the number of bits we need to encode the data *given* the model $\mathcal{M}$. In our application, the model $\mathcal{M}$ is the Boolean CP decomposition of the data tensor. As MDL requires us to explain the data exactly, we also need to encode the differences between the data and its (approximate) decomposition; this is the $\boldsymbol{\mathcal{D}} \mid \mathcal{M}$ part.

To compute the encoding length, we modify the *Typed XOR Data-to-Model encoding* for encoding Boolean matrix factorizations [7] to work with tensor factorizations.

The encoding length of the Boolean CP decomposition is simply the length of encoding the size of the factor matrices, the possible length to encode the number of ones in each factor and the number of ones in the remaining error tensor. To compute the overall length we use standard technique from in [7]. For details on how this encoding is done for this work please see our extended work [4].

Having the encoding in place, we can simply compute the change of description length for every rank $1 \leq r \leq B$ and return r where this value is minimized. The corresponding (truncated) matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are the factors of the final CP decomposition that our algorithm returns.

C. Encoding the Data for the Tucker decomposition

Similar to obtaining a CP decomposition from the blocks returned by WALK'N'MERGE these blocks also define a trivial Tucker decomposition of the same tensor. The factor matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ are defined the same way as for the CP. The core $\boldsymbol{\mathcal{G}}$ of the Tucker decomposition is a B-by-B-by-B size tensor with ones in its hyperdiagonal. Our goal is to obtain a more compact decomposition starting from this trivial one by merging some of the factors and adjusting the dimensions and content of the core accordingly. We want to allow the merge of two factors even if it would increase the error slightly. The model $\mathcal{M}$ we want to encode is the Boolean Tucker decomposition of the data tensor, that is, a tuple $(\boldsymbol{\mathcal{G}}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$. Encoding the size of the data tensor as well as the content of the factor matrices is done in the same way as for the CP decomposition. The only additional task we have is to also encode the number of ones in the core tensor. Finally the positive and negative error tensors are identical to the ones in the CP decomposition and hence are encoded in the same way.

Given the encoding scheme we can use a straight forward heuristic to obtain the final Tucker decomposition starting from the trivial one determined by the output of WALK'N'MERGE. In every mode and for every pair of factors we compute the description length of the resulting decompositions if we were to merge these two factors. Ideally we would compute all possible merging sequences and pick the one with the highest overall gain in encoding length. This is of course infeasible, hence we follow a greedy heuristic and apply every merge that yields an improvement (for more details, see [4]).

V. EXPERIMENTAL EVALUATION

A. Other methods and Evaluation Criteria

We used two real-valued scalable CP decomposition methods: CP_APR [8] (implementation from the Matlab Tensor Toolbox v2.5[1]) and PARCUBE [9][2]. CP_APR is an alternating Poisson regression algorithm that is specifically developed for sparse (counting) data (which can be expected to follow the Poisson distribution) with the goal of returning sparse factors. The aim for sparsity and, to some extend, considering the data as a counting data, make this method suitable for comparison; on the other hand, it aims to minimize the (generalized) K–L divergence, not squared error, and binary data is not Poisson distributed.

The other method we compare against, PARCUBE, uses clever sampling to find smaller sub-tensors. It then solves the CP decomposition in this sub-tensor, and merges the solutions back into one. We used a non-negative variant of PARCUBE that expects non-negative data, and returns non-negative factor matrices. PARCUBE aims to minimize the squared error.

To compute the error, we used sum of absolute differences for WALK'N'MERGE and sum-of-squared-errors for the other methods. This presents yet another apples-versus-oranges comparison: the squared error can help the real-valued methods, as it scales all errors less than 1 down, but at the same time, small errors cumulate unlike with fully binary data. To alleviate this problem, we also rounded the reconstructed tensors from CP_APR and PARCUBE to binary tensors. We tried different rounding thresholds between 0 and 1 and selected the one that gave the lowest (Boolean) reconstruction error. With some of the real-world data, we were unable to perform the rounding for the full representation due to time and memory limitations. For these data sets, we estimated the rounded error using stratified sampling, where we sampled $10\,000$ ones and $10\,000$ zeros from the data, computed the error on these, and scaled the results.

B. Synthetic Data

We generated sparse 1000-by-1500-by-2000 synthetic binary tensor as follows: We first fixed parameters for the Boolean rank of the tensor and the noise to apply. We generated three (sparse) factor matrices to obtain the noise-free tensor. As we assume that the rank-1 tensors in the real-world data are relatively small (e.g. synonyms of an entity), the rank-1 tensors we use were approximately of size 16-by-16-by-16, with each of them overlapping with another block. We then added noise to this tensor. We separate the noise in additive and destructive noise. The amount of noise depends on the number of 1s in the noise-free data, that is 10% of destructive noise means that we delete 10% of the 1s, and 20% of additive noise means that we add 20% more 1s. To generate the data, we varied three parameters – rank, additive noise, destructive noise, and overlap of the latent blocks – and created five random copies for each set parameters.

The rank of the decomposition was set to the true rank of the data for all methods. For WALK'N'MERGE we set the merging threshold to $1-(n_d+0.05)$, where n_d was the amount

[1] http://www.sandia.gov/~tgkolda/TensorToolbox/
[2] http://www.cs.cmu.edu/~epapalex/

of destructive noise, the length of the random walks was set to 5, and we only considered blocks of size 4-by-4-by-4 or larger. The results for varying rank and different types of noise are presented in Figure 1. Varying the amount of overlap did not have any effect on the results of WALK'N'MERGE, and we omit the results. Results for PARCUBE were consistently worse than anything else and they are omitted from the plots.

Rank. For the first experiment (Figure 1(a)) we varied the rank while keeping the additive and destructive noise at 10%. With rank-5 decomposition, WALK'N'MERGE fits to the input data slightly worse than CP_APR (unrounded) but clearly better than $\text{CP_APR}_{0/1}$ (rounded) and $\text{PARCUBE}_{0/1}$, the latter being clearly the worse with all ranks. For larger ranks, WALK'N'MERGE is clearly better than variations of CP_APR. Note that here rank is both the rank of the data and the rank of the decomposition. When comparing the fit to the original data (dashed lines), WALK'N'MERGE is consistently better than the variants of CP_APR or $\text{PARCUBE}_{0/1}$, to the extend that it achieves perfect results for ranks larger than 5.

Additive noise. In this experiment, rank was set to 10, destructive noise to 10%, and additive noise was varied. Results are presented in Figure 1(b). In all results, WALK'N'MERGE is consistently better than any other method, and always recovers the original tensor perfectly.

Destructive noise. For this experiment, rank was again set to 10 and additive noise to 10% while the amount of destructive noise was varied (Figure 1(c)). The results are similar to those in Figure 1(b), although it is obvious that the destructive noise has the most significant effect on the quality of the results.

Discussion. In summary, the synthetic experiments show that when the Boolean structure is present in the data, WALK'N'MERGE is able to find it – in many cases even exactly. That CP_APR is not able to do that should not come as a surprise as it does not try to find such structure. That $\text{PARCUBE}_{0/1}$ is almost consistently the worse is slightly surprising (and the results from the unrounded PARCUBE were even worse). From Figure 1(b) we can see that the results of $\text{PARCUBE}_{0/1}$ start improving when the amount of additive noise increases. This hints that PARCUBE's problems are due to its sampling approach not performing well on these extremely sparse tensors.

C. Real-World Data

1) Datasets: To assess the quality of our algorithm, we tested it with three real-world data sets: The Enron data[3] (146-by-146-by-38) contains information about who sent e-mail to whom (rows and columns) per months (tubes). The TracePort data set[4] (501-by-10 266-by-8 622) contains anonymized passive traffic traces (source and destination IP and port numbers) from 2009. The Facebook data set[5] [10] (63 891-by-63 890-by-228) contains information about who posted a message on whose wall per weeks.

2) CP Factorization: We start by reporting the reconstruction errors with CP decompositions using the same algorithms we used with the synthetic data. The results are in Table I.

[3] http://www.cs.cmu.edu/~enron/

[4] http://www.caida.org/data/passive/passive_2009_dataset.xml

[5] The data is publicly available from the authors of [10], see http://socialnetworks.mpi-sws.org

TABLE I. RECONSTRUCTION ERRORS ROUNDED TO THE NEAREST INTEGER. NUMBERS PREFIXED WITH * ARE OBTAINED USING SAMPLING.

	Enron	TracePort		Facebook
Algorithm	$r = 12$	$r = 15$	$r = 1370$	$r = 15$
WALK'N'MERGE	1 753	10 968	7 613	612 314
PARCUBE	2 089	33 741	$4 \cdot 10^{55}$	$8 \cdot 10^{140}$
$\text{PARCUBE}_{0/1}$	1 724	11 189	$* 2 \cdot 10^{7}$	* 1 788 874
CP_APR	1 619	11 069	5 230	626 349
$\text{CP_APR}_{0/1}$	1 833	11 121	* 1 886	* 626 945

The Enron data reverses the trend we saw with the synthetic data, as both CP_APR and $\text{PARCUBE}_{0/1}$ are better than WALK'N'MERGE, possible because Enron does not have strong Boolean CP structure. With TracePort and $r = 15$ WALK'N'MERGE is again the best, if only slightly. With $r = 1370$, WALK'N'MERGE improves, but not as much as CP_APR and especially $\text{CP_APR}_{0/1}$. The very high rank probably lets CP_APR to better utilize the higher expressive power of continuous factorizations, explaining the significantly improved results. For Facebook and $r = 15$ the situation is akin to TracePort with $r = 15$ in that WALK'N'MERGE is the best followed directly with CP_APR. PARCUBE's errors were off the charts with both TracePort ($r = 1370$) and Facebook; we suspect that the extreme sparsity (and high rank) fooled its sampling algorithm.

Running time. The running time of WALK'N'MERGE depends on the structure of the input tensor and on the parameters used, complicating systematic studies of running times. But to give some idea, we report the running times for the Facebook data, as that is the biggest data set we used. The fastest algorithm for $k = 15$ was PARCUBE, finishing in a matter of minutes (but note that it gave very bad results). Second-fasters was WALK'N'MERGE. We tried different density thresholds d, effecting the running time. The fastest was $d = 0.2$, when WALK'N'MERGE took 85 minutes, the slowest was $d = 0.70$, taking 277 minutes, and the average was 140 minutes. CP_APR was in between these extremes, taking 128 minutes for one run. Note, however, that WALK'N'MERGE didn't return just the $r = 15$ decomposition, but in fact all decompositions up to $r = 3300$. Neither PARCUBE or CP_APR was able to handle so large ranks with the Facebook data.

3) Tucker Decomposition: For the Enron dataset we obtained a decomposition with a core of size 9-by-11-by-9 from the MDL step. While this might feel small, the reconstruction error was 1775, i.e. almost as good as the best BCP decomposition. (MDL does not try to optimize the reconstruction error, but the encoding length.)

With the Tucker decomposition, we also used fourth data set (see [11]). This data set contains noun phrase–context pattern–noun phrase triples that are observed forms of subject–relation–object triples. With this data our goal is to find a Boolean Tucker decomposition such that the core $\mathcal{G}$ corresponds to the latent subject–relation–object triples and the factor matrices tell us which surface forms are used for which entity and relation. The size of the data is 39 500-by-8 000-by-21 000 and it contains 804 000 surface term triplets. The running time of WALK'N'MERGE on this dataset was 52 minutes, and computing the Tucker decomposition took another 3 hours.

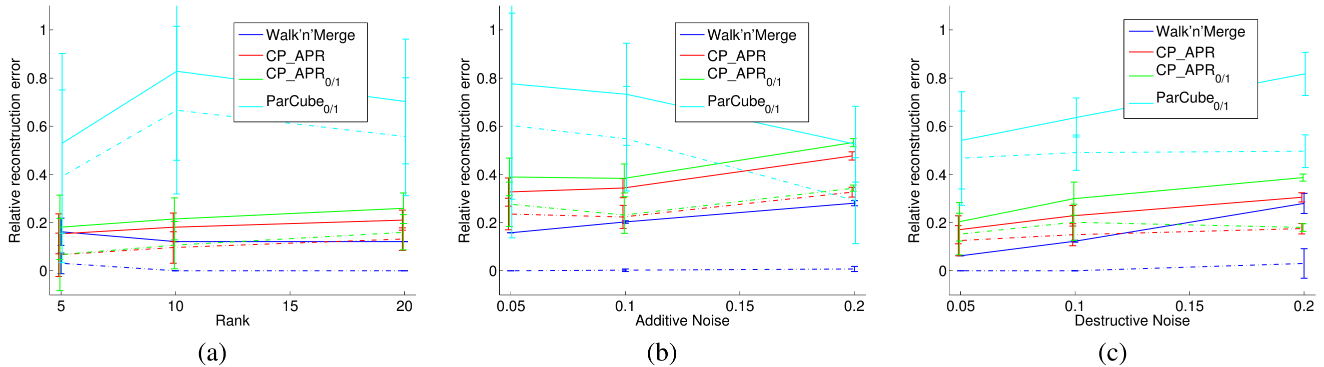

Fig. 1. Results on synthetic data sets using CP-type decompositions. (a) Varying rank. (b) Varying additive noise. (c) Varying destructive noise. Solid lines present the relative reconstruction error w.r.t. input tensor; dashed lines present it w.r.t. the original noise-free tensor. All points are mean values over five random datasets and the width of the error bars is twice the standard deviation.

An example of a factor of the subjects is {`claude de lorimier`, `de lorimier`, `louis`, `jean-baptiste`}, corresponding to Claude-Nicolas-Guillaume de Lorimier, a Canadian politician from the 18th Century (and his son, Jean-Baptiste). An example of an object-side factor is {`borough of lachine`, `villa st. pierre`, `lachine quebec`}, corresponding to the borough of Lachine in Quebec, Canada, and an example of a factor in the relations is {`was born was`, `[[det]] born in`}. In the core $\mathcal{G}$ the element corresponding to these three factors is 1, that is, according to our algorithm, de Lorimier was born in Lachine, Quebec – as he was.

VI. Related Work

Normal tensor factorizations are well-studied, dating back to the late Twenties. The Tucker and CP decompositions were proposed in Sixties [12] and Seventies [13], [14], respectively. The topic has nevertheless attained growing interest in recent years, both in numerical linear algebra and computer science communities. For a comprehensive study of recent work, see [15], and the recent work on scalable factorizations [9].

One field of computer science that has adopted tensor decompositions is computer vision (see [16], [17]).

The theory of Boolean tensor factorizations was studied in [2], although the first algorithm for Boolean CP factorization was presented in [1]. A related line of data mining research has also studied a specific type of Boolean CP decomposition, where no 0s can be presented as 1s (e.g. [18]). For more on these methods and their relation to Boolean CP factorization, see [2].

VII. Conclusions

We have presented Walk'n'Merge, an algorithm for computing the Boolean tensor factorization of large and sparse binary tensors. Analysing the results of our experiments sheds some light on the strengths and weaknesses of our algorithm. First, it is obvious that it does what it was designed to do, that is, finds Boolean tensor factorizations of large and sparse tensors. But it has its caveats, as well. The random walk algorithm, for example, introduces an element of randomness, and it seems that it benefits from larger tensors. The algorithm, and its running time, is also somewhat sensible to the parameters, possibly requiring some amount of tuning.

References

[1] I. Leenen, I. Van Mechelen, P. De Boeck, and S. Rosenberg, "INDCLAS: A three-way hierarchical classes model," *Psychometrika*, vol. 64, no. 1, pp. 9–24, Mar. 1999.

[2] P. Miettinen, "Boolean Tensor Factorizations," in *ICDM '11*, 2011, pp. 447–456.

[3] R. Bělohlávek, C. Glodeanu, and V. Vychodil, "Optimal Factorization of Three-Way Binary Data Using Triadic Concepts," *Order*, Mar. 2012.

[4] D. Erdős and P. Miettinen, "Scalable Boolean tensor factorizations using random walks," arXiv, Tech. Rep., 2013.

[5] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila, "The Discrete Basis Problem," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 10, pp. 1348–1362, Oct. 2008.

[6] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 5, pp. 465–471, Sep. 1978.

[7] P. Miettinen and J. Vreeken, "MDL4BMF: Minimum description length for boolean matrix factorization," Max-Planck-Institut für Informatik, Tech. Rep. MPI–I–2012–5-001, Jun. 2012.

[8] E. C. Chi and T. G. Kolda, "On Tensors, Sparsity, and Nonnegative Factorizations," *SIAM J. Matrix Anal. Appl.*, vol. 33, no. 4, pp. 1272–1299, Dec. 2012.

[9] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "ParCube: Sparse Parallelizable Tensor Decompositions," in *ECML PKDD '12*, 2012, pp. 521–536.

[10] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the Evolution of User Interaction in Facebook," in *WOSN '09*, 2009, pp. 37–42.

[11] D. Erdős and P. Miettinen, "Discovering facts with Boolean tensor tucker decomposition," in *CIKM*, 2013.

[12] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[13] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an N-way generalization of 'Eckart-Young' decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[14] R. A. Harshman, "Foundations of the PARAFAC procedure: Models and conditions for an 'explanatory' multimodal factor analysis," UCLA Working Papers in Phonetics, Tech. Rep., 1970.

[15] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[16] A. Shashua and T. Hazan, "Non-negative tensor factorization with applications to statistics and computer vision," in *ICML '05*, 2005.

[17] Y.-D. Kim and S. Choi, "Nonnegative Tucker Decomposition," in *CVPR '07*, 2007, pp. 1–8.

[18] L. Cerf, J. Besson, C. Robardet, and J.-F. Boulicaut, "Closed patterns meet n-ary relations," *ACM Trans. Knowl. Discov. Data*, vol. 3, no. 1, 2009.