Real-Time Centroid Decomposition of Streams of Time Series

Master Thesis

OLIVER STAPLETON

University of Bern

Supervisor
Dr. Mourad KHAYATI
eXascale Infolab, Department of Informatics, University of Fribourg

Co-Supervisor
Prof. Dr. Philippe CUDRÉ-MAUROUX
eXascale Infolab, Department of Informatics, University of Fribourg

February 7, 2017

Abstract

The Centroid Decomposition (CD) is a matrix decomposition technique that has been successfully applied to recover blocks of missing values in batches of time series. The state-of-the-art solution to compute CD achieves quadratic run-time complexity with the length of time series. Therefore, the CD technique is not efficient when applied to streams of time series. The inefficiency is mainly due to the fact that each update to the streams requires a re-computation of the entire CD from scratch. In this thesis, we introduce a novel technique called cached-CD that achieves a linear run-time complexity with the length of the streams. The proposed solution i) allows an efficient computation of the CD for streams of time series and ii) speeds up the recovery of blocks of missing values in batches of time series. The cached-CD leverages the fact that, between two consecutive executions of the CD algorithm, the input matrices for both use cases (streaming and batch recovery) change only slightly, yielding very similar maximizing sign vectors for consecutive executions. Through caching of these maximizing sign vectors and re-using them as initializing values during the next execution, the cached-CD reduces the run-time complexity from quadratic to linear. We perform experiments on real-world time series data to evaluate i) the efficiency of our solution in comparison to existing solutions for computing the CD for streams of time series and ii) the improved run-time of the recovery of missing blocks for batch time series performed by the cached-CD in comparison to existing solutions.

Additionally, we have developed a graphical tool called c-ReVival that implements our proposed solution. c-ReVival allows i) to perform the computation of cached-CD for streams of time series in a scalable fashion, ii) to visualize the recovery of blocks of missing values in time series using cached-CD and iii) to analyze the main properties of the CD technique.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Mourad Khayati, for his constant and valuable support, feedback and advice throughout the work on this thesis.

I would also like to extend my gratitude to Prof. Dr. Philippe Cudré-Mauroux for his co-supervision of the thesis.

Thank you also to the sources that provided us the real-world data sets for c-ReVival and the evaluation part. Namely, these were the Federal Office for the Environment and the Federal Office of Meteorology and Climatology.

Last but not least, I would like to thank my family and friends for their patience and encouragement during the time I spent working on the thesis.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Time series are sequences of coupled data points consisting of a time stamp and an associated value. Time series usually contain sensory, transactional, or web data. Missing values can occur in time series due to multiple reasons, such as temporary connectivity issues between a sensor and its data base, or machine failure that could result in a measurement not being recorded or getting lost. Many studies have shown that the a priori recovery of these missing values is beneficial in many data analysis applications, e.g., compression, prediction, similarity, etc [1–3]. In the case of discrete missing values, basic statistical methods such as a linear interpolation might suffice to accurately recover the missing values. However, when dealing with longer blocks of missing values, the application of basic statistical methods yields poor recovery accuracy. In such cases, it is pivotal to benefit from the correlation to other time series. The recovery becomes more challenging in the case of streams of time series.

The Centroid decomposition (CD) has been been applied to recover blocks of missing values in correlated time series [4]. CD is able to accurately reconstruct the type, the shape and the amplitude of the missing blocks by learning from the history of the time series that contains the missing blocks together with the history of other correlated time series. At a formal level, the CD technique decomposes an input matrix $\mathbf{X}$ into two separate matrices, i.e., the loading matrix $\mathbf{L}$ and the relevance matrix $\mathbf{R}$, such that $\mathbf{X} = \mathbf{L} \times \mathbf{R}^T$. The naive application of CD for streams of time series requires to recompute from scratch the new $\mathbf{L}$ and $\mathbf{R}$ each time new data arrives. Due to its quadratic run-time with the number of values per time series, the application of CD is inefficient when dealing with long streams of time series.

1.1 Context of work and motivation

The data analysis on batches of historical data has been extensively studied in the literature [1–3]. Up to a certain length of time series, the proposed algorithms are efficient. However, there are limitations when applying an algorithm with non linear (e.g., quadratic) run-time complexity to large time series with potentially millions of values or more. Additionally, due to the emergence of much more connected technologies, such as the Internet of Things (IoT), we have faced a paradigm shift towards real-time data analysis in recent years. This has caused a massive increase in the volume and velocity of data being produced. A lot of this data comes as data streams [5]. The latter are nothing else than ever growing time series that receive updates at a certain, mostly constant, interval. The analysis of data streams is similar to

that of time series, except that it happens in real-time and therefore is more expensive to perform [6, 7]. For example, the CD technique, previously described, is one of these data analysis techniques that are inefficient when applied to streams of time series, mainly due to the fact that the existing solution to compute CD is not incremental. In fact, the state-of-the-art solution recomputes the decomposition of the entire stream of time series from scratch each time a new set of values is added, even though the impact of the addition is extremely minor.

1.2 Contributions

The main contributions of this thesis are:

a) Cached-CD: We propose a novel and efficient technique called the cached Centroid Decomposition (cached-CD) that reduces the run-time complexity, to compute CD for streams of time series, from quadratic to linear. More specifically, cached-CD generally improves the computation efficiency in cases where it is necessary to iteratively recompute the CD in between minor updates to the input matrix. This new approach extends the regular CD by caching the so called maximizing sign vectors of CD($\mathbf{X}_0$) (the Centroid Decomposition of matrix $\mathbf{X}$ at time 0) and reuses them during the computation of CD($\mathbf{X}_1$), after the input matrix $\mathbf{X}$ has been altered slightly, e.g. when only selected values in the input matrix have changed or a new row has been added to it.

The efficiency improvement of cached-CD speeds up both the computation of the CD for streams of time series and the recovery of missing values in batches of time series. For the streaming use case, the cached-CD reduces the run-time complexity from quadratic to linear with respect to the number of values per time series. As an example, for a stream of three time series, where we already know the CD for $n-1$, the cached-CD computes the CD 2'500 times faster than the current state-of-the-art CD algorithm at $n = 50k$ (cf. Chapter 6). For the recovery use case, cached-CD speeds up the computation by a factor that depends on the amount of separate necessary computations of CD for the recovery. For example, using the cached-CD for the recovery on an input matrix consisting of four time series with $10k$ values each, cached-CD based recovery achieves a run-time improvement by a factor of 10 (cf. Chapter 6).

b) Updating-CD: We implement the updating Centroid Decomposition (updating-CD), an approach of on-line computing the CD of streams of time series and compare its performance against our proposed cached-CD algorithm.

c) c-ReVival: We implement an online graphical tool called c-ReVival (Cached RecoVery of missing Values). c-ReVival allows i) to visualize the recovery process of missing values in real-world data using cached-CD, ii) to compare the obtained decomposition on a stream of time series using cached-CD against the regular-CD and the updating-CD, and iii) to illustrate the main properties of the CD technique.

1.3 Outline

The structure of this thesis is as follows. Chapter 2 introduces the used notation and provides the background to the thesis by covering the CD technique. Chapter 3 covers the updating-CD technique and describes an implementation of the algorithm together with a running example. In Chapter 4, the novel cached-CD technique is introduced together with possible implementations of algorithms for the streaming and recovery use cases. Chapter 5 introduces the c-ReVival tool and describes the purpose and functionality of its components. The performance of the cached-CD algorithms for both streaming computation and recovery are evaluated in Chapter 6. Additionally, some observations made with the c-ReVival tool are discussed. Chapter 7 concludes the thesis and highlights possible future work that could be conducted on the topic of using the cached-CD for the on-line recovery of missing values in time series.

2
Background

This chapter serves as an introduction to the Centroid Decomposition. It covers the relevant background to the techniques and algorithms on which we base our own propositions, implementations and improvements.

2.1 Notations

The following notations are used throughout the thesis. Variables in bold upper-case letters refer to matrices, regular upper-case letters to vectors (possibly rows and columns of matrices), and lower-case letters to individual elements of a vector or matrix. Double subscript indices denote rows and columns (in that order) of a matrix, e.g., vector X_{i*} is the i-th row of matrix $\mathbf{X}$ and x_{ij} is the j-th element of the row X_{i*}. Single subscript indices are used to distinguish either different states of the variable based on the current iteration/time, or to distinguish between multiple variables using the same denotation. E.g., $\mathbf{X}_0$ and $\mathbf{X}_1$ denote matrix $\mathbf{X}$ at time $t = 0$ and $t = 1$, respectively, and $Z_1, ..., Z_m$ denote all m different sign vectors used during the computation of CD($\mathbf{X}$). $\mathbf{X}^T$ stands for the transpose of matrix $\mathbf{X}$.

We consider an $n \times m$ input matrix $\mathbf{X}$ with n rows and m columns. A time series $\mathbf{X}_{*i} = ((t_1, v_1), \ldots, (t_n, v_n))$ is a set of n temporal values that are ordered with respect to their timestamps t_j ($j \in \{1, \ldots, n\}$). When dealing with multiple time series, we assume the time series to be aligned, i.e. having the same set of timestamps, and we omit the timestamps in time series altogether by writing $\mathbf{X}_{*i} = (4, -1, 3, 2)$ instead of $\mathbf{X}_{*i} = ((0, 4), (1, -1), (2, 3), (3, 2))$. The time series are considered columns of the input matrix. Thus, vector X_{*i} is the column of $\mathbf{X}$ that contains all the values of i-th time series, and vector X_{j*} is a row that contains the j-th value of each time series in $\mathbf{X}$.

2.2 Centroid Decomposition (CD)

2.2.1 Definition

The Centroid Decomposition (CD)[8] is a matrix decomposition technique that decomposes a given input matrix $\mathbf{X}$ into a loading matrix $\mathbf{L}$ and a relevance matrix $\mathbf{R}$, such that $\mathbf{X} = \mathbf{L} \times \mathbf{R}^T$.

Algorithm 1 describes the pseudo code of the CD technique. The algorithm takes as input an $n \times m$ matrix $\mathbf{X}$ and computes the $n \times m$ loading matrix $\mathbf{L}$ and the $m \times m$ relevance matrix $\mathbf{R}$ by iteratively appending intermediate column vectors to $\mathbf{L}$ and $\mathbf{R}$. For each of the m iterations i, the maximizing sign vector Z_i for the current state of $\mathbf{X}$ is computed. $Z_i \in \{-1, 1\}^n$ is the sign vector that produces the maximal centroid value, $\|\mathbf{X}^T \times Z_i\|$, in iteration i.

Algorithm 1: CD($\mathbf{X}$)

Input : $n \times m$ matrix $\mathbf{X}$
Output : $n \times m$ loading matrix $\mathbf{L}$, $m \times m$ relevance matrix $\mathbf{R}$

```
for i ← 1 to m do
    Z_i ← MaximizingSignVector(X);
    L_*i ← NextColumnL(X, Z_i);
    R_*i ← NextColumnR(X, Z_i);
    X ← X − L_*i × R_*i^T;
end
return L, R;
```

Figure 2.1 shows schematically the Centroid Decomposition of an input matrix $\mathbf{X}$. Note how, for each iteration i, a maximizing sign vector Z_i, loading column vector L_{*i} and relevance column vector R_{*i} is computed.

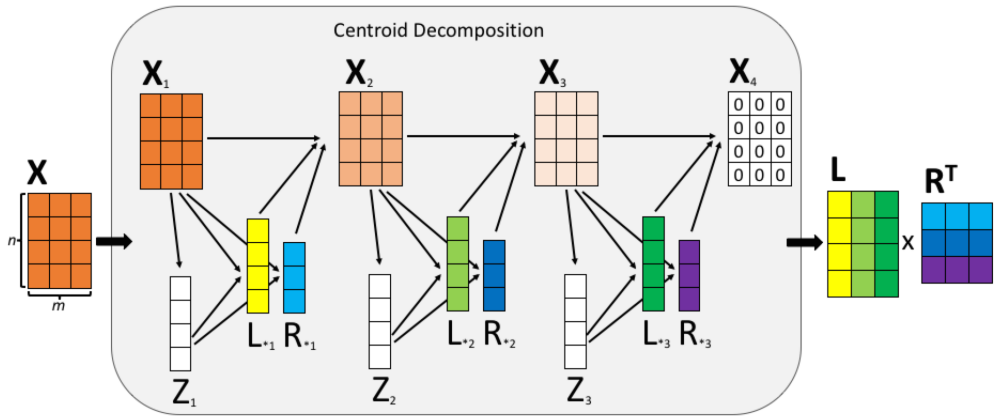


Figure 2.1: Schematic diagram of CD

2.2.2 Maximizing sign vector

A sign vector $Z \in \{-1, 1\}^n$ is a column vector that consists of only the values 1 and -1. For any given matrix $\mathbf{X}$ with dimensions $n \times m$, the maximizing sign vector Z is the vector $Z \in \{-1, 1\}^n$ for which $\|\mathbf{X}^T \times Z\|$ is maximized. A maximizing sign vector is computed for each iteration of the CD algorithm. Thus, the efficiency of CD heavily depends on the computation of the maximizing sign vector. Strategies to compute the maximizing sign vectors usually start by initializing some sign vector Z, and iteratively flipping the sign of individual elements in Z until the maximizing sign vector is found (cf. Section 6.2.2).

2.2.3 Scalable Sign Vector (SSV)

The SSV algorithm [9] is an efficient and greedy solution to compute the maximizing sign vector for a given input matrix $\mathbf{X}$. SSV has a linear space complexity and thus, outperforms existing techniques to compute the maximizing sign vector such as the Quadratic Sign Vector (QSV) algorithm [8]. Algorithm 2 describes the pseudo code of SSV. It starts by initializing column vector $Z^T = [1, 1, \ldots, 1]$ of length n, after which it iteratively computes column vector V out of $\mathbf{X}$ and Z. Next, the signs of the values in the vectors Z and V for each position $j \in \{0, 1, \ldots, n\}$ are compared. SSV greedily flips the sign of z_{pos} (the element at position pos in Z) from plus to minus, or vice-versa, such that $|v_{pos} \cdot z_{pos}|$ is maximal. Once Z and V have pair-wise the same sign at all positions, the SSV terminates and returns Z, the maximizing sign vector.

Algorithm 2: SSV($\mathbf{X}$)

Input : $n \times m$ matrix $\mathbf{X}$
Output : maximizing sign vector Z (length n)

```
1  pos ← 0;
2  repeat
3      if pos = 0 then
4          Z^T ← [1, ..., 1];
5      else
6          changeSign(z_pos);
7      end
8      V ← computeV(Z, X);
9      pos ← 0;
10     find pos such that sign(z_pos) ≠ sign(v_pos) and |v_pos · z_pos| is maximized;
11 until pos = 0;
12 return Z;
```

Since we will introduce an another algorithm of finding maximizing sign vectors later, to avoid confusion, we define regular-CD as the CD (Algorithm 1) that uses SSV on line 2 to find the maximizing sign vectors.

2.3 Recovery of missing values

The Centroid Decomposition technique can be applied to recover missing values in time series data. To achieve this, multiple time series need to first be aligned in respect to their time stamps. Usually, there is exactly one time series with missing values (called the base time series), and several other, complete time series (called reference time series). The time series are merged to a matrix, where each time series represents a column. The RecM algorithm introduced by Khayati et al.[4] iteratively applies the CD algorithm combined with dimensionality reduction to the input matrix to recover (blocks of) missing values in the base time series through correlation.

The pseudo code of RecM is very similar to that seen in Algorithm 7 (cached-RecM), except for line 4, where the regular-CD is used instead of cached-CD.

3

Updating Centroid Decomposition (updating-CD)

In this chapter, we describe a technique, called updating-CD, to compute the CD of an input matrix with updated rows. This solution exploits the properties of the decomposition to avoid recomputing CD from scratch each time new data arrives.

We slightly extend our notations. We denote the matrix containing the time series before appending the new row of values (row vector A) as $\mathbf{X}_0$, and $\mathbf{X}_1$ after appending A. Subsequently, we denote CD($\mathbf{X}_0$) = $\mathbf{L}_0 \times \mathbf{R}_0^T$ and CD($\mathbf{X}_1$) = $\mathbf{L}_1 \times \mathbf{R}_1^T$.

3.1 Idea

Chu and Blevins introduce a rank-1 updating framework [10], that embeds a method for an updated computation of the CD of the updated input matrix $\mathbf{X}_1$, given the row vector A of the update and CD($\mathbf{X}_0$) before the update. This approach is based on the fact that in CD, the loading matrix $\mathbf{L}$ is a stationary point, i.e., CD($\mathbf{L}$) = $\mathbf{L} \times \mathbf{I}^T$ (where $\mathbf{I}$ is the identity matrix). The authors show that computing CD($\mathbf{X}_1$) can be achieved by orthogonal rotations of the loading matrix $\mathbf{L}_0$ (part of CD($\mathbf{X}_0$)) based on A. Namely, the orthogonal rotations are used to construct an intermediate matrix $\mathbf{S}$, from which $\mathbf{L}_1$ and $\mathbf{R}_1$ of the updated matrix $\mathbf{X}_1$ are derived by, among other steps, computing the CD($\mathbf{S}$). The gain that the authors aim to achieve is that $\mathbf{S}$ is very similar to $\mathbf{L}_0$ (a stationary point). They argue that it should theoretically require less computations to compute CD($\mathbf{S}$) than CD($\mathbf{X}_1$), even though $\mathbf{S}$ and $\mathbf{X}_1$ have the same dimensions.

3.2 Implementation

We implemented the updating-CD algorithm based on the approach introduced by Chu and Blevins. Its pseudo code can be seen in Algorithm 3. The algorithm is part of the c-ReVival tool (see Section 5.1.4), where we compare its performance in on-line computation to the cached-CD and the regular-CD.

The algorithm takes as parameters the CD (loading and relevance matrices) of $\mathbf{X}_0$, $\mathbf{L}_0$ and $\mathbf{R}_0$, together with the row vector A that contains the row of values to appended to $\mathbf{X}_0$ (to form $\mathbf{X}_1$). The algorithm terminates by outputting the CD (loading and relevance matrices) of $\mathbf{X}_1$, $\mathbf{L}_1$ and $\mathbf{R}_1$. The algorithm works as follows: On lines 1 and 2, vector Q is built which is used to determine whether or not A is within the

Algorithm 3: updating-CD($\mathbf{L}_0$, $\mathbf{R}_0$, A)

Input : $n \times m$ matrix $\mathbf{L}_0$, $m \times m$ matrix $\mathbf{R}_0$; row vector A (m values to be added)
Output : $(n+1) \times m$ matrix $\mathbf{L}_1$, $m \times m$ matrix $\mathbf{R}_1$

1 $N \leftarrow \mathbf{R}_0^T A$;
2 $Q \leftarrow A - \mathbf{R}_0^T N$;
3 **if** $||Q|| \neq 0$ **then**
4 // A is not in rowspace of $\mathbf{X}_0$
5 $Q \leftarrow scalarDivision(Q, ||Q||)$;
6 $\mathbf{S} \leftarrow$ append as column value 0 to $\mathbf{L}_0$;
7 $U \leftarrow$ append as column value $||Q||$ to N^T;
8 $\mathbf{S} \leftarrow$ append as row vector U to $\mathbf{S}$;
9 **else**
10 // A is within rowspace of $\mathbf{X}_0$
11 $Q \leftarrow 0$;
12 $\mathbf{S} \leftarrow$ append as row vector N^T to $\mathbf{L}_0$;
13 **end**
14 $\mathbf{L}_S$, $\mathbf{R}_S \leftarrow$ *regular-CD*($\mathbf{S}$);
15 $\mathbf{L}_1 \leftarrow \mathbf{L}_S$;
16 $\mathbf{V} \leftarrow$ append as column vector Q to $\mathbf{R}_0$;
17 $\mathbf{R}_1 \leftarrow \mathbf{V} \times \mathbf{R}_S$;
18 **return** $\mathbf{L}_1$, $\mathbf{R}_1$;

rowspace of $\mathbf{X}_0$. The construction of matrix $\mathbf{S}$ and modification of vector Q depend on this (either on lines 5 to 8, or 11 and 12). After $\mathbf{S}$ has been constructed, the CD($\mathbf{S}$)=$\mathbf{L}_S \times \mathbf{R}_S$ is computed. Finally, based on $\mathbf{L}_S$, $\mathbf{R}_S$, $\mathbf{R}_0$ and A, the Centroid Decomposition of $\mathbf{X}_1$ (loading matrix $\mathbf{L}_1$ and relevance matrix $\mathbf{R}_1$) is constructed (not computed!) and outputted.

3.3 Running example

Example 1. *Consider a matrix* $\mathbf{X}_0 = \{X_{*1}, X_{*2}, X_{*3}\}$ *consisting of three streaming time series at time* $t = 0$. *There are four values in each time series:* $X_{*1} = (1, 4, -3, -4)$, $X_{*2} = (2, 4, -4, 3)$ *and* $X_{*3} = (2, 6, -3, -2)$. *For* $\mathbf{X}_0$, *we happen to know the Centroid Decomposition (loading matrix* $\mathbf{L}_0$ *and relevance matrix* $\mathbf{R}_0$*):*

$$CD(\underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ -3 & -4 & -3 \\ -4 & 3 & -2 \end{bmatrix}}_{\mathbf{X}_0}) = \underbrace{\begin{bmatrix} 2.73 & 1.22 & 0.22 \\ 8.09 & 1.42 & 0.68 \\ -5.41 & -1.97 & 0.90 \\ -2.79 & 4.61 & 0.00 \end{bmatrix}}_{\mathbf{L}_0} \times \underbrace{\begin{bmatrix} 0.63 & 0.37 & 0.68 \\ -0.49 & 0.87 & -0.02 \\ -0.60 & -0.32 & 0.73 \end{bmatrix}}_{\mathbf{R}_0{}^T}$$

At this point, a value is added to each time series. Combined, these added values form a row vector

$A = [-1, 1, -3]$ *that is appended to* $\mathbf{X}_0$*, forming* $\mathbf{X}_1$*.*

$$\underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ -3 & -4 & -3 \\ -4 & 3 & -2 \end{bmatrix}}_{\mathbf{X}_0}, \underbrace{\begin{bmatrix} -1 & 1 & -3 \end{bmatrix}}_{A} \rightarrow \underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ -3 & -4 & -3 \\ -4 & 3 & -2 \\ -1 & 1 & -3 \end{bmatrix}}_{\mathbf{X}_1}$$

We have all the required parameters $(\mathbf{L}_0, \mathbf{R}_0, A)$ *to execute updating-CD (Algorithm 3).* A *is within the rowspace of* $\mathbf{X}_0$*, so the construction of* Q *and* $\mathbf{S}$ *(lines 11 and 12) gives the following:*

$$Q = \begin{bmatrix} 0 \end{bmatrix}, \mathbf{S} = \begin{bmatrix} 2.73 & 1.22 & 0.22 \\ 8.09 & 1.42 & 0.68 \\ -5.41 & -1.97 & 0.90 \\ -2.79 & 4.61 & 0.00 \\ -2.31 & 1.42 & -1.90 \end{bmatrix}$$

Next, (regular) CD($\mathbf{S}$*) is computed:*

$$CD(\underbrace{\begin{bmatrix} 2.73 & 1.22 & 0.22 \\ 8.09 & 1.42 & 0.68 \\ -5.41 & -1.97 & 0.90 \\ -2.79 & 4.61 & 0.00 \\ -2.31 & 1.42 & -1.90 \end{bmatrix}}_{\mathbf{S}}) = \underbrace{\begin{bmatrix} -2.65 & 1.36 & 0.32 \\ -8.01 & 1.90 & 0.45 \\ 5.17 & -2.59 & 0.76 \\ 3.07 & 4.22 & 1.33 \\ 2.56 & 1.63 & -1.33 \end{bmatrix}}_{\mathbf{L}_S} \times \underbrace{\begin{bmatrix} -0.99 & 0.09 & -0.07 \\ 0.07 & 0.97 & 0.25 \\ -0.09 & -0.24 & 0.97 \end{bmatrix}}_{{\mathbf{R}_S}^T}$$

Based on the intermediary constructions and computations, $\mathbf{L}_1$ *and* $\mathbf{R}_1$ *can be constructed.*

$$updatingCD(\mathbf{L}_0, \mathbf{R}_0, A) = \underbrace{\begin{bmatrix} -2.65 & 1.36 & 0.32 \\ -8.01 & 1.90 & 0.45 \\ 5.17 & -2.59 & 0.76 \\ 3.07 & 4.22 & 1.33 \\ 2.56 & 1.63 & -1.33 \end{bmatrix}}_{\mathbf{L}_1}, \underbrace{\begin{bmatrix} -0.61 & -0.27 & -0.75 \\ -0.28 & 0.95 & -0.12 \\ -0.75 & -0.14 & 0.65 \end{bmatrix}}_{\mathbf{R}_1} = CD(\mathbf{X}_1)$$

The Centroid Decomposition of $\mathbf{X}_1$ *is successfully obtained.*

$$\underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ -3 & -4 & -3 \\ -4 & 3 & -2 \\ -1 & 1 & -3 \end{bmatrix}}_{\mathbf{X}_1} = \underbrace{\begin{bmatrix} -2.65 & 1.36 & 0.32 \\ -8.01 & 1.90 & 0.45 \\ 5.17 & -2.59 & 0.76 \\ 3.07 & 4.22 & 1.33 \\ 2.56 & 1.63 & -1.33 \end{bmatrix}}_{\mathbf{L}_1} \times \underbrace{\begin{bmatrix} -0.61 & -0.28 & -0.75 \\ -0.27 & 0.95 & -0.14 \\ -0.75 & -0.12 & 0.65 \end{bmatrix}}_{{\mathbf{R}_1}^T}$$

4
Cached Centroid Decomposition (cached-CD)

This chapter introduces cached-CD, a novel approach of caching the maximizing sign vectors between two separate executions of the CD algorithm to improve its performance. When applied to streams of time series, cached-CD reduces the run-time complexity from quadratic to linear with the length of time series that compose the input matrix (cf. Chapter 6).

4.1 Idea

The idea behind cached-CD is based on two properties of the regular-CD computation.

1. The computation of the maximizing sign vectors in regular-CD is time intensive. Figure 4.1 describes the break-down time cost of the CD operations during their execution. In fact, we take a 100×4 input matrix populated with random values and compute its CD with an implementation of regular-CD (cf. Algorithm 1), which uses SSV (cf. Algorithm 2) to find the 4 maximizing sign vectors. The results of Figure 4.1 show that the most processing and time intensive part of computing the CD of an input matrix is finding its maximizing sign vectors, i.e., 98% of the time execution.

2. Input matrices that differ only on few rows (e.g., when adding a row) have very similar maximizing sign vectors. To illustrate this observation, we take a $n \times m$ input matrix $\mathbf{X}$, where initially $n = 1$ and $m = 4$, consisting of random numbers (positive and negative, in the range [-50, 50]), compute the $CD(\mathbf{X})$ and log the maximizing sign vectors (4 column vectors with n values each). Then, we incrementally increase n by adding a row (4 random numbers in the range [-50, 50]) to $\mathbf{X}$, recompute the $CD(\mathbf{X})$ and log the maximizing sign vectors. For each n, we measure the dissimilarity between its 4 maximizing sign vectors with the following vectors:

   - 4 'neutral' sign vectors of length n consisting just of 1s (as the sign vectors would be initialized in the SSV algorithm)
   - the 4 corresponding maximizing sign vectors obtained in the computation of CD($\mathbf{X}$) for $n - 1$
   - 4 random sign vectors of length n consisting of a random distribution of 1s and -1s

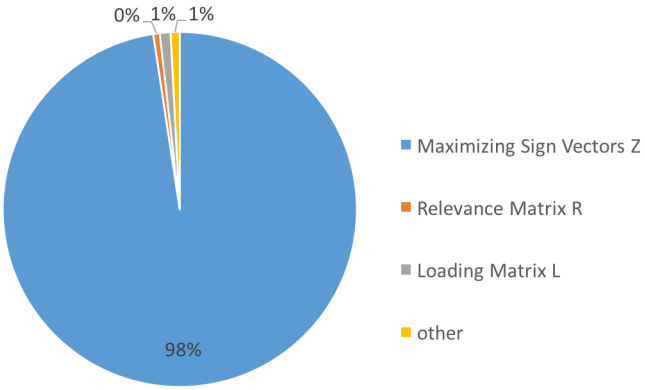


Figure 4.1: Break-down cost of CD operations

Note that we append the value 1 to the corresponding maximizing sign vectors of the previous computation, as they are only of length $n-1$ and we are comparing them to sign vectors of length n. In Figure 4.2, we use the Hamming distance [11], as dissimilarity function, to count the number of pairwise elements that are not equal (either 1 instead of -1, or vice-versa) between two sign vectors. For the sake of simplicity, we show only the measurements for $n = \{10, 20, \ldots, 100\}$. The results show that the 'neutral' sign vectors perform fairly equally to the random sign vectors, with Hamming distance just below $(n \times m)/2$. On the other hand, the Hamming distance between the maximizing sign vectors for the computation of CD($\mathbf{X}$) and those of the previous computation of CD is extremely low and does not grow proportionally to n. In fact, the distance even remains constant.

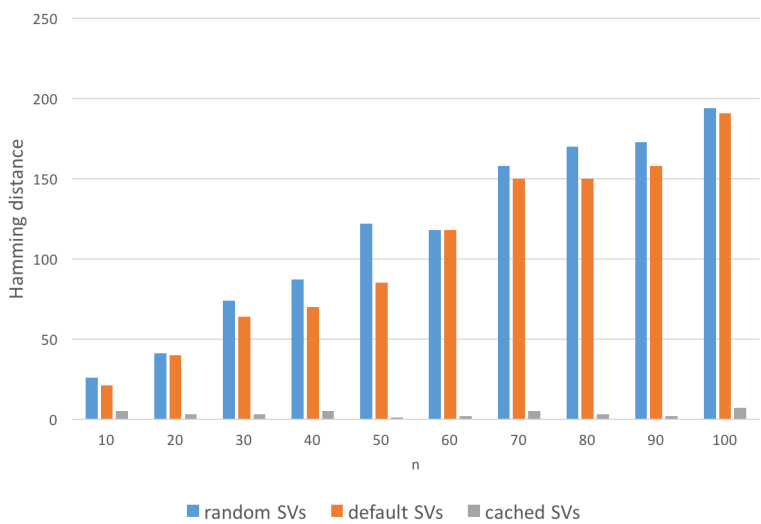


Figure 4.2: Maximizing sign vectors Hamming distance comparison

Our cached-CD approach exploits these two properties. More specifically, when computing the m

maximizing sign vectors of the current input matrix, cached-CD uses the m cached maximizing sign vectors from the previous CD computation as the initial sign vectors. At any time, there are exactly m maximizing sign vectors from the last computation of the cached-CD stored in the cache. Thus, the memory usage of the cached-CD is very low (cf. Section 4.2.1). This solution replaces the naive initialization strategy of the SSV algorithm that uses the default sign vector of all 1's.

4.2 Caching

4.2.1 Caching algorithm

Algorithm 4 describes the pseudo code of cached-CD. The sign vector, used to initialize the search for the maximizing sign vector, is read from the cache (line 2), passed on to the algorithm that finds the maximizing sign vector (line 3), and written back to the cache after it has been updated (line 4). Note that cached-CD reads and writes to the same cache position. For this reason, a cache overflow could occur only if the maximizing sign vectors of one computation of the CD would be too large in size to be stored in the cache simultaneously.

The combined number of elements of all the sign vectors produced during the computation of cached-CD($\mathbf{X}$) is equal to the amount of elements in the input matrix $\mathbf{X}$. For example, for $dim(\mathbf{X}) = n \times m$, CD($\mathbf{X}$) needs exactly m sign vectors with n elements each. However, unlike the elements in $\mathbf{X}$ (which could theoretically be any real number), the elements of the sign vectors are by definition either 1 or -1, which essentially makes the sign vectors binary arrays. Through optimisation, it would therefore be possible to reduce the memory-usage of the cached-CD to $n \cdot m$ bits. Assuming the input matrix $\mathbf{X}$ consists of values of 8 byte (64 bit) double-precision floating-point format, then the cache size required to store the maximizing sign vectors is only $^1/_{64}$ of the cache size that would be required to store the input matrix $\mathbf{X}$.

Algorithm 4: cached-CD($\mathbf{X}$)

Input : $n \times m$ matrix $\mathbf{X}$
Output : $n \times m$ matrix $\mathbf{L}$, $m \times m$ matrix $\mathbf{R}$

1 **for** $i \leftarrow 1$ **to** m **do**
2 $\quad Z_i \leftarrow$ read from cache position i;
3 $\quad Z_i \leftarrow$ *CSV*($\mathbf{X}$, Z_i);
4 $\quad$ store in cache position $i \leftarrow Z_i$;
5 $\quad L_{*i} \leftarrow$ *NextColumnL*($\mathbf{X}$, Z_i);
6 $\quad R_{*i} \leftarrow$ *NextColumnR*($\mathbf{X}$, Z_i);
7 $\quad \mathbf{X} \leftarrow \mathbf{X} - (L_{*i} \times R_{*i}^T)$;
8 **end**
9 **return** $\mathbf{L}, \mathbf{R}$;

In each iteration, the cached-CD algorithm finds the maximizing sign vector based not only on the current state of $\mathbf{X}$ (as would the regular-CD using SSV), but also based on the initial sign vector that was read from the cache (lines 2 and 3).

Figure 4.3 shows a schematic diagram of cached-CD being applied before and after a row is added to the input matrix $\mathbf{X}$. Note how the sign vectors Z_i are read from and written back to the same position in the cache for each execution of cached-CD. Note also how the loading columns L_{*i}, maximizing sign vector Z_i, and the loading matrix $\mathbf{L}$ each grow by a row, just as the input matrix $\mathbf{X}$.

In the next section, we define the Custom Sign Vector (CSV) algorithm, an extension of the SSV that takes a custom sign vector (e.g. a sign vector loaded from cache) as a parameter and uses it as the initializing sign vector before iteratively changing signs of elements to find the maximizing sign vector.

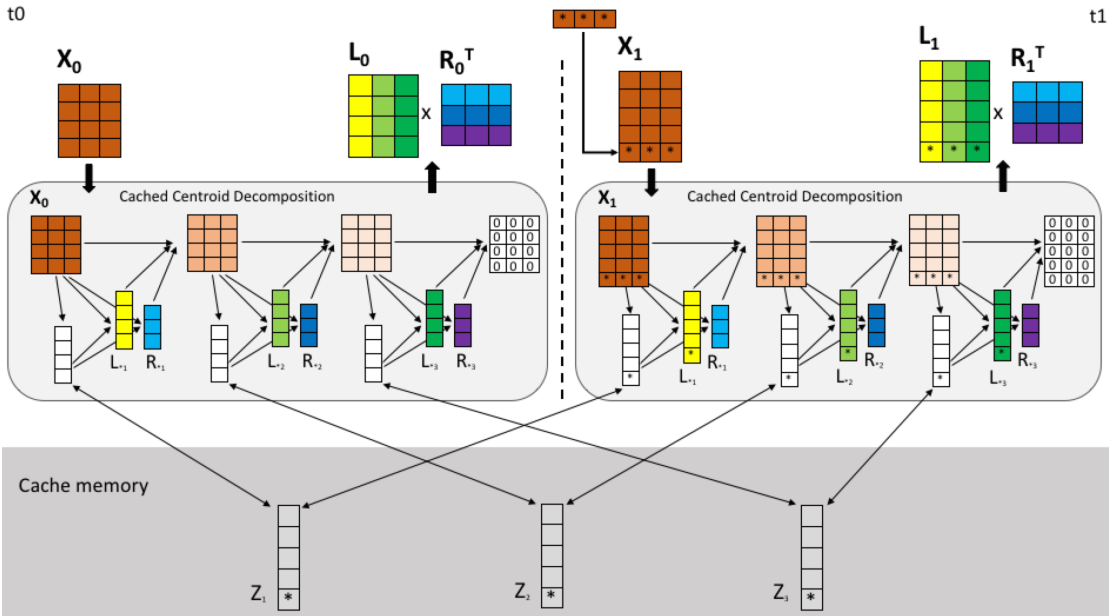


Figure 4.3: Schematic diagram of cached-CD

4.2.2 Custom Sign Vector (CSV)

Algorithm 5 describes the pseudo of the the Custom Sign Vector (CSV) algorithm, a modification of the SSV algorithm. Instead of initializing the first sign vector entirely with 1s (as the SSV would), the CSV takes a custom initial sign vector as an input parameter. CSV takes into consideration the fact that the length of the custom initial sign vector might not match the dimensions of the input matrix $\mathbf{X}$. This is because the input matrix might have grown since the last computation of the cached-CD and require larger sign vectors than those cached (the length n of a sign vector is identical to amount of rows n in the input matrix). CSV appends additional 1s at the end of the sign vector until the length is correct. This characteristic is especially important when dealing with streaming time series, as these grow in between two computations of the cached-CD.

4.2.3 Running example

Example 2. *Assume we have a stream of 3 time series with currently 4 elements each. Thus, at time $t = 0$, we have $m = 3$ and $n = 4$.*

$$\mathbf{X}_0 = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ -3 & -4 & -3 \\ 5 & 4 & -2 \end{bmatrix}$$

We apply the regular-CD to $\mathbf{X}_0$ and we get the 3 following maximizing sign vectors:

Algorithm 5: CSV($\mathbf{X}$, Z)

```
Input  : n × m matrix X, initial sign vector Z
Output : maximizing sign vector Z (length n)
while length(Z) > n do
    remove last element of Z;
end
while length(Z) < n do
    append as row value 1 to Z;
end
pos ← 0;
repeat
    if pos ≠ 0 then
        changeSign(z_pos);
    end
    V ←computeV(Z);
    find pos such that sign(z_pos) ≠ sign(v_pos) and |v_pos · z_pos| is maximal;
until pos = 0;
return Z^T ;
```

$$Z_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix}, Z_2 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, Z_3 = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

At time $t = 1$, we obtain new values in our stream: (1, 2, -3). The added values form a new row in the input matrix $\mathbf{X}_1$.

$$\mathbf{X}_1 = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ -3 & -4 & -3 \\ 5 & 4 & -2 \\ 1 & 2 & -3 \end{bmatrix}$$

We compute now the CD of $\mathbf{X}_1$ by taking advantage of the fact that we know Z_1, Z_2 and Z_3 of CD($\mathbf{X}_0$). The cached-CD takes the (cached) maximizing sign vectors Z_1, Z_2, and Z_3, respectively, as the initial sign vectors in the CSV algorithm (instead of $[1, 1, \ldots, 1]$, as we would in the regular-CD with SSV). Because the input matrix has one row more than the cached maximizing sign vectors, the CSV adds a 1 at the end before commencing with switching signs to finally obtain the maximizing sign vectors for $t = 1$.

- *Z_1 computation through CSV during cached-CD($\mathbf{X}_1$):*
  - *Initialization: append (1) to the sign vector*

- *$\rightarrow$ maximizing sign vector found*

$$\begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \\ \boldsymbol{1} \end{bmatrix} = Z_1$$

- *Z_2 computation through CSV during cached-CD($\mathbf{X}_1$):*
  - *Initialization: append (1) to the sign vector*
  - *$\rightarrow$ maximizing sign vector found*

$$\begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \\ \boldsymbol{1} \end{bmatrix} = Z_2$$

- *Z_3 computation through CSV during cached-CD($\mathbf{X}_1$):*
  - *Initialization: append (1) to the sign vector*
  - *Iteration 1: switch sign at position 2*
  - *Iteration 2: switch sign at position 1*
  - *Iteration 3: switch sign at position 3*
  - *$\rightarrow$ maximizing sign vector found*

$$\begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \\ \boldsymbol{1} \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ \boldsymbol{-1} \\ 1 \\ -1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} \boldsymbol{1} \\ -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ -1 \\ \boldsymbol{-1} \\ -1 \\ 1 \end{bmatrix} = Z_3$$

In total, to find the 3 maximizing sign vectors needed for the computation of CD($\mathbf{X}_1$), there were exactly 3 sign switches necessary (after initialization) in the 3 instances where the CSV algorithm was executed. The application of the regular-CD (where SSV uses the standard initial sign vector of all 1s) would yield a combined total of 6 necessary sign switches.

4.2.4 Complexity

Cached-CD takes as input $n \times m$ matrix. In this thesis, we specifically deal with streams of (growing) time series. Thus, the number of time series, m, is therefore usually fixed, while the number of values per time series, n, is incessantly increasing. For this reason, the complexity is expressed with respect to n (unless stated otherwise).

a) run-time complexity: when using SSV to compute the maximizing sign vectors, the regular-CD has a quadratic run-time complexity with respect to n [9]. If the maximizing sign vectors of the previous computation of the CD of a very similar input matrix are cached, then the CSV is a lot faster and requires a lot less iterations than the SSV does. In such a case the run-time of the cached-CD is linear with respect to

n (cf. Section 6.1). We conclude the that run-time of the cached-CD depends not only on n, but also on how similar the current input matrix $\mathbf{X}_1$ is to the previous one ($\mathbf{X}_0$, for which we know the maximizing sign vectors). It was not within the scope of this thesis to determine how 'close' these input matrices must be to guarantee the run-time complexity of cached-CD to be linear. However, our experiments in Section 6.1 show, that either adding a single row to the input matrix (when using cached-CD to continuously compute the CD of a stream of time series) or updating just a few selected values in the input matrix (when using cached-CD for CD based recovery of missing values) both result in linear run-time complexity. A formal proof has yet to follow.

a) space complexity: the memory complexity of the cached-CD is simple to compute. As already discussed in Section 4.2.1, the combined amount of elements of the maximizing sign vectors that need to be kept in the memory between two computations of the cached-CD is equal to the amount of elements in the input matrix $\mathbf{X}$. Since the amount of values in $\mathbf{X}$ is $n \cdot m$, and we assume m to be constant, the amount of values stored in the cache at any time is exactly linear to n. Thus, we can conclude that the cached-CD has linear space complexity in respect to n.

4.3 Streaming algorithm

We define $\mathbf{S}$ as a stream object that contains m streams of time series. The time series are synchronized and they are updated with a new value simultaneously. The time series within $\mathbf{S}$ are aligned like a grid, essentially creating an $n \times m$ matrix, where n increases every time $\mathbf{S}$ receives an update. The function *getMatrix(*$\mathbf{S}$*)* returns the $n \times m$ matrix of all time series and values currently in $\mathbf{S}$. $\mathbf{S}$ triggers an event each time it is updated, i.e., we can 'listen' for updates of $\mathbf{S}$ in an algorithm. Algorithm 6 describes the pseudo code of the streaming procedure that takes a stream object $\mathbf{S}$ as input and recomputes and prints the current CD each time $\mathbf{S}$ receives an update.

Algorithm 6: streaming-CD($\mathbf{S}$)

```
Input  : Streaming object S
while true do
    waitForUpdate(S);
    X ← getMatrix(S);
    L, R ← cached-CD(X);
    print(L, R);
end
```

4.4 Cached-CD based recovery algorithm

The utilization of cached-CD for CD based recovery of missing values in batches of time series is very straightforward. We adapt the RecM algorithm introduced by Khayati et al.[4] and construct cached-RecM

(Algorithm 7).

Algorithm 7: cached-RecM($\mathbf{X}, T_j^m, \epsilon$)

Input : $n \times m$ matrix $\mathbf{X}$; set of missing time stamps T_j^m in $X_{,j}$; termination threshold ϵ
Output : $n \times m$ matrix $\tilde{\mathbf{X}}$ with recovered values

```
1  linearly interpolate/extrapolate all missing values in X
2  repeat
3      X̃ ← X
4      L, R ← cached-CD(X);
5      L_k ← truncate L by factor k;
6      X_k ← L_k × R^T;
7      foreach t ∈ T_j^m do
8          x̃_tj ← w_tj;
9          // x_tj element of X; w_tj element of X_k
10     end
11 until ‖X − X̃‖_F < ε;
12 return X̃;
```

The algorithm starts by initializing the missing values of the input matrix $\mathbf{X}$ using either linear interpolation or extrapolation, depending on the position of the missing values in $\mathbf{X}$. Next, the cached-CD of the current state of $\mathbf{X}$ is computed, to receive $\mathbf{L}$ and $\mathbf{R}$. $\mathbf{L}_k$ is the truncation of $\mathbf{L}$, where the values in the k last columns are replaced by 0s. The truncation of $\mathbf{X}$, $\mathbf{X}_k = \mathbf{L}_k \times \mathbf{R}^T$ is computed and the input matrix $\mathbf{X}$ is updated with the values of $\mathbf{X}_k$ in exactly the positions of the initially missing values. Cached-RecM iterates until the calculated Frobenius distance [12] between $\mathbf{X}$ before and after the update computed by an iteration of RecM falls below a defined threshold value ϵ.

An evaluation of cached-RecM ("cached-CD based recovery") in comparison to RecM ("regular-CD based recovery") is conducted in Section 6.1.3.

5
c-ReVival

In the scope of this thesis we have implemented a graphical tool called c-ReVival (Cached REcoVery of mIssing VALues). This tool allows to:

- Visualize the properties of CD based recovery on synthetic time series.
- Compare different strategies for finding the maximizing sign vector.
- Display real-world data sets with multiple time series:
  - Browse aligned and non-aligned time series with different granularities.
  - Browse the missing blocks of values throughout the entire history of data.
  - Work with raw data as well as normalized data (Min-Max, Z-Score).
  - Display globally and partially the similarities between time series.
- Display the result of the cached-CD based recovery on real-world batch times series.
- Compare the scalability of the cached-CD compared to the updating-CD and the regular-CD.

At a technical level, c-ReVival has a server-client architecture, and it is accessed via a web browser [1]. The server is connected to a PostgreSQL database, where all the time series and meta data are stored. Messages between client and server are exchanged using HTTP. The code for the server-side logic is written and executed in PHP; the code for the client-side logic is written and executed in JavaScript directly in the web browser.

5.1 Components

c-ReVival consists of six individual components that each serve a concrete purpose.

[1] c-ReVival tool can be accessed on a server of the eXascale Infolab: `http://revival.exascale.info/`

5.1.1 Displaying data sets

Two components are dedicated to displaying time series data. The first component [2] (see Figure 5.1) allows the user to explore entire real-world data sets, consisting of batch time series (TS). Currently, the contains two real-world data sets:

Table 5.1: Description of real-world data sets

Name	Source	Description	# of TS	# of values per TS
Hydrology	FOEN [3]	The TS represent water level data of rivers in 7 places. The unit of the data is meters. The data is aggregated to 4 values per day from 1974 to 2009.	7	52'589
Temperature	MeteoSwiss [4]	The TS represent temperature data of 6 cities. The unit of the data is degrees Celsius. The data is aggregated to 4 values per day from 1990 to 2015.	6	37'984

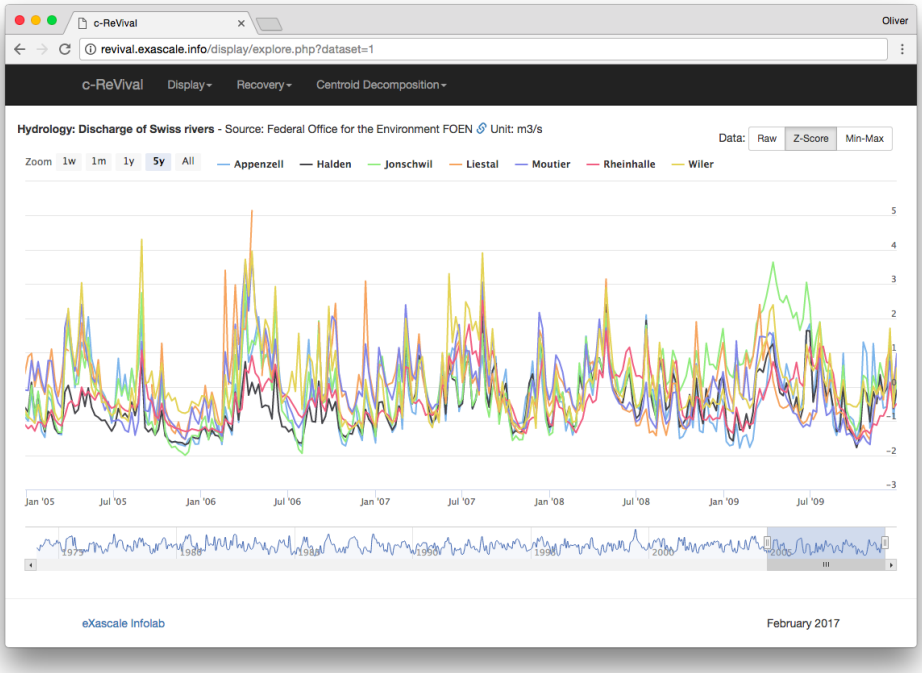


Figure 5.1: Displaying batch time series

[2] `http://revival.exascale.info/display/datasets.php`
[3] Federal Office for the Environment: `http://www.hydrodaten.admin.ch/en`
[4] Federal Office of Meteorology and Climatology: `https://gate.meteoswiss.ch/idaweb/more.do`

The available data sets together with some meta data are listed in the list view of the tool. When clicking on one of the data sets in the list, a graph chart of the data set appears. Each graph represents an individual time series, and can be hidden and made visible by clicking on its label above the chart. The zoom level can either be selected by clicking on one of the corresponding buttons in the top left corner, or by selecting a time range directly within the chart. Below the chart, there is a range selector to navigate through the entire time range of the data. In the top right corner, the data representation can be chosen. By default, the data in the chart is z-score normalized, but it is also possible to use Min-Max normalization or to show the raw data values.

The second component [5] (see Figure 5.2) visualizes an example of streaming time series data. There are three time series that receive updates at a constant rate.

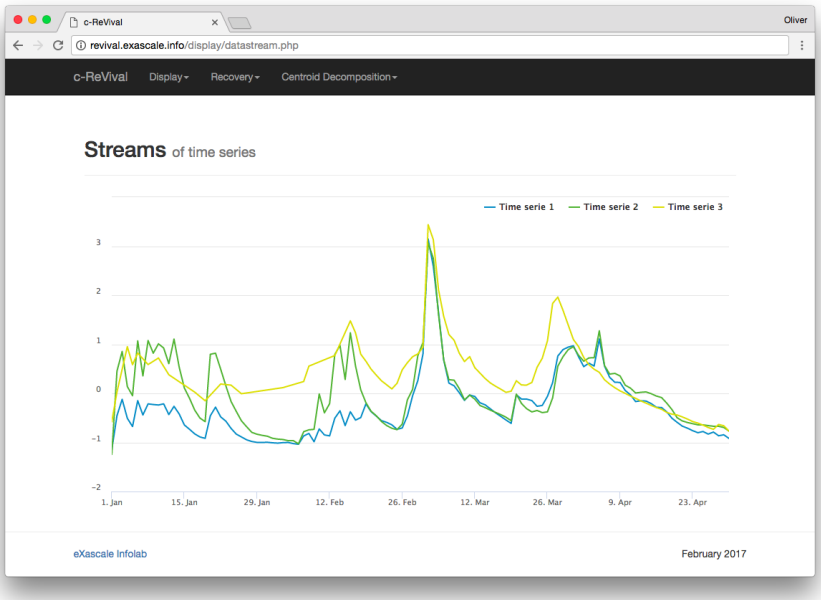


Figure 5.2: Displaying streaming time series

5.1.2 Recovery of missing blocks

The main feature of c-ReVival is to visualize the cached-CD based recovery of missing blocks. One component [6] (see Figure 5.3), illustrates the important properties of the recovery algorithm cached-RecM (cf. Algorithm 7) on synthetic data. The current state of the input matrix $\mathbf{X}$ is illustrated both as a graph chart (left-hand side) and printed in absolute values (right-hand side). The development of the Frobenius distance between $\mathbf{X}$ at the beginning and at the end of each separate computation of the CD is shown in the chart on the bottom of the page. The red line in that chart marks the chosen threshold value ϵ. The recovery process terminates after the Frobenius distance drops below ϵ. Additionally, further charts that display characteristics of each individual iteration can be made visible below the aforementioned charts. Namely, there is an additional chart that shows the progress of the CSV algorithm finding each of the four the maximizing sign vectors required per computation of the CD.

[5] `http://revival.exascale.info/display/datastream.php`
[6] `http://revival.exascale.info/recovery/static.php`

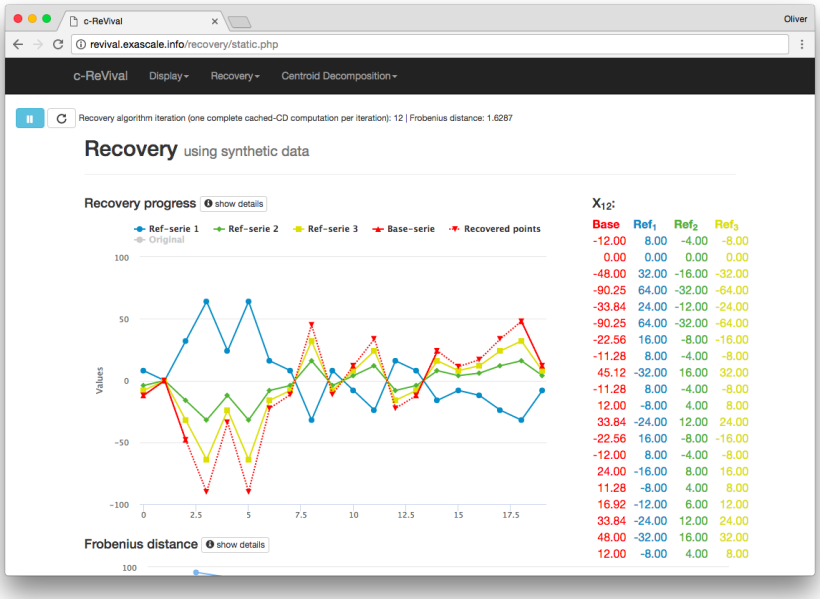


Figure 5.3: Recovery on synthetic data

Figure 5.4 shows the component [7] that allows the user to perform cached-CD based recovery on the large real-world data sets also used in the display component (cf. Section 5.1.1). Additionally, there are modified versions of some of the data sets, where extra values have been removed in order to have more and especially larger blocks of missing values. When clicking on a data set from the list, it gets displayed as a chart with the same controls as in the display component. In addition, there is a panel on the right-hand side to set the parameters for the recovery process. The base time series is the time series for which the missing values are to be retrieved. Any of the time series of the data set can be selected. The reference time series are the time series to be used for the retrieval of the missing values of the base time series. If the reference series have missing values themselves, these values are linearly interpolated before the actual retrieval takes place. There are two modes for selecting the reference time series. The manual mode allows the user to manually select the references time series from a multi-select dropdown menu. The globally correlated mode allows the user to choose the number of reference time series to use, after which the system selects those time series with the highest correlation to the base time series.

The time range, for which to recover the missing values, can be set to predefined values of either one week, one month, one year, or to a manual duration. As a starting point for the recovery, automatically the time of the left edge of the chart is used. If manual duration was selected, the recovery will cover the entire currently visible time range of the chart. The threshold value epsilon ϵ can also be adjusted. The selection of ϵ has great impact on the duration of the recovery, since the smaller epsilon, the more iterations of the recovery algorithm are necessary.

[7] `http://revival.exascale.info/recovery/datasets.php`

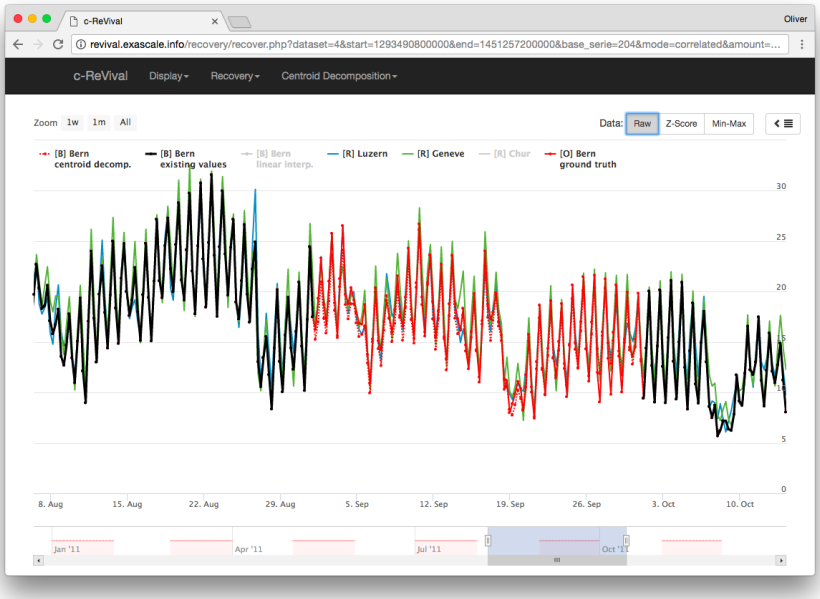


Figure 5.4: Recovery on real-world data

5.1.3 Maximizing sign vector strategy comparison

This rather experimental component [8] (see Figure 5.5) visualizes the comparison of four (theoretical) strategies for finding the maximizing sign vector given an input matrix. It was implemented to get a better understanding of i) the order in which the SSV algorithm (cf. Section 2.2.3) switches signs, and ii) if there are any alternative strategies to or variations of SSV that could speed up the computation of the maximizing sign vector. The three considered strategies are:

- *DSV*: The "double SSV" is identical to the SSV, except that it switches the sign of two elements (instead of one) of the sign vector Z at each iteration.
- *TSV*: The "triple SSV" is identical to the SSV, except that it switches the sign of three elements (instead of one) of the sign vector Z at each iteration.
- *PSV*: The "positive SSV" is identical to SSV, except that it switches the sign of an element of the current sign vector Z if its corresponding element in the weight vector V has the same sign (versus the opposite sign, as in the regular SSV).

[8] `http://revival.exascale.info/cd/signvectors.php`

Figure 5.5: Maximizing sign vector strategy comparison

The comparison consists of four charts per example which are iteratively updated after the corresponding play button has been clicked. The chart on the very left shows for each strategy the sign vectors (represented vertically; blue stands for value 1 and red for value -1) and the order in which their signs are switched. The most right sign vector shows the maximizing sign vector (computed by brute-force approach). The chart in the middle shows a graph that tracks for each strategy and each algorithm the product $|Z^T \times V|$. The red line shows the value of the maximizing sign vector. If a strategy terminates correctly, its last $|Z^T \times V|$ will be on this line. The two bar charts on the right show the current count of a) iterations (top) and b) sign switches (bottom) per strategy. On the far right the input matrix $\mathbf{X}$ of the example is printed.

5.1.4 Streaming computation

The last component [9] (see Figure 5.6) compares the run-time of three approaches for computing the CD for a stream of time-series. For $n = \{1, 2, \ldots\}$, the run-time of each algorithm to compute CD($\mathbf{X}_n$) is measured, given the fact that CD($\mathbf{X}_{n-1}$) is already computed. $\mathbf{X}_n$ is $\mathbf{X}_{n-1}$ plus the appended row vector A_n. The three implemented approaches are:

- The regular-CD, that simply computes CD($\mathbf{X}_n$) from scratch each time.
- The updated-CD, that uses the previously computed loading and relevance matrices, $\mathbf{L}_{n-1}$ and $\mathbf{R}_{n-1}$, of CD($\mathbf{X}_{n-1}$), in combination with row vector A_n to construct an intermediate matrix $\mathbf{S}$, compute its (regular) CD($\mathbf{S}$), and construct CD($\mathbf{X}_n$).
- The cached-CD, that re-uses from cache the maximizing sign vectors obtained during CD($\mathbf{X}_{n-1}$) to speed up the computation of CD($\mathbf{X}_n$).

[9] `http://revival.exascale.info/cd/streaming.php`

The values of the streaming time series are taken from the Hydrology data set that was introduced in Section 5.1.1. The computation of the CD using each of the three algorithms can be started by clicking the play button. Initially, the input matrix $\mathbf{X}$ has the dimensions $m = 3$ and $n = 8$. n grows as the rows are added to $\mathbf{X}$, m is constant. The chart shows the run-time (y-axis) for each algorithm to compute the CD after each addition of a new row (x-axis). Underneath the chart, a collapsed panel is added for each row that was added. It contains the details of the CD computation for each algorithm.

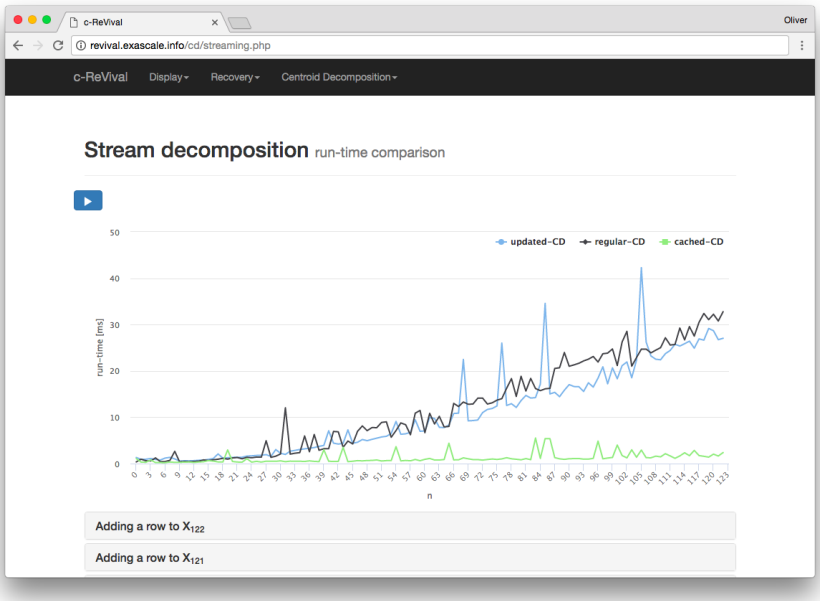


Figure 5.6: Stream decomposition run-time comparison

6 Evaluation

In this chapter, we evaluate the performance of our proposed cached-CD approach. More specifically, we compare i) the decomposition scalability for streams of time series of our cached-CD solution against the updating-CD and the regular-CD, and ii) the efficiency of recovery of missing values using cached-CD against using regular-CD. Additionally, we qualitatively evaluate two components of the c-ReVival tool: i) the comparison of different sign vector strategies, and ii) the accuracy of CD based recovery on real-world data sets.

6.1 Cached Centroid Decomposition (cached-CD)

6.1.1 Setup

6.1.1.1 Data sets

For all experiments, we used a subset of the real-world data sets featured in c-ReVival (cf. Section 5.1.1). Table 6.1 describes the data sets we have used for the experiments.

Table 6.1: Description of the real-world time series.

ID	Source	Base TS	Reference TS	values per TS
Hyd1	Hydrology data set	Appenzell	Jonschwil, Wiler, Halden	52’589
Hyd2	Hydrology data set	Appenzell	Liestal, Moutier, Rheinhalle	52’589
Temp1	Temperature data set	Bern	Luzern, Geneve, Chur	37’984
Temp2	Temperature data set	Bern	Chur, Lugano, St.Gallen	37’984

6.1.1.2 Environment

We use an Apache2 web server hosted on an Ubuntu machine for the computations. All algorithms are implemented in PHP, and are the same as those used in the c-ReVival tool. Table 6.2 shows the

specifications of the server used for the experiments. The data sets are stored and accessed on the same server in a PostgreSQL database.

Table 6.2: Description of machine

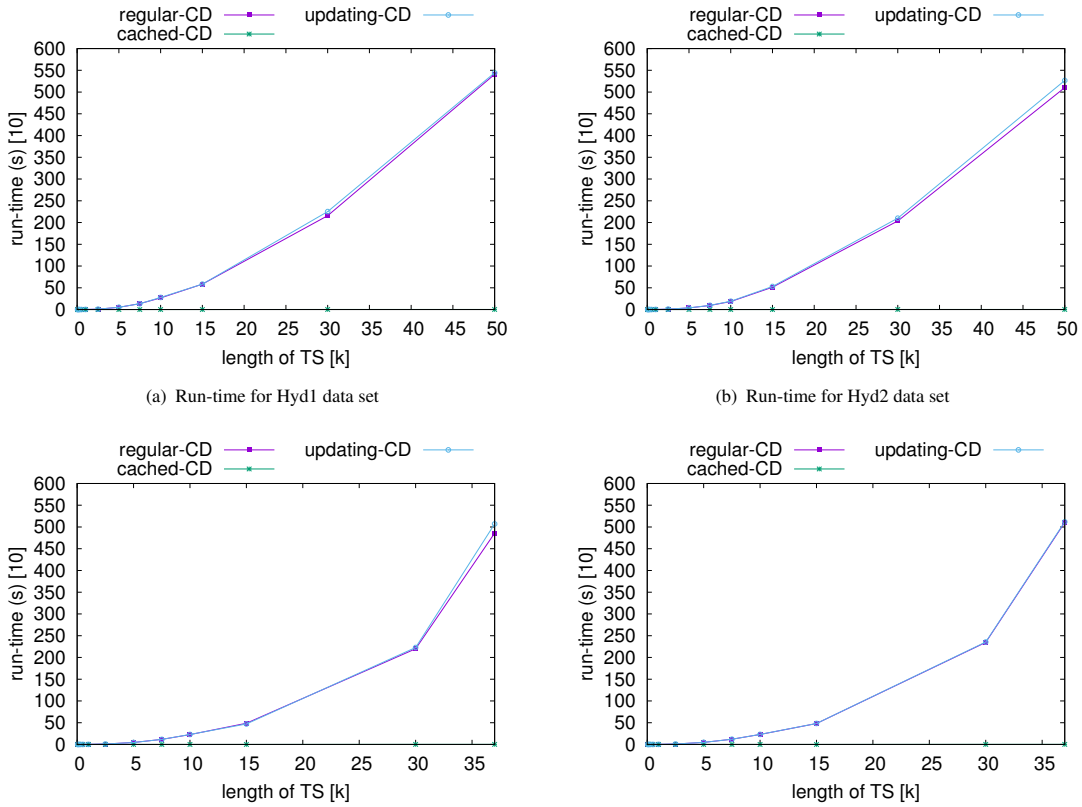
Operating System	Environment	Processor	RAM
Linux Ubuntu	Apache2	quad-core at 3.40 GHz	8 GB at 1600 MHz

6.1.2 Streaming computation

6.1.2.1 Scalability

In this first set of experiments, we measure the scalability of the three different approaches of computing the CD for streams of time series, i.e., regular-CD, updating-CD and cached-CD.

For each of the previously defined test data sets, we build a matrix $\mathbf{X}$, consisting of all reference time series. For several selected values of n, we compute first the CD of $\mathbf{X}_{n-1}$. Then, we compute CD($\mathbf{X}_n$) with all three approaches independently, by using cached values of the previous computation. More specifically, we use the maximizing sign vectors of CD($\mathbf{X}_{n-1}$) for cached-CD($\mathbf{X}_n$), and the loading and relevance matrices($\mathbf{L}_{n-1}$ and $\mathbf{R}_{n-1}$) of CD($\mathbf{X}_{n-1}$) for updating-CD($\mathbf{X}_n$). For each approach, we measure exactly the run-time for computing CD($\mathbf{X}_n$), and count the number of sign switches necessary to find the maximizing sign vectors.



(a) Run-time for Hyd1 data set

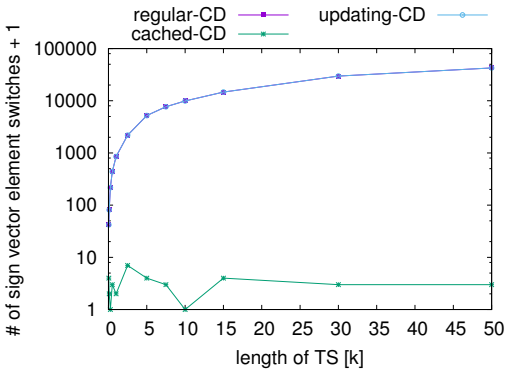
(b) Run-time for Hyd2 data set

(c) Run-time for Temp1 data set

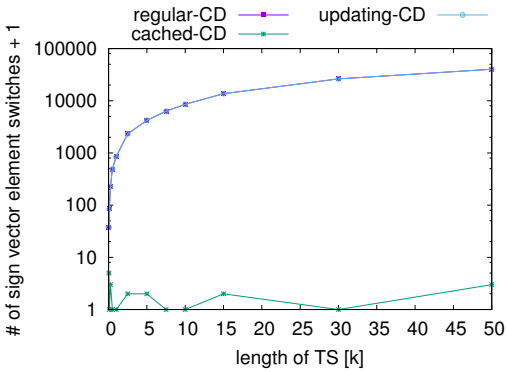
(d) Run-time for Temp2 data set

Figure 6.1: Run-time of regular-CD vs. updating-CD vs. cached-CD with varying n

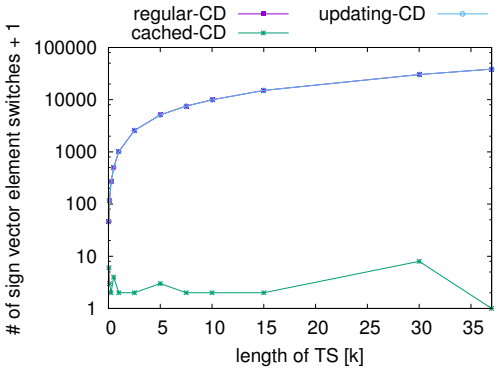
In the experiment of Figure 6.1, we measure the run-time varying n. The results of this experiment show that, for each of the data sets, the cached-CD algorithm scales extremely well compared to updating-CD and the regular-CD. The updating-CD and the regular-CD are not scalable, as they both have a nearly equal quadratic run-time with respect to n. For instance, for $n = 10k$ in the Hyd1 data set, the cached-CD's run-time is 834 times faster than that of regular-CD (0.32s vs. 266.42s) and 871 times faster than that of updating-CD (0.32s vs. 278.06s). These factors increase as n grows. For $n = 50k$, cached-CD is already 2586 times faster than regular-CD (2.09s vs. 5408.57s) and 2603 times faster than updating-CD (2.09s vs. 5443.29s). Cached-CD is essentially reducing the run-time from hours to seconds.



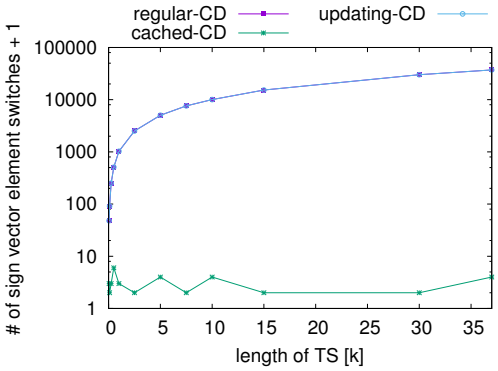
(a) # of sign vector element switches for Hyd1 data set



(b) # of sign vector element switches for Hyd2 data set



(c) # of sign vector element switches for Temp1 data set



(d) # of sign vector element switches for Temp2 data set

Figure 6.2: Sign switches of regular-CD vs. updating-CD vs. cached-CD with varying n

In the experiment of Figure 6.2 we measure the number of sign switches in the sign vectors by varying the value of n. Note the log scale in the y-axis. The results of this experiment confirm the assumption, that the scalability of cached-CD is achieved thanks to a reduced number of sign vector element switches needed in comparison to regular-CD and updating-CD. For the latter two, the number of sign vector switches grows linearly with n, while for the cached-CD it remains constant. We can conclude from this experiment, that the caching approach of the cached-CD (and, thus, by using CSV instead of SSV) does achieve a significant improvement over the regular-CD. The SSV algorithm is clearly the bottleneck regarding run-time of the regular-CD, as it grows quadratically to n.

The high run-time of the updating-CD technique is expected. In fact, while the updating-CD is an incremental approach for the computation of CD, it still ends up having to use regular-CD to compute CD($\mathbf{S}$) (line 12 of Algorithm 3). Because $\mathbf{S}$ has identical dimensions as $\mathbf{X}$, the computations of CD($\mathbf{S}$) and CD($\mathbf{X}$) will yield a very similar run-time. The regular-CD is even a little faster than the updating-CD, as the updating-CD has the additional overhead of computing $\mathbf{S}$ first. For the updating-CD to be more efficient, an alternative decomposition approach for computing CD($\mathbf{S}$) would need to be found. This extension would need to take advantage of the fact that $\mathbf{S}$ is very similar to a loading matrix $\mathbf{L}$, which, as we pointed out (cf. Section3.1), is also a stationary point.

In the experiment of Figure 6.3, we evaluate the scalability of the cached-CD for larger n. We discard the regular-CD and the updated-CD from this experiment due to their high execution time (up to several hours). Since the time series of our test data sets are limited in length, we concatenate the time series

to form two longer time series. Starting at $n = 1$, we iteratively increase n and compute for each n cached-CD($\mathbf{X}_n$). The results confirm that the run-time of cached-CD is linear with the length of the times series (n).

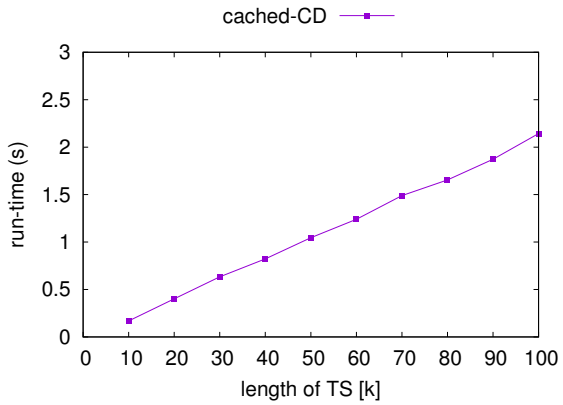


Figure 6.3: Run-time of cached-CD for varying n

So far, we have focused on evaluating the performance of regular-CD, updating-CD and cached-CD with respect to the length of time series (n). To verify that the improvements of cached-CD in terms of run-time complexity with n do not sacrifice its scalability with the amount of input time series (m), we conduct a further experiment. In Figure 6.4, we evaluate the scalability of cached-CD, regular-CD and updating CD by fixing $n = 1k$ but varying m. For this experiment, we use all the time series from our test data sets combined. We start with $m = 2$ time series in the input matrix and measure the run-time of the computation of CD with all three approaches, just as in the previous experiment. We iteratively increase m by adding a time series to the input matrix and re-run the experiment. The results of this experiment show that cached-CD does not compromise on scalability with m: It scales just as well as regular-CD and updating-CD.

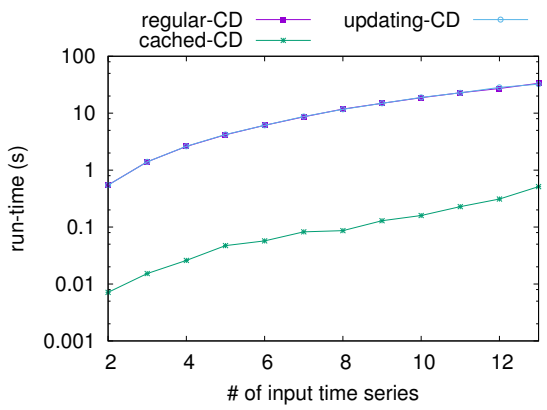
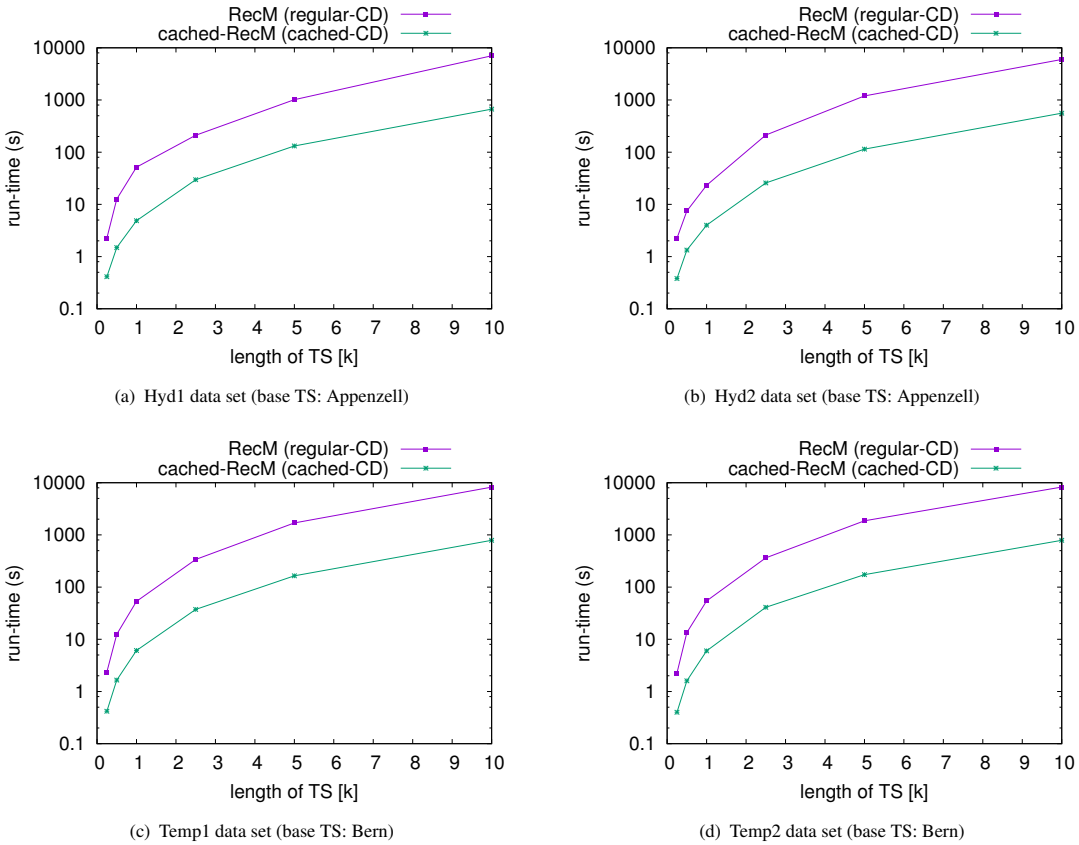


Figure 6.4: Run-time of regular-CD vs. updating-CD vs. cached-CD for varying m

6.1.3 Recovery

6.1.3.1 Scalability

In the second set of experiments, we compare the impact of the chosen CD computation technique (regular-CD vs. cached-CD) on the recovery run-time for various length n of input time series. RecM is the recovery algorithm that uses regular-CD, and cached-RecM is the recovery algorithm that uses cached-CD. We set the threshold to $\epsilon = 0.01$ and perform the same experiment on each of our four test data sets.



(a) Hyd1 data set (base TS: Appenzell)

(b) Hyd2 data set (base TS: Appenzell)

(c) Temp1 data set (base TS: Bern)

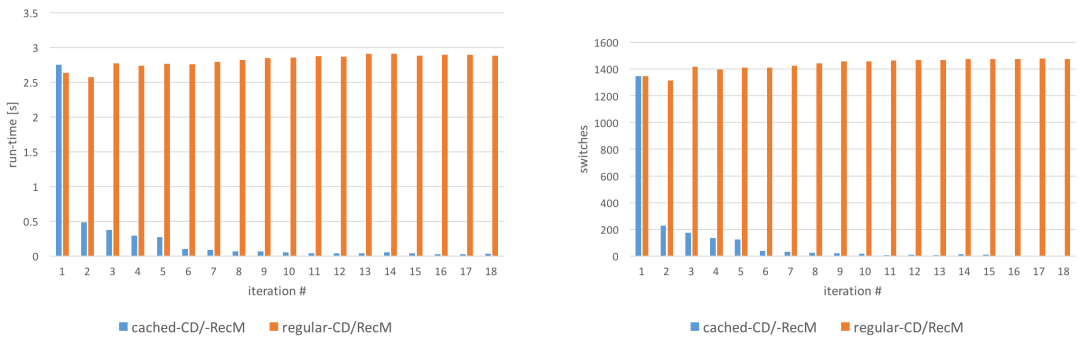
(d) Temp2 data set (base TS: Bern)

Figure 6.5: Run-time of RecM vs. cached-RecM for varying n

The results in Figure 6.5 show that the cached-CD based recovery (cached-RecM) is faster than the regular-CD based recovery (RecM) by quite a multiple for all compared measurements. For example, at $n = 10k$, cached-RecM is more than 10 times faster than RecM for all our tested data sets (Hyd1: 669s vs. 7028s; Hyd2: 558s vs. 5981s; Temp1: 788s vs. 8238s; Temp2: 788s vs. 8263s). The run-time complexity for both algorithms is quadratic to n. The equal run-time complexity is due to the fact that for the first computation of CD (remember, RecM/cached-RecM make multiple computations of CD before they terminate), RecM and cached-RecM both start the recovery process with no maximizing sign vectors in cache. This makes the cached-CD equivalent to regular-CD in this first recovery iteration, meaning also that all sign vectors are initialized with 1s within the CSV algorithm in cached-CD. Only for the second and all subsequent recovery iterations of cached-RecM, the caching effect can be leveraged, yielding linear run-time complexity for all the cached-CD computations after the first one. However, the overall run-time

complexity remains quadratic for both competitors.

In the experiment of Figure 6.6 we show the run-time and the needed sign vector element switches of each computation of CD made during cached-RecM (using cached-CD) and RecM (using regular-CD). For $n = 2.5k$, using both cached-RecM and RecM, the recovery in the Temp1 data set (with base time series Bern and reference time series Luzern, Geneve, and Chur) takes 18 iterations until the obtained 'recovery progress' (measured with the Frobenius norm) of the iteration drops below the selected threshold $\epsilon = 0.01$. The equal number of sign vector element switches in the first iteration is explained by the fact that both algorithms compute the CD of the input matrix for the first time and use identical sign vectors in the process. For all following iterations, cached-RecM substantially performs less sign vector element switches than RecM, yielding the result that cached-CD based recovery is faster than regular-CD based recovery by a fairly constant factor. In fact, the exact factor by which cached-RecM is faster than RecM depends on the number of iterations needed for the recovery. Both algorithms compute exactly the same recovery and have an equal number of iterations, they just use a different technique (cached-CD or regular-CD, respectively) to compute the CD within the iterations. Assuming the recovery needs k iterations to terminate, cached-RecM has exactly one iteration with quadratic run-time complexity (and thus, equal run-time than RecM) and $k-1$ iterations with linear complexity, while RecM has all k iterations at quadratic run-time complexity. We conduct a separate experiment to illustrate the impact of the number of iterations performed by cached-RecM/RecM, k, on the number of rows (n).



(a) Run-time per iteration number

(b) Sign vector element switches per iteration number

Figure 6.6: Iteration run-time and sign switches of RecM vs. cached-RecM

Figure 6.7 shows the number of iterations k of cached-RecM/RecM for each of our test data sets given n and the threshold of $\epsilon = 0.01$. The result of this experiment shows that there is no direct relation between n and k. The number of iterations k needed seems to depend more on the characteristics of the data set such as the amount/placement of missing values within the input matrix. Since there is no apparent relation between k and n, it is not possible to express the factor by which the run-time of cached-RecM (cached-CD based recovery) is faster than RecM (regular-CD based recovery) with respect to the number of rows n of the input matrix.

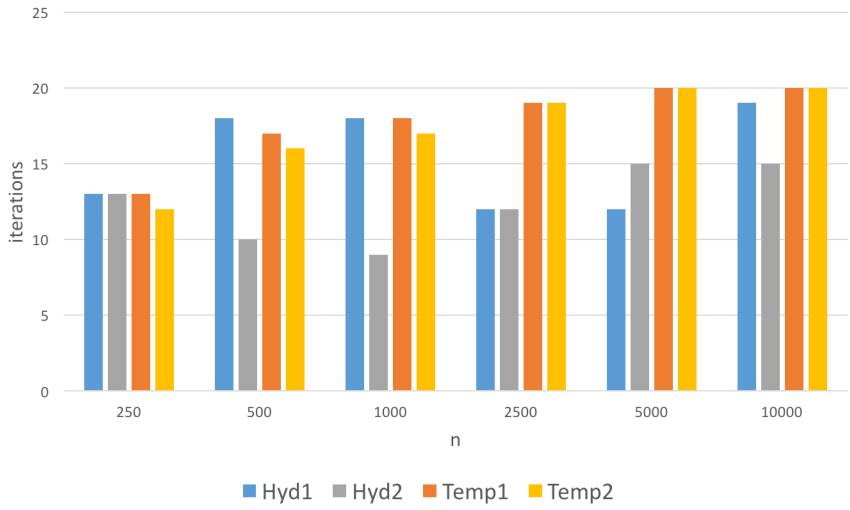


Figure 6.7: Number of RecM iterations for a varying n

6.2 c-ReVival

The main functionality of c-ReVival is to visualize the cached-CD based recovery and the properties of the CD technique on time series. In this section, we qualitatively evaluate two components of the tool.

6.2.1 Recovery using CD

The recovery component of c-ReVival uses cached-RecM for the recovery of blocks of missing values in real-world data sets. The visualization of the results show some interesting characteristics of cached-CD based recovery.

6.2.1.1 Temperature data set

The Temperature data set (cf. Section 5.1.1) has a very high average pairwise correlation between its time series (0.98). This has to do with nature of the data, namely measured temperatures in Swiss cities. The high correlation results in the recovery being very accurate. Figure 6.8 shows a recovery of a block of missing values in the Temperature data set performed with c-ReVival. For the recovery, we set the threshold to $\epsilon = 0.01$. The green and blue lines show the reference time series. The black line shows the values we used for the base time series. We removed some of the original values of the base time series to be able to compare our recovery to the ground truth. The removed values were not part of the recovery process and are shown by the solid red line. Finally, the cached-CD based recovery is represented by the dotted red line.

The visualization allows to compare the recovered values (red dotted line) with those that were removed before the recovery process started (solid red line). The recovery appears to be extremely accurate for the Temperature data set. The shapes and amplitude of the curves are nicely recovered with just minimal deviation.

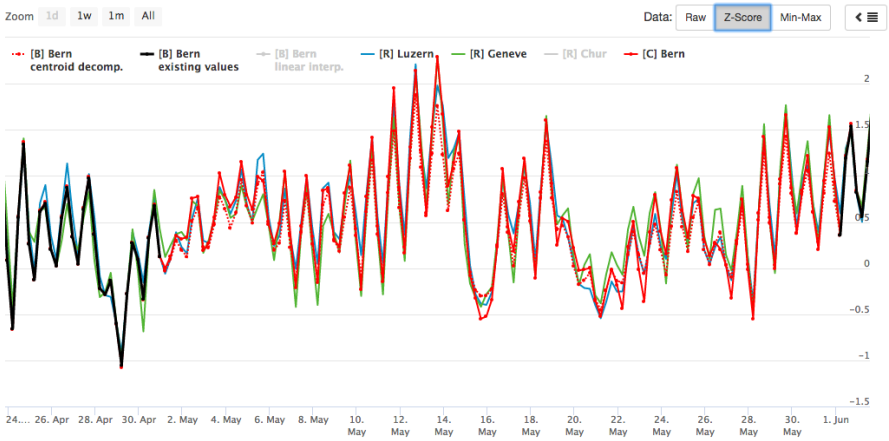


Figure 6.8: Recovery performed on the Temperature data set: case 1.

Figure 6.9 shows a further recovery of missing values in the Temperature data set. The visualization allows for the same observations as those made in Figure 6.8.

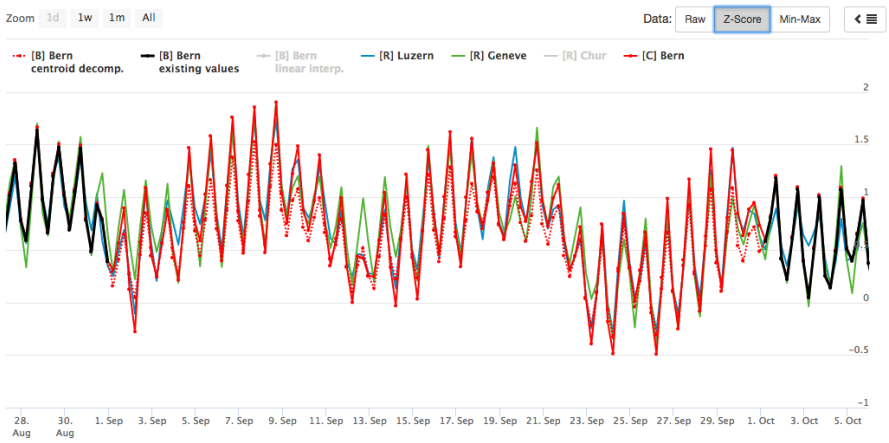


Figure 6.9: Recovery performed on the Temperature data set: case 2

6.2.1.2 Hydrology data set

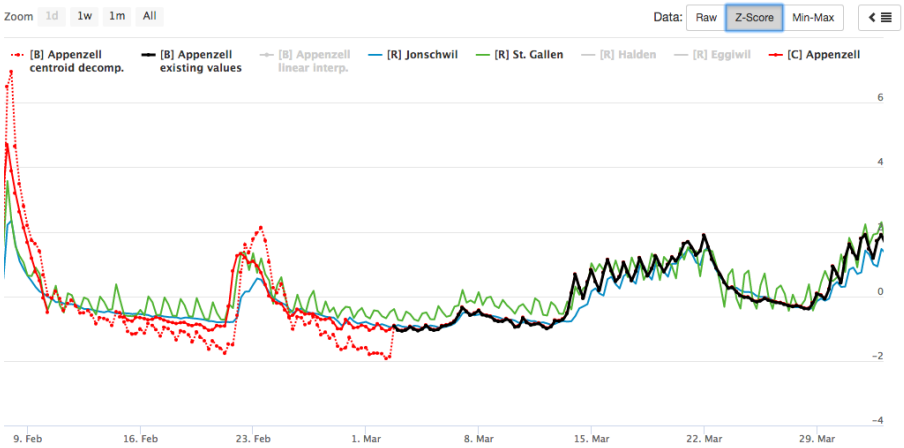
The Hydrology data set (cf. Section 5.1.1) is a lot less regular than the Temperature data set, which also causes the average pairwise correlation between pairs of time series to be lower (0.69 vs. 0.98). This is due to the nature of the measurements, namely water levels of rivers in different cities. Since rain can be quite a regional phenomena, the time series might be shifted in time with respect to observable curves in the data, or even have trends that appear only in a subset of the time series. Given these characteristics of the data set, the expectations of the recovery accuracy can not be quite as high as for the Temperature data set. Figure 6.10 shows an excerpt of cached-CD based recovery of blocks of missing values performed with c-ReVival

Again, the dotted red line shows the recovery as computed by the cached-CD with a threshold of $\epsilon = 0.01$. The solid red line shows the removed values (original values we removed from the base time

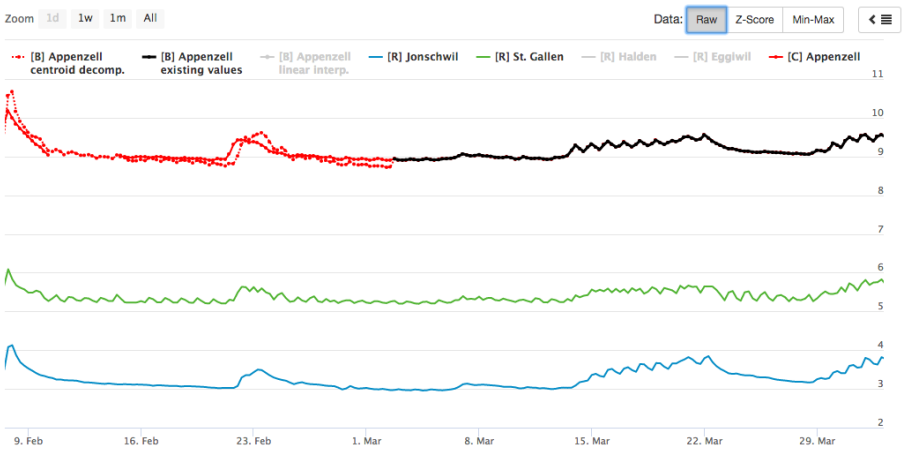
series to compare to our recovery). The black line shows the existing values of the base time series used for recovery, and the other colored lines show the reference series.

When looking at the z-score normalized data representation (Figure 6.10(a)), as expected, the recovery does not quite have the same accuracy as that of the Temperature data set. The recovery reconstructs all the trends of the original, but either over- or underfits the original values slightly. The recovery seems to exaggerate the amplitude of the curves. However, the raw data representation (Figure 6.10(b)) shows nicely that the inaccuracy of the recovery observed in the z-score normalized data representation is only minor, and that at any timestamp the recovery is very close to the original.

A further recovery process on the Hydrology data set, visualized in Figure 6.11, confirms the aforementioned observations. In the z-score normalized data representation (Figure 6.11(a)), the recovery (dotted red line) gets most of the trends and curves right, but struggles slightly with the amplitude of the curves by over- or undershooting the peaks. Again, the raw data representation (Figure 6.11(b)) shows that the effect of over- and undershooting is only minor and the recovery still appears very accurate.

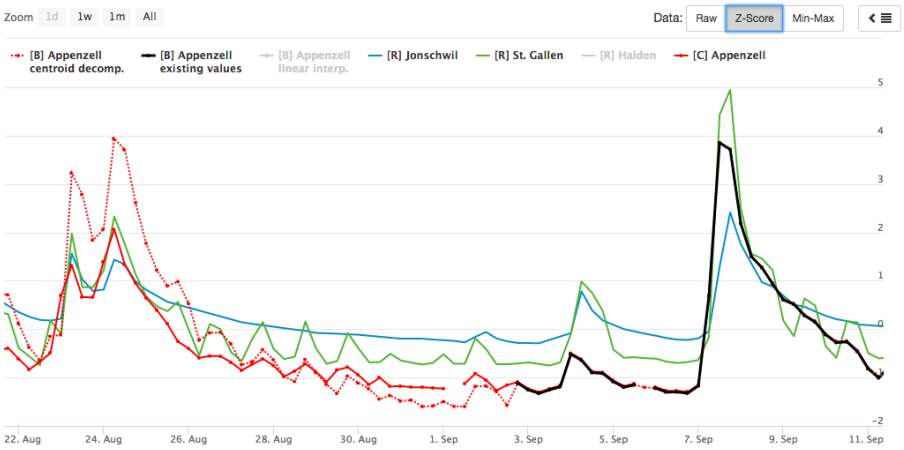


(a) Raw data representation.

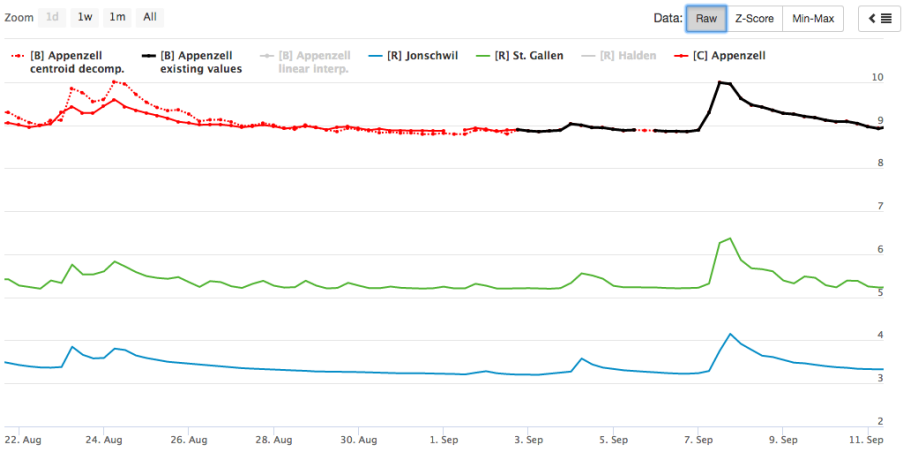


(b) z-score normalized representation.

Figure 6.10: Recovery performed on the Hydrology data set: case 1



(a) Raw data representation.



(b) z-score normalized representation.

Figure 6.11: Recovery performed on the Hydrology data set: case 2

We can conclude that the observed accuracy of CD based recovery does depend on characteristics of the data set. However, even when the accuracy seems slightly off in terms of amplitude/absolute distance between the original and the recovered values, the CD recognizes all the trends in the data, even if these are time-shifted between the time series.

6.2.2 Maximizing sign vector strategies

The sign vector strategies component of c-ReVival compares the strategy of finding the maximizing sign vector of the SSV algorithm to some of its (theoretical) variations: The DSV, TSV, and PSV (cf. Section 5.1.3). The goal of this component was initially to find methods of speeding up how the SSV algorithm finds the maximizing sign vectors. As explained in Section 4.1, the SSV presents to be the bottleneck to efficiently compute the CD of an input matrix.

A good indicator of the performance of a strategy for finding the maximizing sign vector Z_{max} is the progress of the product $Z^T \cdot V$ (maximizing product) for each iteration. Ideally, a strategy should reach the

maximizing product (and thus, Z_{max}) within as few iterations as possible. Figure 6.12 shows the evolution of $Z^T \cdot V$ for each strategy and iteration. The horizontal red line marks the value for $Z_{max}^T \cdot V$ (which was computed using the in-efficient brute-force approach of finding the maximizing sign vector Z_{max}). We can disregard the PSV algorithm straight away, as it clearly does not find the correct maximizing sign vector because it iteratively reduces the product $Z^T \cdot V$ before terminating. Interestingly, both the DSV and TSV do not even terminate for our input matrix $\mathbf{X}$. Instead, they seem to enter a 'race condition': Because the DSV and TSV can both switch more than one sign per iteration (two in the DSV, and three in the TSV), they can fall into an infinite loop where they continuously switch the sign of some specific element(s) in one iteration, only to switch it/them back again in the next iteration. Note that in the chart in Figure 6.12, we only show the first 8 iterations. Since the DSV and TSV do not terminate for our input matrix, the race-condition would continue to infinity.

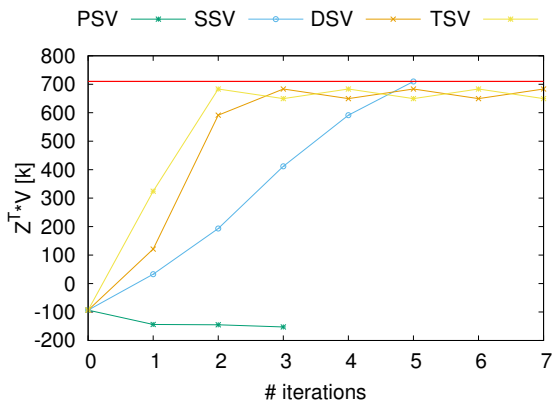


Figure 6.12: Maximizing sign vector strategy comparison

The take-away message from this component is the fact that the SSV finds the maximizing sign vectors more efficiently than any variations of the flipping sign strategies. More specifically, neither changing the criteria for deciding which element to switch (PSV), nor switching the sign of more than one element at a time (2 for the DSV, and 3 for the TSV) in a single iteration of the algorithm even produces the correct maximizing sign vector.

7
Conclusion and Future Work

In this thesis, we have introduced three main contributions. The first contribution is a new technique called cached-CD that i) allows an efficient computation with linear run-time complexity of the Centroid Decomposition (CD) for streams of time series and ii) reduces the run-time for the CD based recovery of missing values by a significant factor. In our experiments, we achieved a run-time reduction from hours to seconds for the computation of CD for streams of time series,

The second contribution of this thesis is the implementation of the updating-CD technique. We have compared its performance for computing the CD for streams of time series with that of the cached-CD and the regular-CD. The results of the comparison have shown that the cached-CD is massively more efficient than both the updating-CD and the regular-CD. In fact, the updating-CD leverages the the fact the loading matrix is a stationary point of the CD, which is not reflected on the way the CD decomposition is computed.

The third contribution is the development of c-ReVival, a graphical tool to visualize the properties of CD and the process of recovering missing values in time series using cached-CD. Through the inclusion of real-world data sets, c-ReVival also shows that it is indeed possible to recover missing blocks using the cached-CD based recovery on real-world data. Last but no least, c-ReVival visualizes the scalability improvement of our proposed cached-CD algorithm in comparison to the regular-CD and the updating-CD algorithms when applied to streams of time series.

In future work, it would be of interest to find a formal proof of the linear run-time complexity of the cached-CD for streams of time series. So far, we have only shown that the run-time of the cached-CD is based primarily on how similar the current input matrix is to the previous one (for which we cached the maximizing sign vectors). The worst-case run-time of the cached-CD is identical to that of the regular-CD (i.e., quadratic), if there are no maximizing sign vectors of a previous computation cached. However, as our experiments show, the average-case scenario is linear for streams of time series.

Another missing piece is the combination of cached-CD based recovery and cached-CD based computation of CD for streams of time series. It is yet to be determined and shown if the cached-CD is indeed capable of performing real-time on-line recovery on streams of updating time series. This problem is a bit more challenging since the recovery needs to be recomputed after each addition of rows to the input matrix, but there only being a certain, constant time between two consecutive additions.

Bibliography

[1] Mourad Khayati and Michael H Böhlen. Rebom: Recovery of blocks of missing values in time series. In *Proceedings of the 18th International Conference on Management of Data*, pages 44–55. Computer Society of India, 2012.

[2] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 171–182. ACM, 2010.

[3] Lei Li, James McCann, Nancy S Pollard, and Christos Faloutsos. Dynammo: Mining and summarization of coevolving sequences with missing values. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 507–516. ACM, 2009.

[4] Mourad Khayati, Michael H Böhlen, and Philippe Cudré Mauroux. Using lowly correlated time series to recover missing values in time series: A comparison between svd and cd. In *International Symposium on Spatial and Temporal Databases*, pages 237–254. Springer, 2015.

[5] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11. ACM, 2003.

[6] Georg Krempl, Indre Žliobaite, Dariusz Brzeziński, Eyke Hüllermeier, Mark Last, Vincent Lemaire, Tino Noack, Ammar Shaker, Sonja Sievi, Myra Spiliopoulou, et al. Open challenges for data stream mining research. *ACM SIGKDD explorations newsletter*, 16(1):1–10, 2014.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[8] Moody T Chu and Robert E Funderlic. The centroid decomposition: Relationships between discrete variational decompositions and svds. *SIAM Journal on Matrix Analysis and Applications*, 23(4):1025–1044, 2002.

[9] Mourad Khayati, Michael Böhlen, and Johann Gamper. Memory-efficient centroid decomposition for long time series. pages 100–111, 2014.

[10] Jason R Blevins and Moody T Chu. Updating the centroid decomposition with applications in lsi. Technical report, Technical report, 2004.

[11] Richard W Hamming. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 29(2):147–160, 1950.

[12] Ake Björck. *Numerical methods for least squares problems*. Siam, 1996.