The present work was submitted to:
Chair of Business Process Management Foundations and Engineering
RWTH Aachen University

Exact Earth Movers' Stochastic Conformance in Rust

Bachelor Thesis

by
Leonhard Mühlmeyer
434569

First Examiner: Prof. Dr. Sander J. J. Leemans
Second Examiner: Prof. Dr. Wil M. P. van der Aalst

Supervisor: Tobias Brockhoff, M.Sc.

Aachen, March 11, 2025

Abstract

Conformance checking is the collective term for techniques that allow businesses to measure how well an event log aligns with a process model. Stochastic process mining advocates to embed of stochastic information into process models such that a probability of every trace through the model can be derived. Accordingly, stochastic conformance checking techniques must also consider probabilities in the model. Earth Movers' Stochastic Conformance (EMSC) was the first proposed stochastic conformance checking technique. The reliance on approximate arithmetic in all publicly known implementations limits the reliability of stochastic conformance checking in critical applications. We show how one can scale trace probabilities and distances to integers, enabling the use of the integer flow theorem to obtain exact solutions via the Network Simplex Algorithm. Alongside this thesis, we thus provide an implementation that not only enables exact EMSC computation but, compared to the standard C implementations, guarantees safety due to the use of Rust. The implementation performs the internal optimization using the Network Simplex Algorithm because of its efficiency in practice. Additionally, we provide an evaluation on real-world event data showcasing that an exact EMSC computation has similar runtime compared to an approximate EMSC computation in most scenarios. We also demonstrate that the runtime of Network Simplex based implementations scales significantly better than the runtime of implementations using other optimization techniques when increasing the instance size.

Contents

1 Introduction **1**

2 Related Work **5**
2.1 Conformance Checking 5
2.2 Stochastic Process Mining 6
2.3 Stochastic Conformance Checking 6

3 Preliminaries **9**
3.1 Notation 9
3.2 Earth Movers' Stochastic Conformance Checking 9
3.3 The Transshipment Problem 12
3.4 The Network Simplex Algorithm 12
3.5 Alternative Transshipment Solving Approaches 17

4 Earth Movers' Stochastic Conformance Checking and Exact Arithmetic **19**
4.1 Transshipment Problem Formulation of EMD 19
4.2 Exact EMSC Using Scaling 20
4.3 Implementation Details and Usability 23

5 Evaluation **27**
5.1 Experimental Setup 27
5.2 Computation Time Comparison With Existing Methods 29
5.3 Deviation Comparison With Existing Methods 30
5.4 Computation Time Comparison of Exact With Approximate Mode 30
5.5 Internal Runtime Distribution per Task 33

6 Discussion **37**

7 Conclusion **39**

8 Appendix **41**
8.1 Evaluation Runtime Data 41

List of Figures

2.1 (a) Illustration of the Conventional Conformance Checking Setting and (b) the Stochastic Conformance Checking Setting 7

3.1 (a) Example Input Network and (b) Network With Added Artificial Root 14
3.2 Execution of Network Simplex on Example Network 16

4.1 Transshipment Problem Instance G, b, c for EMD of L_1 and L_2 with flow x induced by r_2 annotated in blue 20
4.2 EMD Transshipment Problem Instance G, b, c and Its Scaled Up Version G, b', c' 23
4.3 Activity Diagram of EMSC Calculation Control Flow 24

5.1 Runtime Comparison of Ebi With Other Implementations 31
5.2 Deviation from Exact EMSC Value 32
5.3 Runtime Comparison of Exact and Approximate Arithmetic 34
5.4 Ebi (Approximate/Exact) - Internal Runtime Distribution per Task . . . 36
5.5 Python EMD - Internal Runtime Distribution per Task 36
5.6 ProM - Internal Runtime Distribution per Task 36

List of Tables

5.1 Real-Life Event Logs Selected for Evaluation 28
5.2 Time Spent on Subtasks (in Seconds) for BPIC 2018 PA and Model Language of 2^8 Samples 35

8.1 Running Time (Seconds and Normalized) - 1 42
8.2 Running Time (Seconds and Normalized) - 2 43
8.3 Running Time (Seconds and Normalized) - 3 44
8.4 Running Time (Seconds and Normalized) - 4 45
8.5 Running Time (Seconds and Normalized) - 5 46

1 Introduction

Over the last decades, the share of businesses that record events in their operation in so-called *Process-aware Information Systems* has increased [12]. By doing so, they expect to gain insights into their processes and to derive guidelines on how to improve them. These records are called *event logs* and display a sample from the underlying "true" process.

Another fundamental aspect of modern Business Process Management is the modeling of processes, a practice that has evolved over the past century [1]. Here, the goal is to visualize how the different activities within the operation are, or ideally should be, temporally interconnected (e.g. whether a car is painted before the wheels are mounted or vice versa). Examples of such process models are *Directly-Follows Graphs*, *Unified Model Language activity diagrams*, *Business Process Model and Notation models*, and *Petri Nets*. Since the late 1990s, process mining research has developed many algorithms to discover a process model from an event log automatically.

Conformance Checking is the collective term for techniques evaluating how well a given process model describes the present behavior in a given event log [8]. It aims at identifying which behavior within the event log cannot be explained by the model.

In recent years, the process mining research community has pointed out that, until now, only process models have been considered in which behavior is either permitted by the model or not. Leemans et al. [21] argue that a fundamental issue of traditional conformance checking is caused by only considering frequencies of behavior on the side of the event log, while inherently ignoring likelihoods on the model side. This renders standard metrics, such as *precision*, meaningless when working with models containing loops, thereby allowing for infinite behavior.

Suppose we model an ideal morning routine in a model that allows to capture probability for each sequence of activities. Note that this is not supported by convention. The behavior allowed by the model is described by the *stochastic language* of the model, in this case

$$[\langle \text{get up, breakfast, use phone} \rangle^{0.9}, \langle \text{use phone, get up, breakfast} \rangle^{0.1}],$$

where $\langle$get up, breakfast, use phone$\rangle$ is such a sequence of activities that, according to the model, occurs in 90% of cases. The person now evaluates how well they adhere to their plan by tracking their morning routine over a week in an event log:

$$[\langle \text{get up, breakfast, use phone} \rangle^{2}, \langle \text{use phone, get up, breakfast} \rangle^{5}].$$

One can see that they did not manage to stick to the intended plan. However, con-

ventional conformance checking would conclude that the recorded morning routine fully conforms with the model.

To account for frequencies on the model side, stochastic models can be used. These are models in which on, at any point during a run, can derive the probability of any event being the next executed event. For example, so-called *Stochastic Labeled Petri Nets* fulfill this property.

Until now a couple of *stochastic conformance checking* techniques have been proposed [25, 26, 28, 30]. *Earth Movers' Stochastic Conformance* (EMSC) [26] was the first proposal. It considers two probability distributions over traces, namely the *stochastic languages* of the model and the event log, and measures how much probability mass from the log distribution needs to be moved to reshape the distribution to the one of the model. Morever, it regards the similarity of two traces, i.e. how far probability mass needs to be moves. This aspect is ignored by the *unit EMSC* (uEMSC) [26] and other techniques [25, 28, 30].

All publicly known implementations of the EMSC are built on *approximate arithmetic* and thus the trace probabilities are internally represented by floating-point types. This is mostly due to performance reasons as operating systems and hardware are heavily optimized for approximate arithmetic operations. Generally, stochastic process mining often involves extremely small probabilities that can fall below the precision limits of floating-point types [23]. Hence, exact arithmetic is essential for ensuring the reliability of EMSC results, as it eliminates errors caused by the imprecision of floating-point calculations. Furthermore, it enables benchmarking of existing implementations by providing an exact reference point for evaluating their precision. In this thesis, we present a method that computes the exact EMSC value of two stochastic languages using *exact arithmetic* and the *Network Simplex* optimization algorithm. Aiming for a more efficient computation, we scale trace probabilities and distances and thereby allow to perform the optimization on integer types.

We implemented our method to compute the exact EMSC value in Rust. Additionally, a mode is provided to calculate the approximate EMSC value using a floating-point type. The implementation performs the optimization using the Network Simplex Algorithm. To this end, we translated and extended the existing C-implementations from the Lemon library [10] and the Python Optimal Transport library [13].

We evaluate how runtimes differ when calculating the exact EMSC instead of using the approximate mode based on publicly available real-life event logs. Additionally we compare runtimes and deviations of the provided approximate mode to other EMSC implementations and investigate in which step of the EMSC computation implementations spend the most time.

We show that, in practice, standard finite-length integer types usually suffice, which results in performance comparable to calculating the EMSC value using the approximate mode. Furthermore, approximate EMSC values calculated by this thesis' implementation are found to be 10^3 to 10^7 times more precise on average than other implementations.

This thesis is structured in the following way. First, we discuss related work in the field of process mining in Chapter 2. Chapter 3 introduces EMSC, defines the under-

lying optimization problem, explains the Network Simplex Algorithm as a means to compute EMSC, and mentions alternative optimization approaches. We elaborate how we enable an exact EMSC computation by scaling trace probabilities and distance in Chapter 4. This is followed by an evaluation of the implementation in Chapter 5, where its performance is compared with the EMSC implementation in ProM [21], the EMD implementation in PM4Py [5], and an unpublished Python EMD implementation on real-world event data. Following a discussion of the findings in Chapter 6, this thesis draws its conclusions in Chapter 7.

2 Related Work

This chapter provides an overview of related work within the field of process mining, beginning with a description of how conformance checking is performed traditionally in Section 2.1. In Section 2.2 we introduce stochastic process mining and elaborate on existing stochastic conformance measures in Section 2.3.

2.1 Conformance Checking

The goal of conformance checking is to examine how the behavior modeled by a model and the behavior recorded in an event log are related. Carmona et al. [8] provide a detailed overview of conformance checking techniques over an event log and a non-stochastic process model. The two most fundamental concepts in conformance checking are *fitness* and *precision*. While fitness measures the share of traces in the log that are replayable on the model, precision measures the share of traces in the model that occur in the event log. Note that models may have loops and hence allow for infinite behavior, which causes precision to become zero.

However, fitness and precision have a binary notion of replayability, i.e. a trace is replayable on a model or not. This disregards how much a non-replayable trace deviates from other replayable traces or in other words: How unfitting is a particular trace? This issue is targeted by several techniques, two of which are introduced in the following.

Rozinat et al. [33] introduce token-based replay conformance checking. The technique works by replaying each trace on the model and recording how many tokens were *produced, consumed, missing,* and *remaining.* Based on these values a fitness measure is computed that penalizes log traces for greater deviations from the behavior allowed by the model. A central issue of token-based replay is that both silent and duplicate labels lead to ambiguities. Additionally, executing token-based replay does not indicate a closest accepted path in the model.

Alignment-based conformance checking, proposed by Adriansyah et al. [2], tackles this. It directly relates a trace in the log to an execution sequence on the model. Similar to token-based replay, it operates by replaying each trace on the model. However, the alignment computation records a sequence of *synchronous moves, log moves,* and *model moves.* An optimal alignment is an alignment containing the fewest log and model moves. For an optimal alignment of a trace and a model, it holds that the fewer log and model moves occur in the alignment, the better the trace fits the model.

2.2 Stochastic Process Mining

Initially, it is important to clarify the stochastic setting. For the remainder of this thesis, we work with the setting of a *stochastic process model* and a *deterministically known log*, i.e. a standard event log. This opposes the setting used by Gal et al. [6, 14] where a *standard process model* and a *stochastically known log* are assumed. Stochastically known logs are logs where at least one event attribute is defined by a probability distribution over a set of potential attribute values.

As the introductory example of a person trying to improve their morning routine demonstrates, there is great meaningfulness to stochastic information when defining how a process should look. This motivates the introduction of probabilistic information into process models. Such *stochastic process models* open the field of *stochastic process mining*. On the one hand, it unlocks potential that has implicitly been ignored by conventional process mining, on the other hand, this renders many established techniques in need of a stochastic revision.

Stochastic process mining research is mainly split into *stochastic process discovery* and *stochastic conformance checking*. Usually, businesses do not own any stochastic process models, while non-stochastic process models are well-established. This motivates the idea of enriching existing process models with stochastic information based on an event log [7, 24]. Other techniques directly discover stochastic models from event logs [3, 32].

2.3 Stochastic Conformance Checking

Both conformance checking techniques introduced in Section 2.1 take the multiplicity of trace variants in the log into account. Leemans et al. [21] indicate that non-stochastic conformance checking is inherently asymmetric. Specifically, on the model side, there is no equivalent notion of multiplicity or probability of variants in the model. This is symbolically visualized in Figure 2.1 (a), where a point on the plot of L denotes the multiplicity of a particular trace in the log. The M marks the subset of all traces allowed by the model. In case the model includes loops, M will include infinitely many traces that do not occur in L. Non-stochastic conformance checking evaluates both directions of inclusion between log and model when calculating fitness and precision, causing an issue for such models.

The stochastic conformance checking setting is illustrated in Figure 2.1 (b). Here, both log and model are viewed as probability distributions over the traces. On the contrary, stochastic conformance checking moves away from this notion of sets, as inclusions do not work on probability distributions. This makes using only a single similarity measure suitable [26].

Earth Movers' Stochastic Conformance (EMSC) [26] is the first proposed stochastic conformance checking technique. It converts a given event log into the respective *stochastic language* by normalizing the quantities of the traces. Additionally, it requires a stochastic language representing the behavior allowed by a given stochastic model. Note that complete stochastic languages can only be computed if the model does not

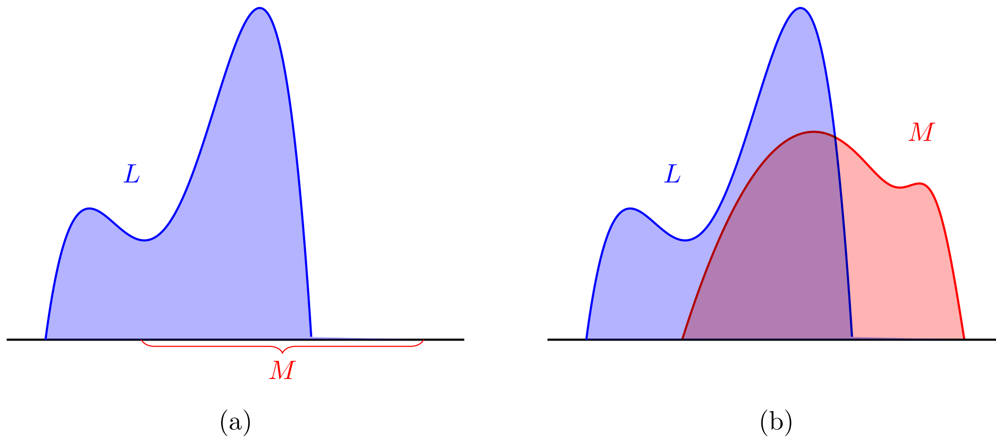

Figure 2.1: (a) Illustration of the Conventional Conformance Checking Setting and (b) the Stochastic Conformance Checking Setting

allows for infinite behavior. In the next step, EMSC calculates the Levenshtein distance between all pairs of traces from the log and the model language [27]. Based on the trace distances, it computes the Earth Movers' Distance (EMD) [35] between the two stochastic languages by optimizing a reallocation cost function. The analogy behind this is a pile of earth (the stochastic log language) which needs to be transformed into a pile of another form (the stochastic model language), where the distance expresses how difficult it is to move a certain share of earth somewhere else. This analogy aligns well with the symbolic illustration of the stochastic process mining setting in Figure 2.1 (b).

Additionally, two further EMD-based methods are presented to cope with infinite model languages, namely *unit Earth Movers' Stochastic Conformance Checking* (uEMSC) and *truncated Earth Movers' Stochastic Conformance Checking* (tEMSC) [26]. For uEMSC the distance function is defined to be one for two equal traces and zero otherwise. This simplifies the calculation because one only needs to check if the log traces are accepted by the model. Hence models with an infinite language do not cause an issue. The tEMSC takes another approach to handling infinite behavior. It is based on an *unfolding process*, which operates by identifying new traces in the model, calculating their respective probability, and adding them to the stochastic language until a certain provided probability mass $p \in [0, 1]$ is covered. If p approaches 1, the tEMSC value approaches the theoretical EMSC value. However, unfolding introduces a bias towards shorter likely traces by definition. To counteract this bias, Li et al. [28] suggest random walk sampling, where a stochastic language is generated by simulating runs through the model and setting trace probabilities based on their occurrence in the sampled traces.

Leemans et al. introduce support for silent and duplicate activities in their follow-up paper on EMSC [21]. Furthermore, the improved version provides diagnostics projected onto the event log and model as well as improved performance by changing the method of optimization.

The *entropic relevance* measure for stochastic conformance checking, proposed by

Polyvyanyy et al. [30], evaluates trace compression based on model likelihoods, addressing precision and recall simultaneously and demonstrating efficient computation.

Leemans et al. [25] introduce precision and recall measures based on the entropy of stochastic deterministic finite automata, which are derived by projecting automata representing log and model languages onto each other. The technique enables the distinction between common and rare behavioral deviations.

Like EMSC, the *Jensen-Shannon Conformance* measure, suggested by Li et al. [28] is defined on two probability distributions over traces. Among the stochastic conformance checking techniques mentioned in this overview, this is the only distance measure that qualifies as a metric.

Although the techniques mention above [25, 28, 30] have desirable properties, EMSC remains the only stochastic conformance checking technique that incorporates inter-trace similarities into its computation. This displays a distinguishing advantage of EMSC.

3 Preliminaries

This chapter first defines the fundamental notations for this thesis in Section 3.1 and explains *EMSC* in Section 3.2. Section 3.3 then introduces the central optimization problem of this thesis, to which an optimization algorithm, namely the *Network Simplex Algorithm*, is presented in Section 3.4.

3.1 Notation

The following notations will be used throughout this thesis.

Definition 3.1.1. *Event log.* A *multiset* is mapping of set of elements to the natural numbers. *Event Logs.* Let Σ denote the finite set of *activities*. Any sequence of activities $t \in \Sigma^*$ is called a *trace*. An *event log* is defined as a finite multiset over such traces.

Definition 3.1.2. *Stochastic Language.* A *stochastic language* is a function $L : \Sigma^* \to [0, 1]$ such that $\sum_{t \in \Sigma^*} L(t) = 1$. It assigns a certain probability to each trace. A trace t is said to be in the stochastic language L, denoted by $t \in L$, if and only if $L(t) > 0$. For a finite stochastic language L with $L(t_1), \ldots, L(t_n) > 0$ and $L(t') = 0$ for all other traces t', we also write $L = [t_1^{L(t_1)}, \ldots, t_n^{L(t_n)}]$.

Definition 3.1.3. *Directed Graph.* A *directed graph* $G = (V, E)$ is a tuple consisting of a finite set of nodes V and a set of arcs $E \subseteq V \times V$, where each arc represents a directed connection from one node to another. A *chain* on G is a sequence of nodes $v_1, \ldots, v_n$ such that for all $i \in \{1, \ldots, n-1\}$ it holds that $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$. A graph is said to be *connected* if each pair of nodes $v \neq v' \in V$ is connected by a chain.

Definition 3.1.4. *Cycle.* Let $G = (V, E)$ be a directed graph. A *cycle* on G is a chain $v_1, \ldots, v_n$ which additionally satisfies $(v_n, v_1) \in E$ or $(v_1, v_n) \in E$.

Definition 3.1.5. *Spanning Tree.* Let $G = (V, E)$ be a directed and connected graph. A *spanning tree* is a subgraph $G' = (V, E')$ of G, $E' \subseteq E$ such that there are no cycles on G'. A *rooted* spanning tree is a spanning tree, where one particular node r is defined as the *root node* of the tree. A node $v' \in V$ is a *successor* of $v \in V$ in the spanning tree G' if and only if v is on the chain connecting v' to the root r in G'.

3.2 Earth Movers' Stochastic Conformance Checking

After a broad introduction to EMSC in Section 2.3, this Section goes into more mathematical detail and introduces a running example, to which further chapters refer.

EMSC can only be applied directly to finite *stochastic languages*. While the conversion of an event log to a finite stochastic language is achieved by normalizing the multiplicities of trace variants, the conversion of a process model to a finite stochastic language is not as straightforward. Section 2.3 pointed out unfolding and random walk sampling as two techniques for generating a finite stochastic language from a model. Note that unfolding formally does not qualify as a stochastic language by Def. 3.1.2 as it not necessarily covers 100% of the probability mass. Random walk sampling works in the following manner: Given a stochastic process model M, let Σ denote the set of activities and let $\hat{t}_1, \ldots, \hat{t}_n$ be n independent random model executions, where $\hat{t}_1, \ldots, \hat{t}_n \in \Sigma^*$. Then, the sampled stochastic language $L_{M,n} = [t^{\frac{m}{n}} \mid t \in \{\hat{t}_1, \ldots, \hat{t}_n\}, m = |\{i \mid \hat{t}_i = t\}|]$ is an unbiased estimator of the model's language [28]. For the remainder of this thesis, we will assume that two finite stochastic languages are given, without a focus on their retrieval.

EMSC is defined based on the EMD [35] between two finite stochastic languages L_1 and L_2. These are essentially probability mass functions over their respective contained traces. Before defining the EMD, it is important to first introduce the distance function d and the reallocation function r. The distance between between traces t_1 and t_2 is defined by a general distance function $d(t_1, t_2)$. Typically, the normalized Levenshtein distance is used, calculated as the number of necessary edit operations (substitution, deletion, insertion of an activity) to turn t_1 into t_2 divided by the maximum length of t_1 and t_2 [27].

Definition 3.2.1. *Reallocation function.* Let L_1 and L_2 be two stochastic languages. A *reallocation function* is a function

$$\begin{aligned} & r : L_1 \times L_2 \to [0,1] \\ \text{such that} \quad & \forall t \in L_1 \sum_{t' \in L_2} r(t,t') = L_1(t), \\ & \forall t' \in L_2 \sum_{t \in L_1} r(t,t') = L_2(t'), \\ \text{and} \quad & \forall t \in L_1 \forall t' \in L_2 \ x_{tt'} \geq 0 \end{aligned}$$

The set of all such reallocation functions is denoted as $\mathcal{R}$.

Here the constraints ensure that the exact probability mass of a log trace is moved to model traces and vice versa. We now define a linear program (LP) for the EMD, based on the reallocation function constraints [26].

Definition 3.2.2. *Earth Movers' Distance.* Let L_1, L_2 be two stochastic languages and let d be a distance function. The $\text{EMD}(d, L_1, L_2)$ is defined as the solution to the following LP:

$$\begin{aligned} \textbf{minimize} \quad & \sum_{t \in L_1} \sum_{t' \in L_2} r(t,t') \cdot d(t,t') \\ \textbf{subject to} \quad & r \in \mathcal{R}. \end{aligned}$$

The EMSC(d, L_1, L_2) value is $1 - \text{EMD}(d, L_1, L_2)$. Additionally, we define

$$cost(r, d, L_1, L_2) = \sum_{t \in L_1} \sum_{t' \in L_2} r(t, t') \cdot d(t, t')$$

to evaluate the cost of a particular reallocation function r.

In the original EMSC paper [26], a general-purpose LP solver calculates the EMD. The follow-up paper [21] claims that this step is the most time-consuming and instead proposes to use of a variant of the *Exterior Point Simplex Algorithm* [29] for the Transshipment Problem (see Section 3.3).

Example. Suppose L_1 and L_2 are the following two stochastic languages:

$$L_1 = [\langle a, b, b, c\rangle^{0.5}, \langle a, c\rangle^{0.2}, \langle a, a, c, b\rangle^{0.3}],$$
$$L_2 = [\langle a, a, b, c\rangle^{0.3}, \langle a, b, b, c\rangle^{0.4}, \langle a, b, c\rangle^{0.3}].$$

To transform the first trace from L_1, namely $\langle a, b, b, c\rangle$, into the first trace of L_2, $\langle a, a, b, c\rangle$, only a single substitution operation is necessary. Thus the obtained normalized Levenshtein distance between the two traces is $\frac{1}{4}$ as the number of edit operations is divided by the maximum length of the traces. All other distances are given by

d	$\langle a, a, b, c\rangle$	$\langle a, b, b, c\rangle$	$\langle a, b, c\rangle$
$\langle a, b, b, c\rangle$	$\frac{1}{4}$	0	$\frac{1}{4}$
$\langle a, c\rangle$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{3}$
$\langle a, a, c, b\rangle$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{1}{2}$

In the following, two exemplary reallocation functions $r_1, r_2 \in \mathcal{R}$ are presented in tabular form:

r_1	$\langle a, a, b, c\rangle^{0.3}$	$\langle a, b, b, c\rangle^{0.4}$	$\langle a, b, c\rangle^{0.3}$
$\langle a, b, b, c\rangle^{0.5}$	0	0.2	0.3
$\langle a, c\rangle^{0.2}$	0	0.2	0
$\langle a, a, c, b\rangle^{0.3}$	0.3	0	0

r_2	$\langle a, a, b, c\rangle^{0.3}$	$\langle a, b, b, c\rangle^{0.4}$	$\langle a, b, c\rangle^{0.3}$
$\langle a, b, b, c\rangle^{0.5}$	0	0.4	0.1
$\langle a, c\rangle^{0.2}$	0	0	0.2
$\langle a, a, c, b\rangle^{0.3}$	0.3	0	0

For example, r_1 reallocates a probability mass of 0.2 from $\langle a, b, b, c\rangle$ to $\langle a, b, b, c\rangle$ and 0.3 to $\langle a, b, c\rangle$. This sums to $0.5 = L_1(\langle a, b, b, c\rangle)$. Note that both functions fulfill the constraints from the reallocation function definition. Inserting both reallocation functions into the cost function yields $cost(r_1, d, L_1, L_2) = 0.325$ and $cost(r_2, d, L_1, L_2) \approx 0.242$, where d is the Levenshtein distance function. This renders r_2 the cheaper and hence better reallocation function. Indeed, r_2 is optimal in the sense that there is

no reallocation function with lower cost. Note that there might be multiple optimal reallocation functions. Since the cost of r_2 is minimal, it represents the EMD value between L_1 and L_2, and the EMSC value is finally derived as $1 - 0.242 = 0.758$.

3.3 The Transshipment Problem

In the late 1930s, Russian mathematician Kantorovich studied several optimization problems to increase production yield [19]. In particular, Kantorovich discussed the "Best Plan of Freight Shipments" [18]. This problem was first formalized as a LP by Hitchcock in the early 1940s [17], known as the Hitchcock-Koopmans Transportation Problem. Here, the problem is characterized by shipping a product from m factories to n cities. Each factory has a certain supply, each city has a certain demand. Connections between factories and cities and potentially between a pair of factories or a pair of cities are associated with a certain cost, that is, how much it costs to ship one unit of product from A to B. This problem can be generalized as the Transshipment Problem (TP), where nodes without any supply or demand are also allowed.

Definition 3.3.1. *The Transshipment Problem.* Let $\mathcal{G} = (V, E)$ be a directed graph. Let $b(v)$ for $v \in V$ define the supply of node v. A node $v \in V$ with $b(v) > 0$ is a supply node, and accordingly, a node with $b(v) < 0$ is a demand node. Nodes $v \in V$ with $b(v) = 0$ are referred to as transshipment nodes. Let $c_{vv'}$ for $(v, v') \in E$ denote the cost of shipping one unit of product from v to v'. Such graphs with associated supplies and costs G, b, c are hereinafter referred to as *networks.* The problem is characterized by the following LP formulation:

$$
\begin{aligned}
\textbf{minimize} \quad & \sum_{(v,v') \in E} c_{vv'} x_{vv'} & \\
\textbf{subject to} \quad & \sum_{v':(v,v') \in E} x_{vv'} - \sum_{v':(v',v) \in E} x_{v'v} = b(v), & \text{for all } v \in V, \\
& x_{vv'} \geq 0, & \text{for all } (v, v') \in E.
\end{aligned}
$$

There are further generalizations, e.g. allowing for the definition of maximal capacities of the arcs. However, capacitated networks will not play a role in this thesis.

3.4 The Network Simplex Algorithm

In the following, we explain the Network Simplex Algorithm is and then execute it on an example network. It was introduced by Dantzig [11] as an adaptee of the standard simplex algorithm, specifically designed for the TP. Even though this explanation focuses on the uncapacitated case, the Network Simplex Algorithm is generally designed to be applied to capacitated networks as well.

Like the standard Simplex Algorithm, the Network Simplex Algorithm is based on continually improving a basic feasible solution (or basis). Let G, b, c be a network with

$G = (V, E)$. A solution of the network is an assignment of the flow variables $x_{vv'}$ for all $(v, v') \in E$ to concrete values. A solution is called feasible if it satisfies the constraints of the TP's LP formulation. It can be shown that if a network has a feasible solution and is bounded from below[1], there is at least one spanning tree solution to the problem [4, 19]. Such a solution is a solution where the subgraph induced by the set of arcs with positive flow $\{(v, v') \mid x_{vv'} > 0\}$ is cycle-free. To obtain a spanning tree, it may be necessary to additionally include some arcs with a flow of zero. Throughout the computation, a representation of the spanning tree is stored and updated according to the changes made to the basic feasible solution. Additionally, the Network Simplex Algorithm keeps track of dual variables π_v, the so-called potential, for each node v.

An initial basic feasible solution to a certain network can either be given as input or be set up algorithmically by adding an artificial node $r \notin V$, called the root node, to the network and connecting it to the existing nodes in the following manner:

$$\begin{aligned} \mathcal{G}' &= (V \cup \{r\}, E'), \\ E' &= E \cup \{(v, r) \mid b(v) \geq 0\} \cup \{(r, v) \mid b(v) < 0\}. \end{aligned}$$

The supply of r is set to $b(r) = -\sum_{v \in V} b(v)$. This displays a key advantage of adding an artificial node since it enables working with imbalanced inputs, i.e. networks where the sum of supplies is not zero. To ensure that the initial feasible solution is more expensive than any other feasible solution, an artificial cost value is identified and assigned to the newly created arcs as follows:

$$\begin{aligned} c_{\text{artificial}} &:= (\max_{(v,v') \in E} c_{vv'} + 1) \cdot |V|, \\ c_{vr} &:= \begin{cases} 0 & \text{if } \sum_{v' \in V} b(v') \geq 0, \\ c_{\text{artificial}} & \text{otherwise} \end{cases} \quad \text{for each } (v, r) \in E', \\ c_{rv} &:= \begin{cases} c_{\text{artificial}} & \text{if } \sum_{v' \in V} b(v') \geq 0, \\ 0 & \text{otherwise} \end{cases} \quad \text{for each } (r, v) \in E'. \end{aligned}$$

Fig. 3.1 (b) visualizes the above-described transformation for an example input network displayed in Fig. 3.1 (a). Nodes are annotated with their supply value (blue) and arcs with their respective cost (black). The artificial cost is identified as $c_{\text{artificial}} = (6 + 1) \cdot 5 = 35$. Supply nodes are connected to the artificial node r and the artificial r node is connected to demand nodes (red). Note that the sum of the supplies is positive and the costs are set accordingly (also red). The initial potentials (dual variables) are defined as $\pi_r = \pi_v = 0$ for $(v, r) \in E'$ and $\pi(v) = c_{\text{artificial}}$, for $(r, v) \in E'$.

One can observe that setting $x_{vr} = b(v)$ for nodes v with $b(v) \geq 0$, $x_{rv} = -b(v)$ for nodes v with $b(v) < 0$ and $x_{vv'} = 0$ for all $(v, v') \in E$ gives a feasible solution to the problem. This solution induces a spanning tree. The artificial node r is defined as the tree's root which will be maintained during the complete computation.

[1] This is for example not the case if there is a cycle in the network with negative cost. Then, infinitely sending the product along the cycle will further and further reduce the overall cost.

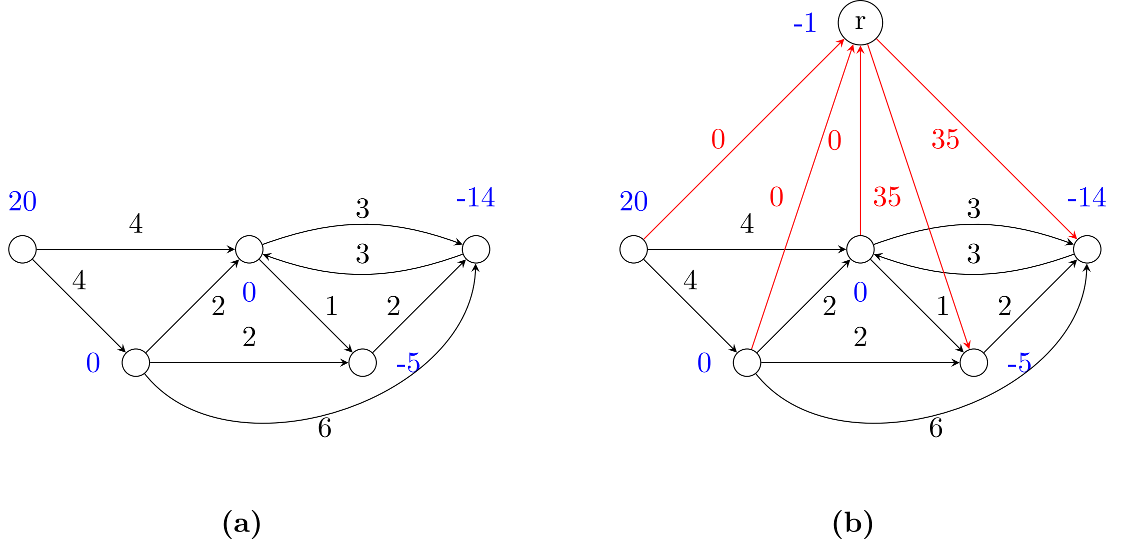

Figure 3.1: (a) Example Input Network and (b) Network With Added Artificial Root

An outline of the Network Simplex Algorithm [11] is given in Alg. 1, the following paragraph outlines the calculations within one iteration.

Algorithm 1 Network Simplex

1: **Generate** Initial Basic Feasible Solution Tree T
2: $(i, j) \leftarrow$ Find Entering Arc in $\mathcal{G} \setminus T$
3: **while** $(i, j) \neq$ NULL **do**
4: $\quad \mathcal{C} \leftarrow$ Find Cycle in $T \cup (i, j)$
5: $\quad \Delta \leftarrow$ Flow of Weakest Edge in $\mathcal{C}$
6: $\quad$ **Update** Flows on $\mathcal{C}$ by $\pm\Delta$
7: $\quad$ **Update** Basic Feasible Solution Tree T
8: $\quad$ **Update** Node Potentials
9: $\quad (i, j) \leftarrow$ Find Entering Arc $\in \mathcal{G} \setminus T$
10: **end while**
11: **Retrieve Solution**

Once an initial basic feasible solution is set, the Network Simplex Algorithm iteratively identifies an arc to enter the basis that will not worsen the current solution: To this end, for each arc $(v, v') \in E'$ that is not part of the current basis, we calculate its reduced cost $\bar{c}_{vv'} = c_{vv'} + \pi_v - \pi_{v'}$[2]. If there is a $\bar{c}_{vv'} < 0$, an arc (v, v') with a minimal $\bar{c}_{vv'}$ enters the basis (otherwise the solution is optimal). This closes a cycle in the current basis along which the algorithm identifies an arc with minimal flow that will then leave the basis. The flow of the leaving arc must be rerouted within the network and is saved

[2]Computing this for all arcs in each iteration is called the *Most Negative Pivot Rule*. At the end of this section we present a more efficient alternative.

as Δ. This Δ is added to all arcs on the identified cycle that share the entering arc's orientation within the cycle and is subtracted from all arcs on the cycle that oppose the entering arc's orientation within the cycle. The representation of the spanning tree is updated according to the described changes and the node potentials are recalculated: For an entering arc (v, v') let $u \in \{v, v'\}$ be the node that is farther from the root node r in the updated spanning tree. If $u = v$ subtract $\bar{c}_{vv'}$ from the potentials of all successors of u, if $u = v'$ add $\bar{c}_{vv'}$ to the potentials of all successors of u. Now continue with the next iteration by identifying the next entering arc. The complexity of the algorithm is $O((|E| + |V|) \cdot |E| \cdot |V| \cdot C^2 \cdot U)$, where n is the number of nodes, m the number of arcs, C denotes the maximal costs, and U the maximal flow [19].

When an optimal solution is found, the total cost can be retrieved as

$$\sum_{(v,v') \in E} c_{vv'} x_{vv'}.$$

The execution of the Network Simplex Algorithm on the network from Fig. 3.1 (b) where the artificial basis has been set up is depicted in Fig. 3.2. For each iteration, the flows along the current basis are displayed in column 2, and column 3 shows the current potentials. In column 4, the reduced costs of all non-basic arcs from the original network are calculated. The minimal reduced cost is highlighted (green) and displays the entering arc in the last column (also green). Across all cells of the figure, the relative position of each node remains the same. All arcs of the induced closed cycle are annotated with $+\Delta$ if they share the entering arc's orientation within the cycle and $-\Delta$ otherwise. The algorithm identifies Δ as the minimal flow of arcs that are annotated with $-\Delta$. The arc with the minimal flow is colored red and will leave the basis in the next iteration. In iteration 5, all reduced costs are positive and hence the solution is optimal. The optimal cost is

$$c_{\text{opt}} = 19 \cdot 4 + 19 \cdot 1 + 0 \cdot 2 + 14 \cdot 2 = 123.$$

Standard integrality theorems of the TP do not reason about the variables used within the optimization, but state that every extreme point of the feasible region is integer-valued [4]. However, one can observe that in this example the Network Simplex Algorithm only used integers throughout the computation. This observation indeed generalizes in the following way.

Theorem 3.4.1. *Integer flow theorem. For a Network Simplex instance where only integers are used for supplies, demands, and costs; all flows, potentials, and reduced costs at any point in the computation will be integers.*

Proof. Let $\mathcal{G} = (V, E), b, c_{vv'}$ for $(v, v') \in E$ be a Network Simplex instance with $b(v) \in \mathbb{Z}$ for all $v \in V$ and $c_{vv'} \in \mathbb{Z}$ for all $(v, v') \in E$.

Induction base: After initialization, all flow variables will be values from the set $\{|b(v)| \mid v \in V\}$. The potential π_v of each node v will either have value 0 or $c_{\text{artificial}} := (\max_{(v,v') \in E} c_{vv'} + 1) \cdot |V|$, which is an integer itself.

Induction step: For any iteration assume that all flows and potentials, calculated by the last iteration, are integers. For any given arc $(v, v') \in E$ its reduced cost is calculated

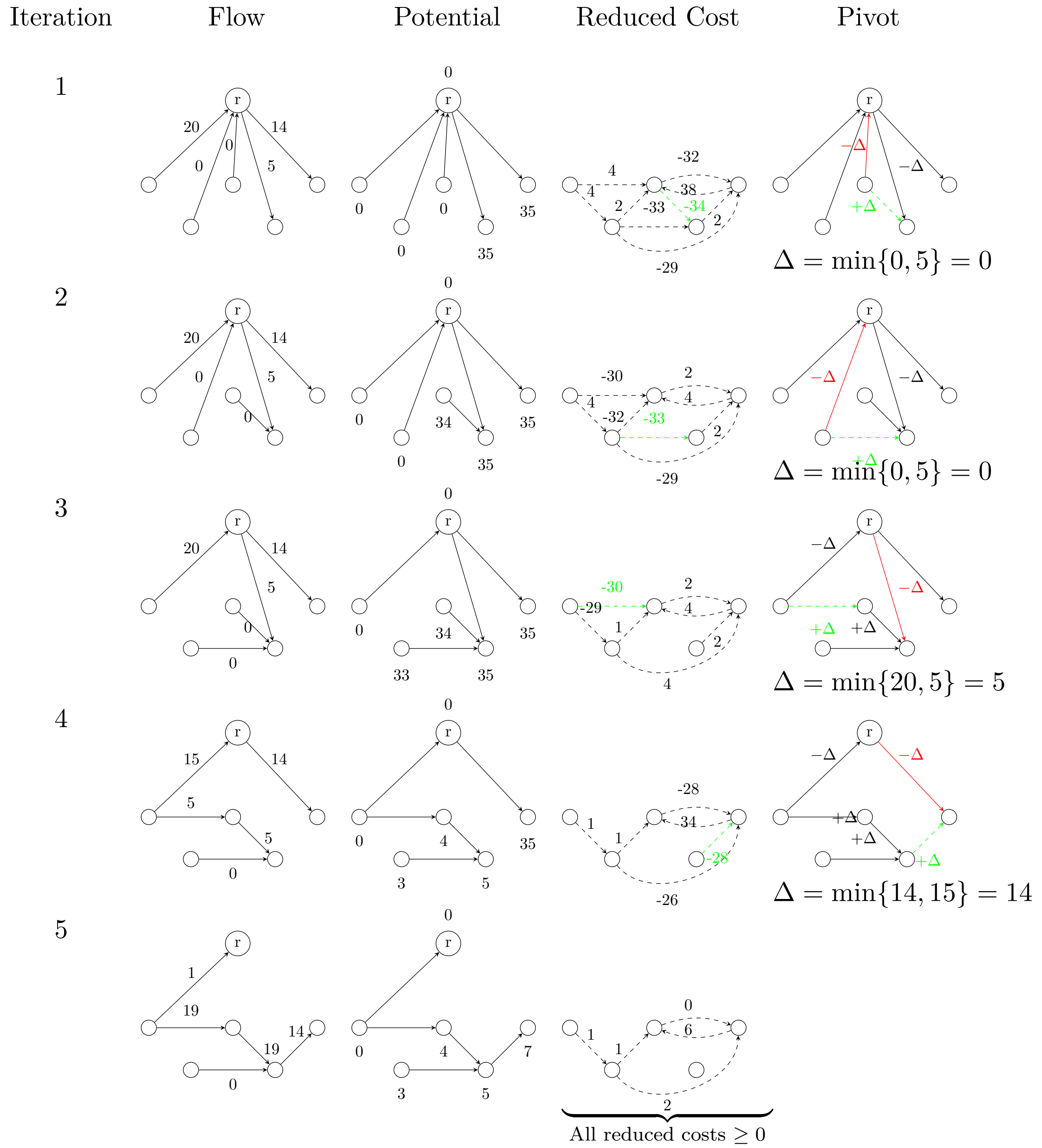

Figure 3.2: Execution of Network Simplex on Example Network

as $\bar{c}_{vv'} = c_{vv'} + \pi_v - \pi_{v'}$. By assumption, each of the summands and subtrahends is an integer. Since $\mathbb{Z}$ is closed under addition and subtraction, also $\bar{c}_{vv'} \in \mathbb{Z}$. Let $\bar{c}_{\min}$ denote the minimal reduced cost. Because Δ is computed as the minimum of a set of flows that are integers by assumption, its value is an integer as well. When updating the flows, each flow either remains the same, is increased by Δ, or decreased by Δ. Similarly, each potential either remains the same, is increased by $\bar{c}_{\min}$, or decreased by $\bar{c}_{\min}$ and hence also all potentials remain integer-valued. This leads to the conclusion that all updated flows and potentials remain integers. □

By induction, this proves that for integer inputs the Network Simplex can be implemented such that it works on integers only, when ensuring that the spanning tree representation is also restricted to using integers.

Sec. 4.1 elaborates how an instance of the TP can be set up as an input to the Network Simplex Algorithm to perform the optimization for the EMD computation.

Furthermore, there are multiple options for identifying the entering arc. These are defined under the term *pivot rules* [19]. The algorithm outlined above uses the *Most Negative Pivot Rule* and identifies the arc with the most negative reduced cost. However, this becomes expensive for larger inputs. Grigoriadis [16] suggests the use of the *Block Search Pivot Rule* which only finds the best arc within a certain block of arcs and considers different blocks in each iteration. This pivot rule is shown to perform significantly better in practice [19].

3.5 Alternative Transshipment Solving Approaches

Since the TP is one of the central optimization problems in operations research, it has been studied extensively over the past decades. This resulted in many research contributions in the field and hence this section only provides some contextualization of the developed techniques.

The Network Simplex Algorithm, as described in Section 3.4, is a combinatorial method specifically designed for the TP. It leverages the inherent structure of transportation networks to achieve dramatic performance improvements. In practice, when applied to the TP, the Network Simplex Algorithm is 200 and 300 times faster than the standard Simplex Algorithm [4].

In contrast to the Network Simplex approach that traverses the boundary of the feasible space, other algorithmic families solving the TP have emerged. For example, interior point methods follow a trajectory through the interior of the feasible region. These methods offer strong theoretical guarantees and can handle large-scale problems efficiently [31]. However, only recently research has focused on the development of an interior point method for the TP. Zanetti et al. [37] state that their implementation can compete with the Network Simplex Algorithm.

Morover, exterior point methods have emerged. These methods permit exiting the feasible region temporarily before restoring feasibility, often leading to a lower iteration count and improved performance [29].

Another family consists of cost scaling methods, which iteratively adjust the cost parameters to refine the flow assignments. Kovács [20] finds the *Partial Augment-Relabel* technique by Goldberg [15] to be a strong competitor to the Network Simplex. However, it only becomes faster than the Network Simplex on sparse networks with many nodes.

4 Earth Movers' Stochastic Conformance Checking and Exact Arithmetic

In this chapter, we present a method for efficiently computing an exact EMSC value. We reformulate the EMD definition into a TP instance in Section 4.1. Based on this, we then introduce the *Integer-Scaled TP*, which is the main contribution of this thesis, in Section 4.2. In Section 4.3 we point out how a reader can access the provided implementation and how it operates internally.

4.1 Transshipment Problem Formulation of EMD

As already indicated, the analogy behind EMD is moving earth from one distribution to another. In the case of EMSC, one tries to move the probability masses of the log traces to the probability masses of the model traces. This gives rise to a reformulation of the EMD between two stochastic languages as an instance of the TP as defined in Sec. 3.3 [34]. Various studies use the Network Simplex Algorithm to compute the EMD, and for the sake of clarity, we explicitly present the construction in this thesis. Given two finite stochastic languages L_1 and L_2, we define the following network:

$$\begin{aligned}
G &= (V, E),\\
\text{where } V &= \underbrace{\{(t,1) \mid t \in \Sigma^* \wedge L_1(t) > 0\}}_{:=V_1} \cup \underbrace{\{(t,2) \mid t \in \Sigma^* \wedge L_2(t) > 0\}}_{:=V_2},\\
E &= V_1 \times V_2,\\
b(v) &= \begin{cases} L_1(t) & \text{if } v = (t,1)\\ -L_2(t) & \text{if } v = (t,2), \end{cases}\\
c_{v_1 v_2} &= d(t_1, t_2), \text{ where } v_1 = (t_1, 1), v_2 = (t_2, 2),
\end{aligned}$$

and d denotes the normalized Levenshtein distance. The nodes are constructed such that the traces from L_1 are the factories or supply nodes producing their respective probability mass and the traces from L_2 are the cities or demand nodes. The set of arcs gathers all connections from L_1-traces to L_2-traces. The respective cost of an arc is set to the normalized Levenshtein distance of the traces that it connects.

It is key to understand that the constraints of the TP align with the reallocation function constraints of the EMD for the TP instance constructed above. The intuitive mapping of legal reallocation functions to flows and vice versa is a bijection. Moreover, if and only if a reallocation function is optimal, its respective flow is optimal as well.

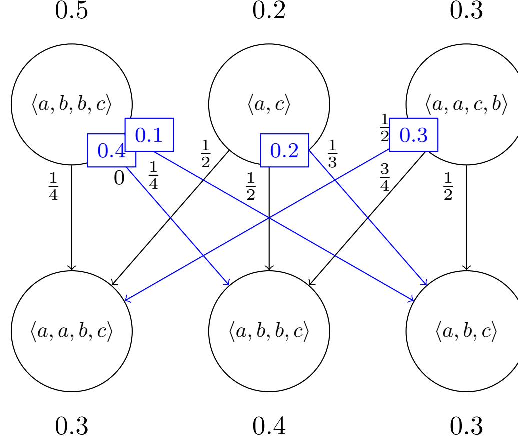

Figure 4.1: Transshipment Problem Instance G, b, c for EMD of L_1 and L_2 with flow x induced by r_2 annotated in blue

Example. Recall the stochastic languages L_1 and L_2 from the example in Section 3.2. The EMD of L_1 and L_2 can also be computed by identifying an optimal solution for the TP instance G, b, c, which is defined based on the two stochastic languages. The generated network is displayed in Figure 4.1. Additionally, the reallocation function r_2 from the example in Sec. 3.2 induces an optimal flow x in the network and thus a solution of the TP instance. The optimal flow is highlighted blue in Figure 4.1 and the flow values are annotated to the arcs in blue rectangles.

4.2 Exact EMSC Using Scaling

Stochastic process mining regularly has to deal with fractions smaller than the range of floating points, which causes calculations to become unreliable [23]. The more arithmetic operations a technique perform, the greater the unreliability. EMSC being a technique that requires many arithmetic operations motivates the need for an exact method of EMSC computation. However, exact computation usually comes with a performance cost: Any data type of fixed size is incapable of representing exact rational numbers. Instead, they are stored on the heap in two arbitrary-precision integers, one representing the nominator, the other the denominator. Compared to standard integers, they are stored on the heap and do not have a fixed length.

Theorem 3.4.1 shows that it is possible to perform the Network Simplex Algorithm solely on integers. This motivates the idea of scaling the supply values of the nodes and the costs of the arcs such that all supplies and distances become integers.

This is formalized as the *Integer-Scaled TP* for the TP on exact rational numbers.

Definition 4.2.1. *Integer-Scaled TP.* Let G, b, c be a network, where $G = (V, E)$ and for all $v, v' \in V$ it holds that $b(v) \in \mathbb{Q}$ and $c_{vv'} \in \mathbb{Q}$. Let $\gcd(a, b)$ denote the greatest common divisor of $a, b \in \mathbb{Z}$. Let $d_b := \{d \mid b(v) = \frac{n}{d}, v \in V, n, d \in \mathbb{Z},$ and $\gcd(n, d) =$

1} the set of supply denominators and let $d_c := \{d \mid c_{vv'} = \frac{n}{d}, (v,v)' \in E, n, d \in \mathbb{Z}$, and $\gcd(n,d) = 1\}$ be the set of cost denominators. We define the scaled network G, b', c' with

$$b'(v) := b(v) \cdot LCM(d_b),$$
$$c'_{v_1v_2} := c_{v_1v_2} \cdot LCM(d_c),$$

where $v, v_1, v_2 \in V$, $(v_1, v_2) \in E$ and LCM is the least common multiple of a given set.

Theorem 4.2.1. *Let $x_{vv'} \in \mathbb{Q}$ for all $(v, v') \in E$ denote any feasible flow on G, b, c. Then x induces the feasible flow $x'_{vv'} = x_{vv'} \cdot LCM(d_b)$ with $x'_{vv'} \in \mathbb{Z}$ for all $(v, v') \in E$ on G, b', c'. Also, G, b', c' is feasible if and only if G, b, c is feasible. The total costs of the flows x and x' behave in the following manner*

$$c_{total}(x) = \frac{c_{total}(x')}{LCM(d_b) \cdot LCM(d_c)}$$

Proof. Let G, b, c be a network with $G = (V, E)$ and let x be a feasible flow on G, b, c. Let x' with $x'_{vv'} := x_{vv'} \cdot \mathrm{LCM}(d_b)$ for all $(v, v) \in E$ be the flow on the Integer-Scaled TP network G, b', c' induced by x. Since x is feasible, $x_{vv'} \geq 0$ for all $(v, v') \in E$ and since $\mathrm{LCM}(d_b)$ is positive by definition, also $x'_{vv'} \geq 0$ for all $(v, v') \in E$. Additionally x satisfies for every node $v \in V$ the conservation constraint

$$\sum_{(v,v') \in E} x_{vv'} - \sum_{(v',v) \in E} x_{v'v} = b(v).$$

Multiplying both sides by $\mathrm{LCM}(d_b)$ yields

$$\sum_{(v,v') \in E} x_{vv'} \cdot \mathrm{LCM}(d_b) - \sum_{(v',v) \in E} x_{v'v} \cdot \mathrm{LCM}(d_b) = b(v) \cdot \mathrm{LCM}(d_b) = b'(v).$$

Hence, the flow x' is feasible on the network G, b', c'.

Since each $x_{vv'}$ is a rational number and $\mathrm{LCM}(d_b)$ is chosen as a common multiple of the denominators of all $x_{vv'}$, the product $x_{vv'} \cdot \mathrm{LCM}(d_b)$ is an integer for all $(v, v') \in E$.

Conversely, let x be an infeasible flow on G, b, c and let x' be the respective induced flow. Then there is at least one constraint that x does not satisfy. If $x_{vv'} < 0$ for some arc $(v, v') \in E$, then also $x'_{vv'} < 0$ since $\mathrm{LCM}(d_b)$ is positive. Otherwise, there is a node $v \in V$ such that

$$\sum_{(v,v') \in E} x_{vv'} - \sum_{(v',v) \in E} x_{v'v} \neq b(v).$$

Since both sides are scaled by $\mathrm{LCM}(d_b)$, the inequality remains. In both cases, x' is an infeasible flow on G, b', c'. Since these arguments reason about all flows, it follows that G, b', c' is feasible if and only if G, b, c is feasible.

The total cost of the flow x in (G, b, c) is given by

$$c_{total}(x) = \sum_{(v,v') \in E} c_{vv'} x_{vv'}.$$

In the scaled network G, b', c', the total cost of the flow x' is

$$\begin{aligned} c_{total}(x') &= \sum_{(v,v') \in E} c'_{vv'} \cdot x'_{vv'} \\ &= \sum_{(v,v') \in E} \Big(c_{vv'} \cdot \text{LCM}(d_c)\Big)\Big(x_{vv'} \cdot \text{LCM}(d_b)\Big) \\ &= \text{LCM}(d_c) \cdot \text{LCM}(d_b) \cdot \sum_{(v,v') \in E} c_{vv'} x_{vv'} \\ &= \text{LCM}(d_c) \cdot \text{LCM}(d_b) \cdot c_{total}(x). \end{aligned}$$

Rearranging, we obtain

$$c_{total}(x) = \frac{c_{total}(x')}{\text{LCM}(d_b) \cdot \text{LCM}(d_c)}.$$

□

In particular, this implies that the optimal cost of the instance G, b, c can be calculated by dividing the optimal cost of the instance G, b', c' by the two calculated LCMs.

Computational Complexity. Computing the LCM of two numbers a and b using the Euclidean Algorithm has complexity $O(\log \min(a, b))$. To obtain the LCM of a set of numbers, one can iteratively calculate pairwise LCMs. Computing the LCM of the supplies then has complexity $O(|V| \log b)$, where b denotes the average absolute supply, scaling the supplies has complexity $O(|V|)$. Analogously, computing the LCM of the costs has complexity $O(|E| \log c)$, where c denotes the average cost, and scaling the costs has complexity $O(|E|)$. In total this results in a complexity of $O(|V| \log b + |E| \log c)$. Considering the complexity $O((|E| + |V|) \cdot |E| \cdot |V| \cdot C^2 \cdot U)$ of the Network Simplex, where C is the maximal cost and U the maximal flow, this additional cost is negligible.

To illustrate how we create an integer-scaled TP instance, the running example of the EMD between L_1 and L_2 is revisited once again.

Example. In the example in Sec. 4.1 the equivalent instance of the TP G, b, c has been created. Since all supply and cost values are rational numbers, the Integer-Scaled TP transformation can be applied.

At first, the supply denominator set is identified as $d_b = \{2, 5, 10\}$. Then all supply values are scaled by $\text{LCM}(d_b) = 10$ and stored as b'. The cost values of the arcs are scaled by $\text{LCM}(d_c) = 12$ into c' for the identified cost denominator set $d_c = \{4, 2, 3\}$.

The network is presented in Figure 4.2 before and after the Integer-Scaled TP instance was generated. The optimal flow x, highlighted in blue in Figure 4.1, can be scaled by

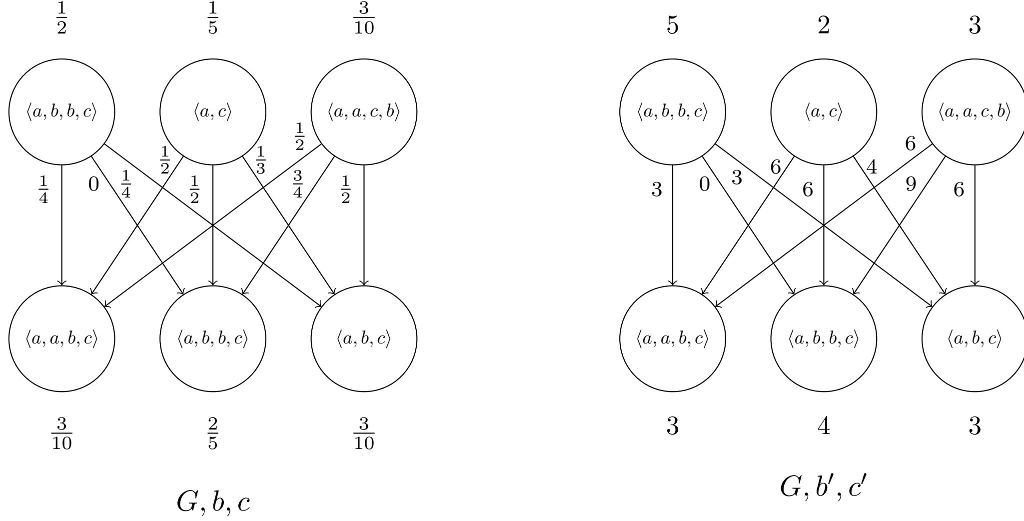

Figure 4.2: EMD Transshipment Problem Instance G, b, c and Its Scaled Up Version G, b', c'

$\text{LCM}(d_b)$ and displays an optimal flow in G, b', c' as well.

4.3 Implementation Details and Usability

This thesis provides a Rust implementation of EMSC that is publicly accessible and integrated into the open-source stochastic process mining framework Ebi[1] [23]. For the remainder of the thesis, the term Ebi implementation will be used to refer to the implementation provided alongside this thesis for simplicity. While the EMSC implementation itself is only defined for *stochastic languages*, Ebi automatically handles conversions, allowing the user to apply EMSC to all pairs of *event logs* (.xes or .xes.gz) and *finite stochastic languages* (.slang). It is exposed to the user from the command `ebi conformance emsc ⟨FILE 1⟩ ⟨FILE 2⟩`. By default, the EMSC value is calculated using exact arithmetic. Users can compute the approximate EMSC value by setting the `-a` flag. Additionally, the command `ebi conformance emsc-sample ⟨FILE 1⟩ ⟨FILE 2⟩ ⟨NUMBER OF SAMPLES⟩` is provided. It allows the user to apply EMSC to a *stochastic model* when specifying the number of traces to be sampled. Internally, Ebi samples ⟨NUMBER OF SAMPLES⟩ many random walks through the model using the technique discussed by Li et al. [28]. For more details on how to use the EMSC implementation, please refer to the Ebi manual linked on the Ebi website[1].

In the following, we sketch the Ebi implementation with a focus on its most important parts. It begins with the TP instance setup and proceeds with remarks on the implementation of the Network Simplex optimization algorithm.

[1] Ebi can installed via `https://ebitools.org` and is accessible from the command line.

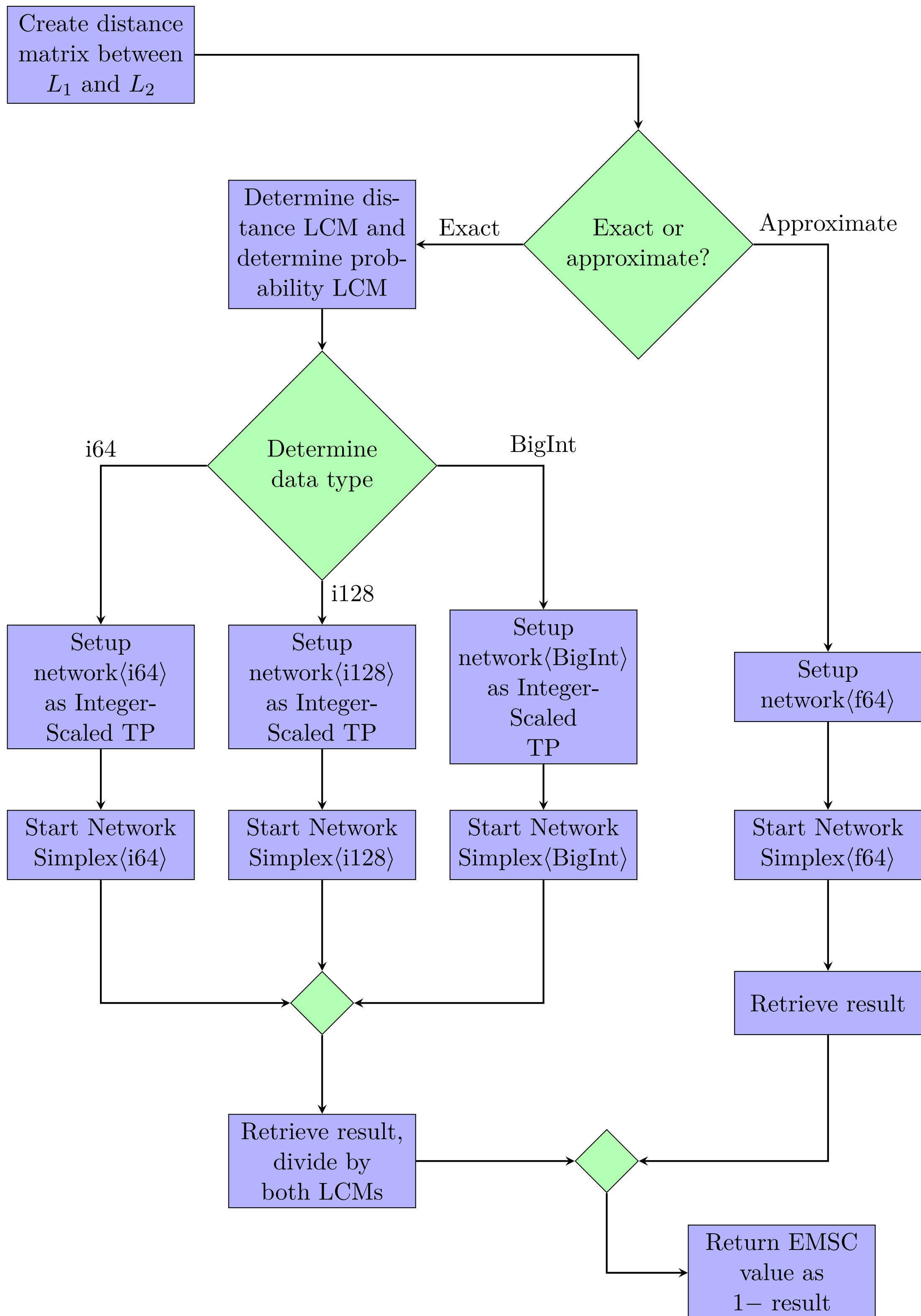

Figure 4.3: Activity Diagram of EMSC Calculation Control Flow

Figure 4.3 displays an activity diagram of the overall control flow of an EMSC computation of two stochastic languages L_1 and L_2.

First, we compute the normalized Levenshtein distance between all pairs of distances (t_1, t_2), where $t_1 \in L_1$ and $t_2 \in L_2$ and store the distances in a matrix.

If the user requests an approximate EMSC calculation, the EMD network is created as elaborated in Section 4.1. Due to floating-point rounding errors the sum of the supplies might be slightly smaller or greater than one. This might cause the first attempt to initialize the Network Simplex to fail, requiring a second attempt with adjusted parameters. The Network Simplex Algorithm is then performed on the created network and once it terminates, the approximate EMD value is retrieved as the optimal cost c_{opt} of the network.

In case the user instead requests an exact EMSC value, the algorithm creates the Integer-Scaled TP instance. First, the algorithm calculates the probability denominator LCM and the distance denominator LCM as described in Section 4.2. These LCMs implicitly determine an upper bound to the values of the variables within the Network Simplex execution. This allows for a decision at runtime whether 64-bit integers or 128-bit integers may suffice. Using the arbitrary-precision integer type BigInt remains the last resort and is only used if 128-bit might not suffice.

However, a potential threat to validity arises in the case of the potentials within the Network Simplex Algorithm. While we observe from experience that the maximal absolute potential values rapidly decrease from the initially high artificial cost, we cannot guarantee that this will always hold. Although we are not aware of any instance where an absolute potential exceeds the artificial cost, it remains to proof an upper bound. In this scenario, an integer overflow could occur potentially leading to an incorrect EMSC value. We have neither observed such behavior in practice nor expect it to occur given the observed trends in potential values during execution.

Then the network setup begins including the scaling of the probabilities by the probability denominator LCM and the distances by the distance denominator LCM. The resulting network is entered into the Network Simplex Algorithms and the EMD value is calculated by dividing the optimal cost c_{opt} of the network by both LCMs.

The EMSC value of L_1 and L_2 is finally returned as $1 - \text{EMD}(L_1, L_2)$. Note that due to Ebi's internally used data structure of stochastic languages, the values of two approximate EMSC executions may vary by small amounts.

While several methods exist for solving the TP, we decided on the Network Simplex Algorithm because of its performance in practice. The EMSC implementation by Leemans et al. [21] uses an exterior point simplex method instead of a general-purpose LP solver as in the authors' previous work [26]. However, in large-scale instances, the implementation quickly becomes infeasible. Cost scaling methods tend to outperform the Network Simplex only in large, sparse networks [20], a condition not met in our dense EMD applications. Recently, an interior point for the TP was introduced [37]. The authors claim they could achieve performance comparable to the Network Simplex Algorithm. Whether this is the case for EMD networks, remains to investigate.

A popular and fast Network Simplex implementation is provided by the Egerváry

Research Group on Combinatorial Optimization within the Lemon graph library [10]. The Ebi implementation includes a Network Simplex implementation written in Rust which was translated from the original C code of the Lemon library. However, some changes were made and are listed in the following. The main distinguishing aspect is the added support for using floating-point types within the Network Simplex. This requires runtime checks of the used type and handles the floating-point case like the Python Opimal Transport library's implementation of the Network Simplex Algorithm [13]. The Lemon library offers multiple pivot rules. However, we only implemented the *Block Search Pivot Rule* because it perform best in practice [19]. Additionally, the process of setting up the input network was streamlined: The network's graph, its supplies, and its costs do not need to be set individually in the Rust implementation. Since this thesis only considers uncapacitated networks, the support for capacitated networks was not adopted by the Rust implementation.

5 Evaluation

This chapter presents an in-depth evaluation of the provided Ebi implementation based on real-life event data. Additionally, we executed the same tests on the following EMSC implementations: The *ProM* [36] implementation by Leemans et al. [21], the EMD implementation integrated into the library *PM4Py* [5], and an unpublished implementation *Python EMD* by Rocha and Brockhoff using the previously mentioned Python Optimal Transport library [13] and Numba to compute the distance matrix..

The real-life event logs considered for this analysis originate from the Business Process Intelligence Challenge (BPIC) hosted by the *Task Force on Process Mining*. The studied logs are publicly available and cover all sublogs of the BPIC 2018, dealing with the processing of applications for agricultural EU funding[1], the Domestic Declaration (DD) log from the BPIC 2020 about travel requests within a university[2], the Road Traffic Fines (RTF) event log[3], as well as the Sepsis (S) event log recorded in a hospital[4]. All event logs with their respective trace and variant counts are listed in Table 5.1.

Section 5.1 elaborates on how the tests were set up. The analysis of the findings is split into four sections for improved structure. In Section 5.2, we evaluate how the approximate mode of the Ebi implementation compares to the other implementations regarding total computation time. We examine their deviations from the exact EMSC values in Section 5.3. The difference in runtime between the exact and the approximate mode of the Ebi implementation is analyzed in Section 5.4. Lastly, Section 5.5 provides more detailed insights into the computation time distribution over the subtasks of the EMSC calculation.

5.1 Experimental Setup

When comparing the different implementations, it is important to fix the setting and eliminate parameters with inexplicable random influences. In this case, a *log-model* or *model-model* context would be problematic. This is first of all due to the implicit randomness of sampling and, secondly, the implementations do not use the same sampling technique. Hence, a *log-log* setting is used throughout this evaluation.

We applied the Inductive Miner [22] to each event log, yielding a block-structured process model. In the next step, we transformed the discovered process models into stochastic process models using the alignment-based weight estimation technique [7].

[1]BPIC 2018: `https://doi.org/10.4121/uuid:3301445f-95e8-4ff0-98a4-901f1f204972`
[2]BPIC 2018: `https://doi.org/10.4121/uuid:3f422315-ed9d-4882-891f-e180b5b4feb5`
[3]Road Traffic Fines: `https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5`
[4]Sepsis: `https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460`

Table 5.1: Real-Life Event Logs Selected for Evaluation

Event Log	Traces	Variants	Weight Estimation[7]
BPIC18 Control Summary (CS)	43,808	59	✓
BPIC18 Department Control Parcel (DCP)	29,297	349	✓
BPIC18 Geo Parcel Documents (GPD)	29,059	9,274	✓
BPIC18 Entitlement Application (EA)	15,260	2,553	✗
BPIC18 Inspection (I)	3,044	2,525	✓
BPIC18 Parcel Document (PD)	14,750	3,615	✓
BPIC18 Payment Application (PA)	43,809	3,832	✓
BPIC18 Reference Alignment (RA)	43,802	515	✓
BPIC20 Domestic Declaration (DD)	10,500	99	✓
Road Traffic Fines (RTF)	150,370	231	✓
Sepsis (S)	1,050	846	✓

Since the weight estimation did not terminate within an hour on the BPIC 2018 Entitlement Application log, we disregard it in the following. Next, we applied random walk sampling [28] to retrieve stochastic languages from the generated stochastic models. Specifically, for each model, eleven stochastic languages were sampled with sample sizes in the range of $2^4, \dots, 2^{14}$. Since other implementations cannot handle the stochastic language format, all stochastic languages were scaled by the sample size into regular event logs.

All tests where executed on an Intel Core i9-9880H processor running macOS 15.0.1 with 16 GB of RAM.

Each pair of ground truth and sampled event log is given as an input to the Ebi implementation both in the exact and the approximate mode, the ProM, the PM4Py, and the Python EMD implementation. We conduct five runs for each test for all implementations except the one integrated into PM4Py. In this case, we only performed two runs due to previously experienced high runtimes. Additionally, the test skipped to the next input log once a single run finished with a runtime above two hours. Here the goal was to retrieve at least a single data point per event log. For the Ebi, ProM, and Python EMD implementations, each run that did not return a result within eight hours was terminated.

We recorded the following information was recorded for each run: The duration of the distance computation, the duration of solving the internal optimization problem, the total duration, and the resulting conformance value. For both modes of the Ebi implementation, we also recorded the duration of the network setup. Additionally for the exact mode, we noted the internally used data type. This targets a more detailed analysis of the feasibility of the exact mode and the Integer-Scaled TP. Since PM4Py and Python EMD return an EMD value, the result was recorded as one minus the EMD value.

All measured total durations are listed in the Appendix in Tables 8.1 through 8.5, both

as absolute values (in seconds) and normalized such that the fastest implementation has a normalized time of 1, marked as (n). Note that PM4Py timed out during testing on the BPIC 2018 Payment Application (PA), Geo Parcel Document (GPD), Inspection (I) and Parcel Document (PD) logs, as well as the Sepsis log. Furthermore, the Ebi implementation in the exact mode timed out on the BPIC 2018 GPD log and the model language consisting of 2^{14} samples. Such missing data points are represented by blank entries in the tables in the Appendix.

5.2 Computation Time Comparison With Existing Methods

In this section, we compare the performance of the Ebi implementation in approximate mode to the ProM, PM4Py, and Python EMD implementations regarding the total computation time.

Based on experience, we know that the PM4Py implementation is notoriously slow. The ProM implementation begins the optimization with an educated guess as its initial basis. We expect that this gives it an edge over other implementations at least for smaller instances. However, experience has shown that the ProM implementation quickly becomes expensive when increasing the size of the instance. Furthermore, we expect similar performance from the Ebi and the Python EMD implementations as Python EMD performs its optimization using the Network Simplex implemented in the Python Optimal Transport library [13], which only differs slightly from the Lemon [10] implementation.

The recorded computation times of all implementations are displayed in Figure 5.1 for all tested input logs.

The performed test can be categorized into two groups: group A, where most samples corresponded to variants that were already previously sampled, and group B where almost every sample added a new variant to the model language. Group A features the tests on BPIC 2018 logs Control Summary (CS), Department Control Parcel (DCP), Reference Alignment (RA), as well as the RTF log. The tests on all other considered logs belong to group B. It is reasonable that the EMSC computation was significantly harder for group B throughout all implementations. While ProM shows some inconsistencies in recorded times of smaller instances, we can generally determine that ProM is faster than Ebi on these rather small instances, supporting our hypothesis. However, for all tests in group B, Ebi eventually becomes faster than ProM when increasing the number of samples. We can observe that PM4Py is indeed orders of magnitude slower than all other implementations. The normalized measured times show that in some cases it was 100,000 times slower than the fastest implementation. Even though not many data points could be collected for PM4Py, the trajectory of computation times on the Sepsis log shows that we cannot expect PM4Py to become more efficient compared to the competitors when further increasing the model language size.

Furthermore, we observed that, unexpectedly, Python EMD was faster than Ebi in all tests. Python EMD had the biggest edge over Ebi, i.e. 69.9 times faster, for the Road Traffic Fines log paired with the model language containing 2^{14} samples. Manual inspection revealed that Python EMD performed about a third of the iterations that Ebi

conducted. The smallest difference between the performance of Ebi and Python EMD is noticed for the BPIC 2018 I log and the model language containing 2^{13} samples. Here Python EMD even required 15% more iterations than Ebi, while being 20% faster. This observation is further discussed in Chapter 6.

5.3 Deviation Comparison With Existing Methods

The existence of an exact EMSC implementation, namely the Ebi implementation in exact mode, enables us to analyze how much the EMSC values calculated by the implementations deviate from their true value. This section elaborates on the same tests as Section 5.2.

All implementations use 64-bit floating point data types and hence deviations are most likely due to rounding errors introduced by arithmetic operations.

Figure 5.2 plots the deviations of the implementations' results on the test cases from the exact EMSC values retrieved by Ebi in the exact mode. Since the computation of the exact EMSC value timed out for the BPIC 2018 Inspection log paired with the stochastic language consisting of 2^{14} samples, no deviations could be calculated for this test case. There are multiple visible data points for Ebi while for all other implementations, only one data point is visible. This is because Ebi is the only implementation not producing results deterministically due to the internal stochastic language data structure. Some data points are not displayed for Ebi. In these cases, no deviation from the exact values has been observed.

The measured deviations are fairly consistent throughout all performed tests and no statistically relevant trends regarding the size of the instances could be detected. It shows that Ebi has the smallest deviation of $1 \cdot 10^{-12}$ on average, followed by Python EMD and ProM with average deviations of $3 \cdot 10^{-9}$ and $6 \cdot 10^{-9}$ respectively, and PM4Py in the last place with an average deviation of $1 \cdot 10^{-4}$. Note that the average deviation calculated for PM4Py includes a bias toward smaller instances as bigger instances were too expensive to compute.

5.4 Computation Time Comparison of Exact With Approximate Mode

The central goal of this thesis is to present an exact EMSC algorithm which ideally is not much slower than its approximate counterpart. This section evaluates how runtimes of the Ebi implementation in the exact mode compare to its approximate mode.

We expect that the Integer-Scaled TP instance creation, presented in Section 4.2, introduces an overhead before the optimization, which should slightly increase computation times across the board. Moreover, we expect that exact runs where finite data types (i.e. i64 and i128) are sufficient are much closer to their respective approximate runs than exact runs that require arbitrary-precision integers.

Figure 5.3 displays the average computation times of the Ebi EMSC implementation

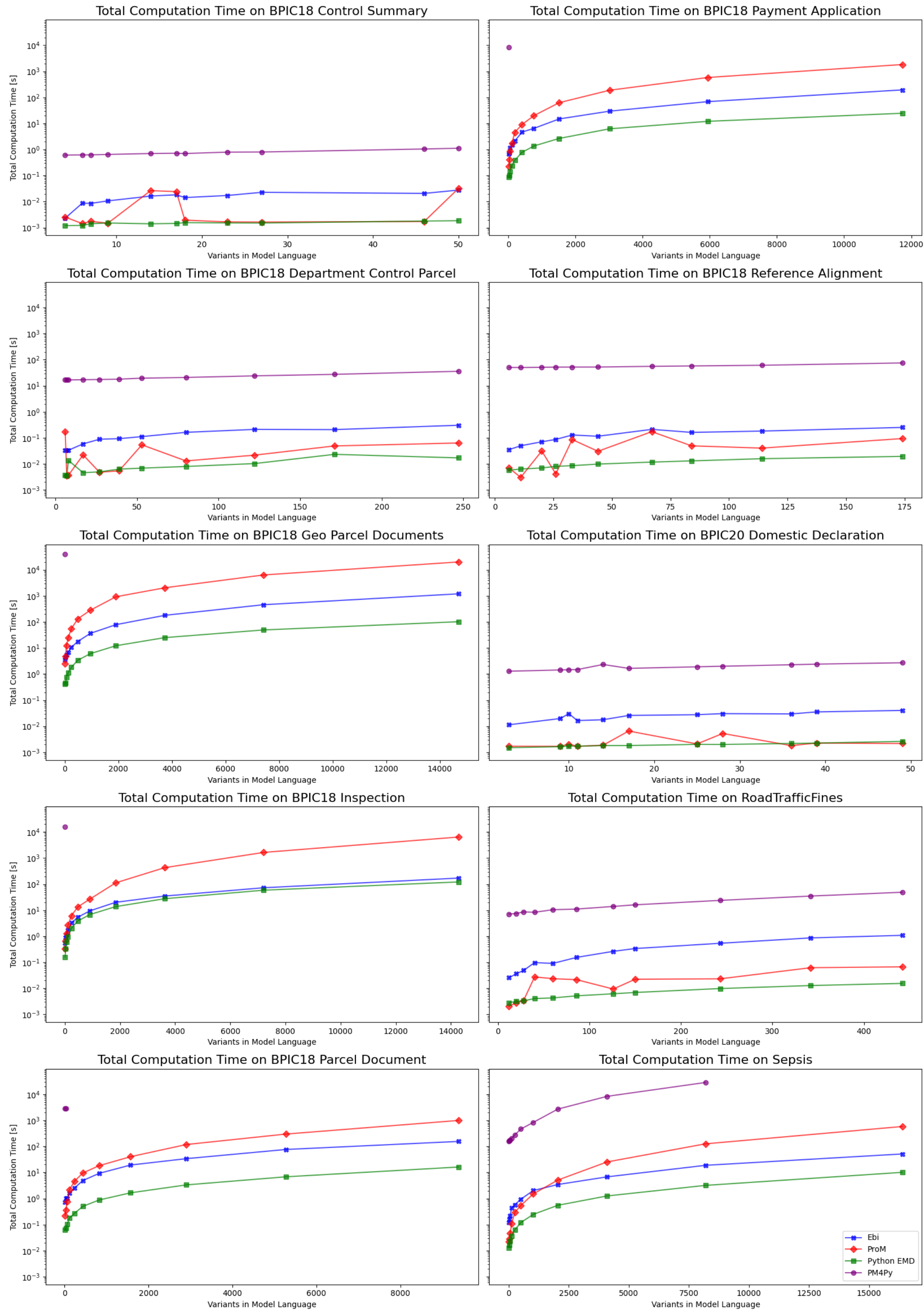

Figure 5.1: Runtime Comparison of Ebi With Other Implementations

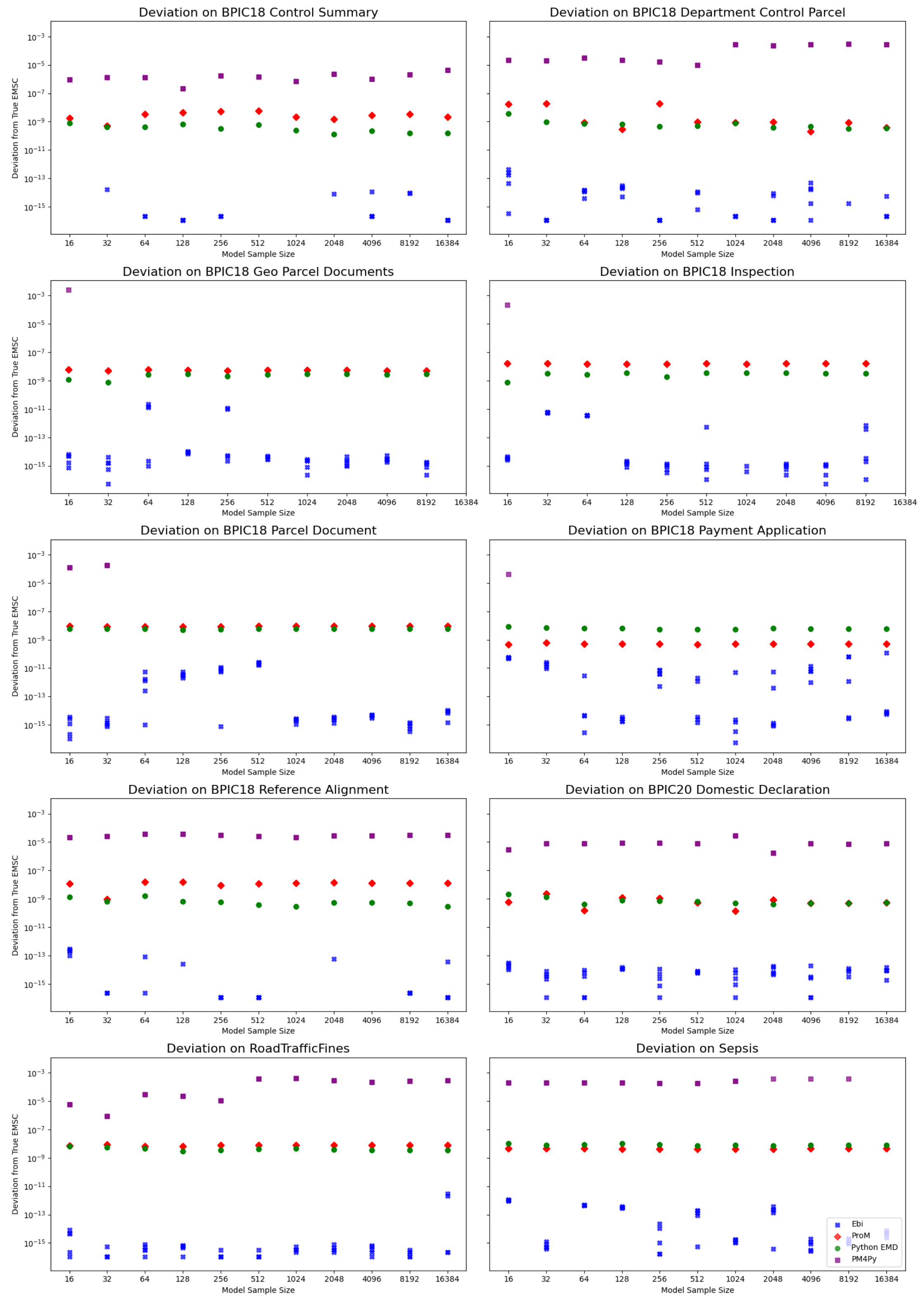

Figure 5.2: Deviation from Exact EMSC Value

in approximate mode (blue) and in the exact mode depending on the used datatype (green for i64, orange for i128, red for BigInt). Additionally, all measured times are listed in the Appendix in Tables 8.1 through 8.5. All exact runs on the BPIC 2018 CS and DCP logs, the BPIC 2020 DD log, and the RTF log internally used 64-bit integers. 128-bit integers were used by all runs on the Sepsis log, the BPIC 2018 PD and RA logs, and runs on the BPIC 2018 PA log up to the model language consisting of 2^8 samples. From the model language consisting of 2^9 samples, BigInts were used instead. BigInts were also used throughout all runs on the BPIC 2018 GPD and I logs.

The used data types indeed explain the measured runtimes: A significant difference in runtime between the two modes is observed only for the BPIC 2018 PA, GPD and I logs. Note that on the BPIC 2018 PA log, runtime suddenly increases when including 2^9 samples into the model language, which perfectly aligns with the detected change of the used data type.

On average, the approximate mode turns out to be 10% faster than the exact mode when finite-size integers are used. In some instances, the exact mode was faster than the approximate mode. Conversely, if the arbitrary-precision integer type BigInt is used, the approximate mode is on average 4.1 times faster than the exact mode. This confirms our initial hypothesis.

5.5 Internal Runtime Distribution per Task

In Section 5.4, we found that using the exact mode of the Ebi implementation was slightly slower than the approximate mode when using finite integers was sufficient. If this is not the case, the approximate mode has a bigger edge regarding computation speed. This section gives a more differentiated analysis of the time distribution over the subtasks of the EMSC computations performed by the implementations on two selected event logs. We decided to include the BPIC 2018 PA log because PA is of particular interest due to Ebi's exact mode switching to arbitrary-precision integers when increasing the sample size. Additionally, the BPIC 2018 PD log is included since it turned out to be the most challenging tested log on which Ebi's exact mode used finite integers for all model languages.

For the Ebi, ProM, Python EMD, and PM4Py implementations the distance computation, optimization, and total computation time are measured. Furthermore, for Ebi, the time required to set up the network was measured to better compare the exact and approximate modes. Subtracting the measured times of the subtasks from the total computation time leaves the time spent on other computations such as retrieving the resulting EMD from the network in Ebi's case. In the approximate mode, Ebi might fail to set up to initialize the network Simplex on the first try due to rounding errors. This is also excluded from the tracked subtasks.

It shows that PM4Py computes distances very inefficiently. On average, 97% of the computation time in all tests where PM4Py terminated before timeout was spent on distance computation. Thus, we will not further analyze PM4Py here.

The time spent on the above-mentioned subtasks is represented as a share of the total

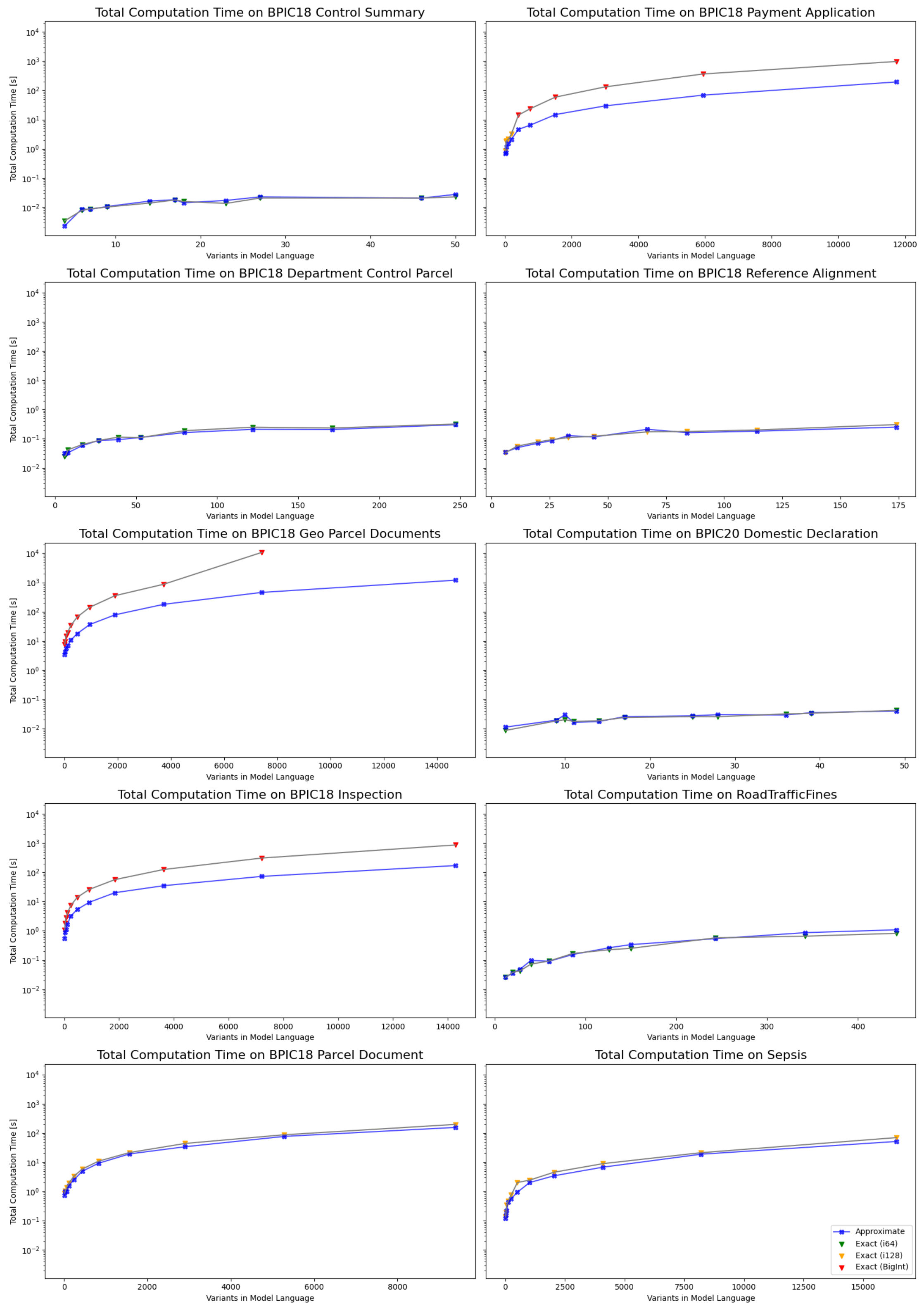

Figure 5.3: Runtime Comparison of Exact and Approximate Arithmetic

Table 5.2: Time Spent on Subtasks (in Seconds) for BPIC 2018 PA and Model Language of 2^8 Samples

Implementation	Distance	Setup	Other	Optimization	Total
Ebi Approximate	0.199	0.179	0.048	1.696	2.123
Ebi Exact (i128)	0.242	1.004	0.094	1.843	3.183
Python EMD	0.187		0.049	0.151	0.387
ProM	0.204		0.001	4.366	4.572

computation time for both Ebi modes in Figure 5.4, for Python EMD in Figure 5.5, and ProM in Figure 5.6. The sample size of the model language to which a log is compared is annotated to the left of each bar, while the total computation time is shown (in seconds) to the right. Additionally, we list the absolute recorded times of the subtasks on BPIC 2018 PA and the model language consisting of 2^8 samples in Table 5.2 as a point of reference to the relative time shares in Figures 5.4 to 5.6.

Comparing the shares of time spent on the network setup in the exact mode of Ebi with the respective shares in the approximate mode confirms the hypothesis that the creation of the Integer-Scaled instance introduces some overhead. On the BPIC 2018 PA log, the optimization time share drastically increases for the tests on model languages consisting of 2^9 samples compared to tests on languages generated by fewer samples. Together with the fact that also the total time increases, as elaborated in Section 5.4, this accentuates that using finite integers reduces the time needed for the optimization.

For both logs considered in this part of the analysis and for both of Ebi's modes, we note that increasing the sample size causally increases the optimization time share and decreases the network setup time share. Interestingly, increasing the sample size at first also increases the distance computation time share. However eventually, the distance computation time share decreases again. This is an indicator that the Network Simplex optimization is the first step in Ebi's EMSC computation to become infeasible.

While Python EMD spends most time outside of the distance computation and optimization for small instances, this drastically changes when increasing the sample size. Overall, the optimization appears to be much more efficient than Ebi's approximate mode. Again, the distance computation time share behaves similarly to Ebi, supporting the conception that the Network Simplex optimization scales worse than calculating the distances.

Comparing the total EMSC computation time of ProM to Ebi's approximate mode shows that ProM is overtaken on both logs when increasing the instance size. Manual inspection revealed that ProM computes distances faster than Ebi for rather small instances. For larger instances, this difference diminishes. Especially for the BPIC 2018 PD log the ProM optimization time share appears to increase rapidly as the instance size grows. Furthermore, the distance computation time share decreases from the beginning. This indicates that EPSA, the optimization technique used within ProM, scales significantly worse than the Network Simplex algorithm.

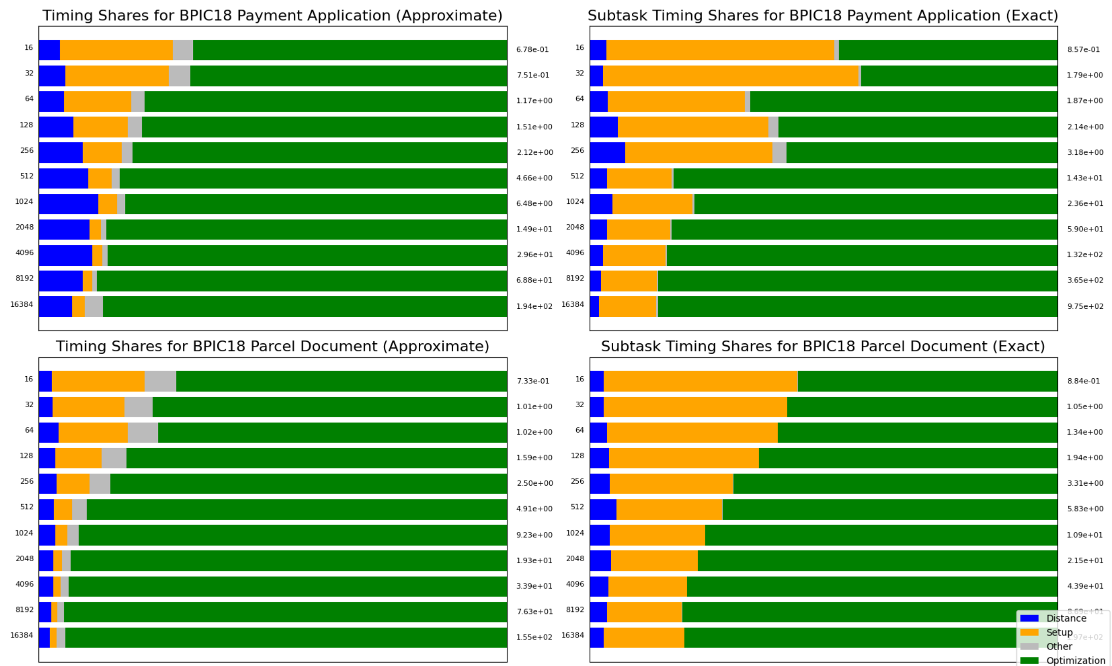

Figure 5.4: Ebi (Approximate/Exact) - Internal Runtime Distribution per Task

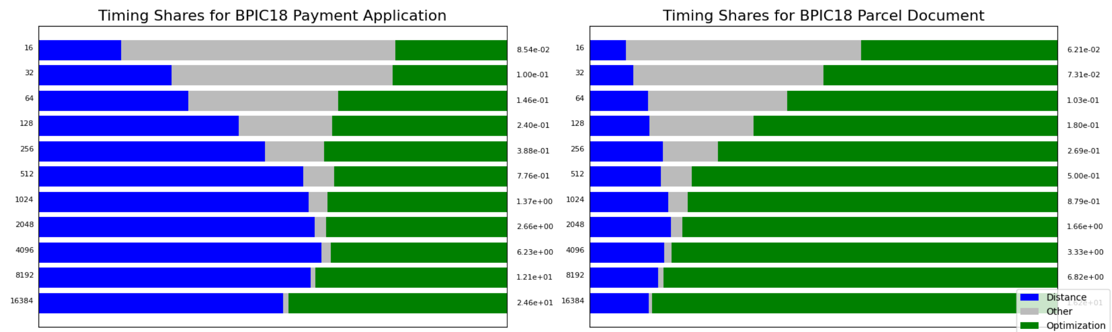

Figure 5.5: Python EMD - Internal Runtime Distribution per Task

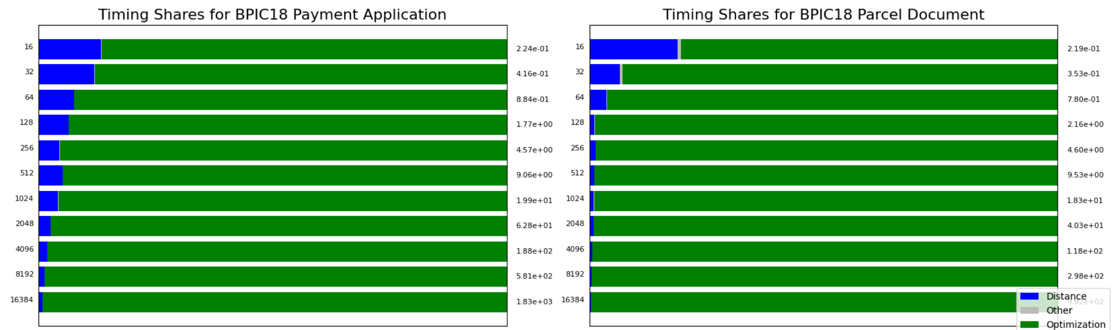

Figure 5.6: ProM - Internal Runtime Distribution per Task

6 Discussion

The main contribution of this thesis is a novel approach to efficiently calculate the exact Earth Movers' Stochastic Conformance value of two stochastic languages. To the author's knowledge, the provided implementation is the first to offer exact EMSC computation capabilities. The key idea is formalized as the Integer-Scaled TP, eliminating all fractions by scaling them by the LCMs of the denominators of the trace probabilities and the distances. This procedure introduces some overhead caused by the LCM calculations and the scaling of all input fractions to integers. However, performing the EMD optimization directly on the arbitrary-precision fraction data type would cause significant overhead for all arithmetic operations. Our evaluation showed that the share of the total exact EMSC computation spent in the optimization grows when the instance size is increased. We suspect that the overhead introduced by the Integer-Scaled TP is small compared to the expected overhead if the optimization instead would be performed on the fractions.

Additionally, we showed that using finite integers is sufficient in most real-world scenarios, which drastically improves optimization runtime because optimization variables no longer need to be stored on the heap. The provided implementation, integrated into the Ebi framework, uses 64-bit or 128-bit integers where applicable. On average, an approximate EMSC computation was only 10% faster than an exact computation if finite integers sufficed. Otherwise, an approximate EMSC computation was 4.1 times faster on average. Even though Rust does not provide larger integer data types such as 256-bit or even 512-bit, further investigation is required to determine whether larger data types offered by community crates could improve performance for those cases, where 128-bit are insufficient. Presumably, performance would be highly architecture-specific but an exact EMSC calculation could be rendered feasible for bigger problem instances.

As long as stochastic languages are derived by conversion from an event log or by random walk sampling a model, the size of the event log and the number of samples constitute an upper bound to the supply and demand values of the EMD network. For any reasonable input, this will not be out of the range of finite integers. In all tests, the only reason why finite integers were insufficient was the LCM of the distance denominators, which resulted in arc costs out of the scope of 128-bit integers. If the stochastic languages include traces of various lengths, the set of primes, generated by the factorization of all lengths, might become large as well. In this case, the distance LCM can grow very quickly. Hence, models with smaller loop probabilities are less likely to require the use of arbitrary-precision integers. A different approach to avoid arbitrary-precision integers is changing the distance function to an un-normalized one which renders scaling the distances obsolete.

On the one hand, we have shown that the provided implementation in its approximate

mode scales better than the ProM implementation by Leemans et al. [21], on the other hand, the unpublished Python EMD implementation by Rocha and Brockhoff, which also uses the Network Simplex Algorithm, was consistently significantly faster. The root cause of this performance gap remains open for investigation.

This may be connected to the higher experienced deviations from the true EMSC value for the Python EMD implementation. Indeed, the provided approximate implementation is more precise than all other tested EMSC implementations: While its deviation was $1 \cdot 10^{-12}$ on average over all tests, Python EMD's deviation increased to $3 \cdot 10^{-9}$, the deviation of the ProM implementation was $6 \cdot 10^{-9}$, and PMpy was the least precise with an average deviation of $1 \cdot 10^{-4}$. For most applications, it presumably will suffice to calculate the approximate EMSC value using the provided implementation due to its low deviation in practice. However, the exact implementation does not showcase drastically higher runtimes than its approximate counterpart and is particularly interesting for benchmarking purposes.

Currently, the Network Simplex Algorithm is initialized with an artificial basis. While this is a simple working approach, it is very general. Charnes et al. [9] suggested the North-West Corner Method, followed by many more approaches from operations research. Future research could build upon existing techniques and refine the initial basis further by incorporating heuristics specifically designed for the EMD network structure.

7 Conclusion

The introduction of stochastic process models opened the field of stochastic conformance checking, which, unlike conventional conformance checking, considers trace probabilities on both the log and the model side. This thesis proposes the Integer-Scaled TP to efficiently calculate an *exact* EMSC value of two finite exact stochastic languages. The probabilities of the traces of the stochastic languages and the distances between the traces of the two languages are scaled individually to integer values when setting up the EMD network.

This thesis provides an implementation in Rust with a mode for an exact EMSC computation based on the Integer-Scaled TP as well as an approximate mode skipping the scaling step. The Network Simplex Algorithm is chosen as the optimization technique for calculating the EMD of the two languages due to its great performance in practice for solving the Transshipment Problem. Depending on the scalars of the trace probabilities and the distances, it is possible to determine whether finite integer types are sufficient to perform the optimization without integer overflows, targeting faster computation. The implementation is integrated into the stochastic process mining framework Ebi where both the exact and the approximate modes are exposed to the user from the command line.

Additionally, we presented an evaluation of the implementation's performance on real-life event data. The evaluation demonstrated that performing the optimization on finite integer types indeed connotes a significant performance improvement compared to optimizing on arbitrary-precision integer types instead, which was only required in a few of the test instances. Moreover, it finds that the approximate mode is faster than the EMSC implementation in PM4Py, and becomes more efficient than the ProM implementation for larger instances whereas the Python EMD implementation was consistently quicker. Our approximate implementation turned out to be the most precise by far regarding experienced deviations from the true EMSC value.

Future Work. Possible directions for future research include exploring other optimization techniques such as interior point methods and the development of a more problem-specific initial feasible basis for the Network Simplex. Such an improved initial feasible basis could be constructed based on heuristics and drastically reduce the number of required iterations. It might be beneficial to analyze the structure of optimal spanning tree solutions when deriving such heuristics. It remains to formally prove an upper bound of the potentials in the EMD networks. Also, the root cause of the performance gap between the provided implementation and the Python EMD implementation should be investigated. Additionally, further potential optimizations, such as the inclusion of larger

finite integer types before defaulting to arbitrary-precision integers, are worth exploring. From a user standpoint, it would be convenient to have an approximate EMSC mode at hand where the maximal deviation ϵ could be specified as an acceptable error and the optimization would terminate once an ϵ-optimal solution is reached.

8 Appendix

8.1 Evaluation Runtime Data

The measured total computation times of all tested implementations are given in the following tables.

Table 8.1: Running Time (Seconds and Normalized) - 1

			Ebi		**Other Implementations**		
Event Log	Samples	Variants	Exact	Approx	ProM	PM4PY	Python EMD
	2^{4}	4	0.003	0.002	0.003	0.614	0.001
	2^{5}	6	0.008	0.009	0.001	0.623	0.001
	2^{6}	7	0.009	0.009	0.002	0.626	0.001
	2^{7}	9	0.010	0.011	0.001	0.646	0.002
	2^{8}	14	0.014	0.016	0.026	0.703	0.001
BPIC18 CS	2^{9}	18	0.016	0.014	0.002	0.702	0.002
	2^{10}	17	0.018	0.018	0.024	0.720	0.001
	2^{11}	23	0.014	0.017	0.002	0.794	0.002
	2^{12}	27	0.021	0.023	0.002	0.803	0.002
	2^{13}	46	0.021	0.021	0.002	1.046	0.002
	2^{14}	50	0.023	0.028	0.032	1.122	0.002
	2^{4}	4	2.9	1.9	2.1	511.5	1.0
	2^{5}	6	6.4	7.2	1.2	512.4	1.0
	2^{6}	7	6.2	6.0	1.2	440.2	1.0
	2^{7}	9	6.9	7.2	1.0	433.3	1.0
	2^{8}	14	10.0	11.6	18.8	499.1	1.0
BPIC18 CS (n)	2^{9}	18	10.1	9.1	1.2	446.9	1.0
	2^{10}	17	12.3	12.5	16.6	493.9	1.0
	2^{11}	23	9.0	11.3	1.1	521.2	1.0
	2^{12}	27	13.7	15.1	1.1	531.4	1.0
	2^{13}	46	11.9	11.9	1.0	603.8	1.0
	2^{14}	50	12.3	15.0	17.3	605.8	1.0
	2^{4}	16	0.857	0.678	0.224	8359.445	0.085
	2^{5}	27	1.794	0.751	0.416		0.100
	2^{6}	49	1.874	1.168	0.884		0.146
	2^{7}	103	2.137	1.506	1.769		0.240
	2^{8}	190	3.183	2.124	4.573		0.388
BPIC18 PA	2^{9}	403	14.332	4.656	9.058		0.776
	2^{10}	758	23.633	6.478	19.911		1.366
	2^{11}	1512	58.979	14.909	62.818		2.662
	2^{12}	3011	132.325	29.642	187.899		6.228
	2^{13}	5942	364.626	68.814	581.418		12.083
	2^{14}	11733	975.116	194.206	1826.236		24.640
	2^{4}	16	10.0	7.9	2.6	97 856.9	1.0
	2^{5}	27	17.9	7.5	4.2		1.0
	2^{6}	49	12.8	8.0	6.1		1.0
	2^{7}	103	8.9	6.3	7.4		1.0
	2^{8}	190	8.2	5.5	11.8		1.0
BPIC18 PA (n)	2^{9}	403	18.5	6.0	11.7		1.0
	2^{10}	758	17.3	4.7	14.6		1.0
	2^{11}	1512	22.2	5.6	23.6		1.0
	2^{12}	3011	21.2	4.8	30.2		1.0
	2^{13}	5942	30.2	5.7	48.1		1.0
	2^{14}	11733	39.6	7.9	74.1		1.0

Table 8.2: Running Time (Seconds and Normalized) - 2

			Ebi		**Other Implementations**		
Event Log	Samples	Variants	Exact	Approx	ProM	PM4PY	Python EMD
BPIC18 DCP	2^4	6	0.025	0.033	0.176	16.747	0.004
	2^5	7	0.029	0.033	0.004	16.645	0.004
	2^6	8	0.042	0.033	0.004	16.728	0.014
	2^7	17	0.064	0.059	0.022	16.960	0.005
	2^8	27	0.087	0.088	0.005	17.357	0.005
	2^9	39	0.114	0.093	0.005	17.830	0.006
	2^{10}	53	0.110	0.111	0.054	19.487	0.007
	2^{11}	80	0.189	0.163	0.013	20.745	0.008
	2^{12}	122	0.249	0.211	0.022	23.883	0.010
	2^{13}	171	0.234	0.207	0.049	27.301	0.023
	2^{14}	247	0.318	0.303	0.063	35.805	0.017
BPIC18 DCP (n)	2^4	6	6.6	8.8	47.3	4493.5	1.0
	2^5	7	8.3	9.3	1.0	4747.5	1.0
	2^6	8	11.4	9.1	1.0	4545.6	3.7
	2^7	17	13.9	12.8	4.8	3704.8	1.0
	2^8	27	18.2	18.4	1.0	3623.1	1.0
	2^9	39	21.0	17.2	1.0	3302.2	1.2
	2^{10}	53	16.0	16.2	7.8	2838.1	1.0
	2^{11}	80	23.8	20.4	1.6	2609.8	1.0
	2^{12}	122	24.3	20.5	2.1	2326.3	1.0
	2^{13}	171	10.1	8.9	2.1	1175.8	1.0
	2^{14}	247	18.7	17.8	3.7	2105.9	1.0
BPIC18 RA	2^4	6	0.032	0.034	0.011	50.069	0.006
	2^5	6	0.035	0.037	0.003	50.683	0.006
	2^6	11	0.056	0.050	0.003	49.824	0.006
	2^7	20	0.078	0.070	0.031	50.939	0.007
	2^8	26	0.094	0.087	0.004	51.683	0.008
	2^9	33	0.110	0.128	0.086	52.024	0.009
	2^{10}	44	0.122	0.115	0.030	52.111	0.010
	2^{11}	67	0.172	0.211	0.172	55.407	0.012
	2^{12}	84	0.177	0.162	0.049	57.317	0.013
	2^{13}	114	0.200	0.182	0.040	61.010	0.016
	2^{14}	174	0.306	0.251	0.094	74.720	0.019
BPIC18 RA (n)	2^4	6	5.6	5.9	1.9	8804.2	1.0
	2^5	6	11.9	12.3	1.0	17 000.4	1.9
	2^6	11	18.4	16.3	1.0	16 416.2	2.1
	2^7	20	11.1	10.1	4.5	7307.8	1.0
	2^8	26	22.7	21.0	1.0	12 527.5	1.9
	2^9	33	12.9	15.0	10.0	6084.5	1.0
	2^{10}	44	12.4	11.7	3.1	5306.7	1.0
	2^{11}	67	14.7	18.0	14.7	4729.3	1.0
	2^{12}	84	13.6	12.4	3.8	4399.0	1.0
	2^{13}	114	12.6	11.5	2.5	3859.0	1.0
	2^{14}	174	15.8	12.9	4.8	3852.8	1.0

Table 8.3: Running Time (Seconds and Normalized) - 3

			Ebi		**Other Implementations**		
Event Log	Samples	Variants	Exact	Approx	ProM	PM4PY	Python EMD
	2^4	16	7.692	3.492	2.539	40 022.216	0.407
	2^5	32	9.441	4.350	4.786		0.452
	2^6	63	14.420	5.482	12.042		0.751
	2^7	126	19.227	6.934	24.200		1.111
	2^8	247	33.927	10.614	54.656		1.809
BPIC18 GPD	2^9	487	65.729	17.796	131.138		3.391
	2^{10}	960	142.292	36.481	282.865		6.066
	2^{11}	1909	353.598	78.293	922.584		12.178
	2^{12}	3738	869.331	178.945	2044.227		24.975
	2^{13}	7421	10 692.608	456.919	6339.901		49.174
	2^{14}	14692		1196.145	19 976.262		101.642
	2^4	16	18.9	8.6	6.2	98 375.7	1.0
	2^5	32	20.9	9.6	10.6		1.0
	2^6	63	19.2	7.3	16.0		1.0
	2^7	126	17.3	6.2	21.8		1.0
	2^8	247	18.8	5.9	30.2		1.0
BPIC18 GPD (n)	2^9	487	19.4	5.2	38.7		1.0
	2^{10}	960	23.5	6.0	46.6		1.0
	2^{11}	1909	29.0	6.4	75.8		1.0
	2^{12}	3738	34.8	7.2	81.8		1.0
	2^{13}	7421	217.4	9.3	128.9		1.0
	2^{14}	14692		11.8	196.5		1.0
	2^4	3	0.009	0.011	0.002	1.282	0.001
	2^5	10	0.020	0.030	0.002	1.466	0.002
	2^6	11	0.018	0.016	0.002	1.478	0.002
	2^7	9	0.018	0.020	0.002	1.432	0.002
	2^8	14	0.019	0.018	0.002	2.325	0.002
BPIC20 DD	2^9	17	0.024	0.026	0.006	1.648	0.002
	2^{10}	25	0.026	0.028	0.002	1.891	0.002
	2^{11}	28	0.026	0.030	0.005	1.992	0.002
	2^{12}	36	0.032	0.029	0.002	2.266	0.002
	2^{13}	39	0.034	0.035	0.002	2.400	0.002
	2^{14}	49	0.042	0.040	0.002	2.706	0.003
	2^4	3	5.8	7.6	1.1	860.2	1.0
	2^5	10	11.7	17.4	1.2	846.8	1.0
	2^6	11	10.6	9.7	1.0	881.2	1.0
	2^7	9	11.3	12.0	1.0	876.2	1.0
	2^8	14	10.3	9.8	1.0	1295.1	1.0
BPIC20 DD (n)	2^9	17	13.4	14.3	3.6	917.0	1.0
	2^{10}	25	12.9	13.8	1.0	946.3	1.0
	2^{11}	28	13.0	15.2	2.6	1005.7	1.0
	2^{12}	36	17.8	16.4	1.0	1265.4	1.2
	2^{13}	39	15.1	15.7	1.0	1073.3	1.0
	2^{14}	49	19.6	18.6	1.0	1255.1	1.2

Table 8.4: Running Time (Seconds and Normalized) - 4

			Ebi		Other Implementations		
Event Log	Samples	Variants	Exact	Approx	ProM	PM4PY	Python EMD
BPIC18 I	2^4	14	1.074	0.556	0.336	15 886.849	0.159
	2^5	32	1.797	0.894	0.666		0.327
	2^6	63	2.770	1.126	1.287		0.594
	2^7	116	4.125	1.696	2.657		0.938
	2^8	240	7.414	3.229	6.129		2.039
	2^9	471	13.929	5.378	13.392		3.856
	2^{10}	909	25.977	9.451	26.652		6.803
	2^{11}	1845	56.363	20.127	112.287		13.741
	2^{12}	3617	124.762	34.896	428.721		27.587
	2^{13}	7209	308.634	72.804	1652.081		58.687
	2^{14}	14275	864.009	170.817	6435.759		121.680
BPIC18 I (n)	2^4	14	6.8	3.5	2.1	100 145.6	1.0
	2^5	32	5.5	2.7	2.0		1.0
	2^6	63	4.7	1.9	2.2		1.0
	2^7	116	4.4	1.8	2.8		1.0
	2^8	240	3.6	1.6	3.0		1.0
	2^9	471	3.6	1.4	3.5		1.0
	2^{10}	909	3.8	1.4	3.9		1.0
	2^{11}	1845	4.1	1.5	8.2		1.0
	2^{12}	3617	4.5	1.3	15.5		1.0
	2^{13}	7209	5.3	1.2	28.2		1.0
	2^{14}	14275	7.1	1.4	52.9		1.0
RTF	2^4	12	0.026	0.026	0.002	7.141	0.003
	2^5	20	0.038	0.036	0.003	7.476	0.003
	2^6	28	0.043	0.049	0.003	8.634	0.003
	2^7	40	0.073	0.098	0.028	8.324	0.004
	2^8	60	0.093	0.091	0.024	10.456	0.004
	2^9	86	0.168	0.155	0.022	11.078	0.005
	2^{10}	126	0.226	0.264	0.009	13.948	0.006
	2^{11}	150	0.252	0.338	0.022	16.123	0.007
	2^{12}	243	0.575	0.539	0.023	23.860	0.010
	2^{13}	342	0.656	0.867	0.062	34.966	0.013
	2^{14}	442	0.823	1.085	0.067	48.927	0.016
RTF (n)	2^4	12	12.4	12.5	1.0	3472.5	1.4
	2^5	20	13.8	13.2	1.0	2708.2	1.1
	2^6	28	12.9	14.6	1.0	2584.3	1.0
	2^7	40	17.9	24.0	6.8	2046.4	1.0
	2^8	60	21.5	21.1	5.5	2427.9	1.0
	2^9	86	32.4	29.9	4.2	2139.3	1.0
	2^{10}	126	36.3	42.5	1.5	2244.6	1.0
	2^{11}	150	36.2	48.6	3.2	2315.0	1.0
	2^{12}	243	58.3	54.6	2.4	2417.7	1.0
	2^{13}	342	51.0	67.4	4.8	2718.6	1.0
	2^{14}	442	53.0	69.9	4.3	3151.9	1.0

Table 8.5: Running Time (Seconds and Normalized) - 5

			Ebi		**Other Implementations**		
Event Log	Samples	Variants	Exact	Approx	ProM	PM4PY	Python EMD
BPIC18 PD	2^{4}	16	0.884	0.733	0.219	2828.304	0.062
	2^{5}	31	1.055	1.013	0.353	2821.804	0.073
	2^{6}	63	1.344	1.020	0.780		0.103
	2^{7}	119	1.937	1.585	2.164		0.180
	2^{8}	236	3.314	2.503	4.601		0.269
	2^{9}	437	5.825	4.908	9.530		0.500
	2^{10}	826	10.871	9.232	18.275		0.879
	2^{11}	1574	21.495	19.283	40.299		1.655
	2^{12}	2905	43.906	33.927	118.298		3.335
	2^{13}	5275	86.942	76.268	297.667		6.818
	2^{14}	9387	196.820	155.331	992.179		16.167
BPIC18 PD (n)	2^{4}	16	14.2	11.8	3.5	45 523.5	1.0
	2^{5}	31	14.4	13.9	4.8	38 601.8	1.0
	2^{6}	63	13.1	9.9	7.6		1.0
	2^{7}	119	10.7	8.8	12.0		1.0
	2^{8}	236	12.3	9.3	17.1		1.0
	2^{9}	437	11.6	9.8	19.1		1.0
	2^{10}	826	12.4	10.5	20.8		1.0
	2^{11}	1574	13.0	11.7	24.3		1.0
	2^{12}	2905	13.2	10.2	35.5		1.0
	2^{13}	5275	12.8	11.2	43.7		1.0
	2^{14}	9387	12.2	9.6	61.4		1.0
S	2^{4}	16	0.136	0.119	0.022	157.259	0.013
	2^{5}	32	0.192	0.160	0.028	163.087	0.016
	2^{6}	64	0.346	0.223	0.046	178.392	0.023
	2^{7}	128	0.473	0.437	0.109	208.396	0.036
	2^{8}	256	0.764	0.575	0.290	276.225	0.062
	2^{9}	512	2.029	0.945	0.542	464.895	0.120
	2^{10}	1024	2.443	2.018	1.500	831.983	0.245
	2^{11}	2048	4.556	3.446	5.106	2697.605	0.549
	2^{12}	4096	8.946	6.782	25.326	8340.254	1.254
	2^{13}	8192	21.171	18.787	125.517	28 422.818	3.191
	2^{14}	16382	69.504	51.314	581.384		10.118
S (n)	2^{4}	16	10.5	9.2	1.7	12 182.5	1.0
	2^{5}	32	12.0	10.0	1.7	10 174.1	1.0
	2^{6}	64	14.8	9.5	2.0	7631.2	1.0
	2^{7}	128	13.1	12.1	3.0	5776.7	1.0
	2^{8}	256	12.2	9.2	4.6	4424.9	1.0
	2^{9}	512	16.9	7.9	4.5	3870.1	1.0
	2^{10}	1024	10.0	8.2	6.1	3390.3	1.0
	2^{11}	2048	8.3	6.3	9.3	4910.9	1.0
	2^{12}	4096	7.1	5.4	20.2	6649.6	1.0
	2^{13}	8192	6.6	5.9	39.3	8906.3	1.0
	2^{14}	16382	6.9	5.1	57.5		1.0

Bibliography

[1] W. M. van der Aalst. “Foundations of Process Discovery.” In: *Process Mining Handbook*. Springer, 2022, pp. 37–75.

[2] A. Adriansyah, B. F. van Dongen, and W. M. van der Aalst. “Conformance Checking Using Cost-Based Fitness Analysis.” In: *2011 ieee 15th international enterprise distributed object computing conference*. IEEE. 2011, pp. 55–64.

[3] H. Alkhammash, A. Polyvyanyy, and A. Moffat. “Stochastic Directly-Follows Process Discovery Using Grammatical Inference.” In: *International Conference on Advanced Information Systems Engineering*. Springer. 2024, pp. 87–103.

[4] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. John Wiley & Sons, 2011.

[5] A. Berti, S. van Zelst, and D. Schuster. “PM4Py: A Process Mining Library for Python.” In: *Software Impacts* 17 (2023), p. 100556. DOI: `https://doi.org/10.1016/j.simpa.2023.100556`.

[6] E. Bogdanov, I. Cohen, and A. Gal. “Conformance Checking over Stochastically Known Logs.” In: *International Conference on Business Process Management*. Springer. 2022, pp. 105–119.

[7] A. Burke, S. J. Leemans, and M. T. Wynn. “Stochastic Process Discovery by Weight Estimation.” In: *International Conference on Process Mining*. Springer. 2020, pp. 260–272.

[8] J. Carmona, B. van Dongen, A. Solti, and M. Weidlich. *Conformance Checking*. Springer, 2018.

[9] A. Charnes and W. W. Cooper. “The Stepping Stone Method of Explaining Linear Programming Calculations in Transportation Problems.” In: *Management science* 1.1 (1954), pp. 49–69.

[10] E. R. G. on Combinatorial Optimization. *LEMON 1.3.1 – Library for Efficient Modeling and Optimization in Networks*. `https://lemon.cs.elte.hu/trac/lemon`. Accessed: 21 February 2025. 2014.

[11] G. B. Dantzig. *Linear Programming and Extensions*. A Rand Corporation Research Study. Princeton, N.J: RAND Corporation, 1963.

[12] M. Dumas, W. M. Van der Aalst, and A. H. Ter Hofstede. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, 2005.

[13] R. Flamary, C. Vincent-Cuaz, N. Courty, A. Gramfort, O. Kachaiev, H. Quang Tran, L. David, C. Bonet, N. Cassereau, T. Gnassounou, E. Tanguy, J. Delon, A. Collas, S. Mazelet, L. Chapel, T. Kerdoncuff, X. Yu, M. Feickert, P. Krzakala, T. Liu, and E. Fernandes Montesuma. *POT Python Optimal Transport (Version 0.9.5)*. 2024. URL: `https://github.com/PythonOT/POT`.

[14] A. Gal. "Everything There Is to Know About Stochastically Known Logs." In: *2023 5th International Conference on Process Mining (ICPM)*. IEEE. 2023, pp. xvii–xxiii.

[15] A. V. Goldberg. "The Partial Augment–Relabel Algorithm for the Maximum Flow Problem." In: *European Symposium on Algorithms*. Springer. 2008, pp. 466–477.

[16] M. D. Grigoriadis. "An Efficient Implementation of the Network Simplex Method." In: *Netflow at Pisa* (1986), pp. 83–111.

[17] F. L. Hitchcock. "The Distribution of a Product from Several Sources to Numerous Localities." In: *Journal of mathematics and physics* 20.1-4 (1941), pp. 224–230.

[18] L. V. Kantorovich. "Mathematical Methods of Organizing and Planning Production." In: *Management science* 6.4 (1960), pp. 366–422.

[19] D. J. Kelly and G. M. ONeill. "The Minimum Cost Flow Problem and the Network Simplex Solution Method." PhD thesis. Citeseer, 1991.

[20] P. Kovács. "Minimum-Cost Flow Algorithms: An Experimental Evaluation." In: *Optimization Methods and Software* 30.1 (2015), pp. 94–127.

[21] S. J. Leemans, W. M. van der Aalst, T. Brockhoff, and A. Polyvyanyy. "Stochastic Process Mining: Earth Movers' Stochastic Conformance." In: *Information Systems* 102 (2021), p. 101724.

[22] S. J. Leemans, D. Fahland, and W. M. Van Der Aalst. "Discovering Block-Structured Process Models from Event Logs - A Constructive Approach." In: *International conference on applications and theory of Petri nets and concurrency*. Springer. 2013, pp. 311–329.

[23] S. J. Leemans, T. Li, and J. N. van Detten. "Ebi - A Stochastic Process Mining Framework." In: *ICPM Doctoral Consortium and Demo Track. CEUR Workshop Proceedings, vol. to appear. CEUR-WS. org*. 2024.

[24] S. J. Leemans, T. Li, M. Montali, and A. Polyvyanyy. "Stochastic Process Discovery: Can It Be Done Optimally?" In: *International Conference on Advanced Information Systems Engineering*. Springer. 2024, pp. 36–52.

[25] S. J. Leemans and A. Polyvyanyy. "Stochastic-Aware Conformance Checking: An Entropy-Based Approach." In: *Advanced Information Systems Engineering: 32nd International Conference, CAiSE 2020, Grenoble, France, June 8–12, 2020, Proceedings 32*. Springer. 2020, pp. 217–233.

[26] S. J. Leemans, A. F. Syring, and W. M. van der Aalst. “Earth Movers’ Stochastic Conformance Checking.” In: *Business Process Management Forum: BPM Forum 2019, Vienna, Austria, September 1–6, 2019, Proceedings 17*. Springer. 2019, pp. 127–143.

[27] V. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals.” In: *Proceedings of the Soviet physics doklady* (1966).

[28] T. Li, S. J. Leemans, and A. Polyvyanyy. “The Jensen-Shannon Distance Metric for Stochastic Conformance Checking.” In: *ICPM workshops, volume to appear of LNBIP*. 2024.

[29] C. Papamanthou, K. Paparrizos, and N. Samaras. “Computational Experience with Exterior Point Algorithms for the Transportation Problem.” In: *Applied Mathematics and Computation* 158.2 (2004), pp. 459–475.

[30] A. Polyvyanyy, A. Moffat, and L. García-Bañuelos. “An Entropic Relevance Measure for Stochastic Conformance Checking in Process Mining.” In: *2020 2nd International Conference on Process Mining (ICPM)*. IEEE. 2020, pp. 97–104.

[31] L. Portugal, F. Bastos, J. Júdice, J. Paixao, and T. Terlaky. “An Investigation of Interior-Point Algorithms for the Linear Transportation Problem.” In: *SIAM Journal on Scientific Computing* 17.5 (1996), pp. 1202–1223.

[32] A. Rogge-Solti, W. M. van der Aalst, and M. Weske. “Discovering Stochastic Petri Nets with Arbitrary Delay Distributions from Event Logs.” In: *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers 11*. Springer. 2014, pp. 15–27.

[33] A. Rozinat and W. M. Van der Aalst. “Conformance Checking of Processes Based on Monitoring Real Behavior.” In: *Information Systems* 33.1 (2008), pp. 64–95.

[34] Y. Rubner, L. J. Guibas, and C. Tomasi. “The Earth Mover’s Distance, Multi-Dimensional Scaling, and Color-Based Image Retrieval.” In: *Proceedings of the ARPA image understanding workshop*. Vol. 661. 1997, p. 668.

[35] L. Rüschendorf. “The Wasserstein Distance and Approximation Theorems.” In: *Probability Theory and Related Fields* 70.1 (1985), pp. 117–129.

[36] B. F. Van Dongen, A. K. A. de Medeiros, H. M. Verbeek, A. Weijters, and W. M. van Der Aalst. “The ProM Framework: A New Era in Process Mining Tool Support.” In: *Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings 26*. Springer. 2005, pp. 444–454.

[37] F. Zanetti and J. Gondzio. “An Interior Point–Inspired Algorithm for Linear Programs Arising in Discrete Optimal Transport.” In: *INFORMS Journal on Computing* 35.5 (2023), pp. 1061–1078.