Process Mining using Integer Linear Programming

Masters Thesis

T. van der Wiel, BICT

Supervisors:
dr.ir. B.F. van Dongen
prof. W.P.M. Nuijten, PhD

Eindhoven, 23rd July, 2010
Abstract

Workflow systems are nowadays extensively used in a large number of businesses for support of their business processes. Often, these systems generate logs concerning the activities performed in the business process. The process mining field uses this data to analyse, verify and improve the business process. Many approaches have been suggested and implemented, amongst which algorithms based on Integer Linear Programming (ILP). This side of the process mining field is relatively new. In this thesis some of the work by Van der Werf, Van Dongen et al. [24] is revisited for implementation in the new ProM 6 framework. Also a further look on some process mining properties in this ILP approach, like fitness, is given in order to improve the general results of the approach.
Preface

This thesis marks the end of my higher education career, started in 2003 at the Avans University of Applied Science. I will try to find new challenges and opportunities to apply my acquired knowledge with the professional methodologies that I was taught.

I would like to thank my parents who made it possible for me to perform my study, both financially and by giving me the drive to give my best and keep looking towards the future.

Furthermore, I would like to thank my graduation supervisor Boudewijn van Dongen and my graduation tutor Wim Nuijten for their advise and support during my masters project.

A special thanks towards Cor Hurkens for his contributions to the mathematical aspects of my project, as his knowledge on the subject of Integer Linear Programming greatly exceeds mine.

Lastly I would like to thank Joos Buijs for providing me the outline for this thesis and his advise and shared experiences on graduation and related topics.

Theun van der Wiel, July 2010.
# Contents

List of Figures ................................................. ix  
List of Tables .................................................. x  
List of Theorems ................................................. xi  

## 1 Introduction  
1.1 Process modeling ........................................... 1  
1.2 Process mining .............................................. 1  
1.3 Theory of regions ........................................... 2  
1.3.1 Problems .................................................. 3  
1.4 ProM ......................................................... 3  
1.5 Approach .................................................... 3  

## 2 Preliminaries  
2.1 Basics ........................................................ 5  
2.1.1 Logs ....................................................... 6  
2.1.2 Models ..................................................... 6  
2.2 Theory of regions ........................................... 7  
2.3 Process Mining .............................................. 8  

## 3 ILP mining  
3.1 Variable fitness ............................................. 9  
3.2 Advanced properties vs. speed ............................... 12  
3.3 Target function ............................................. 13  
3.4 Place reduction ............................................. 13  
3.4.1 And-join/-split problem ................................ 13  
3.5 Maximization ................................................ 14  
3.6 Conclusion .................................................. 15  

## 4 ProM 6 Plug-in implementation  
4.1 Architecture ................................................. 17  
4.1.1 ILP solver independence .................................. 17  
4.1.2 Package management ..................................... 18  
4.1.3 Extendability ............................................. 18  
4.2 Solver comparison ........................................... 19  
4.2.1 Language problems ....................................... 20  
4.2.2 Speed & Memory ......................................... 20  

## 5 Results  
5.1 Fitness ....................................................... 25  
5.2 Readability .................................................. 27  
5.2.1 Net size .................................................. 28  
5.2.2 Arc properties .......................................... 30  

---

Process Mining using Integer Linear Programming
List of Figures

1.1 Petri net of log in Table 1.1 ................................................. 2
1.2 Fitness of different approaches ........................................... 4
2.1 Region with all possible arcs [24] ......................................... 7
3.1 Basis representation result .................................................... 10
3.2 Separating representation result ............................................ 10
3.3 ILP for causal dependency result .......................................... 11
3.4 ILP for causal dependency .................................................... 12
3.5 Variable fitness ILP ............................................................ 12
3.6 Place before each transition .................................................. 14
3.7 Place after each transition .................................................... 14
3.8 Place before & after each transition ....................................... 14
3.9 Place per causal dependency ................................................. 14
3.10 Variable fitness result with $f = 0.7$ ..................................... 15
3.11 Arc reduction result .......................................................... 15
3.12 Place reduction result ....................................................... 15
4.1 Structural overview ............................................................ 19
4.2 Memory usage of Java-ILP & LpSolve, Java-ILP & CPLEX and OPL & CPLEX respectively per ILP solve ........................................ 22
4.3 Solving times for various logs (ILP for causal dependency) .......... 23
4.4 Solving times for various logs (Single ILP for causal dependency) .. 23
5.1 Event fitness of original log on noise net ................................. 26
5.2 Advanced fitness of original log on noise net ......................... 27
5.3 Ten fold cross validation on noise log ..................................... 27
5.4 Net size with various approaches ........................................... 29
5.5 Mined net ................................................................. 30
5.6 Petri nets of the log $L = \{a; b; c; d^1; c; b; a; d^1; b; c; a; d^1; b; a; c; d^1\}$ .................................................. 30
5.7 Original net ................................................................. 32
5.8 Mined net (Place reduction strategy) ...................................... 32
5.9 Mined net (Arc minimization strategy) .................................... 32
5.10 Mined net (Arc minimization with place reduction strategy) ........ 33
5.11 Mined net (Original & Relaxation strategy) ......................... 33
5.12 Mined net of incomplete log ............................................... 34
List of Tables

1.1 Example of an event log [24] .................................................. 2
4.1 Average ± standard deviation of memory usage per ILP solve in kB .............. 21
5.1 Average connector degree ($\rho_{\text{degree}}$) ........................................ 31
5.2 Density ($\rho_{\text{density}}$) ............................................................. 31
List of Theorems

Definition 2.1: Bag ................................................................. 5
Definition 2.2: Sequence .......................................................... 5
Definition 2.3: Parikh vector ...................................................... 5
Definition 2.4: Sequence concatenation ....................................... 6
Definition 2.5: Grouped workflow log ......................................... 6
Definition 2.6: Prefix-closed language ....................................... 6
Definition 2.7: Weighted prefix-closed language ........................... 6
Definition 2.8: Pre- and postset ................................................ 6
Definition 2.9: Enabledness ....................................................... 6
Definition 2.10: Firing sequence ................................................ 6
Definition 2.11: Region ............................................................. 7
Definition 2.12: ILP formulation ................................................ 8
Definition 2.13: Causal dependency ........................................... 8
Metric 3.1: Event fitness ($f_{event}$) .......................................... 10
Lemma 3.2: ILP mining yields $f_{event} = 1$ ................................... 11
Definition 3.3: Variable fitness ILP .......................................... 11
Lemma 3.4: Variable fitness ILP has minimum fitness ..................... 12
Definition 3.5: Single ILP formulation ....................................... 12
Definition 3.6: Naive maximization ILP ..................................... 14
Definition 5.1: Petri net average connector degree ......................... 31
Definition 5.2: Petri net density ............................................... 31
Chapter 1

Introduction

1.1 Process modeling

Nowadays, a large number of businesses use workflow systems for support of their business processes. These systems are set up or built around a process model that describes the workflow; the activities and their relations as they are performed in the business. There are several formalisms to specify these process models. One of these formalisms is that of Petri nets [19], introduced in 1962 by Carl Adam Petri, offering a graphical notation that is fully formalized. For these reasons, Petri nets are also very suitable for analysis and other computations.

A Petri net is a directed bipartite graph, i.e. a graph consisting of two sets of nodes and directed edges that always link a node from one set to a node from the other set. The first set of nodes is called the transitions. Each transition represents an event that may occur and is denoted by a bar, mostly with the name of the event written inside. The second set of nodes is called the places and they represent the pre- and/or post-conditions of the transition. A place is denoted by a circle, possibly containing one or more dots. The dots represent so called tokens, which are used to define the execution semantics. As mentioned, the edges, commonly referred to as arcs, run from a place to a transition or vice versa. The set of places from which an arc runs to a transition, is called the preset of this transition. The postset of a transition is composed of all places to which an arc runs from this transition. The distribution of tokens in a Petri net is called the marking of the net. A transition in the net may be executed when its preset is a subset of the marking, i.e. all places in its preset have at least one token. Firing a transition results in a new marking by removing one token from each place in its preset and adding a token in each place in its postset. Petri nets are non-deterministic, i.e. when two transition can be executed it is undefined which one will be executed first. This property makes Petri nets well suited for modeling concurrent behavior. In Figure 1.1 an example of a Petri net is shown.

1.2 Process mining

Over time workflow models may become outdated or get lost. As a result, the process executed in the business is unclear. Often, the only up to date information available then is the activity log produced by the workflow system. The field of process mining focuses on the analysis, verification and improvement of the process models in the workflow systems. The analysis of these logs - also called process discovery - uses many algorithms that try to reveal the structure and relationships between the different activities in the logs. The requirement on the logs is that every event in a log must belong to a single execution of a process, also called a case. Often, the date and time at which each event occurs, as well as the user who performed the event, is logged. These properties are not required though, as long as there is an ordering on the events of each case. In Table 1.1 a small example of a log is presented.

A classical example of a process discovery algorithm is the \( \alpha \)-algorithm [6]. This algorithm
constructs relationships on events that directly precede one-another. These relationships are then used to construct a Petri net. The first relationship is the precedence, denoted by \( a \succ b \), containing all pairs of events \( a \) and \( b \) for which there is an occurrence of \( a \) in the log, directly followed by \( b \). Several other relationships are defined over the precedence relation, like the causal dependency relationship, that contains all pairs of events \( a \) and \( b \) for which \( a \succ b \) and \( a \not\succ b \) holds. For example, the precedence relation of the log in Table 1.1 is \( \{(A, B), (A, C), (A, E), (B, C), (B, D), (C, B), (C, D), (E, D)\} \). Based on this, the causal dependencies are defined as \( \{(A, B), (A, C), (A, E), (B, D), (C, D), (E, D)\} \). Two more relations are defined like the causal dependency relationship. The \( \alpha \)-algorithm creates places for which certain conditions over the three relations based on the precedence relation hold. Figure 1.1 shows the result when applying the \( \alpha \)-algorithm on the log in Table 1.1. Note that for each pair \((a, b)\) in the causal dependencies, a place exists that is in the postset of \( a \) and in the preset of \( b \).

### 1.3 Theory of regions

A particular set of algorithms are based on the theory of regions introduced by Ehrenfeucht and Rozenberg [13]. Amongst these algorithms, there are some that use Integer Linear Programming (ILP) or Constraint Programming techniques (see Chapter 3) to generate a Petri net of a log that can precisely reproduce this log, or - if such a net does not exist - one with only minimal extra behavior.

Van der Werf, Van Dongen et al. [24] have given a new implementation that builds on the original idea of using Integer Linear Programming for process mining which gives a solution where the number of places is independent of the size of the log, but is related to the causal dependencies in the log (which is at most \( \frac{|T|^2 - |T| - 1}{2} \) where \( T \) is the set of unique events in the log).

---

<table>
<thead>
<tr>
<th>case id</th>
<th>activity id</th>
<th>originator</th>
<th>case id</th>
<th>activity id</th>
<th>originator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>Sandy</td>
<td>4</td>
<td>C</td>
<td>Sandy</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>Patrick</td>
<td>5</td>
<td>A</td>
<td>Sandy</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>Patrick</td>
<td>1</td>
<td>D</td>
<td>Pearl</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>Sandy</td>
<td>3</td>
<td>C</td>
<td>Sandy</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>Patrick</td>
<td>4</td>
<td>B</td>
<td>Patrick</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>Patrick</td>
<td>5</td>
<td>E</td>
<td>Gary</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>Sandy</td>
<td>3</td>
<td>D</td>
<td>Pearl</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>Patrick</td>
<td>5</td>
<td>D</td>
<td>Pearl</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>Gary</td>
<td>4</td>
<td>D</td>
<td>Pearl</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>Pearl</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.1:** Example of an event log [24]

**Figure 1.1:** Petri net of log in Table 1.1
1.3.1 Problems

The ILP mining technique has a number of drawbacks that make it less usable in real life situations. One of the main disadvantages of the ILP based process mining is that a fitness value of 1 (see Chapter 3) is guaranteed, i.e. all behavior represented in the log is present in the resulting Petri net. In most real life logs, so called ‘noise’ (see Section 3.1) corrupts the data. Detecting and filtering this noise will give results that more closely represent the underlying process.

Furthermore, the exact representation of the log may include many unimportant details. Especially in larger processes this can lead to the mining of highly unreadable nets. A better balance between the log fitness and readability is therefore necessary.

Lastly, ILP mining tends to be slower than other algorithms, as a large amount of calculations is needed to solve ILP problems. The fact that these problems are NP-hard makes it difficult to improve on this point.

1.4 ProM

ProM is a generic open-source framework [4] that allows for process mining tools to be implemented in a supportive standard environment. Many plug-ins have been implemented for process mining, analysis and conversion based on the MXML [11] log format. Recently development started on ProM 6, an entirely new process mining framework based on the more extensive XES [14] log format. Due to this major change, all plug-ins have to be changed or reimplemented, among which also the ILP Miner plug-in [24].

1.5 Approach

The remainder of this thesis focuses on improving the balance between the fitness of the log on the resulting Petri net and the readability of the produced Petri nets and on improving the problems of fitness like noise abstraction. The chapters are organized as follows.

In Chapter 3 several improvements are suggested based on the ILP for causal dependency as presented in [24]. An investigation on how to make ILP based process mining capable of abstracting from noise is presented. Furthermore, several ideas on how to balance between the fitness, which is the focus of earlier work, and readability are introduced. Figure 1.2 shows the allowed behavior in terms of event fitness (see Chapter 3) of the following approaches with respect to the original behavior $L$ in the log, i.e. larger circles represent more allowed behavior.

1. The original behavior $L$ in the log.
2. Theory of regions allows only minimal more behavior.
3. Integer Linear Programming based on causal dependencies [24] stops restricting behavior after finding the most expressive places based on causal dependencies in the log.
4. The variable fitness improvement (see Section 3.1) is similar to 3, but ignores parts of the behavior in the log to abstract from so called 'noise'.
5. The arc minimization improvement (see Section 3.3) is similar also to 3, but lets go of behavior restriction. Instead, the focus is entirely on readability, by minimizing the number of arcs in the resulting Petri net.
   The place reduction improvement (see Section 3.4) stops restricting the behavior even sooner, to obtain better readable results.
6. Maximization (see Section 3.5) tries to restrict the behavior in such a way that as much of the original behavior in the log is allowed, but never any behavior that is not seen in the log. However, this approach has some practical issues.
The architecture of the new plug-in implementation in the ProM 6 framework is discussed in Chapter 4. The plug-in itself is designed to serve as a framework for easy implementation of the improvements presented in Chapter 3, as well as for any future additions. Furthermore, we discuss the differences between several ILP solvers, such that a well reasoned decision can be made between performance and price.

An overview of the effectiveness of each improvement is presented in Chapter 5, by means of various metrics and verification techniques.

In Chapter 6 one can find the conclusions drawn from the results in Chapter 5.
Chapter 2

Preliminaries

In this chapter, an overview of the various mathematical notations in the remainder of this thesis is presented. Section 2.1 concerns basic set operations, used in the other definitions. In Section 2.1.1 the formal definition of a log is given, along with several extensions that can be made. Section 2.1.2 introduces the definition of a Petri net as well as various operations on nets, used for analytical purposes and log replay. Lastly, a short overview of the theory of regions is presented in Section 2.2 with the most important concepts thereof.

2.1 Basics

Let $S$ be a set. The size of $S$ denoted by $|S|$ is $\#_{s \in S}$. For convenience we assume there is an ordering $\text{ord} : S \rightarrow \{0, \ldots, |S|\}$ on any set $S$ such that for any two elements $s_1, s_2 \in S$ holds that $\text{ord}(s_1) = \text{ord}(s_2) \iff s_1 = s_2$.

**Definition 2.1 (Bag)** A bag $\pi$ over a set $S$ is a set of pairs $(e, m(e))$ with $e \in S$ and $m$ is a function $m : e \rightarrow \mathbb{N}$ denoting the multiplicity of $e$.

If $S = \{a, b, c\}$ we write e.g. $\{a^2, b^1\}$ for the bag $\{(a, 2), (b, 1), (c, 0)\}$. The mathematical operators $+$ and $-$ and the logical operators $=$, $<$, $\leq$, $>$ and $\geq$ are defined in a standard way, iterating over the elements in the bag.

Given a pair $p = (p_1, p_2)$, we write $\pi_1(p)$ for $p_1$ and $\pi_2(p)$ for $p_2$.

**Definition 2.2 (Sequence)** Let $n \in \mathbb{N}$. A sequence over a set $S$ is a function $\sigma : \{1, \ldots, n\} \rightarrow S$.

If $n > 0$ and $\sigma(1) = e_1, \ldots, \sigma(n) = e_n$, we write $\sigma = \langle e_1, \ldots, e_n \rangle$, $\sigma_i$ for $\sigma(i)$ and $|\sigma|$ for $n$.

For any element $a$ we omit the use of the angled brackets for explicit sequence definition if the use of sequences is clear from the context.

The set of all finite sequences over the set $S$ is denoted by $S^*$. Let $\sigma$ be a sequence over the set $S$. The Parikh mapping [18] of a sequence $\sigma$, denoted by $\Psi(\sigma)$ maps the number of occurrences of $s \in S$ in $\sigma$ to $S$. For calculating convenience, we define the Parikh vector similar to the Parikh mapping.

**Definition 2.3 (Parikh vector)** We define the Parikh vector of $\sigma$ by $\overrightarrow{\sigma} = [o_1, \ldots, o_{|S|}]$ as a row vector in $\mathbb{N}^{|S|}$ space for which $o_i, 1 \leq i \leq |S|$ denotes the number of occurrences of $s \in S, \text{ord}(s) = i$ in $\sigma$, i.e. $o_i = \#_{1 \leq j \leq |\sigma|} (\sigma_j = s)$.

For example for a sequence $\sigma = \langle a, a, b, a, c \rangle$ over the set $S = \{a, b, c\}$, the Parikh vector of $\sigma$ is

$$\overrightarrow{\sigma} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix} \quad (\text{assuming } \text{ord}(a) = 1, \text{ord}(b) = 2, \text{ord}(c) = 3).$$
CHAPTER 2. PRELIMINARIES

Definition 2.4 (Sequence concatenation) Given two sequences \( \tau, \nu \in S^* \), sequence concatenation denoted by \( \sigma = \tau; \nu \) is defined as \( \sigma : \{1, \ldots, |\tau| + |\nu|\} \to S \) such that for all \( 1 \leq i \leq |\tau| \), \( \sigma_i = \tau_i \) and for all \( 1 \leq i \leq |\nu| \), \( \sigma_{i + |\tau|} = \nu_i \), i.e. \( \sigma \) is the interconnection of \( \tau \) and \( \nu \).

2.1.1 Logs

Definition 2.5 (Grouped workflow log) Let \( T \) be an alphabet, i.e. a finite non-empty set. A grouped workflow log \( L \) is a finite set of sequences over \( T \), i.e. \( L \subseteq T^* \).

A grouped workflow log is also called a language. A language consists of words \( (w \in L) \), also called traces when concerning a log. Each word consists of letters of the alphabet, or events of a certain eventclass in the case of a log.

Definition 2.6 (Prefix-closed language) Given a language \( L \) over alphabet \( T \), it is called prefix closed if and only if

\[ \forall_{w \in T^*, a \in T} (w; a \in L \Rightarrow w \in L) \]

A normal workflow log may contain duplicate traces, which can be grouped into single traces to obtain a grouped workflow log. However, this way the information about the frequency of certain behavior is lost. A solution to this problem is to count the number of traces that is combined when converting to a grouped log.

Definition 2.7 (Weighted prefix-closed language) Let \( T \) be an alphabet and the workflow log \( L \) be a bag over \( T^* \). The weighted prefix-closed language \( L_W \) of this process log is defined by

\[ L_W = \{ (w \in T^*, s \in \mathbb{N}) | \exists_{w' \in L} (w \leq \pi_1(w') \wedge \pi_2(w') > 0) \wedge s = \sum_{w'' \in L \, w \leq \pi_1(w'')} \pi_2(w'') \} \]

For example, the workflow log \( \{ab^2, ac^3\} \) would give the weighted prefix-closed language \( \{a^3, ab^2, ac^3\} \).

2.1.2 Models

We use the P/T-nets as defined by Van der Aalst, Weijters and Maruster [6] as a 3-tuple \( N = (P, T, F) \) where

- \( P \) is a finite set of places.
- \( T \) is a finite set of transitions with \( P \cap T = \emptyset \).
- \( F \subseteq (P \times T) \cup (T \times P) \) is a set of directed arcs.

The nodes of a net consist of all places and transitions \( (X = P \cup T) \).

Definition 2.8 (Pre- and postset) Let \( X \) be a set of nodes and \( A \subseteq X \times X \) a binary relation over \( X \) defining the arcs. For each node \( n \in X \) we define \( \bullet n = \{ x \in X | (x, n) \in A \} \) as its preset and \( n \bullet = \{ x \in X | (n, x) \in A \} \) as its postset.

A marking \( m \) of a net \( N = (P, T, F) \) is a bag over \( P \), denoting the distribution of tokens over the net. A pair \( (N, m) \) is called a marked P/T-net.

Definition 2.9 (Enabledness) Let \( (N = (P, T, F), m) \) be a marked P/T-net. A transition \( t \in T \) is called enabled, denoted by \( (N, m)[t] \), if and only if \( \bullet t \leq m \), i.e. a transition is enabled if all places from which an arc runs to this transition have one or more tokens.

If a transition \( t \) is enabled in the marked net \( (N, m) \), it may fire, denoted by \( (N, m)[t](N, m') \). Firing a transition results in a new marked net \( (N, m') \) with \( m' = m - \bullet t + t \bullet \).

Definition 2.10 (Firing sequence) Let \( (N = (P, T, F), m) \) be a marked P/T-net. A sequence \( \sigma = \langle t_1, \ldots, t_n \rangle \) is a firing sequence in \( (N, m) \) if and only if

\[ \exists_{m_1, \ldots, m_{n-1} \in P((N, m)[t_1])(N, m_1) \ldots (N, m_{n-1})[t_{n-1}]}) \]

The set of all firing sequences in a marked net \( (N, m) \) is denoted by \( \mathcal{T}(N, m) \).
2.2 Theory of regions

The language-based theory of regions [7, 10, 16] is used to create a marked Petri net \((N, m)\) from a prefix-closed language \(L\) of a grouped workflow log \(L \subseteq T^*\), abiding to the following properties:

- In the net \(N\) there is exactly one transition for each element in \(T\), i.e. \(\pi_2(N) = T\).
- Each word \(w \in L\) is a firing sequence in the marked Petri net, i.e. \(\forall w \in L (w \in T(N, m))\)
- The number of firing sequences in \((N, m)\) is minimal, i.e. there does not exist a net \((N', m')\) that abides the first two properties and for which \(\exists w \in T \setminus L (w \not\in T(N', m') \land w \in T(N, m))\).

Finding a net for which the first two properties hold is trivial; the marked net \(((\emptyset, T), (0, 0), 0)\) will accept any trace \(w \in T^*\). In order to satisfy the third property, places should be added to the net such that the behavior is restricted. The theory of regions calculates these places in such a way that the first two properties still hold, using so-called regions.

**Definition 2.11 (Region)** A region of a prefix-closed language \(L\) over \(T\) is a triple \((\vec{x}, \vec{y}, c)\) with \(\vec{x}\) and \(\vec{y}\) being column vectors in \(\{0, 1\}^{|T|}\) space and \(c \in \{0, 1\}\), such that for all words \(w; a \in L, a \in T\) holds that \(c + \vec{x}^T \vec{w} \cdot \vec{y} \geq 0\) (see Definition 2.3 of the Parikh vector for \(\vec{w}^+\) and \(\vec{w}^-\)). In other words, a region represents a place with possibly an arc from and/or to each transition and possibly a token, such that when adding this place to the net, all traces in the log are still firing sequences in the net.

We use standard basis vectors denoted by \(e_i\) to retrieve the value of the \(i\)th dimension of a vector by using the dot product operation. For example, to retrieve the 2nd value of the vector \(\vec{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\), we write \(\vec{v} \cdot e_2 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 1 \cdot 0 + 2 \cdot 1 + 3 \cdot 0 = 2\).

Each possible region represents a place \(p\) with its arcs \(A = \{(x_i \in T, p) | \vec{x} \cdot e_{\text{ord}(x_i)} = 1\} \cup \{(p, y_i \in T) | \vec{y} \cdot e_{\text{ord}(y_i)} = 1\}\) that can be added to the marked net \(((P, T, F), m)\), such that the new net becomes \(((P \cup p, T, F \cup A), m')\) with \(m' = m \cup p\) if \(c = 1\), or \(m' = m\) otherwise. Figure 2.1 shows the place of a region over the alphabet \(T = \{t_1, t_2, t_3, t_4\}\), i.e. the arcs and initial token are added according to \(\vec{x}, \vec{y}\) and \(c\).

In [8] and [16] two approaches are presented that use the language-based theory of regions to construct a Petri net. The first approach, called the basis representation tries to find a set of places that forms a basis for the non-negative integer solution space of the inequalities system of Definition 2.11. This basis is worst-case exponential in the number of equations, making this approach exponential in the number of events in the log. As one might expect, this is not feasible for real-life sized logs. The second approach presented in [8] and [16], called the separating representation, only places are searched for that separate the behavior in the log from the behavior that is not part of the process. They present a method that automatically detects the unwanted behavior based on the log. Both approaches assume that the behavior represented in the log is complete. As this is generally not the case, these methods tend to produce overfitted models, i.e. the resulting net allows only behavior actually seen in the log [3].
2.3 Process Mining

One approach to calculate the regions of a language is by using Integer Linear Programming (ILP). Van der Werf et al. [24] present a target function that ensures that all solutions maximally restrict the behavior in the net such that it still expresses the behavior in the log. It is shown that by removing arcs running to a place, or by adding arcs running from a place, the behavior of the net will be further restricted. The target function they introduce minimizes the number of incoming arcs and maximizes the number of outgoing arcs of a place. Furthermore, this approach does not generate a completely overfitted model. Instead it searches a single place for each causal dependency (see Definition 2.13) in the log. The advantage of this approach is that it does not assume that the log is complete and the number of places is only quadratic in the number of events in the log.

Like the authors of [24], we will not use arc weights, i.e. integer numbers denoting the amount of tokens removed or created by an arc. These arc weights are not commonly used in the field of process mining, hence we will ignore them. It is trivial to extend our approach to incorporate these arc weights, by replacing all binary variables (0 or 1) by integer variables (N).

Van der Werf [22] shows that the ILP formulation presented in [24] contains a small error that causes the algorithm to favor adding tokens over adding arcs. The correct ILP formulation is shown in Definition 2.12. Note that in this thesis the mathematical notation is favored above the matrix notation amongst others because it is closest to most programming languages in which it should be implemented.

**Definition 2.12 (ILP formulation)**

Let $L$ be a prefix-closed language of the log $L \subseteq T^*$ over the alphabet $T$. The ILP formulation is defined as:

\[
\text{Minimize } c + \sum_{w \in L} (c + \mathbf{1}^{w^\Psi (\mathbf{x} - \mathbf{y})}) \\
\text{such that } \forall_{w,a \in L, a \in T} (c + \mathbf{1}^{w^\Psi \mathbf{x}} - \mathbf{1}^{w^\Psi \mathbf{y}} \geq 0) \\
\mathbf{0} \leq \mathbf{x} \leq \mathbf{1} \\
\mathbf{0} \leq \mathbf{y} \leq \mathbf{1} \\
0 \leq c \leq 1
\]

**Definition 2.13 (Causal dependency)**

Let $L \subseteq T^*$ be a workflow log over alphabet $T$. We say there is a causal dependency between $a \in T$ and $b \in T$ denoted by $a \rightarrow_L b$ if $\exists_{\sigma_1, \sigma_2 \in T^*} (\sigma_1; a; b; \sigma_2 \in L) \land \forall_{\sigma_1, \sigma_2 \in T^*} (\sigma_1; b; a; \sigma_2 \notin L)$.

The ILP formulation in Definition 2.12 will generate a place that restricts the net as much as possible, i.e. the arcs running to and from this place are chosen such that the behavior seen in the log is still in the net, but as little as possible extra behavior is still allowed. However, this best solution will remain the same such that we can only find the optimal place. For this reason several approaches exist that add extra constraints such that other places will be minimal each time the ILP problem is solved. The ILP for causal dependency formulation presented by Van der Werf, Van Dongen et al. [24] uses causal dependencies to add a constraint of the form $\mathbf{x}^\cdot \mathbf{e}_{ord}(a) = \mathbf{y}^\cdot \mathbf{e}_{ord}(b) = 1$ for a causal dependency $a \rightarrow_L b$. The ILP formulation is solved for each causal dependency in the log, generating a place per causal dependency. The resulting net will have less places than the methods using the theory of regions, as not all possible solutions of the inequation system (Definition 2.11) are used. As a result, this approach will allow for more behavior unseen in the log. The overfitting problem of the original methods is thereby reduced significantly.
Chapter 3

ILP mining

The Integer Linear Programming approach introduced by Van der Werf et al. gives a partial ordering over all regions of a language that allows for searching unrelated and minimal regions. This way, the unnecessary addition of places that have little added value to the behavioral restriction of the net is avoided. In order to illustrate this we will present an example based on the log $L_{example} = \{\langle a, b, b, e \rangle, \langle a, c, d, e \rangle, \langle a, d, c, e \rangle\}$ \cite{8}. When searching for all regions according to the basis representation, the resulting net has 55 places. Figure 3.1 shows the net with the 13 non-redundant places. The separating representation is already a large improvement, as the resulting net shown in Figure 3.2 has only 6 places. Note that these two approaches do use arc weights. In Figure 3.3 the result of the ILP for causal dependency approach is shown. This result has even one place less, as there is no direct causal relation between $A$ and $E$. This model also allows for more behavior like an unlimited repetition of $B$, the execution of $B$ and $D$ in one trace and an execution of $E$ for each $B$.

The ILP mining technique focuses on finding a Petri net that very precisely represents the behavior as it is seen in the log. In real life situations however, one is not always interested in only this property. Often there is a trade-off between three properties of a net:

- fitness; the model retrieved via process mining techniques should represent the behavior of the log in order to give insight in the underlying process.

- structural correctness; the model retrieved via process mining techniques should abide to certain properties, making its behavior predictable and easy to see.

- simplicity/readability; the model retrieved via process mining techniques should focus on the important aspects of the underlying process, instead of showing all details in the log at once. This is the most vague trade-off point, since what is simple and important?

Most process-mining algorithms focus on one or two of the trade-off points. In order to make process-mining available to non process-mining experts, it is critical to find the right balance in the trade-off points. Ideally this balance is maintained automatically by the process-mining algorithm. In this thesis the focus is on improving the simplicity or readability of the ILP mining results and the abstraction of noise in the logs. The algorithms based on the theory of regions focus only on fitness. However, the flexibility of Integer Linear Programming allows for more complex algorithms that try to find a balance. Future work may address the structural correctness of ILP mining results as well.

The theory of regions, and therefore also the ILP formulation in Definition 2.12, guarantees that the log used can be replayed by the generated Petri net. Due to this fact, the mined Petri net will allow for more behavior when the exact behavior cannot be matched. However, in many cases it is not required to capture the exact behavior of a log in the mined model. One often wants a good approximation of the general behavior, abstracting away from exceptional situations and noise. To this end we will suggest a modification to allow the end user to set a minimum match.
of behavior. In order to do this, we will need a measure for the behavior. There already exist metrics that give a measure to the relation between a log and a Petri net, for example the fitness metric introduced by Rozinat and Van der Aalst [20], however this metric is token based and takes other aspects of the net into account as well. Since we are not interested in these properties, a new fitness metric is presented below that is an abstraction of the event enabledness.

**Metric 3.1 (Event fitness (f\text{event}) )** Let \( L = \{ \sigma_1^n, ..., \sigma_k^n \} \) be a log and \((N, m)\) a marked P/T-net of \( L \). For each pair \((\sigma_i, n_i) \in L, 1 \leq i \leq k\),

\[
d(\sigma_i) = \sum_{j=1}^{\lfloor \sigma_i \rfloor} \begin{cases} 0 & \text{if } \bullet \sigma_i \setminus \pi_2((N, m)[\sigma_i(1)](N, m_1)...[\sigma_i(j-1)](N, m_{j-1})) = \emptyset \\ 1 & \text{otherwise} \end{cases}
\]

is the number of events unable to fire. Note that the sequence that is fired in the net \((N, m)\) does not have to be a firing sequence of \((N, m)\). We simply fire according to the definition of firing by removing the preset of a transition from the marking. Any missing tokens in a place cannot be removed, and the place is therefore left empty in the resulting marking.

The event fitness metric \( f_{\text{event}} \) represents the fraction of events \( e \) in the log that is able to fire based on the token production and consumption of the events that appear before \( e \), assuming that
Lemma 3.2 (ILP mining yields $f_{\text{event}} = 1$) Given a weighted prefix-closed language $L_W$ over the alphabet $T$, the ILP formulation as presented by Van der Werf, Van Dongen et al. [24] always yields a marked P/T-net $(N, m)$ with $f_{\text{event}} = 1$.

Proof By contradiction. We assume that there is a trace $w; a \in L_W$, $a \in T$ such that after firing $w$ in $(N, m)$, obtaining the new net $(N, m')$ there is a place $p \in \bullet a$ such that $p$ does not have a token, i.e. $p \notin m'$. (Note that this will hold for any event $a, x; a; y \in L_W$ since $x; a; y \in L_W \Rightarrow x; a \in L_W$)

The ILP formulation abides to the definition of regions, thus it holds that in each place $c + \frac{w}{\Psi} \cdot \frac{x}{\Psi} - \frac{w}{\Psi} \cdot \frac{y}{\Psi} \geq 0$.

We know from our assumption that the number of initial and produced tokens in $p$ by the events in trace $w$ is less than the number of consumed tokens by the events of trace $w; a$, i.e. $c + \frac{w}{\Psi} \cdot \frac{x}{\Psi} < \frac{w}{\Psi} \cdot \frac{y}{\Psi}$, which is a contradiction. This means that $d(w; a) = 0$ for any $w; a \in L_W$ and thus $f_{\text{event}} = 1$. \hfill $\square$

3.1 Variable fitness

The first idea presented here tries to make the resulting net less restrictive, thereby no longer guaranteeing an event fitness of 1, by at random removing certain constraints.

Making a Petri net without guaranteeing $f_{\text{event}} = 1$ is simple; any will do. Therefore, we want the end user to specify a certain fitness value $f$ that can be adjusted according to their needs. The resulting net should have a fitness of at least $f$. This is done by at random ignoring certain constraints such that only minimal more then $100 \cdot f\%$ of the log is covered. Since the target function ensures minimal behavior, removing other constraints will further restrict the net.

Definition 3.3 (Variable fitness ILP) Let $f \in \mathbb{R}, 0 \leq f \leq 1$ be a user specified fitness value and $L_W$ be a weighted prefix-closed language of the log $L \subset T^*$ over alphabet $T$. We randomly choose $\gamma \subseteq L_W$ such that $\sum_{(w,s) \in \gamma} s \geq f \sum_{(w,s) \in L} s$. We define the variable fitness ILP as:

Minimize $c + \sum_{w \in L_W} (c + \frac{w}{\Psi} \cdot (\frac{x}{\Psi} - \frac{y}{\Psi}))$ such that $\forall_{w, a \in \gamma, a \in T} (c + \frac{w}{\Psi} \cdot (\frac{x}{\Psi} - \frac{y}{\Psi}) = 0)$

$\frac{0}{\Psi} \leq \frac{x}{\Psi} \leq 1$ $x \in \{0, 1\}^{\left|T\right|}$

$\frac{0}{\Psi} \leq \frac{y}{\Psi} \leq 1$ $y \in \{0, 1\}^{\left|T\right|}$

$0 \leq c \leq 1$ $c \in \{0, 1\}$

For example take the log $L = \{a; b, a; c, a; b; c, a; c; b\}$ over alphabet $T = \{a, b, c\}$. The weighted prefix-closed language of this log is $L_W = \{a^4, a; b^2, a; b; c^4, a; c^2, a; c; b^4\}$ and the causal dependencies in the log are $a \rightarrow b$ and $a \rightarrow c$. Solving the ILP for causal dependency on this data
for one place. This ensures that the slack variables $L = \gamma$, a solution is to reduce the ILP problem to an LP-problem. As LP-problems are much easier to real-life logs could be mined using the single ILP approach (see Table 4.1 for log sizes). A possible often much bigger than (hand-)generated examples. In explanation, only the smallest of our four logs, the memory on conventional computers will often be insufficient, as the size of these logs is when the slack variable is 1.

Constraint the slack variable should be added in such a way that it would be automatically true no solution to the problem since otherwise the target function would have preferred that. In any

12 Process Mining using Integer Linear Programming

CHAPTER 3. ILP MINING

We define the single ILP model as:

Definition 3.5 (Single ILP formulation) Let $L$ be a prefix closed language of log $L \subset T^*$ over alphabet $T$, $n$ be the number of places to be searched for and a slack weight $W = 2 + \sum_{w \in L} (1 + |w|)$. We define the single ILP model as:

Minimize $\sum_{i=1}^{n} (c_i + \sum_{w \in L} (c_i + \overline{w} \cdot \overline{x_i} - \overline{y_i}) + W \cdot s_i)$

such that $\forall_{i=1}^{n} \forall_{w,a \in L, a \in T} (c_i + \overline{w} \cdot \overline{x_i} - \overline{a} \cdot \overline{y_i} + W \cdot s_i \geq 0)$

Definition 2.11

Note that the slack weight is chosen such that it is larger than any solution target in the ILP for one place. This ensures that the slack variables $s_i, 1 \leq i \leq n$ are only set to 1 when there is no solution to the problem since otherwise the target function would have preferred that. In any constraint the slack variable should be added in such a way that it would be automatically true when the slack variable is 1.

The drawback of using a single ILP model is the size of the problem. When using real-life logs, the memory on conventional computers will often be insufficient, as the size of these logs is often much bigger than (hand-)generated examples. In explanation, only the smallest of our four real-life logs could be mined using the single ILP approach (see Table 4.1 for log sizes). A possible solution is to reduce the ILP problem to an LP-problem. As LP-problems are much easier to
solve, the solving time and memory consumption may significantly decrease. However, since the resulting solution will have real instead of integer values, it is no longer clear how to interpret it. For example, having 0.4 arc between two transitions does not make much sense.

One possible solution to this is to simply round the solution values to the nearest integer value. It is no longer possible to guarantee amongst others the event fitness value of 1 (Lemma 3.2) this way as the rounded solution may no longer be a solution to the ILP problem. Furthermore, the numerical instability causes that the solution found this way is far off from the integer solution.

Recently there has been done some research that uses multiplication on the solution found in the LP-problem to obtain integer values that can be used in Petri nets with arc weights, however these weights can become rather large and it becomes a lot harder to see the semantics of the net.

### 3.3 Target function

Another way to increase the simplicity of the resulting net is by defining a different target function that focuses less on replicating the exact behavior of the log and more on the readability of the results. This can be achieved by eliminating unnecessary join and split constructs, as they make a net more difficult (see Section 5.2.2).

One possible new target function minimizes the number of arcs in the net, in order to introduce join and split constructs (multiple incoming or outgoing arcs respectively) only when necessary. This can be done per place to give a local optimum and reduce only or-joins and -splits, or to the net as a whole when using the single ILP formulation (Definition 3.5) to obtain the global optimum including and-joins and -splits.

The new target function is defined as follows: Minimize $1(\overrightarrow{x} + \overrightarrow{y})$.

### 3.4 Place reduction

Another approach to further reduce the complexity of the resulting net is by simply adding less places and thereby also less arcs. We still need to differentiate between places as otherwise the ILP formulation would produce the same - minimal - place each time.

Similar to the ILP for causal dependency approach, a place is created where a certain subset of the arcs connecting the place must be present. Instead of doing this for each causal dependency, a place is created for each transition. This reduces the number of places from $\frac{|T|(|T|-1)}{2}$ to $|T|$, making it linear instead of quadratic in the number of unique events in the log.

The ILP constraint that is added to link a place initially to a transition $t$ can be either $\overrightarrow{y} \cdot e_{ord(t)} = 1$ to add the place in front of the transition, or $\overrightarrow{x} \cdot e_{ord(t)} = 1$ to add the place after the transition.

#### 3.4.1 And-join/-split problem

There is however a problem with this approach when it comes to rediscovering certain workflow patterns. For example take the log in Table 1.1. The workflow net that has the exact behavior of this log and can be found using the ILP for causal dependency approach is shown in Figure 3.9. This net contains four basic workflow patterns, namely the and-join, the and-split, the or-join and the or-split. As can be seen in Figure 3.6 and 3.7, this new approach is incapable to rediscover either the and-join or the and-split.

A logical solution to this problem is to search for both a place before each transition as well as a place after each transition. This doubles the amount of places, but is still linear in the amount of unique events in the log. Figure 3.8 shows the result of combining these approaches. The so-called end places which have no outgoing arcs could be removed in a post-processing step as they do not restrict the behavior.
Lastly, an attempt to get better readability by reducing the fitness, as is the idea for the variable fitness approach, is to let the behavior in the log be the maximum allowed instead of the minimum. This way we might lose some noise or less important details that are in the log. If the behavior in the log $L$ cannot be exactly matched then the resulting net $(N, m)$ should allow as much as possible, but nothing unobserved, i.e. let $L$ be the prefix closed language of $L$, $\forall w \in T(N,m)(w \in L)$ stating that all possible firing sequences of the resulting net should be in the log. This is the exact opposite of the second property of the theory of regions (see Section 2.2).

The attentive reader will probably notice that checking if all possible traces of a net are in the log is impossible when still constructing the net. Therefore this idea is not practically executable, as it would require many iterations and still not guarantee that the optimal solution is found. A possible solution to this problem is considering a very naive approach where we say that any place must contain at most enough tokens to fire the next event in a trace in the log. However, this will introduce a new problem. The constraints no longer restrict the target function, such that it will simply give a minimal solution, i.e. no incoming arcs and all outgoing arcs. For this reason a different target function is required. Unfortunately, changing the minimization function into a maximization function does not help much, as it always results in a so called flower model, i.e. a model with a single place that has a loop to each transition. This allows for any sequence over the alphabet to fire, and the results obtained with this approach are therefore so far off the original goal that they are not useful, since the results of the suboptimal naive approach have too little in common with the original objectives.

**Definition 3.6 (Naive maximization ILP)** Let $L$ be a prefix-closed language of the log $L \subset T^*$ over the alphabet $T$. The naive maximization ILP formulation is defined as:

Maximize $c + \sum_{w \in L}(c + \overline{w})\Phi(\overline{x} - \overline{y})$

such that $\forall w, a \in L, a \in T(c + \overline{w}\Phi - \overline{w}a\Phi \overline{y} \leq 0)$

\[
\begin{align*}
0 \leq \overline{x} & \leq 1 \\
0 \leq \overline{y} & \leq 1 \\
0 \leq c & \leq 1
\end{align*}
\]

$x \in \{0, 1\}^{|T|}$

$y \in \{0, 1\}^{|T|}$

$c \in \{0, 1\}$
3.6 Conclusion

At the beginning of this chapter we introduced the small example log $L_{example}$ to illustrate the differences between the various approaches. We have applied our improvements to this log to compare the results. Figure 3.10 shows the result of the variable fitness approach. As one can see, the $B$ transition was chosen to be noise and can no longer execute. The other behavior is preserved. Figure 3.11 shows the result when applying the arc reduction approach. The behavior of this net is less restrictive than that of the ILP for causal dependency approach (see Figure 3.3), as expected. It seems to introduce extra places, although this is not really the case as can be read in Section 5.2.3. Lastly, the place reduction technique yields the net as shown in Figure 3.12. Again, as expected, this net is slightly less restrictive than the ILP for causal dependency approach. Note that it is hard to compare readability on such a small example. Therefore, we give a more detailed analysis on the improvements presented in Chapter 5.

In order to calculate and analyse the results of ILP process mining algorithms on real-life data, we need an automated tool as the amount of calculations needed is unfeasible to do by hand. For this reason the ILP Miner plug-in as presented in the next chapter is created in the ProM 6 process mining tool framework.
Chapter 4

ProM 6 Plug-in implementation

An existing implementation based on the work of Van der Werf et al. [24] was present in the ProM framework. However, it was necessary to reimplement the plug-in as it was implemented in an ad-hoc manner. In order to make the plug-in capable of supporting various ILP solvers and different mining approaches as presented in Chapter 3, a new plug-in that acts as a framework was built in ProM 6.

4.1 Architecture

When rebuilding the ILP Miner plug-in for ProM 6, the following design goals were kept in mind:

- For testing purposes, the plug-in should be ILP solver independent.
- The end user of ProM 6 should not have to install any extra software outside of the ProM 6 framework.
- The plug-in should be easily extensible, i.e. new variants of the ILP problem should be easy to implement and make available in the plug-in.

4.1.1 ILP solver independence

In order to show the ILP solver independence, we decided to implement two different solvers into our plug-in, namely CPLEX and LpSolve. Other solvers can be simply added at a later time, if necessary.

The IBM ILOG CPLEX solver seemed a logical choice as we have in-house knowledge of this system. There is however also a large disadvantage in using this professional solver; the academic license did not allow for the distribution of the solver libraries. It would therefore be necessary for each end user to buy and install the IBM ILOG CPLEX solver when using the new ILP Miner plug-in in ProM 6.

In order to avoid the problems of a professional ILP solver, we have decided to let the end user choose which ILP solver should be used. This way, the end-users that are in the possession of IBM ILOG can use the more advanced processing power of this tool, whereas the other end-users can still use the plug-in in combination with an open source ILP solver like LpSolve [2], which was used in the previous implementation.

To achieve the seamless use of various ILP solvers with very different interfaces, an abstraction had to be made. This abstraction layer uses the Java ILP interface [1]. This interface provides a generic way to specify the ILP problem and parse it to the ILP solver of choice. Although the generic language is less expressive, it is very well possible to express the mining ILP problem. Besides the IBM ILOG and the LpSolve solvers, it supports four other solvers, three of which are free to use. The ProM 6 plug-in is built in such a way that it supports basically all these
CHAPTER 4. PROM 6 PLUG-IN IMPLEMENTATION

solvers, although the libraries have to be made available via ProM. We have performed some initial experiments with the GLPK solver, but it caused so many memory errors that we have not continued testing.

Lastly, in order to test further performance increase, we have implemented a second interface language to the CPLEX solver, named OPL. However, this interface is not solver independent and therefore it is necessary to make a separate implementation of each ILP mining approach if it should be available via this approach as well.

4.1.2 Package management

One of the new features in ProM 6 is the package manager. It allows for developers to publish the latest versions of their plug-in on their own server, such that they no longer rely on the release of ProM 6 builds to spread their plug-ins. The ProM 6 framework will scan for these updates whenever the end-user wants.

Another main advantage of the package manager is the fact that it can anticipate on the operating system on which ProM 6 is installed. This way, operating system dependent libraries or programs can be selectively downloaded, greatly decreasing the download size of a plug-in.

The ILP Miner plug-in comes with the open-source LpSolve ILP solver. Therefore it makes explicit use of the package manager functionality.

4.1.3 Extendability

The design of the plug-in is based on extendibility. It should be as easy as possible to implement a new strategy for the construction of a Petri net. For this reason, the plug-in will generate context related data, such as the (weighted) prefix closed language of the log and a mapping between log transitions and an index that can be used in the ILP problem matrices more easily.

Furthermore, it contains a user interface that allows the end-user to specify the ILP solver and the strategy to be used, based on a list of available strategies and the specific settings for this strategy. The plug-in will then instantiate the chosen strategy and execute it.

There is a differentiation between two kinds of strategies:

- A strategy model, which can extend another strategy model. It will define its own data and execution strategy to find Petri net places.

- A strategy extension, which adds constraints to a strategy model ILP problem before it is executed.

Figure 4.1 shows a structural overview of the plug-in. The main class is ILPMiner, which also contains the code that lets ProM 6 recognize the plug-in and its input and output. It is also responsible for showing the settings GUI and for instantiating the correct strategy as provided in the settings by the end-user. A strategy consists of a class that implements the ILPModel interface, and contains a special Java annotation [15] upon which the plug-in recognizes it as a strategy. All classes that implement an interface of the plug-in are placed in the templates package in order to separate the main plug-in code from other code. There are two kinds of ILPModel interfaces, namely the ILPModelJavaILP interface that can execute the strategy on a variety of ILP solvers and the ILPModelCPLEX interface that uses OPL CPLEX only. This last version offers a small speed increase on large models but is suffering a bit of memory problems (see also Section 4.2.2) and is mainly available for testing purposes.

By default there are no extra settings for the strategy chosen by the end-user. However in some cases, like the variable fitness approach discussed in Section 3.1, extra input is required. Therefore an ILPModel can override the standard GUI by generating some controls that are linked to an implementation of the ILPModelSettings interface. The plug-in will automatically show these controls and assure that the settings are parsed when the strategy is instantiated.

Lastly, in order to reduce programming, strategy extensions are introduced. Extensions assure extra properties on the ILP result, like empty after completion or pureness [24], that can be applied to a set of strategies and have the following characteristics:
• No extra data is required, but it uses that which is used in the strategy.
• Only constraints are added to the ILP formulation and never modified or removed.

Like strategies, strategy extensions implement the ILPModelExtension interface and need to be annotated with a special annotation in order for the plug-in to recognize it. A strategy extension has a reference to the ILPModel implementation to which it can be applied (including all inheriting classes) and optionally a reference to a required extension that has to be used in order for the extension to work, like elementary nets that are always also pure nets [24]. The plug-in automatically shows all available extensions that can be applied to the strategy in the GUI and ensures that any required extensions are automatically selected.

4.2 Solver comparison

In order to help the end user in the choice between a free open-source solver (LpSolve) and a paid solver (CPLEX), a comparison is presented that shows the benefits of using a particular solver. These comparisons have been made on a single computer\(^1\) such that a fair comparison can be made. Although also some results are presented in comparison with the previous plug-in as presented in [24], it must be noted that the search is not identical as it searches for so called reset-

\(^1\)These calculations were performed on a 2.66GHz Dual Xeon 5,430 with 16Gb RAM, using Java-ILP 1.1, LpSolve 5.5.0.15, OPL 6.3 and CPLEX 12.1.
and inhibitor-arcs, along with some other minor differences. The number of places found though is approximately the same, as all tests use the ILP for causal-dependency strategy [24].

4.2.1 Language problems

First a note has to be made on OPL; the language that can be used to parse an ILP problem to CPLEX. This advanced language allows the programmer to use complex data-constructs, data-separation from the ILP model and advanced constructs like loops and logical operators such as && and ||. However, this model has to be parsed to the standard form ILP, in order for CPLEX to handle it. Not all constructs can be efficiently converted, making the actual problem that has to be solved much more complex. As a good modeler can make better models than a computer, it should be kept in mind that advanced tools can be convenient, but are not by definition better.

A simple example will show how some optimization in the ILP model will greatly improve performance.

forall(w in 0..6)
{
  (c[w] == 1 && sum(t in 0..8) x[w][t] == 0 && y[w][Cds[w].To] == 1) || s[w] == 1;
}

Example 1: OPL with logical operators

The example used concerns a trivial ILP model specified in OPL that generates 7 similar constraints using a loop (forall(w in 0..6)). Note the use of ranges and arrays in OPL to facilitate amongst others the use of loops.

The use of the && operator is not a problem. Just splitting the constraint x && y; in x; y; is sufficient. The use of the || operator however is a different story. In Example 1 the fact that all variables are binary, i.e. 0 or 1, can be used to rewrite the constraint into three smaller ones shown in Example 2.

forall(w in 0..6)
{
  c[w] + s[w] >= 1;
  sum(t in 0..8) (x[w][t] - s[w]) <= 0;
  y[w][Cds[w].To] + s[w] >= 1;
}

Example 2: OPL without logical operators

The size of the standard form ILP problem after presolving will be reduced this way from 58 variables and 70 constraints in example 1 to 28 variables and 41 constraints in example 2. As one might expect, this leads to a significant speed increase as the solving time is exponential in the number of variables.

4.2.2 Speed & Memory

In order to give a comparison between the different solvers and implementation techniques, various logs from real-life workflow systems and generated logs from a workflow net have been used on
the ILP-Miner plug-in described in this chapter. These same logs have also been used on a differ-
ent plug-in that uses ILP mining techniques (henceforth called ‘Old implementation’), although
this plug-in has no user-adjustable properties and is capable of finding inhibitor and reset arcs.
When analyzing the logs, the ‘Petri net’ variant was used with the ‘Number of places’ set to ‘Per
Causal Dependency’ and the ‘Search for separate initial places’ option on, without any other ILP
extensions. These settings are available for all solvers and are therefore a good comparison. The
results presented below do not include the graph visualization after the mining.

First, a look at the memory usage reveals some interesting details. When looking at the data
shown in Table 4.1, we see that there are several values that are very large for a certain solver,
compared with the other two. After further inspection it seems that, although the generated
data was removed after calculation, ProM 6 or the Java Virtual Machine (JVM) keeps some data
in memory that has nothing to do with the actual plug-in. For this reason, we cannot directly
conclude that a certain solver is less memory efficient. When looking to the average memory
consumption over all 19 logs tested, we can conclude that due to the extra buffering layer in
Java-ILP, a lot of extra memory is used.

When the memory usage of some logs in between each solved ILP problem is plotted in a graph
for each solver (see Figure 4.2), we see some interesting developments. The memory increases for
each solve to drop back to the starting value after 4 to 5 solves. Furthermore, we see in the
case of OPL & CPLEX a steady increase in memory consumption. We were unable to locate the
problem in Java, so it appears to be a problem of the CPLEX library. For speed increase, the
OPL language was used to separate the data values from the logical constraints. This way you can
reuse an OPL model. However, keeping the model alive after solving seems to keep some memory
in use for CPLEX. This explains the fact that the OPL & CPLEX implementation sometimes
 crashes when the log is too big, i.e. too many places have to be calculated. Contact with the
creators of the CPLEX library has unfortunately not yet led to a solution.

<table>
<thead>
<tr>
<th>cases/ events /places</th>
<th>Java-ILP &amp; LpSolve</th>
<th>Java-ILP &amp; CPLEX</th>
<th>OPL &amp; CPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>a120n00_1</td>
<td>200/ 1,236 /12</td>
<td>7,152 ± 33</td>
<td>339,792 ± 383</td>
</tr>
<tr>
<td>a120n00_2</td>
<td>600/ 3,696 /12</td>
<td>7,508 ± 43</td>
<td>16,335 ± 72</td>
</tr>
<tr>
<td>a120n00_3</td>
<td>1,000/ 6,154 /12</td>
<td>36,432 ± 50</td>
<td>27,557 ± 52</td>
</tr>
<tr>
<td>a120n00_4</td>
<td>1,400/ 8,666 /12</td>
<td>48,920 ± 234</td>
<td>57,930 ± 222</td>
</tr>
<tr>
<td>a120n00_5</td>
<td>1,800/ 11,146 /12</td>
<td>80,383 ± 79</td>
<td>71,653 ± 1,389</td>
</tr>
<tr>
<td>a220n00_1</td>
<td>100/ 1,597 /22</td>
<td>100,805 ± 2,368</td>
<td>117,947 ± 2,759</td>
</tr>
<tr>
<td>a220n00_2</td>
<td>300/ 5,031 /22</td>
<td>383,197 ± 6,911</td>
<td>399,377 ± 7,948</td>
</tr>
<tr>
<td>a220n00_3</td>
<td>500/ 9,185 /22</td>
<td>442,988 ± 10,812</td>
<td>434,287 ± 9,902</td>
</tr>
<tr>
<td>a220n00_4</td>
<td>700/ 13,550 /22</td>
<td>475,508 ± 11,539</td>
<td>491,754 ± 13,538</td>
</tr>
<tr>
<td>a220n00_5</td>
<td>900/ 17,418 /22</td>
<td>529,370 ± 14,252</td>
<td>514,513 ± 9,261</td>
</tr>
<tr>
<td>a320n00_1</td>
<td>100/ 2,636 /32</td>
<td>211,916 ± 4,825</td>
<td>145,232 ± 4,137</td>
</tr>
<tr>
<td>a320n00_2</td>
<td>300/ 7,710 /32</td>
<td>516,172 ± 10,801</td>
<td>553,165 ± 12,311</td>
</tr>
<tr>
<td>a320n00_3</td>
<td>500/ 12,392 /32</td>
<td>624,165 ± 10,136</td>
<td>613,234 ± 10,736</td>
</tr>
<tr>
<td>a320n00_4</td>
<td>700/ 18,450 /32</td>
<td>686,698 ± 13,752</td>
<td>705,432 ± 14,340</td>
</tr>
<tr>
<td>a320n00_5</td>
<td>900/ 23,589 /32</td>
<td>461,660 ± 3,363</td>
<td>482,317 ± 6,448</td>
</tr>
<tr>
<td>Real Life 1</td>
<td>84/ 616 /15</td>
<td>55,694 ± 700</td>
<td>33,139 ± 564</td>
</tr>
<tr>
<td>Real Life 2</td>
<td>14,279/119,021/15</td>
<td>144,587 ± 3,818</td>
<td>137,308 ± 3,993</td>
</tr>
<tr>
<td>Real Life 3</td>
<td>369/ 6,484 /95</td>
<td>38,154 ± 3,095</td>
<td>280,576 ± 8,894</td>
</tr>
<tr>
<td>Real Life 4</td>
<td>672/ 9,597 /14</td>
<td>28,490 ± 1,504</td>
<td>64,263 ± 2,594</td>
</tr>
<tr>
<td>Average</td>
<td>1,290/ 14,640 /26</td>
<td>263,239</td>
<td>297,169</td>
</tr>
</tbody>
</table>

Table 4.1: Average ± standard deviation of memory usage per ILP solve in kB

The total calculation time is what matters most when deciding to buy a license for CPLEX or
using the open-source LpSolve. Figure 4.3 shows the calculation time for various logs, using the
‘Petri net’ variant with the ‘Number of places’ set to ‘Per Causal Dependency’ and the ‘Search for
separate initial places’ option on, without any other ILP extensions. Figure 4.4 shows calculation time of the same logs, using the ‘Petri net - Single ILP’ variant with the ‘Number of places’ set to ‘Per Causal Dependency’ and the ‘Search for separate initial places’ option on, without any other ILP extensions. Since ILP solving times are exponential in the number of variables, calculating the same logs as a single ILP problem is expected to be slower than calculating all places separately. What is interesting to see is the fact that calculation times for relatively small problems are more or less the same for LpSolve and CPLEX, but for larger problems CPLEX can become significantly faster. Also good to see is that, except for trivial problems, all solvers are faster then the old implementation, that was even unable to find a solution for the larger logs.

---

**Figure 4.2:** Memory usage of Java-ILP & LpSolve, Java-ILP & CPLEX and OPL & CPLEX respectively per ILP solve.
Figure 4.3: Solving times for various logs (ILP for causal dependency).

Figure 4.4: Solving times for various logs (Single ILP for causal dependency).
Chapter 5

Results

In Chapter 3, several variations on the ILP mining algorithm were presented that focused on improving the readability of the resulting nets and on the abstraction of noise commonly present in logs, in order to find a balance between the fitness and the readability. In this chapter we will show what the improvements and disadvantages of each variation are with respect to these goals.

5.1 Fitness

The variable fitness approach presented in Section 3.1 tries to abstract from noise in the logs by at random ignoring a predefined amount of words in the prefix-closed language of the log.

In order to give an indication of the ability of the variable fitness approach to abstract from noise, we first place some notes on what noise is, and what its properties are. Weijters and Van der Aalst [21] introduced noise as a combination of four changes on the original log:

- Removal of the start of a trace.
- Removal of the end of a trace.
- Removal of a part of the body of a trace.
- Interchange of two random elements of a trace.

Assuming that the original log was large enough, i.e. contains several traces that express the same behavior, then it is trivial to see that any of these four methods will at most add causal relations in the log. Only in rare cases (for example the interchange of two directly adjacent elements) or the introduction of a huge amount of noise, it can occur that causal relations are changed or destroyed. Of course, this implies that algorithms that use these relations for the construction of a net, will simply add the 'extra' behavior. This would mean that if the original behavior is also present, fitness metrics like the event fitness are of less use.

We have conducted some experiments with logs generated from a net, both with noise and without noise. The logs with noise are used to mine a net with the variable fitness approach as presented in Section 3.1. We then calculate the event fitness of the noise-free log on the resulting net. We repeat this task for various fitness values.

Expected is that when lowering the fitness value, the noise is filtered, giving a better fitness value for the noise-free log. When the fitness value becomes too low, the 'good' behavior is also filtered, giving a low fitness value. The results of these test can be seen in Figure 5.1. As discussed above, we see that the noise-free log will fit in the net when mining with a fitness value of 1, i.e. not filtering any noise at all. We do however see that the fitness value of the noise free log remains good up to a mining fitness value of 0.5. Above this value the net was most likely overfitting, i.e. allowing more behavior then necessary.
When using a more advanced fitness metrics available in ProM 6, that also looks at overfitting and structural correctness of the net, based on the work of Rozinat and Van der Aalst [20], we get a different picture as can be seen in Figure 5.2. Here we can see a slight increase in the fitness, indicating that the behavior of the noise-free logs is more closely related to the net when using a lower fitness setting. The results are however not completely conclusive, as the differences are not big and can be caused by other aspects like structural correctness.

Secondly we have performed a ten fold cross validation with noise logs, where we split the log in ten parts and for all ten possible combinations of nine parts we apply ILP mining. In each case we use the remaining part to calculate the event fitness. The results are shown in Figure 5.3. What we hope to see is a linear decay of the average fitness, when testing each 10th part of the noise log, when mining the net with a lower fitness setting. This is unfortunately not the case, probably since our metric is calculated in a way that even allows events to fire for which the prefix was unable to fire. However, a clear decay is present, showing that the allowed behavior of the net indeed decreases.

Further comparison of the mined nets reveals that due to the removal of some constraints, several places had a smaller number of incoming arcs. This is due to the target function that reduces behavior as much as possible. Similarly new outgoing arcs were added. The total number of arcs in the net \( (P, T, F, m) \) will stabilize towards \( |P| \cdot (|T| + 1) \), which is quite a large number. Therefore the readability suffers for too small fitness values.

Overall, the results of the variable fitness approach are not conclusive, but provide a starting point for further research on the removal of noise in ILP mining. We have shown that using the correct metrics is not trivial when dealing with noise. Also, different methods of testing the difference between a log and a net can be used, each giving different results.

**Figure 5.1:** Event fitness of original log on noise net
CHAPTER 5. RESULTS

Figure 5.2: Advanced fitness of original log on noise net

Figure 5.3: Ten fold cross validation on noise log

5.2 Readability

The goal of each improvement is to balance the fitness and the readability of the resulting nets. However, as mentioned before, there is no clear definition of what is readable. In short one could say that it is the ability for non expert people to be able to understand the behavior expressed in
the net. As we do not have the possibility to extensively validate this, we will use several metrics like the net size and various arc properties [17].

5.2.1 Net size
First of all, the sheer size of the resulting net may cause people to lose focus of what is expressed. One would have to make too big of an effort to understand only a small part, making it impossible to get the big picture. Figure 5.4 shows the resulting net for various approaches on the same log. This log is medium sized, but contains a lot of exceptional behavior, making the process as a whole very hard to understand. The following strategies are displayed;


Strategy 2: Maximization (Section 3.5) with arc minimization goal (Section 3.3) (142 places & 413 arcs).

Strategy 3: Place reduction with a place before & after each transition (Section 3.4) (36 places & 369 arcs).

Strategy 4: Arc minimization goal (Section 3.3) with a place per causal dependency (138 places & 465 arcs).

Strategy 5: Variable relaxation & solution rounding (Section 3.2) with a place per causal dependency (94 places & 925 arcs).

Strategy 6: Arc minimization goal (Section 3.3) with a place before & after each transition (Section 3.4) (73 places & 131 arcs).
Figure 5.4: Net size with various approaches
CHAPTER 5. RESULTS

Although all nets but 6 are most likely too big for any person to see the behavior of the net, all techniques presented here are an improvement to the readability when looking at size alone. There are however some drawbacks that must be kept in mind.

Strategy 2: The maximization approach is very naive. No guarantees on the fitness can be given at all.

Proof The log \( \{a; a^n, a; b^1\} \) over the alphabet \( \{a, b\} \) yields a net as shown in Figure 5.5 with an arbitrary small event fitness by increasing \( n \). (\( f_{\text{event}} = \frac{2^n-2}{2^n} \) since only \( b \) can ever fire) □

Strategy 3: Place reduction often stops searching for places even sooner than the ILP for Causal Dependency, thereby less restricting the net.

Strategy 4: Arc minimization preserves \( f_{\text{event}} = 1 \), but possibly allows much extra behavior. For example the log \( \{a; x | x \in \{t_1, \ldots, t_n\}\} \) would produce a net with an or-split after \( a \) to all \( x \in \{t_1, \ldots, t_n\} \) with the ILP for causal dependency approach, but would produce a net with an and-split after \( a \) to all \( x \in \{t_1, \ldots, t_n\} \) with the arc minimization approach. This allows all words \( a; X \) with \( X \) being a random sequence over \( \{t_1, \ldots, t_n\} \) with \( \forall x_1, x_2 \in X x_1 \neq x_2 \), which includes the original language.

Strategy 5: Solution rounding destroys many properties amongst which the guaranteed event fitness value \( f_{\text{event}} = 1 \). For example take the partial net in Figure 5.6.a of the log \( L = \{a; b; c; d_1^1, c; b; a; d_1^1, b; c; a; d_1^1, b; a; c; d_1^1\} \). Figure 5.6.b shows a possible solution of the LP problem. After rounding, the net in Figure 5.6.c is obtained, that has an event fitness \( f_{\text{event}} = \frac{3}{4} \), as \( d \) can never fire.

Strategy 6: Combining Arc minimization and place reduction leads to parallelism and therefore a lot of extra possible behavior.

The only improvement that has no obvious drawbacks is the place reduction technique. Although adding more places will further reduce the behavior, each place that is added will maximally reduce the behavior. The transition constraint enforces that places are added through the entire net, such that the behavior is reduced throughout the net.

5.2.2 Arc properties

In order to further compare the readability, the metrics presented by Mendling, Reijers and Cardoso [17] to be the most significant in the readability of a net can be adopted for Petri nets. The two metrics both concern the number of arcs in the net; the average connector degree states the average number of arcs at a join or split connector and the density states the number of arcs in...
the net with respect to the maximal number of arcs that could be in the net. Since these metrics were tested on EPC like nets, they do not quite apply to Petri nets as for example Petri nets do not have special connector nodes. Any node in a Petri net can be either a join, split or both. For this reason we first define a ‘translation’ of the metrics, such that they can be applied to Petri nets.

Definition 5.1 (Petri net average connector degree) Given a node \( n \in P \cup T \) of a Petri net \( N = (P, T, F) \), we define that \( n \) is joining by

\[
\iota(n) = \begin{cases} 
1 & \text{if } |\bullet n| > 1 \\
0 & \text{otherwise}
\end{cases}
\]

and that \( n \) is splitting by

\[
\varsigma(n) = \begin{cases} 
1 & \text{if } |n \cdot| > 1 \\
0 & \text{otherwise}
\end{cases}
\]

The average connector degree of \( N \) is defined by

\[
\rho_{\text{degree}}(N) = \frac{\sum_{n \in P \cup T} \iota(n) \cdot |\bullet n| + \varsigma(n) \cdot |n \cdot|}{\sum_{n \in P \cup T} \iota(n) + \varsigma(n)}
\]

Definition 5.2 (Petri net density) Let \( N = (P, T, F) \) be a petri net. The density of \( N \) is defined as

\[
\rho_{\text{density}}(N) = \frac{|F|}{|P| \cdot |T|}
\]

Note that the density is not the same for a Petri net and its EPC counterpart as the number of nodes used to express certain behavior is different in both notations, whereas this number is used to calculate the density. We think it is more important how many arcs there actually are instead of how many there would be in a different notation.

In Table 5.1 and 5.2 the average connector degree and the density respectively of some real life logs is shown per strategy as presented in the previous section.

<table>
<thead>
<tr>
<th>Real Life 1</th>
<th>Real Life 2</th>
<th>Real Life 3</th>
<th>Real Life 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>4.30</td>
<td>3.56</td>
<td>8.66</td>
</tr>
<tr>
<td>Maximization</td>
<td>3.29</td>
<td>3.31</td>
<td>4.15</td>
</tr>
<tr>
<td>Place reduction</td>
<td><strong>2.64</strong></td>
<td>3.24</td>
<td>4.57</td>
</tr>
<tr>
<td>Arc minimization</td>
<td>3.40</td>
<td>4.32</td>
<td>5.07</td>
</tr>
<tr>
<td>Relaxation</td>
<td>3.67</td>
<td>4.83</td>
<td>7.65</td>
</tr>
<tr>
<td>Arc min. + Place red.</td>
<td>3.60</td>
<td><strong>2.68</strong></td>
<td><strong>3.15</strong></td>
</tr>
</tbody>
</table>

**Table 5.1:** Average connector degree (\( \rho_{\text{degree}} \))

<table>
<thead>
<tr>
<th>Real Life 1</th>
<th>Real Life 2</th>
<th>Real Life 3</th>
<th>Real Life 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0.1692</td>
<td>0.2889</td>
<td>0.0500</td>
</tr>
<tr>
<td>Maximization</td>
<td>0.0994</td>
<td>0.0979</td>
<td>0.0159</td>
</tr>
<tr>
<td>Place reduction</td>
<td>0.0727</td>
<td>0.1458</td>
<td>0.0225</td>
</tr>
<tr>
<td>Arc minimization</td>
<td>0.1033</td>
<td>0.1199</td>
<td>0.0195</td>
</tr>
<tr>
<td>Relaxation</td>
<td>0.2500</td>
<td>0.2196</td>
<td>0.0451</td>
</tr>
<tr>
<td>Arc min. + Place red.</td>
<td><strong>0.0486</strong></td>
<td><strong>0.0598</strong></td>
<td><strong>0.0092</strong></td>
</tr>
</tbody>
</table>

**Table 5.2:** Density (\( \rho_{\text{density}} \))

As one can see, all presented improvements have a lower average connector degree and density with respect to the original causal dependency strategy for most of the logs used. Therefore, all improvements have an increased readability. Some techniques impose a larger improvement than others, like the arc minimization with place reduction, but have other problems as discussed before.
5.2.3 Control-flow constructs

In Petri nets certain control-flow constructs, or patterns, can be found. It is however not trivial to recognize these patterns in a log of this net. Many algorithms have trouble finding certain patterns, which often leads to more complex mining results, as the algorithm will construct a workaround or leave the pattern out completely. Van Dongen, Alves de Medeiros and Wen [12] have presented some results on which patterns can be mined be certain algorithms, amongst which also the ILP for causal dependency algorithm [24]. The main patterns on which ILP mining can reconstruct a net are sequences, parallelism, choices, loops and non-free-choice [5] and additionally parallelism. Figure 5.7 shows a Petri net containing all these patterns, as well as an invisible task, that is used to obtain the log: 

\{
\langle A, B_1, D, F, G, I_1, J \rangle, \quad \langle A, B_1, C, D, E, E, F, G, I_1, J \rangle, \\
\langle A, B_2, H, C, D, E, F, I_2, J \rangle, \quad \langle A, B_2, C, H, D, F, I_2, J \rangle, \\
\langle A, B_2, D, H, F, I_2, J \rangle, \quad \langle A, B_2, D, E, F, H, I_2, J \rangle
\}

This log was used to mine a net with all improvements presented in this thesis. The results are shown in Figure 5.8 (Place reduction strategy), 5.9 (Arc minimization strategy), 5.11 (Relaxation strategy) and 5.10 (Arc minimization with place reduction strategy).
Figure 5.10: Mined net (Arc minimization with place reduction strategy)

Figure 5.11: Mined net (Original & Relaxation strategy)
As expected, the original ILP for causal dependency is able to reconstruct most patterns of the original net (See Figure 5.11). The invisible task is logically not reconstructed, as it would require the addition of an extra transition, which is not allowed in the theory of regions. However, as the invisible task, which allows to skip transition $C$, is isolated in a sequence, the mining algorithm can reduce it to a loop for $C$. This allows the execution of $C$ as well as skipping $C$ (entering the loop 0 times), but also allows extra behavior by removing the upper bound of executions of $C$. Furthermore, the place between $B_1$ and $G$ has been lost. This is due to the fact the $G$ can not directly follow $B_1$, and thus there is no causal dependency $B_1 \rightarrow G$ in the log. Due to this fact, the implicit non-free-choice information is lost. Testing the log as presented by Van Dongen et al. [12] for their pattern recognition analysis reveals the same result, and therefore ILP mining is unable to reconstruct non-free-choice constructs in at least certain cases, if ever.

Figure 5.12: Mined net of incomplete log

The maximization with arc minimization and the arc minimization strategies are unable to reconstruct the choice pattern. Instead, a parallel execution construct is introduced, which leaves remaining tokens in the net when replaying the log. This does not influence the event fitness, but increases the allowed behavior. Furthermore, it is interesting to see that the invisible task construct is not reduced to a loop. This is caused by the first problem, as the loop contained a choice pattern whether or not to execute $C$.

The maximization with arc minimization strategy is also unable to reconstruct the loop pattern properly. The introduction of an extra place causes a deadlock, such that $E$ is never able to fire. The example again also shows that the naïve approach simply does not work as it introduces many arcs (grey) to transition $A$ that immediately cause a deadlock.

The arc minimization strategy also shows that the allowed behavior in the log increases significantly, as many tokens remain when replaying the log that can be used to execute extra transitions. Trivial examples are the arcs from transition $A$ (grey) that add many tokens to compensate for the introduction of parallelism instead of choices, of which many paths are not executed during log replay. Also the place $x = \{E\}, y = \{F\}, c = 0$ is unbounded. It sounds strange that the arc minimization strategy introduces many arrows, however the net shown in Figure 5.9 is not executed as a single ILP problem. Therefore the property is local and does hold for each place. The local number of arcs for a transition are not even taken into account. But also when executing the problem as a single ILP problem may find a solution that contains more arcs then the result of for example the place reduction strategy shown in Figure 5.8. This is due to the fact that duplicate places are removed in a postprocessing step. Otherwise, the net in Figure 5.8 would contain 8 more places and 37 more arcs.

The place reduction and the relaxation strategy are both able to reconstruct the same constructs as could be reconstructed using the original ILP for causal dependency [24] strategy.

Lastly, again it is made clear that the combination of arc minimization and place reduction leads to useless models. The only constructs that can be reconstructed using this strategy are loops and (too much) parallelism.

A sidenote has to be made on the parallelism, as it requires the log to be complete, i.e. all possible interchanges of the parallel branches must be present. For example, removing the trace $\langle A, B_2, C, H, D, F, I_2, J \rangle$ from the log, which states that $H$ can occur between $C$ and $D$, would result in the net as shown in Figure 5.12.
5.2.4 Overall comparison

Taking all aspects into account, the place reduction technique is recommended to decrease the size of the resulting nets. Some post-processing could be done to remove any end-places, i.e. places without outgoing arcs (especially after transitions near the end of the process, like after $E$ in Figure 3.8), as they do not restrict the behavior at all.

Depending on the interest of the end-user, arc minimization can also be a good tool to increase the readability of the net, as in several cases it performs better than the place reduction strategy, with respect to reducing the average connector degree or density. Especially when the calculations are made in a single ILP problem, the resulting net is a lot better readable. However, several nice properties are lost when using this strategy, like increased allowed behavior and resulting from this a loss of the ability to rediscover certain control-flow patterns.

Also the relaxation technique yields some improvements in readability and is able to reconstruct all control-flow patterns. However, as the event fitness value $f_{event} = 1$ for the resulting nets can no longer be guaranteed, this strategy is not always suitable.
Chapter 6

Conclusions

In this thesis the theory of regions based Integer Linear Programming process mining is addressed. A new plug-in for the ProM 6 process mining framework has been developed based on the work of Van der Werf, Van Dongen et al. [24] that focuses on ILP solver independence and extendibility, such that new ILP formulations can be easily implemented.

Through examples the performance of some ILP solvers is presented. The ILP Miner plug-in has both LpSolve, an open-source ILP solver, and the commercial IBM ILOG CPLEX available. Test results show that especially for larger problems the CPLEX solver is significantly faster. The end-users can now decide for themselves if they wish to pay for a speed boost or not.

Often, process mining algorithms focus on a certain aspect of the results, for instance the structural correctness, the fitness with respect to the log, or the most important parts of the process. The ILP mining approach so far focuses mainly on the fitness. It ensures the replayability of the entire log and has only minimal extra behavior. In order to improve this approach, a more balanced result with respect to the aforementioned aspects could increase the useability of the ILP Miner plug-in. For this reason several variations are presented on the ILP problem definition in order to increase the readability and noise abstraction of the resulting models. Some results based on real-life logs are presented and show a clear improvement of the readability. The results on noise reduction are not conclusive, but bid a starting point for further investigation.

6.1 Future work

The three main properties that make process mining results useful are the fitness of the original log, the structural correctness of the resulting model and the readability of the model. Balancing the fitness with respect to the readability has been the focus of this thesis. However, even though some work has been done on the structural correctness, like the empty after completion property, the balanced ILP mining strategy that combines fitness and readability with structural correctness does not yet exist. Other work on inhibitor and reset arcs [23] could also be combined to further increase the readability of the results without changing the semantics.

Since predicting a good balance between the three aspects mentioned before, it is vital to know the particular interests of the end-user. To this end, the current techniques should be extended to focus on certain aspects according to some simple user adjustable values. Optimally, these values could be changed after the mining procedure as this mining technique is still not particularly fast on real-life sized logs.
Bibliography


Appendix A

ILP Miner Plug-in Manual

The ILP Miner plug-in, implemented in the ProM 6 framework, standardizes the access to several variations of the ILP mining algorithm. In this manual we will show based on a simple example how to use the plug-in. It is however possible that a specific variant requires some special settings that are not described in this manual. We will therefore present a general guideline on how to act in case of this manual being insufficient.

After downloading and installing the ProM 6 framework, we must first make sure that the ILP Miner plug-in is present. For this reason we will first start the ProM 6 Package Manager that comes with the framework. After starting the Package Manager, we can select four tabs on the left hand side (see Figure A.1). On the ‘Up to date’ tab, a list of plug-ins is shown, of which the latest version is present in the framework. If the ILP Miner is not present in this list, we need to either install or update it. Find and select the ILP Miner on the ‘Out of date’ or the ‘Not installed’ tab as shown in Figure A.1.

After pressing the Install or Update button in the description on the right hand side, the Package Manager will download and install the plug-in as shown in Figure A.2. The ILP Miner plug-in has now moved to the ‘Up to date’ tab. Note that the LpSolve and the CPLEX plug-ins have also been automatically installed, as they are required for the ILP Miner plug-in.

Now that the plug-in has been installed, we can close the Package Manager and start the ProM 6 framework. When starting for the first time, no objects like logs or mining results will be present, as shown in Figure A.3.

In order to use the ILP Miner plug-in, we first need a log. After pressing the Import button in the top right corner, a file dialog will be shown as in Figure A.4. The ProM 6 framework can load both the old MXML log format as well as the new XES log format [14]. If you still need to convert your data to one of these formats, we refer you to the new XESame (Formerly known as XEStm) plug-in [9].

After the log has been fully loaded, it will be added to the object pool, as shown in Figure A.5, and we can perform various actions on it. If not done automatically, select the log from the object pool and press the ‘Play’ button (triangle pointing right) in the object details panel on the right hand side. ProM 6 will now show a list of plug-ins that can be applied to the log shown in the input list on the left hand side. Locate and select the ILP Miner 2 plug-in as shown in Figure A.6. We can see in the output list that the ILP Miner plug-in will produce a Petri net and a marking. Press the Start button at the bottom, in order to continue to the settings screen of the ILP Miner plug-in.

After starting the ILP Miner plug-in, first a number of settings has to be set. The first setting concerns the ILP solver to be used as shown in A.7. In Chapter 4 a comparison is presented between the different solvers. The standard solver, Java-ILP & LpSolve, is available to anyone and can be used right away. In order to use the CPLEX solver (both with Java-ILP or OPL), a license is required. When this solver is selected, the plug-in will automatically check if the license is correctly installed and provide support on how to correctly install it if this is not the case.

The second setting concerns the approach to be used when searching for places. Most variations
are discussed in Chapter 3, but as the plug-in has been built for easy implementation of new approaches, other variants may be available. We advise to read any documentation provided by the author (see the variant description) if it is unclear what the results will be. When you are not sure, we advise to select the standard ‘Petri Net’ variant as shown in Figure A.8.
Depending on the chosen variant, a number of extra settings may appear. For example, the ‘Petri Net (Variable Fitness)’ variant has three extra settings as shown in Figure A.9. The ‘Number of places’ concerns the method used to determine the number of places to be searched for, i.e. the basis representation [8, 16] (Not recommended), per causal dependency [24], or the various place reduction combinations (see Section 3.4). If the checkbox ‘Search for separate initial places’ is checked, the result will be a workflow net instead of a normal Petri net. These two settings apply to most variants. Lastly, the variable fitness approach has a slider on which the minimal fitness can be set.

Lastly, a set of extensions is presented varying per ILP variant. These extensions can be turned on to ensure certain properties on the ILP mining results.

When fully satisfied with the setting, pressing the Finish button will start the mining algorithm. The progress bar as shown in Figure A.10 shows the estimated progress whenever possible, as
mining may take quite some time depending on several properties of the log.

After the mining algorithm has finished, the results will be automatically displayed as in Figure A.6:
A.11. Process Mining using Integer Linear Programming
Figure A.9: ILP Miner additional settings

Figure A.10: ILP Miner progress
Figure A.11: ILP Miner result