Experimental Results on Process Mining Based on Regions of Languages

Robin Bergenthum, Jörg Desel, Christian Kölbl, and Sebastian Mauser

Department of Applied Computer Science,
Catholic University of Eichstätt-Ingolstadt,
{firstname.lastname}@ku-eichstaett.de

Abstract. In [2] we presented a process mining method based on the principle of separating behaviour recorded in an event log from not observed behaviour by so called regions of languages. This workshop paper describes an implementation of the method and shows experimental results. Furthermore some extensions improving the basic process mining algorithm are explained, implemented and analyzed by practical tests.

1 Introduction

In the past few years the analysis of business processes became more and more popular. Many of today's information systems record information about performed activities of workflows in log files, so called event logs. Not only classical workflow management systems and ERP systems, but also web services, middleware systems, embedded systems in high-tech equipment and many other information systems produce event logs. Different methods have been suggested to parse event logs carrying information about workflows to understand what is actually going on. Techniques extracting useful, structured information from the vast amount of information recorded in an event log are subsumed under the term *process mining*. In this paper we focus on constructing a process model, which matches the actual workflow of the recorded information system, from an event log. This prevalent aspect of process mining is known as process or control flow discovery. There are many process discovery techniques in literature (see e.g. [8, 7]), often implemented in the ProM framework [5]. In the following we consider process models given by Petri nets.

A typical event log records the executed activities of a workflow together with cases (initiating process instances) the activities belong to. Each case in a log is composed of activities ordered by their execution times. Thus, abstracting from further information, an event log essentially consists of a set of recorded cases each defining a sequence of activities. That means, a case can be seen as a word over the alphabet of activities and an event log can be seen as a language. With this point of view process discovery is similar to a well-known problem in Petri net theory, namely the reproduction of a language by a Petri net – the so called *synthesis problem up to language equivalence*.

Concerning the synthesis problem, since the 1980s, a lot of results and methods have been explored [1]. In a natural way well-known methods derived from the *theory of regions of languages* [1] can be applied to synthesize a Petri net from a language given by an event log as shown in [2]. In this paper we present an implementation of the most promising process discovery algorithm presented in [2]. This approach searches for so called separating regions to exclude behaviour not observed in the event log from recorded behaviour. Each separating region adds restrictions to a net so that it is still able to reproduce the given language but certain unwanted behaviour, a so called wrong continuation, is excluded. In detail this is done by first adding one transition for each activity, and then adding places (and arcs) given by separating regions to restrict the behaviour of the net step by step or let's say place by place. The search for a separating region excluding a wrong continuation is a linear programming problem. We show experimental results for this mining method in the paper.

On top of the implementation of this basic mining algorithm, we present, implement and test some extensions tuning the algorithm to practical needs. A long list of such extensions has been proposed in [3]. Guided by practical experiences with our new implementation of the new mining algorithm, we selected the most interesting ones of this list to be studied in detail in this paper. The main problems tackled by these extensions are overfitting, performance and ”spaghetti” process models. These problems are in particular linked to the fact that the basic process discovery approach of this paper is a precise one in the sense that it roughly speaking constructs a Petri net reproducing the words of the event log and having minimal additional behaviour (in particular, if each of the finitely many wrong continuations can be excluded by a separating region, then the net exactly represents the log).

In contrast, many classical process discovery approaches are imprecise. In order to be efficient in time and memory consumption and to keep the resulting process models small, these process models allow for much more additional behaviour as necessary – they overapproximate the given event log. Our precise approach overcomes many of the limitations of such classical imprecise approaches. These have problems with complex control flow structures such as non-free-choice constructs, unbalanced splits and joins, nested loops, etc., although in reality processes often exhibit such features. The second main problem of imprecise methods is their tendency to underfit the event log, i.e. the resulting model allows for much more behaviour without any indication to be reasonable real behaviour. These problems are resolved by the precise algorithm presented in this paper.

On the other hand, precise methods are often criticized for their tendency to overfit [7]. Of course a log is usually incomplete, i.e. there may be possible cases not occurring in the log. On the first glance this argues against precise approaches. But without additional information it is unclear, which possible cases are missing in the log. It is hardly possible to extract information about such missing cases solely from the event log. The overapproximation performed by most existing imprecise approaches seems to be quite arbitrary in many cases, at least hardly controllable and predictable. In contrast, when using the precise approach as a basis, it is possible to specifically introduce overapproximation in a targeted direction (on top of the precise solution) as shown in [3]. This makes the overapproximation reliable and configurable to address different needs of the user. These ideas are an interesting research field for us, but in this paper we do only rudimentarily consider such extensions introducing overapproximation to adequately deal with incomplete logs. Thus, the presented methods basically face the problem of overfitting, but the tests shown exemplarily in this paper seem promising anyway.

A particular problem with precise process mining methods is, that they may be inefficient in time and memory consumption. Although our basic mining algorithm has a polynomial runtime and memory consumption, this may not be sufficient for practical applications. Many existing imprecise methods are linear. But the experimental results in this paper show that our method works well for quite large logs. Therefore, so far we only worked on few extensions concerning the runtime of the mining procedure. But we plan to integrate some further extensions improving the runtime, especially dealing with modular aspects.

Lastly, the key drawback of a precise process discovery method is, that it usually generates a quite large process model. The number of components of the model – in our situation mainly the number of places and arcs of the Petri net – often has to be large in order to potentiate an exact reproduction of the event log. This is the real problem arising from the overfitting of precise algorithms. The size of the model has to be in such a range, that the model can easily be interpreted by the user. Practitioners and process analysts in industry are usually interested in concise and controllable reference models. ”Spaghetti”-models that can only be evaluated with computer support are mostly not satisfactory. The danger of mining ”spaghetti”-models is especially problematic in logs of unstructured processes in less restrictive environments [7]. Most of the extension of the basic mining algorithm presented in this paper deal with this problem. Of course there sometimes is a trade-off between simplifying the mined net and maintaining a preferably precise model. But since our basic mining algorithm already yields quite satisfactory nets as shown in this paper, we so far only implemented extensions preserving the precise character of the procedure. Imprecise extensions are an interesting field of further research. In order to get a compact model, also abstraction and visualization techniques for the mined model are an interesting future research topic [7].

It remains to mention, that the approach of using regions of languages for process mining was also picked up in a paper [9] that will be presented at the main ATPN2008 conference. This paper combines the ideas with the concept of introducing causal dependencies according to the alpha algorithm. In contrast to the mere alpha algorithm, through the region approach, it is guaranteed that the log is reproduced by the mined net and the places of the net are optimized in some sense (by integer linear programming methods). Although there are similarities to our approach, the two techniques are guided by completely different techniques (alpha algorithm vs. separating regions). In particular, the approach in [9] is still imprecise.

The paper is organized as follows. Section 2 provides the basic algorithm and its implementation, all necessary definitions and all important aspects for the practical use of the algorithm. In Section 3 we give an overview about the extensions we added to the basic algorithm and discuss their complexity. Section 4 shows experimental results.

2 Algorithm and Definitions

In this section we recall the mining method based on *separating regions* from [2] carried by a running example.

An *alphabet* is a finite set A. The set of all *strings (words)* over an alphabet A is denoted by A^*. A subset $L \subseteq A^*$ is called *language over* A. For a word $w \in A^*$, $|w|$ denotes the *length of* w and $|w|_a$ denotes the number of occurrences of $a \in A$ in w. Given two words v, w, we call v *prefix* of w if there exists a word u such that $vu = w$.

Definition 1 (Event log). *Let T be a finite set of* activities *and C be a finite set of* cases. *An* event *is an element of $T \times C$. An* event log *is an element of $(T \times C)^*$.*

Given a case $c \in C$ we define the function $p_c : T \times C \to T$ by $p_c(t, c') = t$ if $c = c'$ and $p_c(t, c') = \lambda$ else. Given an event log $\sigma = e_1 \dots e_n \in (T \times C)^*$ we define the process language $L(\sigma)$ of σ *by $L(\sigma) = \{p_c(e_1) \dots p_c(e_i) \mid i \leq n, c \in C\} \subseteq T^*$.*

For better understanding we take the running example log from [2], where each letter stands for an activity and the number of an event refers to the case the particular activity belongs to (Example 1). Here *abbe* is the word the case number *1* mirrors having *a*, *ab* and *abb* as prefixes. Since a log has to define a prefix-closed language the prefixes also belong to the process language (every prefix is a series of activities that actually happen during the case).

event log (activity,case):
(a,1) (b,1) (a,2) (b,1) (a,3) (d,3) (a,4) (c,2) (d,2) (e,1) (c,3) (b,4) (e,3) (e,2) (b,4) (e,4)
process language:
a ab abb *abbe* ac acd *acde* ad adc *adce*

Example 1.

The letters (activities) T occuring in the log build up the transition-set (of a Petri net) without any restrictions. To restrict the behaviour of the transitions we have to add places, that prohibit wrong words but still permit the words of the process language $L(\sigma)$ of the log. We call such places *feasible places*[2].

Definition 2 (Feasible place). *Let (N, m_p), $N = (\{p\}, T, F_p, W_p)$ be a marked p/t-net with only one place p (F_p, W_p, m_p are defined according to the definition of p). The place p is called* feasible (w.r.t. $L(\sigma)$)*, if $L(\sigma) \subseteq L(N, m_p)$, otherwise* non-feasible.

The aim is to add as many feasible places as necessary so that the given net represents the process language with minimal additional behaviour. Feasible places are defined by so called *regions* of the language. A region can be seen as a tuple or vector $\mathbf{r}$ of natural numbers, where the first entry represents the initial marking of the corresponding place and the other entries define the consumption and production of tokens by the transitions for this place. The set of feasible places is given by the set of places corresponding to regions [2].

Definition 3 (Region). *Denoting $T = \{t_1, \dots, t_n\}$ the activities of an event log σ, a* region *of $L(\sigma)$ is a tuple $\mathbf{r} = (r_0, \dots, r_{2n}) \in \mathbb{N}^{2n+1}$ satisfying for every $wt \in L(\sigma)$ ($w \in L(\sigma), t \in T$):*

$$(*) \qquad r_0 + \sum_{i=1}^{n} (|w|_{t_i} \cdot r_i - |wt|_{t_i} \cdot r_{n+i}) \geq 0.$$

Every region $\mathbf{r}$ of $L(\sigma)$ defines a place p_r via $m_0(p_r) := r_0$, $W(t_i, p_r) := r_i$ and $W(p_r, t_i) := r_{n+i}$ for $1 \leqslant i \leqslant n$. The place p_r is called corresponding place to $\mathbf{r}$.

Now we know, that every place we add to the mined net (only feasible places) does not restrict the process language in any way, and by the notion of regions it is possible to translate the task of computing feasible places into the task of computing non-negative integer solutions of a homogenous inequation system $\mathbf{A}_{L(\sigma)} \cdot \mathbf{r} \geq \mathbf{0}$ defined by the (*)-inequations.

The inequation system of Example 1 would look like Table 1. With this inequation system we ensure, that only feasible places are added w.r.t. the given process language. As we can see, the inequations for *acde* and *adce* coincide so that the matrix actually has one row less (for a word wt only the Parikh image of w is considered).

a	$r_0 - r_6$	≥ 0
ab	$r_0 + r_1 - r_6 - r_7$	≥ 0
abb	$r_0 + r_1 + r_2 - r_6 - 2r_7$	≥ 0
abbe	$r_0 + r_1 + 2r_2 - r_6 - 2r_7 - r_{10}$	≥ 0
ac	$r_0 + r_1 - r_6 - r_8$	≥ 0
acd	$r_0 + r_1 + r_3 - r_6 - r_8 - r_9$	≥ 0
acde	$r_0 + r_1 + r_3 + r_4 - r_6 - r_8 - r_9 - r_{10}$	≥ 0
ad	$r_0 + r_1 - r_6 - r_9$	≥ 0
adc	$r_0 + r_1 + r_4 - r_6 - r_9 - r_8$	≥ 0
adce	$r_0 + r_1 + r_4 + r_3 - r_6 - r_9 - r_8 - r_{10}$	≥ 0

Table 1. In the first inequation, a consumes r_6 tokens from the initial marking r_0. At ”ab” we have the initial marking r_0, r_1 tokens produced by a and $r_6 + r_7$ tokens consumed by a and b, ...

The next step we have to make is the prohibition of words, that are not in the process language $L(\sigma)$. For this we will ”invent” the wrong continuations. Wrong continuations are kind of an opposite concept to feasible places.

Definition 4 (Wrong Continuation). *Let w be a word of a process language $L(\sigma)$ and t be an activity of the event log σ. A word $v = wt$ is a wrong continuation if $wt \notin L(\sigma)$.*

Although there are infinitely many words in the complement of $L(\sigma)$, we only have to regard a finite set of them, the wrong continuations. This is because prohibiting a wrong continuation wt every word wtv is prohibited too (prefix closure). Using Definition 4 we are able to build regions w.r.t. wrong continuations wt which prohibit all wrong words (not occurring in the log) wtv. Such regions are called *separating regions*.

Definition 5 (Separating region). *Let $\mathbf{r}$ be a region of $L(\sigma)$ and let wt be a wrong continuation. The region $\mathbf{r}$ is a* separating region (w.r.t. wt) *if*

$$(**) \qquad r_0 + \sum_{i=1}^{n} (|w|_{t_i} \cdot r_i - |wt|_{t_i} \cdot r_{n+i}) < 0.$$

Given a wrong continuation wt, we add one inequation $\mathbf{b}_{wt} \cdot \mathbf{r} < \mathbf{0}$ given by (**) to the inequation system $\mathbf{A}_{L(\sigma)} \cdot \mathbf{r} \geq \mathbf{0}$ and solve the resulting system. Each solution yields a separating region w.r.t. wt defining a feasible place. Step by step, for each wrong continuation, we compute one such place (if it exists) and add it to the constructed net. That means, if there is a nonnegative integer solution of the inequation system $\mathbf{A}_{L(\sigma)} \cdot \mathbf{r} \geq \mathbf{0}$, $\mathbf{b}_{wt} \cdot \mathbf{r} < \mathbf{0}$, then a feasible place prohibiting the wrong continuation wt is added (otherwise no place is added), and we have to check the next wrong continuation. It is possible that a calculated place for a wrong continuation already blocks another or even a lot of other wrong continuations. So before solving the inequation system in a step of the algorithm, it is checked whether the considered wrong continuation is already prohibited by previously added places. In this case the step is skipped.

As an example the inequation $\mathbf{b}_{wt} \cdot \mathbf{r} < \mathbf{0}$ for the wrong continuation *abc* in our running example would be:

$$r_0 + r_1 + r_2 - r_6 - r_7 - r_8 < 0$$

With an implementation of the presented method we obtained the net in Figure 1 from the log of Example 1.

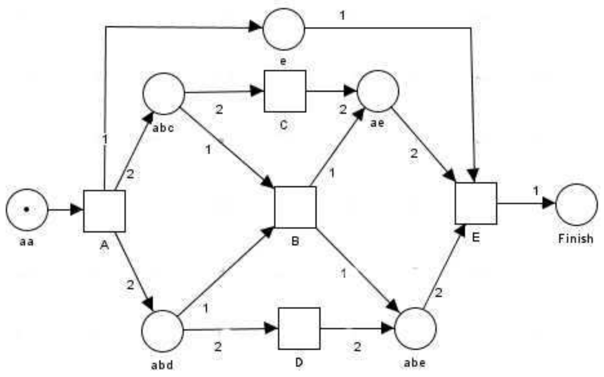

Fig. 1. Mined net. The words under the places are the wrong continuations for which the places were computed.

We implemented the algorithm within a diploma thesis at our Department of Applied Computer Science. The implementation comprehends a self coded version of the *Simplex algorithm* [6] with two primal steps to solve the inequation systems. Since the inequation systems are homogeneous we can use the standard Simplex searching for rational solutions. These can be multiplied by the common denominator of the components of the solution vector to get an integer solution. Furthermore we have an interface to work with the MXML fileformat (used in the toolset ProM [5]).

The implemented algorithm is exponential in the worst case, because of the Simplex algorithm. But there are also rational linear programming solvers, which have polynomial runtime, such as the method of Karmarkar. Using such solver the whole mining algorithm has polynomial time consumption, since there are at most $|L(\sigma)| \cdot |T|$ wrong continuations, i.e. steps of the algorithm. But probabilistic and experimental results show that the Simplex algorithm has a faster average runtime [6] than existing polynomial solver. Thus, we decided to use the Simplex in our implementation, but nevertheless we say that our basic mining algorithm has polynomial runtime.

In [3] we gave an overview of heuristics and extensions that could be important for the practical use of the implemented mining algorithm especially during the processing phase to gain performance and better readable nets. Since this list was very long we needed to decide which extensions and improvements could be the most important ones. For this we played with some logs distributed with the ProM Framework [5] and other small applied or artificial examples. We performed further tests in a collaborative project with Prof. Dr. Andreas Harrer (Professorship of Computer Science at the KU-Eichstätt). There we used the mining algorithm on logs recorded by e-learning systems. Past these tests and applications we figured that the size of the developed nets and the performance to develop was capable already for the basic mining algorithm. So we decided to implement mainly precise extensions with the main focus on simplifying the mined nets and avoiding too complicated structures.

3 Improvements

We distinguish in extensions which preserve the polynomial runtime of the basic algorithm and extensions which require exponential runtime due to the need of integer linear programming methods. As an integer linear programming solver we used the lpsolve 5.5.0.12 Java package [4].

3.1 Polynomial extensions

Finding cycles The implemented method to find cycles is a simple one. It examines each case c from an event log σ. We denote $c = c_1c_2...c_n$. Within one case c the method searches, if there occurs a sequence a of two activities more that once in the case c, i.e. $a = c_{i-1}c_i = c_{j-1}c_j, i \neq j$. In this case the sequence a is assumed to be a candidate to be part of a cycle. All such a are stored, because to allow cycles these a need not be prohibited. That means, if for a wrong continuation $wt \notin L(\sigma)$, $w \in L(\sigma)$, there holds $wt = w'a$ for one of the stored series of activities a, then wt is not considered as a wrong continuation, i.e. no place separating wt is computed.

By not marking this cycle candidate as a wrong continuation we allow the respective sequences, but we do not enforce the introduction of a cycle in the mined net. That means, the cycle may be prohibited by another place, but then this is a hint, that there actually is no cycle, and thus prohibiting the cycle in this way is a reasonable choice.

Order of the wrong continuations This heuristic is quite difficult to generalize in terms of usage. Since the algorithm works step by step considering one wrong continuation in each step, it is dependant on an ordering of the wrong continuations. We tried to improve the runtime of the algorithm and/or minimize the number of places by sorting the wrong continuations in the most practical order. We use ordering principles that are based on our experiences so far. One such principle is to consider at last wrong continuations only consisting of one element. We also tried advanced systematics like testing several (sub-)orderings leading to different solution nets.

Deleting implicit places To reduce the number of places after the actual mining algorithm, we implemented post-processing methods to delete implicit places. Implicit places can be deleted without changing the behavior of the net. We implemented three methods: A very simple and efficient one compares places pairwise to check if one place is less restrictive than the other one. The less restrictive place is of course redundant. A second method uses linear programming techniques to see if a place is less restrictive than a linear combination of the other places. Such place is also implicit. A third method checks all places in a certain order, if they are implicit. The third approach is no more polynomial.

Objective function Since there are several possible solutions of an inequation system defined by a wrong continuation, there are different possibilities which place is actually added in a step of our algorithm. The choice of a concrete solution in one step can be guided by an objective function. Since the Simplex algorithm is able to compute an optimal solution of the inequation system w.r.t. a linear objective function, considering such function in order to, e.g., minimize the arc weights and the initial marking of the resulting place does not significantly decrease the performance of the whole algorithm. For example to minimize the sum of ingoing arc weights and the initial marking of the computed place, we can consider the linear programming problem $!min \sum_{i=0}^{n} r_i$, $\mathbf{A}_{L(\sigma)} \cdot \mathbf{r} \geq \mathbf{0}$, $\mathbf{b}_{wt} \cdot \mathbf{r} < \mathbf{0}$. But by using the rational Simplex solver adding an objective function to the linear programming problem is only a heuristic. The optimization of the objective function by the rational solver means the calculation of the optimum by using rational numbers. The solution is then multiplied by the common denominator to get an integer solution of the homogeneous inequation system defining a separating region. So this separating region must not necessarily persist as the actual optimum, but in practice this heuristic often leads to a near optimal solution.

3.2 Exponential Extensions

Objective function An exact method is to use an integer solver with the branch and bound technique to solve the inequations system equipped with a linear objective function. As this is not implemented in our algorithm we use lpsolve. With this way of computing a separating region we get the exact optimal integer solution, but at the cost of runtime, since the branch and bound technique needs exponential time consumption.

Classes of nets In our case we implemented a method to build a net with no arc weights. This can be done by adding the restrictions $r_i \leq 1$ for $i = 1 \ldots 2n$ to the inequation system, such that the region vectors only consist of 0 or 1 values. In this case we cannot use a rational solver, because we are not able to multiply the solutions by anything. Well, we actually would be able, but it would not necessarily result in a solution of our inequation system, because the inequation system is no longer homogenous. Therefore, we here again use the integer solver of lpsolve.

Outlook Using integer linear programming, it is also possible to minimize the number of places of the net or to compute a ”best solution” in the case that only a fixed maximum number of places is allowed in the net. Such extensions reducing the number of places are interesting, but may lead to problems in performance. We are also able to use more complicated net classes, especially correctness properties for the mined nets can be postulated, e.g. the claim that the net has to be empty after finishing a case (this is part of the well-known soundness property).

4 Tests

We present practical tests to examine the quality of the mined nets and runtime tests.

4.1 Quality Tests

In the tests we tried to figure out, where the explained mining algorithm does a better job than well known algorithms like the alpha algorithm, or the alpha++ algorithm, an extension of the alpha algorithm mining certain implicit dependencies (in particular non-free choice constructs). Both algorithms are implemented in the ProM Framework [5]. The log files used in the tests are mostly taken from the ProM distribution. In this case we refer to the names of the logs. We also checked other algorithms mining Petri nets, where in some cases the results were not satisfactory, e.g. the region miner implemented in ProM, and in some cases it would be interesting to compare the algorithms in detail, e.g. the genetic miner, heuristic miner or multiphase miner. We restrict us to the alpha and alpha++ algorithm in this section because the alpha algorithm is the best known mining algorithm. A second aim of the tests is to analyze the extensions presented in the last section.

In a first test we examined a log file of a workflow with a mutual exclusion of two subprocesses (pn_ex_02.xml). Thereby, the alpha algorithm builds a Petri net that delivers a deadlock after the second transition (Figure 2). The alpha++ algorithm modeled a mutual exclusion, but it additionally introduced optional loops of the sub-processes and allows a direct termination of the main process without executing the sub-processes (Figure 3). Thus, the actual workflow is not displayed. Our basic algorithm correctly mined the mutual exclusion without unnecessary abstraction (Figure 4).

In an example of a log with non-free choice behaviour (sw_ex_14.xml), both alpha algorithms delivered satisfactory solutions, and our method generated a similar representation of the process. Next, we considered a log of a workflow with optional tasks (ic_ex_01.xml), where transitions in the representing Petri Net are able to fire but doesn't have to. While this is not solvable by the other methods our basic mining approach gives us the correct solution as a net. Another tested workflow log was a trigger on expire task (sw_ex_04.xml) that was embedded in a very difficult control flow. In this workflow there exist tasks to prematuraly terminate activities. Also here our method reflects the real workflow better than the alpha algorithms. Similar to the optional tasks are terminating tasks, where a process should be able to terminate in every state by executing the terminating activity. So if there are cases like $ABCD$ or $ACBD$ then ABD or ACD should also be able, when D represents the termination of the process. This problem was only well solved by our algorithm.

We now test the described extension to find cycles. As mentioned this means, that we tell the basic algorithm that detected possible cycles are not wrong, but not that they are explicitly desired. The calling for cycles as mandatory desired behaviour would also be possible (but it is not implemented so far) by simply adding corresponding homogeneous inequations to the inequation systems (similar to setting the arc weights to 1). The next considered workflows contained loops and we tried two different approaches of our method: The one with the finding cycles expansion and the other without.

The first test case was a mutual exclusion inside a loop (pn_ex_11.xml). Both algorithms solved the problem, while the second one counted the maximal number of repetitions of the loop appearing in the log and set according arc weights (Figure 5). The counting place p10 causes the net not to be sound. However, the first one was able to find the cycle and represented it in the sound Petri net shown in Figure 6. The alpha

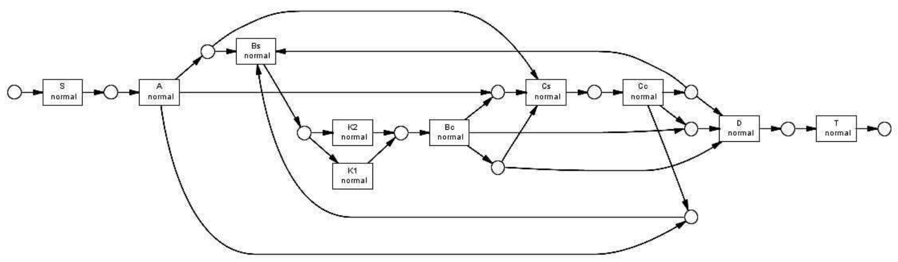

Fig. 2. Mutual exclusion of two sub-processes solved by the alpha algorithm

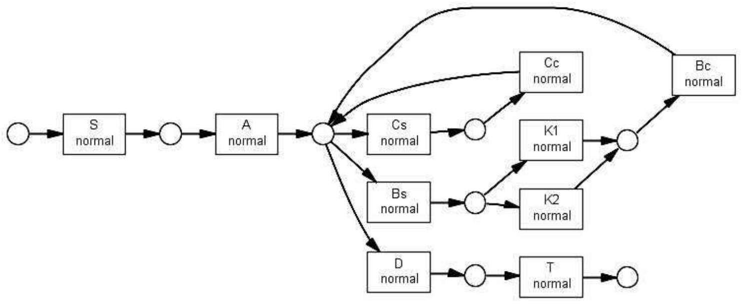

Fig. 3. Mutual exclusion of two sub-processes solved by the alpha++ algorithm

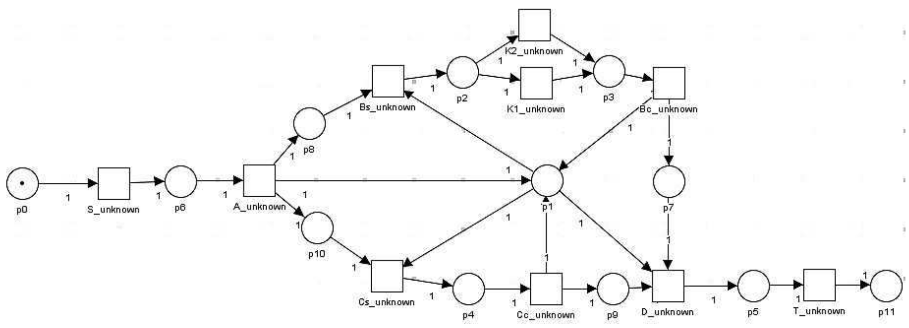

Fig. 4. Mutual exclusion of two sub-processes solved by our algorithm

and alpha++ algorithm couldn't solve the problem (certainly also because of the mutual exclusion).

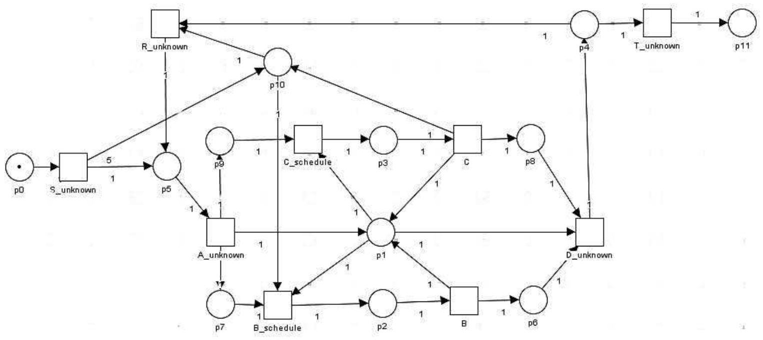

Fig. 5. Mutual exclusion of two sub-processes in a loop solved by our algorithm without the finding cycles expansion.

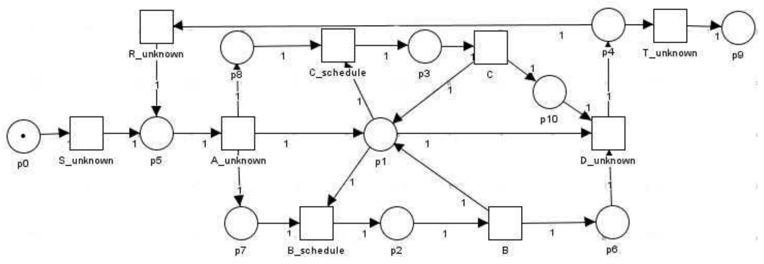

Fig. 6. Mutual exclusion of two sub-processes in a loop solved by our algorithm with the finding cycles expansion.

A pure loop based test case was a workflow with a 2/3 loop (pn_ex_08.xml) (Figure 7, 8), where one has the alternative between a 2- or a 3-loop construct. This test case was correctly solved by each algorithm.

A third setting with a loop is from the experiments with the e-learning logs, where we had a self loop of an activity (actually an optional task) and a difficult cycle. The alpha algorithm collapsed at both problems just as well as the alpha++ algorithm, while the second one delivered a not so bad solution (Figure 9). The accurate solution found by our algorithm can be seen in Figure 10. The net is not sound because we used no inivisible task.

Another problem that cannot be solved by the alpha algorithms was a workflow having a loop that had to be executed at least one time (sw_ex_05.xml). Even with such "at least one"-loop our mining algorithm has no problems.

After testing a lot of different loop constructions we are confident, that our approach is able to very well handle loops in a workflow-log.

Since we are working with arc weights (which are often undesired), we wanted to see what solutions we will get, if we are leaving arc weigths, i.e. demanding $w \leq 1$. In most examples considered so far our algorithm did not introduce arc weights anyway, consequently, to see effects, for this approach we used our running example (Example

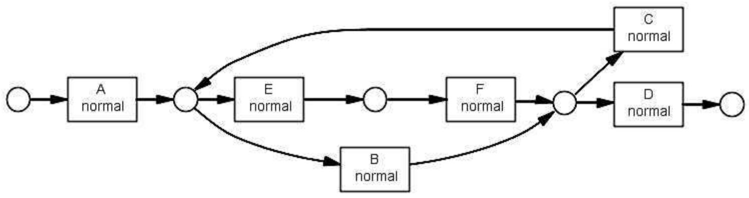

Fig. 7. 2/3-loop solved by Alpha++ Algorithm.

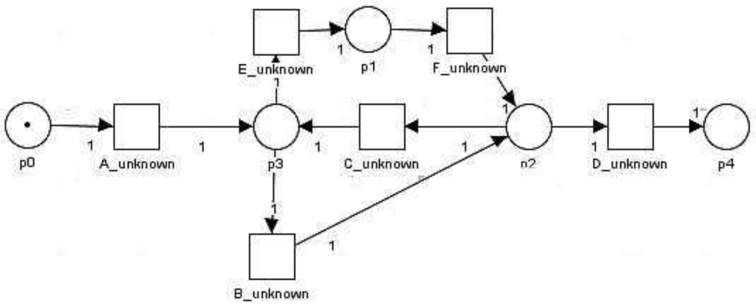

Fig. 8. 2/3-loop solved by our algorithm.

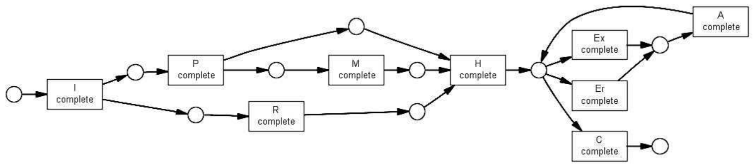

Fig. 9. Optional task and cycle solved not correctly by the alpha++ algorithm.

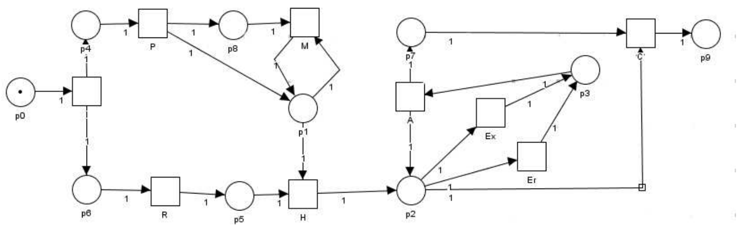

Fig. 10. Optional task and cycle solved correctly by our algorithm

1). The alpha algorithm produces a net, that allows an infinite firing of B in any state, similar to the alpha++ algorithm (Figure 11). This is definitely not desired. With unrestricted arc weights our algorithm delivers a nice net as seen before (Figure 1). With the $w \leq 1$ restriction the mined net (Figure 12) is a bit problematic and not that readable due to the fact that the desired behavior can hardly be represented without arc weights. The net still reproduces the desired language, but unfortunately with some additional behavior, but best as possible (also that more than one place is initially marked may be seen as problematic).

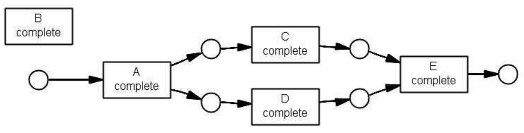

Fig. 11. Running example synthesized by the alpha algorithm.

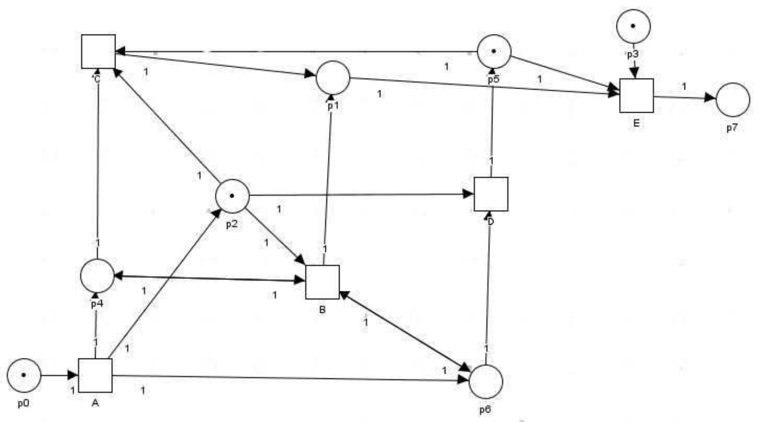

Fig. 12. Running example solved by our algorithm with $w \leq 1$.

We also implemented a heuristic to choose a good order of wrong continuations. We ran a test with the log of the running example and we saved one place (Figure 13).

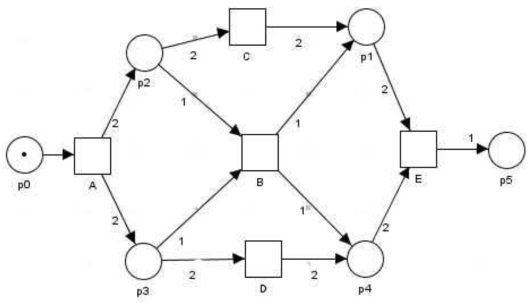

Fig. 13. Running example solved by sorting the wrong continuations in a practical order.

This place could also be deleted by postprocessing algorithms deleting implicit places. But only the inefficient third method described in the last section is able to find this implicit place. In general, the presented basic mining algorithm only in rare cases computes implicit places, in particular when choosing a practical order of the wrong continuations. We never had to use one of the three implemented post-processing methods in one of the previous examples.

We also tested the extensions, where the choice of the introduced places is guided by an objective function. Although it seems to be desirable to, e.g., minimize arcs connected to places, in our examples these extensions only rarely yielded more compact and better readable nets.

Finally, we can say that a practical order of wrong continuations and the heuristic of the cycle calculation distinctly improved the basic algorithm. Usually, when we use the finding cycles expansion, arc weights rarely occur, but if there are still arc weights, then from our experiences they bring a big simplification in the sense that a mined net without arc weights is very clumsy and unclear or incorrect. Thus, in our opinion, arc weights may be helpful. An objective function only brought use in few cases. Therefore, in the standard setting of our mining algorithm no objective function is used, arc weights and the calculations of cycles are used just as well as the approved order (w.r.t. the size) of the wrong continuations.

4.2 Runtime Tests

For the runtime tests we used the standard setting of our algorithm (w.r.t. the applied extensions). We realized the tests on some log-files distributed with the ProM Framework. You can see the results in the table below. As a benchmark we used experimental results presented for the algorithm in [9], because this algorithm is on the one hand similar to our algorithm and on the other hand based on the alpha algorithm (see Introduction).

The important sizes in this table are ”# trans.” (transitions), as it determines the number of variables per inequation system, and ”# constr.” as it defines the number of inequations of the inequation system. With the quotient # events / # cases you get the average length of a case. Since cases may generate equal words, we specify the number of different cases by the column ”# words”.

Our algorithm is overall faster in calculation speed (pay attention that different computers are used), and has a similar growth as the benchmark. Although the runtime of our algorithm is a critical point, we think it works quite well, especially if considering that, until now, no performance oriented programming on the implementation took place (but it will be done in the future, e.g. using Dual Simplex). The complexity seems approximately linear in the number of words (but this may not be the case in general). The dependency of the computation time from the number of transitions seems more problematic. Altogether, we are able to solve a lot of problems, even bigger ones, if we run the calculation over a longer period. Thus, also large log files may be mined, e.g., overnight.

5 Future Work

Our next steps will be the development, implementation and test of further extensions discussed in [3] to our mining approach (see the Introduction). We intend to soon release a mining tool freely available for personal use.

log	**# trans.**	**# cases**	**# events**	**# words**	**# constr.**	**runtime**[1] **(hh:mm:ss.sss)**		benchmark time[2] (hh:mm:ss.sss)
a12f0n00_1	12	200	1236	5	19	0.046		0.406
a12f0n00_2	12	600	3696	5	19	0.047		0.922
a12f0n00_3	12	1000	6154	5	19	0.047		1.120
a12f0n00_4	12	1400	8666	5	19	0.049		1.201
a12f0n00_5	12	1800	11146	5	19	0.054		1.234
a22f0n00_1	22	100	1833	99	901	4.248		1:40.063
a22f0n00_2	22	300	5698	291	2091	12.360		5:07.344
a22f0n00_3	22	500	9463	476	2823	18.302		7:50.875
a22f0n00_4	22	700	13215	660	3488	20.459		10:24.219
a22f0n00_5	22	900	16952	836	4052	44.065		12:29.313
a32f0n00_1	32	100	2549	100	1633	19.624		32:14.047
a32f0n00_2	32	300	7657	300	3815	1:27.904		1:06:24.735
a32f0n00_3	32	500	12717	500	5368	2:11.207		1:46:34.469
a32f0n00_4	32	700	17977	700	6721	2:32.454		2:43:40.641
a32f0n00_5	32	900	23195	900	7854	1:57.218		2:54:01.765
a42f0n00_1	42	100	3269	100	2723	1:45.781		n/a
a42f0n00_2	42	300	9794	300	7443	8:17.308		n/a
a42f0n00_3	42	500	16369	500	11812	34:34.859		n/a
a42f0n00_4	42	700	22817	700	15704	45:51.791		n/a
a42f0n00_5	42	900	29169	900	19263	43:08.843		n/a

References

1. E. Badouel and P. Darondeau. Theory of Regions. In *Petri Nets, LNCS 1491*, pages 529–586, 1996.
2. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In *BPM, LNCS 4714*, pages 375–383, 2007.
3. R. Bergenthum, R. Lorenz, and S. Mauser. Towards Applicability of Language Based Synthesis for Process Mining. In *Algorithmen und Werkzeuge für Petrinetze (AWPN)*, pages 45–50, 2007.
4. Lp solve reference guide. http://lpsolve.sourceforge.net/5.5/.
5. Process mining group eindhoven technical university: Prom-homepage. http://is.tm.tue.nl/ cgunther/dev/prom/.
6. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.
7. W. M. P. van der Aalst and C. W. Günther. Finding structure in unstructured processes: The case for process mining. In *ACSD, IEEE Computer Society*, pages 3–12, 2007.
8. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.
9. J. van der Werf, B. van Dongen, C. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. In *ICATPN, LNCS 5062*, pages 368–387, 2008.

[1] These calculations were performed on a 2.66GHz Core 2 Duo E6750 machine running Java 1.6 (no Threads used). The memory consumption never exceeded 256 MB. The program used LpSolve 5.5.0.12.

[2] These calculations were performed on a 3GHz Pentium 4 machine running Java 1.5. The memory consumption never exceeded 256 MB. The program used LpSolve 5.5.0.10.