Tuning Alignment Computation: An Experimental Evaluation

Sebastiaan J. van Zelst, Alfredo Bolt, and Boudewijn F. van Dongen

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{s.j.v.zelst, a.bolt, b.f.v.dongen}@tue.nl

Abstract. Conformance checking aims at assessing whether a process model and event data, recorded in an event log, conform to each other. In recent years, alignments have proven extremely useful for calculating conformance statistics. Computing optimal alignments is equivalent to solving a shortest path problem on the state space of the synchronous product net of a given Petri net and event data. State-of-the-art alignment-based conformance checking implementations exploit the A^*-algorithm, a heuristic search method for shortest path problems, and include a wide range of parameters that likely influence their performance. In this paper, we present an exploratory empirical evaluation of parametrization of the A^*-algorithm used in alignment computation. Our initial results show that the performance of alignment computation greatly depends on adequate parametrization of the underlying search algorithm.

Keywords: Process Mining, Conformance Checking, Alignments

1 Introduction

Process mining [2] is concerned with the analysis and improvement of business processes, based on real process execution data stored in *event logs*. The field consists of three main branches: *process discovery*, *conformance checking* and *process enhancement*, where conformance checking, i.e. this paper's focus, aims at assessing to what degree the behaviour described by a process model is in line with behaviour captured in an event log.

Early work in conformance checking focused on replaying behaviour in an event log within a Petri net by "playing the token-game" [12]. These techniques however often yield ambiguous and/or unpredictable results. Recently, *alignments* were introduced [3,5], which rapidly developed into the de-facto standard in conformance checking. When computing alignments, we convert a given process model, together with the behaviour in an event log, into a *synchronous product net* and subsequently solve a shortest path problem on the corresponding state space. The major advantage of this approach is the fact that deviations and/or mismatches are quantified in an exact, unambiguous manner.

The process mining tool-kit ProM [15] has proven valuable for the implementation and evaluation of several process mining techniques, for both research and industry. Alignments are implemented in ProM, and, the current implementation can be considered state-of the-art. The implementation uses A^* [7] as an underlying solution to the shortest path problem and provides several parametrization options. Examples of such parameters, which we assess in this paper, are the type of heuristic used and the second order sorting criteria of the underlying priority queue. Studies towards the effect of these parameters, in context of alignment computation, are however missing.

In this paper we formalize the A^*-algorithm and associated parametrization in terms of alignment computation, in line with the underlying implementation in ProM. As such, this paper acts as a high-level description of the current implementation. Using the ProM-based extension of RapidMiner (`http://rapidminer.org`), i.e. RapidProM [4] (`http://rapidprom.org`), we study the impact of the parametrization of the A^*-search algorithm on the average computation time and memory usage. Our experiments show that for the class of models tested, i.e. Petri nets without loops and invisible transitions, there is a clear trade-off in terms of computation time and memory usage when using different heuristic functions. Additionally, the second-order sorting criterion of the priority queue used internally greatly impacts performance.

The remainder of this paper is organized as follows. In Section 2, we present preliminaries. In Section 3, we discuss related work. In Section 4, we explain how to find optimal alignments using A^*. In Section 5, we discuss parametrization of A^* algorithm. In Section 6, we evaluate the proposed parametrization. Section 7 concludes the paper.

2 Preliminaries

In this section we present preliminary concepts needed for a basic understanding of the paper. We assume the reader to be reasonably familiar with concepts such as partial functions, sets, bags, sequences and Petri nets.

For each countable set X we assume the existence of set $I_X = \{1, 2, ..., |X|\}$ and bijective function $\iota \colon I_X \to X$, i.e. each set is indexed. We write $x \in X$ to denote any arbitrary element in X and we let $x_i \in X$ denote that specific element $x \in X$ for which $\iota(i) = x$ with $i \in I_X$. We denote the set of all possible multisets over set X as $\mathcal{B}(X)$. We denote the set of all possible sequences over set X as X^*. The empty sequence is denoted $\langle\rangle$. Concatenation of sequences σ_1 and σ_2 is denoted as $\sigma_1 \cdot \sigma_2$. Given tuple $\boldsymbol{x} = (x^1, x^2, ..., x^n)$ of Cartesian product $X^1 \times X^2 \times ... \times X^n$, we define $\pi^j(\boldsymbol{x}) = x^j$ for all $j \in \{1, 2, ..., n\}$. We overload notation and extend projection to sequences. Given sequence $\boldsymbol{\sigma} \in (X^1 \times X^2 \times ... \times X^n)^*$ of length k with $\boldsymbol{\sigma} = \langle (x^{1^1}, x^{2^1}, ..., x^{n^1}), (x^{1^2}, x^{2^2}, ..., x^{n^2}), ..., (x^{1^k}, x^{2^k}, ..., x^{n^k}) \rangle$, we have $\pi^j(\boldsymbol{\sigma}) = \langle x^{j^1}, x^{j^2}, ..., x^{j^k} \rangle \in X^{j*}$, for all $j \in \{1, 2, ..., n\}$. Given a sequence $\sigma = \langle x^1, x^2, ..., x^k \rangle \in X^*$ and a function $f : X \to Y$, we define $\pi^f(\sigma) = \langle f(x^1), f(x^2), ..., f(x^k) \rangle$. Given $Y \subseteq X$ we define $\downarrow_Y \colon X^* \to Y^*$ recursively with $\downarrow_Y (\langle\rangle) = \langle\rangle$ and $\downarrow_Y (\langle x \rangle \cdot \sigma) = \langle x \rangle \cdot \downarrow_Y (\sigma)$ if $x \in Y$ and

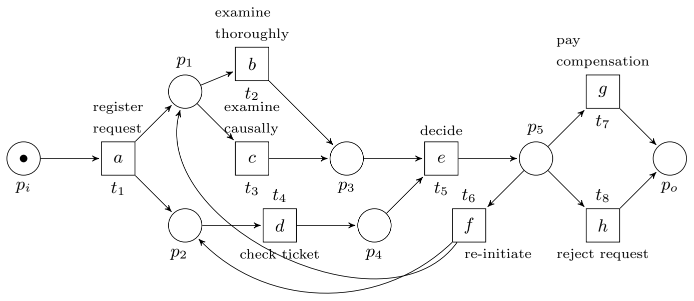

Fig. 1. Example labelled Petri net N_1

$\downarrow_Y (\langle x \rangle \cdot \sigma) = \downarrow_Y (\sigma)$ if $x \notin Y$. We write $\sigma_{\downarrow_Y}$ for $\downarrow_Y (\sigma)$. We let $\epsilon \in \mathbb{R}_{>0}$ denote a positive infinitesimal with $\epsilon > 0$, $n\epsilon > m\epsilon$ if $n > m$ and $\lim\limits_{n \to \infty} (n\epsilon) < 1$.

An introduction to event logs is outside of the scope of this paper, hence, we refer to [2]. We define an event log as a multiset of sequences of business process activities. Let $\mathcal{A}$ denote the universe of business process activities, an event log L is defined as $L \in \mathcal{B}(\mathcal{A}^*)$. A sequence $\sigma \in L$ is called a *trace*.

For a general introduction of Petri nets we refer to [11]. For the purpose of this paper we use the notion of *labelled Petri nets*.

Definition 1 (Petri net). *Let P denote a set of places, let T denote a set of transitions and let $F \subseteq (P \times T) \cup (T \times P)$ denote the flow relation. Let $\mathcal{L}$ denote the universe of labels and let $\lambda \colon T \to \mathcal{L}$ denote a labelling function. A labelled Petri net is a quadruple $N = (P, T, F, \lambda)$. $\mathcal{N}$ denotes the universe of Petri nets.*

Marking M of Petri net $N = (P, T, F, \lambda)$ is a multiset of P, i.e. $M \in \mathcal{B}(P)$. The initial marking of N is denoted as M_i. An example of a labelled Petri net (with both full/abbreviated transition labels) is depicted in Figure 1. When a transition is enabled, we write $(N, M)[t\rangle$, e.g. $(N_1, [p_i])[t_1\rangle$ and $(N_1, [p_3, p_4])[t_5\rangle$. If firing a sequence of transitions $\sigma \in T^*$, starting in marking M, yields marking M', we write $(N, M) \xrightarrow{\sigma} (N, M')$.

Alignments allow us to compare the behaviour recorded in event logs to the behaviour described by a Petri net. Conceptually, an alignment represents a mapping between the activities observed in a trace $\sigma \in L$ and the execution of transitions in the Petri net.

Definition 2 (Alignment). *Let $\sigma \in \mathcal{A}^*$. Let $N = (P, T, F, \lambda)$ be a labelled Petri net and let $M_i, M_f \in \mathcal{B}(P)$ denote N's initial and final marking. Let $\gg \notin \mathcal{A} \cup T$. A sequence $\gamma \in ((\mathcal{A} \cup \{\gg\}) \times (T \cup \{\gg\}))^*$ is an alignment if:*

1. *$(\pi^1(\gamma))_{\downarrow_\mathcal{A}} = \sigma$; activity part (excluding $\gg$'s) equals σ.*
2. *$(N, M_i) \xrightarrow{(\pi^2(\gamma))_{\downarrow_T}} (N, M_f)$; transition part (excluding $\gg$'s) in Petri net language.*

$$\gamma_1 : \begin{array}{||c|c|c|c|c|} a & b & d & e & h \\ \hline t_1 & t_2 & t_4 & t_5 & t_8 \end{array} \quad \gamma_2 : \begin{array}{||c|c|c|c|c|c|} a & b & d & e & \gg & h \\ \hline t_1 & t_2 & t_4 & \gg & t_5 & t_8 \end{array} \quad \gamma_3 : \begin{array}{||c|c|c|c|c|c|c|c|c|} a & b & d & e & \gg & \gg & \gg & \gg & h \\ \hline t_1 & t_2 & t_4 & t_5 & t_6 & t_4 & t_2 & t_5 & t_8 \end{array}$$

Fig. 2. Example alignments for $\langle a, b, d, e, h \rangle$ and N_1.

3. $\forall (a, t) \in \gamma (a \neq \gg \vee t \neq \gg)$; $(\gg, \gg)$ *is not valid in an alignment.*

We let $\Gamma(N, \sigma, M_i, M_f)$ *denote the universe of alignments of Petri net* N *and trace* σ *given markings* M_i *and* M_f.

Given an alignment, an individual element of a sequence is referred to as a *move*. If a move is of the form $(a, \gg)$ we refer to it as a *log move*, which indicates that we are not able to map an observed activity to the execution of a transition. A move of the from $(t, \gg)$ represents a *model move* and indicates that, according to the model, an activity was required to be executed, yet it didn't occur. Finally, a move of the form (a, t) represents a *synchronous move*, given $a = \lambda(t)$. Consider example trace $\langle a, b, d, e, h \rangle$ and consider Petri net N_1 in Figure 1. Observe that the three sequences γ_1, γ_2 and γ_3, presented in Figure 2, are all alignments. All three alignments are in $\Gamma(N_1, \langle a, b, d, e, h \rangle, [p_i], [p_o])$. Observe that γ_1 minimizes any moves of the form $(\gg, t)$ and $(a, \gg)$, hence, we prefer γ_1 over γ_2 and γ_3. To be able to rank and compare alignments we define a cost-function over moves. The cost of the alignment itself is the sum of the costs of its moves.

Definition 3 (Alignment Cost). *Let* $\sigma \in \mathcal{A}^*$, *let* $N = (P, T, F, \lambda)$ *be a labelled Petri net with* $M_i, M_f \in \mathcal{B}(P)$, *let* $\gg \notin \mathcal{A} \cup T$ *and let* $c_m \colon (\mathcal{A} \cup \{\gg\}) \times (T \cup \{\gg\}) \rightarrow \mathbb{R}_{>0}$. *Given alignment* $\gamma \in \Gamma(N, \sigma, M_i, M_f)$, *the costs* κ^{c_m} *of* γ, *given move cost function* c_m, *is defined as* $\kappa^{c_m}(\gamma) = \sum_{i=1}^{|\gamma|} c_m(\gamma(i))$.

In general one can opt to use an arbitrary instantiation of c_m, however, in the remainder of the paper we adopt the *unit-cost function*:

1. $c_m(a, t) = \epsilon \Leftrightarrow a \in \mathcal{A}, t \in T$ and $\lambda(t) = a$
2. $c_m(a, t) = \infty \Leftrightarrow a \in \mathcal{A}, t \in T$ and $\lambda(t) \neq a$
3. $c_m(a, t) = 1$ otherwise

The unit function only assigns finite values to model-, log- and synchronous moves. Moves (a, t) of the form $a \neq \gg \wedge t \neq \gg \wedge \lambda(t) \neq a$ have value ∞. As we assume unit-costs, we omit c_m as superscript and simply refer to $\kappa(\gamma)$. We write γ^{opt} to refer to *the optimal alignment*, i.e. $\gamma^{opt} = \mathbf{arg\ min}_{\gamma \in \Gamma(N, \sigma, M_i, M_f)} \kappa(\gamma)$.

3 Related Work

We primarily focus on work in conformance checking, for an overview of process mining we refer to [2].

Early work in conformance checking uses token-based replay [12]. The techniques try to replay a given trace in a model and add missing tokens if a transition is not able to fire. After replaying the full trace, remaining tokens are counted and a conformance statistic is computed based on missing and remaining tokens.

Alignments are introduced in [5]. The work proposes to transform a given Petri net and a trace from an event log into a synchronous product net, and, subsequently solve the shortest path problem on the corresponding state space. Its implementation in ProM may be regarded as the state-of-the-art technique in alignment computation and serves as a basis for this paper.

In [1,10] decomposition techniques are proposed together with computing alignments. The input model is split into smaller, transition-bordered, submodels for which local alignments are computed. Using decomposition techniques greatly enhances computation time. The downside of the techniques is the fact that they are capable to decide whether a trace fits the model or not, rather than quantifying to what degree a trace fits.

Recently approximation schemes for alignments, i.e. computation of near-optimal alignments, have been proposed in [14]. The techniques use a recursive partitioning scheme, based on the input traces, and solve multiple Integer Linear Programming problems. The techniques identify deviations between sets of transitions, rather than deviations between singletons (which is the case in [5]).

4 Computing Optimal Alignments

Given a Petri net N with initial marking M_i and final marking M_f, and a trace σ we aim at finding $\gamma^{opt} = \mathbf{arg\ min}_{\gamma \in \Gamma(N,\sigma,M_i,M_f)} \kappa(\gamma)$. Computing the optimal alignment is equivalent to solving a *shortest path problem based on the synchronous product net of N and σ*. A formal definition of such synchronous product net and an equivalence proof of the two problems is outside the scope of this paper. Hence, we refer to [5] for these definitions and proofs. We clarify the use of a synchronous product net by means of an example.

Consider Figure 3 which depicts the synchronous product net of Petri net N_1 and trace $\langle a, b, d, e, g \rangle$. The trace is transformed into a sequential Petri net fragment, depicted in the upper part of Figure 3 (coloured dark grey). Each dark grey transition represents a log move, as reflected by their corresponding label. The lower part of Figure 3 (coloured white) is based on the original model. Each white transition represents a model move. Finally, the middle transitions (coloured light grey) represent synchronous moves and connect each trace-based transition to each equal-labelled transition in the model part. A firing sequence of the synchronous product net from $[p_i, p'_i]$ to $[p_o, p'_o]$ corresponds to a sequence of moves, which in fact is an alignment [5].

Each transition in the synchronous product net corresponds to a move in an alignment, and moreover, to an arc in the state space of the synchronous product. Since each move has an associated cost, we are able to assign the weight of each arc in the state space with the cost of the associated move. The goal of finding an optimal alignment is thus equivalent to solving a shortest path problem on the state space of the synchronous product net, using $[p_i, p'_i]$ as an initial state and $[p_o, p'_o]$ as a final state. In the remainder of this paper we assume the existence

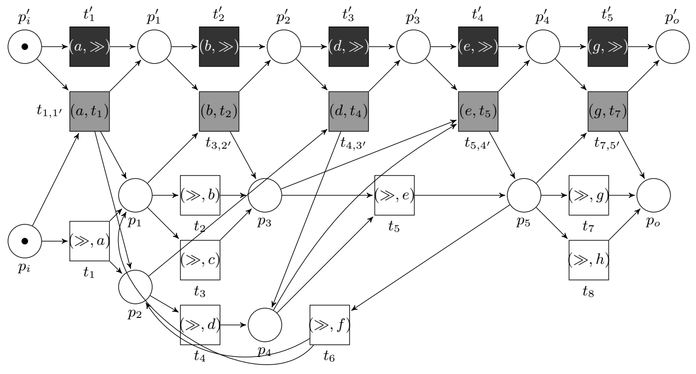

Fig. 3. Synchronous product net of trace $\langle a, b, d, e, g \rangle$ and example Petri net N_1.

of a synchronous product oracle $\otimes\colon \mathcal{N} \times \mathcal{A}^* \to \mathcal{N}$ that, given a Petri net and a trace, computes a synchronous product. We write $N \otimes \sigma$ instead of $\otimes(N, \sigma)$. Furthermore we define $\mathcal{N}^{\mathcal{S}} = \{N^S \mid \exists N \in \mathcal{N}, \sigma \in \mathcal{A}^*(N^S = N \otimes \sigma)\}$. Since the synchronous product inherits the initial marking of the given Petri net and just adds one marked place, i.e. p'_i in Figure 3, we use p'_i to refer to that place. Similarly, we refer to p'_o.

Many algorithms exist that solve a shortest path algorithm on a weighted graph with a unique start vertex and a set of end vertices. In this paper we predominantly focus on the A^* algorithm [7]. The core of the A^* algorithm is the usage of a heuristic function that approximates, for each vertex in the given graph, the expected remaining distance to the *closest end vertex*. The A^* algorithm is *admissible*, i.e. it guarantees to find a shortest path, if the heuristic always underestimates the actual distance. To formally define a heuristic function, we first define $\mathcal{P}$ as the universe of Petri net places. A heuristic function is a function $h\colon \mathcal{N}^{\mathcal{S}} \times \mathcal{B}(\mathcal{P}) \times \mathcal{B}(\mathcal{P}) \to \mathbb{R}_{>0}$, where $h(N^S, M, M')$ denotes the estimated distance to go from M to M' in the state space of N^S. We assume any heuristic function to underestimate the true distance from M to M'. In case we have $M(p) > 0$ or $M'(p') > 0$, and p or p' is not part of N^S, then $h(N^S, M, M') = \infty$, i.e. the heuristic is only defined on the state space of N^S.

In Algorithm 1 we present an algorithmic skeleton for optimal alignment computation using A^*. The algorithm expects a Petri net N with associated initial and final marking M_i, M_f, and a trace σ as an input. Moreover it expects a heuristic function h. The algorithm first creates the synchronous product net N^S. Subsequently it constructs the initial and final marking of the synchronous product net and initializes a set C which contains markings already visited in the search. We use a priority queue Q which stores triples of the form $(M, g, h) \in \mathcal{B}(P^S) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{>0}$. In such triple, M represents in the synchronous product net, g represents the best known cost so far of reaching marking M from M_i^S and h represents the estimated distance to M_f^S from M, i.e. the heuristic for

Algorithm 1: A* (Alignments)

```
   input : $N = (P, T, F, \lambda), M_i, M_f \in \mathcal{B}(P), \sigma \in \mathcal{A}^*, h\colon \mathcal{N}^S \times \mathcal{B}(\mathcal{P}) \times \mathcal{B}(\mathcal{P}) \to \mathbb{R}_{>0}$
   output: $\gamma^{opt} \in \Gamma(N, \sigma, M_i, M_f)$
   begin
1      $N^S = (P^S, T^S, F^S, \lambda^S) = N \otimes \sigma$;
2      $M_i^S \leftarrow M_i \uplus [p'_i]$;
3      $M_f^S \leftarrow M_f \uplus [p'_o]$;
4      $C \leftarrow \emptyset$;
5      initialize priority queue $Q \subseteq \mathcal{B}(P^S) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{>0}$ sorted ascending by the sum of the
         two last tuple arguments;
6      $Q.enqueue(M_i^S, 0, h(N^S, M_i^S, M_f^S))$;
7      while $|Q| > 0$ do
8          $(M, g, h) \leftarrow Q.dequeue()$;
9          $C \leftarrow C \cup (M, g, h)$;
10         if $M = M_f^S$ then
11             return reconstructed alignment by traversing from $M_f^S$ back to $M_i^S$;
12         foreach $t \in \{t' \in T^S \mid M[t'\rangle\}$ do
13             $M' \leftarrow (M \setminus \bullet t) \uplus t\bullet$;
14             if $\nexists (M', g', h') \in C$ then
15                 $g' \leftarrow g + c_m(\lambda^S(t))$;
16                 if $\exists (M'', g'', h'') \in Q(M' = M'' \wedge g' < g'')$ then
17                     replace $(M'', g'', h'')$ by $(M', g', h'')$ in $Q$;
18                     $parent(M') \leftarrow (M, t)$;
19                 else if $\nexists (M'', g'', h'') \in Q(M' = M'')$ then
20                     $Q.enqueue(M', g', h(N^S, M', M_f^S))$;
21                     $parent(M') \leftarrow (M, t)$;
22     return failure;
```

M. Sorting of Q is based on the sum of the last two tuple arguments, i.e. for some tuple (M, g, h) this is $g + h$. While the queue is not empty we remove its head and add it to collection C (line 7). If the head represents the final marking we return the corresponding alignment by reconstructing it using the *parent* pointers, set in line 18 and line 21. If not, we fire each enabled transition in the given marking and investigate the new marking (**foreach** in line 12). If there is already a tuple in C containing the new marking we do nothing (line 14). If not, we check whether Q already contains a tuple with the new marking. If so, we replace that entry if we now reach the marking with a lower path cost, i.e. $g' < g''$ (line 16). If no tuple containing the newly obtained marking exists in Q, we simply insert it together with its associated path costs and heuristic value (line 19). In case of adding or replacing an entry in Q we update a pointer *parent* from the newly reached marking M' to tuple (M, t), stating that we are able to reach M' cheaper by firing t in M.

If we use an underestimating heuristic function, Algorithm 1 is guaranteed to find an optimal path from M_i^S to M_f^S. Hence, line 22 is never reached. If we reach a marking that is already present in C, we ignore this completely. Note that this is only feasible, if we are guaranteed that once we add a triple (M, g, h) to C we are never able to reach that state using distance $g' < g$. This, in turn, is guaranteed if the following *consistency property* holds: $d(M, M') + h(N, M', M_f) \geq$

$h(N, M, M_f)$, where $d(M, M')$ represents the length of the shortest path in the state space of N from M to M'.

We are able to parametrize and/or change several characteristics of the algorithmic skeleton, e.g. the second order sort criterion of Q and the heuristic function. In the upcoming section we discuss each of these changes in detail.

5 Parametrization

In this section we present parametrization and/or changes applicable to Algorithm 1. All changes presented here are in line with the implementation in ProM.

Heuristic Function Observe that a trivial, naive, admissible and consistent heuristic for any marking in the synchronous product net is simply making all remaining activities corresponding to that marking synchronous, given that there exists at least one transition t with a corresponding label. For example, consider marking $[p_1, p_2, p'_1]$ in the synchronous product net in Figure 3. Regardless of how we ended up in the marking, the remaining activity labels are the sequence $\langle b, d, e, g\rangle$. Since for each label in that sequence there exists at least one synchronous transition with a similar label, a naive underestimate of the remaining costs is 4ϵ. Note that this estimator completely ignores whether or not these equally labelled transitions are actually able to fire, at some point in the future, given the current marking. Moreover, the heuristic completely ignores the model part of the alignment (white transitions), i.e. several markings have an equal heuristic.

Alternatively, we exploit the *state equation* of Petri nets as a basis for a heuristic. Let $\mathbf{A}$ denote the incidence matrix of a Petri net $N = (P, T, F, \lambda)$ ($\mathbf{A}$ is an $|T| \times |P|$ matrix with $\mathbf{A}(i, j) = 1$ implies $p_j \in t_i\bullet$, etc.). Furthermore, let $\boldsymbol{x}$ denote at $|T|$-sized column vector of integers. Let M_i and M_f denote two markings and let $\sigma \in T^*$ s.t. $(N, M_i) \xrightarrow{\sigma} (N, M_f)$. Furthermore let $\boldsymbol{m_i}$ and $\boldsymbol{m_f}$ denote two $|P|$ sized column vectors representing M_i and M_f, with $\boldsymbol{m_i}(i) = M_i(p_i)$ and $\boldsymbol{m_f}(i) = M_f(p_i)$ $\forall i \in \{1, 2, ..., |P|\}$. The state equation states that when we instantiate $\boldsymbol{x}$ as the Parikh vector of σ, i.e. if transition t_i occurs k times in σ, $\boldsymbol{x}(i) = k$, then $\boldsymbol{x}$ is a solution to $\boldsymbol{m_f} = \boldsymbol{m_i} + \mathbf{A}^\intercal \boldsymbol{x}$. The opposite however does not hold, i.e. if we find a solution to $\boldsymbol{m_f} = \boldsymbol{m_i} + \mathbf{A}^\intercal \boldsymbol{x}$, $\boldsymbol{x}$ is not necessarily a Parikh representation of a $\sigma' \in T^*$ s.t. $(N, M_i) \xrightarrow{\sigma'} (N, M_f)$.

Nonetheless we are able to utilize the state equation for the purpose of calculating a heuristic. Given any marking M (with vector form $\boldsymbol{m}$) within the synchronous product net, we try to find a minimal solution to $\boldsymbol{m_f} = \boldsymbol{m} + \mathbf{A}^\intercal \boldsymbol{x}$, where $\mathbf{A}$ and $\boldsymbol{x}$ are defined in terms of the synchronous product net. Let $\boldsymbol{c}$ denote a $|T|$-sized vector where each index i has value $c_m(\lambda(t_i))$ for each transition t_i in the synchronous product net. Vector $\boldsymbol{x}$ that minimizes $\boldsymbol{c}^\intercal \boldsymbol{x}$ is a minimal solution to $\boldsymbol{m_f} = \boldsymbol{m} + \mathbf{A}^\intercal \boldsymbol{x}$. Observe that, by contradiction, the value $\boldsymbol{c}^\intercal \boldsymbol{x}$ is always a lower bound for the actual cost to reach M_f from M. Such solution is easily found by, for example, using Integer Linear Programming (ILP) [13].

The ILP-based heuristic is expected to be closer to the actual costs of the shortest path in the synchronous product net's state space, compared to the trivial heuristic. Hence we expect, when using the ILP-based heuristic, that the A^* algorithm visits less states during execution and moreover on average stores less states in the queue. A trade-off is obviously the increase in computation time of the heuristic. Note that we are able to relax the computation time by relaxing the Integer Linear Program to a Linear Program, i.e. $\boldsymbol{x} \in \mathbb{R}^{|T|}_{\geq 0}$. This as a consequence leads to more severe underestimation of the true costs.

When using Integer Linear Programming to find a minimal solution to $\boldsymbol{m_f} = \boldsymbol{m} + \mathbf{A}^\intercal \boldsymbol{x}$, we are able to estimate, and in some cases derive exactly, the heuristic of future states. Assume that we find a minimal-cost solution $\boldsymbol{x}$ to $\boldsymbol{m_f} = \boldsymbol{m} + \mathbf{A}^\intercal \boldsymbol{x}$ based on some marking M in the state space of the synchronous product net. Take any $t \in T^S$ with $M[t\rangle$. We know that for marking $M' = (M \setminus \bullet t) \uplus t\bullet$, we have $\boldsymbol{m_f} = \boldsymbol{m'} + \mathbf{A}^\intercal(\boldsymbol{x} - \mathbf{1}_t)$, i.e. $\boldsymbol{m_f} = \boldsymbol{m'} + \mathbf{A}^\intercal \boldsymbol{x'}$ has a solution with $\boldsymbol{x'} = \boldsymbol{x} - \mathbf{1}_t$. Now assume that there exists some vector $\boldsymbol{y}$ that is a strictly smaller solution to $\boldsymbol{m_f} = \boldsymbol{m'} + \mathbf{A}^\intercal \boldsymbol{x'}$ than $\boldsymbol{x} - \mathbf{1}_t$. Since we know that $M \xrightarrow{t} M'$, we know that $\boldsymbol{y} + \mathbf{1}_t$ is a solution to $\boldsymbol{m_f} = \boldsymbol{m} + \mathbf{A}^\intercal \boldsymbol{x}$ with a strictly lower cost than $\boldsymbol{x}$. This however contradicts minimality of $\boldsymbol{x}$, hence $\boldsymbol{c}(\boldsymbol{x} - \mathbf{1}_t)$ is a lower bound for $h(N^S, M', M_f)$, i.e. $\boldsymbol{c}(\boldsymbol{x} - \mathbf{1}_t) \leq h(N^S, M', M_f)$. In the more specific case that we fire t_i in M with $\boldsymbol{x}(t_i) > 0$, we deduce, by similar rationale, that $\boldsymbol{c}(\boldsymbol{x} - \mathbf{1}_t) = h(N^S, M', M_f)$. Thus, based on solving one ILP for a marking M, we in some cases already know the heuristic for the next state M', and in some cases we know a lower bound. Therefore, if we know the exact heuristic we do not need to solve new ILP's and simply add/update the heuristic of M' to $\boldsymbol{c}(\boldsymbol{x} - \mathbf{1}_t)$. If it is a lower-bound, we also add/update the heuristic of M' to $\boldsymbol{c}(\boldsymbol{x} - \mathbf{1}_t)$, and mark a boolean flag related to M' stating that we have a lower bound. In case the tuple related to marking M' gets on top of the queue we actually solve the underlying ILP to get the exact heuristic. This potentially moves the tuple further down the queue. Hence, we postpone and potentially reduce heuristic computation, yet we increase the number of polling operations to the queue.

Second-Order Queueing Criterion Within the algorithmic skeleton we use queue Q that uses the sum of the g and h values as a sorting criterion, i.e. in terms of A^* this is referred to as the f-value. Multiple markings of the synchronous product net exist that have the same f-value. By default the ordering of markings with an equal f-value is random. Alternatively, we pose two second-order sorting criteria. If we use the h-values as a second-order criterion, we effectively traverse the states with an equal f-value in a *depth-first* manner, i.e. we explore states that *seem to have a better solution* first. Alternatively, if we use the g-values as a second-order criterion, we traverse states with an equal f-value in a *breadth-first* manner, i.e. we explore states that have a minimal actual distance g to the start state first.

Restricting Transition Firing Reconsider the synchronous product net shown in Figure 3 with marking $[p_i, p'_i]$. Observe that there are three firing sequences in the net to achieve marking $[p_1, p_2, p'_1]$, i.e. $\langle t_{1,1'} \rangle$, $\langle t'_1, t_1 \rangle$ and $\langle t_1, t'_1 \rangle$. The cost associated with $\langle t_{1,1'} \rangle$ is ϵ whereas the cost for $\langle t'_1, t_1 \rangle$ and $\langle t_1, t'_1 \rangle$ is 2. We observe that both possible permutations of the sequence containing t_1 and t'_1 have the same cost and can both be part of a (sub-optimal) alignment. In the general sense, assume we have some alignment $\gamma = \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \in \Gamma(N, \sigma, M_i, M_f)$ s.t. γ_2 only consists of log and model moves. In sequence γ_2, if we swap any adjacent log and model move, yielding γ'_2, then also $\gamma' = \gamma_1 \cdot \gamma'_2 \cdot \gamma_3 \in \Gamma(N, \sigma, M_i, M_f)$ and, trivially, $\kappa(\gamma) = \kappa(\gamma')$.

Hence, to find the optimal alignment we need to traverse one specific permutation of the path to the optimum, rather than all possible permutations. The A^* algorithm already limits the number of visited states and also choosing an appropriate second-order criterion helps in preventing this. In its basic form, every enabled transition in a certain state is added to Q. However, we are able to limit this number of states by exploiting the previously mentioned property. We are able to manipulate the **foreach**-loop of Algorithm 1 (line 12) in two ways:

- If the transition leading to the current marking relates to a *model move* we only consider those transitions t that relate to a *model* or *synchronous move*.
- If the transition leading to the current marking relates to a *log move* we only consider those transitions t that relate to a *log* or *synchronous move*.

Observe that the pointers stored for each marking allow us to make such decision. In the first option we are not able to schedule a log move after a model move. The other way around is however possible, i.e. we are allowed to schedule a model move after a log move. The second option behaves exactly opposite, i.e. we are not allowed to schedule a model move after a log move.

6 Evaluation

In order to observe the effect of the parameters presented in Section 5 in terms of performance, i.e. computation time and queue size needed, we designed an exploratory experiment. The experiment is designed as a scientific process mining workflow [6] and was implemented as a `RapidMiner` workflow, based on the `RapidProM` extension (workflow and results available at: `https://github.com/s-j-v-zelst/research/releases/download/2017_ataed/experiments.tar.gz`). We present the corresponding results here. Prior to this we present the experimental set-up.

6.1 Experimental Set-up

To analyse the effect of different values of the parameters presented in Section 5, we use a scientific workflow that, conceptually, performs the following steps.

1. We generate 11 (block-structured) Petri nets with k labelled transitions, where k is drawn from a triangular distribution with parameters $\{10, 30, 50\}$, for increasing levels of Parallelism (from 0 to 50% in steps of 5%) [8]. The base

Table 1. Parameters used in experiments based on Section 5.

Parameter	*Type*	*Values*

Heuristic (h)	Categorical	`NAIVE`
		`ILP without lower-bound estimation`
		`ILP with lower-bound estimation`
		`LP without lower-bound estimation`
		`LP with lower-bound estimation`

Second-order Queueing Criterion	Categorical	`RANDOM` `DFS` (sort on `h`-value) `BFS` (sort on `g`-value)

Transition Restriction	Categorical	`NONE`
		`MODEL`
		`LOG`

distribution of the three constructs sequence, exclusive choice and parallelism are 46%, 35% and and 19% respectively. This distribution is estimated based on the percentage of process models that contain such constructs, obtained from [9]. When increasing and/or decreasing the level of parallelism, we distribute the probabilities of generating the other constructs accordingly.

2. For each Petri net, generate an event log with n cases, where n is a random number between 100 and 1000.
3. For each event log, add increasing levels (from 0 to 50% in steps of 10%) of one type of noise, i.e. *remove activity*, *add activity* or *swap activities*.
4. For each "noisy" event log, do conformance checking against the Petri net it was generated from, using all parameter combinations listed in Table 1.

All generated Petri nets are block-structured and have no loops. We do not consider more advanced Petri nets containing duplicate labels and/or invisible transitions. Hence, the state space of the models considered is strictly finite. Nonetheless, 56.700 conformance checking operations were performed.

6.2 Results

In this section we present the results of the experiments performed. We first assess computation time performance, after which we focus on memory usage in terms of queue-size. Due to space limitation we only show results per category, i.e. we do not focus on parameter interaction. Moreover, we only present results in which (seemingly) significant differences are observed, and, parallelism levels up to 35%. For the optimizations based on the *state equation*, i.e. using linear programming and/or lower-bound estimation we did not encounter notable differences in terms of computation time and memory usage. The same holds for the parameters based on the use of *transition restriction*.

Computation Time In Figure 4 we depict the average optimal alignment computation time in milliseconds, when using either the *naive* heuristic, or the *ILP-based* heuristics, without relaxation and/or lower-bound estimation. Each plot in the figure represents a different level of parallelism within the generated process models. We show parallelism from 0% to 35%. On the x-axis we plot

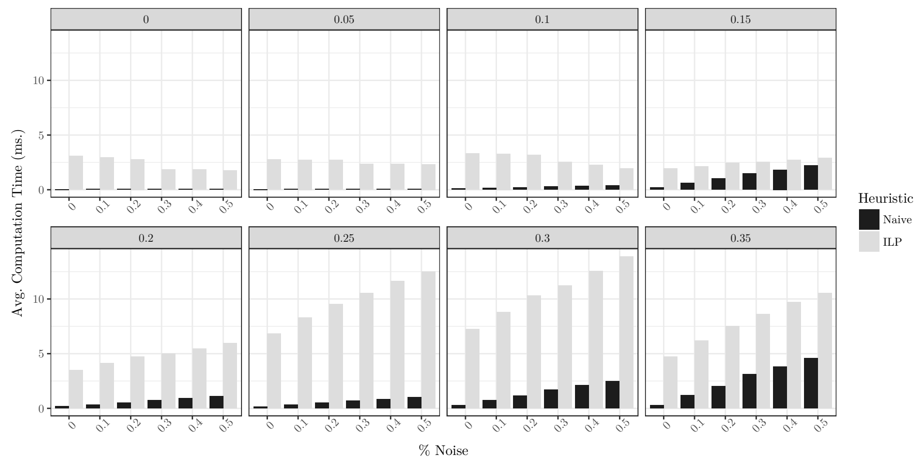

Fig. 4. Average computation time (ms.) per percentage of noise, plotted per level of parallelism. Level of parallelism is displayed on top of the charts.

the percentage of noise added to the generated event logs. On the y-axis we plot average computation time. We observe that for parallelism levels 0%, 5%, 10% and 15%, computation time of ILP is relatively stable and significantly larger than the naive estimator. The computation time for the naive estimator starts increasing when more noise is introduced for models with at least 15% parallelism. For ILP we observe similar behaviour starting from parallelism levels of 20% and higher. Interestingly, for 15% of parallelism the rate of growth of the naive version is clearly higher than the rate of growth of ILP. In all other cases, i.e. for parallelism levels of 20% and up, ILP's growth rate seems higher than the naive variant.

In Figure 5 we depict the average optimal alignment computation time for each type of second order queueing criterion. For parallelism levels greater (or equal to) 25% we observe a significant difference in computation time between DFS and BFS/Random. We observe that the difference also increases with the increase of noise within the event logs. BFS and Random are comparable in computation time. The results of this experiment show that a preference for states within the queue based on the estimated remaining distances is beneficial for reaching a target state faster.

Memory Usage Here we focus on memory usage in terms of average queued states. In Figure 6 we depict the average amount of queued states, when using either the *naive* heuristic, or the *ILP-based* heuristics, *without relaxation and/or lower-bound estimation.* As expected the state equation based heuristic, i.e. using ILP, traverses the state space more efficiently leading to significantly less states queued. Whereas the naive heuristic rises in terms of queued states both with increases of parallelism and noise, the ILP-based heuristic seems unaffected by

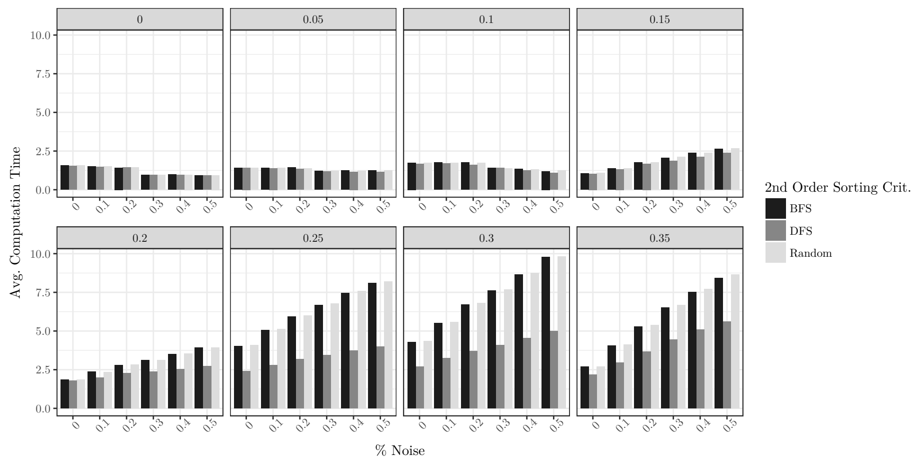

Fig. 5. Average computation time (ms.) per percentage of noise, plotted per level of parallelism.

the amount of noise introduced. The number of queued states slightly increases when the level of parallelism is increased, however, it is orders of magnitude smaller than the increase of the naive heuristic. Based on these results, together with the computation time results, there seems to be a clear trade-off between using ILP or a naive heuristic in terms of computation time versus memory usage.

In Figure 7 we depict the average queue size for each type of second order queueing criterion. We observe that, like in the case of computation time, the

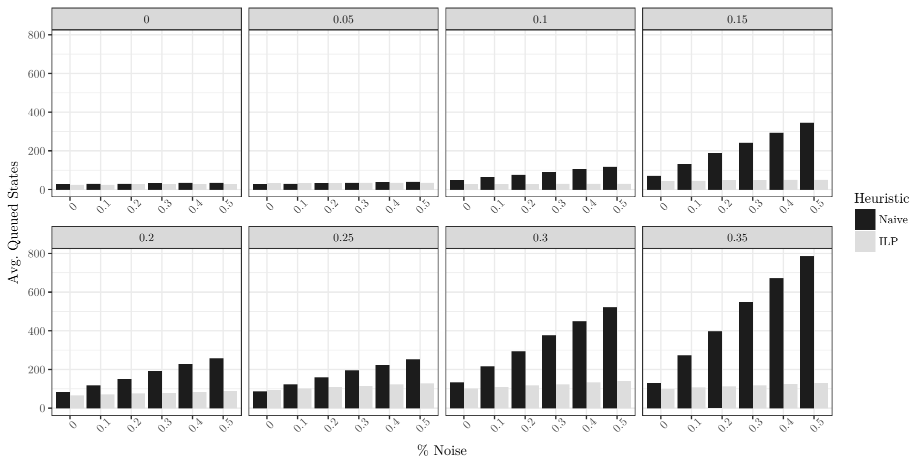

Fig. 6. Queued states per percentage of noise, plotted per level of parallelism.

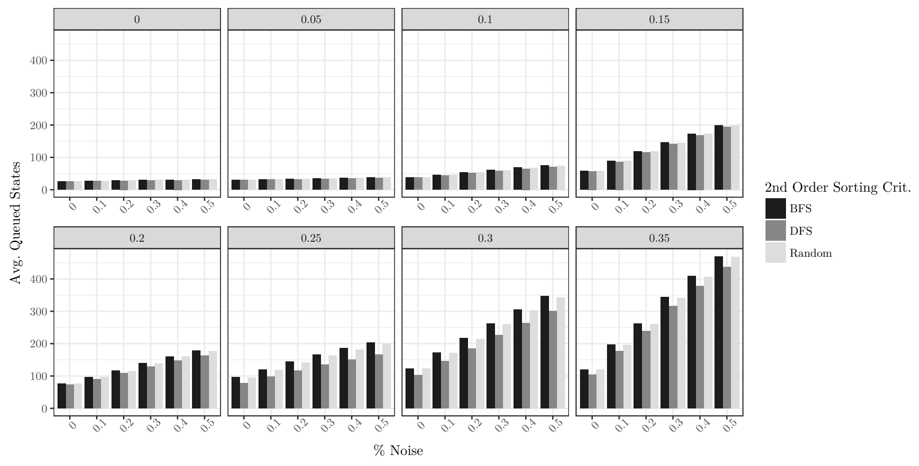

Fig. 7. Queued states per percentage of noise, plotted per level of parallelism.

`DFS` variant outperforms the other two. The difference is however less prominent.

7 Conclusion

In this paper we have presented multiple ways to parametrize the A^* search algorithm used in optimal alignment computation. Some parametrization concerns 2^{nd}-order sorting criteria within an internal priority queue used, whereas other parametrization utilizes Petri net theory. The parametrization discussed is in line with, and limited to, the current implementation of optimal alignment computation in the process mining tool-kit ProM. As such, this paper acts as a high-level description of the current implementation. We have performed an exploratory evaluation of the effects of different parameter combinations w.r.t. the number of states queued and the computation time of finding optimal alignments. Our results show that, for the class of models considered, using a naive heuristic outperforms the more advanced state-equation based heuristic in terms of computation time. Moreover using the heuristic value as a second-order sorting criterion for the internal priority queue is beneficial for memory usage and computation time.

Future Work The results of this exploratory study show that parametrization of the heuristic search has an impact on its performance. We plan to extend the current evaluation using a larger variety of Petri nets and a larger number of iterations per model-log combination.

Several approximation schemes exist for A^*, e.g. using a scaling function within the heuristic. We plan to assess the impact of these approximation schemes on alignment computation as well. We additionally plan to examine the use of alternative informed search methods such as *Iterative Deepening* A^* (IDA^*).

Finally, we plan to adopt (prefix-)alignment computation in an online setting.

References

1. van der Aalst, W.M.P.: Decomposing Petri Nets for Process Mining: A Generic Approach. Distributed and Parallel Databases 31(4), 471–507 (2013)
2. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
3. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying History on Process Models for Conformance Checking and Performance Analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
4. van der Aalst, W.M.P., Bolt, A., van Zelst, S.J.: RapidProM: Mine Your Processes and Not Just Your Data. CoRR abs/1703.03740 (2017)
5. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science (Jul 2014)
6. Bolt, A., de Leoni, M., van der Aalst, W.M.P.: Scientific Workflows for Process Mining: Building Blocks, Scenarios, and Implementation. STTT 18(6), 607–628 (2016)
7. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. Systems Science and Cybernetics 4(2), 100–107 (1968)
8. Jouck, T. and Depaire, B.: PTandLogGenerator: A Generator for Artificial Event Data. In: Azevedo, L., Cabanillas, C. (eds.) Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Brazil, September 21, 2016. CEUR Workshop Proceedings, vol. 1789, pp. 23–27. CEUR-WS.org (2016)
9. Kunze, M., Luebbe, A., Weidlich, M., Weske, M.: Towards Understanding Process Modeling - The Case of the BPM Academic Initiative. In: Dijkman, R.M., Hofstetter, J., Koehler, J. (eds.) Business Process Model and Notation - Third International Workshop, BPMN 2011, Lucerne, Switzerland, November 21-22, 2011. Proceedings. Lecture Notes in Business Information Processing, vol. 95, pp. 44–58. Springer (2011)
10. Munoz-Gama, J., Carmona, J., van der Aalst, W.M.P.: Single-Entry Single-Exit Decomposed Conformance Checking. Inf. Syst. 46, 102–122 (2014)
11. Murata, T.: Petri nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (Apr 1989)
12. Rozinat, A., van der Aalst, W.M.P.: Conformance Checking of Processes Based on Monitoring Real Behavior. Inf. Syst. 33(1), 64–95 (2008)
13. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience series in discrete mathematics and optimization, Wiley (1999)
14. Taymouri, F., Carmona, J.: A Recursive Paradigm for Aligning Observed Behavior of Large Structured Process Models. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9850, pp. 197–214. Springer (2016)
15. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Soffer, P., Proper, E. (eds.) Information Systems Evolution - CAiSE Forum 2010, Hammamet, Tunisia, June 7-9, 2010, Selected Extended Papers. Lecture Notes in Business Information Processing, vol. 72, pp. 60–75. Springer (2010)