REACH: Researching Efficient Alignment-based Conformance Checking

Jacobo Casas-Ramos[a,*], Manuel Mucientes[a], Manuel Lama[a]

[a] *Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS)*
Universidade de Santiago de Compostela, Santiago de Compostela, Spain

Abstract

Conformance checking techniques compare how a process is supposed to be executed according to a model with how it is executed in reality according to an event log. Alignment-based approaches are the most successful solutions for conformance checking. Optimal alignments are a way of finding the best match between the real and the modeled behavior and identifying the differences. However, finding these optimal alignments is a challenging task, especially for complex cases where the log and the model have many events and paths. The difficulty lies in the computational complexity required to find these alignments. To address this problem, we propose an efficient algorithm named REACH based on the A* search algorithm. The core components of the proposal are the use of a partial reachability graph for faster execution of process models for alignment computation and a set of optimization techniques for reducing the number of states explored by the A* algorithm. These improve performance by both reducing the required computation time per state and the number of states to process respectively. To evaluate the performance and scalability, we conducted tests using 227 pairs of logs and models, comparing the results obtained with those from 10 state-of-the-art approaches. Results show that REACH outperforms the other proposals in runtimes, and even aligns logs and models that no other algorithm is able to align.

Keywords: Process Mining, Conformance Checking, Alignments

1. Introduction

Companies need to automate and digitalize their processes to become more competitive, cut costs and avoid delays in their operations. In this context, a process is a set of activities with coordination requirements among them, which are executed by a set of resources to achieve an objective (Carmona et al., 2018). These processes are described by means of process models that clearly detail the activities to be performed as well as when and which resources will execute them. However, in practice the execution of the processes differs from the process models that were designed to automate the process, making it difficult to understand what is happening in the process and to take decisions.

*Corresponding author
Email addresses: jacobocasas.ramos@usc.es (Jacobo Casas-Ramos), manuel.mucientes@usc.es (Manuel Mucientes), manuel.lama@usc.es (Manuel Lama)

Process mining is an emerging discipline whose aim is to get information about what is really happening in the execution of a process, giving an understanding of the real processes that take place in an organization (van der Aalst et al., 2012). To achieve this, process mining techniques use an event log as input. An event log is a set of traces, each containing a sequence of events. Each event has information about the activity that has been executed, the timestamp of that activity, the trace identifier, the resource that has performed the activity, and, optionally, contextual information about the event execution. With this in mind, three fundamental descriptive process mining techniques have emerged: *(i)* process discovery, which aims to retrieve the underlying process model that represents the behavior recorded in an event log; *(ii)* conformance checking, where a process model is compared with a log of the same process to analyze and quantify the deviations between the modeled and the observed behavior, as recorded in the log; and *(iii)* process enhancement, where a process model is modified and improved based on the information from the log. In this paper, we focus on conformance checking, particularly in the computation of the alignments between the process model and the log traces.

Several conformance checking approaches have appeared in recent years. These approaches can be classified in token replay-based (Berti and van der Aalst, 2021; Rozinat and Van der Aalst, 2008) and alignment-based (Adriansyah, 2014; de Leoni et al., 2018; Lu et al., 2015; de Leoni and van der Aalst, 2013; Taymouri and Carmona, 2016; van Dongen, 2018; de Leoni and Marrella, 2017). The former approaches try to execute all events on the model, registering all states of the process model and modifying the execution state when it is needed for a proper event execution on the model —possibly reporting errors that would be false positives. Alignment-based approaches are widely regarded as the most effective solutions for conformance checking, as they return much more accurate results, pinpointing the optimal model deviations. These methods align each trace with the closest path allowed by the model, even if they do not match perfectly. Some of these techniques (Adriansyah, 2014; de Leoni et al., 2018; Lu et al., 2015; de Leoni and van der Aalst, 2013; Taymouri and Carmona, 2016; van Dongen, 2018) are based on the A* algorithm (Hart et al., 1968), a graph search algorithm that uses a heuristic to guide the search while ensuring optimal results. However, alignment-based approaches encounter challenges when dealing with intricate models and extensive logs, resulting in extended processing times and even not being able to compute some alignments. Several approaches tackle this problem by relaxing restrictions and returning non-optimal alignments (Leemans et al., 2018; Reißner et al., 2017; Reißner et al., 2020a; Reißner et al., 2020b), which might not be suitable for applications where an exact description of the differences between the process model and the log is required.

In this paper, we introduce REACH, an extension of the A* algorithm designed to compute optimal alignments efficiently. It incorporates a series of optimizations to enhance performance and scalability. They significantly reduce the number of states that need to be explored to reach the optimal solution and the processing time spent on each state. The main contributions of the proposal are:

- A new heuristic that quickly finds required activities —activities that must be executed to reach the

end of the model— and compares them with the remaining trace in order to provide more accurate estimates and speed up the algorithm without losing its optimality.

- Techniques for reducing the number of states explored by the A* algorithm. These include optimizations that check and force the execution of required moves by exploring the rest of the trace and the model. Furthermore, a greedy algorithm that returns a sub-optimal alignment is used as an upper bound of cost for the main algorithm. These optimizations filter states that will not lead to the optimal alignment, as another state will lead to an alignment of less cost. This also removes all neighbors generated by the ignored states recursively, greatly reducing the computational cost.
- Efficient execution of process models for alignment computation using a partial and incremental reachability graph. It works by saving information about how process models run for alignment computation. It prevents performing repeated model operations at each step of the A* algorithm.

The remainder of this article is organized as follows. Section 2 studies and compares previous work. Next, Section 3 defines all the required concepts on which this work is based. The algorithm is described in detail in Section 4. Section 5 describes the experiments and discusses the results, compared to the current state of the art. Finally, the conclusions and the future work are presented in Section 6.

2. Related work

Conformance checking is a very active field in process mining. One of its most popular approaches is token-based replay over Petri nets (Berti and van der Aalst, 2021; Rozinat and Van der Aalst, 2008). It involves executing each event of the Petri net as they appear in the trace. If an event cannot be executed, the tokens required to execute it are inserted and counted. Once the full trace has been replayed, all the remaining tokens that are not in a sink place are counted. The fitness metric, which is a measure of conformance between the log and the model, is computed based on these counts. Unfortunately, this technique is less accurate than alignment-based approaches, as it assumes the model is always correct. Furthermore, the provided diagnostics are hard to understand for the end-user, as they are tied to the Petri net model representation and replay. In (van den Broucke et al., 2014) they propose a decomposition-based extension of token-based replay that is more efficient, but it shares the aforementioned disadvantages.

A recent development in this field is stochastic conformance checking (Leemans et al., 2021), which involves comparing event logs and process models while recognizing that logs represent only a subset of possible behaviors. The major disadvantage of the approach is that stochastic process model discovery is a necessary prerequisite for applying this technique. It is a computationally expensive task that has much higher memory requirements than traditional process discovery techniques. The algorithm depends on an input parameter (probability mass) that makes the returned metrics more stable when it is increased. However, the runtime of the algorithm increases very quickly with respect to this parameter for most datasets. This is

even worse if the model presents concurrency and looping behavior.

Alignment-based techniques (Adriansyah, 2014) can identify and explicitly list all discrepancies, enabling the detection of optimal trace executions through the model. These techniques still rely internally on the execution of process models, but they can find the path through the model with the least number of discrepancies possible. There are also techniques, like behavioral alignments (Garcia-Banuelos et al., 2018), that provide textual summaries of conformance without actually computing alignments. These descriptions might be easier for end-users, but they are not as useful as alignments. Alignments link event data to the model, and can be used for post-processing tasks such as fitness calculation, performance analysis, model repair, or prediction tasks.

Alignments are considered the standard for conformance checking. In order to find the optimal alignments, most approaches use a pathfinding algorithm like A* for aligning the trace and the model. A* requires a heuristic, which defines the way to explore the state space of the problem. On the one hand, simple heuristics lead to faster exploration but more explored states (Adriansyah, 2014; de Leoni et al., 2018; Lu et al., 2015), which makes computing alignments for medium or large models impractical without other states reduction techniques. On the other hand, complex heuristics focus on reducing the number of states at the cost of more processing time per state. An illustrative instance of a complex heuristic is rooted in Integer Linear Programming (ILP), utilizing constraints extracted from the model, the remaining trace, and an optimization cost function. Given a state (marking in the model and remaining trace events), the ILP solver is capable of determining the minimum cost that a solution may have, so it is suitable as a heuristic for A* (de Leoni and van der Aalst, 2013; Taymouri and Carmona, 2016; van Dongen, 2018). However, a drawback of complex heuristics is that the computational cost for each discovered state often surpasses the efficiency gained through state reduction.

Leoni et al. (de Leoni and Marrella, 2017) convert the alignment problem to the Planning Domain Definition Language and use an external automated planner to compute the alignments. Their algorithm can modify the planning framework for alignment computation. Nevertheless, their approach depends on the blind A* heuristic, which guarantees optimality but significantly underestimates the remaining cost.

Considering the exponential complexity of optimal algorithms, researchers have introduced non-optimal techniques for computing alignments. These methods were proposed as a compromise between the quality of the results and the computational cost. These techniques do not guarantee that the alignment with the least cost will be found, but they provide approximations with a quality that depends on the model and the log. One such technique is (Taymouri and Carmona, 2018), utilizing an evolutionary algorithm to offer improved alignment approximations, albeit without the assurance of discovering all optimal alignments. Another approach that uses local search to reduce processing time and memory usage is (Taymouri and Carmona, 2020), with the added limitation of only being able to return one of the alignments. It is worth mentioning (van Dongen et al., 2017), which also performs an iterative search in order to find a possibly

non-optimal alignment.

A prevalent method for non-optimal alignments involves decomposing models into smaller, more computationally efficient parts and aggregating partial results, even though this may not always result in optimal alignments (Munoz-Gama et al., 2014; van der Aalst, 2013; Sani et al., 2020). Alternatively, a recomposing technique that is capable of obtaining optimal alignments from the partial pseudo-alignments was developed in (Lee et al., 2018), but it was only executed with manual decompositions of models. Another decomposing optimal technique is described in (Munoz-Gama et al., 2013), but it is limited to sound and safe workflow nets.

Finally, another type of non-optimal technique is based on building automata capable of aligning the log and the model (Reißner et al., 2017; Leemans et al., 2018; Reißner et al., 2020a; Reißner et al., 2020b). These approaches, while non-optimal, give good approximations of the optimal alignments for most cases. The method in (Leemans et al., 2018) splits the activities into multiple subsets to handle very complex models with many activities. Consequently, it increases performance on models with lots of activities at the expense of lower cost accuracy of the resulting alignments.

The state of the art still struggles when facing complex models and logs, leading to large runtimes, and even not being able to compute some alignments. To address this issue, several proposals have relaxed restrictions and returned non-optimal alignments. However, in this paper, we present an A*-based algorithm that increases performance while still providing optimal alignments. This algorithm introduces several techniques for reducing the number of states of the search space to be explored: a new heuristics to better guide the states to explore, a greedy algorithm to find an upper cost bound of the optimal alignment, and two optimizations that check each state to force certain mandatory moves instead of generating all possible neighboring states. Furthermore, our approach introduces an efficient way to execute process models that relies on a partial and incremental reachability graph in order to speed up the computation required for each state. These techniques speed up computation and mitigate the scalability problem currently present in the state-of-the-art proposals.

3. Preliminaries

To perform conformance checking a log and a process model are needed. Logs consist of events, organized into traces, with each trace representing the execution of the associated process model. Every event must have the execution timestamp, the trace identifier that groups the events of the same process execution, and the executed activity. Table 1 shows an example of a log with two traces from an e-learning platform, where each row of the table is an event. For the sake of simplicity, we define traces as a sequence of executed activities, sorted by the execution timestamps of each event or in the order of appearance in the log in case of ties.

Table 1: Example of a log corresponding to an e-learning process. A gray font highlights the events of the trace Case234.

Timestamp	Trace ID	Activity
2021-09-01 22:16:29	Case234	Enroll
2021-09-03 16:12:24	Case675	Enroll
2021-10-04 08:09:56	Case234	Class
2021-10-19 00:00:13	Case234	Class
2021-11-01 04:10:07	Case675	Class
2021-11-15 02:09:48	Case234	Test
2021-11-29 13:53:30	Case675	Test
2021-12-13 07:24:41	Case675	Class
2022-01-17 23:03:31	Case675	Exam
2022-01-19 06:42:33	Case234	Exam
...		

Definition 1 (Trace)**.** A trace $\sigma = \langle a_1, \ldots, a_n \rangle$ is a sequence of activities a_i extracted from events that share the case identifier. Each trace or case contains activities belonging to the same process execution, which are ordered by the execution timestamp of their event. Note that if two activities have the same execution timestamp, they are sorted by the order in which they have been registered. The $+\!\!+$ operator concatenates two sequences. Given a trace σ, the notation $\sigma[i{:}]$ refers to the subsequence from position i (zero-indexed, inclusive) to the end of the sequence.

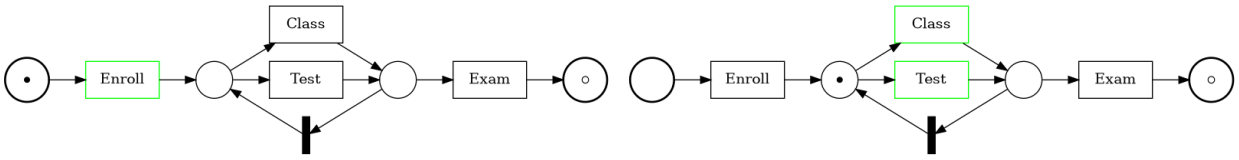
In Table 1, traces refer to the actions of a student in a virtual course. To track the actions of an individual student, we can extract a trace by filtering the recorded events (log table rows) with the same trace identifier. A student executed the activity in the column "Activity" at the moment indicated by the column "Timestamp". For instance, the trace identified as Case234 documents the sequence of activities executed by the student: ⟨Enroll, Class, Class, Test, Exam⟩.

Definition 2 (Log)**.** An event log $L = [\sigma_1, \ldots, \sigma_n]$ is a multiset of traces σ_i. Each trace corresponds to one execution of the process.

A process model describes the allowed behavior by giving activities a structure with initial and final states. In the realm of conformance checking, Petri nets are the dominant technique for representing process models.

Definition 3 (Petri net)**.** A Petri net is a tuple $PN = (P, T, F, \lambda)$ where

- P is the set of places.
- T is the set of transitions. These may contain silent transitions τ, which are related to the model structure, and non-silent transitions, which are related to the execution of activities. T_s denotes the set of silent transitions and $T_{ns} = T \setminus T_s$ is the set of non-silent transitions.



(a) Initial marking of the Petri net.

(b) Marking after executing the transition Enroll.

Figure 1: Example of a Petri/workflow net that models a very simplified process of an e-learning course. Places are represented as circles and transitions as rectangles —a silent transition is shown as a thin black rectangle. The filled circles inside some places indicate the number of tokens they contain. The end place is shown with a circumference inside. Enabled transitions are shown with green borders. In this process, students can enroll in the course and, once enrolled, they can attend any number of classes and/or take tests, performing at least one of those activities. Lastly, a final exam is required to complete the course. In this example, the Enroll transition is executed, leading the Petri net from its initial marking (a) to the next marking (b). Due to the execution of Enroll, a token is consumed in the initial place, and one is produced in its output place, enabling the Class and Test transitions. A complete process execution for this model would be ⟨Enroll, Class, Exam⟩.

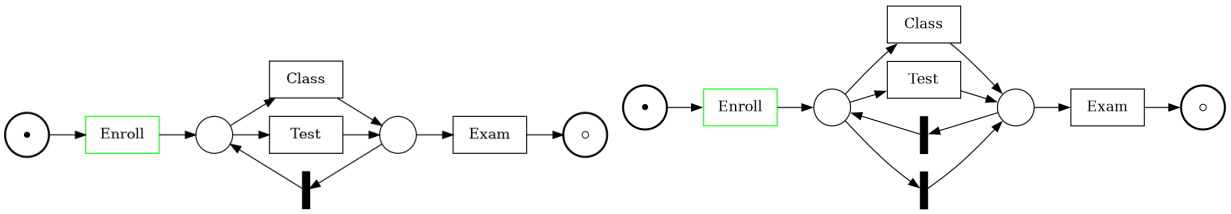
- $P \cap T = \emptyset$.
- $F \in (P \times T) \cup (T \times P)$ is the set of directed arcs that connect places to transitions and vice versa.
- $\lambda : T_{ns} \rightarrow A$ maps every non-silent transition to a label —an activity that may appear in the log. Multiple transitions can have the same label, as our algorithm supports handling duplicate activities in the process model. We also use λ_r to perform the reverse mapping —map an activity to the set of non-silent transitions with the given label.

Petri nets are directed bipartite graphs where nodes are places and transitions. We denote $\bullet t$ as the input places ($\bullet t = \{p \in P \mid (p,t) \in F\}$) and $t\bullet$ as the output places ($t\bullet = \{p \in P \mid (t,p) \in F\}$) of a transition $t \in T$. These are the places directly linked by arcs to and from transition t, respectively. The same operator can be applied to places to retrieve the connected transitions. In order to execute Petri nets, it is necessary to introduce the concepts of token and marking. A place $p \in P$ can store any number of tokens in the marking M, as given by the function $tokens(M, p)$. Hence, we define a marking as a multiset of places that represents the number of tokens in each place.

Definition 4 (Marking)**.** Let $PN = (P, T, F, \lambda)$ be a Petri net and let $\mathcal{B}$ be the power multiset function. A marking $M \in \mathcal{B}(P)$ of that Petri net is a multiset of places. This multiset represents the places that contain tokens and the number of tokens they contain. $M_0 \in \mathcal{B}(P)$ is the initial marking of the Petri net.

During the execution of Petri nets, tokens are added to and removed from places. Specifically, at the start of the process, the Petri net is initialized with the tokens of the initial marking M_0. A transition $t \in T$ is enabled at a marking M iff $\forall p \in \bullet t : tokens(M, p) > 0$. This condition means that the transition is enabled when there is at least one token available for consumption in each place connected to the transition through an input arc. To execute that transition t, all input places $p \in \bullet t$ consume one token; and all output places $p \in t\bullet$ receive a token. When a marking including tokens in a place marked as final is reached, the process is considered as finished. Figure 1(a) shows an example of a Petri net.

Workflow nets (van der Aalst, 1996) are a class of Petri nets focused on modeling processes: they have a



(a) Designed process model (b) Process model discovered from observed behavior

Figure 2: (a) The process model from our running example and (b) the real process model discovered from the event log.

single input place and a single output place, and all transitions are in a path from the start to the end places.

Definition 5 (Workflow net)**.** Let $PN = (P, T, F, \lambda)$ be a Petri net. It is a workflow net if and only if:

- There is a single input place $p_i \in P$. $\bullet p_i = \emptyset$.
- There is a single output place $p_o \in P$. $p_o \bullet = \emptyset$.
- Adding a transition t with $\bullet t = p_o$ and $t \bullet = p_i$, would cause the Petri net to become a strongly connected graph.

A workflow net has proper completion if for any reachable marking M by executing any sequence of enabled transitions $\langle t_0, \ldots, t_n \rangle$, $tokens(M, p_o) > 0 \implies M = [p_o]$, i.e., any sequence of fired transitions that reach the end will only have one token in the output place.

In practice, process executions frequently deviate from the intended process models. For instance, in the context of the virtual course from Figure 1(a), students are expected to attend classes before taking the final exam. However, in reality, some students may choose to skip these lessons and still manage to complete the course, even though this behavior is not accounted for in the designed process model. This is shown in Figure 2. Alignments provide the necessary information to identify deviations of traces from the process model. An alignment can be computed for each trace given a process model, as a trace is an execution of the model. An alignment defines a path that traverses both the trace and the process model from start to end. It provides detailed conformance information useful for multiple tasks like model repair, auditing and prediction, even if the trace and the model do not match perfectly. This is achieved by associating the executed activities in the trace with the transitions in the model. Alignments are built from legal moves, which consume an activity of the trace and execute a non-silent transition with a matching label in the process model, or perform an asynchronous move by only executing a transition on the model (model move) or advancing on the trace (log move).

Definition 6 (Legal move)**.** Let M be the current marking in the model, let σ be the trace to align, and let i be the first index —zero-based— of the trace that still needs to be aligned so that $\sigma[i]$ is the next activity to align. Furthermore, let $M[t\rangle$ denote that marking M enables transition t. A legal move is one of the following.

Log:	Enroll	»	Exam	Test
Model:	Enroll	Class	Exam	»

Log:	Enroll	»	Exam	Test
Model:	Enroll	Test	Exam	»

Figure 3: Two optimal alignments using the default cost function for the trace Enroll, Exam, Test and the model in Figure 1. Each column represents a move, indicating the activity of the trace in the first row and the executed transition of the model in the second row. Moves are executed from left to right to take the trace and the model from the initial state to the end. » is used for asynchronous moves indicating that no action is taken. A move in the model uses » in the trace row to show that only the transition in the model is executed, and a move in the log uses » in the model row. These alignments show the user how there was a skipped required activity that could have been either Class or Test, and the executed activity Test should not have been executed after the final Exam was taken, according to the model. The cost for each of these alignments, assuming a cost function that assigns a cost of 1 to asynchronous moves and 0 to synchronous moves, is 2 in both cases, and their fitness is $f = 1 - \frac{2}{3+3} = 0.\widehat{6}$.

- A synchronous move $(\sigma[i], t)$ is available iff $M[t\rangle$ and $\lambda(t) = \sigma[i]$. This move will update i with the next index of the trace. It will also update M by executing the transition t, i.e., $M - \bullet t + t\bullet$. Note that if the end of the trace was reached ($i = |\sigma|$), no more synchronous moves can be made.
- A log move ($\sigma[i]$,») is available iff $i < |\sigma|$. This will increase i by one, advancing on the trace.
- A model move (», t) is available iff $M[t\rangle$. As a result of this move, M is updated to $M - \bullet t + t\bullet$. If t is a silent transition τ, the move is called a silent move.

Log and model moves are referred to as asynchronous moves.

Definition 7 (Alignment)**.** An alignment $\gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ is a sequence of legal moves (Definition 6). A complete alignment's moves $\gamma_1, \ldots, \gamma_n$ advance through the model and the trace. Alignments start from the initial marking of the model and the start of the trace and reach the end of the model and the end of the trace.

Definition 8 (Cost function, Alignment cost, Optimal alignment)**.** A cost function $C : (A \cup \{\text{»}\}) \times (T \cup \{\text{»}\}) \to (0, \infty)$ assigns a cost to each possible legal move, where A is the set of activities in the log and T is the set of transitions in the process model. Generally, the cost for synchronous moves is lower than the cost for asynchronous moves. Each valid alignment is assigned a cost $c_\gamma = c_{\gamma_1} + \cdots + c_{\gamma_n}$ which is the sum of all the costs of its moves. Hence, the optimal alignment for a given trace and model is the one with the minimum cost. There may be several different optimal alignments.

Figure 3 shows two alignments for the model depicted in Figure 1 alongside a trace. When conducting conformance checking, it's frequently beneficial to provide quality metrics. These metrics offer a concise summary of the results, condensing all calculated alignments into a single, easily interpretable value. One of the most well-established metrics is fitness. It gives an idea of the similarity between the log and the model based on the cost of the computed alignment. Fitness can be calculated for individual alignments or for the entire log. All optimal alignments of a trace have the same fitness as it is derived from the cost of the alignment.

Definition 9 (Alignment fitness)**.** Let γ be an alignment between the trace σ and the model M, with cost

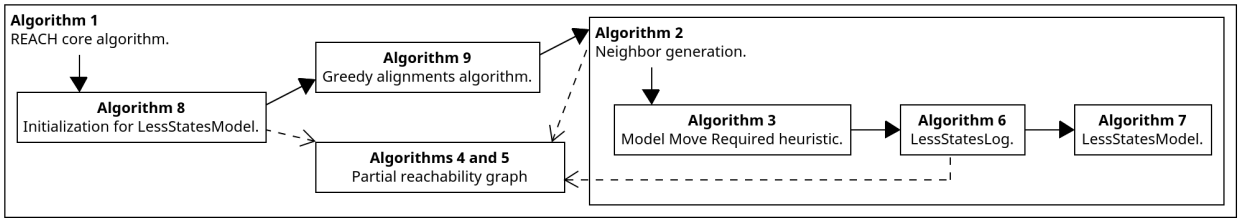


Figure 4: High-level diagram of REACH algorithms and their interactions. Continuous arrows show the control flow of the algorithm, while discontinuous ones show the usage of another algorithm, returning the control flow to the caller.

c_γ (Definition 8). The fitness for that alignment is defined as:

$$f_\gamma = 1 - \frac{c_\gamma}{\sigma_c + M_c}$$

where σ_c is the sum of costs for the asynchronous log moves for all of the trace activities and M_c is the sum of costs of the asynchronous model moves required for the minimum cost path through the model or, in other words, the cost of the optimal alignment of the empty trace and the model.

Fitness indicates how much the trace matches the model, quantifying all differences. Thus, it compares the cost of the given alignment with the cost of the alignment with only asynchronous moves: it first traverses the whole model via its shortest path adding the executed transitions as model moves, and then adds log moves for the whole trace. A fitness of 1 (perfect fitness) shows that the trace was executed correctly for the model (fitting trace), while lower fitness values indicate that discrepancies between the trace and the model were found. To calculate fitness for an entire log, the fitness values of individual traces are averaged. This gives an idea at a glance of how well the log conforms to the process model.

Definition 10 (Log Fitness)**.** Let $\gamma_L = [\gamma_1, \ldots, \gamma_n]$ be a multiset of optimal alignments between each trace of the log $L = [\sigma_1, \ldots, \sigma_n]$ and the model M. The fitness of the log (f) is the average fitness of the alignments: $f = \left(\sum_{i=1}^{n} f_{\gamma_i}\right)/n$, where f_{γ_i} is the fitness of the alignment γ_i (Definition 9).

4. Conformance checking based on alignments

The goal of the REACH algorithm is to find the best alignment or all the best alignments for a model PN and each trace in a log L. Figure 4 shows an overview diagram containing the algorithms detailed in this section and how they relate to each other. Alg. 1 receives the inputs and runs the preprocessing steps of Algs. 8 and 9. Subsequently, the main search loop commences, iteratively invoking Alg. 2 to create neighbors of previously discovered states. Algs. 3, 6 and 7 are optimizations applied each time a neighbor is generated. Some algorithms depend on the partial reachability graph for the execution of operations on the model, as indicated by the discontinuous arrows of Figure 4.

Prior to initiating the processing of each log trace, the algorithm performs some preprocessing tasks. Firstly, once per model, it computes the shortest path through the model following the cost function (Alg. 1:3). While this is not required to find the optimal alignments, it is done to properly compute the fitness metric. Once each optimal alignment is computed, its fitness can be quickly calculated following Definition 9. Secondly, the log is simplified, detecting and counting duplicate traces, so that only one of those is processed. This task is executed only once per log (Alg. 1:4). Lastly, an optimization for reducing the number of states generated that will be explained in Alg. 8 is initialized.

A* is a complete and optimal search algorithm (Hart et al., 1968). This implies that, if the problem has a solution, the algorithm will discover it —complete search algorithm—, and it will always discover the optimal solution if it exists. It stands out for its optimal efficiency with the main drawback being its exponential space complexity. It requires an admissible and consistent heuristic function to select the best path to pursue while guaranteeing optimality. This algorithm fits very well the alignment problem. A directed graph where the nodes are partial alignments is defined, for which each available legal move (Definition 6) generates a new connected neighbor. The start node is the empty partial alignment, which contains no moves. Within this state graph, the algorithm needs to find the minimum cost path between the start and goal nodes, where a goal node is a complete alignment —an alignment that reaches the end of the model and the end of the trace. This is the kind of problem that A* solves with optimal efficiency, meaning that no other optimal algorithm would find the solution expanding fewer nodes if provided the same information.

The proposed conformance checking approach (Alg. 1) is grounded in the A* algorithm (Adriansyah, 2014), a method for navigating the state space to identify the optimal alignment. Each state is a (partially) built alignment, meaning an alignment whose sequence of legal moves begins with the initial marking and the start of the trace. Concretely, states are defined as $S = (M,\, i,\, \gamma,\, c,\, h)$, where:

- M represents the state of execution of the process model. For Petri nets, it is the current marking.
- i is the zero-based index of the trace from which the remaining activities are not yet aligned.
- $\gamma = \langle \gamma_0, \ldots, \gamma_{i-1} \rangle$ is the partial or complete alignment, consisting of the sequence of legal moves taken.
- c is the current cost, which is the sum of the costs of the movements made.
- h is the heuristic value, i.e., an optimistic estimate of the remaining cost to complete the alignment.

Before executing the main algorithm, a greedy search is performed (Alg. 1:15-16). If it finds an alignment, it is used as a maximum cost limit to avoid generating unnecessary states and speed up the algorithm. The greedy search will be discussed further in Section 4.2.2. The algorithm begins with the initial state (Alg. 1:17), which is an alignment without any move, at the initial marking of the model and at the first event of the trace, with cost 0. States are kept in a priority queue that sorts states by increasing order of $S.c + S.h$ (Alg. 1:19), where $S.c$ is the cost and $S.h$ is the heuristic. $S.c + S.h$ is used to guide the exploration of A*, ensuring optimal results. The main loop (Alg. 1:21) explores each state in the order given by the priority queue while a solution is not found. The algorithm can also retrieve the set of all optimal alignments by

Algorithm 1 REACH core algorithm.

Input: The process model PN, the log L to align to PN, and a boolean *opt* that is true to return all optimal alignments, otherwise one of them is given.

Output: Optimal alignments.

1: **procedure** EXECUTE(PN, L, *opt*)
2: $results \leftarrow \emptyset$
3: $PN.leastCost \leftarrow$ SHORTESTPATH(PN) ▷ Needed for computing fitness
4: $L \leftarrow$ SIMPLIFYLOG(L)
5: LESSSTATESMODELINIT(PN) ▷ Alg. 8
6: **for all** $\sigma \in L$ **do**
7: $alignments \leftarrow$ ALIGN(PN, σ, *opt*) ▷ Align the trace
8: $results \leftarrow$ AGGREGATE($results$, $alignments$) ▷ Aggregate the results
9: **end for**
10: **return** $results$
11: **procedure** SHORTESTPATH(PN)
12: $S \leftarrow$ ALIGN(PN, EMPTYTRACE, *false*)[0] ▷ Align the empty trace, returning the final state
13: **return** $S.c$ ▷ Return the cost c of the optimal alignment
14: **procedure** ALIGN(PN, σ, *opt*)
15: $greedyAlignment \leftarrow$ ALIGNGREEDY(PN, σ) ▷ Alg. 9
16: $maxCost \leftarrow greedyAlignment.c$ ▷ The cost of the greedy alignment limits the optimal cost
17: $S \leftarrow$ INITIALSTATE(PN, σ)
18: $open \leftarrow \emptyset$ ▷ Priority queue that stores states
19: ADD($open, S$) ▷ Insert state, sorted by $S.c + S.h$
20: $alignments \leftarrow \emptyset$ ▷ Found optimal alignments
21: **while** $alignments = \emptyset$ || (opt && $S.c + S.h < min(\{S.c \mid S \in alignments\} \cup \{+inf\}) + min\_c$) **do**
22: $S \leftarrow$ POLL($open$) ▷ Extract the next best state
23: **if** ¬ISFINAL(S) **then**
24: ADDNEIGHBORS($\sigma, S, open, maxCost$) ▷ Alg. 2
25: **else**
26: $alignments \leftarrow alignments \cup S$ ▷ Record the final state of the optimal alignment
27: **end while**
28: **return** $alignments$

iterating while $S.c + S.h$ is lower than the cost of any found solution ($min(\{S.c \mid S \in alignments\} \cup \{+inf\})$) plus the minimum cost of an asynchronous move (min_c) —as A* requires synchronous moves to have a negligible cost strictly greater than 0. At each step, a state is removed from the priority queue (Alg. 1:22). If the given state is not final, the algorithm creates the neighbors of that state by using all the feasible legal moves (Definition 6). For each neighbor, it is necessary to update the process model state, the trace progress, the cost, and the heuristic. Otherwise, if the state is final, it is an optimal alignment and it is added to the alignments set (Alg. 1:23-26).

The ADDNEIGHBORS function detailed in Alg. 2 adds all the children states from a parent based on the available legal moves —called from Alg. 1:24. Given the set of enabled transitions of the parent state (Alg. 2:2), and the next trace activity of the parent state (Alg. 2:4), neighbors are created by executing all available legal moves. Specifically, one neighboring state is generated for each move:

Algorithm 2 Neighbor generation.

Input: A current or parent state S and a priority queue *open* for the new states to be inserted into.
Output: None, it adds neighbors to *open*.

1: **procedure** AddNeighbors(σ, S, *open*, *maxCost*)
2: $etrs \leftarrow$ EnabledTransitions($S.M$) ▷ Alg. 4: enabled transitions from the current state
3: **if** $S.i < |\sigma|$ **then** ▷ If the end of σ was not reached yet
4: $activity \leftarrow \sigma[S.i]$ ▷ The next activity recorded in the trace
5: $trs \leftarrow \lambda_r(activity)$ ▷ An activity may map to multiple model transitions
6: **for all** $tr \in trs \cap etrs$ **do**
7: AddMove(σ, S, *open*, Sync, tr, *maxCost*) ▷ Generate synchronous movements
8: **end for**
9: **if** $\neg$LessStatesLog(σ, S) **then** ▷ Alg. 6
10: AddMove(σ, S, *open*, Log, *activity*, *maxCost*) ▷ Generate asynchronous log movements
11: **if** $\neg$LessStatesModel(σ, S) **then** ▷ Alg. 7
12: **for all** $tr \in etrs$ **do**
13: AddMove(σ, S, *open*, Model, tr, *maxCost*) ▷ Generate asynchronous model movements
14: **end for**
15: **procedure** AddMove(σ, S_{parent}, *open*, *type*, tr, *maxCost*)
16: $S \leftarrow$ NewState()
17: **if** *type* = Sync **then** ▷ Record the performed legal move in γ for the current state
18: $S.\gamma \leftarrow S_{parent}.\gamma \mathbin{+\!\!+} (\lambda(tr), tr)$
19: **if** *type* = Log **then**
20: $S.\gamma \leftarrow S_{parent}.\gamma \mathbin{+\!\!+} (tr, \gg)$
21: **if** *type* = Model **then**
22: $S.\gamma \leftarrow S_{parent}.\gamma \mathbin{+\!\!+} (\gg, tr)$
23: **if** *type* $\neq$ Model **then** ▷ Synchronous or log movements advance on the trace
24: $S.i \leftarrow S_{parent}.i + 1$
25: **if** *type* $\neq$ Log **then** ▷ Synchronous or model movements advance on the model
26: $S.M \leftarrow$ ExecuteTransition($S_{parent}.M, tr$) ▷ Alg. 5
27: **if** *type* $\neq$ Sync **then** $S.c \leftarrow S_{parent}.c + 1$ ▷ Update cost and heuristic for the new state
28: **else** $S.c \leftarrow S_{parent}.c + epsilon$
29: $S.h =$ Heuristic(σ, S) ▷ Alg. 3
30: **if** ShouldAdd(S, *open*, *maxCost*) **then** ▷ Add the state to the queue
31: Add(*open*, S)
32: **return** S

- A synchronous move for each enabled transition of the model that shares the label with the next activity of the trace —note that model transitions can have duplicate labels— (Alg. 2:6-7).
- An asynchronous movement in the log if the end of the trace is not yet reached (Alg. 2:10).
- As many asynchronous movements in the model as transitions are enabled from the current marking of the model (Alg. 2:12-13).

The conditions at Alg. 2:9 and Alg. 2:11 are optimizations. These optimizations reduce the exploration of unnecessary states by skipping certain asynchronous moves in the log and the model when specific conditions are met (Section 4.2.2). For each move, a new neighboring state S must be created and $S.\gamma$ must be updated, the sequence of legal moves (Alg. 2:15-22). Each call to AddMove defines the new state that can advance

to the next activity on the trace, and/or execute a transition in the model, depending on the kind of move (Alg. 2:23-26). The cost and heuristic values are then updated for the new state, just before inserting it in the priority queue, where it will be sorted by the sum of both values (Alg. 2:27-31). The condition at Alg. 2:30 avoids inserting in the open queue states that are not capable of reaching the optimal alignment, as they match a previously discovered state —equal $S.M$ and $S.i$ values— with a higher or equal $S.c$, or they exceed the $S.c + S.h$ threshold established by the greedy algorithm.

To achieve optimality, the A* algorithm employed in REACH must meet three conditions. First, the cost change when advancing to a neighbor must be strictly positive. To satisfy this requirement, the cost of synchronous and silent moves is always a negligible value greater than 0 (*epsilon*). A standard cost of 1 is applied to all other asynchronous moves. Second, the heuristic must be admissible, meaning that it must return an underestimate of the remaining cost to the closest goal node in terms of cost. In third place, the heuristic must also be consistent —also called monotone— in order to provide optimal results. A consistent heuristic returns an estimate for any node that is lower or equal to the least cost of advancing to a neighbor plus its heuristic estimate, $S_{parent}.h \leq S.c - S_{parent}.c + S.h$, where S is a state successor of S_{parent}. This is because using a consistent heuristic ensures that once a node is explored, it will not be reached again with a lower cost. Note that a consistent heuristic is also admissible. The admissibility and monotonicity of the proposed heuristic will be discussed in Section 4.1.

4.1. Heuristic

Heuristics substantially affect the performance of the A* algorithm and, as such, are the focus of many state-of-the-art papers. We propose a new heuristic, called Model Move Required (MMR) that balances the time required to compute the heuristic —which would mean a higher processing time per state—, and the accuracy of the estimates it provides —which enables reducing the number of states that need to be explored to achieve optimality. It works by finding a subset of the required transitions —transitions that must be executed to reach the end of the model— and by comparing their labels to the remaining trace activities.

Alg. 3 presents the MMR heuristic, which calculates an optimistic approximation of the cost to complete an alignment from a given state. The heuristic needs to identify a subset of the non-silent transitions that are necessary to execute from the current marking to reach a final state (Alg. 3:2). The first step is to figure out the required transitions for the state for which the heuristic will be computed (Alg. 3:8). Each token of that state marking is added to the queue of *places* to visit (Alg. 3:12-14). Then, the main loop starts visiting each *place* of that queue until it is empty (Alg. 3:15). Each visited *place* is removed from the queue, and already visited places are skipped or otherwise marked as visited (Alg. 3:16-19). Next, *place*• is retrieved, i.e., the successor *transitions* of *place*. If the list only contains one transition, it is registered as a required transition (Alg. 3:20-24). All the successor places of the transition are added to the queue of *places* to visit (Alg. 3:25). Figure 5 presents an example that illustrates the behavior of the required transitions procedure.

Algorithm 3 Model Move Required heuristic

Input: An state S.
Output: The heuristic value.

procedure HEURISTIC(σ, S)
  $requiredTrs \leftarrow$ REQUIREDTRANSITIONS($S.M$) ▷ Subset of the required transitions (cached)
  $required \leftarrow \{\lambda(t) \mid t \in requiredTrs\}$ ▷ Set of required unique activities of the model
  $remaining \leftarrow$ TOSET($\sigma[S.i:]$) ▷ Set of unique activities of the trace suffix starting at $S.i$
  $missing \leftarrow required \setminus remaining$ ▷ Set of required activities not found in the remaining trace
  $minCostMoves \leftarrow remaining \setminus missing$ ▷ Set of required extra moves, assumed synchronous
  return $|missing| + |minCostMoves| * epsilon$ ▷ Heuristic, under the standard cost function
procedure REQUIREDTRANSITIONS(M)
  $requiredTransitions \leftarrow \emptyset$ ▷ The output of this function is the set of required transitions
  $places \leftarrow \emptyset$ ▷ Queue that stores places to visit
  $placesVisited \leftarrow \emptyset$ ▷ Queue that stores visited places
  for all $place \in$ TOKENPLACES(M) **do** ▷ For each token place in the current marking
    ADD($places$, $place$) ▷ Insert the initial place in the queue to visit it later
  end for
  while ¬EMPTY($places$) **do** ▷ While there are more places to visit
    $place \leftarrow$ POLL($places$) ▷ Extract the next place to explore
    if $place \in placesVisited$ **then**
      continue ▷ Skip the place if already visited
    ADD($placesVisited$, $place$) ▷ Mark the place as visited
    $transitions \leftarrow place\bullet$ ▷ Get the successors of the place
    if $|transitions| \neq 1$ **then**
      continue ▷ Skip the place if it has more than one output arc or no output arcs
    if ¬ISSILENT($transitions[0]$) **then** ▷ If the transition is not silent
      ADD($requiredTransitions$, $transitions[0]$) ▷ Register the transition as required
    ADDALL($places$, $transitions[0]\bullet$) ▷ Queue each output place for exploration
  end while
  return $requiredTransitions$

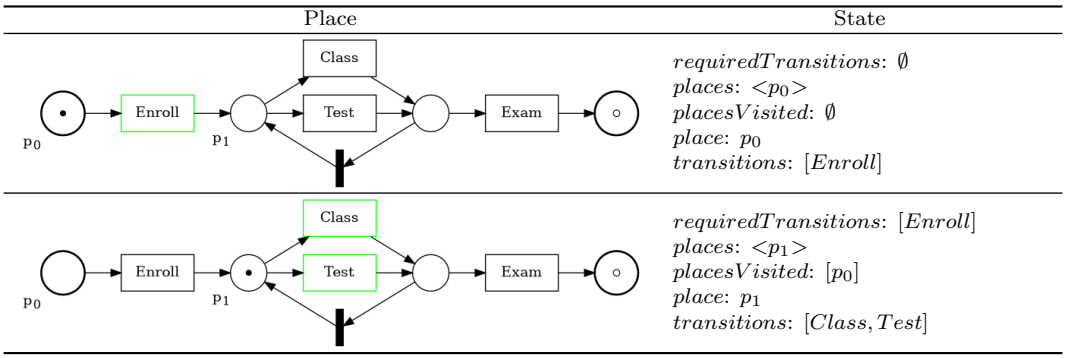


Figure 5: Iterations of RequiredTransitions (Alg. 3:8) for the initial state of the running example. For each iteration —row of the table— of the main while loop (Alg. 3:15), the place being explored is highlighted on the left column and the state of the defined variables is shown on the right column. The *requiredTransitions*, *places* and *placesVisited* variables show the values before the iteration, while the *place* and *transitions* variables show the values retrieved during the iteration. The initial marking only has one token on p_0, which is added to *places*. The *place* to visit in the first iteration (p_0) is extracted from *places*. This iteration checks that the successor of p_0 is only one transition. As this is true ($|transitions|= 1$), it marks the transition (*Enroll*) as required and adds all output places of the transition ($[p_1]$) to *places*. The second iteration processes p_1 as it is the first element of the *places* queue. It has two successors, so it does not add any more places to *places*. It does not mark them as required as only one of them has to be executed so neither is mandatory. The *places* queue is empty, so the algorithm stops. The only required transition found for this simple example is *Enroll*.

When the queue of *places* to visit is empty, the algorithm collected a subset of all the required transitions. Following the collection of the required transitions, their unique labels are compared with the remaining distinct activities in the trace to calculate the heuristic (Alg. 3:4). For each mandatory model activity that does not appear in the remaining trace (Alg. 3:5), the cost of the asynchronous move is added to the heuristic. These activities are required to be executed in order to reach the end of the model, so not having them in the remaining trace implies at least another move in the model in order to reach the end. For each of the unmatched trace activities remaining (Alg. 3:6), the cost of a synchronous move is added, as another move with a minimum cost of a synchronous move has to be made for each of them in order to complete the alignment.

In order for this heuristic to be valid, there must be only one reachable final marking of the model: one token in the end place. This condition, known as proper completion, guarantees that all transitions identified through the heuristic procedure will be necessary to reach the end of the model. Hence, this heuristic assumes that the model is a workflow net with proper completion —without the soundness requirement. This is a consistent heuristic that uses the standard cost function: assigns a positive value very close to 0 (*epsilon*) for synchronous moves and a cost of 1 to asynchronous moves. The number of unmatched required activities can only decrease by a maximum of 1 between parent and child states (S_{parent} and S), and it may only decrease by 1 when the cost increases by 1 by performing an asynchronous move ($S.c - S_{parent}.c$), so the consistency condition ($S_{parent}.h \leq S.c - S_{parent}.c + S.h$) is verified.

Algorithm 4 Partial reachability graph: enabled transitions.

Input: The current marking M. The single partially built reachability graph p for the workflow net is received as input even if the calling pseudocode does not specify it (p is initialized as an empty map). Assumes that EnabledTransitions is called before ExecuteTransition for any marking.

Output: The set of enabled transitions of the marking M.

```
1: procedure EnabledTransitions(M, p)
2:     if M ∈ p then                          ▷ If M had its enabled transitions listed previously
3:         etrs ← {t | (t, m) ∈ p[M]}         ▷ Retrieve the list of enabled transitions from p
4:     else
5:         etrs ← {t | M[t⟩}                  ▷ Compute the enabled transitions from M
6:         p[M] ← {(etr, null) | etr ∈ etrs}  ▷ Record the enabled transitions, mapped to null
7:     return etrs
```

Algorithm 5 Partial reachability graph: execute transition.

Input: The current marking M and the transition to execute t. The single partially built reachability graph p for the workflow net is received as input even if the calling pseudocode does not specify it (p is initialized as an empty map). EnabledTransitions assumes that EnabledTransitions is called before ExecuteTransition for any marking.

Output: The new marking M' after executing t on the marking M.

```
1: procedure ExecuteTransition(M, t, p)
2:     if p[M][t] ≠ null then                 ▷ If t was previously executed from M
3:         M' ← p[M][t]                       ▷ Retrieve the next marking from the partial reachability graph
4:     else
5:         M' ← M − •t + t•                   ▷ Compute the new marking after executing the t transition
6:         p[M][t] ← M'                       ▷ Record the transition execution in the partial reachability graph
7:     return M'
```

4.2. Optimizing the algorithm

This section focuses on our contributions to the core A*-based alignments algorithm: *(i)* partial reachability graph, and *(ii)* several states reduction optimizations.

4.2.1. Partial reachability graph

All state-of-the-art algorithms need to execute the workflow net to be able to compute optimal alignments. Thus, all algorithms perform operations over the workflow net for creating an initial marking (Alg. 1:17), listing all enabled transitions for a marking (Alg. 2:2), executing a transition (Alg. 2:26) and checking if a marking is final (Alg. 1:23). Those operations affect performance, as they are executed several times for each iteration of A*. Our approach introduces a significant difference from the state of the art. We propose the dynamic construction of a partial reachability graph (PRG) as new model markings are reached. This change is aimed at leveraging information from prior model operations to enhance the speed of future iterations. To make the execution of the workflow net faster, a directed graph is created that shows the different explored states of the system. This graph is stored in p, which is initially empty (Alg. 4). When explored, each marking

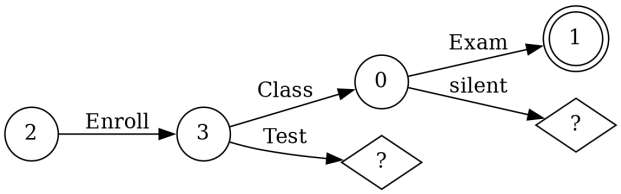


Figure 6: Example of the partial reachability graph computed while executing the trace ⟨Enroll, Class, Exam⟩ on the running example (Figure 1). Before executing each transition of the example trace, all enabled transitions of the marking are computed. The nodes of the graph are markings. Known markings are circles and unexplored ones are diamonds. Final markings are represented with a double circle. The node contents show a unique identifier if explored and a question mark otherwise. Each arc between two nodes is labeled with the enabled transition whose execution generates the target marking from the source marking.

is stored as an entry in p and represents a vertex in the graph (Alg. 4:5-6). When each marking M is explored, each enabled transition from M is represented by an entry in the $p[M]$ mapping, which maps to the new marking after executing the transition (Alg. 5:5-6), or null if it was never executed (Alg. 4:5-6). The enabled transitions serve as arcs connecting states within the graph. This builds at runtime a data structure similar to a reachability graph (Davidrajuh, 2013). Nevertheless, only the explored states are generated. Hence, it avoids building the full reachability graph which would be a computationally intensive task, especially for models with lots of branching and concurrent transitions. Instead, it only stores information about states that are needed in the algorithm's exploration. Once repeated model operations start occurring, e.g. due to loops in the model or to the beginning of the alignment computation over a new trace of the log, the partial reachability graph can quickly access enabled transitions and new markings (Alg. 4:2-3 and 5:2-3). This information is kept while aligning all the traces of the log to the model, which is the main reason for the speedup. Figure 6 shows the partial reachability graph for the running example and a simple trace.

The advantages of this PRG become evident as the algorithm progresses and repeated model operations become commonplace. For instance, when loops occur within the model or when aligning a new trace from the log, the PRG swiftly facilitates access to enabled transitions and the corresponding new markings.

4.2.2. State reduction-based optimizations

The primary challenge in computing conformance checking alignments using the A* algorithm is the substantial number of generated states that must be stored and evaluated to ensure optimal results. To mitigate this problem, we propose new optimizations focused on states reduction that will alleviate the state explosion that occurs for complex datasets: *(i)* States Reduction forcing asynchronous Model moves (SRModel), *(ii)* States Reduction forcing asynchronous Log moves (SRLog), and *(iii)* an initial greedy search that finds a cost limit.

The SRModel optimization is only applicable when seeking a single optimal alignment, and it effectively reduces the number of generated neighbors by constraining allowed movements (Alg. 2:9, Alg. 6). Thus, if the current state has enabled transitions and their activities do not appear in the remaining trace (Alg. 6:5), this

Algorithm 6 LessStatesLog (SRModel).

Input: The trace σ and an state S.
Output: A boolean indicating whether to skip generating log move states from S.

1: **procedure** LessStatesLog(σ, S)
2: **return**
3: $|\text{EnabledTransitions}(S.M)| > 0$
4: and $|\text{EnabledTransitions}(S.M) \cap$
5: $\bigcup\{\lambda_r(act) \mid act \in \sigma[S.i:]\}| = 0$

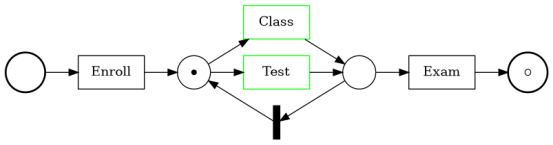
Algorithm 7 LessStatesModel (SRLog).

Input: The trace σ and an state S, for which AliveActivities returns the alive activities from that state (Alg. 8).
Output: A boolean indicating whether to skip generating model move states from S.

1: **procedure** LessStatesModel(σ, S)
2: **return**
3: $\neg$LessStatesLog(σ, S) and $i < |\sigma|$ and
4: $\sigma[i] \notin$ AliveActivities($S.M$)

Log:	Enroll
Model:	Enroll

(a) State represented as a partial alignment.



(b) The model marking of the state.

Figure 7: SRModel optimization example for the trace ⟨Enroll, Exam⟩ and the model from Figure 1. The explored state of (a) would normally generate three asynchronous moves: one model move for each of the enabled model transitions, and one log move on the Exam activity of the trace. However, the SRModel optimization checks whether the enabled model transitions cannot be executed in the remaining trace. As this is true for the given state, it can force a model move by only generating two of the three neighboring states.

optimization can be activated: the asynchronous move in the log can be avoided, as the optimal alignment must have a model move on one of the enabled transitions. This optimization always provides an optimal alignment because a synchronous or model move is required to reach the end of the model, but no synchronous move is or will ever be available —the remaining trace activities do not match the labels of the currently executable transitions in the model— until a model move is performed. Forcing the model moves in this situation reduces the number of states to explore without affecting the optimality of the algorithm. An example of this optimization is shown on Figure 7.

Similarly to SRModel, the SRLog optimization reduces the number of states to explore by studying the remaining trace and model. It identifies the alive transitions of the model, which are those that can be enabled from the current marking through valid transition execution sequences. Alive activities are the unique labels of the alive transitions. SRLog checks if the next activity of the trace —which must have one or more activities left— is not one of the alive activities of the current state. In this case, the algorithm forces an asynchronous movement on the log (Alg. 2:11, Alg. 7) because that movement has to be made in order to reach a complete alignment. This optimization always provides an optimal alignment because a synchronous or log move is required to reach the end of the trace, but no synchronous move is or will ever be available —the alive activities of the model do not match the next activity in the trace— until a log move is performed. By forcing the algorithm to make it as soon as possible, all states that would otherwise need to be generated later are being avoided. In cases where both SRModel and SRLog are applicable to a particular state, we

exclusively employ SRModel. The simultaneous application of both would result in the filtering out of all neighboring states. SRLog is applied to the generated neighbors if SRModel is not available for them. SRLog requires an additional model pre-processing task in which all the alive activities of all states are stored. This pre-processing task is only needed once per model, and it is performed in two phases as is shown in Alg. 8.

The initial phase of SRLog involves exploring the model and recording all the unique reachable markings within the workflow net. Each of those markings is associated with a single state, which also includes information about the activities that are directly enabled, and the predecessor states (Alg. 8:2). This exploration phase stops when a repeated marking or a marking without enabled transitions is reached.

In this phase, a list to store the discovered states that still need to be processed is used (Alg. 8:5). Each iteration explores a new discovered state, and the loop stops when there are no more states in the queue (Alg. 8:9-10). The first action for each discovered state is to check if its marking was previously explored, registering it otherwise. This is performed by PutIfAbs, which also returns the previously stored state for that marking if it exists (Alg. 8:11). This previously stored state will later be used to merge already discovered predecessor states and enabled activities into the single state associated with that shared marking (Alg. 8:25-27). To further explore new states, it is necessary to iterate over each enabled transition of the current state which might lead to a new child state (Alg. 8:13-15). It is checked if the current child marking is new by comparing to previously explored markings. In either case, the label of the transition is registered as an alive activity of the parent state and the predecessors of the child state are updated (Alg. 8:16-22).In the event that the child state is indeed new, it is added to the state queue to be explored later (Alg. 8:23). The final state is also recorded, which will be the starting point for phase 2 of this algorithm (Alg. 8:29).

During the second phase of SRLog, the algorithm calculates all alive activities associated with each marking. These are activities that might be executed at any point in the future, even if they are not enabled at this point and require executing other activities first. This is done by recursively inheriting all the alive activities from states to their predecessors, starting from the recorded final state. The *states* list is initialized to the final state (Alg. 8:32). The main loop starts, taking the last state from the *states* list as long as it is not empty (Alg. 8:33). For each explored state, S_{prev} is not null if and only if $S.M$ was previously explored, and $S.M$ is marked as explored (Alg. 8:35). Then, all predecessor states of S inherit the alive tasks from S (Alg. 8:38). The predecessors are appended to the end of the *states* list if they were not previously explored or if new enabled tasks were found (Alg. 8:40). Figure 8 shows step-by-step the execution of the initialization of SRLog over the running example. In addition, Figure 9 shows the execution of the SRLog algorithm, using the same model and an example trace.

Note that both SRModel and SRLog have no relation to the log-move-first and model-move-first optimizations defined in (Carmona et al., 2018). The optimizations we introduce involve an analysis of the remaining trace and/or model, identifying essential moves and actively enforcing them. This stands in contrast to (Carmona et al., 2018) in which only the last alignment move is examined to dictate the sequence

Algorithm 8 Initialization for LessStatesModel (SRLog).

Input: A process model PN.
Output: The map from each reachable marking to the set of alive activities in the model.

```
1:  procedure LessStatesModelInit(PN)
2:      S ← { M : InitialMarking(PN),
3:            alive : ∅,                                   ▷ Alive activities from S
4:            pred : ∅ }                                   ▷ Predecessor states
5:      states ← list()                                    ▷ Unexplored state list
6:      explored ← map()                                   ▷ Marking to state mapping
7:      finalState ← null
8:      Add(states, S)
9:      while |states|> 0 do                               ▷ Phase 1: discovery
10:         S ← PollLast(states)          ▷ Retrieve and remove the last element of the states list.
11:         S_prev ← PutIfAbs(explored, S.M, S)   ▷ Check if the state already exists and save it otherwise
12:         etrs ← EnabledTransitions(S.M)                 ▷ Alg. 4
13:         for all etr ∈ etrs do                          ▷ Explore all enabled transitions
14:             S_c ← NewState(S)
15:             S.M ← ExecuteTransition(S_parent.M, tr)    ▷ Alg. 5
16:             S_back ← Get(explored, S_c.M)              ▷ Find out if this is a previously explored marking
17:             if S_back ≠ null then          ▷ Handle discovered loops to previous states without recursion
18:                 if etr ∈ T_ns then S.alive ← S.alive ∪ {λ(etr)}
19:                 S_back.pred ← S_back.pred ∪ {S}
20:             else       ▷ Handle new states by also updating alive and pred, and adding them to states
21:                 if etr ∈ T_ns then S.alive ← S.alive ∪ {λ(etr)}
22:                 S_c.pred ← S_c.pred ∪ {S}
23:                 AddLast(states, S_c)      ▷ Queue for exploration by adding them at the end of states
24:         end for
25:         if S_prev ≠ null then          ▷ Merge collected data if returning to a previously explored state
26:             S_prev.alive ← S_prev.alive ∪ S.alive
27:             S_prev.pred ← S_prev.pred ∪ S.pred
28:         else if IsFinal(S.M) then                      ▷ Remember the final state
29:             finalState ← S
30:     end while
31:     explored ← map()                                   ▷ Marking to state mapping
32:     Add(states, finalState)
33:     while |states|> 0 do                               ▷ Phase 2: alive activities
34:         S ← PollLast(states)
35:         S_prev ← PutIfAbs(explored, S.M, S)   ▷ Check if the state already exists and save it otherwise
36:         for all S_c ∈ S.pred do                        ▷ Explore predecessor states
37:             prevAlive ← S_c.alive                      ▷ Remember previously alive activities
38:             S_c.alive ← S_c.alive ∪ S.alive            ▷ Inherit previously alive activities from successor
39:             if S_prev = null or S_c.alive ≠ prevAlive then   ▷ Check stop condition
40:                 AddLast(states, S_c)                   ▷ Mark for exploration
41:         end for
42:     end while
43:     return explored
```

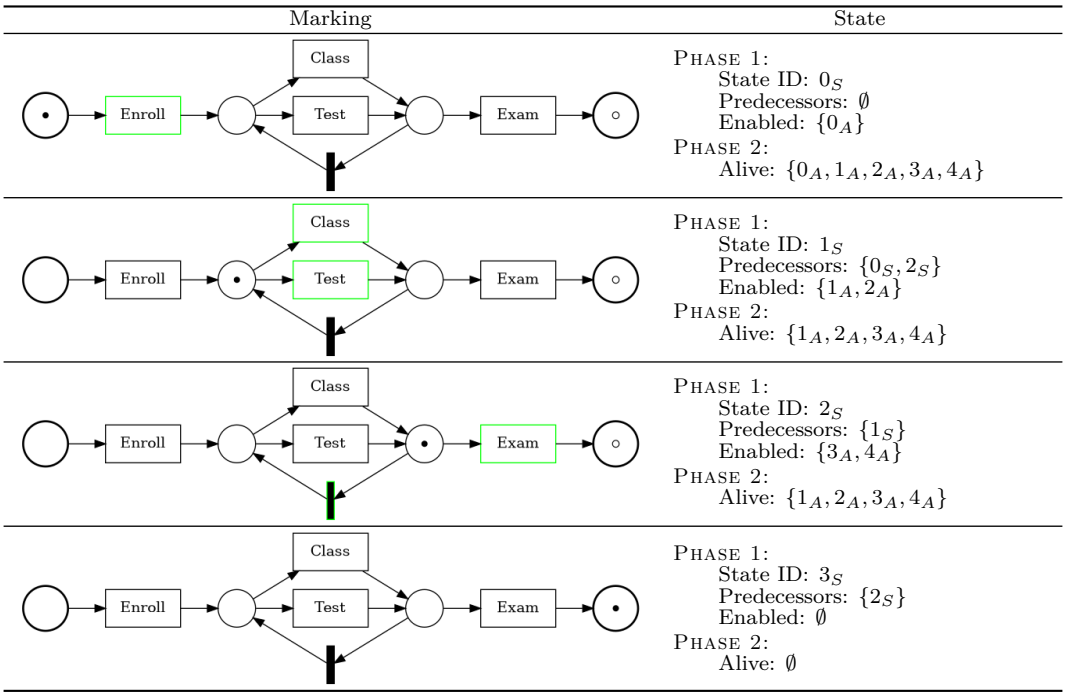
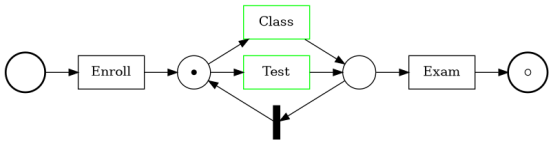


Figure 8: Execution steps of Alg. 8 (initialization for SRLog) for the model of the running example. During the first phase, the following information for each state is obtained: its ID, the IDs of its predecessor states, and the directly enabled activities. The second phase completes the full set of alive activities by inheriting enabled activities from successors. The S and A subscripts help to distinguish between state and activity IDs respectively.

Log:	Enroll
Model:	Enroll

(a) State represented as a partial alignment.



(b) The model marking of the state.

Figure 9: SRLog optimization example for the running example from Figure 1 and the trace ⟨Enroll, Enroll, Class, Exam⟩. The alive transitions computed using Alg. 8 (Figure 8) for the current state are Class, Test, the silent one, and Exam. The explored state of (a) would normally generate three moves: one model move for each of the enabled model transitions, and one log move on the Enroll event of the trace. However, the SRLog optimization checks whether the Enroll event is not alive in the model. As this is true for the given state, it can force a log move by only generating one of the three neighboring states.

of log and model moves. Both SRModel and SRLog can be enabled at the same time during the execution of the algorithm for improved performance, whereas only one of log-move-first or model-move-first can be applied on each execution. Note that neither log-move-first nor model-move-first are compatible with the SRModel or SRLog optimizations.

Before the primary alignments algorithm, the greedy search optimization (Alg. 9) is employed to quickly establish an upper cost bound for the optimal alignment. The algorithm starts from the same initial state as the REACH algorithm (Alg. 9:2-5). It records a tuple of the visited marking and trace progress to quickly advance avoiding infinite loops (Alg. 9:6). It generates the neighbors of the initial state also following the

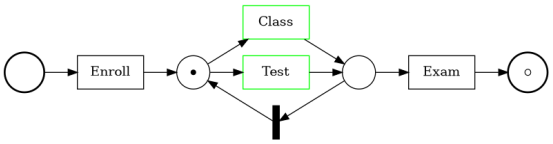
Algorithm 9 Greedy alignments algorithm.

Input: The process model PN and a trace σ.
Output: The final state of the greedy alignment, or null if no greedy alignment is found.

```
1:  procedure AlignGreedy(PN, σ)
2:      S ←InitialState(PN, σ)
3:      visited ← ∅
4:      alignment ← null
5:      while alignment = null and S ≠ null do          ▷ Explore states until stop condition is matched
6:          Add(visited, (S.M, S.i))                    ▷ Mark the tuple of the marking and the trace index as visited
7:          if ¬IsFinal(S) then
8:              open ← ∅                                ▷ Reset the open queue on each iteration, disabling backtracking
9:              AddNeighbors(S, open)                   ▷ Alg. 2: consider all neighbors as usual
10:             while S ≠ null do
11:                 S ← Poll(open)                      ▷ Extract them by priority
12:                 if ¬Contains(visited, (S.M, S.i)) then   ▷ Ignore already visited neighbors
13:                     break
14:             end while
15:         else
16:             alignment ← S                           ▷ Use it as the result if final
17:     end while
18:     return alignment
```

Log:	Enroll
Model:	Enroll

(a) State represented as a partial alignment.



(b) The model marking of the state.

Figure 10: Greedy alignment optimization example for the running example from Figure 1 and the trace <Enroll, Class, Exam>. The greedy alignment computed using Alg. 9 is made of three synchronous moves. The explored state of (a) would normally generate four moves: one synchronous move, one model move for each of the enabled model transitions, and one log move on the Class event of the trace. However, the greedy alignments optimization checks whether the cost of each generated neighbor is greater than the cost of the greedy alignment. As this is true for all of the asynchronous moves of the given state, it can force a synchronous move by only generating one of the four neighboring states.

REACH algorithm (Alg. 2), and adds them to the newly created *open* queue (Alg. 9:8-9). Considering only the neighbors generated on that iteration, the one of minimum $S.c + S.h$ is chosen (Alg. 9:11), i.e., the greedy algorithm performs a Best First Search guided by the heuristic. If it was already visited (Alg. 9:12), it goes back to Alg. 9:10 to process the next best state. If the chosen state was not visited before, the algorithm moves to that state and continues the exploration from there. It does so until a complete alignment is achieved or until there are no more unvisited states left to explore (Alg. 9:5). The cost of this suboptimal greedy alignment is used as the upper bound of $S.c + S.h$ for the optimal alignment retrieved by the REACH algorithm, filtering states that will not lead to the optimal solution. Figure 10 shows an example of how this filtering is made.

5. Evaluation

We have extensively evaluated our approach and compared it against other state-of-the-art algorithms. We provide REACH as a web service, as a binary executable and as the original source code[1] to allow the replication of results. In this section, we describe the datasets and the experiments, discussing the results.

5.1. Setup

To ensure consistency, we developed and tested REACH in Java 8, aligning it with the Java-based implementations of the state-of-the-art algorithms. All the algorithms have been executed in an Oracle Java 8 Virtual Machine in a computer equipped with an Intel Core i5-9600K, 32 GB RAM, 1024 GB SSD, and Ethernet 1Gb BaseT. For the execution of each algorithm, a maximum of 24 GB RAM was reserved.

5.2. Logs and process models

The algorithm takes two inputs: a log and a model, referred to as a log-model pair for the sake of simplicity. We used logs and models from the following published papers:

- (Reißner et al., 2020b) includes 17 logs extracted from different years of the Business Process Intelligence Challenge (BPIC) and from 4TU.ResearchData[2]. They are taken from different domains such as healthcare, government, finance and IT service management, with different levels of complexity to create a fair test environment for conformance checking algorithms.
  The process models were discovered with the Inductive Miner infrequent algorithm (Leemans et al., 2014). Discovering models that perfectly fit the log would be too easy to solve as the optimal alignment would only require synchronous moves. To increase the difficulty and test the performance limits of the state of the art, we discover models with various fitness levels. These models differ in the extent to which they describe the log behavior, with lower percentages indicating lower fitness levels. Achieving optimal alignments generally becomes more challenging as the fitness decreases. Thus, for each log, 10 models will be discovered, ranging from 10% to 100% of behavior[3]. For the log BPIC18, the discovery algorithm could not obtain a model for percentages over 30%, so only 3 models were discovered. The total of log-model pairs extracted from this source is 163.
- (van Dongen, 2018) includes logs and models, so no model discovery algorithm was required. The main advantage of including these log-model pairs is that they include models discovered with different algorithms and even large artificial models. From this source, we expanded our dataset with an additional 64 log-model pairs.

[1] `https://tec.citius.usc.es/reach`
[2] `https://data.4tu.nl/`
[3] In this paper, when we mention the percentage of considered behavior, we are referring to the complement of the threshold parameter of the Inductive Miner infrequent algorithm.

Table 2: Logs information. The columns are: number of activities (A), number of traces (T), total number of events (E), minimum number of events per trace (N), and maximum number of events per trace (X). Note that the S subindex indicates that redundant traces are ignored.

Name	A	T	E	T_S	E_S	N	X
BPIC12	24	13,087	262,200	4,366	182,467	3	175
BPIC13CP	4	1,487	6,660	183	1,810	1	35
BPIC13INC	4	7,554	65,533	1,511	29,010	1	123
BPIC14F	9	41,353	369,485	14,948	232,067	3	167
BPIC15F1	70	902	21,656	295	10,367	5	50
BPIC15F2	82	681	24,678	420	17,820	4	63
BPIC15F3	62	1,369	43,786	826	28,553	4	54
BPIC15F4	65	860	29,403	451	16,502	5	54
BPIC15F5	74	975	30,030	446	18,789	4	61
BPIC17F	18	21,861	714,198	8,767	333,965	11	113
BPIC18	41	43,809	2,514,266	28,457	1,819,830	24	2,973
BPIC19_1	11	1,044	5,898	148	1,619	2	21
BPIC19_2	38	15,182	319,233	4,228	255,982	1	990
BPIC19_3	39	221,010	1,234,708	7,832	82,611	1	179
BPIC19_4	15	14,498	36,084	281	1,614	1	17
RTFMP	11	150,370	561,470	231	1,891	2	20
SEPSIS	16	1,050	15,214	846	13,775	3	185
sepsis (Mannhardt, 2016)	16	1,050	15,214	846	13,775	3	185
Fitting logs (Maruster et al., 2006)	42	4,000	83,402	2,935	76,482	5	102
Noisy logs (Maruster et al., 2006)	42	16,000	322,670	12,051	298,281	2	102
Fitting logs (Munoz-Gama, 2013)	317	1,200	49,792	1,126	48,573	14	59
Noisy logs (Munoz-Gama, 2013)	363	5,300	638,555	5,149	633,965	15	245
Fitting logs (Munoz-Gama, 2014)	110	34,000	1,062,208	22,649	906,695	12	167
Noisy logs (Munoz-Gama, 2014)	68	32,000	1,035,889	22,747	910,515	12	147
bpi12 (van Dongen, 2012)	24	13,087	262,200	4,366	182,467	3	175
road_fines (de Leoni and Mannhardt, 2015)	11	150,370	561,470	231	1,891	2	20

Adding the log-model pairs obtained from (Reißner et al., 2020b) and (van Dongen, 2018) results in a total of 227 log-model pairs. The complete set of logs used in our evaluation, along with relevant statistics showcasing their diversity, is presented in Table 2.

5.3. Impact of the optimizations in REACH

In this section, we analyze the effect of the proposed optimizations on the performance of the A* algorithm by selectively enabling and disabling the optimizations. Note that we always check the optimality of the algorithm for each experiment by comparing the returned alignment costs against other versions of the algorithm or against the state of the art.

We tested the proposed optimizations of Sections 4.2.1 and 4.2.2. For each combination of optimizations,

Table 3: Impact of the states reduction optimizations and the partial reachability graph optimization (PRG) on the performance of the REACH algorithm for the complete dataset. Performance has been evaluated in terms of the average execution time in milliseconds (time) and the average number of states discovered (states). To better display the performance increase, both measurements are divided into simple and complex log-model pairs as indicated by the subscript. Simple log-model pairs are those that take less than 10 seconds to solve without optimizations. In addition, both metrics are taken as absolute values (a, b) and percentages of improvement with respect to the execution with no optimizations applied (c).

PRG			✓	
SRModel		✓		✓
SRLog			✓	✓
$states_{simple}$	$1.73 \cdot 10^6$	$1.58 \cdot 10^6$	$1.47 \cdot 10^6$	$1.31 \cdot 10^6$
$states_{complex}$	$2.96 \cdot 10^7$	$2.47 \cdot 10^7$	$2.64 \cdot 10^7$	$2.00 \cdot 10^7$
$time_{simple}$	$3.02 \cdot 10^3$	$2.96 \cdot 10^3$	$2.65 \cdot 10^3$	$2.66 \cdot 10^3$
$time_{complex}$	$2.90 \cdot 10^4$	$2.51 \cdot 10^4$	$1.97 \cdot 10^4$	$1.73 \cdot 10^4$

a Absolute metrics with different optimizations enabled.

PRG	✓	✓	✓	✓
SRModel		✓		✓
SRLog			✓	✓
$states_{simple}$	$1.73 \cdot 10^6$	$1.58 \cdot 10^6$	$1.47 \cdot 10^6$	$1.31 \cdot 10^6$
$states_{complex}$	$2.96 \cdot 10^7$	$2.47 \cdot 10^7$	$2.64 \cdot 10^7$	$2.00 \cdot 10^7$
$time_{simple}$	$3.02 \cdot 10^3$	$2.97 \cdot 10^3$	$2.66 \cdot 10^3$	$2.66 \cdot 10^3$
$time_{complex}$	$2.78 \cdot 10^4$	$2.46 \cdot 10^4$	$2.00 \cdot 10^4$	$1.74 \cdot 10^4$

b Absolute metrics with different optimizations enabled.

PRG	✓	✓	✓
SRModel	✓		✓
SRLog		✓	✓
$states_{simple}$	8.6	15.1	24.0
$states_{complex}$	16.4	10.5	32.4
$time_{simple}$	1.7	12.2	11.9
$time_{complex}$	15.3	31.2	40.1

c Relative improvement (%).

we verified that the algorithm produced alignments of consistent cost and evaluated the enhancements in terms of: *(i)* the number of states generated before reaching an optimal solution; and *(ii)* the processing time. Table 3 shows some statistics for the complete dataset. If the execution for a single log-model pair takes longer than 5 minutes for any combination of optimizations, we do not consider it to keep the test times reasonable and the statistics fair.

The partial reachability graph (PRG) optimization is aimed at reducing the computation time required for the workflow net execution. Hence, it does not change the total number of states that need to be processed when enabled. It improves execution times by reducing the number of operations when interacting with the process model. The PRG optimization demonstrates its maximum effectiveness when no other optimizations are enabled, particularly when aligning complex log-model pairs, resulting in an average execution time reduction from 29.0 to 27.8 seconds. This improvement is reduced when other optimizations are enabled but, even then, the PRG optimization does not have a negative effect on the average execution time. It is affected by other optimizations as they lower the number of model operations by reducing the number of discovered states.

The States Reduction forcing asynchronous Model moves (SRModel) optimization achieves a reduction in the number of states discovered at the cost of increasing the processing time per state. For simple log-model pairs, which are those solved in under 10 seconds without any optimizations, SRModel reduces the average number of states by 8.6%. For complex problems, the reduction is even more substantial, amounting to 16.4% fewer states. For simple log-model pairs, the reduction of states is on par with the increased time per state,

Table 4: Slowest initialization times in milliseconds for the SRLog optimization, out of all 163 log-model pairs from (Reißner et al., 2020b). The model column indicates what percentage of behavior the model supports.

Log	Model	Time (ms)
BPIC19_2	100%	89,502
BPIC15F5	60%	18,951
BPIC15F5	70%	18,746
BPIC15F5	50%	18,334
BPIC15F5	100%	13,441
BPIC15F5	80%	12,675
BPIC15F5	90%	12,394
BPIC15F5	40%	11,389
BPIC15F5	30%	10,845
BPIC15F5	20%	8,641
SEPSIS	90%	166

leading to approximately the same mean total time (1.7% decrease in execution time). Nevertheless, it should be noted that this states reduction has a much greater effect on the complex log-model pairs, resulting in a reduction of the average time of 15.3%. Note that these percentages are measured with the PRG optimization enabled, as REACH will also enable all optimizations. The improvements in states discovered and execution time remain consistent with the PRG optimization disabled. This means that the SRModel optimization helps REACH to solve harder problems, without losing average performance on simple models. The SRModel optimization has a greater impact for bigger models, especially when the alignment between the model and the trace is very poor, which happens when the fitness is lower, i.e., for models that do not support much behavior of the log.

The States Reduction forcing asynchronous Log moves (SRLog) optimization reduces the number of discovered states by 15.1% for simple models and 10.5% for complex models. It outperforms SRModel in state reduction for simple models, while SRModel achieves better state reduction for complex models. Nevertheless, SRLog stands out for its performance improvements for more complex models, reaching an average reduction of 31.2% of the execution time. SRLog is more effective in models with many branches since forcing an asynchronous move in the log will remove all the states generated by each enabled activity on the model. As SRModel, this optimization has a great impact on models with low fitness since the probability that the activities of the remaining trace do not appear in the model is much higher. Its performance improvement is also evident in simple models, with an average execution time reduction of 12.2%. This is due to the fact that a significant portion of the computational cost of SRLog can be executed during an initialization phase, thereby reducing the time spent on each state.

The SRLog optimization requires an initialization phase whose execution time is generally very low for most of the log-model pairs. This initialization has been taken into account in the times reported on Table 3. To further investigate this initialization time, Table 4 shows in descending order the 11 highest execution times for the initialization of the SRLog optimization for the models from (Reißner et al., 2020b). For all

other log-model pairs, the execution times of this initialization are less than 100 milliseconds. Therefore, the SRLog optimization incurs negligible penalties in terms of execution time. Moreover, it is worth mentioning that in almost all cases in which the initialization takes more than 200 milliseconds —8 out of 10—, the algorithm would still take more than 5 minutes to finish if the optimization was disabled, so SRLog is improving the algorithm's performance. In the other 2 log-model pairs —BPIC15F5 with models with 100% and 90% behavior supported—, the initialization takes less than 15 seconds while the algorithm takes around 28 seconds.

SRModel and SRLog can be combined to complement each other and avoid redundant tasks. Enabling both optimizations results in an even better reduction in the number of discovered states —24.0% for simple models and 32.4% for complex models— and total execution time —11.9% for simple log-model pairs and 40.1% for complex ones.

In conclusion, the proposed optimizations for reducing the number of discovered states improve very significantly the efficiency of REACH over the test dataset, being the cause of its high performance when compared to the state-of-the-art proposals.

5.4. State of the art comparison

We compare REACH with several publicly available algorithms of the state of the art. We have included techniques that obtain the optimal alignments —ProMNoILP (Adriansyah, 2014) (ProM, PNetReplayer v6.11.191), ProMILP (de Leoni and van der Aalst, 2013) (ProM, PNetReplayer v6.11.191), ProMLP (Carmona et al., 2018) (ProM, PNetReplayer v6.11.191), eMEQ (van Dongen, 2018) (incremental version, ProM, Alignment v6.10.122), RecomposingReplay (Lee et al., 2018) (ProM, DecomposedReplayer v6.9.97), PartialReplayer (Lu et al., 2015) (ProM, PartialOrderReplayer v6.9.177), AutoConf (Reißner et al., 2017) (AutomataConformance v1.2 (Reißner et al., 2020b))—, and techniques that do not necessarily return the optimal alignments —AutoSComp (Reißner et al., 2020a) (AutomataConformance v1.2), AutoTRSComp (Reißner et al., 2020b) (AutomataConformance v1.2) and AutoHybrid (Reißner et al., 2020b) (AutomataConformance v1.2).

In the experiments, each algorithm is provided with a log in XES format, and one of its discovered models, in PNML format, and it returns one alignment per trace. All algorithms are configured in multi-thread mode to get the speedup of parallel processing. Furthermore, for each log-model pair, we measure the execution time from the moment the algorithm receives the inputs until it obtains the alignment for each trace of the log. Hence, we also account for the time needed for parsing the inputs (log and model) and writing the alignments. This is because some algorithms may involve pre/post-processing steps that should not be excluded from the total execution time as it would be unfair to other algorithms. Finally, once the alignments are obtained, for each trace we check whether its alignment is valid and, if applicable, whether its cost is the same as the alignments for that trace returned by the other optimal algorithms. Note that the alignments

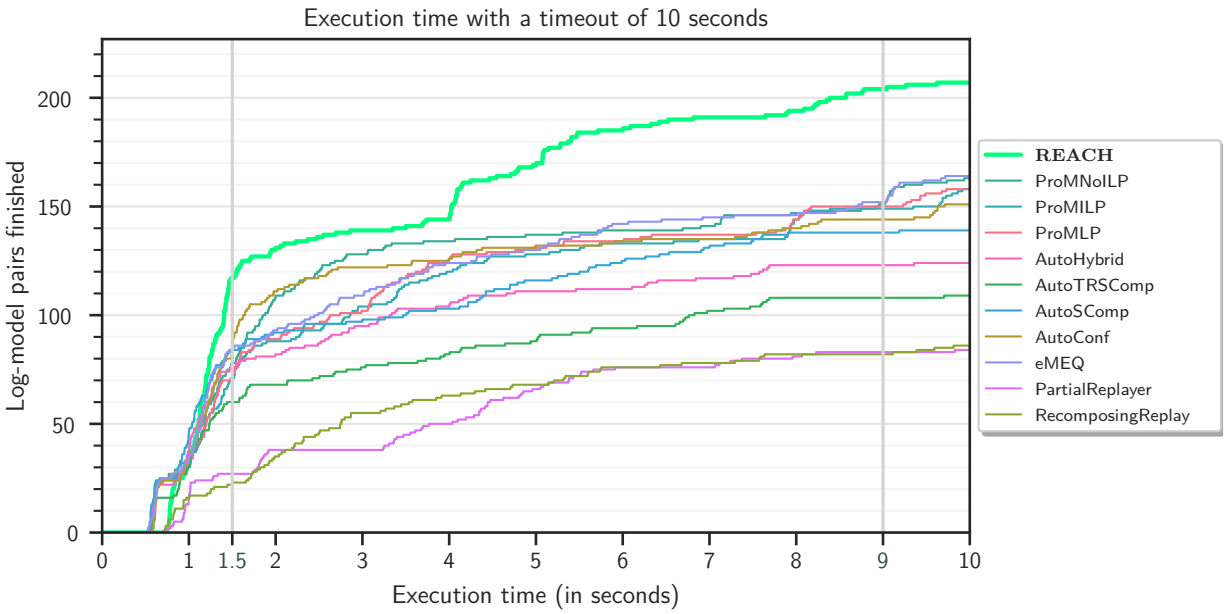


Figure 11: Solved problems of each algorithm with a time limit of 10 seconds.

returned by non-optimal algorithms are considered a solution, regardless of whether their cost for each trace is optimal or not. Although performing the comparison in this way is unfair for optimal algorithms like REACH, the aim is to show whether they obtain better results even with this disadvantage.

For practical reasons, we set a time limit for the computation times of the log-model pairs. We have considered two time limits —10 and 300 seconds—, and for each of them we analyze the performance using several visualizations. The first one is a graph that shows the number of log-model pairs that have been successfully computed —the algorithm returns the best alignment it has found— over time. Note that if an algorithm is not able to compute a log-model pair, e.g., it gets stuck processing a trace or cannot align some structures and throws an error, it is considered as if the time limit was reached. The second visualization is a ranking that compares the time taken by each of the algorithms to align each log-model pair. Furthermore, we also provide tables that show how the fitness and precision of input log-model pairs can affect the number of problems solved by each algorithm, highlighting the algorithm that solves the highest number of pairs within the given fitness or precision range.

5.4.1. Results with a time limit of 10 seconds

Figure 11 shows the results of the algorithms for a time limit of 10 seconds. It can be seen that all the algorithms compute the simplest alignments in less than 1 second. However, after that point, the curve representing REACH sharply rises above those of the other algorithms. This indicates that REACH is able to process a significantly larger number of log-model pairs in less time. In just 1.5 seconds, REACH successfully processes 117 pairs. Meanwhile, the second-best algorithm —eMEQ—, which is also optimal, computes

Table 5: Number of solved problems of the tested algorithms with a time limit of 10 seconds, splitting log-model pairs by fitness and precision.

	Fitness				Precision		
Algorithm	0.0-0.25	0.25-0.5	0.5-0.75	0.75-1.0	0.0-0.5	0.5-0.75	0.75-1.0
REACH	**0**	**22**	**64**	**121**	**1**	**32**	**62**
ProMNoILP	**0**	16	45	102	**1**	21	55
ProMILP	**0**	15	44	99	**1**	26	52
ProMLP	**0**	16	42	100	**1**	24	53
AutoHybrid	**0**	4	35	85	**1**	16	40
AutoTRSComp	**0**	2	31	76	**1**	16	31
AutoSComp	**0**	15	42	82	**1**	19	49
AutoConf	**0**	9	42	100	**1**	28	53
eMEQ	**0**	16	43	105	**1**	27	60
PartialReplayer	**0**	14	19	51	**1**	10	36
RecomposingReplay	**0**	15	9	62	**1**	16	35
Number of problems	3	28	66	130	1	33	62

85 pairs, closely followed by the optimal algorithm AutoConf and the non-optimal algorithm AutoSComp, both of which compute 84 pairs in that time. This faster behavior of REACH is even greater for log-model pairs that require more computation time, showing the scalability of our approach. Thus, in 9 seconds REACH is able to successfully complete 204 pairs, while the second-best algorithm —eMEQ— computes 152. Upon reaching the 10-second time limit, REACH has processed 207 out of 227 log-model pairs, whereas the second-best algorithm, eMEQ, is only able to complete 164 pairs. Note that the next best optimal algorithm —ProMNoILP— achieves a solution for 163 log-model pairs. Therefore, with a limit of 10 seconds, REACH has a performance that is 26% better than the best state-of-the-art algorithm.

Table 5 displays the solved log-model pairs per algorithm in a timeout of 10 seconds, categorizing problems by fitness and precision levels. It should be noted that the algorithm used to compute precision is Alignment Based Precision Checking with one optimal alignment (Adriansyah et al., 2013), and that this algorithm is not able to compute some log-model pairs due to the high execution time and memory requirements —this is totally independent of the alignment algorithm. This analysis allows us to examine how fitness and precision can affect the performance of each alignment algorithm. Regarding fitness, REACH consistently outperforms the state of the art. There is no clear effect on the number of solved problems depending on the fitness value. However, no algorithm can solve log-model pairs fitnesses lower than 0.25 in 10 seconds or less. This outcome is expected because these represent the most challenging problems: a lower fitness means a higher cost of the optimal alignment, and alignments of higher cost —with more misalignments to repair— are much more difficult to compute. In terms of precision, REACH also solves more problems than the state of the art, with the exception of precision values lower than 0.5, where there is only one example solved by all algorithms.

Table 6: Performance ranking of the tested algorithms.

a With a time limit of 10 seconds.

	rank
REACH	3.405
ProMNoILP	5.084
AutoSComp	5.132
AutoConf	5.460
eMEQ	5.518
AutoHybrid	5.597
ProMLP	5.888
ProMILP	6.081
AutoTRSComp	6.319
RecomposingReplay	8.566
PartialReplayer	8.949

b With a time limit of 300 seconds.

	rank
REACH	3.242
ProMNoILP	4.641
eMEQ	4.998
ProMLP	5.390
AutoSComp	5.399
AutoConf	5.678
ProMILP	5.797
AutoHybrid	5.945
AutoTRSComp	6.888
RecomposingReplay	8.980
PartialReplayer	9.042

Similarly to fitness, the precision level of input log-model pairs does not affect on the number of problems solved by REACH.

We have also compared the execution times that each algorithm needs to solve each of the 227 log-model pairs. To establish whether there are statistically significant differences between REACH and the other tested algorithms, we performed a non-parametric test using the execution times. First, a Friedman's Aligned Ranks test with a significance level of 0.05 has been applied. The ranking is calculated by ordering the execution times of all the algorithms for each log-model pair —the best algorithm obtains a rank of 1—, and then averaging the ranking of each algorithm in all the log-model pairs. The results of this test are summarized in Table 6a. The table clearly shows that REACH achieves the top ranking. In the second position, we find a non-optimal algorithm, AutoSComp, followed by the second-best optimal algorithm, PromNoILP.

As the *p-value* of the Friedman test is lower than 10^{-5}, there are significant differences in performances among the algorithms. Thus, we applied Holm's post hoc test to perform a pairwise comparison between REACH and the tested algorithms. The results of this test confirm that there are statistically significant differences between REACH and the other algorithms, with *p-values* consistently lower than 10^{-5}. Therefore, we can conclude that REACH is, on average, the fastest algorithm in our experiments over 227 diverse log-model pairs.

5.4.2. Results with a time limit of 300 seconds

Figure 12 depicts the results of the algorithms for a time limit of 300 seconds. REACH computes 216 log-model pairs in less than 45 seconds, reaching the time limit in only 11 of them. Conversely, other fast algorithms require more time to compute alignments, delivering results progressively and nearing the 5-minute time limit —and cannot return optimal alignments for some log-model pairs for which our approach is successful. Concretely, after 45 seconds of execution, the second-best optimal algorithms —eMEQ and

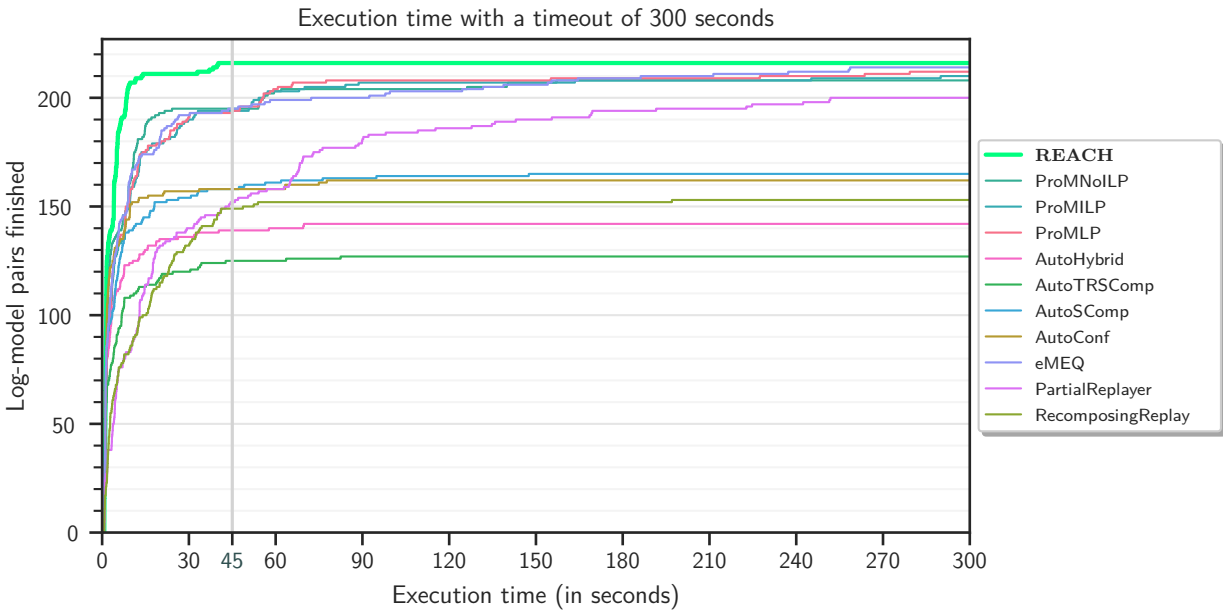


Figure 12: Solved problems of each algorithm with a time limit of 300 seconds.

ProMNoILP— align 195 log-model pairs, and the next best optimal algorithm —ProMILP— solves 194 pairs, while REACH aligns 9% more log-model pairs. When the time limit of 5 minutes is reached, the fastest algorithms after REACH are eMEQ, ProMLP and ProMNoILP, with 214, 212, and 210 solved log-model pairs respectively —in the best case, they solve 2 log-model pairs less than REACH solves in less than 45 seconds.

For the 227 tested log-model pairs, REACH is the fastest optimal algorithm in 125 pairs. In the remaining pairs, the computation time of REACH is, at most, 3 seconds slower than the fastest algorithm for each pair, except for 3 log-model pairs from Noisy logs (Munoz-Gama, 2013), for which only eMEQ is capable of finishing within the given timeout. These models are synthetic with extensive parallelism, leading to an excessive number of states that most algorithms cannot efficiently explore. As stated in (van Dongen, 2018), the logs for those models were built with vast amounts of swapped activities towards the end of the traces, which is a known weakness of A*-based methods. The ILP-based heuristic proposed in (van Dongen, 2018) is very effective in detecting swapped activities. However, this method spends more time computing the heuristic on each state, which makes it slower on average in the complete test dataset.

Our algorithm is the only one capable of computing alignments for the most complex models of the BPIC15 log. The primary contributor to the success of these log-model pairs is the SRLog optimization, which can prevent timeouts on its own. These examples have a large number of transitions (around 130), of which a considerable amount are silent transitions (around 60), leading to the generation of a large number of model moves needed from each state explored by the A* algorithm. The proposed optimization benefits from this situation, as it reduces the number of states when possible by forcing an asynchronous move in the log and avoiding all those unnecessary model moves.

Table 7: Number of solved problems of the tested algorithms with a time limit of 300 seconds, splitting log-model pairs by fitness and precision.

	Fitness				Precision		
Algorithm	0.0-0.25	0.25-0.5	0.5-0.75	0.75-1.0	0.0-0.5	0.5-0.75	0.75-1.0
REACH	**3**	22	**65**	126	**1**	**33**	**62**
ProMNoILP	**3**	22	64	119	**1**	30	60
ProMILP	2	22	64	122	**1**	**33**	61
ProMLP	**3**	**23**	63	123	**1**	32	61
AutoHybrid	0	4	37	101	**1**	25	43
AutoTRSComp	0	2	31	94	**1**	22	38
AutoSComp	0	19	45	101	**1**	27	55
AutoConf	**3**	9	45	105	**1**	29	55
eMEQ	**3**	22	62	**127**	**1**	**33**	**62**
PartialReplayer	0	21	60	119	**1**	32	61
RecomposingReplay	0	17	39	97	**1**	29	50
Number of problems	3	28	66	130	1	33	62

Table 7 displays the solved log-model pairs per algorithm within a timeout of 300 seconds, categorizing problems by fitness and precision levels. Focusing on fitness, eMEQ and ProMLP solve one more problem than REACHin ranges 0.75-1.00 and 0.25-0.50, respectively, but overall, REACH consistently matches the performance of the best state-of-the-art algorithms. We have observed no correlation between the fitness range of the input problem and the number of problems solved by REACH. In terms of precision, REACH is matched by eMEQ, while algorithms like ProMILP, PromLP or PartialReplayer exhibit very similar performance solving one or two less log-model pairs than REACH. Again, we have observed no correlation between the precision range of the input problem and the number of solved problems of REACH. As shown in Figure 12 even though REACH solves almost the same number of problems as some algorithms, it does so much faster than the state of the art.

In the rankings (Table 6b), REACH holds a position with a score of 3.242, surpassing the next best algorithm from the state of the art (ProMNoILP) with a ranking of 4.641. This reinforces the confidence that REACH is much faster, as this rank compares the time to compute alignments for each log-model pair. To confirm again that there are statistically significant differences between REACH and the other algorithms for a time limit of 300 seconds, we have repeated the Friedman's and Holm's tests —based on the ranking from Table 6b. Regarding Friedman's test, REACH has the best ranking with a *p-value* lower than 10^{-5}, indicating again that there are statistically significant differences in the performances of the algorithms. Furthermore, Holm's test allows the rejection of the null hypothesis in all the pairwise comparisons between REACH and the other algorithms, since the *p-values* are lower than 10^{-5} in all cases.

5.5. Limitations

Although REACH achieves superior performance compared to state-of-the-art algorithms, it still presents certain limitations. REACH does not succeed in solving 11 of the 227 tested log-model pairs due to the nature of the A* search: the number of states to explore explodes based on the complexity of the models and logs, as well as the cost of the optimal alignment. There are several factors that affect the performance of the algorithm in those cases, among which we highlight:

- The parallelism allowed by the process model, i.e., the average number of enabled activities for each reachable marking. Each of those activities must generate a new neighboring state by performing a model move when that marking is reached. Each state added to the search space when exploring the neighbors of a state exponentially increases the time complexity, as each generated state is recursively explored if the search algorithm requests it. This is aggravated by the presence of silent transitions, which allow reaching different markings of the process model without increasing the cost, effectively raising the number of neighboring states. Loops also contribute to this by removing the length limit of paths through the model.
- The length of each trace directly affects the number of moves required in the optimal alignment. As alignments are constructed from start to end, adding a move at each state transition, the depth of the solution in the search space increases with the addition of an event to the trace. Increasing the depth of the explored search space exponentially raises the time complexity of the algorithm.
- The cost of the optimal alignment, i.e., the minimum number of errors that must be repaired for the trace to follow a valid path through the model, forces the A* search to explore more states. This is because the solution will include more asynchronous movements of cost 1, compelling the search algorithm to ensure that there are no complete alignments of lower cost than the optimal one.

Among the 227 tested log-model pairs, REACH was not the fastest optimal algorithm in 100 instances. Notably, these cases primarily correspond to the simplest problems in the experiment. This observation is visually supported by Figure 11, where it can be seen that REACH takes slightly longer to solve most of the problems that take less than one second. The median delay of REACH with respect to the fastest optimal algorithm for each of these pairs is 204ms, and the fastest algorithm is not always the same, varying among ProMNoILP, ProMILP, ProMLP, AutoConf, eMEQ and RecomposingReplay. The delay observed in simple log-model pairs for REACH is primarily attributed to the extended initialization time required by the proposed optimizations. These optimizations have been designed with the aim of enhancing performance in the context of complex log-model pairs.

Even with all optimizations applied, REACH was unable to complete the alignments computation for 11 of the tested log-model pairs within less than five minutes. The only optimal algorithm capable of solving three of these pairs is eMEQ (van Dongen, 2018). This algorithm uses a complex heuristic based on Integer

Linear Programming (ILP), which proves more effective than REACH at detecting misalignments toward the end of the trace. For each state, the ILP solver can estimate the minimum cost of a solution without overestimating it, thus suiting A* heuristics. Although this estimation is computationally expensive, it is more accurate than REACH , making eMEQ capable of solving three log-model pairs that REACH cannot finish within the given time limit.

6. Conclusions and future work

We have presented REACH, an A*-based algorithm that computes in a very efficient way optimal alignments. The main contributions of our proposal include techniques designed to minimize the number of states explored by the A* algorithm and the utilization of a partial reachability graph for faster execution of process models during alignment computation. We have tested our proposal with 227 log-model pairs from different domains and discovered using different algorithms. We verified the performance of each contribution of our algorithm by partially enabling the proposed optimizations, and we have compared the performance of our proposal with 10 state-of-the-art conformance-checking algorithms. Results show that REACH aligns 95% of the tested log-model pairs in less than 45 seconds. Remarkably, our proposed optimizations empower REACH to complete alignments in just 45 seconds for two log-model pairs that no other algorithm can solve within a 5-minute time frame. Moreover, for a 10-second time limit, it also aligns 26% more pairs than all the other optimal state-of-the-art algorithms. REACH exhibits exceptional speed, outperforming all state-of-the-art approaches by aligning 55% of the log-model pairs more rapidly than any other algorithm. Our performance improvements enable the efficient computation of optimal alignments, allowing users to perform more precise conformance checking. By eliminating the need to rely on fast but less accurate methods, our approach opens up new possibilities for the application of conformance checking in real-world scenarios.

However, REACH still presents some limitations that should be addressed in future work. It currently proposes a balanced heuristic between accuracy and computing time, but this may not return the fastest solution for some log-model pairs: (i) for some high-complexity pairs, Integer Linear Programming (ILP) can be used to define more accurate heuristics (although slower) than our heuristic, or (ii) very easy to compute optimal alignments, that could be solved faster by algorithms that compute simple heuristics very quickly, albeit inaccurately. The SRLog optimization also slightly increases the computation time for the simplest problems due to the relatively slow initialization phase. Therefore, as future work, we plan to develop a more advanced heuristic based on ILP, focusing on reducing the time spent on each state. Nevertheless, simpler problems are solved faster when using simpler heuristics and disabling the SRLog optimization. Hence, we will propose a new classification technique that, based on the characteristics of the input log and model, selects the heuristic and optimizations that best tackle the given problem.

CRediT authorship contribution statement

Jacobo Casas-Ramos: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - Original Draft, Visualization. **Manuel Mucientes**: Conceptualization, Methodology, Resources, Writing - Review & Editing, Supervision, Funding acquisition. **Manuel Lama**: Conceptualization, Methodology, Resources, Writing - Review & Editing, Supervision, Funding acquisition.

Acknowledgement

This research was partially funded by the Spanish Ministerio de Ciencia e Innovación (grant number PID2020-112623GB-I00), and the Galician Consellería de Cultura, Educación e Universidade (grant numbers ED431C 2018/29 and ED431G2019/04). These grants are co-funded by the European Regional Development Fund (ERDF). Jacobo Casas-Ramos is supported by the Spanish Ministerio de Universidades under the FPU national plan (grant number FPU19/06668).

References

Adriansyah, A. (2014). *Aligning observed and modeled behavior*. PhD thesis, Technische Universiteit Eindhoven.

Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B. F., and van der Aalst, W. M. P. (2013). Alignment based precision checking. In *Business Process Management 2012 International Workshops*, pages 137–149. Springer.

Berti, A. and van der Aalst, W. M. P. (2021). A novel token-based replay technique to speed up conformance checking and process enhancement. *Transactions on Petri Nets and Other Models of Concurrency*, 15:1–26.

Carmona, J., van Dongen, B., Solti, A., and Weidlich, M. (2018). *Conformance Checking*. Springer International Publishing.

Davidrajuh, R. (2013). Extended reachability graph of Petri net for cost estimation. In *2013 8th EUROSIM Congress on Modelling and Simulation*, pages 378–383.

de Leoni, M., Lanciano, G., and Marrella, A. (2018). Aligning partially-ordered process-execution traces and models using automated planning. In *28th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 321–329.

de Leoni, M. and Marrella, A. (2017). Aligning real process executions and prescriptive process models through automated planning. *Expert Systems with Applications*, 82:162–183.

de Leoni, M. and van der Aalst, W. M. P. (2013). Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In *11th International Conference on Business Process Management (BPM)*, pages 113–129. Springer.

de Leoni, M. M. and Mannhardt, F. (2015). Road Traffic Fine Management Process.

Garcia-Banuelos, L., van Beest, N. R. T. P., Dumas, M., Rosa, M. L., and Mertens, W. (2018). Complete and interpretable conformance checking of business processes. *IEEE Transactions on Software Engineering*, 44(3):262–290.

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

Lee, W. L. J., Verbeek, H., Munoz-Gama, J., van der Aalst, W. M., and Sepúlveda, M. (2018). Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining. *Information Sciences*, 466:55–91.

Leemans, S. J., van der Aalst, W. M. P., Brockhoff, T., and Polyvyanyy, A. (2021). Stochastic process mining: Earth movers' stochastic conformance. *Information Systems*, 102:101724.

Leemans, S. J. J., Fahland, D., and van der Aalst, W. M. P. (2014). Discovering block-structured process models from event logs containing infrequent behaviour. In *Business Process Management 2013 International Workshops*, pages 66–78. Springer.

Leemans, S. J. J., Fahland, D., and van der Aalst, W. M. P. (2018). Scalable process discovery and conformance checking. *Software & Systems Modeling*, 17(2):599–631.

Lu, X., Fahland, D., and van der Aalst, W. M. P. (2015). Conformance checking based on partially ordered event data. In *Business Process Management 2014 Workshops*, pages 75–88. Lecture Notes in Business Information Processing.

Mannhardt, F. (2016). Sepsis Cases - Event Log.

Maruster, L., Weijters, A., van der Aalst, W., and van den Bosch, A. (2006). A rule-based approach for process discovery: Dealing with noise and imbalance in process logs. *Data Mining and Knowledge Discovery*, 13(1):67–87. Pagination: 21.

Munoz-Gama, J. (2013). 'Conformance Checking in the Large' (BPM 2013).

Munoz-Gama, J. (2014). Single-Entry Single-Exit Decomposed Conformance Checking (IS 2014).

Munoz-Gama, J., Carmona, J., and van der Aalst, W. M. P. (2013). Hierarchical conformance checking of process models based on event logs. In *34th International Conference Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, pages 291–310. Springer.

Munoz-Gama, J., Carmona, J., and van der Aalst, W. M. P. (2014). Single-entry single-exit decomposed conformance checking. *Information Systems*, 46:102–122.

Reißner, D., Armas-Cervantes, A., Conforti, R., Dumas, M., Fahland, D., and La Rosa, M. (2020a). Scalable alignment of process models and event logs: An approach based on automata and S-components. *Information Systems*, page 101561.

Reißner, D., Armas-Cervantes, A., and La Rosa, M. (2020b). Efficient conformance checking using alignment computation with tandem repeats. *arXiv preprint arXiv:2004.01781*.

Reißner, D., Conforti, R., Dumas, M., La Rosa, M., and Armas-Cervantes, A. (2017). Scalable conformance checking of business processes. In *On the Move to Meaningful Internet Systems: OTM 2017 Conferences*, pages 607–627.

Rozinat, A. and Van der Aalst, W. M. (2008). Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95.

Sani, M. F., van Zelst, S. J., and van der Aalst, W. M. P. (2020). Conformance checking approximation using subset selection and edit distance. In *32nd International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 234–251. Springer.

Taymouri, F. and Carmona, J. (2016). A recursive paradigm for aligning observed behavior of large structured process models. In *14th International Conference Business Process Management (BPM)*, pages 197–214. Springer.

Taymouri, F. and Carmona, J. (2018). An evolutionary technique to approximate multiple optimal alignments. In *16th International Conference Business Process Management (BPM)*, pages 215–232. Springer.

Taymouri, F. and Carmona, J. (2020). Computing alignments of well-formed process models using local search. *ACM Transactions on Software Engineering and Methodology*, 29(3):1–41.

van den Broucke, S. K., Munoz-Gama, J., Carmona, J., Baesens, B., and Vanthienen, J. (2014). Event-based real-time decomposed conformance analysis. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, pages 345–363.

van der Aalst, W. M. P. (1996). Structural characterizations of sound workflow nets. *Computing science reports*, 96(23):18–22.

van der Aalst, W. M. P. (2013). Decomposing Petri nets for process mining: A generic approach. *Distributed and Parallel Databases*, 31(4):471–507.

van der Aalst, W. M. P., Adriansyah, A., de Medeiros, A. K. A., Arcieri, F., Baier, T., Blickle, T., and Bose, J.C. et al (2012). Process Mining Manifesto. In *Business Process Management 2011 International Workshops*, pages 169–194.

van Dongen, B. (2012). BPI Challenge 2012.

van Dongen, B., Carmona, J., Chatain, T., and Taymouri, F. (2017). Aligning modeled and observed behavior: A compromise between computation complexity and quality. In *29th International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 94–109. Springer.

van Dongen, B. F. (2018). Efficiently computing alignments. In *Business Process Management*, pages 197–214, Cham. Springer International Publishing.