Computing alignments with maximum synchronous moves via replay in coordinate planes

Hui Yan, Uzay Kaymak, Pieter Van Gorp, Xudong Lu, Shan Nan, Huilong Duan

Key Laboratory of Biomedical Engineering of Hainan Province, School of Biomedical Engineering, Hainan University, Haikou, China
Eindhoven University of Technology, Eindhoven, The Netherlands
School of Biomedical Engineering and Instrumental Science, Zhejiang University, Hangzhou, China

Abstract
Optimal alignments are the basis of conformance checking. For long, researchers have been devoted to the efficiency issue of computing optimal alignments. This paper focuses on the optimality issue. Specifically, we aim to find alignments with maximum synchronous moves and minimum deviations. This paper introduces a coordinate-plane search space, which allows enumerating all the possible alignments. The alignments with maximum synchronous moves are translated into the lowest-cost paths, such that heuristic strategies (such as the Dijkstra algorithm) can be applied. Both theoretical proof and experimental results show that 100% optimality can be achieved.

1. Introduction

Business processes play an important role in improving the efficiency of individual organizations, because they provide an effective way to document, understand and further analyze processes, in practice. Information technology has accelerated the widespread use of information systems, which provides abundance of historical data on which events occurred during process executions. Conformance checking aims to compare the behavior of a business process model and the behavior observed in the historical data for detecting and pinpointing what deviated where. Such diagnosis suggests two analytic angles: (a) how to improve the model to reflect reality better? and, (b) how to improve control to enforce a better conformance? Undesirable deviations suggesting fraud or inefficiencies are examples of the second analytic angle.

The state-of-the-art techniques of conformance checking use alignment-based approaches. Alignment-based conformance checking aligns the events of a particular case with the “closest” matching path described by the process model. However, finding the closest matching path, i.e., computing optimal alignment(s), faces a twofold challenge: efficiency and optimality. It is known as a non-deterministic polynomial-time hardness (NP-hard) problem. Because a business process model could involve parallel, alternative, and iterative routings, the possible candidate executions are combinatorial of all the routings. Moreover, even compared to one candidate execution, the particular case could have missing, redundant, and dislocated behaviors. Thus, for a particular case, there are an exponential number of candidate executions.
Most studies [4,2,16] have focused on the efficiency aspect using heuristic strategies. Gaining efficiency has led to the loss of optimality. Song et al. in [21] have reported that their method was able to find the optimal alignment for most cases. It means that for some cases, the returned results are not optimal. It was also observed in [6] that though the method in [2] was able to return alignments with the lowest cost, they do not always have maximum synchronous moves. Up till the present moment, the work in [2] was still regarded as "guaranteed optimality" according to [26]. But, it was commented in [9] as "ad hoc implementations of the A* algorithm based on an ad hoc heuristics".

Instead of some cost-based definitions in [2,26], we explicitly define optimal alignments as the ones having the maximum number of synchronous moves among all the possible alignments. A synchronous move is a pair-wise match between an event in the case and an activity in the process model, while an asynchronous move is a deviation. Thus, our proposed definition, which addresses maximum synchronous moves, is consistent with the essence of alignment-based approaches, which addresses the closest matching. Besides, this quantity-based definition makes optimality accurate and measurable.

The key issue is how to ensure that all the possible alignments are checked and efficiency is still guaranteed. To tackle this issue, we propose an optimal alignments computation (OAC) approach in this paper. The core of OAC approach is coordinate-plane spaces (CPSs), which allow enumerating all the possible alignments on one hand, and support various heuristic algorithms to efficiently obtain optimal alignments on the other hand.

One CPS can be seen as a coordinate plane with the observed case being in the horizontal axis and the process model being in the vertical axis. Then, synchronous moves and asynchronous moves can be seen as edges in that CPS. Thus, CPSs enable enumerating all the possible alignments. To find the optimal ones, we propose translating the optimal alignments problem into a lowest-cost paths problem. The latter can be solved by several efficient path-searching algorithms. To make the lowest-cost paths our optimal alignments, the costs of edges in CPSs need to be properly designed such that the equivalence between the two can be achieved. Furthermore, this equivalence is guaranteed by a rigorous theoretical proof. The main contributions are as follows.

1. A quantity-based (maximum number of synchronous moves) definition of optimal alignments is proposed.
2. A new approach for computing optimal alignments based on CPSs, which allows enumerating all the possible alignments and supports advanced strategies, is proposed. As the core of this paper, it makes computing optimal alignments with both optimality and efficiency.
3. The conditions under which the optimal alignments are the lowest-cost paths in CPSs are developed with rigorous theoretical proof.
4. The Dijkstra algorithm is applied in computing optimal alignments for execution spaces without loops. A method gradually unfolding loops is proposed for execution spaces with loops.
5. The proposed OAC approach was evaluated using various experiments, which show OAC always achieves 100% optimality.

The rest of paper is organized as follows: related work is reported in Section 2. The preliminaries are presented in Section 3. CPSs are presented in Section 4. The design of the costs for edges is studied in Section 5. A proof that the design of costs works is provided in Section 6. Implementations of algorithms to compute optimal alignments are presented in Section 7. Evaluations with real-life datasets are described in Section 8. Section 9 concludes this paper.

2. Related work

Studies in computing optimal alignments can be divided into three categories: decomposition-based [23,15], planner-based [9], and heuristic replay [4,2].

2.1. Decomposition-based approach

Researchers [19,21] proposed decomposition strategy to tackle the computational complexity of computing optimal alignments. The basic idea is decomposing the process model into several independent sub-models. The sub-models could be single-entry single-exit regions [17] or without alternative or iterative routings [21]. Then existing alignment algorithms (i.e., [3,8]) or linear time algorithm [21] can be applied to find the optimal alignments for the sub-models. The first shortcoming of this approach is that the optimal alignments obtained are the combination of local optimal alignments, which is not guaranteed to be globally optimal [27]. The second shortcoming is that this approach only applies to process models that can be easily decomposed into sub-models without alternatives. As a counter example, when process models contain alternatives embedded in parallelism, it is difficult to partition. Though this approach leads to shorter computing time, it cannot always guarantee the optimality of the alignments found.

2.2. Planning problem-based approach

De Leoni et al. in [9] formalized optimal aligning computation into planning problems. Given a process model in Petri nets and a log trace, a synthesized plan is constructed such that solution steps correspond to alignment steps. The purpose is to
minimize the total cost of the alignment. The theoretical results show that solutions can always be found by planners in a finite amount of time. However, as observed in another study [27], this approach is only available for safe Petri nets; in other words, process models with loops cannot use this approach.

2.3. Replay-based approach

2.3.1. Token replay

Rozinat et al. [19] proposed a token-based replay approach, which takes each log trace in isolation and replays it against the process model (in Petri nets), namely, firing transitions sequentially according to the ordering of events in the trace. If there is a transition that should be fired but not enabled, missing tokens are added to enable the transition. This approach can identify two types of inconformity: (i) missing tokens: how many times a token needs to be added to a place in the Petri net in order to continue replay; and (ii) remaining tokens: how many tokens remain in the Petri net once a trace has been fully replayed. In case there is an event associated with more than one enabled transition, heuristics is used to find out which candidate is better. As the heuristic information used is ad hoc, it may lead to misleading results. Thus, its optimality is not guaranteed.

2.3.2. Tree-based heuristic replay

Studies in [3,16] constructed a search space graph (weighted directed graph) of replaying log trace separately on a Petri net model. The search graph consists of edges representing three types of moves: synchronous moves and asynchronous moves representing skipping and inserting activities. The edges representing asynchronous moves are assigned to a non-negative cost. Each alignment is represented by a path from a source node to a goal node, which represents an initial replay state and a final replay state. The A* algorithm was adopted to find the lowest-cost path from a source node to a target node in the search space. Notably, the formalized search space contains only splitting branches (no join branches). Thus, it is essentially a tree, which has more than one goal node. Each leaf node of the tree is one goal node indicating that the entire log trace has been aligned. Instead of exploring all the goal nodes, the search [3] terminates when the first goal node is encountered. The alignments found at the first goal node are regarded as optimal alignments. The main drawback of this approach is that the assumption that the first found goal node corresponds to the optimal alignments lacks theoretical derivation. For multiple goal nodes corresponding to different process traces, the first found goal node has the lowest cost among all the other goal nodes. However, it is not always true that the lowest-cost path has the fewest number of deviations. For example, a second found goal node could have the same cost with the first one. Yet, they might have the same number of deviations, while the second found path could have more synchronous moves than the first one.

2.3.3. Graph-based heuristic replay

To overcome multiple-goal nodes in the tree-based approach [3], Adriansyah et al. [2] proposed a graph-based search space: the transition system of the product of two Petri nets [30]. One Petri nets is a given process model, and the other net is a given log trace represented in Petri nets. The product net is a union of the two Petri nets with extra synchronous transitions that are constructed by pairing transitions in one net with transitions in the other net that refers to the same activity name [2].

The transition system (also called reachability graph) of the product net is generated such that all possible behavior allowed by the product net is encoded in the transition system. As the transition system has one initial marking and one final marking, the search space is genuinely graph-based instead of tree-based. Each edge in the transition system is assigned according to standard cost function: cost value being 0 for synchronous moves or being a non-negative value for asynchronous moves. Thus, all paths from the initial marking to the final marking in the transition system yield all possible alignments. The total cost of edges in one path can represent the cost of the alignment. The A* algorithm was used to compute the path with the lowest cost from the initial marking to the final marking on the transition system.

As for optimality, though this work provides guaranteed optimality according to [9], the so-called optimal alignments with lowest cost do not necessarily have maximum synchronous moves. The first reason is that the non-negative cost value for all the moves on model are equal. It causes misleading results, as observed in [6]. Suppose that a Petri net has two process traces: \( \sigma_{M_1} = \langle a, b, c \rangle \) and \( \sigma_{M_2} = \langle b, d \rangle \), and a given log trace is \( \sigma_l = \langle < b > \). Based on the standard cost function in [2], the cost for \( \sigma_{M_1} \) is 2 and the cost for \( \sigma_{M_2} \) is 1. The algorithm selects \( \sigma_{M_2} \) as the optimal alignment as it has a lower cost. However, \( \sigma_{M_1} \) and \( \sigma_{M_2} \) have the same number of synchronous moves. Thus, for process models with different-length traces, the optimality is not guaranteed in the sense of maximum number of synchronous moves. The second reason is that when generating the transition system of the product net with one net involving loops, there should be infinite final markings in theory. Since each final marking makes an individual goal node, in searching for lowest-cost paths, the global lowest-cost path can only be found by involving all the final markings. However, the work in [2] took one goal node.

As for efficiency, the work in [2] suffers from time complexity for two reasons. The A* algorithm requires an underestimating heuristic value (know as the cost function) to guide the search efficiently. The work in [2] adopted Integer Linear Programming to compute the estimation, which is exponential in the rank of the input matrix. This is the first reason for time complexity. The second reason is the search space, i.e., transition system of the product net, is the full execution of the input. The size of the search space is considerable when the input process model contains parallelisms. When the
worst-case scenario occurs where the given log trace deviates from the process model to a great extent, the algorithm needs to visit almost all the states to find the lowest-cost optimal alignments. Thus, the worst-case complexity is exponential with respect to the size of the state space.

In line with the method in [2], van Dongen et al. [28,27] strive to decrease the time complexity in two ways: improving the heuristic estimation and applying a decomposition approach to reduce the computing time. However, these efforts only improve the efficiency of the computing. The optimality level is still the same as the work in [2].

3. Preliminaries

This section introduces log traces, transition systems, execution spaces, alignments, and optimal alignments.

3.1. Log traces

**Definition 1 (Log trace).** Let $A_L$ be the set of all possible sequences consisting of elements of $A_L$. A log trace $\sigma_L \in A_L$ is a sequence of events $A_L, \sigma_L = (A_L, \succ_L)$ where:

- $A_L$ is a set of events, each of which has at least three elements: a unique case identifier, an activity name (label), and a timestamp.
- $\succ_L \subseteq A_L \times A_L$ impose a total ordering on the events in $A_L$. The order is typically based on timestamp.

Furthermore, events in log traces could include information on other process perspectives, like resources, costs, duration, etc.

3.2. Transition systems

A transition system, also called reachability graph [13] or marking graph [31] or statespace [25], is one of the most basic process modeling notations. The following definition is adapted from [24].

**Definition 2 (Transition system).** A transition system of a process model $PM$ is a tuple $TS = (S, \text{FO}_PM, T)$ where $\text{FO}_PM$ is a set of flow objects in $PM$, $S$ is the (possibly infinite) set of states, also called markings, $T \subseteq S \times \text{FO}_PM \times S$ is the set of transitions between states.

A transition system could work as a placeholder for more advanced modeling languages, e.g. Petri nets, UML, BPMN (Business Process Model and Notation) [12], EPCs, etc. [24]. Petri nets have their own mechanism for generating transition systems [31]. Take the BPMN model in Fig. 1 for example, its transition system is shown in Fig. 2a, which is generated using the BPMN execution rules specified in [28].

3.3. Execution spaces

As the essence of alignment-based conformance checking is to align events to tasks in process models, behaviors regarding routings of activities, e.g. gateways, can be hidden. Thus, the states and transitions in a TS that are not relating to tasks can be omitted but the connection between the predecessor and successor states are kept. A new representation only for the execution of tasks can be derived.

**Definition 3 (Execution spaces).** An execution space of a process model $PM$ is a directed graph $M = (A_M, R_M)$ where:

![Fig. 1. A BPMN process model.](image-url)
A $A_M = (ID_M, FO_M)$ is a set of executed tasks, each of which refers to an execution of a task $FO_{Task}$. Start Event or End Event. $FO_M$ is a union of tasks $FO_{Task} \subseteq FO_{PM}$, Start Event, and End Event of PM. $ID_M$ is the set of identities of $A_M$.

Let $A_M$ be the set of all possible sequences consisting of elements of $A_M$. $\sigma_M \subseteq A_M$ denote a full execution sequence in $M$.

Fig. 2b shows the execution space of the BPMN model in Fig. 1. This execution space is transformed from the transition system in Fig. 2a by hiding transitions relating to Gateways G1, G2, G3, and G4. Notably, the transitions drawn as edges in Fig. 2a are drawn as node in Fig. 2b for the purpose of explicit. In Fig. 2b, an executed task indicating entering a flow object is always followed by an executed task for leaving the same flow object. However, if performing a certain task takes some time, there could be other executed tasks between the Enter-Leave pairs. For illustrative purposes, we assume executions of tasks are instantaneous. Furthermore, the executed tasks indicating Leave can be hidden in the execution spaces to have a more concise view. In the following sections, an execution space only contains executed tasks of type Enter.

### 3.4. Alignments

**Definition 4 (Aligned activity).** Let $\sigma_L$ be a log trace over a set of events $A_L$, let $M = (A_M, R_M)$ be an execution space specified by a set of executed tasks $A_M$ and their relations $R_M$. Let $A_L = A_L \cup \{\bot\}, A_M = A_M \cup \{\bot\}$. An aligned activity $(x, y) \in A_L \times A_M$ is a pair-wise match between an event in $A_L \cup \{\bot\}$ and an executed task in $A_M \cup \{\bot\}$. There are four types of $(x, y)$:

- $(x, y)$ is a log move, if $x \in A_L$ and $y = \bot$,
- $(x, y)$ is a model move, if and $x = \bot$ and $y \in A_M$,
- $(x, y)$ is a synchronous move, if $x \in A_L$ and $y \in A_M$ under the condition that $x$ relates to $y$,
- $(x, y)$ is an illegal move otherwise.

Model moves and log moves are also called asynchronous moves. In conformance analysis, a *model move* is interpreted as *omission* because it represents a missing event according to the model; a *log move* is interpreted as *addition* because it represents an additional event compared to the model. **Fig. 2.** From transition system to execution space.
Definition 5 (Alignments). Let $\sigma_L$ be a log trace over $A_L$ and let $M = (A_M, R_M)$ be an execution space specified by a set of activities $A_M$ and their relations $R_M$. An alignment $\gamma$ between $\sigma_L$ and $M$ is a sequence of aligned activities $(x, y)$. $\Gamma_{\sigma_L, M}$ is the set of all alignments between a trace $\sigma_L$ and an execution space $M$.

3.5. Optimal alignments

We explicitly define optimal alignments as the ones with maximum synchronous moves and minimum asynchronous moves to reflect a maximum resemblance between a log trace and a process model.

Definition 6 ($\Omega$). $\Omega(\gamma)$ is the function calculating the number of synchronous moves in an alignment $\gamma$.

Definition 7 (Dev). $\text{Dev}(\gamma)$ is the function calculating the number of asynchronous moves in an alignment $\gamma$.

Definition 8 (Optimal alignments). Optimal alignments is a set of alignments, $\Gamma_{\text{opt}}$:

$$\Gamma_{\text{opt}} = \{ \gamma_{\text{opt}} \mid \forall \gamma \in \Gamma_{\sigma_L, M} \Omega(\gamma_{\text{opt}}) \geq \Omega(\gamma) \land \text{Dev}(\gamma_{\text{opt}}) \leq \text{Dev}(\gamma) \}.$$  

We say that $\gamma_{\text{opt}}$ is an optimal alignment between $\sigma_L$ and $M$ if and only if for all $\gamma \in \Gamma_{\sigma_L, M} : \Omega(\gamma) \leq \Omega(\gamma_{\text{opt}})$ and $\text{Dev}(\gamma) \geq \text{Dev}(\gamma_{\text{opt}})$.

4. Coordinate-plane spaces

This section introduces the concept of CPSs, which makes sure all the possible alignments are possible to be checked so that real optimal ones can be found.

Given a log trace and a process model, the replay [24] uses the log trace to drive the execution of the process model. In case that the event cannot match the task of the model, an artificial event desired by the model or an artificial task desired by the log trace is inserted so that the execution could continue. There are multiple places where artificial events or tasks can be inserted, which results in all kinds of alignments. All the possible alignments are able to be generated using the replay method.

Coordinate planes can be used as the space for replay. Given a log trace $\sigma_L = \langle a, c, b, d, e \rangle$ and an execution space $M$ shown in Fig. 4a. For illustrative purpose, $M$ is separated into two process traces, $\sigma_{M1} = \langle a, b, d, e \rangle$ and $\sigma_{M2} = \langle a, b, c, e \rangle$. As shown in Fig. 3, two coordinate planes are presented where their Y-axes are $\sigma_{M1}$ and $\sigma_{M2}$ respectively. The paths from the source point $n_{\text{source}}$ to the goal point $n_{\text{goal}}$ consist of edges, which is among three types: horizontal, vertical and diagonal. The horizontal edge represents a log move. The vertical edge represents a model move. The diagonal edge represents a synchronous move.

As the two Y-axes of Fig. 3a and b share some common parts Task "a", Task "b" and Task "e", the two coordinate planes can also be combined by merging the common parts. The combined coordinate system is shown in Fig. 4b. Herein, a definition of coordinate-plane space (CPS) for replaying a log trace in an execution space is introduced as follows.

Definition 9 (Coordinate-plane space). Given an execution space $M$ and a log trace $\sigma_L$, its coordinate-plane space $G_{M, \sigma_L} = (N, E, \Pi_{\sigma_L}, \Pi_M)$, where:

- $n \in N$ is a node representing one state during replaying $\sigma_L$ on $M$. $n_{\text{source}} \in N$ is the source node, which has no incoming edges, representing an initial state of the replay. $n_{\text{goal}} \in N$ is the goal node, which has no outgoing edges, representing the completion state of the replay.

![Fig. 3. Replaying a log trace in an execution space in coordinate planes.](image-url)
E denotes edges between nodes in \( N \), where \( E_B \) is the set of edges on the synchronous dimension in \( G_{M;\sigma} \), \( E_M \) is the set of edges on the model dimension in \( G_{M;\sigma} \), and \( E_L \) is the set of edges on the log dimension in \( G_{M;\sigma} \).

pr\(_L\) is a function projecting a node \( n \) to an element of the log trace \( \sigma_L \). It yields an event: \( pr_L(n) \in A_L \), which is at the state of completion of replay at node \( n \).

\( \pi_M \in \Pi_M \) is a function projecting a node \( n \) to an element of the execution space \( M \). It yields an executed task: \( \pi_M(n) \in A_M \), which is at the state of completion of replay at node \( n \).

e\(_{ij} \) denotes an edge between \( n_i \) and \( n_j \), and \( n_j \) is a successor of \( n_i \). \( e_{ij} \) represents an aligned activity, with one of the three types:
- Log move: \( (x, \perp) \) where \( x = \pi_{\sigma_L}(n_j) \), called \( e_{ij} \in E_L \) is an edge on the log dimension;
- Model move: \( (\perp, y) \) where \( y = \pi_M(n_j) \), called \( e_{ij} \in E_M \) is an edge on the model dimension;
- Synchronous move: \( (x, y) \) where \( x = \pi_{\sigma_L}(n_i) \) and \( y = \pi_M(n_i) \), called \( e_{ij} \in E_B \) is an edge on the synchronous model and log dimension.

Notably, this definition is suitable for execution spaces without loops. For execution spaces with loops, there are more than one \( n_{goal} \) that involves the termination state of the execution spaces. So, the CPSs are also multiple.

5. Design of costs of edges in CPSs

In the previous section, all the possible alignments are able to be enumerated using CPSs. To find the optimal ones without inefficient enumeration, we propose translating the optimal alignments into the lowest-cost paths in this section. Firstly, the formulation of the cost of a path in CPSs is presented. It will establish a link between the cost and the synchronous moves. Then, based on the formulation, we study the conditions under which the lowest-cost paths have the maximum number of synchronous moves. These conditions result in the design of the costs of edges, namely \( \delta(e) \) when \( e \) is in \( E_L, E_M \) and \( E_B \).

5.1. Formulation of the cost function

**Definition 10 (Cost of an alignment).** Cost of an alignment \( \gamma \) is \( \text{Cost}(\gamma) \):

\[
\text{Cost}(\gamma) = \sum_{i=1}^{\left|\gamma\right|} \delta(\gamma(i)),
\]

where \( \left|\gamma\right| \) is the length of the alignment \( \gamma \) and \( \delta(\gamma(i)) \) is the cost of the \( i \)-th element in the alignment.
where γ(i) is the i-th aligned activity in γ. δ is a cost function for each aligned activity. |γ| denotes the number of elements in γ.

γ(i) is one of the three types: a synchronous move, a model move, or a log move. We can put together each individual type, constituting a set \( E_B \subset E_a \), a set \( E_L \subset E_l \), and a set \( E_M \subset E_M \). Then, \( \text{Cost}(\gamma) \) equals to:

\[
\text{Cost}(\gamma) = \sum_{e_i \in E_B} \delta(e_1) + \sum_{e_2 \in E_L} \delta(e_2) + \sum_{e_3 \in E_M} \delta(e_3).
\]

(3) can be visualized in Fig. 5, where the individual set is moved together. Note that \( S_1 \) in Fig. 5 is the complete set of all the possible log moves for \( \sigma_1 \). \( S_2 \) in Fig. 5 is the complete set of all the possible model moves for \( \sigma_M \). \( S_1 \) and \( S_2 \) are fixed for any alignment between \( \sigma_1 \) and \( \sigma_M \). The varying parts are \( E_L \) and \( E_M \), but we can establish the relationship between them, the fixed part and \( E_B \). Like projections [5] from geometry, projecting \( e \in E_B \) on X-axis results in \( e \)'s horizontal component, denoted by \( \bar{e} \). As \( S_1 \) consists of \( E_L \) and the horizontal component of \( E_B \), it results in \( E_L = S_1 \setminus \{ \bar{e} \mid e \in E_B \} \). Similarly, \( E_M = S_2 \setminus \{ e \mid e \in E_B \} \).

It leads to (4):

\[
\text{Cost}(\gamma) = \sum_{e_i \in E_B} \delta(e_1) + \sum_{e_2 \in S_1 \setminus \{ e \mid \bar{e} = \bar{e}_1 \}} \delta(e_2) + \sum_{e_3 \in S_1 \setminus \{ e \mid \bar{e} = \bar{e}_2 \}} \delta(e_3).
\]

Further, we can isolate the varying part \( E_B \). Then, \( \text{Cost}(\gamma) \) equals to:

\[
\text{Cost}(\gamma) = \sum_{e_i \in E_B} \left( \delta(e_1) - \delta(e_1) \right) + \sum_{e_2 \in S_1} \delta(e_2) + \sum_{e_3 \in S_1} \delta(e_3).
\]

Thus, for any alignment between \( \sigma_1 \) and \( \sigma_M \), its cost equals to (5), where \( S_1 \) and \( S_2 \) are fixed and \( E_B \) varies. For \( \sum_{e_2 \in S_1} \delta(e_2) \), since both \( S_1 \) and \( \delta(e_2) \) are fixed, \( \sum_{e_2 \in S_1} \delta(e_2) \) is a constant for all the possible alignments. We suggest setting \( \delta(e) \) to one for \( e \in S_1 \), so \( \sum_{e_2 \in S_1} \delta(e_2) = C_1 \).

Similarly, \( \sum_{e_3 \in S_2} \delta(e_3) \) is fixed given one process trace \( \sigma_M \). However, there are more than one process traces in an execution space \( M \). To make \( \sum_{e_3 \in S_2} \delta(e_3) \) a constant \( C_M \) no matter to which process trace \( S_2 \) corresponds, a straightforward solution is assigning \( C_2 \) to the number of executed tasks of the longest process trace \( \sigma_{\text{max}} \) in \( M \). To decide \( \delta(e) \) for \( e \in S_2 \), we introduce a Floor function, which assigns an ordinal number, called Floor value, to each executed task of \( M \).

**Definition 11 (Floor function).** Given an executed task \( \pi_M(n) \) of an execution space \( M \), its Floor value \( \text{Floor}(\pi_M(n)) \) is the largest ordinal number starting from the Start Event in the multiple process traces to which it belongs.

In Fig. 6b, the Floor values of executed tasks of the execution space in Fig. 6a are labeled. For example, the executed task “Give medicine as ACS NSTEMI” belongs to multiple process traces. In the shortest process trace, its ordinal number is 3. In the longest process trace, its ordinal number is 11. So the Floor value of “Give medicine as ACS NSTEMI” is 11, according to **Definition 11**. Notably, the Floor value of “Perform Exercise ECG” being 7, 8, 9, or 10 does not change the design intention that \( \sum_{e_3 \in S_2} \delta(e_3) \) is constant.

Then, \( \delta(e_3)(e \in S_2) \) is the difference of Floor values of two neighboring executed tasks.
\[ \delta(e_j) = \text{Floor}(\pi_M(n_j)) - \text{Floor}(\pi_M(n_i)). \]  

(6)

Thus, for an alignment that corresponds to any process trace of \( M, \sum_{i \in S_2} \delta(e_j) \) is always a constant.

\[
\sum_{e \in S_2} \delta(e) = \text{Floor}(\pi_M(n_1)) - \text{Floor}(\pi_M(n_{\text{source}})) + \text{Floor}(\pi_M(n_2)) - \text{Floor}(\pi_M(n_1)) + \ldots + \text{Floor}(\pi_M(n_{M_{\text{max}}}))

- \text{Floor}(\pi_M(n_{M_{\text{max}} - 1})),

= \text{Floor}(\pi_M(n_{M_{\text{max}}})) - \text{Floor}(\pi_M(n_{\text{source}})) = C_2.

(7)

Therefore,

\[
\text{Cost}(\gamma) = \sum_{e_i \in E_B} \left( \delta(e_i) - \delta(e_i \uparrow) - \delta(\bar{e}_i) \right) + C_1 + C_2.

(8)

with \( C_1 \) and \( C_2 \) being constants.

5.2. Distance-based cost setting

According to (8), \( \text{Cost}(\gamma) \) only relates to \( \sum_{e_i \in E_B} \left( \delta(e_i) - \delta(e_i \uparrow) - \delta(\bar{e}_i) \right) \) up to a constant. \( \text{Cost}(\gamma) \) will be less when \( e \in E_B \) increases, provided that \( \delta(e_i) - \delta(e_i \uparrow) - \delta(\bar{e}_i) \) is negative for each element \( e_i \in E_B \). Thus, we want
\(\delta(e_1) - \delta(e_1) - \delta(e_1) < 0\), which leads to:
\[
\delta(e_y) < \delta(e_y) + \delta(e_y).
\]  \hspace{1cm} (9)

The triangular inequality theorem \[14\] will suffice for (9). Treating \(\delta(e_y)\), \(\delta(e_y)\) as the length of two shorter edges in a right-angled triangle, \(\delta(e_y)\) could just be the length of the longest edge. Then, the cost of an edge can be its length, i.e., the Euclidean distance, in that triangle. Thus, \(\delta(e_y) = \sqrt{(\delta(e_y)^2 + (\delta(e_y)^2))}\). In summary, a distance-based cost setting \(\delta(e_y)\) for \(e_y\) in \(\mathbb{E}_L, \mathbb{E}_M\) and \(\mathbb{E}_B\) of CPSs is defined as follows.

**Definition 12 (Distance-based cost setting).** In a CPS for a log trace \(\sigma_1\) and an execution space \(M\), cost of \(e_y\) is:
\[
\delta(e_y) = \begin{cases} 
1, & \text{if } e_y \in \mathbb{E}_L; \\
\Delta \text{Floor} = \text{Floor}(\pi_M(n_i)) - \text{Floor}(\pi_M(n_i)), & \text{if } e_y \in \mathbb{E}_M; \\
1 + (\text{Floor}(\pi_M(n_i)) - \text{Floor}(\pi_M(n_i)))^2, & \text{if } e_y \in \mathbb{E}_B. 
\end{cases}
\]

\[6. \text{Proof that the lowest-cost alignments have the maximum synchronous moves}\]

In this section, we provide a theoretical proof that the lowest-cost paths in CPSs have the maximum number of synchronous moves among all the possible paths.

A CPS \(G_{M, \sigma_1}\) can be seen as multiple coordinate planes sharing the same source and goal nodes and stacking together, \(g_{\sigma_{M_1}, \sigma_1}, g_{\sigma_{M_2}, \sigma_1}, \ldots, g_{\sigma_{M_N}, \sigma_1}\). Their vertical dimensions are \(\sigma_{M_1}, \sigma_{M_2}, \ldots, \sigma_{M_{\text{NumM}}}, \sigma_1\), respectively. Then, the lowest-cost alignments in \(G_{M, \sigma_1}\) are \(\gamma_1, \gamma_2, \ldots, \gamma_{\text{NumM}}\). Lemma 1 and Lemma 2 are used to establish a relationship between \(\gamma_k(1 < k \leq N\text{umM})\) and \(\gamma_1\) in terms of the number of synchronous moves.

**Lemma 1.** Let \(\gamma_1\) be the lowest-cost alignment in \(g_{\sigma_{M_1}, \sigma_1}\) whose vertical dimension is the longest process trace of \(M, \sigma_{M_1}\). If \(\gamma_k(1 < k \leq N\text{umM})\) contains a synchronous move \(e_y \in \mathbb{E}_B\) with \(\Delta \text{Floor} = \text{Floor}(a_j) - \text{Floor}(a_i) > 1\), then \(\gamma_1\) also has a synchronous move \(e_{y_j} \in \mathbb{E}_B\) with \(\Delta \text{Floor} = \text{Floor}(a_j) - \text{Floor}(a_i) = 1\) arriving at the same node \(n_j\) as \(e_y\).

**Proof.** Based on Definition 11, when \(\Delta \text{Floor} = \text{Floor}(a_j) - \text{Floor}(a_i) > 1\), \(a_i\) belongs to more than one process trace. And, one of them is the longest process trace. Herein, \(a_i\) belongs to \(\sigma_{M_k}\) and \(\sigma_{M_1}\).

In \(g_{\sigma_{M_k}, \sigma_1}\), the existence of \(e_y\) means that an event \(\sigma_{L}(j)\) makes up a synchronous move with \(a_i\). Since \(g_{\sigma_{M_k}, \sigma_1}\) and \(g_{\sigma_{M_1}, \sigma_1}\) have the same horizontal axis \(\sigma_{L}\), \(\sigma_{L}(j)\) can also make up a synchronous move with the same executed task \(a_i\) and event \(\sigma_{L}(j)\) in \(g_{\sigma_{M_1}, \sigma_1}\). Suppose the predecessor of \(a_i\) in \(\sigma_{M_k}\) is \(a_u\). Then, the existence of \(e_y\) indicates the existence of \(e_{y_j}\) with the same target node \(n_j\) in \(g_{\sigma_{M_1}, \sigma_1}\). So the existence of one synchronous move in \(\gamma_k\) indicates the existence of one synchronous move in \(\gamma_1\). (QED).

**Lemma 2.** Let the longest process trace of \(M\) be \(\sigma_{M_1}\). Then, the number of synchronous moves in \(\gamma_2, \gamma_3, \ldots, \gamma_{\text{NumM}}\) is not greater than the number of synchronous moves in \(\gamma_1\).

**Proof.** \(\gamma_2, \gamma_3, \ldots, \gamma_{\text{NumM}}\) are the lowest-cost alignments in \(g_{\sigma_{M_1}, \sigma_1}, g_{\sigma_{M_1}, \sigma_1}, \ldots, g_{\sigma_{M_{\text{NumM}}}, \sigma_1}\). In the corresponding 2D spaces, the vertical axes are \(\sigma_{M_1}, \sigma_{M_2}, \ldots, \sigma_{M_{\text{NumM}}}\). The increment of Floor values (Floor) between two neighboring executed tasks in \(\sigma_{M_1}, \sigma_{M_2}, \ldots, \sigma_{M_{\text{NumM}}}\) could be \(1, 2, 3, \ldots\). The lowest-cost alignment \(\gamma_k(1 < k \leq \text{NumM})\) contains synchronous moves having Floor either being 1 or greater than 1. For synchronous moves with Floor being 1, the executed tasks are also in \(\sigma_{M_1}\). For synchronous moves with Floor greater than 1, according to Lemma 1, a synchronous move \(e_y\) with Floor greater than 1 indicates the existence of synchronous move \(e_{y_j}\) in \(\gamma_1\). Besides, \(\sigma_{M_1}\) has more executed tasks than \(\sigma_{M_k}\). Thus, \(\gamma_1\) contains equal or more synchronous moves than \(\gamma_k\). (QED).

**Theorem 1.** Let \(\Gamma\) be the set of all possible alignments between an event trace \(\sigma_1\) and an execution space \(M\). The set of the lowest-cost alignments among \(\Gamma\) is denoted by \(\Gamma_{\text{min-Cost}}\) where the costs of edges adopt the Distance-based cost setting in Definition 12. Let Max\(\Omega\) be the maximum number of synchronous moves of all the alignments in \(\Gamma\). Then,
\[
\{\Gamma_{\text{min-Cost}}|\forall \gamma \in \Gamma [\text{Cost}(\gamma_{\text{min-Cost}}) \leq \text{Cost}(\gamma)] \} \forall \gamma \in \Gamma, \Omega(\gamma) \leq \text{Max}\Omega.
\]  \hspace{1cm} (10)

\[\Rightarrow \{\forall \gamma \in \Gamma_{\text{min-Cost}}, \Omega(\gamma) = \text{Max}\Omega\}.\]
The lowest-cost alignments in all the 2D spaces are \( \gamma_1, \gamma_2 \ldots \gamma_{\text{NumM}} \). Let \( \gamma_1 \) be the lowest-cost alignment in \( G_{M, \delta_1} \) whose vertical dimension is the longest process of \( M \). The proof is by case analysis. For any \( \gamma_k (1 < k \leq \text{NumM}) \), there are three cases:

1. \( \text{Cost}(\gamma_k) < \text{Cost}(\gamma_1) \).
2. \( \text{Cost}(\gamma_k) > \text{Cost}(\gamma_1) \).
3. \( \text{Cost}(\gamma_k) = \text{Cost}(\gamma_1) \).

Now, we have to show that: the theorem holds for all the three cases and that the lower-cost alignment has \( \text{Max}\Omega \) number of synchronous moves.

**Case 1:** \( \gamma_1 \) consists of log moves with cost being 1, model moves with cost being 1, and synchronous moves with cost being \( \sqrt{2} \). \( \gamma_k \) consists of log moves with cost being 1, model moves with cost being 1, 2, 3 ... and synchronous moves with cost being \( \sqrt{2}, \sqrt{5}, \sqrt{10} \ldots \) The model moves, log moves and synchronous moves can be regarded as edges in \( G_{M, \delta_1} \). Besides the edges that are shared by both \( \gamma_1 \) and \( \gamma_k \), there are some remaining edges in \( \gamma_1 \) and in \( \gamma_k \). Because \( \gamma_1 \) and \( \gamma_k \) share the same \( n_{\text{source}} \) and \( n_{\text{goal}} \) of \( G_{M, \delta_1} \), the remaining edges in \( \gamma_1 \) and in \( \gamma_k \) make up two paths between two same nodes in \( G_{M, \delta_1} \).

When \( \text{Cost}(\gamma_k) < \text{Cost}(\gamma_1) \), at least one of the triangles in Fig. 7 should be formed by the remaining edges in \( \gamma_1 \) and \( \gamma_k \). Fig. 7 lists conditions where \( \gamma_k \) has lower cost than \( \gamma_1 \) with the synchronous move \( e_0 \) of \( \gamma_k \) being \( \sqrt{5}, \sqrt{10}, \sqrt{17} \ldots \) However, \( \gamma_k \) has the same number of synchronous moves as \( \gamma_1 \). Thus, the theorem holds in this case that \( \gamma_k \) has the same number of synchronous moves as \( \gamma_1 \), which is \( \text{Max}\Omega \) based on Lemma 2.

**Case 2:** In Case 2, when \( \text{Cost}(\gamma_k) > \text{Cost}(\gamma_1) \), \( \gamma_1 \) is the lowest-cost alignment. The theorem holds, based on Lemma 2 that \( \gamma_1 \) has \( \text{Max}\Omega \) synchronous moves.

**Case 3:** According to (8), \( \text{Cost}(\gamma) \) only relates the number of synchronous moves \( e_1 \) in \( \gamma \) and \( \delta(e_1) - \delta(e_1) \). \( \delta(e_1) - \delta(e_1) \) can only be fixed values: \( \sqrt{2} - 2(\Delta \text{Floor} = 1), \sqrt{5} - 3(\Delta \text{Floor} = 2), \sqrt{10} - 4(\Delta \text{Floor} = 3) \) ... If \( \text{Cost}(\gamma_k) = \text{Cost}(\gamma_1) \), then \( \gamma_k \) and \( \gamma_1 \) have the same set of \( e_1 \). Thus, \( \gamma_k \) has equal number of synchronous moves as \( \gamma_1 \). Based on Lemma 2, \( \gamma_1 \) has \( \text{Max}\Omega \) synchronous moves. So, \( \gamma_1 \) and \( \gamma_k \) both with the lowest cost among \( \Gamma \) have the maximum number of synchronous moves. In other words, the theorem holds in this case.

The proof is complete: the lowest-cost paths are guaranteed to have maximum number of synchronous moves. (QED).

The optimal alignments \( \Gamma_{\text{opt}} \) should have not only the maximum number of synchronous moves, but also the minimum number of asynchronous moves. It can be easily achieved by sorting \( \Gamma_{\text{min}, \text{Cost}} \) on the number of asynchronous moves and taking the fewest ones.

7. Implementations of algorithms to compute optimal alignments

This section contains two subsections: computing the optimal alignments in one CPS and find the global optimal ones in multiple CPSs. The second subsection deals with execution spaces with loops.

7.1. Applying the Dijkstra algorithm for one CPS

In previous sections, optimal alignments have been translated into the lowest-cost paths in one CPS. The CPS has a single source node and a single goal node. This lowest-cost paths in single-source and single-goal directed graphs can be computed.
efficiently using the informed best-first algorithms (e.g. Dijkstra algorithm and A* algorithm) [10]. Various best-first algorithms have the same basic algorithmic descriptions, including node expansion, selecting the lowest-cost node in a set, etc. They differ mainly in the cost function, which influence its efficiency. Due to 100% optimality, the Dijkstra and A* algorithms are the most common ones. Because the Dijkstra algorithm adopts a simpler cost function than the A* [11], we implement the Dijkstra algorithm. It is worth noticing that other path-searching algorithms can be applied as well.

Algorithm 1 shows the Dijkstra algorithm. Basically, it prioritizes the nodes with higher merits in exploring (from the source node) neighboring nodes until the goal node. For the detailed description, we refer to [10]. Two issues are needed to be elaborate. One is the node expansion operator (in line 7) that explores neighboring nodes. The other is the cost function (line 9) that is used to prioritize nodes in the directed graph.

Algorithm 1 The Dijkstra algorithm for computing optimal alignments in \( G_{M, \sigma_L} \)

1: Let OPEN be an open set of nodes, place the node \( n_{source} \) of \( G_{M, \sigma_L} \) in OPEN;
2: while OPEN is not empty do
3: Remove from OPEN a node \( n \) whose cost is the lowest and place it in a set called CLOSED;
4: if \( n \) is a goal node then
5: Return \( n \); Succeed in finding the optimal alignment, the algorithm ends.
6: b.f end if
7: Perform node expansion on \( n \);
8: for each successor \( n' \) of \( n \) do
9: Calculate \( f(n') \);
10: if \( n' \) was neither in OPEN nor in CLOSED then
11: Add \( n' \) to OPEN and assign \( f(n') \) as \( n' \)’s cost;
12: else
13: if \( n' \) is not new.
14: Compare the new cost of \( n' \) to the old cost and keep the lower-cost \( n' \);
15: end if
16: end for
17: end while

7.1.1. Node expansion

For one node \( n_i \), its neighbors have three directions: horizontal, vertical and diagonal. Instead of sprouting \( n_i \) to all its neighbors, the neighbor on the diagonal direction should be prioritized. The reason is that a diagonal edge indicates a synchronous move. Asynchronous moves are only generated when synchronous moves are not possible according to the replay [24]. Thus, there are two situations: synchronous move is possible or not.

Definition 13 (Node expansion). For a given node \( n_i \) in the CPS \( G_{M, \sigma_L} \) for an execution space \( \sigma_L \), let \( \sigma_L(i) \) be the event to which \( n_i \) is projected on the log dimension, then \( \sigma_L(i) \)'s successor is \( \sigma_L(i + 1) \); let \( \sigma_M(i) \) be the executed task to which \( n_i \) is projected on the model dimension, then \( \sigma_M(i) \)'s successors in \( M_i \) a set of executed tasks \( S_i \subset \sigma_M \), an operation node expansion of \( n_i \) is:

- if \( \exists s_{good} \leftrightarrow \sigma_L(i + 1) \) with \( s_{good} \in S_i \), in which “\( \leftrightarrow \)” denotes an aligning relation between \( s_{good} \) and \( \sigma_L(i + 1) \):
  - generate an edge \( e_g \) connecting \( n_i \) to \( n_{g} \), in which \( \pi_M(n_i) = s_{good} \) and \( \pi_M(n_{g}) = \sigma_L(i + 1) \);
  - for each executed task \( s_i \in S_i \) except \( s_{good} \), generate an edge \( e_{ig} \) connecting \( n_i \) to \( n_{g} \), in which \( \pi_M(n_i) = s_i \) and \( \pi_M(n_{g}) = \sigma_L(i) \).
- else
  - for each executed task \( s_i \in S_i \), generate an edge \( e_{ig} \) connecting \( n_i \) to \( n_{g} \), in which \( \pi_M(n_i) = s_i \) and \( \pi_M(n_{g}) = \sigma_L(i) \);
  - generating an edge \( e_{ik} \) on the log dimension, which connects \( n_i \) to \( n_k \), \( \pi_M(n_k) = \sigma_M(i) \) and \( \pi_M(n_k) = \sigma_L(i + 1) \).

The “if” describes the first situation, and the “else” describes the second situation. Notably, in the first situation, even though one executed task \( s_{good} \) gets aligned to \( \sigma_L(i + 1) \), the remaining executed tasks of \( S_i \) still could contribute to a more meritorious path in a later stage than \( e_g \) contributed by \( s_{good} \). Therefore, for each remaining executed task \( s_i \) in \( S_i \), an edge on the model dimension is generated. In the second situation, a log move is generated.

7.1.2. The cost function \( f(n) \)

In the Dijkstra algorithm, \( f(n) \) takes the cost of currently formed path as an estimate for the merit of becoming the lowest-cost path. The cost \( f(n) \) of a given node \( n \) in \( G_{M, \sigma_L} \) can be written as:

\[
f(n) = \sum_{e \in \text{Priority} \to n} \delta(e),
\]
in which \( E_{\text{source} \rightarrow n} \) is a set of edges consisted in the path from \( n_{\text{source}} \) to \( n \), denoted by \( p_{\text{source} \rightarrow n} \).

7.2. Computing optimal alignments in multiple CPSs

For an execution space having loops, different times of unfolding the loops result in the variety in length as each unfolding makes the execution space larger. This variety makes the CPS method not applicable because \( n_{\text{goal}} \) of CPS represents the termination state of the execution space. To deal with it, we could unfold the loops such that the flatten execution spaces don’t have loops. Then, the CPS method can be applied for each of them.

To find the one from multiple flatten execution spaces whose optimal alignments have more synchronous moves than the rest, the unfolding process should cover all the possible unfolding scenarios. It contains two aspects, the times of unfolding the same loop and the orders of different loops. This can be done in an incremental way. For instance, take the execution space in Fig. 8a which contains \( k \) loops. Leaving out all the loops yields \( M_0 \). Based on \( M_0 \), we can unfold each of \( k \) loops once, which yields \( M_1, M_2, \ldots, M_k \). Based on the unfold-once execution spaces, unfold-twice execution spaces can also be generated. Based on \( M_1 \), each of the \( k \) loops can be unfolded. Notably, it cannot always unfolding all the loops. For example, after the loop from \( a_5 \) to \( a_3 \), the loop \( a_2 \) to \( a_1 \) cannot be unfolded. Via this incremental manner, all the possible unfolding can be generated. Fig. 8b shows the resulting multiple execution spaces.

Though the number of flatten execution spaces can be infinite, we can stop the unfolding process when the best one is found. It is possible due to the parent–child relations in the tree as shown in Fig. 8b. Since the child execution space is larger than its parent execution space, we could stop the unfolding of that branch when the child is no better than the parent. The underlying principle is that when the parent execution space has reached a maximum aligning level with the log trace, its children execution spaces will only obtain a decreased aligning level due to additional tasks.

8. Experiments

We implemented the proposed method in Python 3.7. The code is available online at https://github.com/HuiY/ComptOptAlignments.git. Besides, the original BPMN models, their execution spaces (in.grs and.png), event logs and the results from these two experiments can be obtained from https://github.com/HuiY/DataSetForPaperCoordinateSystem.git.

8.1. Process model without loops

The first experiment is for execution spaces without loops. The purpose is to validate the optimality, i.e., the returned optimal alignments always having the maximum synchronous moves. The process model is a diagnosis validation process for unstable angina in Catharina hospital in the Netherlands [29]. Its execution space is shown in Fig. 6b, which contains three
Exclusive Splits and three Exclusive Joins. The log traces are artificially generated. We firstly generated three perfectly fitting traces with different length: 5, 10 and 14. Then we introduced deviations by randomly removing and inserting events (artificial omissions and additions) into the three perfectly fitting traces. The information of the artificial log traces is shown in Table 1, which covers deviation ratio\(^1\) ranging from 6% to 86.7%. In total, we generated 25 logs, each of which contains 20 log traces.

### Table 1
Log information.

| Log | Length | Omission | Addition | NumDev | NumBoth | Log | Length | Omission | Addition | NumDev | NumBoth |
|-----|--------|----------|----------|--------|---------|-----|--------|----------|----------|--------|---------|--------|
| 1   | 6      | 0        | 1        | 1      | 5       | 13  | 15     | 0        | 1        | 1      | 14      |
| 2   | 5      | 1        | 1        | 2      | 4       | 14  | 14     | 1        | 1        | 1      | 2       |
| 3   | 6      | 1        | 2        | 3      | 4       | 15  | 15     | 1        | 1        | 2      | 3       |
| 4   | 5      | 2        | 2        | 4      | 3       | 16  | 14     | 2        | 2        | 4      | 12      |
| 5   | 11     | 0        | 1        | 1      | 10      | 17  | 15     | 2        | 3        | 5      | 12      |
| 6   | 10     | 1        | 1        | 2      | 9       | 18  | 14     | 3        | 3        | 6      | 11      |
| 7   | 11     | 1        | 2        | 3      | 9       | 19  | 15     | 3        | 4        | 7      | 11      |
| 8   | 10     | 2        | 2        | 4      | 8       | 20  | 14     | 4        | 4        | 8      | 10      |
| 9   | 11     | 2        | 3        | 5      | 8       | 21  | 15     | 4        | 5        | 9      | 10      |
| 10  | 10     | 3        | 3        | 6      | 7       | 22  | 14     | 5        | 5        | 10     | 9       |
| 11  | 11     | 3        | 4        | 7      | 7       | 23  | 15     | 5        | 6        | 11     | 9       |
| 12  | 10     | 4        | 4        | 8      | 6       | 24  | 14     | 6        | 6        | 12     | 8       |
|     |        |          |          |        |         | 25  | 15     | 6        | 7        | 13     | 8       |

Fig. 9. Comparing the number of correct traces resulted from running the Dijkstra algorithm with standard cost setting vs. distance-based cost setting. Log 1–4 (length of 5–6) have increasing deviation number from 1 to 4. Log 5–12 (length of 10–11) have increasing deviation number from 1 to 8. Log 13–25 (length of 14–15) have increasing deviation number from 1 to 13.

\(^1\) Deviation ratio is calculated by the number of omissions and additions divided by the length of alignment.
Table 2
Efficiency and optimality for Replay a Log on Petri Nets in ProM 6.10 and the coordinate-plane search space method of this paper.

<table>
<thead>
<tr>
<th>Log trace ID</th>
<th>Length</th>
<th>Replay plug-in in ProM 6.8</th>
<th>Our method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ω</td>
<td>(\mathcal{E}_{dev})</td>
<td>s</td>
</tr>
<tr>
<td>1539302414925</td>
<td>66</td>
<td>20 54 1213 38</td>
<td>22 52 15 16067 1150</td>
</tr>
<tr>
<td>1539303857517</td>
<td>62</td>
<td>22 46 843 7</td>
<td>25 42 15 14719 843</td>
</tr>
<tr>
<td>154768734202</td>
<td>60</td>
<td>23 42 778 5</td>
<td>25 40 15 14354 725</td>
</tr>
<tr>
<td>154772738650</td>
<td>68</td>
<td>22 52 1002 7</td>
<td>24 50 15 16257 887</td>
</tr>
<tr>
<td>154769394308</td>
<td>62</td>
<td>23 44 717 4</td>
<td>26 40 15 14712 971</td>
</tr>
<tr>
<td>1547915248799</td>
<td>62</td>
<td>22 46 754 6</td>
<td>24 44 15 14776 913</td>
</tr>
<tr>
<td>1547698148253</td>
<td>82</td>
<td>27 64 1640 53</td>
<td>32 60 115 143588 8946</td>
</tr>
<tr>
<td>1547986862398</td>
<td>64</td>
<td>25 42 651 4</td>
<td>27 40 15 14721 1182</td>
</tr>
<tr>
<td>1539314415211</td>
<td>118</td>
<td>32 94 2668 19</td>
<td>40 102 99 177565 1367</td>
</tr>
<tr>
<td>1547997805806</td>
<td>62</td>
<td>24 42 629 2</td>
<td>27 38 15 14533 853</td>
</tr>
<tr>
<td>154803283992</td>
<td>62</td>
<td>24 42 642 2</td>
<td>26 40 15 14714 754</td>
</tr>
<tr>
<td>154803400723</td>
<td>54</td>
<td>23 36 415 2</td>
<td>25 34 15 12961 819</td>
</tr>
<tr>
<td>1548037113729</td>
<td>78</td>
<td>24 58 1432 13</td>
<td>26 56 15 18890 1094</td>
</tr>
<tr>
<td>1539316889981</td>
<td>92</td>
<td>25 78 2271 71</td>
<td>29 76 103 144482 9922</td>
</tr>
<tr>
<td>1539734942389</td>
<td>72</td>
<td>21 58 1454 11</td>
<td>24 54 15 17512 1136</td>
</tr>
<tr>
<td>1539737717686</td>
<td>70</td>
<td>22 62 1758 27</td>
<td>29 58 115 122743 6639</td>
</tr>
<tr>
<td>1539739275781</td>
<td>76</td>
<td>24 64 613 12</td>
<td>31 62 123 141896 9687</td>
</tr>
<tr>
<td>1539740347920</td>
<td>60</td>
<td>21 46 804 4</td>
<td>23 44 15 14644 1418</td>
</tr>
<tr>
<td>1539831113678</td>
<td>52</td>
<td>21 38 636 3</td>
<td>23 36 15 12709 698</td>
</tr>
<tr>
<td>1539832275246</td>
<td>72</td>
<td>25 58 1499 23</td>
<td>30 54 103 113237 6175</td>
</tr>
</tbody>
</table>

Fig. 10. An execution space with loop structures.
We compared our distance-based cost setting and the standard cost setting \[10,15\] both using the Dijkstra algorithm. Standard cost setting means cost value being zero for synchronous moves and being one for asynchronous moves. For each version, the algorithm returns a lowest-cost alignment for each log trace. By counting the number of synchronous moves and comparing to the supposed count (the column NumBoth in Table 1), we can determine whether the returned alignment is optimal or not. The result is shown in Fig. 9. Our distance-based cost setting always resulted in 20 correct traces for all 25 logs. The standard cost setting failed in finding all the alignments with maximum synchronous moves when the deviation ratio is up to 28% in log 16.

8.2. Process model with loops

This second experiment is for execution spaces with loops. The process model and an event log are from the 1st Conformance Checking Challenge 2019 (CCC2019) \[1\]. CCC2019 published a BPMN 2.0 model describing a real-life Central Venous Catheter (CVC) installation training process for medical students \[18\]. They also published an event log containing 20 real-life traces with the length ranging from 52 to 118.

We compared our results with that of an existing approach \[2\]. The latter is recognized as the only method with guaranteed optimality according to \[9\]. This method is implemented as a plugin “Replay a log on Petri net for all optimal alignments” in the ProM 6.10. Notably, the selected algorithm is “Graph-based state space replay to obtain all optimal alignments” with cost setting being defaulted. This plugin accepts BPMN models as input, so the two methods take the exactly same input. For each log trace, the Prom plugin outputs the resulting lowest-cost alignments, the number of queued states (\(Q\)), and the time (T(ms)) it took. The numbers of synchronous moves (denoted as \(X\)) and asynchronous moves (denoted as \(E_{dev}\)) in each result alignment can be obtained by counting. The detailed results are shown in columns 3 to 6 in Table 2.

We implemented our method, namely the Dijkstra algorithm using the distance-based cost setting, in Python 3.7 in a computer with the core Intel Core i5-7200U 2.5 GHz (4 cores), 8 GB RAM. As our algorithm accepts execution spaces as input, we firstly generated \[28\] the execution space of the BPMN CVC installation training process model. It is shown in Fig. 10. Running the algorithm, we record the number of synchronous moves (\(\Omega\)), the number of deviations (\(E_{dev}\)), the number of sub-coordinate planes (\(\text{Spaces}\)), the number of queued nodes (\(Q\)) and the consumed time for each result alignment. They are shown in columns 7 to 11 in Table 2.

The numbers of synchronous moves \(\Omega\) of the two methods are plotted in Fig. 11. It shows that our method aligns more synchronous moves than the ProM 6.8 plugin for all the log traces. As the columns 4 and 8 of Table 2 show, our method also achieved fewer deviations for 19 traces. The exception is the trace with “ID: 1539314415211”), which has eight more deviations in our method than the Prom method. This is reasonable as our method finds more (eight) synchronous move than the Prom method. Thus, our method achieves better optimality than the existing “Relay a log on Petri net for all optimal alignments” \[2\] in general. Though our method takes more nodes and more time than the Prom plugin, which are shown in the columns 5 vs. 10 and 6 vs. 11, they are still acceptable.
9. Conclusion

To tackle the twofold (efficiency and optimality) challenge in computing optimal alignments, we translated the optimal alignments into the lowest-cost paths in coordinate planes. Firstly, we proposed a new concept, CPSSs, which allow enumerating all the possible alignments between a log trace and an execution space without loops. Secondly, to speed up the search (for optimal alignments) efficiency, we studied the conditions under which the optimal alignments are the lowest-cost paths in one CPS. These conditions are based on theoretical derivation and lead to the design of the costs of edges: distance-based cost setting. Thirdly, we implemented the Dijkstra algorithm in finding the optimal alignments, including cost function specification and node expansion operators.

For execution spaces with loops, unfolding the loops causes larger no-loop execution spaces, each of which has its own optimal alignments. They can be computed using the single-source and single-goal CPS method. To find the global optimal alignment, we investigated the mechanism covering all the possible unfolding scenarios and when to stop the unfolding.

We carried out experiments for execution spaces without and with loops respectively. For the execution space without loops, artificially generated log traces were used to test whether our method is able to find the correct number of synchronous moves. The result showed our method has a 100% optimality even the deviation ratio is up to 86.7%. For the execution space with loops, we compared our method with an existing one that is recognized as the only one having the guaranteed optimality. The results showed that our method achieves more synchronous moves and fewer deviations at the same time.

As the main focus of this paper is the optimality, one limitation is that efficiency is not fully evaluated. In future work, more datasets can be used to evaluate the efficiency when the optimality is guaranteed using our CPSSs. For given execution spaces and log traces, other lowest-paths search algorithms can also be applied. They will achieve the same optimality using the proposed CPSSs and distance-based cost setting. In the future work, the A* algorithm will be studied for a better efficiency. For on-line conformance checking where the events come in streaming, dynamic versions of lowest-path search algorithms can be applied using the CPSSs. In future, other cost settings can be studied to achieve multiple-perspective (time, data, etc.) conformance checking.

CRediT authorship contribution statement

Hui Yan: Conceptualization, Writing - original draft, Methodology, Software. Uzay Kaymak: Methodology, Writing - review & editing. Pieter Van Gorp: Methodology, Writing - review & editing. Xudong Lu: Formal analysis, Supervision. Shan Nan: Formal analysis, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research leading to these results has received funding from the Brain Bridge Project sponsored by Phillips Research.

References