Efficiently Computing Alignments
Algorithm and Datastructures

Boudewijn F. van Dongen

Eindhoven University of Technology B.F.v.Dongen@tue.nl

Abstract. Conformance checking is considered to be anything where observed behaviour needs to be related to already modelled behaviour. Fundamental to conformance checking are alignments which provide a precise relation between a sequence of activities observed in an event log and a execution sequence of a model. However, computing alignments is a complex task, both in time and memory, especially when models contain large amounts of parallelism.
In this tool paper we present the actual algorithm and memory structures used for the experiments of [15]. We discuss the time complexity of the algorithm, as well as the space and time complexity of the main data structures. We further present the integration in ProM and a basic code snippet in Java for computing alignments from within any tool.

Keywords: alignments, conformance checking, process mining

1 Introduction

Conformance checking is considered to be anything where observed behaviour needs to be related to already modelled behaviour. Conformance checking is embedded in the larger contexts of Business Process Management and Process Mining, where conformance checking is typically used to compute metrics such as fitness, precision and generalization to quantify the relation between a log and a model. Fundamental to conformance checking are alignments [2]. Alignments provide a precise relation between a sequence of activities observed in an event log and a execution sequence of a model. For each trace, this precise relation is expressed as a sequence of "moves". Such a move is either a "synchronous move" referring to the fact that the observed event in the trace corresponds directly to the execution of a transition in the model, a "log move" referring to the fact that the observed event has no corresponding transition in the model, or a "model move" referring to the fact that a transition occurred which was not observed in the trace. Computing alignments is a complex task, both in time and memory.

In [15] a technique is presented to efficiently compute alignments for Petri nets, using the extended marking equation. In this paper, we present the algorithm in pseudocode and the datastructures used. We discuss the time and memory complexity for the algorithm and we provide a code snippet for actually

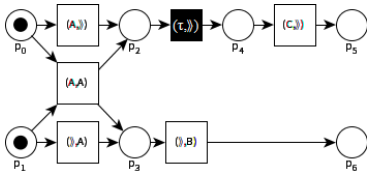

Fig. 1. Example synchronous product.

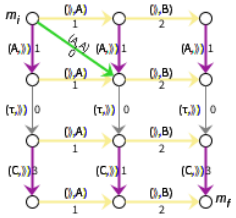

Fig. 2. Example search space.

using the code which is publicly available. We assume the reader to be familiar with some basic notions of Petri nets.

The problem of finding alignments can be expressed the following way: *Given a synchronous product Petri net and a cost function, provide the cheapest firing sequence from the initial marking to the final marking.* Consider the example in Figure 1. Here, a synchronous product for a sequential Petri net with three transitions, labelled A, τ and C and a trace containing two events A and B is depicted. The reachability graph of this model is shown in Figure 2 and for an alignment, we look for the shortest path from the top-left marking $m_i = [p_0, p_1]$ to the bottom-right marking $m_f = [p_5, p_6]$. The distances at each edge are determined by a cost function which associates costs to firing transitions in the synchronous product. For a complete formal specification of this question, we refer to [15], however the following key elements are important for alignments:

- The synchronous product is a Petri net with both an initial and final marking. The final marking is assumed to be reachable from the initial marking through at least one firing sequence,
- Each transition in the synchronous product corresponds to a so-called move in the alignment. These moves are "model-moves", "log-moves", "synchronous-moves" or "τ-moves",
- Synchronous and τ moves typically have cost 0, while model and log moves have cost >0,
- The cost to reach the final marking can be underestimated using the (extended) marking equation. This requires solving a (integer) linear program.
- The size of the reachability graph of the synchronous product equals the size of the reachability graph of the model times the length of the trace that is being aligned.

The remainder of this paper is structured as follows. In Section 2, we discuss some related work. Then in Section 3, we discuss the algorithm in detail and in Section 4 the data structures needed. In Section 5 we discuss the inclusion in ProM and we conclude the paper in Section 6.

2 Related Work

In [12], Rozinat et. al. laid the foundation for conformance checking. They approached the problem by firing transitions in the model, regardless of available

tokens and they kept track of missing and remaining tokens after completion of an observed trace. However, these techniques could not handle duplicate labels (identically labelled transition occurring at different locations in the model) or "invisible" transitions, i.e. τ-labelled transitions in the model purely for routing purposes.

As an improvement to token replay, alignments were introduced in [3]. The work proposes to transform a given Petri net and a trace from an event log into a synchronous product net, and solve the shortest path problem using $A^\star$ [8] on the its reachability graph. This graph may be considerable in size as it is worst-case exponential in the size of the synchronous product, in some cases, the proposed algorithm has to investigate the entire graph despite of tweaks identified in [16].

In [15], an incremental technique is presented to compute alignments more efficiently using so-called splitpoints and the extended marking equation. However, no pseudo code is presented, nor a discussion on the main algorithm and data structures.

Several techniques exist to compute alignments. Planner-based techniques [6] are available for safe Petri nets. For safe, acyclic models, constraint satisfaction [9] can be used. When not using a model, but its reachability graph as input, automata matching [11] can be applied. Other conformance checking work includes decomposition-based approaches [1,10] and approximation schemes [13].

3 Algorithm

The core algorithm for alignments is an $A^\star$ search technique, which is shown in Algorithm 1. In lines 2 to 10, the algorithm is initialized. Then the mean search loop between 11 and 47 iteratively expands the search space. In every iteration, a marking is selected for expansion (line 12). This marking m is chosen such that it minimizes the cost to reach that marking from the initial marking $g(m)$ plus the estimated remaining cost $h(m)$. The algorithm stops if the final marking m_f is reached.

As explained in [15], estimating the remaining distance is done in two stages. First, an underestimate is "guessed" (line 40) and second, it is computed exactly (line 21). Lines 16 through 27 handle the case in which the selected marking m has a "guessed" estimate in which cast it is part of the set Y. The incremental nature of the work presented in [15] is shown in lines 17 through 20. In this part, a splitpoint is added to the set K after which the algorithm is restarted from scratch. The value added to the set K is s which represents the index of the last event explained by the closed markings in set A. This can be implemented without any recursion, so there is no memory overhead here.

Lines 28 through 46 make up the regular $A^\star$ algorithm. The selected marking m is expanded by iterating over all enabled transitions and firing them to obtain new markings m'. If a marking m' was already closed, we continue with the next marking. If it was not closed, then it is added the the open set (if it wasn't already in there) in line 33 and a temporary value for g is computed in line 34.

Each marking $m'$is then assigned a value for $g(m')$ and $h(m')$, where the latter depends on whether it can be derived from $h(m)$ (line 38) or whether it has to be "guessed" (lines 40-41). Furthermore, the predecessor relation p is updated in line 43.

In this algorithm, the set of splitpoints K is automatically updated whenever the algorithm reaches a marking which does not have an exact heuristic, but only a guessed value and s is currently not in the set of splitpoints. This incremental way of computing alignments has proven to be the most efficient way to date [15]. However, removing lines 17 through 20 from the algorithm and ignoring the set K would yield the alignment algorithm as presented in [2,3].

3.1 Time Complexity

$A^\star$ relies on a heuristic function to underestimate the remaining cost. In traditional alignments, this underestimation function is an (integer) linear program with the number of rows equal to the number of places in the synchronous product and the number of columns equal to the number of transitions. For many markings reached in the search for an optimal alignment such a linear program is solved. This leads to the fact that for the experiments in [15], 99% of the time of the classic $A^\star$ is spent on solving such linear programs using LPSolve [5].

In the incremental $A^\star$ technique of [15], an incremental heuristic function is proposed that uses the extended marking equation with a number of splitpoints. The heuristic function is a linear program of which the number of rows and columns scale linearly in the size of K. Solving a linear equation system is polynomial in the rank of its matrix (hence in the number of rows and columns). The algorithm proposed in [15] and detailed in Algorithm 1 maximizes reuse of previously computed solutions to these linear programs. while the linear programs are more complex than for classical $A^\star$, far fewer have to be solved. The total fraction of the time spent in the LP Solver for the experiments in [15] was 77% for the incremental $A^\star$. The remaining 23% of the time is spent on the internal computations within Algorithm 1, therefore it is important to choose the right data structures for use with this algorithm.

As the algorithm computes an alignment for a single trace, it is rather straightforward to parallelize it when aligning an entire log. By simply distributing traces over multiple cores of a CPU, walltime can be gained. However, as usual with multi-threading on a single CPU, this comes at a cost of synchronization. For the experiments presented in [15], going from 1 thread to 4 on a CPU with 4 physical cores and 4 hypercores, causes the wallclock time to reduce to 27%. The sum over the CPU times per trace increases by 35% in that case.

In the algorithm, there are several core data structures each with a specific set of requirements. In the next section, we discuss each data structure independently.

Procedure 1 $A^\star$ Algorithm for Alignments with estimated heuristics

Let $SN = ((P, T, F, \lambda), m_i, m_f)$ be a synchronous product and let $c : T \to \mathbb{N}$ be a cost function and $h : \mathsf{RS} \to \mathbb{N}$ be a heuristic underestimating the cost of getting from any marking to the final marking.

function ASTAR(SN, O, c, K)
  $A \leftarrow \emptyset$ ▷ Initialise closed set
  $X \leftarrow \{m_i\}$ ▷ Initialise open set as a priority queue favouring markings for which the exact heuristic is known.
  $Y \leftarrow \emptyset$ ▷ Initialise estimated heuristics
  $s \leftarrow 0$ ▷ Initialise the number of events explained
  $p(m_i) = (\tau, \tau)$ ▷ Initialise predecessor function
  $\forall_{m \in \mathsf{RS}(SN)}\ d(m) = \infty$ ▷ Initialise cost so far function d
  $g(m_i) = 0$
  $\forall_{m \in \mathsf{RS}(SN)}\ f(m) = \infty$ ▷ Initialise estimated total cost function f
  $f(m_i) = h(m_i, K)$ ▷ Compute estimate for initial marking with the splitpoints in K
  while $X \neq \emptyset$ **do** ▷ While not all states visited
    $m \leftarrow m \in X$ minimizing $f(m)$ ▷ Get the most promising marking m
    if $m = m_f$ **then** ▷ final marking reached
      break while
    end if
    if $m \in Y$ **then** ▷ Heuristic of m is not exact
      if $s \notin K$ **then** ▷ Check if s is not already a splitpoint in K
        $K \leftarrow K \cup \{s\}$ ▷ Add the maximum events explained to k
        return Astar(SN, O, c, K) ▷ Restart with a longer list of splitpoints.
      end if
      $x \leftarrow h(m)$ ▷ Compute the true estimate
      $Y \leftarrow Y \setminus \{m\}$ ▷ Remove estimated heuristic
      if $x > \hat{h}(m)$ **then** ▷ Heuristic increased
        $f(m) \leftarrow g(m) + h(m)$ ▷ Update estimated total cost function
        continue while ▷ Note: m may not be minimizing f any more
      end if ▷ If heuristic did not chance, continue with m
    end if
    $A \leftarrow A \cup \{m\}$ ▷ Add m to the closed set
    $X \leftarrow X \setminus \{m\}$ ▷ Remove m from the open set
    $s \leftarrow max(s, \text{events explained by } m)$ ▷ keep track of the maximum number of events explained
    for all $t \in T'$ with $m[t\rangle m'$ **do** ▷ For each relevant transition enabled in m
      if $m' \notin A$ **then** ▷ Reaching a marking not yet visited, or found shorter path
        $X \leftarrow X \cup \{m'\}$ ▷ Add m' to the open set
        $a \leftarrow g(m) + c(t)$ ▷ Compute the cost so far of reaching m' via m
        if $a < g(m')$ **then** ▷ If this current cost is better than known cost so far
          $g(m') \leftarrow a$ ▷ Update cost so far function
          if $h(m', K)$ can be derived from $h(m, K)$ **then**
            $f(m') \leftarrow g(m') + h(m', K)$ ▷ Update estimated total cost function
          else
            $Y \leftarrow Y \cup \{m'\}$ ▷ Add m' to the estimated heuristics set
            $f(m') \leftarrow g(m') + h(m, K) - c(t)$ ▷ Update estimated total cost function
          end if
          $p(m') \leftarrow (t, m)$ ▷ Update predecessor function
        end if
      end if
    end for
  end while
  $\gamma \leftarrow \langle t_0, \ldots, t_n \rangle$ such that $t_n = \#_1(p(m_f))$, $t_{n-1} = \#_1(p(\#_2(p(m_f))))$ etc. until the initial marking is reached recursively.
  return $f(m_f), \gamma$ ▷ Return distance and alignment
end function

4 Data Structures

The alignment algorithm contains a number of core data structures on which it operates. In this section, we discuss each data structure and we provide insights into the operations performed on these structures as well as the memory requirements for them. We start with the the most prominent concept in the algorithm, namely the marking.

4.1 Marking m

A marking is represented by a 31 bit number (the Java type of signed integer). Apart from the initial and final marking, which are represented as arrays, no marking is ever represented explicitly in memory. Instead, whenever marking m is used in the code, it is constructed by following the p relation back to the origin of the search graph. Markings are simply numbered from 0 upwards. This implies that there is a bit of computational overhead in line 12 of the algorithm when marking m is retrieved from the open set. Here, the integer id of marking m is returned after which the marking itself is constructed into an array with a byte value for each place (we assume no more than 128 tokens in a place at any point in time).

For each marking, we store the value of the g function, the value of the h function and the predecessor relation p in a total of 128 bits in an array indexed by the marking id (of which 2 bits are still unused). These values only have to be retrieved or written for a given marking hence using an array with the marking id as index makes these operations O(1).

4.2 Predecessor Relation p

The predecessor relation p stores the transition t in the synchronous product that was fired to reach a marking as well as the marking in which t was fired. We use this predecessor relation in two parts. First, we use it to compute the actual alignment (line 48). Furthermore, we use it throughout the code to construct a marking explicitly as we only store marking ids, not explicit markings. We store the predecessor transition in 31 bits unsigned (where one value is reserved for indicating a non-existing predecessor). The predecessor marking is also stored using its 31 bit id.

4.3 Cost-so-far Function g

The cost through which a marking is reached is stored as a 31 bit unsigned value. This implies that the cost to reach a marking can be at most 2,147,483,647. This is important when selecting a cost function for non-synchronous moves. Selecting too high cost values for certain moves will lead to internal overflow (which is detected and reported as such by the Java code).

4.4 Heuristic Function h

Like the g function, we also store the h function in 31 unsigned bits. In addition, we store a flag of 1 bit to indicate if the heuristic is exact (line 24 and line 42) or if the heuristic is estimated (line 45).

Together, p, g and h make up 125 bits. We use one more bit to store a flag indicating if the marking is in the closed set or not. The closed set is not represented in another way and in line 33 of the algorithm, this flag is checked to check if m' is in the closed set.

To store these functions we allocate 16 kByte arrays initially. Whenever 1,024 markings have been visited, we allocate an additional block of 16 kByte.

4.5 Open Set X

Perhaps the most interesting data structure is the open set X. This set should allow for efficient removal of the element minimizing f and favouring exact heuristics over estimated ones (line 12), efficient checking of inclusion (line 33) and efficient insertion (line 34). Furthermore, because of the estimated heuristics, it should allow for updating the locations of markings in the set whenever their estimated heuristic function is overwritten by an exact heuristic (line 24).

This set is implemented as a hashtable-backed, balanced binary heap using a sorting procedure which uses the f score as a first order sorting criterion. The second order sorting criterion is the availability of an exact estimate of a marking and the third order sorting is based on the g score (favouring markings with a higher g score). The heap itself is an array of markings (array of integers). The hash-table is needed to efficiently check at what location a marking is in the queue.

The memory use of this heap is limited to the array of markings (4 bytes per marking in the queue) and the hashtable. The latter uses an implementation from an external package which consumes 13 bytes per element for which the table has capacity. With the default load factor of 50%, this implies approximately 26 bytes per marking in the queue.

Checking if a marking exists in the queue as well as inspecting the head element are O(1) operations. All others (insertion, removal, updating) are O(log(n)) operations. However, the performance of this set depends on the hash function used. We experimented with a large collection of hash functions for arrays of bytes (recall markings are arrays of bytes with one byte per place). The default Java function performed equally well as others, both in terms of minimal hash collisions and time.

As markings are not represented explicitly, computing hash codes for a specific marking is a more complex task than usual. Each marking for which the hashcode needs to be computed needs to be instantiated fully first by following the predecessor function p. This implies that the open set should aim to minimize the hash collisions so that, for any insertion, only one hash code has to be computed, i.e. the hashcode of the marking to be inserted.

Table 1. Time and collision test for hashcode on 1,048,576 semi-random arrays, expecting 128 hash collisions.

Reference	hashCode name	CPU time (s)	Collisions
[4]	AdHASH	0.506	115
[7]	FNV1	0.519	137
[7]	FNV1a	0.532	119
https://tanjent.livejournal.com/755617.html	MurMur3	0.270	135
http://www.burtleburtle.net/bob/c/lookup3.c	Jenkings	0.309	125
http://www.cse.yorku.ca/~oz/hash.html	Bernstein	0.315	113
Default Java	JAVA31	0.311	126

4.6 Hashing markings

To maximize the speed of the algorithm, we tested a large collection of hashcodes. The results are shown in Table 1. For a total of 1,048,576 semi-random arrays, we expect 128 hash collisions, i.e. for 128 arrays, we expect that they have the same hashcode as another array, whilst not being equal. As shown, the tested hashcodes perform equally well in terms of collisions. The CPU time shows that the default Java implementation for hashing an array is in the lower range, so we decided to use this. Murmur3 is slightly faster, but with more collisions. Note that the speed of the Java default may have to do with the fact that there is a bytecode translation available in Java which is optimized for speed.

4.7 Closed Set A

The closed set A contains all markings which have been visited before. In the code, each marking carries a flag to indicate if it was closed or not. The set A is therefore not directly represented. Instead, the code uses a hashset to store the union of A and X, i.e. the set of all visited markings. The two operations on this set are insertion (line 28) and checking the presence of an element (line 32). The memory use is limited to 4 bytes per element for which the set has capacity, which, with a default load factor of 50%, implies 8 bytes per element. Insertion is a O(log(n)) operation, whereas checking presence of an element is amortized O(1). As for the open set, hash collisions should be avoided as much as possible as computing hash codes for stored markings is relatively expensive.

4.8 Solution Vector Cache for Function h

As indicated in [15], the solution vectors for the linear programs can be reused to obtain new solutions. This requires us to store the solution vectors for the linear programs in memory. We do so in a hashmap, mapping markings to these vectors. Each vector is represented by a minimal number of bits. For each solution vector, we need as many bits per transition as the maximum number in the vector requires (i.e. if the vector contains a value 4, then we need 3 bits per transition to store this vector). We limit the number of firings to 255, so the maximum number of bits we need per transition is 8. A header is used to store a flag of 1 bit indicating if the solution was computed (line 10, line 21) or derived (line 38)

and we use 3 bits to store the number of bits per transition used, i.e. we have a 4 bit header per vector. The vectors are stored in a hashmap mapping markings to the arrays.

4.9 Set of Estimated Heuristics Y

The set of estimated heuristics Y does not need explicit representation. Instead, a 1-bit flag is stored in the function h to indicate if an estimate is exact or estimated.

4.10 Memory use of the LP Solver

LPSolve [5] is used as an LP solver to obtain values for the heuristic function. LPSolve stores the linear programs in memory. This linear program is essentially a large, but very sparse, matrix of double values. We estimate that, for each non-zero element 16 bytes are used, 4 to store the row, 4 to store the column and 8 for the actual value. Then some internal arrays are needed to perform the calculations, but is is limited.

Overall, the memory overhead of the algorithm is very low. For all experiments conducted in [15], the actual memory used internally in the algorithm presented as Algorithm 1 was 8 Megabytes, not counting the overhead of storing the Petri net or the event log in Java memory.

5 Implementation

The algorithm presented as Algorithm 1 has been implemented in ProM [14] as part of the "Alignment" package. It is one of the algorithms available for the plugin entitled "Replay a log on Petri net for Conformance Analysis" and it is selected by default if it is installed. Figure 3 shows a screenshot of ProM with the plugin highlighted.

The use of this plugin is not different from the traditional alignment plugin presented in [2, 3]. Both the input and the output are the same and the implementation seamlessly integrates with ProM.

The algorithm can also be used in a stand-alone fashion from the command line. The code snippet depicted in Table 2 shows how to do this in Java code.

6 Conclusions and Future Work

This paper presents, in detailed pseudo-code, the efficient alignment algorithm presented in [15]. Computing an alignment for a given process model and trace is, in general, a complex and time consuming task. In part, this complexity stems from the fact that the search space of the $A^\star$ algorithm is worst case the size of the reachability graph of the model times the length of the trace. A second contributing factor to the complexity is the heuristic function used, but also the choice of data structures in the core algorithm.

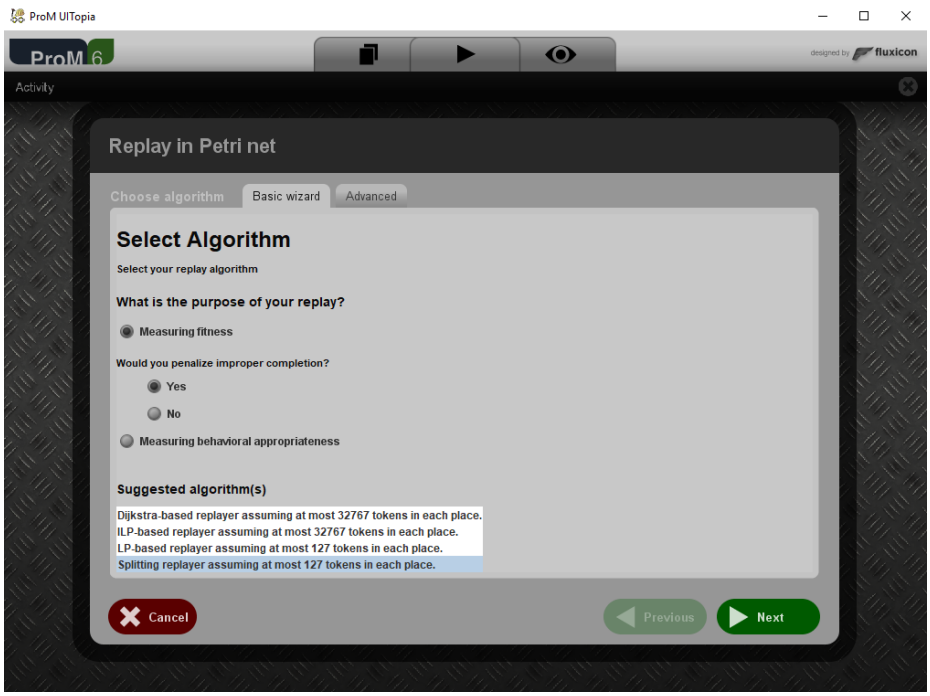

Fig. 3. User interface of ProM showing the algorithm of this paper selected.

Table 2. Java code for standalone execution of the Replayer.

```
public static void doReplay(XLog log, Petrinet net, Marking initialMarking,
                 Marking finalMarking, XEventClasses classes, TransEvClassMapping mapping) {
   // Setup default parameters with 2 threads
   ReplayerParameters parameters = new ReplayerParameters.Default(2, Debug.NONE);
   // Setup the replayer
   Replayer replayer = new Replayer(parameters, net, initialMarking, finalMarking, classes,
                                                                              mapping, false);
   // Set a timeout per trace in milliseconds
   int toms = 10 * 1000;
   // preprocessing time to be added to the statistics if necessary
   long preProcessTimeNanoseconds = 0;
   // Setup a threadpool for the multithreaded execution
   ExecutorService service = Executors.newFixedThreadPool(parameters.nThreads);
   // Setup an array to store the results
   Future<TraceReplayTask>[] futures = new Future[log.size()];
    for (int i = 0; i < log.size(); i++) {
      // Setup the trace replay task
      TraceReplayTask task = new TraceReplayTask(replayer, parameters, log.get(i), i, toms,
                              parameters.maximumNumberOfStates, preProcessTimeNanoseconds);
      // submit for execution
      futures[i] = service.submit(task);
   }
   // initiate shutdown and wait for termination of all submitted tasks and obtain results.
   service.shutdown();
   for (int i = 0; i < log.size(); i++) {
      TraceReplayTask result;
      result = futures[i].get();
   }
   switch (result.getResult()) {
      case DUPLICATE :
         assert false; // cannot happen in this setting
         throw new Exception("Result cannot be a duplicate in per-trace computations.");
      case FAILED :
         // internal error in the construction of synchronous product or other error.
         throw new RuntimeException("Error in alignment computations");
      case SUCCESS :
         // process succcesful execution of the replayer
         SyncReplayResult replayResult = result.getSuccesfulResult();
         // obtain the exit code of the replay algorithm
         int ec = replayResult.getInfo().get(Replayer.TRACEEXITCODE).intValue();
         if ((ec & Utils.OPTIMALALIGNMENT) == Utils.OPTIMALALIGNMENT) {
            // Optimal alignment found.
            // Handle further processing here.
         } else if ((ec & Utils.FAILEDALIGNMENT) == Utils.FAILEDALIGNMENT) {
            // failure in the alignment. Error code shows more details.
            // Handle further processing here.
         }
         // Additional exitcode information for failed alignments:
         if ((ec & Utils.ENABLINGBLOCKEDBYOUTPUT) == Utils.ENABLINGBLOCKEDBYOUTPUT) {}
            // in some marking, there were too many tokens in a place.
         if ((ec & Utils.COSTFUNCTIONOVERFLOW) == Utils.COSTFUNCTIONOVERFLOW) {}
            // in some marking, the cost function went through the upper limit of 2^24
         if ((ec & Utils.HEURISTICFUNCTIONOVERFLOW) == Utils.HEURISTICFUNCTIONOVERFLOW) {}
            // in some marking, the heuristic function went through the upper limit of 2^24
         if ((ec & Utils.TIMEOUTREACHED) == Utils.TIMEOUTREACHED) {}
            // alignment failed with a timeout
         if ((ec & Utils.STATELIMITREACHED) == Utils.STATELIMITREACHED) {}
            // alignment failed due to reacing too many states.}
         if ((ec & Utils.COSTLIMITREACHED) == Utils.COSTLIMITREACHED) {}
            // no optimal alignment found with cost less or equal to the given limit.
         if ((exitCode & Utils.CANCELED) == Utils.CANCELED) {}
            // user-cancelled.
         break;
}  }
```

We discuss the time complexity of the algorithm as well as the properties the various data structures have. As far as we know, this is the first time the actual algorithm and the properties of these data structures have been presented.

References

1. W.M.P. van der Aalst. Decomposing Petri Nets for Process Mining: A Generic Approach. *Distributed and Parallel Databases*, 31(4):471–507, 2013.
2. W.M.P. van der Aalst, A.Adriansyah, and B.F. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.
3. A.Adriansyah. *Aligning Observed and Modeled Behavior*. PhD thesis, Eindhoven University of Technology, Department of Computer Science, 2014.
4. M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology — EUROCRYPT '97*, pages 163–192, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
5. M. Berkelaar, K. Eikland, and P. Notebaert. *lpsolve : Open source (Mixed-Integer) Linear Programming system.*
6. M.de Leoni and A.Marrella. Aligning real process executions and prescriptive process models through automated planning. *Expert Systems with Applications*, 82:162 – 183, 2017.
7. D. Eastlake, G. Fowler, K.-P. Vo, and L. Noll. The fnv non-cryptographic hash algorithm. 2015.
8. P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.
9. M.T.Gómez López, D. Borrego, J. Carmona, and R.M. Gasca. Computing alignments with constraint programming: The acyclic case. In *Proceedings of ATAED 2016, Torun, Poland, June 20-21, 2016.*, volume 1592 of *CEUR Workshop Proceedings*, pages 96–110. CEUR-WS.org, 2016.
10. J. Munoz-Gama, J. Carmona, and W.M.P. van der Aalst. Single-Entry Single-Exit Decomposed Conformance Checking. *Inf. Syst.*, 46:102–122, 2014.
11. D. Reißner, R. Conforti, M. Dumas, M. La Rosa, and A. Armas-Cervantes. Scalable conformance checking of business processes. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, pages 607–627, Cham, 2017. Springer.
12. A.Rozinat and Wil M.P. van der Aalst. Conformance Checking of Processes Based on Monitoring Real Behavior. *Inf. Syst.*, 33(1):64–95, 2008.
13. F.Taymouri and J.Carmona. A Recursive Paradigm for Aligning Observed Behavior of Large Structured Process Models. In Proceedings of *BPM 2016*, Rio de Janeiro, Brazil, September 18-22, 2016, volume 9850 of *LNCS*, pages 197–214. Springer, 2016.
14. B.F. van Dongen, A.K.A. de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The prom framework: A new era in process mining tool support. In *Proceedings of ATPN 2005*, pages 444–454, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
15. B.F. van Dongen. Efficiently computing alignments using the extended marking equation. In *Accepted for BPM 2018, Sydney Australia*, 2018.
16. S.J. van Zelst, A.Bolt, and B.F. van Dongen. Tuning Alingment Computation: An Experimental Evaluation. In *Proceedings of ATAED 2017, Zaragoza, Spain, June 25-30, 2017.*, pages 1–15, 2017.