String Comparators Based Algorithms for Process Model Matchmaking[1]

Yacine Belhoul, Mohammed Haddad, Eric Duchêne and Hamamache Kheddouci
Lab. GAMA, Université Claude Bernard Lyon 1, Université de Lyon, 69622, Villeurbanne, France,
yacine.belhoul@etu.univ-lyon1.fr, {mohammed.haddad, eric.duchene, hamamache.kheddouci}@univ-lyon1.fr

***Abstract*—Retrieving matchings between process models becomes a significant challenge in many applications. Recent attempts have been done to measure similarity of process models based on graph-edit distance. This problem is known to be difficult and computational complexity of exact algorithms for graph matching is exponential. Thus, heuristics must be proposed to obtain approximations. In this paper, we propose an approach to find relevant process models based on their decomposition into paths of possible execution sequences. Then, we propose a schema to compute the similarity between two process models using the proposed decomposition. Moreover, we give particular attention to the problem of ranking a collection of process models according to a particular query.**

***Keywords*-process model matchmaking; process model retrieval; string comparators metrics;**

I. Introduction

Process-Aware Information Systems [1] have recently become an important research area since business world dynamics is continuously increasing making the companies confronting frequent changes in their business environment [2]. Enterprises show the need of continuously re-engineering their Business Processes. Hence, their performance in markets is closely related to their business processes optimization, flexibility and ability to be upgraded. The increasing importance of Process-Aware Information Systems and Service-Oriented Architectures leaded the companies to their own process model repositories. Thus, computing similarity between process models is a critical task that should be completed to manage these repositories. Comparing two processes mainly consists in determining differences between two process models, generally a reference model and the enterprise model. Furthermore, ranking a set of process models following their similarity with the user process model helps administrators to manage collections of process models or several versions of same models.

Inexact, error-tolerant or error-correcting graph-matching is a challenging problem in the manipulation of relational structures that arises in many areas of machine intelligence [3]. Determining the similarity between different graph structures is one of the important issues that relates to the problem. This task is frequently defined as the problem of computing the graph edit distance. Some recent proposals for process matching [4], [5] showed the effectiveness of using graph-based modeling of the problem of process discovery and ranking. However, the graph matching algorithms that underly the propositions induce high computational complexity which reduces their application in practice.

The principle of edit distance computation is to identify a set of basic edit operations on nodes and edges of a graph, and to associate costs to these operations [6]. The edit distance is found by determining the sequence of edit operations that will make the two graphs isomorphic, with minimum cost. The set of edit operations is given by insertions, deletions, and relabeling of both nodes and edges. Unfortunately, computing the exact value of the edit distance has been proved to be a difficult task by Wang *et al.* [7]. To make the algorithm tractable, heuristics that reduce the complexity of the algorithm while keeping an acceptable precision level are needed. Bunke *et al.* showed the intimate relationship between the size of the maximum common subgraph and edit distance [8]. In particular, they demonstrated that problems of finding the maximum common subgraph and computing the graph edit distance are equivalent. This is an important observation since the problems may be related to find efficient approximations each other.

Indeed, our work focus on decomposing the process model into its paths of possible execution sequences based on the process model graph. We propose a generic mechanism using string comparator metrics to compute a distance between a user query and a given process model based on their possible execution sequences. Our contributions include: (i) An algorithm to extract possible execution sequences of a process model. (ii) A mechanism to compute distance or similarity between two process model graphs based on their possible execution sequences. (iii) A new similarity metric based on Jaro similarity to improve its performance. (iv) A solution to rank a collection of process models according to a given query.

The rest of this paper is organized as follows. After some preliminaries given in Section II, we review related work in Section III. The proposed approach is presented in Section IV. Section V describe our experimental evaluation. We conclude our paper in Section VI.

II. Preliminaries

In this section, we introduce some concepts that are used to define a process model. After, we present pertinent metrics used in the literature to compare strings.

[1]This work has received support from the National Agency for Research on the reference ANR- 08-CORD- 009

A. Model

The process model used to represent a given service is based on the one proposed in [4] that covers the core features of practical languages, such as BPMN, EPC, and OWL-S.

A process model is defined by a directed attributed graph $G = (V, A)$, where V is a set of nodes and A is a set of arcs that defines the precedence of the sequence relations between nodes. We distinguish between two kinds of nodes: *activities* and *connectors*.

An activity node represents an atomic task. Connector nodes represent control flow constraints. Two kinds of rules are defined: split and join rules. For each kind of rule, two kinds of operators exist: XOR or AND. A XOR-Split represents a choice between one of several alternative branches which are merged by a corresponding XOR-Join. An AND-Split connector triggers its outgoing concurrent branches which are synchronized by a corresponding AND-Join connector. A Split connector must have at least two outgoing arcs and, symmetrically, Join connectors must have at least two incoming arcs. The model imposes that for each Split connector, there will be exactly one corresponding Join connector. Observe that loop operations are induced by arcs *i.e.* if an arc links two nodes between which there already exists a path, then a loop is formed.

B. String comparator metrics

We will use standard notations and definitions of combinatorics on words or strings, equally: given a finite alphabet $\mathcal{A}$, we will denote $\mathcal{A}^*$ the set of all words on $\mathcal{A}$. The length of a word $w \in \mathcal{A}^*$ will be denoted by $|w|$. A word w is called a *factor* (resp. a *prefix*) of another word u if there exist two words x, y such that $u = xwy$ (resp. $u = wy$). A word x is said to be a *subword* of y if there exist two words $u_1, \ldots, u_n$ and $v_0, \ldots, v_n$ such that $x = u_1 \ldots u_n$ and $y = v_0 u_1 v_1 \ldots u_n v_n$. In other words, x is a subword of y if he is obtained from y by deleting some of its factors. Given two words x and y, a *longest common subword* is a word z of maximal size which is both a subword of x and y. It will be denoted $lcs(x, y)$. For more details, see [9].

1) Hamming distance d_H: This well-known distance is applied on two words of the same size. Roughly speaking, the Hamming distance between two words u and v is the number of differences between the letters (characters) of u and v taken at the same positions. More formally, if $u = u_1 \ldots u_n$ and $v = v_1 \ldots v_n$, where u_i and v_i are letters of the alphabet, then $d_H(u, v)$ is the number of indices i such that $u_i \neq v_i$. Note that in case where the two words have different sizes, this distance can be extended by setting $d_H(u, 0) = |u|$. (e.g., $d_H(aabbba, ababbb) = 3$).

2) Subword distance d_S: Given two words u and v of arbitrary sizes n and m, the subword distance $d_S(u, v)$ is defined as follows:

$$d_S(u, v) = n + m - 2|lcs(u, v)|$$

The subword distance is the smallest number of insertions and deletions needed to transform u into v. Note that in the above formula, the computation of the longest common subword of u and v can be done recursively in time $\mathcal{O}(|u||v|)$. Yet, in the case where we search the subword distance between k words, the problem is NP-hard [10].

3) Levenshtein distance d_L: Levenshtein distance [11] is certainly the most frequently used metric for string comparison. In the literature, it is also called *edit distance*. It can be seen as a mix of the two previous metrics, since it corresponds to the number of edit operations (insertions, deletions and substitution) needed to transform u into v. Several surveys [12], [13] summarize the interest of the Levenshtein distance in various domains.

This distance can be computed according to the following recursive property: given two words $u = (u_1, \ldots, u_n)$ and $v = (v_1 \ldots v_m)$, with prefixes $U_{1,\ldots n}$ (resp. $V_{1,\ldots m}$), $d_L(U_i, V_j)$ is defined as follows :

$$\begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ min(d_L(U_{i-1}, V_{j-1}), d_L(U_{i-1}, V_j) + 1, & \\ \quad d_L(U_i, V_{j-1}) + 1) & \text{if } u_i = v_j \\ min(d_L(U_{i-1}, V_{j-1}) + 1, d_L(U_{i-1}, V_j) + 1, & \\ \quad d_L(U_i, V_{j-1}) + 1) & \text{if } u_i \neq v_j \end{cases}$$

The above formula assumes that the edit costs equal 1. It can straightforwardly be adapted for any other values. A simple dynamic program derived from it yields a $\mathcal{O}(nm)$ algorithm.

4) The Jaro similarity d_J: The Jaro similarity was introduced in 1989 [14]. Since then, it is considered as one of the most accurate metric when dealing with short words. Compared to the Levenshtein distance, it integrates the notion of common characters between two words. Its value is set in the range $[0, 1]$, the value 1 corresponding to a perfect similarity.

Its computation integrates the concept of *common characters*: given two words $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_m)$, we say that u_i and v_j are common if $u_i = v_j$ and $j - i \leq max(n, m)/2 - 1$ (roughly speaking, it means that there are not too far from each other). We will denote by $cc(u, v)$ the number of common characters between u and v. Let $u' = (u'_1, \ldots, u'_k)$, be the characters in u which are common with v (in the same order they appear in u) and let $v' = (v'_1, \ldots, v'_l)$, be the characters in v which are common with u. The transposition number $t(u, v)$ is half the Hamming distance between u' and v', so $t(u, v) = d_H(u', v')/2$. According to these notations, the Jaro similarity is defined as follows:

$$d_j(u, v) = \frac{1}{3}\left(\frac{cc(u, v)}{n} + \frac{cc(u, v)}{m} + \frac{cc(u, v) - t(u, v)}{cc(u, v)}\right)$$

III. Related work

Several studies have been proposed for the problem of determining the similarity of process models [15], [16], [5], [17], [18].

In [19], workflows are modeled as automata and the authors proposed a metric for measuring the similarity. The similarity computation problem is reformulated as automata intersection problem similarly to the maximum common subgraph. Execution time by target process size which is expressed as finding shared traces between the two automata representing these processes. Trace equivalence is often used to determine if two process models are similar or not [20]. In addition, the concept of bisimulation [21], [22] extends trace equivalence by considering stronger constraints. In [23], authors propose a similarity metric that integrates the occurrence probability of shared traces between two processes. Another work based on traces [24], assigns weights to every trace based on logs of execution logs. This is done to distinguish between traces according to their importance.

Work proposed in [25] defines a measure that evaluates the similarity of state-charts by combining the similarities of state labels and state depths. The behavioral similarity of these states is obtained by comparing the vicinities of states. In [16], authors proposed a representation of process models based on finite set of structural relationships between the activities: sequence, conditional, parallel branching, *etc.*. The distance between two processes is then defined as the number of shared relations between them. The edit distance [26] is also used to measure the difference between traces [24], [27]. However, some asymmetry issues are still present. To deal with that, some recent work [5] proposes algorithms to measure the similarity of process models based on graph-edit distance. Other work considering graph techniques based on decomposition or summarization were proposed in [28], [29].

In the present work, we propose a method based on string comparator metrics to approximate the distance between process models. Our approach is based on the decomposition of process models into their expected execution sequences.

IV. Our approach

We describe in this section our approach for process models matchmaking. Let A and B be two process models. We first want to compute the distance between A and B. We can achieve this using a graph matching algorithm [30]. However this problem is NP-complete and the computational complexity of such algorithm is exponential in the size of the two process models. To overcome this problem we propose to decompose the process graph into its paths of possible execution sequences.

In this paper, we consider two specifics problems. The first is the problem of computing the distance between two process models. The second deals with ranking a collection of process models according to a given query.

In the following, we first give the details of our decomposition of a process model, then we present our solutions for the above cited problems.

A. Execution sequences decomposition

We present here how to decompose a process model into its paths of possible execution sequences (ESs). To achieve this, we develop an algorithm to extract possible execution sequences from *START* node to *END* node of a process model represented by a graph G according to behavior definition of each node. For example, an *XOR-Split* means that only one activity among children activities is executed. However, an *AND-Split* means that all children activities are executed in any order. Two examples of execution sequences of two process models are given in Figure 1.

The parameters of the algorithm are from which node the exploration begins ($fromNode$) and to which node ends ($toNode$). The algorithm requires also the current list of ESs ($currentESs$) to continue from this list. The pseudocode of the algorithm is given by Algorithm 1 ($ESsAlgorithm$). The algorithm is recurrently called according to the type of $fromNode$. If it is a $START$, XOR_Join or AND_Join node, the algorithm is called to explore from the next child activity (line 5). In case of $XOR - Split$, the algorithm is called from each child of $fromNode$ until the end of this XOR (given by *GetEND* function) by using the current list of ESs (line 13). All returned lists of ESs are added to the list of ESs of this XOR ($xorESs$). To continue the exploration, the algorithm is called from the end of XOR to $toNode$ using $xorESs$ (line 16). In case of $AND - Split$, the algorithm is called from each child of $fromNode$ until the end of this AND by using an empty list of current ESs (line 23). All possible concatenations between returned values are done by $ConcatLists$ function to obtain the list of ESs of this AND ($andESs$). After, the algorithm is called from the end of this AND to $toNode$ using $andESs$ (line 26). It remains to treat the case of ordinary activity. In this case, the value of $fromNode$ is added to all ESs of $currentESs$ list and the algorithm is called from the next child activity of $fromNode$ to $toNode$ using $currentESs$ list (line 30). Notice that to avoid infinite executions, we limit to k the number of execution of loops. The value of k is set to 1 to reduce the computational complexity and can be increased to improve performances of our solution (see the example in Figure 1). We show in this paper that by setting k to 1, our solution achieves good performance.

B. Comparing two process models

We propose here a solution to compute the distance between two process models A and B. First, the two process models are decomposed into their possible ESs using Algorithm 1. Then, we construct a bipartite graph where the two partitions of nodes are respectively the set of ESs of A and the set of ESs of B. Consider the example illustrated in

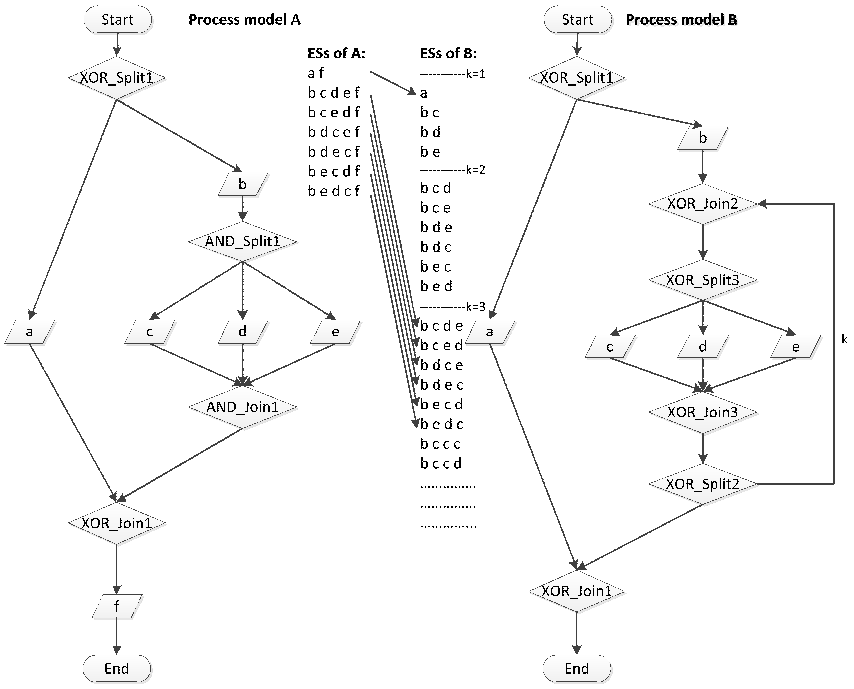


Figure 1. Example of how our solution proceeds. Each ES of A is related by an arrow to the ES of B that matches it better.

Algorithm 1 Execution sequences algorithm: *ESsAlgorithm*

Input: A graph G, from which node the exploration begins ($fromNode$) and to which node ends ($toNode$), and the current list of ESs ($currentESs$)
Output: List of possible ESs from $fromNode$ to $toNode$

1: *fromNode.NBExploration ++ //Limit the number of executions of a loop to k*
2: **if** ($fromNode = toNode$) **then**
3: **return** $currentESs$
4: **end if**
5: **if** ($fromNode \in \{START, XOR\_Join, AND\_Join\}$) **then**
6: **return** ESsAlgorithm(G,$fromNode.child(1)$,$toNode$,$currentESs$) *//To continue the exploration from the next child*
7: **end if**
8: **if** ($fromNode = XOR\_Split$) **then**
9: $xorESs := null$
10: $endXOR$:=*GetEND(XOR_Split)*
11: **for** ($i := 0; i < fromNode.NBChildren; i++$) **do**
12: **if** ($fromNode.child(i).NBExploration < k$) **then**
13: $xorESs$.addAll(ESsAlgorithm(G, $fromNode.child(i)$, $endXOR$, $currentESs$))
14: **end if**
15: **end for**
16: **return** ESsAlgorithm(G, $endXOR$, $toNode$, $xorESs$)
17: **end if**
18: **if** ($fromNode = AND\_Split$) **then**
19: $andESs := currentESs$
20: $endAND$:=*GetEND(AND_Split)*
21: **for** ($i := 0; i < fromNode.NBChildren; i++$) **do**
22: **if** ($fromNode.child(i).NBExploration < k$) **then**
23: $andESs$:= ConcatLists($andESs$, ESsAlgorithm(G, $fromNode.child(i)$, $endAND$, $null$))
24: **end if**
25: **end for**
26: **return** ESsAlgorithm(G, $endAND$, $toNode$, $andESs$)
27: **end if**
28: **if** ($fromNode$ is an ordinary activity) **then**
29: AddList($currentESs$,$fromNode$)
30: **return** ESsAlgorithm(G,$fromNode.child(1)$,$toNode$,$currentESs$)
31: **end if**
32: **end**

Figure 2. We compute the distances (Levenshtein distance is used in this example) between every node of ESs_A and every node of ESs_B (the bipartite graph is complete). Note that if the two partitions do not have the same size, we complete the smaller one with empty ESs. One the complete bipartite graph is obtained, we compute the distance between A and B as the sum of distances on the set of edges given by the Minimal Weight Bipartite Maximum Matching (MWBMM). In the example given by Figure 2, the distance between A and B is equal to 3.

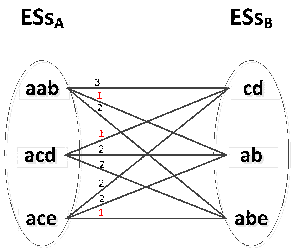


Figure 2. Solution for computing the distance between two process models. The weight of each edge connecting an ES of A and an ES of B is the distance between them.

This solution can be easily modified to compute the similarity between A and B. In this case, We compute the similarities between every node of ESs_A and every node of ESs_B. The similarity between A and B is the sum of similarities on the set of edges given by the maximal weight bipartite maximum matching. Notice that the computational complexity of the algorithm that finds the the

maximal or minimal weight bipartite maximum matching is $\mathcal{O}((\max(n_A, n_B))^3)$ [31] where $n_A = |ESs_A|$ and $n_B = |ESs_B|$.

C. Ranking process models

In this problem, we are interested to rank a collection of process models according to a given query. In this case, we propose a schema to avoid finding the MWBMM which induces a cubic computational complexity. Our schema matches each ES of A with the ES of B whose it has the minimum distance (see Figure 1). The computational complexity of our schema is $\mathcal{O}(n_A \times n_B)$.

Our solution is composed of three steps (see Figure 3). In the first, each process model of the collection is loaded and transformed to a graph representation. To avoid using an exponential algorithm [30] to compute the distance between the entire graphs of A and B, we propose to decompose the two graphs into their paths of possible ESs, then we propose a method to compute a distance between the two graphs using theirs ESs. Thus, in the second step of our solution, we extract possible execution sequences of the process model using Algorithm 1. In the last step, we search for each ES of a process model A, the ES of the process B that matches it better (an example is given in Figure 1). Then, we propose a generic mechanism to compute the distance or similarity between the two process models using their ESs (more details are given in Section IV-C1). We considered four pertinents string comparators metrics described in Section II-B. We also propose a new comparator metric by modifying Jaro similarity to improve its performances. We compare obtained results of existing string comparator metrics to the results of the proposed similarity. Each comparator metric gives a value of the distance or similarity between the query and the current process model. This process is repeated for the user query and each process model of the collection. At the end of this step, a rank of process models that match better the query is given for each considered metric. We can see that choosing the first process model in a rank gives the process model that matches better the given query. However there is different ranks given by each considered metric. To evaluate performances of each solution, we compare obtained results with an expert user results.

1) Distance based rank algorithm: We describe in this section our algorithm of computation of the distance between a given user query A and a process model B of the collection. After that we show how to modify the algorithm if a similarity is considered. First, the algorithm finds for each ES of A, the ES of B that matches it better. To achieve this, we use one of the string comparator metrics. Let D_{ij} be a distance between the i^{th} ES of query A and j^{th} ES of the process model B. We define, D_{iB} to be the distance between the i^{th} ES of A and all ESs of B. Since we are searching the ES of B which matches better the i^{th} ES of A, we consider D_{iB} the minimum of all distances between the i^{th} ES of query A and all ESs of B (An example is given in Figure 1). Thus, $D_{iB} = \min\{D_{ij} : j \in B\}$. We define D_{AB} to be the mean distance between ESs of A and ESs of B. Hence, $D_{AB} = \frac{1}{size(A)} \sum_{i \in A} \frac{D_{iB}}{MaxD_{iB}}$, where $size(A)$ is the number of ESs of the query A and $MaxD_{iB}$ is the maximal value that can take D_{iB}. If Levenstein or Hamming distance is considered, returned D_{iB} is always less than $Max(size(A(i)), size(B(j)))$, where $size(A(i))$ is the number of activities of the i^{th} ES of A. Thus $MaxD_{iB} = Max(size(A(i)), size(B(j)))$. This computation of D_{AB} yields a value in [0,1] and gives more weight to a process model that matches better long-size ESs of the query. However, the weight can be other practical considerations as frequencies of ESs, the priority of the process model, ...etc. This process is repeated for each process model of the collection. At the end of the algorithm, we sort process models according to increasing distances D_{AB} to obtain a rank of process models that match better the query A. The pseudocode of our algorithm is given by Algorithm 2. Our algorithm is generic and can be used for all string distances proposed in the literature. We have only to define $MaxD_{iB}$ which is the maximum value that can take D_{iB} according to each distance. In case of subword distance, $MaxD_{iB} = size(A(i)) + size(B(j))$.

Algorithm 2 Distance based rank algorithm

Input: *a query A, a collection of process models*
Output: *a rank of the process models that match better A*
1: **while** there is a process model in the collection **do**
2: *B := the current process model*
3: $D_{AB} := 0$ *//to initialize the distance to 0*
4: **for** each ES i of query A **do**
5: $D_{iB} := MaxD_{iB}$ *//to initialize D_{iB} to the Max value it can take*
6: **for** each ES j of process model B **do**
7: *Compute D_{ij} according to the considered distance*
8: **if** $(D_{ij} < D_{iB})$ **then**
9: $D_{iB} := D_{ij}$ *//to compute the minimum distance*
10: Update $MaxD_{iB}$ *//according to the considered distance*
11: **end if**
12: **end for**
13: $D_{AB} := D_{AB} + \frac{D_{iB}}{MaxD_{iB}}$
14: **end for**
15: $D_{AB} := \frac{D_{AB}}{Size(A)}$ *//to have the mean distance*
16: **end while**
17: *Sort process models according to increasing distances D_{AB}*
18: **end**

The algorithm is modified as follows to compute D_{AB} in case where a similarity is considered.
$D_{AB} = \frac{1}{size(A)} \sum_{i \in A} D_{iB}$, where D_{iB} is the maximum of all similarities between the i^{th} ES of query A and all ESs of B (i.e. $D_{iB} = \max\{D_{ij} : j \in B\}$). Thus, we inverse the inequality in line 8 of Algorithm 2. For any similarity, D_{iB} is in the interval [0,1], thus $MaxD_{iB} = 1$. Notice that the computation of D_{AB} gives always a value between 0 an 1. It remains to sort process models according to decreasing similarities D_{AB} to obtain a rank of process models that match better the query A. The modified algorithm is also generic and can be used for all similarities.

We also propose a new similarity:

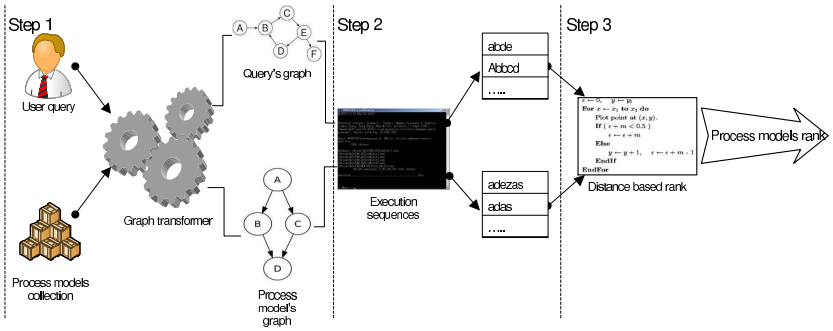


Figure 3. Solution for ranking a collection of process models

2) Proposed similarity: To improve the precision of Jaro similarity, we propose to consider u_i and v_j to be common if $u_i = v_j$ and $|i - j| < min(n, m)/2$. Thus, u_i and v_j are considered to be common if they are nearer. We also propose to consider the opposite direction when comparing u and v since $cc(u, v) \neq cc(v, u)$. Thus we propose a new similarity:
$S_P(u, v) = \frac{1}{4}(\frac{cc(u,v)}{n} + \frac{cc(v,u)}{m} + \frac{cc(u,v)-t(u,v)}{cc(u,v)} + \frac{cc(v,u)-t(u,v)}{cc(v,u)})$.

Notice that $t(v, u)$ is not considered since it is equal to $t(u, v)$. The performances of the proposed similarity are compared in this paper to those of string comparator metrics described in Section II-B.

V. Experimental Evaluation

We present in this section a set of experiments that validate the performance of our solution. All components of the proposed architecture presented in Figure 3 are implemented using the Java programming language and Eclipse SDK version 3.5.2. Evaluating precision of matching algorithms is often impossible. However, the goals of matching depend heavily on user requirements. Thus, we compare obtained results of different considered distances and similarities algorithms with the match result given by an expert user. The used dataset [32] contains 623 semantic process models specified using the graph based representation of process models defined above. We randomly extracted 240 process models from this dataset and considered them as target process models. We have then selected 16 processes from the collection and we considered them as queries. Each query is compared to a collection of process models using proposed algorithms. The query is manually compared by the expert user to the collection of process models to decide which of them match this query.

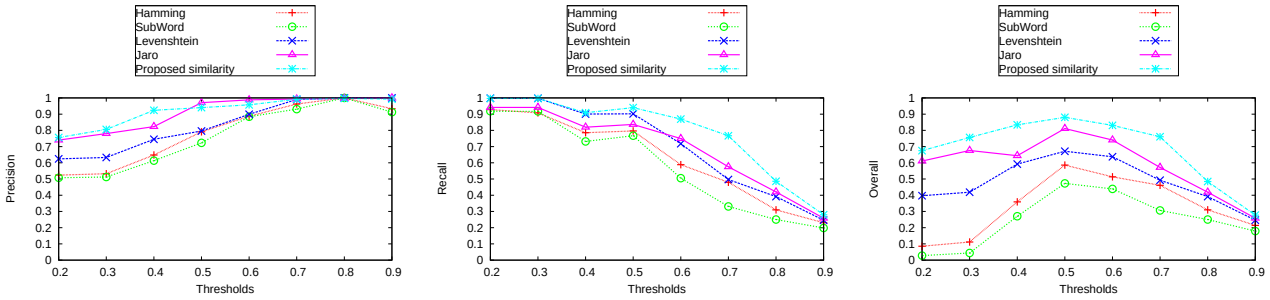
A. Performance Measure

We measured performance of each distance and similarity based algorithm using recall, precision and overall measures. Let Rel be the set of relevant process models given by the expert user and Ret be the set of process models returned by a distance or similarity based algorithm. We use an acceptance threshold to decide that a process model belongs to the Ret set. Thus, a process model is added to Ret if it has a similarity computed with Algorithm 2 (D_{AB}) greater than the threshold. In case where a distance is used, a process model is added to Ret if $\frac{1}{1+D_{AB}}$ is greater than the threshold. An important problem is the choice of this threshold. We compare the different algorithms using the three measures (precision, recall and overall) under different values of the threshold chosen between 0.2 and 0.9. We will then propose an empirical value of the threshold where all the algorithms present the best results according to considered measures.

- Precision is given by : $Precision = \frac{|Rel \cap Ret|}{|Ret|}$. It reflects the share of real correspondences among all returned (found) process models. A solution having high value of precision means that it does not return much false positives.
- Recall is given by : $Recall = \frac{|Rel \cap Ret|}{|Rel|}$. It specifies the share of real correspondences that is returned. A high value of recall of a given solution means that it does not return much false negatives.
- We also consider the overall measure proposed in [33] which is a combined measure of match quality. $Overall = Recall \times (2 - \frac{1}{Precision})$. Notice that overall makes sense only if precision is not less than 0.5. Otherwise, the overall is negative.

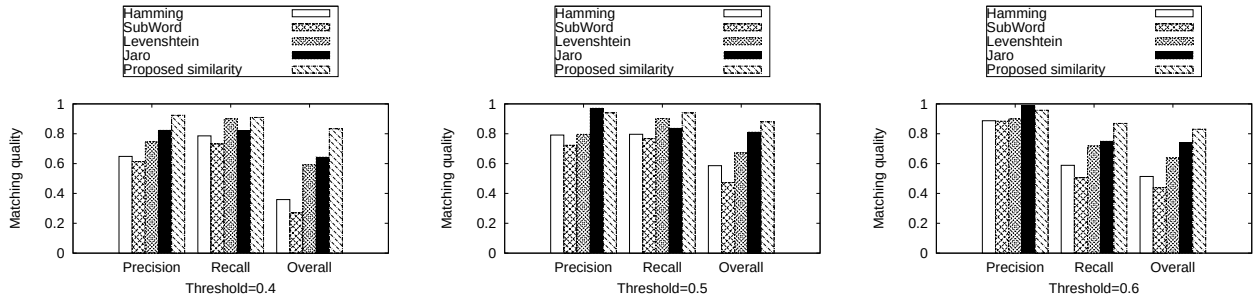
B. Results and interpretation

For performance evaluation, we measured precision, recall and overall for the different algorithms according to different values of the acceptance threshold. As shown in Figure 4(a), precisions of the algorithms increase according to the threshold because high values of thresholds minimize the number of false positives in the Ret set, what gives higher precision. However, recalls of algorithms decrease when the threshold increases (see Figure 4(b)). This is due to the fact that high thresholds reduce the number of true positives in the Ret set, what leads to low recalls. Using a reduced value of threshold leads to low precisions, and using a high threshold leads to



(a) Precision vs. thresholds (b) Recall vs. thresholds (c) Overall vs. thresholds

Figure 4. Performance measures according to different thresholds



(a) Threshold=0.4 (b) Threshold=0.5 (c) Threshold=0.6

Figure 5. Best performance thresholds

low recalls. Figure 4(c) confirms this observation and shows poor performances in terms of overall for low and high values of thresholds. Good performances of algorithms are achieved for the threshold values 0.4, 0.5 and 0.6. To confirm this, we present performances of algorithms according to these thresholds (Figure 5(a) for 0.4, Figure 5(b) for 0.5 and Figure 5(c) for 0.6). We can see that high performances of algorithms especially for overall measure are achieved when considering the 0.5 threshold.

When comparing the different algorithms we observe that lower performances are given by the algorithm using subword distance since it does not take into account the substitution edit operation. Thus, the distance returned by subword distance may be far from the real value. The Hamming distance based algorithm achieves better result than subword since it considers substitution operation (only in the same order). The algorithm using Levenstein distance is better than subword and Hamming algorithms since it considers all edit operations in any possible order. The Jaro similarity based algorithm achieves better results than the above discussed algorithms. This can be explained by the fact that Jaro similarity comparison is based on common activities and the appearance order of activities is not so important in a certain limit of distance between these activities. Since in many cases in process models, the order of activities is not very important (example, rent a car after hotel reservation is similar to hotel reservation after rent a car), the Jaro based algorithms detects this kind of matching and hence presents high performances. Finally, the algorithm based on our similarity gives better results since we consider nearer common activities and we take into account the number of common activities in the opposite direction (i.e. between the current process model and the query). These improvements increase the accuracy of results given by our similarity based algorithm.

VI. Conclusion

In this paper we proposed a solution for process model matchmaking based on comparator metrics of possible execution sequences. We implemented our solution and compared results of the different algorithms. The experimental results show that all algorithms present good performances for the empirical threshold 0.5. The proposed similarity overcome other comparator metrics by achieving high precision, recall and overall.

To improve performances of our solution in real applications, it will be very interesting to consider semantic relationships of activities of the process models. This can be achieved using a domain ontology containing the semantic relationships of activities.

References

[1] M. Dumas, W. van der Aalst, and A. ter Hofstede, "Process-aware information systems: Bridging people and software through process technology," *Wiley and Sons*, 2005.

[2] D. Strong and S. Miller, "Exceptions and exception handling in computerized information processes," *ACM Transactions on Information Systems*, vol. 13, no. 2, pp. 206–233, 1995.

[3] M. Eshera and K. Fu, "An image understanding system using attributed symbolic representation and inexact graph-matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 5, pp. 604–618, 1986.

[4] R. M. Dijkman, M. Dumas, and L. Garcia-Banuelos, "Graph matching algorithms for business process model similarity search," in *In proceedings of BPM'09*, 2009, pp. 48–63.

[5] A. Gater, D. Grigori, and M. Bouzeghoub, "Owl-s process model matchmaking," in *ICWS'10, IEEE Computer Society*, 2010, pp. 640–641.

[6] K. Tai, "The tree-to-tree correction problem," *Journal of the ACM (JACM)*, vol. 26, no. 3, pp. 422–433, 1979.

[7] J. Wang, K. Zhang, and G. Chirn, "The approximate graph matching problem," in *Proceedings of the 12th IAPR International Conference on Computer Vision & Image, Pattern Recognition*, vol. 2. IEEE, 1994, pp. 284–288.

[8] H. Bunke and A. Kandel, "Mean and maximum common subgraph of two graphs," *Pattern Recognition Letters*, vol. 21, no. 2, pp. 163–168, 2000.

[9] M. Lothaire, "Algebraic Combinatorics on Words," *Encyclopedia of Mathematics and its Applications, 90, Cambridge University Press, Cambridge*, 2002.

[10] D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *J. ACM (ACM Press) 25(2)*, pp. 322–336, 1978.

[11] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics-Doklady 10*, pp. 707–710, 1966.

[12] K. Kukich, "Techniques for automatically correcting words in text," *ACM Compute. Surveys, 24*, pp. 377–439, 1992.

[13] E. Ristad and P. Yianilos, "Learning string edit distance," in *Machine learning: Fourteenth international conference*, 1997.

[14] M. A. Jaro, "Advances in record linking methodology as applied to the 1985 census of Tampa Florida," *Journal of the American Statistical Society 64*, pp. 1183–1210.

[15] B. V. Dongen, R. Dijkman, and J. Mendling, "Measuring similarity between business process models," in *Advanced Information Systems Engineering, CAiSE*, 2008.

[16] R. Eshuis and P. Grefen, "Structural matching of bpel processes," in *Fifth European Conference on Web Services*, 2007, pp. 171–180.

[17] a. M. D. R. M. Dijkman, B. F. van Dongen, R. Kaarik, and J. Mendling, "Similarity of business process models: Metrics and evaluation," *Information Systems*, vol. 36, no. 2, pp. 498–516, 2011.

[18] M. Weidlich, R. M. Dijkman, and M. Weske, "Deciding behaviour compatibility of complex correspondences between process models," in *BPM'10, LNCS 6336*, 2010, pp. 78–94.

[19] A. Wombacher, B. Mahleko, P. Fankhauser, and E. Neuhold, "Matchmaking for business processes based on choreographies," in *In Proc. of IEEE International Conference on e-Technology, e-Commerce and e-Service*, 2004.

[20] J.Hidders, M.Dumas, W. M. der Aalst, A. H. ter Hofstede, and J.Verelst, "When are two workflows the same?" in *Australasian Symposium on theory of Computing - Vol 41, ACM vol. 105*, 2005, pp. 3–11.

[21] W. van der Aalst and T. Basten, "Inheritance of workflows: An approach to tackling problems related to change," *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 125–203, 2002.

[22] R. van Glabbeek and W. Weijland, "Branching time and abstraction in bisimulation semantics," *Journal of the ACM*, vol. 43, no. 3, pp. 555–600, 1996.

[23] A. de Medeiros, W. van der Aalst, and A. Weijters, "Quantifying process equivalence based on observed behavior," in *Data Knowledge Engeenering*, 2008, pp. 55–74.

[24] W. van der Aalst, A. A. de Medeiros, and A. Weijters, "Process equivalence: Comparing two process models based on observed behavior," in *BPM'06, LNCS 4102*, 2006.

[25] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and merging of statecharts specifications," in *Proceedings of the 29th international conference on Software Engineering*, 2007.

[26] A. Wombacher and M. Rozie, "Evaluation of workflow similarity measures in service discovery," *Service Oriented Electronic Commerce*, pp. 51–71, 2006.

[27] S. Pinter and M. Golani, "Discovering workflow models from activities' lifespans," *Computers in Industry*, vol. 53, no. 3, pp. 283–296, 2004.

[28] A. Gater, D. Grigori, M. Haddad, M. Bouzeghoub, and H. Kheddouci, "A summary-based approach for enhancing process model matchmaking," in *SOCA 2011*, pp. 1–8.

[29] S. Lagraa, H. Seba, R. Khennoufa, and H. Kheddouci, "A Graph Decomposition Approach to Web Service Matchmaking," in *WEBIST 2011*, pp. 31–40.

[30] K. Riesen and H. Bunke, "Approximate graph edit distance computation by means of bipartite graph matching," *Image and Vision Computing*, vol. 27, pp. 950–959, June 2009.

[31] H. Kuhn, "The hungarian method for the assignement problem," *Naval Research Logistics*, vol. 2, pp. 83–97, 1955.

[32] A. Gater, "Semantic process model benchmark," *Technical report, PRiSM Laboratory*, 2011.

[33] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Proceedings of the 18th International Conference on Data Engineering*, 2002.