

Fast business process similarity search

Zhiqiang Yan · Remco Dijkman · Paul Grefen

Published online: 24 February 2012

© The Author(s) 2012. This article is published with open access at Springerlink.com

Abstract Nowadays, it is common for organizations to maintain collections of hundreds or even thousands of business processes. Techniques exist to search through such a collection, for business process models that are similar to a given query model. However, those techniques compare the query model to each model in the collection in terms of graph structure, which is inefficient and computationally complex. This paper presents an efficient algorithm for similarity search. The algorithm works by efficiently estimating model similarity, based on small characteristic model fragments, called features. The contribution of this paper is threefold. First, it presents three techniques to improve the efficiency of the currently fastest similarity search algorithm. Second, it presents a software architecture and prototype for a similarity search engine. Third, it presents an advanced evaluation of the algorithm. Experiments show that the algorithm in this paper helps to perform similarity search about 10 times faster than the original algorithm.

Keywords Business process · Feature · Similarity · Search

1 Introduction

Nowadays, business process management techniques develop quickly in both academic and industrial fields. To increase the flexibility and controllability of the man-

Communicated by P.K. Chrysanthis.

Z. Yan · R. Dijkman (✉) · P. Grefen
Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: r.m.dijkman@tue.nl

Z. Yan
e-mail: z.yan@tue.nl

P. Grefen
e-mail: p.w.p.j.grefen@tue.nl

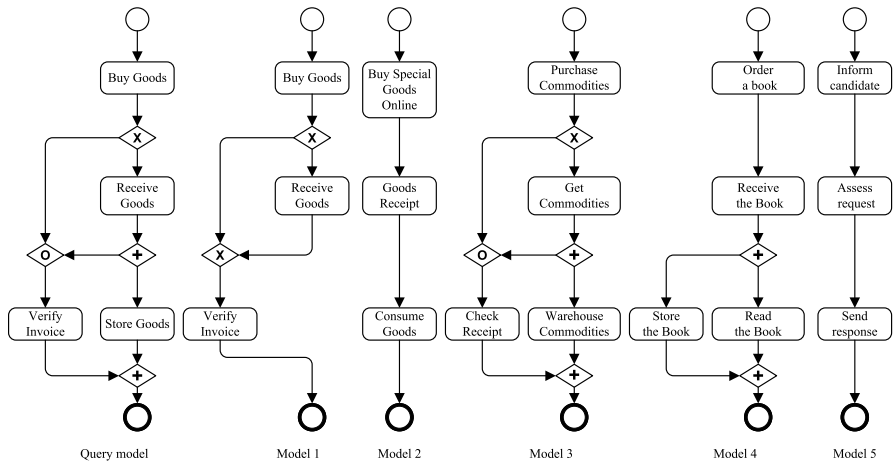


Fig. 1 Searching a collection of business process models

agement of organizations, business processes are used to describe the services that an organization provides and the internal processes that implement those services. As a result, it is common to see collections of hundreds or even thousands of business process models. For example, the collection of SAP reference models consists of more than 600 business process models [6], and the collection of the reference models for Dutch Local Government contains a similar number of models [9]. As business process model collections increase in size, tools and techniques are required to manage them. This includes tools and techniques for quickly searching through a collection, for business process models that meet certain criteria. These criteria can be specified by means of a query language [1, 5], but also by means of (a part of) a query business process model for which similar models must be retrieved [7, 8, 27].

This paper focuses on the second class of search techniques, which are also called *similarity search* techniques. Figure 1 shows an example of business process similarity search. It shows one *query model* and five *process models* in the BPMN notation. Given a query model, a similarity search technique should only return those process models that are similar to the query model and it should return those similar process models in order of their similarity to the query model. In the example, the technique could return models 1, 2 and 3.

There currently exist similarity search techniques [7, 8, 27]. However, these techniques focus on defining a metric to compute the similarity between two process models. To rank the business process models in a collection, the similarity of each of the process models to the query model must be computed. Subsequently, the process models must be ordered according to their similarity. At the same time, business process model collections are increasing in size. For example, Suncorp-Metway Ltd [16] maintains a collection of more than 6000 business process models. Comparing a query model with such amount of models is time consuming and can cause a similarity search operation to take multiple seconds or even minutes, depending on the metric and algorithm that is used, while a query should be performed within

milliseconds by a search engine. (Compare, for example, the response time that you would require of an Internet search engine.)

Therefore, similarity search algorithms must be developed that are both accurate *and* fast. This paper presents such an algorithm. It is developed by extending an existing fast similarity search algorithm [32], by:

1. introducing preprocessing techniques, to reduce the search space that must be processed and, therewith, the number of iterations that must be performed by the algorithm;
2. introducing incremental computation in the algorithm, thus reducing the complexity of each iteration; and
3. introducing prediction techniques to predict which choices in the algorithm will lead to the best result, thus reducing the number of iterations that must be performed to arrive at the best result.

These techniques are presented in Sect. 5. In addition, the paper extends previous work [32] by presenting a detailed software architecture and a prototype for a business process model repository that implements the algorithm (Sect. 7) and an advanced evaluation of the algorithm for the case in which query models are taken from a different collection than the one that is being searched (Sect. 8.2).

The similarity search algorithm works in five steps, as shown in Fig. 2. First, features are identified in the process models that have to be searched. Features are simple but representative abstractions of a business process model, e.g., tasks and task succession. Second, the query model and the process models are compared by looking at the features that they have in common, i.e.: that are similar enough such that we say that they are ‘matched’. For example, suppose that we use tasks and task succession as features in Fig. 1. We can observe that model 1 has six matching features with the query model: the task features ‘Buy Goods’, ‘Receive Goods’ and ‘Verify Invoice’; and the succession features (‘Buy Goods’, ‘Receive Goods’), (‘Buy Goods’, ‘Verify Invoice’) and (‘Receive Goods’, ‘Verify Invoice’). Models 2 and 3 have fewer matches or have weaker similarity with respect to their matches (e.g., ‘Buy Goods’, ‘Buy Special Goods Online’ and ‘Purchase Commodities’ are similar but not identical labels). Models 4 and 5 have an even weaker match or no match at all with respect to the features that they have in common with the query model. Third, an estimation of the similarity of process models to the query model is made, based on the ratio of matching features, and models are classified based on their estimated similarity. For example, depending on the exact metrics that are used, model 1 could be considered as relevant based on matching features, models 2 and 3 considered as potentially relevant, and models 4 and 5 as irrelevant. Fourth, an improved greedy algorithm for determining process similarity is used to efficiently compute similarities between the query model and *potentially relevant* models (e.g., models 2 and 3). Finally, the models in the collection are ranked according to their (estimated) similarity to the query model.

Two experiments were performed to evaluate the algorithm in this paper. The first experiment evaluates the use case in which a model is taken from a collection and subsequently similar models in the same collection are searched. The results of this experiment show that, for this use case, the algorithm helps to retrieve similar models 6.7 times faster than the original algorithm, without impacting the quality of the

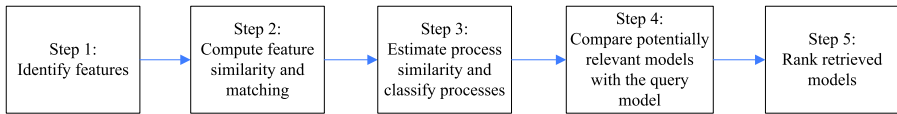


Fig. 2 Steps of the algorithm

results. It helps to retrieve similar models 8.6 times faster if a quality reduction of 1% is acceptable. The second experiment evaluates the use case in which the model that is searched is not from the same collection. It shows that, for this use case, the algorithm helps to retrieve similar models at least 8 times faster with a 4% quality reduction; and 10.7 times faster with a 7% quality reduction.

The rest of the paper is organized as follows. Section 2 defines the concept of feature and presents features that can be used for business process similarity estimation (step 1 in Fig. 2). Section 3 defines metrics for measuring the similarity of features and checking whether features match (step 2 in Fig. 2). Section 4 presents metrics to determine whether a model is relevant, irrelevant or potentially relevant to a query, based on the features that match with features from the query model (step 3 in Fig. 2). Section 5 presents the greedy algorithm for business process similarity search along with indexing techniques and efficiency improvements (step 4 in Fig. 2). Section 6 explains how business processes can be ranked according to their similarity to a given query model (step 5 in Fig. 2). Section 7 presents a software architecture and a prototype for a business process model repository that uses the algorithm in this paper for doing similarity search. Section 8 presents the experiments that were performed to evaluate the properties of the algorithm that was introduced. Section 9 presents related work and Sect. 10 concludes the paper.

2 Business process model features

In this paper features are defined as simple but representative abstractions of business process models. Their simplicity allows similarity computation based on them to be fast and their representativeness ensures that their similarity is strongly related to similarity of the business process models themselves. This makes features very suitable as means to quickly estimate the similarity of business process models. Provided that we choose business process model features carefully, we can further speed up similarity search by building an index of business process models based on those features. In this section, we present the business process model features that we explore in this paper.

Labels can be conveniently used as features, because they are simple strings and therefore qualify as simple abstractions. In addition, indexing mechanisms for strings are well-known, which enables indexing of label features. However, it is harder to use the structure of a business process model as a feature. In fact, considering the structure of a graph when computing the similarity between business process models in our previous work is what makes the problem computationally hard. Therefore,

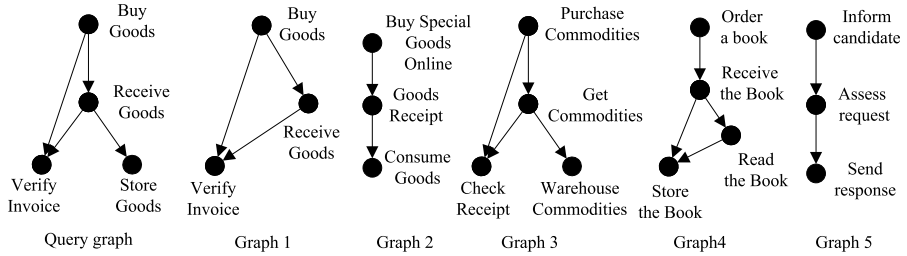


Fig. 3 Business process graphs

we consider the structure of a business process model in terms of the simpler structural features: start, stop, sequence, split, and join. We define these features on the abstraction of a business process graph.

Definition 1 (Business Process Graph, Pre-set, Post-set) Let \mathcal{L} be a set of labels. A business process graph is a tuple (N, E, λ) , in which:

- N is the set of nodes;
- $E \subseteq N \times N$ is the set of edges; and
- $\lambda : N \rightarrow \mathcal{L}$ is a function that maps nodes to labels.

Let $G = (N, E, \lambda)$ be a business process graph and $n \in N$ be a node: $n\bullet = \{m | (m, n) \in E\}$ is the pre-set of n , while $\bullet n = \{m | (n, m) \in E\}$ is the post-set of n .

A business process graph is a graph representation of a business process model. As such, it is an abstraction of a business process model that focuses purely on the structure of that model, while abstracting from other aspects. We define our similarity search techniques on business process graphs to be independent of a specific notation.

Optionally, certain types of nodes can be disregarded in a business process graph. For example, Fig. 3 shows the business process graphs for the models from Fig. 1. Only tasks are considered in these graphs, while events and gateways are disregarded. We disregard events and gateways, because they do not provide (much) additional information when computing model similarity. For example, when two process models have an AND-split, this does not provide us with much information about whether or not these processes are similar. As another example, when an event ‘receive application’ appears in a model, often an activity ‘process application’ appears in the model, thus the event does not provide much additional information. On the other hand, the efficiency of similarity computation is directly related to the number of nodes in the model. Therefore, reducing the number of nodes helps to improve the efficiency. Considering that events and gateways do not provide much additional information when computing similarity, this is a good trade-off.

Based on this the structural features are defined in Definition 2.

Definition 2 (Structural Business Process Model Features) Let $G = (N, E, \lambda)$ be a business process graph.

- A start feature is a node $n \in N$ that has an empty pre-set;

- A stop feature is a node $n \in N$ that has an empty post-set;
- A sequence feature of size s is a list of nodes $[n_1, n_2, n_3, \dots, n_s] \subseteq N$, such that $(n_1, n_2) \in E, (n_2, n_3) \in E, \dots, (n_{s-1}, n_s) \in E$, for $s \geq 2$;
- A split feature of size s is a split node n and a set of nodes $\{n_1, n_2, \dots, n_{s-1}\} \subseteq N$, such that $(n, n_1) \in E, (n, n_2) \in E, \dots, (n, n_{s-1}) \in E$, for $s \geq 3$;
- A join feature of size s is a join node n and a set of nodes $\{n_1, n_2, \dots, n_{s-1}\} \subseteq N$, such that $(n_1, n) \in E, (n_2, n) \in E, \dots, (n_{s-1}, n) \in E$, for $s \geq 3$.

For example, for graph 1 in Fig. 3, the label feature set is {Buy Goods, Receive Goods, Verify Invoice}, the start feature set is {Buy Goods} (using node labels to identify nodes), the stop feature set is {Verify Invoice}, the sequence feature set is {(Buy Goods, Receive Goods), (Buy Goods, Verify Invoice), (Receive Goods, Verify Invoice)}, the split feature set is {(Buy Goods, {Receive Goods, Verify Invoice})}, and the join feature set is {{{Buy Goods, Receive Goods}, Verify Invoice}}.

Many more possible features can be considered in business process models, depending on the business process model aspects that are taken into account (e.g., the organizational aspect or the data aspect), the desired performance of the algorithm (adding more features decreases performance) and the desired quality of the results (adding more features is expected to increase the quality of the search results). In this paper we focus on the most basic process model features. Extensions are possible and are a topic for future work.

3 Feature similarity, matching and indexing

It is possible to use the similarity of the features of two business process models as an estimator of the similarity of the business process models themselves. To this end, metrics must be defined that quantify the similarity of the business process model features. We say that two features that are sufficiently similar are *matching features* and we show how we can determine feature matching based on their similarity. The ratio of matching features will be used in the next section as an estimator of the similarity of business process models. To be able to quickly identify matching features, and therewith similar business process models, feature indices must be defined.

This section first presents metrics to quantify feature similarity. Second, it explains how a feature match can be determined based on feature similarity and, third, it presents feature-based indices.

3.1 Feature similarity

Label feature similarity can be measured in a number of different ways [8, 27]. For illustrative purposes we will use a syntactic similarity metric, which is based on string edit-distance, in this paper. However, in realistic cases more advanced metrics should be used that take synonyms and stemming [8, 27] and, if possible, domain ontologies into account [11]. Label feature similarity is defined as follows in previous work [7, 8].

Definition 3 (Label Feature Similarity) Let $G = (N, E, \lambda)$ be a business process graph and $n, m \in N$ be two nodes and let $|l|$ represent the number of characters in a label l . The string edit distance of the labels $\lambda(n)$ and $\lambda(m)$ of the nodes, denoted $ed(\lambda(n), \lambda(m))$ is the minimal number of atomic string operations needed to transform $\lambda(n)$ into $\lambda(m)$ or vice versa. The atomic string operations are: inserting a character, deleting a character or substituting a character for another. The label feature similarity of $\lambda(n)$ and $\lambda(m)$, denoted $lsim(n, m)$ is:

$$lsim(n, m) = 1.0 - \frac{ed(\lambda(n), \lambda(m))}{\max(|\lambda(n)|, |\lambda(m)|)}$$

For example, the string edit distance between ‘Transportation planning and processing’ and ‘Transporting’ is 26: delete ‘ion planning and process’. Consequently, the label feature similarity is $1.0 - \frac{26}{38} \approx 0.32$. Optional preprocessing steps, such as lower-casing and removing special characters, can improve the results of feature similarity measurements.

The drawback of measuring similarities only by labels is that similar tasks can have different labels. Therefore, it may be hard to determine task similarity solely based on label similarity. For example, in Fig. 3, ‘Buy Special Goods Online’ of Graph 2 and ‘Purchase Commodities’ of Graph 3 are related to ‘Buy Goods’ of Query graph. However, compared with ‘Buy Goods’, ‘Buy Special Goods Online’ is more verbose and ‘Purchase Commodities’ uses synonyms. Therefore, they may not match based on the label similarity. To deal with this situation, we use structural information together with the labels.

We can measure the structural similarity of two nodes, by determining the similarity of the (structural) roles that they have in their business process graphs. We distinguish five different roles that nodes can have: start, stop, regular (sequence), split or join. We do not distinguish the type of splits or joins (e.g., XOR or AND), because we established in previous work [7, 8] that the similarity of the types of two splits or two joins is a bad indication for whether they are similar.

Definition 4 (Role Feature) Let $n \in N$ be a node and $\mathcal{R} = \{\text{start, stop, split, join, regular}\}$ be a set of roles that a node can have. The roles of n are determined by the function $roles: N \rightarrow \mathbb{P}(\mathcal{R})$, such that

$$\begin{aligned} \text{start} \in \text{roles}(n) &\Leftrightarrow |\bullet n| = 0 \\ \text{stop} \in \text{roles}(n) &\Leftrightarrow |n \bullet| = 0 \\ \text{split} \in \text{roles}(n) &\Leftrightarrow |n \bullet| \geq 2 \\ \text{join} \in \text{roles}(n) &\Leftrightarrow |\bullet n| \geq 2 \\ \text{regular} \in \text{roles}(n) &\Leftrightarrow |\bullet n| = 1 \wedge |n \bullet| = 1 \end{aligned}$$

Roles of nodes are considered to be similar or not with respect to the input and output paths of the nodes. The definition of role feature similarity is inspired by string edit-distance, i.e., mainly considering the differences between numbers of input (output) paths of two nodes. Formally, role feature similarity is defined as follows:

Definition 5 (Role Feature Similarity) Let $n, m \in N$ be two nodes. The role feature similarity of these two nodes, denoted $\text{rsim}(n, m)$, is defined as:¹

$$\text{rsim}(n, m) = \begin{cases} 1 & \text{if } \text{start} \in \text{croles} \wedge \text{stop} \in \text{croles} \\ \text{avg}(1 - \frac{\text{abs}(|n\bullet| - |m\bullet|)}{|n\bullet| + |m\bullet|}, 1) & \text{if } \text{start} \in \text{croles} \wedge \text{stop} \notin \text{croles} \\ \text{avg}(1, 1 - \frac{\text{abs}(|\bullet n| - |\bullet m|)}{|\bullet n| + |\bullet m|}) & \text{if } \text{start} \notin \text{croles} \wedge \text{stop} \in \text{croles} \\ \text{avg}(1 - \frac{\text{abs}(|n\bullet| - |m\bullet|)}{|n\bullet| + |m\bullet|}, & \\ 1 - \frac{\text{abs}(|\bullet n| - |\bullet m|)}{|\bullet n| + |\bullet m|}) & \text{otherwise} \end{cases}$$

Where $\text{croles} = \text{roles}(n) \cap \text{roles}(m)$.

This formula covers all possible combinations of roles that nodes can have. For example, the situation in which both nodes are split nodes as well as join nodes is covered by the case ‘otherwise’ ($\text{start} \notin \text{croles} \wedge \text{stop} \notin \text{croles}$). The situation in which both nodes are regular nodes is covered by the same case and leads to a role feature similarity score of 1.

The drawback of measuring role similarity in this way is that it does not discount for the fact that there is a large difference between the frequency of the occurrence of the different role features. Therefore, using the role similarity metric in this way is ineffective. Since, if we give a bonus for matching role features, most nodes would receive that bonus. Therewith, the effect of the bonus would be minimal.

For that reason we refine the role similarity metric to take this effect into account. We do that by not considering features that appear too frequently in the dataset; we say that those features lack ‘discriminative power’.

Definition 6 (Discriminative Role Features) Let \mathcal{N} be the set of nodes of all business process models in a collection. We say that a role feature $r \in \mathcal{R}$ is discriminative, denoted $\text{discriminative}(r)$ if and only if the fraction of the nodes that have the feature is sufficiently small:

$$\frac{|\{n | n \in \mathcal{N}, r \in \text{roles}(n)\}|}{|\mathcal{N}|} \leq \text{dcutoff}$$

Where dcutoff is a cutoff value that determines when the fraction of nodes that have the feature is sufficiently small. This cutoff value is a parameter that can be set as desired, to produce the best results.

In general, a good setting for dcutoff is easy to determine, because there is a large difference between the frequency of features with a low frequency of occurrence and features with a high frequency of occurrence. For example, in the set of business process models that we use for evaluation in this paper, there are 374 nodes in total. Of these nodes, 178 have the ‘stop’ role, 153 have the ‘start’ role, 58 the ‘regular’ role,

¹ avg returns the average value; abs returns the absolute value.

52 the ‘split’ role, 36 the ‘join’ role. Here, we have far more nodes with the ‘start’ and ‘stop’ roles than other nodes. Hence, if we set the *dcutoff* anywhere between 0.16 and 0.40, ‘start’ and ‘stop’ role features are not considered discriminative, while other role features *are* considered discriminative. We incorporate the discriminative power of role features into their similarity using the following formula.

Definition 7 (Role Feature Similarity with Discriminative Power) Let $n, m \in N$ be two nodes. Their role feature similarity with discriminative power, denoted $rdsim(n, m)$, is defined as:

$$rdsim(n, m) = \begin{cases} rsim(n, m) & \text{if } \forall r \in \text{roles}(n) \cap \text{roles}(m) : \text{discriminative}(r) \\ 0 & \text{otherwise} \end{cases}$$

3.2 Feature matching

We say that two features are matched if they are sufficiently similar. What is considered to be sufficient is determined by cutoff parameters that can be set accordingly. If two business process models have sufficiently many matching features, we consider them similar. This is explained in the next section.

We consider two node features to match, if their component features (label features and role features) match. Strong label feature similarity is a strong indication that two nodes are matched, while a combination of role feature similarity and (less strong) label feature similarity is also an indication that two nodes are matched. We distinguish between these two cases when determining a node feature match, such that we can set different thresholds for label similarity in case there is also role similarity and in case there is no role similarity.

Definition 8 (Node Feature Match) Let $n, m \in N$ be two node features with their respective label features and role features. The node features match, if they satisfy one of the following two rules:

- their label features are similar to a high degree, i.e., $lsim(n, m) \geq lcutoff_{high}$;
- their role features are similar, and their label features are similar to a medium degree, i.e., $rdsim(n, m) \geq rcutoff$ and $lsim(n, m) \geq lcutoff_{med}$.

Where $lcutoff_{high}$, $rcutoff$ and $lcutoff_{med}$ are parameters that determine what is considered to be a similar to what degree. The parameters can be set as desired, to produce the best results.

We consider two structural features to match, if their component features (node features) match.

Definition 9 (Structural Feature Match) Two start features with nodes n and m match, if and only if their node features are matched. A stop feature match is defined similarly.

Two sequence features of size s with lists of nodes $Ln = [n_1, n_2, n_3, \dots, n_s]$ and $Lm = [m_1, m_2, m_3, \dots, m_s]$ are matched if and only if for each $1 \leq i \leq s$: the node features of n_i and m_i are matched.

Two split features of size s with split nodes n and m and sets of nodes $Sn = \{n_1, n_2, \dots, n_{s-1}\}$ and $Sm = \{m_1, m_2, \dots, m_{s-1}\}$ are matched if and only if the node features of nodes n and m are matched and there exists a mapping $\text{Map} : Sn \rightarrow Sm$ holds that for each $(sn, sm) \in \text{Map}$: the node features of sn and sm are matched. A join feature match is defined similarly.

Features of different types or sizes are never matched with each other. We can use these two definition to define general feature matching.

Definition 10 (Feature Match) Let f_1 and f_2 be two features. f_1 and f_2 match, denoted $\text{match}(f_1, f_2)$, if and only if they are of the same type and they match according to Definition 8 in case they are node features or Definition 9 in case they are structural features.

3.3 Feature indexing

Node feature matching is mainly based on label similarity and, indirectly, structural feature matching is as well, because it is based on node feature matching. Therefore, if we can find similar labels more efficiently, we can do feature matching more efficiently. We use two indexing techniques to find similar labels more efficiently.

First, we use an M-Tree index [2] on node labels. An M-Tree index is specifically meant for quickly finding items that are similar to a given item to a given degree. In our case, we use it to quickly find nodes with labels that have a similarity (Definition 3) to a given node label that is higher than a specified cutoff ($\text{lcutoff}_{\text{high}}$ or $\text{lcutoff}_{\text{med}}$ in Definition 8).

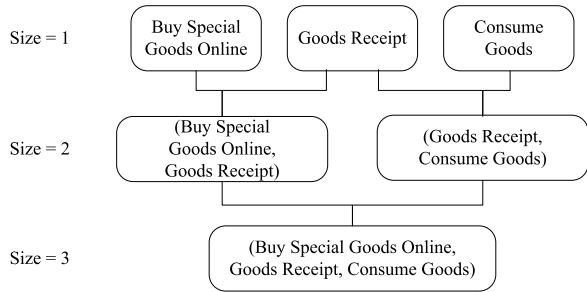
Second, we use an inverted index [20] that maps node labels to nodes, such that, given a node label, we can quickly find the nodes with that label. We use the inverted index, because multiple nodes with the same label may exist in a collection of business process models. For example, in the set of business process models that we use for evaluation in this paper, there are 374 labels, but only 190 distinct ones. The inverted index can prevent comparing identical labels repeatedly.

Furthermore, we can build a ‘parent-child’ index that exploits the fact that features of a larger size (in terms of the number of nodes) are composed of features of a smaller size. For example: sequence features of size 2 are composed node features (of size 1); sequence features of size 3 are composed of sequence features of size 2 and, indirectly, of node features (of size 1). We call the (larger) composed features ‘child’ features and the (smaller) component features ‘parent’ features.

Definition 11 (Parent Feature, Child Feature) If feature A can generate feature B by adding some node(s), feature A is a parent feature of feature B, and feature B is a child feature of feature A. If feature A can generate feature B by adding a single node, feature A is a direct parent feature of feature B, and feature B is a direct child feature of feature A.

Using these properties, we can build a ‘parent-child’ index. Figure 4 illustrates this. Node feature ‘Buy Special Goods Online’ is a direct parent feature of sequence

Fig. 4 Feature index



feature (‘Buy Special Goods Online’, ‘Goods Receipt’), and sequence feature (‘Buy Special Goods Online’, ‘Goods Receipt’) is a direct child feature of node feature ‘Buy Special Goods Online’. The ‘parent-child’ index can help to increase the efficiency of searching, if we consistently start by matching ‘smaller’ features, keeping track of those matches and subsequently, when matching ‘larger’ features, re-using the matches of the ‘smaller’ features that were already identified.

For example, suppose that we need to search for models that are similar to a model that consists of only a single sequence: (‘receive goods’, ‘consume goods’). The model, therefore, contains node features ‘receive goods’ and ‘consume goods’, as well as sequence feature (‘receive goods’, ‘consume goods’). Starting the similarity search in the ‘parent-child’ index, we first find matches (by using an M-Tree lookup) between node features ‘receive goods’ and ‘Goods Receipt’ and between node features ‘consume goods’ and ‘Consume Goods’. We cache these matches. When subsequently looking for a match for the sequence feature (‘receive goods’, ‘consume goods’), we first look at the cached matches of its parent features (which are ‘Goods Receipt’ and ‘Consume Goods’) and their common child features, the sequence (‘Goods Receipt’, ‘Consume Goods’). Subsequently, we only have to establish the match between this sequence and the given sequence.

4 Feature-based similarity estimation

We use the fraction of matching features between two business process models to estimate their similarity, as shown in Definition 12.

Definition 12 (Estimated Business Process Model Similarity) Given a query process graph G_q and another process graph G , with feature sets F_q and F derived from G_q and G . The estimated business process similarity, denoted $ESim(G_q, G)$ is the number of features in G_q or G that are matched by a feature in the other process graph, divided by the number of all features in G_q and G :

$$\frac{|\{f_q \in F_q | \exists f \in F : match(f_q, f)\}| + |\{f \in F | \exists f_q \in F_q : match(f_q, f)\}|}{|F_q| + |F|}$$

Note that we count the number of features in G_q that match a feature in G separately from the number of features in G that match a feature in G_q , because the match

is not necessarily one-to-one. For example, a label feature ‘Fill-out Request Forms’ can match with label features ‘Fill-out Requester’s Detail’ and ‘Fill-out Request Details’ in the other process graph.

Based on the estimated graph similarity, we can classify graphs as relevant, irrelevant or potentially relevant to a query graph. We do that by defining the minimal estimated similarity that a graph must have to the query graph to be considered relevant and the minimal estimated similarity that a graph must have to be considered potentially relevant. We return relevant graphs directly, check the potentially relevant graphs with expensive similarity search algorithms [7, 27], and discard irrelevant graphs.

Definition 13 (Graph Relevance Classification) Given a query process graph G_q and another process graph G , we classify G as:

- relevant to G_q if and only if $\text{ESim}(G_q, G) \geq \text{ratio}_r$
- potentially relevant to G_q if and only if $\text{ratio}_r > \text{ESim}(G_q, G) > \text{ratio}_p$
- irrelevant to G_q if and only if $\text{ratio}_p \geq \text{ESim}(G_q, G)$

Where ratio_r and ratio_p are parameters that determine when a process graph is considered to be relevant, potentially relevant or irrelevant and can be set as desired, to produce the best results.

5 The improved greedy algorithm for process similarity search

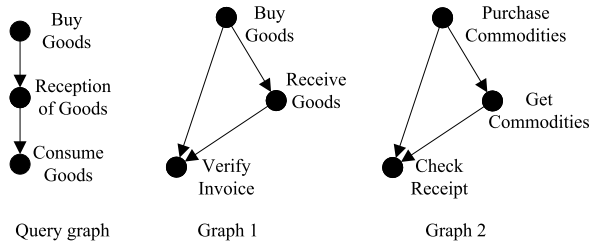
In Sect. 4, we classify process models as relevant, potentially relevant or irrelevant to a given query model. Potentially relevant models still need to be checked by algorithms that can compute exact process similarity. This section explains how to do this. In previous work, a metric is defined to measure process similarity, and algorithms are given to compute the similarity automatically [7]. In this section, we briefly introduce the metric and the currently fastest algorithm to compute the similarity automatically, the greedy algorithm [7]. Then we propose three improvements to the greedy algorithm to further improve its performance.

5.1 The greedy algorithm for process similarity search

The similarity of two business process graphs is defined as a metric based on the graph edit distance is defined, as described in Definition 14.

Definition 14 (Graph Similarity) Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two graphs. The graph edit distance between two graphs is the minimal number of atomic operations needed to transform G_1 into G_2 or vice versa. Atomic operations include inserting, deleting, and substituting nodes and edges. Let $M : N_1 \rightarrow N_2$ be a partial injective mapping that maps N_1 to N_2 . Let $n_1 \in N_1$ be a node in G_1 . n_1 is a substituted node if and only if $\exists n_2 \in N_2, M(n_1) = n_2$, and accordingly n_2 is also a substituted node. A node $n \in N$ is a skipped node if and only if it is not a substituted node. Let $n_{11}, n_{12} \in N_1$ and $(n_{11}, n_{12}) \in E_1$ be two nodes and an edge

Fig. 5 Business process graphs



of G_1 . (n_{11}, n_{12}) is a skipped edge if and only if $\bar{A}(n_{21}, n_{22}) \in E_2$, $M(n_{11}) = n_{21} \wedge M(n_{12}) = n_{22}$. Similarly, we can define the skipped edge in G_2 .

Let *subn*, *skipn* and *skipe* be the sets of substituted nodes, skipped (inserted or deleted) nodes, and skipped (inserted or deleted) edges respectively. The fraction of inserted or deleted nodes, denoted *fskipn*, the fraction of inserted or deleted edges, denoted *fskipe* and the average distance of substituted nodes, denoted *fsubn*, are defined as follows:

$$fskipn = \frac{|skipn|}{|N_1| + |N_2|} \quad fskipe = \frac{|skipe|}{|E_1| + |E_2|}$$

$$fsubn = \frac{2.0 \cdot \sum_{(n,m) \in M} 1.0 - lsim(n, m)}{|subn|}$$

The partial graph similarity induced by the mapping M , denoted as $GSim(G_1, G_2, M)$, is defined as follows:

$$GSim(G_1, G_2, M) = 1.0 - \frac{wskipn \cdot fskipn + wskipe \cdot fskipe + wsubn \cdot fsubn}{wskipn + wskipe + wsubn}$$

where *wsubn*, *wskipn* and *wskipe* are the weights that we assign to substituted nodes, skipped nodes, and skipped edges respectively. These parameters can be set as desired to produce the best results.

The graph similarity of two graphs, denoted as $GSim(G_1, G_2)$, is the maximal possible similarity induced by a mapping between these graphs.

As an example, consider the query graph and graph 1 in Fig. 5. Let the weights $wsubn = 1.0$, $wskipn = 0.5$ and $wskipe = 0.5$. Let the mapping $M = \{(\text{“Buy Goods,” “Buy Goods”}), (\text{“Reception of Goods,” “Receive Goods”})\}$. Then, the partial graph similarity induced by M for the query graph, G_q , and graph 1, G_1 , can be computed based on Definition 14. Note that there are 2 skipped nodes (“Consume Goods” and “Verify Invoice”), 3 skipped edges, and that the label similarity of “Reception of Goods” and “Receive Goods” is 0.62. Consequently, $GSim(G_q, G_1, M) = 1.0 - \frac{0.5 \cdot 0.33 + 0.5 \cdot 0.6 + 1.0 \cdot 0.19}{0.5 + 0.5 + 1.0} \approx 0.68$. This is also the maximal possible graph similarity induced by any mapping and, hence, this is the graph similarity of the two graphs, i.e., $GSim(G_q, G_1) = GSim(G_q, G_1, M)$.

Algorithm 1 describes the greedy algorithm for computing the graph similarity of two process graphs. The algorithm finds the mapping M for which two process graphs have the highest similarity. The algorithm works as follows. Initially, all possible node

pairs are added to *openpairs* (line 3) and no node pair to the mapping *M* (line 4). Then, in each iteration, $\text{GSim}(G_1, G_2, M \cup \{(n, m)\})$ is computed for all $(n, m) \in \text{openpairs}$ to select the pair that increases the partial graph similarity the most (line 7 and 8). That pair is added to the mapping *M* (line 10) and all pairs that contain one of the nodes from that pair are removed from *openpairs* (line 11), such that each node can be mapped at most once. The algorithm ends when there is no node pair in *openpairs* that can increase the graph similarity (line 7 and 8).

Algorithm 1: Greedy Algorithm [7]

```

1 input: two business process graphs  $G_1 = (N_1, E_1, \lambda_1)$ ,  $G_2 = (N_2, E_2, \lambda_2)$ 
2 init
3    $\text{openpairs} \leftarrow N_1 \times N_2$ ;
4    $M \leftarrow \emptyset$ ;
5 begin
6   while
7      $\exists (n, m) \in \text{openpairs}, \text{GSim}(G_1, G_2, M \cup \{(n, m)\}) > \text{GSim}(G_1, G_2, M) \wedge$ 
8      $\nexists (x, y) \in \text{openpairs},$ 
9      $\text{GSim}(G_1, G_2, M \cup \{(x, y)\}) > \text{GSim}(G_1, G_2, M \cup \{(n, m)\})$ 
10    do
11       $M \leftarrow M \cup \{(n, m)\}$ ;
12       $\text{openpairs} \leftarrow \{(x, y) \in \text{openpairs} \mid x \neq n, y \neq m\}$ ;
13    end
14  return  $\text{GSim}(G_1, G_2, M)$ ;

```

We illustrate the algorithm using again the example based on the query graph and graph 1 from Fig. 5. Initially, there are 9 (3 times 3) pairs in *openpairs*. In the first iteration, the pair that increases the similarity most is (“Buy Goods”, “Buy Goods”), because it has the highest label similarity. This pair is added to the mapping *M* and all elements from *openpairs* that contain one of the nodes “Buy Goods” are removed from, such that there are 4 (2 times 2) pairs left. In the second iteration, (“Reception of Goods”, “Receive Goods”) is chosen, since the pair increases the partial graph similarity most. Then, there is only one (one by one) pair left, but it cannot increase the partial graph similarity and the function ends. In the example, the graph similarity is computed 14 times (9 times in the first iteration, 4 times in the second iteration and 1 time in the third and last iteration).

5.2 Improvements

Below, we optimize the algorithm, by reducing the number of times that the graph similarity has to be computed and by reducing the complexity of computing the similarity itself. We present three improvements for Algorithm 1.

5.2.1 Selecting only the top-k similar node pairs

We can reduce the number of times that the graph similarity has to be computed, by initially reducing the number of *openpairs*. In Algorithm 1, *openpairs* is assigned $N_1 \times N_2$ initially (line 3) to include all the possible node mappings between the query graph and the graph in the dataset. However, only pairs with high similarity scores are valuable because those can be expected to increase the similarity the most. The time complexity of the algorithm is directly related to the size of *openpairs*. Consequently, reducing the size of this set has direct impact on the execution time. Therefore, in this section we aim to reduce the size of *openpairs* as follows. For each node in the query graph, we find the top k most similar nodes in the graph and only put these node pairs in the *openpairs*. Definition 15 presents the formal definition of the top k most similar nodes.

Definition 15 (Top-K Most Similar Nodes) Let n be a node, let N be a set of nodes, let k be the number of similar nodes that should be considered for each node, and let sim be a function to compare the similarity of two nodes (e.g., $lsim$). The set that contains the top k similar nodes for the node n , denoted as $TOP(k, n, N, sim)$, is the set that makes the following conditions hold:

- $TOP(k, n, N, sim) \subseteq N$
- $|TOP(k, n, N, sim)| = \min(k, |N|)$
- $\forall p \in TOP(k, n, N, sim), \nexists o \in N / TOP(k, n, N, sim),$ such that $sim(n, o) > sim(n, p)$.

The parameter k can be set as desired to get the best results.

For example, consider the query graph and graph 1 from Fig. 5. Let $k = 1$, let n be the node “Reception of Goods” in Query graph, let N be the node set of graph 1, and let $sim = lsim$. Then, $TOP(1, n, N, lsim) = \{\text{“Receive Goods”}\}$.

There is a drawback to computing the top k node pairs by only using label similarity. The graph edit similarity considers both the label similarity and the structural similarity. However, the Top-K heuristic does not take structural information into account and therefore may not result in the optimal node pair. For example, considering the query graph and graph 2 in Fig. 3, the labels “Buy Goods” and “Purchase Commodities” are related, but their label similarity is only 0.24. Consequently, the pair (“Buy Goods”, “Purchase Commodities”) may not be put into *openpairs* when using the Top-K heuristic, even if it may increase the similarity score later on in the execution of the algorithm.

To partly account for this issue, we can also take into account structural information in the Top-K heuristic. We can do this by comparing the sizes of the pre-sets and post-sets of nodes as defined in Definition 5. Then, we can compute the similarity of two nodes by considering both the label and role similarities, as described in Definition 16.

Definition 16 (Node Similarity) Let $n, m \in N$ be two nodes. The node similarity is a weighted average value of $lsim(n, m)$ and $rsim(n, m)$, i.e.,

$$nsim(n, m) = \frac{w_l \cdot lsim(n, m) + w_r \cdot rsim(n, m)}{w_l + w_r}$$

where w_l and w_r are parameters that can be set as desired to produce the best results.

For example, let $w_l = 1.0$ and $w_r = 0.5$. Considering the nodes “Buy Goods” and “Purchase Commodities” in query graph and graph 2 of Fig. 3, their node similarity is $\frac{1.0 \cdot 0.24 + 0.5 \cdot 0.67}{1.0 + 0.5} \approx 0.38$. Consequently, for $k = 1$ (“Buy Goods”, “Purchase Commodities”) would be put in *openpairs*, while, if we had used label similarity instead of the node similarity, for the node “Buy Goods” in Query graph, the only node pair in the *openpairs* would have been (“Buy Goods”, “Get Commodities”).

5.2.2 Incrementally computing the graph similarity

We can reduce the computation time of the graph similarity, by computing it incrementally instead of anew in each iteration. In Algorithm 1, when we add a new node pair (n, m) into the mapping M , we need to re-compute the partial graph similarity according to the new mapping $M \cup \{(n, m)\}$, i.e., $\text{GSim}(G_1, G_2, M \cup \{(n, m)\})$ (line 7 and 8). However, $\text{GSim}(G_1, G_2, M \cup \{(n, m)\})$ is related to $\text{GSim}(G_1, G_2, M)$, so we should compute it incrementally. Therefore, this section investigates the definition of graph similarity and deduces an incremental way to compute the partial graph similarity.

From Definition 14, we know that the partial graph similarity is related to three fractions, i.e., *fskipn*, *fskipe* and *fsubsn*. These fractions change when M changes. Let us see how these fractions change one by one after putting a node pair to M .

First, no matter which node pairs are in M and which new node pair is put into M , the size of *skipn* always reduces by two, because two nodes are matched and removed from the skipped node set. Thus, we can compute the increment of *fskipn*, as defined in Definition 17.

Definition 17 (Skipped-node Fraction Increment) Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two graphs. Let M be a partial injective mapping that maps N_1 to N_2 . Let (n, m) be a node pair in *openpairs*. After putting (n, m) into M , the increment of *fskipn*, denoted as $\Delta_{|fskipn|}$, is defined as follows:

$$\Delta_{fskipn} = \frac{-2}{|N_1| + |N_2|}$$

Second, the size reduction of *skipe* is related to the new node pair (n, m) and the mapping M . We can compute this by only considering edge pairs that are related to (n, m) , instead of all possible edge pairs. The size reduction is equal to the size of the intersection of $\bullet n \times \bullet m$ and M and the size of the intersection of $n \bullet \times m \bullet$ and M .

Definition 18 (Skipped-edge Increment) Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two graphs. Let M be a partial injective mapping that maps N_1 to N_2 . Let (n, m) be a node pair in *openpairs*. After putting (n, m) into M , the increment of $|skipe|$, denoted as $\Delta_{|skipe|}$, is defined as follows:

$$\Delta_{|skipe|} = -2 \cdot (|\{(x, y) \in M | x \in \bullet n \wedge y \in \bullet m\}| + |\{(x, y) \in M | x \in n \bullet \wedge y \in m \bullet\}|)$$

The increment of *fskipe* is defined in Definition 19.

Definition 19 (Skipped-edge Fraction Increment) Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two graphs. Let M be a partial injective mapping that maps N_1 to N_2 . Let (n, m) be a node pair in openpairs. After putting (n, m) into M , the increment of *fskipe*, denoted as $\Delta_{|fskipe|}$, is defined as follows:

$$\Delta_{fskipe} = \frac{\Delta_{|skipe|}}{|E_1| + |E_2|}$$

Third, contrary to *skipn*, the size of *subn* increases by two after putting (n, m) into M . The increment of *fsubn* also involves the label similarities of (n, m) and pairs in M , as defined in Definition 20.

Definition 20 (Substituted-node Fraction Increment) Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two graphs. Let M be a partial injective mapping that maps N_1 to N_2 . Let (n, m) be a node pair in openpairs. After putting (n, m) into M , the increment of *fsubn*, denoted as Δ_{fsubn} , is defined as follows:

$$\Delta_{fsubn} = \frac{\sum_{(x,y) \in M} lsim(x, y)}{|subn|} - \frac{\sum_{(x,y) \in M \cup \{(n,m)\}} lsim(x, y)}{|subn| + 2}$$

where $|subn| = 2 \cdot |M|$.

From the analysis above, we can derive that after putting (n, m) into M , the graph similarity increment can be computed by performing only two computations, those of $lsim(n, m)$ and $\Delta_{|skipe|}$. The other components of the computation are either constants that can be computed before executing the algorithm, or functions of M that only have to be computed once each time M changes. Proposition 1 shows how the graph similarity increment can be computed as a function of $lsim(n, m)$ and $\Delta_{|skipe|}$, two constants c_1 and c_2 and two functions of M : $\varphi_1(M)$ and $\varphi_2(M)$.

Proposition 1 (Graph Similarity Increment) Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two graphs. Let M be a partial injective mapping that maps N_1 to N_2 . Let (n, m) be a node pair in openpairs. After putting (n, m) into M , the graph similarity increment, denoted as Δ , is defined as follows:

$$\Delta = \varphi_1(M) \cdot \mathbf{lsim}(\mathbf{n}, \mathbf{m}) + c_1 \cdot \mathbf{\Delta}_{|skipe|} + \varphi_2(M) + c_2,$$

where:

$$c_1 = \frac{-wskipe}{(wskipn + wskipe + wsubn) \cdot (|E_1| + |E_2|)},$$

$$c_2 = \frac{2 \cdot wskipn}{(wskipn + wskipe + wsubn) \cdot (|N_1| + |N_2|)},$$

$$\varphi_1(M) = \frac{wsubn}{(wskipn + wskipe + wsubn) \cdot (|M| + 1)}, \quad \text{and}$$

$$\varphi_2(M) = \frac{-w_{\text{subn}} \cdot \sum_{(x,y) \in M} \text{lsim}(x, y)}{(w_{\text{skipn}} + w_{\text{skipe}} + w_{\text{subn}}) \cdot (|M| + 1) \cdot |M|}$$

We prove Proposition 1 as follows.

Proof The graph similarity increment is equal to the graph similarity after adding (n, m) , i.e.: $\Delta = \text{GSim}(G_1, G_2, M \cup \{(n, m)\}) - \text{GSim}(G_1, G_2, M)$. We can rewrite this as follows.

$$\begin{aligned} \Delta &= \text{GSim}(G_1, G_2, M \cup \{(n, m)\}) - \text{GSim}(G_1, G_2, M) \\ &= \left(1.0 - \frac{w_{\text{skipn}} \cdot f_{\text{skipn}}' + w_{\text{skipe}} \cdot f_{\text{skipe}}' + w_{\text{subn}} \cdot f_{\text{subn}}'}{w_{\text{skipn}} + w_{\text{skipe}} + w_{\text{subn}}} \right) \\ &\quad - \left(1.0 - \frac{w_{\text{skipn}} \cdot f_{\text{skipn}} + w_{\text{skipe}} \cdot f_{\text{skipe}} + w_{\text{subn}} \cdot f_{\text{subn}}}{w_{\text{skipn}} + w_{\text{skipe}} + w_{\text{subn}}} \right) \\ &= \frac{-w_{\text{skipn}} \cdot \Delta_{f_{\text{skipn}}} - w_{\text{skipe}} \cdot \Delta_{f_{\text{skipe}}} - w_{\text{subn}} \cdot \Delta_{f_{\text{subn}}}}{w_{\text{skipn}} + w_{\text{skipe}} + w_{\text{subn}}} \\ &= \frac{1}{w_{\text{skipn}} + w_{\text{skipe}} + w_{\text{subn}}} \cdot \left(\frac{2 \cdot w_{\text{skipn}}}{|N_1| + |N_2|} - \frac{w_{\text{skipe}}}{|E_1| + |E_2|} \cdot \Delta_{|\text{skipe}|} \right. \\ &\quad \left. + w_{\text{subn}} \cdot \left(\text{lsim}(n, m) \cdot \frac{1}{|M| + 1} - \frac{\sum_{(x,y) \in M} \text{lsim}(x, y)}{(|M| + 1) \cdot |M|} \right) \right) \\ &= \varphi_1(M) \cdot \text{lsim}(n, m) + c_1 \cdot \Delta_{|\text{skipe}|} + \varphi_2(M) + c_2 \quad \square \end{aligned}$$

From Proposition 1 we can see that only $\text{lsim}(n, m)$ and $\Delta_{|\text{skipe}|}$ are related to the new pair (n, m) . We already know how to compute them as described in Definitions 3 and 18. Therefore, we can compute the graph similarity incrementally. For example, considering query graph and graph 1 from Fig. 5. Let the weights $w_{\text{subn}} = 1.0$, $w_{\text{skipn}} = 0.5$ and $w_{\text{skipe}} = 0.5$. Let the mapping $M = \{(\text{“Buy Goods”}, \text{“Buy Goods”})\}$. Then, after putting $(\text{“Reception of Goods”}, \text{“Receive Goods”})$ into M , the graph similarity increment is $\frac{1}{2.0} \cdot \left(\frac{1.0 \cdot 0.63}{2} - \frac{0.5 \cdot (-2)}{5} - \frac{1.0 \cdot 1.0}{2} + \frac{2.0 \cdot 0.5}{6} \right) \approx 0.1$.

5.2.3 Pre-selecting similar node pairs

We can reduce the number of times the graph similarity must be computed, by ‘predicting’ the pair in the mapping that would increase the similarity the most. In Algorithm 1, to decide which node pair to add into the mapping M next, we need to compute $\text{GSim}(G_1, G_2, M \cup \{(n, m)\})$ for all the (n, m) in *openpairs* and find the maximal value (lines 7 and 8). Proposition 1 discloses that the graph similarity increment is related to two variables: $\text{lsim}(n, m)$ and $\Delta_{|\text{skipe}|}$ only. Consequently, if we can ‘predict’ the value of those two variables, we can predict the value of the overall similarity increase.

This section first proposes an efficient manner to compute the value of $\Delta_{|\text{skipe}|}$. Then, it uses the values of $\text{lsim}(n, m)$ and $\Delta_{|\text{skipe}|}$ to pre-select a few candidate pairs that potentially have the largest similarity increment. Last, it finds the node pair with the maximal graph similarity increment from candidate pairs. By doing this, only

the graph similarity increments of these candidate pairs, instead of all the pairs in *openpairs*, need to be computed and compared to find the pair with the maximal graph similarity increment.

We can efficiently compute the value of $\Delta_{|skipe|}$ as follows. During iterations, given a pair (n, m) , $lsim(n, m)$ is constant in spite of the changes of the mapping M . However, $\Delta_{|skipe|}$ is related to both (n, m) and M . In each iteration, we need to know the $\Delta_{|skipe|}$ values for all the pairs in *openpairs* before the pre-selection. We can compute the $\Delta_{|skipe|}$ values based on Definition 18 in each iteration, but it is time consuming because of the consideration of all pairs in *openpairs*. Instead, we build a cache to store the $\Delta_{|skipe|}$ values for all the pairs in *openpairs*. Initially, all the $\Delta_{|skipe|}$ values are 0, because there is no node pair in M . When a node pair (o, p) is added to M , we only need to update the $\Delta_{|skipe|}$ values for $(n, m) \in openpair$ that makes $n \in \bullet o \wedge m \in \bullet p$ or $n \in o \bullet \wedge m \in p \bullet$ hold. Proposition 2 presents the rule to update the $\Delta_{|skipe|}$ values.

Proposition 2 (Difference of Skipped-edge Increment) *Let $G_1 = (N_1, E_1, \lambda_1)$ and $G_2 = (N_2, E_2, \lambda_2)$ be two process graphs as defined in Definition 1. Let M be a partial injective mapping that maps N_1 to N_2 . Let (o, p) and (n, m) be two node pairs in *openpairs* ($o \neq n \wedge p \neq m$). After putting (o, p) into M , the difference of the $\Delta_{|skipe|}$ value for (n, m) is defined as follows:*

$$-2 \cdot (|\{(x, y) \in \{(o, p)\} | n \in \bullet x \wedge m \in \bullet y\}| + |\{(x, y) \in \{(o, p)\} | n \in x \bullet \wedge m \in y \bullet\}|).$$

We prove Proposition 2 as follows.

Proof Let $M' = M \cup \{(o, p)\}$.

$$\begin{aligned} &\Delta'_{|skipe|} - \Delta_{|skipe|} \\ &= -2 \cdot (|\{(x, y) \in M' | x \in \bullet n \wedge y \in \bullet m\}| + |\{(x, y) \in M' | x \in n \bullet \wedge y \in m \bullet\}|) \\ &\quad - (-2 \cdot (|\{(x, y) \in M | x \in \bullet n \wedge y \in \bullet m\}| + |\{(x, y) \in M | x \in n \bullet \wedge y \in m \bullet\}|)) \\ &= -2 \cdot (|\{(x, y) \in \{(o, p)\} | x \in \bullet n \wedge y \in \bullet m\}| \\ &\quad + |\{(x, y) \in \{(o, p)\} | x \in n \bullet \wedge y \in m \bullet\}|) \\ &= -2 \cdot (|\{(x, y) \in \{(o, p)\} | n \in x \bullet \wedge m \in y \bullet\}| \\ &\quad + |\{(x, y) \in \{(o, p)\} | n \in \bullet x \wedge m \in \bullet y\}|). \quad \square \end{aligned}$$

As an example, consider query graph and graph 1 from Fig. 5. Let the mapping $M = \emptyset$. Then, after putting (“Buy Goods”, “Buy Goods”) into M , only the $\Delta_{|skipe|}$ values for (“Reception of Goods”, “Receive Goods”) and (“Reception of Goods”, “Verify Invoice”) need to be modified to -2 .

By now, we already know the values of $lsim(n, m)$ and $\Delta_{|skipe|}$. Then, let us see how to use them to pre-select candidate pairs. The value range of $\Delta_{|skipe|}$ is typically limited. For example, in the validation dataset of this article, it can only be 0, -2 , -4 , or -6 (the values are always even, because only two edges can match each other

at one time). Therefore, we can first consider $\Delta_{|skipe|}$ and then $lsim(n, m)$. For each possible value of $\Delta_{|skipe|}$, the node pair (n, m) with the maximal $lsim(n, m)$ value is selected as a candidate pair (see Definition 1). To quickly get the pair (n, m) with maximal label similarity, we can sort the node pairs in *openpair* descendingly with respect to their label similarities in advance. We get a few candidate pairs, of which the one with the maximal graph similarity increment is the pair we are looking for.

Next, we can pre-select the potential node pairs to be put into M . For example, query graph and graph 1 in Fig. 3 are considered. When $M = \{(\text{“Buy Goods”}, \text{“Buy Goods”})\}$, the distinct values for $\Delta_{|skipe|}$ are -2 or 0 . The node pairs $(\text{“Reception of Goods”}, \text{“Receive Goods”})$ and $(\text{“Consume Goods”}, \text{“Receive Goods”})$ are selected as candidate pairs respectively for each $\Delta_{|skipe|}$ value. Finally, $(\text{“Reception of Goods”}, \text{“Receive Goods”})$ is put into M , because it provides a higher graph similarity increment.

There are algorithms solving multiple items with the highest overall values, e.g., the well-known threshold algorithm (TA) [13]. These algorithms are also applicable for node pair pre-selection (two items in this case, $\Delta_{|skipe|}$ and $lsim(n, m)$). In this section, we present a simple but effective algorithm for the sake of explanation.

5.3 The improved greedy algorithm for process similarity search

This section integrates the improvements proposed in Sect. 5.2 into the Greedy Algorithm that is presented in Sect. 5.1 and presents the improved algorithm after integration, as shown in Algorithm 2.

Initially, instead of considering all the possible node pairs, only the top k most similar nodes are considered for each query node (see line 10) as explained in Sect. 5.2.1. These node pairs are sorted with respect to their label similarities. The mapping is empty at first (see line 11). Three more variables are defined (see lines 12–14: *result*, *skippedgecache* and *candidatepairs*. *result* is the partial graph similarity for the current mapping M . *skippedgecache* is a list that records the numbers of potentially matched edges for each pair in *openpairlist*. *candidatepairs* is a mapping that, for each possible value in *skippedgecache*, records the node pair with maximal node similarity.

In each iteration, the node pair with the maximal graph similarity increment in *candidatepairs* is added to the mapping; the variables are adapted according to the current state after that. The function ends when there is no more node pair for which the graph similarity increases.

Figure 6 shows an example in which the similarity of the query graph and graph 1 from Fig. 5 are computed according to the improved algorithm. Let $k = 2$, $wskipn = wskipe = w_l = 0.5$, and $wsubn = w_r = 1.0$. In the figure the nodes are identified by the first letters of the words in their labels. The figure shows the values for the variables of the algorithm for three iterations. Initially, the top-2 most similar nodes for each node in the query model are determined. Based on that information, the *openpairlist* is constructed. Initially, none of the pairs in the *openpairlist* will reduce the number of skipped edges. Consequently, the *skippedgecache* contains only 0s. The only pair in *candidatepairs* is $(\text{“Buy Goods”}, \text{“Buy Goods”})$, which is consequently put into M in the first iteration. As a result, the *openpairlist*, *skippedgecache* and *candidatepairs* variables are updated. There are now two distinct values (-2 and 0)

Algorithm 2: Improved Greedy Algorithm

input: two business process graphs $G_1 = (N_1, E_1, \lambda_1)$, $G_2 = (N_2, E_2, \lambda_2)$

```

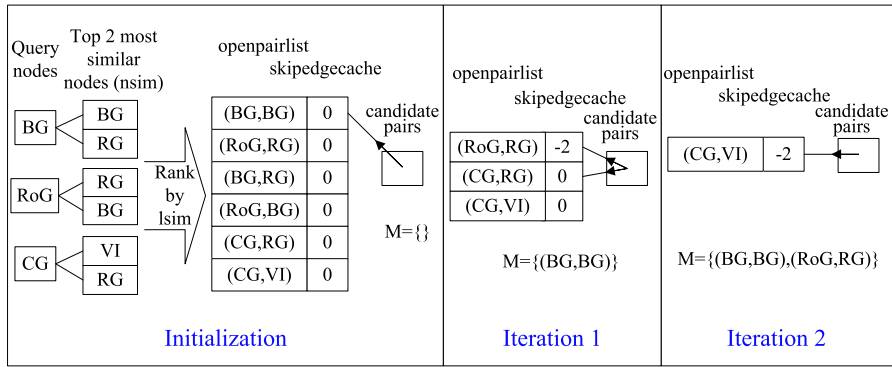
1 function sortedTopkPairs( $N_1, N_2$ );
2 begin
3   foreach  $n_1 \in N_1$  do
4     | select the top  $k$   $n_2 \in N_2$  with respect to  $nsim(n_1, n_2)$ ;
5     | put the top  $k$   $(n_1, n_2)$  pairs in openpairlist;
6   end
7   return sorted openpairlist with respect to  $lsim(n_1, n_2)$  in descending order;
8 end

9 init
10  openpairlist  $\leftarrow$  sortedTopkPairs( $N_1, N_2$ );
11   $M \leftarrow \emptyset$ ;
12   $result \leftarrow 0$ ;
13  skipedgecache  $\leftarrow [0, 0, \dots, 0]$ ;
14  candidatepairs  $\leftarrow [(0, \text{Head}(\text{openpairlist}))]$ ;
15 begin
16  while  $\exists (n, m) \in \text{candidatepairs}, \Delta > 0 \wedge \nexists (x, y) \in \text{candidatepairs}, \Delta' > \Delta$ 
17    do
18    |  $M \leftarrow M \cup \{(n, m)\}$ ;
19    |  $result += \Delta$ ;
20    | foreach  $(x, y) \in \text{openpairlist} \wedge (x = n \vee y = m)$  do
21    | | remove  $(x, y)$  from openpairlist;
22    | | remove the according item for  $(x, y)$  from skipedgecache
23    | end
24    | foreach  $(x, y) \in \text{openpairlist} \wedge ((x \in \bullet n \wedge y \in \bullet m) \vee (x \in n \bullet \wedge y \in m \bullet))$ 
25    | do
26    | | update skipedgecache// $skippedge- = 2$  or  $skippedge- = 4$ 
27    | end
28    | foreach distinct value in skipedgecache do
29    | | update candidatepairs//store maximal  $lsim(n, m)$ 
30    | end
31  end
32  return  $result$ ;
33 end

```

in *skipedgecache*, and two pairs in *candidatepairs*. (“Reception of Goods”, “Receive Goods”) provides higher graph similarity increase and is chosen to be in M in the second iteration. Then, there is only one pair left in *candidatepairs*, but it cannot increase the partial graph similarity and the algorithm ends.

During the iterations, only four partial graph similarities induced by different mappings are computed, and only two partial graph similarities are compared to select the pair with the maximal graph similarity increment. Some additional computation is re-



k=2, wskipn=wskipe=w=0.5, and wsubn=w_r=1.0

BG=Buy Goods, RoG=Reception of Goods, RG=Receive Goods, CG=Consume Goods, and VI=Verify Invoice

Fig. 6 Running example for query graph and graph 1 in Fig. 5

quired in Algorithm 2 as compared to the original algorithm (e.g., ranking node pairs with respect to the label similarity). However, the improved algorithm is much less time consuming, as will be shown in the evaluation results in Sect. 8.

6 Ranking

Using the similarity estimation metric ESim from Sect. 4 and the similarity measurement metric GSim from Sect. 5, we can rank the models in a collection in the order of their similarity to a query model.

Given a query business process model and a set of business process models, we classify the set of business process models as ‘relevant’, ‘potentially relevant’ or ‘irrelevant’, according to Definition 13. We only rank the models in the ‘relevant’ and the ‘potentially relevant’ sets, by first presenting the models in the ‘relevant’ set, in the order of their estimated similarity ESim to the query model, and then presenting the models in the ‘potentially relevant’ set in the order of their similarity GSim to the query model. Ranking models in a set, results in a sequence that is ordered in descending order of similarity score (most similar item first). Sequences can be concatenated to produce a complete search result. Given two sequences L and M , their concatenation, denoted $L ++ M$, is the sequence in which the elements from L are put in front of the elements from M operand. We only consider the potentially relevant models that are sufficiently similar to the query model (i.e. we only consider the models G , for which $GSim(G, G_q) > cutoff$, where $cutoff$ is a parameter). More precisely, the ranking is defined as follows.

Definition 21 (Ranking) Let G_q be a query graph and let G_s be a set of graphs. Furthermore, let $cutoff$ be a parameter that determines the minimum similarity score. The ranking of the graphs from G_s according to their similarity to G_q is a mathematical sequence $Gr ++ Gp$, where:

- Gr is the sequence that consists of all models from G_s that are relevant to G_q , such that for each Gr_i, Gr_j from Gr holds: if $i < j$ then $\text{ESim}(Gr_i, G_q) \geq \text{ESim}(Gr_j, G_q)$; and
- Gp is the sequence that consists of all models G from G_s that are potentially relevant to G_q and for which $\text{GSim}(G, G_q) > \text{cutoff}$, such that for each Gp_i, Gp_j from Gp holds: if $i < j$ then $\text{GSim}(Gp_i, G_q) \geq \text{GSim}(Gp_j, G_q)$.

The improvement in the time complexity when using the similarity estimation step, can be characterized as follows. Let k be the total number of process models in a collection and n be the average number of nodes in a process model. If node features are used for similarity estimation, the similarity estimation searches the most similar node in a tree-based index, for each node in the query model. There are $k \cdot n$ nodes in the tree at most, when all nodes in the process model collection are distinct from each other. Therefore, the time complexity of the similarity estimation step (using the node features only) has an upper bound of $O(n \cdot \log(k \cdot n))$. The time complexity of the greedy algorithm for process similarity search is $O(n^3)$ [7]. Therefore, the improvement in time complexity is characterized as: $O(p \cdot n^3 - n \cdot \log(k \cdot n))$, where p is the fraction of models that can directly be classified as relevant or irrelevant, after the similarity estimation step.

7 Implementation

This section describes the architecture that we propose for implementing the search algorithm in a business process model repository. The architecture in this paper is based on the more general architecture for business process model repositories that we propose in [33] and focuses on the similarity search aspect. As such, it provides a more detailed design of a single aspect of the architecture for business process model repositories. As a proof of concept, we implemented a web-based prototype of the architecture and the search algorithm.² In this section, first, we present the general three-layer architecture of the tool in terms of a UML component diagram. Second, we make the architecture more concrete, by presenting details of the interfaces of the components and implementations of some of the components. Third, we present the sequence diagrams that describe the behavior of the architecture: one sequence diagram for building an index of business process models and their features and one for searching similar processes. Last, we present the prototype that implements the architecture.

Figure 7 present the general architecture of the tool, which is based on our reference architecture [33]. However, where the reference architecture presents a general architecture that contains all functions that can be implemented by a business process model repository at a high level of abstraction, this paper presents a detailed architecture for the search function only. The architecture consists of three layers: the presentation layer, the process repository management layer and the storage layer.

²Access the prototype at: <http://is.tm.tue.nl/research/apromore.html>. Please take Firefox or Google Chrome as your web browser, since IE does not support the script we use.

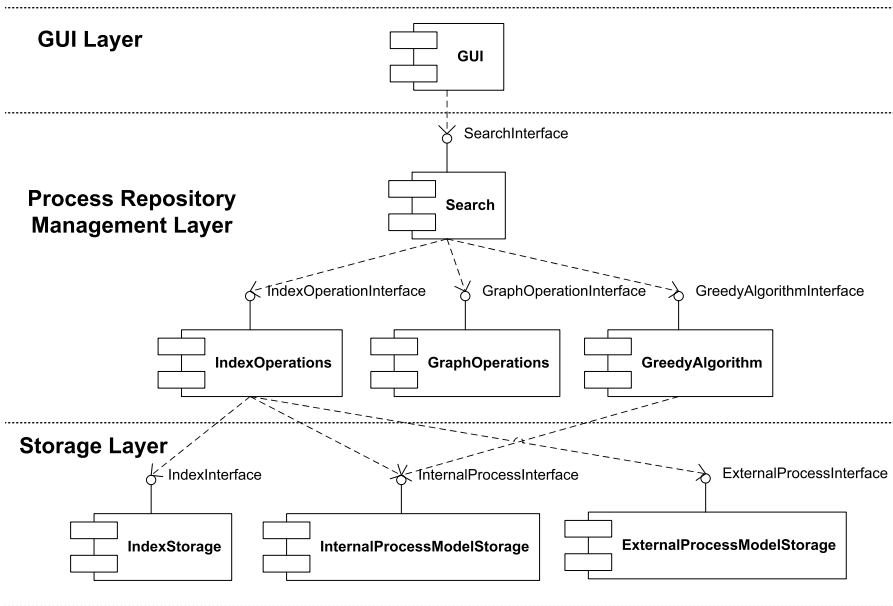


Fig. 7 Architecture of the tool

The presentation layer provides a (graphical) interface for users to interact with the tool. The process repository management layer provides the model management functions. The storage layer stores process data, including indexing information, an internal representation of the business process models that focuses on performance and an external representation of the business process models that focuses on interoperability. Although a business process model repository would typically contain a number of model management functions (such as checking in and checking out models, version management and configuration management), this paper focuses only on the similarity search function. However, the implementation of the search function is consistent with the possible implementation of other functions. The search function is implemented by a single component, such that additional components that implement other functions can be added easily.

Figure 8 shows the architecture in more detail. The process repository management layer consists of four components. The main component is the component that provides the search interface. This interface provides two operations: one for creating an index and one for searching the models that are similar to a given model. The process repository management layer contains three more components: one that provides process graph management functions, one that provides access to the improved greedy algorithm and one that provides operations for managing the indices of process graph features. The greedy algorithm component provides only a single operation for determining process graph similarity, as it is explained in Sect. 5. The process graph component provides two operations: one for transforming a given process model into a process graph and one for deriving the features of a given process graph, as it is explained in Sect. 2. The operation for transforming a process model into a

process graph can be overloaded to enable the conversion of multiple process modeling notations. Our prototype supports both the EPC and the BPMN notation. The index management component provides operations for adapting the indices. It has an operation for initializing the indices (i.e.: creating empty indices) and an operation “constructIndex” that adds a single feature to the existing indices. The component also provides operations for retrieving process models based on the indices. It has an operation “estimate” that estimates which graphs are relevant or potentially relevant to a query graph through indices (Sects. 3.3 and 4). This operation is supported by an operation “getESim” that computes the estimated similarity of two graphs, based on their matched features (Sect. 4) and an operation “getMatchedFeatures” that returns, given a feature, the matching features. More information about the way in which these operations work together to create an index or perform a search will be given in terms of sequence diagrams later on in this section.

The storage layer consists of three components: the indexing component and the internal and external process model component. The indexing component is the core of our design. It stores features and an index based on features. As examples, Fig. 8 contains two types of features. However, subclasses of “Feature” can be created as desired to also store other features. “NodeFeature” stores a label and a number of input and output edges; “Seq2Feature” stores sequences of two nodes. The class diagram describes two types of indices, “InvertedFeatureIndex” and “FeatureRelationIndex”. The former stores the relation between features and the business process graphs in which they are contained. The latter stores hierarchical relations between features as explained in Sect. 4. More precisely, it stores which feature is (direct) parent of which features (Fig. 4). The internal process model component stores the business process models in the format that is used in the repository for efficient computation, which is the process graph in this case (Definition 1). The external process model stores the business process models in their original format. Process models are described in the “ProcessModel” class, which has several subclasses, indicating that process models can be described in different notations, e.g., EPC and BPMN. The class can be extended as desired to store other types of models. In order for those models to work in the repository, the process repository management layer must contain functions to convert them to business process graphs. Note that process models, the corresponding process graphs and features of those process graphs are related via the “processId” that must be unique for a given process model.

For conciseness, Fig. 8 only describes the most important components in detail. We excluded details about the other components, because they are not essential to understand the design and because they would differ in different repositories, for example, to cater for different GUI requirements or to include business process models in different notations. For the same reason, not all operations that are made available by the repository are shown. For example, the external process model storage component only provides an operation to read all models, but obviously also operations should be provided to create, read, update and delete singular models. These operations, however, are not essential to understanding the design. As another example, additional operations should be provided to add a single model to the indexes or to remove a single model from the indexes, such that indexes can be updated incrementally when a model is added to or removed from the repository.

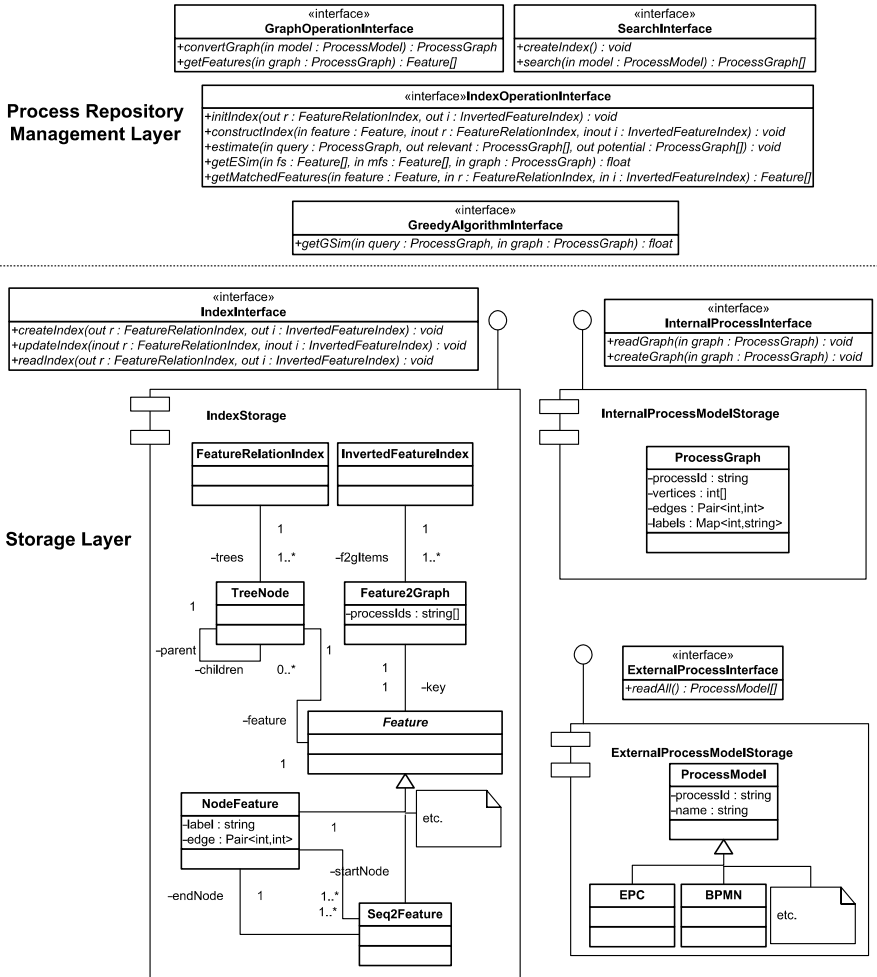


Fig. 8 Class diagram of the tool

As mentioned above, two functions have to be performed by the repository to enable efficient similarity search: building the index and searching for similar processes with respect to a query process. Figures 9 and 10 present the sequence diagrams for these two functions respectively.

To build the index, the GUI invokes the “createIndex” function. First, this function initializes the indexes, by creating empty indexes. Second, it reads all process models from the repository. Third, it converts each process model into a process graph, retrieves all of its features (Sect. 2) and stores the process graph into the repository. Fourth, it inserts each feature of a process graph into the index (Sect. 3.3) and updates the index in the repository. Note that, at this moment, the index is constructed

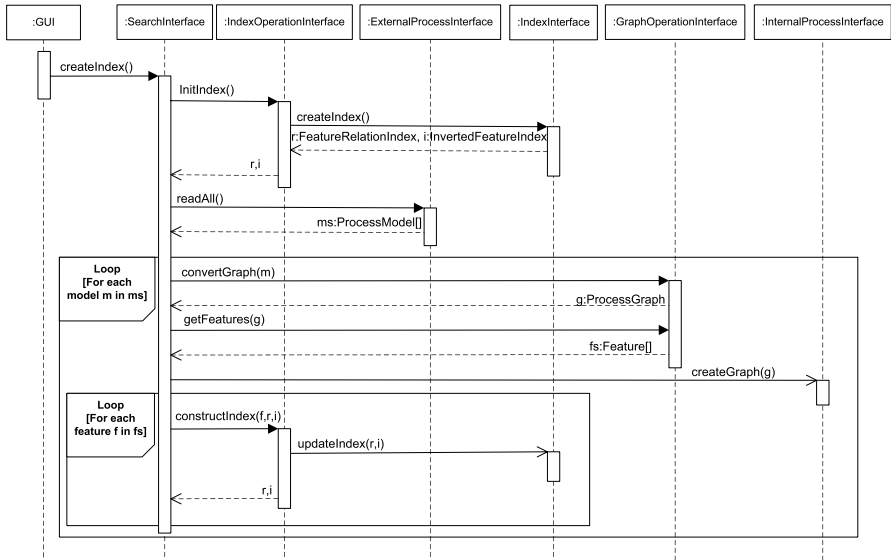


Fig. 9 Sequence diagram for building index

in-memory, instead of in the storage layer. This is not efficient and must be improved in future work.

To search similar processes, the GUI invokes the “search” function. First, this function transforms the given model into a process graph. Second, it invokes the “estimate” function, to compute the relevant and potentially relevant process graphs for the given process graph. The “estimate” function first computes all features of the given query graph. It then reads the indexes into memory. (Again, processing of the indexes is done in-memory, which must be improved in future work.) The “estimate” function then retrieves, from the indexes, the features that match features from the process graph. It then uses this information to compute the estimated similarity of the query graph and the process graphs that have at least one matching feature. Finally, it uses the estimated similarity to determine which graphs are relevant and which graphs are potentially relevant. These two lists are returned. The search component then invokes the improved greedy algorithm (Sect. 5) to compute the similarity of the query graph to each of the potentially relevant process graphs. Finally, the list of results is returned to the user.

As a proof of concept, we implemented a web-based prototype of the architecture and the search algorithm. Figure 11 presents a screenshot of the tool, displaying its main functionality. The GUI consists of three main parts. On the left side the names of the models in the repository are displayed. When a model is selected, it is displayed on the right side. Above the model display there is a toolbar that provides access to the functions that can be performed on the model. One of those functions is the similarity search function. When the similarity search button is clicked, the list of model names on the left is changed to contain only those models that are similar to the given model. Models are automatically indexed by the repository when they are added.

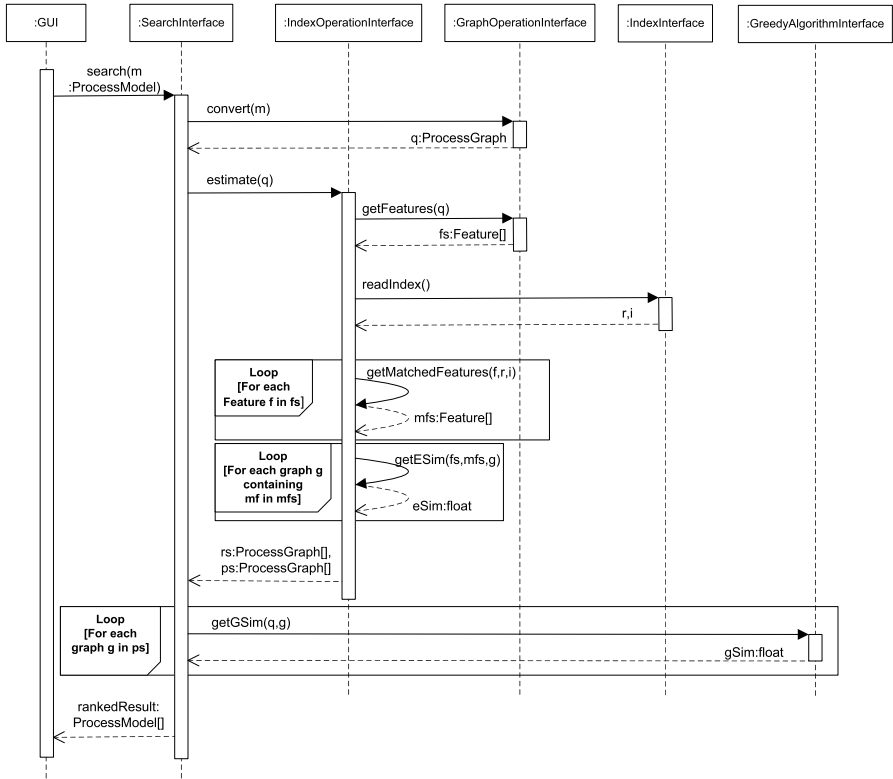


Fig. 10 Sequence diagram for process similarity search

8 Evaluation

This section presents the evaluations of the algorithm described in this paper. The evaluations determine the execution time of the algorithm and the quality of the search results that it returns. In particular we compare the execution time and search result quality of the algorithm from this paper to those of the greedy algorithm [7]. Two evaluations are performed. One homogeneous evaluation in which the query models are taken from the collection that is searched and one heterogeneous evaluation in which the query models are taken from a different model collection.

8.1 Homogeneous evaluation

In this subsection, we present the homogeneous evaluation. We first explain the setup of the evaluation and then the results.

8.1.1 Evaluation setup

We have two experimental setups: one for evaluating the quality of retrieved results and one for evaluating the execution time, respectively.

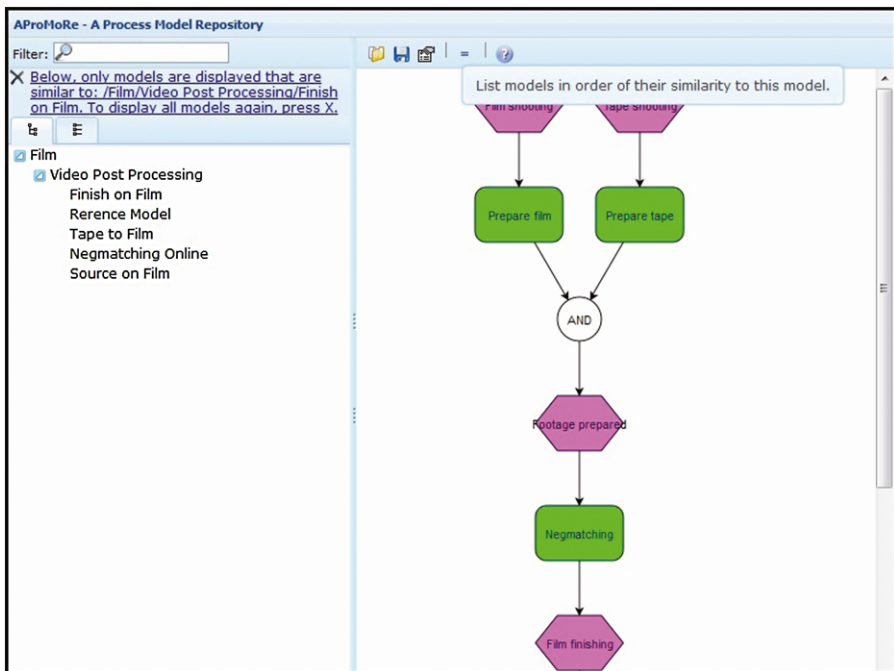


Fig. 11 A screenshot of the tool

Both experiments are performed on the collection of SAP reference models. This is a collection of 604 business process models (described as EPCs) that capture the business processes that are supported by SAP [6]. On average each process model in the collection contains 21.6 nodes with a minimum of 3 and a maximum 130 nodes. The average size of node labels is 3.8 words.

To evaluate the quality of retrieved results, we use the same evaluation dataset as in [7]. This dataset consists of 100 process models that were extracted from the collection of SAP reference models. In addition to that we extracted 10 process models as query models. Consequently, there are 1000 combinations of a query model and a model in the dataset for which the similarity can be determined. For each of those combinations three human observers judged whether the process model is a relevant search result for a particular query model. Next, we can determine the quality of the search results that are returned by a particular algorithm by comparing them to the relevance judgement that is given by the human observers. We can quantify the quality in terms of the R-Precision [4].

Definition 22 (R-Precision) Let \mathcal{D} be the set of process models, \mathcal{Q} be the set of query models and $\text{relevant} : \mathcal{Q} \rightarrow \mathbb{P}(\mathcal{D})$ be the function that returns the set of relevant process models for each query model (as determined by the human observer).

Given the list of search results $D = [d_1, d_2, \dots, d_n]$ for a query q with $d_i \in \mathcal{D}$, the R-Precision is the precision of the first R results, where $R = |\text{relevant}(q)|$ is the total

Table 1 Result quality of the homogeneous evaluation

Feature (n)	Occurrences	Matches	Rel	PoR	Ir	R-Prec
Previous Work [7]	–	–	0	100	0	0.84
1: Node(1)	374	581	5.5	10.9	83.6	0.84
2: 1+Seq(2)	+267	+197	8.1	8	83.9	0.83
3: 2+Seq(3)	+175	+96	7.8	10.1	82.1	0.83
4: 2+Split(3)	+87	+93	7.8	10.1	82.1	0.83
5: 4+Split(4)	+23	+11	7.8	10.1	82.1	0.83
6: 2+Join(3)	+58	+18	7.8	10.1	82.1	0.83
7: 6+Join(4)	+14	+1	7.8	10.1	82.1	0.83

number of process models that is relevant to the query:

$$\text{R-Precision} = \frac{|[d_i \in D | i \leq n, i \leq R, d_i \in \text{relevant}(q)]|}{R}$$

We compare the R-Precision of the greedy algorithm that we developed in previous work [7] to the R-Precision of the improved greedy algorithm that is described in this paper. We use the greedy algorithm, because it is the fastest algorithm of the ones we studied [7] and, therefore, provides a lower-bound for improvements in execution time.

To evaluate the execution time, we compare the 10 queries with all 604 business process models in the collection of SAP reference models, instead of just the 100 process models. We do this, because to compute the execution time we do not need the human judgement and computing the execution time for a larger set of models leads to a more realistic result. We record the average execution time per query.

8.1.2 Evaluation results

Table 1 shows the quality of the results that are retrieved by the greedy algorithm and those returned by the improved greedy algorithm in combination with similarity estimation, based on various feature types.

The rows in the table show the features that are used to do the feature-based similarity estimation. In the first row no feature-based similarity estimation is done. This row lists the performance of the greedy algorithm. In the second row similarity estimation is done based only on node features (of size 1). In the third row similarity estimation is done based on node features plus sequence features of size 2 and so on.

The columns in the table show the properties of the features and similarity estimation based on the features. First, they show the number of times features of a given type occur in the set of process models and the number of times features of a certain type match in the set of process models. For example, in the set of process models, there could be four nodes labeled ‘A’. These nodes count as four occurrences of the node feature type. Because of their high label feature similarity, these nodes can be considered to match. This leads to six matches, because each of the four nodes can be matches to each of the others. Second, the columns show the average number of

Table 2 Execution time of the homogeneous evaluation

Features (n)	Rel	PoR	Ir	T_{est}	T_{com}	$T_{\text{total}}^{\text{avg}}$	$T_{\text{total}}^{\text{min}}$	$T_{\text{total}}^{\text{max}}$
Previous Work [7]	0	604	0	0.00 s	0.60 s	0.60 s	0.16 s	1.45 s
1: Node(1)	7	73	524	0.05 s	0.04 s	0.09 s	0.03 s	0.14 s
2: 1+Seq(2)	13.7	44.9	554.4	0.05 s	0.02 s	0.07 s	0.03 s	0.09 s
3: 2+Seq(3)	9.5	73.2	521.3	0.05 s	0.05 s	0.10 s	0.03 s	0.15 s
4: 2+Split(3)	9.5	73.2	521.3	0.05 s	0.05 s	0.10 s	0.03 s	0.15 s
5: 4+Split(4)	9.5	73.2	521.3	0.05 s	0.05 s	0.10 s	0.03 s	0.15 s
6: 2+Join(3)	9.5	73.2	521.3	0.05 s	0.05 s	0.10 s	0.03 s	0.15 s
7: 6+Join(4)	9.5	73.2	521.3	0.05 s	0.05 s	0.10 s	0.03 s	0.15 s

process models that, after the similarity estimation step, are estimated as being relevant (Rel), potentially relevant (PoR) and irrelevant (Ir) over the ten queries. Third, the columns show the average R-Precision (R-Prec) over the ten queries.

The table shows that when similarity estimation is done based only on node features on average 5.5 models are estimated to be relevant, 10.9 models to be potentially relevant and 83.6 models as irrelevant. Therefore, in this situation, the improved greedy algorithm only has to be used to measure the similarity of about 11% of the total number of process models, about 6% of the models are immediately judged as relevant and the remaining models are judged as irrelevant. In this case the quality of the returned results in terms of R-Precision remains the same. If sequences of size two are also used to perform the similarity estimation, only 8% of the process models has to be compared using the improved greedy algorithm. However, this does lead to a slightly lower R-Precision. Inclusion of other types of features does not improve the similarity estimation any further.

Table 2 shows the execution time of the similarity search both when only using the greedy algorithm and when using certain feature types for similarity estimation and the improved greedy algorithm. All the experiments are done on a computer with the Intel Core2 Duo processor T7500 CPU (2.2 GHz, 800 MHz FSB, 4 MB L2 cache), 4 GB DDR2 memory, the Windows Vista operating system and the SUN Java Virtual Machine version 1.6.

The execution time consists of two parts: the time it takes to estimate the similarity and classify process models as relevant (Rel), potentially relevant (PoR) or irrelevant (Ir), denoted T_{est} ; and the time it takes to compute the similarity for the models classified as potentially relevant, denoted T_{com} . Table 2 shows the average estimation and execution times over the ten search queries. In addition to that it shows the average total time over the ten queries and the (minimum) time of processing the query that takes the least time and the (maximum) time of processing the query that takes the most time.

The table shows that, on average, estimating similarity based on node features helps to retrieve similar models 6.7 times faster and from Table 1 we know that this does not impact the quality of the search results. Also including sequence features of size two helps retrieve similar models 8.6 times faster, but from Table 1 we know that this reduces the quality of the results by about 0.01 in terms of R-Precision as a tradeoff.

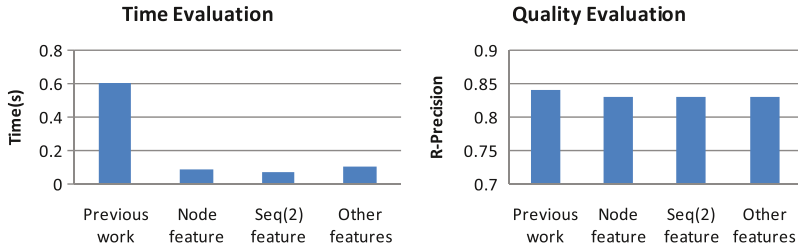


Fig. 12 Results of the homogeneous evaluation

The table also shows that, on average, the total search time for the greedy algorithm is quite acceptable and takes only 0.60 seconds. However, in the worst case the total search time for the greedy algorithm is already 1.45 seconds. This is slower than the response time that one would expect of a search engine. In addition to that, the search time of the greedy algorithm is linear over the number of models in the collection, meaning that if we were to search a collection of 6000 models (which is the size of the collection of business process models of Suncorp-Metway Ltd [16]) the search time would already be around 14 seconds in the worst case.

Figure 12 summarizes the results of the homogeneous evaluation, by showing both the quality and the execution time evaluation. We can see that the algorithm in this paper reduces the execution time (from 0.6 s to less than 0.1 s per query on average) with a stable quality (0.83–0.84 on average) in terms of R-Precision.

Similarity estimation depends on the following parameters:

- $dcutoff$, which is a parameter that determines whether a role feature is considered to be discriminative (Definition 6).
- $lcutoff_{high}$, $rcutoff$ and $lcutoff_{med}$, which are parameters that determine what is considered to be a sufficiently similar for a feature to match (Definition 10).
- $ratio_r$ and $ratio_p$, which are parameters that determine which class a process model belongs to based on the fraction of features that match with the query model (Definition 13).

We vary each of these parameters from 0 to 1 in increments of 0.1 and ran the experiments with all possible combinations of parameter values within this range. We use the parameters that, on average, give the highest R-Precision or the fewest potentially relevant models with respect to the queries to show attractive trade-offs. The values that we use are $dcutoff = 0.3$, $lcutoff_{high} = 0.8$, $rcutoff = 1.0$ and $lcutoff_{med} = 0.2$. The other two parameters also depend on the type of features we use. For the node feature (the second row in Table 1 or 2), $ratio_r = 0.5$; otherwise, $ratio_r = 0.2$. For the node and sequence (with two nodes) features (the second and third rows in Table 1 or 2), $ratio_p = 0.1$; otherwise, $ratio_p = 0.0$.

The improved greedy algorithm for process similarity search depends on the following parameters:

- $wskipn$, $wskipe$ and $wsubn$, which denote the weights given to node deletion, node substitution and edge deletion (Definition 14).
- k , which denotes how many most similar nodes are considered for a search node (Definition 15).

- w_l and w_r , which denote the weights given to the label and role similarities (Definition 16).

For the first group, we used the same values as in [7], i.e., $w_{skipn} = 0.1$, $w_{skipe} = 0.4$ and $w_{subn} = 0.9$. For the second group, we varied each of these parameters from 1 to 10 in increments of 1, and it returns best results when $k = 3$. For the third group, we varied each of these parameters from 0 to 1 in increments of 0.1 and ran the experiments with all possible combinations of parameter values within this range. We used the parameters that give best results, i.e., $w_l = 1.0$ and $w_r = 0.6$.

Note that the parameter settings are specifically tuned to give the best results for this dataset. Parameter values that are generically applicable should be obtained through additional experiments on other process model collections. However, note that a change in parameter settings should not change the conclusions about the comparison between the greedy algorithm and the algorithm in this paper, because both algorithm profit equally from the optimization of the parameters for the evaluation dataset.

8.2 Heterogeneous evaluation

In this subsection, we present the heterogeneous evaluation. We first explain the setup of the evaluation and then the results.

8.2.1 Evaluation setup

In the heterogeneous evaluation, the model collection was extracted from the same collection as for the homogeneous evaluation. However, the query models were taken from a different collection of business process models, which represent the processes of a large manufacturing company. Ten query models were extracted. On average each of these ten process models contains 20.3 nodes with a minimum of 9 and a maximum 35 nodes. The average size of node labels is 4.6 words.

The main difference between the heterogeneous and the homogeneous evaluation is that, in the homogeneous evaluation, it is more clear which models are similar to a given model. For example, the SAP Reference Model contains 7 purchasing models that resemble each other strongly. Consequently, given one of the purchasing models, it is very easy to find the other, similar, ones. For the heterogeneous evaluation, this is more difficult: given a purchasing model (that is not from the SAP Reference Model), similar models are less easy to identify. Therefore, the similarity estimation step will initially lead to more models that are potentially relevant. Consequently, we expect that the similarity estimation step will lead to a smaller efficiency improvement.

Both the SAP reference models and the manufacturing models cover a large number of business functions, which are distributed over different “branches” of the business process model collections (e.g., the collection of SAP reference models has 29 branches and a total of 604 business process models). Models that belong to different branches are typically not similar. To develop a collection of business process models that can be feasibly compared by human observers and that also contains models that are similar, selections from both model collections were made, by first determining

Table 3 Setup for the heterogeneous evaluation

Branch/Business Function	nr. of query models	nr. of document models
Procurement	3	37
Delivery and invoicing	1	
Production planning		17
Sales	4	43
Business planning	2	
Management	2	

similar branches and then selecting models from similar branches. As shown in Table 3, 10 manufacturing models are selected as query models, and 97 SAP reference models are selected as models in dataset, taking overlapping or related branches from the models. The metrics for evaluating quality and time are the same as the previous evaluation, i.e., the R-Precision and the average execution time per query.

8.2.2 Evaluation results

Table 4 shows the results of the greedy algorithm and the algorithm in this paper. The experiments are run on the same computer as the homogeneous evaluation. The rows show the features that are used to estimate the similarity. The columns show the classification of models, result quality and the execution times per query. Process Models are classified as relevant (Rel), potentially relevant (PoR) or irrelevant (Ir) models. Result quality is also measured by R-Precision (R-Prec). The average execution time (T_{total}^{avg}) consists of the estimation time based on features (T_{est}) and the computation time by the improved greedy algorithm (T_{com}). Besides these the columns also show the execution time for the queries that take least (T_{total}^{min}) and most (T_{total}^{max}) time.

The table shows that, by using node features only, around 20% of process models need to be checked with the improved greedy algorithm; the execution time is reduced by 8 times; while the quality is reduced by 0.02 in terms of R-Precision. These findings support our expectation that in the heterogeneous case more models will need to be checked with the greedy algorithm than in the homogeneous case (in the homogeneous case 10% of the process models need to be checked). By further including sequence with two nodes features, around 12% of process models need to be checked with the improved greedy algorithm; the execution time is reduced by 10.7 times; while the quality is reduced by 0.04 in terms of R-Precision. Similar to the previous evaluation, the results do not improve anymore by including more features.

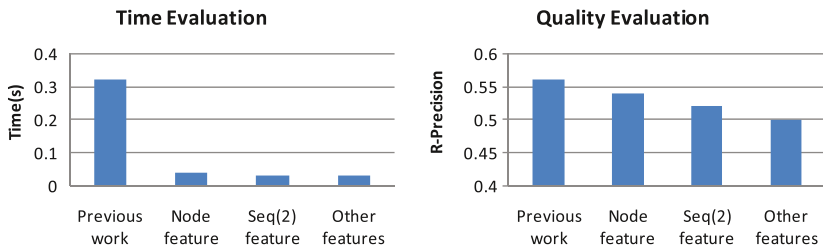
Figure 13 summarizes the results of the heterogeneous evaluation, by showing both the quality and the execution time evaluation. We can see that the algorithm in this paper significantly reduces the execution time with a small decrease in quality.

For the heterogeneous evaluation we changed the values for the parameters $lcutoff_{high} = 0.2$, $lcutoff_{med} = 0.1$, $wskipe = 0.0$ and $wsubn = 0.1$ to obtain the best results. The values of other parameters stay the same as for the homogeneous evaluation.

From both the homogeneous and heterogeneous experiments, we can see that the quality of the homogeneous experiment is higher (0.84 v.s. 0.54). This is because

Table 4 Results of the heterogeneous evaluation

Features (n)	Rel	PoR	Ir	R-Prec	Test	T_{com}	T_{total}^{avg}	T_{total}^{min}	T_{total}^{max}
Previous [7]	0	100	0	0.56	0.00 s	0.32 s	0.32 s	0.20 s	0.51 s
1: Node(1)	74.2	20.4	5.4	0.54	0.02 s	0.02 s	0.04 s	0.02 s	0.06 s
2: 1+Seq(2)	83	11.6	5.4	0.52	0.02 s	0.01 s	0.03 s	0.02 s	0.05 s
3: 2+Seq(3)	85	9.6	5.4	0.50	0.02 s	0.01 s	0.03 s	0.02 s	0.05 s
4: 2+Split(3)	85	9.6	5.4	0.50	0.02 s	0.01 s	0.03 s	0.02 s	0.05 s
5: 4+Split(4)	85	9.6	5.4	0.50	0.02 s	0.01 s	0.03 s	0.02 s	0.05 s
6: 2+Join(3)	85	9.6	5.4	0.50	0.02 s	0.01 s	0.03 s	0.02 s	0.05 s
7: 6+Join(4)	85	9.6	5.4	0.50	0.02 s	0.01 s	0.03 s	0.02 s	0.05 s

**Fig. 13** Results of the heterogeneous evaluation

similar tasks are typically labeled with the same terms in the same collection, but with different terms in different collections. Therefore, it is easier to establish task similarity based on label similarity in homogeneous datasets. Moreover, we currently use string edit distance to compute label similarity in this paper, which is a naive label similarity metric that cannot deal well with different (synonymous) terms being used in similar labels. The result quality of the heterogeneous experiment should therefore be improved by considering synonyms [8, 27] and domain ontologies [11]. We can also see that the execution time of the heterogeneous experiment is much less (0.60 s v.s. 0.32 s). This is because the size of query models. Although the average sizes are almost the same for both sets of query models (21.6 v.s. 20.3), the maximal size differs a lot (130 v.s. 35). Consequently, the slowest query of the homogeneous experiment takes 1.45 s, while the slowest query of the heterogeneous experiment only takes 0.51 s. This causes the homogeneous experiment to take more time on average.

9 Related work

The work presented in this paper is related to: business process similarity search, business process querying, general graph similarity (isomorphism) search, schema matching and ontology matching. We present work on these topics as related work.

Business process similarity search techniques have been developed from different angles [11, 17–19, 23, 26, 29]. These techniques mainly vary with respect to the

information, incorporated in the business process models, that they use to determine similarity [7] and the underlying formalism that they use to determine similarity [10]. The work described in this paper complements existing business process similarity search techniques, because it focuses on estimating business process similarity, rather than measuring it exactly, and using that estimate to improve the time performance of existing techniques. As such it can be combined with any of the existing techniques to improve their performance. Lu and Sadiq [18] also use features to determine similarity, but because their goal differs from the goal of this paper (they want to measure similarity exactly), their features are larger than ours, potentially consisting of a complete process model. This makes their features suitable for measuring similarity exactly, but not for estimating it quickly. Kunze and Weske [15] combine metric trees with process similarity metrics based on edit distances to reduce comparison operations. They compare process similarity based on complete process models, while we estimate process similarity based on different types of features. Furthermore, this method completely relies on metric trees, which requires the process similarity metrics satisfying the positivity, symmetry, and triangle inequality postulates. It is not always the case when we consider synonyms [8, 27] and domain ontologies [11]. Although we also choose metric trees to index labels in this paper, it is optional. For example, we can use the inverted index to index labels for the label similarity metrics considering synonyms.

Process model querying is another related topic. Instead of computing similarity between models, it retrieves process models that satisfy a given query. A query can be described by a query language for process models [1, 3, 5] or a (fragment of a) process model [14]. Awad [1] develops BPMN-Q, a language to query business processes, by extending the BPMN notation. Beeri et al. [3] proposes BP-QL, a language to query business processes modeled in BPEL. Choi et al. [5] proposes IPM-EPDL, a query language for a proprietary process modeling notation based on XML. A notable relation to the work in this paper is the work by Jin et al. [14], who also develop indexing techniques, using sequences in the process models, to improve the efficiency of process model querying.

General graph search has been applied in various application domains, including fingerprint search, DNA search and chemical compound search. In these domains, (sub)graph isomorphism algorithms are used as a basis of graph search, by checking whether a query graph is a subgraph of a graph in the dataset. To avoid comparing two entire graphs, which is time consuming, graph fragments are used as features to build index. This idea is also the basis for this paper. Willett et al. [28] describe feature-based similarity search in a chemical compound databases. ShaSha et al. [25] propose a path-based approach; Yan et al. [30] use discriminative frequent structures to index graphs; Zhao et al. [34] prove that using tree structures and a small number of discriminative graph structures to index graphs is a good choice. Yan et al. [31] also investigate the relationship between feature-based and structure-based methods and built a connection between the two. The main difference between the work that has been done in this area and the work in this paper, is the different nature of business process graphs as compared to graphs in other domains. In particular, there is practically no restriction to the number of possible node labels in a business process graph and matching nodes do not necessarily have the identical labels. In comparison

DNA nodes have four possible labels, chemical compound nodes have 117 possible labels, and in both cases matching nodes have identical labels. Also, business process graphs have different structural properties and patterns. These characteristics require that feature types are defined specifically for business process graphs. In addition to that processing feature similarity is different, because business process graphs do not require features to match exactly for graphs to be similar, while graphs in other domains do require features to match exactly.

The problem of process model similarity search can be related to that of schema matching [24]. There are, however, important differences between process models and schemas. Firstly, data models and schemas generally have labeled edges (associations or schema elements) in addition to labeled nodes. Secondly, the types of nodes and the attributes attached to nodes are different in process models when compared to schemas or data models (e.g. there are no control nodes in data models). During our experiments, we implemented a graph matching technique originally designed for schema matching, namely Similarity Flooding [21]. After adapting the technique to deal with process models, we tested it on the dataset discussed in this paper using various parameter settings [8]. The similarity flooding technique led to a poor score—0.56 of mean average precision for the best settings (with a first-10 precision of 0.6). We attribute this poor performance to the fact that edges in process models do not have labels, while schema matching techniques, such as similarity flooding, heavily rely on edge labels. Madhusudan et al. [19] introduce a structural metric for process model comparison based on similarity flooding. However, Madhusudan et al. rely on a semantic notation in which process models have labels attached to their edges.

The problem of process model similarity search can also be related to that of ontology matching [12]. However, the nature of ontologies and business processes is different; a process model consists of labeled tasks and control flow relations, while an ontology provides a vocabulary, which records the relationship of its terms, e.g., generalization and specialization. This makes it hard to directly use techniques from the area of ontology matching in the area of business process similarity search. However, in future work, it would be worthwhile to investigate the possible use of ontologies and ontology matching for matching tasks and task labels. Ehrig et al. [11] apply such a technique, using WordNet synonyms [22] as an ontology. In previous work we also applied WordNet synonyms to measure the semantic similarity of two labels [8, 27]. However, we still need to develop indexing techniques to use those similarity metrics efficiently.

10 Conclusion

This paper presents an algorithm that improves the efficiency of business process similarity search. The algorithm contains three improvements to an existing algorithm for fast business process similarity search [32]. In addition to that it presents a preprocessing step, in which the similarity of business process models is estimated. The estimation is used to quickly classify business process models as relevant, irrelevant or potentially relevant to a query. The actual similarity computation, which is computationally expensive, then has to be performed for fewer models, namely only those

models that were classified as potentially relevant. The classification is done based on simple, but representative, parts of business process models, also called features.

The greedy algorithm for process similarity search, developed in previous work, is improved in three ways. First, the number of node pairs that must be compared to determine similarity is reduced by initially selecting only a subset of all possible combinations of nodes. Second, the algorithm for similarity computation is improved by computing the similarity incrementally, rather than anew in each iteration of the algorithm. Third, the number of node pairs that must be compared is further reduced by ‘predicting’ the node pairs that should increase the similarity the most in each iteration of the algorithm.

The evaluations that are performed on the algorithm show that, as a consequence of the improvements, the search time of the fastest algorithm for business process similarity search that currently exists can be reduced by a factor 10 with a quality reduction of less than 0.04 (In terms of R-Precision). These reductions are computed as the average over ten search queries. The time reduction for the most complex query is a factor 24.5 and the reduction for the least complex query is a factor 2.5.

The evaluations also show that individual nodes and sequences of two nodes are effective features to quickly compare and classify business process models. Other features that have been used are sequences of three nodes and splits and joins. However, these features do not further improve the quality of the search results or reduce the search time.

There are some research topics that are left for future work. First, in this paper the similarity of nodes in business process models is mainly based on string similarity. However, nodes can be labeled differently using synonyms, in particular when the query models and the models in the dataset are from different organizations. Therefore, we propose that in future work more advanced metrics for label similarity that consider synonyms [8, 27] and domain ontologies [11] are applied. Second, the algorithm in this paper mainly focuses on tasks and connections between them. However, process models often contain more information that may be exploited when determining their similarity, e.g., resources and data used. We propose that the extent to which such information can be used to determine process similarity is investigated in future work. Third, the architecture for fast process similarity search can be extended to incorporate technical measures that improve the efficiency of similarity search. For example, the architecture may allow for distributed processing of search queries.

Acknowledgement The research reported in this paper is supported by the China Scholarship Council (CSC).

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Awad, A.: BPMN-Q: A language to query business processes. In: Proceedings of EMISA 2007, Nanjing, China, pp. 115–128 (2007)

2. Bartolini, I., Ciaccia, P., Patella, M.: String matching with metric trees using an approximate distance. In: Proceedings of the 9th SPIRE, Lisbon, Portugal (2002)
3. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes. In: Proceedings of VLDB 2006, Seoul, Korea, pp. 343–354 (2006)
4. Buckley, C., Voorhees, E.M.: Evaluating evaluation measure stability. In: Proceedings of the ACM SIGIR Conference, pp. 33–40 (2000)
5. Choi, I., Kim, K., Jang, M.: An xml-based process repository and process query language for integrated process management. *Knowl. Process Manag.* **14**(4), 303–316 (2007)
6. Curran, T.A., Keller, G.: *SAP R/3 Business Blueprint*. Addison-Wesley, Bonn (1999)
7. Dijkman, R.M., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. In: Proceedings of the 7th BPM, pp. 48–63 (2009)
8. Dijkman, R.M., Dumas, M., van Dongen, B., Uba, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. *Inf. Syst.* **36**(2), 498–516 (2011)
9. Documentair structuurplan, <http://www.model-dsp.nl>
10. Dumas, M., García-Bañuelos, L., Dijkman, R.M.: Similarity search of business process models. *Bull. Tech. Comm. Data Eng.* **32**(3), 23–28 (2009)
11. Ehrig, M., Koschmider, A., Oberweis, A.: Measuring similarity between semantic business process models. In: Proceedings of the 4th Asia-Pacific Conference on Conceptual Modelling, Ballarat, Victoria, Australia, pp. 71–80 (2007)
12. Euzenat, J., Shvaiko, P.: *Ontology Matching*. Springer, Berlin (2007)
13. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: Proceedings of the 21th ACM International Symposium on Principles of Database Systems (2001)
14. Jin, T., Wang, J., Wu, N., La Rosa, M., ter Hofstede, A.: Efficient and accurate retrieval of business process models through indexing. In: Proceedings of the 18th CoopIS, Part I, Crete, Greece. *Lecture Notes in Computer Science*, vol. 6426, pp. 402–409. Springer, Berlin (2010)
15. Kunze, M., Weske, M.: Metric trees for efficient similarity search in large process model repositories. In: Proceedings of the 1st International Workshop on Process in the Large, Hoboken, NJ, USA (2010)
16. La Rosa, M., Dumas, M., Uba, R., Dijkman, R.M.: Business process model merging: An approach to business process consolidation. QUT ePrint 38241, Queensland University of Technology, Australia (2009)
17. Li, C., Reichert, M.U., Wombacher, A.: On measuring process model similarity based on high-level change operations. In: Proceedings of the 27th International Conference on Conceptual Modeling, Barcelona, Spain, pp. 248–264 (2008)
18. Lu, R., Sadiq, S.: On the discovery of preferred work practice through business process variants. In: Proceedings of the 26th International Conference on Conceptual Modeling, Soringer, Heidelberg, Germany, pp. 165–180 (2007)
19. Madhusudan, T., Zhao, L., Marshall, B.: A case-based reasoning framework for workflow model management. *Data Knowl. Eng.* **50**(1), 87–115 (2004)
20. Manning, C.D., Raghavan, P., Schütze, H.: *Introduction to Information Retrieval*. Cambridge University Press, Cambridge (2008)
21. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarityflooding: Aversatile graph matching algorithm and its application to schema matching. In: International Conference on Data Engineering, pp. 117–128. IEEE Computer Society, Los Alamitos (2002)
22. Miller, G.: WordNet: A lexical database for English. *Commun. ACM* **38**(11), 39–41 (1995)
23. Minor, M., Tartakovski, A., Bergmann, R.: Representation and structure-based similarity assessment for agile workflows. In: Weber, R., Richter, M. (eds.) 7th International Conference on Case-Based Reasoning, pp. 224–238 (2007)
24. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. *VLDB J.* **10**(4), 334–350 (2001)
25. ShaSha, D., Wang, J., Giugno, R.: Algorithmics and applications of tree and graph searching. In: Proceedings of the 21th ACM International Symposium on Principles of Database Systems, pp. 39–53 (2002)
26. van der Aalst, W., Medeiros, A., Weijters, A.: Process equivalence: Comparing two process models based on observed behavior. In: Proceedings of BPM 2006, pp. 129–144. Springer, Vienna (2006)
27. van Dongen, B.F., Dijkman, R.M., Mendling, J.: Measuring similarity between business process models. In: Proceedings of the 20th CAiSE, pp. 450–464 (2008)
28. Willett, P., Barnard, J., Downs, G.: Chemical similarity searching. *J. Chem. Inf. Comput. Sci.* **38**, 983–996 (1998)

29. Wombacher, A.: Evaluation of technical measures for workflow similarity based on a pilot study. In: Proceedings of CoopIS 2006, pp. 255–272. Springer, Berlin (2006)
30. Yan, X., Yu, P.S., Han, J.: Graph indexing: A frequent structure-based approach. In: Proceedings of the 2004 ACM SIGMOD, pp. 335–346 (2004)
31. Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: Proceedings of the 2005 ACM SIGMOD, pp. 766–777 (2005)
32. Yan, Z., Dijkman, R.M., Grefen, P.W.P.J.: Fast business process similarity search with feature-based similarity estimation. In: Proceedings of the 18th CoopIS, Part I, Crete, Greece. Lecture Notes in Computer Science, vol. 6426, pp. 60–77. Springer, Berlin (2010)
33. Yan, Z., Dijkman, R.M., Grefen, P.W.P.J.: Business process model repositories—framework and survey. *Inf. Softw. Technol.* **54**(4), 380–395 (2012)
34. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: Tree + Delta \geq Graph. In: Proceedings of the 2007 ACM VLDB, pp. 938–949 (2007)