Available online at www.sciencedirect.com
ScienceDirect
Information Systems 32 (2007) 685–712
www.elsevier.com/locate/infosys

Mining unconnected patterns in workflows

Gianluigi Greco[a,*], Antonella Guzzo[b,c], Giuseppe Manco[b], Domenico Saccà[b,c]

[a]*Department of Mathematics, University of Calabria, Via P.Bucci 30B, 87036 Rende, Italy*
[b]*ICAR, CNR, Via P.Bucci 41C, 87036 Rende, Italy*
[c]*DEIS, University of Calabria, Via P.Bucci 41C, 87036 Rende, Italy*

Received 21 September 2005; received in revised form 11 March 2006; accepted 22 May 2006
Recommended by M. Weske

Abstract

General patterns of execution that have been frequently scheduled by a workflow management system provide the administrator with previously unknown, and potentially useful information, e.g., about the existence of unexpected causalities between subprocesses of a given workflow. This paper investigates the problem of mining *unconnected* patterns on the basis of some execution traces, i.e., of detecting sets of activities exhibiting no explicit dependency relationships that are frequently executed together. The problem is faced in the paper by proposing and analyzing two algorithms. One algorithm takes into account information about the structure of the control-flow graph only, while the other is a smart refinement where the knowledge of the frequencies of edges and activities in the traces at hand is also accounted for, by means of a sophisticated graphical analysis. Both algorithms have been implemented and integrated into a system prototype, which may profitably support the enactment phase of the workflow. The correctness of the two algorithms is formally proven, and several experiments are reported to evidence the ability of the graphical analysis to significantly improve the performances, by dramatically pruning the search space of candidate patterns.

Keywords: Frequent patterns discovery; Graph mining; Workflow management

1. Introduction

A workflow is a partial or total automation of a business process, in which a collection of *activities* must be executed by humans or machines, according to certain procedural rules. Workflows may be conveniently defined, analyzed and supported by means of *Workflow Management System* (WfMS). These systems represent the most effective technological infrastructure for managing business processes in several application domains (cf. [1–4]), and they are, therefore, more and more utilized into enterprises. In fact, enhancing the functionalities of WfMSs has become a very active research area in recent years, and several efforts have been already spent in order to provide facilities for the human system administrator while designing complex processes as well as to offer an "intelligent" support in the decisions which have to be taken by the enterpriser during the enactment [5–9].

In this paper, we continue on the way of enhancing the functionalities of WfMSs by proposing some data

*Corresponding author. Tel.: +39 0984 496429; fax: +39 0984 839054.
E-mail addresses: ggreco@mat.unical.it (G. Greco), guzzo@icar.cnr.it (A. Guzzo), manco@icar.cnr.it (G. Manco), sacca@icar.cnr.it (D. Saccà).

0306-4379/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2006.05.001

mining techniques, which are specifically tailored to help the human system administrator to gain some previously unknown, and potentially useful information from the low-level data collected into the *log* files during past enactments of the system. Differently from classical *process mining* techniques, where the focus is on extracting the "hidden" model underlying the low-level data (see, e.g., [10–13] and the discussion in Section 6), in this paper we deal with the enactment phase of the workflow life-cycle: we assume that a workflow model has been already designed and installed over some platforms, and we want to provide an effective support for the decisions to be taken during each execution, based on the history of past executions. To this aim, specific data mining techniques can be used depending on the perspectives workflow specifications are looked from (cf. [14]).

In a simple scenario where one looks at workflows from a *data perspective* only, classical mining techniques, such as the "market basket analysis", can be used to find interesting and potentially useful knowledge about the business data at hand. However, these classical techniques do not fit scenarios where one looks at workflows from a *control-flow perspective*, in which the focus is instead on the causal relations and on the constraints on the occurrence of the tasks. For instance, the administrator may be interested in knowing whether there are subprocesses frequently scheduled in the same enactment, or whether there are correlations (in previous enactments) between the order of execution of a set of activities and the execution of a specific final activity.

In this perspective, important correlations among activities are already hardwired in the workflow model, independently from any execution history and, therefore, the mining may be effectively driven by the workflow structure to detect unexpected and useful information only.

And, in fact, the peculiarities of the domain raise the necessity to deal with the natural graphical representation of control flows, so that specific mining algorithms have to be conceived. An example applicative context is next illustrated by considering the automatization of a "sales ordering" process.

Example 1. Consider the workflow schema depicted in Fig. 1, according to the graphical notation of the *event-driven process chains* (*EPCs*) [15], where the process is represented as chains of *events* (drawn as hexagons) and *functions* (drawn as boxes) connected by means of logical *connectors*. The schema supports a sales order process as follows. As soon as a customer issues a request to purchase a given product, the enterpriser checks the availability of the required items. Three different stocks are available: stocks X and Y, containing items of type A; and stock Z which contains items of type B. If the requested items are not available in the respective stocks, the order is rejected and the corresponding transaction is aborted. Otherwise, the order is accepted and the items are shipped to the customer. In this latter case, two further events may occur: either the trial period elapses without any claim by the customer, or the customer issues a claim for defective item (in which case a substitution has to be accomplished).

In this scenario, it could be crucial to characterize the discriminant factors that will lead to the rejection of the order or to the shipping of a defective item. In fact, specialized mining techniques may reveal important information to be used to diagnose the business process and to identify problems within the supply chain. For instance, by looking at the traces

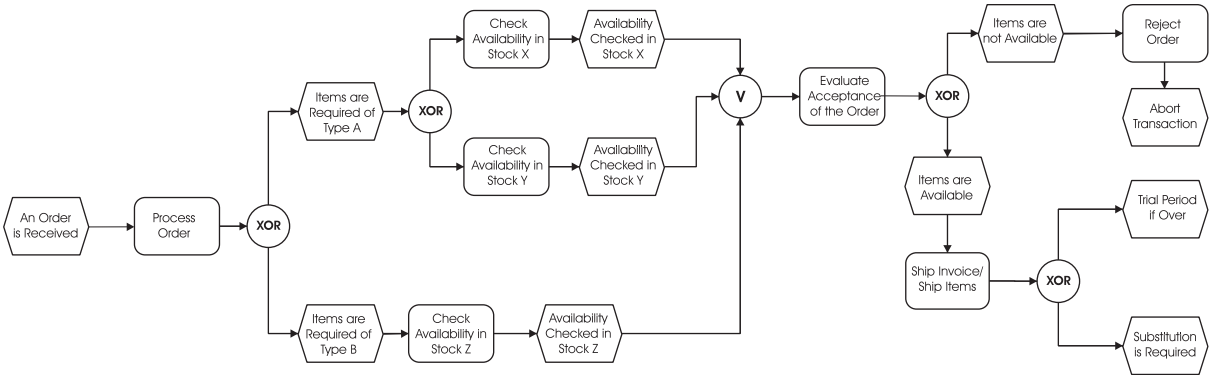


Fig. 1. An example workflow schema.

stored in the log files, one may discover that requests for items of type B frequently ended in the previous enactments with an "*Abort Transaction*" event. Clearly enough, this mined knowledge is useful for the enterprise, for it evidences some dimensioning/structuring problems with stock Z.

A first contribution towards the definition of techniques tailored for the scenarios depicted above has been proposed by Greco et al. [16], where the problem of identifying the structures of the executions, also called *patterns*, that have been scheduled more frequently by the workflow system (short: *frequent patterns*) has been proposed.

Settled within the graphical representation of workflows, frequent patterns may be roughly viewed as subgraphs corresponding to subprocesses of the process at hand. And, in fact, the algorithms proposed in [16] are able to discover frequent patterns that are also *connected*. These algorithms are useful, for instance, in scenarios similar to the one described in Example 1: there, a correlation between the requests for items of type B and the "*abort transaction*" event necessarily entails (because of the workflow structure) that all the functions and events involved in the path connecting them frequently co-occur in the log traces as well. However, albeit very valuable, connected patterns are not the sole important information which can be extracted from the analysis of the execution traces, as evidenced below.

Example 2. Consider again the workflow schema depicted in Fig. 1. By looking at some execution traces, one may also notice that orders involving items of type A frequently end with a request for substitution. Notice that, although the "*items are required of type A*" and "*substitution is required*" events co-occur frequently, it does not necessarily mean that the events contained within any path between them are frequent as well, i.e., that a frequent connected pattern has to be detected. Indeed, it could be the case that only a small percentage of the requests dispatched by a single stock end with a request for substitution; yet, considered regardless of the stock which serves them, the frequency of the orders for items of type A ending with a request for substitution could be significantly high, thus highlighting a problem in the supplier and/or in the production process. In this circumstance, a frequent *unconnected* pattern can be detected that provides the enterprise with the suggestion of revising the supplying process for item A. But, by analyzing patterns in the way paved in [16], one would not identify this criticality, since no particular problem can be detected in any of the two stocks.

In this paper, we continue on the way paved by Greco et al. [16], by generalizing the problem of discovering frequent connected patterns to more general patterns of execution, and in particular by investigating the problem of discovering possible "correlations" among unconnected patterns, once frequent connected ones have been already discovered.

We point out that the techniques we propose in the paper assume that a limited set of executions is given as input; so they are not well suited for performing correctness checks on the basis of all possible executions, e.g., for determining modelling anomalies such as unsuccessful paths. Indeed, such anomalies can be better detected at design time by means of reasoning tools, especially if approaches to the modelling are used which impose correctness constraints leading to effective checks (as in the case of Petri nets-based models [14]). Thus, we assume that possible anomalies in the modelling have been already detected and eventually removed, so that we can rather focus on extracting unexpected knowledge about the actual execution flow of the processes.

For instance, our techniques can be used for singling out sets of subprocesses that are very often executed together; for identifying subprocesses which are related by some unexpected (w.r.t. the workflow structure) causal relation; for identifying association rules among the tasks of the workflow; and for identifying critical subprocesses that lead with high probability to (un)desired final configurations. More in general, these techniques can be used for identifying a synthetic but focused picture of the actual behavior of the workflow system within a specific application scenario, which can be profitably exploited during further coming enactments.

1.1. Contributions and organization

Technically, the basic tool one needs for discovering all the kinds of knowledge described above is the ability of dealing with *unconnected* patterns, which are arbitrary subsets of connected patterns exhibiting no explicit dependency relationships. The aim of this paper is precisely to investigate this research issue by designing and implementing efficient solutions for the *frequent unconnected patterns* discovery

(short: FUPD) problem, in which a set of frequent connected patterns is given in input, and all the subsets of this set that are frequent as well have to be discovered. In more details:

- We introduce and formalize the FUPD problem of discovering frequent unconnected patterns in workflow executions.
- We present a first solution for FUPD, that is, the *ws-unconnected-find* algorithm, consisting in the application of a levelwise algorithm (in the *a priori* style [17]) which combines all the unconnected patterns and then checks for their frequency.
- We show how the structure of the workflow together with some elementary information such as the frequency of the occurrences of elementary activities suffices for significantly pruning the search space. Accordingly, we enhance and optimize the *ws-unconnected-find* algorithm, thus obtaining an efficient and practically fast algorithm, called *ws*-unconnected-find*.
- We perform an in-depth theoretical analysis of the algorithms for formally proving their correctness. In particular, we prove that the pruning of the search space performed by the *ws*-unconnected-find* algorithm is sound, i.e., no frequent unconnected patterns are discarded.
- Finally, all the algorithms described in the paper have been implemented and tested. In particular, we evaluate the effectiveness of the pruning strategies introduced in the *ws*-unconnected-find* algorithm, by performing several experiments and comparisons.

The rest of the paper is organized as follows. In the next section, we define the formal model of workflow and review the problem of mining frequent connected patterns of execution. In Section 3, we propose a levelwise algorithm for discovering frequent unconnected patterns. Section 4 describes how the analysis of the workflow structure and instances allows to deduce tight bounds on the frequency of candidate patterns, and consequently to specialize the a priori-based algorithm. The algorithms have been implemented and integrated into a software architecture which is discussed in Section 5, where results of experimental activity are also reported. Finally, Section 6 discusses some related works, and Section 7 draws our conclusions.

2. Workflow model and mining problems

In this section, we introduce the formal framework we shall exploit throughout the paper. In particular, we first present the basic notions and definitions to deal with workflow specifications, and we subsequently overview the approach in [16] to face the problem of mining frequent connected patterns of execution.

2.1. Workflow model: some preliminary observations

Despite the efforts of the Workflow Management Coalition [18], a plethora of languages and models still exist to support WfMSs, each one being characterized by specific advantages and disadvantages. For instance, many leading tools in the field of business process engineering (e.g., SAP R/3 [19] and ARIS [20]) model processes by means of EPCs [15], where the control-flow structure of the process is represented as a chain of *events* and *functions* connected by means of logical *connectors*. However, EPCs have been not intended as a *formal* specification and, in fact, they are not equipped with a clear and non-ambiguous semantics (cf. [21]). Other systems are, instead, based on Petri nets [22,14] possibly enhanced with specific extension and restrictions, in order to have a formal semantics exhibiting parallelism, concurrency, synchronization, non-determinism and mutual exclusion despite the graphical nature. Finally, other systems use proprietary languages which may be possibly inspired by formalisms such as the *concurrent transaction logic* [23,24], the *state chart model* [25,26], the *active object oriented* paradigm [27], and the *process algebra* [28].

Fortunately, for the kind of application discussed in the paper, i.e., to deal with the problem of mining frequent patterns of executions, the choice of a modelling language is not a crucial issue, since in order to single out frequent patterns of execution two inputs are required only, that are: (1) a graphical representation of the workflow schema, and (2) a log of enactments compliant with the schema. Therefore, we just need to look at workflow specifications from the *control-flow* perspective (cf. [14]), and actually to focus on the definition of the tasks occurring in the process at hand and of the relationships of precedence among them. More advanced modelling features are of limited interest in this context and, thus, they can be disregarded without loosing in generality—the reader interested in expanding on these aspects is referred to, e.g., [29–31,14].

In the light of the observations above, we decided to model a *workflow schema* $\mathscr{WS}$ as a tuple $\langle A, E, a_0, A_F\rangle$, where A is a finite set of *activities* (also called nodes in the following), $E \subseteq (A - A_F) \times (A - \{a_0\})$ is a relation of precedences among activities (whose elements are simply called edges), $a_0 \in A$ is the starting activity, and $A_F \subseteq A$ is the set of final activities. The tuple $\langle A, E\rangle$ is roughly referred to as the *control-flow* graph of $\mathscr{WS}$, in the following. As an example, a control flow graph which will be referred to throughout the paper is shown in Fig. 2(a), for which a is the starting activity, and $A_F = \{\mathtt{p}, \mathtt{q}\}$. Note that focusing on the control-flow graph only makes our approach completely orthogonal w.r.t. the underlying WfMSs, since other more elaborate specifications, such as EPCs or Petri nets, can be eventually formulated in our syntax, by possibly discarding their specific more advanced issues. As an example, we leave to the reader the (straightforward) task identifying the control-flow graph in the EPC shown in Fig. 1.

Actually, beside the syntax, another problem come into play with the specification of the languages, which pertains the semantics of the model (cf. [32]). This is particularly true in the presence of branching and synchronization patterns, e.g., for the *synchronized merge* construct whose non-local behavior causes serious semantics problems, faced in pragmatic ways by the different vendors (see [33]). However, the heterogeneity in the semantics for a workflow schema is again not a serious issue for our aims. Indeed, we are not interested in developing verification or analysis techniques which require the formalization of a suitable executable semantics. Rather, we stress that the problem of mining frequent patterns just requires that some enactments are given at hand, no matter of the way they have been actually computed and, thus, independently from the intended semantics underlying the workflow schema. In fact, our approach is to a large extent "syntactic", and the proposed techniques result to be orthogonal w.r.t. any specific semantics.

Therefore, no specific executable semantics is discussed and, again, the reader interested in expanding on this subject is referred to the literature (e.g., [29,30,14]). On the other hand, we assume that a workflow schema exists which has been firstly modelled and then enacted in some system; that correctness/structuring constraints are issued on it; and that the executions stored in the log actually reflect this (possibly unknown) semantics. In particular, by abstracting from the specificity of the logs, we may assume (as often done in the literature) that each enactment is registered in the log as a sequence of nodes identifiers, also called *trace*, denoting the ordering of occurrence of the various tasks. Then, the only (trivial) requirement is that this order must respect the relationships of precedence in the control-flow graph, i.e., an activity a may precede an activity b in a trace if and only if there is a directed path from a to b in the control flow. As an example, Fig. 2(b) reports some traces for the workflow schema whose control-flow graph is shown in Fig. 2(a).

Let t be a trace for a workflow schema $\mathscr{WS} = \langle A, E, a_0, A_F\rangle$. Starting from t, we can identify the subgraph of the control flow which was enacted as the subgraph $I(t)$ of $\langle A, E\rangle$ induced over the nodes in t. The graph $I(t)$ is called an *instance* of $\mathscr{WS}$, denoted by $\mathscr{WS} \models I(t)$. And, consistently, a log for a workflow schema $\mathscr{WS}$ may be equivalently viewed as either a bag of traces or a bag of instances of $\mathscr{WS}$. As an example, Fig. 3 shows the instances

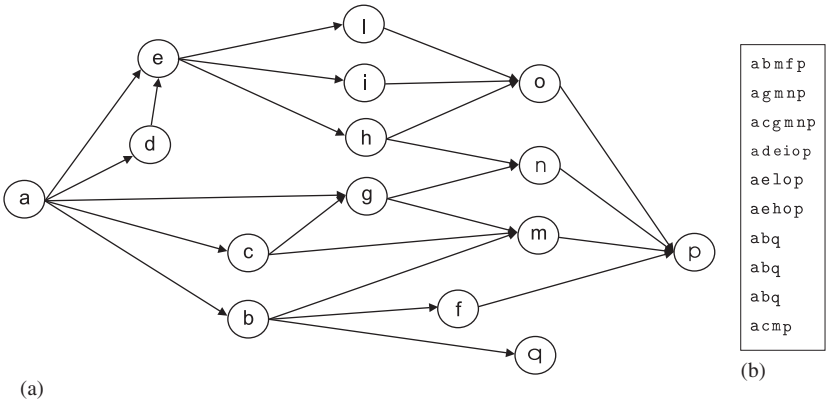


Fig. 2. An example workflow schema: (a) the control-flow graph, and (b) some traces for it.

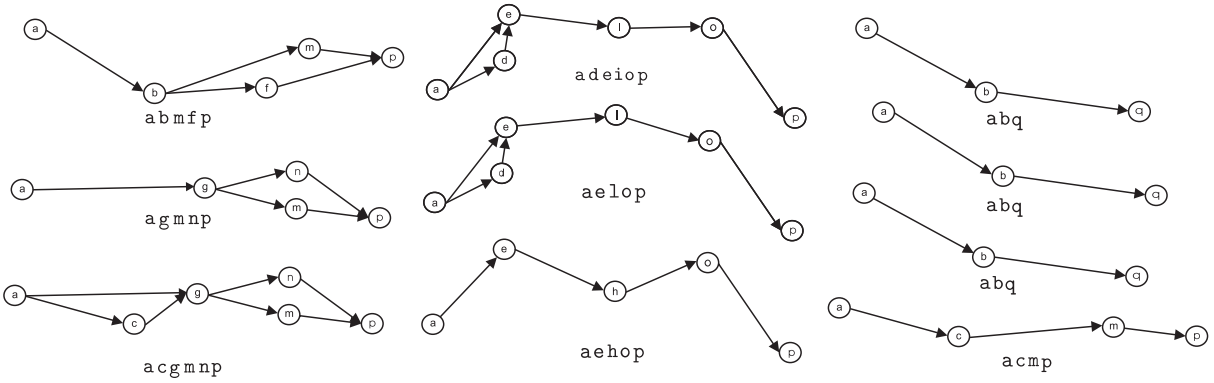


Fig. 3. Example instances.

associated with the traces in Fig. 2(b) for the workflow schema whose control flow is depicted in Fig. 2(a). Throughout the paper we shall deal with such a graphical representation rather than with traces, in order to better exploit the topological properties of the instances in the algorithms. In this respect, it is worth noting that some commercial tools, such as ARIS process performance monitor [34], precisely deal with instances in form of graphs, and that more elaborate techniques have been proposed to reconstruct instance graphs from traces, even when the underlying process model is unknown [35].

2.2. Formal framework

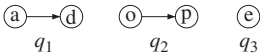
Let us assume that a workflow schema $\mathscr{WS}$ is given together with a log file, i.e., a bag of instances $\mathscr{F} = \{I_1, \ldots, I_n\}$ such that $\mathscr{WS} \vDash I_i$, for each $1 \leqslant i \leqslant n$. In this section, we formalize the basic problems we deal with in the paper. Roughly speaking, among the instances of $\mathscr{F}$, we are interested in discovering the most frequent patterns of execution as next defined.

Definition 3 (*Pattern*). A graph $p = \langle A_p, E_p \rangle$ is an $\mathscr{F}$-pattern (cf. $\mathscr{F} \vDash p$) if there exists $I = \langle A_I, E_I \rangle \in \mathscr{F}$ such that $A_p \subseteq A_I$ and p is the subgraph of I induced by the nodes in A_p. Whenever $\mathscr{F}$ is clear from the context an $\mathscr{F}$-pattern is simply said to be a *pattern*.

In the following, we use the notation $p_1 \cup p_2$ to denote the pattern $\langle A_{p_1} \cup A_{p_2}, E_{p_1} \cup E_{p_2} \rangle$, and $p_1 \subseteq p_2$ to denote the case where both $A_{p_1} \subseteq A_{p_2}$ and $E_{p_1} \subseteq E_{p_2}$ hold. Sometimes, a pattern $p_1 \cup \cdots \cup p_n$ is equivalently represented as the sets of all its components, i.e., as $\{p_1, \ldots, p_n\}$.

A crucial property for patterns is the connectedness. Formally, a pattern p is *connected* if its undirected version is connected, otherwise p is *unconnected*.

Example 4. Let us consider again the schema depicted in Fig. 2. Then, the following subgraphs are connected patterns:



On the other hand, it is easy to see that the patterns $\{q_1, q_2\} = q_1 \cup q_2$ and $\{q_1, q_3\} = q_1 \cup q_3$ are not connected.

Let $supp(p) = |\{I | \{I\} \vDash p \wedge I \in \mathscr{F}\}|/|\mathscr{F}|$ be the *support* of an $\mathscr{F}$-pattern p, i.e., the number of instances in the log $\mathscr{F}$ containing p as a subgraph. Then, given a real number *minSupp*, we consider the following two relevant problems on workflows:

FCPD: *Frequent connected pattern discovery*, i.e., finding all the connected patterns whose support is greater than *minSupp*.

FUPD: *Frequent unconnected pattern discovery*, i.e., finding all the subsets of connected patterns (short: unconnected patterns) whose support is greater than *minSupp*.

Importantly, as a side effect of considering a frequency threshold, we have that the formalization of the problems FCPD and FUPD is to large extent immune to noise, exceptions and anomalies in the logs, which are in fact filtered out on the basis of their statistic relevance. Therefore, the problems above are well-defined even in the presence of logs that do not fully comply with the control-flow graph or that are, for instance, incomplete.

Example 5. Let us consider again the workflow shown in Fig. 2, and the log consisting of the instances depicted in Fig. 3. Let $minSupp = 30\%$. Then, the pattern q_2 of Example 4 is a frequent connected pattern, because it occurs in three of the 10 instances. Conversely, q_1 is unfrequent, since it occurs in two instances only. Also, notice that none of the nodes l, i, h is frequent, whereas q_3 is frequent. Moreover, it is easy to see that $\{q_2, q_3\}$ is a frequent unconnected pattern, since each instance including q_3 also includes q_2.

Note that the FCPD problem has been already faced in [16], by resorting on the idea of exploiting the control graph of $\mathscr{WS}$ in order to reduce the number of patterns to generate. Specifically, [16] introduces the notion of *weak patterns* and shows that working with w-patterns rather than $\mathscr{F}$-patterns is not an actual limitation, since each frequent $\mathscr{F}$-pattern is bounded by w-patterns. Indeed, for each frequent $\mathscr{F}$-pattern p, a frequent w-pattern p' exists such that $p \subseteq p'$, and, moreover each weak pattern $p' \subseteq p$ is frequent as well. Thus, weak patterns can be used in a smart exploration of the search space, by adopting a breadth-first search strategy. Roughly speaking, frequent weak patterns can be extracted incrementally, by starting from frequent "elementary" weak patterns (i.e., weak patterns obtained by processing a single node and then satisfying all the constraints), and by extending each frequent weak pattern using two basic operations: adding a frequent edge and merging with another frequent elementary weak pattern. The correctness of the approach follows from the observation that the space of all connected weak patterns can be traversed by means of the precedence relation $\lessdot$, defined as follows: $p_1 \lessdot p_2$ if and only if p_2 can be obtained from p_1 by either adding an edge, or merging p_1 with an elementary weak pattern it does not include. Details on the algorithm exploiting this observation, called w-find, can be found in [16]. Here, we just note that the frequent connected patters obtained by w-find on the instances of Fig. 3 with support 30% are those shown in Fig. 4.

In the following section, we shall make use of the set of all the frequent connected patterns for solving the more general FUPD problem.

3. Mining unconnected patterns

Assume that the set $C(\mathscr{F})$ of all the frequent (w.r.t. *minSupp*) connected patterns in the set of instances

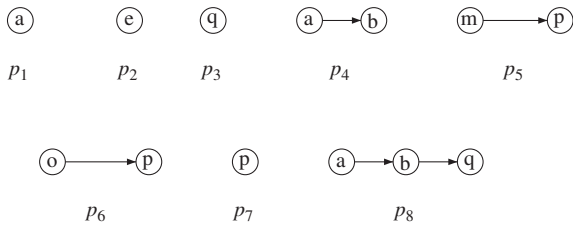


Fig. 4. Example frequent patterns on the instances of Fig. 3.

$\mathscr{F}$ for a schema $\mathscr{WS}$ has been already computed (using, e.g., the w-find algorithm). In this section, we investigate how to exploit $C(\mathscr{F})$ in order to efficiently deal with the FUPD problem. For the sake of simplicity, let us assume throughout the section that a threshold *minSupp* is given, so that by *frequent patterns* we simply mean patterns whose support is greater than *minSupp*.

We present a solution to the FUPD problem that relies on the application of a levelwise algorithm (in the *a priori* style [17,36]) for combining all the patterns in $C(\mathscr{F})$, by subsequently checking for their frequency. The algorithm, called *ws-unconnected-find*, is shown in Fig. 5. It receives in input the workflow schema $\mathscr{WS}$ and the set $C(\mathscr{F})$ of the frequent connected $\mathscr{F}$-patterns, and returns all the subsets of $C(\mathscr{F})$ that are frequent as well.

While combining different connected patterns, *ws-unconnected-find* exploits some smart strategies which are able to reduce the search space and avoid unnecessary computations. Specifically, the algorithm takes benefit of the structural information of the workflow schema $\mathscr{WS}$ only. An extension of the algorithm which is also capable of exploiting statistical information about the set $\mathscr{F}$ of instances is presented in Section 4.

The first important observation for understanding how *ws-unconnected-find* is designed is that in order to solve the FUPD problem, we can focus without loss of generality on the sets of frequent connected patterns containing the starting activity, say a_0, of $\mathscr{WS}$. Indeed, since the starting activity is executed in each instance, each set of connected patterns that is frequent in $\mathscr{F}$ can be extended with the initial activity or, generally, with a pattern containing the initial activity, without modifying its frequency. Therefore, the following property trivially holds.

Proposition 6. *For any two patterns p and q such that $p \cup q$ is a frequent unconnected pattern, there exists a pattern p' containing both p and the starting activity*

Input: A workflow schema $\mathcal{WS}$, a set $\mathcal{F}$ of instances of $\mathcal{WS}$, the minimal support *minSupp*, the set $C(\mathcal{F})$ of frequent connected $\mathcal{F}$-patterns.
Output: The set of all the frequent starting patterns.
Method: Perform the following steps:

```
1   InitializeStructures();
2   L_0 := { p | p ∈ C(F), a_0 ∈ p };      //***frequent connected starting patterns
3   k := 0, R := L_0; C' := C(F) − L_0;
4   repeat
5     U := UpdateCandidateList(L_k)
6     L_{k+1} := ComputeFrequentPatterns(U);
7     R := R ∪ L_{k+1};
8     forall r ∈ U − L_{k+1} do begin
9       p := starting(r);
10      forall p' ∈ L_{k+1} s.t. p ⊂ p' do
11        discarded(p') := discarded(p') ∪ {terminating(r)};
12    end
13  until L_{k+1} = ∅;
14  return R;
```

Procedure `InitializeStructures`;

```
IS1   forall p ∈ C(F) do
IS2     discarded(p) := { q | q ∈ C(F), p ∩ q ≠ ∅ };
```

Function `ComputeFrequentPatterns`(U: set of candidates): set of frequent patterns;

```
CFP1  return {r | r ∈ U, supp(r) > minSupp};
```

Function `UpdateCandidateList`(L_k: set of frequent patterns): set of candidate patterns;

```
UCL1  U := ∅
UCL2  forall p ∈ L_k                                  //***starting pattern
UCL3    forall q ∈ C' − discarded(p) do begin         //***terminating pattern
UCL4      r := p ∪ q; starting(r) = p; terminating(r) = q;
UCL5      discarded(r) := discarded(p) ∪ discarded(q);
UCL6      U := U ∪ {r};
UCL7    end;
UCL8  return U;
```

Fig. 5. Algorithm *ws-unconnected-find* ($\mathcal{WS}$, $\mathcal{F}$, *minSupp*, $C(\mathcal{F})$).

a_0, *i.e.*, $p' \supseteq p \cup \{a_0\}$, *such that* $p' \cup q$ *is frequent as well.*

In the light of this property, each frequent set of connected patterns can be generated from the frequent sets of connected patterns for which a pattern contains the starting activity. These kinds of frequent sets are called *starting* patterns in the following. And, consistently, *ws-unconnected-find* singles out the set of all the frequent starting patterns, thereby avoiding the computation of the subsets of the frequent starting patterns that do not contain a_0 (which, are in fact, necessarily frequent as well).

Frequent starting patterns are incrementally built by the algorithm by combining the frequent starting patterns already discovered in the computation. Interestingly, the combination can be carried out in a very efficient manner with a smart exploration of the space of all the connected starting patterns. To see how this is possible, consider two starting patterns r and p; we say that r directly precedes p, denoted by $r \prec p$, if there exist a connected pattern q such that $r = p \cup q$ and such that q does not contain the starting activity. Patterns that does not contain the starting activity are called *terminating* patterns. The above relation can be extended transitively: we say that r precedes p, denoted by $r \prec^* p$, if either $r \prec p$ or there exists a starting pattern q such that $r \prec^* q$ and $q \prec^* p$. It is not difficult to see that starting patterns can be constructed by means of a chain over the $\prec$ relation.

Thus, the search space of starting patterns forms a lower semilattice, and the bottom-up exploration of such a lattice constitutes the second key ingredient of *ws-unconnected-find*: at each step $k+1$, the algorithm generates the set of all the possible unconnected starting patterns made of $k+1$ distinct unconnected

patterns, by combining the patterns already generated and tested to be frequent (stored in L_k and made of k distinct unconnected starting patterns) with a terminating pattern. Let us now detail the different steps of *ws-unconnected-find*.

The algorithm starts by defining L_0 as the set of frequent patterns in $C(\mathscr{F})$ that contain a_0, and C' as the set of all the terminating connected patterns—notice that C' is, in fact, $C(\mathscr{F})$ minus the starting patterns in L_0.

Then, at each step the algorithm generates a number of candidates (stored in U) in the main cycle (steps 4–13). Each generated pattern p is obtained by means of the function *UpdateCandidateList*, by combining a starting pattern in L_k with a connected terminating pattern q in C' which is not in the set *discarded*(p). This latter set is used for optimization purposes. In fact, since we are interested in frequent unconnected components, given a pattern p we can compute in advance a set of patterns that must not be combined with p, denoted by *discarded*(p). Specifically, this set contains (i) all the patterns which have a non-empty intersection with p, and (ii) a set of patterns which merged with p are guaranteed to produce an unfrequent pattern. The set *discarded*(p) is initialized in the procedure *InitializeStructures*. Moreover, notice that each pattern r generated at the step k is also equipped with two functions, *starting*(r) and *terminating*(r), which store the starting and terminating patterns, respectively, that have been used for generating r.

After all the candidates have been computed in the set U, the function *ComputeFrequentPatterns* is invoked (step 6) for filtering the elements in U which frequently occur in $\mathscr{F}$, thus creating the set L_{k+1} containing all the frequent unconnected patterns made of $k+1$ connected patterns. This task is simply accomplished by means of a scan in the logs $\mathscr{F}$.

Finally, the generated starting frequent patterns are added to the actual result R, and in the steps 8–12 the set *discarded*(p') is updated in order to include patterns that combined with p' will certainly lead to an unfrequent pattern. Intuitively, if a pattern r is known to be unfrequent (i.e., it belongs to $U - L_{k+1}$), then each pattern $p' \supset starting(r)$ cannot be extend with *terminating*(r). This observation justifies the expression in step 11.

Example 7. Assume $minSupp = 30\%$ and consider again the set of instances depicted in Fig. 3. Then, the *ws-unconnected-find* algorithm initially considers the frequent patterns shown in Fig. 4. In particular, p_1, p_4 and p_8 are added to L_0, since they are starting patterns, whereas the remaining patterns are added to C'. The algorithm then performs three main iterations.

In the first iteration, at step 5, each pattern $p \in L_0$ is combined with a pattern in C' which is not in *discarded*(p). For example, p_1 is combined with p_2, p_3, p_5, p_6 and p_7. As a result, 12 candidate patterns are computed in U, and the execution of step 6 yields $L_1 = \{p_1 \cup p_2, p_1 \cup p_5, p_1 \cup p_6, p_1 \cup p_7\}$. In the second iteration, candidate patterns are obtained by combining each element in L_1. As a result, U contains eight new candidate patterns, of which $p_1 \cup p_2 \cup p_6$ and $p_1 \cup p_2 \cup p_7$ are inserted (as frequent) in L_2. In the last iteration, no new candidate patterns are obtained.

We conclude by stating a number of properties of the algorithm. Specifically, we shall prove its correctness and point out that its computational cost is, in fact, related to the number of unconnected components contained in frequent unconnected patterns—this number influences the number of scans to the log file as well.

Lemma 8. *The ws-unconnected-find algorithm on input* $\mathscr{WS} = \langle A, E, a_0, A_F \rangle$, *minSupp, and* $C(\mathscr{F})$ *is such that*

(1) *for each pattern* $r \in R$, $a_0 \in starting(r)$;
(2) *for each pattern* $r \in R$ *and for each pattern* $\ell \in discarded(r)$, *either*

- *there exists* $\bar{r} \in r$ *such that* $\bar{r} \cup \ell$ *is not an unconnected pattern, or*
- $r \cup \ell$ *is not frequent*;

(3) *for each pattern* $r \in R$ *and for each pattern* $\ell \in C'$ *such that there exists a pattern* $\bar{r} \in r$ *with* $\bar{r} \cup \ell$ *being connected,* ℓ *is in discarded*(r).

Proof. We prove the properties by structural induction on the number k of iterations in the main loop (steps 4–13). Clearly, (1) trivially holds for $k = 0$, since R coincides in fact with L_0. To see that (2) and (3) hold for $k = 0$ too, it is sufficient to consider how the set *discarded* is initialized in *InitializeStructures* for all the patterns in $C(\mathscr{F})$ and, therefore, for the starting patterns in L_0. In particular, for each $r \in R = L_0$, and for each $\ell \in discarded(r)$, we have that $r \cap \ell \neq \emptyset$ and, hence, $r \cup \ell$ is not unconnected.

Let us now assume that (1)–(3) hold at step k. We show that they hold in $k+1$ as well. In particular, it is sufficient to limit our attention on the patterns in L_{k+1} which are eventually added to R in the step 7, because for all the other patterns the properties are ensured by induction. Moreover, since L_{k+1} is obtained by just filtering out unfrequent patterns from the set U (see step 6), we have to focus on the steps that are performed in the function *UpdateCandidateList* on input L_k.

As for property (1), consider a pattern r added to U in step UCL6. Pattern r is the result of the union of p and q (see UCL4), that are, respectively, a pattern in L_k and a pattern in $C(\mathscr{F}) - L_0 - \mathit{discarded}(p)$. Notice that $\mathit{starting}(r)$ is set to p, while $\mathit{terminating}(r)$ is set to q. Then, by inductive hypothesis, $a_0 \in \mathit{starting}(p)$ holds.

As for property (2), observe that $\mathit{discarded}(r) = \mathit{discarded}(p) \cup \mathit{discarded}(q) \cup \{\ell' \mid \exists \widetilde{r} \in U - L_{k+1}$ s.t. $\ell' = \mathit{terminating}(\bar{r}) \wedge \mathit{starting}(\bar{r}) = r\}$, because of the steps 11 and UCL4. Then, consider an arbitrary pattern $\ell \in \mathit{discarded}(r)$. In the case where ℓ is in $\mathit{discarded}(p)$, by inductive hypothesis there exists $\bar{r} \in p$ such that either $\bar{r} \cup \ell$ is not an unconnected pattern or $\bar{r} \cup \ell$ is not frequent. Since p is a subset of r, it is the case that $\bar{r}$ belongs to r as well, and the property follows. A similar line of reasoning applies in the case where ℓ is in $\mathit{discarded}(q)$. Therefore, it remains to be considered the case where $\ell = \mathit{terminating}(\widetilde{r})$, for an unfrequent pattern $\widetilde{r}$ such that $\mathit{starting}(\widetilde{r}) = r$. In this case, the property follows because $r \cup \ell$ is a superset of $\widetilde{r}$ and must be, therefore, unfrequent as well.

Finally, as for property (3), consider a pattern $\ell \in C'$ such that there is $\bar{r} \in r$ with $\bar{r} \cup \ell$ being connected. We must show that ℓ is in $\mathit{discarded}(r)$. In the case where $\bar{r}$ belongs to p the property follows by induction. Moreover, in the case where $\bar{r} = q$, since q is in $C(\mathscr{F})$, the pattern ℓ is inserted in $\mathit{discarded}(q)$ by initialization and is subsequently added to $\mathit{discarded}(r)$ in step UCL4. □

Theorem 9. *The ws-unconnected-find algorithm on input $\mathscr{WS} = \langle A, E, a_0, A_F\rangle$, minSupp, and $C(\mathscr{F})$ terminates in at most $|A|$ scans in the log file, and computes the set R of all frequent starting patterns.*

Proof. It is easy to see that R contains only frequent starting patterns because of the property (1) in Lemma 8 and because of the test for the frequency performed in the function *ComputeFrequentPatterns*. Hence, we have to show that for each unconnected starting pattern p, it is the case that p is in R too. The proof is by structural induction on the number of the components in p (short: size of p). The base case where p consists of one (connected) pattern only, trivially holds since it must be the case that p is in $C(\mathscr{F})$. Therefore, let us assume that R contains the set of all the frequent starting patterns of size k and consider a frequent pattern p of size $k+1$. Since the space of all unconnected patterns is a lower semilattice, there exists a pattern $\bar{p}$ such that $\bar{p} \prec p$. This entails that p is the union of $\bar{p}$ (of size k) with some terminating pattern p_t. Moreover, since p is frequent, $\bar{p}$ is frequent as well and it has been, therefore, computed by the algorithm in the previous step (and inserted in L_k), by inductive hypothesis. Hence, in step $k+1$, the pattern p is eventually generated in UCL4: in fact, since $\bar{p} \cup p_t$ is a frequent unconnected pattern it must be the case that p_t belongs to $C' - \mathit{discarded}(p)$ (otherwise $\bar{p} \cup p_t$ is not a frequent unconnected pattern, by properties (2) and (3) in Lemma 8, thereby having a contradiction).

To conclude the proof, notice that the algorithm requires a scan in the log file each time the function *ComputeFrequentPatterns* is invoked. Therefore, the number of scans coincides with the total number of iterations (steps 4–13) that are in their turn bounded by the maximum number of components which an unconnected starting pattern can be made of. The size of A is, in fact, a trivial upper-bound for such a number. □

4. Optimizing candidate generation

The algorithm *ws-unconnected-find* takes already advantage of some important observations in order to perform a smart exploration of the search space. In this section, we show how to further exploit the peculiarities of the workflow graph, and specifically some statistics on the set of instances $\mathscr{F}$, in order to identify, before their actual testing w.r.t. the logs, those patterns which are necessarily (un)frequent. The resulting algorithm, called *ws*-unconnected-find*, is able to further prune the search space and to solve the FUPD problem in a more efficient way. Actually, in the description of this algorithm, we shall assume that the control-flow graph of the workflow schema at hand is acyclic, i.e., we do not consider repetitions and loops. However, this simplification has been just pursued for the sake of exposition and without loosing in generality, since dealing with cyclic graphs can be faced by means of some syntactic expedients which are discussed afterwards the algorithm is presented.

4.1. Overview of the approach

The key idea in the design of the *ws*-unconnected-find* algorithm is to combine the dynamic information obtained from the frequency of single nodes and edges, with the static information derived from the workflow schema in order to predict (un)frequent patterns. To get an intuition of the approach, consider again the schema in Fig. 2 and the following extreme scenario. Consider the activities a and p, and notice that they are frequent but not necessarily any path from a to p is frequent as well (this is what happens, e.g., by considering the instances of Fig. 3 and $minSupp = 30\%$). At the other hand, as every execution starting from a will eventually terminate in p, we can then conclude that the frequency of any pattern containing a is invariant if the pattern is extended with p. Therefore, we can conclude that nodes a and p form a frequent unconnected pattern without looking at the actual co-occurrences in the log file. Actually, many situations are less evident than the above trivial case. For instance, by analyzing both the instances and the graph structure (with the techniques we shall develop in the paper), we could conclude that m frequently occurs together with a, since a necessary condition for the execution of m is the execution of a. Incidentally, note also that m and b cannot co-occur frequently, since the only path connecting them is below the frequency threshold (and hence the frequency of m cannot be related to that of b).

The aim of this section is to systematically study the circumstances in which some analysis on the instances and the graph structure helps in pruning the search space. Specifically, we develop a graph-theoretic approach for predicting whether two patterns are coupled, that only looks at the workflow structure and at the frequency of the elementary activities alone.

In the following, we assume the existence of a set $\mathscr{F}$ of instances of a workflow schema $\mathscr{WS}$. Then, let q be a pattern of $\mathscr{WS} = \langle A, E, a_0, A_F\rangle$ which occurs in $f(q)$ instances of $\mathscr{F}$ and p be a pattern which occurs in $f(p)$ instances of $\mathscr{F}$ such that: q and p are unconnected; and, there is a path from some activity in p to some activity in q. Our aim is to compute as efficiently as possible the number of instances in $\mathscr{F}$ executing both the components p and q, denoted by $f_p(q)$. Clearly, the most trivial and inefficient way for computing $f_p(q)$ is to make a scan of the log $\mathscr{F}$. However, we shall show how some proper data structures and algorithms can be used for effectively identifying a suitable lower bound and an upper bound for $f_p(q)$, denoted by $l_p(q)$ and $u_p(q)$, respectively, in an efficient way which does not require the access to the log.

To this aim, it is worth noticing that in real-life scenarios, the execution of workflow activities is typically bound to specific constraints, which rule the relationship among the activities. Thus, we can exploit both the topology of the schema $\mathscr{WS}$ and the constraints issued over the nodes. Notice that, in this respect, it is not convenient to focus on some specific modelling language, since it would restrict the applicability of the proposed algorithms to some specific WFMSs. Rather, it makes sense to consider a simple language where the basic features which are common to most of the proposals in the literature are allowed, so that techniques can be developed which can be profitably integrated in different application scenarios. Consequently, to make our approach as much general as possible, we consider the very basic constructs of sequence, iteration, split and join which are common to most of the current workflow languages, as it emerged by recent comparative studies (e.g., [32]) based on workflow patterns [37,38]. Moreover, we do not present any specific executable semantics for constraints, since our aim is just to consider their "static" properties (recall the discussion in Section 2). Thus, as for the preconditions, we assume that activities in $A - \{a_0\}$ are partitioned in$A^{\vee}_{in} \cup A^{\wedge}_{in}$, such that for any instance $I = \langle A_I, E_I\rangle$, and activity a in A_I:

- if a_i is in $A^{\wedge}_{in}$, then $\{(a_j, a_i) \in E \mid a_j \in A\} \subseteq E_I$, i.e., a_i is an *and-join* that can be executed only after all its predecessors; and,
- if a_i is in $A^{\vee}_{in}$, then $|\{(a_j, a_i) \in E \mid a_j \in A\} \cap E_I| > 0$, i.e., a_i is an *or-join* that can be executed as soon as one of its predecessors is. Note that this requirement is fairly trivial and it must hold no matter of the specific semantics used by the system to solve the problem with the vicious cycle (cf. [33]).

As for post-conditions, activities in $A - A_F$ are partitioned in $A^{\wedge,\vee}_{out} \cup A^{\otimes}_{out}$, such that for any instance $I = \langle A_I, E_I\rangle$, and activity a in A_I:

- if a_i is in $A^{\otimes}_{out}$, then $|\{(a_i, a_j) \in E \mid a_j \in A\} \cap E_I| \leqslant 1$, i.e., a_i is an *exclusive-fork* that enables at most one outgoing activity; and,
- if a_i is in $A^{\wedge,\vee}_{out}$, then $|\{(a_i, a_j) \in E \mid a_j \in A\} \cap E_I\}| \geqslant 0$, i.e., a_i enables some of the outgoing activities. Note that this is in fact not a constraint.

Example 10. Consider again the control-flow structure of Fig. 2(a). In the following, we assume that constraints on this graph are such that: $A^{\wedge}_{in} = \{\mathtt{d},\mathtt{c},\mathtt{b},\mathtt{f}\}$ and $A^{\vee}_{in} = \{\mathtt{e},\mathtt{g},\mathtt{l},\mathtt{i},\mathtt{h},\mathtt{m},\mathtt{n},\mathtt{o},\mathtt{p}\}$, while $A^{\otimes}_{out} = \{\mathtt{a},\mathtt{h}\}$, $A^{\wedge,\vee}_{out} = \{\mathtt{f},\mathtt{l},\mathtt{i},\mathtt{g},\mathtt{d},\mathtt{m},\mathtt{n},\mathtt{o},\mathtt{b},\mathtt{c},\mathtt{e}\}$,and $A_F = \{\mathtt{p},\mathtt{q}\}$. The reader may check that instances in Fig. 3 conform with these constraints.

4.2. Computing frequency bounds for activities

Let us start investigating how frequency bounds can be efficiently computed in the case where patters are made of single activities of $\mathscr{WS}$. Given an activity $a \in A$, let G_a be the subgraph of the control flow of $\mathscr{WS}$ induced by all the nodes b such that there is a path from b to a in $\mathscr{WS}$. Note that all such nodes can be easily determined by reversing the edges in $\mathscr{WS}$ and computing the transitive closure of a. Moreover, the graph obtained from G_a by reversing all its edges is denoted by $\overleftarrow{G}_a$.

The starting point of our approach is to compute for each node b in G_a, the number of instances in $\mathscr{F}$ executing both the activities a and b, denoted by $f_a(b)$. Actually, as already pointed out, we turn to the computation of a lower bound $l_a(b)$ and of an upper bound $u_a(b)$ for $f_a(b)$.

In order to accomplish this task we need some auxiliary data structures besides the workflow schema which are used for storing the occurrences of each activity and edge (connecting activities) in the log $\mathscr{F}$.

Definition 11 (*Frequency graph*). Let $\langle A, E\rangle$ be the control flow of a workflow schema $\mathscr{WS}$ and let $\mathscr{F}$ be a set of instances of $\mathscr{WS}$. The *frequency graph* $\mathscr{WS}_{\mathscr{F}} = \langle A, E, f_A, f_E\rangle$ is a weighted graph such that:

- $f_A : A \mapsto \mathbf{N}$ maps each activity a to the number of instances in $\mathscr{F}$ executing it, and
- $f_E : E \mapsto \mathbf{N}$ maps each edge e to the number of instances in $\mathscr{F}$ containing this edge.

Whenever no confusion arises, given an activity $a \in A$ (resp. an edge $e \in E$), $f_A(a)$ (resp. $f_E(e)$) will be simply denoted by $f(a)$ (resp. $f(e)$).

Fig. 6 shows the frequency graph associated with the schema of Fig. 2, built by taking into account the set of instances described in Fig. 3.

In order to derive the aforementioned bounds, we first determine a topological sort $\langle a = b_1, b_2, \ldots, b_k\rangle$ of the nodes in $\overleftarrow{G}_a$. Notice that as $\mathscr{WS}$ is acyclic, $\overleftarrow{G}_a$ is also acyclic and a topological sort exists for it. Then we proceed as shown in Fig. 7.

In the step 1, the lower and upper bounds of the activity a are fixed to the known value $f(a)$. Then, each node b_i in G_a, is processed according to the topological sort of $\overleftarrow{G}_a$. In step 3, the set of all the activities $C(b_i)$ that can be reached by means of an edge starting in b_i and that are in G_a is computed. Step 4 is responsible for computing the upper bound $u_a(b_i)$, whereas steps 5–9 are responsible for computing the lower bound $l_a(b_i)$.

Intuitively, the upper bound $u_a(b_i)$ can be computed by optimistically assuming that each edge outgoing from b_i is in some path reaching a. This justifies the formula of step 4.

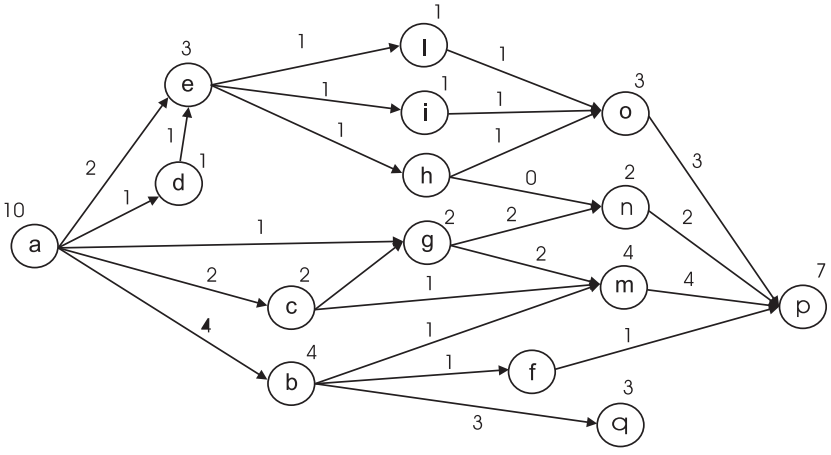


Fig. 6. Frequency graph associated with the schema of Fig. 2.

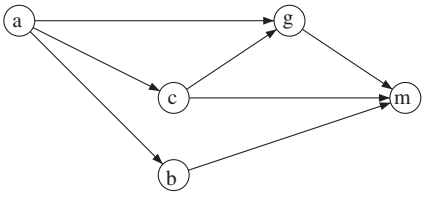
Input: An annotated workflow schema $\mathcal{WS}_{\mathcal{F}} = \langle A, E, f_A, f_E \rangle$, an activity a, the graph G_a and a topological sort $\langle a = b_1, b_2, \ldots, b_k \rangle$ of the nodes in $\overleftarrow{G}_a$.
Output: for each node $b \in G_a$, the values $l_a(b)$ and $u_a(b)$.
Method: Perform the following steps:

1 $l_a(a) := f(a)$; $u_a(a) := f(a)$;
2 **forall** $i = 2..k$ **do begin**
3 $C(b_i) := \{b \mid (b_i, b) \in E \wedge b \in G_a\}$;
4 $u_a(b_i) := \min(f(b_i), f(a), U)$, with $U = \sum_{e=(b_i,c) \mid c \in C(b_i)} \min(f(e), u_a(c))$.
5 **if** $b_i \in A^{\wedge,\vee}_{out}$ **then**
6 $l_a(b_i) := \max(L^{\wedge}_1, L^{\vee}_1)$, with
$L^{\wedge}_1 = \max_{c_j \in C(b_i) \cap A^{\wedge}_{in}} \{l_a(c_j)\}$, and
$L^{\vee}_1 = \max_{c_j \in C(b_i) \cap A^{\vee}_{in}} \{\max(0, l_a(c_j) - \sum_{e=(d,c_j) \in E \wedge d \neq b_i} f(e))\}$
7 **else** // *case of* $b_i \in A^{\otimes}_{out}$
8 $l_a(b_i) := L^{\wedge}_2 + L^{\vee}_2$, with
$L^{\wedge}_2 = \sum_{c_j \in C(b_i) \cap A^{\wedge}_{in}} \{l_a(c_j)\}$, and
$L^{\vee}_2 = \sum_{c_j \in C(b_i) \cap A^{\vee}_{in}} \{\max(0, l_a(c_j) - \sum_{e=(d,c_j) \in E \wedge d \neq b_i} f(e))\}$
9 **end**
10 **end**

Fig. 7. The *compute_frequency_bounds*($\mathcal{WS}_{\mathcal{F}}, a$) algorithm.

Concerning $l_a(b_i)$, observe that each node $c_j \in C(b_i)$ is executed with a by at least $l_a(c_j)$ instances. Therefore, we need to know how many of the instances executing b_i contribute to $l_a(c_j)$. Two cases arise: (i) $b_i \in A^{\wedge,\vee}_{out}$, so the nodes connected to b_i may occur simultaneously within an instance, and (ii) $b_i \in A^{\otimes}_{out}$, then all c_j are executed exclusively from each other. This explains why in the first alternative $L^{\wedge}_1$ and $L^{\vee}_1$ are computed by maximizing the contribution of each c_j, whereas in the second alternative the single contributions are summated. Finally, observe that when $c_j \in A^{\vee}_{in}$, it may be not the case that all of the $l_a(c_j)$ instances execute b_i, thus requiring to differentiate the formulas for $L^{\vee}_1$ and $L^{\wedge}_1$ (and, in the same way, for $L^{\vee}_2$ and $L^{\wedge}_2$).

Example 12. Let us consider the graph $G_{\mathtt{m}}$ induced by node m:



A topological sort for $\overleftarrow{G}_{\mathtt{m}}$ is $\langle \mathtt{m}, \mathtt{g}, \mathtt{b}, \mathtt{c}, \mathtt{a} \rangle$. Then, we leave the careful reader to check that by applying *compute_frequency_bounds*($\mathcal{WS}_{\mathcal{F}}, \mathtt{m}$), the following bounds for node m can be obtained:

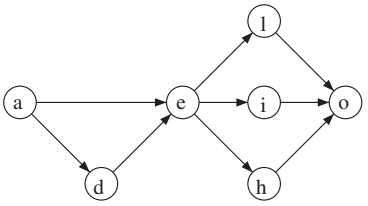
$$l_{\mathtt{m}}(\mathtt{g}) = 2, \quad u_{\mathtt{m}}(\mathtt{g}) = 2, \quad l_{\mathtt{m}}(\mathtt{b}) = 1, \quad u_{\mathtt{m}}(\mathtt{b}) = 1,$$

$$l_{\mathtt{m}}(\mathtt{c}) = 1, \quad u_{\mathtt{m}}(\mathtt{c}) = 2, \quad l_{\mathtt{m}}(\mathtt{a}) = 3, \quad u_{\mathtt{m}}(\mathtt{a}) = 4.$$

Before proceeding with the exposition, we want to provide some intuition on how these bounds may be practically used for pruning the search space.

Consider, for instance, the lower bound $l_{\mathtt{m}}(\mathtt{a}) = 3$. This entails that pattern $\{\mathtt{m}\}$ frequently (w.r.t. $minSupp = 30\%$) occurs with pattern $\{\mathtt{a}\}$. Therefore, we can conclude that $\{\mathtt{m}\} \cup \{\mathtt{a}\}$ is a frequent unconnected pattern without testing for its frequency over the set of the instances. Conversely, given the bound $u_{\mathtt{m}}(\mathtt{b}) = 1$ we can conclude that $\{\mathtt{m}\} \cup \{\mathtt{b}\}$ has no chance of being frequent, and therefore it can be pruned from the search space.

As an example scenario in which bounds cannot be instead used for pruning the search space, let us conclude by analyzing also the bounds for node o. Consider the dependency graph $G_{\mathtt{o}}$, which is depicted below:



Then, by applying the algorithm, we obtain $u_{\mathtt{o}}(\mathtt{a}) = u_{\mathtt{o}}(\mathtt{e}) = 3$, $u_{\mathtt{o}}(x) = 1$ for $x \in \{\mathtt{d}, \mathtt{l}, \mathtt{i}, \mathtt{h}\}$, and $l_{\mathtt{o}}(y) = 1$ for each node y in $G_{\mathtt{o}}$. Thus, even though $\{\mathtt{e}\} \cup \{\mathtt{o}\}$ is a frequent unconnected pattern, lower bounds do not help in detecting such pattern without resorting to the logs. This is essentially due to the fact that, since $\mathtt{e} \in A^{\vee}_{out}$, its lower bound depends from the lower bounds of h, i and l (each of which belongs to $G_{\mathtt{o}}$, with frequency 1).

We conclude the description of the algorithm for computing bounds between pairs of activities by discussing its main properties.

Theorem 13. *The following properties hold for the algorithm in Fig. 7:*

(1) *The parameters* U, $L_1^{\vee}$, $L_2^{\vee}$, $L_1^{\wedge}$ *and* $L_2^{\wedge}$ *are well defined, i.e.,* $u_a(b_i)$ *and* $l_a(b_i)$ *are computed by exploiting already processed values.*
(2) *For each node* $b_i \in G_i$, *the values* $l_a(b_i)$ *and* $u_a(b_i)$ *are, respectively, a lower and an upper bound for the frequency* $f_a(b_i)$.
(3) *The procedure can be computed in time* $O(|G_a|^2)$.

Proof. (1) We show that the property hold by structural induction on the processing of the topological sort $\langle a = b_1, \ldots, b_k\rangle$ for $\overleftarrow{G}_a$. For the base case, i.e., $i = 1$ and $b_1 = a$, the values of $u_a(a)$ and $l_a(a)$ are both set to the actual value $f(a)$. Then, assume that all the values are correctly computed for nodes $b_1, \ldots, b_{i-1}$ with $i \leqslant 2$, and assume that the execution of the algorithm is at the ith iteration (loop in steps 2–10). We have to show that the formulas for computing $u_a(b_i)$ and $l_a(b_i)$ are well defined. Actually, by careful looking at expressions in steps 4, 6, and 8, we can see that both $u_a(b_i)$ and $l_a(b_i)$ depends only on the values of $u_a(c)$ and $l_a(c)$ for some node $c \in C(b_i)$. By construction in step 3, notice that $C(b_i)$ contains only nodes in G_a that are reachable by means of an edge by b_i. Therefore, by construction of the topological sort $\langle b_1, \ldots, b_k\rangle$, any node in $C(b_i)$ is in the topological sort for some index $h < i$. Thus, nodes in $C(b_i)$ have been already processed and the values for the upper bound and the lower bound have been correctly computed, because of the inductive hypothesis.

(2) As before, we proceed by structural induction. In the case $i = 1$, the values of $u_a(a)$ and $l_a(a)$ coincide with the actual value $f(a)$. Then, consider the ith iteration of the algorithm and assume that $l_a(b_j) \leqslant f_a(b_j) \leqslant u_a(b_j)$, for each $1 \leqslant j < i$.

Let us first consider the value of $u_a(b_i)$ computed in step 4 of the algorithm and show that $f_a(b_i) \leqslant u_a(b_i)$. Assume, for the sake of contradiction that $u_a(b_i) < f_a(b_i)$. Then, by looking at the expression of $u_a(b_i)$, there are only three possible scenarios:

- $u_a(b_i) = f(b_i)$, which is impossible since we would have $f(b_i) < f_a(b_i)$;
- $u_a(b_i) = f(a)$, which is impossible since we would have $f(a) < f_a(b_i)$;
- $u_a(b_i) = U$, with $U = \sum_{e=(b_i,c)|c\in C(b_i)} \min(f(e), u_a(c))$. In this case, by inductive hypothesis, we have that $u_a(c) \geqslant f_a(c)$ (notice that c is in $C(b_i)$, and recall from the point (1) above that nodes in $C(b_i)$ have been already processed). Therefore, we have that $f_a(b_i) > u_a\ (b_i) = U \geqslant \sum_{e=(b_i,c)|c\in C(b_i)} \min(f(e), f_a(c))$. But this is a contradiction. Indeed, each time an instance executes both a and b_i, thereby contributing to the frequency $f_a(b_i)$, it is the case that there exists a node c such that (b_i, c) is executed as well. By construction, c belongs to$C(b_i)$ and therefore the execution of c contributes to both the frequencies $f((b_i, c))$ and $f_a(c)$. It follows that the value of $f_a(b_i)$ is bounded by $\sum_{e=(b_i,c)|c\in C(b_i)} \min(f(e), f_a(c))$.

Let us now turn to evaluate the correctness of the value of $l_a(b_i)$. In this case, we have to distinguish whether b_i is in $A_{out}^{\wedge,\vee}$, or is in $A_{out}^{\otimes}$. We shall consider the former case only, since the latter scenario can be proven by using similar arguments. Assume, for the sake of contradiction, that $l_a(b_i) > f_a(b_i)$. Then, by looking at the expression of $l_a(b_i)$, there are only two scenarios:

- $l_a(b_i) = L_1^{\wedge} = \max_{c_j\in C(b_i)\cap A_{in}^{\wedge}} \{l_a(c_j)\}$. By inductive hypothesis, since c_j is in $C(b_i)$, we have that $l_a(c_j) \leqslant f_a(c_j)$, for each c_j in the expression above. Therefore, we can write $f_a(b_i) < l_a(b_i) = l_a(\bar{c}) \leqslant f_a(\bar{c})$, where $\bar{c}$ is a node in $C(b_i)$ and, hence, such that $(b_i, \bar{c})$ is an edge in the control flow of $\mathscr{WS}$. However, it is impossible that $f_a(b_i) < f_a(\bar{c})$ holds for such pairs of connected nodes, because $\bar{c}$ is a $A_{in}^{\wedge}$ node and, so, any instance activating $\bar{c}$ has to activate b_i as well.
- $l_a(b_i) = L_1^{\vee}$. Then,

$$l_a(b_i) = \max_{c_j\in C(b_i)\cap A_{in}^{\vee}} \left\{ \max\left(0, l_a(c_j) - \sum_{e=(d,c_j)\in E\wedge d\neq b_i} f(e)\right)\right\}.$$

In this case, we can assume, w.l.o.g, that $l_a(b_i) \neq 0$ (otherwise we get an immediate contradiction with $f_a(b_i)$ being greater or equal than 0). Therefore, we have $f_a(b_i) < l_a(b_i) = l_a(\bar{c}) - \sum_{e=(d,\bar{c})\in E\wedge d\neq b_i} f(e)$, for a suitable node $\bar{c}$ in $C(b_i) \cap A_{in}^{\vee}$. Again, $\bar{c}$ has been already processed and it holds that $l_a(\bar{c}) \leqslant f_a(\bar{c})$. Then, we have: $f_a(b_i) + \sum_{e=(d,\bar{c})\in E\wedge d\neq b_i} f(e) < l_a(\bar{c}) \leqslant f_a(\bar{c})$. But this is a contradiction. Indeed, since $\bar{c}$ is in $A_{in}^{\vee}$, each time an instance activates $\bar{c}$ and does not activate b_i (i.e., the edge $(b_i, \bar{c})$ is not being exploited), it must be the case that an edge in

the set $\{(d, c_j) \in E \wedge d \neq b_i\}$ is activated as well. Therefore, each time there is a contribution to $f_a(b_i) - f_a(\bar{c})$, there is a contribution as well for $f(e)$ for some edge $e \in \{(d, c_j) \in E \wedge d \neq b_i\}$.

(3) Notice that the main loop of the algorithm is executed k times, where k is in fact the size of the graph G_a. Then, at each step i, the computation of the values $u_a(b_i)$ and $l_a(b_i)$ just requires elementary calculations and selection of minimum and maximum values in set of $|C(b_i)|$ elements. Since, $|C(b_i)| \leqslant |G_a|$, for each i, the quadratic bound easily follows. □

4.3. Computing frequency bounds for patterns

Let us now turn to the slightly more general problem of approximating the value of $f_p(b)$, for any pattern p and any activity b, by means of suitable lower and upper bounds. Notice that the value $f_p(b)$ is the number of instances in $\mathscr{F}$ executing both the pattern p and an activity b that precedes one of the activities in p. To this aim, we simply reuse the technique described in the previous section with some adaptations. Let us first introduce some auxiliary notation.

Definition 14. Let $\mathscr{WS}$ be a workflows schema. For each connected pattern $p = \langle A_p, E_p \rangle$ on $\mathscr{WS}$, define:

$$\texttt{INBORDER}(p) = \{a \in A_p \mid \nexists b \in A_p : (b, a) \in E_p, \exists c \in \mathscr{WS} - p : (c, a) \in \mathscr{WS}\}$$

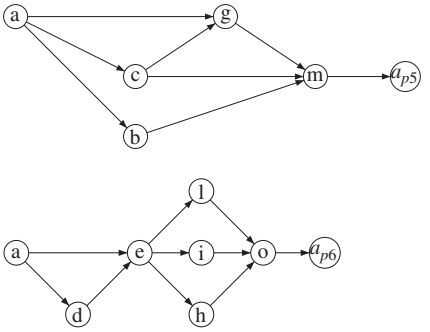
and

$$\texttt{OUTBORDER}(p) = \{a \in A_p \mid \exists c \in \mathscr{WS} - p : (a, c) \in \mathscr{WS}\}.$$

Let $\mathscr{WS}(p)$ be the workflow schema derived from $\mathscr{WS}$ by adding a new and-join node, say a_p, corresponding to the pattern p, and by adding an edge from each node b in p to a_p. In the frequency graph of $\mathscr{WS}(p)$ set $f(a_p) = f(p)$, and $f(e) = f(p)$ for each $e = (b, a_p) \in E$. Moreover, given an activity a, we denote by $G(p)_a$ be the subgraph of the control flow of $\mathscr{WS}(p)$ induced by all the nodes b such that there is a path from b to a in $\mathscr{WS}$, and we denote by $\overleftarrow{G}(p)_a$ the graph obtained from $G(p)_a$ by reversing all its edges.

Then, the computation of the bounds for the co-occurrence of any activity b with the patter p is carried out by means of the function *compute_frequency_bounds*$(\mathscr{WS}(p)_{\mathscr{F}}, a_p)$. This function will be simply denoted in the follows as *compute_frequency_bounds*$(\mathscr{WS}_{\mathscr{F}}, p)$.

Example 15. Let us consider the patterns p_5 and p_6 of Fig. 4. According to the workflow schema shown in Fig. 2, $\texttt{INBORDER}(p_5) = \{\texttt{m}\}$ and $\texttt{INBORDER}(p_6) = \{\texttt{o}\}$. By adding the dummy nodes a_{p_5} and a_{p_6}, we obtain the following induced subgraphs:



By construction, we have $f(a_{p_5}) = f(p_5) = 4$ and $f(a_{p_6}) = f(p_6) = 3$. Moreover, given that $|\texttt{INBORDER}(p_5)| = |\texttt{INBORDER}(p_6)| = 1$, it is not difficult to check that lower bounds and upper bounds for co-occurrences for p_5 and p_6 with other activities coincide with those computed when the co-occurrences for activities m and o are considered. That is, by means of the invocation of *compute_frequency_bounds*$(\mathscr{WS}(p)_{\mathscr{F}}, a_{p_5})$, we obtain $u_{p_5}(x) = u_{\texttt{m}}(x)$ and $l_{p_5}(x) = l_{\texttt{m}}(x)$ for each $x \in G(p_5)_{a_{p_5}}$; whereas, by means of *compute_frequency_bounds*$(\mathscr{WS}(p)_{\mathscr{F}}, a_{p_6})$ we get $u_{p_6}(x) = u_{\texttt{o}}(x)$ and $l_{p_6}(x) = l_{\texttt{o}}(x)$ for each $x \in G(p_6)_{a_{p_6}}$.

The correctness of the approach is stated by the following result.

Theorem 16. *Let $\mathscr{WS}$ be a workflow schema, $\mathscr{F}$ be a set of instances, and be p a pattern. For any activity b, let $l_{a_p}(b)$ and $u_{a_p}(b)$ be the lower and upper bound of the occurrence of activity a_p together with b, computed by means of the algorithm compute_frequency_bounds$(\mathscr{WS}(p)_{\mathscr{F}}, a_p)$. Then, $l_{a_p}(b)$ and $u_{a_p}(b)$ are indeed lower and upper bounds for $f_p(b)$.*

Proof. We show by structural induction on the topological sort $\langle b_0 = a_p, b_1, \ldots, b_n \rangle$ of nodes in $\overleftarrow{G}(p)_{a_p}$ that, for each $b_i \in G_{a_p}$, it holds that $l_{a_p}(b_i) \leqslant f_p(b_i) \leqslant u_{a_p}(b_i)$. For the base case $b_0 = a_p$, notice that a_p is an and-join node reached by each node $b \in \texttt{INBORDER}(p)$. Hence, b_0 is activated each time all the elements in $\texttt{INBORDER}(p)$ are activated. Since each instance I including p must activate each node in $\texttt{INBORDER}(p)$, I also activates a_p. Hence, $f_p(a_p) = f(a_p)$. The thesis holds by observing that, by construction, $l_{a_p}(a_p) = u_{a_p}(a_p) = f(a_p) = f(b_0)$.

For the inductive case, assume that $l_{a_p}(b_j) \leqslant f_p(b_j) \leqslant u_{a_p}(b_j)$ for $0 \leqslant j < i$. We prove now that $f_p(b_i) \leqslant u_{a_p}(b_i)$ and $f_p(b_i) \geqslant l_{a_p}(b_i)$. In order to prove that $f_p(b_j) \leqslant u_{a_p}(b_j)$, since $f_p(b_i)$ is trivially bounded by $f(b_i)$, it only remains to show that $U \geqslant f_p(b_i)$—see, again, the bounds as for they appear in Fig. 7. By inductive hypothesis, $U \geqslant \sum_{e=(b_i,c)|c\in C(b_i)} \min(f(e), f_p(c))$. Then, the thesis holds by observing that for each instance I including both b_i and p there exists $c \in C(b_i)$ such that both c and (b_i, c) are included in I: in such a case, I contributes to $f_p(c)$, and hence $\sum_{e=(b_i,c)|c\in C(b_i)} \min(f(e), f_p(c)) \geqslant f_p(b_i)$.

Finally, in order to prove that $l_{a_p}(b_i) \leqslant f_p(b_i)$, two cases have to be distinguished:

- $b_i \in A^{\wedge,\vee}_{out}$. We show that $f_p(b_i) \geqslant \max(L^{\wedge}_1, L^{\vee}_1)$. Indeed, if $l_{a_p}(b_i) = L^{\wedge}_1$, then there exists $\overline{c} \in C(b_i) \cap A^{\wedge}_{in}$ such that $l_{a_p}(b_i) = l_{a_p}(\overline{c})$. By inductive hypothesis, $l_{a_p}(b_i) = l_{a_p}(\overline{c}) \leqslant f_p(\overline{c})$. But $f_p(\overline{c}) \leqslant f_p(b_i)$, since $\overline{c} \in A^{\wedge}_{in}$ and consequently each instance executing $\overline{c}$ must also execute b_i. If, by the converse, $l_{a_p}(b_i) = L^{\vee}_1$, then there exists $\overline{c} \in C(b_i) \cap A^{\vee}_{in}$ such that $l_{a_p}(b_i) = l_{a_p}(\overline{c}) - \sum_{e=(d,\overline{c})\in E \wedge d \neq b_i} f(e)$. By inductive hypothesis, $l_{a_p}(\overline{c}) \leqslant f_p(\overline{c})$, and consequently $l_{a_p}(b_i) \leqslant f_p(\overline{c}) - \sum_{e=(d,\overline{c})\in E \wedge d \neq b_i} f(e)$. The thesis holds by observing that, since $\overline{c} \in A^{\vee}_{in}$, each instance passing from both $\overline{c}$ and p either passes from b_i or from some other node d as a consequence, $f_p(\overline{c}) - \sum_{e=(d,\overline{c})\in E \wedge d \neq b_i} f(e) \leqslant f_p(b_i)$.
- $b_i \in A^{\otimes}_{out}$. Again, we show that $f_p(b_i) \geqslant L^{\wedge}_2 + L^{\vee}_2$. Indeed, each instance traversing both b_i and p must traverse exactly one $b_j \in C(b_i)$. As a consequence, $f_p(b_i) = \sum_{b_j \in C(b_i) \cap A^{\wedge}_{in}} f_p(b_j) + \sum_{b_j \in C(b_i) \cap A^{\vee}_{in}} f_j$ where, for a given $b_j \in C(b_i) \cap A^{\vee}_{in}$, the term f_j is the number of instances traversing b_i, b_j and p. Notice that $f_j \geqslant 0$ and f_j must count at least $f_p(b_j) - \sum_{e=(d,b_j)\in E \wedge d \neq b_i} f(e)$ instances: indeed, the latter is the number of instances traversing both b_j and p that cannot enable other ancestors of b_j than b_i. The thesis finally holds by observing that, by inductive hypothesis, $L^{\wedge}_2 \leqslant \sum_{b_j \in C(b_i) \cap A^{\wedge}_{in}} f_p(b_j)$ and $L^{\vee}_2 \leqslant \sum_{b_j \in C} (b_i) \cap A^{\vee}_{in} \{\max(0, f_p(b_j) - \sum_{e=(d,b_j)\in E \wedge d \neq b_i} f(e))\}$. □

4.4. Algorithm ws*-unconnected-find

After that the frequency bounds for a given pattern (w.r.t. any activity) are computed, we can face the most general problem. Let q be a pattern of $\mathscr{WS}$ with frequency $f(q)$ and p be a pattern with frequency $f(p)$ such that q and p are unconnected. A lower bound and an upper bound for $f_p(q)$ are as follows:

- $l_p(q) = \max(0, \max_{b\in q} \{l_p(b) - (f(b) - f(q))\}, f(q) - ! \sum_{b\in q}(f(q) - l_p(b)))$
- $u_p(q) = \min(f(q), \min_{b\in \texttt{OUTBORDER}(q)} \{u_p(b)\}, f(q) + \sum_{b\in q} (f(q) - l_p(b)))$.

Here, recall that $\texttt{OUTBORDER}(p)$ refers to all the nodes in q having outgoing edges in $\mathscr{WS} - q$. The intuition behind the above formulas is the following. The value $u_p(q)$ is obtained by taking into account the contribution of each node b of q from which there is a path to a node in p. However we may exclude from the computation all internal nodes of q (i.e., those not in $\texttt{OUTBORDER}(p)$) as they are always executed together with at least one node in $\texttt{OUTBORDER}(p)$. Concerning the computation of $l_p(q)$, observe that there are at least $l_p(b)$ instances executing $b \in q$ and p. So, as $f(b) \geqslant f(q)$, there are at least $l_p(b) - (f(b) - f(q))$ instances connecting q and p and executing b. It turns out that a suitable lower bound is provided by the node exhibiting the maximum such value.

Example 17. Let us consider again the patterns p_4 and p_5 of Fig. 4. According to the above formulas and by considering the bounds shown in Example 15, we obtain $l_{p_5}(p_4) = 1$ and $u_{p_5}(p_4) = 2$.

The correctness of the proposed bounds is stated by the following theorem.

Theorem 18. *Let $\mathscr{WS}$ be a workflow schema, $\mathscr{F}$ be a set of instances, and p and q two patterns. Then, $l_p(q)$ and $u_p(q)$ are lower and upper bounds of $f_p(q)$.*

Proof. Let us first consider the expression for the lower bound $l_p(q)$. We first show that $f_p(q) \geqslant \max_{b\in q} \{l_p(b) - (f(b) - f(q))\}$. For $b \in q$, let us denote by $f_p(b \mid \neg q)$ the number of activities executing both b and p, but not the entire q. Clearly, the identity $f_p(b) = f_p(q) + f_p(b \mid \neg q)$ by definition. Moreover, given that b is contained in q it follows that $f_p(b \mid \neg q) \leqslant f(b) - f(q)$, where the term on the right of the inequality is the number of instances executing b but not the entire q. As a consequence, $f_p(b) \leqslant f_p(q) + (f(b) - f(q))$. By Theorem 16, $f_p(b) \geqslant l_p(b)$; thus, we obtain by simple algebraic manipulations that $l_p(b) - (f(b) - f(q)) \leqslant f_p(q)$. Finally, since the inequality holds for each $b \in q$, we obtain $f_p(q) \geqslant \max_{b\in q} \{l_p(b) - (f(b) - f(q))\}$.

We now show that $f_p(b) \geqslant f(q) - \sum_{b\in q}(f(q) - l_p(b))$. Indeed, notice that each term of the form

$(f(q) - l_p(b))$ with $b \in q$ is an upper bound on the instances that execute q and for which the node b is not executed with p. Therefore, $\sum_{b \in q}(f(q) - l_p(b))$ is an upper bound on the number of instance executing q and not p; it follows that subtracting such a value from $f(q)$ (total number of instances executing q), we get an admissible lower bound for $f_p(q)$.

Let us now consider the expression for the upper bound $u_p(q)$. Assume again, for the sake of contradiction, that $u_p(q) < f_p(q)$. There are three possible scenarios only:

(1) $u_p(q) = f(p)$, which is impossible since we would have $f(p) < f_p(q)$;
(2) $u_p(q) = \min_{b \in \texttt{OUTBORDER}(q)}\{u_p(b)\}$. In this case, there exists a node $\bar{b}$ in $\texttt{OUTBORDER}(q)$ such that $u_p(q) = u_p(\bar{b}) < f_p(q)$. Due to the correctness of the expression for $u_p(\bar{b})$, we have that: $f_p(\bar{b}) \leqslant u_p(\bar{b}) < f_p(q)$, which is impossible in its turn, since each instance executing q activates $\bar{b}$ by definition.
(3) $u_p(q) = f(q) + \sum_{b \in q}(f(q) - l_p(b))$. In this case, observe that, since $l_p(q) \geqslant f(q) - \sum_{b \in q}(f(q) - l_p(b))$, we have $2f(q) - l_p(q) \leqslant u_p(q)$. Recalling that $u_p(q) < f_p(q)$ by hypothesis, we obtain $2f(q) - l_p(q) < f_p(q)$. Recall now that $l_p(q) \leqslant f_p(q)$, because of the first part of the proof. Therefore, the above expression can be rewritten as $2f(q) < 2f_p(q)$. But this is a contradiction, since each instance executing both q and p clearly contributes to $f(q)$ (so that $f_p(q) < f(q)$). □

Generalized upper and lower bounds can be finally used for pruning the search space of the *ws-unconnected-find* algorithm. In fact, if for any two patterns p and q such that $u_p(q) < minSupp$ holds, then it is always the case that p and q never occur frequently together. Conversely, if $l_p(q) \geqslant minSupp$ then p and q can be combined into a pattern that is frequent as well.

Thus, the algorithm *ws-unconnected-find* may be optimized by incorporating the aforementioned ideas in the algorithm *ws*-unconnected-find*, by suitably adapting the procedures *InitializeStructures*, UpdateCandidateList and ComputeFrequentPatterns as shown in Fig. 8.

Specifically, InitializeStructures computes the frequency graph and all the frequency bounds for any pattern, by exploiting the *compute_frequency_bounds* algorithm. Procedure *ComputeFrequentPatterns* optimizes the computation of frequent patterns: it exploits frequency bounds for statically detecting frequent patterns in LF and unfrequent ones in LU, and verifies the frequency in the log $\mathscr{F}$ only for patterns which cannot be tested with the frequency bounds only (that is, all patterns in $U - (LF \cup LU)$.

Finally, the *UpdateCandidateList* procedure recomputes the set U of candidate patterns, by additionally computing for each newly generated candidate $r = p \cup q$ the values $u_q(p)$ and $l_q(p)$, to be exploited in the next stage for statically pruning the list of candidates.

Example 19. Recall from Example 7 that $L_0 = \{p_1, p_4, p_8\}$ and $C' = \{p_2, p_3, p_5, p_6, p_7\}$. Also, recall that the *ws-unconnected-find* algorithm initially generates 12 candidates (cf. Example 7). Then, it can be observed that in the first iteration of the *ws*-unconnected-find* algorithm, the optimized ComputeFrequentPatterns procedure of Fig. 8 removes from the set U of candidates the unfrequent patterns $LU = \{p_4 \cup p_2, p_4 \cup p_5, p_4 \cup p_6, p_4 \cup p_7, p_8 \cup p_2, p_8 \cup p_5, p_8 \cup p_6, p_8 \cup p_7\}$, and the frequent ones $LF = \{p_1 \cup p_2, p_1 \cup p_5\}$.

4.5. Dealing with loops

The *ws*-unconnected-find* algorithm depends on the assumption that the control-flow graph is acyclic: indeed, the proofs of the correctness results are based on the fact that for any activity a, the graph G_a does not contain a itself. As repetitions and loops often occur in real-life workflows, it is relevant to show that the scope of *ws*-unconnected-find* is not actually limited as the algorithm can be adapted to deal with cyclic control flows. The basic idea for performing such an extension is to unfold cycles within the instances of execution as informally discussed below.

Let $\mathscr{T} = \{t_1, \ldots, t_n\}$ be a log for a cyclic workflow schema $\mathscr{WS} = \langle A, E, a_0, F\rangle$. Let us assume for simplicity that there exists only one cycle and it has the format of a circuit, say $[a_1, a_2, \ldots, a_k, a_1]$—call such activities *cyclic*. Then the activities $a_1, \ldots, a_k$ may occur several times in a same trace of $\mathscr{T}$. Let n be the maximum number of repetitions of the above cycle over all the traces in $\mathscr{T}$. We next build an acyclic schema $\mathscr{WS}^+$ that, though different from $\mathscr{WS}$, is equivalent to it as far as the specific problem of mining patterns of executions is concerned.

The set of activities in $\mathscr{WS}^+$ contains all the acyclic activities of $\mathscr{WS}$ as well as $n+1$ distinct copies of each cyclic activity a_i, denoted by $a_i[0], \ldots, a_i[n]$. Edges of $\mathscr{WS}^+$ are defined as follows.

Procedure `InitializeStructures`;
- IS1 $\mathcal{WS}_{\mathcal{F}} := compute\_frequency\_graph(\mathcal{WS}, \mathcal{F})$;
- IS2 **forall** $p \in C(\mathcal{F})$ **do begin**
- IS3 $discarded(p) := \{ q \mid q \in C(\mathcal{F}), p \cap q \neq \emptyset \}$;
- IS4 $\langle l_p, u_p \rangle := compute\_frequency\_bounds(\mathcal{WS}_{\mathcal{F}}, p)$
- IS5 **end**;

Function `ComputeFrequentPatterns`(U: set of candidates): set of frequent patterns;
- CFP1 $LF := \{ r \mid r \in U, l_{terminating(r)}(starting(r)) \geq minSupp \}$;
- CFP2 $LU := \{ r \mid r \in U, u_{terminating(r)}(starting(r)) < minSupp \}$;
- CFP3 **return** $LF \cup \{ r \mid r \in U - (LF \cup LU), r$ is frequent w.r.t. $\mathcal{F} \}$;

Procedure `UpdateCandidateList`(L_k: set of frequent patterns): set of candidate patterns
- UCL1 $U := \emptyset$;
- UCL2 **forall** $p \in L_k$ **do** //***starting pattern
- UCL3 **forall** $q \in C' - discarded(p)$ **do begin** //***terminating pattern
- UCL4 $r := p \cup q$; $starting(r) = p$; $terminating(r) = q$;
- UCL5 $discarded(r) := discarded(p) \cup discarded(q)$;
- UCL6 $l_q(p) = \max(0, \max_{b \in q}\{l_p(b) - (f(b) - f(q))\}, f(q) - \sum_{b \in q}(f(b) - l_p(b)))$;
- UCL7 $u_q(p) = \min(f(q), \min_{b \in \texttt{OUTBORDER}(q)}\{u_p(b)\}, f(q) + \sum_{b \in q}(f(q) - l_p(b)))$;
- UCL8 $U := U \cup \{r\}$;
- UCL9 **end**;
- UCL10 **return** U;

Fig. 8. The *ws*-unconnected-find* ($\mathcal{F}, \mathcal{WS}, minSupp, C(\mathcal{F})$) algorithm.

First, all the edges in $\mathcal{WS}$ among acyclic activities are preserved in $\mathcal{WS}$. Then, each edge in $\mathcal{WS}$ from an acyclic activity to a cyclic one, say (b, a_i), is replaced by $(b, a_i[0])$. Further, for each edge in $\mathcal{WS}$ from a cyclic activity to an acyclic one, say (a_i, b), and for each $j = 0, \ldots, n$, the edge $(a_i[j], b)$ is included in $\mathcal{WS}^+$ if there is a trace t in $\mathcal{T}$ containing exactly j repetitions of the cycle and a_i is the last cyclic activity in t. Finally, for each edge (a_i, a_{i+1}) in the cycle and for each $j = 0, \ldots, n$, if there is a trace t in $\mathcal{T}$ containing at least j occurrences of the sequence $[a_i, a_{i+1}]$ then add $(a_i[j], a_{i+1}[j'])$, where $j' = j + 1$ or j depending whether $a_{i+1}[0]$ precedes $a_i[0]$ in t or not. As an example, the Fig. 9(b) shows the control flow obtained by applying the procedure above to the graph in Fig. 9(a) for a log $\mathcal{T}$ with $n = 2$.

5. System architecture and experiments

All the algorithms proposed in the paper have been implemented and integrated into a stand-alone system architecture developed in Java. In this section, we shall first discuss this architecture, and subsequently evaluate over it the effectiveness of our approach, by studying in particular the behavior of the *ws*-unconnected-find* algorithm and its pruning capabilities.

5.1. System architecture

The system architecture is sketched in Fig. 10. Notably, it is organized around various knowledge and data-bases that, in fact, guarantee the interoperability of the different software components.

The knowledge bases are of two kinds: persistent repositories and auxiliary volatile data structures. The persistent repositories store the workflow schemas, the logs and the patterns. In particular, the schemas in the *Schema Repository* can be either designed by means of a graphical *User Interface* or imported by some external sources by means of suitable *Import/Export modules*. Actually, these modules are not meant to be integrated into our system architecture, since they strongly depend on the specific application needs and since our techniques are to large extent independent on the specific modelling language. However, while different parsers are required for different source languages, their design can be easily carried in many practical scenarios, because of the choice of the internal modelling language, which has been kept simple and which just allows the basic features which are common to most of the proposals in the literature (recall that we basically need the control-flow graph

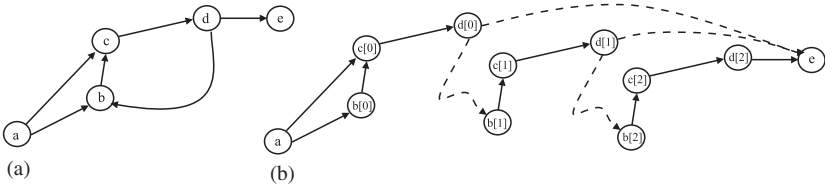


Fig. 9. Handling cyclic control-flow graphs.

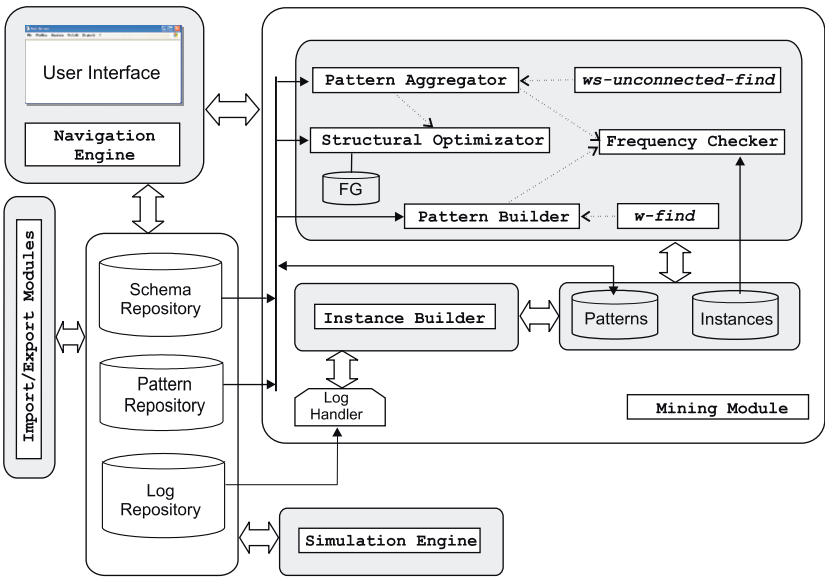


Fig. 10. System architecture.

only). Similarly, the *Log Repository* stores traces imported by external sources or generated by means of a *Simulation Engine*, which will be in-depth discussed in Section 5.2. Finally, the *Pattern Repository* stores all the patterns extracted through workflow mining tasks.

While persistent repositories are conceived to interact with the user, volatile data structures are devoted to guarantee effectiveness and performances of the implemented mining algorithms described in the previous sections. Indeed, logs are not imported in main memory as a whole. Rather, a *Log Handler* is responsible of sequentially (on-demand) reading some desired set of traces, so that the *Instance Builder* module can translate them into instances (see Section 2) and store them into main memory. Also, patterns discovered by means of the mining algorithms are handled in main memory so that they can be navigated by means of the *User Interface*, and they can be possibly stored on the repository.

The core of the architecture consists of the mining module. For the sake of clarity and conciseness, the basic components of the mining module are labelled with the names of the algorithms and procedures previously presented in the paper. Thus, the *w-find* (resp. *ws-unconnected-find*) module is devoted for discovering a frequent connected (resp. unconnected) patterns, by implementing the techniques described in the previous sections. In particular, *w-find* relies on a *Pattern Builder* module which looks at the structural specification of the process and performs levelwise search of the space of all the possible patterns. Each time a new pattern is discovered, a *Frequency Checker* module is used to assess whether the patter is frequent. Based on the connected patterns discovered by *w-find* and kept in memory, *ws-unconnected-find* starts composing patterns by means of the *Pattern Aggregator* module. Eventually, the *Structural Optimizator* module can help in speeding up this process by performing the optimizations described in Section 4.4. Then, all the generated patterns are again tested for their frequency by means of the *Frequency Checker*.

An important aspect of our architecture is the graphical *User Interface* which represents a very intuitive front-end for the final user. The interface

relies on a *Navigation Engine* module providing higher-level functionalities for browsing and exploiting all the discovered knowledge and for reusing it in the redesign/analysis phase of workflow. Two screenshots of this interface are reported in Fig. 11, which basically report the workflow of Fig. 1 as for it is modelled in the system. The workflow may be modified in the central side of the screen in an interactive way. Moreover, to help the user in dealing with huge specifications, in the left-down side, a zoom-in view of the workflow is provided, while in the left-top side of the screen, all the elements occurring in the workflow specification are listed. Interestingly, the results of the mining algorithms are provided to the user in an intuitive visual manner. Indeed, after a mining algorithm is invoked by setting some main parameters (e.g., in the bottom of Fig. 11), the discovered patterns appear in a pop-up list (e.g., in the top of Fig. 11), so that the current selected pattern is evidenced in both the list and the workflow (as a subgraph colored in blue) and it is available for further analysis. For instance, it can be saved in a jpeg format, or stored in the repository.

5.2. Simulation engine

An important module of our system architecture is the simulation engine. Basically, it allows to generate random instances and schemas over which mining algorithms can be tested. Also, as a side functionality, mining synthetic data can serve the purpose of identifying structural problems in the specifications. Even though this is not the main aim of our techniques (as discussed in Section 2), we believe that this possibility is relevant for the case the schema at hand has been designed by not considering correctness constraints leading to effective checks.

Since the method for generating instances and schemas suitable for the invocation of the *w-find* algorithm has been discussed in [16], here we just provide details on the generation of instances and schemas suitable for testing *ws-unconnected-find* and its optimized version. This generation can be tuned according to

(1) the size of $\mathscr{F}$,
(2) the average number d of frequent connected patterns to use in the generation of frequent unconnected patterns, and
(3) the average number u of frequent patterns to exploit in the generation of unfrequent unconnected patterns.

The main step in the whole process is the generation of a schema containing the desired number of frequent and unfrequent unconnected patterns. To this purpose, we first generate a set S containing d connected frequent patterns, where each pattern $p \in S$ has a frequency f_p associated (which is greater than a fixed frequency threshold). These patterns are the basic components of the workflow schema, and their composition shall define the structure of unconnected patterns.

Frequent unconnected patterns are generated according to the following strategy. Iteratively, a pair p, q of patterns is randomly picked from S and merged into $r = p \cup q$. Then, r is then relabelled and added to S, while p and q are retained in S with probability p_f. Precisely, r is obtained by connecting $\texttt{OUTBORDER}(p)$ to $\texttt{INBORDER}(q)$ in such a way that $p \cup q$ is frequent but unconnected. Let f_p and f_q be the frequencies of p and q, respectively. Each node in $\texttt{OUTBORDER}(p)$ is connected to a new node $a \in A^{\vee}_{in} \cap A^{\otimes}_{out}$. Similarly, a new node $b \in A^{\vee}_{in} \cap A^{\otimes}_{out}$ is connected to each node in $\texttt{INBORDER}(q)$. Then, f_r (the total number of instances involving either p or q is then set to $\max(f_p, f_q)$, and a connection between a and b is set by adding at most $n = \min(f_p, f_q)$ unfrequent nodes to r, and by connecting a and b by means of paths traversing such nodes. Further nodes can be connected either to a or b in order to retain frequencies.

Unfrequent unconnected patterns are built, starting from frequent (either connected or unconnected) patterns according to a similar strategy. Two frequent patterns p and q randomly chosen from S generate an unfrequent unconnected pattern r by connecting $\texttt{OUTBORDER}(p)$ and $\texttt{INBORDER}(q)$ with exactly one edge exhibiting a low frequency. Further nodes are added and connected either to $\texttt{OUTBORDER}(p)$ or to $\texttt{INBORDER}(q)$ in order to retain frequencies. The resulting graph r still has $f_r = \max(f_p, f_q)$, but $p \cup q$ has frequency 1. Again, p and q are retained into S with a fixed probability p_u, while r is added to S.

The above mentioned parameters u and d influence the number of frequent and unfrequent unconnected patterns to be generated. Starting from a set d of connected patterns, unconnected frequent patterns are generated until S reaches size u. Thus, at the end of this step S contains u components, each of which composed by several unconnected frequent patterns. These components are used to iteratively generate unfrequent unconnected patterns, until a single graph is obtained. In order to limit the growth

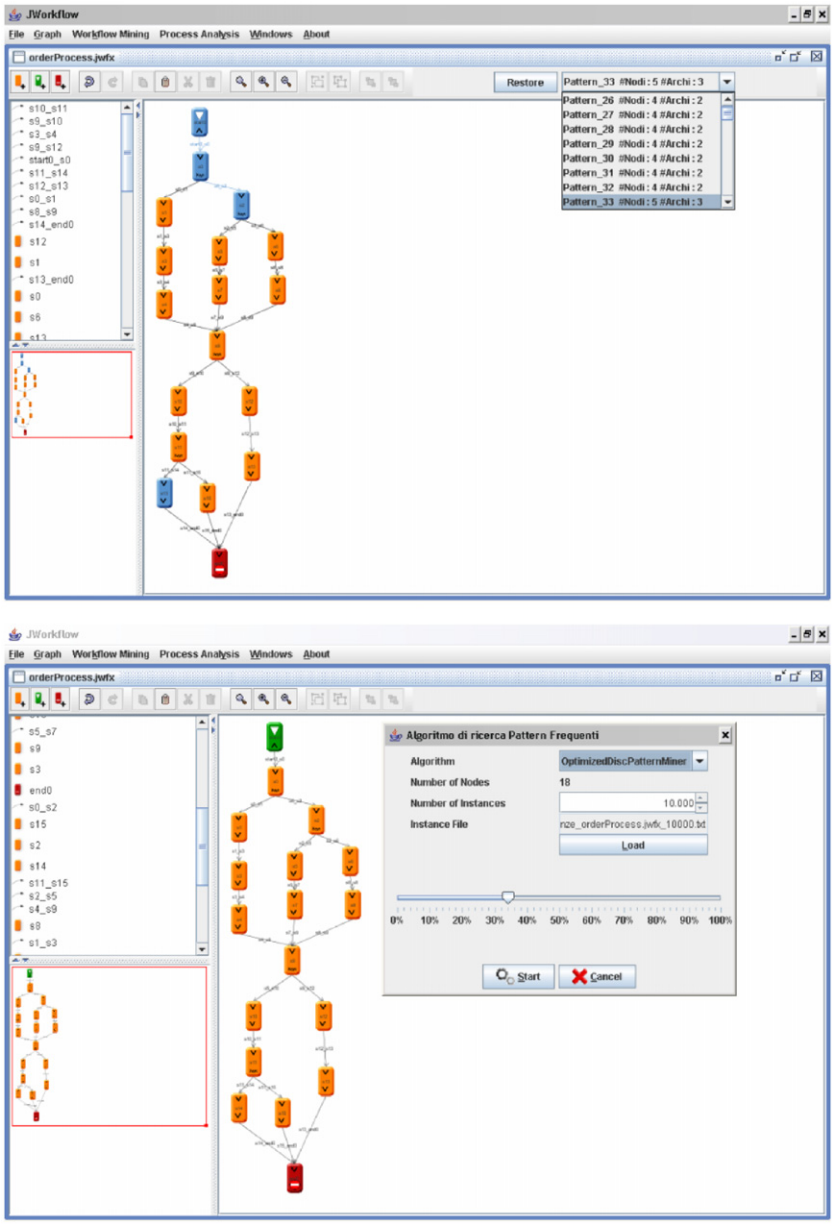


Fig. 11. Screenshots of the graphical *User Interface*.

of the graph, p_u and p_f are maintained relatively low (typically, $p_f = p_u = 0.2$). Finally, the desired instances are generated from the graph, by taking into account the frequency requirement of each node and edge.

Fig. 12 shows an example schema generated by fixing $d = 20$ and $u = 10$. As we can see, exclusive-fork nodes are extensively exploited to build both frequent and unfrequent unconnected patterns. Despite its apparent complexity, the schema exhibits several regularities in the form of "diamond connections" (i.e., groups of paths starting from the same exclusive-fork node and ending in the same or-join node).

On the basis of the above described generation procedure, we can expect that the larger is the difference between d and u, the higher is the number of unconnected frequent patterns contained within synthesized data. On the other side, the lower is the difference, the higher is the number of unconnected unfrequent patterns. It is worth noticing that the workflow topology (number of nodes and node connectivity) is directly influenced by the above

parameters. Indeed, at each step, the generation of a new component introduces new nodes, and the degree of each node in the border of the involved components is increased. For example, by fixing $d = 15$ and ranging u from 2 to 14, we obtain workflow schemas whose size ranges from 45 to 90 nodes and from 1300 to 5000 edges. Moreover, by ranging d from 10 to 40 we obtain schemas whose size ranges from 30 to 400 nodes, and from 500 to 10^5 edges. Notice also that the frequency of each unconnected frequent pattern is related to the number of unfrequent components and the number of desired total instances. Indeed, if u is the desired number of frequent unconnected patterns to compose infrequently, the number of instances in $\mathscr{F}$ necessary to compose them with frequency at least f is $|\mathscr{F}| \approx u \times f$.

5.3. Results

We conducted a number of experiments aimed at evaluating whether the computation of upper and lower bounds avoids the generation of unnecessary candidate patterns to check for frequency against the log data. Actually, it is not our objective to compare the proposed approach with existing general purpose graph-based mining techniques. Indeed, it has been already shown in [16] that specialized algorithms (including *w*-find), specifically designed to handle workflow constraints, can significantly outperform traditional graph mining approaches, even when they are suitably reengineered to cope with workflow instances. Rather, our aim is to study the behavior of the *ws-unconnected-find* algorithm and, specifically, to evaluate whether the optimization proposed for the procedures InitializeStructures, UpdateCandidateList and ComputeFrequentPatterns are effective in practice.

In a first set of experiments, we evaluated the ratio $f = n_{cc}/n_{cp}$ between the number n_{cc} of candidate patterns checked against the logs and the total number n_{cp} of candidate patterns. Low values of f represent a higher pruning capability of the algorithm *ws*-unconnected-find* w.r.t *ws-unconnected-find*. Fig. 13(a) shows the behavior of f for $d = 10$, $minSupp = 5\%$ and increasing values of $|\mathscr{F}|$ and u. As we can see, f is quite low, except when $u = 8$. Figs. 13(c) and (d) exhibit the number of unfrequent and frequent unconnected patterns

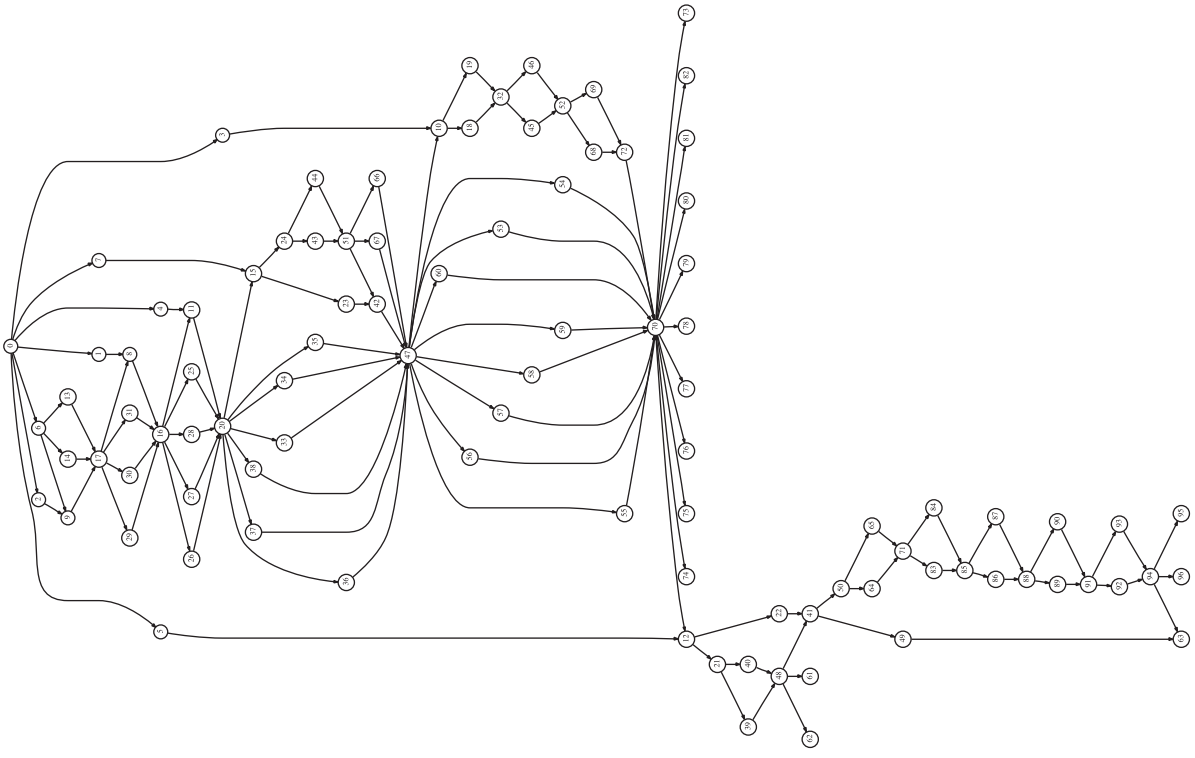


Fig. 12. An example synthetic schema.

discovered by resorting to upper and lower bounds, respectively.

Fig. 13(b) exhibits the ratio f for increasing values of *minSupp* and u, when $|\mathscr{F}| = 1.000$ and $d = 15$. Peaks within the graphs are mainly due to the fact that we are mining unconnected components: at lower support values, patterns are mined as frequent connected (indeed, the frequency of paths connecting the components is greater than the given threshold). As soon as support threshold increases, a higher number of unconnected frequent patterns is detected by the algorithm. Despite of these irregularities, we can notice that increasing values of u influence the pruning ability. In particular, by Figs. 14(a) and (b) we can see that, with high values of u, upper bounds provide substantial pruning ability. Again, the peaks in Fig. 14(b) find their explanation in the transformation of unconnected patterns into connected patterns.

Figs. 15(a) and (b) reports the pruning capabilities of the algorithm (formally, the number of pruned unfrequent and frequent patterns, respectively) for increasing values of *minSupp* and d, with u fixed to 2 and $|\mathscr{F}|$ to 1.000. As a matter of fact, upper bounds have high pruning capabilities that appear to be independent of *minSupp*, to large extent. However, for high values of *minSupp*, upper bounds seems to be not effective. Interestingly (but not surprisingly), results for lower bounds (cf. Fig. 15(b)) are dual; indeed, lower bounds appears to be quite effective at high values of *minSupp*, when several disconnections among frequent patterns are guaranteed to exists with high probability. Therefore, the combination of the two bounds guarantees that *ws*-unconnected-find* is able to significantly prune the search space, no matter of the required frequency threshold.

As a final remark, it is worth mentioning that upper bounds tend to be effective in the first steps of

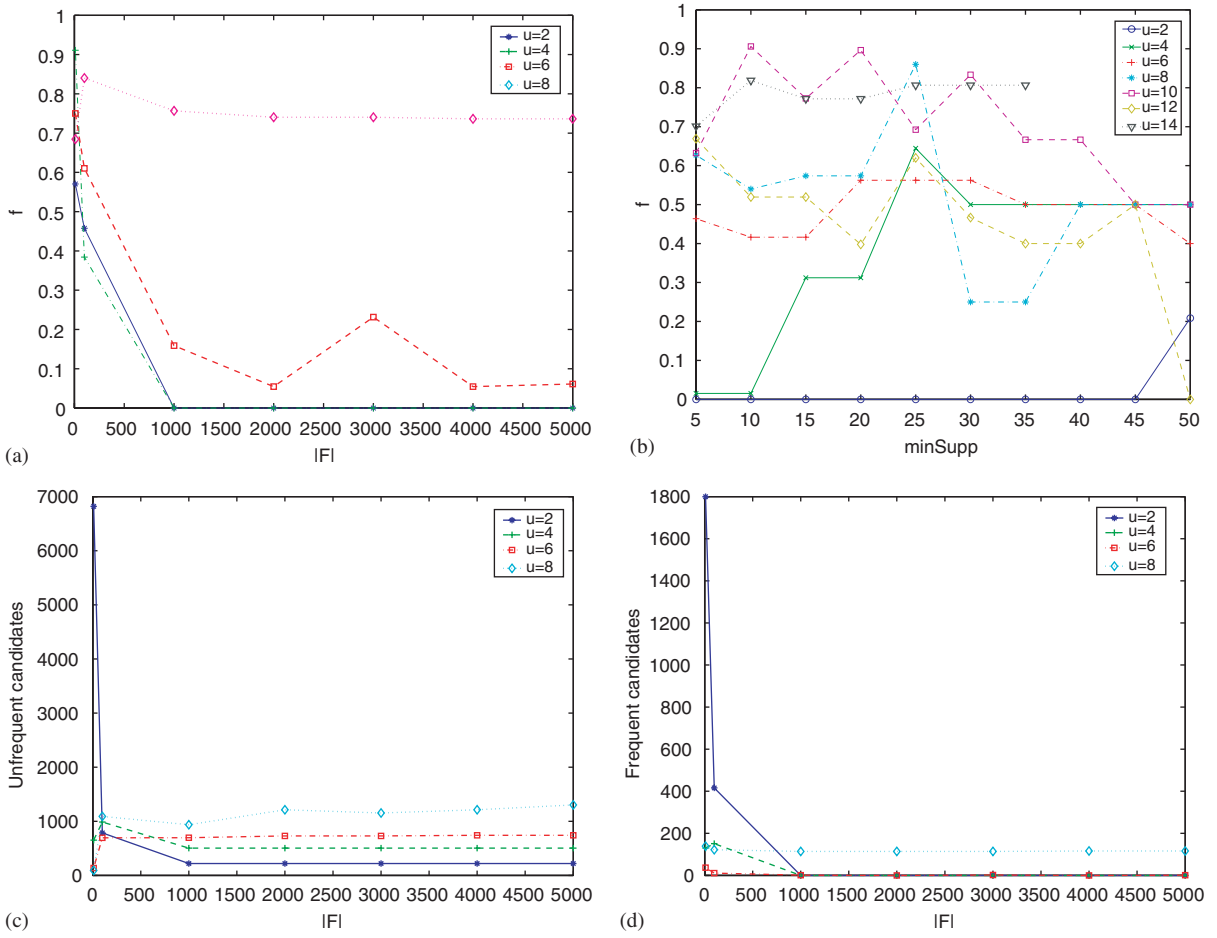


Fig. 13. Performance graphs: (a) ratio f, increasing values of $|\mathscr{F}|$ and u; (b) ratio f, increasing values of *minSupp* and u; (c) pruning by upper bound, for increasing values of $|\mathscr{F}|$ and u; (d) pruning by lower bound, for increasing values of $|\mathscr{F}|$ and u.

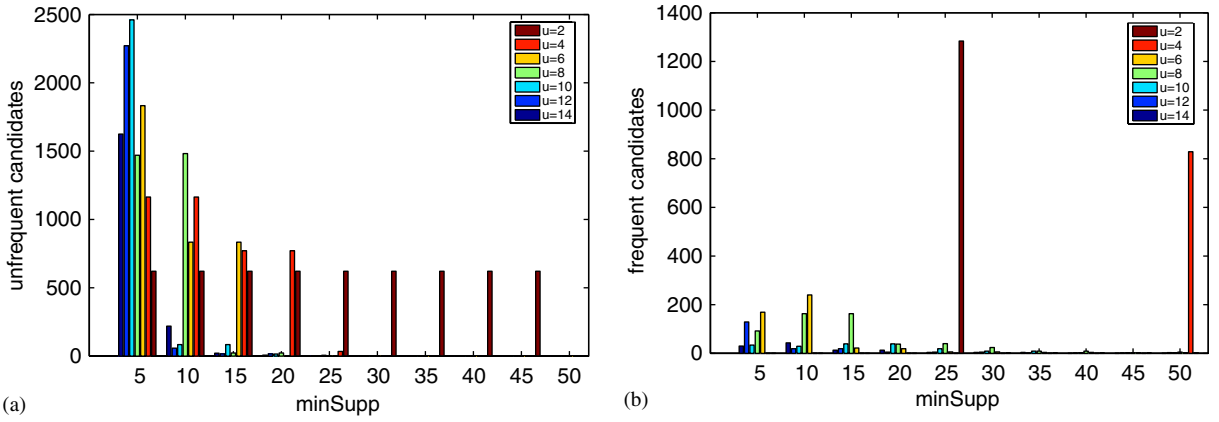


Fig. 14. Pruning effectiveness, for increasing values of *minSupp* and *u*: (a) pruning by upper bound; (b) pruning by lower bound.

the algorithm (i.e., in the computation of L_k for low values of k), whereas lower bounds effectiveness distributes throughout the entire execution of the algorithm. Fig. 16 shows an example distribution for $d = 15$ and different values of u, with $minSupp = 5\%$.

6. Related work: workflow mining and graph mining

The natural representation of a workflow execution as a directed graph (in which nodes represent the activities, and edges represent the relationships between them) can in principle allow to tackle the FUPD problem by exploiting graph-based structure mining techniques, for identifying frequent substructures in a given graph dataset [39]. Notably, these techniques represent the most related literature for our research, as no specific proposal for mining complex patterns in workflows has been proposed so far. Therefore, we shall next review and discuss some graph-mining approaches, which are receiving more and more attention because of their interesting applications in web analysis and in bioinformatics.

Typically, the approaches proposed in the literature rely on a representation of the available data as a set of labelled graphs $\{G_k(V_k, E_k, l_k)|k = 1, \ldots, n\}$ where V_k and E_k represent nodes and arcs, respectively, and l_k represents a mapping from nodes (or, in some approaches, edges) to labels. Then, they investigate the task of discovering the subgraphs occurring at least in a fixed fraction of $\{G_k(V_k, E_k, l_k)|\, k = 1, \ldots, n\}$. Seminal approaches to graph mining exploit multi-relational techniques [40], by combining the modelling capability of first-order logic programming with some smart strategies for exploring the search space. However, these solutions turned out to be effective only to some extent, because of their limits in properly facing some of the peculiarities and of the challenges related to graph mining. In fact, in order to efficiently face the problem of discovering frequent substructures from a set of graphs, mining algorithms have been subsequently proposed that comply with the following key requirements, which are specific for graph mining purposes:

(1) large graphs must only be considered after small graphs;
(2) each graph must be considered exactly once.

The first point has been conveniently addressed in the literature by means of a *levelwise* refinement approach, and by exploring the search space by either a breadth-first search [36,41,42], or a depth-first search [43–46]. Instead, the second point turned out to be more challenging. Indeed, in order to ensure that a given graph is not considered twice, one has to check whether the graph to hand is isomorphic to some previously processed graph. However, when different nodes/edges can share the same label, checking for graph isomorphism becomes a difficult problem for which no polynomial-time algorithm is known and for which, therefore, suitable heuristics have to be conceived.

In the current literature, the issue of processing each graph once is tackled be means of a generate-and-test approach. Firstly, a normal form is associated with each graph (which is therefore encoded in a different data-structure), and a criterium for sorting these encodings is devised. Interestingly, encodings are defined such that their comparison can be efficiently done, and such that isomorphic graphs

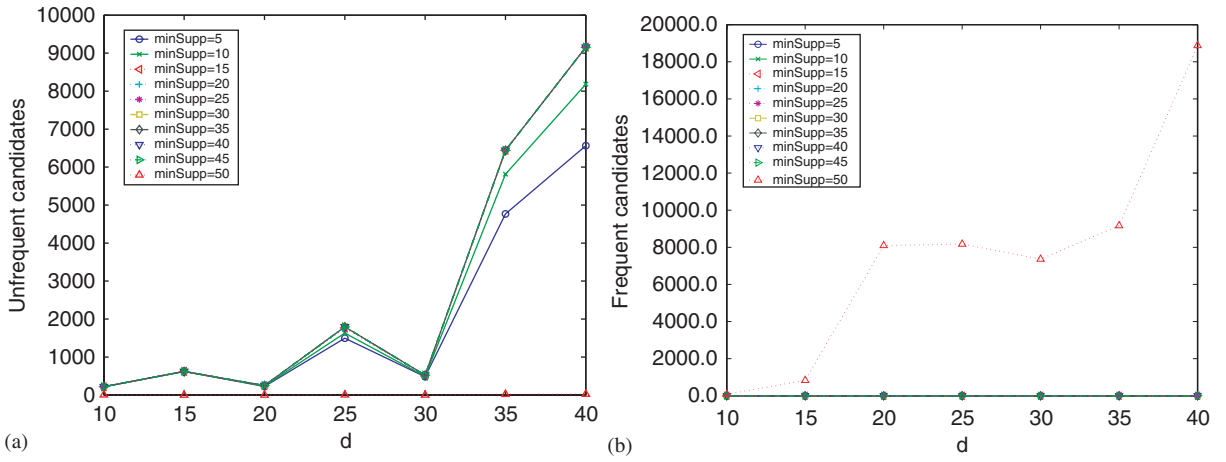


Fig. 15. Pruning effectiveness, for increasing values of d and *minSupp*: (a) pruning by upper bound; (b) pruning by lower bound.

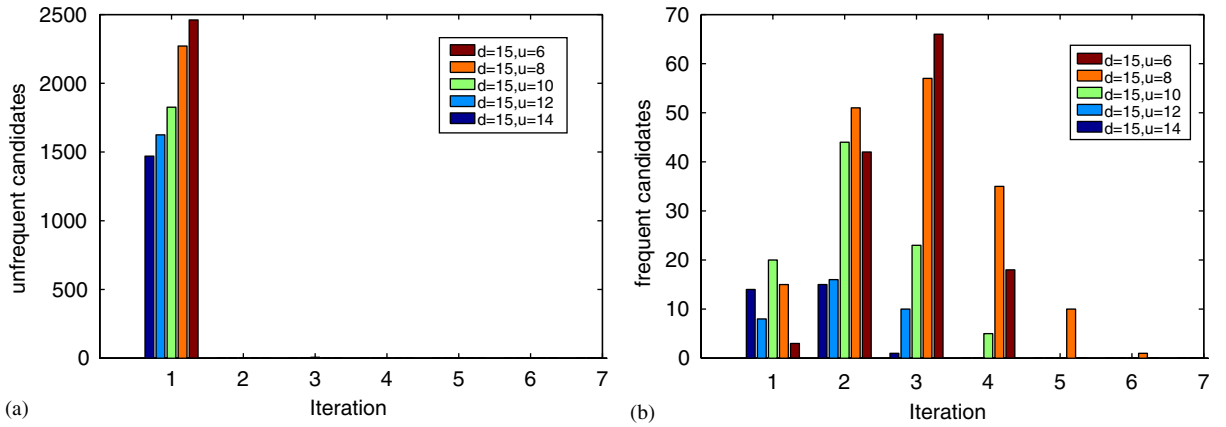


Fig. 16. Distribution of pruning abilities along the iterations: (a) pruning by upper bound; (b) pruning by lower bound.

have the same set of possible encodings. Then, once a new graph is generated and suitably encoded, an exhaustive procedure is used for determining whether another encoding has been already stored which can be associated to the graph and which precedes the current encoding according to the sorting criterium. Graphs for which this encoding exists are discarded. Normal forms often rely on *graph invariants*, i.e., properties which characterize the topological structure of a graph. Graph invariants are easier to compare, and usually are identical for isomorphic graphs. Indeed, the various approaches in the literature differ in the definition of the invariants as well in the strategy adopted for generating candidates. Examples of graph invariants are DFS codes [43,45] and canonical representations of the adjacency matrix [41,42,44].

Given the existence of these techniques, the reader may now wonder whether they can be adopted to deal with the problem of mining frequent workflow instances. Clearly enough, this is possible after some modifications are performed in order to fit the peculiarities of the specific applicative domain of workflow systems. Nonetheless, there are some issues which make the aforementioned approaches unpractical, and which are discussed below.

- First, in a workflow setting each node represents a distinct object, that is, a distinct activity; therefore, no two nodes may share the same label. This fact makes the graph isomorphism problem easier than in the general case, since simple heuristics can be detected to cope with. Therefore, more efficient algorithms can be devised.

- Second, the generation of patterns with (general purposes) graph-based approaches would not benefit of the exploitation of the execution constraints imposed by the workflow schema, such as precedences among activities, synchronization and parallel executions of activities (see, e.g., [7,14,11]). By contrast, it has been shown in [16], that specialized algorithms capable of handling such constraints can significantly outperform traditional graph mining approaches, even when they are suitably reengineered to cope with workflow instances.
- Third, all these approaches are able to discover connected patterns only. Thus, they do not cover the case of both connected and disconnected patterns. To the best of our knowledge, the only approach suitable for such a general task is [47], where patterns to be discovered may include *wildcards*. But again, this approach would not benefit of the specific constraints which can be defined over a workflow schema.

Before leaving this section, we discuss some further works which are related, even though in an indirect manner, to our research. First, the solutions proposed in this paper share some similarities with the approach recently developed in [48], for the problem of mining frequent *itemsets* (i.e., set of items). Indeed, Calders [48] argued that the use of frequency bounds for itemsets may significantly reduce the access to the database to count their actual number of occurrences. However, frequency bounds are only computed on the basis of the *antimonotonicity* property of the a priori algorithm, and strongly rely on the assumption of knowing the supports of all the subsets of a given itemset. For instance, by knowing the frequencies for itemsets $\{A\}$, $\{B\}$, $\{C\}$, $\{A,B\}$, $\{B,C\}$, and $\{A,D\}$, the set of rules in [48] are able to deduce possibly useful bounds for the itemset $\{A,B,C\}$; however, given the frequencies for itemsets $\{A,B\}$ and $\{C,D\}$ only, rules in [48] cannot derive non-trivial bounds for $\{A,B,C,D\}$. Our approach is, instead, specifically tailored for solving such circumstances. Indeed, it is able to deduce tight bounds for unconnected patterns on the basis of the knowledge of the frequencies of its connected components and of its atomic constituents (nodes and edges), only. Clearly, this is possible by exploiting some structural information (defined over the workflow schema) which is instead missing for the itemset mining problem. And, in fact, finding deduction rules for situations in which some of the subsets are missing, and when we only have partial knowledge of the supports has been left as an open problem in [48].

Finally, it is very relevant to note that a different, more traditional way of applying mining techniques in workflow systems consists in exploiting the information collected during the enactment of a process not yet supported by a WfMS (e.g., the transaction logs of ERP systems like SAP) to derive a model explaining all the recorded events [10–13]. Indeed, creating a work-flow design is a complicated time-consuming process and typically there are discrepancies between the actual work-flow processes and the processes as perceived by the management (cf. [10]). Accordingly, some specialized mining techniques have been devised in the literature which start by gathering information about the work-flow processes as they take place, rather than with a work-flow design, and address the problem of extracting the ''hidden'' model underlying the stored low-level data (e.g., transaction *logs*). Possibly, their output, i.e., the ''mined'' synthetic model, is used to (re)design a detailed workflow schema, capable of supporting automatic further enactments of the process. Therefore, these kinds of technique, called *process mining* techniques in the literature, are completely orthogonal with the techniques discussed in the paper, since they offer a valuable support at design time, rather than in the enactment phase.

7. Conclusion

In this paper we have addressed the problem of mining frequent unconnected workflow patterns. Basically frequent unconnected patterns are sets of connected patterns that frequently occur together in some log data (this may happen whenever the paths between the components fail to be frequent, e.g., because there are two or more alternative paths each one being unfrequent). Since each pattern represents a set of activities that are frequently executed together and may be abstractly seen as a subprocess in the workflow schema, unconnected patterns can be used by the system administrator to identify interesting and useful correlations among subprocesses which are apparently not related with each other. This information is useful to gain further semantic knowledge about the process.

To face the problem, we have proposed two algorithms which take benefit of the peculiarities of the application. The *ws-unconnected-find* algorithm takes into account information about the structure of the workflow, only. The *ws*-unconnected-find*

algorithm is a smart refinement where the knowledge of the frequencies of edges and activities in the instances at hand is also accounted for, in order to prune the search space of candidate patterns. The refinement has been made possible by means of the exploitation of some graph-theoretic techniques which allow to deduce tight bounds for the co-occurrences of patterns without resorting to the log. The correctness of both algorithms is formally proven, and several experiments evidence the ability of the graphical analysis to significantly improve the performances.

We conclude by observing that models proposed in this paper are essentially *propositional* models, for they assume a simplification of the workflow schema in which many real-life details are omitted. However, we believe that it can be very useful to enhance the models to cope with more complex constraints, such as time constraints, and rules, e.g., for exception handling. Designing data mining techniques looking at workflows under both the data and the control-flow perspective is left as subject for further research. Also, as another avenue of further research, it would be interesting to asses whether some of the observations and techniques exploited in the paper can be used for performing similar optimizations in different contexts in which the model for the data is assumed to be a graph [42,36,49].

References

[1] A. Bonifati, F. Casati, U. Dayal, M.C. Shan, Warehousing workflow data: challenges and opportunities, in: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01), 2001, pp. 649–652.

[2] U. Dayal, M. Hsu, R. Ladin, Business process coordination: state of the art, trends and open issues, in: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01), 2001, pp. 3–13.

[3] M. Sayal, F. Casati, U. Dayal, Ming-Chien Shan, Integrating workflow management systems with business-to-business interaction standard, in: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), 2002, pp. 287–296.

[4] M. Weske, G. Vossen, C. Bauzer Medeiros, F. Pires, Workflow management in geoprocessing applications, in: Proceedings of the Sixth ACM International Symposium on Advances in Geographic Information Systems (GIS'98), 1998, pp. 88–93.

[5] M. Castellanos, F. Casati, M.-C. Shan, U. Dayal, ibom: a platform for intelligent business operation management, in: Proceedings of the 21th International Conference on Data Engineering (ICDE'05), 2005, pp. 1084–1095.

[6] M. Gillmann, W. Wonner, G. Weikum, Worklow management with service quality guarantee, in: Proceedings of the ACM Conference on Management of Data (SIGMOD02), 2002, pp. 228–239.

[7] P. Koksal, S.N. Arpinar, A. Dogac, Workflow history management, SIGMOD Recod 27 (1) (1998) 67–75.

[8] W. van der Aalst, M. Weske, D. Grunbauer, Case handling: a new paradigm for business process support, Data Knowledge Eng. 53 (2) (2005) 129–162.

[9] M. Weske, W. van der Aalst, H.M.W. Verbeek, Advances in business process management, Data Knowl. Eng. 50 (1) (2004) 1–8.

[10] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, A.J.M.M. Weijters, Workflow mining: a survey of issues and approaches, Data Knowl. Eng. 47 (3) (2003) 237–267.

[11] J.E. Cook, A.L. Wolf, Automating process discovery through event-data analysis, in: Proceedings of the 17th International Conference on Software Engineering (ICSE'95), 1995, pp. 73–82.

[12] W.M.P. van der Aalst, A. Hirnschall, H.M.W. Verbeek, An alternative way to analyze workflow graphs, in: Proceedings of the 14th International Conference on Advanced Information Systems Engineering, 2002, pp. 534–552.

[13] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: Proceedings of the Sixth International Conference on Extending Database Technology (EDBT'98), 1998, pp. 469–483.

[14] W.M.P. van der Aalst, K.M. van Hee, Workflow Management: Models, Methods, and Systems, MIT Press, Cambridge, 2002.

[15] G. Keller, M. Nüttgens, A.W. Scheer, Semantische Process-modellierung auf der Grundlage Ereignisgesteuerter Process-ketten (EPK), Veröffentlichungen des Instituts für Wirtschaftsinformatik (iWi), Heft 89, University of Saarland, 1992.

[16] G. Greco, A. Guzzo, G. Manco, D. Saccà, Mining and reasoning on workflows, IEEE Trans. Knowl. Data Eng. 17 (4) (2005) 519–534.

[17] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Databases, 1994.

[18] L. Fisher (Ed.), Workflow Management Coalition, Workflow Handbook 2003, Future Strategies, Lighthouse Point, Florida, 2003.

[19] J. Hernandez, The SAP R/3 Handbook, 1997.

[20] A.W. Scheer, Business Process Engineering, ARIS-Navigator for Reference Models for Industrial Enterprises, Springer, Berlin, 1994.

[21] W.M.P. van der Aalst, Formalization and Verification of Event-driven Process Chains, Computing Science Reports 98/01, Eindhoven University of Technology, Eindhoven, 1998.

[22] W.M.P. van der Aalst, The application of petri nets to workflow management, J. Circuits Syst. Comput. 8 (1) (1998) 21–66.

[23] A. Bonner, Workflow, transactions, and datalog, in: Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99), 1999, pp. 294–305.

[24] H. Davulcu, M. Kifer, C.R. Ramakrishnan, I.V. Ramakrishnan, Logic based modeling and analysis of workflows, in: Proceedings of the 17th ACM Symposium on Principles of Database Systems (PODS'98), 1998, pp. 25–33.

[25] D. Wodtke, G. Weikum, A formal foundation for distributed workflow execution based on state charts, in: Proceedings of

the Sixth International Conference on Database Theory (ICDT'97), 1997, pp. 230–246.
[26] D. Wodtke, J. Weissenfels, G. Weikum, A. Dittrich, The Mentor project: steps towards enterprise-wide workflow management, in: Proceedings of the IEEE International Conference on Data Engineering (ICDE'96), 1996, pp. 556–565.
[27] G. Kappel, P. Lang, S. Rausch-Schott, W. Retschitzagger, Workflow management based on object, rules, and roles, IEEE Data Eng. Bull. 18 (1) (1995) 11–18.
[28] M.P. Sing, Semantical considerations on workflows: an algebra for intertask dependencies, in: Proceedings of the International Workshop on Database Programming Languages (DBPL'95), 1995, pp. 6–8.
[29] F. Casati, S. Ceri, B. Pernici, G. Pozzi, Conceptual modeling of workflows, in: Advances in Object-Oriented Data Modeling, MIT Press, Cambridge, 2000, pp. 281–306.
[30] D. Georgakopoulos, M. Hornick, A. Sheth, An overview of workflow management: from process modeling to workflow automation infrastructure, Distributed and Parallel Databases 3 (2) (1995) 119–153.
[31] H. Schuldt, G. Alonso, C. Beeri, H. Schek, Atomicity and isolation for transactional processes, ACM Trans. Database Syst. 27 (1) (2002) 63–116.
[32] W. van der Aalst, A. Hofstede, Yawl: yet another workflow language, 2002.
[33] Wil van der Aalst, Jörg Desel, Ekkart Kindler, On the semantics of EPCs: a vicious circle, EPK 2002, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2002, pp. 71–79.
[34] IDS Scheer, Aris process performance manager (aris ppm): measure, analyze and optimize your business process performance (whitepaper). Saarbruecken, Germany, ⟨http://www.ids-scheer.com⟩, 2002.
[35] B.F. van Dongen, W.M.P. van der Aalst, Multi-phase process mining: building instance graphs, in: Proceedings of the Conceptual Modeling—ER 2004, 2004, pp. 362–376.
[36] A. Inokuchi, T. Washi, H. Motoda, An a priori-based algorithm for mining frequent substructures from graph data, in: Proceedings of the Fourth European Conference on Principles of Data Mining and Knowledge Discovery, 2000, pp. 13–23.
[37] W. Aalst, A. van der, B. Hofstede, A. Kiepuszewski, Advanced workflow patterns, in: O. Etzion en, P. Scheuermann (Ed.), Seventh International Conference on Cooperative Information Systems (CoopIS 2000), Lecture Notes in Computer Science, vol. 1901, Springer, Berlin, 2000, pp. 18–29.
[38] W.M.P. Van der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, A.P. Barros, Workflow patterns, Distributed Parallel Databases 14 (1) (2003) 5–51.
[39] T. Washio, H. Motoda, State of the art of graph-based data mining, SIGKDD Explorations 5 (1) (2003) 59–68.
[40] L. Dehaspe, H. Toivonen, Discovery of frequent DATALOG patterns, Data Min. Knowl. Discovery 3 (1) (1999) 7–36.
[41] A. Inokuchi, T. Washi, H. Motoda, Complete mining of frequent patterns from graphs: mining graph data, Mach. Learn. (2003) 321–354.
[42] M. Kuramochi, G. Karypis, Frequent subgraph discovery, in: Proceedings of the IEEE International Conference on Data Mining (ICDM'01), 2001, pp. 313–320.
[43] X. Yan, J. Han, gSpan: graph-based substructure pattern pining, in: Proceedings of the IEEE International Conference on Data Mining (ICDM'02), 2001, pp. 721–724.
[44] J. Huan, W. Wang, J. Prins, Efficient mining of frequent subgraph in the presence of isomorphism, in: Proceedings of the Third IEEE International Conference on Data Mining (ICDM'03), 2003, pp. 549–552.
[45] X. Yan, J. Han, CloseGraph: mining closed frequent graph patterns, in: Proceedings of ACM International Conference on Knowledge Discovery and Data Mining (KDD'03), 2003, pp. 286–295.
[46] S. Nijssen, J.N. Kok, A quickstart in frequent structure mining can make a difference, in: Proceedings of the ACM SIGKDD'05 Conference on Knowledge Discovery and Data Mining, 2005.
[47] N. Vanetik, E. Gudes, Mining frequent labeled and partially labeled graph patterns, in: Proceedings of the 20th International Conference on Data Engineering (ICDE'04), 2004, pp. 91–102.
[48] T. Calders, Deducing bounds on the support of itemsets, in: R. Meo, P. Lanzi, M. Klemettinen (Eds.), Database Support for Data Mining Applications, Lecture Notes in Artificial Intelligence, vol. 2682, Springer, Berlin, 2004, pp. 214–233.
[49] M. Zaki, Efficiently mining frequent trees in a forest, in: Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining (SIGKDD02), 2002, pp. 71–80.