Mining Unconnected Patterns in Workflow

Gianluigi Greco[1], Antonella Guzzo[2], Giuseppe Manco [2], Domenico Saccà[3]

RT-ICAR-CS-05-09 **Settembre 2005**

Mining Unconnected Patterns in Workflows

Gianluigi Greco[*] Antonella Guzzo[*] Giuseppe Manco[†] Domenico Saccà[*,†]

Abstract

Recently, there is a growing interest in endowing Workflow Management Systems with advanced mechanism, based on data mining techniques, for monitoring and diagnosing workflow executions. Quite relevant in this context is the analysis of the most characterizing patterns of execution, which may allow to detect the sequences of activities typically leading to a successful/unsuccessful termination. The natural representation of a workflow (and its execution) as a directed graph, allows to address the problem as a frequent pattern discovery problem on graphs. Thus, some ad-hoc techniques have been devised for the discovery of frequent connected patterns of execution. On the other side, the problem of discovering frequent unconnected patterns has not been addressed yet, despite its relevance to the process of workflow mining: indeed finding sequences of activities which frequently occur together although they are not contiguous is crucial to discover meaningful execution patterns. This paper investigates the problem of mining unconnected patterns in workflows and presents for its solution two algorithms, both adapting the Apriori approach to the graphical structure of workflows. The first one is a straightforward extension of the level-wise style of Apriori whereas the second one introduces sophisticated graphical analysis of the frequencies of workflow instances. The experiments show that graphical analysis improves the performance of pattern mining by dramatically pruning the search space of candidate patterns.

Keywords: Workflow Management, Frequent Patterns Discovery, Graph Mining.

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. A *Workflow Management System* (WfMS) is a set of tools for defining, analyzing and managing the execution of workflows. WfMSs represent the most effective technological infrastructure for managing business processes in several application domains [6, 17, 3].

There is a growing body of proposals aiming at enhancing this technology in order to provide facilities for the human system administrator while designing complex processes as well as in order to offer an "intelligent" support in the decisions which have to be done by the enteprise during the enactment [7, 5, 12, 14]. In such a context, data mining techniques have proved to be very effective [16, 4, 15, 1]. In particular, a novel line of research has been introduced in [8], by investigating the ability of predicting the "most probable" workflow execution from the analysis of the information collected during the enactment. In this perspective, data mining techniques can help the administrator, by looking at all the previous instantiations (collected into *log* files in any commercial system), in order to extract unexpected and useful knowledge about the process, and in order to take the appropriate decisions in the executions of further coming instances.

The natural representation of a workflow execution as a directed graph (in which nodes represent activities, and edges the relationships between such activities), can in principle allow to address the above mentioned problem by exploiting structure mining techniques [18, 19, 13, 10]. However, the adaptation of such methods to workflow mining results unpractical from both the expressiveness and the efficiency viewpoint. Indeed, generation of patterns with such traditional approaches does not benefit from the exploitation of the executions' constraints imposed by the workflow schema, such as precedences among activities, synchronization and parallel executions of activities (see, e.g, [12, 17, 4]). By contrast, specialized algorithms capable of handling such constraints can significantly outperform traditional graph mining approaches, even when they are suitably reenginered to cope with workflow instances [9].

An example is the algorithm presented in [8], which exploits the connected structure of the workflow for finding patterns of execution which have been scheduled more frequently by the workflow system. These

[*]DEIS, University of Calabria, Via Pietro Bucci, 87036 Rende, Italy. Email {ggreco,guzzo}@di.deis.unical.it.

[†]ICAR-CNR, National Research Council, Via Pietro Bucci, 87036 Rende, Italy. Email {manco,sacca}@icar.cnr.it.

structures are called *frequent connected patterns* (short: frequent $\mathcal{F}$-patterns) and represent subprocesses whose frequency of occurrence in a set $\mathcal{F}$ of logs is above a given threshold σ. However, one limitation of the this algorithm is the focus on connected subgraphs, which correspond to subprocesses frequently occurring together and whose dependency relationships can be explicitly inferred. This makes the proposed approach eventually incomparable with more general approaches (such as, e.g., [11]), which would allow the mining of general patterns of execution.

In this paper, we extend the approach in [8] and study the problem of discovering correlations among general patterns of execution in a workflow. In particular, we focus our attention on unconnected patterns, which are arbitrary subsets of connected patterns exhibiting no explicit dependency relationship. Thus, we assume that a set $\mathcal{P}$ of frequent $\mathcal{F}$-patterns is given and we are interested in discovering whether any of the subsets of $\mathcal{P}$ is frequent as well. This problem, called *frequent unconnected patterns* discovery (short: `FUPD`), occurs very often in practical scenarios and is crucial for the identification of the critical subprocesses that lead with high probability to (un)desired final configurations.

It is worth noting that `FUPD` has a trivial solution consisting in the application of a level-wise algorithm (in the *a-priori* style) which combines all the unconnected patterns in $\mathcal{P}$ and then checks for their frequency. However, this approach would not benefit from the peculiarities of the workflow graph that can be profitably used for pruning the search space. We show how the structure of the workflow together with some elementary information such as the frequency of occurrences of elementary activities suffices for pruning the search space and for deriving an efficient and practically fast algorithm, called *ws*-unconnected-find*.

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 introduce the problem of mining unconnected patterns, and propose an a-priori like algorithm for discovering them. Section 4 describes how the analysis of the workflow structure allows to deduce tight bounds on the frequency of candidate patterns, and consequently to specialize the a-priori based algorithm. Section 5 discusses of several experiments that confirm the validity of the approach. Finally in Section 6 we draw our conclusions by pointing to further enhancements to the proposed approach that are worth future research efforts.

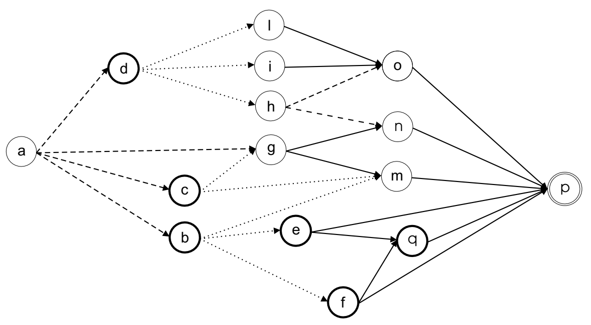


Figure 1: An example of workflow schema.

2 Workflow Model and Problem Formulation

We begin by formally defining the domain under consideration. A *workflow schema* $\mathcal{WS}$ is a tuple $\langle A, E, a_0, A_F \rangle$, where A is a finite set of *activities*, $E \subseteq (A - A_F) \times (A - \{a_0\})$ is an acyclic relation of precedences among activities, $a_0 \in A$ is the starting activity, and $A_F \subseteq A$ is the set of final activities. Moreover, A is partitioned into $A^{\vee}_{in} \cup A^{\wedge}_{in} \cup A^{\vee}_{out} \cup A^{\wedge}_{out} \cup A^{\otimes}_{out}$, where $A^{\vee}_{in}$ are the *or-join* nodes in A, $A^{\wedge}_{in}$ are the *and-join* nodes in A, $A^{\vee}_{out}$ are the *or-fork* nodes in A, $A^{\wedge}_{out}$ are the *and-fork* nodes in A, and $A^{\otimes}_{out}$ are the *exclusive-fork* nodes in A. The tuple $\langle A, E \rangle$ is often referred to as the *control graph* of $\mathcal{WS}$.

Informally, an activity in $A^{\wedge}_{in}$ acts as synchronizer (also called a *join* activity in the literature), for it can be executed only after all its predecessors are completed, whereas an activity in $A^{\vee}_{in}$ can start as soon as at least one of its predecessors has been completed. Moreover, once finished, an activity $a \in A^{\wedge}_{out}$ activates all its outgoing activities, $a \in A^{\vee}_{out}$ activates some of the outgoing activities, while $a \in A^{\otimes}_{out}$ activates exactly one outgoing activity.

A workflow schema has a quite natural graphical representation, by means of a directed acyclic graph.

Example 1. *An example of workflow schema is shown in Figure 1. In this schema, we adopt the graphical convention of representing nodes in $A^{\vee}_{in}$ with plain circles and nodes in $A^{\wedge}_{in}$ with bold circles; moreover, nodes in $A^{\otimes}_{out}$ exhibit dashed outgoing arcs, whereas nodes in $A^{\vee}_{out}$ exhibit dotted arcs and nodes in $A^{\vee}_{out}$ exhibit bold arcs. Finally, nodes in A_F are represented by means of a double circle. To summarize figure 1 represents a schema $\mathcal{WS}$ where $A^{\wedge}_{in} = \{d, c, b, f, e, n, q\}$ and $A^{\vee}_{in} = \{a, g, l, i, h, m, o, p\}$, while $A^{\otimes}_{out} = \{a, h\}$, $A^{\wedge}_{out} = \{l, i, g, e, f, m, n, o, q\}$, $A^{\vee}_{out} = \{b, c, d\}$, and $A_F = \{p\}$.* $\triangleleft$

The enactment of a workflow gives rise to an instance, i.e., to a proper subgraph of the schema which is derived satisfying the constraints imposed by the instances included.

Definition 1. *Let $\mathcal{WS}$ be a workflow schema. Any connected subgraph $I = \langle A_I, E_I \rangle$ of the control flow graph is an* instance *of $\mathcal{WS}$ (denoted as $\mathcal{WS} \models I$) if the following conditions hold:*

(i) $a_0 \in A_I$,

(ii) $A_I \cap A_F \neq \emptyset$,

(iii) *for each* $a \in A_I$, $|\{b \mid (b,a) \in E_I\}| > 0$,

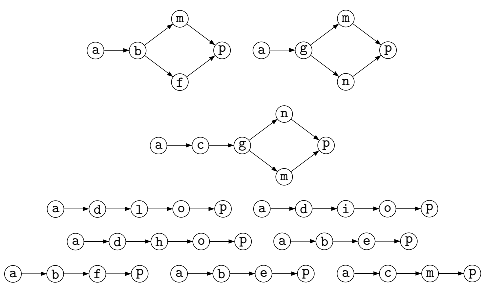
(iv) *for each* $a \in A_I \cap A^{\wedge}_{in}$, $\{b \mid (b,a) \in E\} \subseteq A_I$,

(v) *for each* $a \in A_I \cap A^{\wedge}_{out}$, $\{b \in A^{\vee}_{in} \mid (a,b) \in E\} \subseteq A_I$, *and*

(vi) *for each* $a \in A_I \cap A^{\otimes}_{out}$, $|\{b \mid (a,b) \in E_I\}| \leq 1$ *and* $|\{b \mid (a,b) \in E_I\}| = 1$ *if* $\{b \in A^{\vee}_{in} \mid (a,b) \in E\} \neq \emptyset$,

The set of all instances of $\mathcal{WS}$ is denoted by $\mathcal{I}(\mathcal{WS})$. □

Example 2. *With reference to the schema of fig. 1, the following are example instances:*



Despite its apparent simplicity, static reasoning on the control graph of a workflow is ineffective. Indeed, even simple reachability problems, such as deciding whether an arc is included in some instance (and consequently whether a workflow admits an instance), or whether two nodes always occur together are intractable [9]. This motivates the need to resort to data mining techniques in order to infer properties of the workflow under consideration on the basis of the observation of the past history. In particular, we assume that each instance is properly stored by the workflow management system in the log file, which can be seen as a set $\mathcal{F} = \{I_1, ..., I_n\}$ such that $\mathcal{WS} \models I_i$, for each $1 \leq i \leq n$. Among the instances of $\mathcal{F}$ we are interested in discovering the most frequent patterns of execution as next defined.

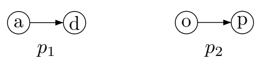
Definition 2. *A graph $p = \langle A_p, E_p \rangle \subseteq \mathcal{WS}$ is a* $\mathcal{F}$-pattern *(cf. $\mathcal{F} \models p$) if there exists $I = \langle A_I, E_I \rangle \in \mathcal{F}$ such that $A_p \subseteq A_I$ and p is the subgraph of I induced by the nodes in A_p. In the case $\mathcal{F} = \mathcal{I}(\mathcal{WS})$, the subgraph is simply said to be a* pattern. □

Let *supp*$(p) = |\{I|\{I\} \models p \wedge I \in \mathcal{F}\}|/|\mathcal{F}|$, be the *support* of a $\mathcal{F}$-pattern p. 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 whose support is greater than *minSupp*.

Example 3. *Let us consider the control graph of fig. 1 the instances of example 2 and* minSupp $= 0.3$. *Then, the patterns*

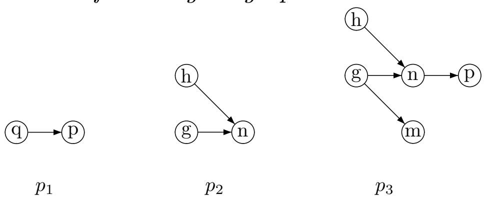


are frequent connected patterns. Also, notice that none of the nodes `l`, `i`, `h` *is frequent, whereas the subgraph $p = p_1 \cup p_2$ is frequent (and hence $\{p_1, p_2\}$ is a frequent unconnected pattern).* ◁

2.1 Mining Connected Patterns.

In [8], an approach to FCPD has been devised by resorting to the notion of *deterministic closure*. The approach is based on the idea of exploiting the control graph of $\mathcal{WS}$ in order to reduce the number of patterns to generate. To achieve this aim, we can only consider connected $\mathcal{F}$-patterns which satisfy both local and global constraints (i.e., they are deterministically closed). Such graphs are denoted as *weak patterns*, or simply w-patterns.

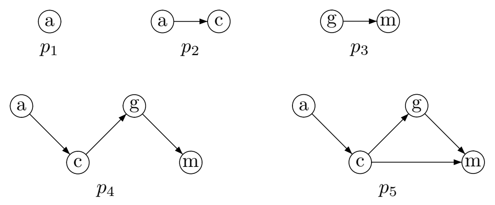
Example 4. *Let us consider the control graph of Figure 1, and the following subgraphs.*



p_1 and p_2 are not w-pattern: indeed, `q` *is a synchronizer (thus triggering the occurrence of both* `e` *and* `f`*), whereas* `g` *and* `n` *are and-forks (thus triggering the occurrence of both* `m` *and* `p`*). Notice that p_3 is instead a w-pattern, since both all the constraints over the involved nodes are satisfied.* ◁

Working with w-patterns rather than $\mathcal{F}$-patterns is not an actual limitation, since each frequent $\mathcal{F}$-pattern is bounded by w-patterns: it can be shown [9] that for each frequent $\mathcal{F}$-pattern p, there exist a frequent w-pattern p' 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 extrated incrementally, by starting from frequent "elementary" weak patterns (i.e., weak patterns obtained as the deterministic closure of single nodes), and by extending each frequent weak pattern using two basic operations: adding a frequent arc 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 $\prec$, defined as follows: $p_1 \prec p_2$ if and only if p_2 can be obtained from p_1 either adding an arc, or by merging a node in p_1 with an elementary weak pattern.

Example 5. *With reference to the workflow graph of Figure 1, let us consider the subgraphs shown below:*



The subgraphs p_1, p_2 and p_3 are elementary patterns: indeed, p_1 is the deterministic closure of a and p_2 is the deterministic closure of c, whereas p_3 can be obtained from g. Also, notice that $p_1 \subset p_2$, although $p_1 \not\prec p_2$. Neither p_4 nor p_5 are elementary patterns, as no single node can generate them. Notice that $p_2 \prec p_4$ and $p_3 \prec p_4$, since $p_4 = p_2 \cup p_3 \cup \{(c, g)\}$. Finally, $p_4 \prec p_5$, since $p_5 = p_4 \cup \{(c, m)\}$. $\lhd$

It can be shown [9] that all the connected weak patterns of a given workflow schema can be constructed by means of chains over the $\prec$ relation, each of which starting from an elementary w-patterns. As a consequence, it turns out that the space of all connected weak patterns is a lower semi-lattice w.r.t. the precedence relation $\prec$. The algorithm w-find, reported in Figure 2, exploits an apriori-like exploration of this lower semi-lattice.

Within the algorithm, EW represents the set of all elementary w-patterns, and EW_p denotes the set of the elementary weak patterns contained p. Moreover, $Compl(\text{EW}_p)$ contains all the elementary patterns which are neither in EW_p nor contained in some element of EW_p. Finally, $E^{\subseteq}$ denotes the subset of arcs

```
Input: A workflow Graph $\mathcal{WS}$, a set $\mathcal{F} = \{I_1, \ldots, I_N\}$ of instances of $\mathcal{WS}$.
Output: A set of frequent $\mathcal{F}$-patterns.
Method: Perform the following steps:
 1      $L_0 := \{e | e \in EW, e$ is frequent w.r.t. $\mathcal{F}\}$;
 2      $k := 0,\ R := L_0$;
 3      $FrequentArcs := \{(a,b) | (a,b) \in E^{\subseteq},\ \langle\{a,b\}, \{(a,b)\}\rangle$ is frequent w.r.t. $\mathcal{F}\}$;
 4      $E_f^{\subseteq} := E^{\subseteq} \cap FrequentArcs$;
 5      repeat
 6        $U := \emptyset$;
 7        forall $p \in L_k$ do begin
 8          $U := U \cup addFrequentArc(p)$;
 9          forall $e \in Compl(EW_p) \cap L_0$ do
10            $U := U \cup addFrequentEWPattern(p, e)$;
11        end
12        $L_{k+1} := \{p | p \in U, p$ is frequent w.r.t. $\mathcal{F}\}$;
13        $R := R \cup L_{k+1}$;
14      until $L_{k+1} = \emptyset$;
15      return $R$;
--------------------------------------------------------------------------------
Function $addFrequentEWPattern(p = \langle A_p, E_p\rangle, e = \langle A_e, E_e\rangle)$: w-pattern;
  $p' := \langle A_p \cup A_e, E_p \cup E_e\rangle$;
  if $p'$ is connected, then return $p'$ else return $addFrequentConnection(p', p, e)$;
--------------------------------------------------------------------------------
Function $addFrequentConnection(p' = \langle A_{p'}, E_{p'}\rangle, p = \langle A_p, E_p\rangle, e = \langle A_e, E_e\rangle)$: w-pattern;
  $S := \emptyset$
  forall frequent $(a,b) \in E_f^{\subseteq} - E_p$ s.t. $(a \in A_p,\ b \in A_e) \vee (a \in A_e,\ b \in A_p)$ do begin
    $q := \left\langle A_{p'}, E_{p'} \cup (a,b)\right\rangle$;
    if $\mathcal{WS} \models q$ then $S := S \cup \{q\}$;
  end
  return $S$
--------------------------------------------------------------------------------
Function $addFrequentArc(p = \langle A_p, E_p\rangle)$: pattern;
  $S := \emptyset$
  forall frequent $(a,b) \in E_f^{\subseteq} - E_p$ s.t. $a \in A_p,\ b \in A_p$ do begin
    $p' := \langle A_p, E_p \cup (a,b)\rangle$
    if $\mathcal{WS} \models p'$ then $S := S \cup \{p'\}$;
  end
  return $S$
```

Figure 2: **Algorithm** w-find($\mathcal{WS}$,$\mathcal{F}$)

in $\mathcal{WS}$ whose source is not an and-fork, i.e., $E^{\subseteq} = \{(a, b) \in E \mid a \in A^{\vee}_{out} \cup A^{\otimes}_{out}\}$. At each stage, the computation of L_{k+1} (steps 5-14) is carried out by extending any pattern p generated at the previous stage ($p \in L_k$), in two ways: by adding frequent edges in $E^{\subseteq}$ (*addFrequentArc* function); or by adding an elementary weak pattern (*addEWFrequentPattern* function).

The above algorithm has many interesting properties. In particular, it can be shown [9] that it outperforms traditional graph mining approaches, even when they are suitably reengineered to cope with workflow instances.

3 Mining Unconnected Patterns

In this paper we shall deal with an efficient solution for FUDP by assuming that the set $C(\mathcal{F})$ of all the frequent (w.r.t. *minSupp*) connected patterns in the set of instances $\mathcal{F}$ has been already computed using the w-find described in the previous section. It is worth noting that FUPD has a straightforward solution consisting in the application of a level-wise algorithm (in the *a-priori* style) [2, 11] which combines all the unconnected patterns in $\mathcal{P}$ and then checks for their frequency. Indeed, further in the section we present an implementation of Apriori that takes into account some basic properties of workflows. However, in order to achieve a larger amount of pruning of the search space, we need to further exploit the peculiarities of the workflow graph. To this end, one might think of combining 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 Figure 1. Observe that the activities `a` and `p` 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 example 2 and *minSupp* = 30%). On 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 files.

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 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`). In order to systematically study such circumstances, we develop a graph theoretic approach for predicting whether two activities are coupled just on the basis of the workflow structure and of the frequency of the elementary activities alone.

A Level-Wise Algorithm. We present a first simple solution to the `FUPD` problem, achieved by means of the algorithm *ws-unconnected-find* shown in Figure 3. The algorithm receives in input the workflow schema $\mathcal{WS}$ and the set $C(\mathcal{F})$ of frequent connected $\mathcal{F}$-patterns, and returns all the frequent unconnected patterns.

Before detailing the mains steps we need some further definitions and notations. Given a unconnected pattern p, we say that p is a *starting* pattern if it contains the starting activity of the workflow schema; otherwise, it is said a *terminating* pattern. Rather than computing all the possible unconnected patterns, we limit on starting patterns and we show how the space of all the connected starting patterns forms a lower semi-lattice that can be profitably explored in a bottom-up fashion. In fact, given two starting patterns r and p we say that r directly precedes p, denoted by $r \prec p$, if there exist a terminating pattern q such that $r = p \cup q$. Moreover, 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. Such an approach is, in fact, exploited by the algorithm in Figure 3 that computes all the frequent starting patterns, by generating at each step k the patterns made of k distinct unconnected patterns (stored in the set L_k).

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

Then, at each step it generates a number of candidates (stored in U) in the main cycle (steps 4–13). Each generated pattern p is obtained by 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 unconnected components, given a pattern p we can compute in advance a set of connected patterns that must not be combined with p, denoted by *discarded*(p). This set contains all the patterns which have a non-null intersection with p, and it 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 $\mathcal{F}$, thus creating the set L_{k+1} containing all the frequent unconnected patterns made of $k + 1$ unconnected patterns — notice that in this implementation this task is simply done by means of a scan in the logs $\mathcal{F}$.

Finally, the generated starting frequent patterns are added to the actual result R, and in the steps 8–12 the set *discarded* is updated for the patterns which are discovered to be frequent.

The computational cost of the algorithm is related to the number of unconnected components contained by the maximal frequent unconnected patterns. This number indeed influences the number of scans to the log file.

Proposition 3.1. *The algorithm ws-*unconnected-find *computes the set of all the unconnected frequent patterns with at most $|C(\mathcal{F})| - |L_0|$ scans in the log file.*

The correctness of the algorithm follows from the following observation.

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: A set of frequent unconnected $\mathcal{F}$-patterns.

Method: Perform the following steps:

```
1    InitializeStructures();
2    $L_0 := \{ p \mid p \in C(\mathcal{F}),\, a_0 \in p \}$;      //***frequent connected starting patterns
3    $k := 0$, $R := L_0$; $C' := C(\mathcal{F}) - L_0$;
4    repeat
5      $U :=$ UpdateCandidateList($L_k$)
6      $L_{k+1} :=$ ComputeFrequentPatterns(U);
7      $R := R \cup L_{k+1}$;
8      forall $r \in U - L_{k+1}$ do begin
9        $p := starting(r)$;
10       forall $p' \in L_{k+1}$ s.t. $p \subset p'$ do
11         $discarded(p') := discarded(p') \cup \{terminating(r)\}$;
12     end
13   until $L_{k+1} = \emptyset$;
14   return $R$;
```

Procedure `InitializeStructures`;

```
IS1    forall $p \in C(\mathcal{F})$ do
IS2      $discarded(p) := \{ q \mid q \in C(\mathcal{F}),\, p \cap q \neq \emptyset \}$;
```

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

```
CFP1    return $\{ r \mid r \in U,\, supp(r) > minSupp\}$;
```

Function `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        $U := U \cup \{r\}$;
UCL7      end;
UCL8    return $U$;
```

Figure 3: **Algorithm** *ws-unconnected-find*($\mathcal{WS}$,$\mathcal{F}$,*minSupp*,$C(\mathcal{F})$)

THEOREM 3.1. *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 initial activity a_0 such that $p' \cup q$ is frequent as well.*

Informally, the theorem states that, since the starting activity is executed in each instance, each unconnected frequent pattern can be extended with the initial activity. As a consequence, each frequent unconnected pattern can be generated starting from the unconnected frequent starting patterns.

EXAMPLE 6. *By assuming minSupp $= 30\%$ and the set $\mathcal{F}$ of instances of example 2, the patterns $\{\mathtt{a}, \mathtt{b}\} \cup \mathtt{p}$, $\mathtt{a} \cup \{\mathtt{m}, \mathtt{p}\}$ and $\{\mathtt{a}, \mathtt{d}\} \cup \{\mathtt{o}, \mathtt{p}\}$, are maximal unconnected frequent patterns.* $\lhd$

4 Optimizing Candidate Generation

In this section we present some techniques for efficiently pruning the search space identified by the level-wise algorithm *ws-unconnected-find.* Our idea is to exploit the structure and the information regarding the frequency of each activity in order to identify, before their actual testing w.r.t. the logs, those patterns which are necessarily (un)frequent.

In the following, we assume the existence of a set $\mathcal{F}$ of instances of a workflow schema $\mathcal{WS}$. Then, let q be a not-necessarily connected component of $\mathcal{WS}$ with frequency $f(q)$ and p be a connected component with frequency $f(p)$ such that q and p are unconnected. Our aim is to compute as efficiently as possible the number of instances in $\mathcal{F}$ executing both the components p and q, denoted by $f_p(q)$.

Obviously, the most trivial and inefficient way for computing $f_p(q)$ is to make a scan of the log $\mathcal{F}$. 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 some efficient way not requiring the access to the log.

We next start with the basic situation in which p and q are patterns each one made of a single activity of $\mathcal{WS}$.

4.1 Computing Frequency Bounds for Activities. Given an activity $a \in A$, let G_a be the subgraph of the control flow of $\mathcal{WS}$ induced by all the nodes b such that there is a path from b to a in $\mathcal{WS}$. Note that all such nodes can be easily determined by reversing the arcs in $\mathcal{WS}$ and computing the transitive closure of a.

The starting point of our approach is to compute for each node b in G_a, the number of instances in $\mathcal{F} = \{I_1, ..., I_n\}$ executing both the activities a and b, denoted by $f_a(b)$. As already said, we actually turn for computing a lower bound $l_a(b)$ and an upper bound $u_a(b)$ — obviously $l_a(b) \leq u_a(b) \leq min(f(a), f(b))$.

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 $\mathcal{F}$.

Definition 3. (Frequency graph) *Let $\langle A, E\rangle$ be the control flow of a workflow schema $\mathcal{WS}$ and let $\mathcal{F} = \{I_1, ..., I_n\}$ be a set of instances of $\mathcal{WS}$. The* frequency graph *$\mathcal{WS}_{\mathcal{F}} = \langle A, E, f_A, f_E\rangle$ is a weighted graph such that*

- $f_A : A \mapsto \mathbf{N}$ *maps each activity a to the number of instances in $\mathcal{F} = \{I_1, ..., I_n\}$ executing it, and*
- $f_E : E \mapsto \mathbf{N}$ *maps each arc e to the number of instances in $\mathcal{F} = \{I_1, ..., I_n\}$ containing this arc.*

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. $f(e)$). $\square$

Figure 4 shows the frequency graph associated with the schema of fig. 1, built by taking into account the set $\mathcal{F}$ of instances described in example 2.

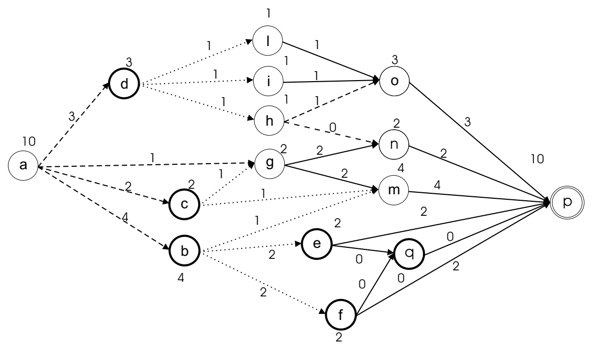


Figure 4: Frequency graph associated with the schema of fig. 1.

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 G_a of $\mathcal{WS}$ - as $\mathcal{WS}$ is acyclic a topological sorts exists for each of its subgraphs including G_a. Then we proceed as shown in Figure 5. In the step 1, the lower and upper bounds of the activity a are fixed to the known value $f(a)$, determined through G_a. Then, each node b_i in G_a is processed according to the topological sort. In 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 – note that $|C(b_i)| \neq 0$. 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 arc outgoing from b_i is in some path reaching c. This justifies the formula of step 4.

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) $b_i \in A^{\vee}_{out} \cup A^{\wedge}_{out}$, 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 $L^{\wedge}_2$).

Observe that the final step in the algorithm possibly find tighter lower bounds by exploiting the fact that, given two nodes b and c in G_a if $(b, c) \in \mathcal{WS}$, b is an and-fork node and c is an or-join node, then $l_a(c) \leq l_a(b)$ the activity b is executed each time the activity c is.

Input: An annotated workflow schema $\mathcal{WS}_{\mathcal{F}}$, 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 G_a.

Output: for each node $b \in G_a$, the values $l_a(b)$ and $u_a(b)$.

Method: Perform the following steps:

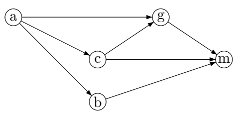
```
1   $l_a(a) := f(a);\ \ u_a(a) := f(a);$
2   forall $i = 2..k$ do begin
3     $C(b_i) := \{b \mid (b_i, b) \in E \wedge b \in G_a\};$
4     $u_a(b_i) := \min(f(b_i), f(a), U)$, with $U = \sum_{e=(b_i,c) \mid c \in C(b_i)} \min(f(e),\, u_a(c))$.
5     if $b_i \in A^{\vee}_{out} \cup A^{\wedge}_{out}$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

Figure 5: The *compute_frequency_bounds*$(\mathcal{WS}_{\mathcal{F}}, a)$ algorithm

THEOREM 4.1. *The following properties hold for the algorithm in Figure 5:*

1. *The parameters U, $L^{\vee}_1$, $L^{\vee}_2$, $L^{\wedge}_1$ and $L^{\wedge}_2$ 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, lower and upper bound of the frequency $f_a(b_i)$.*
3. *The procedure can be computed in time $O(|G_a|^2)$.*

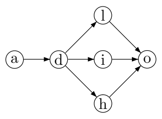
EXAMPLE 7. *Let us consider the graph $G_{\mathtt{m}}$ induced by node $\mathtt{m}$:*



The inferred topological sort is $\langle \mathtt{m}, \mathtt{g}, \mathtt{b}, \mathtt{c}, \mathtt{a} \rangle$. Hence, by applying compute_frequency_bounds$(\mathcal{WS}_{\mathcal{F}}, m)$, we obtain the following bounds for node $\mathtt{m}$:

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

According to the these bounds, it is easy to see that $\mathtt{m} \cup \mathtt{a}$ is a frequent unconnected pattern, whereas $\mathtt{m} \cup \mathtt{b}$ is not (even though $\mathtt{b}$ and $\mathtt{m}$ are frequent patterns). It is interesting to analyze also the bounds for node $\mathtt{o}$. Given the dependency graph $G_{\mathtt{o}}$,



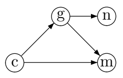
we obtain $u_{\mathtt{o}}(\mathtt{a}) = u_{\mathtt{o}}(\mathtt{d}) = 3$, $u_{\mathtt{o}}(x) = 1$ for $x \in \{\mathtt{l}, \mathtt{i}, \mathtt{h}\}$, and $l_{\mathtt{o}}(y) = 1$ for each node y in $G_{\mathtt{o}}$. Thus, even if $\mathtt{d} \cup \mathtt{o}$ is a frequent unconnected pattern, lower bounds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

4.2 Computing Frequency Bounds for Patterns. Let us now turn to the 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 $\mathcal{F}$ executing both the component p and each 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 $\mathtt{INBORDER}(p)$ denote the set of the activities in p having incoming arcs from $\mathcal{WS} - p$. Let $\mathcal{WS}(p)$ be the workflow schema derived from $\mathcal{WS}$ by adding a new and-join node, say a_p, corresponding to the component p, and by adding an arc from each node b in $\mathtt{INBORDER}(p)$ to a.

In the frequency graph of $\mathcal{WS}(p)$ set $f(a_p) = f(p)$, and $f(e) = f(p)$ for each $e = (b, a_p) \in E$. Then, the function *compute_frequency_bounds*$(\mathcal{WS}_{\mathcal{F}}, p)$ is defined as *compute_frequency_bounds*$(\mathcal{WS}(p)_{\mathcal{F}}, a_p)$.

EXAMPLE 8. *Let us consider the pattern p, structured as shown below:*



According to the workflow schema shown in fig. 1, INBORDER(p) = {c, g*} (indeed, both nodes have incoming arcs from nodes which are not in p). In order to compute frequency bounds for p, we act on the frequency graph and connect both* g *and* c *with a new dummy node* a_p*. Thus, in the new frequency graph* $\mathcal{WS}(p)$*, we obtain* $G_{a_p} = G_{\mathsf{c}} \cup G_{\mathsf{o}}$ *and hence lower and upper bounds for each node w.r.t.* a_p *can be computed.* ◁

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

THEOREM 4.2. *Let* $\mathcal{WS}$ *be a workflow schema,* $\mathcal{F}$ *be a set of instances, and 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*$(\mathcal{WS}(p)_{\mathcal{F}}, a_p)$*. Then,* $l_{a_p}(b)$ *and* $u_{a_p}(b)$ *are indeed lower and upper bounds of* $f_p(b)$*.*

4.3 Algorithm *ws*-unconnected-find.* Once the frequency bounds for a given pattern (w.r.t. any activity) are computed, we can face the more general problem. Let q be a general component of $\mathcal{WS}$ with frequency $f(q)$ and p be a connected component with frequency $f(p)$ such that q and p are unconnected. A lower bound and an upper bound of $f_p(q)$ are as follows:

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

Here, `OUTBORDER`(p) refers to all the nodes in q having outgoing arcs in $\mathcal{WS} - q$. 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 in the upper bound computation all internal nodes of q (i.e., those not in `OUTBORDER`(p)) as they are always executed together with at least one node in `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 $f(b) \geq f(q)$, 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.

THEOREM 4.3. *Let* $\mathcal{WS}$ *be a workflow schema,* $\mathcal{F}$ *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* $f_p(q)$*.*

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 $u_p(q) <$ *minSupp* then it is always the case that p and q never occur frequently together. Conversely, if $l_p(q) \geq$ *minSupp* then p and q can be combined into a pattern that is frequent as well.

Thus, the algorithm *ws-disconected-find* can be optimized (see Figure 6), by suitably adapting the procedures `InitializeStructures`, `UpdateCandidateList` and `ComputeFrequentPatterns`. Specifically, the former also compute the frequency graph and all the frequency bounds for any pattern, by exploiting the above formulas. The second, instead, verifies the frequency in the log $\mathcal{F}$ only for patterns which cannot be tested with the frequency bounds only.

5 Experiments and Discussion

In this section we study the behavior of the *ws*-unconnected-find* algorithm, by examining its pruning capability. Experiments are 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.

In our experiments, we use synthesized data, in which both the workflow schema and the instances are artificially generated. The generation can be tuned according to: i) the size of $\mathcal{F}$, ii) the average number d of frequent connected patterns to use in the generation of frequent unconnected patterns, and iii) the average number u of frequent patterns to exploit in the generation of unfrequent unconnected patterns. Data are generated according to the following strategy. First, a set S of d frequent connected patterns are generated, according to a fixed frequency threshold; next, iteratively, a pair p, q of patterns is randomly chosen from S and merged into $r = p \cup q$. p and q are retained from S with a fixed probability p_f, while r is relabeled and added to S. r is obtained by connecting `OUTBORDER`(p) to `INBORDER`(q) in such a way that $p \cup q$ is frequent but unconnected. More in details, let f_p and f_q be the frequencies of p and q, respectively. Each node in `OUTBORDER`(p) is connected to a new node $a \in A^{\vee}_{in} \cap A^{\otimes}_{out}$. Similarly, a new node

```
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     $U := 0$;
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         $l_q(p) = \max(0, \max_{b \in p}\{l_q(b) - (f(b) - f(p))\})$;
UCL7         $u_q(p) = \min(f(p), \sum_{b \in \mathtt{OUTBORDER}(q)} u_p(b))$;
UCL8         $U := U \cup \{r\}$;
UCL9       end;
UCL10    return $U$;
```

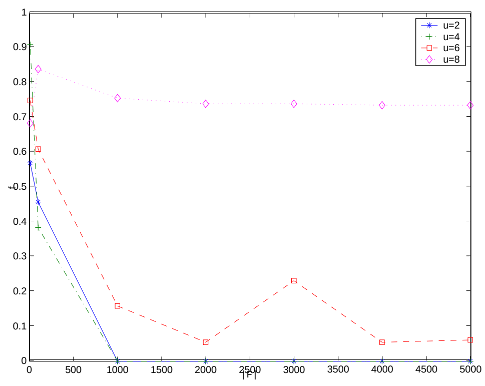
Figure 6: The *ws*-unconnected-find*($\mathcal{F}$,$\mathcal{WS}$,*minSupp*,$C(\mathcal{F})$) algorithm

$b \in A_{in}^{\vee} \cap A_{out}^{\otimes}$ is connected to each node in INBORDER(q). f_r 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.

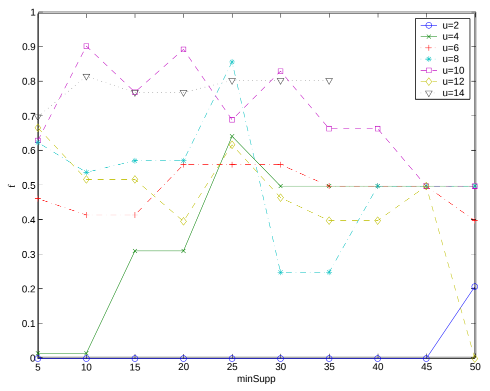
Unfrequent unconnected patterns are built, starting from frequent (either connected or unconnected) patterns according to a similar strategy. Two frequent patterns p and q randomly chosen from S generate an unfrequent unconnected pattern r by connecting OUTBORDER(p) and INBORDER(q) with exactly one edge exhibiting a low frequency. Further nodes are added and connected either to OUTBORDER(p) or to 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 u and d parameters 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 of the graph, p_u and p_f are mantained relatively low (tipically, $p_f = p_u = 0.2$). Finally, the desired instaces are generated from the graph, by taking into account the frequency requirement of each node and edge.

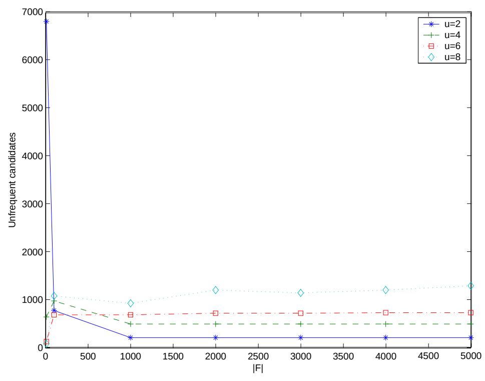
On the basis of the above described generation procedure, we can expect that, the larger the difference between d and u, the higher is the number of unconnected frequent patterns contained within synthesized data. On the other side, the lower is the difference, the higher is the number of unconnected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



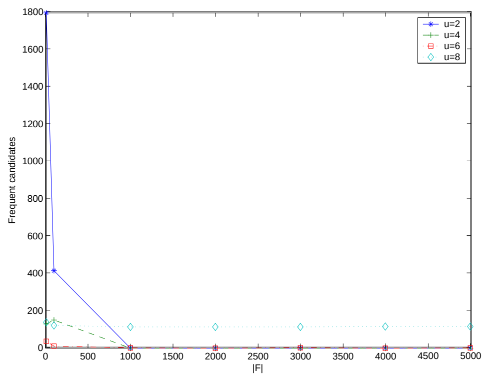
(a) Ratio f, increasing values of $|\mathcal{F}|$ and u.



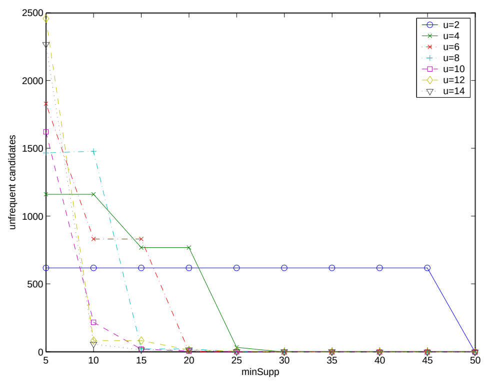
(b) Ratio f, increasing values of *minSupp* and u.



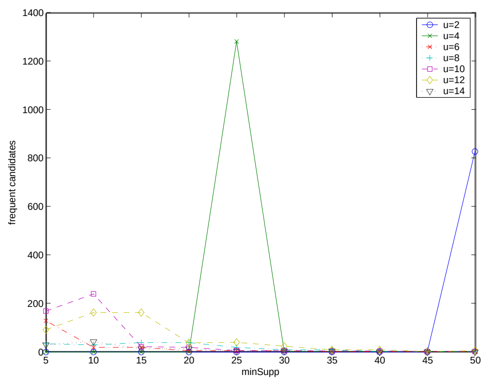
(c) Pruning by upper bound, for increasing values of $|\mathcal{F}|$ and u.



(d) Pruning by lower bound, for increasing values of $|\mathcal{F}|$ and u.



(e) Pruning by upper bound, for increasing values of *minSupp* and u.



(f) Pruning by lower bound, for increasing values of *minSupp* and u.

Figure 7: Performance Graphs.

tern 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 $\mathcal{F}$ necessary to compose them with frequency at least f is $|\mathcal{F}| \approx u \times f$.

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*. Figure 7(a) shows the behavior of f for $d = 10$, $minSupp = 5\%$ and increasing values of $\mathcal{F}$ and u. As we can see, f

is quite low, except when $u = 8$. Figures 7(c)and 7(d) exhibit the number of unfrequent and frequent unconnected patterns discovered by resorting to upper and lower bounds, respectively.

Figure 7(b) exhibits the ratio f for increasing values of *minSupp* and u, when $|\mathcal{F}| = 1.000$ and $d = 15$. Peaks within the graphs are mainly due to the fact that we are mining unconnected components: at low support values, patterns are mined as frequent connected (i.e., the frequency of paths connecting the components is greater than the given threshold). As soon as support threshold increases, frequencies of paths tend to decrease, and hence 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 figures 7(e) and 7(f) we can see that, with high values of u, upper bounds provide substantial pruning ability.

More in general, upper bounds are better in pruning, as also demonstrated by figures 8(a) and 8(b). In these graphs, the number of pruned unfrequent and frequent patterns is shown for increasing values of *minSupp* and d, with u fixed to 2 and $\mathcal{F}$ to 1.000. Interestingly, lower bounds are quite effective at high values of *minSupp*, which guarantee several disconnections among frequent patterns.

As a final remark, it is worth mentioning that upper bounds tend to be effective in the first steps of 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. More extensive graphical analysis, omitted here for lack of space, gives evidence of the claim.

6 Conclusions

In this paper we have addressed the problem of mining frequent unconnected workflow patterns. We have developed a graph theoretic approach for predicting whether activities in a workflow are coupled in the executions, on the basis of the workflow structure and of the frequency of the elementary activities alone. The approach has been adopted in a level-wise algorithm for mining frequent patterns, and revealed as a powerful tool for pruning the search space of candidate patterns.

We conclude by sketching some directions of future research. The models proposed in this paper and in [8] 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 the models can be easily updated to cope with more complex constraints, such as time constraints, pre-conditions and post-conditions, and rules for exception handling. Furthermore, we believe that many of the observations we exploit in the paper can be used for performing similar optimizations in different contexts in which the model of the data is assumed to be a graph [13, 11, 20].

References

[1] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *Proc. 6th Int. Conf. on Extending Database Technology (EDBT'98)*, pages 469–483, 1998.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.

[3] A. Bonifati, F. Casati, U. Dayal, and M.C. Shan. Warehousing workflow data: Challenges and opportunities. In *Proc. 27th Int. Conf. On Very Large Data Bases (VLDB'01)*, pages 649–652, 2001.

[4] J.E. Cook and A.L. Wolf. Automating process discovery through event-data analysis. In *Proc. 17th Int. Conf. on Software Engineering (ICSE'95)*, pages 73–82, 1995.

[5] U. Dayal, M. Hsu, and R. Ladin. Business process coordination: State of the art, trends and open issues. In *Proc. 27th Int. Conf. On Very Large Data Bases (VLDB'01)*, pages 3–13, 2001.

[6] D.Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[7] M. Gillmann, W. Wonner, and G. Weikum. Worklow management with service quality guarantee. In *Proc. ACM Conf. on Management of Data (SIGMOD02)*, pages 228–239, 2002.

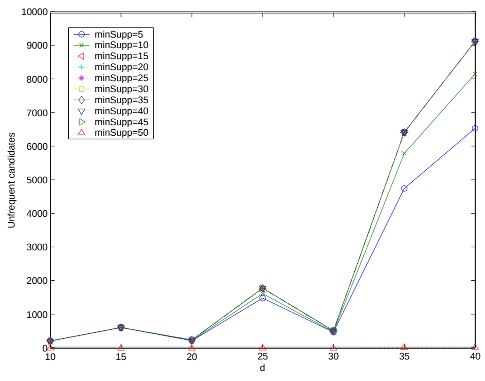
[8] G. Greco, A. Guzzo, G. Manco, and D. Saccà. Mining frequent instances on workflows. In *Proc. Pacific-Asia Conference on Knowledge Discovery (PAKDD'03)*, pages 209–221, 2003.

[9] G. Greco, A. Guzzo, G. Manco, and D. Saccà. Mining and reasoning on workflows. Technical Report 12, ICAR-CNR, 2004. Available at `http://www.icar.cnr.it/isi`.

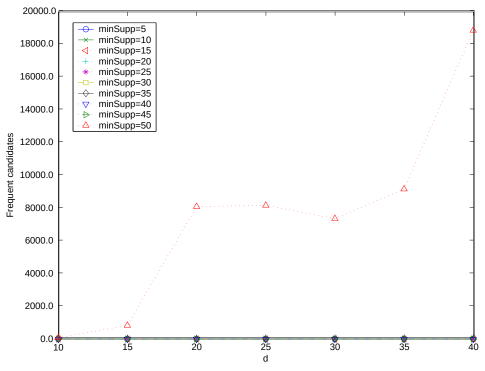
[10] A. Inokouchi et al. A fast algorithm for mining frequent connected subgraphs. Technical Report RT0448, IBM Research, Tokyo Research Laboratory, 2002.

[11] A. Inokuchi, T. Washi, and H. Motoda. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *Proc. 4th European Conf. on Principles of Data Mining and Knowledge Discovery*, pages 13–23, 2000.

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



(a) Pruning by upper bound, for increasing values of *d* and *minSupp*.



(b) Pruning by lower bound, for increasing values of *d* and *minSupp*.

Figure 8: Performance Graphs (cont.).

[13] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. IEEE Int. Conf. on Data Mining (ICDM'01)*, pages 313–320, 2001.

[14] P. Senkul, M. Kifer, and I.H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 694–702, 2002.

[15] W.M.P. van der Aalst, A. Hirnschall, and H.M.W. Verbeek. An alternative way to analyze workflow graphs. In *Proc. 14th Int. Conf. on Advanced Information Systems Engineering*, pages 534–552, 2002.

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

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

[18] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, 5(1):59–68, 2003.

[19] X. Yan and J. Han. gSpan: Graph-based substructure pattern pining. In *Proc. IEEE Int. Conf. on Data Mining (ICDM'02)*, pages 721–724, 2001.

[20] M. Zaki. Efficiently Mining Frequent Trees in a Forest. In *Proc. 8th Int Conf. On Knowledge Discovery and Data Mining (SIGKDD02)*, pages 71–80, 2002.

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Discovering Expressive Process Models by Clustering Log Traces

Gianluigi Greco[1], Antonella Guzzo[2], Luigi Pontieri[2], Domenico Saccà[3]

Rapporto Tecnico N.: 1 **RT-ICAR-CS-06-04**

Data: **Aprile 2006**

[1] Dipartimento di Matematica, Università degli Studi della Calabria, via P. Bucci 30B, Rende (CS)
[2]Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Cosenza, via P. Bucci 41C, 87036 Rende(CS)
[3] DEIS,Università degli Studi della Calabria, via P. Bucci 41C, Rende (CS)