Probabilistic Workflow Mining

Ricardo Silva[*]
Center for Automated Learning and Discovery
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
rbas@cs.cmu.edu

Jiji Zhang *
Department of Philosophy
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
jiji@andrew.cmu.edu

James G. Shanahan
Clairvoyance Corporation
5001 Baum Boulevard, Suite 700
Pittsburgh, PA 15213
jimi@clairvoyancecorp.com

ABSTRACT

In several organizations, it has become increasingly popular to document and log the steps that makeup a typical business process. In some situations, a normative workflow model of such processes is developed, and it becomes important to know if such a model is actually being followed by analyzing the available activity logs. In other scenarios, no model is available and, with the purpose of evaluating cases or creating new production policies, one is interested in learning a workflow representation of such activities. In either case, machine learning tools that can mine workflow models are of great interest and still relatively unexplored. We present here a probabilistic workflow model and a corresponding learning algorithm that runs in polynomial time. We illustrate the algorithm on example data derived from a real world workflow.

Categories and Subject Descriptors

G.3 [**Mathematics of Computing**]: PROBABILITY AND STATISTICS

General Terms

Algorithms

Keywords

Workflow mining, graphical models, causal models

1. MOTIVATION

Most large social organizations are complex systems. Every day they perform various types of processes, such as assembling a car, designing and implementing software, organizing a conference, and so on. A process is a set of tasks to be accomplished, where every task might have pre-requisites within the process that have to be fulfilled before execution.

For instance, implementing a database query system should not be performed before the necessary data structures are designed. One should not add the doors to a car before the seats are in place. That is, some tasks are essentially *sequential*. But it is fair to say that building the speakers of a car bears no implication on the manufacturing of the tires, and vice-versa, i.e., some tasks can be executed in *parallel*. Moreover, there are tasks that are *mutually exclusive*: for instance, one has to decide if a given share of coffee harvest is to be exported, or sent to the internal market. Some tasks might also be executed in cycles.

To analyze productivity, identify outliers, cut unnecessary expenses, and design other production policies, *models of work* are important, i.e., abstract representations of typical process instances modeling the causal and probabilistic dependencies among tasks. Such models are based on the concepts of sequential, parallel, iterative (cyclic) and mutually exclusive tasks and are used to evaluate costs, monitor processes, and predict the effect of new policies [7]. For these reasons, empirically building process models from data is of great interest. Such a problem has been called *process mining*, or simply *workflow mining* [8, 3, 4], because the usual representation of work processes is workflow graphs.

In this paper, we describe a probabilistic model for workflow graphs, and algorithms for learning such graphs from data. The setup is similar to other graphical models. In Section 2, we introduce a formal description of workflow graphs and the associated generative models. Section 3 describes a data mining algorithm for learning the structure of workflow graphs from data. An empirical study is given in Section 4. Related work is discussed in Section 5.

2. APPROACH

In this section, we first give a description of the family of graphs that are allowed in our framework. This is followed by a probabilistic parameterization of such graphs. We then describe the role of temporal information in our approach, followed by our treatment of hidden variables and noise. We conclude this section with a concept (called *faithfulness*) that links empirically observable constraints to graphs.

2.1 WORKFLOW GRAPHS

For simplicity, in this paper we will work with acyclic graphs only. A future extension of this work will cover the cyclic case.

[*]This work was carried out while on internship at Clairvoyance Corporation

KDD'05, August 21–24, 2005, Chicago, Illinois, USA.

In a typical process, each task T has *pre-requisites*, a set of other tasks whose *execution* will determine the probability of T being executed. A workflow graph G is a directed acyclic graph (DAG) where each task is a node, and the parents of a node are its *direct pre-requisites*. That is, the decision to execute T does not depend on any (other) task in G given its parents.

Motivated by other workflow representations (see [8] for a review) which are used to model a large variety of real-world processes, we adopt a constrained DAG representation. Let a *AND/OR workflow graph* (AO graph) be a constrained type of DAG, with any node being in one of the following classes:

- *split node*, a node with multiple children;
- *join node*, a node with multiple parents;
- *simple node*, a node with no more than one parent and no more than one child;

We require that an AO graph must have exactly one node that has no parents (a *start node*) and exactly one node that has no children (an *end node*). Informally, split nodes are meant to represent the points where choices are made (i.e., where one among mutually exclusive tasks will be chosen) or where multiple parallel threads of tasks will be spawned. As a counterpart, join nodes are meant to represent *points of synchronization*. That is, a join node is a task J that, before allowing the execution of any of its children, waits for the completion of all active threads that have J as an endpoint. This particular property is very specific to workflow graphs, which we call *synchronization property*.

However, not any split-join pattern is permitted. Every split node T has also to obey the following constraints in an AO graph:

- there must be a node that is a descendant of all children of T. The end node obviously is one such node. Among all such nodes, we assume there is a *unique* minimal one that is not a descendant of any other such node. There may also be a node that is a descendant of more than one, but not all children of T. We call such a node a *partial join* for T;
- Let S_1 and S_2 be any two directed chains from T to a node V that only intersect at T and V. Then all nodes in G that are descendants of nodes in $S_1 \cup S_2 \backslash \{T\}$ are either ancestors of V, or descendants of V.

This property is desirable in order to give join nodes the semantics of real synchronization tasks, i.e., join nodes as tasks that finalize threads started by the most recent split node. It essentially enforces *nesting* of threads. A case where this assumption is not respected is illustrated by Figure 1.

These constraints are the most characteristic constraints of workflow graphs adopted in the literature, and provide distinctive features to be explored by workflow mining algorithms.

2.2 A PARAMETRIC MODEL OF WORKFLOW GRAPHS

Each task T is an event. It either happens or it does not happen. By an abuse of notation, we will use the same symbols to represent binary random variables and task events,

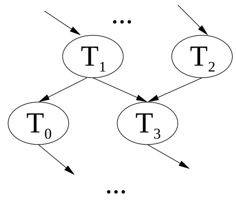

Figure 1: This construction is not allowed because T_1 creates another thread that is not nested between the split point that generated $\{T_1, T_2\}$ and its synchronization point T_3.

where $T = 1$ represents the event "*T happened*". We define a parametric model for a DAG by the conditional probability of each node given its parents, i.e. by assuming the Markov condition (Spirtes et al., 2000). There is, however, a special logical constraint in workflow graphs.

Let an *OR-split* be a split node that forces a unique choice of task to be executed among its children, i.e., all of its children are mutually exclusive. Any other type of split node is called an *AND-split*[1]. Children of OR-splits will have a special parameterization.

Let Pa_T represent the parents of task T in an AO graph G. By another abuse of notation, let Pa_T also be a random variable representing the joint state of the parents of a task T, i.e., $Pa_T = j$ is a particular combination of binary assignments to the elements of Pa_T. In particular, $Pa_T = 0$ represents the event where all parents of T are assigned the value 0. The basic parametetrization is as follows:

- if T is not a child of an OR-split, $P(T = 1|Pa_T = j) = \Theta_{tj} < 1$ for $j > 0$, and $P(T = 1|Pa_T = 0) = 0$.
- if T is a child of an OR-split, then by assumption Pa_T has an unique element V. Let $Choice(V)$ be an auxiliary multinomial random variable in $\{1, ..., c\}$, where c is the number of children of V. Each $Choice(\bullet)$ random variable has its own multinomial distribution, where the domain of this function is the set of OR-splits of G. Finally, define T as being the ith child of Pa_T. Then $P(T = 1|Pa_T = 1, Choice(Pa_T) = i) = \Theta_t < 1$, and 0, otherwise;

The requirement that $P(T = 1|Pa_T = 0) = 0$ encodes the modeling assumption that a necessary condition for a task to be executed is that at least one of its parents is executed. We call this property *backward determinism*, typically present in real-world processes [7][2]. Also important, backward determinism will allow us to design an algorithm to learn workflow graphs *in polynomial time*.

2.3 TEMPORAL INFORMATION

We assume that the data available for our learning algorithm is a *workflow log* [1]. A workflow log consists of records

[1]This is an unfortunate choice of names, since OR-splits actually behave as XOR operators, while an AND-split is technically an OR choice. We adopt this denomination since it is already widespread in this field.

[2]The assumption $\Theta_{tj} < 1$ is not an essential assumption and was introduced here for the purposes of simplifying the presentation. It does capture the common phenomenon that any process can be aborted non-deterministically.

of which tasks were performed for which process instances at which starting time. For example, the following log

$WorkflowLog = \{(Car_1, BuildChassis,$ 09:10$am),$
$(Car_2, BuildDoors,$ 10:17$am),$ $(Car_2, AddSeats,$ 10:20$am),$
$(Car_1, Build\ Doors,$ 10:47$am)\}$

contains information concerning two *instances* (Car_1 and Car_2) going through a series of *tasks* ($BuildChassis$, $Build\ Doors$, $AddSeats$) starting at differente *times*.

Workflow logs are by-products of *workflow management systems* [7]. We assume that our data are workflow logs.

2.4 HIDDEN VARIABLES AND NOISE

We allow the possibility that non-simple nodes can be hidden variables (i.e., split or join nodes might not be recorded at all in the log). However, for identification purposes, we make the following assumptions:

1. no hidden AND-split is a child of a hidden AND-split, and no hidden OR-split is a child of a hidden OR-split;
2. no hidden task is both a split and join node;
3. no hidden join is followed by a simple task and no hidden OR-split follows a simple task, where there are no hidden partial joins;

These assumptions do not restrict the ability of the AO graphs to represent any combination of sequential, parallel or exclusive patterns that appear in practice. Mathematically, however, they assure that any AO graph can be distinguished from any other AO graph given enough data, as it will be explained in Section 3. Furthermore, we allow the possibility of *measurement error*. For each task T that is measurable, we account for the possibility that T is not recorded in a particular instance even though T happened. That is, let T_M be a binary variable such that $T_M = 1$ if task T is recorded to happen. Then we have the following measurement model:

- $P(T_M = 1|T = 1) = \eta_{TM} > 0$
- $P(T_M = 1|T = 0) = 0$

Note that we assume measurement error happens only in one direction. Although that might not be the case in every application, this greatly simplifies our problem, and will allow us to learn the structure of workflow graphs without fitting latent variable models.

In this sense, every task is hidden. However, in this paper, the name "hidden task" will be applied only to tasks that cannot be measured at all. The description of a workflow model as a specialized hidden Markov model will be treated in Section 5. Notice also that for every OR-split T in G, $Choice(T)$ is a hidden variable, and will not be explicitly represented in AO graphs, unlike hidden splits and joins.

To identify hidden AND-splits, we need to assume that the immediate observable descendants of a hidden AND-split T (i.e., those that do not have an observable proper ancestor that is a descendant of T) should not be tied by any temporal constraint, i.e., given observable descendants T_1 and T_2, the probability that T_1 is executed (starts) before T_2 is positive.

We assume that there is also a fixed measurement noise for the temporal ordering information. For each pair of tasks T_1, T_2, there is some probability ϵ that T_1 is recorded before T_2 even though in the true workflow graph T_2 is an ancestor of T_1. We will assume that the noise level is the same for each pair.

2.5 STRUCTURAL INDEPENDENCE

The Markov condition gives us a way of parameterizing a probabilistic model as a AO graph. If one is interested in calculating the effect of a new policy that changes the probability distribution of some specific set of tasks, then the Causal Markov condition needs to be assumed [6].

If one is interested in a learning algorithm that will recover the right structure, at least asymptotically, we have to have some extra assumptions linking the probabilistic distribution of the tasks to the corresponding graphical structure. For the general case of learning the structure of DAGs, a sufficient condition for consistent learning is the faithfulness condition. This condition states that a conditional independence statement holds in the probability distribution if and only if it is entailed in the respective DAG by d-separation [6].

We want a similar assumption, because observed conditional independencies can provide information about the workflow graph underlying the data, but only if conditional independencies are a result of the workflow structure (i.e., if they are entailed by the workflow graph). We cannot just assume faithfulness to d-separation: due to backward determinism, a chain such as $T_1 \rightarrow T_2 \rightarrow T_3$ encodes that T_2 is independent of T_1 given $T_3 = 1$ (because if T_3 happened, then by assumption T_2 happened, which means that T_1 does not add any information concerning the distribution of T_2), but T_2 is not d-separated from T_1 given T_3.

Instead, we assume a variation of faithfulness. First, two definitions: an *augmented* AO graph is a modification of a AO graph G such that, for each OR-split T we introduce a new node, $Choice(T)$, as a child of T, and make every original child of T a child of $Choice(T)$ only. We denote the augmented version of G by $Augmented(G)$. Also, given $Augmented(G)$, we say that task A is a *sure-ancestor* of task B if for every ancestor C of B, C is an ancestor of A and A d-separates C and B, or A is an ancestor of C. We then assume that T_i is independent of T_j given a set of tasks $\mathbf{T}$ if and only if either of the following situations hold in the workflow graph G associating such tasks:

- T_i and T_j are d-separated given $\mathbf{T}$ in $Augmented(G)$;
- T_i and T_j are d-separated given a sure-ancestor of some $T_k \in \mathbf{T}$ such that $T_k = 1$;
- T_i (or T_j) is a sure-ancestor of some $T_k \in \mathbf{T}$ such that $T_k = 1$;

The idea embedded in faithfulness is that conditional independences should be given by the graphical structure, not by the particular choice of parameters defining the probability of a task being accomplished. Sure-ancestry entails independencies because in an AO graph G, if A is a sure-ancestor of B, then $P(A = 1|B = 1) = 1$ in any probability model parameterized by G.

3. LEARNING AO GRAPHS

Assume for now we have an *ordering oracle* O for a workflow graph G such that $O(T_1, T_2)$ returns $true$, $false$ or $exclusive$ as follows:

- if T_1 and T_2 are immediate observable descendants of an AND-split, then $O(T_1, T_2) = O(T_2, T_1) = true$;
- if T_1 is an ancestor of T_2, then $O(T_1, T_2) = true$;
- if $O(T_1, T_2) = true$, then T_2 is not an ancestor of T_1;
- $O(T_1, T_2) = exclusive$ if and only if T_1 and T_2 are mutually exclusive;

Notice that according to this oracle it is possible to have $O(T_1, T_2) = true$ even though T_1 is not an ancestor of T_2, as long as T_2 is not an ancestor of T_1.

Analogously, assume for now we have an independence oracle I for a workflow graph G such that $I(T_i, T_j, T_k)$ is true if and only if T_i and T_j are independent given $T_k = 1$. The motivation for defining such oracles is given by the following theorem:

THEOREM 1. *Let G_1 and G_2 be two AO graphs with respective ordering and independence oracles $\{O_1, I_1\}$ and $\{O_2, I_2\}$ over a same set of observable tasks* $\mathbf{T}$*. If O_1 and O_2, and I_1 and I_2 agree on all queries concerning members of* $\mathbf{T}$*, then $G_1 = G_2$ up to a renaming of the hidden tasks.*

The proof of this theorem is given in Appendix A. In simple terms, given certain *partial* information of ordering and conditional independences among the observable tasks, one is able to uniquely recover the proper AO graph.

3.1 MAIN ALGORITHM

With these oracles, we claim that the algorithm *LearnOrderedWorkflow*, given in Figure 2, will return the correct workflow structure.

This algorithm makes references to other sub-algorithms given in Section 3.2. We will first provide a higher-level description of its steps. The algorithm works by iteratively adding child nodes to a partially built graph in a specific order. Initially, the ordering oracle will tell us which nodes are "root causes" of all other measurable tasks, i.e., which nodes do not have any measurable ancestor. Such nodes are identified in Step 3 of Figure 2. If we have more than one measurable node as a "root cause", and because an AO graph requires a single starting point and explicit control nodes (i.e., AND-splits and OR-splits), it is the case that unobserved splits have to be added to the graph. This is done by $HiddenSplits$.

At each main iteration (Steps 7 - 12), we have a set of nodes called **CurrentBlanket**, which contains all and only the "leaves" of the current workflow graph H, i.e., all the task nodes that do not have any children in H. The initial choice of nodes for **CurrentBlanket** are exactly the root causes. The next step is to find which measurable tasks should be added to H. We are interested in building the graph by selecting only a set of tasks **NextBlanket** such that:

- there is no pair (T_1, T_2) in **NextBlanket** where T_1 is an ancestor of T_2 in G;
- no element in **NextBlanket** has an ancestor in G that is not in H;
- every element in **NextBlanket** has an ancestor in G that is in H;

Algorithm *LearnOrderedWorkflow*
Input: O, an ordering oracle for a set $\mathbf{T}$ of tasks; I, an independence oracle for $\mathbf{T}$;
Output: H, an AO graph

1. Let H and G_O be two empty graphs, where H has no nodes and $\mathbf{T}$ are the nodes of G_0
2. For every pair of tasks T_i and T_j such that $O(T_1, T_2)$ if $true$ but not $O(T_2, T_1)$, add the edge $T_1 \rightarrow T_2$ to G_0
3. Let **CurrentBlanket** be the subset of $\mathbf{T}$ whose elements do not have a parent in G_O
4. Add nodes in **CurrentBlanket** to H
5. $H \leftarrow HiddenSplits(H, \mathbf{CurrentBlanket}, O)$
6. $G_O \leftarrow G_O - \mathbf{CurrentBlanket}$
7. While G_O has nodes
8. $\mathbf{NextBlanket} \leftarrow GextNextBlanket(\mathbf{CurrentBlanket}, G_O, O, I)$
9.   Add nodes in **NextBlanket** to H
10. $Ancestors \leftarrow Dependencies(\mathbf{CurrentBlanket}, \mathbf{NextBlanket}, O, I)$
11. $H \leftarrow InsertLatents(H, \mathbf{CurrentBlanket}, \mathbf{NextBlanket}, Ancestors, O)$
12. $G_O \leftarrow G_O - \mathbf{NextBlanket}$
13. Let **CurrentBlanket** be the subse of $\mathbf{T}$ whose elements do not have a child in H
14. $H \leftarrow HiddenJoins(H, \mathbf{CurrentBlanket}, O)$
15. Return H

Figure 2: An algorithm for learning AO graphs.

We claim that *GetNextBlanket*, as described later, returns a set corresponding to these properties. We still need to identify which elements in **NextBlanket** should be descendants of which elements in **CurrentBlanket**, and this is accomplished by *Dependencies*.

It is quite possible that between nodes in **CurrentBlanket** and nodes in **NextBlanket** there are several hidden join/split tasks. Such tasks are detected and added to H by $InsertLatents$.

This procedure is iterated till all observable tasks are placed in H. To complete the graph, we just have to make sure that all tasks are synchronized in a finalization task, as required by all AO graphs. If the end task is not visible, several threads will remain open if we do not add latent joins. This is accomplished by the final $HiddenJoins$ call. A sample execution of this algorithm is given in Appendix B.

3.2 ALGORITHM DETAILS

While mutually exclusive tasks are directly identifiable from the ordering oracle, this is not true concerning parallel tasks. If two tasks are potentially parallel, they still might be executed always in the same order. The only way we can identify parallelism is by identifying a previous task that make these two tasks independent. This is the purpose of algorithm *GetNextBlanket*, as described in Figure 3.

This algorithm select tasks, but does not indicate which elements are descendants of which previous tasks. This is the role of *Dependencies* (Figure 4). The fact that the independence oracle condition only positive values of T_{2M} (Step 3 of *Dependencies*) is a necessary and sufficient condition.

Algorithm *GetNextBlanket*

Input **CurrentBlanket**, a set of tasks;
G_O, a DAG encoding ancestral relationships;
O, an ordering oracle;
I, an independence oracle;

Output **NextBlanket**, a subset of the tasks in G_O

1. For every pair of adjacent tasks (T_1, T_2) in G_O
2. Remove the edge between T_1 and T_2 if and only if $I(T_{1M}, T_{2M}, T_{iM})$, where T_{iM} is the measure of some task $T_i \in$ **CurrentBlanket** and $O(T_i, T_1) \neq exclusive$, $O(T_i, T_2) \neq exclusive$
3. Return all nodes from G_O that do not have parents

Figure 3: Identifying the next set of elements to be added.

It is necessary because by our assumptions there might be measurement error when we observe value 0. It is sufficient because by backward determinism (if a task happens, all elements in a chain before it also happened), we do not need to condition on multiple tasks. Figure 5 illustrates an example of this case.

Algorithm *Dependencies*

Input **CurrentBlanket**, a set of tasks;
NextBlanket, another set of tasks;
O, an ordering oracle;
I, an independence oracle;

Output *AncestralGraph*, a DAG

1. Let *AncestralGraph* be a graph with nodes in **CurrentBlanket** $\cup$ **NextBlanket**
2. For every task T_0 in **NextBlanket**
3. For every task T_1 in **CurrentBlanket**, add edge $T_1 \rightarrow T_0$ to *AncestralGraph* if and only if:

   i. $O(T_0, T_1) \neq exclusive$

   ii. There is no task $T_2 \in$ **CurrentBlanket** s.t. $O(T_1, T_2) \neq exclusive$, $O(T_0, T_2) \neq exclusive$, and $I(T_{0M}, T_{1M}, T_{2M})$, where T_{iM} is the measure of task T_i;
4. Return *AncestralGraph*

Figure 4: Determining ancestors for a set of new tasks.

This algorithm runs in $O(N^3)$, N being the number of measurable tasks. It also requires simpler statistical tests of conditional independence than general DAG search algorithms, since we condition only on singletons.

Finally, there are several points in $LearnOrderedWorkflow$ where we need to introduce hidden tasks. The algorithm $HiddenJoins$ is shown in Figure 6. Notice that here we tag nodes according to their role ("AND-join" and "OR-join"). We do not show an explicit description of $HiddenSplits$: this algorithm is analogous, with the exception that edges are added in the opposite direction. It is very similar in principle to an algorithm given by [4]. The algorithm $InsertLatents$ builds upon $HiddenJoins$ and $HiddenSplits$. It is given in Figure 7. The final steps of this algorithm just verify if a measurable task that has measurable children actually d-separates them. If not, a hidden task is introduced.

3.3 PRACTICAL IMPLEMENTATION

The independence oracle can be implemented by statistical tests of independence, such as the χ^2 test. Given the

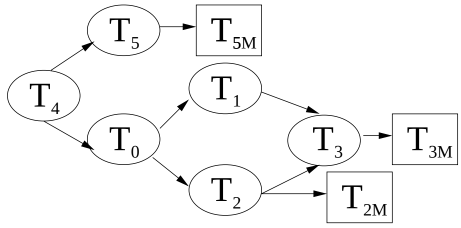

Figure 5: An example of why conditioning on a single element is enough. Here, T_{3M} and T_{5M} are independent measures given $T_{2M} = 1$. If T_{2M} is 1, by assumption we know that $T_2 = 1$, because measurement error is one-sided. T_0 is 1 by backward determinism, which means that we are effectively asking if T_{3M} and T_{5M} are independent given $T_0 = 1$, which is entailed by the graphical structure.

parameter ϵ for the noise level, binomial tests can be used to create an ordering oracle by testing if the probability of task T_i antecedes task T_j given the instances where both are recorded is larger than ϵ.

To learn a good level of ordering noise, one can do a grid search for ϵ over the interval $[0, 0.5]$ and heuristically choose the one that maximizes some measure of fitness, such as a posterior probability for the output model (using a Dirichlet prior for the parameters, for instance), or some other measure that relies on independence constraints only, which is the basis of our model. For instance, by adjusting ϵ one could try to bring the set of independence constraints that are entailed by the output graph as close as possible to the ones judged to hold in the data. This does not require fitting a latent variable model and is not subject to constraints other than independence constraints. Learning ϵ will be treated in detail in a future work.

An important practical issue is how to avoid outputing invalid AO graphs, which can be due to deviations from the assumptions or statistical mistakes. Due to lack of space, we omit a discussion of the necessary conditions that the ordering and independence oracles should satisfy to generate a valid AO graph.

4. EXPERIMENT

Workflow data is not as easy to obtain as other data sources. In this paper, we perform a simulated study based on a theoretical workflow that models the annual process of writing final reports at Clairvoyance Corporation. The process basically consists of parallel threads of preparing documents, preparing summaries, booking flights and hotel rooms for an annual workshop hosted by the parent company of Clairvoyance in Japan. The graph was constructed by manually analysing e-mail logs exchanged among the company's employees over the course of four projects. The details are given [5].

There are 15 observable and 2 hidden tasks, with no mutually exclusive tasks and no measurement noise (the algorithm still assumes the possibility of noise). One task (*Printing materials*) naturally happens much later than the actions of booking flights and hotels, even though there is no temporal constraint that dictates that printing should be performed only after travel is arranged. Many other work-

```
Algorithm HiddenJoins
Input    H, a DAG;
         S, a set of nodes;
         O, an ordering oracle;
Output   H, a DAG

(H, NewJoin) ← JoinStep(H, S, O)
Return H
```

```
Algorithm JoinStep

If S has only one element S_0
   Return (H, S_0)
Let M be a graph having elements of S as nodes,
and with an undirected edge between a pair of
nodes {S_1, S_2} if and only O(S_1, S_2) ≠ exclusive
Let NewLatent be a new latent node, and
add it to H
If M is disconnected
   M' ← M
   Tag NewLatent as "OR-join"
Else
   M' ← the complement of M
   Tag NewLatent as "AND-join"
For each component C of M'
   If C has only one node C_0
      Add edge C_0 → NewLatent to H
   Else
      (H, NextLatent) ← JoinStep(H, C, O)
      Add edge NextLatent → NewLatent to H
Return (H, NewLatent)
```

Figure 6: An algorithm for inserting required join nodes.

flow approaches [8] would be deceived by this temporal information, i.e., they would regard the two tasks as strictly sequential when in fact they are not.

The graph is parameterized by a single parameter α that gives the probability of a task being executed given its prerequisites. In our model, a necessary condition for any task is that all of its parents have to be performed. We simulated samples of size of 100, 200 and 500 and with $\alpha = \{0.9, 0.95\}$[3]. We do not introduce noise in the time order of the samples, since this will only be explored in full detail in the future.

The independence oracle is implemented by a χ^2 test using a significance level of 0.05. We ran 10 trials for each configuration, and evaluated the true model against the output of our algorithm, assuming the ordering information is correct, by the following criteria: number of edges between measurable tasks in the true graph that are not in the estimated graph (edge omission, out of 12 possible edges); number of edges between measurable tasks in the estimated graph that are not in the true graph (edge omission); number of measurable pairs that share a common parent in the true graph but not in the estimated graph (sibling omission). Sibling comissions did not happen in our experiments. The results are: for sample size 100 and $\alpha = 0.95$, the average edge omission was 5.1 (2.1 of standard deviation); the average edge comission was 1.7(0.7) and the average sibling omission was 2.6(1.4). For sample size 100, $\alpha = 0.9$, we had 4.9(2.6), 0.7(1.1) and 2(1.4). For sample size 200, and $\alpha = 0.95$, we got 0.4(0.5), 0.1(0.3), 0.1(0.3). For sample size 200 and $\alpha = 0.9$, we got edge omission error of 0.2(0.4) and no other error. For sample size 500, we got the exact graph in all 10 trials for both values of α. In the experiments, missing edges usually implied sequential tasks being treated as parallel.

[3]The value of α cannot be too small, or otherwise we will need large sample sizes in order to have a relatively large number of instances that are completed. Workflows with large chains will usually have some deterministic steps.

```
Algorithm InsertLatents
Input    H, a DAG H;
         CurrentBlanket, NextBlanket, two sets;
         AncestralGraph, a DAG;
         O, an ordering oracle;
Output   a DAG H

For every task T ∈ NextBlanket
   Let Siblings be the set of elements in
   NextBlanket that have a common parent
   with T in AncestralGraph
   Let AncestralSet be the set of parents of
   Siblings in AncestralGraph
   (H, JoinNode) ← HiddenJoins(H, AncestralSet, O)
   (H, SplitNode) ← HiddenSplits(H, Siblings, O))
   Add edge JoinNode → SplitNode to H
   NextBlanket ← NextBlanket − Siblings
For every set C of observable tasks, |C| > 1,
that are children of a single hidden node Pa_H
that is child of an observable task Pa in H
   If all pairs in C_M are independent conditioned on
   Pa_M = 1, C_M being the set of respective measures
   of C and Pa_M the measure of Pa,
      Add edges Pa → C_i for every C_i ∈ C
      Remove latent Pa_H
Return H
```

Figure 7: An algorithm to introduce required hidden tasks between two layers of measurable tasks.

The results are convincing, but it is still of interest to obtain more robust outcomes with smaller sample sizes. We plan to pursue Bayesian approaches in an extended version of this framework.

5. RELATED WORK

Agrawal et al. [1] introduced the first algorithm for mining workflow logs. Greco et al. [3] approach the problem using clustering techniques. A broad survey on the current work in workflow mining, or process mining, is given by van der Aalst and Wejters [8]. None of the approaches in that survey are based on a coherent probabilistic model. Instead, they use a variety of heuristics to deal with noise, while focusing on deterministic models such as Petri nets.

Herbst and Karagiannis [4] use a representation very similar to AO graphs with cycles. While some probability distribution is informally applied to define the likelihood of a workflow graph, this likelihood is not used anywhere in learning the structure of workflow graphs as defined in our paper.

It is clear that workflow models could be represented by off-the-shelf methods such as dynamic Bayesian networks and stochastic Petri nets. In particular, the factorial hidden Markov model [2] seems to naturally apply to the problem of modeling parallel threads of tasks. However, workflow modeling has its own particular issues that are not efficiently explored by generic dynamic Bayesian networks: instances have a well defined beginning and end; the synchronization property; backward determinism, which naturally applies to

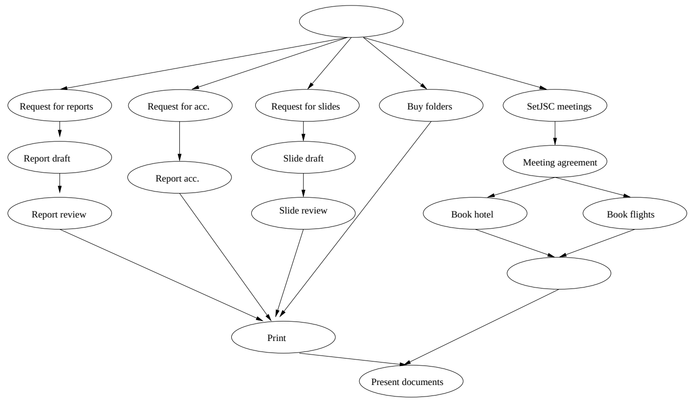

Figure 8: A simplified workflow model of the process of document preparation at Clairvoyance Corporation. ("Acc." stands for "accomplishments", and JSC refers to Clairvoyance's parent company.)

many real-world problems; the fact that the "hidden states" of a workflow model are in general associated with one "visible symbol" only. Even if a same task might be generated under different contexts, as explored by [4], this is the exception, not the rule, and it seems wasteful to arbitrarily allow hidden states of a workflow-like dynamic system to be able to generate any symbol. A generic dynamic model would not be as statistically efficient as a constrained model.

Moreover, one is often interested in understanding the causal chains of a business process. For instance, a generic factorial hidden Markov model with a fixed number of chains would be a very opaque model to provide such understanding, even if the fit is good.

6. CONCLUSION

We have presented an algorithm for learning workflow graphs that makes use of a coherent probability model. To the best of our knowledge, this is the first approach with such a property. Results from a real world workflow are very encouraging.

Several extensions are planned for a near future: more extensive experiments, learning with cycles, showing consistency of the learning algorithm and Bayesian variations. A very interesting problem is to determine identifiability conditions for learning semantic roles for tasks, i.e., how tasks can appear in multiple parts of a workflow model depending on context. Ultimately, we also want to extract a task ontology from text data obtained from groupware and e-mail software, therefore creating workflow logs from free text data.

7. REFERENCES

[1] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from work-flow logs. *Proc. of 6th International Conference on Extending Database Technology*, pages 469–483, 1998.

[2] Z. Ghahramani and M. Jordan. Factorial hidden Markov models. *Machine Learning*, 29, 1997.

[3] G. Greco, A. Guzzo, L. Pontieri, and D. Sacca. Mining expressive process models by clustering workflow traces. *Proc. of the 8th PAKDD*, 2004.

[4] J. Herbst and K. Karagiannis. Workflow mining with InWoLvE. *Computers and Industry*, 53:245–264, 2004.

[5] R. Silva, J. Zhang, and J. G. Shanahan. Probabilistic workflow mining. *Clairvoyance Corportation Technical Note CC-TN-04-20*, http://www.cs.cmu.edu/∼rbas/workflow.ps, 2004.

[6] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction and Search*. Cambridge University Press, 2000.

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

[8] W. van der Aalst and A. Wejters. Process mining: a research agenda. *Computers and Industry*, 53:231–244, 2004.

APPENDIX

A. PROOF OF THE THEOREM 1

THEOREM 1. *Let G_1 and G_2 be two AO graphs with respective ordering and independence oracles $\{O_1, I_1\}$ and $\{O_2, I_2\}$ over a same set of observable tasks $\mathbf{T}$. If O_1 and O_2, and I_1 and I_2 agree on all queries concerning members of $\mathbf{T}$, then $G_1 = G_2$ up to a renaming of the hidden tasks.*

We will do induction on the number of observable tasks to prove the proposition. For that purpose, we need a few lemmas. The first two lemmas show that the start tasks in the two graphs are identical. Let s_1 be the start task in G_1, and s_2 the start task in G_2.

LEMMA 1. *Either $s_1 = s_2 \in \mathbf{T}$, or they are both hidden tasks.*

Proof One of the following two cases must obtain.
Case 1: s_1 and s_2 are both observable. Then s_1 is the unique common predecessor of all observable tasks according to O_1, and s_2 is the unique common predecessor of all observable tasks according to O_2. Because O_1 and O_2 agree, $s_1 = s_2$.
Case 2: One of them, say s_1 without loss of generality, is hidden, then by our assumption it must be a split and there is NO observable task that is a common predecessor of all observable tasks according to O_1 (this follows from our assumption about the immediate observable descendants of an AND-split). Because O_1 and O_2 agree, there is no common observable predecessor according to O_2. It follows that s_2 is not observable.
Therefore, either $s_1 = s_2 \in \mathbf{T}$, or they are both hidden. □

LEMMA 2. *s_1 is an AND-split iff. s_2 is an AND-split. Similarly, s_1 is an OR-split iff. s_2 is an OR-split.*

Proof By Lemma 1, we only need to consider two cases:
Case 1: s_1 and s_2 are both hidden. It suffices to show that it cannot be the case that one of them is an OR-split while the other is an AND-split. For the sake of contradiction, suppose, without loss of generality, s_1 is an OR-split and s_2 is an AND-split. Then there exist two immediate observable descendants of s_1, $T_1, T_2 \in \mathbf{T}$, that are mutually exclusive according to O_1. Because O_2 agrees with O_1, T_1 and T_2 are also immediate observable descendants of s_2 in G_2, which means they are in the split-join session initiated by s_2 in G_2. Furthermore, they are also mutually exclusive according to O_2, so they cannot belong to different threads in that split-join session, since s_2 is an AND-split. So there must be another immediate observable descendant of s_2, $T_3 \in \mathbf{T}$, such that it is in parallel with both T_1 and T_2 according to O_2. It follows that T_3 is in the split-join session initiated by s_1 in G_1, and is in parallel with both T_1 and T_2 according to O_1. But this is impossible, because T_1 and T_2 are in

different threads of that OR-split-join session initiated by s_1. Hence either they are both AND-splits, or they are both OR-splits.
Case 2: $s_1 = s_2 = T \in \mathbf{T}$. By symmetry, we only need to rule out three scenarios: (i) T is an OR-split in G_1 but an AND-split in G_2; (ii) T is an OR-split in G_1 but a simple task in G_2; (iii) T is an AND-split in G_1 but a simple task in G_2. (i) can be ruled out by rehearsing the arguments in case 1. In the case of (ii) and (iii), notice that T may not be followed by an observable task in G_2, for otherwise that observable task will be the unique common predecessor of all observable tasks but T according to O_2 but will not be such according to O_1. Furthermore, by our assumption, T, as a simple task in G_2, may not be followed by a hidden OR-split, so it can only be followed by a hidden AND-split in G_2. (ii) can thus be ruled out by rehearsing the arguments in case 1, since T is an OR-split in G_1. For (iii), notice that some immediate observable descendants of T will be independent conditional on $T = 1$ according to I_1, but dependent conditional on $T = 1$ according to I_2. Hence (iii) contradicts the assumptions, too. □

Suppose, for the moment, that s_1 and s_2 are both splits. Let j_i be the (full) join that synchronizes the split initiated by s_i in G_i, $i = 1, 2$. We define a *thread* of the split-join session between s_i and j_i to be the subgraph between s_i and any parent of j_i (over the ancestors of that parent of j_i). A thread, under this definition, can contain any number of (observable) partial joins of the split initiated by s_i. It is easy to see that each thread is either an AO graph or of the simple form $s_i \rightarrow T$, where s_i is hidden. Furthermore, by our enforcement of nesting of splits and joins, it is easy to see that different threads will only intersect at the starting point s_i. The next two lemmas concern the observable tasks that appear in the split-join session, and in particular, in each thread of the session.

LEMMA 3. *Suppose s_1 and s_2 are both splits. For any $T \in \mathbf{T}$, T is in the split-join session initiated by s_1 in G_1 iff. T is in the split-join session initiated by s_2 in G_2.*

Proof Let $\mathbf{IOD}_i$ be the set of immediate observable descendants of s_i in G_i, $i = 1, 2$. Because O_1 and O_2 agree, $\mathbf{IOD}_1 = \mathbf{IOD}_2$. Hereafter we will drop the subscripts and write $\mathbf{IOD}$. By our assumption, any member in $\mathbf{IOD}$ must be in the split-join session initiated by s_i, otherwise there exists some observable task that lies in between. So for any $T \in \mathbf{T}$, if $T \in \mathbf{IOD}$, then it is in the split-join session in G_1 iff. it is in the split-join session in G_2. If $T \notin \mathbf{IOD}$, there are two cases to consider: (i) s_1 and s_2 are both AND-splits. In this case, if T is in the session initiated by s_1 in G_1 but not in the session initiated by s_2 in G_2, then there exist $T_1, T_2 \in \mathbf{IOD}$ such that T is independent of T_2 conditional on T_1 according to I_1, but T is dependent of T_2 conditional on T_1 according to I_2. (Specifically, let T_1 be an immediate observable descendant of s_1 in the same thread as T is in G_1, and T_2 be an immediate observable descendant of s_1 in any other thread.) Hence a contradiction. By symmetry, it may not be the case either that T is in the session initiated by s_2 in G_2 but not in the session initiated by s_1 in G_1. (ii) s_1 and s_2 are both OR-splits. In this case, if T is in the session initiated by s_1 in G_1 but not in the session initiated by s_2 in G_2, then T will be mutually exclusive with some member in $\mathbf{IOD}$ according to O_1, but will not be mutually exclusive with any member in $\mathbf{IOD}$ according to O_2. Hence a contradiction. By symmetry, it may not be the case either that T is in the session initiated by s_2 in G_2 but not in the session initiated by s_1 in G_1. □

LEMMA 4. *Suppose s_1 and s_2 are both splits. For any $T_1, T_2 \in \mathbf{T}$ that are in the split-join session initiated by the start task in both graphs, they are in a same thread of that session in G_1 iff. they are in a same thread of that session in G_2.*

Proof Let $\mathbf{IOD}$ be the set of immediate observable descendants of s_1 (and s_2) according to O_1 (and O_2). By Lemma 2, we only need to consider two cases:
Case 1: s_1 and s_2 are both AND-splits. We first show that if $T_1, T_2 \in \mathbf{IOD}$, then it is not the case that they are in the same thread in one of the graphs but not in the other graph. Suppose otherwise and, without loss of generality, that T_1 and T_2 are in the same thread in G_1 but not in the same thread in G_2. It follows that $O_1(T_1, T_2)$ and $O_1(T_2, T_1)$ are both true. Because O_1 agrees with O_2, we also have $O_2(T_1, T_2)$ and $O_2(T_2, T_1)$. Since T_1 and T_2 belong to the same thread initiated by s_1 in G_1 and are both in $\mathbf{IOD}$, there must be an OR-split that lies between s_1 and T_1, T_2, as an AND-split cannot immediately follow another AND-split. This implies that there exists $T_3 \in \mathbf{IOD}$ that is mutually exclusive with both T_1 and T_2 according to O_1. However, because T_1 and T_2 belong to different threads initiated by s_2, an AND-split, in G_2, it is impossible that a task can be mutually exclusive with both of them according to O_2. Hence a contradiction. Thus, if $T_1, T_2 \in \mathbf{IOD}$, then they are in a same thread in G_1 iff. they are in a same thread in G_2.
Now suppose at least one of them, say T_1 without loss of generality, is not in $\mathbf{IOD}$. If T_1 and T_2 belong to different threads in G_1, then there exists $T_3 \in \mathbf{IOD}$ such that T_1 and T_3 are in parallel and T_1 is independent of T_2 conditional on T_3 according to I_1. On the other hand, if T_1 and T_2 belong to the same thread in G_2, the only way that T_3 could render them independent is that T_3 and T_2 are two children of an OR-split, but in that case they will be mutually exclusive. So T_1 and T_2 must belong to the same thread in G_2, too. By symmetry, the converse also holds.
Case 2: s_1 and s_2 are both OR-splits. If T_1 and T_2 belong to different threads in G_1, then they are mutually exclusive according to O_1, which means they are also mutually exclusive according to O_2. So, if on the other hand T_1 and T_2 belong to the same thread in G_2, then there must be an AND-split in between s_2 and the (yet another) OR-split that splits T_1 and T_2 because an OR-split cannot immediately follow another OR-split. This implies that there exists T_3 such that it is not mutually exclusive with either T_1 or T_2 according to O_2. However, because T_1 and T_2 belong to different threads in G_1, it is impossible that T_3 is not mutually exclusive with either T_1 or T_2 according to O_1. Hence a contradiction. □

Finally, we need a lemma about j_i's that complete the split-join sessions initiated by s_i's.

LEMMA 5. *Suppose s_1 and s_2 are both splits. Let j_1 be the (full) join that synchronize the splits initiated by s_1 in G_1, and j_2 be the (full) join that synchronize the splits initiated by s_2 in G_2. Then either j_1 and j_2 are the same observable task or they are both hidden.*

Proof Two cases to consider:
Case 1: Suppose j_1 and j_2 are both observable. So j_i is the descendant of all observable tasks within the split-join session initiated by s_i and the ancestor of all other observable tasks, $i = 1, 2$. By Lemma 3, the set of observable tasks within the split-join session initiated by s_1 is the same as the set of observable tasks within the split-join session initiated by s_2. It follows that $j_1 = j_2$, otherwise O_1 does not totally agree with O_2.
Case 2: Suppose one of them, say j_1 without loss of generality, is hidden. In this case, if j_2 is observable, then j_2 must immediately follow j_1 in G_1, otherwise O_1 and O_2 do not agree. By our assumption, j_2 may not be a simple task. If it is an OR-split, then in G_2 a hidden-OR must immediately follow j_2 (by arguments very similar to those in previous lemmas), which, however, is ruled out by our assumption. If j_2 is an AND-split in G_1, then some tasks after j_2 will be independent conditional on j_2 according to I_1, but dependent conditional on j_2 according to I_2. A contradiction. Therefore, j_2 must be hidden, too. □

We now prove the main proposition by induction on the number of observable tasks n. It is easy to see that $n \geq 2$ by our assumptions.
Base case: $n = 2$. Let T_1 and T_2 be the two observable tasks. Only four AO graphs are compatible with our assumptions (up to a renaming of latent tasks): (1) $T_1 \rightarrow T_2$; (2) $T_2 \rightarrow T_1$; (3) T_1 and T_2 are two threads of an AND split-join session with a hidden split (start task) and a hidden join (end task); (4) T_1 and T_2 are two threads of an OR split-join session with a hidden split (start task) and a hidden join (end task). Obviously each graph entails a different ordering relationship between T_1 and T_2. So, if O_1 and O_2 agree, then G_1=G_2 up to a renaming of the hidden tasks.
Inductive Step: Suppose the proposition holds for $n \leq m$. Let $n = m + 1 \geq 3$. There are three cases:
Case 1: s_1 is a simple task in G_1. By Lemmas 1 and 2, $s_1 = s_2 = T$ and T is also a simple task in G_2. It is easy to see that the subgraph of G_1 over $\mathbf{T}\backslash\{T\}$ and the subgraph of G_2 over $\mathbf{T}\backslash\{T\}$ are also AO graphs (since $n \geq 3$). By the inductive hypothesis, they are identical up to a renaming of hidden tasks. It follows that $G_1 = G_2$ up to a renaming of hidden tasks.

Case 2: s_1 is a split, and the split is joined before reaching the end task in G_1. By Lemmas 1, 2 and 5, s_2 is also a split, and the split is joined before reaching the end task in G_2. Let $\mathbf{T}_i$ be the set of observable tasks that belong to the split-join session initiated by s_i in G_i (including the initial split and the final join), $i = 1, 2$. It follows from Lemmas 1, 2, 3 and 5 that $\mathbf{T}_1 = \mathbf{T}_2$. By the inductive hypothesis, the subgraph of G_1 over $\mathbf{T}_1$ is the same as the subgraph of G_2 over $\mathbf{T}_2$ up to a renaming, and the subgraph of G_1 over $\mathbf{T}\backslash\mathbf{T}_1$ is the same as the subgraph of G_2 over $\mathbf{T}\backslash\mathbf{T}_2$ up to a renaming. (Note that there is a special case where the subgraphs over $\mathbf{T}\backslash\mathbf{T}_i$ only contain one observable task, and hence the inductive hypothesis is not applicable. But in that case, the two subgraphs are trivially identical.) It follows that $G_1 = G_2$ up to a renaming of hidden tasks.

Case 3: s_1 is a split, and the split is joined at the end task in G_1. By Lemmas 1, 2 and 5, s_2 is also a split, and the split is joined at the end task in G_2. By Lemma 3, for each thread of that split-join session in G_1, there is a thread of the split-join session in G_2 such that the two threads involve the exactly same observable tasks, and vice versa. By the inductive hypothsis, the two threads (subgraphs) are the same up to a renaming of hidden tasks. (Again, there is a special case where the inductive hypothesis is not applicable. That is, the threads are of the form $s_i \rightarrow T$, and s_i's are hidden. In this case the two subgraphs are trivially identical.) So in total $G_1 = G_2$ up to a renaming of hidden tasks. **Q.E.D**

B. AN ALGORITHMIC EXAMPLE

We will now go through an example of how *LearnOrdered Workflow* works. Assume for now that the graph G in Figure 9 corresponds to the true generative model, from which we know the ordering oracle O and the independence oracle I for tasks $\{1, \ldots, 12\}$. We will demonstrate how *LearnOrderedWorkflow* is able to reconstruct G out of O and I.

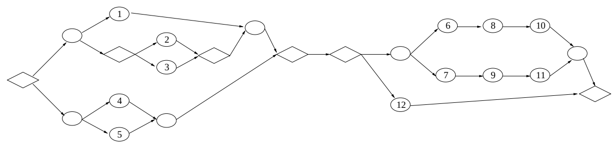

Figure 9: Unlabeled nodes represent hidden tasks. Each OR-split/join is represented as a rhombus.

Suppose that the directionality graph G_O is given in Figure 10. Notice that even though elements in $\{8, 10\}$ are concurrent to elements in $\{9, 11\}$, there is a total order among these elements: $8 \rightarrow 9 \rightarrow 10 \rightarrow 11$, according to O. 6 and 7 are not connected because by assumption they should happen in either order a frequent number of times. We consider this assumption to be reasonable (at the moment of the split, tasks should be independent, and therefore no fixed time order implied). However, contrary to a naive workflow mining algorithm, we do not require, for instance, that 6 and 11 are recorded in random orders. This type of assumption seems considerably more artificial, because tasks in one chain might take much longer than tasks in another chain, and a specific order may arise naturally.

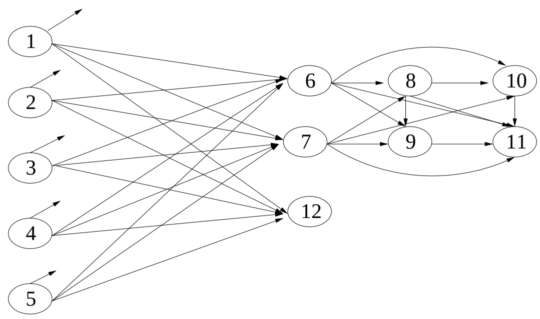

Figure 10: An ordering relationship for the graph in Figure 9. We do not represent explicitly the edges between elements in $\{1, 2, 3, 4, 5\}$ and $\{8, 9, 10, 11\}$ in order to avoid cluttering the graph (symbolized by the unconnected edges out of $\{1, 2, 3, 4, 5\}$).

In the initial step, the set **CurrentBlanket** will contain tasks $\{1, 2, 3, 4, 5\}$. The *HiddenSplits* algorithm will work as follows: a graph M will be created based on O and tasks $\{1, 2, 3, 4, 5\}$. M and its complemented are shown in Figure 11. Since M is disconnected, it will be the basis for the recursive call. We are going to insert an hidden OR-split separating $\{1, 2, 3\}$ and $\{4, 5\}$ at the return of the recursion, as depicted in Figure 12.

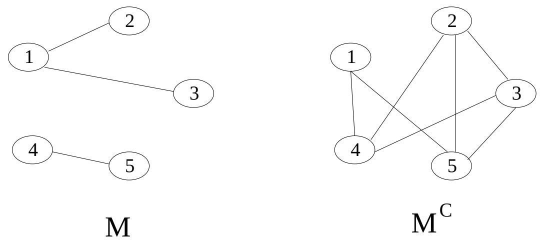

Figure 11: Graphs M and its complement M^C in *HiddenSplits* for the first CurrentBlanket set.

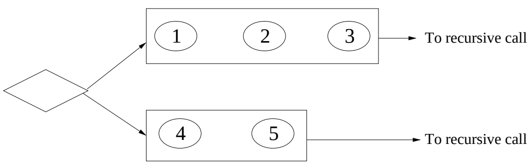

Figure 12: The first call of *HiddenSplitsStep* will separate set $\{1, 2, 3, 4, 5\}$ as $\{1, 2, 3\}$ and $\{4, 5\}$.

Consider the new call for *HiddenSplitsStep* with argument $\mathbf{S} = \{1, 2, 3\}$. The corresponding graphs M and M^C are now shown in Figure 13. M is not disconnected, but M^C is. This will lead to an insertion of an AND-split separating sets $\{1\}$ and $\{2, 3\}$ and another recursive call for $\{2, 3\}$.

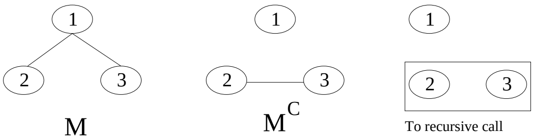

Figure 13: Graphs M and M^C corresponding to $\mathbf{S} = \{1, 2, 3\}$ in *HiddenSplitsStep*.

At the end of the first *HiddenSplits*, H will be given by the graph show in Figure 14. We now proceed to insert the remaining nodes into H.

From the ordering graph of Figure 10, we will choose as the next blanket the set $\{6, 7, 12\}$. Since they are not connected by any edge in Figure 11, we did not need to do any independence test to remove edges between them. When computing the direct dependencies between $\{1, \ldots, 5\}$ and $\{6, 7, 12\}$, since no conditional independence holds between elements in $\{6, 7, 12\}$ conditioned on positive measurements of any element in $\{1, 2, 3, 4, 5\}$, all elements in $\{1, 2, 3, 4, 5\}$ will be the direct dependencies of each element in $\{6, 7, 12\}$.

We now have to perform the insertion of possible latents between $\{1, 2, 3, 4, 5\}$ and $\{6, 7, 12\}$. There is only one set **Siblings** in *InsertLatents*, $\{6, 7, 12\}$, and one **AncestralSet**, $\{1, 2, 3, 4, 5\}$. When inserting hidden joins for elements in **AncestralSet**, we will perform an operation analogous to our previous example of *HiddenSplits*, but with arrows directed in the opposite way. The modification in shown in Figure 15(a), while Figure 15(b) depicts the modification of the relation between $\{6, 7, 12\}$. The last step of our *InsertLatents* iteration simply connects the childless node of Figure 15(a) to the parentless node of Figure 15(b).

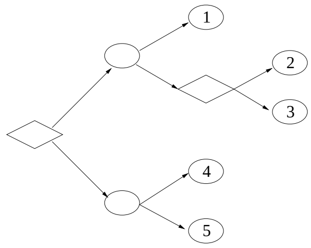

Figure 14: The partially constructed graph H.

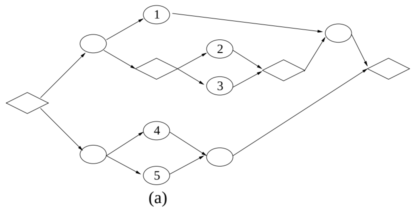

Figure 15: Inserting latents between two layers of observable tasks.

Again, we proceed to add more observable tasks in the next cycle of *LearnOrderedWorkflow*. The candidates are $\{8, 9, 10, 11\}$.

By Figure 10, all elements in $\{8, 9, 10, 11\}$ are adjacent. However, by conditioning on singletons from $\{6, 7, 12\}$ we can eliminate edges $\{8 \to 9, 9 \to 10, 8 \to 11, 10 \to 11\}$. The parentless nodes in this set are now 8 and 9, instead of 8 only. **CurrentBlanket** is now $\{6, 7, 12\}$ and **NextBlanket** is $\{8, 9\}$.

When determining direct dependencies, we first select $\{6, 7\}$ as the possible ancestors of $\{8, 9\}$. Since 8 and 7 are independent conditioned on 6, and 9 and 6 are independent conditioned on 7, only edges 6 8 and 7 9 are allowed. Analogously, the same will happen to $8 \to 10$ and $9 \to 11$. Graph H, after introducing all observable tasks, is shown in Figure 16. After introducing the last hidden joins in the final steps of *LearnOrderedWorkflow*, we reconstruct exactly the original graph in Figure 9.

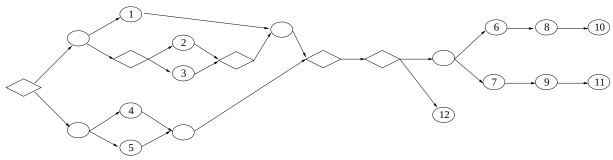

Figure 16: The graph H after introducing all observable tasks and just before introducing the last hidden joins.