Process Discovery Using Prior Knowledge

Aubrey J. Rembert, Amos Omokpo,
Pietro Mazzoleni, and Richard T. Goodwin

IBM T.J. Watson Research Center
Yorktown Heights NY 10598, USA

Abstract. In this paper, we describe a process discovery algorithm that leverages prior knowledge and process execution data to learn a control-flow model. Most process discovery algorithms are not able to exploit prior knowledge supplied by a domain expert. Our algorithm incorporates prior knowledge using ideas from Bayesian statistics. We demonstrate that our algorithm is able to recover a control-flow model in the presence of noisy process execution data, and uncertain prior knowledge.

1 Introduction

Process discovery is a research area at the intersection of business process management and data mining that has as one of its main objectives the development of algorithms that find novel relationships within, and useful summarizations of, process execution data. These relationships and summarizations can provide actionable insight such as the need for process redesign, organizational restructuring, and resource re-allocation. Control-flow discovery is a sub-area of process discovery concerned with the development of algorithms for learning the dependency structure between activities from process execution data.

In this paper, we consider the problem of learning control-flow models in the form of Information Control Nets (ICN) from the combination of noisy process execution logs, and uncertain prior knowledge encoded as augmented ICNs. Most control-flow discovery algorithms do not incorporate prior domain knowledge. Prior knowledge from domain experts or a repository of process models from the same domain can be a valueable resource in the discovery of control-flow models. This is especially true if there are important process segments that are executed infrequently. For example, in a banking process, if a transaction involving more than $100,000 is performed, a separate part of the banking process is executed. If the underlying control-flow discovery algorithm is designed to handle noise, then important, infrequent process executions may not get reflected in the discovered control-flow model. On the other hand, if the control-flow discovery algorithm is not designed to handle noise, then the discovered control-flow model will incorporate important, infrequent process executions, as well as erroneous, infrequent process executions.

The main contributions of this paper are that we present a control-flow discovery algorithm that uses prior knowledge in the form of an augmented Information Control Net, and process execution data to automatically discover a control-flow

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 328–342, 2013.

model in the form of an Information Control Net. Our control-flow discovery algorithm can deal with noise in the process execution data, and uncertainty in the prior knowledge using ideas from Bayesian statistics. Additionally, our control-flow discovery algorithm can deal with cycles and discovers the semantics of splits and joins.

2 Related Work

The area of process discovery is over fifteen years old. It was first investigated by Cook and Wolf [1] in the context of software processes. Next, process discovery was investigated by Agrawal et. al [2] in the context of business processes. The work of Cook and Wolf, and Agrawal et. al laid a foundation for process discovery. However, their work does not make use of models that can explicitly represent the nature of concurrent and decision splits, and synchronous and asynchronous joins, and does not leverage prior knowledge. The first phase of our algorithm builds on the algorithm developed by Agrawal et. al [2] by incoporating an approach to leverage prior knowledge.

In the paper [3], van der Aalst et al. describe the α-algorithm, a process discovery algorithm that explores the theoretical limits of the WF-Net (a Petri-net variant) approach to process discovery. It is based on a complete and noise-free log of process traces. The authors describe some control-flow patterns that are impossible for the α-algorithm to discover.

The paper by Fahland and van der Aalst [4] proposes an approach to include prior knowledge in process discovery. Their approach takes as input a potentially noisy process execution log and prior knowledge encoded as a Petri net. It produces a Petri net that contains the prior knowledge Petri net with additional sub-models that represent subtraces in the process execution data that did not fit the prior knowledge Petri net. Their approach assumes that the control-flow model provided by a domain expert is accurate and that only the addition of sub-control-flow models to the domain expert supplied control-flow model can be made. Our approach assumes that there is a level of uncertainty associated with provided domain knowledge. This leaves room for a domain expert's control-flow model to be erroneous. In our approach to process discovery using prior knowledge, the resulting control-flow model omits erroneous structures in a domain expert's control-flow model if it is not supported by enough evidence in the process execution data.

In the paper [5], Medeiros et. al. introduce a genetic algorithm for process discovery that takes as input process models represented as Causal Matrices as an initial population. This initial population can be a set of process models supplied by a domain expert. However, the genetic algorithm approach for process discovery uses a global score and search procedure, which makes it difficult to distinguish important, yet infrequent process fragments from noise.

3 Process Execution Logs

Process execution logs are the process execution data that our control-flow discovery algorithm uses to learn control-flow models. A *process instance* is an execution of a process. A *process trace* or simply *trace* is a list of events generated by a process instance. The events in a trace are denoted by the triple, (P, A, X), where P is unique trace identifier, A is the name of the activity, and X is the timestampe of the event. A *process execution log*, denoted by $\mathcal{L}$, is a multiset of process traces.

The dependencies that exist in a process will be implicitly embedded in the process execution log it generates. For instance, if activity b is dependent on activity a in a control-flow model, then, in each trace containing events from activities a and b, the event generated by a will always appear before that of b, unless there are measurement or ordering errors in process execution log generation. By an abuse of notation, we represent activities and events with the same symbol; this abuse of notation will be clear from the context.

Definition 1 (Precede). *Given a trace T containing events a and b, event a* precedes *event b in T, denoted by $a \prec_T b$, if a occurs before b in T. (The T from $\prec_T$ can be dropped when the context is clear)*

Definition 2 (Dependent). *Let $\mathcal{L} = \{T_1, \ldots, T_{|\mathcal{L}|}\}$ be a process execution log. Activity b is* dependent *on activity a, denoted by $a \rightarrow b$, if event a precedes event b a statistically significant number of times.*

Definition 3 (Independent). *Let $\mathcal{L}$ be a process execution log. Activity a is* independent *of activity b (and vice versa), if it is not the case that $a \rightarrow b$, or $b \rightarrow a$.*

Definition 4 (Mutually Exclusive). *Let $\mathcal{L}$ be a process execution log. Activity a is* mutually exclusive *of activity b, if a and b are negatively correlated in $\mathcal{L}$ (i.e. they hardly ever appear together in the same process trace).*

Given a process execution log, $\mathcal{L}$, we can define a dependency graph, and an independency graph that represent the dependencies between activities. These two graphs implicitly represent the structure and semantics of the underlying control-flow model. The dependency graph represents the structure, and the independency graph represents the semantics of the splits and joins.

Definition 5. *Given a process execution log, $\mathcal{L}$, and a set of unique activities in $\mathcal{L}$, denoted by $A_\mathcal{L}$, a directed graph $D_\mathcal{L} = (A_\mathcal{L}, F)$ is a* dependency graph *over $\mathcal{L}$, if for each pair of activities $a, b \in A_\mathcal{L}$, there exists a path $a \rightsquigarrow b$ in $D_\mathcal{L}$ where there exists a dependency relationship, $a \rightarrow b$, between activities a and b in $\mathcal{L}$.*

Definition 6. *Given the process execution log, $\mathcal{L}$, and a set of unique activities, $A_\mathcal{L}$, an undirected graph $U_\mathcal{L} = (A_\mathcal{L}, H)$ is a* independency graph *over $A_\mathcal{L}$, iff there exists an undirected edge between each pair of activities a and b in $U_\mathcal{L}$ where there exists an independency relationship between a and b in $\mathcal{L}$.*

4 Information Control Nets

The result of control-flow discovery using prior knowledge is an Information Control Net (ICN). An ICN is an edge-colored, directed graph $G = (A, E, \delta)$ used to model the control-flow of a business process, where A is a finite set of activities, $E \subseteq A \times A$ is a set of control-flow links, and $\delta = \delta_{in} \cup \delta_{out}$ is a set of mappings used to represent edge colors. Sets A and E define the structure of an ICN, while set δ defines the semantics of its splits and joins. Let $a, b \in A$ be activities. The *predecessors* of a are denoted by $pred(a) = \{b|(b, a) \in E)\}$. The *successors* of a are denoted by $succ(a) = \{b|(a, b) \in E)\}$.

Activities can be classified as *simple*, *split* or *join*. A simple activity has at most one predecessor, and at most one successor. A split activity has multiple successors, and a join activity has multiple predecessors. It is important to note that in the ICN model a single activity can be both a split activity and a join activity. There are two unique activities, s and t, called the starting and terminating activities, respectively, in every ICN. Starting activity s has no predecessors, and terminating activity t has no successors.

A control-flow link (a, b) is said to be *activated* if once activity a has finished executing, activity b is eligible for execution. In some instances, where activity a is a split activity, activity a must choose a subset of its control-flow links to activate. If the proper constraints are satisfied, the target activities of activated control-flow links can be executed. We describe those contraints in Section 4.1.

4.1 ICN Normal Form of δ

The ICN Normal form of δ is a canonical representation, invented by Ellis in the paper [6], that enables our edge coloring scheme. The mappings $\delta_{in}(a)$ and $\delta_{out}(a)$ partition the sets $pred(a)$ and $succ(a)$, respectively, in such a way that they describe which activities can execute concurrently and which activities cannot. Let $\mathcal{C}$ be a set of disjoint sets of activities such that $\delta_x(a) = \mathcal{C}$, and let each $C_i \in \mathcal{C}$ be a set of activities. Additionally, let the cardinality of each $C_i \in \mathcal{C}$ be $s(i)$, the cardinality of $\mathcal{C}$ be ℓ, and c_i^j be an activity in the set C_i. The *ICN normal form* of $\delta_x(a)$ is represented by Equation 1, where x can take on either the value *in* or *out*.

$$\delta_x(a) = \{\{c_1^1, \ldots, c_1^{s(1)}\}, \ldots, \{c_\ell^1, \ldots, c_\ell^{s(\ell)}\}\}, \tag{1}$$

In Equation 1, when $x = in$, activity a can execute if and only if each activity c_i^j in exactly one set $C_i = \{c_i^1, \ldots c_i^{s(i)}\} \in \mathcal{C}$ has finished executing and activated control-flow link (c_i^j, a). When $x = out$, activity a can choose only one set $C_i \in \mathcal{C}$. Based on this choice, each activity $c_i^k \in C_i$ is enabled to execute when control-flow link (a, c_i^k) becomes activated. Activities c_i^j and c_i^k in the same set C_i can be executed concurrently. Alternatively, given sets $C_i, C_j \in \delta_x(a)$ such that activities $c_i^k \in C_i$, $c_j^l \in C_j$, and $i \neq j$, it is the case that c_j^k and c_i^l can never be executed concurrently.

We now sketch our edge-coloring scheme. Given an activity a such that $\delta_{out}(a) = \mathcal{C} = \{C_1, \ldots, C_\ell\}$, let $E_{succ(a)} = \{(a, b) | b \in succ(a)\}$ be the set of control-flow links to the activities in $succ(a)$ from a. Let each $C_i \in \mathcal{C}$ define a color. Each edge in $E_{succ(a)}$ is colored according to the set, $C_i \in \mathcal{C}$, its target activity belongs to. However, if a is connected to a join activity j then the color of the (a, j) control-flow link is determined by $E_{pred(j)}$, which is defined analogously to $E_{succ(a)}$. Additonally, in ICNs, some activities are *observable*, while others are *hidden*. Observable activities are executed by (human/machine) actors and generate events that are recorded in process execution data, while hidden activities are not executed by actors and do not generate events that are recorded. Hidden activities are a convention used to represent control-flow patterns that cannot be directly represented in ICN normal form using only observable activities. For purposes of this paper, we substitute edge color for edge slashes (edges with the same number of slashes are the same color).

Example 1. Consider the ICN in Figure 1. Let the hollow circles represent hidden activities $h1$ and $h2$. The figure shows that $\delta_{out}(a) = \{\{b, h1\}\}$ and $\delta_{in}(a) = \{\{\}\}$. This means that a can be executed at any time, and once it has finished executing, the control-flow links (a, b) and $(a, h1)$ are activated. Thus, enabling b and $h1$ to execute concurrently. This figure also shows that $\delta_{in}(h2) = \{\{c\}, \{d\}\}$. This means $h2$ can execute when either the control-flow link $(c, h2)$ is activated as a result of c finishing execution, or when the control-flow link $(d, h2)$ is activated as a result of d finishing execution. After $h2$ executes, the control-flow link $(h2, f)$ is activated. Note that f cannot execute until the both the (e, f) and $(h2, f)$ edges are activated.

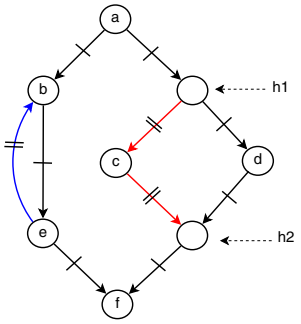

Fig. 1. An Information Control Net (ICN)

4.2 Augmented ICN

The prior knowledge is specified in an augmented ICN. That prior knowledge can be from a domain expert, or a repository of control-flow models from the same domain in which we wish to perform process discovery. An augmented ICN is an ICN with degrees of belief specified on its edges, and edge colors. The degree of belief specified on an edge reflects either how strongly a domain expert believes in the dependency between two activities, or, given a repository of control flow

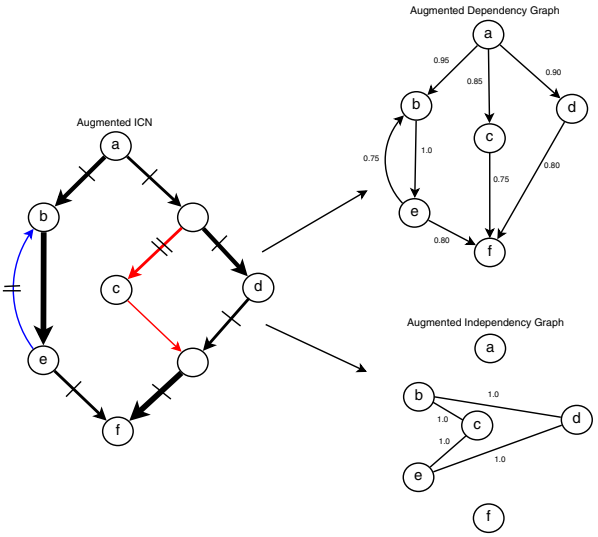

Fig. 2. An Augmented ICN with Corresponding Augmented Dependency Graph and Augmented Independency Graph

models from the same domain, the proportion of control flow models that contain that edge. The degree of belief specified on an edge color signifies the strength of belief in the concurrency of two activities. Degrees of belief can be represented by the thickness of an edge (i.e. the thicker the edge the higher the degree of belief), and the intensity of an edge color (i.e. the more intense the edge color the higher the degree of belief). The thickness of an edge, as well as the intensity of an edge color correspond to a probability in the interval (0, 1].

An augmented ICN can be broken down into its components, which are an augmented dependency graph $D_{\mathcal{K}}$, and an augmented independency graph $U_{\mathcal{K}}$. The augmented dependency graph $D_{\mathcal{K}}$ is a colorless digraph that contains all of the observable activities in the corresponding augmented ICN, and edge labels of the degree of belief. The augmented independency graph is an undirected graph that contains all of the observable activities in the augmented ICN as vertices. An undirected edge in the augmented independency graph represents a concurrency relationship between the incident activities.

In addition to the degree of belief, the experience level of the domain expert, and/or number of control-flow models in the repository must be taken into account. The quantity $n_{\mathcal{K}}$ represents the number of traces that a domain expert's belief is based on or the number of control-flow models in the repository. This experience level will also help in determining how eager our control-flow discovery algorithm is to change the structure and semantics of an augmented ICN. Figure 2 shows an augmented ICN along with its corresponding augmented dependency graph and augmented independency graph.

5 Dependency Extraction

We can now state the control-flow discovery with prior knowledge problem. Given a process execution log $\mathcal{L}$, an augmented dependency graph $D_{\mathcal{K}}$, and an augmented indenpendency graph $U_{\mathcal{K}}$, construct an Activity Precedence Graph G, that encodes the statistically significant activity dependencies in the combination of $\mathcal{L}$, $D_{\mathcal{K}}$, and $U_{\mathcal{K}}$. The algorithm we present to solve the control-flow discovery with prior knowledge problem consists of two phases. The first phase, called *Dependency Extraction*, learns activity dependencies and independencies from the combination of a process execution log and prior knowledge. The output of the Dependency Extraction phase is a dependency graph $D_{\mathcal{L}}$ and a independency graph $U_{\mathcal{L}}$. The second phase, called *Split/Join Semantics Discovery*, is concerned with transforming the dependency graph and independency graph into an ICN. The Split/Join Semantics Discovery phase is described in the 2009 paper by Rembert and Ellis [7], and will not be presented in this paper due to space concerns.

The *Dependency Extraction* algorithm computes the pair-wise precedence relationships between activities found in the process execution log. The input to the Dependency Extraction algorithm is a process execution log $\mathcal{L}$, a user-defined threshold μ, an augmented dependency graph $D_{\mathcal{K}}$, an augmented independency graph $U_{\mathcal{K}}$, and a domain expert's experience level $n_{\mathcal{K}}$.

The outputs of this algorithm are a dependency graph and an independency graph that contain the most probable dependency relationships reflected in both the process execution log and the prior knowledge. The Dependency Extraction algorithm is depicted in Algorithm 5.1

Before we proceed with the description of the algorithm, we first characterize the type of noise we expect to see in process traces. Our characterization is adapted from Silva et. al. [8]. By an abuse of notation, we let a be a binary random variable such that $a = 1$ means that activity a was executed, and $a = 0$ means that activity a was not executed. Let a_R be a binary random variable such that $a_R = 1$ means that the event corresponding to activity a was recorded and $a_R = 0$ means that the event was not recorded. We assume the following type of measurement error. The conditional probability $p(a_R = 1|a = 1) = \omega > 0$, captures the uncertainty associated with an activity executing in a process instance, and its corresponding event being recorded in the appropriate process trace. We assume $p(a_R = 1|a = 0) = 0$, which expresses that an event cannot be included in a process trace if a corresponding activity was not executed. In addition to measurement error (activities being executed but not recorded in a process trace), we consider ordering errors. An ordering error happens in a process instance when an activity a finishes executing before an activity b has, but in the corresponding process trace event b is recorded as finishing before event a has finished. Between each pair of activities, we consider and ordering error rate of ϵ.

The *DependencyExtraction* algorithm is based on ideas presented by Agrawal et. al [2]. The first idea that we leverage from Agrawal is the notion of *cycle unrolling* in the process execution log. Cycle unrolling entails treating events with the same label in a process trace as different events. This is done by relabeling events with an occurence counter. For instance, the first occurence of activity a is relabeled a_1, the second occurence as a_2, and so on. The process execution log $\mathcal{L}$ with unrolled cycles is denoted by $\mathcal{L}^*$.

Mutually exclusive activities can be difficult to detect if they occur within a cycle. This is because cycles can enable mutually exclusive activities to appear in the same process trace. However, we can leverage correlation to determine the strength of association between activities across process traces. In a cycle unrolled process execution log, we can compute the ϕ-coefficient between activities and store the result in the square matrix $\mathbf{M}^\phi = [\phi_{ij}]$. The rows and columns of $\mathbf{M}^\phi$ correspond to unique activities in $\mathcal{L}^*$, and the value ϕ_{ij} corresponds to the ϕ-coefficient of activities indexed by i and j. The ϕ-coefficient of activities a and b is given by Equation 2

$$\phi_{ab} = \frac{(N_{ab}N_{\overline{ab}}) - (N_{a\overline{b}}N_{\overline{a}b})}{\sqrt{N_a N_{\overline{a}} N_b N_{\overline{b}}}}, \tag{2}$$

where:

- N_{ab} is the number of process traces that both activities a and b occurr in,
- $N_{\overline{ab}}$ is the number of traces that don't contain either a or b,
- $N_{a\overline{b}}$ is the number of traces that contain a, but not b,
- $N_{\overline{a}b}$ is the number of traces that contain b, but not a,
- N_a is the number of traces that contain a,
- $N_{\overline{a}}$ is the number of traces that don't contain a,
- N_b is the number of traces that contain b, and
- $N_{\overline{b}}$ is the number of traces that don't contain b.

If activities are found to be negatively correlated, we assume that they are mutually exclusive. It easy to see that if $N_{ab} = 0$ (i.e. activities a and b don't occur in any of the same process traces), then Equation 2 will have a negative value.

Binomial Distribution and Beta Prior. For all the positively correlated pairs of activities, we compute the likelihood that one activity is dependent on another with the results stored in a cycle unrolled dependency graph and a cycle unrolled independency graph. Let $D_{\mathcal{L}^*}$ be a cycle unrolled dependency graph and $U_{\mathcal{L}^*}$ be a cycle unrolled independency graph, both of which contain all of the relabeled activities as vertices. The cycle unrolled dependency graph is a directed graph, and the cycle unrolled independency graph is an undirected graph, both of which are initially edgeless.

To add directed edges to $D_{\mathcal{L}^*}$ and undirected edges to $U_{\mathcal{L}^*}$, we leverage the Binomial distribution and the Beta prior. The binomial distribution is used determine the probability of k successes in N bernoulli trials given a parameter

μ. Let a and b be activities that we wish to discover the dependency relationship between, N be the total number of traces in the process execution log, N_{ab} be the number of traces that both a and b occur in, and $k_{a \prec b}$ ($k_{b \prec a}$) be the number of traces that a precedes b (b precedes a). Additionally, let μ be a user-defined parameter that represents the proportion of times that a must precede b in order for b to be considered dependent on a; μ can be considered an *activity precedence to occurrence ratio*. It is important to note that one should set μ so that a certain amount of ordering error can be effectivly handled.

The binomial distribution can be written

$$p(D_{ab}|\mu) = \binom{N_{ab}}{k_{a \prec b}} \mu^{k_{a \prec b}} (1-\mu)^{N_{ab} - k_{a \prec b}} \tag{3}$$

, where

$$\binom{N_{ab}}{k_{a \prec b}} = \frac{N_{ab}!}{(N_{ab} - k_{a \prec b})! k_{a \prec b}!}. \tag{4}$$

The conditional probability $p(D_{ab}|\mu)$ is the likelihood of D_{ab} given μ, where $D_{ab} = \{T_1^{a \prec b} = 1, T_2^{a \prec b} = 0, \ldots, T_{N_{ab}}^{a \prec b} = 1\}$ represents the set of traces that activities a and b occur together in. The trace-level precedence indicator $T_i^{a \prec b}$ takes a value of 1, if a was executed before b, and 0 otherwise. It should be noted that $k_{a \prec b} = \sum_{i=1}^{N_{ab}} T_i^{a \prec b}$.

We use the binomial distribution in activity dependency discovery by conducting a binomial test for each pair unique activities in the cycle unrolled process execution log $\mathcal{L}^*$. The null hypothesis of this test is: if $p(D_{ab}|\mu)$ is greater than a user-defined significance level, then $\frac{k_{a \prec b}}{N_{ab}}$ is not significantly different from μ, thus $a \rightarrow b$. The first alternative hypothesis is: if $p(D_{ab}|\mu)$ is less than a user-defined significance level in the top tail of the binomial distribution, then $\frac{k_{a \prec b}}{N_{ab}}$ is significantly larger than μ, therefore it is also the case that $a \rightarrow b$. The second alternative hypothesis is: if $P(D_{ab}|\mu)$ is less than a user-defined significance level in the bottom tail of the binomial distribution, then $\frac{k_{a \prec b}}{N_{ab}}$ is significantly smaller than μ, therefore b is not dependent on a. The user-defined significance-level is typically 0.05 or 0.025 for two-tailed binomial tests. If the null hypothesis is accepted, or the null hypothesis is rejected and the first alternative hypothesis is accepted, then a directed (a, b) edge is added to the cycle unrolled dependency graph $D_{\mathcal{L}^*}$. If it is found that b is not dependent on a and vice-versa, then an undirected edge (a, b) is added to $U_{\mathcal{L}^*}$.

The approach just described is called the *Likelihood Estimate* approach. The Likelihood Estimate approach is based solely on data and does not take into account the experience and expertise of domain experts when discovering the dependency between activities. In situations where domain expertise is not available, or contains gaps, the Likelihood Estimate approach is used. However, since the Likelihood Estimate approach does not take into account the prior knowledge of a domain expert, we need an approach that does. This can be done using Bayesian statistics.

In Bayesian statistics, we can leverage prior knowledge to help with determining the dependencies between activities. In the Likelihood Estimate approach, the uncertainty is associated with the data. However, in the Bayesian approach, the uncertainty is associated with the activity precedence to occurrence ratio μ. So, instead of calculating $p(D_{ab}|\mu)$, we calculate $p(\mu|D_{ab})$, which according to Bayes' Theorem is:

$$p(\mu|D_{ab}) = \frac{p(D_{ab}|\mu)p(\mu)}{p(D_{ab})}. \tag{5}$$

From Equation 5, we can see that the Bayesian approach leverages the Likelihood Estimate approach. Additionally, since the data is given, $p(D_{ab}) = 1$. For the assesment of $p(\mu)$, we choose the Beta distribution as a prior because it is conjugate to the Binomial distribution. The Beta distribution is defined as

$$p(\mu) = Beta(\mu|v, w), \tag{6}$$

such that

$$Beta(\mu|v, w) = \frac{\Gamma(v + w)}{\Gamma(v)\Gamma(w)}\mu^{v-1}(1 - \mu)^{w-1}, \tag{7}$$

where the Gamma function is defined as $\Gamma(x + 1) = x!$. The quantities v and w are called the hyperparameters of the Beta distribution, and are used to control its shape.

To incorporate prior knowledge, we take both the augmented dependency graph and the augmented independency graph and use the edge labels as priors when trying to determine the dependency relationship between activities. Essentially, the prior knowledge of experts represent virtual occurences of activity pairs occurring in a particular order. Since $k_{b\prec a} = N_{ab} - k_{a\prec b}$, then

$$p(\mu|D_{ab}) = \frac{\Gamma(N_{ab} + v + w)}{\Gamma(k_{a\prec b} + v)\Gamma(k_{b\prec a} + w)}\mu^{k_{a\prec b}+v-1}(1 - \mu)^{k_{b\prec a}+w-1}, \tag{8}$$

where the hyperparameters v and w are based on the prior degree of belief on the domain expert.

We now show how to assess the hyperparameters v and w for a pair of activities, given an augmeneted dependency graph $D_{\mathcal{K}}$ and an augmented independency graph $U_{\mathcal{K}}$. For activities a and b, let there be a directed (a, b) edge in the augmented dependency graph $D_{\mathcal{K}}$. By another abuse of notation, let $a \rightarrow b$ be a binary random variable such that $p(a \rightarrow b = 1|D_{\mathcal{K}}, U_{\mathcal{K}})$ is the degree of belief specified on the dependency relationship between activities a and b. Additionally, let edge (a, b) be based on $n_{\mathcal{K}}$ process traces, which, as described above, captures the level of experience of the domain expert. Given these assumptions, we set v and w to be: $v = p(a \rightarrow b = 1|D_{\mathcal{K}}, U_{\mathcal{K}})n_{\mathcal{K}}$, and $w = p(a \rightarrow b = 0|D_{\mathcal{K}}, U_{\mathcal{K}})n_{\mathcal{K}}$. For activities b and c, let there be an undirected (b, c) edge in an augmented independency graph $U_{\mathcal{K}}$. In this case, we assess v and w to be: $v = p(a \rightarrow b = 1|D_{\mathcal{K}}, U_{\mathcal{K}}) \cdot 0.5 \cdot n_{\mathcal{K}}$, and $w = v$.

The binomial test is again used to determine activity dependence. The null hypothesis of this test is: if $p(\mu|D_{ab})$ is greater than a user-defined significance

level, then there is no significant difference between μ and $\frac{k_{a \prec b}+v}{N_{ab}+v+w}$, and therefore $a \rightarrow b$. The first alternative hypothesis is: if $p(\mu|D_{ab})$ is less than a user defined significance level and in the top tail of the distribution, then μ is significantly less than $\frac{k_{a \prec b}+v}{N_{ab}+v+w}$, and $a \rightarrow b$. The second alternative hypotheis is: if $p(\mu|D_{ab})$ is less than a user defined significance level and in the bottom tail of the distribution, then μ is significantly greater than $\frac{k_{a \prec b}+v}{N_{ab}+v+w}$, and b is not dependent on a. If the probability returned from the binomial distribution is above a user-defined significance-level, then we accept the null hypothesis, otherwise we reject it and accept the alternative hypothesis. Like the Likelihood Estimate approach, If the null hypothesis is accepted, or the first alternative hypothesis is accepted, we add an directed edge (a, b) in the cycle unrolled dependency graph $D_{\mathcal{L}^*}$, and if b is not dependent on a, and vice-versa, we add an undirected edge (a, b) in $U_{\mathcal{L}^*}$.

When constructing $D_{\mathcal{L}^*}$ and $U_{\mathcal{L}^*}$ using prior knowledge $D_{\mathcal{K}}$ and $U_{\mathcal{K}}$, it is the case that the relabeled activities in the cycle unrolled process execution log $\mathcal{L}^*$ will not match the activities in $D_{\mathcal{K}}$ and $U_{\mathcal{K}}$. To handle this issue, when matching activities from the augmented dependency graph and the augmented independency graph, we ignore the count appended to the activities in $\mathcal{L}^*$. For example, the edge (a, b) in an augmented dependency graph will match the pair of activities a_1 and b_1, as well as the pair a_2 and b_7. If b_7 is found to be dependent on a_2 based on edge (a, b) in the augmented dependency graph and the binomial test, then edge (a_2, b_7) is added to $D_{\mathcal{L}^*}$.

Re-rolling the Cycle Unrolled Dependency and Independency Graphs. Our algorithm re-rolls the cycle unrolled dependency and independency graphs. The first step in this process is to minimize the number of edges in $D_{\mathcal{L}^*}$ without loosing the appropriate dependency information. This is done by using a heuristic proposed by Agrawal et. al. [2], which computes the transitive reduction of each induced subgraph $D^i_{\mathcal{L}^*}$ formed over the activity relabeled process trace T_i. For each $D^i_{\mathcal{L}^*}$, mark all the edges in the transitive reduction. Remove all edges from $D_{\mathcal{L}^*}$ that remains unmarked.

The next step in our process is to collapses $D_{\mathcal{L}^*}$ and $U_{\mathcal{L}^*}$ such that all of the activities that were relabelled to unroll cycles are merged into their original activity in both graphs. For instance, activity a_2 is collapsed into activity a_1, and all of the incoming and outgoing edges of a_2 become incoming and outgoing edges of a_1. This process continues until there are no more activities with an activity counter label greater than 1. The activity counter label, is then dropped from all activities. The collapsed versions of $D_{\mathcal{L}^*}$ and $U_{\mathcal{L}^*}$ are $D_{\mathcal{L}}$ and $U_{\mathcal{L}}$, respectively.

The final step in cycle re-rolling algorithm is the capture of cycles in $D_{\mathcal{L}}$. We capture cycles in $D_{\mathcal{L}}$ because in the split/join semantics discovery phase of the *LearnICN* algorithm activities that are the target of a backegdge are treated slightly different than other activities. Cycles are captured by discovering and marking all backegdes in $D_{\mathcal{L}}$. Backedges are discovered using a simple depth-first search exploration of $D_{\mathcal{L}}$ initiated at the unique starting activity s.

Algorithm 5.1. *DependencyExtraction*($\mathcal{L}$, $D_{\mathcal{K}}$, $U_{\mathcal{K}}$, $n_{\mathcal{K}}$, μ)

$\mathcal{L}^* \leftarrow UnrollCyclesInLog(\mathcal{L})$
$A^* \leftarrow$ unique activities in $\mathcal{L}^*$
$\mathbf{M}^{\phi} \leftarrow ComputeCorrelation(\mathcal{L}^*)$
$D_{\mathcal{L}^*} \leftarrow (A^*, \emptyset)$
$U_{\mathcal{L}^*} \leftarrow (A^*, \emptyset)$
for each pair of positively correlated activities $a, b \in \mathbf{M}^{\phi}$ **do**
  if the pair (a, b) is unmarked **then**
    Let N_{ab} be the number of times that a and b occur in the same process trace
    $v \leftarrow 0$
    $w \leftarrow 0$
    if edge $(a, b) \in D_{\mathcal{K}}$ **then**
      $v \leftarrow p(a \rightarrow b = 1 | D_{\mathcal{K}}, U_{\mathcal{K}}) \cdot n_{\mathcal{K}}$
      $w \leftarrow p(a \rightarrow b = 0 | D_{\mathcal{K}}, U_{\mathcal{K}}) \cdot n_{\mathcal{K}}$
    else if edge $(a, b) \in U_{\mathcal{K}}$ **then**
      $v \leftarrow p(a \rightarrow b = 1 | D_{\mathcal{K}}, U_{\mathcal{K}}) \cdot 0.5 \cdot n_{\mathcal{K}}$
      $w \leftarrow v$
    end if
    if $BinomialTest(k_{a \prec b} + v, N_{ab} + v + w, \mu)$ accepts null hypothesis or first alternate hypothesis **then**
      add directed edge (a, b) to $D_{\mathcal{L}^*}$
    else if $BinomialTest(k_{b \prec a} + w, N_{ab} + v + w, \mu)$ accepts null hypothesis or first alternate hypothesis **then**
      add directed edge (b, a) to $D_{\mathcal{L}^*}$
    else
      add undirected edge (a, b) to $U_{\mathcal{L}^*}$, if its not already there
    end if
    mark the pair (a, b)
  end if
end for
for each process trace $T_i \in \mathcal{L}^*$ **do**
  Let $D^i_{\mathcal{L}^*}$ be the trace dependence graph for T_i
  Compute the transitive reduction of $D^i_{\mathcal{L}^*}$
  Mark the edges in $D_{\mathcal{L}^*}$ that are in the transitive reduction of $D^i_{\mathcal{L}^*}$
end for
Remove all unmarked edges from $D^*_{\mathcal{L}}$
$D_{\mathcal{L}} \leftarrow Collapse(D_{\mathcal{L}^*})$
$U_{\mathcal{L}} \leftarrow Collapse(U_{\mathcal{L}^*})$
Mark all backedges in $D_{\mathcal{L}}$
Return $D_{\mathcal{L}}$, $U_{\mathcal{L}}$

6 Experiments

We tested the hypothesis that, in the presence of noise, our process discovery algorithm that leverage prior knowledge learns more accurate APGs than ICN learning algorithms that do not. We tested our algorithm using the telephone repair example process execution log from the ProM example [9]. In our experiments, we let the activity occurrence to precedence ratio be $\mu = 0.90$, and the significance level be 0.05 for a two-tailed test. We experimented with two types of domain knowledge (perfect and imperfect), as well as no domain knowledge. We consider perfect domain knowledge to have the same structure as the reference activity precedence graph and certainty of edges and edge colors between (0.9 and 1.0) We created imperfect domain knowledge by removing and adding edges from the augmented dependency graph and the augmented independency graph. Additionally, the degree of belief for edges in both graphs is drawn uniformly from the range $(0.5, 0.9)$.

We tested the three versions of the algorithm at seven different log sizes (200, 400, 600, 800, 1000, 1200, 1400), three different experience levels (100, 200, 400), and three different measurement error levels (0.95, 0.90, 0.85) and a constant ordering error level of (0.05). To determine how well the control-flow discovery algorithm works, we compared the learned dependency graph and independency graph to the reference dependency graph and reference independency graph. The reference dependency graph and reference independency graphs were computed from the reference ICN. The reference ICN is the true control-flow model.

To compare the learned dependency graph with the reference dependency graph, we computed the edge recall, edge precision, color recall, and color precision. Edge recall is the number of edges that the learned dependency graph and reference dependency graph share divided by the number of edges in the reference dependency graph. Edge precision is the number of edges that the learned dependency graph and reference dependency graph share diveded by the number of edges in the learned dependency graph. Edge F-measure is a combination of edge precision and edge recall. Table 1 shows the edge F-measure for the dependency graphs learned from noisy process execution logs of the telephone repair process with a domain expert experience level of 200 (i.e. $n_{\mathcal{K}} = 200$). As can be seen from the results in Table 1, the control-flow discovery approach that leverages imperfect prior knowledge performs better, in terms of Edge F-measure, than the control-flow discovery approach that does not use prior knowldge. However, as the log size increases, this disparity is reduced because evidence provided by the data will eventually be the main determiner of edge recall and edge precision. Color recall is the size of the intersection between the edges in the learned independency graph and reference independency graph divided by the number of edges in the reference independency graph. Color precision is the number of edges that the learned independency graph and reference independency graph share diveded by the number of edges in the learned independency graph. Table 2 shows the color F-measure for the independency graphs learned from noisy process execution logs of the telephone repair process with a domain expert experience level of 200 (i.e. $n_{\mathcal{K}} = 200$). In Table 2, the color F-measure for Imperfect domain knowledge is slightly smaller than the color F-measure for no prior knowledge. This is primarily due to color recall for Imperfect domain knowledge.

Table 1. Edge F-measure of Learned Dependency Graph with $\epsilon = 0.05$ and $n_{\mathcal{K}} = 200$

	Measurement Error								
	0.85			0.9			0.95		
Prior / Traces	None	Imperfect	Perfect	None	Imperfect	Perfect	None	Imperfect	Perfect
200	0.744	0.841	0.844	0.778	0.871	0.874	0.870	0.935	0.937
400	0.766	0.846	0.848	0.844	0.904	0.905	0.938	0.980	0.984
600	0.798	0.863	0.867	0.878	0.925	0.927	0.950	0.991	0.993
800	0.831	0.879	0.883	0.900	0.943	0.948	0.950	0.989	0.997
1000	0.827	0.860	0.864	0.902	0.947	0.949	0.952	0.998	1.000
1200	0.821	0.893	0.895	0.915	0.968	0.970	0.952	0.996	1.000
1400	0.854	0.940	0.945	0.925	0.989	0.993	0.952	0.996	1.000

Table 2. Color F-measure of Learned Dependency Graph with $\epsilon = 0.05$ and $n_{\mathcal{K}} = 200$

	Measurement Error								
	0.85			0.9			0.95		
Prior / Traces	None	Imperfect	Perfect	None	Imperfect	Perfect	None	Imperfect	Perfect
200	0.809	0.974	0.984	0.777	0.961	0.979	0.789	0.947	0.964
400	0.899	0.971	0.984	0.932	0.957	0.973	0.966	0.958	0.966
600	0.917	0.972	0.980	0.947	0.944	0.953	0.984	0.957	0.984
800	0.955	0.958	0.974	0.960	0.942	0.961	0.996	0.982	0.996
1000	0.970	0.963	0.978	0.956	0.949	0.956	0.999	0.987	0.999
1200	0.887	0.947	0.968	0.953	0.945	0.963	0.990	0.999	1.000
1400	0.875	0.949	0.961	0.946	0.970	0.986	0.998	0.977	1.000

The slightly reduced color recall numbers are due to ordering errors in the log being boosted by reversed edges in the imperfect augmented dependency graph. However, despite the errors in the imperfect domain knowledge, a more correct model was found as more process execution data was provided. Additionally, as can be seen in both the edge F-measure, and the color F-measure, when there are small data sizes, having some prior knowledge increases the edge F-measure and color F-measure. This means that portions of the true process that are executed infrequently can be boosted by the presence of domain knowledge, therefore those infrequent portions of a process trace won't be considered noise. The results of our experiments confirm our hypothesis.

7 Summary and Future Work

In this work, we have presented a process discovery algorithm that leverages prior knowledge in the form of augmented Information Control Nets. We have shown that our process discovery algorithm is robust to noise in the process execution data in the form of measurement errors and ordering errors. Additionally, our process discovery algorithm is able to deal with uncertainty and errors in the prior knowledge it is provided. Through experimentation, we have shown that our approach is useful when important, infrequent portions of a process need to be discovered. Given enough certainty and experience, our approach will not consider the infrequency of those executions as noise. ICNs were developed nearly 30 years ago by Ellis [6]. Since the ICN normal form is nearly identical to the Causal Matrix [5] formalism, for the future, we'd like to explore process discovery with prior knowledge using Causal Matrices, which can be transformed into Petri nets.

References

1. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Software Engineering Methodology 7(3), 215–249 (1998)
2. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 469–483. Springer, Heidelberg (1998)

3. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
4. Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 229–245. Springer, Heidelberg (2012)
5. Medeiros, A., Weijters, A., Aalst, W.: Genetic process mining: an experimental evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
6. Ellis, C.A.: Formal and informal models of office activity. In: IFIP Congress, pp. 11–22 (1983)
7. Rembert, A.J., Ellis, C(S.): Learning the control-flow of a business process using ICN-based process models. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 346–351. Springer, Heidelberg (2009)
8. Silva, R., Zhang, J., Shanahan, J.G.: Probabilistic workflow mining. In: KDD 2005: Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp. 275–284. ACM Press, New York (2005)
9. Verbeek, H.E., Bose, J.C.: Prom 6 tutorial, reviewexample. (2010)