[[1]](#footnote-1)

An Algorithm for Mining Indirect Dependencies Between Loops and Loops Structure via Petri Nets

HongWei Sun, Wei Liu, Liang Qi, YuYue Du, and XiaoJun Ren

***Abstract*—As an emerging technology, process mining is used to extract, detect, and improve actual processes by extracting knowledge from event logs generated from information systems. In the production process, we can obtain the optimal process based on practical experience. Indirect dependencies may exist between different structures in the optimal process model. However, the existing process mining algorithms cannot effectively mine the indirect dependencies between different structures. To make up for this deficiency, an algorithm named *AlphaID* is proposed in this paper, which can mine the indirect dependencies of loop-choice driven loop structure. First, two efficient algorithms are proposed to identify loop sequences and choice sequences from event logs, respectively. Then, the concept of association rules is proposed to describe indirect dependencies between different structures. Next, we expand the ordinary Petri netand redefine the new transition firing rules to represent the process model obtained by the algorithm. At last, the correctness and effectiveness of the algorithm are verified by an artificial example and a real example, and the *AlphaID* algorithm is fully realized by integrating the algorithm into the ProM platform as a plug-in.**

***Index Terms*—indirect dependency, loop structure, ProM, Petri net, process mining,**

# INTRODUCTION

A

T present, as an emerging technology, the process mining algorithm is used by more and more modern enterprises. Process mining is not a simple fusion of existing methods but a close relationship between data and process models. Process mining is used to detect compliance, detect deviation, predict delay, support decision making and assist process reengineering. In other words, process mining is used to extract, detect, and improve actual processes by extracting knowledge from event logs generated from information systems [1]. Among them, process discovery is still one of the most challenging tasks in process mining, which can help us find the optimal process and improve our work efficiency. For a process model, we have four main quality dimensions: simplicity, fitness, precision, and generalization [1]. The simplicity represents a process model can express all the behavior in a log, the simpler the better. The fitness represents the ability of the trace (activities sequence) to be repeated in a process model. The precision is the ability of the process model to replay logs. A low precision means that a process model allow “too much” behaviors. A process model should generalize and not restrict behaviors to logs [1]. If a process model does not generalize, it will be overfitting. If a process model allows too many behaviors beyond behaviors in the log, it will be underfitting. Therefore, these four quality dimensions should be used in a balance.

In process mining, we mine process models from four perspectives: (1) the case perspective, (2) the organizational perspective, (3) the process perspective, and (4) the time perspective. The process perspective focuses on the control flow (sequence of activities). The organizational perspective focuses on inter-organizational relationships and resource allocation. The case perspective focuses on the attributes of the case. The time perspective focuses on the time and number of events in the event log. To analyze problems from the time perspective can help us optimize the process and improve work efficiency.

In recent years, some process mining algorithms have been proposed. A classical process mining algorithm *α*-algorithm is proposed in [2], which mines a process model based on the order dependence between activities. The *α*-algorithm can mine most of the process models [3] correctly. However, the *α*-algorithm is not efficient in mining short loops (with less than two activities) or non-free-choice structures, and in dealing with noise problems. Considering the shortcomings of the *α*-algorithm, many extended algorithms are proposed. The *α*+ algorithm is proposed in [4]. The *α*+ algorithm can obtain the process model containing a short loop structure when the loop sequence is complete. A process mining algorithm, *α*++ algorithm, which can mine non-freely selected structures, is proposed in [5]. Indirect dependencies between activities can be found by *α*++ algorithm. In [6], in order to mine invisible transitions, an algorithm of *α*# mining is proposed. Next, [7] proposes an algorithm *α*$ that can mine invisible transitions in process models with non-free-choice structures. [8] proposes a process mining algorithm *α*\*, which can effectively mine the process model containing repeated activities in an event log. With the further development of process mining technology, other advanced mining algorithms have been proposed. In [9], the Heuristic Miner (*HM*) algorithm is proposed to mine causal net and deal with noise. A gene mining algorithm is proposed in [10] to mine complex and special structures. An integer linear programming (*ILP*) is proposed in [11], which could mine process models with short loop structures. [12] proposes the Inductive Miner (*IM*) algorithm, it can mine block structure. A process model obtained by *IM* algorithm has high fitness. In [13] and [14], two region-based process mining algorithms are proposed, which can express complex process models and balance overfitting and underfitting as much as possible. A fuzzy process mining algorithm is proposed in [15], which is very effective for processing noisy or incomplete event logs.

From [5], we know that dependencies between activities include not only direct dependencies, but also indirect dependencies. At present, most process mining algorithms are based on the direct dependence between activities. However, effectively mining indirect dependencies between activities is also a challenge in process mining. It is also imperative to mine the indirect dependencies between different structures in the process model. The indirect dependence between different structures can effectively improve the process model and improve production efficiency.

The existing process mining algorithms do not consider the indirect dependence between different structures from the perspective of control flow.To make up for the vacancy, this paper proposes a method to mine the process model containing a loop-choice driven loop structure.The main contributions of this paper are as follows:

1. Two algorithms are proposed to identify loop sequences and choice sequences from event logs respectively.
2. In order to clearly express the indirect dependency between different structures, we introduce association rules and propose an algorithm to obtain effective association rules.
3. To clearly represent the process model, we extend the ordinary Petri net and redefine the new transition firing rules.
4. An algorithm named *AlphaID* is proposed to mine the process model containing loop-choice driven loop structure.
5. The correctness and effectiveness of the algorithm have been fully verified by experiments, and the *AlphaID* algorithm is integrated into the ProM platform as a plug-in.

The rest of this paper is organized as follows.In Section II, some of the basic knowledge related to our research are reviewed. An *AlphaID* algorithm is proposed for mining the indirect dependencies between the loop-choice driven loop structuresin Section III. In Section IV, the efficiency of the *AlphaID* algorithm are verified by two simulation experiments. In Section V, we conclude the work of this paper and discuss the research direction of future work.

# Preliminaries

In this section, we first review the following concepts of tuples [16], sequence [16-17], [38] multi-set [17], [39], trace activity, Petri net[18-21], [28-29], [31-37], trace [25-26], event log [17], [27], [30], and workflow net[22-24], [38]. Then the concept and related definition of *α****-***algorithm are introduced.

## Basic Knowledges

*Definition 1* (*Tuples*): Let *Ψ* be a set*. k* *=* (*a*1, *a*2, *...*, *an* )∈*Ψ*×...×*Ψ* is a tuple consisting of *n* elements.

For instance,let *k* *=* (*a*1, *a*2, *a*3, *a*4) be a quad set.

*Definition 2* (*Multi-Set*): Let *Ψ* be a set. A multi-set *R* over *Ψ* is a function *R*: *Ψ* → N, where N is a set of natural numbers. The set of all multi-sets over *Ψ* is denoted by ***B***(*ψ*).

For instance, let *Ψ* = {*m*, *n*, *p*, *q*, *r*}be a set, then *R* = {*m*3, *n*4, *p*5, *q*2} represents a multi-set over *Ψ* with *R*(*m*) = 3, *R*(*n*) = 4, *R*(*p*) = 5, and *R*(*q*) = 2.

*Definition 3* (*Sequence*) [38]:Let *Ψ* be a set. *σ* = <*σ*[1], ..., *σ*[*n*]> is a sequence over *Ψ*,where *σ*[*i*] ∈ *Ψ* denotes the *i*-th element of *σ*. |*σ*| = *n* denotes the length of *σ*. *σ*(*a*) denotes the number of occurrences of *a* in *σ*. *Ψ*∗ is the set of all finite sequences over *Ψ*. For all *σ* ∈*Ψ*∗ and 1 ≤ *i* ≤ *j* ≤ |*σ*|, σ*ij* denotes the subsequence of *σ* from *σ*[*i*] to *σ*[*j*].

For instance, let *σ* = <*a*, *b*, *c*, *d*, *a*, *e*, *f*>be a sequence, so *σ*[2] = *b*, |*σ*| =7, *σ*(*a*) = 2 and σ25 = < *b*, *c*, *d*, *a* >.

*Definition 4* (*Event Logs and Traces*):Let *Ψ* be the set of all activities, *A* ⊆ *Ψ* is a set, and *A*∗ is a finite sequence set over the set *Ψ.* Then *σ* ∈ *A*∗ is defined as a trace and a multi-set of traces is called an event log *L* ∈ ***B***(*A*∗).

For instance, let *Ψ* = {*m*, *n*, *p*, *q*} be a set of activities. σ1 = <*n*, *p*, *q*> and *σ*2 *=* <*m*, *n*, *p*, *q*> are two traces over *A*, an event log is *L*1= <*σ*1, *σ*2>.

*Definition 5* (*Traces* *Activities*): Let *Ψ* be the set of all activities, *L* be an event log, and *σ* ∈ *L* be a trace, then *φset*(*σ*) represents a set of activities in *σ*, and *φset*(*L*) represents a set of activities in *L*.

For instance, let *L* = {*σ*1 = <*a*, *b*, *r*, *d*, *n*, *r*, *d*, *n*, *g*, *m*>, *σ*2= <*a*, *b*, *r*, *d*, *n*, *q*, *o*, *s*, *t*, *m*>, *σ*3= <*a*, *b*, *t*, *m*>}is an event log with three traces. Thus *φset*(*σ*1) = {*a*, *b*, *r*, *d*, *n*, *g*, *m*}. *φset*(*L*) = {*a*, *b*, *r*, *d*, *n*, *g*, *q*, *o*, *s*, *t*, *m*}.

*Definition 6* (*Petri Net*): A four-tuple *PN* =(*P*, *T*; *F*, *M*0)is called Petri net, where

1. *N* = (*P*, *T*; *F*) is a net, where *P* is a finite set of places and *T* is a finite set of transitions, *P* ∪ *T* ≠ ∅and *P* ∩ *T* = ∅;
2. *F* ⊆(*P* *×* *T*) ∪ (*T* *×* *P*) is a set of directed arcs from places to transitions or from transactions to places;
3. *M*: *P* → *N* is called a marking, where *M*0 represents the initial marking, and for ∀*p* ∈ *P*, the number of tokens in *p* is represented by *M*(*p*);
4. *cod*(*F*) ∪ *dom*(*F*) = *T* ∪ *P*, where *cod*(*F*)*=*{*x* ∈ *P* ∪ *T |*∃*y* ∈ *P* ∪ *T*: (*y*, *x*) ∈ *F*}, and *dom*(*F*) *=* {*x* ∈ *P* ∪ *T* *|*∃*y* ∈ *P* ∪ *T*: (*x*, *y*) ∈ *F*}; and
5. Transition firing rules:
6. For ∀*t* ∈ *T*, if ∀*p* ∈ •*t*, *M*(*p*) ≥ 1, then the transition *t* is denoted as *M*[*t>*, which means that it is enabled in the mark *M*;
7. If *M*[*t>*, the transition *t* can be fired, and a new marking *M′* is obtained, denoted as *M*[*t*> *M*′, where

*M*′ (*p*) = 

where •*t* and *t*• represent a pre-set and pos-set of *t*.

A transition of *T* is represented by an empty rectangle, a place of *P* is represented by a circle, and every pair of nodes (*m*, *n*) in the flow relationship is represented by a directed arc that leads from *m* to *n*, and a token is represented by a black dot.

Fig. 1. A Petri net model: (A) a parallel structure; (B) a choice structure; (C) a loop structure; (D) Petri net is a sequential structure.

Fig. 1 shows a Petri net model. This model contains four simple structures of the Petri nets. Where (A) represents a concurrent structure, (B) represents a choice structure, (C) represents a loop structure, and (D) represents a sequential structure. In Fig. 1, activities *b* and *c* in a concurrent structure mean that activity *c* can directly follow activity *b* (i.e., *bc*), or activity *b* can directly follow activity *c* (i.e., *cb*) in a trace; Activities *f* and *g* in a choice structure mean that *f* and *g* can only be executed in any one trace; Activities *i* and *j* in a loop structure mean that *i* and *j* can appear any times in a trace; Activities *d* and *e* in a sequential structure mean that *d* and *e* are executed in sequence. A process model can consist of four basic structures [27], and any one of these structures can contain other structures or itself.

*Definition 7* (*Workflow* *Net*) [38]: *WPN* = (*PN*, *i*, *o*) is a workflow net, where:

1. *PN* = (*P*, *T*; *F*, *M*0) is a Petri net;
2. it has a unique initial place *i*, and •*i* = ∅;
3. it has a unique ending place *o*, and *o*• = ∅; and
4. for ∀*y* ∈ *P* ∪ *T*, *y* is on a path from *i* to *o*.

The workflow net is a special Petri net containing initial and ending places.

* 1. *Loop Structure*

The loop structure is a research focus in this paper, and it will be introduced in this section. If a process model has a general loop structure, we divide a loop structure into the main loop structure and the callback loop structure, respectively, as shown in Fig. 2(a). If there is no transition in the main loop structure, it is called a free loop structure, as shown in Fig. 2 (b). The length of a loop structure is represented by the number of transitions in a loop structure. We assume that the transition number of a loop structure is *n*. When *n* > 2, it is referred as a long loop structure; and when *n* ≤ 2, it is referred as a short loop structure; in particular, if n=1, it is referred as a self-loop, as shown in Fig. 2 (c).



Fig. 2. (a) A long loop. (b) A free loop. (c) A self-loop structure

* 1. *α-Algorithm*

According to the relationship between activities in a log, a classical process model mining algorithm called *Alpha* is proposed, and the relevant definitions are given below.

*Definition 8* (*Order Relations*): Let *L* ∈ ***B***(*A\**) be an event log over *A*. For ∀*c*, *d* ∈ *A*:

1. Follow relations: denoted as *c* >*L* *d* if and only if there is a trace *σ* = <*t*1, *t*2, *t*3, …, *tn*>, and *i* ∈ {1, 2, 3, …, *n*-1}, *σ* ∈ *L*, then *ti* = *c* and *ti+*1 = *d*;
2. Causal relations: denoted as *c* →*L* *d* if and only if *c* >*L* *d* and *d* ≯*L* *c*;
3. Concurrent relations: denoted as *c* ||*L* *d* if and only if *c* >*L* *d* and *d* >*L* *c*; and
4. Exclusive relations: denoted as *c* #*L* *d* if *c* ≯*L* *d* and *d* ≯*L* *c*.

For instance, *L* = {*σ*1 = <*ta*, *tb*, *tc*, *td*, *te*, *tf*>, *σ*2 = <*ta*, *tc*, *tb*, *td*, *te*, *tf*>} be an event log, where {*ta*, *tc*, *te*, *tb*, *tf*, *td*,} is a set of activities. Then the following relations contain *ta* >*L­ tb*, *tb* >*L­ tc*, *tc* >*L­ td*, *td* >*L­ te*, *ta* >*L­ tc*, *tc* >*L­ tb*, *tb* >*L­ td*, and *te* >*L­ tf*; the casual relations contain *ta* →*L tb*, *ta* →*L tc*, *tb* →*L td*, *tc* →*L td*, *td* →*L te*, and *te* →*L tf*; the concurrent relations contain *tb* ||*L­* *tc*; and the exclusive relations contain *ta* #*L td*, *ta* #*L te*, *ta* #*L tf*, *tb* #*L te*, *tb* #*L tf*, *tc*#*L te*, and *tc* #*L tf*.

*Definition 9* (*α-algorithm*): Let *L* be an event log over *T*. *α* (*L*) is defined as follows [1]:

1. *TL* = {*t* ∈ *T* | ∃*σ*∈*L**t* ∈ *σ*};
2. *TI* = {*t* ∈ *T* | ∃*σ*∈*L  t* = *first*(*σ*)};
3. *To* = {*t* ∈ *T* | ∃*σ*∈*L t* = *last*(*σ*)};
4. *X L* = {(*A*, *B*) |*A* ⊆ *TL* ∧ *A* ≠ ∅ ∧ *B* ⊆ *TL* ∧ *B* ≠ ∅ ∧∀ *a*∈*A* ∀*b*∈*B* *a* →*L* *b* ∧∀*a*, *a*∈*A a* 1 #*L* *a* 2 ∧∀*b*, *b*∈ B *b* 1 #*L* *b* 2 };
5. *YL* = {(*A*, *B*)} ∈ *X**L* |∀(*A′* , *B′* ) ∈ *X**A* ⊆ *A*′ ∧ *B* ⊆ *B*′ ⇒ (*A*, *B*) = (*A*′, *B*′)};
6. *PL* = {*p*(*A*, *B*) |(*A*, *B*) ∈ *YL* ∪ {*iL*, *oL*}};
7. *FL* = {(*a*, *p*(*A*, *B*)) |(*A*, *B*) ∈ *YL* ∧ *a* ∈ *A* } ∪ {(*p*(*A*,*B*), *B*) |(*A*, *B*) ∈ *YL* ∧ *b* ∈ *B*} ∪ {( *iL*, *t*) |*t* ∈ *TI*} ∪ {(*io*, *t*) |*t* ∈ *To*}; and
8. *α*(*L*) = (*PL*, *TL*, *FL*).

where, if *σ* is a trace, and *σ* = <*ta*, *tb*, *tc*, …, *tn*>, then *first*(*σ*) = *ta* and *last*(*σ*) = *tn*.

*Definition 10* (*Complete Event Log*) : Let *WPN* = (*PN*, *i*, *o*) be a workflow net. *L* is an event log of *WPN* if and only if *L* ∈ *B*(*T*∗) and every trace *σ* ∈ *L* is a firing sequence of *WPN* starting in state [*i*] and ending in state [*o*]. *L* is called a complete event log of *WPN* if and only if

(1) for ∀ *L*′ ∈ *B*(*T*∗) of *WPN*: >*L*′ ⊆ >*L* , and

(2) for ∀ *t* ∈ *T*, there is ∃*σ* ∈ *L* such that *t* ∈ *σ*.

*Definition 11* (*Activities′*  *indirect dependency*):Let *Ψ* be a set of all activities, *A* ⊆ *Ψ* be a subset over *Ψ*, *L* ∈ ***B***(*A\**) be an event log over *A*, for ∀*σ* ∈ *L*, *t*1 = *σ*[*i*] ∈*Ψ*, *i* ∈ {1, 2, 3, …, *n*}, and *t*2 = *σ*[*j*] ∈*Ψ*, *j* ∈ {*i*+2, *i*+3, *i*+4, …, *n*}, an indirect dependency between activities *t*1 and *t*2 is defined as *t*1 » *t*2.

1. Mining Indirect Dependencies

In this section, we propose a method for mining the indirect dependencies between the loop-choice branches driven loop structures. First, two algorithms are proposed to identify loop sequences and choice sequences from event logs, respectively. Then, in order to clearly describe the indirect dependencies between different structures, we introduce association rules. Next, we expand the ordinary Petri net and redefine the new transition firing rules. The process mining algorithm *AlphaID* is proposed at last.

* 1. *Identify Loop Structure*

To identify a loop structure from the given event log, we redefine the loop activity and divide the loop activities into a loop start activity and a loop end activity, and further obtain a sequence that satisfies the loop structure.

*Definition 12* (*Activity Number*) [39]: Let *σ* ∈ *Ψ*∗ be a trace, where *Ψ* be a set of all activities, for *m* ∈ *σ* is an activity, and *q* ∈ *σ* is a sequence, then *num*(*m*, *σ*) represents the number of *m* in *σ*, *num*(*q*, *σ*) represents the number of *q* in *σ*.

For instance, let *Ψ* = {*c*, *d*, *e*, *f*, *g*} be a set of activities, *σ* = <*c*, *d*, *e*, *d*, *e*, *f*, *g*> be a trace over *Ψ*, then *num*(*d*, *σ*) = 2, and *num*(*f*, *σ*) = 1. *q* = <*d*, *e*> be a sequence over *σ*, then *num*(*q*, *σ*) = 2.

*Definition 13* (*Loop Activity*): Let *L* ∈ ***B***(*A\**) be an event log over *A*, *L* does not containduplicate activities, and *σ* ∈ *L* be a trace. If *a* ∈ *σ*, and *num*(*a*, *σ*) > 1, then *a* ∈ *σ* is called a loop activity. The set of loop activities is represented by *SLA*, where *SLA* = {*a* ∈*Ψ* | ∃*a* ∈ *σ* ∧ *num*(*a*, *σ*) > 1}.

For instance, let *σ* = <*f*, *g*, *h*, *g*, *h*, *d*, *e*> ∈ *L* be a trace, and *num*(*g*, *σ*) = 2 > 1, *num*(*h*, *σ*) = 2 > 1, then *g* and *h* are called loop activities, and *SLA* = {*g*, *h*}.

*Definition 14* (*Loop Sequence*): Let *SLA* be a set of loop activities, ƒ = <ƒ[1], ƒ[2], …, ƒ[*j*], …., ƒ[*n*]> ∈ *SLA*∗ be a loop sequence over *SLA*, where for ƒ[*j*] ∈ *SLA*, if |ƒ| = 1, the loop ƒ is called self-loop sequence; if |ƒ| ≥ 2, where ƒ[*j*] >*L* ƒ[*j* + 1], *j* ∈ {1, 2, …, *n*-1} and ƒ[*n*] >*L* ƒ[1]. The set of loop sequences is represented by *SLS*, where *SLS* = {ƒ ∈ *SLA*∗| |ƒ| = 1∪ (ƒ[*j*] >*L* ƒ[*j* + 1] ∧ ƒ[*n*] >*L* ƒ[1], *j* ∈ {1, 2, …, *n*-1})}.

For instance, let *σ* = <*a*, *o*, *p*, *q*, *o*, *p*, *q*, *e*, *m*, *m*, *m*, *g*> ∈ *L* be a trace, where *SLA* = {*o*, *p*, *q*, *m*}, *o* >*L* *p*, *p* >*L* *q*, *q* >*L* *o*, and *m* >*L* *m*, then ƒ1 = <*o*, *p*, *q*>, and ƒ2 = <*m*> , *SLS* = {<*o*, *p*, *q*>, <*m*>}.

*Definition 15*: Let *Ψ* be a set of activities, *a* ∈ *Ψ* be an activity, and *s* ∈ *Ψ*∗ be a sequence over *Ψ*, then *a* + *s* represents a new sequence <*a*, *s*>, and *s* + *a* represents a new sequence <*s*, *a*>.

For instance, let *s* = <*g*, *h*, *l*, *k*, *m*> is a sequence, then *s*1 = *a* + *s* = <*a*, *g*, *h*, *l*, *k*, *m*>, *s*2 = *s* + *a* = < *g*, *h*, *l*, *k*, *m*, *a*>.

According to the definition of log completeness, the definition of loop completeness is obtained. This definition lays the foundation for finding indirect dependencies between loop structures.

*Definition 16* (*Loop -Complete* *Log*): Let *WPN* = (*PN*, *i*, *o*) is a workflow net and *L* be an event log of *WPN.* *L* is a loop-complete event log of *WPN* if and only if *L* is complete, and for ∀*L*′ of *WPN*: if there is ∃*σ*′ ∈ *L*′, and ƒ is a subsequence of *σ*′, then there is ∃*σ* ∈ *L*, and ƒ is a subsequence of *σ* .



Fig. 3. Two loop structures exist on branches of a concurrent structure

As shown in Fig. 3, loop structures exist on branches of a concurrent structure, an incomplete loop log may appear.

In Fig. 3, a complete event log similar to *L* = {<*m*, *n*, *l*, *f*>, <*m*, *n*, *e*, *l*, *f*>, <*m*, *e*, *n*, *l*, *f*>, <*m*, *n*, *e*, *l*, *f*>, <*m*, *n*, *l*, *e*, *f*>, <*m*, *n*, *l*, *e*, *d*, *f*>, <*m*, *n*, *e*, *l*, *d*, *e*, *n*, *l*, *f*>, <*m*, *n*, *l*, *e*, *d*, *n*, *l*, *f*>} may appear. However, the loop sequence ƒ1 = < *n*, *l*, *d* > does not appear in log *L*, and *L* is not a loop completeness log. Logs like this are not the focus of this paper, we will focus on loop completeness logs in the following sections.

*Definition 17* (*Loop start and end sets*): Let *SLA* be a set of loop activities, ƒ = <ƒ[1], ƒ[2], …, ƒ[*i*], …., ƒ[*n*]> ∈ *SLA*∗ be a loop sequence over *SLA*, the loop start activity set is represented by *Als*, and the loop end activity set is represented by *Ale*, where

1. *Als* = {*ai*∈ *SLA* | *ai* = ƒ*i*(1)}; and
2. *Ale* = {*ai*∈ *SLA* | *ai* = ƒ*i*(*n*)}, *i* ∈ {1, 2, …, *n*}.

Specially, if ∃*a*∈ *SLA*, *a*∈ *Als*, and *a*∈ *Ale*, then the loop ƒ is called a self-loop sequence.

For instance, let *SLs* = {<*a*, *b*, *d*>, <*m*, *f*, *n*>, <*p*, *i*, *r*, *t*>} is a set of loop sequences, then *Als* = {*a*, *m*, *p*}, and *Ale* = {*d*, *n*, *t*}.

Now, Algorithm 1 for identifying *Als* and *Ale* is proposed. For *c*, *d* ∈ *Ψ*, and if *c* >*L­ d*, then *Adir\_set* represents a set of (*c*, *d*), where *Adir\_set* = {(*c*, *d*)| *c*, *d*∈*Ψ* ∧ *c* >*L­ d*}.

In Algorithm 1, step 1 initializes variable *Adir\_set*, *Als*, *Ale*, and *SLA*. Steps 2-8 traverse all traces in a loop completeness log and get a set of activity pairs that satisfy the following relationship. Steps 9-13 obtain all the loop activities in a log according to the characteristics of the activities in the trace and add the loop activities into *SLA*. Steps 14-20 traverse all activities of *SLA*, get the loop start activities and add them into *Als.* Steps 21-27 traverse all activities of *SLA*, get the loop end activities and add them into *Ale.* At last, *Als* and *Ale* are obtained in step 28.

**Example 1:** Let *L* = {<*a*, *q*, *c*, *d*, *o*, *p*, *h*, *j*>, <*a*, *q*, *c*, *d*, *e*, *c*, *d*, *o*, *p*, *h*, *i*, *p*, *h*, *j*>, <*a*, *q*, *c*, *d*, *e*, *c*, *d*, *e*, *c*, *d*, *o*, *p*, *h*, *i*, *p*, *h*, *i*, *p*, *h*, *j*>}is a loop completeness log. Firstly, *Adir\_set* = {(*a*, *q*), (*d*, *e*), (*c*, *d*), (*e*, *c*), (*d*, *o*), (*q*, *c*), (*o*, *p*), (*p*, *h*), (*h*, *i*), (*i*, *p*), (*h*, *j*)} can be obtained from Algorithm 1. According to *Definition 12*, *num*(*c*, *σ*2) = *num*(*d*, *σ*2) = *num*(*p*, *σ*2) = *num*(*h*, *σ*2) = 2 and *num*(*e*, *σ*3) = *num*(*c*, *σ*3) = *num*(*d*, *σ*3) = *num*(*p*, *σ*3) = *num*(*h*, *σ*3) = *num*(*i*, *σ*3) = 2, thus the set of loop activities *SLA* = {*c*, *d*, *e*, *p*, *h*, *i*}. Then *c* ∈ *SLA*, (*q*, *c*) ∈ *Adir\_set* and *q* ∉ *SLA*, thus *Als* = {*c*}, and *p* ∈ *SLA*, (*o*, *p*) ∈ *Adir\_set* and *o* ∉ *SLA*, thus *Als* = {*c*, *p*}. Similarly, *Ale* = {*e*, *i*}is easy to get.

|  |
| --- |
| **Algorithm 1** Start and End Activity Sets of Loop Structure |
| **Input:** A loop completeness log *L* ∈ *B*(*A*∗)  **Output:**  *Als*, *Ale* |
| 1. *Adir\_set* ← ∅, *Als* ← ∅, *Ale* ← ∅, *SLA* ← ∅  2. **for** each *δ* ∈ *L* **do**  3. **for** each *a*, *b* ∈ *σ* **do**  4. **if** *δ* = <*t*1, *t*2, …, *ti*>, *i* ∈ {1, 2, 3, …, *n*-1}**then**  5. *Adir\_set* ← *Adir\_set* + (*a*, *b*);  6. **end if**  7.  **end for**  8. **end for**  9. **for** each *δ* ∈ *L* **do**  10.  **if ∃***a* ∈ *δ*, and  *num*(*a*, *σ*) > 1 **then**  11.  *SLA* ← *SLA* + *a*;  12. **end if**  13. **end for**  14. **for** each *a* ∈ *SLA* **do**  15. **for** each (*c*, *d*) ∈ *Adir\_set* **do**  16. **if** *a* = *d*, and *c* ∉ *SLA* **then**  17. *Als* ← *Als* + *a*;  18. **end if**  19.  **end for**  20. **end for**  21. **for** each *a* ∈ *SLA* **do**  22. **for** each (*c*, *d*) ∈ *Adir\_set* **do**  23. **if** *a* = *c*, and *d* ∈ *Als* **then**  24. *Ale* ← *Ale* + *a*;  25. **end if**  26.  **end for**  27. **end for**  28. **return**  *Als*, *Ale* |

Through Algorithm 1, we can get *Als* and *Ale*, and then take them as inputs to propose an algorithm for obtaining a loop sequence . For *m*, *n* ∈ *Ψ*,(*m*, *n*) ∈ *Adir\_set*, and if *m*, *n*∈ *SLA*, then *Aloop\_dir* represents a set of (*m*, *n*), where *Aloop\_dir* = {(*m*, *n*) ∈ *Adir\_set* | *m*, *n* ∈ *SLA*}. From example 1, we can get *Aloop\_dir* = {(*c*, *d*), (*p*, *h*), (*e*, *c*), (*h*, *i*), (*d*, *e*), (*i*, *p*)}.

|  |
| --- |
| **Algorithm 2** Loop Sequence |
| **Input:** A loop completeness log *L* ∈ *B*(*A*∗), *Als*, *Ale*,  *Adir\_set*  **Output:** *SLS* |
| 1. *Aloop\_dir* ← ∅, ƒ ← ∅  2. **for** each (*a*, *b*) ∈ *Adir\_set* **do**  3. **if** *a* ∈ *SLA* and *b* ∈ *SLA*  **then**  4. *Aloop\_dir* ← *Aloop\_dir* + (*a*, *b*);  5. **end if**  6. **if** *a* ∈ *SLA*, *b* ∈ *SLA*, and (*b*, *a*) ∈ *Adir\_set*  **then**  7. *Aloop\_dir* ← *Aloop\_dir* - (*a*, *b*) - (*b*, *a*);  8. **end if**  9. **end for**  10. **for** each *a* ∈ *Als* **do**  11.ƒ ← ƒ + *a*;  12.  **if** *a* ∈ *Ale* **then**  13.  **break;**  14.  **for** each (*m*, *n*) ∈ *Adir\_set* **do**  15.  **if** *a* = *m*, and *n* ∈ *SLA* **then**  16. ƒ ← ƒ + *n*;  17. **for** each (*c*, *d*) ∈ *Aloop\_dir* **do**  18. **if** *n* = *c*, and *d* ∉ *Ale* **then**  19. ƒ ← ƒ + *d*;  20. **end if**  21. **if** *n* = *c*, and *d* ∈ *Ale* **then**  22. ƒ ← ƒ + *d*;  23.  **break;**  24. **end if**  25.  **end for**  26.  **end if**  27. **end for**  28.  *SLS* ← *SLS* + ƒ;  29. ƒ ← ∅;  30. **end for**  31. **return**  *SLS* |

In Algorithm 2, step 1 initializes variable *Aloop\_dir*, and ƒ. Steps 2-9 traverse all *Adir\_set* and get the set of activity pairs that satisfy the definition of *Aloop\_dir*. Steps 10-27 traverse all *Als*, traverse all *Adir\_set* and traverse all *Aloop\_dir*, then the activities that satisfy the conditions are added to a loop sequence ƒ, and the complete loop sequence ƒ can be obtained. Step 28 adds the obtained loop sequence ƒ to a set of loop sequences *SLS*. At last, *SLS* is obtained in step 31.

**Example 2:** Let *L* = {<*a*, *k*, *c*, *f*, *g*, *h*, *j*>, <*a*, *k*, *c*, *d*, *e*, *k*, *c*, *f*, *g*, *h*, *i*, *m*, *n*, *j*>, <*a*, *k*, *c*, *d*, *e*, *k*, *c*, *d*, *e*, *k*, *c*, *f*, *m*, *n*, *i*, *m*, *n*, *j*>, <*a*, *k*, *c*, *d*, *e*, *k*, *c*, *d*, *e*, *k*, *c*, *f*, *m*, *n*, *i*, *m*, *n*, *i*, *m*, *n*, *j*>}is a loop completeness log. We can get *Adir\_set* = {(*a*, *k*), (*k*, *c*), (*d*, *e*), (*c*, *d*), (*e*, *k*), (*f*, *m*), (*c*, *f*), (*n*, *i*), (*m*, *n*), (*i*, *m*), (*n*, *j*)}, *SLA* = {*k*, *c*, *e*, *d*, *m*, *n*, *i*}, *Als* = {*k*, *m*}, and *Ale* = {*e*, *i*} from Algorithm 1. In steps 2-9 of Algorithm 2, we can get *Aloop\_dir* ={(*k*, *c*), (*d*, *e*), (*m*, *n*), (*e*, *k*), (*c*, *d*), (*i*, *m*), (*n*, *i*)}. Then *k* ∈ *Als*, ƒ1 = <*k*>, and (*k*, *c*) ∈ *Adir\_set*, *c* ∈ *SLA*, thus ƒ1 = <*k*, *c*>, and (*c*, *d*) ∈ *Aloop\_dir*, *d* ∉ *Ale*, thus ƒ1 = <*k*, *c*, *d*>, and (*d*, *e*) ∈ *Aloop\_dir*, *e* ∈ *Ale*, thus ƒ1 = <*k*, *c*, *d*, *e*>. Similarly, we can get ƒ2 = <*m*, *n*, *i*>. At last, *SLS* = {<*k*, *c*, *d*, *e*>, <*m*, *n*, *i*> }is obtained.

* 1. *Identify Choice Structure*

If a process model has a non-free choice structure, there may be indirect dependencies between different activities. In this subsection, in order to obtain the non-free choice structure from a given log, we redefine the activities in the non-free choice structure and define the choice sequence. Notice that the choice structure located in the sequential structure is the object of our study.

Fig. 4. A choice structure exists in a sequential structure.

*Definition 18* (*Choice Activity*): Let *L* ∈ ***B***(*A\**) be an event log over *A*, *σ* ∈ *L* be a trace, and ∃*σ*1, *σ*2 ∈ *L*, if *b* ∈ *σ*1, *b* ∉ *σ*2, and *b* ∉ *SLA*, then *b* ∈ *σ* is named a choice activity. The set of choice activities is represented by *SCA*, where *SCA* = {*b* ∈*Ψ* |∃*σ*1, *σ*2 ∈ *L* ∧ *b* ∈ *σ*1 ∧ *b* ∉ *σ*2 ∧ *b* ∉ *SLA*}.

For instance, Fig. 4 is a process model, *L* is a complete event log, i.e., *L* = {*σ*1 = <*a*, *b*, *c*, *d*, *g*>, *σ*2 = <*a*, *b*, *e*, *f*, *g*>}can be obtained. According to *Definition 18*, we can find that *e*, *f* ∈ *σ*2, *e*, *f* ∉ *σ*1, and *c*, *d* ∈ *σ*1, *c*, *d* ∉ *σ*2. Then we can get *SCA* = {*c*, *d*, *e*, *f*}.

*Definition 19* (*Choice start and end sets*): Let *SCA* be a set of choice activities, the choice start activity set is represented by *Acs*, and the choice end activity set is represented by *Ace*, where

1. *Acs* = {*ai* ∈ *SCA* | ∃*b*∈*Ψ* (*bi*, *ai*) ∈ *Adir\_set* ∧ *bi* ∉ *SCA* }; and
2. *Ace* = {*ai* ∈ *SCA* | ∃*b*∈*Ψ* (*ai*, *bi*) ∈ *Adir\_set* ∧ *bi* ∉ *SCA* ∧ *bi* ∉ *Acs*}, *i* ∈ {1, 2, …, *n*}.

For instance, let *L* = {*σ*1 = <*a*, *n*, *m*, *l*>, *σ*2 = <*a*, *n*, *d*, *e*, *l*>, *σ*3 = <*a*, *n*, *f*, *g*, *h*, *l*>, *σ*4 = <*a*, *n*, *i*, *j*, *k*, *l*>} is a completeness log, we can get *SCA* = { *d*, *m*, *e*, *f*, *j*, *g*, *k*, *h*, *i*, }. Then the choice start activity set *Acs* = {*m*, *d*, *f*, *i*}, and the choice end activity set *Ace* = {*e*, *h*, *k*} can be obtained by *Definition 19* From the results we can find the number of activities in *Acs* is one more than the number of activities in *Ace*, which means that there is only one activity on a branch of the choice structure.

According to *Definition 18* and *Definition 19*, to obtain the choice start activity set *Acs* and the choice end activity set *Ace*, we propose the following algorithm.

In Algorithm 3, step 1 initializes variable *Acs*, *Aes*, and *SCA*. Steps 2 - 4 traverse all *δ* and get the set of the choice activities *SCA*. The loop activities may exist in *SCA*, therefore, in steps 5 - 9, we remove the activities that belong to the loop activity set *SLA*. Steps 10 - 16 obtain the choice start activity set *Acs* with traversing *SCA* and *Adir\_set*. Steps 17 -23 traverse all *SCA* without *Acs* and get the choice end activity set *Acs*. At last, *Acs* and *Ace* are returned in step 24.

**Example 3:** Let *L* = {*σ*1 = <*a*, *t*, *c*, *p*, *n*>, *σ*2 = <*a*, *t*, *c*, *s*, *e*, *p*, *n*>, *σ*3 = <*a*, *t*, *f*, *g*, *r*, *p*, *n*>, *σ*4 = <*a*, *t*, *i*, *j*, *k*, *l*, *p*, *n*>}is a complete log. Firstly, we can get *Adir\_set* = {(*a*, *t*), (*t*, *c*), (*t*, *s*), (*d*, *e*), (*t*, *f*), (*f*, *g*), (*g*, *r*), (*t*, *i*), (*i*, *j*), (*j*, *k*), (*k*, *l*), (*l*, *p*), (*p*, *n*)}, *φset*(*L*) = {*a*, *t*, *c*, *s*, *e*, *f*, *g*, *r*, *i*, *j*, *k*, *l*, *p*, *n*}, *φset*(*σ*1) = {*a*, *t*, *c*, *p*, *n*}, *φset*(*σ*2) = {*a*, *t*, *c*, *s*, *e*, *p*, *n*}, *φset*(*σ*3) = {*a*, *t*, *f*, *g*, *r*, *p*, *n*}, *φset*(*σ*4) = {*a*, *t*, *i*, *j*, *k*, *l*, *p*, *n*}. Then we can get *SCA* = { *k*, *c*, *s*, *i*, *e*, *j*,*f*, *g*, *r*, *l*} from steps 2-9 in

|  |
| --- |
| **Algorithm 3** Start and End Activity Sets of the Choice Branch |
| **Input:** A loop completeness log *L* ∈ *B*(*A*∗), *Adir\_set*,  *SLA*  **Output:** *Acs*, *Ace* |
| 1. *Acs* ← ∅, *Ace* ← ∅, *SCA* ← ∅  2. **for** each *δ* ∈ *L* **do**  3.  *SCA* ← *SCA* + *φset*(*L*) - *φset*(*σ*);  4. **end for**  5. **for** each *a* ∈ *SCA* **do**  6. **if** *a* ∈ *SLA* **then**  7.  *SCA* ← *SCA* - *a*;  8. **end if**  9. **end for**  10. **for** each *a* ∈ *SCA* **do**  11. **for** each (*m*, *n*) ∈ *Adir\_set* **do**  12. **if** *n* = *a*, and *m* ∉ *SCA* **then**  13. *Acs* ← *Acs* + *a*;  14. **end if**  15.  **end for**  16. **end for**  17. **for** each *a* ∈ *SCA* - *Acs* **do**  18. **for** each (*m*, *n*) ∈ *Adir\_set* **do**  19. **if** *n* = *a*, and *n* ∉ *SCA* **then**  20. *Ace* ← *Ace* + *a*;  21. **end if**  22.  **end for**  23. **end for**  24. **return**  *Acs*, *Ace* |

Algorithm 3. Next, for *c* ∈ *SCA*, (*t*, *c*) ∈ *Adir\_set*, and *t*∉ *SCA*, then *Acs* = {*c*}, then Algorithm 3 traverse all activities in *SCA*, and get the choice start activity set *Acs* = {*c*, *s*, *f*, *i*}. Similarly, *Aes* = {*e*, *r*, *l*} can be obtained.

*Definition 20* (*Choice-branch Sequence*) Let *SCA* be a set of choice activities, *γ* = <*γ*[1], *γ*[2], …, *γ*[*i*], …., *γ*[*n*]> ∈ *SCA*∗ be a choice-branch sequence over *SLA*, where for *γ*[*i*] ∈ *SCA*, if |*γ*| = 1, then only one activity exists on the choice branch; if |*γ*| ≥ 2, *γ*[*i*] >*L* *γ*[*i* + 1], *i* ∈ {1, 2, …, *n*-1}.The set of choice-branch sequence is represented by *SCS*, where *SCS* = {*γ* ∈ *SCA*∗| |*γ*| = 1∪ (*γ* [*i*] >*L* *γ* [*i*+1], *i* ∈ {1, 2, …, *n*-1})}.

For instance, let *L* = {*σ*1 = <*a*, *m*, *n*, *i*>, *σ*2 = <*a*, *m*, *q*, *w*, *i*>, *σ*3 = <*a*, *m*, *f*, *g*, *h*, *i*>}is a complete log, we can get *SCA* = {*n*, *q*, *w*, *f*, *g*, *h*}, *Adir\_set* = {(*m*, *c*), (*m*, *d*), (*q*, *w*), (*m*, *f*), (*f*, *g*), (*g*, *h*), (*h*, *i*)}, *Acs* = {*n*, *q*, *f*}, and *Ace* = {*w*, *h*}. Then we can obtain *γ*1 = <*n*>, *γ*2 = <*q* *w*>, *γ*3 = <*f*, *g*, *h*>, and *SCS* = {<*n*>, <*q*, *w*>, <*f*, *g*, *h*>} according to *Definition 20*.

We propose an algorithm to get choice-branch sequences from a complete event log as follows.

In Algorithm 4, step 1 initializes variable *SCS*, and *γ*. Steps 2 - 13 indictate that according to *Definition 15*, if *a* ∈ *Acs*, we can get *γ* = *γ* + *a*. Then, for each (*a*, *b*) ∈ *Adir\_set*, if *b*∈ *SLA*, and *b* ∉ *Ace*, *γ* = *γ* + *b*, algorithm continues to execute; if *b*∈ *SLA*, and *b* ∈ *Ace*, *γ* = *γ* + *b*, algorithm jumps out to execute, we can get the choice-branch sequence *γ*. Step 14 indictates that we put the choice-branch sequence *γ* into the set of choice-branch sequence *SCS*. Finally, step 17 returns *SCS*.

|  |
| --- |
| **Algorithm 4**  Choice - Branch Sequence |
| **Input:** A loop completeness log *L* ∈ *B*(*A*∗), *Acs*, *Ace*, *Adir\_set*, *SCA*  **Output:** *SCS* |
| 1.  *SCS* ← ∅, *γ* ← ∅  2. **for** each *a* ∈ *Acs* **do**  3.*γ* ← *γ* + *a*;  4.  **for** each (*p*, *q*) ∈ *Adir\_set* **do**  5.  **if** *p* = *a*, and *q* ∈ *SLA*, and *q* ∉ *Ace* **then**  6.  *γ* ← *γ* + *q*;  7. **end if**  8. **if** *p* = *a*, and *q* ∈ *Ace* **then**  9. *γ* ← *γ* + *q*;  10.  **break;**  11. **end if**  12.  **end for**  13. **end for**  14.  *SCS* ← *SCS* ∪ *γ* ;  15. *γ* ← ∅;  16. **end for**  17. **return**  *SCS* |

An example is given to illustrate the execution process of Algorithm 4.

**Example 4:** Let *L* = {*σ*1 = <*a*, *p*, *d*, *q*, *e*, *f*, *o*, *m*>, *σ*2 = <*a*, *p*, *q*, *d*, *e*, *r*, *h*, *o*, *m*>, *σ*3 = <*a*, *p*, *d*, *q*, *e*, *s*, *j*, *k*, *o*, *m*>, *σ*4 = <*a*, *p*, *q*, *d*, *e*, *s*, *j*, *k*, *o*, *m*>, *σ*5 = <*a*, *p*, *q*, *d*, *e*, *f*, *o*, *m*>}is a complete log. *Adir\_set* = {(*a*, *p*), (*p*, *q*), (*p*, *d*), (*d*, *q*), (*q*, *d*), (*d*, *e*), (*q*, *e*), (*e*, *f*), (*e*, *r*), (*r*, *h*), (*e*, *s*), (*s*, *j*), (*j*, *k*), (*k*, *o*), (*f*, *o*), (*h*, *o*), (*k*, *o*), (*o*, *m*)}, *SCA* = {*f*, *r*, *h*, *s*, *j*, *k*}, *Acs* = {*r*, *f*, *s*}, and *Aes* = {*h*, *k*}can be obtained from Algorithm 1 and Algorithm 3. Then ,we can get *γ*1 = <*f*>, *γ*2 = <*r*, *h*>, *γ*3 = <*s*, *j*, *k*>, and *SCS* = {<*f*>, <*r*, *h*>, <*s*, *j*, *k*>}.

* 1. *Association Rules*

Discovering association rules is an important task in data mining. Now, we apply association rules to process mining in his section,. The association rules are used to discover indirect dependencies between activities or different structural sequences. Currently, most algorithms mine process models through direct dependencies between activities. However, if the indirect dependencies can be found from the event log, we will get a more precise process model.

*Definition 21* (*Association Pair*): Let *L* ∈ ***B***(*A\**) be an event log over *A*, ƒ1, ƒ2 ∈ *SLS* be loop sequences, *γ* ∈ *SCS* be a choice-branch sequence, *a* ∈ ƒ1, *b* ∈ ƒ2, *a*, *b* ∈ *Ale*, *c* ∈ *γ*, *c* ∈ *Acs*, and *σ* ∈ *L* be a trace. An association pair is denoted as ω = (X, Y), where X = ƒ1*num*(*a*, *σ*) ∧ *γ num*(*c*, *σ*), Y = ƒ2*num*(*b*, *σ*). The set of ω is denoted as *Aps*.

According to *Definition 21*, if consider the pair of ω, X and Y, that are not directly connected, and there are other activities M between them. Then the pair of ω is said to be indirectly associated. Fig. 5 illustrates an indirect association.



Fig. 5. Indirect association between X and Y.

For instance, let ω = (<*a*, *b*, *d*>2 ∧ <*e*, *f*>1, <*l*, *i*, *s*, *k*>2) be an association pair. We can get ƒ1 = <*a*, *b*, *d*>, *γ* = <*e*, *f*>, ƒ2 = <*l*, *i*, *s*, *k*>, and *d*, *k* ∈ *Ale*, *e* ∈ *Acs*. The association pair of ω indicates that the loop sequence ƒ1 is executed two times and the choice-branch *γ* is executed one time, then the loop sequence ƒ2 will be executed two times.

*Definition 22* (*Sub-trace*): Let *L* ∈ ***B***(*A\**) be an event log over *A*, ƒ∈ *SLS* be a loop sequence, *γ* ∈ *SCS* be a choice-branch sequence, *σ* ∈ *L* be a trace. *γ* and ƒ are called sub-trace of *σ* if only if *γ* ⊆ *σ*, and ƒ ⊆ *σ*.

For instance, let *σ* = <*a*, *b*, *e*, *d*, *b*, *e*, *d*, *b*, *e*, *g*, *h*, *m*, *n*, *k*> be a trace, ƒ = <*b*, *e*, *d*> be a loop sequence. Since ƒ ⊆ *σ*, we can get a sub-trace over *σ* is ƒ.

*Definition 23* (*Sub-trace number*): Let *L* ∈ ***B***(*A\**) be an event log over *A*, ƒ∈ *SLS* be a loop sequence, *σ* ∈ *L* be a trace, and if ƒ ⊆ *σ*, then the number of ƒ in *σ* is represented as *num*(ƒ, *σ*).

For instance, let *σ* = <*b*, *m*, *n*, *q*, *m*, *n*, *q*, *m*, *n*, *g*, *h*, *s*, *r*, *k*> be a trace, ƒ = <*m*, *n*, *q*> be a loop sequence, then *num* (ƒ, *σ*) = 2.

*Definition 24*  Let *L* ∈ ***B***(*A\**) be an event log over *A*, *σ* = <*β*1, ƒ, ƒ, *β*2, *β*3, *β*4, …, *βn*> ∈ *L* be a trace. For ƒ∈ *SLS*, if ƒ, *βi* ⊆ *σ*, *i* ∈ {1, 2, …, *n*}, then η = *delete* (ƒ, *σ*) = {<*β*1>, <*β*2, *β*3, *β*4,…, *βn*>}. Similarly, for *γ* ∈ *SCS*, *σ* = <*β*1, *β*2, *β*3, *γ*, *β*4, …, *βn*> ∈ *L*, then η = *delete* (*γ*, *σ*) = {<*β*1, *β*2, *β*3>, <*β*4,…, *βn*>}.

For instance, let *σ* = <*m*, *b*, *n*, *d*, *b*, *n*, *d*, *b*, *n*, *e*, *f*, *i*, *j*, *k*> be a trace, if ƒ = <*b*, *n*, *d*> ⊆ *σ* be a loop sequence, then *delete* (ƒ, *σ*) = {<*m*>, <*e*, *f*, *i*, *j*, *k*>}, and if *γ* = <*i*, *j*> ⊆ *σ* be a choice-branch sequence, then *delete* (*γ*, *σ*) = {<*m*, *b*, *n*, *d*, *b*, *n*, *d*, *e*, *f* >, <*k*>}.

*Theorem 1*: For ∀*σ* ∈ *L*, ∀ƒ*m*, ƒ*n* ⊆ *σ*, and ƒ*m*, ƒ*n* ∈ *SLS*, if ∃*c*, *d* ∈ *Ale*, *c* ∈ ƒ*m*, *d* ∈ ƒ*n*, and *c* *#L* *d*, then *num*(ƒ1, *σ*) = *num*(*c*, *σ*), and *num*(ƒ2, *σ*) = *num*(*d*, *σ*).

*Proof*: When the loop structures exist in a sequential structure, and there is no loop nesting. In fig. 2(a), a loop structure contains the main loop structure and the callback loop structure. A loop sequence ƒ1 = <*b*, *c*, *d*, *e*, *m*>, *b* ∈ *Als*, and *m* ∈ *Als*, we can find that if ∃*b* ∈*σ*, ƒ1 ⊆ *σ* may not exist, however, if ∃*m* ∈*σ*, ƒ1 ⊆ *σ* must exist. Similarly, fig.2(b) and fig. 2(c) are special cases of fig. 2(a), therefore, in ∀*σ* ∈ *L*, we can use the number of loop end activities to represent the number of loop sequences.

By analyzing the existing mining algorithms, we find that most mining algorithms rely on direct following relationships to mine process models. There are also a few algorithms that can mine indirect dependencies between activities, however, no algorithm has been used to mine indirect dependencies between multiple conditions and a result. For instance, a loop sequence count and a choice branch count decide anther loop sequence count. With above in mind, we propose an algorithm to mine loop and choice branch driven - loop structures as follows.

|  |
| --- |
| **Algorithm 5**  Loop and Choice Branch Driven loop |
| **Input:** A loop completeness log *L* ∈ *B*(*A*∗), *Ale*, *Acs*, *SLS*, *SCS*  **Output:** |
| 1. ω ← ∅, η ← ∅, *Aps* ← ∅  2. **for** each ƒ*i*  ∈ *SLS* **do**  3.  **for** each *σ* ∈ *L* **do**  4.  **if** ƒ*i* ⊄ *σ* **then**  5.  **continue**;  6.  **end if**  7.  **if** ƒ*i* ⊆ *σ* **then**  8. **if** *a* ∈ƒ*i*,and  *a* ∈ *Ale* **then**  9.  *num*(*a*, *σ*);  10. **end if**  11. η =  *delete* (*f*, *σ*);  12.  **for** each *γ* ∈ *SCS* **do**  13. **if** *γ*⊄ η[2]**then**  14.  **continue**;  15. **end if**  16.  **if** *γ* ⊆η[2] **then**  17. **if** *b* ∈ *γ*,and  *b* ∈ *Acs* **then**  18.  *num*(*b*, *σ*);  19. **end if**  20. η1 = *delete* (*γ*, η[2]);  21.  **for** each ƒ*j* ∈ *SLS* **do**  22. **if** ƒ*j*⊄ η1[2]**then**  23.  **continue**;  24. **end if**  25.  **if** ƒ*j* ⊆η1[2] **then**  26. **if** *e* ∈ƒ*j*,and  *e* ∈ *Ale* **then**  27.  *num*(*e*, *σ*);  28. **end if**  29. ω ← ω + <ƒ*i* *num*( *a* , *σ*) ∧ *γ**num*( *b*, *σ*), ƒ*j* *num*( *e*, *σ*)>;  30. *Aps* ← *Aps* + ω;  31. **end if**  32.  **end for**  33. **end if**  34. **end for**  35. **end if**  36. **end for**  37. **end for**  38. **return**  *Aps* |

In Algorithm 5, step 1 initializes variable ω, η, and *Aps*. Steps 2-11 traverse the set of loop sequences *SLS*, and *σ*. If a loop sequence ƒ*i­* is a sub-trace of *σ*, then a trace *σ* is divided into two sub-traces η, and the number of the loop sequence ƒ*i­* can be obtained from *num*(*a*, *σ*), where *a* ∈ ƒ*i*,and *a* ∈ *Ale*. Steps 12-20 traverse the set of choice-branch sequences *SCS­*, and get the choice-branch sequence *γ* behind the loop sequence ƒ*i­*. If a choice-branch sequence *γ* a sub-trace of η[2], then a sub-trace η[2] is divided into two sub-traces η1, and the number of the choice-branch sequence *γ* can be obtained from *num*(*b*, η[2]), where *b* ∈ *γ*,and *b* ∈ *Acs*. Steps 21-28 traverse the set of loop sequences *SLS*. If a loop sequence ƒ*j* is a sub-trace of η1[2], and the number of the loop sequence ƒ*j* can be obtained from *num*(*e*, *σ*), where *e* ∈ƒ*j*,and *a* ∈ *Ale*. Steps 29-30 indicate that according to *Definition 21*, we can get the association pairs ω and the set of association pairs *Aps*. Steps 38 returns *Aps* at last.

Now, an example is given to illustrate the execution process of Algorithm 5

**Example 5:** Let *L* = {*σ*1 = <*a*, *u*, *s*, *f*, *g*, *t*, *m*, *v*, *p*>, *σ*2 = <*a*, *u*, *c*, *s*, *q*, *u*, *c*, *s*, *f*, *h*, *r*, *t*, *m*, *v*, *o*, *m*, *v*, *p*>, *σ*3 = <*a*, *u*, *c*, *s*, *q*, *u*, *c*, *s*, *f*, *h*, *r*, *m*, *v*, *p*>, *σ*4 = <*a*, *u*, *c*, *s*, *q*, *u*, *c*, *s*, *f*, *j*, *k*, *t*, *m*, *v*, *o*, *m*, *v*, *o*, *m*, *v*, *p*>, *σ*5 = <*a*, *u*, *c*, *s*, *q*, *u*, *c*, *s*, *e*, *u*, *c*, *s*, *f*, *g*, *t*, *m*, *v*, *o*, *m*, *v*, *o*, *m*, *v*, *o*, *m*, *v*, *p*>}is a complete log. According to algorithm 1 - 4, we can obtain *SLS* = {<*u*, *c*, *s*, *q*>, <*m*, *v*, *o*>}, *SCS* = {<*g*>, <*h*, *r*>, <*j*, *k*>}, *Ale* = {*q*, *o*}, and *Acs* = {*g*, *h*, *j*}. First, ƒ1 = <*u*, *c*, *s*, *q*>, ƒ1 ⊆ *σ*2, *e* ∈ƒ1,and  *e* ∈ *Ale*, then we can get *num*(*q*, *σ*2) = 1, and η = {<*a>*, <*u*, *c*, *s*, *f*, *h*, *r*, *t*, *m*, *v*, *o*, *m*, *v*, *p>*}. Next, since *γ =* <*h*, *r*>, *γ* ⊆ η[2] = <*u*, *c*, *s*, *f*, *h*, *r*, *t*, *m*, *v*, *o*, *m*, *v*, *p*>, *h* ∈ *γ*,and *h* ∈ *Acs*, we can get *num*(*h*, *σ*2) = 1, and η1 = {<*u*, *c*, *s*, *f*>, <*t*, *m*, *v*, *o*, *m*, *v*, *p>*}. Then, ƒ2 = <*m*, *v*, *o*>, ƒ2 ⊆η1[2] = <*t*, *m*, *v*, *o*, *m*, *v*, *p>*, *o* ∈ƒ2,and *o* ∈ *Ale*, we can get *num*(*m*, *σ*2) = 1, ω = <<*u*, *c*, *s*, *q*>1 ∧ <*h*, *r*>1, <*m*, *v*, *o*>1>, and *Aps* = {ω = <<*u*, *c*, *s*, *q*>1 ∧ <*h*, *r*>1, <*m*, *v*, *o*>1>}. Similarly, we obtain *Aps* ={<<*u*, *c*, *s*, *q*>1 ∧ <*h*, *r*>1, <*m*, *v*, *o*>1>, <<*u*, *c*, *s*, *q*>1 ∧ <*g*, *k*>1, <*m*, *v*, *o*>2>, <<*u*, *c*, *s*, *q*>2 ∧ <*g*>1, <*m*, *v*, *o*>3>} at last.

According to Algorithm 5, we can find all association pairs that satisfy *Definition 21* from a given event log. As we know, the purpose of association rule learning mining is to analyze the potential association relationships between data items from the object data set, similar to finding a sufficient condition that if X occurs, then Y will occur, which always reveals the information that is beneficial to us [1]. Association rules can be expressed as X ⇒ Y, where X is often referred to as an antecedent of the association rules, Y is often referred to as a consequent of association rules. The combination of X and Y can be any variables. Based on the above, we can combine the association pairs in Algorithm 5 into association rules to represent the indirect dependencies between different structures. The association rule is defined as Mω = (X ⇒ Y), where X = ƒ1*num*(*f* 1, *σ*) ∧ *γ num*(*γ*, *σ*), Y = ƒ2*num*(*f* 2, *σ*). In addition, the set of association rules is represented by *ΡR*, the set of the antecedent of association rules X is represented by *R\_pre* and the set of the consequent of association rules Y is represented by *R­\_pos*.

In data mining, there are generally three metrics to evaluate association rules: *support*, *confidence*, and *lift*. To obtain a valid association rule, *minsup* and *minconf* are generally defined in data mining, where *minsup* represents the minimum support degree and *minconf* represents the minimum confidence degree. If an association rule is valid, its *minsup* should be the value of *support*, i.e., *minsup*  *Support*(*Mω*); similarly, its *minconf* should be lower than the value of *confidence*, i.e., *minconf*  *confidence* (*Mω*).

Therefore, in order to mine the indirect dependencies between different structures, we apply association rules to process mining and extend the two measures of evaluating association rules, i.e., the *support* and *confidence*. The specific definitions are given below.

*Definition 25* (*Extended-support*) Let *L* ∈ ***B***(*A\**) be an event log over *A*, *ω* be an association pair, *Aps* be a set of association pairs, *Mω* ∈ *ΡR* is an association rule, and the number of occurrences of ω in *L* is represented by *Aps*(*ω*). Then the extended support of association rule *Mω* is represented by *Extended-support*(*Mω*), and it is calculated as follows.



*Extended-support*(*Mω*) =

*Extended-support* indicates the applicability of association rules. In general, the higher the level of *Extended-support*, the more useful it is for us to mine valid association rules. From *Definition 25*, the value of *Extended-support* (*Mω*) is 0 to 1.

For instance, Let *L* = <*σ*1 = <*a*, *b*, *o*, *r*, *g*, *l*, *m*, *n*, *p*>, *σ*2 = <*a*, *b*, *c*, *o*, *e*, *b*, *c*, *o*, *r*, *h*, *i*, *m*, *n*, *p*>, *σ*3 = <*a*, *b*, *c*, *o*, *e*, *b*, *c*, *o*, *r*, *h*, *i*, *l*, *m*, *n*, *q*, *m*, *n*, *p*>, *σ*4 = <*a*, *b*, *c*, *o*, *e*, *b*, *c*, *o*, *r*, *j*, *k*, *l*, *m*, *n*, *q*, *m*, *n*, *q*, *m*, *n*, *p*>, *σ*5 = <*a*, *b*, *c*, *o*, *e*, *b*, *c*, *o*, *r*, *j*, *k*, *l*, *m*, *n*, *q*, *m*, *n*, *q*, *m*, *n*, *p*>> is an event log. According to Algorithm 5, we can get *Aps* = {<ω1 = <*b*, *c*, *o*, *e*>1 ∧ <*h*, *i*>1, <*m*, *n*, *q*>1>, <ω2 = <*b*, *c*, *o*, *e*>1 ∧ <*g*, *k*>1, <*m*, *n*, *q*>2>}. Then the number of *Aps*(ω1) = 1, *Aps*(ω2) = 2, and |*L*| = 5, thus we can get *Extended-support*(*Mω*1) = 1/5 =0.2, *Extended-support*(*Mω*2) = 2/5 = 0.4.

*Definition 26* (*Extended**-confidence*) Let *L* ∈ ***B***(*A\**) be an event log over *A*, *Mω* = (X⇒Y) be a association rule, and *σ* ∈ *L* be a trace, where X = ƒ1*num*(*f* 1, *σ*) ∧ *γ num*(*γ*, *σ*), Y =ƒ2*num*(*f* 2, *σ*), *NX­*(*ω*) represents the number of X of association pair ω occurrences in L. *Aps*(*ω*) represents the number of association pair *ω* occurrences in *L*, the *Extended-confidence* is defined and calculated as follows.



*Extended-confidence* (*Mω*) =

*Extended-confidence* indicates the reliability of the association rules. If an association rule with a high *Extended –confidence*, represents that this association rule is very reliable, i.e., we get the sufficient condition that if X occurs, then Y occurs.

For instance，Let *L* = <*σ*1 = <*a*, *b*, *p*, *r*, *g*, *l*, *s*, *n*, *q*>, *σ*2 = <*a*, *b*, *t*, *p*, *e*, *b*, *t*, *p*, *r*, *h*, *i*, *s*, *n*, *q*>, *σ*3 = <*a*, *b*, *t*, *p*, *e*, *b*, *t*, *p*, *r*, *h*, *i*, *l*, *s*, *n*, *o*, *s*, *n*, *q*>, *σ*4 = <*a*, *b*, *t*, *p*, *e*, *b*, *t*, *p*, *r*, *h*, *i*, *l*, *s*, *n*, *o*, *s*, *n*, *o*,*s*, *n*, *q*>, *σ*5 = <*a*, *b*, *t*, *p*, *e*, *b*, *t*, *p*, *r*, *j*, *k*, *l*, *s*, *n*, *o*, *s*, *n*, *o*, *s*, *n*, *o*, *s*, *n*, *q*>, *σ*6 = <*a*, *b*, *t*, *p*, *e*, *b*, *t*, *p*, *r*, *j*, *k*, *l*, *s*, *n*, *o*, *s*, *n*, *o*, *s*, *n*, *o*, *s*, *n*, *q*>> is an event log. *Aps* = {<ω1 = <*b*, *t*, *p*, *e*>1 ∧ <*h*, *i*>1, <*s*, *n*, *o*>1>, <ω2 = <*b*, *t*, *p*, *e*>1 ∧ <*h*, *i*>1, <*s*, *n*, *o*>2>, <ω3 = <*b*, *t*, *p*, *e*>1 ∧ <*j*, *k*>1, <*s*, *n*, *o*>3>} can be obtained from Algorithm 5. Such as X1­­ = <*b*, *t*, *p*, *e*>1 ∧ <*h*, *i*>1, X2­ = <*b*, *t*, *p*, *e*>1 ∧ <*j*, *k*>1, then *Aps*(ω1) = 2, *Aps*(ω2) = 1, *Aps*(ω3) = 2, and = 3, = 2. Thus, *Extended-confidence*



(*Mω*1) = 2/3 = 0.67, *Extended-confidence*(*Mω*2) = 1/3 = 0.33, *Extended-confidence*(*Mω*3) = 2/2 = 1 can be obtained at last.

From the definition of extension support and extension confidence, if the association pair *ω* appears in a given log *L*, i.e., *Extended-support*(*Mω*) > 0, and the *Extended-confidence* in a given log is 1, i.e., *Extended-confidence*(*Mω*) = 1, we say that the association rule *Mω* is valid in a process model. Based on this conclusion, an algorithm to obtain eligible association rules from association pairs is proposed as follows.

|  |
| --- |
| **Algorithm 6**  Identify Association Rules |
| **Input:** A loop completeness log *L* ∈ *B*(*A*∗), *Aps*,  *Acs*, *Ale*  **Output:** *ΡR*, *R\_pre* , *R­\_pos* |
| 1. *ΡR* ← ∅, *R\_pre* ← ∅, *R­\_pos* ← ∅, *NX*­(*ω*) ← 0, *Aps*(*ω*) ← 0  2. **for** each ω = <ƒ1*num*(*f* 1, *σ*) ∧ *γ num*(*γ*, *σ*), ƒ2*num*(*f* 2, *σ*) > ∈ *Aps* **do**  3. **for** each *σ* ∈ *L* **do**  4. **if** ƒ1 ⊆ *σ* and *γ* ⊆ *σ* **then**  5. **for** each *a* ∈ *Ale* **do**  6. **if** *a* ∈ƒ1and*num*(*a*, *σ*) = *num*(*f* 1, *σ*)**then**  7.  **for** each *b* ∈ *Acs* **do**  8. **if** *b* ∈ *γ*and*num*(*b*, *σ*)= *num*(*γ*, *σ*)**then**  9. *NX*­(*ω*) ← *NX­*(*ω*) + 1;  10.  **if** ƒ2 ⊆ *σ* **then**  11. **for** each *c* ∈ *Ale* **do**  12.  **if** *c* ∈ƒ2and*num*(*c*, *σ*)= *num*(*f* 2, *σ*)**then**  13. *Aps*(*ω*) ← *Aps*(*ω*) + 1;  14. **end if**  15.  **end for**  16. **end if**  17.  **end if**  18. **end for**  19. **end if**  20.  **end for**  21.  **end if**  22.  **end for**  23.  *Extended-support*(*Mω*) = *Aps*(*ω*) / |*L*|;  24. *Extended-confidence* (*Mω*) = *Aps*(*ω*) / *NX*­(*ω*);  25. **if** *Extended-support*(*Mω*) > 0 and  *Extended-confidence* (*Mω*) = 1**then**  26. *ΡR* ← *ΡR*+ {(ƒ1*num*(*f* 1, *σ*) ∧ *γ num*(*γ*, *σ*))⇒ ƒ2*num*(*f* 2, *σ*)};  27. *R\_pre* ← *R\_pre* + (ƒ1*num*(*f* 1, *σ*) ∧ *γ num*(*γ*, *σ*));  28. *R­\_pos* ← *R­\_pos* + ƒ2*num*(*f* 2, *σ*);  29. **end if**  30. *NX*­(*ω*) ← 0, *Aps*(*ω*) ← 0;  31. **end for**  32. **return** *ΡR*, *R\_pre*, *R­\_pos* |

In Algorithm 6, step 1 initializes variable *ΡR*, *R\_pre*, *R­\_pos*, *NX*­(*ω*), and *Aps*(*ω*). Steps 2-9 indicate that the number of X of association pair *ω* occurrences in *L* is obtained, i.e., *NX*­(*ω*). Steps 10-13 indicate that the number of association pair *ω* occurrences in *L* is obtained, i.e., *Aps*(*ω*). Steps 23-24 indicate that the values of *Extended-support*(*Mω*) and *Extended-confidence*(*Mω*) for each association pair *ω* are obtained according to *Definition 25* and *Definition 26*. Steps 25-29 indicate that if *Extended-support*(*Mω*) > 0 and *Extended-confidence*(*Mω*) = 1 are satisfied, the set of association rules *ΡR*, the antecedent of association rules set *R\_pre*, and the consequent of association rules set *R­\_pos* are obtained. Step 32 returns *ΡR*, *R\_pre*, and *R­\_pos* at last.

Now, an example is given to illustrate the execution process of Algorithm 6 as follows.

Fig. 6. A process model with two loop structures and a choice structure

**Example 6:** Fig. 6 represents a process model. An event log can be obtained by executing this process model. *L* = <*σ*1 = <*a*, *p*, *q*, *t*, *f*, *k*, *s*, *m*, *o*>, *σ*2 = <*a*, *p*, *q*, *d*, *p*, *q*, *t*, *f*, *k*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*3 = <*a*, *p*, *q*, *d*, *p*, *q*, *t*, *f*, *k*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*4 = <*a*, *p*, *q*, *d*, *p*, *q*, *d*, *p*, *q*, *t*, *g*, *h*, *k*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*5 = <*a*, *p*, *q*, *d*, *p*, *q*, *d*, *p*, *q*, *t*, *g*, *h*, *k*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*6 = <*a*, *p*, *q*, *d*, *p*, *q*, *d*, *p*, *q*, *t*, *i*, *j*, *k*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*7 = <*a*, *p*, *q*, *d*, *p*, *q*, *d*, *p*, *q*, *t*, *i*, *j*, *k*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*8 = <*a*, *p*, *q*, *d*, *p*, *q*, *t*, *g*, *h*, *k*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*9 = <*a*, *p*, *q*, *d*, *p*, *q*, *t*, *g*, *h*, *k*, *s*, *m*, *r*, *s*, *m*, *r*, *s*, *m*, *o*>, *σ*10 = <*a*, *p*, *q*, *d*, *p*, *q*, *t*, *i*, *j*, *k*, *s*, *m*, *r*, *s*, *m*, *o*>>. We can get *Aps* = {<ω1 = <*p*, *q*, *d*>1 ∧ <*f*>1, <*s*, *m*, *r*>1>, <ω2 = <*p*, *q*, *d* >1 ∧ <*f*>1, <*s*, *m*, *r*>2>, <ω3 = <*p*, *q*, *d*>2 ∧ <*g*, *h*>1, <*s*, *m*, *r* >2>, <ω4 = <*p*, *q*, *d*>2 ∧ <*i*, *j*>1, <*s*, *m*, *r* >3>, <ω5 = <*p*, *q*, *d*>1 ∧ <*g*, *h*>1, <*s*, *m*, *r* >1>, <ω6 = <*p*, *q*, *d*>1 ∧ <*g*, *h*>1, <*s*, *m*, *r* >2>, <ω7 = <*p*, *q*, *d*>1 ∧ <*i*, *j*>1, <*s*, *m*, *r* >1>}. Such as X1­­ = <*p*, *q*, *d* >1 ∧ <*f*>1, X2­ = <*p*, *q*, *d* >2 ∧ <*g*, *h*>1, X3­ = <*p*, *q*, *d* >2 ∧ <*i*, *j*>1, X4­ = <*p*, *q*, *d* >1 ∧ <*g*, *h*>1, and X5­ = <*p*, *c*, *d* >1 ∧ <*i*, *j*>1, then the *Aps*(ω1) = 1, *Aps*(ω2) = 1, *Aps*(ω3) = 2, *Aps*(ω4) = 2, *Aps*(ω5) = 1, *Aps*(ω6) = 1, *Aps*(ω7) = 1, and = 2, = 2, = 2, = 2, = 1. Thus, *Extended-*



*support*(*Mω*1) = 1/10 = 0.1 > 0, *Extended-confidence*(*Mω*1) = 1/2 = 0.5; *Extended-support*(*Mω*2) = 1/10 = 0.1 > 0, *Extended-*

*confidence*(*Mω*2) = 1/2 = 0.5; *Extended-support*(*Mω3*) = 2/10 = 0.2 > 0, *Extended-confidence*(*Mω*3) = 2/2 = 1; *Extended-*

*support*(*Mω*4) = 2/10 = 0.2 > 0, *Extended-confidence*(*Mω*4) = 1/2 = 0.5; *Extended-support*(*Mω*5) = 1/10 = 0.1 > 0, *Extended-confidence*(*Mω*5) = 1/2 = 0.5; *Extended-support*(*Mω*6) = 1/10 = 0.1 > 0, *Extended-confidence*(*Mω*6) = 1/2 = 0.5; and *Extended-support*(*Mω*7) = 1/10 = 0.1 > 0, *Extended-confidence*

(*Mω*7) = 1/1 = 1 can be obtained. According to the Algorithm 6, *ΡR* = {<*p*, *q*, *d*>2 ∧ <*g*, *h*>1 ⇒ <*s*, *m*, *r*>2, <*p*, *q*, *d*>1 ∧ <*i*, *j*>1 ⇒<*s*, *m*, *r*>1}, *R\_pre* = {<*p*, *q*, *d*>2 ∧ <*g*, *h*>1, <*p*, *q*, *d*>1 ∧ <*i*, *j*>1}, and

*R­\_pos* ={<*s*, *m*, *r*>2, <*s*, *m*, *r*>1} can be obtained at last.

* 1. *Improved Petri Net*

As a formal representation method, a Petri net is often used to describe the process mining model. The definition of ordinary Petri net cannot directly express the indirect dependency in the process model. Thus, we improve the definition of Petri and propose new transition firing rules in this section.

*Definition 27* (*Improved Petri Net*): A seven-tuple *IPN* =(*P*, *T*; *F*, *M*0, *R\_pre*, *R­\_pos*, *ΡR*)is called improved Petri net, where

1. *N* = (*P*, *T*; *F*) is a net, where *P* is a finite set of places and *T* is a finite set of transitions, *P* ∪ *T* ≠ ∅and *P* ∩ *T* = ∅;
2. *F* ⊆(*P* *×* *T*) ∪ (*T* *×* *P*) is a set of directed arcs from places to transitions or from transactions to places;
3. *M*: *P* → *N* is called a marking, where *M*0 represents the initial marking, and for ∀*p* ∈ *P*, the number of tokens in *p* is represented by *M*(*p*);;
4. *cod*(*F*) ∪ *dom*(*F*) = *P* ∪ *T*, where *cod*(*F*) *=* {*x* ∈ *P* ∪ *T|*∃*y* ∈ *P* ∪*T*: (*y*, *x*) ∈ *F*}, and *dom*(*F*) *=* {*x* ∈ *P* ∪ *T|*∃*y* ∈ *P* ∪ *T*: (*x*, *y*) ∈ *F*};
5. *R\_pre* is a set of the antecedent of association rules X = ƒ*inum*(*fi*, *σ*) ∧ *γ num*(*γ*, *σ*);
6. *R­\_pos* is a set of the consequent of association rules Y = ƒ*jnum*(*fj*, *σ*); and
7. *ΡR* is a set of association rules.
8. Transition firing rules:
9. if ∀ƒ*in*∧ *γ*1 ⇒ ƒ*jm* ∈ *ΡR*, where ƒ*i*and ƒ*j*are loop sequences, *γ* is a choice sequence, and ƒ*in*∧ *γ*1 ∈ *R\_pre*, ƒ*jm* ∈ *R­\_ pos*. If ƒ*i* has been executed *n* times and *γ* has been executed at the same time, and *p* ∈ •*t*, where *t* ∈ ƒ*j­* and *t* ∈ *Als*: *M*(*p*) ≥ 1, then the loop sequence ƒ*j*­ is enabled in the mark *M*, and it is denoted as *M*[ƒ*j m >*, said it can be executed *m* times,; and
10. if ƒ*in*∧ *γ*1 ⇒ ƒ*jm* ∉ *ΡR*, the rules are consistent with *PN*.

Fig. 7. An *IPN* process model with indirect dependency

From example 6, we can know that Fig. 6 is a process model with indirect dependencies, i.e., this model can be represented by the *IPN* model as shown in Fig. 7. We can get *ΡR* = {<*b*, *c*, *d*>2 ∧ <*g*, *h*>1 ⇒ <*l*, *m*, *n* >2, <*b*, *c*, *d*>1 ∧ <*i*, *j*>1 ⇒<*l*, *m*, *n* >1}, *R\_pre* = {<*b*, *c*, *d*>2 ∧ <*g*, *h*>1, <*b*, *c*, *d*>1 ∧ <*i*, *j*>1}, and *R­\_pos* ={<*l*, *m*, *n* >2, <*l*, *m*, *n* >1}. For ƒ1 = <*b*, *c*, *d*>, *γ*1 = <*g*, *h*>, ƒ2 = <*l*, *m*, *n*>, if ƒ1 has been executed two times and *γ*1 has been executed at the same time, then *p*9 ∈ •*l* : *M*(*p*9) ≥ 1, then the loop sequence ƒ2 is enabled in the mark *M*, and it can be executed two times. Similarly, for ƒ1 = <*b*, *c*, *d*>, *γ*2 = <*i*, *j*>, ƒ2 = <*l*, *m*, *n*>, if ƒ1 has been executed once time and *γ*2 has been executed at the same time, then *p*9 ∈ •*l* : *M*(*p*9) ≥ 1, then the loop sequence ƒ2 is enabled in the mark *M*, and it can be executed once time.

* 1. *AlphaID Algorithm*

Now, all loop and choice branch driven loop structures can be obtained correctly. To propose an algorithm is necessary to construct this structure. Therefore, we propose an algorithm called *AlphaID* algorithm to mine this structure as follows.

*Definition 28* (*AlphaID algorithm*): Let *L* be an event log over *T*. *AlphaID* (*L*) is defined as follows:

1. *TL* = {*t* ∈ *T* | ∃*σ*∈*L**t* ∈ *σ*};
2. *TI* = {*t* ∈ *T* | ∃*σ*∈*L t* = *first*(*σ*)};
3. *To* = {*t* ∈ *T* | ∃*σ*∈*L t* = *last*(*σ*)};
4. *SLA* = {*a* ∈ *TL*| ∃*a* ∈ *σ* ∧ *num*(*a*, *σ*) > 1};
5. *Als* = {*ai*∈ *SLA* | *ai* = ƒ*i*(1)};
6. *Ale* = {*ai*∈ *SLA* | *ai* = ƒ*i*(*n*)}, *i* ∈ {1, 2, …, *n*};
7. *SLS* = {ƒ ∈ *SLA*∗| |ƒ| = 1 ∪ (ƒ[*i*] >*L* ƒ[*i* + 1] ∧ ƒ[*n*] >*L* ƒ[1], *i* ∈ {1, 2, …, *n*-1})};
8. *SCA* = {*a* ∈ *TL*| ∃*σ*1, *σ*2 ∈ *L* ∧ *a* ∈ *σ*1 ∧ *a* ∉ *σ*2 ∧ *a* ∉ *SLA*};
9. *Acs* = {*ai* ∈ *SCA*| ∃*b*∈*Ψ* (*bi*, *ai*)∈ *Adir\_set* ∧ *bi* ∉ *SCA*};
10. *Ace* = {*ai* ∈ *SCA*| ∃ *b*∈*Ψ* (*ai*, *bi*)∈ *Adir\_set* ∧ *bi* ∉ *SCA* ∧ *bi* ∉ *Acs*};
11. *SCS* = {*γ* ∈ *SCA*∗| |*γ*| = 1∪ (*γ* [*i*] >*L* *γ* [*i*+1], *i* ∈ {1, 2, …, *n*-1})}.
12. *X L* = {(*A*, *B*) | *A* ⊆ *TL* ∧ *A* ≠ ∅ ∧ *B* ⊆ *TL* ∧ *B* ≠ ∅ ∧ ∀ *a*∈*A* ∀*b*∈*B* *a* →*L* *b* ∧∀*a,**a*∈*A a* 1 #*L* *a* 2 ∧∀*b,**b*∈ *B* *b* 1 #*L* *b* 2};
13. *YL* = {(*A*, *B*)}∈ *X**L* |∀(*A′* , *B′* ) ∈ *X**A* ⊆ *A*′ ∧ *B* ⊆ *B*′ ⇒ (*A*, *B*) = (*A*′, *B*′)};
14. *PL* = {*p*(*A, B*) |(*A,* *B*) ∈ *YL* ∪ {*iL*, *oL*}};
15. *FL* = {(*a*, *p*(*A*,*B*)) |(*A*, *B*) ∈ *Y*L ∧ *a* ∈ *A* } ∪ {(*p*(*A*,*B*), *B*) |(*A, B*) ∈ *YL* ∧ *b* ∈ *B*} ∪ {(*iL*, *t*) |*t* ∈ *TI*} ∪ {(*io*, *t*) |*t* ∈ *To*};
16. *R\_pre* = {ƒ*inum*(*fi*, *σ*) ∧ *γ num*(*γ*, *σ*)| ƒ*i* ∈ *SLS*, *γ* ∈ *SCS*};
17. *R­\_pos* = {ƒ*j num*(*fj*, *σ*) | ƒ*j* ∈ *SLS*};
18. *ΡR* = {ƒ*i num*(*fi*, *σ*) ∧ *γ num*(*γ*, *σ*) ⇒ ƒ*j num*(*fj*, *σ*) | ƒ*i num*(*fi*, *σ*) ∧ *γ num*(*γ*, *σ*) ∈ *R\_pre* and ƒ*jnum*(*fj*, *σ*) ∈ *R\_pos*, and *Extended-support*(*Mω*) 0, *Extended-confidence*(*Mω*) = 1}; and
19. *AlphaID* (*L*) = (*PL*, *TL*, *FL*, *R\_pre*, *R­\_pos*, *ΡR*).

where, if *σ* is a trace, and *σ* = <*ta*, *tb*, *tc*, …, *tn*>, then *first*(*σ*) = *ta* and *last*(*σ*) = *tn*. *Adir\_set* represents a set of (*m*, *n*), where *Adir\_set* = {(*m*, *n*)| *m*, *n* ∈ *TL* ∧ *m* >*L­ n*}. *Mω* ∈ *ΡR* is an association rule.

The *AlphaID* algorithmis proposed based on *α-*algorithm. From the *AlphaID* algorithm, we can see that only steps 4-11 and 16-19 are different from *α-*algorithm. Next, let's analyze the effects of these steps. Step 4 gets all the loop activities in a log and puts them into a loop activity set *SLA*. Steps 5 and 6 obtain the start activity set *Als* and end activity set *Ale* of the loop structure, respectively. Step 7 aims to get the loop sequence set *SLS*. Step 8 is to get all the choice activities in a log and puts them in a choice activity set *SCA*. Steps 9 and 10 obtain the start activity set *Acs* and end activity set *Ace* of the choice structure, respectively. Step 11 aims to obtain the choice-branch sequence set *SCS*. Step 16 gets the antecedent set of association rules *R\_pre*. Step 17 obtains the consequent set of association rules *R\_pos*. Step 18 is to get the set of association rules *ΡR*. Step 19 shows that the *AlphaID* algorithm obtains the final *IPN* model.

Now, we come to a simple analysis of the complexity of the *AlphaID* algorithm. The complete logs for a complex process model may contain many traces. However, the complete log required by the *AlphaID* algorithm only needs to satisfy the traces that contain all direct and indirect follow relations. Thus, this greatly reduces the impact of the number of traces on the complexity of the *AlphaID* algorithm. In general, a qualified completeness log does not contain too many traces. According to [5], the complexity of the *AlphaID* algorithm depends on the number of tasks. Let |ω| = *i*, |*L*| = *j*, |*Ale*| = *k*, |*Acs*| = l, and |*TL*| = *n*. We assume that the size of completeness log *L* less than *n*, i.e., *j* *n.* Since Algorithm 6 contains five layers of loops, so the worst of *AlphaID* algorithms complexity is O(*n*5). We know that *i*, *k*, *l* *n*, so the actual complexity of *AlphaID* algorithm is O(*i j k lk*), and it is much lower than O(*n*5).

1. Simulation Experiments

In this section, we verify the effectiveness and correctness of the *AlphaID* algorithmin mining process models with indirect dependencies from two aspects of artificial experiment and real experiment respectively. By comparing with the existing algorithms, we find that the algorithm proposed in this paper has some advantages in dealing with the process model containing a loop-choice branch driven loop structure. The *AlphaID* algorithm is integrated into the open-source process mining framework Prom as a plug-in named *AlphaID*. The plug-in takes a complete event log as input, and the output is an *IPN* model. The software and hardware platform required by the plug-in is Intel (R) Core(TM) i5-6200U @ 3.60GHz CPU, 8.0GB RAM, JDK 1.8.11, and Windows 7\_64bit. This plug-in is available from https://github.com/shw476951164/AlphaID-algorithms-

event-logs.

* 1. *Artificial Experimental Analysis*

In this section, an artificial experiment is given to verify the effectiveness of the proposed algorithm in this paper. For a process model containing the loop-choice branch driven loop structure, we give the complete event logs L00-L07, and the specific information is shown in Table I. Now, we use the algorithms *AlphaID*, *Alpha++*, *ILP*, *HM* and *IM* to mine process models with the logs in Table I. At the same time, the process model obtained by each algorithm is analyzed from the aspects of correctness, fitness, and precision.

TABLE I

Artificial Model Completeness Log Properties

|  |  |  |  |
| --- | --- | --- | --- |
| log | trace | events | activities |
| L00 | 89 | 1880 | 23 |
| L01 | 222 | 5704 | 23 |
| L02 | 500 | 13653 | 23 |
| L03 | 900 | 25053 | 23 |
| L04 | 1900 | 53553 | 23 |
| L05 | 2700 | 76353 | 23 |
| L06 | 3600 | 102003 | 23 |
| L07 | 4600 | 130503 | 23 |

* + 1. *Correctness*

To verify the correctness of the proposed algorithm in mining process models with indirect dependencies, we take the log given in Table I as the input of the algorithm and use several different algorithms to obtain the process models. A process model shown in Fig. 8 is obtained by *AlphaID* algorithm. Fig. 9 is a model obtained by *Alpha++* algorithm and *ILP* algorithm. Fig. 10 and 11 show the process model obtained by *IM* and *HM*, respectively.

Fig. 8 shows an *IPN* model that can correctly reflect the log behavior shown in Table I. Association rules are clearly represented in the *IPN* model, which is used to mine the indirect dependencies of loop-choice branch driven loop structure. For instance, we can find <*f*, *g*, *h*>1 ∧ <*j*>1 ⇒<*p*, *q*, *r*, *s*>1 from the *IPN* model, it means that a loop sequence <*f*, *g*, *h*> is executed once, and a choice structure branch <*j*> is executed, then a loop sequence <*p*, *q*, *r*, *s*> can only be executed once. We can find that the process model mined by *Alpha*++ and *ILP* is the same, as shown in Fig. 9. The behavior of the logs in Table I can also be represented by this process model, however, it does not correctly reflect the indirect dependencies contained in the logs. Such as the process model can allow traces containing <*f*, *g*, *h*>1 ∧ <*j*>1 ⇒<*p*, *q*, *r*, *s*>2 to appear,but such traces are not likely to appear in the logs in Table I, thus the precision of the process model is not high. Fig. 10 is a process model obtained by *IM* algorithm,four places, and three invisible transitions are added to the process model [38], which makes the model more complex. Fig. 11 is a C- net, it is obtained by *HM* algorithm. This process model can be converted to *PN* model, which is the same as the alpha-+ + algorithm. And, more importantly, the process model also cannot correctly express the indirect dependencies of loop-choice branch driven loop structure.

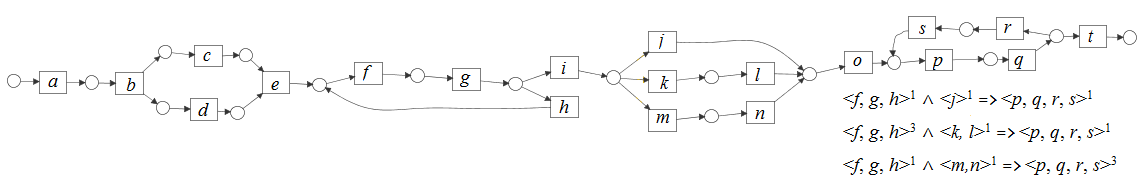


Fig. 8. An *IPN* process model mined by *AlphaID* algorithm from logs L00-L07.

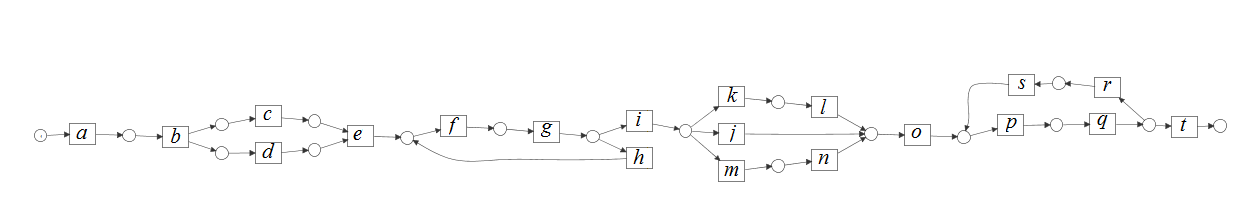


Fig. 9. A process model mined by *ILP* and *Alpha++* algorithms from logs L00-L07.

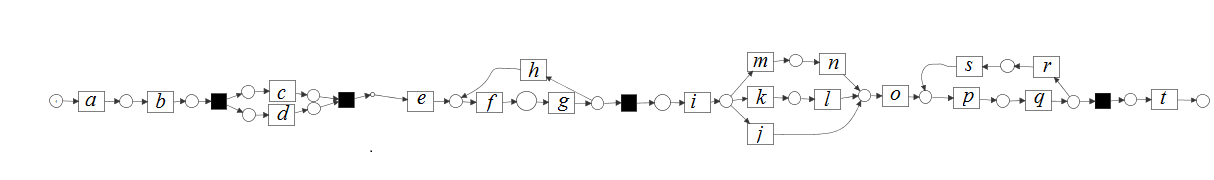
.

Fig. 10. A process model mined by *IM* algorithm from logs L00-L07.

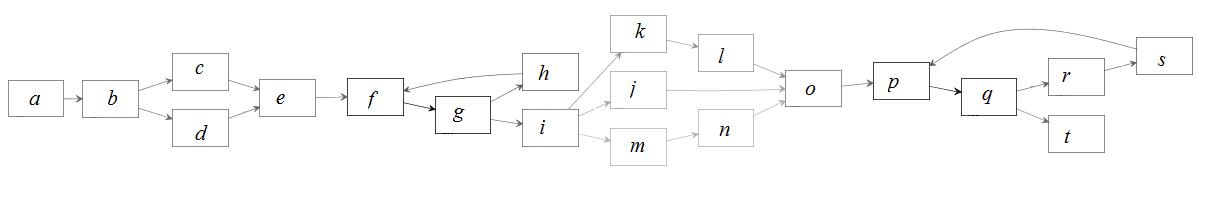


Fig. 11. A process model mined by *HM* algorithm from logs L00-L07.

As a result, the process model with a loop-choice branch driven loop structure can be correctly found by the *AlphaID* algorithm proposed in this paper, and can clearly represent the indirect dependencies can be clearly represented by the process model.

* + 1. *Fitness*

The process models obtained by mining algorithms should be able to represent the behavior of a given event log representation. Among them, the fitness is one of the evaluation indexes of the process model. The higher the fitness, the stronger the ability of the the event log to replay in the model.The “*Replay a Log on Petri Net for Conformance Analysis*”[40] plug-in in the Prom platform is used to obtain the fitness between the process models mined by different algorithms and the event logs given in Table I. The details of fitness values are shown in Fig. 12.

In Fig. 12, we can find that the process models obtained by different algorithms have a value of 1 with the event logs given in Tabel 1, which means that the logs L00-L07 can be replayed in-process models with corresponding algorithms.

Fig. 12. The values of fitness

* + 1. *Precision*

The behaviors represented by the process models obtained by different algorithms should not include behaviors other than the given event log behavior. In other words, precision represents the ability of the model to replay logs. For instance, we can find that there are indirect dependencies of loop-choice branch driven loop structure <*f*, *g*, *h*>1 ∧ <*m*, *n*>1 ⇒<*p*, *q*, *r*, *s*>3 in the process model in Fig. 8, it means that a loop sequence <*f*, *g*, *h*> is executed once, and a choice structure branch <*m*, *n*> is executed, then a loop sequence <*p*, *q*, *r*, *s*> can only be executed three times. However, in the process models obtained by other mining algorithms, such as *ILP*, *Alpha ++*, *HM*, *IM*, there is no limit to the number of loops of the loop structure. Then, we can find that, in Fig. 9-11, the process models allow to include traces that do not exist in the log, therefore their accuracy is low. The “*Check Precision based on Align-ETConformance* ” [41 - 42] plug-in in the Prom platform is used to obtain the precision between the process models mined by different algorithms and the event logs given in Table I. The details of precision values are shown in Fig. 13.

Fig. 13. The values of precision.

In Fig. 13, the process model obtained by the *AlphaID* algorithm has a high precision between the given event log. Association rules are added to the process model, which reduces the types of event log reaction behaviors that the process model can represent, then the process model can replay logs more accurately. The precision of the process model mined by *HM*, *ILP* and *Alpha*++ algorithms is consistent with given by logs, ranging from 0.86 to 0.90. The process model obtained by *IM* contains invisible transitions, which can replay logs more accurately. And the precision is higher than *ILP* algorithm, however, the precision is lower than *AlphaID* algorithm.

In short, the algorithm proposed in this paper can effectively obtain the process model with loop-choice branch driven loop structure, and can clearly express the indirect dependency on the process model, which is not available in other algorithms.

* 1. *Real Experimental Analysis*

In this section, we take the ball bearings production process as an example to demonstrate the effectiveness of the proposed algorithm *AlphaID*. The production process model of ball bearings contains the structure of the loop-choice branch driven loop. Eight sets of event logs L08-L15 are obtained from the actual ball bearings production system. For event logs, the name of the event is our primary concern, not the other attributes. The log details are shown in Table II. The production process of ball bearings mainly consists of the following parts: In the first part, we first make embryos for steel ball bearings according to different raw materials. Then we anneal the steel ball bearings and next to turn the ball bearings. In the second part, this part is a loop operation to continue the rough grinding, rough washing, rough soft grinding operations. Then the ball bearings are cut and cut. In the third part, it is a selection operation, according to the product quality requirements to choose heat treatment or cold treatment. Then we polish the ball bearings. The fourth part is a loop operation to continue the fine grinding,fine washing, fine soft grinding operation. The number of loop operations hinges on the number of ball bearings rough washing and the choice of cold and hot treatment.The last part is the detection grouping of the ball bearings, anti-rust treatment, then we pack the ball bearings, and finally store them.

Through the investigation of the actual production of the ball bearing, we find that only the following two conditions, the ball bearing passing rate is higher. Under the condition of the ball bearing the same raw materials: In the first case, if the loop sequence <*rough grinding*, *rough washing*, *rough soft grinding*> is executed twice, and the heat treatment branch is selected, then the loop sequence <*fine grinding*,*fine washing*, *fine soft grinding*> is executed twice; In the anther case, if the loop sequence <*rough grinding*, *rough washing*, *rough soft grinding*> is executed three times, and the cold treatment branch is selected, then the loop sequence <*fine grinding*,*fine washing*, *fine soft grinding*> is executed four times. These indicate indirect dependencies between them. Similar to the artificial experiment, we use *AlphaID*, *Alpha++*, *ILP*, *IM*, and *HM* to mine the eight groups of logs given in Table II. And, we analyze the process model mined by each algorithm from the perspective of correctness, fitness, and precision.

Fig. 13 is a process model mined by the *AlphaID* algorithm proposed in this paper. This model is different from other models in that it is an *IPN* model.we can see from this model, on the one hand, the model can correctly represent the process of ball-bearing production, on the other hand, it also finds two indirect dependencies of optimal production. More importantly, the optimal production process found is consistent with actual production experience. One of the optimal production processes is the loop sequence <*rough grinding*, *rough washing*, *rough soft grinding*> is executed twice, and the heat treatment branch is selected, then the loop sequence <*fine grinding*,*fine washing*, *fine soft grinding*> is executed twice; The other optimal production process in the loop sequence <*rough grinding*, *rough washing*, *rough soft grinding*> is executed three times, and the cold treatment branch is selected, then the loop sequence <*fine grinding*,*fine washing*, *fine soft grinding*> is executed four times. In other words, if the above two conditions are satisfied, the ball bearing production has the highest acceptance rate.The conditions for improving product quality can be obtained from the model, which is convenient for us to improve the production process of ball bearings.

TABLE II

Real Model Completeness Log Properties

|  |  |  |  |
| --- | --- | --- | --- |
| log | trace | events | activities |
| L08 | 143 | 3939 | 20 |
| L09 | 297 | 8181 | 20 |
| L10 | 484 | 13332 | 20 |
| L11 | 979 | 26967 | 20 |
| L12 | 1639 | 45147 | 20 |
| L13 | 2640 | 72720 | 20 |
| L14 | 3509 | 96657 | 20 |
| L15 | 4301 | 118473 | 20 |

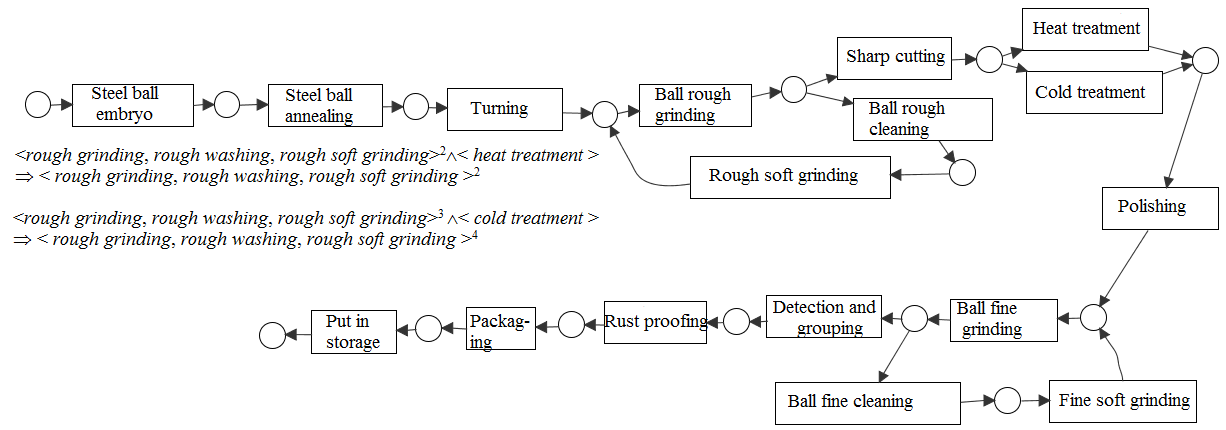
******

Fig. 13. An *IPN* process model mined by *AlphaID* algorithm from logs L08-L15.

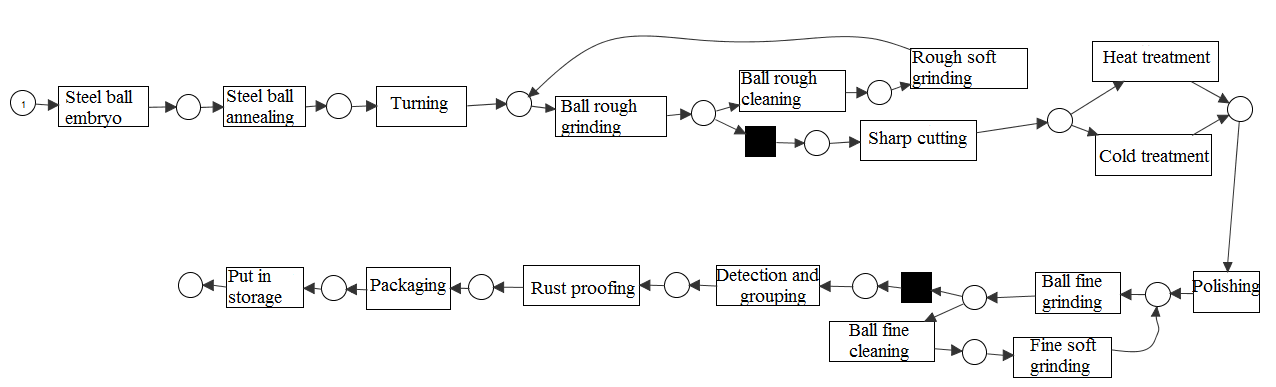


Fig. 14. A process model mined by *IM* algorithm from logs L08-L15.

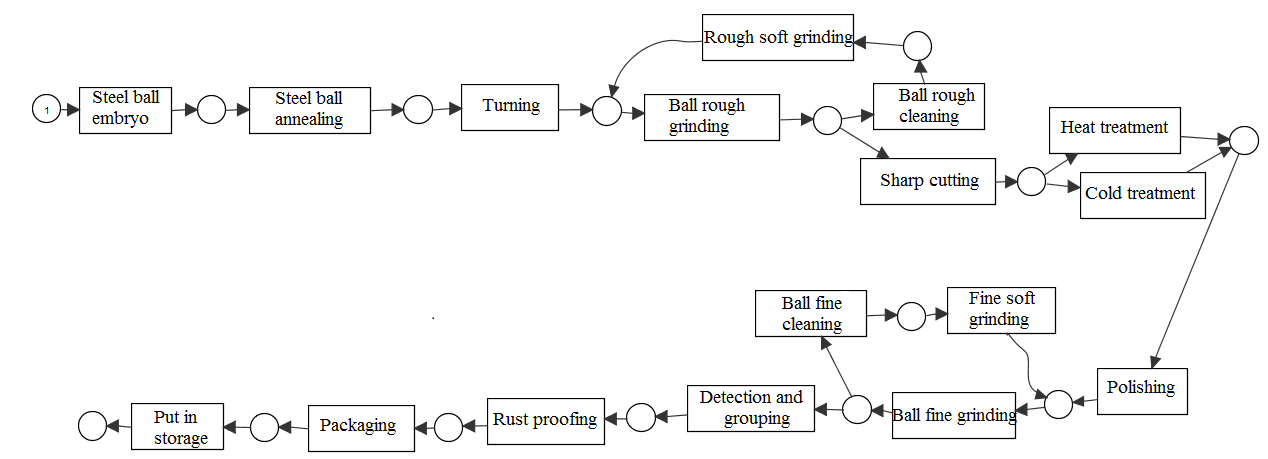
******

Fig. 15. A process model mined by ALPHA++*,* ILP*,* HMalgorithm from logs L08-L15.

In Fig. 14, the process model is obtained by the *IM* algorithm, which can correctly represent the ball bearing production process but cannot correctly express the indirect dependency of the structure of the loop-choice branch driven loop. Some implicit transitions exist in this process model, making its simplicity not high. In this process model, the loop sequence <*rough grinding*, *rough washing*, *rough soft grinding*> is executed twice, and the heat treatment branch is selected, then the loop sequence <*fine grinding*,*fine washing*, *fine soft grinding*> may be executed twice or five times. However, this situation is inconsistent with the actual production requirements, which should be avoided in the actual production process. The trace that does not exist in the process of actual production is allowed to appear in the process model, making its precision not high.

As shown in Fig. 15, the process models mined by *ILP*, *Alpha++*, and *HM* algorithms are consistent. We find that the structure of the loop-choice branch driven loop could not be correctly obtained from this process model. Similar to Fig. 14, this process model also allows impossible traces to occur in the actual production process, resulting in low precision.

Fig. 16. The values of fitness.

Fig. 17. The values of precision.

Similar to an artificial experiment, for the fitness and precision between the process model obtained by different algorithms and each set of event logs in Table II, we can also use the "*Replay a Log on Petri Net for Conformance Analysis*" and "*Check Precision based on Align-ETConformance*" plug-ins to obtain respectively. Fig. 16 and Fig. 17 respectively represent the details of fitness and precision. In Fig. 16, we find that the value of fitness between all models and logs is 1. This shows that the event logs L08-L15 can be replicated in all models. In Fig. 17, the precision between the process model obtained by the *AlphaID* algorithm proposed in this paper and logs is the highest compared with other algorithms.

In a word, the process model with the structure of the loop-choice branch driven loop can be found effectively by the algorithm proposed in this paper. And the process model can provide good advice for actual production.

1. Conclusion

we propose an algorithm for mining indirect dependencies in this paper. This algorithm is named *AlphaID*, and it can mine the process model that contains the structure of the loop-choice branch driven loop. In order to clearly describe the indirect dependencies between different structures, we introduce association rules and improve *PN* model. The *IPN* model is used to effectively represent the process model obtained by the algorithm. At the same time, we integrate the *AlphaID* algorithm into the ProM platform as a plug-in, At the same time, we integrated the *AlphaID* algorithm into the Prom platform as a plug-in, and the algorithm is fully implemented in ProM.In this paper, we focus on the indirect dependence of the number of loops of one loop structure and the different branches of choice structures on the number of loops of another loop structure.

In this paper, we verify the proposed algorithm with an artificial example and a real-life example. Both experiments show that the *AlphaID* algorithm is effective in mining the indirect dependence of the loop-choice branch driven loop structure. At the same time, compared with other algorithms, the fitness of the process model obtained by *AlphaID* algorithm is 1, and its precision is higher. Notice that the indirect dependencies found in the process model, they can effectively guide optimal production, but also help to solve the bottleneck problem in the real process model. However, the process model structure studied in this paper is relatively simple. In the future, we will study more complex process models and discover indirect dependencies between different structures, such as parallel structures.

References

[1] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance, and Enhancement of Business Processes*. Berlin, Germany: Springer, 2011, pp. 1–10.

[2] W. M. P. van der Aalst, T. Weijters, and L. Maruster, “Workflow mining: Discovering process models from event logs”, *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, Sep. 2004.

[3] A. K. A. de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters, “Workflow Mining: Current Status and Future Directions.” On The *Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences*, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003 Springer Berlin Heidelberg, 2003.

[4] A. K. A. de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters, “Process mining: Extending the α-algorithm to mine short loops,” Ph.D.dissertation, Dept.Technol.Manage, Eindhoven Univ. Technol, Eindhoven, The Netherlands, 2004.

[5] L. J. Wen, W. M. P. van der Aalst, J. M. Wang, and J. Sun, ‘‘Mining process models with non-free-choice constructs,” *Data Mining Knowl. Discovery*, vol. 15, no. 2, pp. 145–180, 2007.

[6] L. J. Wen, J. Wang, W. M. P. van der Aalst, B. Huang, and J. Sun, ‘‘Mining process models with prime invisible tasks,” *Data Knowl. Eng*., vol. 69, no. 10, pp. 999–1021, 2010.

[7] Q. Guo, L. Wen, J. Wang, Z. Yan, and P. S. Yu, “Mining invisible tasks in non-free-choice constructs,” in *Business Process Management* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2015, pp. 109–125.

[8] J. Li, D. Liu, and B. Yang, “Process mining: Extending α-algorithm to

mine duplicate tasks in process logs,” in *Advances in Web and Network* *Technologies*, *and Information Management*. Berlin, Germany: Springer, 2007, pp. 396–407.

[9] J. E. Cook and A. L. Wolf, ‘‘Discovering models of software processes from event-based data,’’ *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, 1998.

[10] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters, ‘‘Genetic process mining,’’ in *Applications and Theory of Petri Nets* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2005, pp. 48–69.

[11] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and

A. Serebrenik, ‘‘Process discovery using integer linear programming,’’

*Fundamenta Informaticae*, vol. 94, nos. 3–4, pp. 387–412, 2008.

[12] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, “Discov-

ering block-structured process models from event logs — A constructive approach,” in *Proc. Int. Conf. Appl. Theory Petri Nets Concurrency*. Berlin, Germany: Springer, 2013, pp. 311–329.

[13] C. W. Günther and W. M. P. van der Aalst, “Fuzzy mining Adaptive process simplification based on multi-perspective metrics,” presented at the 5th Int.Conf. Bus.Process Manage.,Brisbane, QLD, Australia, Sep.2007.

[14] J. Carmona, J. Cortadella, and M. Kishinevsky, ‘‘A region-based algorithm for discovering Petri nets from event logs,’’ in *Proc. Int. Conf. Bus. ProcessManage*. Berlin, Germany: Springer, 2008, pp. 358–373.

[15] R. Bergenthum, J. Desel, S. Mauser, and R. Lorenz, ‘‘Synthesis of Petri nets from term-based representations of infinite partial languages,’’ *Fundamenta Informaticae*, vol. 95, no. 1, pp. 187–217, 2009.

[16] W. Zheng, Y. Du, L. Qi, and L. Wang, ‘‘A method for repairing process models containing a choice with concurrency structure by using logic Petri nets,’’ *IEEE Access*, vol. 7, pp. 13106–13120, 2019.

[17] Y. Du, L. Qi, and M. Zhou, ‘‘Analysis and application of logical Petri nets to E-commerce systems,’’ *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 44, no. 4, pp. 468–481, Apr. 2014.

[18] T. Murata, “Petri nets: properties, analysis, and applications,” Proc. IEEE, vol. 77, no. 4, pp. 541–580, 1989.

[19] L. Qi, M. C. Zhou, and W. J. Luan, “A two-level traffic light control

strategy for preventing incident-based urban traffic congestion,” in *IEEE*

*Trans. Intelligent Transportation Systems*, vol. 19, no. 1, pp. 13–24,

2018.

[20] L. Qi, M. C. Zhou, and W. J. Luan, “Impact of driving behavior on traffic delay at a congested signalized intersection,” in *IEEE Trans. Intelligent Transportation Systems*, vol. 18, no. 7, pp. 1882–1893, 2017.

[21] N. Ran, H. Su, and S. Wang, ‘‘An improved approach to test diagnose-ability of bounded Petri nets,’’ *IEEE/CAA J. Autom. Sinica.*, vol. 4, no. 2, pp. 297–303, Apr. 2017.

[22] G. Liu, ‘‘Some complexity results for the soundness problem of workflow nets,’’ *IEEE Trans. Services Comput.*, vol. 7, no. 2, pp. 322–328, Apr./Jun. 2014.

[23] F. Yang, N. Q. Wu, Y. Qiao, and R. Su, ‘‘Polynomial approach to optimal one-wafer cyclic scheduling of treelike hybrid multi-cluster tools via Petri nets,’’ *IEEE/CAA J. Autom. Sinica*., vol. 5, no. 1, pp. 270–280, Jan. 2018.

[24] G. Liu, ‘‘Complexity of the deadlock problem for Petri nets modeling

resource allocation systems,’’ *Inf. Sci*., vol. 363, pp. 190–197, 2016.

[25] J. Lekić and D. Milićev, ‘‘Discovering block-structured parallel process models from causally complete event logs,’’ *J. Elect. Eng.*, vol. 67, no. 2, pp. 111–123, 2016.

[26] Y. Y. Wang and Y. Y. Du, ‘‘Conformance checking based on extended footprint matrix,’’ *J. Shandong Univ. Sci. Technol.*, Nature Sci., vol. 37, no. 2, pp. 9–15, 2018.

[27] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, “Discovering

block-structured process models from event logs: a constructive

approach,” Application and Theory of Petri Nets and Concurrency,

pp. 311–329, 2013.

[28] ShouGuang Wang, Dan You, MengChu Zhou, Carla Seatsu. “Characterization of Admissible Marking Sets in Petri Nets with Uncontrollable Transitions”, in *IEEE Trans. Automatic Control*. vol.61,no.7,pp.1953-1958.

[29] ShouGuang Wang, Dan You, MengChu Zhou. “A Necessary and Sufficient Condition for a Resource Subset to Generate a Strict Minimal Siphon in S4PR,”  *IEEE Trans. Automatic Control*. 2017, DOI: 10.1109/TAC.2017.2677859

[30] Lu Wang, Yuyue Du and Liang Qi, "Efficient Deviation Detection Between a Process Model and Event Logs," *IEEE/CAA J. Autom. Sinica*, 2019, 6(6):1352-1364.

[31] Ning Ran, Hongye Su, and Shouguang Wang, "An Improved Appro-

ach to Test Diagnosability of Bounded Petri Nets," *IEEE/CAA J.*

*Autom. Sinica*, 2017,4(2): 297-303.

[32] Xin Guo, Shouguang Wang, Dan You, Zhifu Li, Xiaoning Jiang:“A SiphonBased Deadlock Prevention Strategy for S3PR.” in *IEEE Access*7: 86

863-86873 (2019)

[33] JianChao Luo, ZhiQiang Liu, ShuoGang Wang, and KeYi Xing, "

Robust Deadlock Avoidance Policy for Automated Manufacturing

System with Multiple Unreliable Resources," *IEEE/CAA J. Autom.*

*Sinica*, 2020, In Press.

[34] W Liu, L Wang, X Feng, et al, “Soundness Analytics of Composed Logi-cal Workflow Nets,” J.*International Journal of Parallel Programming*, vol. 47, pp. 709-724, 2019.

[35] Y Y Du, L Qi, M C Zhou. A vector matching method for analysing

logic Petri nets[J]. *Enterprise Information Systems*, 2011, 5(4):

449-468.  
[36] W. Liu, Y. Y. Du, M. C. Zhou, and C. Yan. "Transformation of

 Logical Workflow Nets." *IEEE Transactions on Systems Man &*

*Cybernetics Systems* 44(10):1401-1412, 2014.

[37] Xiaoyu Lu, Mengchu Zhou, Ahmed Chiheb Ammari and Jingchu Ji, "

Hybrid Petri Nets for Modeling and Analysis of Microgrid Systems,"

*IEEE/CAA J. Autom. Sinica*, 2016, 3(4):349-356.

[38] Z. He, Y. Du, L. Wang, L. Qi, and H. Sun, “An alpha-FL algorithm for discovering free loop structures from incomplete event logs,” *IEEE Access.*, vol. 6, pp. 27885−27901, 2018

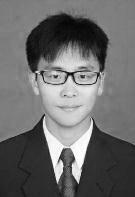
[39] H. Sun, Y. Du, L. Qi and Z. He, "A Method for Mining Process Models With Indirect Dependencies via Petri Nets," in IEEE Access, vol. 7, pp. 81211-81226, 2019.

[40] A. Rozinat and W.M.P. van der Aalst., ‘‘Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models,’’ in *Computer Science*, C. Bussler et al., editor, BPM 2005 Workshops, volume 3812 of Lecture Notes, pages 163-176. Springer- Verlag, Berlin, 2006.

[41] A. Adriansyah, J. Munoz-Gama, and J. Carmona, ‘‘Measuring precision of modeled behavior,’’ in *Information Systems and e-Business Management.*, 2014, 13(1):37-67.

[42] A. Adriansyah, J. Munoz-Gama, J. Carmona, B. F. van Dongen, and

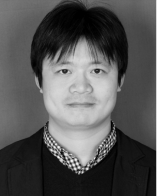
W. M. P. van der Aalst, ‘‘Measuring precision of modeled behavior,’’ *Inf.Syst. e-Bus. Manage.*, vol. 13, no. 1, pp. 37–67, Feb. 2015.

**HONGWEI SUN** received the B.S. and M.S. degrees from the school of Engineering, Qufu Normal University, Rizhao, China, in 2013 and 2016, respectively, where he is currently pursuing the Ph.D. degree in computer software and theory from Shandong University of Science and Technology, Qingdao, China. He is currently an Assistant with the College of Computer Science and Engineering, Weifang University of Science and Technology, Weifang, China. His current research interests are process mining, Petri nets, and workflow.

 **WEI LIU** received the B.S., M.S., and Ph.D. degrees from the Shandong University of Science and Technology, Qingdao, China, in 1999, 2005, and 2012, respectively. He is currently a Professor with the Shandong University of Science and Technology. His research interests are in formal engineering methods, Petri nets, workflows, and Web services. He has led or participated in many projects supported by the National Natural Science Foundation, the National Key Basic Research Development Program, and other important and key projects at provincial levels. He has authored over 30 papers in journals and conference proceedings, such as the IEEE TRANSACTIONS ON SYSTEMS MAN AND CYBERNETICS: SYSTEMS and *Enterprise Information Systems*. He was a recipient of the Science and Technology Advancement Awards at the province level and the Excellent Doctoral Dissertation Award from Shandong Province, China.

**LIANG QI** received the B.S. degree in information and computing science and the M.S. degree in computer software and theory from the Shandong University of Science and Technology, Qingdao, China, in 2009 and 2012, respectively, and the Ph.D. degree in computer software and theory from Tongji University, Shanghai, China, in 2017. From 2015 to 2017, he was a Visiting Student with the Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ, USA. He is currently with the Shandong University of Science and Technology. He has authored over 35 papers in journals and conference proceedings, including the IEEE TRANSACTIONS ON SYSTEM, MAN AND CYBERNETICS: SYSTEMS, the IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS, the IEEE/CAA JOURNAL OF AUTOMATICA SINICA, the IEEE TRANSACTIONS ON COMPUTATIONAL SOCIAL SYSTEMS, the IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING, and the IEEE TRANSACTIONS ON CYBERNETICS. His interests include Petri nets, machine learning, intelligent transportation systems, and optimization. He received the Best Student Paper Award-Finalist at the 15th IEEE International Conference on Networking, Sensing and Control (ICNSC’2018).

**YUYUE DU** received the B.S. degree from Shandong University, Jinan, China, in 1982, the M.S. degree from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1991, and the Ph.D. degree in computer application from Tongji University, Shanghai, China, in 2003. He is currently a Professor with the College of Information Science and Engineering, Shandong University of Science and Technology, Qingdao, China. He has taken in over ten projects supported by the National Natural Science Foundation, the National Key Basic Research Developing Program, and other important and key projects at provincial levels. He has published over 140 papers in domestic and international academic publications, and they are embodied over 80 times by SCI and EI and cited over 270 times by others. His research interests include formal engineering, Petri nets, real-time systems, Web services, and workflows. He is a member of the Professional Committee of Petri nets of the China Computer Federation.

**XIAOJUN REN**received the B.S. and M.S. degrees in Electronic Information Technology and Computer Software from Shandong University of Science and Technology, China, in 2007 and 2010, respectively; and his Ph.D. in Data Mining from Dongseo University, Korea, in 2016. Currently, he is a full-time lecturer at Weifang University of Science and Technology. His research interests include sensor networks, machine learning, database technology and statistics

1. (*Corresponding author: W. Liu*.)

   H. W. Sun is with the College of Information Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China, and he is currently an Assistant with the College of Computer Science and Engineering, Weifang University of Science and Technology, Weifang, 262700, China (e-mail: sunhongwei\_doctor.@yeah.net).

   W. Liu is with the College of Information Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China (e-mail: liuwei\_doctor.@yeah.net).

   L. Qi is with the College of Information Science and Engineering,

   Shandong University of Science and Technology, Qingdao 266590, China

   (e-mail: qiliangsdkd@163.com).

   Y. Y. Du is with the College of Information Science and Engineering,

   Shandong University of Science and Technology, Qingdao 266590, China

   (e-mail: yydu001@163.com) .

   X. J. Ren is with the College of Computer Science and Engineering, Weifang University of Science and Technology, Weifang, 262700, China

   (e-mail: renxiaojun\_05@yeah.net) [↑](#footnote-ref-1)