Bamboo Joint (BJ): A Long-Distance Dependency Mining Optimization in the Fodina Method

Supervisor: Dr.Artem Polyvyanyy, Student: Qingtan Shen (1130945)

October 31, 2022

Abstract

Process mining has been more and more popular with the development of artificial intelligence in recent years. Many automated process mining algorithms were published, but most of these algorithms do not consider the long-distance dependency relationship between different events. In this paper, a new long-distance dependency mining approach named Bamboo Joint (BJ) Approach will be introduced. This approach is a development based on the Fodina method, which belongs to the heuristic mining family. A model with better performance can be achieved by using this new approach when comparing to the Fodina method.

Keywords: process mining, process discovery, heuristic mining, long-distance dependency mining, event logs, Petri Nets

1 Introduction

1.1 Process Mining

With the increase in information system storage and computer calculation efficiency in these years, process mining techniques have already been applied in many large technical or consulting companies. Process mining can be treated as a field that combines traditional model-based process analysis and machine learning techniques together. It can be seen as a bridge between data mining, machine learning, and business process management. Process mining can be used to visually restore the actual business flow of an enterprise by collecting and cleaning event logs stored in various information systems. It can be used to help enterprise to find loopholes and bottlenecks when their event process is unusual. It also makes continuous monitoring become more efficiency.

Figure 1: The relationship between process mining, data science and process science [1]

The process mining concept was firstly published at the Eindhoven University of Technology in 1999, while event datasets were hard to collect at that period, and techniques related to process mining are not mature, so the performance of these techniques is unstable when applying them in the real world. However, lots of research achievements have been gained in the field of process mining during the last twenty years, and process mining is also divided into some smaller research areas, such as process discovery algorithm, conformance checking method in process mining, and mining additional perspectives, etc. Researchers and scientists with different backgrounds pay more attention to process mining recently, more and more concepts and algorithms were created to make this area become more mature.

1.2 Automated Process Discovery Algorithm

Automated process discovery is one of the most important parts of process mining operations, which takes an event log as its input and produces the output as a business process model that captures the control-flow relations between tasks implied by the event log [2]. For instance, here we have an event log, which is often XES standard. A process discovery algorithm is a function that maps this event log onto a process model. This process model can be used to represent the behavior in the event log. And our goal is to find an algorithm to map event logs automatically. Various automated process discovery algorithms have been posted in the last 20 years. These algorithms can be grouped by larger process mining families by their characters, such as the alpha-algorithm family, the inductive algorithm family, the heuristic algorithm family, etc.

Some measurements should also be made for our models generated by an automated process discovery algorithm. Firstly, each trace or the similar trace should be shown by the model generated from the algorithm. Secondly, the model could also show traces that are identical or similar to the traces of the process that produced the log, but these traces do not exist in the log. Thirdly, other traces except for the two conditions we just mentioned should not be generated from the model [essay1]. Professional names are given for these requirements to measure whether a model is qualified, which are precision, recall, and generalization. Also, a model generated from an algorithm should be as simple as possible, so here is the last measurement defined for model comparison, which is named simplicity.

Although the automated process discovery algorithm has developed for decades years, there are still two biggest problems we should meet when applying our automated algorithms to the real world as an industry or business environment. The first one is that models produced by automated algorithms may be extremely large, and it is hard to catch the key relationship between events in this spaghetti-like complex model. Another problem is that models produced by automated algorithms sometimes cannot fit the event log very well, which means many traces that exist in the original event log cannot be generated by the model we produced using the automated algorithm. Also, the model produced by automated algorithms can be over-generalized, which means many traces that exist in the model produced by algorithms do not appear in the original event log. A brilliant automated process discovery algorithm should overcome these difficulties.

1.3 Long Distance Dependency Relationship in Heuristic Mining

The heuristic mining algorithm is one of the most commonly used process mining methods to analyze event logs in the real world. Sometimes, heuristic mining algorithms are used to get directly follow graphs. We can also obtain a Petri Net or Business Process Model Net by using the split and join relationship which is an output in the second step of this algorithm. However, one shortage of heuristic mining algorithms is that the model produced by this method does not take into account indirect dependencies between events [3]. If the input behaviors in some parts are not free-choice, we cannot get a precise model. Long distance dependency relationship is one condition that can make more constraints between events, which can change input behaviors from free-choice to unfree-choice.

Long-distance dependencies are not represented in the dependency graph created at the first step of the heuristic mining algorithm. These relationships show the cases in which task A indirectly depends on task B, which means the choice for split or join events may depend on choices made in other parts of this process model [4].

In this paper, the main goal is to introduce the Bamboo Joint approach to mine long-distance dependency relationships between tasks. This BJ approach is based on an automated process discovery algorithm named the Fodina method [5], which belongs to the heuristic mining family. So, it can be

also considered as an optimization of the Fodina method, in order to get a better model by adding more actions in its long-distance dependency mining part.

The structure of this paper will be introduced briefly. Section 2 is the Motivating Example part, where an event log sample will be applied by using the original Fodina method and the BJ method after optimization. Section 3 is the Literature Review part, which will introduce some automated process discovery algorithms and different algorithm families, the development of the heuristic mining family will be mainly discussed there. Many basic concepts such as Event Log and Petri Net will be introduced in Preliminaries in section 4. The details of the long-distance dependency relationship mining approach will be introduced in section 5. The conformance-checking method and the experiment result for the BJ approach will be shown in section 6. Discussion and Conclusion of this approach will be shown in section 7 and section 8 respectively.

2 Motivation Example

In this part, an event log sample will be used to test the performance of the BJ approach. Firstly, a Petri Net model will be produced by the original Fodina algorithm. Then a new Petri Net will be produced by using the new algorithm after the optimization of the long-distance dependency mining part. There are three dimensions for these two Petri Net models that should be compared, which are precision, recall, and simplicity.

The event log sample comes from the Process Discovery Contest 2020, which has one thousand traces with tens of tasks. It can be seen as an event log from the industry environment, so it is suitable to be used as an input to test the performance of different algorithms. The form of this event log is XML Encryption Standard. This event log is put into two algorithms and produces these two Petri Net as the figures show.

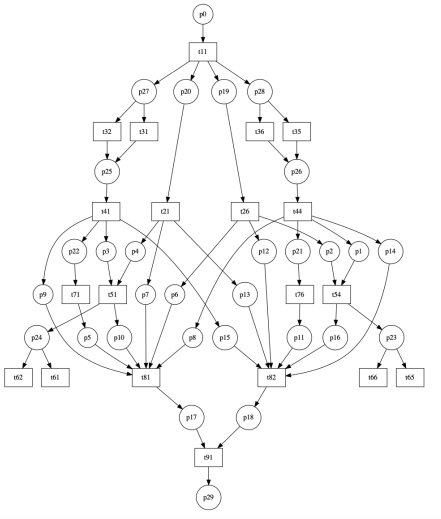

Figure 2: Petri Net I

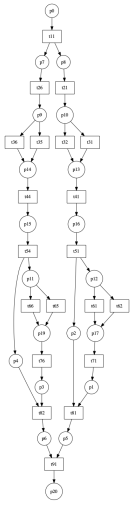

Figure 3: Petri Net II

Petri Net I is produced by the original Fodina algorithm. Petri Net II is produced by the BJ algorithm with an optimization approach. Precision of these two models will be compared at first. The precision of Petri Net I is 0.557. While the precision of Petri Net II is 0.812. Precision is defined as the ratio of the number of behaviors in both the Petri Net model and the event log example divided by the number of behaviors that only exist in the Petri Net model. Therefore, larger precision is considered as more behaviors of this Petri Net structure is similar to the behaviors in our training set, which means the model is much more suitable for this event log. Therefore, it can be clearly seen that the precision of Petri Net II is much larger than the precision of Petri Net I, which means Petri Net II is a better model than Petri Net I when considering the precision dimension.

The next dimension we will compare is named recall. The recall of Petri Net I is 0.995. While the recall of Petri Net II is 0.995. Recall is defined as the ratio of the number of behaviors in both the

Petri Net model and the event log example divided by the number of behaviors that only exist in the event log. Therefore, a larger recall is considered as more traces in this event log can be reproduced by the new Petri Net model, which also means the model has a better performance. In this example, the recall values of these two models are nearly the same, and they are really close to 1. Therefore, these two algorithms both have a good performance when considering the recall values of their models.

Similarity value is the last dimension we need to compare. It is also the most obvious dimension, which can be compared from these two Petri Nets directly. There are three different types of components in the Petri Net structure, which are place, transition, and arc. Similarity can be defined by two different measurement methods. It can be calculated as the sum of the place number and transition number in the Petri Net, or calculated as the sum of the place number and arc number, and transition number. A smaller similarity value means the model generated by the algorithm is much easier to understand and readable. Petri Net I has 20 transitions, 29 places, and 64 arcs. Petri Net II has 20 transitions, 20 places, and 46 arcs. If we use the second similarity approach, the similarity value for Petri Net I is 113, while the similarity value for Petri Net II is 86. When only considering model simplicity, Petri Net II is better than Petri Net I.

When comparing these comparisons together, both precision and simplicity dimensions have good performances when using the BJ approach. From these results, we can conclude that this new approach can delete some useless structures of the Petri Net model generated by the original Fodina algorithm, which can show a better performance on both precision and simplicity dimensions when generating models. The detail of this algorithm will be shown in section 5 later.

3 Literature Review

Many kinds of automated process discovery algorithms have been created in the past twenty years. In this section, the most basic and important algorithms will be introduced as a review of the process algorithm research area. Here is a table shows the development of automated process discovery algorithms in the period from 2012 to 2017.

Method	Main study	Year	Related studies	Model type	Model language	Semantic Constructs				Implementation		Evaluation		
						AND	XOR	OR	Loop	Framework	Accessible	Real-life	Synth.	Art.
HK	Huang and Kumar [25]	2012		Procedural	Petri nets	✓	✓		✓	Standalone		✓	✓	
Declare Miner	Maggi et al. [26]	2012	[27], [28], [29], [30], [31], [32]	Declarative	Declare					ProM	✓	✓	✓	
MINERful	Di Ciccio, Mecella [33]	2013	[34], [35], [36], [37]	Declarative	Declare					ProM, Standalone	✓	✓	✓	
Inductive Miner - Infrequent	Leemans et al. [38]	2013	[39], [40], [41], [42], [43], [44], [45]	Procedural	Process trees	✓	✓	✓	✓	ProM	✓	✓		
Data-aware Declare Miner	Maggi et al. [46]	2013		Declarative	Declare					ProM	✓	✓		
Process Skeletonization	Abe, Kudo [47]	2014	[48]	Procedural	Directly-follows graphs		✓		✓	Standalone		✓		
Evolutionary Declare Miner	vanden Broucke et al. [49]	2014		Declarative	Declare					Standalone			✓	
Evolutionary Tree Miner	Buijs e al. [16]	2014	[50], [51], [52], [53], [54]	Procedural	Process trees	✓	✓	✓	✓	ProM	✓	✓		✓
Aim	Carmona, Cortadella [55]	2014		Procedural	Petri nets	✓	✓		✓	Standalone		✓		✓
WoMan	Ferilli [56]	2014	[57], [58], [59], [60], [61]	Declarative	WoMan					Standalone		✓		✓
Hybrid Miner	Maggi et al. [62]	2014		Hybrid	Declare + Petri nets					ProM	✓	✓		
Competition Miner	Redlich et al. [63]	2014	[64], [65], [66]	Procedural	BPMN	✓	✓		✓	Standalone		✓		✓
Directed Acyclic Graphs	Vasilecas et al. [67]	2014		Procedural	Directed acyclic graphs		✓			Standalone		✓	✓	
Fusion Miner	De Smedt et al. [68]	2015		Hybrid	Declare + Petri nets					ProM	✓	✓	✓	✓
CNMining	Greco et al. [69]	2015	[70]	Procedural	Causal nets	✓	✓		✓	ProM	✓	✓	✓	
alpha$	Guo et al. [12]	2015		Procedural	Petri nets	✓	✓		✓	ProM	✓	✓		✓
Maximal Pattern Mining	Liesaputra et al. [71]	2015		Procedural	Causal nets	✓	✓		✓	ProM		✓		✓
DGEM	Molka et al. [72]	2015		Procedural	BPMN	✓	✓		✓	Standalone		✓		✓
ProDiGen	Vazquez et al. [73]	2015	[74]	Procedural	Causal nets	✓	✓		✓	ProM			✓	✓
Non-Atomic Declare Miner	Bernardi et al. [75]	2016	[76]	Declarative	Declare					ProM	✓	✓	✓	
RegPFA	Breuker et al. [77]	2016	[78]	Procedural	Petri nets	✓	✓		✓	Standalone	✓	✓	✓	
BPMN Miner	Conforti et al. [79]	2016	[80]	Procedural	BPMN	✓	✓	✓	✓	Apromore, Standalone	✓	✓		✓
CSMMiner	van Eck et al. [81]	2016	[82]	Procedural	State machines	✓	✓		✓	ProM	✓	✓		
TAU miner	Li et al. [83]	2016		Procedural	Petri nets	✓	✓		✓	ProM		✓		✓
PGminer	Mokhov et al. [84]	2016		Procedural	Partial order graphs	✓	✓			Standalone, Workcraft	✓	✓		✓
SQLMiner	Schönig et al. [85]	2016	[86]	Declarative	Declare					Standalone	✓	✓		
ProM-D	Song et al. [87]	2016		Procedural	Petri nets	✓	✓		✓	Standalone		✓	✓	
CoMiner	Tapia-Flores et al. [88]	2016		Procedural	Petri nets	✓	✓		✓	ProM				✓
Proximity Miner	Yahya et al. [89]	2016	[90]	Procedural	Causal nets	✓	✓		✓	ProM	✓	✓		
Heuristics Miner	Augusto et al. [19]	2017	[20], [21], [22], [91]	Procedural	BPMN	✓	✓		✓	Apromore, Standalone	✓	✓	✓	✓
Split miner	Augusto et al. [92]	2017		Procedural	BPMN	✓	✓		✓	Apromore, Standalone	✓	✓		
Fodina	vanden Broucke et al. [93]	2017	[94]	Procedural	BPMN	✓	✓		✓	ProM	✓	✓	✓	
Stage miner	Nguyen et al. [95]	2017		Procedural	Causal nets	✓	✓		✓	Apromore, Standalone	✓	✓		
Decomposed Process Miner	Verbeek, van der Aalst [96]	2017	[97], [98], [99], [100], [101], [102], [103]	Procedural	Petri nets	✓	✓	✓	✓	ProM	✓	✓		✓
HybridILPMiner	van Zelst et al. [24]	2017		Procedural	Petri nets	✓	✓		✓	ProM	✓			✓

Figure 4: Process discovery algorithms created from 2012 to 2017 [6]

3.1 Types of Automated Process Discovery Algorithm

The alpha algorithm [7] is one of the most basic process discovery algorithms. Log events can be scanned for their special relationship. Three kinds of relationships can be determined using the alpha algorithm, which are following, parallel, and unrelated. A table can be drawn to show the relationship between different events, and a Petri Net will be produced as an output of this algorithm. Although the concurrency problem can be solved by the alpha algorithm, it has difficulties when dealing with more complex problems, for instance, these problems may have much noise, infrequent behavior, long-distance dependency relationship, and complex routing constructs.

Genetic process mining [8] is another kind of process discovery algorithm. Unlike deterministic manner as the alpha-algorithm, genetic mining algorithm belongs to the evolutionary approaches. It uses iterative operations and randomization to get new alternatives. Four main steps should be

followed in this algorithm, which are initialization, selection, reproduction, and termination. The genetic mining algorithm is suitable to generate models for noisy or incomplete event logs because of its flexibility and robustness. However, more costs will be taken when applying this algorithm to larger models and event logs.

The inductive mining algorithm [9] is one of the leading process discovery algorithms because of its scalability and formal guarantees. The divide-and-conquer method is the main idea for this algorithm. Usually, process tree structures are preferred to show the output of this algorithm. The input event log is split into smaller logs iteratively by using an inductive mining algorithm. The split node should be found in the directly-follow graph. Four kinds of split operations can be operated, which are exclusive-choice cut, sequence cut, parallel cut, and redo-loop cut. Some conditions cannot be handled by using inductive mining, such as activities with fixed-length repetitions, or some silent or duplicate activities in event logs, which are shortcomings of this algorithm.

3.2 Heuristic Mining Algorithm Development

Heuristic mining algorithms [10] are another important part of automated process discovery algorithms. For the basic heuristic algorithm, event frequencies are considered when creating a process model. It should be promised that infrequent paths cannot be contained in the model, so robustness is the biggest advantage for heuristic mining algorithms when comparing to other methods. Usually, a causal net is used to show the output of this algorithm. The first step for heuristic mining is to create a dependency graph, which shows the dependency relationship of tasks that meet a certain threshold. The next step is to define the relation of splits and joins between the nodes which are connected in the dependency graph, and finally we get a causal net as our result.

The flexible heuristic mining (FHM) algorithm [4] is an updated version of the basic heuristic mining algorithm. We can choose to define the condition that all the tasks that appeared should be connected, which means each non-initial activity should have at least one activity as its cause and each non-final activity should have at least one dependent activity. Also, the length-two loop condition and long-distance dependency condition are also considered in this FHM algorithm. However, the long-distance dependency method is incomplete in the FHM algorithm, only uncomplex long-distance dependencies can be mined.

Fodina method [5] is also an updated version of the heuristic mining algorithm, which was mentioned in an essay published in 2017. Two steps should be followed when using this algorithm. In the first step, the dependency graph will be constructed. During this period, many restrictions can be chosen to consider or not. For example, the dependency graph can be constructed with no binary conflicts, and we can also choose to mine long-distance dependency when building the graph. Length-two loop mining is also included in this step. In the second step, although the basic idea of split and join operation is almost the same as the FHM algorithm, the Fodina method uses more steps to detect whether two tasks have long-distance dependency relationship. Therefore, the Fodina method has more probabilities to show a better performance than the FHM algorithm when mining activities with long-distance dependency relationship between tasks.

3.3 Motivation for Researching on Long Distance Dependency in the Fodina Method

More thresholds are defined in the Fodina method than in other heuristic mining algorithms, and it is beneficial for us to change them to consider different model construction restrictions when doing process mining. It is the main reason to choose the Fodina algorithm as our basic method. Then we found that a problem exists in the logic of this method: the Fodina method uses the same weight on normal dependency relationships and long-distance dependency relationships in the dependency map during the filtering procedure, which will cause all the long-distance dependency relationships shown in the model because of the larger threshold set for long-distance dependency relationships, while some useful normal dependency relationships will be deleted because of the lower frequencies. To solve this problem, the first consideration is to add different weights to normal dependency relationships and long-distance dependency relationships. But the weights calculated in process mining problems are more difficult than calculating in machine learning problems because we can only do conformance checking for a single model and there are no mathematical formulas to calculate these weights. What's more, different tasks in the event log may have different weights so the number of weights can be

extremely large. Therefore, adding many different weights may not be a suitable and efficient method to solve this problem.

Here is another view for solving this problem. The long-distance dependency map used in the Fodina method will be focused on when creating an optimization approach. Then the main idea to solve this problem is changed from adding different weights to checking each group of long-distance dependency relationships. When relationships of tasks contain in one group are necessary, then the structure of relationships in this group will be kept in the generated model. Otherwise, its structure will be deleted from the model. This new approach has the same effect as adding different weights to different relationships. More importantly, the time complexity of this new approach is only $O(n)$. This new approach will be shown more in detail in section 5 and section 6.

4 Preliminaries

Some definitions in process mining such as event log, Petri Net, and conformance checking are shown in this section.

4.1 Event Logs

An event log is a mathematical model which contains historic information about the executions of dynamic systems [11]. An event log consists of a multiset of traces. The whole information in each trace can be considered as a series of operations or an execution of some systems. Each trace is composed of a number of events, which can be treated as the occurrence of the activity or just one operation exists in a business process.

Here is an example to show the relationship between event log, trace, and activity. An event log is defined as L, two different traces are defined as $T1$ and $T2$ both belonging to the event log L. Then $T1$ has four activities named as a, b, c, e. $T2$ has three activities named a, c, and e. Then we can use these formulas to show their relationship. $T1 \in L$,$T2 \in L$, these two formulas show the relationship between the trace and the event log. Also, $a, b, c, e \in T1$; $a, c, e \in T2$, these two formulas show the relationship between the single activity and the trace. Trace $T1$ can be shown in this format $T1 =< a, b, c, e >$, also trace $T2$ can be shown in this format $T2 =< a, c, e >$. The event log L can be shown in this format $L = [T1, T2] = [< a, b, c, e >, < a, c, e >]$.

The order of activities in each trace contains temporal relations between these tasks. For one trace which contains events m_i and n_j, the position of m_i in this trace is i, while the position of n_j in this trace is j. If we observe that $i < j$, then it can be concluded that event m_i is observed before n_j. Take $T1$ as an example, the position of activity b in $T1$ is 2, the position of activity c in $T1$ is 3, and $2 < 3$, so activity b is observed before activity c in trace $T1$.

An event log also can capture the fact when some series of business operations are observed and recorded several times. $L1 = [< a, b, c, e >^5, < a, c, e >^{10}]$ can be taken as an example, $< a, b, c, d >^5$ means five traces are the same in $L1$ and they are $< a, b, c, e >$, Also $< a, c, e >^{10}$ means ten traces are the same in $L1$ and they are $< a, c, e >$. So there are $5 + 10 = 15$ traces in $L1$, and there are $4 * 5 + 3 * 10 = 50$ activities in $L1$.

4.2 Petri Net

Petri Nets [12] are the oldest process modeling language, and they have the best performance when modeling concurrency conditions. Although many other modeling languages also have the ability to model complex processes, such as the Business Process Modelling Notation (BPMN) and the Engineering Procurement Construction (EPC), most of the conformance checking methods are defined using the Petri Net. So, the Petri Net is chosen as the output type for our algorithm comparison.

Petri Net is defined as $N := (P, T, F)$, which is a triple set [13]. T represents a set of all the transitions, which can also be distinguished into two different parts, observable transitions, and silent transitions. Observable transitions have reasonable meanings, while silent transitions have no domain interpretation. P represents a set of places; each place means a possible state of the process and transitions are connected by places in the Petri Net. Both P and T are sets with finite disjoints. and if one element exists in p, then this element cannot exist in T, and vice versa. F is the set of flow relations, which can be shown as $F \subseteq (P * T) \cup (T * P)$. It means the elements in F must be chosen

from the connections between P sets and T sets. A Petri Net example will be used to show these definitions more clearly.

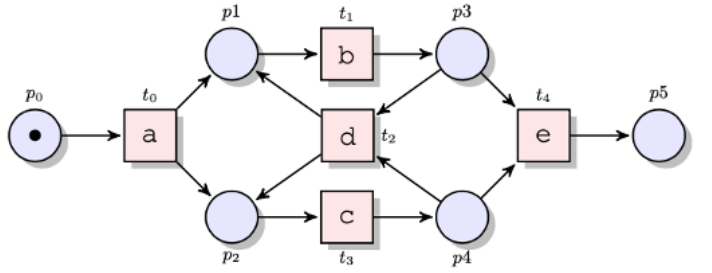

Figure 5: The Petri Net example [14]

In this Petri Net, it is easy to see that there are six places, so the P set can be displayed as $\{p_0, p1, p2, p3, p4, p5\}$. Also, there are five transitions exist, so the T set is equal to $\{a, b, c, d, e\}$. Fourteen connections between places and transitions are shown in this net. Therefore, our flow relation F is equal to $\{(p_0, a)(a, p1)(a, p2)(p1, b)(p2, c)(b, p3)(c, p4)(p3, d)(p4, d)(d, p1)(d, p2), (p3, e), (p4, e), (e, p5)\}$.

After describing the definition of Petri Net, there are also four gateways in Petri Net that should be mentioned, which are AND-split, XOR-split, And-join, and XOR-join. The same example will be used to clarify the differences between these four gateways. AND-split gateway can be applied to activate several places simultaneously after coming out of a transition. For example, $(a, p1), (a, p2)$ is an AND-split gateway, which means $p1$ and $p2$ are activated at the same time after going through the transition a. XOR-split gateway can be used if only one route can be chosen from several routes after coming out of a place. In this example, $(p3, d), (p3, e)$ consists of an XOR-split gateway, which means only one transition, transition d or transition e can be activated after going through the place $p3$, and it is impossible to activate them simultaneously.

AND-join gateway can be applied on several active places, which work together to activate one single transition. In this example, $(p3, e)$ and $(p4, e)$ consists of an AND-join gateway, which means the only condition for activating transition e is that both $p3$ and $p4$ should be active. The condition that only $p3$ or $p4$ is active cannot activate transition e. XOR-join gateway can be applied on several transitions, either of these transitions can activate a place. In this example, $(a, p1), (d, p1)$ consists of an XOR-join gateway, which means in the condition that either transition a or transition d is active can activate $p1$.

4.3 Causal Net

Causal net is another kind of modeling usually used in heuristic mining families, including the Fodina algorithm. It's also the reason why it should be discussed in this part. A causal net is defined as a tuple (T, I, O). T means a finite set of tasks. I is the input pattern function which stores all the input nodes' information for every task in the T set. O is the output pattern function which stores all the output nodes' information for every task in T set. Here is an example:

I(input)	T(activities)	O(output)
$\{\}$	a	$\{\{b\}, \{c\}, \{b, c\}, \{c, d\}, \{b, c, d\}\}$
$\{\{a\}\}$	b	$\{\{e\}\}$
$\{\{a\}\}$	c	$\{\{e\}\}$
$\{\{a\}\}$	d	$\{\{e\}\}$
$\{\{b\}, \{c\}, \{b, c\}, \{c, d\}, \{b, c, d\}\}$	e	$\{\}$

Table 1: Input and output nodes table

Here is a causal net, a table can be used to show the definition of this net. This table contains

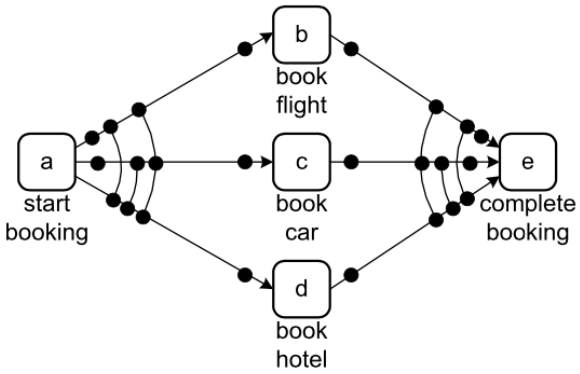

Figure 6: The causal net example [13]

the same information which contains in the tuple we define for the causal net. The second column in this table represents T in the tuple. The first column shows all the input node combinations for each activity, which has the same meaning as I in the tuple. The third column shows all the output node combinations for each task, which has the same meaning as O in the tuple. Take activity a as an example, the input for activity a is nothing, while the output for activity a has five different conditions, which are only b, only c, both b and c, both c and d, and all the b, c, and d.

4.4 Dependency Graph

A dependency graph is also a modeling method used in heuristic mining families, including the Fodina method. So brief introduction to this modeling method is required. We already know that (T, I, O) represents a causal net, then dependency graph (DG) is a relation on T, it can be written as $DG \subseteq T*T$, with $DG = \{(a, b) \mid (a \in T \wedge b \in a\square) \vee (b \in T \wedge a \in \square b)\}$ [FHM]. $a\square$ means the set which contains all the single tasks as outputs of activity a. $\square b$ means the set which contains all the single tasks as inputs of activity b. The main difference between one dependency graph and one causal net is that a dependency graph just shows the connection information between activities. It does not show the split and join information on each task as a causal net. Here is an example:

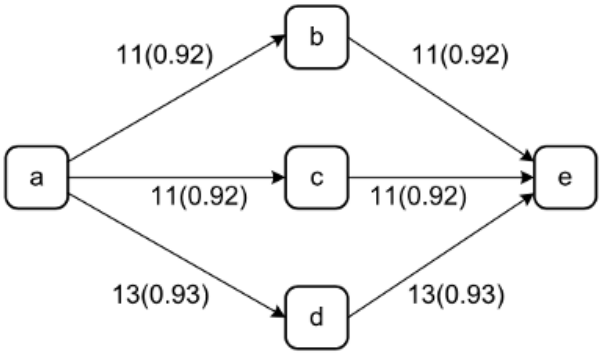

Figure 7: The dependency graph example [15]

This example is a dependency graph changed from the example of the causal net in section 4.3. The graph is much easier and our dependency graph should only show the connection information of these tasks, which can be expressed as $DG = \{(a, b), (a, c), (a, d), (b, e), (c, e), (d, e)\}$.

4.5 Heuristic Dependency-based Process Discovery

The heuristic dependency-based algorithm is a large family, which includes heuristic mining, flexible heuristic mining, Fodina mining, etc. Four main steps should be followed sequentially when applying these algorithms. Also, these four steps can be treated as the basic logic to follow when designing new heuristic mining methods. Therefore, it's worth showing more details here.

The first step for heuristic mining is to count the number of basic relations between activities. Three kinds of relationships between different activities are defined in this step. $|a > b|$ means the number of times activity a is followed directly by activity b. $|a >> b|$ means the number of repetition count from activity a to activity b, then followed by activity a again. $|a >>> b|$ shows the number of times activity a is followed indirectly by activity b. After getting these counting numbers, we can move to the next step.

The second step is to construct a dependency graph using the counting numbers collected in step I. The main goal is to determine whether any two activities have a relationship directly, which can be presented as $|a > b|$ or $|b > a|$ for activities a and b. A dependency graph can be drawn after finishing this step.

The third step is to collect the semantic information. It can be seen as adding input and output details for each activity to convert the dependency graph generated in step II to a causal net. Different gateways (XOR-join, AND-join, XOR-split, AND-split) should be chosen carefully when building the causal net.

The last step is the long-distance dependency mining, which is an optional step. $|a >>> b|$ is often used to determine whether activities a and b have a long-distance dependency relationship. However, values for $|a >>> b|$ sometimes are relatively large based on the event log size, so more restrictions are set in this determination. More details will be discussed in section 5 later.

4.6 Conformance Checking

Process discovery algorithms take an event log as their input and wish to output a process model that satisfies these four quality dimensions of process mining, which are precision, recall, simplicity, and generalization [16].

Precision is defined to measure the proportion of behaviors in the process model which can also be discovered in the event log. A low precision value implies an underfitting problem, which means many unreasonable behaviors exist in the process model. Deleting useless structures in the model might be a good way to improve precision value.

Recall is defined to measure the proportion of behaviors in the event log which can also be discovered from the process model. A low recall value implies that many behaviors in the event log cannot be shown by the process model. Both precision and recall values are between zero and one, which can be calculated using an event log and a process model as input. Although many different methods exist when calculating precision and recall, almost all scientists agree on the definitions of precision and recall.

Unlike recall and precision definitions, different viewpoints of generalization definition exist because generalization needs to deduce the behaviors not shown in the event log and establish the relationship between these behaviors and the process model. The definition for generalization is abstract, so we only show its traditional definition: A generalization measure [17] aims to quantify the probability that new unseen cases will fit the model. A low generalization value means the model might be overfitting.

Simplicity is used to measure the complexity of a model. If two models have the same performance on precision and recall dimensions, then the simpler model is better. Simplicity is calculated using the sum of nodes and arcs in a model. Then the model is simpler if the simplicity value for this model is smaller.

In this paper, the Entropia method will be used to calculate precision and recall for our models, which will be introduced in more detail in section 6. For the Petri Net, simplicity is defined in two different ways, one way is calculated as the sum of places and transitions in the Petri Net, and another way is defined as the sum of places, transitions, and arcs in the Petri Net. Both measurements will be used to compare models in this paper. The generalization dimension is ignored for our model comparison.

5 Approach

In this section, details of the original Fodina method and the BJ approach are described. All the code of the approaches described in this section can be viewed by using this link: https://github.com/QINGTANS/Bamboo-Joint-Approach, the README file in this web page shows the basic information for the python files there.

5.1 Original Fodina method

Although the Fodina method belongs to the heuristic mining family, its basic idea is a little different from the four basic procedures of heuristic dependency-based mining we introduced in section 4.5. The first procedure of Fodina mining is to count the number of basic relations, this procedure is also combined with dependency graph construction. And the third procedure is semantic information connection, which starts after those two operations. What's more, long-distance dependency mining is not operated at the end of this algorithm. This mining behavior continues throughout the whole period of the Fodina algorithm. Therefore, the description of the Fodina method is separated into two parts, the first part consists of counting basic relations and dependency graph construction, while the second part contains semantic information connection.

5.1.1 First Part: Counting Basic Relations and Dependency Graph Construction

Three optional operations in the first part of this algorithm should be introduced at the beginning. The first optional operation is to assume that binary conflicts should be avoided when creating the model. The second optional operation is to make sure that all the activities should be fully connected in the process model. The third optional step is to mine the long-distance dependency relationship in the event log. More details will be shown for these three options in the following paragraphs.

Input: An event log L with activity set T_L, a set of tasks T_C with mapping $\mu : \sigma \in L \mapsto T_C$, basic relations $|a > b|$, $|a >> b|$, and $|a >>> b|$, settings t_d, t_{l1l}, t_{l2l}, t_{ld} (thresholds), $noL2lWithL1l$, $noBinaryConflicts$, $connectNet$, and $mineLongDependencies$.

Output: A dependency graph D with start and end tasks t_s and t_e; a set $ldeps$ indicating which dependencies are long distance.

1: $D \leftarrow \{\}$
2: $\forall a \in T_C : \frac{|a>a|}{|a>a|+1} \geq t_{l1l}, D \leftarrow D \bigcup \{(a, a)\}$
3: $\forall a, b \in T_C : \frac{|a>b|}{|a>b|+|b>a|+1} \geq t_d, D \leftarrow D \bigcup \{(a, b)\}$
4: $\forall a, b \in T_C : \frac{|a>>b|+|b>>a|}{|a>>b|+|b>>a|+1} \geq t_{l2l} \wedge (\neg noL2lWithL1l \vee (a, a) \notin D \vee (b, b) \notin D)$, $D \leftarrow D \bigcup \{(a, b), (b, a)\}$
5:
6: $t_s \leftarrow \text{argmax}_{x \in T_C} \sum_{\sigma \in L : x = \mu(\sigma_1)} 1$
7: $t_e \leftarrow \text{argmax}_{x \in T_C} \sum_{\sigma \in L : x = \mu(\sigma_{|\sigma|})} 1$
8: $\forall a \in T_C, D \leftarrow D \setminus \{(a, t_s), (t_e, a)\}$
9:

Figure 8: Basic steps of the Fodina method in dependency graph construction part [5]

Steps from line 1 to line 4 can be concluded as the first procedure of the Fodina algorithm. This procedure can be treated as counting frequency behaviors. Firstly, we should define an empty dependency graph, with three thresholds (range from 0 to 1) to determine whether direct dependency relationships or loop dependencies exist between activities. Then $|a > a|$ will be calculated, which means the number of conditions that activity a directly follows itself. If the value of the formula shown in line 2 is not smaller than the threshold we set for length-one loop dependency, their relationship is confirmed and should be added to the dependency graph. After that $|a > b|$ and $|b > a|$ will be calculated, which means the number of conditions that activity b directly follows activity a; activity a directly follows activity b respectively. Then use the formula in line 3 to estimate whether every two different activities have a direct dependency relationship. If the value obtained from the formula is no smaller than the threshold set for direct dependency, we can add the dependency of the two activities in the dependency graph. Then $|a >> b|$ and $|b >> a|$ will be calculated, $|a >> b|$ is the number of conditions that task b follows task a, also this task b follows another task a. $|b >> a|$

is the number of conditions that task a follows task b, also this task a follows another task b. For a single activity, length-one loop dependency and length-two loop dependency cannot appear together in the dependency graph, which is another restriction when finding length-two loop dependency. When fitting this requirement and the calculation value in line 4 is no smaller than the threshold chosen for length-two dependency, then we can add these two activities as length-two dependency relationships into the dependency graph.

Step from line 6 to line 8 is the second procedure during which the start event and the end event are chosen. The frequencies of different tasks at the first position of each trace in an event log are collected. The task with the largest frequency value is chosen as the start event in the dependency graph. The same logic is applied when choosing the end event, recording frequencies of tasks at the last position of each trace in an event log, then choosing the task with the highest frequency as the end event in the dependency graph. Then after confirming the start event and the end event, we should delete the dependency which shows this start event is output from other events in the dependency graph. Also, the dependency which demonstrates this end event is input to other events should be deleted from the dependency graph.

10: **if** $noBinaryConflicts$ **then**
11: **for** $a, b \in T_C : (a, b) \in D \wedge (b, a) \in D$ **do**
12: $D \leftarrow D \setminus \{(a, b), (b, a)\}$
13: **if** $|a >> b| > 0$ **then** $D \leftarrow D \bigcup \{(a, a)\}$
14: **if** $|b >> a| > 0$ **then** $D \leftarrow D \bigcup \{(b, b)\}$
15: **for** $c \in T_C : c \neq a \wedge c \neq b$ **do**
16: **if** $(c, a) \in D \vee (c, b) \in D$ **then** $D \leftarrow D \bigcup \{(c, a), (c, b)\}$
17: **if** $(a, c) \in D \vee (b, c) \in D$ **then** $D \leftarrow D \bigcup \{(a, c), (b, c)\}$
18: **if** $connectNet$ **then**
19: **repeat**
20: % US, UE is the set of unconnected activities from the start/end task respectively
21: **if** $US \neq \emptyset$ **then**
22: $(bestin, un) \leftarrow \text{argmax}_{(c,u):c \in T_C, u \in US \wedge (c,u) \notin D \wedge c \neq t_e \wedge u \neq t_s} \frac{|c>u|}{(|c>u|+|u>c|+1)}$
23: $D \leftarrow D \bigcup \{(bestin, un)\}$
24: **if** $UE \neq \emptyset$ **then**
25: $(un, bestout) \leftarrow \text{argmax}_{(u,c):c \in T_C, u \in UE \wedge (u,c) \notin D \wedge u \neq t_e \wedge c \neq t_s} \frac{|u>c|}{(|c>u|+|u>c|+1)}$
26: $D \leftarrow D \bigcup \{(un, bestout)\}$
27: **until** All tasks lie on path from start to end
28:
29: $ldeps \leftarrow \{\}$
30: **if** $mineLongDependencies$ **then**
31: **for** $a, b \in T_C : \frac{2|a>>b|}{|a|+|b|+1} - \frac{2||a|-|b||}{|a|+|b|+1} \geq t_{ld}$ **do**
32: **if** $PathExistsFromToWithoutVisiting(t_s, t_e, a) \wedge$
33: $PathExistsFromToWithoutVisiting(t_s, t_e, b) \wedge$
34: $PathExistsFromToWithoutVisiting(a, t_e, b)$ **then**
35: $D \leftarrow D \bigcup \{(a, b)\}$
36: $ldeps \leftarrow ldeps \bigcup \{(a, b)\}$

Figure 9: Setting restrictions part of the Fodina method [5]

Then the first optional restriction about binary conflicts is considered. Binary conflict means two different tasks have direct dependencies in two directions. For example, here is a dependency graph $\{(a, b), (a, c), (b, a), (c, d)\}$, task a and task b are binary conflicts because both dependencies (a, b) and (b, a) can be found in this graph. If binary conflicts exist in the dependency graph, steps from line 10 to line 17 will be applied to convert the dependency of these conflicted tasks into another kind of dependency relationship. If task a and task b are binary conflicts between each other, dependencies (a, b) and (b, a) should be deleted from the dependency graph at the first step. Then we should check whether $|a >> b|$ or $|b >> a|$ is greater than 0. If $|a >> b| > 0$, we can add a one-length loop on task a; If $|b >> a| > 0$, we can add a one-length loop on task b. After that, we should also check whether an activity c is exists which is the input event of both a and b in the dependency graph. If we found that c, the dependency relationships $(c, a), (c, b)$ will be added to the dependency graph. Also, if we find an

activity d which is the output event of both a and b in the dependency graph, the direct dependency $(a, d), (b, d)$ will be added to the dependency graph.

After that, the second restriction about the net connection is considered. The restriction is to make sure all the tasks except the start and the end events are connected to other tasks for both input direction and output direction, also the start event should be connected at its output direction and the end event should be connected at its input direction. Steps from line 18 to line 27 are applied to make all the tasks fully connected. Firstly, two sets are used to collect the unconnected tasks from the input direction and output direction respectively. Then for each task such as task a in the input direction set, we find the task, such as task b, with the highest probability to become the input task a, then we add a dependency (b, a) in the dependency graph. The probability value is proportional to the formula value shown in line 22, which is calculated using $|a > b|$ and $|b > a|$. The same operations are applied to the output direction set.

The last procedure for this part is the long-distance dependency mining, which is also optional. Steps from line 29 to line 36 are created for this mining operation. A long-distance dependency set is created to collect the long-distance dependency between activities. The check method should apply to the combination conditions of any two different tasks. The formula in line 31 is the first test step. The meaning of $|a >>> b|$ has been mentioned in section 4.5, $|a|$ means the frequencies of task a in the whole event log. If the formula value is no smaller than the threshold we set for long-distance dependency, then it can be seen as a potential long-distance dependency. After filtering some dependencies in the first test step, the second test is to find if these three paths exist in the event log. Tasks a and b with long-distance dependency (a, b) will be used to show these paths. The first two paths are from the start event to the end event without visiting task a or task b respectively. The third path is from the event a to the end event without visiting task b. If all three paths can be found in the event log, the potential long-distance dependency becomes the real long-distance dependency. Then both the dependency graph and long-distance dependency set should add this new dependency (a, b). After that, we get our final dependency graph

5.1.2 Second Part: Semantic Information Connection

Operations in the semantic information connection part will use the dependency graph and long-distance dependency created in the dependency graph construction part. The algorithm in this part is used to convert the dependency graph into a causal net by adding split and join information at each node. This part can also be divided into two small operations, which are the pattern-finding step and pattern filtering step. These two operations will be discussed separately.

All kinds of tasks in the dependency graph should be collected into a new set. A loop is created for the new set in order that split and join semantics can be added on each kind of activity singly. Split and join information is divided into input semantics and output semantics for each activity. Only the logic of output semantics mining will be introduced because input semantics mining has the same method as output semantics mining. An Example will be used when explaining the finding pattern function, which is the steps from line 5 to line 26 and it is the most important and difficult step in the Fodina algorithm. We have this event log: $L = [< a1, b, c, d, h, g1, a2, f, g2, e >^2, < a, c, d, g >]$, $a1$ and $a2$ (also $g1$ and $g2$) are the same activities appear at different positions in a trace, the dependency graph: $DG = \{(a, d), (a, f), (a, g), (a, h), (b, d), (b, h)\}$, the long-distance dependency: $LD = \{(a, h)\}$. From the example DG, we get $Tc = \{a, b, d, f, g, h\}$ because these six tasks appear in the dependency graph. Then comes to the algorithm step from lines 1 to 3: we choose to mine the output of activity a as an instance.

Then we come to the pattern-finding function. A set named extracted pattern is created to store the output semantic results for the output of activity a in line 6. Line 8 is the next step for an output direction mining goal. In this step, tasks shown as an output combined with activity a in the dependency graph are found and gathered into a new set C. Therefore, $C = \{d, f, g, h\}$ because the components $(a, d), (a, f), (a, g), (a, h)$ are all exists as an element in the dependency graph. Then there is a loop operation in line 9, which means each activity a in each trace of the event log should be considered respectively. So, we should consider task $a1$ in trace $T1 =< a1, b, c, d, h, g1, a2, f, g2, e >$, then task $a2$ in trace $T1$, then task $a1$ and $a2$ in trace $T2 =< a1, b, c, d, h, g1, a2, f, g2, e >$ respectively, then task a in trace $T3 =< a, c, d, g >$.

Activity $a1$ in trace $T1$ is considered in this paragraph, a new set P is created for recording the output pattern information for activity a in this single trace. Then there is another loop in line 12 for

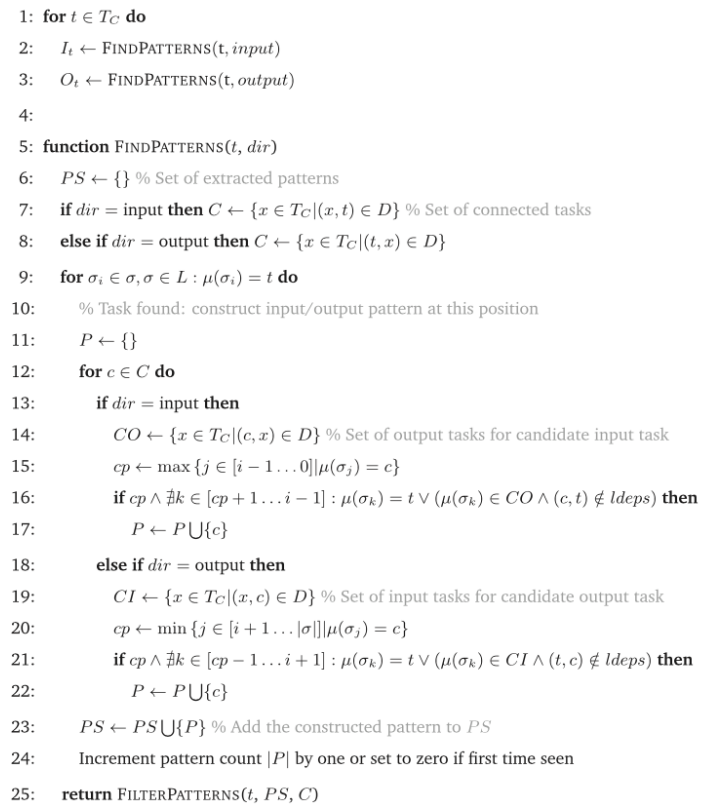

Figure 10: Pattern-finding function in the Fodina method [5]

each task in set C, so task d, f, g, and h will be considered separately. Steps from line 13 to line 17 mine the input information for an activity, which will be ignored in the explanation part because of the same logic. Steps from lines 18 to 25 will be introduced in detail as the output information mining part. Activity d is applied in steps between line 18 and line 22 at first. In line 19, conditions where activity d behaved as an output task in the dependency graph, will be put into a new set CI. Because (b, d) exists in DG, so the input task b in this dependency is put into CI, then $CI = \{b\}$. Then in step line 20, we found the nearest distance between task $a1$ and task d when d is at the fourth position in $T1$. In line 21, we find tasks between the first position ($a1$) and the fourth position (nearest d) is $BT = \{b, c\}$, then we should check whether any element in set BT exists in set CI. Because task b exists in both CI and BI, and dependency (a, d) not exists in the long-distance dependency set. Then task b has a larger probability than task $a1$ as an input activity of task d because the distance between position b and position d (which is $4 - 2 = 2$) is closer than the distance between position $a1$ and position d (which is $4 - 1 = 3$). So, our assumption about task d as an output of task $a1$ is not correct. Then f as the second task in set C is considered. There is no dependency relationship in DG that task f is an output activity, so CI is empty. Then task f exists as the eighth position in $T1$, we find that there is another activity a ($a2$) between $a1$ and f, in which f has a larger probability related to $a2$ instead of $a1$. So, the hypothesis that f is an output of $a1$ is not correct. Then g as the third task in set C is considered. Set CI is also empty in this condition. However, there are two g activities $(g1, g2)$ in $T1$. We only choose $g1$ because the distance between $a1$ and $g1$ ($6 - 1 = 5$) is closer than the distance between $a1$ and $g2$ ($9 - 1 = 8$). Also, CI is empty in this condition, and no activities same as task a exists between $a1$ and $g1$. So g becomes an output activity of a. Then task h is the last task in set C we need to consider. $CI = \{b\}$ at this time because (b, h) exists in the dependency graph. And h is in the fifth position of $T1$, task b (the second position) exists between $a1$ and h. However, at this condition (a, h) exists in the LD, which means a and h have the long-distance dependency, so h can also be the output task of $a1$ although these two activities ($5 - 1 = 4$) have larger distance than b and h ($5 - 2 = 3$). So we get the output tasks for $a1$ in $T1$, which are g and h, written as $\{g, h\}$, we put them into PS, then $PS = \{\{g, h\}\}$.

Then $a2$ in trace $T1$ will be considered. For the first task d in set C, no task d exists after $a2$ position in $T1$, so d cannot be the output task of $a2$. For the second task f in set C, we find task f is just one position after $a2$, no other tasks exist between them, so f is an output task of $a2$. For the third task g in set C, we find $g2$ located after $a2$, CI is also empty at this time, so g is another output task of $a2$. For the last task h in set C, there is no h appears after $a2$ in $T1$, so the output task set for $a2$ in T1 is $\{f, g\}$, we add them into PS, then $PS = \{\{g, h\}, \{f, g\}\}$

Same mining steps will be operated in $T2$, the results will also be added to PS, then $PS = \{2 * \{g, h\}, 2 * \{f, g\}\}$. Then we consider task a in trace $T3$. We can easily find that the output task for a in $T3$ is $\{d, g\}$ without any difficulties. Then we add this set into PS, now $PS = \{2 * \{g, h\}, 2 * \{f, g\}, \{d, g\}\}$. All steps of output tasks mining for activity a are completed and can be treated as the final output in this procedure.

```
26: function FilterPatterns(t, PS, C)
27:   PF ← {} % Set of retained patterns
28:   if |PS| > 0 then
29:     tr ← Σ_{P∈PS} |P| / (|t|×|PS|) % |P| indicates the number of times this pattern was seen,
        |t| indicates the number of times this activity occurs in the log, |PS| indicates the total
        number of patterns found
30:     if t_pat ≤ 0 then tr ← tr + t_pat * tr
31:     elsetr ← tr + t_pat * (1 − tr)
32:     for P ∈ PS : |P|/|t| ≥ tr do
33:       PF ← PF ⋃ {P}
34:   for c ∈ C : ∄P ∈ PS : c ∈ P do
35:     PF ← PF ⋃ {{c}}
36:   return PF
```

Figure 11: Pattern-filtering function in the Fodina method [5]

Here comes to the pattern filtering part (steps from line 26 to 36) after finishing the pattern finding part, the result $PS = \{2 * \{g, h\}, 2 * \{f, g\}, \{d, g\}\}$ got from the example will also be used in this part to explain the algorithm. The basic logic in the filtering part is to delete some sets which have the lower frequencies in PS set. A threshold t_{pat} will be used to control in what kind of extent an element in PS set should be deleted. This filtering operation will be applied to each single kind of activities.

Firstly, a new empty set PF is created to collect the remaining elements after filtering process. Then a value will be calculated using the formula in line 29. Both $\{g, h\}$ and $\{f, g\}$ appear two times in PS, while $\{d, g\}$ appears one time, so $P = [2, 2, 1]$. Activity a appear 5 $(2 + 2 + 1 = 5)$ times in the event log L, so $t = 5$. The total element number in PS is 5 $(2 + 2 + 1 = 5)$, so $|PS| = 5$. Then $tr = \frac{2}{5*5} + \frac{2}{5*5} + \frac{1}{5*5} = \frac{1}{5}$. We set the threshold $t_{pat} = \frac{1}{5}$, so the final $tr = \frac{1}{5} + \frac{1}{5} * (1 - \frac{1}{5}) = \frac{9}{25}$. This final tr will be used as the filter value in next steps.

From line 32, each element in list P should be divided by value t $(t = 5)$, then $P = [2/5, 2/5, 1/5]$ after calculation. If the elements in P are greater than the final tr, we keep them and add them into PF, or we will delete them. So elements $\{g, h\}$ and $\{f, g\}$ are kept $(\frac{2}{5} > \frac{9}{25})$, and element $\{d, g\}$ is deleted $(\frac{1}{5} < \frac{9}{25})$. After that, the final output semantic for activity a is $PF = \{\{g, h\}, \{f, g\}\}$. A causal net can be drawn as the final output of the Fodina algorithm after getting input and output semantics of all the tasks appear in the dependency graph.

5.2 The Optimization Approach

The steps of the BJ long-distance mining approach are shown in this part. This approach can be applied just after the Fodina mining procedures. The input, output as well as functions defined in this approach will be discussed at first. After that, an example will be used to show my steps in more detail.

Five inputs should be caught before running this BJ approach. The first input is the long-distance dependency map *ldeps*, which is confirmed during the dependency graph construction part in the original Fodina mining algorithm. Event Log L is never changed, which is also the input element of the original Fodina algorithm. The third input is split and join semantics PF, which is the final output of the Fodina mining. The last two inputs ts and te are the start and end events deduced at the

Algorithm 1: Long-distance Dependency Mining Approach

Input: long-distance dependency *ldeps* , event log L , split and join semantics *PF,* start task *ts,* end task *te*

Output: *FinalPetriNet* with the best performance

```
1
2  FinalLdeps ⟵ ldeps
3  PetriModel ⟵ GetPetri(PF, ts, te)
4  precision, recall ⟵ Entropia(PetriModel, L)
5  if ldeps is not empty then
6      GroupLdeps ⟵ group elements in ldeps using the input task in each element
7      R1 ⟵ Combine(precision, recall)
8      for group ∈ GroupLdeps do
9          NewLdeps ⟵ delete all elements in group from FinalLdeps
10         NewPF ⟵ FodinaWithLdeps(NewLdeps, L)
11         NewPetriModel ⟵ GetPetri(NewPF, ts, te)
12         NewPrecision, NewRecall ⟵ Entropia(NewPetriModel, L)
13         R2 ⟵ Combine(NewPrecision, NewRecall)
14         if R1 ≤ R2 then
15             R1 ⟵ R2
16             FinalLdeps ⟵ NewLdeps
17
18 FinalPF ⟵ FodinaWithLdeps(FinalLdeps, L)
19 FinalPetriNet ⟵ GetPetri(FinalPF, ts, te)
20
21 return FinalPetriNet
```

beginning of the original Fodina algorithm. The only output for this approach is the $FinalPetriNet$, which is the Petri Net model with the best performance in all three dimensions: precision, recall, and simplicity.

Four functions are defined in this approach, which are named as *GetPetri*, *Entropia*, *Combine*, and $FodinaWithLdeps$. The *GetPetri* function is used to convert a causal net to a Petri Net, more details will be shown in section 5.4. *Entropia* function is a method used to get precision and recall values of a Petri Net model, which will be introduced in more detail in section 6.1. The *Combine* function is a way to combine precision and recall values, or even simplicity values together to get a single value, and this single value can be used to choose the better Petri Net model. More details on the *Combine* function will be shown in section 6.2.

The $FodinaWithLdeps$ function is almost the same as the original Fodina algorithm, but there is a little difference. The $FodinaWithLdeps$ method does not have the steps to confirm the long-distance dependency map because a long-distance dependency map has already been applied to this function as an input, and this map will be used in the split and join semantics operation later. In other words, only steps from line 29 to line 36 in the dependency graph construction part of the Fodina algorithm will be deleted, other steps are the same.

An example will be shown to show the logic of my approach. We assume that the long-distance dependency map we get from the original Fodina algorithm is $ldeps = \{(a,b),(a,c),(b,c),(b,d),(c,f),(c,g)\}$. Steps from line 2 to line 4 are easy to understand, the start value we set for $FinalLdeps$ is $ldeps$. Then a Petri model $M1$ is created using the split and join information PF, the start event ts, the end event te, which are all from the original Fodina method. The precision value pre and recall value rec are obtained after using the *Entropia* method with two input variables: the model $M1$ and the event log L.

Other steps will continue to operate if the original $ldeps$ are not empty. Line 6 is a group operation for a long-distance dependency map. After grouping the example $ldeps$, the $GroupLdeps$ we get is $\{g1,g2,g3\}$, $g1 = \{(a,b),(a,c)\}$ because the input tasks of these two components in $ldeps$ are the same. And $g2 = \{(b,c),(b,d)\}$, $g3 = \{(c,f),(c,g)\}$. Then *Combine* function is used to get a single value from pre and rec, this single value $R1$ will be used in model comparison later. In line 8, a loop is built according to elements in $GroupLdeps$, steps from line 8 to line 16 will be operated three times because

three groups exist in $GroupLdeps$. We consider $g1$ in the first loop, which has two elements: (a, b) and (a, c). $NewLdeps$ is shown as $\{(b, c), (b, d), (c, f), (c, g)\}$ after deleting these two elements from the $FinalLdeps$. After that, $NewPF$ will be obtained using the $FodinaWithLdeps$ function with two inputs: $NewLdeps$ and the event log L. Then in line 11, we get a new Petri Net $M2$ when using $NewPF$, ts, te as inputs of the $GetPetri$ function. In line 12, we get a new precision value $newpre$ and a new recall value $newrec$ from function $Entropia$ using model $M2$ and the event log L as inputs. Then, $R2$ is calculated for model comparison using the $Combine$ function with $newpre$ and $newrec$ as its inputs. Then we come to the comparison part from line 14 to line 16. If $R2$ is greater than $R1$, which means the model with $NewLdeps$ performs better. So, the $FinalLdeps$ should be changed to $NewLdeps$. Then $R1$ should be changed to $R2$ for comparison in the next loop. In this example, we consider $g1$ loop at first, $R1(LD1 = \{(a, b), (a, c), (b, c), (b, d), (c, f), (c, g)\}) < R2(LD2 = \{(b, c), (b, d), (c, f), (c, g)\})$, then $FinalLdeps$ changes from $LD1$ to $LD2$, $R1$ changes to $R2$. Then we consider $g2$ loop, after deleting elements (b, c) and (b, d) from $FinalLdeps$(same as $LD2$), we get $LD3 = \{(c, f), (c, g)\}$, after these operations we assume that: $R1(LD2 = \{(b, c), (b, d), (c, f), (c, g)\}) > R2(LD3 = \{(c, f), (c, g)\})$, then we should keep the value in $FinalLdeps$ and $R1$, which means $FinalLdeps = LD2$, $R1 = R1(LD2 = \{(b, c), (b, d), (c, f), (c, g)\})$. In the last loop, we delete (c, f) and (c, g) elements from $FinalLdeps$ (same as $LD2$), then we get $LD4 = \{(b, c), (b, d)\}$. After operations we get $R1(LD2 = \{(b, c), (b, d), (c, f), (c, g)\}) < R2(LD4 = \{(b, c), (b, d)\})$. Then $FinalLdeps$ will change to $LD4$. After completing loops for all the groups, the long-distance dependency map we get is $LD4 = \{(b, c), (b, d)\}$. Then in steps line 18 and line 19, $FodinaWithLdeps$ and $GetPetri$ are applied to get the best Petri Net model using $LD4$ as the final long-distance dependency map. This Petri Net model is also the final output of my approach.

5.3 Proof the Group Deleting Order is Possible to be Random

A theoretical method will be used to prove that different element orders followed by deleting steps in the long-distance dependency map have little effect on model generation, so a random order can be chosen for deleting steps in the BJ optimization approach. When deleting an element group from the long-distance dependency map, the Petri Net model might be changed into two different types. The first type is mainly focused on deleting the useless long-distance dependency structures of the model. While the second part is focused on changing the input and output relationship for the remaining transitions in the Petri Net.

When considering the first type of structure changes, it can be proved that whenever we delete a particular group in the long-distance dependency map, the places and arcs deleted in the Petri Net due to this operation are always the same. For example, a long-distance dependency map $LM = [g1, g2]$, right now the Petri Net model is $M1$. If we delete group $g1$ in LM, then structure $S1$ is deleted from $M1$ to get a new model $M2$, which can be expressed as $M2 = (M1 - S1)$. Then we compare $M1$ and $(M1 - S1)$ models to check if $S1$ has a significant influence on the Petri Net. Group $g2$ will be deleted in the next step, the structure $S2$ is deleted from $M2$ to get a new model $M3$, which can be shown as $M3 = (M1 - S1 - S2)$. Then we compare $M1$ and $(M1 - S1 - S2)$ models to check if $S2$ has a significant effect on the Petri Net. Now we change the group order in LM and delete $g2$ at first, then the structure that will be deleted is still $S2$. So, two models $M1$ and $(M1 - S1)$ will be compared. After deleting group $g1$, $(M1 - S1)$ and $(M1 - S1 - S2)$ will be considered. Because two long-distance dependency structures with different input transitions in the Petri Net are hardly related to each other, then the relationship between the single input group of the long-distance dependency structure and the main part of the Petri Net will be considered. The main part we defined as $MA = (M1 - S1 - S2)$, which means the Petri Net without all long-distance dependency structures. So, when we delete $g2$ after $g1$, the two steps are applied to determine whether $S1$ has a significant effect on MA, then check whether $S2$ has a significant effect on MA; when we delete $g1$ after $g2$, we determine whether $S2$ has a significant effect on MA firstly, then check whether $S1$ has a significant effect on MA. The problems they solve are the same, which means they must have the same result if $S1$ and $S2$ are independent with each other. Therefore, deleting orders can be done randomly when considering the first structure change condition if all the long-distance dependency structures are independent with each other.

The second type of structural change is considered in this paragraph. Long-distance dependency map $LM = [g1, g2]$ is still used as an example. Unlike changes that do not happen in the main part of the model we just discussed, in this part, we only consider the condition that the main structure of the Petri Net has changed when deleting groups from the long-distance dependency map, which

means we only pay attention to the changes in MA. Then it should be clarified how main structures change according to the BJ approach. Let's assume group $g1$ combines all the conditions that task a is shown as the input in the long-distance dependency map, which can be expressed as $g1 = [(a, c), (a, d)]$. Now we assume there are four elements with task a as their input in the dependency map, shown as $d1 = [(a, c), (a, d), (a, e), (a, f)]$. After deleting group $g1$ in the long-distance dependency graph, then $d1_{new} = [(a, e), (a, f)]$, In the split and join steps of the Fodina method, in the group of elements with the same input tasks in the dependency graph, the percentage of frequencies shown in the event log is used to detect whether these dependency relationships can be shown in the Petri Net. And a threshold value is defined for the filtering procedure.

Then we only consider the dependency relationship (a, e): when d1 changes to $d1_{new}$, the frequency percentage of (a, e) is increased because the denominator is smaller due to the deletion part. There are also three different conditions that should be discussed separately. We define the percentage threshold as $t1$, the frequency percentage can be expressed as $Per(a, e)$. The first condition is when using $d1$ as the dependency map, $Per(a, e) < t1$. When dependency map changes to $d1_{new}$, we can also get $Per(a, e) < t1$. In this condition the dependency relationship (a, e) cannot be shown on the Petri Net for both conditions, so the main part MA does not change. The second condition is that $Per(a, e) > t1$ when using $d1$ or $d1_{new}$ as the dependency map, which means (a, e) exists on the Petri Net for both conditions, and the main part MA does not change. Here is the last condition: $Per(a, e) < t1$ when using $d1$ as the dependency map, while $Per(a, e) > t1$ when using $d1_{new}$ as the dependency graph. It means that dependency relationship (a, e) has been added to the main structure MA when deleting $g1$ in the long-distance dependency map, so it is the only condition of the three that the main part MA has been changed. If MA is not changed in the first two conditions which happens more times than the third condition in the experiment, the deleting order in the long-distance dependency is not necessary. Therefore, the third condition is the only one that needs to focus on. Because all the structure changes are adding places and arcs to the original Petri Net. Then we can define the main structure of the Petri Net according to $d1$ as model MB. We assume that after deleting group $g1$ in the long-distance dependency map, the main structural changes to model $(MB + P1)$, where $P1$ is the added structure. After deleting $g2$, we get model $(MB + P1 + P2)$. Then we change the deleting order, we get model $(MB + P2)$ when deleting $g2$. Model $(MB + P1 + P2)$ is obtained when deleting $g1$. The input task for $P1$ is task a, while the input task for $P2$ is task b. In most cases, $P1$ and $P2$ have low probabilities related to each other, so we can only consider the added part $P1$ or $P2$ has a significant impact on MB. So random order when deleting groups in a long-distance dependency map is reasonable in this condition.

After the whole proof, it makes sure that deleting groups in the long-distance dependency map with multiple orders will not cause a great difference in the result. So, we can choose one group order randomly to make the optimization approach more efficient. The time complexity will change from $O(n!)$ to $O(n)$.

5.4 Method to Convert Causal Net into Petri Net

Algorithm 2: Convert Causel Net into Petri Net

Input: split and join semantics *PF,* start task *ts,* end task *te*
Output: Petri Net *PN*

```
1
2  function GetPetri(PF, ts, te)
3      OD ⟵ convert PF into a dictionary only has output information for tasks
4      OnetoOne, OnetoMore, MoretoOne ⟵ separate OD into three parts
5      ConnectModel1 ⟵ CreateConnection(MoretoOne)
6      ConnectModel2 ⟵ CreateConnection(ConnectMethod1, OnetoMore)
7      ConnectModel3 ⟵ CreateConnection(ConnectMethod1, OnetoOne)
8      ConnectPetri ⟵ create connections for tasks with no input, but have outputs based on
       ConnectModel3
9      PN ⟵ write ConnectPetri to a file as XML format
10     return PN
```

This algorithm above shows the steps of the function *GetPetri*, which is used to convert the output causal net from the Fodina method into a Petri Net, in order to do conformance checking later. There are some input variables in this function, which are split and join semantics PF, start task ts and end task te. This function will return a Petri Net file in XML format, this file can be applied in conformance checking directly.

The first step for this function is to convert our split and join semantics into one output dictionary. The original split and join semantics are stored in two dictionaries, the first dictionary is an input dictionary, which contains the input information for every single task. And the second dictionary is an output dictionary, which stores the output information for every single task. Our goal is to combine these two dictionaries into one dictionary showing output conditions for tasks only. After this operation, three kinds of elements exist in the output dictionary OD, the first one is a single task that has only one output, the second one is a single task that has more than one output, and the third one is a group of tasks has only one output. These kinds of outputs should be separated from the single output dictionary.

Then it comes to the connection confirmation part. Connection structures are created for the condition that a group of tasks with a single one output at first. And-join relations are added in this part. The new connection relationships from the condition that a single task with more than one output are added to the connection structures just created in step 5. And-split relations are mainly added in this part. After that, the connection relationships in the condition of a single task with only one output are added to the connection structures created in step 6. This creation steps order is the most efficient way to build Petri Nets.

After these steps, the condition that some tasks with no input, but have outputs should be considered. We use start events as their inputs and connect them with start events to make a final connection structure. It is sure that no matter how the long-distance dependency relationship in PF changes, connection methods between the start event and these tasks with no inputs will not be changed, which means these events cannot affect the precision and recall comparison. After completing connection structures, it comes to the writing file part. Places and transitions are created firstly in XML format, then arcs connected between places and transitions are created in this file. After these steps, we get the Petri Net, which can be used as an input for the Entropia testing tool to get precision and recall directly.

6 Experiment Evaluation

6.1 Entropia Conformance Checking Tool

Entropia [14] is a command-line that contains a family of conformance-checking methods. In this essay, exact matching precision and exact matching recall [18] will be used to calculate the precision and recall of a model. Function *Entropia* in Algorithm 1 is used to represent this testing step. An event log and a Petri Net model are applied as two inputs of the Entropia method. Then precision and recall values will be returned after calculation.

In the exact matching method, there are two kinds of traces named relevant traces and retrieved traces. Relevant trace relates to all traces that appear in the event log, while retrieved trace relates to all traces that can be deduced in the Petri Net model. Then we use $m(rel \cap ret)$ to represent the magnitude of the shared behavior specified by relevant and retrieved behaviors of the model. Precision is defined as the ratio of the shared behavior value divided by the magnitude of the retrieved behavior, which can be shown as $precision = m(rel \cap ret)/m(ret)$. Recall is defined as the ratio of the shared behavior value divided by the magnitude of the relevant behavior, which can be shown as $recall = m(rel \cap ret)/m(rel)$.

Precision and recall are values between 0 and 1. Low precision means many traces generated from the Petri Net model cannot be found in the event log. Low recall means many traces exist in the event log that cannot be shown in the Petri Net model. They are two main difficulties we should overcome when designing a discovery process mining algorithm.

6.2 Thresholds Setting in the Fodina Algorithm

To generate enough samples in our experiment, thresholds in the Fodina method will be set at first. These thresholds can be divided into a binary group and a value group. The binary group consists of thresholds that should be set as true or false. All four thresholds in this group are applied in the dependency graph construction part. These threshold meanings have been discussed in detail in section 5.1. This section only shows the reason why these threshold values are chosen for generating samples. Firstly, we set $noBinaryConflicts = True$, which makes sure that all length-two loops are converted into other structures when building the model. Because the optimization mainly focuses on long-distance dependency. Without considering length-two loops has little influence on the long-distance mining part, another reason is that length-two loops in a model may affect the conformance checking process due to its complex structure. Then, we set $noL21WithL1l = True$, which means there are no length-one loop exists in a length-two loop. It is also a choice for simplifying the model. Thirdly, we set $connectNet = False$, which means that each task that appears in the event log does not need to be fully connected by other tasks. It is useful to delete some tasks that occur only a few times in the event log. Fourthly, we set $mineLongDependency = True$, which means to do the long-distance dependency mining and it is our goal exactly.

Four binary thresholds are used to control the Fodina operation, while the other five threshold values are applied as a filter to delete some unnecessary dependency relationships. Thresholds t_{L1L} and t_{L2L} are used in length-one loop mining and length-two loop mining, which are not related to the long-distance dependency, so the fit value 0.9 is given to these two thresholds and never changes during the experiment. The other three thresholds in value type that have the main effect in long-distance dependency mining, are t_d, t_l, and t_{pat}. They will be discussed in more detail.

t_d is between 0 and 1, and it is used to determine whether this direct dependency relationship can be stored in the dependency map. If the value we calculated from two events with a dependency relationship is larger than t_d, it is considered they have a dependency relationship and put them in the dependency map. If the value is smaller than t_d, these events are considered with no dependency relationship and cannot add them in the dependency map. Therefore, a larger t_d means a dependency map has a smaller size of dependency relationship, and two events with more probabilities of dependency relationship remaining in the map, the model generated will be simpler. Smaller t_d means dependency relationships with lower probabilities can be added to the map, then the model generated will be more complex. Therefore, t_d value should be chosen carefully to construct a reasonable model, and the dependency map should contain enough elements for model construction, also dependency relationships with low probabilities cannot be added to the dependency map for model accuracy and model simplicity. After some experiments using different t_d in the Fodina method, we choose a value between 0.5 and 0.8.

t_l is the threshold to determine whether two events have a long-distance dependency relationship, which is also between 0 and 1. If the value calculated from two events is greater than t_l, then it is confirmed these events have a long-distance dependency relationship, and this relationship between the two tasks will be added to the dependency map and long-distance dependency map. To meet the experiment requirement, the long-distance dependency map should contain enough elements when applying the BJ optimization approach, so t_l cannot be too high. Also, the long-distance dependency map should not contain long-distance dependency with low probability, so t_l cannot be set too low. After several experiments, we set t_l between 0.4 and 0.9. The reason why the lowest value in t_l is smaller than the lowest value in t_d is that more restrictions are set for long-distance dependency determination except t_l threshold, while t_d is the only restriction set for direct dependency determination. Therefore, t_d must be chosen more seriously for restriction purposes.

After setting t_d and t_l, the last threshold t_{pat} is considered. t_{pat} value is between -1 and 1, which can be treated as a filter when adding split and join semantics in the Casual Net model based on the dependency map and event frequencies. Larger t_{pat} means restrictions is more serious and only dependency relationship with higher frequencies can be kept after the filtering procedure. Although t_{pat} is a threshold, the filter value is different when considering every single task, and the filter value for one task is calculated according to t_{pat} threshold and frequency number of the dependency relationships which contain this task. Smaller t_{pat} means little restriction set in the split and join mining period and our causal net contains many infrequency dependency relationships. So middle values (between 0.2 and 0.8) are chosen in our experiment.

After considering these thresholds, six pairs' thresholds are chosen to apply to 25 event logs. These

event logs are stored in Process Discovery Contest 2020 at this website https://www.tf-pm.org/resources/logs. For each event log, it has 1000 traces, and each traces contains decades of tasks. Also it should be confirmed that the output split and join semantics must be different for the same event log when thresholds are different. Here is the threshold table containing thresholds we apply on each event log.

t_d	t_l	t_{pat}
0.8	0.9	0.7
0.8	0.9	0.5
0.7	0.8	0.6
0.7	0.8	0.4
0.6	0.8	0.5
0.6	0.8	0.3

Table 2: The thresholds setting table

6.3 Discuss the Method to Combine Precision, Recall and Simplicity

A method to combine precision, recall, and similarity into a single value will be introduced in this section. This operation is expressed as function *Combine* in Algorithm 1. This single value will be used in model comparison to determine which model is better. There are many methods to combine precision and recall into a single value, such as F measure [19], which is shown as $F_1 = \frac{2*precision*recall}{precision+recall}$ and F_β measure [20], which is shown as $F_\beta = \frac{(1+\beta^2)*precision*recall}{(\beta^2*precision)+recall}$, and β is a real factor in this formula. But these methods do not consider the effect of similarity. Therefore, a new formula needs to define for these three dimensions combination.

Here is an example, for a Petri Net model $M1$, its precision is $p1$, recall is $r1$; for another Petri Net model $M2$, its precision is $p2$, recall is $r2$, then a threshold is defined as t. Then we have a formula $s = (p2 + r2) - (p1 + r1) + t$, single value s will be used in model comparison. If s value is greater or equal to 0, $M2$ is chosen, otherwise, $M1$ is chosen. In the BJ approach, $M1$ is always generated before $M2$ by the Fodina method, which means $M1$ always has more elements in its long-distance dependency map than $M2$, so $M2$ is often simpler than $M1$. Before setting threshold t, it's important to know which model seems to be simpler than the other. If $M2$ is simpler than $M1$, t should be positive. If $M2$ is more complex than $M1$, t should be negative. Threshold t is set mainly to balance the small errors caused by the Entropia conformance checking method. For example, if the sum of precision and recall in $M1$ and $M2$ ($M2$ has more probabilities simpler than $M1$, so t is positive) is nearly the same, which means $(p2 + r2) - (p1 + r1) \in (-t, t)$, then $s \in (0, 2t) > 0$, so consider the simplicity dimension, because $M2$ has more probabilities to be simpler than $M1$, so we choose $M2$ after considering the simplicity dimension. Also using $M1$ and $M2$ as an example, another condition is that the sum of precision and recall for $M1$ is much greater than $M2$, which can be expressed as $(p2 + r2) - (p1 + r1) \in (-\infty, -t)$, then $s \in (-\infty, 0) < 0$. It means if the precision and recall value of $M1$ is much greater than $M2$, we should choose $M1$ without considering the simplicity. The last condition is the sum of precision and recall for $M1$ is much smaller than $M2$, which can be shown as $(p2 + r2) - (p1 + r1) \in (t, +\infty)$, then $s \in (2t, +\infty) > 0$. It means if the precision and recall value of $M2$ is much greater than $M1$, also $M2$ is simpler than $M1$, so $M2$ will be chosen with no doubt.

Similarity value is not used in this combined function because it has a different range $(0, +\infty)$ from precision range $(0, 1)$ and recall range $(0, 1)$. It is difficult to set a fixed coefficient for similarity value because it is hard to make the similarity value always in the same scale with precision and recall in different sizes of Petri Nets. So, the threshold t is set in my method to separate the precision and recall results comparison into three conditions. when comparing models, the weight of simplicity can also be changed in some conditions. After considering the Entropia method error and the event log size, we set $t = 0.1$. This threshold can be set higher if adding more weights on the simplicity dimension when doing the model comparison.

6.4 Algorithm Comparison

In this section, the final plots of algorithm comparison will be shown and discussed. In the precision-recall plot, recall and precision values are shown on the x-dimension and y-dimension respectively in Figure 12.

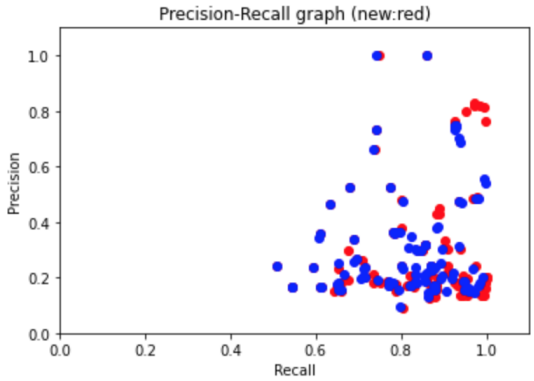

Figure 12: Precision-Recall plot

There are two kinds of nodes in this graph, the blue nodes represent Petri Net models generated by the original Fodina method, while the red nodes show Petri Net models generated by the BJ approach. It's shown that the recall value of these samples is between 0.7 and 1, and their precision value is almost between 0.15 and 0.4. In most parts of the plot, red points and blue points are close to each other, especially in the right bottom part, where recall is around 0.9 and precision is nearly 0.2. It means differences between these two algorithms cannot be found in this part. While in the right top of the plot, there are only some red points located, with no blue points surrounding them. Precision for this small red points cluster is around 0.8, recall is around 0.95. They are the best points when considering precision and recall together. In these samples, models obtained from the BJ approach have better performance than the original Fodina method in precision and recall dimensions, which means the new long-distance dependency optimization approach works better than the Fodina method when mining some event log samples.

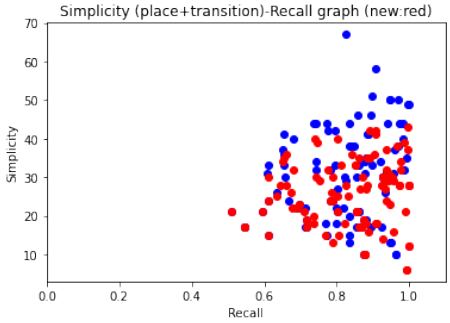

Figure 13: Simplicity-Recall plot I

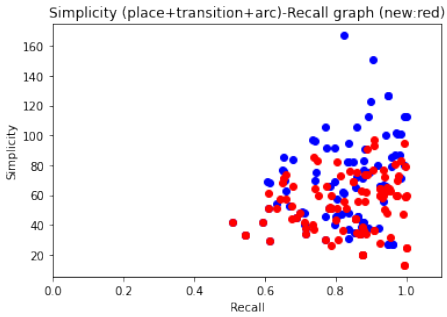

Figure 14: Simplicity-Recall plot II

Then simplicity dimension is considered in the next four plots. The relationship between recall and simplicity is shown in the two plots located in the Figure 13 and Figure 14. Simplicity is defined as the sum of place and transition in the Figure 13, and it is defined as the sum of place, transition, and arc in the Figure 14. Points with two colors located in these plots, red points and blue points represent models generated by the Fodina method and the BJ approach respectively. These two plots give the information that there is no significant change in recall value from blue points to red points, while simplicity values drop down dramatically from blue nodes to red nodes, which means models generated by the BJ approach have lower simplicity values than models generated by the Fodina method. Therefore, the BJ optimization approach can generate much simpler models without reducing recall value. The relationship between precision and simplicity is shown in the Figure 15 and

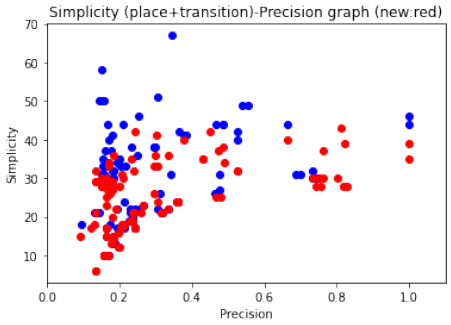

Figure 15: Simplicity-Precision plot I

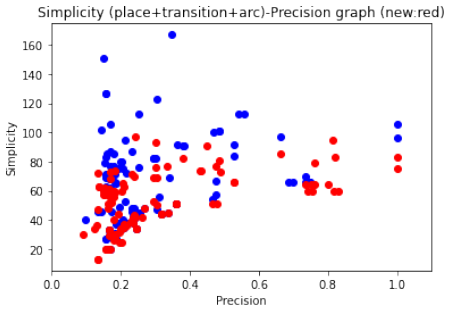

Figure 16: Simplicity-Precision plot II

Figure 16. These two plots show that the changes in precision value from blue points to red points are small, while the simplicity values decrease obviously, which means the BJ approach can construct simpler models without reducing precision value.

In conclusion, the BJ approach generates models with better precision and recall than the original Fodina method in some conditions. Making models become simpler without reducing their precision and recall is another benefit of this new approach, which means this method can delete useless structures in the Petri Net models.

7 Discussion

Although this BJ optimization approach can generate a better model in many conditions, there are still some disadvantages to this BJ approach. Firstly, in the step of converting Casual Nets into Petri Nets, it is assumed that all information is transformed from the Casual Net to the Petri Net, and no extra information is generated in the Petri Net. So, in the future design, a new method will be created to generate a Petri Net directly using a dependency map, then we will get a more accurate model.

Secondly, if a group in a long-distance dependency map contains many elements, we cannot be sure that keeping all of them or deleting all of them will generate the best model. Maybe keeping half and deleting half of these elements will get the best model. Therefore, the future plan is to divide large groups into a few parts and delete one part at each time to check whether this small part is necessary for the Petri Net model.

Thirdly, in the experiment evaluation part, it is not accurate to use the sum of precision and recall value to combine the model performance in precision and recall dimensions. Therefore, I will try to combine the F measure and simplicity dimension using a new formula to do a model comparison.

Fourthly, we know the Fodina method gives the same weight to direct dependency relationships and long-distance dependency relationships in the dependency map during the filtering procedure. Although the BJ approach will be used to solve this problem. My future plan is to change the weights of different kinds of dependency relationships in the dependency map when filtering elements in the split and join mining procedure, then the efficiency of this future algorithm and the model performance which generated from the future method will be used to compare with the BJ approach this essay introduced. Also more dimensions in the conformance checking, such as generalization dimension, will be used in our checking period.

8 Conclusion

In this essay, we discussed the long-distance dependency mining optimization approach based on the Fodina method. This method can simplify Petri Net models by deleting useless long-distance dependency relationship structures. It can also increase the precision and recall of the model by modifying the direct dependency relationship in the main structure of the model in some conditions. And the time complexity of this algorithm is $O(n)$, where n is the group number in the original long-distance dependency map. Therefore, this method is useful and efficient when mining long-distance dependency relationships in the event log.

References

[1] Wil van der Aalst. "Data Science in Action". In: *Process Mining: Data Science in Action*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–23. ISBN: 978-3-662-49851-4. DOI: 10.1007/978-3-662-49851-4_1. URL: https://doi.org/10.1007/978-3-662-49851-4_1.

[2] Wil van der Aalst. "Process Discovery: An Introduction". In: *Process Mining: Data Science in Action*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 163–194. ISBN: 978-3-662-49851-4. DOI: 10.1007/978-3-662-49851-4_6. URL: https://doi.org/10.1007/978-3-662-49851-4_6.

[3] Anna Kalenkova et al. "Automated Repair of Process Models Using Non-local Constraints". In: *Application and Theory of Petri Nets and Concurrency*. Ed. by Ryszard Janicki, Natalia Sidorova, and Thomas Chatain. Cham: Springer International Publishing, 2020, pp. 280–300. ISBN: 978-3-030-51831-8.

[4] A.J.M.M. Weijters and J.T.S. Ribeiro. "Flexible Heuristics Miner (FHM)". In: *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. 2011, pp. 310–317. DOI: 10.1109/CIDM.2011.5949453.

[5] Seppe K.L.M. vanden Broucke and Jochen De Weerdt. "Fodina: A robust and flexible heuristic process discovery technique". In: *Decision Support Systems* 100 (2017). Smart Business Process Management, pp. 109–118. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2017.04.005. URL: https://www.sciencedirect.com/science/article/pii/S0167923617300647.

[6] Adriano Augusto et al. "Automated Discovery of Process Models from Event Logs: Review and Benchmark". In: *IEEE Transactions on Knowledge and Data Engineering* 31.4 (2019), pp. 686–705. DOI: 10.1109/TKDE.2018.2841877.

[7] W. van der Aalst, T. Weijters, and L. Maruster. "Workflow mining: discovering process models from event logs". In: *IEEE Transactions on Knowledge and Data Engineering* 16.9 (2004), pp. 1128–1142. DOI: 10.1109/TKDE.2004.47.

[8] A. Medeiros, A. Weijters, and Wil Aalst. "Genetic process mining: An experimental evaluation". In: *Data Mining and Knowledge Discovery* 14 (Apr. 2007), pp. 245–304. DOI: 10.1007/s10618-006-0061-7.

[9] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. "Discovering Block-Structured Process Models from Event Logs - A Constructive Approach". In: *Application and Theory of Petri Nets and Concurrency*. Ed. by José-Manuel Colom and Jörg Desel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 311–329. ISBN: 978-3-642-38697-8.

[10] A. J. M. M. Weijters and W. M. P. van der Aalst. "Rediscovering Workflow Models from Event-Based Data Using Little Thumb". In: *Integr. Comput.-Aided Eng.* 10.2 (Apr. 2003), pp. 151–162. ISSN: 1069-2509.

[11] Artem Polyvyanyy et al. "Impact-Driven Process Model Repair". In: *ACM Trans. Softw. Eng. Methodol.* 25.4 (Oct. 2016). ISSN: 1049-331X. DOI: 10.1145/2980764. URL: https://doi.org/10.1145/2980764.

[12] Wolfgang Reisig and Grzegorz Rozenberg. "Informal Introduction to Petri Nets". In: *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 1–11. ISBN: 3540653066.

[13] Wil van der Aalst. "Process Modeling and Analysis". In: *Process Mining: Data Science in Action*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 55–88. ISBN: 978-3-662-49851-4. DOI: 10.1007/978-3-662-49851-4_3. URL: https://doi.org/10.1007/978-3-662-49851-4_3.

[14] Artem Polyvyanyy et al. "Entropia: A Family of Entropy-Based Conformance Checking Measures for Process Mining". In: *ICPM Doctoral Consortium / Tools*. 2020.

[15] Wil van der Aalst. "Advanced Process Discovery Techniques". In: *Process Mining: Data Science in Action*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 195–240. ISBN: 978-3-662-49851-4. DOI: 10.1007/978-3-662-49851-4_7. URL: https://doi.org/10.1007/978-3-662-49851-4_7.

[16] Anja Syring, Niek Tax, and Wil Aalst. "Evaluating Conformance Measures in Process Mining Using Conformance Propositions". In: Nov. 2019, pp. 192–221. ISBN: 978-3-662-60650-6. DOI: 10.1007/978-3-662-60651-3_8.

[17] Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. "Replaying history on process models for conformance checking and performance analysis". In: *WIREs Data Mining and Knowledge Discovery* 2.2 (2012), pp. 182–192. DOI: https://doi.org/10.1002/widm.1045. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1045. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1045.

[18] Artem Polyvyanyy et al. "Monotone Precision and Recall Measures for Comparing Executions and Specifications of Dynamic Systems". In: *ACM Trans. Softw. Eng. Methodol.* 29.3 (June 2020). ISSN: 1049-331X. DOI: 10.1145/3387909. URL: https://doi.org/10.1145/3387909.

[19] Yutaka Sasaki et al. "The truth of the F-measure". In: *Teach tutor mater* 1.5 (2007), pp. 1–5.

[20] C. J. Van Rijsbergen. *Information Retrieval*. 2nd. Butterworth-Heinemann, 1979.