Graph Learning for Process Discovery

TESI DI LAUREA MAGISTRALE IN

COMPUTER SCIENCE AND ANGINEERING

|  |  |
| --- | --- |
| Student ID:  Advisor:  Co-advisor:  Academic Year: | 010499057  Matteo Matteucci  Pietro Portolani  202122 |

Author: **Matteo Badenchini**

# Abstract

**Keywords.**

# Abstract in lingua italiana

# Contents

[Abstract i](#_Toc112774703)

[Abstract in lingua italiana iii](#_Toc112774704)

[Contents 5](#_Toc112774705)

[Introduction 8](#_Toc112774706)

[1. Process Mining 9](#_Toc112774707)

[1.1 Overview and objectives 9](#_Toc112774708)

[1.2 Data 11](#_Toc112774709)

[1.3 Models 12](#_Toc112774710)

[1.3.1 Directly Follows Graphs 12](#_Toc112774711)

[1.3.2 Petri Nets and Workflow Nets 12](#_Toc112774712)

[1.3.3 Transition Systems 14](#_Toc112774713)

[1.3.4 Dependency Graphs and Causal Nets 14](#_Toc112774714)

[1.3.5 Process Trees 15](#_Toc112774715)

[2. Process Discovery 17](#_Toc112774716)

[2. 17](#_Toc112774717)

[2.1 Overview and objectives 17](#_Toc112774718)

[2.2 Challenges of Process Discovery 18](#_Toc112774719)

[2.3 Alpha relations and footprint 19](#_Toc112774720)

[2.4 Metrics 20](#_Toc112774721)

[2.4.1 Fitness 20](#_Toc112774722)

[2.4.2 Precision 22](#_Toc112774723)

[2.4.3 Simplicity 22](#_Toc112774724)

[2.4.4 Generalization 22](#_Toc112774725)

[2.5 State of the Art 22](#_Toc112774726)

[2.5.1 Algorithmic Miners 23](#_Toc112774727)

[2.4.2 Deep learning-based miner 28](#_Toc112774728)

[3. Bibliography 32](#_Toc112774729)

[4. List of Figures 35](#_Toc112774730)

# Introduction

# Process Mining

In this chapter we will introduce the fundamental concepts related to the field of process mining, the goals it tries to achieve, the data, and the structures used to model real-life processes. The content of this chapter is of paramount importance since it defines the framework in which the project is developed. In the next chapter we will give a more in-depth explanation of the topic of process discovery, which denotes the main theme of the present work.

## Overview and objectives

Process mining can be defined as the bridge between data science and process science[[1]](#footnote-1) [1]. The goal of process mining is to discover, analyze and improve existing (business) processes by extracting the relevant information from event data stored inside information systems.

A business process is a collection of tasks and activities performed by people or (autonomous) systems that produces an outcome that contributes to the business goal. Processes are frequently described in form of graphs (e.g., Petri Nets), and this representation embodies the so-called model of the process. Event data is the result of the execution of such models.

Process mining represents the core of the Business Process Management (BPM) lifecycle, which describes the distinct phases of managing a particular business process.

Diagram

Description automatically generatedW. van der Aalst defines the BPM lifecycle as composed of three different phases [2]: *(re)design*, *implement/configure*, and *run and adjust*.

Figure 1

After being designed in the *(re)design* phase, the model is transformed into a running system in the *implementation/configuration* phase. Once an implementation of the model is available, the *run and adjust* phase starts: the model is executed, and event data is collected. This data can then be used to extract the actual behavior of the model, which is later analyzed and compared against the expected behavior of the blueprint. Finally, if necessary, the model is redesigned, in the attempt to optimize (or remove) bottlenecks, include observed behavior and remove unobserved one.

The *run and adjust* phase is especially important, since it allows to close the cycle, and therefore to incorporate into the design information about the actual execution of the process in a systematic way.

In the field of process mining, we can identify three different branches (or type of tasks) that conduct the distinct phases of the BPM lifecycle: play-in, play-out and replay:

* Play-in, also known as discovery, refers to the practice of building the model of a process, starting from its observed behavior (event data). It represents the starting point of the BPM lifecycle, since it generates the model that is then put into production and subsequently analyzed and refined.
* Play-out consists instead in generating the behavior of a known process starting from the model that describes it. In this sense it reaches the opposite goal of play-in: it tries to produce event data (usually in form of logs) by simulating the model of the process.
* Replay is responsible for testing the model of a given process against the actual behavior observed. It serves different purposes: to spot bottlenecks, to find deviations from the blueprint and to try to predict problems that can arise from using such a model.

## Data

The starting point of any process mining task is event data, which can come, for example, from information and ERP systems, csv files, APIs, emails, or transaction logs. Data is usually gathered from different sources, and then refined and assembled into event logs, each one expressing the observed behavior of a different process.

Inside an event log, each entry represents a single event, which corresponds to the name of an activity (label) inside the process model. Besides the activity name, an event is usually characterized by other attributes, the timestamp and the case ID being among the most important ones. The case id denotes the unique identifier of a single run of the process model (case): a sequence of activities that, if performed, allows to go from the start of the process to the end of it. Each event belongs to a single case, but different cases may contain the same activity/event (an example of this are the start and end activity, which are always executed for every case).

Other attributes (e.g., resource availability, cost, user/patience name) are possible, and usually present, and are used to better characterize the execution of the process. Attributes can be at case level, if they are shared by all the events of the case (e.g., the patient’s name in case of medical records), or at event level, if they are unique for the event. Figure 2 shows an example of a real-life event log, in which besides case id, activity name and timestamp (*case item*, *event concept: name* and *event time: timestamp* respectively) we can see many more attributes that characterize the process.

A sequence of activities belonging to the same case is also known as a trace, and a log can be seen as a collection (or multiset) of traces. A trace, taken without its case, can appear multiple times inside a log, since different runs of a model can produce the same outcome reached by means of the same sequence of activities. A variant, instead, is a unique trace inside a log.

Let us clarify these definitions with an example. Suppose to have a log written in form of a multiset of traces (so a set in which each element is reported with its multiplicity), and be L = {<a, b, c, d>3, <a, c, b, d>2, <a, e, d>} such a log. Inside L we can identify three different variants, namely <a, b, c, d>, <a, c, b, d> and <a, e, d>, for a total of six traces.

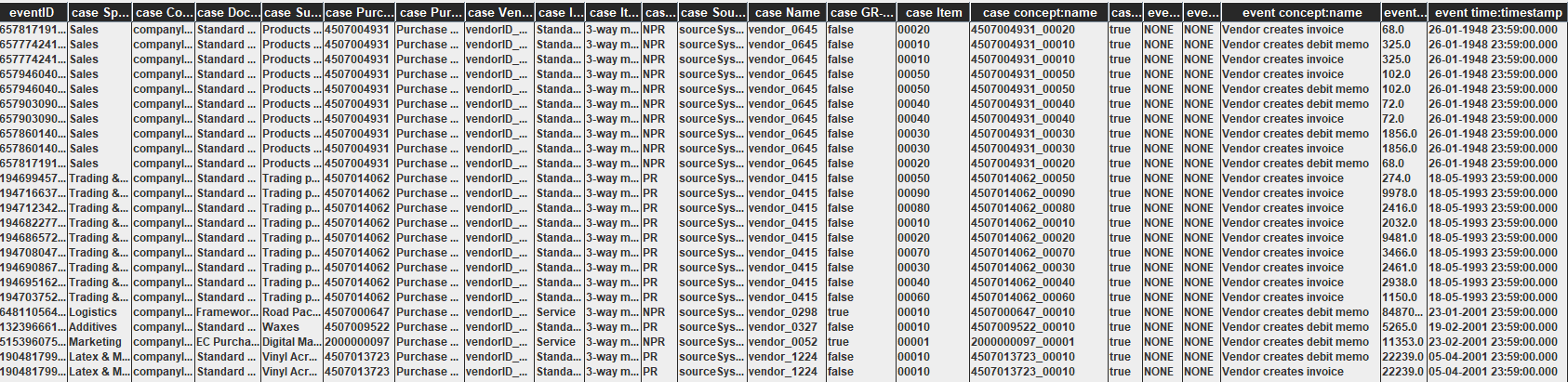


Figure 2

## Models

Processes can be represented in a variety of forms, depending on one’s needs. In this paragraph we will discuss some of the most popular models, be they the final representation that can be presented to the end user or the starting point (or the intermediate representation) of a discovery algorithm.

### Directly Follows Graphs

The directly follows graph is one of the most straightforward representations of a process and the easiest to build starting from event logs. It consists of a directed graph whose set of nodes corresponds to the set of unique activity names in the event log. Arcs connect activities that directly follow each other inside a trace. The result is a model that captures the temporal relations among activities, but not necessarily causal dependencies (e.g., in case of parallel activities). Logs can be filtered beforehand (by removing infrequent activities and/or traces) to avoid building so-called spaghetti models, keeping only the most interesting information.

### Petri Nets and Workflow Nets

A petri net is a bipartite and directed graph composed of places, transitions and arcs connecting places to transitions and vice versa. A place contains a given number of tokens, which represent the resources that can be used to carry out the execution of the process. Places are usually displayed as circles, transitions as boxes and tokens as black dots inside a place.

Diagram

Description automatically generatedIn the process mining context, transitions are associated with the activities of the log, while places are the nodes that control the order of execution of the activities by receiving tokens from (a subset of) the transitions in their backward star and passing them to (a subset of) the transitions in their forward star.

Figure 3

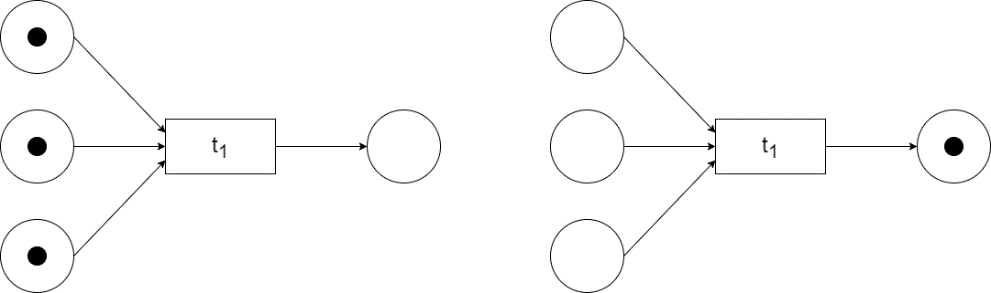


Figure 4

A transition is said to be enabled if each input place (those in its backward star) contains at least a token. When a transition is enabled, it can *fire* by consuming a token from each input place and producing a token for each output place. As you can in Figure 3, transition t1 is not enabled, since one of its input paces does not contain any token. In Figure 4, instead, we can see how the transition t1, once enabled, consumes all input tokens, and produces a new token for each output place (in this example just the one).

The distribution of the tokens inside a net is called marking and it defines the state of the net. A marking can be reachable or unreachable, depending on whether it is possible for the net to reach such a configuration starting from the initial state.

In principle, a place can contain an infinite number of tokens. We can however define an upper bound to such a number and say that a place is k-bounded if there is no reachable marking for which said place holds more than k tokens. A petri net is defined k-bounded if all its places are k-bounded. When k is equal to one, the place (or the net, if the property holds for all its places) is said to be safe.

Petri nets can suffer from deadlock problems: these can arise whenever there is a reachable marking in which no transition is enabled[[2]](#footnote-2). The complete absence of such problems can be verified in live petri nets, nets such that, for every transition, it is possible to reach a marking that enables it starting from the initial reachable marking [3].

When a petri net has a well-defined start (source) and end (sink) place, it is called workflow net. Workflow nets are typically preferred when modeling a process, since the presence of a well-defined source and sink leads to models that are more easily interpretable by humans.

### Transition Systems

A transitions system is a type of model that tries to represent a process as a set of states connected to one another. As already discussed, a log can be seen as a set of traces, each of which specifies a possible execution of a model. In the context of a transitions system, the position of the execution inside a trace defines a state of the process, and the set of all possible states generated in this way (considering all the traces available for a given process) represents the set of nodes in the transition system.

Given two states, if it is possible to transition from the first state to the other by means of a single action, meaning by executing a single activity, the nodes corresponding to such states are connected. The arc connecting two states has a label that corresponds to the name of the activity that needs to be executed to transition from a state to the other.

### Dependency Graphs and Causal Nets

A dependency graph is a directed graph that embeds the causal dependencies among activities in an event log. In this graph, each node corresponds to a unique activity in the event log, while the arcs connect nodes that are causally dependent from one another. Differently from directly follows graphs, two nodes are connected if and only if the corresponding activities are such that one enables the other, that is, given two activities A and B, A must be concluded before B starts, and the execution of B depends on whether A is executed in the first place. Another way to picture such a structure is to think of a petri net from which the places have been removed.

Causal nets are strictly related to dependency graphs: nodes and arcs have the same meaning. However, in this model, each activity (node) has a set of possible input bindings and a set of possible output bindings. Given an activity A, an input binding represents a set of activities that enable A, while an output binding represents the set of activities that activity A can enable [4].

|  |  |
| --- | --- |
| Figure 5 | Figure 6 |

Figure 5 depicts an example of output bindings: activity *a* can enable either the set {*b, d*} or {*c, d*}, but not both. An output binding creates an obligation: once a binding is chosen, all the activities in it must be executed. Figure 6 describes an example of input bindings: activity *e* can be enabled by either the couple {*b, d*} or {*c, d*}, but not both.

Let us consider the bindings in Figure 5 and Figure 6 as belonging to the came causal net: if activity *a* enables *b* and *d*, then activity *e* will be enabled by *b* and *d*, and not *c* and *d*. In other words, an obligation created by an output binding must be consumed by a (set of) input bindings for the activities involved.

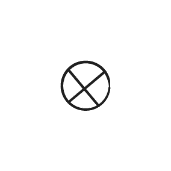
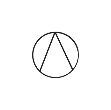
In a causal net the start activity has a no input bindings, while the end activity has no output bindings.

### Process Trees

Process trees are another powerful, yet easy to interpret, way of modeling a process. It describes a process as a tree whose leaves are the activities of process, and the internal nodes are instead operators that connect the different activities in a hierarchical way.

There are four possible operators, each of which defines a specific relation among its children:

* A picture containing shape

  Description automatically generatedSequence : n-ary operator; all the child nodes are are executed one after the other
* Exclusive choice : n-ary operator; only one of the child nodes is executed
* Parallel : n-ary operator; all the child nodes are executed
* Loop A picture containing icon

  Description automatically generated : binary operator; the left child defines the condition, while the right one represents the body of the loop.

# Process Discovery

In this chapter we will discuss the main ideas and objectives of process discovery, the branch of process mining devoted to the construction of models from event logs. Before delving into the description of some of the most famous algorithms, also known as miners, we will introduce the concept of alpha relations, which provides useful guidelines when trying to build Petri/Workflow nets, and the metric used to evaluate the goodness of a model. Then we will give an overview of some of the most famous algorithmic miners available in commercial tools, highlighting their strengths as well as their weaknesses. Lastly, we will describe the most recent attempt to the problem of process discovery: a deep learning-based miner that tries to overcome some of the limitations of its algorithmic counterparts by adopting a more human-like and explainable approach.



## Overview and objectives

As briefly discussed in the previous chapter, process discovery is the task of process mining that has the goal to map the content of an event log onto a process model (Petri net, process tree and so on) that is representative of the behavior seen in the event log.

Diagram

Description automatically generated

Figure 7

Ideally the result of such task is a sound workflow net that is able to replay all the traces seen in an event log without overfitting or underfitting. In other words, the resulting model should also have the ability to model unseen behavior, without generalizing too much. To further clarify this concept let us consider the log L = {<a, b, c, d>3, <a, c, b, d>2, <a, e, d>}. A model that can describe this log is for example the so-called *flower model* of Figure 7, that is considered to underfit the log since it can allow for the replay of any combination of traces that contain the activities {*a, b, c, d, e*}. On the other end, a model that enumerates all the possible traces in the log, such as the one depicted in Figure 8, is said to overfit the log, since it is only able to describe the content of the log from which it was built, without allowing for any other (unseen) behavior.

Shape

Description automatically generated

Figure 8

To assess the quality of the discovered model, four main metrics have been defined, namely fitness, precision, simplicity, and generalization, that we will describe in more detail further in this chapter.

## Challenges of Process Discovery

Process discovery is one of the most difficult tasks of process mining. Three are the main challenges that miners face when trying to discover a model from a log: incompleteness of the dataset, noise and what is called *representational bias*, so the set of the a priori assumptions made about the search space of the models.

For starter, logs only contain partial information about the process that generated them: capturing all the possible behavior would require infinite executions of the process, since only given enough time, a process will display every possible sequence of events. This is especially true for logs coming from real life data, since it is not infrequent for business processes to be abruptly interrupted by humans or faulty equipment, or for executed event not to be correctly recorded.

On a similar note, the problem of the noise is related to the fact that usually logs contain infrequent behavior that is not representative of the main characteristics of a process.

Lastly, as briefly mentioned, there is the problem of representational bias. Miners are designed to extract relevant information from logs by looking for specific patterns and converting them into meaningful structures in the target model; that is to say, they have to make assumptions about the search space of the possible models. An example in which representational bias can occur, is for example when a miner only tries to discover a model by allowing for activities to only appear once in the structure, while another would be making decisions about the way activities are connected based only on local interaction and not looking at the process as a whole.

All miners suffer from such problems. The difference is in the way they deal with them and produce results that are acceptable enough for the industry.

## Alpha relations and footprint

To define the structure of a Petri/Workflow net from the information contained inside event logs, we first need a way to establish an ordering of its transitions. Alpha relations are a set of rules designed to achieve this goal; they work by classifying the relationship between couples of activities into four categories based on their relative position inside an event log. These relations are:

* Direct succession, denoted with x > y. Given two activities *x* and *y*, they are in direct succession if and only if there exist a case inside the log for which *x* is directly followed by *y*.
* Causality, denoted with x 🡪 y. Given two activities *x* and *y*, they are causally related if and only if x > y and not y > x. In other words, if x precedes y for some cases, the inverse should not be true for any other case.
* Parallel, denoted with x//y. Given two activities *x* and *y*, they are parallel if and only if x > y and y > x. This relation describes concurrency inside the log, the situation in which two activities are executed simultaneously; because of that, there is no fixed ordering for termination of the activities, and they can appear in either configuration of the directly follow relation.
* Choice, denoted with x # y. Given two activities *x* and *y*, they are in a choice relation if and only if not x > y and not y > x. In this situation, the two activities cannot be both activated from the same source, but a choice must be taken.

These relations are used by some discovery algorithms as a reference to build the final model, while others only exploit them for preprocessing the output, as we will discuss in this chapter.

The set of all the relations of a log is the *footprint* of the log. If we consider for example the log L = {<a, b, c, d>3, <a, c, b, d>2, <a, e, d>}, we can see that , , , , , and , while all the other possible relations are of type choice. Based on that, the footprint of this log is the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| a |  |  |  |  |  |
| b |  |  |  |  |  |
| c |  |  |  |  |  |
| d |  |  |  |  |  |
| e |  |  |  |  |  |

Where the symbol indicates the specular relation with respect to .

## Metrics

As mentioned above, there are four main criteria that allow practitioners to assess the quality of the discovered models: fitness, precision, simplicity, and generalization. In loose terms, fitness measures how much the behavior observed in a log can be replayed by the model, in terms of percentage of traces correctly executed; precision on the other hand, can be seen as a limitation on the behavior allowed by the model - a precise model would generalize to similar behavior, while a flower model for example could produce any combination of the considered activities; simplicity is an indication of the complexity of the result, usually in terms of the number of connections among activities; generalization indicates how well a model can produce unseen (plausible) behavior[[3]](#footnote-3).

### Fitness

Fitness is usually described by means of one of the following metrics: token-based replay fitness [5] and alignment-based replay fitness [6]. The former is model dependent and tries to execute each trace of the log on the model, forcing the execution by adding tokens when necessary and keeping track of such modifications. The latter is model independent and guarantees that semantically equivalent models will have the same fitness values.

Let us say we want to evaluate a Petri net. To compute the token-based fitness, we need to keep track of how the markings of the net change during the execution, and in particular the number of tokens produced *p*, the number of tokens consumed *c*, the number of missing tokens *m* and the number of remaining tokens *r* after the execution. Every time a transition fires, *p* is incremented by a value equal to the number of places reached by the transition; every time a transition is enabled, *c* is incremented by a value equal to the number of input places for that transition; every time a token needed to carry out the execution is missing, such token is introduced in the net and *m* incremented by one; at the end of the replay, *r* is set equal to the number of tokens remaining inside the net. Then, the metric is computed according to the following equation:

where L denotes the log, N the model being evaluated, denotes a trace in the log and the frequency of such trace.

In the alignment-based approach we try to find, for each trace in the log, the corresponding sequence of activities through the model, going from beginning to end, such that this sequence is as close as possible to the trace (or optimal). For a Petri net, such a sequence of activities matches to a path in the reachability graph of the net, such that the labels of the arcs traversed correspond to the transitions that have fired in the net during the execution. To choose the optimal sequence, a ranking of all the possible sequences constructed on the activities of the trace is computed; this ranking is based on a cost function that can be as simple as the edit distance[[4]](#footnote-4).

For the alignment-based approach to work, *relaxed soundness* [7] is required: for each activity in the model, it should exist at least one firing sequence that connect the starting activity of the process to the end activity of the process.

Given a trace, the optimal sequence is found, as well as the worst possible one, that is the one that differs from the trace in every position. The fitness for that trace is computed as

where L denotes the log, N the model being evaluated, denotes a trace in the log, the frequency of such trace and and are, respectively, the costs of the optimal and worst sequence for each trace.

### Precision

Similarly to what happens for the fitness, precision can be computed using two main approaches, one token-based [8] and one alignment-based [9], and their inner workings are like those described above. The idea behind the two approaches, however, is similar: for each trace, the different prefixes are replayed on the model, whenever possible. At each reached marking (or position in the alignment), the set of transitions that are enabled in the process model is compared to the set of activities that follow the prefix: the more the sets are alike, the higher is the precision.

### Simplicity

Simplicity tries to describe the complexity of the solution in terms of the arc degree of the nodes in the model [10]. If we consider a Petri net, we compute the average degree for each place and transition of the net, defined as the sum of the output and input arcs of each node divided by the number of nodes. Then the simplicity of the model is given by

where N is the model considered and k a parameter decided by the user such that .

### Generalization

Generalization measures how well the model can describe the unseen behavior of a process, and not only the one displayed in the event log [11]. It is based on how often the different parts of the model are used during the replay of the log: if some parts are infrequently visited, generalization is bad, while if all parts are frequently used generalization is likely to be high. If we imagine the model being a process tree, then the formula to compute the generalization if the following:

## State of the Art

In this section we will introduce some of the most famous miner used in commercial tools, the algorithmic miners, and the state of the art for what concerns the process discovery method based on deep learning.

### Algorithmic Miners

#### Alpha algorithm

The Alpha algorithm was, historically, the first miner to be able to discover the concurrency of activities inside a process. It works by taking in input an event log, and after constructing the footprint, it directly exploits the alpha relations among activities to build a Petri net; consequently, the produced model generates the same footprint of the log.

The first step is to extracts all the activity names from the log and use them to construct the set T of all the transitions of the model. Once T is available, it proceeds to define the sets TI and TO of starting and ending activities.

The second and most important step is to construct the places of the model. At this stage, the places are defined as pairs of sets of transitions (A, B) such that, for every transition ta in A there exist a transition tb in B that directly follows ta, and all the transitions in A and B respectively are in choice relation among each other. In symbols:

Where XL is the set of all such pairs. This set is then filtered by keeping only the pair of sets that are maximal, so the pairs such that by removing any element of A or B, the condition stated before no longer holds; these remaining pairs are the final places of the model. For each place, A contains the incoming transitions, while B the outgoing transitions, so the model is built accordingly.

|  |  |
| --- | --- |
| Diagram  Description automatically generated  Figure 9 | A picture containing text, sport  Description automatically generated  Figure 10 |

In its basic form, this algorithm has many limitations:

* It is unable to detect loops of length ≤ 2. If not taken care of in pre or post processing, this leads to unconnected transitions that, according to the semantics of Petri nets, are always enabled.
* It can generate implicit places: places that connect transitions that if removed do not alter the behavior of the net. This problem can be easily solved in post processing.
* By rigidly following the alpha relations, it is unable to detect nonlocal dependencies. Alpha relations are based on the directly follows relation, so if in the real process there are two activities are such that one eventually follows the other (meaning they are at distance ≥ 1), this dependency is not identified by the Alpha algorithm.
* It cannot identify free-choice constructs, such as those in Figure 9 and Figure 10.
* It suffers from representational bias: the assumption of unique activities (so no transitions with the same label are allowed) and the absence of OR-split and joins (that cannot be detected by the alpha relations). Both these problems could be solved with the introduction of silent transitions.
* The resulting model is not guaranteed to be sound.
* It is not robust to noise (infrequent traces or activities), so preprocessing may be needed.

Besides its limitations, the Alpha algorithm represents a good baseline when doing process discovery.

#### Heuristics Miner

The Heuristic Miner [12] works in two phases: at first it learns a dependency graph from the input log; then it transforms the graph into a causal net by learning the appropriate bindings by means of some predefined heuristics.

The dependency graph is built according to two indicators: the direct succession , and the dependency measure , defined as follows for two generic activities *a* and *b* in the log L:

where denotes a trace in the log, the frequency of such trace and the i-th activity of trace . As we can see, the dependency measure is based on the direct succession, and it can be observed that , with a value of meaning strong causality among the two activities. These two measures are used to prune the original dependency graph constructed directly from the log, by keeping only the arcs for which and are above a certain predefined threshold.

Once the dependency graph is available, the bindings are learnt by means of some heuristics (e.g., use a time window around each activity to keep only the most frequent activities among the one closest in time in the log) or optimization criteria (e.g., define a function to rank the possible combination of bindings and keep the best ones) that depend on the version of the miner.

The resulting causal net can be converted to a Petri net, but in this case the Petri net is not guaranteed to be sound. Beside this problem, since the two indicators used by the algorithm in the first phase are based on the direct succession relation, the Heuristic miner suffers from some of the limitations that afflict the Alpha miner, for example the inability to detect nonlocal dependencies among activities. When used on large datasets, this miner also frequently leads to spaghetti models, which are difficult to interpret.

#### Region-Based Miner

The Region-Based miner, as the Heuristic one, works in two steps: it first learns an intermediate representation of the model and then based on this intermediate representation it builds the final model. In this case the intermediate representation is a transition system, while the final model a Petri net.

The most crucial decision in the construction of the transition system regards the way the states are defined: they can embed information about the frequency and the order of the activities in the log, for example by means of a time window around each activity (that can be symmetric or focused just on the past or future interactions of the activity) or, in a more abstract way, consider the states as multisets of activities, each one reported with the frequency with which it appears inside a particular trace or the whole log. This decision has an important impact on the final model: if the states include information about the activities’ order, for example, the resulting model could overfit, while more abstract representations help mitigate the problem.

Once the transitions system is defined, the algorithm proceeds to identify regions inside it. A region, in this context, is defined as a set of states of the transition system, such that if a transition exists the region, then all equally labeled transitions exit the region, and if a transition enters the region, all equally labeled transitions enter the region[[5]](#footnote-5). In particular, the algorithm is interested in finding all the non-trivial minimal regions inside the transition system. A trivial region is a region that contains all the states of the transition system or that contains none, while a minimal region is a region that cannot be decomposed into smaller, non-trivial ones.

The set of all the minimal, non-trivial regions constitutes the set of all the places in the Petri net; the set of all the transition labels in the transition system instead defines the set of all the transitions of the Petri net. Once places and transitions are defined the arcs connecting them are generated.

As already mentioned, this approach can produce overfitting models, depending on the definition of state for the transition system, but is able to discover complex process patterns that are undetected by the miners previously described. Other limitations regard the number of minimal non-trivial regions that can be found: if none is found, the result is an unconnected Petri net, in which all transitions are always enabled. In general, a transition in the resulting Petri net can be disconnected from the rest of the net.

#### Inductive Miner

The inductive miner [13] is one of the best performing algorithmic miners available, being able to guarantee the soundness of the output model. It works by iteratively partitioning the directly follows graph into subsets of activities and using these subsets to build a process tree.

The criteria followed to partition the directly follows graph are closely related to the definition of the four possible operators in a process tree:

* The *sequence cut* partitions the directly follows graph into subsets of nodes such that every activity in one subset follows every activity of another subset based on the direct succession relation. The subsets generated are linked by the *sequence* operator.
* The *exclusive choice* cut generates partitions that are completely disconnected (the activity in each partition do not interact with the activities of any other partition). The subsets generated are linked by the *choice* operator.
* The *parallel cut* generates partitions such that the activities contained in each subset can follow each other in any order (e.g., if two subsets A and B are found, it is possible for any activity inside A to follow and be followed by any activity inside B). The subsets generated are linked by the *parallel* operator.
* The *redo-loop cut* finds a partition in which its subsets are such that the execution of the activities inside some of the subset is conditioned on the execution of the activities of other subsets. For example, let us consider three subsets A, B and C that are found by this cut. If the execution of any activity in B or C is conditioned on the execution of A, then A represent the condition of the loop, while B and C the body of the loop. The subsets generated are linked by the *loop* operator.

|  |  |
| --- | --- |
| Diagram  Description automatically generated  Figure 11 | Diagram, engineering drawing  Description automatically generated  Figure 12 |

In Figure 11 it is possible to see an example directly follows graph, and Figure 12 the resulting process tree found by the miner. In this example is easy to see how the sets {*a}* {*b, c, f, e*} and {*d*} can be individuated using the *sequence cut*. The same cut can be applied to the subset {*e, f*}, while the subset {*b, c*} can be partitioned using the *exclusive choice* cut. Finally, subsets {*b, c*} and {*e, f*} can be individuated using the *redo-loop* cut.

The resulting process tree can be translated into a sound Petri net, by introducing silent transitions when needed.

Limitations of this miner are mainly due to its representation bias: the leaves must have a unique label (so there cannot be repetitions of activities in different areas of the resulting model) and the process must be described with a block-structured representation. Also, fitness and precision can be increased by increasing the number of silent transitions inside the final model, that has the side effect of reducing the simplicity of the model.

### Deep learning-based miner

So far, state-of-the-art techniques have approached the problem of process discovery as an unsupervised learning problem, through graph synthesis algorithms such as those discussed above. This approach has the downside of introducing biases in the resulting models, since each algorithm tries to find a solution by means of different design criteria and/or heuristics [14]. Recently, to overcome this issue, researchers have tried to switch to a more human-like approach by leveraging the ability of Neural Networks to learn patterns through data alone. The most famous work in this area is the one by D. Sommers [15], who formulated the problem as a supervised learning problem: a deep learning-based algorithm that tries to learn how to build process models from examples of (synthetic) event logs and corresponding template models (search space) and then transfers this knowledge to build models for real-life unseen logs. This algorithm takes in input a graph that encodes the traces of a given event log and the template of a Petri net (more on that later), it elaborates this input by means of subsequent passes through graph neural networks and shallow multilayer perceptrons, and outputs a Petri net which is a subgraph of the one encoded in the input.

#### Data

The first step is to encode the process discovery problem as a graph. This graph is a directed multigraph composed of two parts: a bidirectional graph that encodes the traces of the event log, called the trace graph, and the template of a Petri net, built according to the alpha relations discovered in the log and augmented with a subset of all the possible places that connect the transitions.

Shape, box and whisker chart

Description automatically generated

Figure 13

The trace graph, of which we can see an example in Figure 13, consists of different chains of nodes, one for each variant in the event log, each node representing an activity of the process (except the initial and final nodes, that represent the fictitious start and end activities). These chains are linked to two artificial nodes marking the start and the end of the traces. The features of each node are initialized with a one-dimensional tensor holding the one-hot encoding for that node in the first n-1 places, and the frequency of the trace in the nth place.

The Petri net template, instead, is built starting from the alpha relations discovered in the event log. The transitions of this net consist in the set of the process activities, to which are added the transitions corresponding to an artificial start and end activities. Places are introduced according to the following criteria:

1. The places associated with the directly follows relations are added. These places are referred to as one-to-one places.
2. Places with a single incoming transition and multiple outgoing transitions are added. These correspond to the places which connect activities that are directly followed by more than one activity. Let us consider for example an event log in which activity *a* is directly followed by activities *b*, *c*, and *d*, and there is no parallel relation among *b*, *c* and *d*. In this case, the places that will be added to the Petri net are 4 and correspond to all the possible combinations of the three activities (so the 4 possible output configurations for activity *a*): {*b*, *c*}, {*b*, *d*}, {*c*, *d*}, {*b*, *c*, *d*}. [figure x] shows an example of this. In the eventuality in which two or more of the following activities are parallel to each other, the places that express the combinations in which such activities are present are discarded (so, continuing the previous example, if *c*//*d*, the only places we add are {*b*, *c*} and {*b*, *d*}). A similar methodology is applied to add all places that have more than one incoming transition and only one outgoing transition. These kinds of places are respectively referred to as one-to-many places and many-to-one places.
3. In the last step, many-to-many places are introduced, so places that have multiple common incoming and outgoing transitions. The criteria followed to create them is like the one in step three but combines the two ideas.

The features associated to the places are initialized to zero. Transitions and places are then connected with directed arcs that go in both directions, allowing for the information to flow in the graph without any limitations (e.g., a preferred direction).

Once the two subgraphs are constructed, they are joined by means of directed arcs that connect the nodes of the trace graph to the corresponding transitions of the template Petri net with the same name.

#### Approach

Given the graph obtained during preprocessing, the goal is to learn to select the subset of places that restricts the transitions of the template petri net to the behavioral patterns present in the log (sequence, parallel and so on). This is achieved by estimating the likelihood that a specific place , where P is the set of all the places in the template Petri net, is best suited to be included into the result given that places have already been selected by the miner, i.e., . The actual likelihood, however, is dependent on the ordering π in which the nodes are considered, and so the joint probability that the places in are selected is defined as with ordering π. Since the order is not relevant to the solution, we can use the marginal joint probability to model the probability over all permutations of : . can then be used to estimate the best subset of places in the template net.

Since it is not actually feasible to compute the marginal joint probability over all the possible permutations of , the actual problem of the computation is addressed in two different ways at training and inference time. During training, teacher forcing is used to impose a breadth first order on the miner, that gives a lower bound on the actual marginal joint probability. In this way, the model learns the joint distribution , with π being the breadth first ordering of the places in the template Petri net. The miner is then trained by maximizing the expected joint log likelihood , with and from dataset , where and are respectively a log and the corresponding input graph constructed as discussed above. With a sufficiently large and varying dataset, approaches , which is the true probability distribution of the correct selection of places from *P* given an event log L.

During inference, since teacher forcing is not available, beam search is applied to keep track of multiple candidate sequences of places and mitigate the effect that suboptimal choices in the beginning can have on the final sequence. Then the most promising one(s) is (are) considered as the result(s) of the discovery.

#### Structure of the miner

The functioning of this miner is possible thanks to its structure. It is composed of four different neural networks: two graph neural networks called *propagation networks* and two shallow multilayer perceptrons, the *select candidate network* and the *stop network*. The details on how a graph neural network and a multilayer perceptron are structured will be given in the following chapter: for now, it is sufficient to consider them as black boxes that, given an input, are able to learn patterns and produce an output.

The first propagation network takes in input one of the graphs constructed during the preprocessing phase and returns an initial embedding of its nodes. After that, the core of the miner is executed: a do-while cycle controlled by the *stop network* evaluates, one at a time, a candidate place among those in *P*, and the decision on whether to include it in the final result is made by the *select candidate network*. Before the condition of the cycle is evaluated, the embeddings of the nodes are further preprocessed by the second *propagation network* that produces new embeddings that contain the information about the decisions made so far by the *select candidate network*. When the *stops network* ends the cycle, the template Petri net is extracted from the processed graph and returned to the user.

#### Results

When compared with other algorithmic miners[[6]](#footnote-6), this miner has demonstrated to be able to reach similar results to the state of the art. It is not guaranteed to generate sound models, not even relaxed sound, but its result yield in general to high precision, generalization, and simplicity (where it outperforms the state of the art), but in general a lower fitness. When tested on synthetic data, it generates models that are closer to the ground truth than those produced by the other miners, thus outperforming them; on real data, the results are comparable to the state of the art.

# Bibliography

|  |  |
| --- | --- |
| [1] | W. v. d. Aalst, Process Mining: Data Science in Action, Springer, 2016. |
| [2] | W. v. d. Aalst, "Business Process Management: A Comprehensive Survey," 2012. |
| [3] | M. Tadao, "Petri Nets: Properties, Analysis and Applications," 1989. |
| [4] | W. v. d. Allst, "Causal Nets: A Modeling Language Tailored towards Process Discovery," 2011. |
| [5] | A. Berti and W. v. d. Aalst, "Reviving Token-based Replay: Increasing Speed While Improving Diagnostics," in *ATAED@ Petri Nets/ACSD*, 2019. |
| [6] | B. v. Dongen, J. Carmona and T. Chatain, "Alignment-based Metrics in Conformance Checking". |
| [7] | J. Dehnert and P. Rittgen, "Relaxed Soundness of Business Processes," 2001. |
| [8] | J. Muñoz and G. J. Carmona, "A fresh look at precision in process conformance," in *International Conference on Business Process Management*, Berlin, Heidelberg, 2010. |
| [9] | A. Adriansyah and e. al, "Measuring precision of modeled behavior," *nformation systems and e-Business Management 13.1,* 2015. |
| [10] | F. R. Blum, "Metrics in process discovery," Computer Science Department, University of Chile, 2015. |
| [11] | J. C. Buijs, B. F. v. Dongen and W. v. d. Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *International Journal of Cooperative Information Systems 23.01,* 2014. |
| [12] | A. Weijters, W. v. d. Aalst and A. K. A. D. Medeiros, "Process mining with the HeuristicsMiner algorithm," 2006. |
| [13] | S. Leemans, D. Fahland and W. v. d. Aalst, "Discovering Block-Structured Process Models From Event Logs - A Constructive Approach". |
| [14] | A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. Maggi, A. Marrella, M. Mecella and A. Soo, "Automated Discovery of Process Models from Event Logs: Review and Benchmark," *IEEE Transaction son Knowledge and Data Engineering,* vol. 31, 2019. |
| [15] | V. M. D. F. Dominuque Sommers, "Process Discovery Using Graph Neural Networks," 2021. |

# List of Figures

[Figure 1 10](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc111733284)

[Figure 2 12](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc111733285)

[Figure 3 13](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc111733286)

[Figure 4 13](file:///C:\Users\matte\Documents\GitHub\GLPD\documents\thesis_10499057.docx#_Toc111733287)

[Figure 5 15](#_Toc111733288)

[Figure 6 15](#_Toc111733289)

1. The discipline that focuses on process modeling and the implementation of processes (not necessarily in a data driven way). [↑](#footnote-ref-1)
2. Such markings are called dead markings. [↑](#footnote-ref-2)
3. The description given are based on the resources found on the website of the process mining framework called PM4PY (<https://pm4py.fit.fraunhofer.de/>), since this is the framework used to assess the quality of the project which is the main content of this document. [↑](#footnote-ref-3)
4. In computer science, it is a measure of the dissimilarity of two strings, computed by counting the minimum number of operations required to transform one string into the other. [↑](#footnote-ref-4)
5. Remember that in a transition system, the arcs connecting the states are labeled with the name of the activity that, if executed, allows to transit from a state to another. [↑](#footnote-ref-5)
6. Split miner, Heuristic miner, ILP miner and Inductive miner. [↑](#footnote-ref-6)