ASM Sc. J., 19, 2024

https://doi.org/10.32802/asmscj.2023.1447

Towards the Implementation of Heuristics Miner in .Net Framework

Z. Lamghari[1,2*]

[1]Department of Computer Science, Faculty of Sciences, Mohammed V University, Rabat, 10000, Morocco

[2]Laboratory of Sciences, Engineering, and Management (LSEM), High School of Technology (EST), Sidi Mohamed Ben Abdellah University, Immouzer Road, Fez, 2427, Morocco

Process mining is an interdisciplinary field that bridges the gap between data mining and business process analysis. It revolves around the core concept of leveraging recorded event logs to automate the generation, validation, and refinement of process model. A majority of currently accessible process mining solutions are distributed in form of black box systems - software with no openly accessible source code - or as libraries for scripting languages like Python or R. In this sense, it is necessary to design and build an understandable foundation of a process mining library with the intention of presenting the possibility of incorporating process mining into many programs written in the language or aiding other programmers interested in the potential of such technologies. Therefore, this paper describes and demonstrates the usability of an advanced algorithm for process mining in the .NET platform, namely, the Heuristic miner for process modelling.

Keywords: process mining; process discovery; heuristics miner algorithm; C# library; .NET Core 6

I. INTRODUCTION

Process mining is a scientific discipline that falls between data mining and business process analysis (Van der Aalst, 2016). The main idea of process mining is to use the execution BPs event logs that are recorded in the information system to automatically generate, check and enhance process models.

Moreover, process mining consists of three types, which are discovery, conformance, and enhancement (Coutinho-Almeida & Cruz-Correia, 2022). Discovery: An automatic process modelling methodology that takes event logs as input and produces a BP model as output. Conformance: compares the newly discovered process model with the existing process model. The purpose is to identify bottlenecks and discover discrepancies. Enhancement: focuses on improving or extending the existing process model using the information stored in event logs. In this context, the discovery technique has always been a major topic in process mining research.

Furthermore, there are several process mining tools. The majority of currently available process mining solutions are distributed as black box systems – software with no openly accessible source code – or as libraries for scripting languages like Python (Waibel, 2022) or R (Janssenswillen *et al.*, 2019), which are generally regarded as unsuitable for more complex software projects.

The objective of this paper, as well as the associated implementation, is to design and build a solid, easily extensible, and understandable foundation for a process mining library in C#, a popular, modern, coherent, and widely used, primarily object-oriented programming language, with the intention of not only presenting the possibility of incorporating process mining into many programs written in the language or assisting other programmers interested in the potential of such technology.

This paper, in particular, describes a discovery algorithm for a library written in the C# programming language and built in the. NET Core 6 framework. The reader is first introduced to the fundamentals of process mining. Just after that, definitions and implementations of commonly used models are introduced, followed by an explanation of the

*Corresponding author's e-mail: zineb_lamghari2@um5.ac.ma heuristic miner algorithm. This algorithm is also demonstrated in the paper. The library is accessible via the GitHub 1platform.

Upcoming sections of this paper are organised as follows: Section 2 presents the theoretical background related to our study field. Section 3 explains the heuristics miner algorithm. This section will lead us to understand the process mining library in terms of components and logical links. Section 4 illustrates the implementation of the heuristics' miner algorithm in the .NET framework. The conclusion is mentioned in section 5.

II. THEORETICAL BACKGROUND

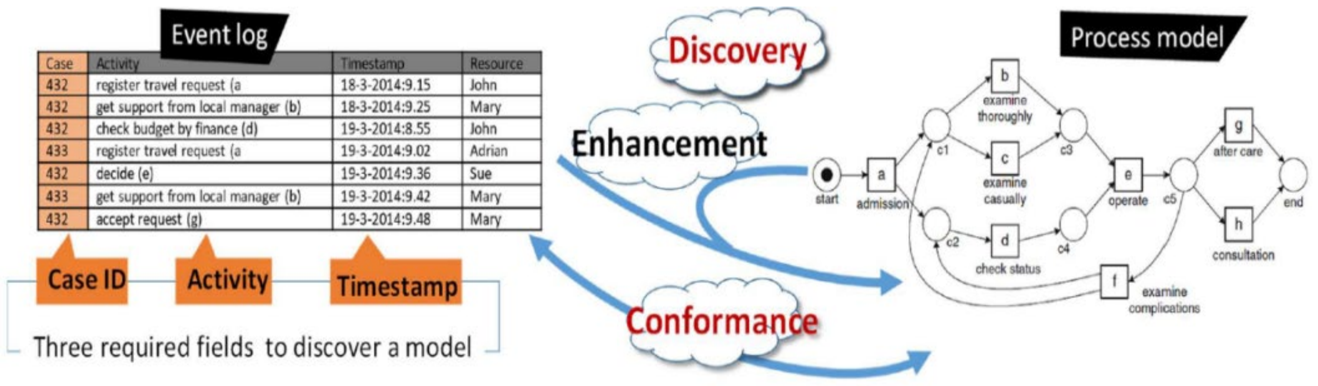
In this section, we present terminologies used throughout our paper. In this sense, we define the following concepts and notations: Event logs, Basic relations, Dependency measures, Dependency graph, Causal Net and Petri Net.

For this purpose, this section content will be organised as follows: Definition 1 (Event logs), Definition 2 (Basic relations), Definition 3 (Dependency measures), Definition 4 (Dependency graph), Definition 5 (Causal Net) and Definition 6 (Petri Net).

A. Event Logs

Process Mining typically assumes that BP (Business Process) execution data are stored as event logs (Van der Aalst, 2016; Coutinho-Almeida & Cruz-Correia, 2022). An event can be considered as the starting point of process mining. The event log structure is illustrated in Figure 1, where the process is made of either cases or finished process instances. Each case is composed of a series of occurrences known as a trace. Depending on the organisation's needs, an event can include any number of extra properties (timestamps, costs, resources, etc.). These extra characteristics are critical for tracking BP improvement. For example, bottlenecks cause can slow down the process flow.

The event logs notation may depend on the information system treatment or objectives. However, the main objective is the quality of these events that can heavily affect the process model representation and, by necessity the main business of the organisation. Therefore, event logs should be treated as first-class.



Case	Activity	Timestamp	Resource
432	register travel request (a	18-3-2014:9.15	John
432	get support from local manager (b)	18-3-2014:9.25	Mary
432	check budget by finance (d)	19-3-2014:8.55	John
433	register travel request (a	19-3-2014:9.02	Adrian
432	decide (e)	19-3-2014:9.36	Sue
433	get support from local manager (b)	19-3-2014:9.42	Mary
432	accept request (g)	19-3-2014:9.48	Mary

Figure 1. Process mining overview (El-Gharib & Amyot, 2019).

Definition 1 (Event log).

Let T be a set of activities,

$\zeta \in T^*$ is an event trace, i.e., sequence of activity identifiers,

$L \subseteq T^*$ is an event log, i.e., a multiset of event traces.

To illustrate the basic concepts used in the following chapters, we use the event log:

[1] https://github.com/lasaris/ProcessM.NET

$$L_1 = [\langle a,c,d,b\rangle_{10}, \langle a,d,c,b\rangle_{10}, \langle a,e,b\rangle_{10}, \langle a,b\rangle_5, \langle a,e,e,b\rangle_2, \langle a,c,b\rangle_1, \langle a,d,b\rangle_1, \langle a,e,e,e,b\rangle_1] \quad (1)$$

The numbers above the traces indicate how many times the trace has occurred (see Equation 1). The event log L1 contains 40 traces, three of which are infrequent and cause noise in the event log (rare behaviour that does not represent typical process behaviour).

B. Basic Relations

We must analyse basic relations (Yulion *et al.*, 2022) if we are to discover a process model based on the event log. Within the scope of this paper, we use three types of basic relations. In (Yulion *et al.*, 2022), the authors provide a standard description of the basic relations (see Equation 2).

Definition 2 (Basic relations)

Let L be an event log over T,

$a, b \in$T:

$a >_\omega b$, if there is a trace $\sigma = t_1t_2t_3 \dots t_n$ and $i \in \{1, \dots, n-1\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$,

$a >>_\omega b$, if there is a trace $\sigma = t_1t_2t_3 \dots t_n$ and $i \in \{1, \dots, n-2\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$, $t_{i+2} = a$ and $a \neq b$,

$a >>>_\omega b$, if there is a trace $\sigma = t_1t_2t_3 \dots t_n$ and $i < j$ and $i, j \in \{1, \dots, n\}$ such that $\sigma \in L$ and $t_i = a$ and $t_i = b$ (until next appearance of a or b) (2)

The first relation $>_\omega$ specifies which activities occur in chronological order, i.e., one activity immediately follows the other. The second relation $>>_\omega$ describes activities that take place in two-length loops. The final relation $>>>_\omega$ refers to direct or indirect successors.

C. Dependency Measures

Since the frequency of successors does not indicate the likelihood of succession, we must define dependency measures (Yulion *et al.*, 2022). As a result, the dependency relationship between the two activities (notation a ⇒w b) can be stated. There are three types of measures once again. In (Yulion *et al.*, 2022), the authors provide a standard description of the dependency measures (see Equation 3).

The dependency measures are implemented in three matrices by Class DependencyMatrix. The DirectDependency'Matrix stores the first dependency measure. The second dependency measure is stored in the LILDependencyMatrix, which is a one-dimensional matrix, and the third dependency measure is stored in the LILDependencyMatrix.

Definition 3 (Dependency measures)

Let L be an event log over T, a, $b \in T$,

$|a >_\omega b|$ is the number of times $a >_\omega b$ occurs in L,

$|a >>_\omega b|$ is the number of times $a >>_\omega b$ occurs in L.

$$a \Rightarrow_\omega b = \left(\frac{|a >_\omega b| - |b >_\omega a|}{|a >_\omega b| + |b >_\omega a| + 1} \right) \text{ if } (a \neq b)$$

$$a \Rightarrow_\omega a = \left(\frac{|a >_\omega a|}{|a >_\omega a| + 1} \right) \text{ if } (a = b)$$

$$a \Rightarrow_\omega^2 b = \left(\frac{|a >>_\omega b| - |b >>_\omega a|}{|a >>_\omega b| + |b >>_\omega a| + 1} \right) \quad (3)$$

The dependency measure's value is always between -1 and 1. The higher the value, the stronger the dependency link between activities. These relationships can be represented using the dependency graph. In (Yulion *et al.*, 2022), the authors provide a standard description of the dependency graph (see Equation 4).

D. Dependency Graph

Definition 4 (Causal Net)

Let L be an event log over T,

σ_a the absolute dependency threshold (90%),

σ_{L1L} the length one loop threshold (90%),

σ_{L2L} the length two loop threshold (90%),

σ_r the relative to the performed threshold (5%),

act the all-tasks-connected heuristic.

The dependency graph DG is defined as follows:

$A = \{t | \exists_{\sigma \in L} [t \in \sigma]\}$ (The set of tasks appearing in the log),

$C_1 = \{(a, a) \in A \times A | (a \Rightarrow_\omega a \geq \sigma_{L1L})\}$ (Length-one loops),

$C_2 = \{(a,b) \in A \times A | (a,a) \notin C_1 \wedge (b,b) \notin C_1 \wedge \mathrm{a} \Rightarrow^2_\omega b \geq \sigma_{L2L}\}$ (Length-two loops),

$C_{out} = \{(a,b) \in A \times A | act \wedge \mathrm{b} \neq \mathrm{end} \wedge \mathrm{a} \neq \mathrm{b} \ \wedge \forall_{y \in A}[a \Rightarrow_\omega b \geq a \Rightarrow_\omega y]\}$ (Each task's strongest follower),

$C_{in} = \{(a,b) \in A \times A | act \wedge \mathrm{b} \neq \mathrm{start} \wedge \mathrm{a} \neq \mathrm{b} \ \wedge \forall_{y \in A}[a \Rightarrow_\omega b \geq y \Rightarrow_\omega b]\}$ (Each task's strongest cause),

$C'_{out} = \{(a,x) \in C_{out} | act \wedge \mathrm{a} \neq \mathrm{x} < \sigma_\omega \wedge \ \exists_{(b,y) \in C_{out}}[(a,b) \in C_2 \wedge \mathrm{b} \Rightarrow_\omega y - a \Rightarrow_\omega x > \sigma_r]\}$ (non-necessary dependencies),

$C_{out} = C_{out} - C'_{out}$ (Only one dependent task is necessary for a length-two loop),

$C'_{in} = \{(a,x) \in C_{in} | act \wedge \mathrm{a} \Rightarrow_\omega \mathrm{x} < \sigma_a \wedge \ \exists_{(b,y) \in C_{in}}[(x,y) \in C_2 \wedge \mathrm{b} \Rightarrow_\omega y - a \Rightarrow_\omega x > \sigma_r]\}$ (non-necessary dependencies),

$C_{in} = C_{in} - C'_{in}$ (Only one cause task is necessary for a length-two loop),

$$\begin{aligned} C''_{out} = \{(a,b) \in A \times A| \big(act \wedge a \Rightarrow_\omega b \\ \geq \sigma_a \\ \wedge \exists_{(a,y) \in C_{out}} [a \Rightarrow_\omega y - a \Rightarrow_\omega b \leq \sigma_r]\big) \\ \vee (\neg act \wedge a \Rightarrow_\omega b \geq \sigma_a)\} \end{aligned}$$

$$\begin{aligned} C''_{in} = \{(a,b) \in A \times A| \big(act \wedge a \Rightarrow_\omega b \\ \geq \sigma_a \wedge \exists_{(y,b) \in C_{in}} [y \Rightarrow_\omega b - a \Rightarrow_\omega b \leq \sigma_r]\big) \\ \vee (\neg act \wedge a \Rightarrow_\omega b \geq \sigma_a)\} \end{aligned}$$

$$DG = C_1 \cup C_2 \cup C''_{out} \cup C''_{in} \qquad (4)$$

E. Causal Net

The causal net is a graph in which nodes represent activities and arcs represent causal relationships. Each node has its own set of input and output bindings. Binding is a collection of activities in the AND relationship. If an activity has multiple bindings pointing in the same direction, they are in an XOR relationship. The causal net has distinct beginning and ending activities. They also keep track of how many times each activity and binding were visited during the log's replay. In (Van der Aalst *et al.*, 2011), the authors provide a standard description of the causal Net (see Equation 5).

Definition 5 (Causal Net)

Let L be an event log over T,

A causal net (C-net) is a tuple C = (A, ai, ao, D, I, O) (5)

where:

$A \subseteq T$ is finite set of activities,

$a_i \epsilon A$ is start activity,

$a_0 \epsilon A$ is end activity,

$D \subseteq A$ x A is dependency relation (see equation 4),

$AS = \{X \subseteq P(A) \mid X = \{\emptyset\} \vee \emptyset \notin X\}^2$,

$I \epsilon A \longrightarrow AS$ defines the set of input bindings per activity,

$O \epsilon A \longrightarrow AS$ defines the set of output bindings per activity.

F. Petri Net

Petri Nets (2015) is a graphical language that is used to illustrate a process. A Petri Net, in particular, is a bipartite network with two types of nodes: transitions and places. In (Raffety *et al.*, 2022), the authors provide a standard description of the Petri Net (see Equation 6).

Definition 6 (Petri Net)

A Petri net is a triplet $N = (P, T, F)$ (6)

where:

P is a finite set of places,

T is a finite set of transitions such that $P \cap T = \emptyset \wedge F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs (flow relation).

A marked Petri net is a pair (N, M), where $N = (P, T, F)$ is a Petri net, and where $M \in \beta(P)$ is a multi-set over P defining the marking of the net. The set of all marked Petri nets is expressed N.

III. HEURISTICS MINER

The Heuristic Miner (HM) algorithm was developed by Weijters, Ribeiro in (Yulion *et al.*, 2022). It employs a heuristic approach to resolve problems encountered by the algorithm. The general idea behind this algorithm aims at recognising the sets of relations in the event logs and then generate a process model based on those relations. The algorithm differs from the heuristic miner in that the latter uses statistical measures to determine the relationships between activities. The algorithm is split into three steps. It first generates a dependency graph by counting all direct successions in the logs (the frequencies of the successions are then stored in a matrix). It then computes the dependency rate between each activity in order to retain only those with a significant causal relationship. The discovered causality graph (Van der Aalst *et al.*, 2011) will be developed after a dependency rate and succession frequency threshold is set. The second step is to use a heuristic approach to identify divergences and synchronisations. Finally, if necessary, the causality graph may be transformed into a Petri net.

The Heuristic miner is a commonly used mining algorithm that can deal with noise and can be used to express the primary behaviour registered in an event log (Silva & Mira Da Silva, 2022). The process model is discovered by the Heuristic miner, which describes the control-flow perspective of the process captured in the event log. When constructing a process model, this algorithm considers the frequencies and sequences of the events. This enables us to exclude unusual behaviour from the discovered model. Benchmark studies have demonstrated its worth, displaying the ability to discover high-quality models (Yulion *et al.*, 2022). The algorithm's input is an event log with one initial and one final activity. This can be accomplished by preprocessing the event log.

The establishment of the dependency graph is the starting point for the Heuristic miner. The first step is to construct a matrix of basic relationships.

For example, Table 1 presents the number of times the basic relations occurred in the event log L_1 (see equation 7).

$$(L_1 = \begin{bmatrix} \langle a,c,d,b\rangle_{10}, \langle a,d,c,b\rangle_{10}, \langle a,e,b\rangle_{10}, \langle a,b\rangle_{5}, \langle a,e,e,b\rangle_{2}, \\ \langle a,c,b\rangle_{1}, \langle a,d,b\rangle_{1}, \langle a,e,e,e,b\rangle_{1} \end{bmatrix}) \quad (7)$$

The relation a >w b is on the left, and the relation a >>>w b is on the right side. The matrix of relation a >>w b is not shown, because it contains only zeros (L1 does not contain the length of two loops). As an example, $|a >_{\omega} c| = 11$, i.e., c is followed by 11 times in the event log L_1 (10 times in $\langle a,c,d,b\rangle_{10}$ and once in $\langle a,c,b\rangle_{1}$).

Table 1. Matrices of basic relations.

	a	c	d	b	e
a	0	11	11	11	5
c	0	0	10	11	0
d	0	10	0	11	0
b	0	0	0	0	0
e	0	0	0	13	4

	a	c	d	b	e
a	0	21	21	40	13
c	0	0	10	21	0
d	0	0	0	21	0
b	0	0	0	0	0
e	0	0	0	0	0

The next step is to compute the activity dependency measures, as defined in section 2.3. The dependency measures are once again stored as matrices. We can omit the dependency measure $a \Rightarrow^{2}_{w} b$.

Table 2 shows the dependency measures. The values in the lower triangular part of the matrix are the inverses of the values in the upper triangular part ($a \Rightarrow_{\omega} c$ and $c \Rightarrow_{\omega} a$). As a result, we can skip the lower triangle computation by simply using the negative value of the upper triangle.

Table 2. Dependency measures.

	a	c	d	b	e
a	0	92%	92%	83%	93%
c	-92%	0	0	92%	0
d	-92%	0	0	92%	0
b	-83%	-92%	-92%	0	-93%
e	-93%	0	0	93%	80%

The next objective is to make the dependency graph. Dependency measurements can be used in two ways: essentially (without all-tasks-connected heuristics) and in conjunction with all-tasks-joined heuristics. When we use the direct approach, we only observe the length-one loop, length-two loops, and absolute dependency thresholds. The default value for these thresholds is usually 0.9. These thresholds indicate that we accept dependency relationships between activities with dependency measures greater than or equal to the threshold. If we use the all-tasks-connected approach, we must first create a model of the best candidates (the strongest input and output relations). Then, we handle the other relations using the relative to the best threshold. We also approve activities with a dependency measure greater than the absolute dependency threshold and a dependency measure close to the best candidate. The dependency graph is created in 12 steps. In steps 4 through 9, the all-tasks-connected heuristic is used. In steps 10 through 12, the process model is expanded with additional reliable arcs (Yulion *et al.*, 2022).

Thus, the DG is generated using the default settings and all-tasks-connected heuristic from Table 2. By changing the settings, we can get different dependency graphs. For example, if we use a length-one loop threshold of 0.7, we can get specific representation of the DG (see Figure 2). The dependency graph also supports a self-loop on activity e. You may notice that activities a and b are missing one set of activities. This is due to the fact that a is the initial activity and b is the final activity.

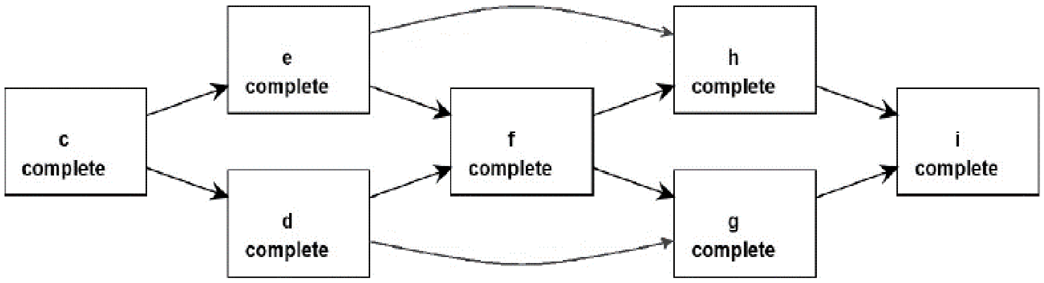


Figure 2. Dependency Graph illustration.

Mining long-distance dependencies is optional and is determined by the settings. It identifies relations that have not yet been included in the dependency graph.

A new frequency-based metric has been defined to address this issue. A long-distance dependency measure considers direct or indirect successors. The fundamental concept is to detect pairs of tasks with comparable frequency, where the second activity follows the first either directly or indirectly. The formal definition of the long dependency measure is cited in (Yulion *et al.*, 2022).

To obtain the long-distance dependency measures, we must apply this operation (see equation 8):

$$a \Rightarrow_{w}^{1} b = \left(\frac{2 * |a >>>_{\omega} b|}{|a|+|b|+1} - \frac{abs(|a|-|b|)}{|a|+|b|+1}\right) \tag{8}$$

where L is an event log over $T, a, b \in T$,

$|a >>>_{\omega} b|$ is the number of times $a >>>_{\omega} b$ b occurs in L,

$|a|$ is the number of times a occurs in L.

$|b|$ is the number of times a occurs in L.

The final step is to transform the dependency graph into a causal net. To accomplish this, we must mine input and output bindings for each activity. We replay the event log and count each unique pattern after this activity to build the output binding of the activity. We stop counting when we find the next occurrence of the activity under consideration, and we only count those activities where the activity under consideration is the pattern's nearest input. The input bindings are discovered in the same manner, but in the opposite direction.

The causal net is the result of the Heuristic miner. For some analyses, such as determining the conformance between an event log and the model, we must transform the causal net to the Petri net. In (Silva & Mira Da Silva, 2022), the authors describe this conversion. However, such a conversion is problematic because it may result in unsound Petri nets, and Petri nets may have a firing sequence that cannot be extended to become a legitimate firing sequence. To produce a simpler model, we skip some unnecessary invisible transitions.

ProM software employs a similar approach to mine. The ProM[2] conversion, on the other hand, does not consider long-distance dependencies. As a result, the model may allow traces. In this context, our implementation solves this problem by allocating additional locations for each long-distance dependency.

IV. IMPLEMENTATION

The purpose of this paper was to describe an advanced algorithm for process modelling, known as the heuristics' miner. It begins with an overview of process mining. It then goes on to explain to the reader the models that are used and the definitions that are required. Following that:

A. Structure

The PMLib library was created with extensibility, portability, and maintainability as primary goals. As a result, the library uses very few external, non-standard libraries, with Deedle (Phillippe, 2020) being the only one used at the time of writing this paper, a library used for importing data from CSV files and storing data in data frames. PMLib is built with .NET Core 3, a Microsoft ecosystem open-source, cross-platform development framework. As described in the .NET Core overview (Microsoft, 2020), .NET Core is a versatile framework for building modern applications.

Deedle, an exploratory data library for .NET, as described in the documentation provided by Blue Mountain Capital (2020), is a valuable tool for data analysis.

B. Structure

The library's design is hierarchical, albeit relatively flat, with the goal of separating functionality into directories and subdirectories so that a potential user only needs to import the parts of the library that are required. A hierarchical structure with few levels fits process mining techniques rather well because the scientific field is divided into three main compartments, the most commonly used of which are play-in and replay (see Figure 3), and these compartments are mostly made up of relatively standalone procedures. The source code for this project is available on the GitHub [3] platform. There are three directories in the solution. We ignored the conformance checking part. We focus only on the discovery and the Model directories. Therefore, we conclude the following hierarchical directory structure:

• ProcessM.NET – solution.

- Discovery / HeuristicMiner

HeuristicMiner

DependencyMatrix

DependencyGraph

HeuristicMinerSettings

- Model/ CausalNet

    CNet

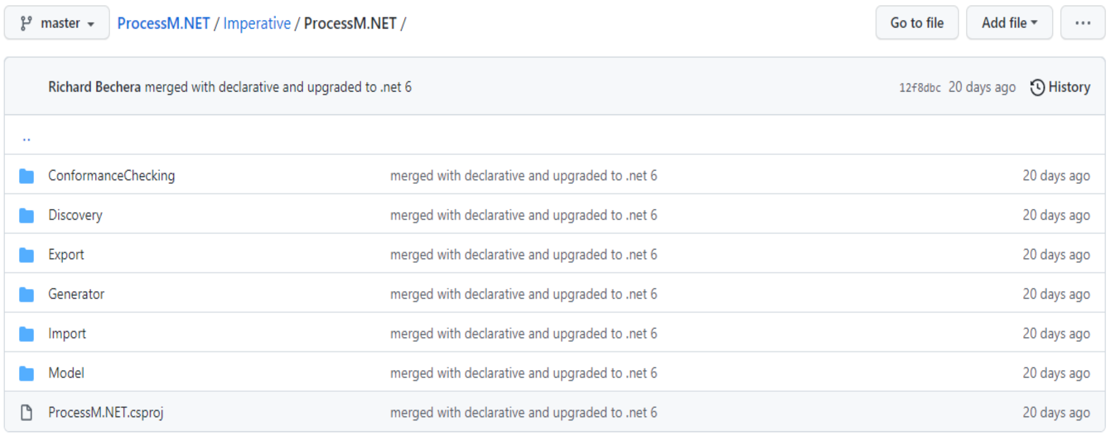
    CNetUtils (transformation to Petri net)

• ProcessM.NET-tests - unit tests and necessary resources for testing

• ProcessM.NET-Demo – demonstration

[2] https://www.promtools.org/doku.php

[3] https://github.com/lasaris/ProcessM.NET



master
ProcessM.NET / Imperative / ProcessM.NET /
Go to file
Add file
...

Richard Bechera merged with declarative and upgraded to .net 6 12f8dbc 20 days ago History

..		
ConformanceChecking	merged with declarative and upgraded to .net 6	20 days ago
Discovery	merged with declarative and upgraded to .net 6	20 days ago
Export	merged with declarative and upgraded to .net 6	20 days ago
Generator	merged with declarative and upgraded to .net 6	20 days ago
Import	merged with declarative and upgraded to .net 6	20 days ago
Model	merged with declarative and upgraded to .net 6	20 days ago
ProcessM.NET.csproj	merged with declarative and upgraded to .net 6	20 days ago

Figure 3. Dependency Graph illustration.

In PMLib (2015), event logs are represented by an ImportedEventLog class, which consists of a Deedle data frame and three string properties – CaseId, Activity, and Timestamp – to highlight the case, activity, and timestamp columns in the data frame. Initially, the string properties are set to null.

Only a data frame is used to create an instance of the ImportedEventLog class. The class then includes methods for configuring the three string properties, two of which – CaseId and Activity – are required before the event log can be processed into a workflow log. The Timestamp property is not required for creating a workflowlog because the order of events in the event log will be used to order activities in cases. However, some process mining techniques, such as bottleneck mining, require the Timestamp property to be defined in order to function properly.

The subdirectory contains two classes for modelling workflow traces: WorkflowTrace and TimestampedWorkflowTrace. For the most part, the classes are similar in that they both serve as simple containers for a string property Case, indicating the case under which the activities in the WorkFlowTrace were grouped, and a collection of strings, Activities, containing the activities of events belonging to the specified case. The classes then provide methods for inserting activities into the trace, and both are created with only a Case string as an input. The ordering of the activities is what distinguishes the two classes. WorkflowTrace stores the activities in the same order that they were added to the trace, using a list of strings as a container.

The relations are implemented in the class SucessorMatrix. There are three matrices in the class. The DirectMatrix stores relational information $>_\omega$. The L2LMatrix stores relational information $>>_\omega$, while the LongDistanceMatrix stores relational information $>>>_\omega$. In addition, the Activity Occurrences class contains an array of activity occurrences.

The dependency graph is implemented in the classes HeuristicMinerSettings and DependencyGraph.

The HeuristicMinerSettings folder also includes the default values for these settings. Because int comparison is faster than string comparison, activities in the DependencyGraph are stored as int. As a result, the class provides two types of mapping, such as int to string and vice versa. There is a distinct StartActivity and EndActivity. InputActivities and OutputActivities, which are lists of hash sets, are included in the class (accessible by activity ID). Finally, LongDependencies is a hash set of tuples that stores long-distance dependencies.

The implementation of causal networks is divided into three classes: CNet, CPlace, and Binding. The Binding class keeps track of a collection of Activities and their Frequency. The CPlace class holds nodes that contain information about the activity Id and frequency. The causal net is represented by the CNet class. Because activities are once again stored as integers, there are two mapping functions. Nodes are represented by a list of CPlaces that are stored in Activities. It has a distinct StartActivity and a distinct EndAcitivity.

Furthermore, the dictionaries InputBindings and OutputBindings are supplemented by the LongDistance dictionary for long-term dependent activities.

The Petri net subdirectory contains a representation of a Petri net and its components– places, transitions, and arcs – for the PMLib library's needs. Because PMLib is a process mining library, the term "Petri net" is understood to be synonymous with "workflow net". It should also be noted that the Petri net model lacks firing functionality, as only the static part of the model is required for a significant portion of the library. The firing functionality is then implemented as an overlay to the static model of a Petri net in directories containing classes that require such behaviour to function properly.

The static part of a Petri net is represented by three classes in the Model directory: Place, Transition, and PetriNet, which implement three corresponding interfaces: IPlace, ITransition, and IPetriNet.

C. Import and Export

The PMLib library also includes Petri net import and export functionality. The Import directory contains two classes: CSVImport and PNMLImport. The Export directory contains two classes: DOTExport and PNMLExport. CSVImport is a static class with a single method – MakeDataFrame.

The method takes a path to a.csv file and several optional arguments before passing the input to Frame.

Deedle library's ReadCsv method generates a data frame containing the imported data. The method then creates and returns an instance of the ImportedEventLog class using the imported data frame.

PNMLImport is a static class with one public method: Deserialise. The method expects a string containing the path to an.xml file containing a Petri net representation in the standardised Petri Net Markup Language (PNML, 2015). If such a Petri net exists in the file, the method parses the XML and returns a Petri net.

DOTExport is a static class with one public method, Serialise. As input, the method expects an instance of a class derived from the IPetriNet interface, as well as an optional string value of indentation sequence. The method then generates a representation of the given Petri net in a dot language (2020), which can be visualised using dot-compliant tools like Graphviz (2022). The dot representation of the given Petri net is then saved as a.dot file, and the method returns the full file name.

PNMLExport is a static class with one public method: Serialise. As an input, the method expects an instance of a class derived from the IPetriNet interface. Following that, the method generates a PNML-compliant XML representation of the given Petri net. Then, the method saves the XML representation of the Petri net into a.xml file and returns the full file name.

V. CONCLUSION

The purpose of this paper was to describe an advanced algorithm for process modelling, known as the heuristics' miner. It begins with an overview of process mining. It then goes on to explain to the reader the models that are used and the definitions that are required. Following that, it explains how the Heuristic Miner works.

As the only one currently available, the library presented in this paper is written in the C#.NET Core 6 framework. It is designed to be simple to understand, even for developers who are unfamiliar with process mining. The implementation has been thoroughly documented. It also includes the unit tests for the miner algorithm and the heuristics. The GitHub1 platform is used to access the library.

The library is a fully functional process mining library that is future-extensible (inductive miner algorithm). This paper's text may be used as study material for these algorithms.

In the future, we hope to implement the inductive miner algorithm using the Process Mining .NET library. The future implementation must include the genetic miner algorithms as well as a set of automatic tests to demonstrate the genetic miner algorithm's proper functionality.

VI. REFERENCES

Coutinho-Almeida, J & Cruz-Correia, RJ 2022, 'Developing a Process Mining Tool Based on HL7, Procedia Computer Science, vol. 196, pp. 501–508.

Phillippe, R 2020, Deedle: Exploratory data library for .NET, New York, New York: Blue Mountain Capital, viewed on 24 May 2020, <https://bluemountaincapital.github.io/Deedle/csharpintro.html>.

El-Gharib, NM & Amyot, D 2019, 'Process Mining for Cloud-Based Applications: A Systematic Literature Review', in the Proceeding of the 27th International Requirements Engineering Conference Workshops, Jeju, 27 September 2019, Jeju, Korea South.

Ellson, J 2022, Graphviz - Graph Visualization Software, viewed on 4 February 2022, < https://www.graphviz.org/doc/info/lang.html>.

Janssenswillen, G, Depaire, B, Swennen, M, Jans, M & Vanhoof, K 2019, 'bupaR: Enabling reproducible business process analysis', Knowledge-Based Systems, vol. 163, pp. 927–930.

Microsoft 2020, NET Core overview, Albuquerque, New Mexico: Microsoft, viewed on 4 February 2020, <https://docs.microsoft.com/en-us/dotnet/core/about>.

PNML: Petri Net Markup Language, Pnml.org, viewed on 4 February 2022, <http://www.pnml.org>.

Raffety, J, Stone, B, Svacina, J, Woodahl, C, Cerny, T & Tisnovsky, P 2021, 'Multi-source Log Clustering in Distributed Systems, Lecture Notes in Electrical Engineering', vol. 739, pp.31–41.

Silva, EC & Mira Da Silva, M 2022, 'Research contributions and challenges in DLT-based cryptocurrency regulation: a systematic mapping study', Journal of Banking and Financial Technology, vol.6, no.1, pp.63-68.

Van der Aalst, WMP, Adriansyah, A & van Dongen, B 2011, 'Causal Nets: A Modeling Language Tailored towards Process Discovery', Lecture Notes in Computer Science, vol. 6901, pp. 28-42.

Van der Aalst, WMP 2016, 'Data Science in Action', Process Mining: Data science in Action, 2nd ed, Berlin Heidelberg, Springer, pp. 3-22.

Waibel, P 2022, Causal Process Mining from Relational Databases with Domain Knowledge, viewed on 20 July 2023, <https://arxiv.org/abs/2202.08314>.

Yulion, D, Yahya, BN & Lewi Engel, VJ 2022, 'Building a robust transition matrix using causal matrix for route recommendation', Procedia Computer Science, vol. 197, pp. 768–775.