USER MANUAL

Hierarchical Process Discovery

Muhammad Saud Khan
Sezin Meiden
Byoung-Kyu Park

Table of Contents

Overview

This **Hierarchical process discovery program** provides a **user** with the functionalities of abstraction in the process discovery. Given an event log, it can be used to discover the patterns or transform the log either according to user-provided patterns or a user selection of discovered patterns. The result can be discovered by transformed logs and graphical process models. Consequently, this program helps the user to overcome the difficulties of analyzing spaghetti-like-models by simplifying it via abstractions.

The development period:

- This program has been developed from Apr to Jun 2020, as a part of "Process discovery using Python" in the summer semester in 2020 in RWTH.

The development team:

● Sezin Maden	sezin.maden@rwth-aachen.de
● Muhammad Saud Khan	saud.khan@rwth-aachen.de
● Byoungkyu Park	byoungkyu.park@rwth-aachen.de

NOTE: For any question, please feel free to contact any of us.

The instructors (Client) in PADS lab:

- Madhavi Shankara Narayana

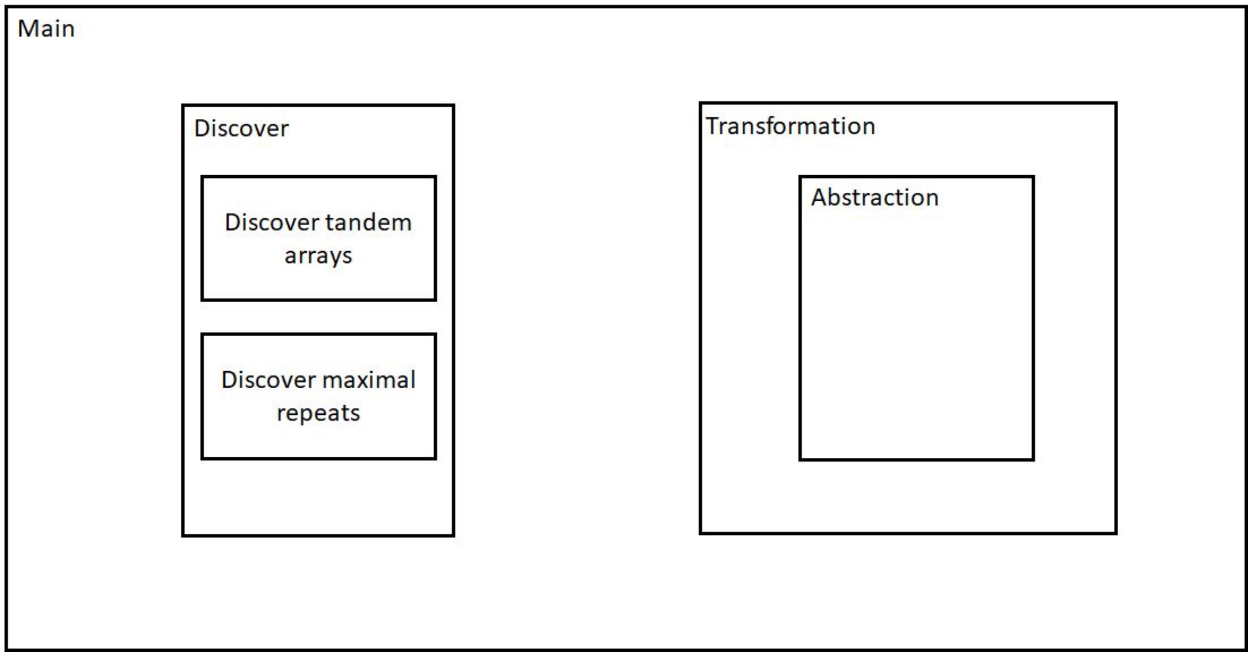
Structure of User Manual

This **User Manual** consists of three different parts. Depending on the environment of the user and his/her knowledge over the process mining, some parts can be skipped.

- **Initial setup**
  This describes how to set up the environment for the program. This includes the installation of python and libraries. Even for users who have already installed Python and the required libraries, this section is required to install the program.

- **Concepts and Terminologies**
  This describes the basic concepts of process mining and terminologies used in this manual. For those who already have knowledge of process mining, you can skip this step.

- **Features of this program / How to run**
  Each module can be run independently if the required conditions are met in advance.
  - **main.py:** to run all modules at once.
  - **discover.py:** to read a given log, find patterns, and store them in a pattern file
  - **abstraction.py:** with a given log and desired patterns, it only displays an abstracted log on console.
  - **transformation.py:** to provide the user with a graphical model and transformed log. The transformed log and model can be exported as required



Initial Setup

The user is assumed to work under a condition that ensures the running of the program without any unexpected interruptions, including enough computing power, sufficient storages, stable electricity, secured and workable network connection, and so on.

Hardware requirement

The "Recommended System Requirements" is not the absolute guideline to run the program, but insufficient computing power and lack of storage may result in performance in many negative ways.

Recommended System Requirements

- Processors: Intel® Core™ i5 processor 4300M at 2.60 GHz or 2.59 GHz (1 socket, 2 cores, 2 threads per core), 8 GB of DRAMIntel® Xeon® processor E5-2698 v3 at 2.30 GHz (2 sockets, 16 cores each, 1 thread per core), 64 GB of DRAMIntel® Xeon Phi™ processor 7210 at 1.30 GHz (1 socket, 64 cores, 4 threads per core), 32 GB of DRAM, 16 GB of MCDRAM (flat mode enabled)
- Disk space: 2 to 3 GB
- Operating systems: Windows® 10, macOS*, and Linux*

Minimum System Requirements

- Processors: Intel Atom® processor or Intel® Core™ i3 processor
- Disk space: 1 GB
- Operating systems: Windows* 7 or later, macOS, and Linux
- Python* versions: 2.7.X, 3.6.X

*(**Source:** https://www.quora.com/What-are-the-minimum-hardware-requirements-for-python-programming)*

Software Requirement

This program is only runnable under the environment of python equipped with the PM4Py and Graphviz libraries. This part is prepared for the user on the window 10 - 64bit.

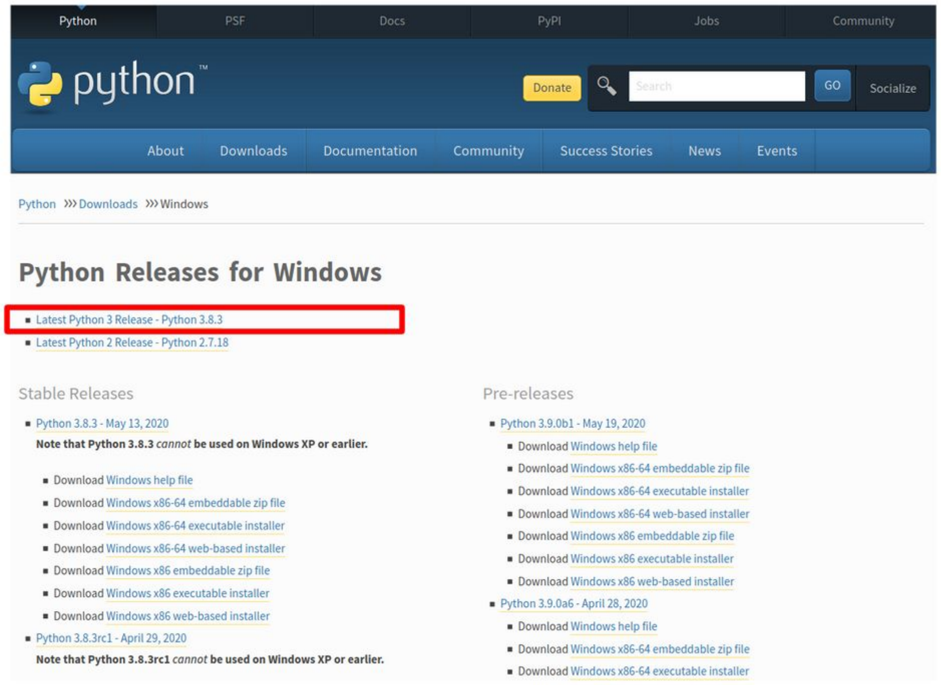
For other operating systems, please refer to the following links

https://docs.python.org/3/
https://pm4py.fit.fraunhofer.de/install

Download and Install Python

For the window user, the download link is as below. Please click the highlighted link to download the Python for window 64bit. Then, the user can execute the installation file.

https://www.python.org/downloads/windows/



Python »» Downloads »» Windows

Python Releases for Windows

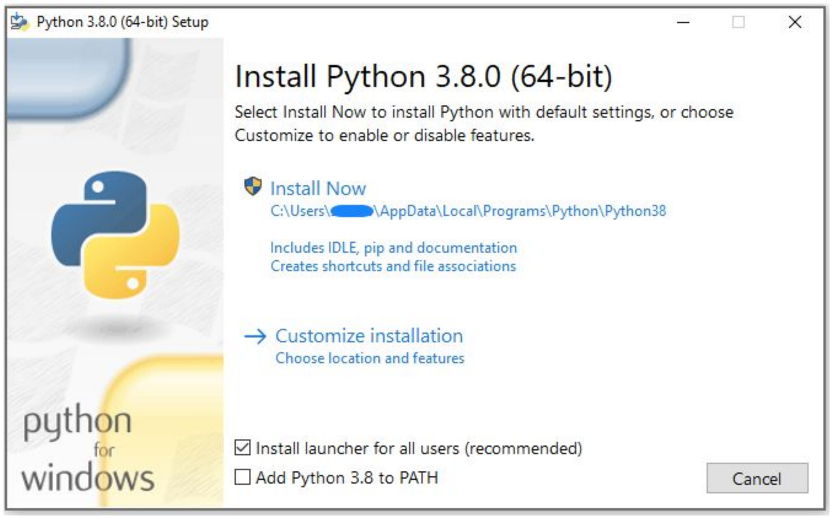
- Latest Python 3 Release - Python 3.8.3
- Latest Python 2 Release - Python 2.7.18

Stable Releases

- Python 3.8.3 - May 13, 2020
  Note that Python 3.8.3 *cannot* be used on Windows XP or earlier.
  - Download Windows help file
  - Download Windows x86-64 embeddable zip file
  - Download Windows x86-64 executable installer
  - Download Windows x86-64 web-based installer
  - Download Windows x86 embeddable zip file
  - Download Windows x86 executable installer
  - Download Windows x86 web-based installer
- Python 3.8.3rc1 - April 29, 2020
  Note that Python 3.8.3rc1 *cannot* be used on Windows XP or earlier.

Pre-releases

- Python 3.9.0b1 - May 19, 2020
  - Download Windows help file
  - Download Windows x86-64 embeddable zip file
  - Download Windows x86-64 executable installer
  - Download Windows x86-64 web-based installer
  - Download Windows x86 embeddable zip file
  - Download Windows x86 executable installer
  - Download Windows x86 web-based installer
- Python 3.9.0a6 - April 28, 2020
  - Download Windows help file
  - Download Windows x86-64 embeddable zip file
  - Download Windows x86-64 executable installer



Install Python 3.8.0 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python38

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.8 to PATH

(***Source:*** *https://docs.python.org/3/using/windows.html*)

For more details of installation and potential problems, please refer to the link:
https://docs.python.org/3/

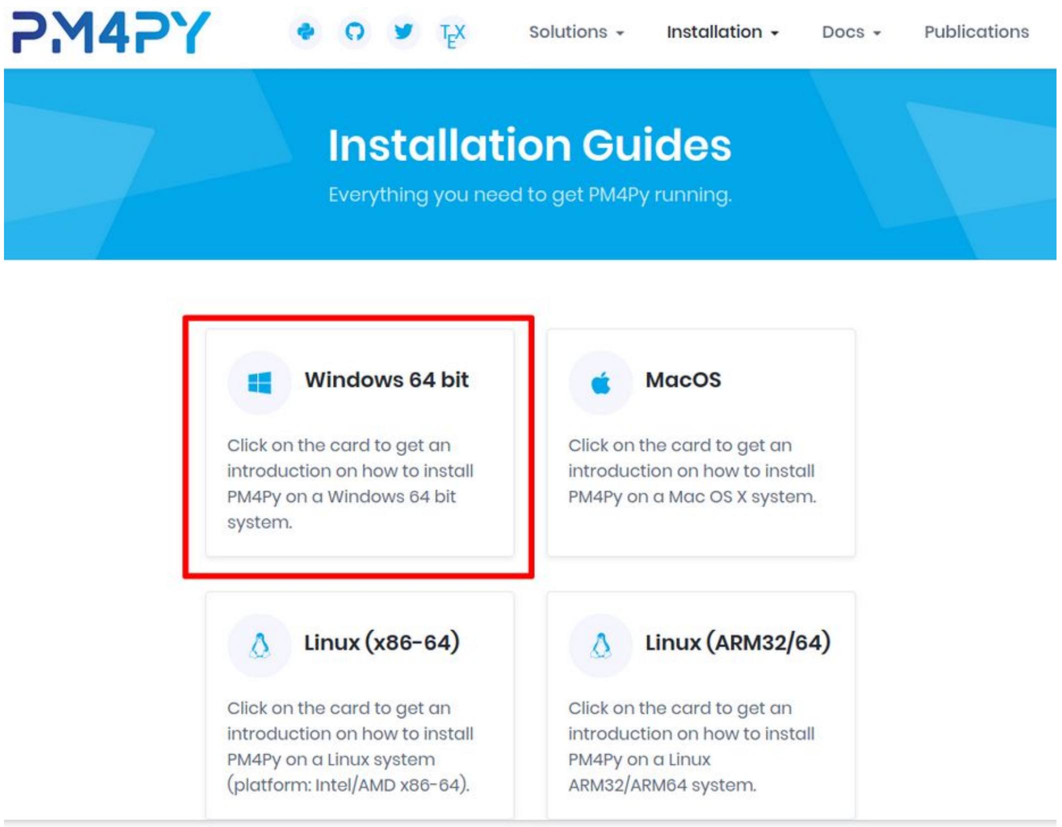
Installation of PM4Py

For users on the 64 bit-Window systems, the below steps are required.

1. Install a Windows C/C++ compiler
2. Install GraphViz
3. Install Anaconda/Miniconda
4. Install PM4Py

For detailed steps and other operating systems, users are requested to visit the web site.

https://pm4py.fit.fraunhofer.de/install



Installation Guides

Everything you need to get PM4Py running.

Windows 64 bit

Click on the card to get an introduction on how to install PM4Py on a Windows 64 bit system.

MacOS

Click on the card to get an introduction on how to install PM4Py on a Mac OS X system.

Linux (x86-64)

Click on the card to get an introduction on how to install PM4Py on a Linux system (platform: Intel/AMD x86-64).

Linux (ARM32/64)

Click on the card to get an introduction on how to install PM4Py on a Linux ARM32/ARM64 system.

After the installation of Anaconda (the 3rd step mentioned above), PM4Py can be installed by running the below command.

```
pip install pm4py
```

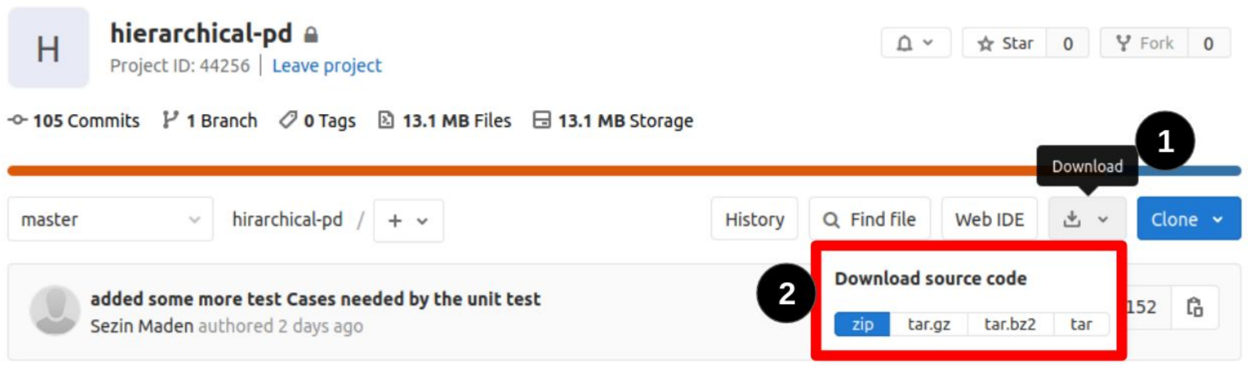
Python version 3.7.3 and PM4Py 1.3.1 were used during the development. The lower versions may not guarantee the best performance of this program. As of May 2020, the latest release of PM4Py is 1.3.1, and Python has version 3.8.3. We recommend the user to adopt the latest python.

Installation of Hierarchical Process Discovery (Source)

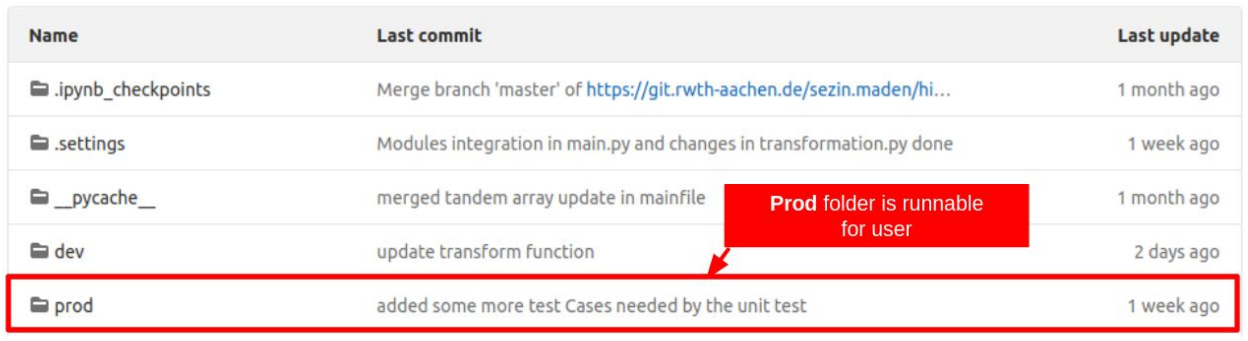
The program in python code can be found in the Gitlab repository of RWTH. The user can directly download from

https://git.rwth-aachen.de/sezin.maden/hirarchical-pd.git

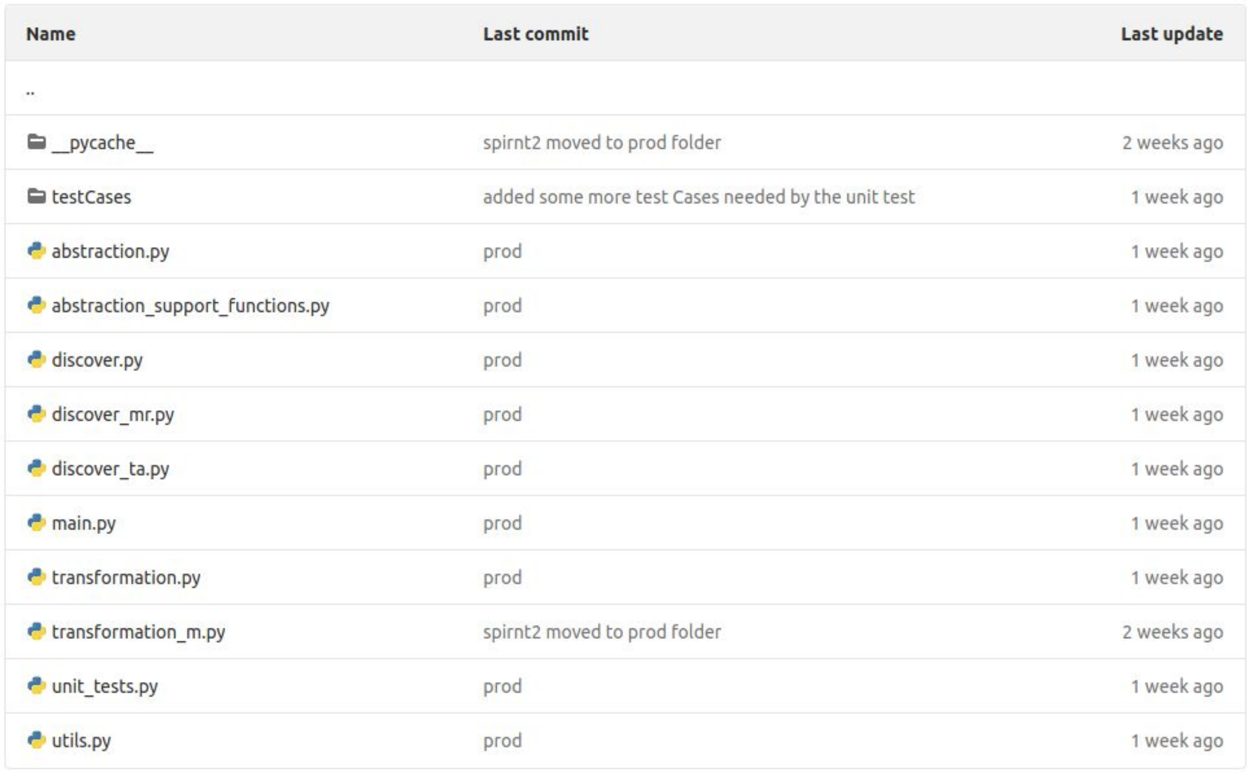
and extract the zip file in your working folder.



Please note that **"prod"** folder contains the runnable python files.



Name	Last commit	Last update
.ipynb_checkpoints	Merge branch 'master' of https://git.rwth-aachen.de/sezin.maden/hi...	1 month ago
.settings	Modules integration in main.py and changes in transformation.py done	1 week ago
__pycache__	merged tandem array update in mainfile	1 month ago
dev	update transform function	2 days ago
prod	added some more test Cases needed by the unit test	1 week ago



Name	Last commit	Last update
..		
__pycache__	spirnt2 moved to prod folder	2 weeks ago
testCases	added some more test Cases needed by the unit test	1 week ago
abstraction.py	prod	1 week ago
abstraction_support_functions.py	prod	1 week ago
discover.py	prod	1 week ago
discover_mr.py	prod	1 week ago
discover_ta.py	prod	1 week ago
main.py	prod	1 week ago
transformation.py	prod	1 week ago
transformation_m.py	spirnt2 moved to prod folder	2 weeks ago
unit_tests.py	prod	1 week ago
utils.py	prod	1 week ago

To grant the authority to access the webpage, the user is requested to contact the development team.

Concepts and Terminologies

Running Environment General

Python

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

(Source:** https://en.wikipedia.org/wiki/Python_(programming_language)**)

PM4Py

PM4Py is a software product, developed by the Fraunhofer Institute for Applied Information Technology (FIT). In particular, PM4Py is developed by the process mining group of Fraunhofer FIT. PM4Py is created with many goals.

(Source:** https://pm4py.fit.fraunhofer.de/about-us**)

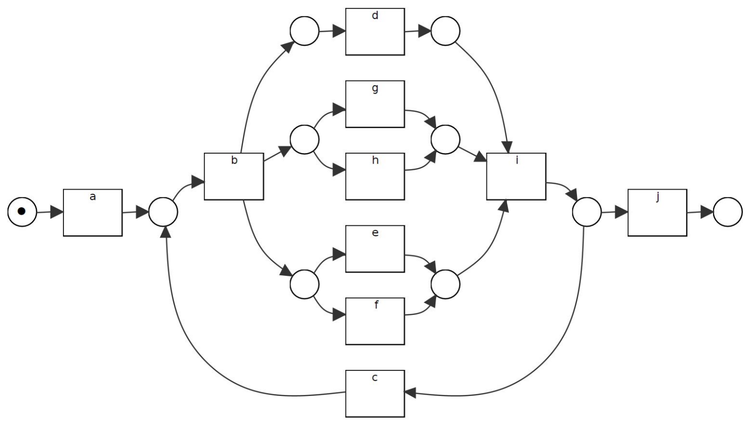
Process Discovery

Process Discovery is a part of Process Mining concerned with the automated discovery of business process models from event data. It provides a set of tools and techniques that are used to understand underlying business processes for the given event log.

For example of discovering a model with an event log,

L2= [<a, b, d, e, g, i, j>, <a, b, e, d, h, i, j>, <a, b, f, d, g, i, j>, <a, b, f, g, d, i, j>, <a, b, g, d, f, i, j>, <a, b, d, f, h, i, c, b, e, h, d, i, j>, <a, b, g, f, d, i, c, b, h, f, d, i, j>, <a, b, h, e, d, i, c, b, d, g, e, i, j>]

A process model for the above log is generated as below



NOTE: The above model was prepared only for reference to help the user understand the concept of process discovery.

For more advanced and detailed topics about Process Mining, please refer to:

- "W. van der Aalst. Process Mining: Data Science in Action. Springer-Verlag, Berlin, 2016" (http://springer.com/9783662498507)

- Coursera Online Course - Process Mining: Data science in Action https://www.coursera.org/learn/process-mining

Event Data

Data recorded with **Case ID, Activity Name, and Timestamp**, which are essential for Process Mining. Additional information can be included such as Resource, Cost, and so on.

Event data come from various sources, such as a database system, a message log, an open API providing data from websites or social media, a comma-separated values file or spreadsheet, a translation log, a business suite/ERP system (SAP, Oracle, and etc.)

(Source:** BPI course, 2020 SE, RWTH**)

For example, a list of event data

(Case ID)	**(Activity Name)**	**(Timestamp)**	**(Other)**
Student Name	**Course Name**	**Exam date**	**mark**
Peter Jones	Business Information systems	16-1-2014	8
Sandy Scott	Business Information systems	16-1-2014	5
Bridget White	Business Information systems	16-1-2014	9
John Anderson	Business Information systems	16-1-2014	8
Sandy Scott	BPM Systems	17-1-2014	7

Peter Jones	Algorithm	20-1-2014	1
Sandy Scott	BPM Systems	17-1-2014	7
Peter Jones	Logic	29-1-2014	4

(Source:** BPI course lecture material, RWTH**)

Case

Sequence/trace of events. It can be identified by **Case ID.**
In the example above, using the case id "Peter Jones", the corresponding case is as below:

(Case ID)	**(Activity Name)**	**(Timestamp)**	**(Other)**
Student Name	**Course Name**	**Exam date**	**mark**
Peter Jones	Business Information systems	16-1-2014	8
Peter Jones	Algorithm	20-1-2014	1
Peter Jones	Logic	29-1-2014	4

Event Log

Collection of the cases/traces

XES(eXtensible Event Stream)

Official IEEE Standard. It has been driven by the IEEE Task Force on Process Mining. The format is supported by tools such as ProM, Celonis, Disco, etc. Conversion from other formats (CSV) is easy if the right data are available.

Supported Formats:

XES:

This program assumes that the given event log contains at least **"concept:name" (or activity)** and **"time:timestamp".** Those are essential and restricted by the library of PM4Py.

NOTE :
Please note **"lifecycle:transition" events** are currently not supported, because the filter function of PM4PY loses the metadata of the log.

It may not import the log properly if any of the information is not included.

Examples from the **running-example.xes**

```
<trace>
        <string key="concept:name" value="3"/>
        <string key="creator" value="Fluxicon Nitro"/>
        <event>
                <string key="concept:name" value="register request"/>
                <string key="org:resource" value="Pete"/>
                <date key="time:timestamp" value="2010-12-30T14:32:00.000+01:00"/>
                <string key="Activity" value="register request"/>
                <string key="Resource" value="Pete"/>
                <string key="Costs" value="50"/>
        </event>
```

CSV:

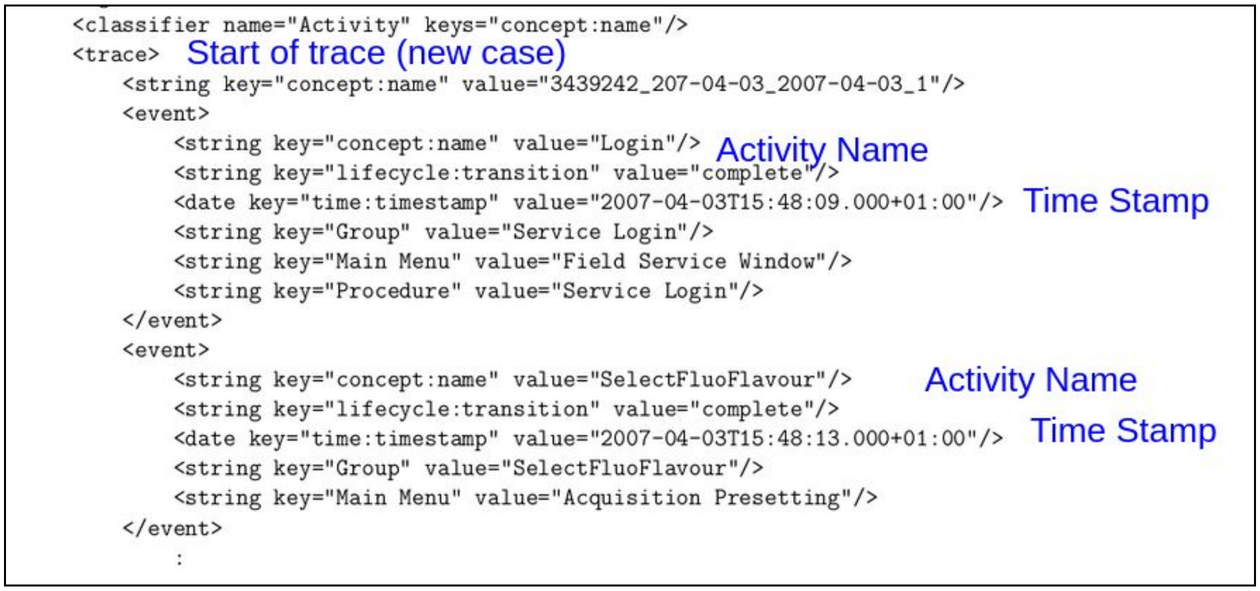
The below information is expected as mandatory due to the requirements of the PM4Py library.

- Activity or concept:name
- case:concept:name
- time:timestamp

Otherwise, it may not import the log in CSV properly.

For more details, please refer to www.xes-standard.org

(Source:** BPI course lecture material, RWTH**)

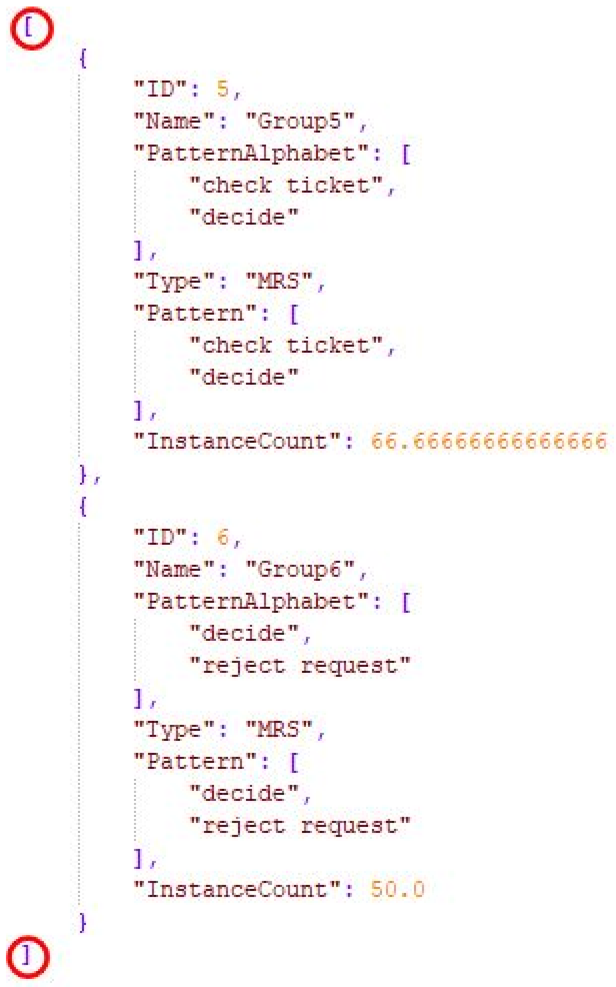


```
<classifier name="Activity" keys="concept:name"/>
<trace>
    <string key="concept:name" value="3439242_207-04-03_2007-04-03_1"/>
    <event>
        <string key="concept:name" value="Login"/>
        <string key="lifecycle:transition" value="complete"/>
        <date key="time:timestamp" value="2007-04-03T15:48:09.000+01:00"/>
        <string key="Group" value="Service Login"/>
        <string key="Main Menu" value="Field Service Window"/>
        <string key="Procedure" value="Service Login"/>
    </event>
    <event>
        <string key="concept:name" value="SelectFluoFlavour"/>
        <string key="lifecycle:transition" value="complete"/>
        <date key="time:timestamp" value="2007-04-03T15:48:13.000+01:00"/>
        <string key="Group" value="SelectFluoFlavour"/>
        <string key="Main Menu" value="Acquisition Presetting"/>
    </event>
        :
```

Pattern file

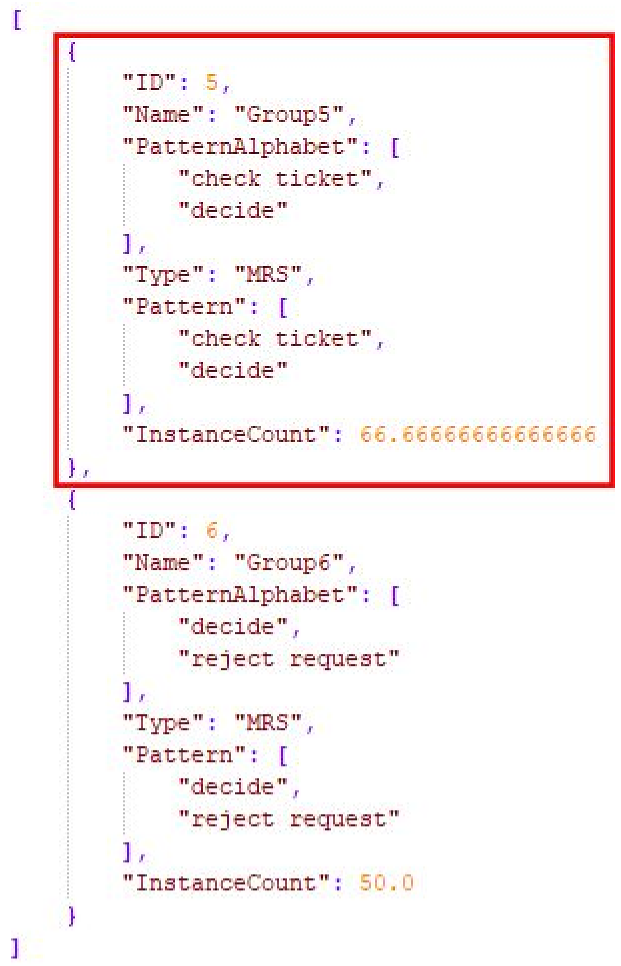
The patterns found in a given event log are stored in a pattern file. They are the sets of maximal repeats or tandem arrays in JSON format.

A valid pattern.json file always starts and ends with square brackets



```
[
    {
        "ID": 5,
        "Name": "Group5",
        "PatternAlphabet": [
            "check ticket",
            "decide"
        ],
        "Type": "MRS",
        "Pattern": [
            "check ticket",
            "decide"
        ],
        "InstanceCount": 66.66666666666666
    },
    {
        "ID": 6,
        "Name": "Group6",
        "PatternAlphabet": [
            "decide",
            "reject request"
        ],
        "Type": "MRS",
        "Pattern": [
            "decide",
            "reject request"
        ],
        "InstanceCount": 50.0
    }
]
```

Every pattern is enclosed in curly brackets and separated from the next pattern by a comma. Every pattern consists of attributes given in the form “AttributeName”: Value. The attributes are separated by commas. Strings have to be enclosed in quotation marks and Lists are marked by enclosing the comma-separated values by square brackets. Numeric values (**ID** and **InstanceCount**) are not enclosed.



```
[
    {
        "ID": 5,
        "Name": "Group5",
        "PatternAlphabet": [
            "check ticket",
            "decide"
        ],
        "Type": "MRS",
        "Pattern": [
            "check ticket",
            "decide"
        ],
        "InstanceCount": 66.6666666666666
    },
    {
        "ID": 6,
        "Name": "Group6",
        "PatternAlphabet": [
            "decide",
            "reject request"
        ],
        "Type": "MRS",
        "Pattern": [
            "decide",
            "reject request"
        ],
        "InstanceCount": 50.0
    }
]
```

The below table describes all possible attributes of patterns, their meanings, and whether or not they are optional.

Attribute	**Type**	**optional/mandatory**	**Description**
ID	Integer	mandatory	A unique integer identifier of the pattern
Name	String	mandatory	A unique String name of the pattern. If the pattern is abstracted this is the new activity name replacing all occurrences of the pattern.
Pattern	List of Strings	mandatory	A list of Strings of activity names describing the pattern by giving an example.

PatternAlphabet	List of Strings	optional	A list of Strings containing all activity names that are contained at least once in the pattern. No activity is contained twice and the order can be different from the order of the pattern.
Type	String	optional	A String symbolizing the pattern type such as super maximal repeat or primitive tandem array. See the description of discovered patterns for more information.
InstanceCount	Numeric	optional	A number between 0 and 100, that describes the percentage of traces in a specific log containing the pattern at least once.

User Pattern

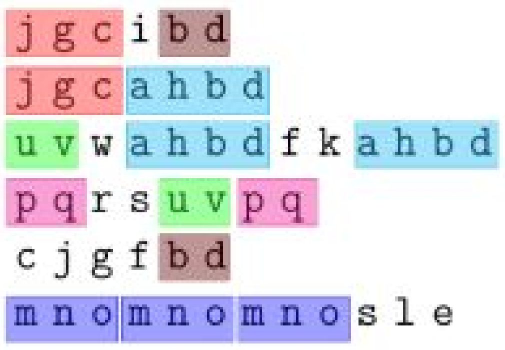
The user can input the desired sequence of activities for abstraction. It is called "User Pattern" in this manual and program. This should be valid as per the explanation of Pattern File in the above table.

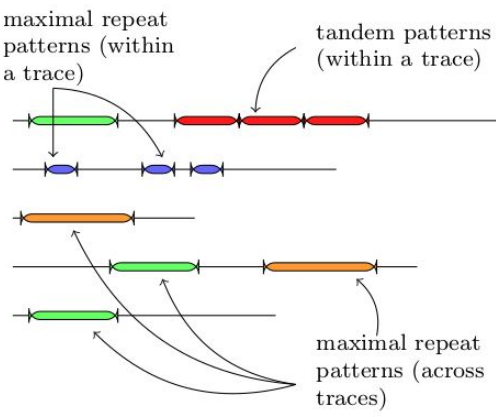
Patterns used in the program

In this program, **maximal repeat** and **tandem array** patterns are identified.

To briefly explain the concepts, referring to the left picture below, **"mno"** is repeated in a row within the same trace, which is "**Tandem Array**". Whereas, other sequences, such as **"jgc"**, **"bd"**, **"uv"**, **"ahbd"**, and **"pq"**, are repeated within or across traces which are "**Maximal Repeat**"

It is just to provide brief concepts of patterns. More precise and strict definitions are followed.




maximal repeat patterns (within a trace)
tandem patterns (within a trace)
maximal repeat patterns (across traces)

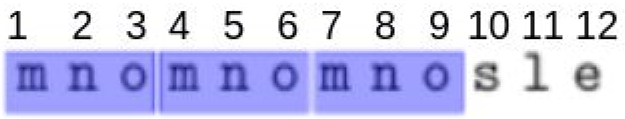
***(Source:** Process Mining in the large, R.P. Jagadeesh Chandra Bose,*
*https://pure.tue.nl/ws/files/3410491/730954.pdf***)**

Tandem Array

A tandem array in a sequence **s** is a subsequence **s(i, j)** of the form **α** α ... k times **α**, where **α**, called the tandem repeat type, is a non-empty sequence, and k >= 2. We denote a tandem array by the triple.

(i, α, k) i : starting position α : repeat type k : number of repetition

the example below can be expressed as **(1,"mno",3)** . In this program, we are only interested in the tandem repeat type.



Maximal Tandem Array

A tandem array in a sequence **s** is a maximal tandem array if and only if there are no additional copies of **α** immediately before or after **s(i, j)**.

(1, "mno", 2) or **(4, "mno", 2)** are not **Maximal Tandem Array,** but **(1,"mno",3)**. In this program, only Maximal Tandem Arrays are considered and abbreviated as **"MTA"**

Primitive Tandem Repeat Type

A tandem repeat type **α** is called a primitive tandem repeat type if and only if **α** is not a tandem array. For example, **"abc"** is a **Primitive Tandem Repeat Type** of **"abcabc"**

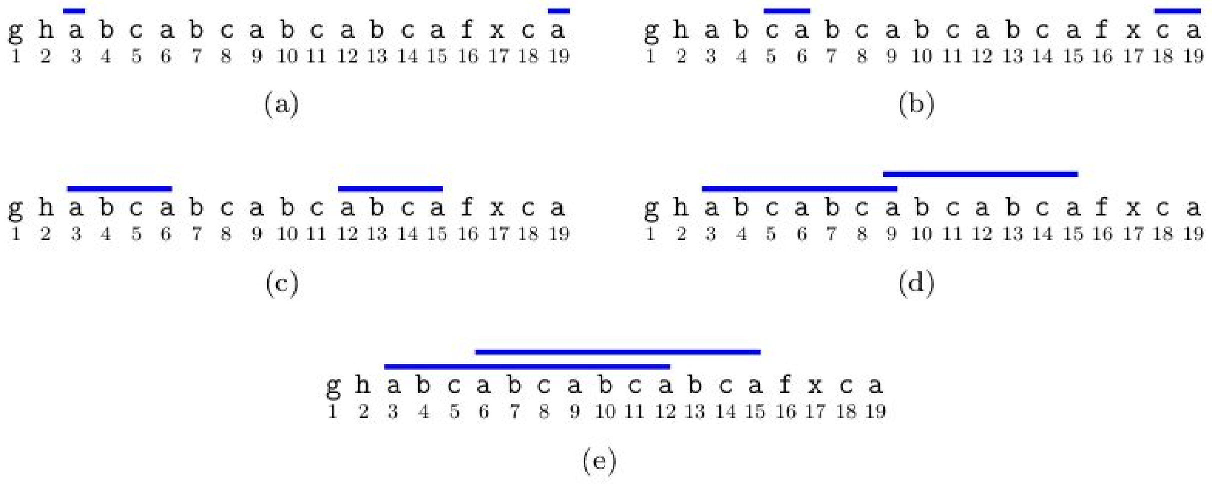
Primitive Tandem Array

A tandem array **(i, α, j)** is a primitive tandem array if and only if **α** is a primitive tandem repeat type. It is represented as **"PTA"** in the user pattern file.

Maximal repeat

A maximal repeat in a sequence is defined as a subsequence **α** that occurs in a maximal pair, for example, if the log is given as **{ghabcabcabcabcafxca}**
The set of maximal repeats in this trace is **{a, ca, abca, abcabca, abcabcabca}.**

In the user pattern file, it is represented as **"MRS"**



***(Source:** Process Mining in the large, R.P. Jagadeesh Chandra Bose,*
*https://pure.tue.nl/ws/files/3410491/730954.pdf***)**

Super Maximal Repeat

A super maximal repeat in a sequence is defined as a maximal repeat that never occurs as a proper subsequence of any other maximal repeat.

For the maximal repeats depicted in above only the maximal repeat **(e) abcabcabca** qualifies to be a super maximal repeat because all other maximal repeats happen to be a subsequence of **(e) abcabcabca.**

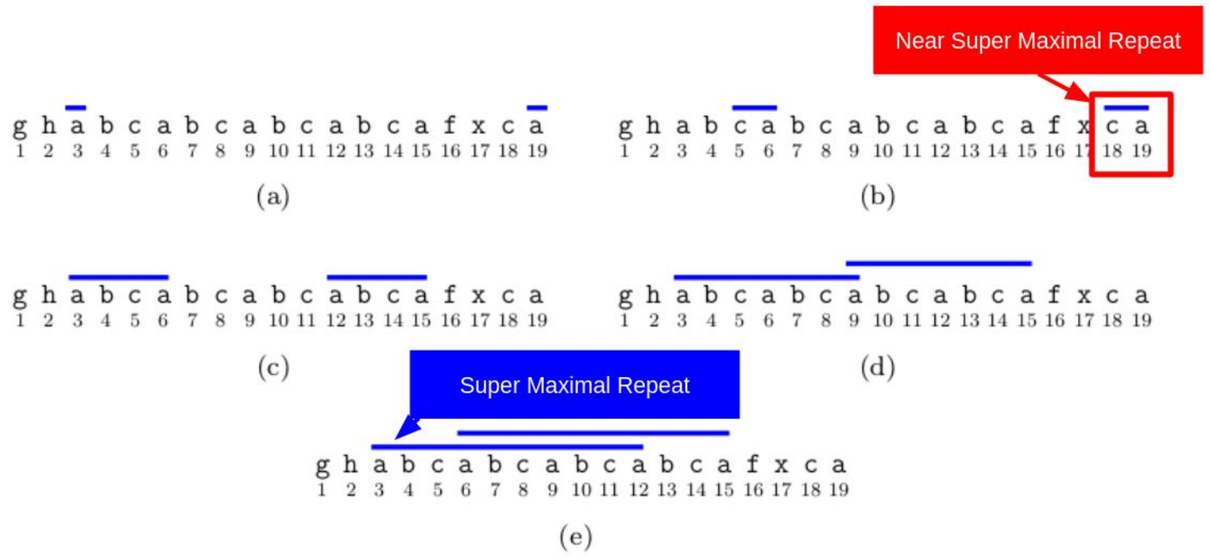
It is abbreviated as **SMRS** in the **Pattern File**

Near Super Maximal Repeat

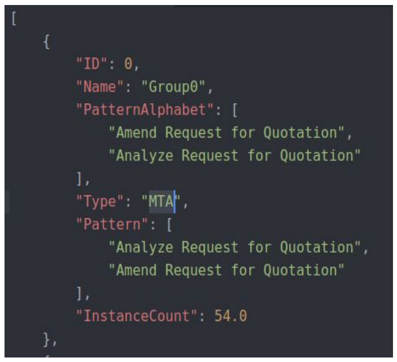
A maximal repeat **α** in a sequence **s** is said to be a near super maximal repeat if and only if there exists at least one instance of **α** at some location in the sequence **s** where it is not contained in another maximal repeat.

The manifestation of the maximal repeat **ca** at position 18 in **(b)** is not contained in any other maximal repeat manifestation. Hence the maximal repeat **"ca"** qualifies to be a near super maximal repeat.

In the **Pattern File**, it is abbreviated as **NSMRS.**

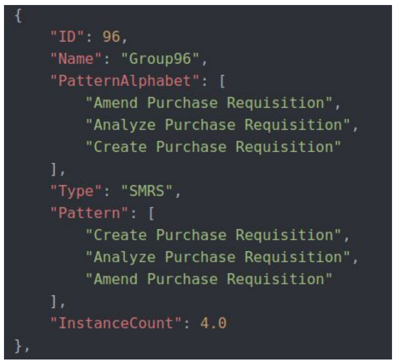


(Source:** Process Mining in the large, R.P. Jagadeesh Chandra Bose,* *https://pure.tue.nl/ws/files/3410491/730954.pdf* ***)



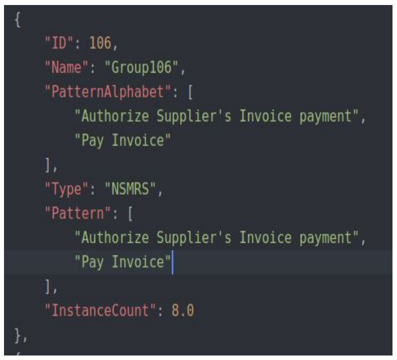
```
[
    {
        "ID": 0,
        "Name": "Group0",
        "PatternAlphabet": [
            "Amend Request for Quotation",
            "Analyze Request for Quotation"
        ],
        "Type": "MTA",
        "Pattern": [
            "Analyze Request for Quotation",
            "Amend Request for Quotation"
        ],
        "InstanceCount": 54.0
    },
```

An example of Maximal Tandem Array in the pattern file



```
{
    "ID": 96,
    "Name": "Group96",
    "PatternAlphabet": [
        "Amend Purchase Requisition",
        "Analyze Purchase Requisition",
        "Create Purchase Requisition"
    ],
    "Type": "SMRS",
    "Pattern": [
        "Create Purchase Requisition",
        "Analyze Purchase Requisition",
        "Amend Purchase Requisition"
    ],
    "InstanceCount": 4.0
},
```

An example of Super Maximal Repeat in the pattern file



```
{
    "ID": 106,
    "Name": "Group106",
    "PatternAlphabet": [
        "Authorize Supplier's Invoice payment",
        "Pay Invoice"
    ],
    "Type": "NSMRS",
    "Pattern": [
        "Authorize Supplier's Invoice payment",
        "Pay Invoice"
    ],
    "InstanceCount": 8.0
},
```

An example of Near Super Maximal Repeat in the pattern file

Program features / How to run

Main Module (main.py)

Description

The main module encapsulates all functionalities of this project in a runnable program. Given an event log, it can be used to discover the patterns mentioned above or transform the log either according to user-provided patterns or a user selection of discovered patterns. It always displays a process model of the most abstracted log and gives the option to save both the model as well as the log.

Required conditions

- A valid log file in XES or CSV format
- If a user pattern file is provided it has to be valid, as described above

Usage

```
main.py [-h] --path xes_file_name [--userPattern pattern_path] [--downloadLog]
[--downloadModel]

-h, --help                            show this help message and exit
--path          xes_file_name         specify the path of xes file. e.g. test.xes
--userPattern   pattern_path
                                      specify the path of the user pattern file. e.g. user.json to
                                      abstract all included patterns in order of occurrence
--downloadLog                         Flag specifying that the resulting log should be saved
--downloadModel                       Flag specifying that the resulting model should be saved
```

Example

Command:

```
$ python main.py --help
```

Result:

```
usage: main.py [-h] --path xes_file_name
          [--userPattern user_pattern_file_name] [--downloadLog]
          [--downloadModel]

To transform the logs with the pattern. The --path argument is mandatory.

optional arguments:
  -h, --help            show this help message and exit
  --path xes_file_name  specify the path of xes file. e.g. test.xes
  --userPattern user_pattern_file_name
                        specify the user pattern path file in json format.
  --downloadLog         flag to export/download transformed log
  --downloadModel       flag to export/download process model
```

Command:

```
$ python main.py --path ./testCases/running-example.xes
```

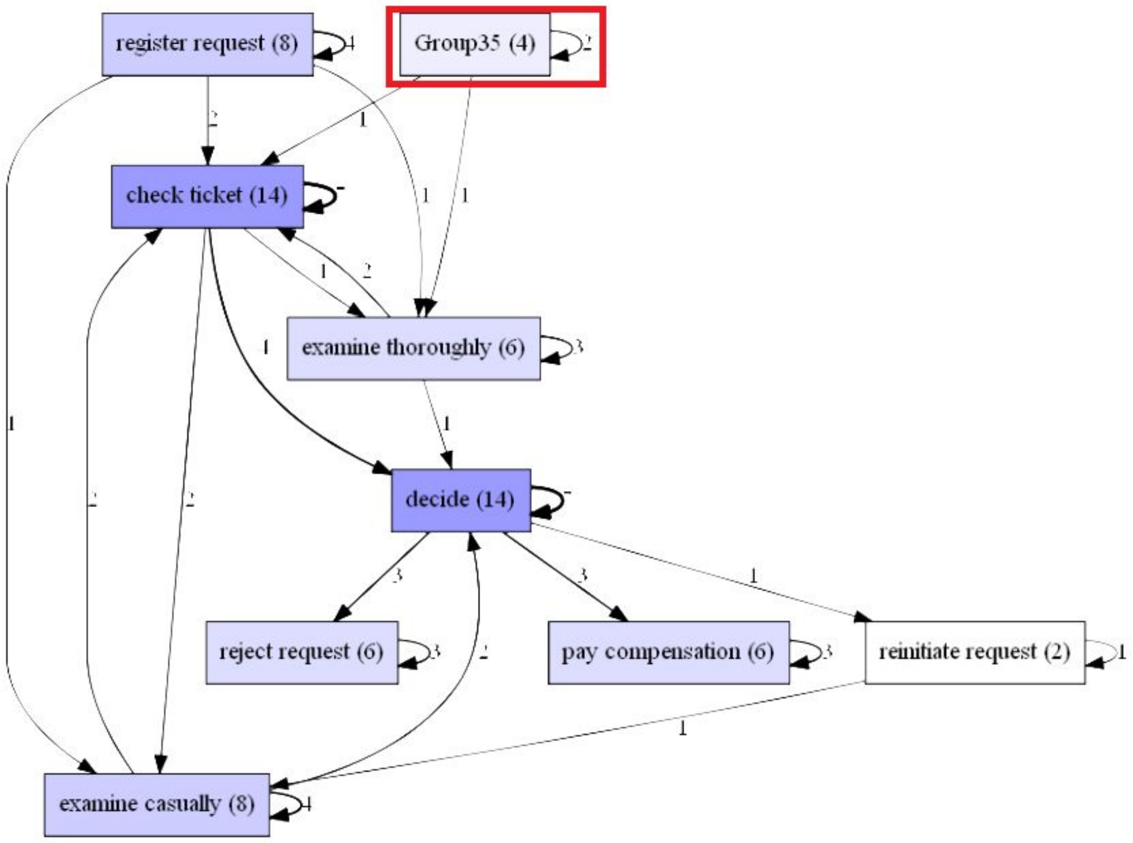
This is the most basic use of the main module. All maximal repeats and tandem arrays in the log are discovered and printed to an external pattern file. Once this is completed the user is presented with the following dialogue:

Result:

```
Please see testCases/running-example_patterns.json file for a list of found patterns
Please enter pattern numbers in order e.g. 1 2 3 1 (Required field):
```

The user can inspect the .json file and enter the ids of the patterns the user wishes to abstract. The ids need to be separated by one space. Then, the patterns are abstracted and an image of the discovered model is displayed. The patterns will be abstracted in the order that their ids are given. The program will exit if the ids are not valid. If the user presses enter without entering anything, nothing will be abstracted.

The image below is an example of a graphical model being abstracted with **Group 35**.

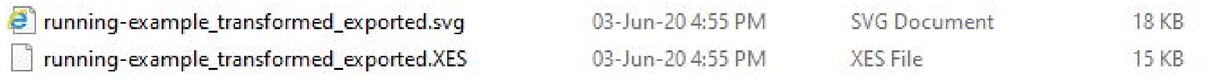


Command:

```
$ python main.py --path ./testCases/running-example.xes --downloadLog --downloadModel
```

The functionality described above can be extended by setting the downloadLog and downloadModel flags. If these flags are set, the program will export a transformed log and the discovered Model respectively. The files can be found in the working directory of the original log.

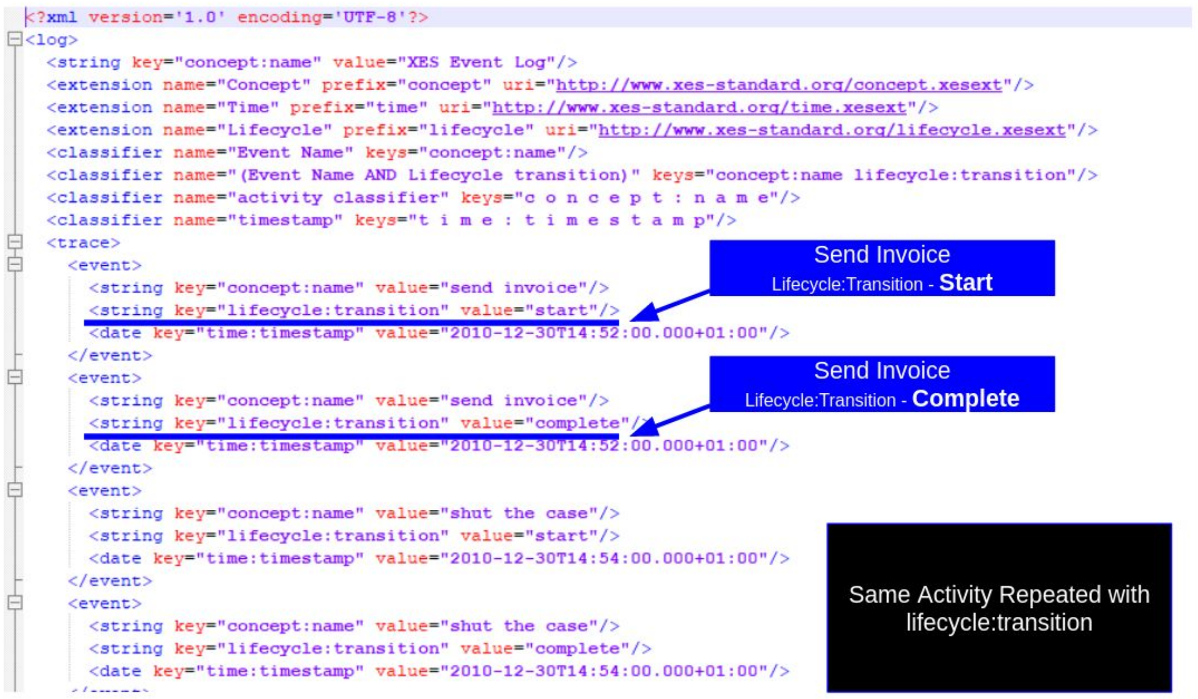
Result:



```
running-example_transformed_exported.svg    03-Jun-20 4:55 PM    SVG Document    18 KB
running-example_transformed_exported.XES    03-Jun-20 4:55 PM    XES File        15 KB
```

The output log is always in the XES format. For more experienced users, it is important to notice that the exported log also always contains start and end lifecycle events for each activity.

The exported event log should look in the below screenshot. The exported process model would be the same as we have seen above.



```
<?xml version='1.0' encoding='UTF-8'?>
<log>
  <string key="concept:name" value="XES Event Log"/>
  <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
  <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
  <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.xesext"/>
  <classifier name="Event Name" keys="concept:name"/>
  <classifier name="(Event Name AND Lifecycle transition)" keys="concept:name lifecycle:transition"/>
  <classifier name="activity classifier" keys="c o n c e p t : n a m e"/>
  <classifier name="timestamp" keys="t i m e : t i m e s t a m p"/>
  <trace>
    <event>
      <string key="concept:name" value="send invoice"/>
      <string key="lifecycle:transition" value="start"/>
      <date key="time:timestamp" value="2010-12-30T14:52:00.000+01:00"/>
    </event>
    <event>
      <string key="concept:name" value="send invoice"/>
      <string key="lifecycle:transition" value="complete"/>
      <date key="time:timestamp" value="2010-12-30T14:52:00.000+01:00"/>
    </event>
    <event>
      <string key="concept:name" value="shut the case"/>
      <string key="lifecycle:transition" value="start"/>
      <date key="time:timestamp" value="2010-12-30T14:54:00.000+01:00"/>
    </event>
    <event>
      <string key="concept:name" value="shut the case"/>
      <string key="lifecycle:transition" value="complete"/>
      <date key="time:timestamp" value="2010-12-30T14:54:00.000+01:00"/>
```

Send Invoice
Lifecycle:Transition - Start
Send Invoice
Lifecycle:Transition - Complete
Same Activity Repeated with lifecycle:transition

Command:

```
$ python main.py --path ./testCases/running-example.xes --userPattern ./testCases/user.json
```

The last big feature is that the user can give patterns **(user.json)** that are abstracted before the discovery step. We recommend using this feature if the user has prior knowledge about the process and would like to make specific abstractions. For this purpose, users can input a valid pattern file in JSON format as described above. All patterns in that file will be abstracted in the order they are occurring in the file. The abstracted patterns will be displayed on the console.

User Patterns on console:

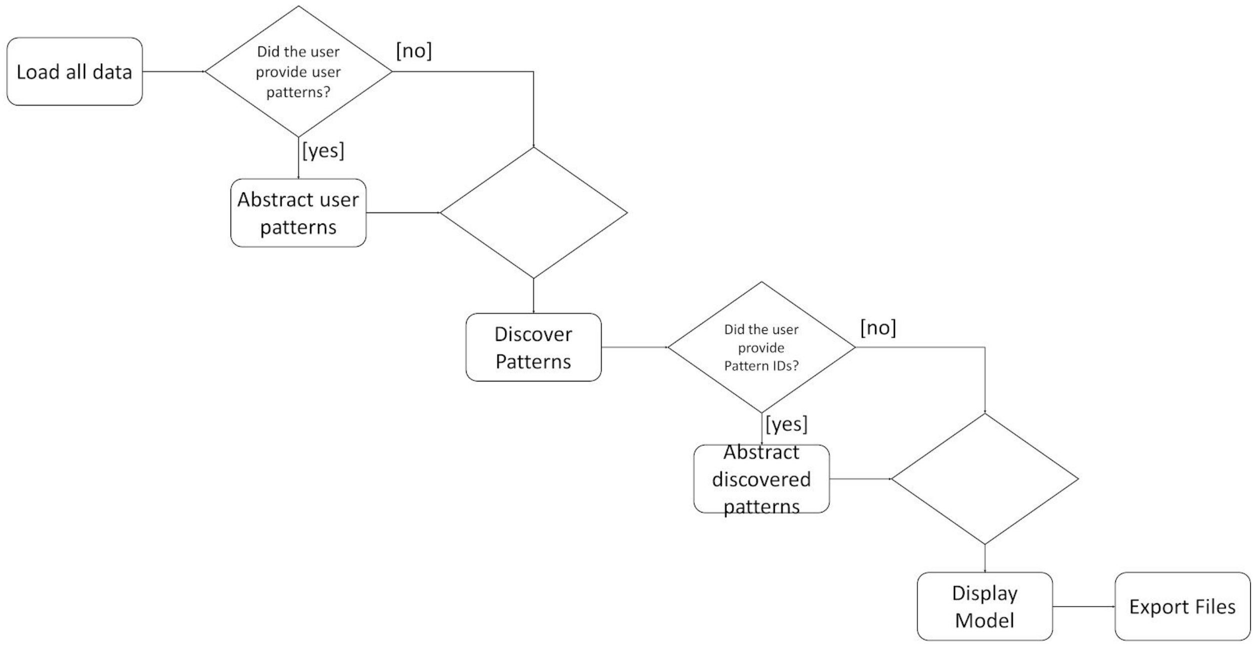
```
**************************************USER PATTERNS****************************************************

        ID      |       Name  |       Pattern

        5       |       Group5  |     ['check ticket', 'decide']
        6       |       Group6  |     ['decide', 'reject request']

Group5 ['check ticket', 'decide']
Group6 ['decide', 'reject request']
```

After the user patterns are abstracted, the program continues with the discovery step as described above (Page 24).


Load all data
Did the user provide user patterns?
[no]
[yes]
Abstract user patterns
Discover Patterns
Did the user provide Pattern IDs?
[no]
[yes]
Abstract discovered patterns
Display Model
Export Files

One can see a typical run through the main program above. The user can decide which path through the program he wants to take. The table below illustrates various user scenarios and the corresponding output:

Did the user provide Pattern IDs?	**Did the user provide user patterns?**	**Output**
Yes	No	The original log is abstracted with the patterns that are specified in the dialogue
No	No	The original log is returned including lifecycle events
Yes	Yes	The original log is first abstracted by the user patterns, a second abstraction step is made with the patterns selected after the discovery step
No	Yes	The original log is only abstracted by the user patterns, all discovered patterns are ignored

Discovery Module (discover.py)

Description

The Discovery Module is realized in the discover.py file. Given a log file, it discovers the maximal repeats, tandem arrays, and their variations that could be relevant for abstraction. The module outputs a .json file named file name followed by the suffix ‘_patterns.json'. The JSON file contains all discovered patterns, as well as, the attributes **ID**, **Name**, **PatternAlphabet**, **Type**, **Pattern**, and the measure called **InstanceCount**.

Required conditions

- A valid log file in XES or CSV format

Usage

discover.py [-h] --path xes_file_name

To transform the logs with the pattern. The --path and --pattern and their arguments are mandatory to run

-h, --help		show this help message and exit
--path	xes_file_name	specify the path of xes file. e.g. test.xes

Example

Command:

```
$ python discover.py --help
```

Result:

```
Usage:
 python discover.py --path example.XES

path:   The path of the input event log in xes format, required argument
```

Command:

```
$ python discover.py --path ./testCases/running-example.xes
```

Result:

```
See testCases/running-example_patterns.json file for a list of found patterns
```

Abstraction Module (abstraction.py)

Description

To abstract a given log with a sequence of patterns. The patterns can be only selected from the results obtained from the discover.py, i.e., maximal repeats and/or tandem arrays. The available patterns are assumed to be stored in a valid pattern file.

Please note that this is mainly to display the abstracted event logs along with timestamps without transforming into a file and process model.

Required conditions

- A valid log file in XES or CSV format
- A valid pattern file in JSON format. Named after the log followed by the suffix '_patterns.json'

  This file can be achieved by running the **discovery module (discovery.py)** or directly prepared **by the user** in the same format of the pattern file.

For example, executing the abstraction over "running-example.xes", "**running-example_patterns.json**" is required. This pattern file can be produced by the discovery module (discovery.py).

Usage

abstraction.py [-h] --path xes_file_name --pattern pattern_ID

To transform the logs with the pattern. The **--path** and **--pattern** and their arguments are mandatory to run

-h, --help		show this help message and exit
--path	xes_file_name	specify the path of xes file. e.g. test.xes
--pattern	pattern_ID	Referring to the xes_file_name.pattern, specify the sequence of the pattern in order which you want to make abstractions with. e.g. 1 2 3 1

Example

1) To show the usages of this module.
Command:

```
$ python ./abstraction.py --help
```

Result:

```
usage: abstraction.py [-h] --path xes_file_name --pattern pattern_ID
                      [pattern_ID ...]

To transform the logs with the pattern. The --path and --pattern are
mandatory.

optional arguments:
  -h, --help            show this help message and exit
  --path xes_file_name  specify the path of xes file. e.g. test.xes
  --pattern pattern_ID [pattern_ID ...]
                        Referring to the xes_file_name.pattern, specify the
                        sequence of the pattern in order which you want to
                        make abstractions with. e.g. 1 2 3 1
```

2) Example of abstraction

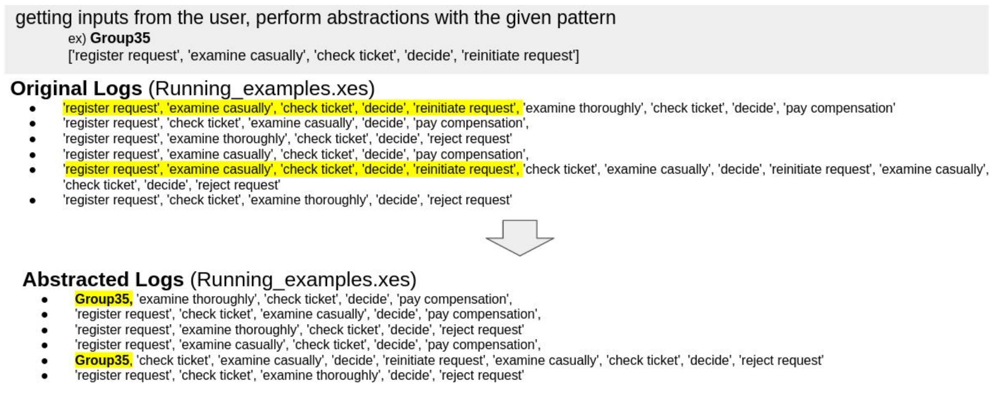
Command:

```
$ python ./abstraction.py --path ./testCases/running-example.xes --pattern 35
```

It performs abstractions with file **running-examples.xes** in the testCases folder by the **pattern 35**.

Group35 ['register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request']
then it shows the original log and abstracted log in a row.

You can replace the **--pattern** value with a sequence of unique **IDs** in a pattern file. The abstraction is performed on the given pattern IDs sequentially.



getting inputs from the user, perform abstractions with the given pattern
ex) **Group35**
['register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request']

Original Logs (Running_examples.xes)

- 'register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request', 'examine thoroughly', 'check ticket', 'decide', 'pay compensation'
- 'register request', 'check ticket', 'examine casually', 'decide', 'pay compensation',
- 'register request', 'examine thoroughly', 'check ticket', 'decide', 'reject request'
- 'register request', 'examine casually', 'check ticket', 'decide', 'pay compensation',
- 'register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request', 'check ticket', 'examine casually', 'decide', 'reinitiate request', 'examine casually', 'check ticket', 'decide', 'reject request'
- 'register request', 'check ticket', 'examine thoroughly', 'decide', 'reject request'

Abstracted Logs (Running_examples.xes)

- **Group35,** 'examine thoroughly', 'check ticket', 'decide', 'pay compensation',
- 'register request', 'check ticket', 'examine casually', 'decide', 'pay compensation',
- 'register request', 'examine thoroughly', 'check ticket', 'decide', 'reject request'
- 'register request', 'examine casually', 'check ticket', 'decide', 'pay compensation',
- **Group35**, 'check ticket', 'examine casually', 'decide', 'reinitiate request', 'examine casually', 'check ticket', 'decide', 'reject request'
- 'register request', 'check ticket', 'examine thoroughly', 'decide', 'reject request'

Original traces

To identify the different traces, **"|"** is used as a delimiter.

Result:

```
original traces
 ['register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request', 'examine thoroughly',
'check ticket', 'decide', 'pay compensation', '|', 'register request', 'check ticket', 'examine casually',
'decide', 'pay compensation', '|', 'register request', 'examine thoroughly', 'check ticket', 'decide', 'reject
request', '|', 'register request', 'examine casually', 'check ticket', 'decide', 'pay compensation', '|', 'register
request', 'examine casually', 'check ticket', 'decide', 'reinitiate request', 'check ticket', 'examine casually',
'decide', 'reinitiate request', 'examine casually', 'check ticket', 'decide', 'reject request', '|', 'register
request', 'check ticket', 'examine thoroughly', 'decide', 'reject request', '|']

register request          2010-12-30 14:32:00+01:00
examine casually          2010-12-30 15:06:00+01:00
check ticket              2010-12-30 16:34:00+01:00
decide                    2011-01-06 09:18:00+01:00
reinitiate request        2011-01-06 12:18:00+01:00
examine thoroughly        2011-01-06 13:06:00+01:00
check ticket              2011-01-08 11:43:00+01:00
decide                    2011-01-09 09:55:00+01:00
pay compensation          2011-01-15 10:45:00+01:00
|                         | <- delimiter of traces
register request          2010-12-30 11:32:00+01:00
check ticket              2010-12-30 12:12:00+01:00
examine casually          2010-12-30 14:16:00+01:00
decide                    2011-01-05 11:22:00+01:00
pay compensation          2011-01-08 12:05:00+01:00
|                         |
register request          2010-12-30 11:02:00+01:00
examine thoroughly        2010-12-31 10:06:00+01:00
check ticket              2011-01-05 15:12:00+01:00
decide                    2011-01-06 11:18:00+01:00
reject request            2011-01-07 14:24:00+01:00
|                         |
register request          2011-01-06 15:02:00+01:00
examine casually          2011-01-06 16:06:00+01:00
check ticket              2011-01-07 16:22:00+01:00
decide                    2011-01-07 16:52:00+01:00
pay compensation          2011-01-16 11:47:00+01:00
|                         |
register request          2011-01-06 09:02:00+01:00
examine casually          2011-01-07 10:16:00+01:00
check ticket              2011-01-08 11:22:00+01:00
```

```
decide                        2011-01-10 13:28:00+01:00
reinitiate request            2011-01-11 16:18:00+01:00
check ticket                  2011-01-14 14:33:00+01:00
examine casually              2011-01-16 15:50:00+01:00
decide                        2011-01-19 11:18:00+01:00
reinitiate request            2011-01-20 12:48:00+01:00
examine casually              2011-01-21 09:06:00+01:00
check ticket                  2011-01-21 11:34:00+01:00
decide                        2011-01-23 13:12:00+01:00
reject request                2011-01-24 14:56:00+01:00
|                             |
register request              2011-01-06 15:02:00+01:00
check ticket                  2011-01-07 12:06:00+01:00
examine thoroughly            2011-01-08 14:43:00+01:00
decide                        2011-01-09 12:02:00+01:00
reject request                2011-01-12 15:44:00+01:00
|                             |
```

Abstracted traces

In the abstracted traces, the two timestamps are presented for abstraction symbols. The 1st timestamp is the one from the first activity in the abstraction, and 2nd is to show the last activity in the designated group.

For example, the first timestamp **2010-12-30 14:32:00+01:00** is to show the first activity in **Group 35**, whereas the timestamp **2011-01-06 12:18:00+01:00** is the one of the last activity in **Group 35**.

Result:

```
 ['Group35', 'examine thoroughly', 'check ticket', 'decide', 'pay compensation', '|', 'register request', 'check ticket', 'examine casually', 'decide', 'pay compensation', '|', 'register request', 'examine thoroughly', 'check ticket', 'decide', 'reject request', '|', 'register request', 'examine casually', 'check ticket', 'decide', 'pay compensation', '|', 'Group35', 'check ticket', 'examine casually', 'decide', 'reinitiate request', 'examine casually', 'check ticket', 'decide', 'reject request', '|', 'register request', 'check ticket', 'examine thoroughly', 'decide', 'reject request', '|']

Group35                       2010-12-30 14:32:00+01:00    2011-01-06 12:18:00+01:00
examine thoroughly            2011-01-06 13:06:00+01:00    timestamp of last activity in Group35
check ticket                  2011-01-08 11:43:00+01:00
decide                        2011-01-09 09:55:00+01:00
pay compensation              2011-01-15 10:45:00+01:00
|                             |
register request              2010-12-30 11:32:00+01:00
```

```
check ticket              2010-12-30 12:12:00+01:00
examine casually          2010-12-30 14:16:00+01:00
decide                    2011-01-05 11:22:00+01:00
pay compensation          2011-01-08 12:05:00+01:00
|                         |
register request          2010-12-30 11:02:00+01:00
examine thoroughly        2010-12-31 10:06:00+01:00
check ticket              2011-01-05 15:12:00+01:00
decide                    2011-01-06 11:18:00+01:00
reject request            2011-01-07 14:24:00+01:00
|                         |
register request          2011-01-06 15:02:00+01:00
examine casually          2011-01-06 16:06:00+01:00
check ticket              2011-01-07 16:22:00+01:00
decide                    2011-01-07 16:52:00+01:00
pay compensation          2011-01-16 11:47:00+01:00
|                         |
Group35                   2011-01-06 09:02:00+01:00   2011-01-11 16:18:00+01:00
check ticket              2011-01-14 14:33:00+01:00
examine casually          2011-01-16 15:50:00+01:00
decide                    2011-01-19 11:18:00+01:00
reinitiate request        2011-01-20 12:48:00+01:00
examine casually          2011-01-21 09:06:00+01:00
check ticket              2011-01-21 11:34:00+01:00
decide                    2011-01-23 13:12:00+01:00
reject request            2011-01-24 14:56:00+01:00
|                         |
register request          2011-01-06 15:02:00+01:00
check ticket              2011-01-07 12:06:00+01:00
examine thoroughly        2011-01-08 14:43:00+01:00
decide                    2011-01-09 12:02:00+01:00
reject request            2011-01-12 15:44:00+01:00
|                         |
```

Transformation Module (transformation.py)

Description

Transformation.py is used to transform a given log with the abstracted patterns. The input file could be CSV or XES format. The final output of this module would be the transformed event log which is always in XES format, having the **start and complete lifecycle events** and a graphical process model. To run transfromation.py, please follow the instructions step by step.

Required conditions

- A valid log file in XES or CSV format
- Corresponding pattern **IDs in order** from patterns.json file that we obtained from the discovery module.

For more detail, please refer to the "Required conditions" of "Abstractions module" (Page 29)

Usage

transformation.py [-h] --path xes_file_name --pattern pattern_ID

To transform the logs with the pattern. The --path and --pattern and their arguments are mandatory to run

-h, --help		show this help message and exit
--path	xes_file_name	specify the path of xes file. e.g. test.xes
--pattern	pattern_ID	Referring to the xes_file_name.pattern, specify the sequence of the pattern in order which you want to make abstractions with. e.g. 1 2 3 1

You can also find help about the parameters by using **--help** command

Example

Command:

```
$ python ./transformation.py --help
```

Result:

```
usage: transformation.py [-h] --path xes_file_name --pattern pattern_ID
```

```
                    [pattern_ID ...] [--downloadLog] [--downloadModel]

To transform the logs with the pattern. The --path and --pattern number are mandatory.

optional arguments:
  -h, --help              show this help message and exit
  --path xes_file_name   specify the path of xes file. e.g. test.xes
  --pattern pattern_ID   [pattern_ID ...]
                          Referring to the xes_file_name.pattern, specify the
                          sequence of the pattern in order which you want to
                          make abstractions with. e.g. 1 2 3 1
  --downloadLog           flag to export/download transformed log
  --downloadModel         flag to export/download process model
```

You can skip step 1 and 2 if you already executed abstraction module

Step 1 (Discovery):
Run discover.py to get sets of tandem arrays and maximal repeats as pattern discovery.

NOTE: This pattern file can be directly prepared by the user in the form of a pattern file. For more information about pattern files, please refer to page 15.

Command:

```
$ python ./discovery.py --path ./testCases/running-example.xes
```

Result:

```
See ./testCases/running-example_patterns.json file for a list of found patterns
```

Here **running-example_patterns.json** is created that contains a set of tandem arrays and maximal repeats, each having a unique pattern number (ID) which could be used in step 2.

Step 2 (Transformation):
Run transformation.py to get a transformed log and graphical process model. The transformation module internally runs an abstraction module to get abstracted traces and their timestamps.

Command:

```
$ python ./transformation.py --path ./testCases/running-example.xes --pattern 35
```

Result:

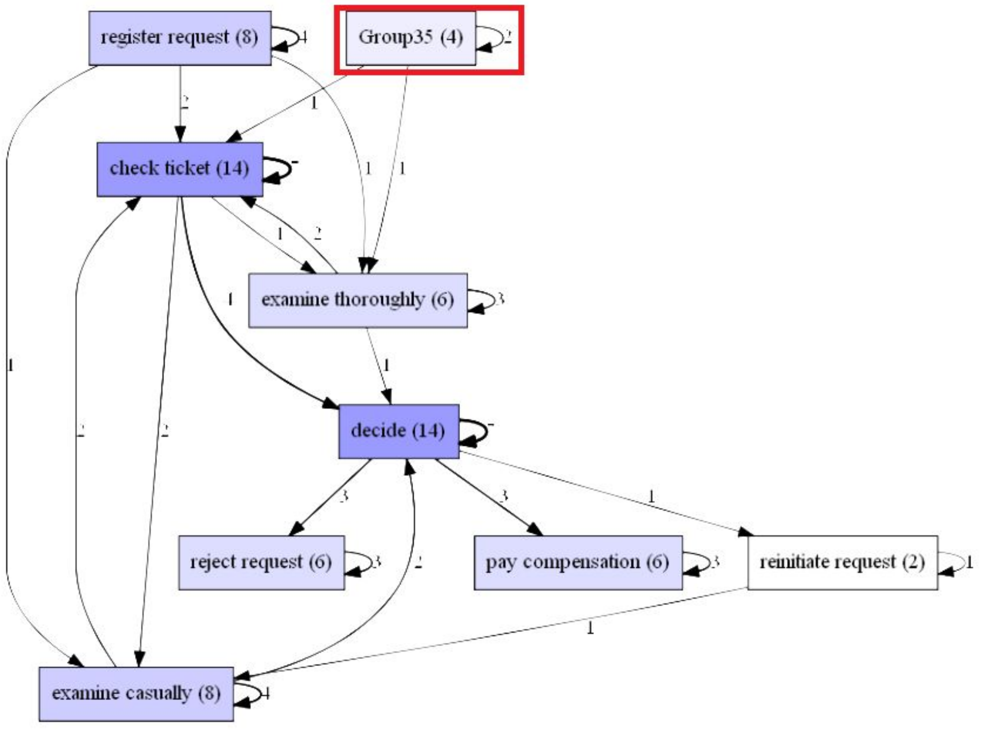
```
Group35 ['register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request']

************************************************************************************************************************
Log is transformed
************************************************************************************************************************

************************************************************************************************************************
Process model is generated for transformed log (Popping up)
************************************************************************************************************************

************************************************************************************************************************
Log exported as .\testCases\running-example_transformed_exported.XES
************************************************************************************************************************

************************************************************************************************************************
Process Model exported as .\testCases\running-example_transformed_exported.XES
************************************************************************************************************************
```



Group35 ['register request', 'examine casually', 'check ticket', 'decide', 'reinitiate request']

The event log is transformed and its process model is generated in the above screenshots. It can be observed that *Group35* is abstracted here. The parameters: **--downloadLog** and **--downloadModel** are optional parameters. It can be used if the user likes to export/ download transformed event log and/ or process models similar to the main module (Page 23).

FAQs

1. **I provided user patterns, why was the log not abstracted?**
   Please first check if the user pattern file is valid according to the specification given in this document. If this does not fix the problem, make sure that the user pattern file matches the log file.

2. **What happens if user patterns are provided and pattern IDs are not given in main.py?**
   The program will only generate the transformed log file with abstracted traces, based on user patterns. The principle is that only given inputs matter. You can also abstract the other way around, i.e., pattern IDs are inputted without user patterns.

3. **Can I run the transformation module directly without running any other modules?**
   Yes, you can run the transformation module directly but you need to provide a valid pattern file in JSON format. Provided that the pattern file can be generated by yourself or by getting from the discovery module.

4. **Can I compare two process models with your program?**
   This program does not provide such a comparison by itself, but you can manually do that by running the program twice. When getting the first model then rename it as you like. Then run the program a second time, you can have a second process model. Then you can compare both models.

Open Issues

- Currently, event logs with lifecycle transitions are not supported in a program, because the pm4py filter method loses the metadata information. PM4Py creates a new log object and only fills it with the traces all other information is lost.

- When activities are grouped the new end timestamp is the starting timestamp of the last activity instead of the end timestamp. This also leads to the issue that if we are first abstracting the user pattern and then discovering patterns, the end timestamp of the user patterns will be not as expected.

- The case ids and other attributes in events are not handled during the entire project.