# Używanie Pythona do miningu procesów

Python 3.x, opyenxes , pygraphviz ( lub grafviz ).

Do tej klasy możesz użyć dowolnego środowiska Pythona, które ma wyżej wymienione biblioteki.   
Można również użyć: [https://colab.research.google.com](https://web.archive.org/web/20200919222901/https:/colab.research.google.com/) .

**Implementacja prostego heurystycznego górnika**

Używając poniższego fragmentu kodu zaimportuj plik repairexample.xes do swojego skryptu Pythona:

from opyenxes.data\_in.XUniversalParser import XUniversalParser

path = 'repairExample.xes'

with open(path) as log\_file:

# parse the log

log = XUniversalParser().parse(log\_file)[0]

Spójrz na zmienną log. Używając log.get\_features () lub log.get\_attributes (), możesz sprawdzić pewne informacje o logu. Ponieważ przeanalizowany log składa się z list zdarzeń, możesz również wybrać pojedyncze zdarzenie i sprawdzić jego atrybuty:

event = log[0][0]

event.get\_attributes()

Aby ułatwić dalszą pracę, utworzymy workflow\_log składający się z nazw zdarzeń:

workflow\_log = []

for trace in log:

workflow\_trace = []

for event in trace[0::2]:

# get the event name from the event in the log

event\_name = event.get\_attributes()['Activity'].get\_value()

workflow\_trace.append(event\_name)

workflow\_log.append(workflow\_trace)

Aby utworzyć prostą sieć heurystyczną zadań (uproszczony model procesu jak w narzędziu Disco), utworzymy strukturę, w której dla każdego zdarzenia zbierzemy zbiór wszystkich zdarzeń poprzedzających to zdarzenie

w\_net = dict()

for w\_trace in workflow\_log:

for i in range(0, len(w\_trace)-1):

ev\_i, ev\_j = w\_trace[i], w\_trace[i+1]

if ev\_i not in w\_net.keys():

w\_net[ev\_i] = set()

w\_net[ev\_i].add(ev\_j)

Przyjrzyj się bliżej słownikowi w\_net:

{'Analyze Defect': {'Inform User', 'Repair (Complex)', 'Repair (Simple)'},

'Archive Repair': {'End'},

'Inform User': {'Archive Repair', 'End', ...},

...}

Reprezentuje powiązania między zdarzeniami:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Analyze Defect | Archive Repair | Inform User | … | End |
| Analyze Defect |  |  | → |  |  |
| Archive Repair |  |  |  |  | → |
| Inform User | → |  |  |  | → |
| … |  |  |  |  |  |
| End |  |  |  |  |  |

Używając [Pygraphviz](https://web.archive.org/web/20200919222901/https:/pygraphviz.github.io/), możemy wygenerować obrazek pokazujący proces:

import pygraphviz as pgv

G = pgv.AGraph(strict=False, directed=True)

G.graph\_attr['rankdir'] = 'LR'

G.node\_attr['shape'] = 'Mrecord'

for event in w\_net:

G.add\_node(event, style="rounded,filled", fillcolor="#ffffcc")

for preceding in w\_net[event]:

G.add\_edge(event, preceding)

G.draw('simple\_heuristic\_net.png', prog='dot')

Jeśli nie masz programu pygraphviz , możesz użyć graphviz.

# Ulepszanie diagramu

W Disco mogliśmy zobaczyć częstotliwości zadań. Policzmy taką częstotliwość:

ev\_counter = dict()

for w\_trace in workflow\_log:

for ev in w\_trace:

ev\_counter[ev] = ev\_counter.get(ev, 0) + 1

Następnie w naszym modelu możemy po prostu zmienić etykietę, aby zawierała wynik obliczeń:

text = event + ' (' + str(ev\_counter[event]) + ")"

G.add\_node(event, label=text, style="rounded,filled", fillcolor="#ffffcc")

Możemy również zmienić przezroczystość odkrytych zadań na podstawie ich częstotliwości:

color\_min = min(ev\_counter.values())

color\_max = max(ev\_counter.values())

G = pgv.AGraph(strict=False, directed=True)

G.graph\_attr['rankdir'] = 'LR'

G.node\_attr['shape'] = 'Mrecord'

for event in w\_net:

value = ev\_counter[event]

color = int(float(color\_max-value)/float(color\_max-color\_min)\*100.00)

my\_color = "#ff9933"+str(hex(color))[2:]

G.add\_node(event, style="rounded,filled", fillcolor=my\_color)

for preceding in w\_net[event]:

G.add\_edge(event, preceding)

G.draw('simple\_heuristic\_net\_with\_colors.png', prog='dot')

Możemy również spróbować odkryć zdarzenia początkowe i końcowe i poprawić model:

from functools import reduce

ev\_source = set(w\_net.keys())

ev\_target = reduce(lambda x,y: x|y, w\_net.values())

ev\_start\_set = ev\_source - ev\_target

print("start set: {}".format(ev\_start\_set))

ev\_end\_set = ev\_target - ev\_source

print("end set: {}".format(ev\_end\_set))

for ev\_end in ev\_end\_set:

end = G.get\_node(ev\_end)

end.attr['shape']='circle'

end.attr['label']=''

G.add\_node("start", shape="circle", label="")

for ev\_start in ev\_start\_set:

G.add\_edge("start", ev\_start)

G.draw('simple\_heuristic\_net\_with\_events.png', prog='dot')

# graphviz zamiast pygraphviz

Można używać graphviz zamiast pygraphviz , ale ma on inną składnię, np.:

import graphviz

G = graphviz.Digraph()

for event in w\_net:

G.node(event, style="rounded,filled", fillcolor="#ffffcc")

for preceding in w\_net[event]:

G.edge(event, preceding)

G.graph\_attr['rankdir'] = 'LR'

G.node\_attr['shape'] = 'Mrecord'

G.edge\_attr.update(penwidth='2')

G.node("End", shape="circle", label="")

G.render('simple\_graphviz\_graph')

display(G)

Obraz zawierający linia, diagram, rysowanie, szkic

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

# Polecenia

Rozszerz odkrywanie procesów o dodatkowe funkcje:

Spróbuj odkryć częstotliwość każdego przejścia (przepływu) i wyświetl liczbę wystąpień zarówno w postaci etykiety, jak i grubości linii.

Dodaj opcję filtrowania, aby wyświetlać lub ukrywać zadania lub przepływy zgodnie z wybranym progiem.

Spróbuj wdrożyć i odkryć relacje zgodnie z algorytmem Alfa (ta część jest opcjonalna).

# Algorytm Alfa

Uproszczony algorytm Alpha Aby eksplorować procesy biznesowe powszechnie znany algorytm Alpha został uproszczony i dostosowany do odkrycia modelu procesu BPMN. Rozważmy następujące ślady (warianty) pewnego dziennika przepływu pracy:

a b c d e g

a b c d f g

a c b d e g

a c b d f g

W przypadku tego dziennika przepływu pracy, w oparciu o algorytm Alpha, możemy znaleźć następujące relacje (x,y - zdarzenia):

(>) bezpośrednia sukcesja: x>y jeśli w pewnym przypadku x jest bezpośrednio następujące po y,

(→) przyczynowość: x→y jeśli x>y i nie y>x,

(||) paralelizm: x||y jeśli x>y i y>x,

(#) wybór/niezwiązany/brak bezpośredniej sukcesji: x#y jeśli nie x>y i nie y>x.

W poniższej implementacji nie używamy bezpośredniej relacji „brak bezpośredniej sukcesji”, ponieważ można po prostu sprawdzić, czy istnieje przyczynowość lub paralelizm.

direct\_succession = {

'a': {'b','c'},

'b': {'c','d'},

'c': {'b','d'},

'd': {'e','f'},

'e': {'g'},

'f': {'g'}

}

causality = {

'a': {'b', 'c'},

'b': {'d'},

'c': {'d'},

'd': {'e','f'},

'e': {'g'},

'f': {'g'}

}

parallel\_events = {('b', 'c'), ('c', 'b')}

możemy także znaleźć następujące zestawy zdarzeń początkowych i końcowych:

start\_set\_events = {'a'}

end\_set\_events = {'g'}

Na potrzeby kodowania możemy również zdefiniować odwróconą przyczynowość dla tych zdarzeń w relacji przyczynowości, które mają tylko jednego następnika (nie jest to część samego algorytmu alfa, ale może być pomocna jako tymczasowa zmienna do wykorzystania wzorców eksploracji alfa):

inv\_causality = {

'd': {'b', 'c'},

'g': {'e', 'f'}

}

Następnie użyj poniższego kodu, aby wygenerować prosty model procesu BPMN:

import graphviz

class MyGraph(graphviz.Digraph):

def \_\_init\_\_(self, \*args):

super(MyGraph, self).\_\_init\_\_(\*args)

self.graph\_attr['rankdir'] = 'LR'

self.node\_attr['shape'] = 'Mrecord'

self.graph\_attr['splines'] = 'ortho'

self.graph\_attr['nodesep'] = '0.8'

self.edge\_attr.update(penwidth='2')

def add\_event(self, name):

super(MyGraph, self).node(name, shape="circle", label="")

def add\_and\_gateway(self, \*args):

super(MyGraph, self).node(\*args, shape="diamond",

width=".6",height=".6",

fixedsize="true",

fontsize="40",label="+")

def add\_xor\_gateway(self, \*args, \*\*kwargs):

super(MyGraph, self).node(\*args, shape="diamond",

width=".6",height=".6",

fixedsize="true",

fontsize="35",label="×")

def add\_and\_split\_gateway(self, source, targets, \*args):

gateway = 'ANDs '+str(source)+'->'+str(targets)

self.add\_and\_gateway(gateway,\*args)

super(MyGraph, self).edge(source, gateway)

for target in targets:

super(MyGraph, self).edge(gateway, target)

def add\_xor\_split\_gateway(self, source, targets, \*args):

gateway = 'XORs '+str(source)+'->'+str(targets)

self.add\_xor\_gateway(gateway, \*args)

super(MyGraph, self).edge(source, gateway)

for target in targets:

super(MyGraph, self).edge(gateway, target)

def add\_and\_merge\_gateway(self, sources, target, \*args):

gateway = 'ANDm '+str(sources)+'->'+str(target)

self.add\_and\_gateway(gateway,\*args)

super(MyGraph, self).edge(gateway,target)

for source in sources:

super(MyGraph, self).edge(source, gateway)

def add\_xor\_merge\_gateway(self, sources, target, \*args):

gateway = 'XORm '+str(sources)+'->'+str(target)

self.add\_xor\_gateway(gateway, \*args)

super(MyGraph, self).edge(gateway,target)

for source in sources:

super(MyGraph, self).edge(source, gateway)

G = MyGraph()

# adding split gateways based on causality

for event in causality:

if len(causality[event]) > 1:

if tuple(causality[event]) in parallel\_events:

G.add\_and\_split\_gateway(event,causality[event])

else:

G.add\_xor\_split\_gateway(event,causality[event])

# adding merge gateways based on inverted causality

for event in inv\_causality:

if len(inv\_causality[event]) > 1:

if tuple(inv\_causality[event]) in parallel\_events:

G.add\_and\_merge\_gateway(inv\_causality[event],event)

else:

G.add\_xor\_merge\_gateway(inv\_causality[event],event)

elif len(inv\_causality[event]) == 1:

source = list(inv\_causality[event])[0]

G.edge(source,event)

# adding start event

G.add\_event("start")

if len(start\_set\_events) > 1:

if tuple(start\_set\_events) in parallel\_events:

G.add\_and\_split\_gateway(event,start\_set\_events)

else:

G.add\_xor\_split\_gateway(event,start\_set\_events)

else:

G.edge("start",list(start\_set\_events)[0])

# adding end event

G.add\_event("end")

if len(end\_set\_events) > 1:

if tuple(end\_set\_events) in parallel\_events:

G.add\_and\_merge\_gateway(end\_set\_events,event)

else:

G.add\_xor\_merge\_gateway(end\_set\_events,event)

else:

G.edge(list(end\_set\_events)[0],"end")

G.render('simple\_graphviz\_graph')

G.view('simple\_graphviz\_graph')

Alpha Plus Alpha+ to proste rozszerzenie algorytmu Alpha, które może obsługiwać krótkie pętle i pętle własne. Na poniższym rysunku modele po lewej stronie to modele odkryte przez algorytm Alpha. Modele po prawej stronie to modele, które można było odkryć za pomocą algorytmu Alpha+.

Obraz zawierający tekst, diagram, zrzut ekranu, Czcionka

Zawartość wygenerowana przez sztuczną inteligencję może być niepoprawna.

W Internecie znajdują się artykuły o algorytmach wykorzystujących sieci Petriego. Nie ma prostego tłumaczenia z sieci Petriego na BPMN, jednak w przypadku Alpha+ sytuacja jest podobna do tej w przypadku Heuristic Miner. Więc w celu odkrycia pętli własnych i krótkich pętli należy skonstruować dodatkową macierz i wykluczyć odkryte relacje z macierzy śladu algorytmu Alpha.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | … |
| aτa |  | ✓ |  |  |
| bτb |  |  |  |  |
| cτc |  |  |  |  |
| … |  |  |  |  |

Wskazówka: może Ci się przydać ta strona: <https://processmining.org/old-version/event-book.html>