PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
PRÓ-REITORIA DE PESQUISA, PÓS-GRADUAÇÃO E INOVAÇÃO
PROGRAMA INSTITUCIONAL DE BOLSAS DE INICIAÇÃO CIENTÍFICA

PIBIC

RELATÓRIO FINAL

MINERAÇÃO DE PROCESSOS E SIMULAÇÃO

EDSON EMILIO SCALABRIN

JAIR JOSÉ FERRONATO

CURITIBA

2021

GABRIEL SCHOLZE ROSA
EDSON EMILIO SCALABRIN

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO – ESCOLA POLITÉCNICA
BOLSA PUCPR

MINERAÇÃO DE PROCESSOS E SIMULAÇÃO

Relatório Final apresentado ao Programa Institucional de Bolsas de Iniciação Científica e Tecnológica, Pró-Reitoria de Pesquisa, Pós-Graduação e Inovação da Pontifícia Universidade Católica do Paraná, sob orientação do(a) Prof. Edson Emilio Scalabrin.

Sumário

SUMÁRIO

RESUMO

Quando se trata de inteligência de negócios, a mineração de processos em conjunto com a simulação são elementos essenciais, pois elas moldam os processos de modo a otimizá-los, tornando sua execução mais rápida e, muitas vezes, gastando menos recursos. Isto pode ser percebido em qualquer parte do cotidiano, como nas rotas de ônibus, que pode ser analisada a taxa de chegada de ônibus e as rotas que assumem de modo a diminuir o tempo de percurso, sugerindo rotas mais rápidas. Assim, o presente artigo propõe uma análise dos métodos presentes de simulação de processo, principalmente por meio da biblioteca PM4PY (Process Mining for Python), em colaboração com a área de simulação, estudada através da biblioteca de Simpy, de modo a comparar tempos de execução registrados com os simulados, assim melhorando ao máximo o período necessário para executar toda uma bateria de atividades. Mineração de processos foi estudada por meio de alguns desafios, testando as funcionalidades da biblioteca, tanto por filtragem de dados, descoberta de processos e checagem de conformidade até chegar a última etapa do estudo: uma tentativa de simulação de um log utilizando a própria biblioteca de mineração de processos, para assim comparar os tempos de execução registrados com os tempo da simulação, comparação frustrada pela falta de tratamento de certas atividades por parte da biblioteca, assim iniciando os estudos de simulação, que se desenvolveram bem, mas a aplicação principal não foi desenvolvida, houve somente a compreensão acerca das funcionalidades. Em suma, boa parte da análise proposta acerca dos dados presentes nos registros de eventos foi satisfatória por meio da biblioteca de mineração de processos, porém em relação ao tempo nem todos os objetivos previstos foram atingidos, mesmo que os estudos tenham sido instrutivos. Acima de tudo, o mais importante aqui é compreender bem o assunto e as bibliotecas

Palavras-chave: Mineração de processos. Simulação de Processos. Inteligência de Negócios.

1. INTRODUÇÃO

A mineração de processos fornece meios para descobrir conhecimentos acerca dos modelos de negócios, assim podendo checar sua conformidade e encontrar problemas durantes os processos, possibilitando a sugestão de melhorias nestas atividades, analisando o ambiente e os objetivos que querem ser alcançados com tal modelo. Ao passar dos anos cada vez mais logs de eventos são produzidos com base em processos do mundo real, do dia a dia do ser humano (D. R. FERREIRA, 2017, p. 65) assim tornando a mineração de processos cada vez mais importante, pois ela pode otimizar até o mais básico da vivência humana. O trabalho de mineração de processos é feito utilizando logs de eventos que são registros com data e hora, nome das atividades e outras informações sobre o processo que está sendo realizado. Por exemplo, em um log de eventos de uma rede de fast-food, ocorre a solicitação da confecção de um sanduíche, que inicia uma série de eventos, como a seleção do tipo de pão, da carne, molhos e saladas. Realizando uma mineração de processos nos processos que ocorrem no fast-food é possível a análise de qual processo mais leva tempo para otimizar a produção economizando o máximo de recursos de forma que o lucro do estabelecimento seja maximizado.

A partir do que foi descoberto na mineração de processos, é possível partir para a área da simulação, aplicando as melhorias sugeridas na etapa de mineração e obtendo os resultados de forma a melhorar o processo. Tomando como base o log de eventos do estabelecimento mencionado, podemos fazer uma simulação do processo de produção de lanches, analisando os recursos e como eles são gastos (carnes, saladas, molhos e bebidas), assim aplicando as melhorias sugeridas pela etapa de mineração de processos.

Assim como na indústria alimentícia, os conceitos de mineração de processos e simulação podem ser aplicados em várias áreas. Dentre elas temos o transporte, onde é possível a execução de uma análise completa sobre as rotas de uma linha de transporte público, por exemplo, de modo a realizar a rota mais rápida até cada um dos destinos, a área da saúde, que pode ajudar controlar de uma melhor maneira a distribuição das vacinas contra a COVID-19 em todo o país, de modo a tornar a vacinação mais eficiente, na área industrial pode-se também otimizar os processos para economizar recursos, por exemplo em um ambiente de construção

onde os recursos são contados para a obra não sair dos limites financeiros da construtora.

A mineração de processos é realizada pela biblioteca PM4PY (Process Mining for Python) enquanto a simulação é realizada pela biblioteca Simpy, ambas utilizando a linguagem de programação Python para operar. O presente artigo visa entender mais os conceitos de mineração de dados e simulação, estudando as duas bibliotecas citadas acima de modo a exemplificar de uma maneira mais simplória cada parte estudada.

2. OBJETIVOS

Analisar processos por meio da mineração destes processos e da simulação, de modo a sempre buscar meios para diminuir recursos gastos para o desenvolvimento do processo, tanto quanto diminuir o tempo necessário para sua execução.

3. REVISÃO DE LITERATURA

A mineração de processos tem como objetivo a extração de modelos de processos, retirados de logs de eventos. Assim é possível analisar os processos pelos dados gerados durante sua execução. As técnicas da mineração permitem as organizações a descobrir, monitorar e melhorar o desempenho de seu modelo de processo. A mineração de processos é muito popular na área de gerenciamento de processos de negócio, ou BPM (Business Process Management). Um processo é uma sequência de tarefas que são conduzidas para atingir um objetivo de negócio. (Lopes, 2017).

Considerando os logs de eventos, podemos conduzir três tipos diferentes de mineração de processos. Primeiro tem-se a técnica de descoberta (Discovery), que gera um modelo, com base no log de eventos, sem nenhuma informação “a-priori”. Por conseguinte, existe a técnica de conformidade (Conformance), onde seria avaliado o alinhamento entre um log de eventos e seu modelo de processos já existente. Por último, há a técnica de melhorar um modelo (Enhancement), que visa

aprimorar um modelo de processos com base na comparação dele com seu log de origem e seu modelo conceitual. A mineração de processos pode ter focos diferentes dependendo da perspectiva do processo. Como exemplo, há casos em que é priorizado a busca de caminhos (*paths*) otimizados para a realizar as tarefas, chamada de *Control-flow*. Não obstante, existe a *Organization,* que analisa os recursos que executam os processos, como os departamentos e as pessoas. Por fim existe o *Case*, que foca em casos específicos dentro dos logs de eventos, e o Time, que analisa o tempo e frequência dentro do log de eventos. (Barbon Jr, 2015). As técnicas listadas acima abrem caminhos quase infinitos para a quantidade de informações possível para se retirar do modelo. É possível extrair informações como as atividades realizadas no processo, data e hora das realizações das atividades, quem realizou qual atividade, são alguns exemplos do que é possível extrair por meio das técnicas acima.

Assim, detectando os problemas por meio da mineração de processos é possível partir para a simulação. Assim como na mineração, a simulação de casos torna possível o reconhecimento da modelagem correta, podendo propor alterações ao modelo originalmente utilizado. Se, em um processo, existem partes que nunca são executadas, existe um erro evidente ade modelagem que deve ser corrigido (Szimanski, 2013).

Uma simulação é feita com o objetivo de estimar o comportamento de um determinado evento, de uma estrutura ou de um processo, baseado nos dados inseridos no modelo. Assim, em um caso de simulação de eventos discretos, devemos avaliar prioritariamente as inserções (entradas ou inputs) feitas no processo e os resultados obtidos.

4. MATERIAIS E MÉTODO

Primeiramente foi realizado um estudo acerca dos conceitos de mineração de processos, para assim entender as principais ideias sobre eles. Isto foi feito por meio de artigos como *A Primer on Process Mining,* de Diogo R. Ferreira, e Time Prediction

Based on Process Mining, de W.M.P. van der Aalst, M.H. Schonenberg, and M. Song, além do slide Mineração de Processos de Sylvio Barbon Junior, do Departamento de Computação da Universidade Estadual de Londrina. Posteriormente, foi utilizada a linguagem de programação Python juntamente com as bibliotecas PM4PY (Process Mining for Python), para realizar a parte de mineração de processos, e Simpy, que é uma biblioteca dedicada a simulação dentro do Python.

Inicialmente, foi estudado o funcionamento do PM4PY, testando por meio de um log de eventos suas funcionalidades de filtragem de dados, descoberta de processos, checagem de conformidade, trabalhos estatísticos e com simulação. A parte de maior dificuldade foi comparar os tempos de atividade em média do log e da simulação realizada deste mesmo log, mesmo com os parâmetros de filtragem para obtenção do tempo estando iguais.

Logo após, foi realizado um estudo acerca do Simpy, que é outra biblioteca de Python dedicada a simulação, estudando as principais funções do Simpy, como a criação de eventos, interação dos processos, recursos e eventos em tempo real.

5. RESULTADOS

Os resultados serão divididos em duas seções: A primeira parte terá foco na Mineração de processo, falando especificamente da utilização da biblioteca PM4PY (Process Mining for Python), enquanto a segunda parte falará de simulação, também focando em outra biblioteca do Python, o Simpy.

5.1. PM4PY (Process Mining for Python)

Antes de iniciar as explicações com códigos, é importante esclarecer alguns conceitos. A mineração de processos abordada neste artigo trabalha com logs de eventos, nestes logs existem cases, que são cases que são casos, onde os eventos são agrupados. Existe também o *trace,* que está relacionado a sequência de atividades que ocorreram em cada case.

Inicialmente, os estudos acerca da biblioteca de mineração de processos em Python foram conduzidos por alguns “desafios” realizados para compreender melhor seu funcionamento. Os primeiros desafios foram mais simples, como importar logs,

contagem de eventos, listar as atividades, filtrar os casos com um tempo determinado, entre outros. Posteriormente, é chegado em uma parte mais importante, que é a descoberta de processos, na qual utilizamos três algoritmos de mineração de processos da biblioteca, o Alpha Miner, Heuristic Miner e Inductive Miner. Os três tem uma aplicação em código semelhante mas utiliza de conceitos diferentes. O Alpha Mineré o mais popular, possibilitanto a descoberta de uma rede de petri, um marco inicial que descreve o estado da rede de petri quando uma execução começa e um marco final, que descreve o mesmo estado da rede quando uma execução é finalizada. Já o Inductive Miner trabalha diferente, ele detecta ‘cortes’ em um log, repetindo nos sublogs encontrados ao aplicar o corte até que o caso base seja encontrado. Por fim, há o Heuristic Miner, que trabalha diretamento com o Directly-Follows Graph (Gráfico de Segmentos Diretos), que é um gráfico em que cada nodo represente um evento no log, fornecendo meios de achar construções comuns entre atividades. Assim, quando aplicado o algoritmo, é retornado uma rede chamada heuristic net que é um objeto que contem as atividades e os relacionamentos entre elas, podendo ser convertida para uma rede de petri comum. A seguir estão os modos de fazer a descoberta por meio de cada um dos algoritmos:

```
from pm4py.algo.discovery.alpha import algorithm as alpha_miner
net, initial_marking, final_marking = pm4py.discover_petri_net_alpha(log)

from pm4py.algo.discovery.inductive import algorithm as inductive_miner

net, initial_marking, final_marking = inductive_miner.apply(log)

from pm4py.algo.discovery.heuristics import algorithm as heuristics_miner
net, im, fm = heuristics_miner.apply(log,
parameters={heuristics_miner.Variants.CLASSIC.value.Parameters.DEPENDENCY_THRESH: 0.99})
```

Em seguida, foi realizado um estudo sobre a parte de checar a conformidade, comparando o modelo de processo com o log de eventos do mesmo processo, tendo duas técnicas para realizar essa verificação: Uma delas é o *token-based replay* e a outra é *alignments.*

De inicio abordaremos o *Token-based Replay* Este primeiro compara traces (conjuntos de eventos) com uma rede de petri para descobrir quais transições são executadas e qual dos *places* sobraram ou tem *tokens* faltando.

```
net, initial_marking, final_marking = alpha_miner.apply(log)
```

```
from pm4py.algo.conformance.tokenreplay import algorithm as token_replay
replayed_traces = token_replay.apply(log, net, initial_marking, final_marking)
```

A saída do token-base replay é armazenada no replayed_traces que contém várias informações, como se o trace está de acorda com o modelo, que retorna um valor verdadeiro ou falso, lista de transições ativadas, marcação de quando o replay for finalizado, tokens faltando, tokens sobrando, consumidos e produzidos.

Já o *Alignments* é um replay baseado em alinhamento tem o objetivo de encontrar o melhor alinhamento entre o trace e o modelo. Para cada trace existe uma lista de dois elementos, um é um evento do trace e o segundo é uma transição do modelo.

```
net, initial_marking, final_marking = inductive_miner.apply(log)
from pm4py.algo.conformance.alignments import algorithm as alignments
aligned_traces = alignments.apply_log(log, net, initial_marking, final_marking)
```

Posteriormente, foram estudadas alguns tipos de simulação, como a CTMC e a Monte Carlos Simulation. A CTMC é uma simulação que permite descobrir a probabilidade de um processo ser finalizado após um determinado tempo. A simulação acontece tomando como base as atividades iniciais e finais do processo.

```
dfg_perf = dfg_discovery.apply(log, variant=dfg_discovery.Variants.PERFORMANCE)
from pm4py.statistics.start_activities.log import get as start_activities
from pm4py.statistics.end_activities.log import get as end_activities
sa = start_activities.get_start_activities(log)
ea = end_activities.get_end_activities(log)
from pm4py.objects.stochastic_petri import ctmc
tang_reach_graph = ctmc.get_tangible_reachability_and_q_matrix_from_dfg_performance
(dfg_perf, parameters={"start_activities": sa, "end_activities": ea})
```

A simulação de Monte Carlo tem o mesmo objetivo que a CTMC que é prever o tempo para o processo ser finalizado. Porém, o faz tanto por meio do Directly-follows graph e do case arrival ratio. Case arrival ratio é a taxa de chegada dos cases, isto é, o tempo que passa entre a chegada de dois cases consecutivos.

```
from pm4py.algo.discovery.dfg import algorithm as dfg_discovery
dfg_perf = dfg_discovery.apply(log, variant=dfg_discovery.Variants.PERFORMANCE)
sa = start_activities.get_start_activities(log) #atividades iniciais do log
ea = end_activities.get_end_activities(log) #atividades finais do log
from pm4py.objects.conversion.dfg import converter
net, im, fm = converter.apply(dfg_perf,
variant=converter.Variants.VERSION_TO_PETRI_NET_ACTIVITY_DEFINES_PLACE,
```

```
   parameters={converter.Variants.VERSION_TO_PETRI_NET_ACTIVITY_DEFINES_PLACE.value.
9  Parameters.START_ACTIVITIES:sa,converter.Variants.VERSION_TO_PETRI_NET_ACTIVITY_
   DEFINES_PLACE.value.Parameters.END_ACTIVITIES: ea})
11 from pm4py.simulation.montecarlo import simulator as montecarlo_simulation
   from pm4py.algo.conformance.tokenreplay.algorithm import Variants
13 parameters = {}
   parameters[montecarlo_simulation.Variants.PETRI_SEMAPH_FIFO.value.Parameters.
15 TOKEN_REPLAY_VARIANT] = Variants.BACKWARDS
   parameters[montecarlo_simulation.Variants.PETRI_SEMAPH_FIFO.value.Parameters.
17 PARAM_CASE_ARRIVAL_RATIO] = 10800
   simulated_log, res = montecarlo_simulation.apply(log, net, im, fm,
parameters=parameters)
```

Como saída tem-se o *simulated_log* e o *res.* O simulated_log são os traces que foram simulados durante o processo de simulação. Já o res é o resultado da simulação, expresso em dicionário python. Com estes resultados é possível obter respostas mais específicas se tratando do tempo, por exemplo é possível filtrar uma transição aleatória no modelo que tem um numero de intervalo que estão se sobreponto a pontos que são distribuídos uniformemente por meio do intervalo de tempo do log.

```
1  import random
   last_timestamp = max(event["time:timestamp"] for trace in log for event in
3  trace).timestamp()
   first_timestamp = min(event["time:timestamp"] for trace in log for event in
5  trace).timestamp()
   n_div = 10
7  i = 0
   while i < n_div:
9     print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
      pick_place = random.choice(list(res["places_interval_trees"]))
11    timestamp = first_timestamp + (last_timestamp - first_timestamp)/n_div * i
      print("Pick_Place")
13    print(pick_place)
      print("Timestamp + tamanho do intervalo")
15    print("\t", timestamp, len(res["places_interval_trees"][pick_place]))
      i = i + 1
```

Após finalizar a parte de simulação abordada pelo PM4PY, retorna-se para a parte de mineração de processos, nisto entra o terceiro desafio. Primeiramente, foi preciso detectar os cases incompletos no log de eventos, os cases que não foram finalizados. Para cada case não finalizado, retorna-se a atividade que esta sendo executada e o tempo passado desde o inicio da execução do case. Para isso, foi desenvolvido um algoritmo em python puro, checando se a atividade final foi alcançada para identificar cases não finalizados e utilizando a função print para mostrar os resultados.

```
1   cwd = os.getcwd()
    filename = os.path.join(cwd, "patient_treatment.xes")
3   log = xes_importer.apply(filename)
```

```
logfiltrado = variants_filter.apply_auto_filter(log)
dfg, sa, ea = pm4py.discover_dfg(logfiltrado)
verificador = False #caso a atividade final seja encontrada, o valor fica True
count=0
currentTrace=0 #salva qual o trace atual
currentEvent=0 #salva qual o evento atual
for trace in log:
    currentEvent= 0 # zera o evento a cada trace, para começar do zero
    for event in trace:
        if event['concept:name']==next(iter(ea)): #filtra pela atividade final
            print("Processo Finalizado")
            verificador = True
            break
    if verificador==False:
        print("_____________________________________")
        print("Trace não finalizado:")
        print(trace)
        print("~~~~~~~~~~~~~~~")
        for event in trace:
            print("Atividade do evento: ")
            print(event['concept:name']) #printa cada atividade por evento
            print("Tempo de duração da atividade: ")
            if (currentEvent!=len(trace)-1):
                actualTime1 = str(event['time:timestamp'])
                nextTime1 = str(trace[currentEvent+1]['time:timestamp'])
                actualTime = actualTime1[0]
                nextTime = nextTime1[0]
                for i in range(1, 19):
                    actualTime = actualTime + actualTime1[i]
                    nextTime = nextTime + nextTime1[i]
                if(nextTime==actualTime):
                    print("0s")
                else:
                    from datetime import datetime
                    fmt = '%Y-%m-%d %H:%M:%S'
                    tstamp1 = datetime.strptime(str(nextTime), fmt)
                    tstamp2 = datetime.strptime(str(actualTime), fmt)
                    if tstamp1 > tstamp2:
                        td = tstamp1 - tstamp2
                    else:
                        td = tstamp2 - tstamp1
                    print(f"{td.total_seconds()} s")
                currentEvent += 1
                print("~~~~~~~~~~~~~~~")
            else:
                if(len(log)!= currentTrace-1):
                    actualTime1 = str(event['time:timestamp'])
                    nextTime1 = str(log[currentTrace+1][0]['time:timestamp'])
                    actualTime = actualTime1[0]
                    nextTime = nextTime1[0]
                    for i in range(1, 19):
                        actualTime = actualTime + actualTime1[i]
                        nextTime = nextTime + nextTime1[i]
                    if (nextTime == actualTime):
                        print("0s")
                    else:
                        from datetime import datetime
```

```
                        fmt = '%Y-%m-%d %H:%M:%S'
                        tstamp1 = datetime.strptime(str(nextTime), fmt)
                        tstamp2 = datetime.strptime(str(actualTime), fmt)
                        if tstamp1 > tstamp2:
                            td = tstamp1 - tstamp2
                        else:
                            td = tstamp2 - tstamp1
                        print(f"{td.total_seconds()} s")
                else:
                    print("0s")
                currentEvent += 1
                print("~~~~~~~~~~~~~~~")
        print("_____________________________________________")
        count += 1
        if count == 20:
            break #mantem apenas 20 traces sendo analisados acelerar processos
    else:
        currentTrace+=1;
        verificador = False
```

Após identificar os cases incompletos e retornar visualmente suas atividades e seu tempo, foi iniciada outra parte de simulação. Neste caso a tentativa era realizar a simulação de um log de eventos (foi usada a simulação de Monte Carlo) e comparar o tempo médio de execução da simulação com o do log original, depois mostrando o tempo médio para finalizar um case, tomando como base os eventos e atividades. Porém, foi aqui que surge a dificuldade. Quando comparados os tempos do log e da simulação gerada com este log base, estes tempos são muito diferentes. Ocorre que no log original, há acontecimentos registrados e para cada atividade, tem um tempo que ela demora para acontecer, depois de ser executada é registrada no log. A simulação Monte Carlo faz uma espécie de replay no modelo descoberto no log original e executa a simulação sem se preocupar com o tempo de cada atividade. Assim, o tempo vai ser menor que o log original. O PM4PY não faz ainda o tratamento dos atributos da simulação, ou seja, tempo de cada atividade, uso de recursos, entre outros. Logo não há meios de simular um log com os atributos da simulação, atributos que adicionam um tempo a mais à simulação, deixando este log simulado com um tempo sempre menor, pois apenas o replay foi executado.

Logo, o estudo do PM4PY foi encerrado e foi dado início ao estuda da biblioteca de simulação mais popular do Python, o Simpy.

5.2. SIMPY (Simulation with Python)

SimPy é um framework baseado em processos, que simula eventos utilizando bibliotecas nativas do Python. Os processos do Simpy são feitos por meio de funções geradoras que podem definir entidades na simulação, como compradores, veículos, entre outros.

O Simpy oferece a possibilidade de manipular simulações, de modo que elas sejam mais rápidas, em tempo real, ou manualmente, como um passo a passo, de modo a analisar cada parte do processo.

5.2.1. INSTALAÇÃO

O processo de instalação do Simpy é simples, basta abrir o menu iniciar do Windows e escrever **”CMD”**, assim abrindo o prompt de comandos do sistema operacional. Em seguida, digite o seguinte comando: **“pip install simpy”**.

Assim, a instalação será feita, lembrando que é necessário ter o *pip* instalado em seu computador para realizar a instalação. O Simpy tem suporte no Python 2 e Python 3.

5.2.2. ENVIRONMENTS

Um Environment, ou ambiente de simulação, gerencia o tempo da simulação e o agendamento e processamento de eventos, fornecendo meios para executar a simulação. A classe base para todos os ambientes é a *Environment,* mas quando se trata de simulações em tempo real, o Simpy fornece a *RealtimeEnvironment.*

O Simpy também provê meios de controlar a simulação. É possível executar a simulação até acabar os eventos, até que um tempo X seja atingido ou até que certo evento seja processado. O method mais importante é o *Environment.run().* O env.run() executa a simulação até que não sobrem eventos para executar. Porém,

na maioria dos casos é recomendado parar a simulação a determinado tempo, que pode ser determinado por meio do parâmetro *until:* `env.run(until=10).`

A simulação será parada assim que o relógio chegar em 10 e não irá processar algum evento agendado para o tempo 10.

Além disso, é possível percorrer evento por evento da simulação, por meio dos métodos *peek()* e *step():*

- peek(): Retorna o tempo do próximo evento agendado, ou infinito se não existirem eventos agendados;
- step(): Processa o próximo evento agendado, caso não exista um, é mostra um *EmpySchedule.*

O Simpy oferece um método que fornece o tempo atual da simulação, que é o *Environment.now,* que retorna um número que pode ser incrementado pelos eventos de *timeout.* Por padrão, a simulação começa em 0, mas é possível passar um initial_time (tempo inicial).

5.2.3. CRIAÇÃO DE EVENTOS

A criação de eventos normalmente necessita que seja importado o *simpy.events,* instanciado a classe de evento e passado ao ambiente. Porém, de modo a reduzir a escrita, o Simpy oferece alguns atalhos:

- **Environment.process()**

O process() é utilizado para instanciar eventos, ou classes de eventos, na simulação, no ambiente.

- **Environment.timeout()**

Esse método deixa a simulação aguardando por um tempo definido por parâmetro. Por exemplo, “env.timeout(5)” fará com que a simulação espere passar 5 unidades de tempo.

5.2.4. CODIGO BÁSICO SIMPY

Confira a seguir um exemplo para compreender como funciona a montagem básica de um código utilizando o Simpy.

```
1     import simpy
      def carro(environment):
3     while True:
          print('Estacionando as: %d' %environment.now)
5         tempo_estacionar = 3
          yield environment.timeout(tempo_estacionar)
7         print('Começa a dirigir as: %d' %environment.now)
          tempo_viagem = 5
9         yield environment.timeout(tempo_viagem)

11       env = simpy.Environment()
         env.process(carro(env))
13       env.run(until=20)
```

Tendo como saída no terminal o seguinte:

Estacionando as: 0

Começa a dirigir as: 3

Estacionando as: 8

Começa a dirigir as: 11

Estacionando as: 16

Começa a dirigir as: 19

Neste exemplo, tem-se um carro que muda de estado constantemente entre estacionando e dirigindo. Passamos um ***environment*** como referência na função carro para criar eventos.

Com o tempo iniciando em 0 por padrão, são adicionadas 3 unidades de tempo na linha 7 para o evento de estacionar, e 5 unidades de tempo para o evento

de dirigir, enquanto o *environment.now* retorna o tempo atual do ambiente. Na linha 12 instanciamos um novo environment que é passado na função *carro,* em seguida ele é iniciado e adicionado ao environment na linha 13. Por fim, na linha 14, chamamos a função run(), que inicia a simulação, passando um tempo limite para a simulação. A instancia do processo também pode ser utilizada para realizar interações no processo, como esperar um processo ser finalizado e interromper outro processo enquanto está esperando por um evento.

5.2.5. INTERAÇÃO DE PROCESSOS

Existem vários tipos de interação de processos, aqui veremos dois exemplos. Um deles é a espera de um processo ser finalizado, o outro é a interrupção de um processo.

5.2.5.1. ESPERANDO O FIM DE UM PROCESSO

Nesse exemplo, vamos utilizar o ciclo de um avião. O avião, enquanto está abastecendo, aguarda um determinado tempo para sair para voo novamente.

```
1 import simpy

3 class Aviao(object):
    #função que instancia as variaveis
5     def __init__(self, env):
        self.env = env
7         self.action = env.process(self.run())

9     def run(self):
        while True:
11           print('Estacionando avião e abastecendo as %d' % self.env.now)
             tempo_de_abastecimento = 5
13           #chama a função abastecer para aguardar o tempo de abastecimento
             yield self.env.process(self.abastecer(tempo_de_abastecimento))
15
             #finalizando o tempo de abastecimento, o aviao voa
16           print('Voando em %d' %self.env.now)
             tempo_voo = 2
18           yield self.env.timeout(tempo_voo)

20    def abastecer(self, tempo):
        yield self.env.timeout(tempo)
22
   env = simpy.Environment()
23 aviao = Aviao(env)
   env.run(until=15)
```

Quando o avião chega para abastecer, a função *abastecer* é chamada, repare que para chamar o processo de abastecer é utilizado o método

Environment.process() para solicitar uma instância do *Process*, fazendo com que o *run* aguarde o término do abastecimento para continuar sua execução.

As seguintes informações são retornadas no terminal:

Estacionando avião e abastecendo as 0

Voando em 5

Estacionando avião e abastecendo as 7

Voando em 12

Estacionando avião e abastecendo as 14

5.2.5.2. INTERROMPENDO PROCESSO

Diferente do exemplo anterior, que era preciso esperar o tempo de abastecimento do avião para ele voar, podemos interromper esse processo, saindo com uma quantidade X de combustível. No exemplo abaixo damos continuidade ao avião do exemplo anterior, agora interrompendo o processo de embarcar dele caso o horário de embarque seja ultrapassado.

```
1 import simpy

3 def stop(env, aviao):
      yield env.timeout(3)
5     aviao.action.interrupt()

7
  class Aviao(object):
9     def __init__(self,env):
          self.env = env
11        self.action = env.process(self.run())
      def run(self):
12        while True:
              print('Estacionando e esperando pessoas embarcarem em %d
                    %self.env.now)
14            tempo_embarque = 5

16            try:
                  yield self.env.process(self.embarque(tempo_embarque))
18            except simpy.Interrupt:
                  print('Horário limite, quem não embarcou perdeu o
voo')
20            print('Voando em %d' %self.env.now)
              tempo_voo = 2
22            yield self.env.timeout(tempo_voo)
```

```
24      def embarque(self,duration):
            yield self.env.timeout(duration)
25
    env = simpy.Environment()
26  aviao = Aviao(env)
    env.process(stop(env,aviao))
28  env.run(until=15)
```

Tendo como saída no console o seguinte:

Estacionando e esperando pessoas embarcarem em 0

Horário limite, quem não embarcou perdeu o voo

Voando em 3

Estacionando e esperando pessoas embarcarem em 5

Voando em 10

Estacionando e esperando pessoas embarcarem em 12

Basicamente, foi declarado na linha 30 o processo que vai fazer a interrupção. A partir disso, é dado *run* na linha 31 e o tempo vai contando, assim que chegar o tempo passado como timeout na linha 4, o processo que está ocorrendo será interrompido. Assim, como o tempo de embarque é 5, inicialmente, e o de interrupção é 3, há uma interrupção no embarque e o avião entra no status voando no tempo 3.

5.2.6. RECURSOS

Recursos são um tipo de contêiner, no qual os progressos podem inserir e retirar coisas. Porém, caso o recurso esteja cheio ou vazio, os processos devem fazer uma fila de esperar. Um recurso é constituído de eventos put() e get(). Os eventos put() são usados por processos que quererem adicionar algo aos recursos, enquanto os eventos get() são usados para tirar algo dos recursos.

Enquanto um processo está esperando por um evento put() ou get(), este processo pode ser interrompido por algum outro, caso isso ocorra ele pode continuar aguardando o evento put/get, chamando o evento novamente, ou pode apenas parar de esperar por estes eventos, chamando o método cancel() para cancelar. Os recursos podem ser usados por um número limitado de processos ao mesmo tempo,

e os processos os solicitam para usá-los ou possuí-los, sendo obrigados a liberá-los após fazer seu uso.

O Simpy tem três tipos de recursos, o padrão (Resource), o Recurso Prioritário (Priority Resource) e o Recurso Preventivo (Preemptive Resource).

I. RESOURCE

Tem como único parâmetro, além de uma referência ao Environment, sua capacidade. Ele armazena o evento de solicitação para cada usuário com um token de acesso, como no exemplo básico abaixo:

```
import simpy
def recurso(env, resource):
    solicitacao = resource.request() #gera um evento request
    yield solicitacao #Espera o acesso
    yield env.timeout(1) #Alguma outra função
    resource.release(solicitacao)

env = simpy.Environment()
res = simpy.Resource(env, capacity=1)
env.process(recurso(env,res))
env.run()
```

II. RECURSOS PRIORITÁRIOS

Permite que processos de solicitação forneçam uma prioridade para cada solicitação que fazem, demostrada por um número. Quanto menor o valor desse número, maior sua prioridade.

```
import simpy

def recursos(nome, env, resource,wait,prio):
    yield env.timeout(wait)
    with resource.request(priority=prio) as req:
        print('%s solicitando as %s com prioridade de=%s'  %(nome,env.now,prio))
        yield req
        print('%s pegou o recurso as %s' %(nome, env.now))
        yield env.timeout(3)

env = simpy.Environment()
res = simpy.PriorityResource(env, capacity=1)
env.process(recursos('a',env,res,wait=0,prio=0))
env.process(recursos('b',env,res,wait=1,prio=0))
env.process(recursos('c',env,res,wait=2,prio=1))
env.run()
```

Tendo como saída no terminal o seguinte:

a solicitando as 0 com prioridade de=0

a pegou o recurso as 0

b solicitando as 1 com prioridade de=0

c solicitando as 2 com prioridade de=1

b pegou o recurso as 3

c pegou o recurso as 6

III. RECURSOS PREVENTIVOS

Por diversas vezes é necessário mais que saltar na fila devido a importância de uma solicitação, fazendo com que usuários existentes sejam expulsos do recurso, é isso que recursos preventivos (PreemptiveResource) permitem fazer:

```
import simpy
def recursoPreventivo(nome,env,recurso,wait,prioridade):
    yield env.timeout(wait)
    with recurso.request(priority=prioridade) as req:
        print('%s solicitando as %s com prioridade de=%s' % (nome,env.now,prioridade))
        yield req
        print('%s pegou o recurso as %s' %(nome,env.now))
        try:
            yield env.timeout(3)
        except simpy.Interrupt as interrupt:
            causa = interrupt.cause.by
            usage = env.now - interrupt.cause.usage_since
            print('%s foi antecipado %s as %s depois de %s' % (nome, causa, env.now, usage))
env = simpy.Environment()
recurso = simpy.PreemptiveResource(env,capacity=1)
env.process(recursoPreventivo('(UserA)',env,recurso,wait=0,prioridade=0))
env.process(recursoPreventivo('(UserB)',env,recurso,wait=1,prioridade=0))
env.process(recursoPreventivo('(UserC)',env,recurso,wait=2,prioridade=-1))
env.run()
```

Podemos ver que sempre que solicitado algo com mais prioridade, o usuário que está acessando o recurso tem seu acesso antecipadamente interrompido para que a solicitação com mais prioridade seja atendida, gerando a seguinte saída no terminal:

(UserA) solicitando as 0 com prioridade de=0

(UserA) pegou o recurso as 0

(UserB) solicitando as 1 com prioridade de=0

(UserC) solicitando as 2 com prioridade de=-1

(UserA) foi antecipado <Process(recursoPreventivo) object at 0x1d965a24888> as 2 depois de 2

(UserC) pegou o recurso as 2

(UserB) pegou o recurso as 5

5.2.7. CONTAINERS

Os containers funcionam como recipientes, eles ajudam a modelar produção e consumo em determinada situação. Por exemplo, um caminhão de gasolina vai aumentar a quantidade de gasolina em uma bomba, enquanto os veículos que abastecem diminuirão essa quantia.

```
import simpy
class PostoGasolina:
    def __init__(self,env):
        self.bomba_gasolina = simpy.Resource(env,capacity=2)
        self.tanque_combustivel = simpy.Container(env, init=100, capacity=1000)
        self.monitor_processos = env.process(self.monitora_combustivel(env))
    def monitora_combustivel(self,env):
        while True:
            if self.tanque_combustivel.level < 100:
                print('Chamando tanque as %s' %env.now)
                env.process(caminhao_tanque(env,self))
            yield env.timeout(15)
def caminhao_tanque(env,PostoGasolina):
    yield env.timeout(10)
    print('Caminhão tanque chegando as %s' % env.now)
    quantidade = PostoGasolina.tanque_combustivel.capacity -
PostoGasolina.tanque_combustivel.level
    yield PostoGasolina.tanque_combustivel.put(quantidade)
def car(name,env,PostoGasolina):
    print("Carro %s chegando as %s" % (name,env.now))
    with PostoGasolina.bomba_gasolina.request() as req:
        yield req
        print('Carro %s abastecendo as %s'%(name,env.now))

        yield PostoGasolina.tanque_combustivel.get(40)
        yield env.timeout(5)
        print("Carro %s saindo as %s" %(name,env.now))

def gerador_carro(env,PostoGasolina):
    for i in range(4):
        env.process(car(i, env, PostoGasolina))
        yield env.timeout(5)


env = simpy.Environment()
Posto_de_gasolina = PostoGasolina(env)
env.process(gerador_carro(env,Posto_de_gasolina))
env.run()
```

O programa anterior permite uma filtragem do nível do container tanto quanto sua capacidade. Assim, quando o nível de gasolina da bomba de gasolina está abaixo de 100, que é o nível inicial, o caminhão tanque é chamado para reabastecer o tanque (linha 11). O caminhão abastece o container com a função put() (linha 17), enquanto os carros que abastecem no posto utilizam a função get() para diminuir o nível de gasolina no container (linha 24).

5.2.8. STORES

Uma store serviria como uma loja, que tem seus fornecedores e seus consumidores, trabalhando assim com objetos concretos, o que não ocorre nos containers. Uma única store pode armazenar vários tipos de objetos. Além de store, podemos usar funções diferentes para encontrar coisas especificas em uma store, como *FilterStore* e *PriorityStore.* A seguir, um exemplo básico do uso das stores:

```
1 import simpy

3 def fornecedor(env,store):
      for i in range(10):
5         yield env.timeout(2)
          yield store.put('produto %s' % i )
7         print('Forneceu produto as ', env.now)

9 def consumidor(nome,env,store):
      while True:
11        yield env.timeout(1)
          print(nome, 'solicitando produto as', env.now)
13        item = yield store.get()
          print(nome,'pegou', item, 'as',env.now)
15
  env = simpy.Environment()
17 store = simpy.Store(env, capacity=2)
  prod = env.process(fornecedor(env,store))
19 consumers = [env.process(consumidor(i,env,store)) for i in range(3)]
  env.run(until=5)
```

Pode-se notar que no exemplo anterior, existe um consumidor e um fornecedor. Enquanto o consumidor pega os itens (produtos) da store (linha 13), os fornecedores, por sua vez, adicionam itens à store (linha 6) juntamente ao seu nome.

O código acima tem como saída o seguinte:

6. DISCUSSÃO

Analisando as conclusões do resultado obtido, percebo que a parte de estudo do PM4PY foi satisfatória, foi possível ter uma boa compreensão acerca das funcionalidades da biblioteca de modo que as atividades propostas foram realizadas satisfatoriamente, e o aprendizado foi significante. Porém, se tratando da biblioteca de simulação o estuda foi mais superficial, a parte básica foi claramente entendida, deixando a possibilidade da construção de um projeto simples. Porém, acredito que os estudos da biblioteca não foram aprofundados o suficiente. A parte de simulação é a que mais da possibilidade do que fazer, são muitos exemplos que podem ser construídos para possibilitar o aprendizado, considero a parte mais fácil de compreender, mesmo sendo a que menos foi aprofundada, apenas foram exploradas as funções mais primitivas da biblioteca.

7. CONSIDERAÇÕES FINAIS

Considerando todas as atividades executadas ao longo do projeto, concluo que nem todos os objetivos foram alcançados. A parte de análise dos processos para encontrar erros por meio da mineração de processos foi bem executada, percorrendo todas as partes do modelo de processos, principalmente a parte de filtrar somente os cases não finalizados para identificar suas atividades e o tempo perdido em sua execução. Porém, o que não foi muito bem utilizado foi a parte de simulação, ainda mais quando se trata do Simpy. Basicamente, o que foi feito no Simpy foi um estudo sobre o funcionamento da biblioteca, sem focar em uma aplicação no problema, apenas exemplificando junto com as definições oferecidas pela biblioteca. Mesmo tendo sido um aprendizado muito bom analisar esta biblioteca, acabou que o foco desta parte do estudo foi diferente do inicialmente proposto.

8. OUTRAS ATIVIDADES REALIZADAS

Além das atividades com códigos recorrentes, foi feita muita leitura acerca da mineração de processos, tentando achar outros métodos para realizar uma predição dos tempos dos logs, pois a simulação feita na biblioteca de mineração de processos fornece uma simulação que não cobre todos os tempos contidos do log, repetindo só partes que realmente importam. Não obstante, a leitura dos artigos sobre mineração de processos também foi substancial pois ajudou muito nos conceitos e na compreensão da biblioteca.

REFERÊNCIAS

Documentation for Simpy. Disponível em: https://simpy-portuguese-br.readthedocs.io/pt_BR/latest/contents.html. Acesso em 02/06/2021.

Process Mining for Python. Disponível em <https://pm4py.fit.fraunhofer.de/>.

FERREIRA, Diogo. 2017. A Primer on Process Mining. 10.1007/978-3-319-56427-2.

LOPES, Nathália C. S. Modelo de Gestão por Processos Baseado em Mineração. Disponível em < https://www.cos.ufrj.br/uploadfile/publicacao/2769.pdf > Acesso em 28 de junho de 2021.

BARBON Jr, Sylvio. Inteligência de Negócios II – Mineração de Processos. Disponível em <http://www.barbon.com.br/wp-content/uploads/2013/11/ProcessMining.pdf>. Acesso em 29 de junho de 2021.

TEIXEIRA Jr, Gilmar. Modelagem de Sistemas de Informação para Mineração de Processos: Características e Propriedades das Linguagens. Disponível em < https://repositorio.bc.ufg.br/tede/bitstream/tede/7596/5/Disserta%C3%A7%C3%A3o%20-%20Gilmar%20Teixeira%20Junior%20-%202017.pdf>. Acesso em 30 de julho de 2021.

FRANCISCO, Rosemary; SANTOS, Eduardo A. P. Aplicação da Mineração de Processos como uma prática para a Gestão do Conhecimento. Disponível em < https://www.researchgate.net/profile/Rosemary-Francisco/publication/267830078_Aplicacao_da_Mineracao_de_Processos_como_uma_pratica_para_a_Gestao_do_Conhecimento/links/54da05f80cf25013d043b057/Aplicacao-da-Mineracao-de-Processos-como-uma-pratica-para-a-Gestao-do-Conhecimento.pdf>. Acesso em 1 de julho de 2021.

SZIMANSKI, Fernando. Melhoria de Modelos de Processos de Negócio com Mineração de Processos e Simulação Baseada em Agentes. Disponível em < https://core.ac.uk/download/pdf/33549539.pdf>. Acesso em 1 de julho de 2021.

ANEXO II (Exclusivo para PIBIC/PIBITI Jr.)

1. **Participação do bolsista nas atividades do PIBIC Jr**

Relacione a sua participação nas reuniões do grupo de pesquisa, no SEMIC e em outros eventos. Dê sua opinião a respeito.

2. **Dificuldades encontradas/críticas ou sugestões**

Descreva as dificuldades encontradas no desenvolvimento das atividades, incluindo críticas ou sugestões.

3. **Parecer do orientador acerca do desempenho do bolsista**

(Até 8 linhas)

4. **Parecer do professor supervisor acerca do desempenho do bolsista**

(Até 8 linhas)

5. **Relato da estudante sobre a participação no programa**

O projeto de iniciação cientifica me ofereceu meios para compreender outros âmbitos do meu ramo, que é a Ciência da Computação, estudando bibliotecas diferentes do Python, aprendendo a aplicar isto em uma situação real, o que contribui muito com o aprendizado que será necessário posteriormente no mercado de trabalho. Foi uma experiência boa para quem acabou de ingressar na computação e estava buscando alguma direção.