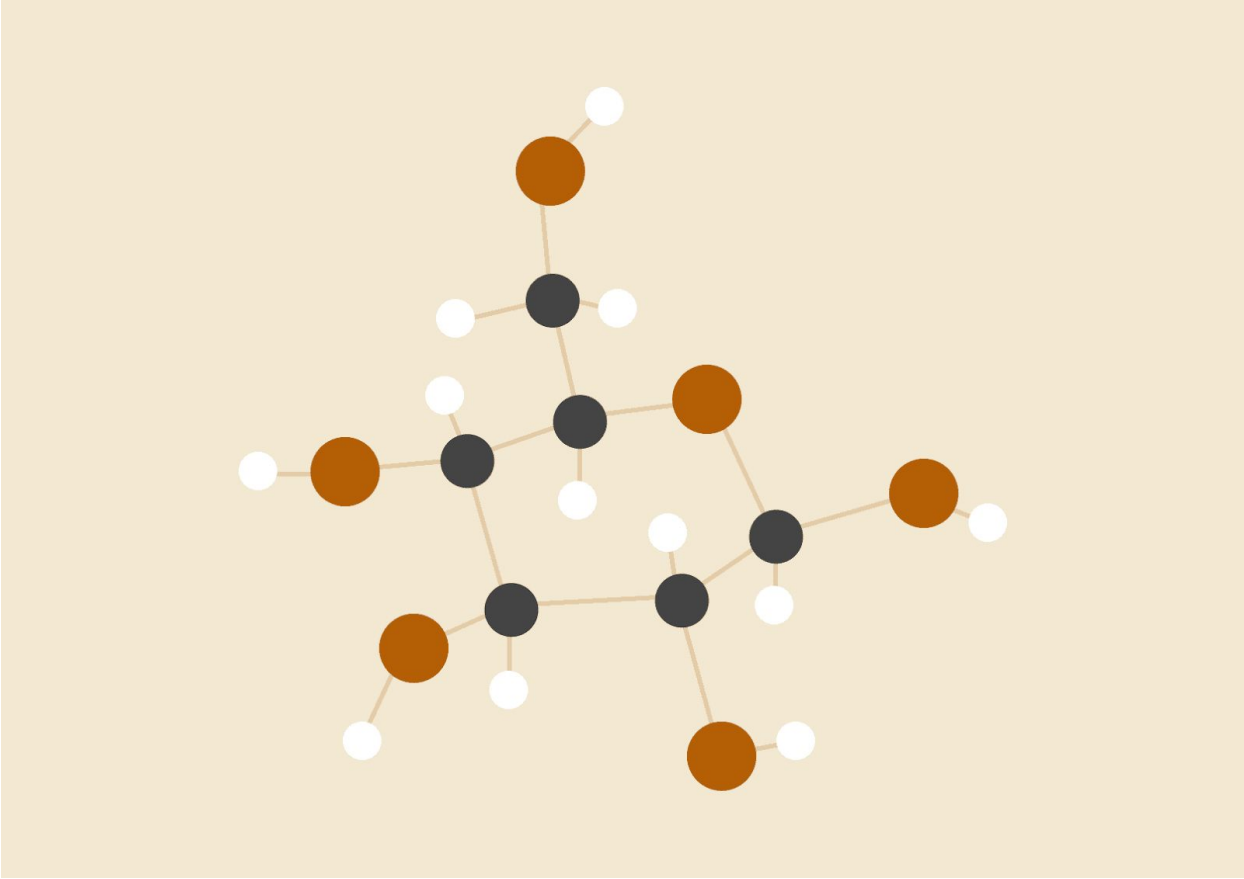
RELATÓRIO PROGRAMAÇÃO PARALELA E CONCORRENTE

Bar dos filósofos



Vinicius Amaro Sampaio

Matrícula: 1368157

2020

CIÊNCIAS DA COMPUTAÇÃO - UECE

INTRODUÇÃO

Neste projeto é feita uma implementação concorrente para o problema bar dos filósofos.

PROBLEMA BAR DOS FILÓSOFOS

Antes de descrever o problema do Bar dos Filósofos, vamos descrever o Problema do Jantar dos Filósofos.

Cinco filósofos se reúnem para jantar e conversar em uma mesa redonda com um garfo colocado sobre a mesa entre cada um deles. Como discutir problemas relacionados filosofia é um exaustivo, eles precisam comer de vez em quando. A fim de fazer isso, um filósofo requer o garfo da esquerda e o da direita, que são compartilhados com os colegas da esquerda e da direita, respectivamente. Depois de um filósofo comer, os dois garfos são colocado de volta na mesa, possibilitando que outro filósofo possa pegar os garfos e iniciar o jantar. É importante notar que um filósofo deve possuir ambos os garfos, caso contrário, ele não pode comer nada. Devido ao fato de que há apenas cinco garfos e cada filósofo precisa de dois deles para comer, é óbvio que, no máximo, dois filósofos pode comer simultaneamente. Uma solução para este problema tem de garantir que vários filósofos possam comer ao mesmo tempo. Além disso, deadlocks (impasses) e starvation (fome) devem ser evitados.

A descrição do Problema do Jantar dos Filósofos é bastante restrita. O problema se resume um grafo em ciclo, onde cada processo sempre precisa de recursos ao seu lado para executar sua seção crítica. Em 1984 , Chandy e Misra apresentaram uma generalização deste problema, o Bar dos Filósofos, mostrado a seguir.

A principal diferença nesse problema é o grafo de modelagem, que ao contrário do grafo circular do jantar dos filósofos, agora temos um grafo genérico. Nenhuma restrição é feita para a estrutura grafo, como distribuição ou grau de conectividade. Os filósofos (processos) são representados pelo nós (vértices) do grafo enquanto as garrafas (recursos

compartilhados) são representados pelas arestas do grafo. A generalização é focada em dois pontos: (1) a estrutura do grafo é arbitrária, e, (2) o número e conjunto de garrafas requeridas podem variar. Em cada rodada, cada processo escolhe um subconjunto de suas garrafas adjacentes para esta sessão de bebida. Este subconjunto escolhido não têm necessariamente de ser a mesma em cada sessão de bebida. Se o filósofo escolher todas as garrafas adjacentes em um grafo circular, podemos simular o problema do jantar dos filósofos. Os estados dos filósofos pode ser: tranquilo, com sede e bebendo, semelhante aos estados filosofando, com fome e comendo no problema do jantar dos filósofos. Cada filósofo pode requerer no mínimo uma e no máximo n garrafas, sendo n o número de arestas associadas a cada vértice.

IMPLEMENTAÇÃO

O projeto foi feito em Python com a seguinte classe:

Além das variáveis contidas na imagem, a classe filósofo estende a classe Thread da biblioteca threading e contém os seguintes métodos:

count_drink: atualiza a quantidade de rodadas que o filósofo participou.

free_drinks: retira as garrafas que o filósofo acabou de beber da variável compartilhada taken, liberando para que ela possa ser escolhida por outras instâncias do filósofo.

drink: dorme por um segundo e, em seguida, chama as funções *count_drink* e *free_drinks*.

number_drinks: sorteia um número entre 2 e *n* que será utilizado como a quantidade de garrafas a serem escolhidas pelo filósofo na rodada atual de bebidas.

rest_time: sorteia um número entre 0 e 2 com precisão de até 3 casas decimais que será usado para determinar a quantidade de segundos que o filósofo irá dormir na fase de descanso.

rest: dorme a quantidade x de segundos determinada pela função *rest_time*.

__init__: construtor da classe.

run: função chamada quando filósofo.start() é chamada. Os comandos dela serão

```
def run(self):
    while(self.drinks != self.k):
        t_calm = self.rest_time()
        self.rest(t_calm)
        t_start = datetime.now()
        n = self.number_drinks()
        d = 0
        flag = False
        while(d != n):

            dFlag = False
            while(not dFlag):
                vertex = choice(self.conec)
                if vertex not in self.chosen_drinks:
                    dFlag = True

            with data_lock:
                if (vertex not in taken) or ([vertex[1],vertex[0]] not in taken):
                    taken.append(vertex)
                    flag = True

            if flag:
                self.chosen_drinks.append(vertex)
                d = d + 1
        t_sede = datetime.now() - t_start
        self.drink()
        logging.info('Philosopher '+str(self.iden)+' rested for '+str(t_calm)+'s
        #logging.debug(','+str(self.iden)+','+str(t_calm)+','+str(t_sede)[6:]+','
        print('\033[94m'+'Philosopher ' +str(self.iden)+' shouts:'+'\033[0m'+'\n'+
        self.chosen_drinks = []
```

executados em paralelo (com exceção das regiões críticas) para todas as instâncias da classe filósofo. Nela temos um loop com de duração igual ao número de rodadas da execução atual.

Para cada rodada de drinks é sorteado um tempo de descanso e, após tal período, é sorteado a quantidade de garrafas que o filósofo deve beber nessa rodada, ele, então, entra em outro loop até que a quantidade de garrafas necessárias para a rodada seja atingida, nesse loop interno ele escolhe uma garrafa entre as disponíveis para ele caso ele já tenha escolhido essa garrafa ele escolhe outra e, caso outro filósofo já tenha escolhido a garrafa que ele quer, ele escolhe outra. Após conseguir todas as garrafas ele chama a função *drink* onde ele bebe e após o tempo de bebida proclama uma quote filosófica escolhida de forma aleatória do arquivo *quotes.txt* (no começo do programa a função *readfiles* é importada do arquivo *parsequotes.py* onde é feito o tratamento do arquivo txt e é gerado um vetor de strings)

A leitura do grafo é feita pela pela biblioteca Pandas na função *GetGraph* onde o caminho do arquivo é passado como argumento. Os arquivos estão no seguinte formato:

0,1,0,0,0,1
1,0,1,0,0,0
0,1,0,1,0,1
0,0,1,0,1,1
0,0,0,1,0,1
1,0,1,1,1,0

MECANISMO DE SINCRONIZAÇÃO

A sincronização é feita da seguinte forma: Uma variável compartilhada (*taken*) e uma variável de do tipo Lock (*data_lock*).

```
data_lock = Lock()
taken = []
```

A variável *data_lock* é utilizada para controlar o acesso a variável *taken* da seguinte maneira:

with data_lock:
#do something with taken

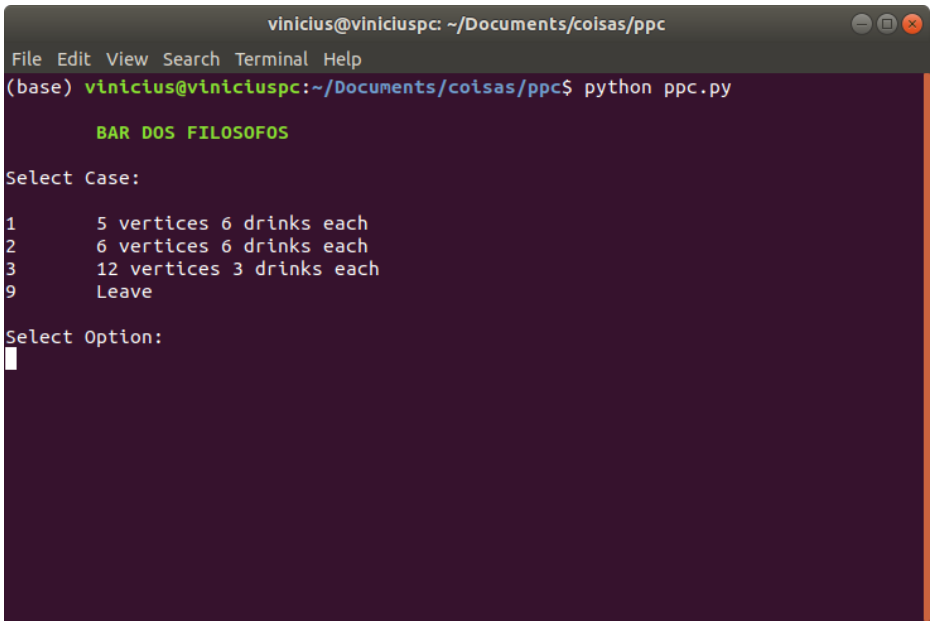
```
with data_lock:
    taken.remove(i)
```

Isso é equivalente a usar *data_lock.acquire()*, fazer as ações criticas e chamar *data_lock.release()*.

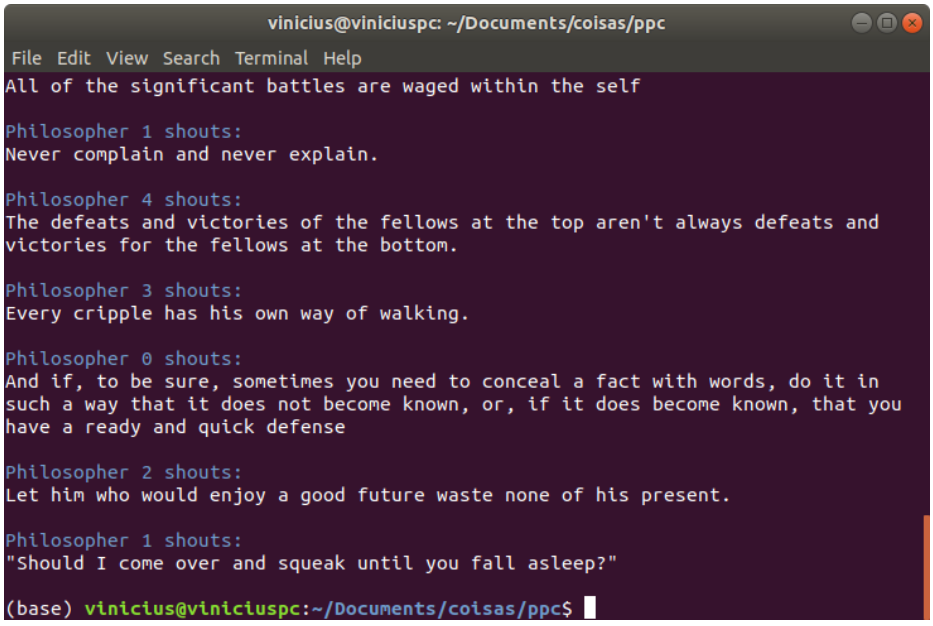
A função *Lock.acquire()* implementa espera bloqueada, ou seja depois que o primeiro thread adquirir a trava os outros threads serão bloqueados ao chegar na região crítica, quando o thread que possui a trava chamar a função *release()* apenas um dos threads bloqueados será acordado.

EXECUÇÃO

Foram feitas duas versões do programa: uma com um menu para escolher o grafo e um arquivo de log para salvar os tempos e a segunda versão sem menus ou prints onde os tempos são salvos em um csv e o caminho do grafo deve ser editado no código fonte (feita para várias execuções sequenciais) chamada versão teste.



Menu do programa

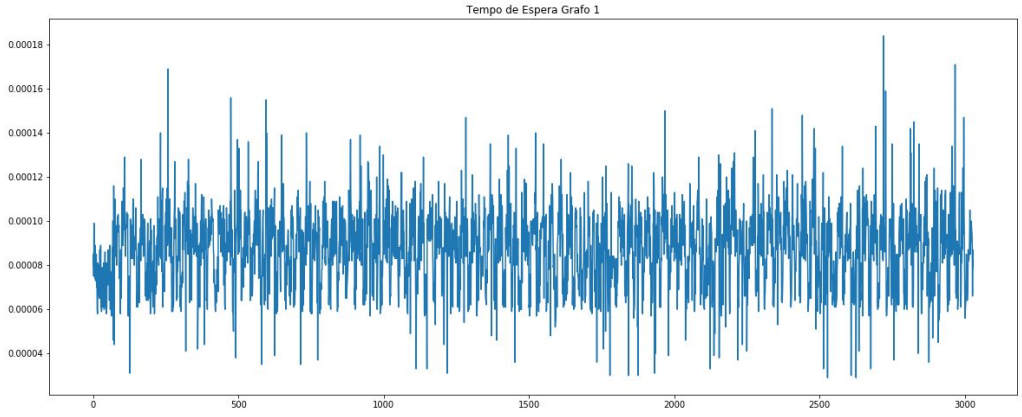


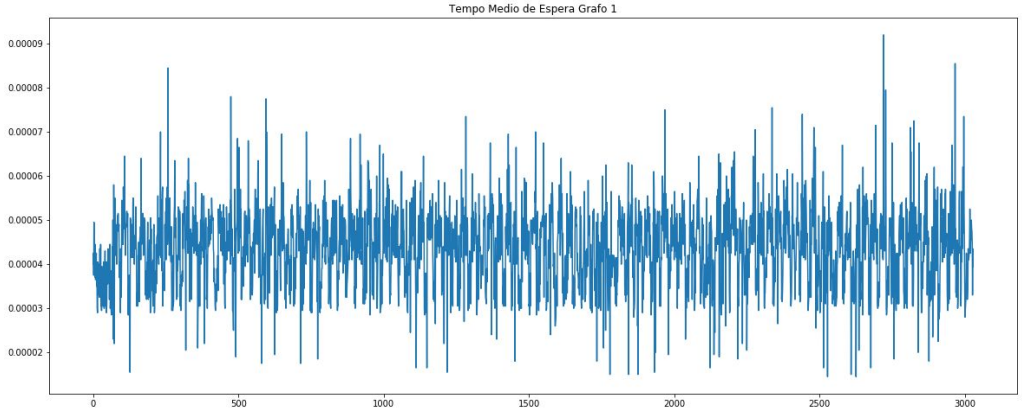
Visualização das quotes filosóficas

Para cada um dos três grafos foram feitas 100 execuções utilizando a versão de teste e foram gerados os seguintes resultados:

GRAFO 1

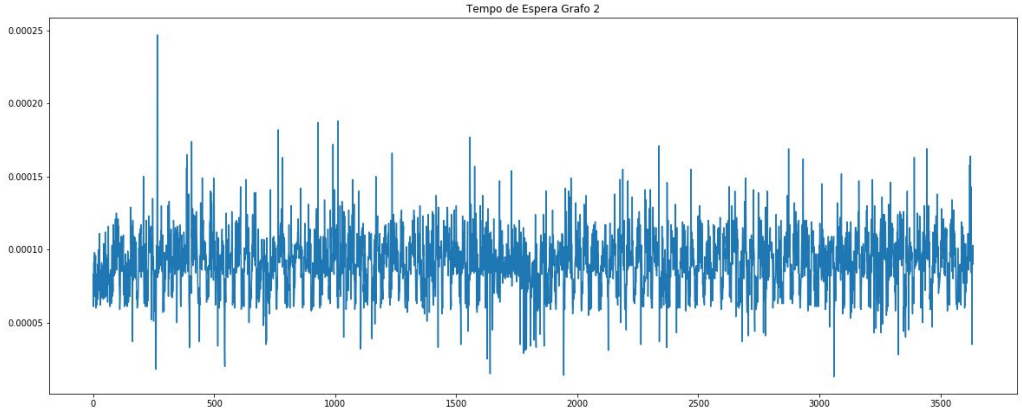
	ID	SLEEP	WAIT	DRINK	QTE
count	3030.000000	3030.000000	3030.000000	3030.000000	3030.0
mean	2.000000	1.004216	0.000087	3.500000	2.0
std	1.414447	0.575850	0.000018	1.708107	0.0
min	0.000000	0.000000	0.000029	1.000000	2.0
25%	1.000000	0.512250	0.000075	2.000000	2.0
50%	2.000000	1.010000	0.000088	3.500000	2.0
75%	3.000000	1.498000	0.000099	5.000000	2.0
max	4.000000	1.999000	0.000184	6.000000	2.0

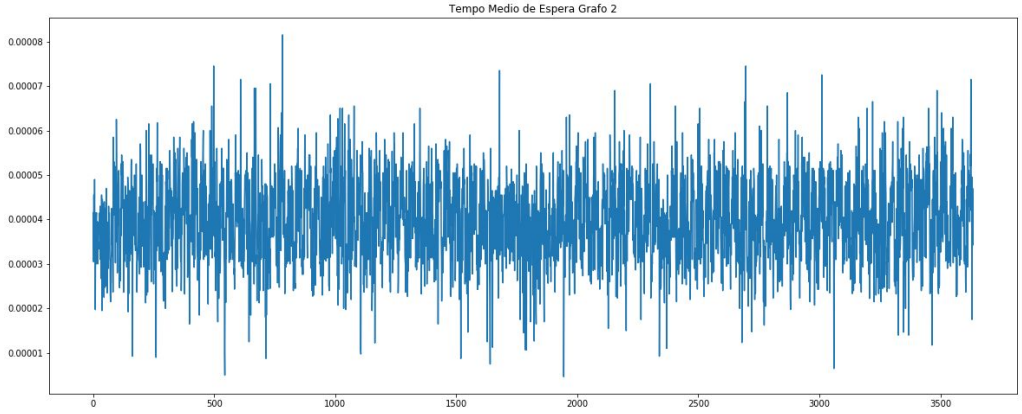




GRAFO 2

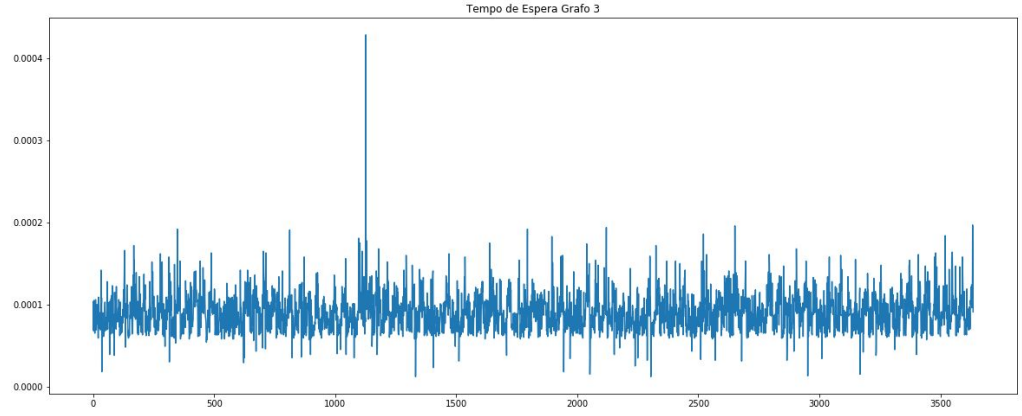
	ID	SLEEP	WAIT	DRINK	QTE
count	3636.00000	3636.000000	3636.000000	3636.00000	3636.000000
mean	2.50000	0.998265	0.000091	3.50000	2.344609
std	1.70806	0.583352	0.000020	1.70806	0.584345
min	0.00000	0.000000	0.000013	1.00000	2.000000
25%	1.00000	0.505000	0.000080	2.00000	2.000000
50%	2.50000	0.993000	0.000092	3.50000	2.000000
75%	4.00000	1.499250	0.000102	5.00000	3.000000
max	5.00000	1.999000	0.000247	6.00000	4.000000

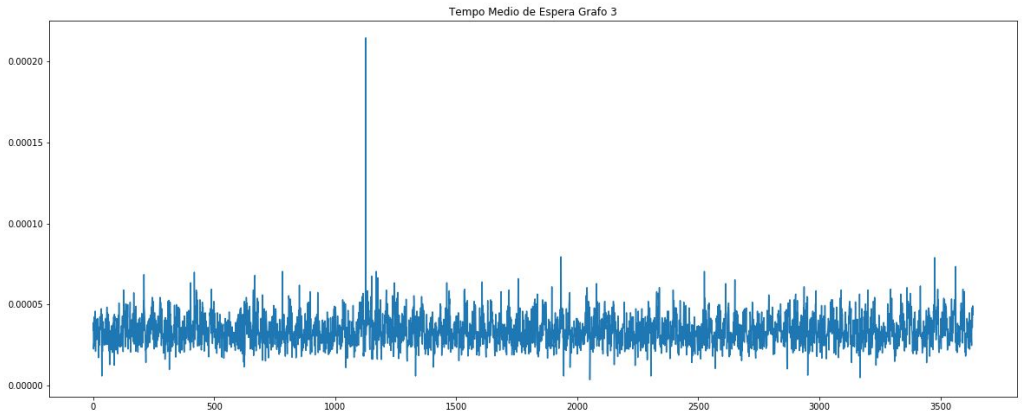




GRAFO 3

	ID	SLEEP	WAIT	DRINK	QTE
count	3636.000000	3636.000000	3636.000000	3636.000000	3636.000000
mean	5.500000	0.995383	0.000089	2.000000	2.768702
std	3.452527	0.566363	0.000023	0.816609	0.927418
min	0.000000	0.000000	0.000012	1.000000	2.000000
25%	2.750000	0.514000	0.000073	1.000000	2.000000
50%	5.500000	0.999000	0.000087	2.000000	3.000000
75%	8.250000	1.492000	0.000102	3.000000	3.000000
max	11.000000	1.998000	0.000429	3.000000	6.000000





CONCLUSÃO

Podemos ver que em sua maioria os tempos de espera não tem grande variância, com exceção de um caso no grafo 3 (provavelmente causado pelo escalonador do sistema, visto que foi o único caso das 100 execuções e considerando que o "spike" continua quando dividido pela quantidade de drinks.

Neste projeto tive a oportunidade de aprender e praticar técnicas de programação concorrente em python. Sem dúvidas, independente da nota, foi um dos projetos que agregou mais conhecimento no curso até agora.

REFERÊNCIAS

1. Problema do Bar dos Filósofos, http://marcial.larces.uece.br/cursos/programacao-concorrente-e-paralela-2019-2/problema-do-bar-dos-filosofos
2. threading — Higher-level threading interface, https://docs.python.org/2/library/threading.html#lock-objects
3. threading — Thread-based parallelism, https://docs.python.org/3/library/threading.html#with-locks