Mini-projeto de Simulação

Mestrado em Engenharia Informática

Universidade de Aveiro

DETI - Aveiro, Portugal

UC - Simulação e Otimização
2022/2023

Ricardo Rodriguez 98388

Mini-projeto de Simulação

May 13, 2023

Contents

1 Introdução

Este relatório visa explicar a metodologia usada para resolver os dois problemas de simulação propostos para o primeiro mini-projeto da unidade curricular de *Simulação e Otimização*, bem como responder a algumas questões levantadas no enunciado.

2 *Primeiro Problema*

2.1 Introdução

O primeiro problema consiste em simular uma instalação de serviços que contém dois servidores do tipo A e um servidor do tipo B, onde os clientes chegam com intervalos de tempo diferentes consoante sejam do tipo 1 ou do tipo 2, ambos os tipos com diferentes probabilidades: 0,8 e 0,2.

Os clientes do tipo 1 podem ser atendidos por qualquer servidor, mas escolhem um servidor do tipo A se houver um disponível. Se todos os servidores estiverem ocupados, os clientes do tipo 1 aguardam numa fila *first-in, first-out* para clientes do tipo 1.

Por outro lado , os clientes do tipo 2 exigem serviços simultaneamente de um servidor do tipo A e um servidor do tipo B. Se ambos os servidores do tipo A ou o servidor do tipo B estiverem ocupados, os clientes do tipo 2 aguardam numa fila *first-in, first-out* para clientes do tipo 2.

Quando um cliente é atendido, é dada preferência a um cliente do tipo 2, caso haja algum presente na fila respetiva, e se ambos os servidores do tipo A e o servidor do tipo B estiverem disponíveis. Caso contrário, a preferência é dada a um cliente do tipo 1.

O problema foi simulado para um tempo total de 1000 minutos, sendo que foi preciso calcular o tempo médio de espera na fila e o número médio de clientes na fila, num determinado tempo, para cada tipo de cliente. Além disso, foi também necessário estimar a proporção de tempo que cada servidor de cada tipo gasta com cada tipo de cliente.

Numa segunda estância, é necessário determinar se adicionar um servidor do tipo A ou um servidor do tipo B reduzirá o atraso máximo na fila para ambos os tipos de clientes, o que será respondido posteriormente.

2.2 Metodologia de implementação

Para resolver o problema proposto, foi seguido um *workflow* semelhante aos problemas apresentados nas aulás práticas da unidade curricular.

Inicialmente, foram definidas as constantes necessárias para o problema, nomeadamente: o número de servidores de cada tipo, a média de chegada de um cliente, as probabilidades de cada tipo de cliente, os tempos de serviço médio para cada tipo de cliente, o tempo máximo de simulação e, por último, o número de vezes por minuto em que serão registados o número de clientes em cada fila, de tipos diferentes.

De seguida, procede-se à inicialização de variáveis que serão usadas enquanto métricas da simulação que, no geral, servem para: calcular o tempo médio de espera na fila; calcular o número médio de clientes na fila, num determinado tempo, para cada tipo de cliente; e, por último, calcular a proporção de tempo que cada servidor de cada tipo gasta com cada tipo de cliente.

```python
# Simulation constants
NUM_TYPE_A_SERVERS = 2
NUM_TYPE_B_SERVERS = 1
INTERARRIVAL_MEAN = 1
TYPE_1_PROB = 0.8
TYPE_1_SERVICE_MEAN = 0.8
TYPE_2_SERVICE_MEAN = (0.5, 0.7)
MAX_SIMULATION_TIME = 1000
TIME_TICK_STEP = 10     # 10 ticks per second

# Performance tracking variables
type1_queue_delay = 0.0                 # Sum of the total delay of customers in type 1 queue
type1_queue_total_num_customers = 0     # Sum of the total number of customers that  passed in type 1 queue
type1_queue_avg_customers = 0           # Sum of the number of customers in type 1 queue throughout all ticks (10 times per unit)
type1_queue = []                        # Keep track of customers in queue and arrival time for type 1 queue
type2_queue_delay = 0.0                 # Sum of the total delay of customers in type 2 queue
type2_queue_total_num_customers = 0     # Sum of the total number of customers that  passed in type 2 queue
type2_queue_avg_customers = 0           # Sum of the number of customers in type 2 queue throughout all ticks (10 times per unit)
type2_queue = []                        # Keep track of customers in queue and arrival time for type 2 queue
server_utilization = {
                        'A' : { index : 0.0 for index in range(1, NUM_TYPE_A_SERVERS+1) },
                        'B' : { index : 0.0 for index in range(1, NUM_TYPE_B_SERVERS+1) }
                    }
```

Figure 1: Variáveis necessárias ao problema

De seguida, após inicializar todos os servidores com os valores *False*, que significa que não estão ocupados, e adicionar o primeiro evento de chegada de um cliente no início da simulação, a simulação é iniciada.

O *loop* da simulação corre enquanto o tempo da simulação não chegou ao limite previamente definido. Nesta fase, calcula-se o próximo evento, com base no tempo mais pequeno, e verifica-se se é necessário calcular o número de clientes em cada tipo de fila, para no final calcular a média do número de clientes em cada tipo de fila.

Após isto, o programa pode processar um evento de chegada ou de partida. No evento de chegada, é calculado o próximo evento de chegada e calcula-se o tipo de cliente do evento de chegada atual, com base nas probabilidades definidas. Caso o cliente seja do tipo 1, verifica-se se existe algum servidor do tipo A disponível, uma vez que é o servidor de preferência do mesmo. Caso haja, o servidor escolhido estará ocupado para servir o tal cliente. Caso não exista um servidor do tipo A disponível, procede-se à procura de um servidor do tipo B nos mesmos moldes. No caso de nenhum servidor estar disponível, o cliente atual é adicionado à fila de espera dos clientes do tipo 1.

Caso o cliente seja do tipo 2, os moldes são os mesmos, diferindo apenas no facto de que tem de haver um servidor do tipo A e um servidor do tipo B disponível para o cliente ser servido. Caso isto não ocorra, o cliente será adicionado à fila de espera dos clientes do tipo 2.

Se o evento atual for uma partida, tornar-se-ão disponíveis os servidores usados pelo cliente, consoante o seu tipo. De seguida, verifica-se se existe alguém na fila dos clientes do tipo 2, que tomam prioridade para serem servidos, e se existem dois servidores, um do tipo A e outro do tipo B, para o cliente do tipo 2 ser servido. Caso isto se verifique, será criado um evento de partida para o cliente que esteve à espera na fila e está a ser servido atualmente.

Por outro lado, se as condições para servir um cliente do tipo 2 na fila de espera não se verificarem, avalia-se se existe algum cliente do tipo 1 que possa ser servido ou por um servidor do tipo A ou por um servidor do tipo B.

No final, procede-se ao cálculo das estimativas supramencionadas para efeitos de análise da simulação.

2.3 Instruções para executar a simulação

Para correr a simulação, basta executar o programa com Python: *python3 ex1.py*

2.4 Resultados

Os resultados obtidos da simulação estão presentes abaixo, sendo que: a primeira figura refere-se à simulação com dois servidores do tipo A e um servidor do tipo B (simulação padrão); a segunda figura à simulação com três servidores do tipo A e um servidor do tipo B; e, por último, a terceira figura se refere à simulação com dois servidores do tipo A e dois servidores do tipo B.

```
======= expected average delay in queue =======
[TYPE 1] 0.41141819946269426 seconds
[TYPE 2] 0.3816592936248045 seconds
======= expected time average number in queue for each type of customer =======
[TYPE 1] 0.0178
[TYPE 2] 0.0166
======= expected proportion of time that each server spends on each type of customer =======
[SERVER A1 & TYPE 1] 0.46660143502496737
[SERVER A1 & TYPE 2] 0.5333985649750326
[SERVER A2 & TYPE 1] 0.22955250894598603
[SERVER A2 & TYPE 2] 0.770447491054014
[SERVER B1 & TYPE 1] 0.1797828775344307
[SERVER B1 & TYPE 2] 0.8202171224655693
```

Figure 2: Resultados obtidos para dois servidores do tipo A e um servidor do tipo B

```
======= expected average delay in queue =======
[TYPE 1] 0.25960616833228817 seconds
[TYPE 2] 0.37211084080620604 seconds
======= expected time average number in queue for each type of customer =======
[TYPE 1] 0.0028
[TYPE 2] 0.0075
======= expected proportion of time that each server spends on each type of customer =======
[SERVER A1 & TYPE 1] 0.43975148169925
[SERVER A1 & TYPE 2] 0.56024851830075
[SERVER A2 & TYPE 1] 0.2059612760760558
[SERVER A2 & TYPE 2] 0.7940387239239441
[SERVER A3 & TYPE 1] 0.06424570262413591
[SERVER A3 & TYPE 2] 0.9357542973758641
[SERVER B1 & TYPE 1] 0.12469319506633013
[SERVER B1 & TYPE 2] 0.8753068049336699
```

Figure 3: Resultados obtidos para três servidores do tipo A e um servidor do tipo B

```
======= expected average delay in queue =======
[TYPE 1] 0.2520375119865399 seconds
[TYPE 2] 0.3274477067095082 seconds
======= expected time average number in queue for each type of customer =======
[TYPE 1] 0.0025
[TYPE 2] 0.0112
======= expected proportion of time that each server spends on each type of customer =======
[SERVER A1 & TYPE 1] 0.44609955886663016
[SERVER A1 & TYPE 2] 0.5539004411333699
[SERVER A2 & TYPE 1] 0.24183286932358153
[SERVER A2 & TYPE 2] 0.7581671306764185
[SERVER B1 & TYPE 1] 0.16326181095136463
[SERVER B1 & TYPE 2] 0.8367381890486354
[SERVER B2 & TYPE 1] 0.05117124994463708
[SERVER B2 & TYPE 2] 0.9488287500553629
```

Figure 4: Resultados obtidos para dois servidores do tipo A e dois servidores do tipo B

Como se pode verificar através da visualização dos resultados obtidos, o aumento do número de servidores, indepententemente do tipo, melhora o tempo médio de espera de cada tipo de cliente.

Contudo, o aumento do número de servidores do tipo B por uma unidade mostra-se ser mais benéfico, tendo em conta que reduz o tempo médio de espera dos clientes do tipo 2 em 0.05 minutos, com uma redução muito pouco significativa do tempo de espera para clientes do tipo 1. A redução de tempo para os clientes do tipo 2 é notória, uma vez que estes precisam de dois servidores em simultâneo para poderem ser servidos.

3 Segundo Problema

3.1 Introdução

O modelo de Lotka-Volterra descreve a evolução de populações de presas e predadores num determinado ambiente. Este baseia-se em duas equações diferenciais, que dependem de alguns parâmetros previamente definidos.

O segundo problema visa traçar a evolução de populações de presas e predadores, de acordo com duas variantes do modelo de Lotka-Volterra: o método de Euler e o método de Runge Kutta.

3.1.1 Metodologia de Implementação

O programa de simulação usa um processador de linha de comandos para obter os parâmetros necessários, nomeadamente variáveis que serão usadas nas fórmulas matemáticas do modelo de Lotka-Volterra e para executar a simulação.

- x0 - Número inicial de presas
- y0 - Número inicial de predadores
- Alpha - Taxa máxima de crescimento per capita da presa
- Beta - Efeito da presença de predadores na taxa de crescimento da presa
- Delta - Efeito da presença de presas na taxa de crescimento do predador
- Gamma - Taxa de mortalidade per capita do predador
- time_step - Intervalo de tempo entre os pontos de amostragem
- max_time - Tempo máximo da simulação

Além destes parâmetros, o utilizador precisa, também, de especificar a variante que quer usar para a simulação.

Relativamente à metodologia das variantes do modelo supramencionado, ambas criam uma lista de valores temporais, de zero até max_time, com intervalos de acordo com a variável $time_step$.

De seguida, ambas as variantes executam o programa até ao tempo final da simulação, com os devidos intervalos, em que se estima o número de presas e o número de predadores em cada intervalo de tempo da simulação.

```python
# Exercise 2.1 -  trace the evolution of x(t), and y(t), using the Forward Euler method given the method input arguments
def lotka_volterra_forward_euler(x0, y0, alpha, beta, delta, gamma, time_step, max_time):
    times = np.arange(0, max_time + time_step, time_step)   # Fills up a zero-valued array from [0, max_time[ in *time_step* (delta_t) steps
    n = len(times)
    x = np.zeros(n)     # Number of preys
    y = np.zeros(n)     # Number of predators
    x[0] = float(x0)           # Initial number of preys
    y[0] = float(y0)           # Initial number of predators
    # Compute the predator-prey population evolution over n iterations
    for i in range(1, n):
        x[i] = x[i-1] + (alpha * x[i-1] - beta * x[i-1] * y[i-1]) * time_step
        y[i] = y[i-1] + (- gamma * y[i-1] + delta * x[i-1] * y[i-1]) * time_step
    return times, x, y
```

Figure 5: Implementação do método Lotka-Volterra - Método de Euler

```python
def lotka_volterra_runge_kutta(x0, y0, alpha, beta, delta, gamma, time_step, max_time):
    times = np.arange(0, max_time+time_step, time_step)
    n = len(times)
    x = np.zeros(n)
    y = np.zeros(n)
    x[0] = x0
    y[0] = y0
    # Compute the predator-prey population evolution over n iterations
    for i in range(1, n):
        k1x = alpha * x[i-1] - beta * x[i-1] * y[i-1]
        k1y = delta * x[i-1] * y[i-1] - gamma * y[i-1]
        k2x = alpha * (x[i-1] + k1x*time_step/2) - beta * (x[i-1] + k1x*time_step/2) * (y[i-1] + k1y*time_step/2)
        k2y = delta * (x[i-1] + k1x*time_step/2) * (y[i-1] + k1y*time_step/2) - gamma * (y[i-1] + k1y*time_step/2)
        k3x = alpha * (x[i-1] + k2x*time_step/2) - beta * (x[i-1] + k2x*time_step/2) * (y[i-1] + k2y*time_step/2)
        k3y = delta * (x[i-1] + k2x*time_step/2) * (y[i-1] + k2y*time_step/2) - gamma * (y[i-1] + k2y*time_step/2)
        k4x = alpha * (x[i-1] + k3x*time_step) - beta * (x[i-1] + k3x*time_step) * (y[i-1] + k3y*time_step)
        k4y = delta * (x[i-1] + k3x*time_step) * (y[i-1] + k3y*time_step) - gamma * (y[i-1] + k3y*time_step)
        x[i] = x[i-1] + time_step/6 * (k1x + 2*k2x + 2*k3x + k4x)
        y[i] = y[i-1] + time_step/6 * (k1y + 2*k2y + 2*k3y + k4y)
    return times, x, y
```

Figure 6: Implementação do método Lotka-Volterra - Método de Runge Kutta

No final, é demonstrado um gráfico que apresenta a evolução populacional de presas e de predadores, consoante os dados introduzidos na linha de comandos.

3.2 Instruções para executar a simulação

Para correr a simulação, é necessário introduzir os argumentos necessários ao programa de simulação consoante a figura apresentada abaixo, de forma ordenada.

```python
parser = argparse.ArgumentParser(description='CLI Interface to the predator-prey populations simulation program using the Lotka-Volterra model')
parser.add_argument('x0', type=float, help='Initial number of preys (population density)')
parser.add_argument('y0', type=float, help='Initial number of predators (population density)')
parser.add_argument('alpha', type=float, help='Maximum prey per capita growth rate')
parser.add_argument('beta', type=float, help='Effect of the presence of predators on the prey growth rate')
parser.add_argument('delta', type=float, help="Effect of the presence of prey on the predator's growth rate")
parser.add_argument('gamma', type=float, help="Predator's per capita death rate")
parser.add_argument('time_step', type=float, help='Time step interval')
parser.add_argument('max_time', type=float, help='Time of the simulaiton')
parser.add_argument('--method', type=str, default='euler', choices=['euler', 'runge_kutta'], help='Lotka Volterra variation method')
args = parser.parse_args()
```

Figure 7: Processamento de argumentos da linha de comando para o segundo problema

Exemplo para correr a simulação com o método de Euler:
python3 ex2.py 10 10 0.1 0.02 0.02 0.4 0.1 1000 –method euler
Exemplo para correr a simulação com o método de Runge Kutta:
python3 ex2.py 10 10 0.1 0.02 0.02 0.4 0.1 1000 –method runge_kutta

3.3 Resultados

Os resultados da simulação obtidos para ambas o método de Euler e o método de Runge Kutta, com os mesmos argumentos que os exemplos do capítulo anterior, estão presentes nas figuras abaixo, respetivamente.

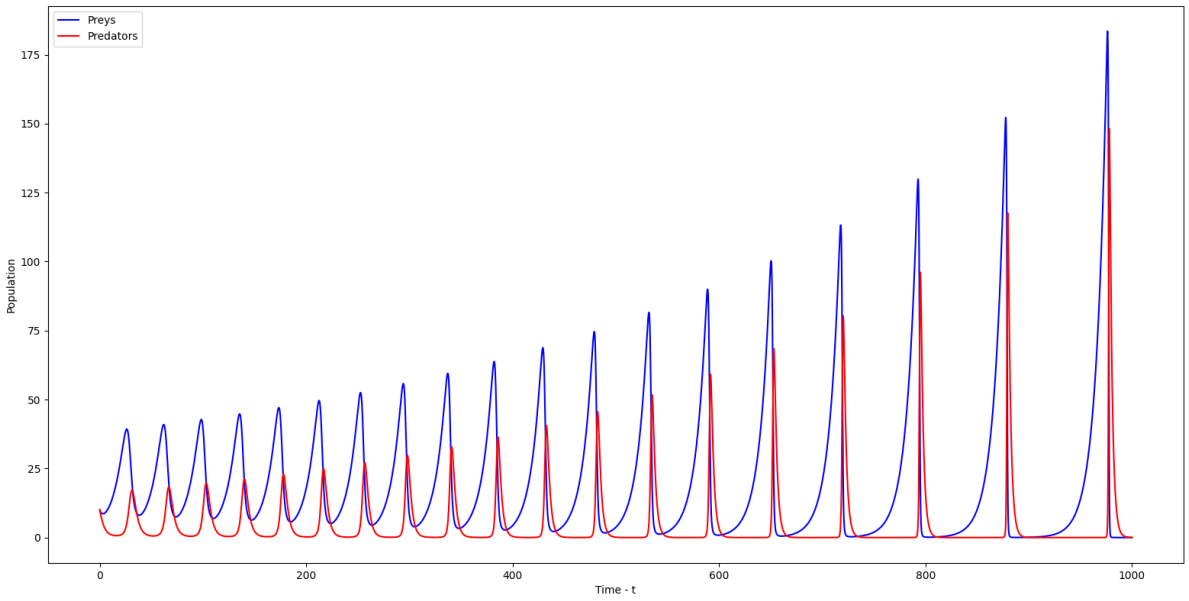

Figure 8: Resultados obtidos para o método de Euler

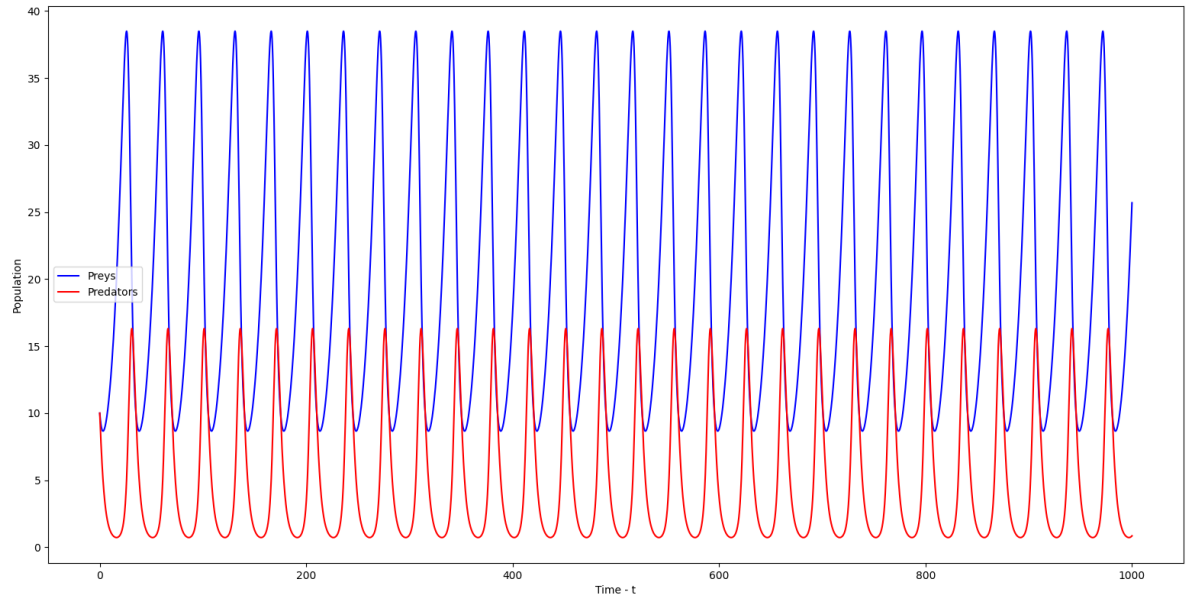

Figure 9: Resultados obtidos para o método de Runge Kutta

4 Conclusão

A análise de cenários reais, como é o caso dos problemas apresentados neste relatório, através da implementação de programas de simulação, com base nos conceitos teórico-práticos aprendidos na vertente de simulação da unidade curricular de *Simulação e Otimização*, revelam-se muito benéficos para a compreensão do mundo e para a tomada de decisões através da análise de dados simulados.