Universidad Autónoma de Coahuila

PROYECTO FINAL: CALIDAD Y PRUEBASDE SOFTWARE/SIMULACION

Redes de Jackson

Equipo:
Ernesto Carrera de la Peña
Jesus Eduardo Garcia Morquecho
Juan Jaime Reyes Herrera
Oscar Uriel Rojas Badillo

Docente:
Carlos Trejo Nasiff

Fecha:
22/11/2022

Introducción

Las líneas de espera (o colas) forman parte de nuestra vida cotidiana. Tanto nuestra vida personal como laboral nos obliga a formar parte de estos sistemas. Pero el tiempo de espera que cada persona tiene que esperar para poder obtener un beneficio o pedir un servicio puede no suponer solamente una molestia sino también un coste monetario, más aún cuando las líneas de espera son analizadas desde el punto de vista empresarial. Esto es lógico puesto que muchas veces la espera puede limitar el tiempo productivo de un agente productivo. Es decir que el problema principal es no poder realizar las tareas programadas mientras se esté esperando, tal vez debido a la falta de recursos necesarios, de servidores capacitados, o simplemente que mantenerse presente en una línea de espera imposibilita estar presente en su área de trabajo. La teoría de colas es el estudio de la espera en las distintas modalidades. Utiliza los modelos de colas para representar los tipos de sistemas de líneas de espera (sistemas que involucran colas de algún tipo) que surgen en la práctica. Las fórmulas de cada modelo indican cuál debe ser el desempeño del sistema correspondiente y señalan la cantidad promedio de espera que ocurrirá en diversas circunstancias (Hillier & Lieberman, 2010) La utilidad de estos modelos radica en que permite analizar el rendimiento de una línea de espera para así lograr encontrar un balance adecuado entre el costo de servicio y el coste que supone esperar por el servicio, sea este un coste real o un coste por oportunidad despreciada. El presente escrito documenta la simulación de estos sistemas, concretamente un modelo de red de líneas de espera abierta (también conocido como red de Jackson), a través del lenguaje de programación Python y las bibliotecas facilitadoras de interfaces graficas en HTML conocida como Tkinter. La propuesta aborda la elaboración de una red de Jackson para 3 nodos o líneas de espera que representan tres servicios proporcionados por un hotel: estacionamiento, recepción y elevador. Se extrajeron los métodos y funciones principales del sistema para llevar a cabo las pruebas unitarias, de integración, de cobertura, etc.

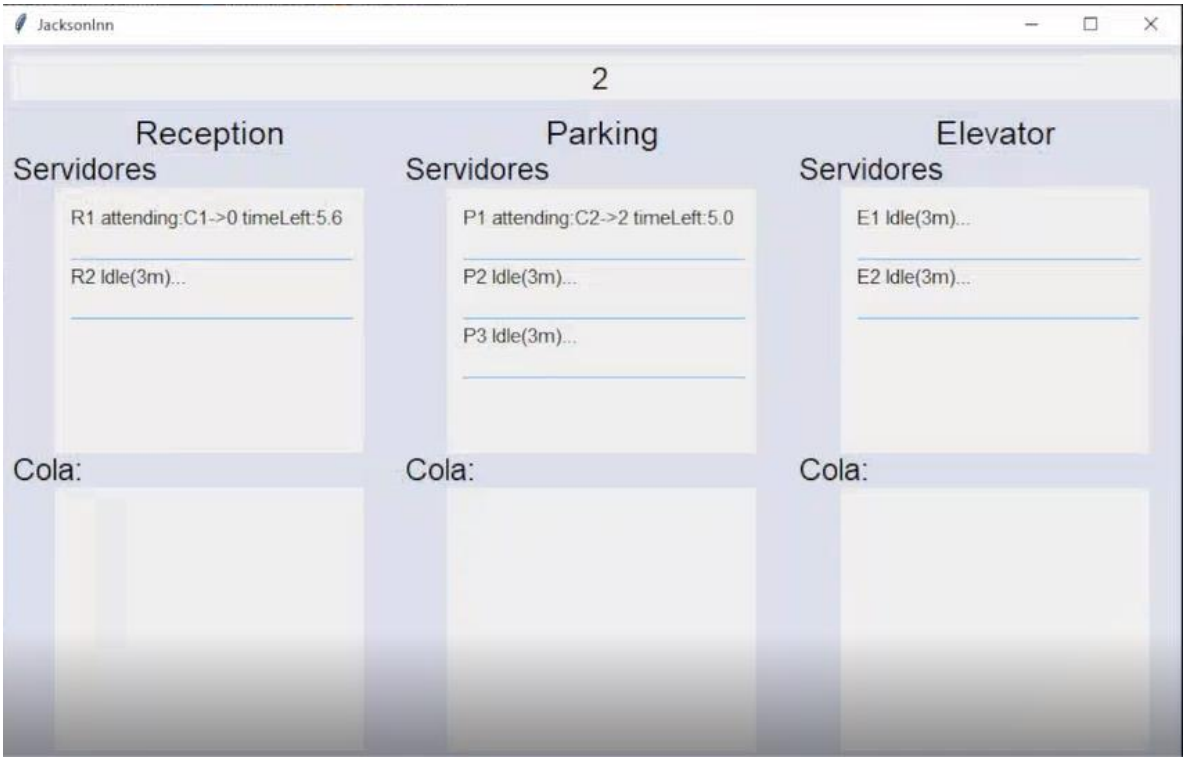
Marco teórico

Líneas de espera

Los actores principales en una situación de colas son el cliente y el servidor. Los clientes llegan a una instalación (servicio) desde de una fuente. Al llegar, un cliente puede ser atendido de inmediato o esperar en una cola si la instalación está ocupada. Cuando una instalación completa un servicio, “jala” de forma automática a un cliente que está esperando en la cola, si lo hay. Si la cola está vacía, la instalación se vuelve ociosa hasta que llega un nuevo cliente. (TAHA, 2012) Medidas de rendimiento Las medidas de rendimiento indican la eficacia en la que una red opera según las propias características del modelo usado, dependiendo siempre de las tasas de llegada, servicio y el número de servidores del sistema. La siguiente tabla muestra los indicadores usados por las redes de colas usados por Krieger. (Gómez, Silva, López, Ramírez, & Ibarra, 2018)

demostración

Se hizo una implementación de los modelos MMC para hacer mediciones y poder observar si se ejecuta con normalidad como lo muestra en la imagen.



Para hacer las pruebas unitarias, integración y mutables, se tomo las funciones del modelo de los servicios del hotel en tiempo real.

λ	Tasa media de llegadas.
$1/\lambda$	Tiempo medio entre llegadas
μ	Tasa media de servicio.
$1/\mu$	Tiempo medio de servicio.
L	Número esperado de clientes en el sistema.
L_q	Número esperado de clientes en la cola *q-esima*.
W	Tiempo total de espera en el sistema
W_q	Tiempo promedio de espera en la cola *q-esima*.
π_o	Probabilidad de que el sistema esté ocupado (en estado estacionario)
π_0	Probabilidad de que el sistema esté vacío (en estado estacionario)
N_i	Número de servidores en la estación *i,i=1,2,...,s*

Modelo MMC

Este modelo presupone una cola infinita de espera y múltiples servidores en la cola. Los clientes

llegan a la cola a una tasa de parámetro λ cada unidad de tiempo que sigue una distribución

exponencial, son atendidos por s servidores a una tasa de parámetro μ operaciones cada unidad

de tiempo que sigue también una distribución exponencial. Según estos parámetros, las medidas

de rendimiento están dadas por las siguientes ecuaciones:

Factor de utilización

$$\rho = \frac{\lambda}{k\mu}$$

Probabilidad de que no haya unidades en el sistema

$$P_0 = \frac{1}{\sum_{n=0}^{k-1} \frac{(\lambda/\mu)^n}{n!} + \frac{(\lambda/\mu)^k}{k!}\left(\frac{k\mu}{(k\mu-\lambda)}\right)}$$

Probabilidad de que haya n unidades en cola

Para $n \leq k$

$$P_n = \frac{(\lambda/\mu)^n}{n!} P_0$$

Para $n > k$

$$P_n = \frac{(\lambda/\mu)^n}{k!\,(k^{n-k})} P_0$$

Número promedio de unidades en cola

$$L_q = \frac{(\lambda/\mu)^k \mu\lambda}{(k-1)!\,(k\mu-\lambda)^2} P_0$$

Número promedio de unidades en el sistema

$$L_s = L_q + \frac{\lambda}{\mu}$$

Tiempo promedio que una unidad pasa en una cola

$$W_q = \frac{L_q}{\lambda}$$

Se dice que una cola MMC cumple la condición de no saturación cuando su factor de utilización es menor a 1.

$$\rho = \frac{\lambda}{s\mu} < 1$$

Redes de cola

Las redes de colas son múltiples líneas de espera interconectadas, en redes los clientes solicitan el servicio de algunas o todas las colas del sistema. Es de destacable importancia el descubrimiento de la propiedad de equivalencia: Propiedad de equivalencia: suponga que una instalación de servicio tiene s servidores, un proceso de entradas Poisson con parámetro y la misma distribución de los tiempos de servicio de cada servidor con parámetro (el modelo M/M/s), en donde s. Entonces, la salida en estado estable de esta instalación de servicio también es un proceso de Poisson con parámetro. (Hillier & Lieberman, 2010) Así, una red de colas es un conjunto de nodos interconectados, cada uno de los cuales está formado por un sistema de colas con uno o más servidores que operan de forma asíncrona y concurrente. Se distingue según sean sistemas abiertos o cerrados. Cuando se trata de una red cerrada, no se permite la entrada de nuevos clientes ni la salida de los clientes existentes, haciendo que el número de clientes sea constante en el tiempo. Por otra parte, cuando se trata de una red abierta los clientes pueden entrar y salir del sistema libremente, el flujo de entrada en el sistema puede darse por cualquier nodo al igual que el flujo de salida del sistema. Estos sistemas de redes pueden ser tanto cíclicos como acíclicos. Cuando un sistema de redes es cíclico se permite pasar en múltiples ocasiones a sistemas de colas ya utilizados, mientras que esto es imposible en sistemas acíclicos. Red en tándem En este sistema de colas el cliente debe visitar diversos servidores antes de completar el servicio requerido. El cliente va pasando por distintos subsistemas en serie donde cada subsistema tiene su propio tipo de cola, pero donde la tasa de entrada debe igualar a la tasa de salida. Además, una red en tándem tiene las siguientes características:

• El número de clientes de cada uno de los servidores es independiente del otro.

• Los tiempos de espera de un cliente en cada cola no son independientes.

• Los tiempos totales de espera (cola + servicio) son independientes.

Red de Jackson Por otra parte una red de Jackson es un sistema de m nodos (sistema de colas de espera) de servicio en donde cada nodo i (i =1,2, . . ., m) recibe clientes que visitaron o visitaran las demás instalaciones en orden independiente (y que además pueden no llegar a todas ellas). Es decir que se trata de un sistema de redes abierto, pues los clientes pueden provenir tanto del exterior del sistema como de otras instalaciones en su interior. Debe de contar con:

• Una cola infinita

• Clientes que llegan de afuera del sistema según un proceso de entrada Poisson con parámetro ai • si servidores con distribución exponencial de tiempos de servicio con parámetro μi .

Sus parámetros de rendimiento están dados por el total de la suma de los parámetros individuales de cada nodo

$$L = \sum_{i=1}^{m} L_i$$

$$Lq = \sum_{i=1}^{m} Lq_i$$

$$W = \sum_{i=1}^{m} W_i$$

$$Wq = \sum_{i=1}^{m} Wq_i$$

La probabilidad de tener xi clientes simultáneamente en cada nodo i esta dado simplemente por la probabilidad para xi de manera conjunta para cada nodo i. (Hillier & Lieberman, 2010) Además de que la condición de no saturación del sistema establece que todo nodo i cumpla con la condición de no saturación de manera independiente.

Plan de acción

Se desarrollarán las ecuaciones para solucionar un sistema simple de redes de Jackson. Se identifican cuales son las variables involucradas en el proceso y el tipo de salida obtenida por la operación. Se simulará el la llegada de clientes a una red de colas independientes no limitadas a su utilización secuencial

A falta de librerías se creo una clase que simulara ser la api con todas las características y funcionalidades necesarias para el correcto funcionamiento de la aplicación elaborada en Python.

Desarrollo

El proyecto tomado como base fue un simulador de un sistema de líneas de espera de

El

TABLA 17.4 Datos del ejemplo de una red de Jackson

Instalación j	s_j	μ_j	a_j	P_{ij}		
				$i=1$	$i=2$	$i=3$
$j=1$	1	10	1	0	0.1	0.4
$j=2$	2	10	4	0.6	0	0.4
$j=3$	1	10	3	0.3	0.3	0

EN base a los analisis descriptivos se simulo mediante una red de Jackson las medidas de MMC, se diseño una clase que simula ser una api, su tarea principal es simular los calculos de tiempos de llegada, tiempos de espera y tamaño de los nodos.

Se hacen dos pruebas con los datos proporcionados (a, miu y servers) de las 3 colas de Jackson, para esto si utilize una computacion externa con una pagina, en el cual se creo el modelo MMC

```
#Prueba 1
a=30 #a
miu=40 #m
servers=3 #c

#Prueba 2
a2=15 #a
miu2=20 #m
servers2=5 #c



class test_excec(unittest.TestCase):
    #Suma numeros de posiciones pasadas
    def test_AcumulativeArray(self):
        cases = [
            ([4, 16,5], [4,20,25]),
            ([1, 2,3], [1,3,6]),
        ]
```

La clase test_excec, es manejada a base de objetos, para esto se creo un reloj, con la finalidad que se fueran generando objetos y destruyendo conforme sea necesario.

Para una prueba unitaria se utilizo acumulativeArray para generar entradas y salidas, se sumaron los digitos mediante entraban a la cola, lo que el acumulartiveArray hace es una matriz de probabilidad acumulativa para poder tomar deciciones en base a eso.

```python
class test_excec(unittest.TestCase):
    #Suma numeros de posiciones pasadas
    def test_AcumulativeArray(self):
        cases = [
            ([4, 16,5], [4,20,25]),
            ([1, 2,3], [1,3,6]),
        ]

        for inp, expected in cases:

            with self.subTest(inp=inp, expected=expected):
                obtained = Exec.getAcumulativeArray(inp)
                self.assertEqual(obtained, expected, "Array(%s) should be %s" % (inp, expected))
```

Para las pruebas unitarias se uso una calculadora en linea que hizo calculos aritmeticos con entradas y salidas esperadas segun lo obtenido de los metodos

```python
class test_modelMMC_adap(unittest.TestCase):
    def test_p(self):
        #Se encarga de sacar la p con la operacion a/(c*m)
        cases = [
            (a,miu,se (variable) servers2: int
            (a2,miu2,servers2,.15)

        ]

        for inp1, inp2,inp3, expected in cases:

            with self.subTest(inp1=inp1, inp2=inp2,inp3=inp3, expected=expected):
                obtained = modelMMCadap.p(inp1,inp2,inp3)
                self.assertEqual(obtained, expected, "p(%s) should be %s" % (inp1,expected))
```

En base a esto se utilize para obtener pruebas

Prubea de Po

```python
    def test_Po(self):
        #Se encarga de sacar Po con la operacion ...
        cases = [
            (a,miu,servers,-6.174119866483886e-34),
            (a2,miu2,servers2,-6.438358212912472e-20)
        ]

        for inp1, inp2,inp3, expected in cases:

            with self.subTest(inp1=inp1,inp2=inp2,inp3=inp3, expected=expected):
                obtained = modelMMCadap.Po(inp1, inp2,inp3)
                self.assertEqual(obtained, expected, "Po(%s) should be %s" % (inp1,expected))
```

Prueba de Lq

```
    def test_Lq(self):
        #Se encarga de sacar Lq con la operacion (Po*((a/m)**c)*p)/(factorial(c)*(1-p)**2
        cases = [
            (a,miu,servers,.25,-6.174119866483886e-34,.014705882), #0.014705882
            (a2,miu2,servers2,.15,-6.438358212912472e-20,0.000193929), #.000193929

        ]

        for inp1, inp2,inp3,inp4,inp5, expected in cases:

            with self.subTest(inp1=inp1,inp2=inp2,inp3=inp3,inp4=inp4,inp5=inp5, expected=expected):
                obtained = modelMMCadap.Lq(inp1, inp2,inp3,inp4,inp5)
                self.assertEqual(obtained, expected, "Lq(%s) should be %s" % (inp1,expected))
```

Prueba de Wq

```
    def test_Wq(self):
        #Se encarga de sacar Wq con la operacion lq/a
        cases = [
            (a,0.014705882,0.0004901960666666667), #0.000490196
            (a2,.000193929,1.29286e-05), #0.000012929

        ]

        for inp1, inp2, expected in cases:

            with self.subTest(inp1=inp1,inp2=inp2, expected=expected):
                obtained = modelMMCadap.Wq(inp1, inp2)
                self.assertEqual(obtained, expected, "Wq(%s) should be %s" % (inp1,expected))
```

Identidad

```
    def test_idle(self):
        #Se encarga de sacar idle con la operacion 1-p
        cases = [
            (.25,.75),
            (.15,.85),

        ]

        for inp1, expected in cases:
```

Para el mutmut, se hizo manual, invirtiendo simbolos, cambiando parametros e invertir operaciones

```python
def pmut(a,m,c)-> float:
        return (c*m)/a #MODIFICACION SE INVIERTE LA OPERACION

def Lqmut(a,m,c,p,Po) -> float:
        return (Po*((a/m)**c)*p)/(factorial(c)*(1+p)**2)  #MODIFICACION 1+P

def Pomut(m,c,p) -> float:
        sum = 0
        for m in range(0,p):           #SE CAMBIA EL RANGO DEL FOR
            sum+=((c*p)**m)/factorial(m)
        sum+=((c*p)**c)/(factorial(c)*(1-p))
        return 1/(sum)

def Wqmut(a,lq) -> float:
        return lq-a #SE CAMBIA LA OPERACION A UNA DIVISION
```

El objetivo de los metodos es localizar los asesinables, aceptables y durante la ejecucion se encontraron nuevos casos de prueba con estas mutaciones

```python
from math import factorial
import numpy as np

def p(a,m,c)-> float:
        return a/(c*m)

def Lq(a,m,c,p,Po) -> float:
        return (Po*((a/m)**c)*p)/(factorial(c)*(1-p)**2)

def Po(m,c,p) -> float:
        sum = 0
        for m in range(0,c):
            sum+=((c*p)**m)/factorial(m)
        sum+=((c*p)**c)/(factorial(c)*(1-p))
        return 1/(sum)

def Wq(a,lq) -> float:
        return lq/a
```

Casos de prueba

```python
def test_Wqmut(self):
    #Se encarga de sacar Wq con la operacion lq/a
    cases = [
        (a,0.014705882,0.0004901960666666667), #0.000490196# error por mutacion
        (a2,.000193929,1.29286e-05), #0.000012929# error por mutacion

    ]

    for inp1, inp2, expected in cases:

        with self.subTest(inp1=inp1,inp2=inp2, expected=expected):
            obtained = modelMMCadap.Wqmut(inp1, inp2)
            self.assertEqual(obtained, expected, "Wq(%s) should be %s" % (inp1,expected))
```

Se hizo un coverage para verificar la covertura del codigo

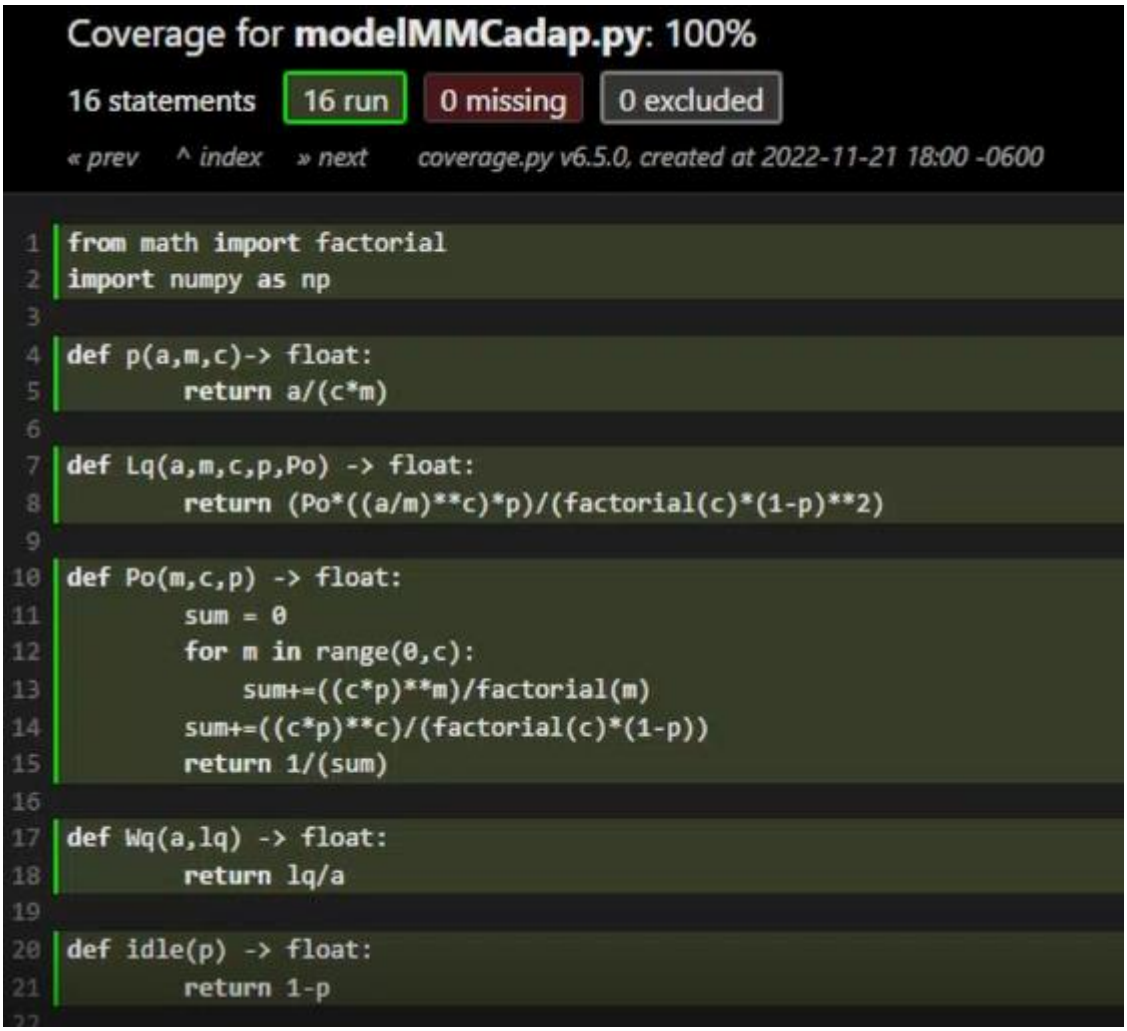
Coverage for **test_proyect.py**: 98%

49 statements | 48 run | 1 missing | 0 excluded

« prev ^ index » next coverage.py v6.5.0, created at 2022-11-21 18:00 -0600

```python
import unittest
import Exec
import modelMMCadap

#Prueba 1
a=30 #a
miu=40 #m
servers=3 #c

#Prueba 2

a2=15 #a
miu2=20 #m
servers2=5 #c



class test_excec(unittest.TestCase):
    #Suma numeros de posiciones pasadas
    def test_AcumulativeArray(self):
        cases = [
            ([4, 16,5], [4,20,25]),
            ([1, 2,3], [1,3,6]),
```

Se obtuvo el 100% en el Test de covertura



Coverage for **modelMMCadap.py**: 100%

16 statements | 16 run | 0 missing | 0 excluded

« prev ^ index » next coverage.py v6.5.0, created at 2022-11-21 18:00 -0600

```
1  from math import factorial
2  import numpy as np
3
4  def p(a,m,c)-> float:
5          return a/(c*m)
6
7  def Lq(a,m,c,p,Po) -> float:
8          return (Po*((a/m)**c)*p)/(factorial(c)*(1-p)**2)
9
10 def Po(m,c,p) -> float:
11         sum = 0
12         for m in range(0,c):
13             sum+=((c*p)**m)/factorial(m)
14         sum+=((c*p)**c)/(factorial(c)*(1-p))
15         return 1/(sum)
16
17 def Wq(a,lq) -> float:
18         return lq/a
19
20 def idle(p) -> float:
21         return 1-p
```

Se utilizaron las pruebas unitarias, de mutación, Set Up y TearDown y se hizo una prueba de cobertura con la librería de coverage.

Se realizaron pruebas unitarias donde se aislaron los métodos utilizados en la clase MMC (después de ser adaptados para un correcto funcionamiento), a partir de esto se realizaron pruebas de mutación manuales cambiando signos las operaciones aritméticas.

Además de realizar las pruebas se verifico una correcta implementación de la clase de testeo con un coverage, así corroborando que el flujo de datos abarcaba todos y cada uno de las partes del método y prueba.

Con un Set Up y TearDown se verifico que el archivo creado con el nombre de resultados y con todos los datos dentro se haya creado correctamente y que todos los datos estén intactos.

Bibliografía

Hillier, F. S., & Lieberman, G. J. (2010). INTRODUCCIÓN A LA INVESTIGACIÓN DE OPERACIONES .

MÉXICO DF: MC GRAW HILL/INTERAMERICANA EDITORES, S.A. DE C.V.

TAHA, H. A. (2012). INVESTIGACIÓN DE OPERACIONES. MÉXICO DF: PEARSON EDUCACIÓN.