Reporte 3:
Teoría de Colas

Jorge Torres

26 de febrero de 2022

1. Objetivo

El objetivo de la práctica consiste en analizar los tiempos de ejecución de tres algoritmos diferentes que encuentran si un número es primo. Se varían tanto el algoritmo utilizado como el orden inicial de la lista de números de forma independiente y se realiza un análisis estadístico relevante para decidir si la diferencia en tiempos de ejecución es significativa o no. Por último, se muestran resultados de la evaluación en gráficas tipo violín para una mejor interpretación.

2. Desarrollo

Basando el desarrollo en el código implementado por E. Schaeffer [1], primero se definen los parámetros de ejecución de los algoritmos, como se observa en el código 1,

Código 1: Parámetros

```
d = 1000
h = 5000
replicas = 30
original = [x for x in range(d, h + 1)]
invertido = original[::-1]
aleatorio = original.copy()
shuffle(aleatorio)
```

donde se crea una lista de números desde 1000 hasta 5000, y se establece que los experimentos tendrán 30 réplicas. Los parámetros `invertido` y `aleatorio` simplemente reordenan la lista de números de forma invertida y aleatoria, respectivamente.

Lo siguiente es definir las tres formas en que se decide si los números son primos o no. Para esto se toman los algoritmos implementados por E. Schaeffer, y se pueden ver en el código 2.

Código 2: Algoritmos para encontrar números primos

```
def primo_1(n):
    if n < 3:
        return True
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

def primo_2(n):
    if n < 4:
        return True
    if n % 2 == 0:
       return False
    for i in range(3, n - 1, 2):
        if n % i == 0:
            return False
    return True

def primo_3(n):
    if n < 4:
```

```
        return True
    if n % 2 == 0:
       return False
    for i in range(3, int(ceil(sqrt(n))), 2):
        if n % i == 0:
            return False
    return True
```

La función `primo 1` simplemente decide que un número es primo si es menor a 3 o si el residuo de la división entre sus predecesores es diferente de cero. La función `primo 2` únicamente revisa entre los número impares de la lista. La tercera función, `primo 3`, utiliza la lógica matemática en la que, para el par de factores p y q de n, el menor de ellos no puede ser mayor a $\sqrt{n}$, mejorando así el algoritmo.

El siguiente paso es definir las listas donde se guardan los tiempos de ejecución y comenzar a iterar entre las diversas formas de encontrar los primos y de ordenar la lista de números, como se observa en el código 3.

Código 3: Ejecución de códigos

```
if __name__ == "__main__":
    resultados_1 = {'Algoritmo 1': [], 'Algoritmo 2': [], 'Algoritmo 3': []}
    resultados_2 = {'Original': [], 'Invertido': [], 'Aleatorio': []}
    with multiprocessing.Pool(processes = cores-1) as pool:
        pool.map(primo_1, original)
        for r in range(replicas):
            t = (time()*1000)
            pool.map(primo_1, original)
            resultados_1['Algoritmo 1'].append((time()*1000)-t)
            t = (time()*1000)
            pool.map(primo_2, original)
            resultados_1['Algoritmo 2'].append((time()*1000)-t)
            t = (time()*1000)
            pool.map(primo_3, original)
            resultados_1['Algoritmo 3'].append((time()*1000)-t)
            t = (time()*1000)
            pool.map(primo_3, original)
            resultados_2['Original'].append((time()*1000) - t)
            t = (time()*1000)
            pool.map(primo_3, invertido)
            resultados_2['Invertido'].append((time()*1000) - t)
            t = (time()*1000)
            pool.map(primo_3, aleatorio)
            resultados_2['Aleatorio'].append((time()*1000) - t)
    df1 = pd.DataFrame(data = resultados_1)
    df2 = pd.DataFrame(data = resultados_2)
```

Por último, se realiza un análisis de varianza utilizando una instrucción de la librería estadística de `scipy` (ver código 4), para determinar si la media en cada tipo de variación (algoritmo y orden), representa una diferencia significativa que permita decir que el tiempo de ejecución depende del algoritmo utilizado o del orden inicial de los números.

Código 4: Test de análisis de varianza

```
stat1, p1 = f_oneway(resultados_1['Algoritmo 1'],
                     resultados_1['Algoritmo 2'],
                     resultados_1['Algoritmo 3'])
print('Variando algoritmo\n', 'stat=%.3f, p=%.3f' % (stat1, p1))
if p1 > 0.05:
    print('Estadisticamente no significativa\n')
else:
    print('Estadisticamente significativa\n')
stat2, p2 = f_oneway(resultados_2['Original'],
                     resultados_2['Invertido'],
                     resultados_2['Aleatorio'])
print('Variando orden de numeros\n', 'stat=%.3f, p=%.3f' % (stat2, p2))
if p2 > 0.05:
    print('Estadisticamente no significativa\n')
else:
    print('Estadisticamente significativa\n')
```

El código en su totalidad se puede obtener de mi repositorio en GitHub.

3. Resultados

Los resultados consisten en un test análisis de varianza para determinar si la diferencia entre tiempos de ejecución es estadísticamente significativa, además de un par de gráficas tipo violín para confirmar visualmente los resultados del análisis.

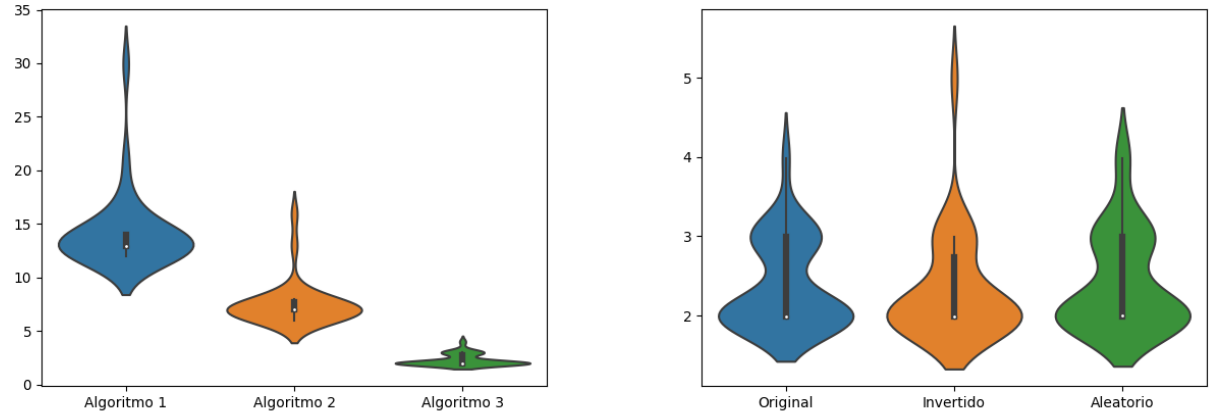
3.1. Análisis de Varianza

Este modelo estadístico permite comprobar la veracidad de la hipótesis nula, H_0, al determinar si las medias de dos o más muestras pertenecen a la misma densidad de población [2]. En este caso, la hipótesis nula es que las medias de los tiempos pertenecen a la misma densidad de población, y por lo tanto no dependen del algoritmo u orden utilizado al no haber una diferencia significativa. Para concluir si la hipótesis se comprueba o no, se hace referencia al valor p calculado por el análisis implementado en el código. Específicamente, si $p > 0{,}05$, la diferencia no es significativa, mientras que si $p < 0{,}05$, la diferencia es estadísticamente significativa.

Al variar el algoritmo utilizado, el resultado del análisis indica un valor de p mucho menor a 0,05. Por otro lado, al variar el orden inicial de los números el resultado indica una p mayor a 0,05. Esto se puede visualizar mejor al hacer la comparación gráfica con los diagramas tipo violín. La interpretación de estos resultados se discute en la sección 4.

3.2. Diagramas Tipo Violín

Un diagrama tipo violín es similar a uno de caja-bigote en el sentido que muestra el rango intercuartil y la media de una muestra o grupo de muestras. La diferencia radica en que el diagrama tipo violín contiene un diagrama de densidad en sus costados para mostrar la forma de densidad de la muestra de datos [3]. En la figura 1 se pueden observar los diagramas tipo violín de los tiempos de ejecución tanto al variar el algoritmo utilizado (figura 1a) como al variar el orden inicial de los números (figura 1b).



(a) Variación del algoritmo

(b) Variación del orden

Figura 1: Diagramas violín

4. Conclusiones

Tomando en cuenta un valor de $p < 0{,}05$ como prueba de que la diferencia en tiempos de ejecución es significativa, entonces se tiene que: 1) Hay una clara dependencia entre el algoritmo utilizado y el tiempo que toma encontrar los número primos, y 2) la media de los tiempos de ejecución al variar el orden inicial no parece presentar una diferencia significativa.

Esto puede ser más evidente al observar los diagramas presentados. En la figura 1a se observa claramente la disminución de las medias (representadas por el punto blanco), al utilizar un algoritmo más eficiente. Por otro lado, en la figura 1b las medias se encuentran en posiciones muy similares, además de que las distribuciones para cada orden no son tan distintas.

Referencias

[1] E. Schaeffer. Queuingtheory. *Repositorio, GitHub*, 2019. URL https://github.com/satuelisa/Simulation/tree/master/QueuingTheory.

[2] Wikipedia. Analysis of variance, 2022. URL https://en.wikipedia.org/wiki/Analysis_of_variance#The_F-test.

[3] Wikipedia. Violin plot, 2022. URL https://en.wikipedia.org/wiki/Violin_plot.