Optimización de flujo en redes

175
Tarea #2

19 de marzo de 2019

Se utilizan algoritmos para grafos de NetworkX, implementando un codigo Python para ejecutar los siguientes algoritmos:

- `all_shortest_paths`
- `betweenness_centrality`
- `dfs_tree`
- `greedy_color`
- `max_weight_matching`

luego se aplica una cantidad moderada de iteraciones para crear tiempos de ejecucion considerables para despues hacer un analisis estadistico.

1. Caminos cortos

Se utiliza el primer grafo de la tarea 2 con el algoritmo `all_shortest_paths` de NetworkX, el cual encuentra todos los caminos cortos de un nodo inicial a un nodos destino, se aplica un total de iteraciones de 8 millones para alcanzar un tiempo de ejecucion mayor a 5 segundos como se muestra a continuacion:

```
tiempoAngel1 = []
for i in range(30):
    start = tm.time()
    for x in range(8000000):
        nx.all_shortest_paths(G, source='C', target='B')
    end = tm.time()
    tiempoAngel1.append(end - start)
```

La figura 1 muestra los resultados obtenidos de la implementacion, se muestra la prueba estadistica de *shapiro wilk* para probar si la distribucion de los datos se comportan normales, dado que el p valor es mayor a 0.05 se puede concluir que los datos se comportan normales, ademas se muestra la media y desvicion estandar.

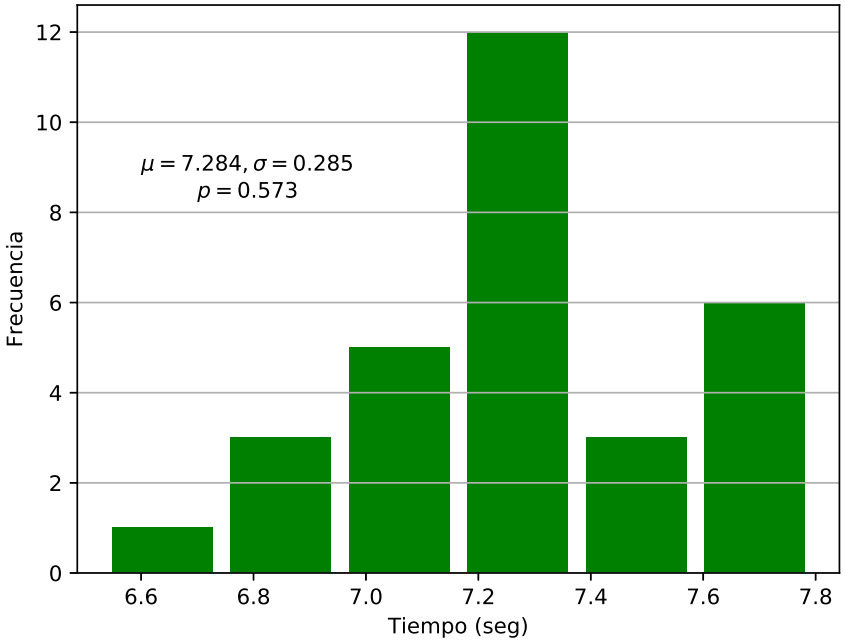

Figura 1: Histograma del tiempo de ejecucion del algoritmo.

2. Nodos centrales

Se implementa subrutina similar a la seccion pasada con la diferencia de utilizar el algoritmo `betweenness_centrality`, el cual es perfecto para encontrar caminos cortos desde un nodo inicial central hasta uno final. Se utiliza el segundo grafo de la tarea 2.

La figura 2 muestra el histograma del tiempo de ejecion, se observa que los datos no se comportan normales debido al p valor muy pequeño de la prueba estadistica.

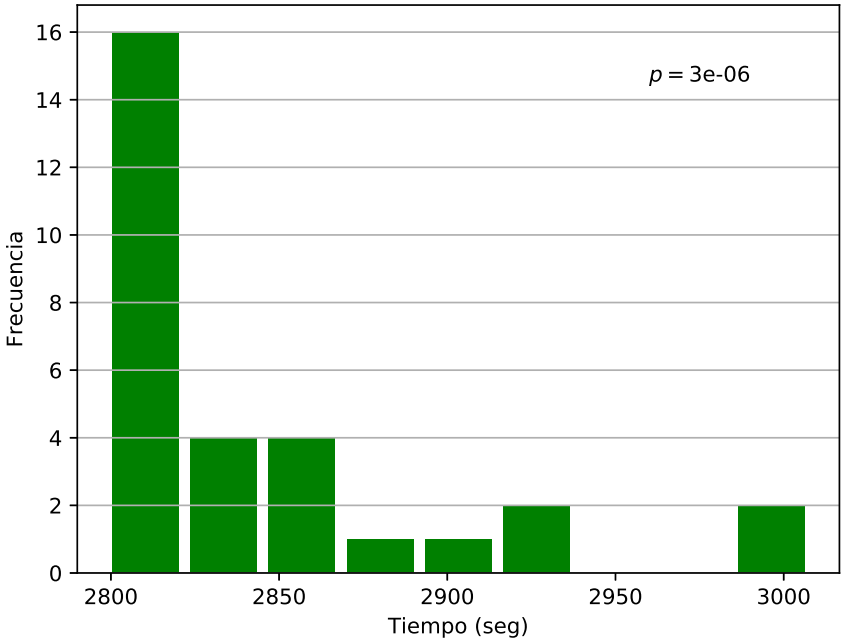

Figura 2: Histograma del tiempo de ejecucion del algoritmo `betweenness_centrality`.

3. Busqueda en profundidad

El algoritmo `dfs_tree` construye un arbol orientado partiendo desde un nodo inicial utlizando la busqueda a profundidad. La figura 3 muestra el histograma, se vuelve a rechazar normalidad de los datos.

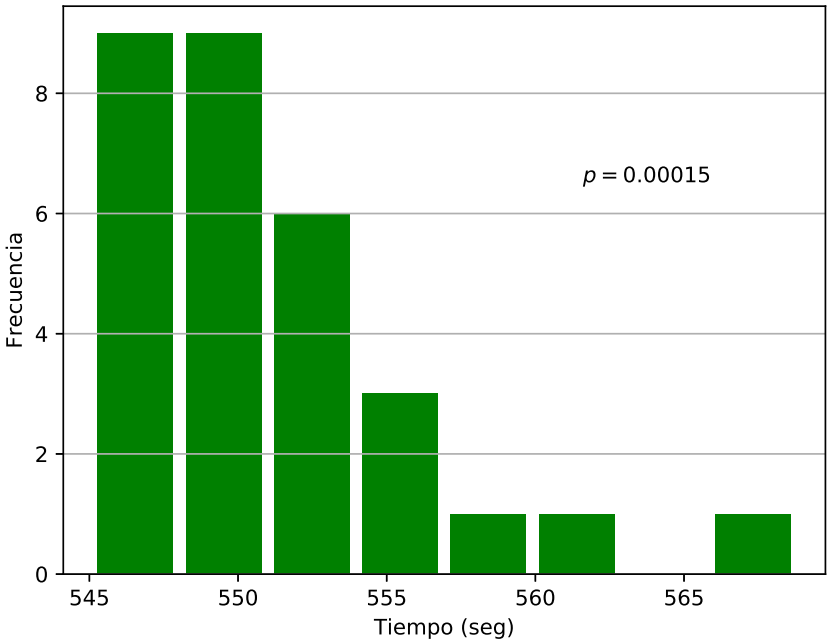

Figura 3: Histograma del tiempo de ejecucion utilizando el algoritmo `dfs_tree`.

4. Colores

La figura 4 muestra el histograma del tiempo de ejecucion de usar el algoritmo `greedy_color` el cual encuentra los colores adecuados para un grafo utlizando estrategias. Se rechaza normalidad de los datos.

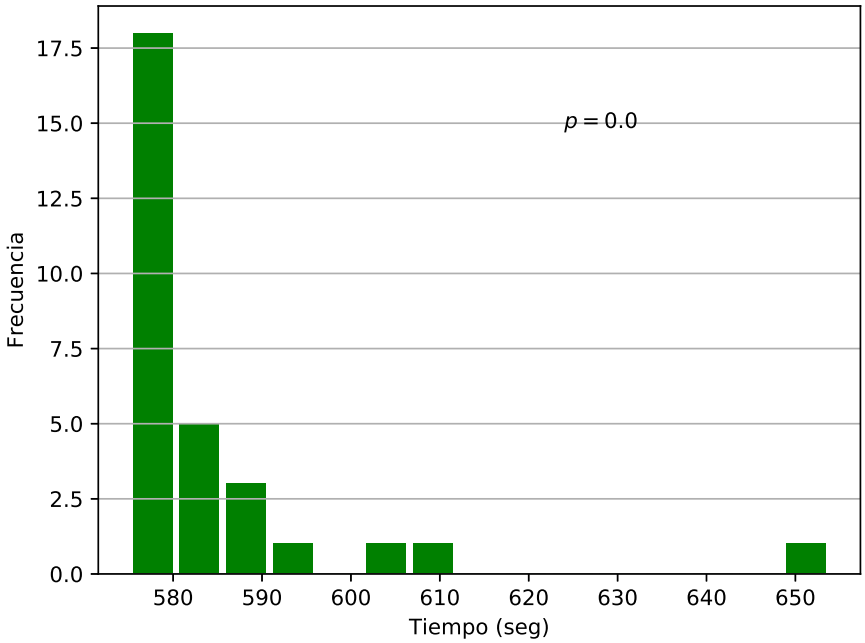

Figura 4: Tiempo de ejecucion del algoritmo `greedy_color`.

5. Maximo peso

Utilizando el algoritmo `max_weight_matching` se consigue el histograma de la figura 5, el algoritmo consiste en encontrar el camino con el maximo peso partiendo desde un nodo inicial. Se rechaza normalidad de los datos.

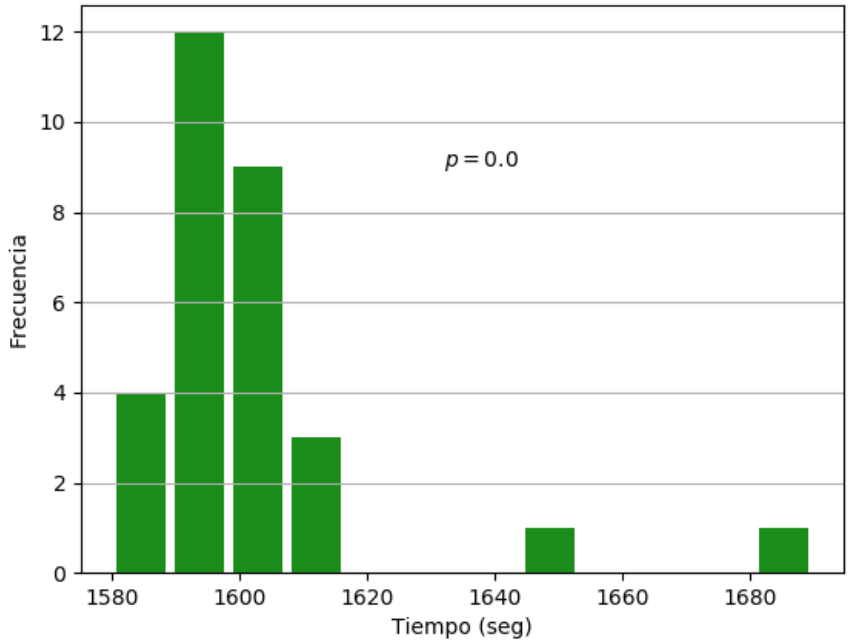

Figura 5: Histograma del tiempo de ejecucion del algoritmo `max_weight_matching`.

6. Conclusiones

La figura 6 muestra una grafica de dispercion de tiempo de ejecucion contra cantidad de nodos de los grafos. Se observa como varian los tiempos dependiedo la cantidad de nodos en el grafo.

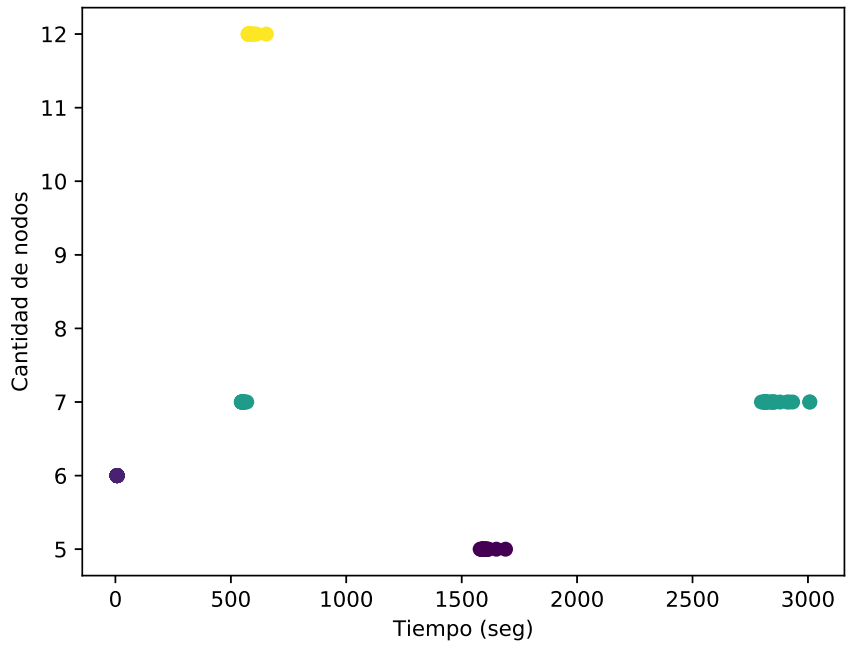

Figura 6: Grafica de dispercion de tiempos contra cantidad de nodos del grafo.

La figura 7 muestra los resultados de los tiempos de ejecucion contra la cantidad de aristas del grafo.

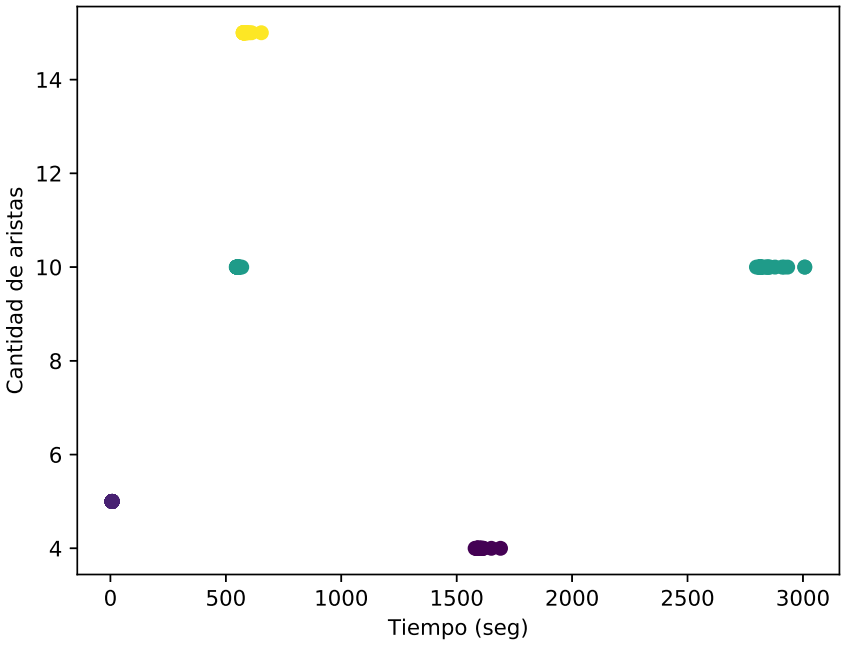

Figura 7: Grafica de dispercion de tiempos contra cantidad de aristas del grafo.

Referencias

[1] Schaeffer E. *Optimización de flujo en redes, 2019.*
https://elisa.dyndns-web.com/teaching/opt/flow/

[2] Tomihisa Kamada, Satoru Kawai. *An Algorithm for Drawing General Undirected Graphs*
Information Processing Letters, 1988.