Tarea 3 Optimización de Flujo en Redes

Orlando Lázaro Ruiloba Torres 5270

19 de marzo de 2019

1. Algoritmo ordenamiento topológico

Este algoritmo devuelve una lista de vértices en ordenamiento topológico, lo cual se traduce en que realiza una permutación no única de los vértices, de modo que una arista de u a v implica que u aparece antes de v en el orden de clasificación topológico [4].

El ordenamiento topológico es una adaptación simple pero útil de una búsqueda en profundidad. Sus pasos se describen a continuación [3]:

1.- Llamar a la búsqueda en profundidad para el grafo en cuestión. La principal razón por la que se invoca a la búsqueda en profundidad, es para calcular los tiempos de finalización para cada uno de los vértices.

2.- Almacenar los vértices en una lista en orden decreciente según el tiempo de finalización.

3.- Devolver la lista ordenada como resultado del ordenamiento topológico.

El código en python se muestra a continuación:

```
#Grafo 1

GDA = nx.DiGraph()

GDA.add_nodes_from(["E1","E2","E3"], bipartite=0)
GDA.add_nodes_from(["E4","E5","E6","E7"], bipartite=1)

GDA.add_edges_from([("E1","E4"),("E1","E5"),("E2","E6"),("E2","E7"),("E3","E4"),("E3","E7")])

replicas = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        t_s = nx.topological_sort(GDA)
    elapsed_time = time() - start_time
    replicas.append(elapsed_time)
print(replicas)
print(len(replicas))

normality_test=stats.shapiro(replicas)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f1,density=True)
first_edge, last_edge = np.min(replicas_f1),np.max(replicas_f1)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f1, bins=bin_edges, rwidth=0.8, color='orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma1.eps")
plt.show()

nx.draw(GDA, pos=nx.bipartite_layout(GDA,["E1","E2","E3"]), node_size = 2000, node_color = 'y',
       node_shape = 'o', with_labels=True)
plt.draw()
```

```
plt.savefig("GDA.eps")
plt.show(1)
```

A continuación en la figura 1, se muestra un histograma que ilustra el comportamiento de este algoritmo para las 30 veces en que se efectuaron las 100000 réplicas con sus respectivos tiempos de cómputo:

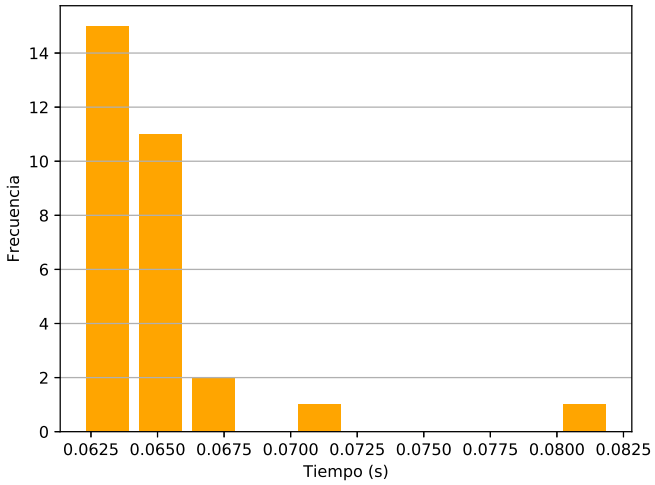

Figura 1: Histograma No.1

2. Algoritmo árbol de expansión mínimo

Un árbol de expansión mínimo es un subgrafo del grafo dado (un árbol) con la suma mínima de los pesos de sus aristas. Este algoritmo genera aristas en un bosque de expansión mínimo de un grafo ponderado no dirigido, este bosque es una unión de los árboles de expansión para cada componente conectado del grafo [6].

El código es el siguiente:

```
#Grafo 2

GNDC = nx.Graph()

GNDC.add_node("F")
GNDC.add_nodes_from(["C1","C2","C3","C4"])

GNDC.add_edge("F","C3", weight = 2)
GNDC.add_edge("C3","C4", weight = 1)
GNDC.add_edge("C4","C2", weight = 4)
GNDC.add_edge("C2","C1", weight = 1)
GNDC.add_edge("C1","F", weight = 2)

replicas_2 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        mst = nx.minimum_spanning_tree(GNDC)
    elapsed_time = time() - start_time
    replicas_2.append(elapsed_time)
print(replicas_2)
print(len(replicas_2))

normality_test=stats.shapiro(replicas_2)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f2,density=True)
first_edge, last_edge = np.min(replicas_f2),np.max(replicas_f2)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

```

```
plt.hist(replicas_f2, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma2.eps")
plt.show()

g = nx.random_layout(GNDC)

nx.draw(GNDC, node_size = 2000, node_color = 'y', node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("GNDC.eps")
plt.show(2)
```

En la figura 2, se muestra el histograma correspondiente:

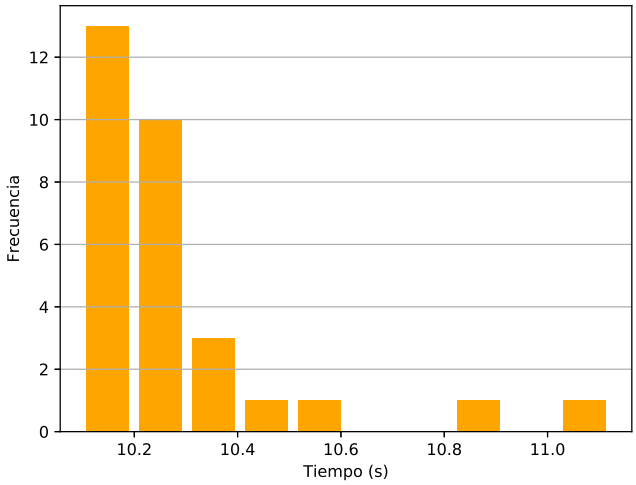

Figura 2: Histograma No.2

3. Algoritmo centralidad de intermediación

Este algoritmo calcula la centralidad de intermediación de cada vértice de un grafo dado [9], esta se encarga de medir en un grafo la tendencia de un vértice único a ser más central que todos los demás vértices en el mismo. Se basa en las diferencias entre la centralidad del vértice más central y la de todos los demás [1].

El código en python es:

```
#Grafo 3

GDC = nx.DiGraph()

GDC.add_nodes_from(["S1","S5"])
GDC.add_edges_from([("S1","S2"),("S2","S5"),("S3","S4"),("S4","S5"),("S5","S1")])

replicas_3 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        b_c = nx.betweenness_centrality(GDC, normalized=False)
    elapsed_time = time() - start_time
    replicas_3.append(elapsed_time)
print(replicas_3)
print(len(replicas_3))

normality_test=stats.shapiro(replicas_3)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f3,density=True)
first_edge, last_edge = np.min(replicas_f3),np.max(replicas_f3)
```

```
23
24 n_equal_bins = 10
25 bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)
26
27 plt.hist(replicas_f3, bins=bin_edges, rwidth= 0.8, color= 'orange')
28 plt.ylabel('Frecuencia')
29 plt.xlabel('Tiempo (s)')
30 plt.grid(axis='y', alpha= 0.5)
31 plt.savefig("Histograma3.eps")
32 plt.show()
33
34 p = nx.spring_layout(GDC)
35
36 nx.draw(GDC, node_size = 1000, node_color = 'y', node_shape = 'o', with_labels=True)
37 plt.draw()
38 plt.savefig("GDC.eps")
39 plt.show(3)
```

El histograma para este algoritmo aparece a continuación:

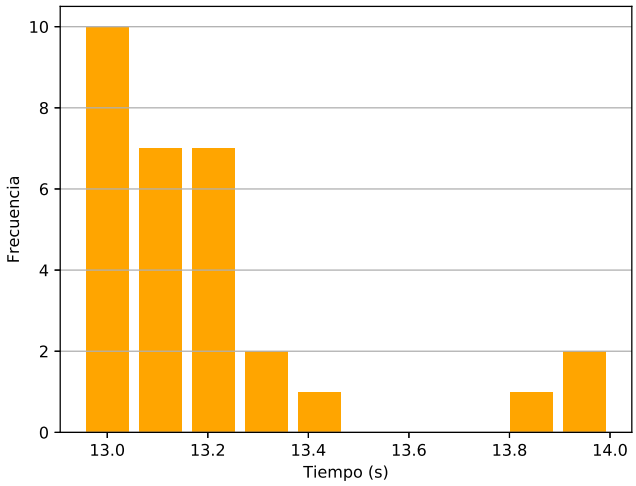

Figura 3: Histograma No.3

4. Algoritmo árbol dfs

Este algoritmo devuelve un árbol orientado, construido a partir de una búsqueda en profundidad desde el origen [7]. La técnica de cruce de DFS (Depth First Search o Búsqueda en Profundidad) de un grafo produce un árbol de expansión como resultado final, es decir, un grafo que no contiene ningún ciclo. En este caso, la estructura de datos se usa con el tamaño máximo del número total de vértices en el grafo para implementar el recorrido DFS [8].

El código en python es el siguiente:

```
1  #Grafo 4
2
3  GNDA = nx.Graph()
4
5  GNDA.add_node("Gerente")
6  GNDA.add_nodes_from(["Subgerente","Asistente","Coordinador","Planificador","Organizador","Supervisor"])
7
8  GNDA.add_edges_from([("Gerente","Subgerente"),("Subgerente","Asistente")])
9  GNDA.add_edges_from([("Asistente","Coordinador"),("Asistente","Planificador")])
10 GNDA.add_edges_from([("Asistente","Organizador"),("Asistente","Supervisor")])
11
12 replicas_4 = []
13 for j in range(30):
14     start_time = time()
15     for i in range (100000):
```

```
        dfs = nx.dfs_tree(GNDA)
    elapsed_time = time() - start_time
    replicas_4.append(elapsed_time)
print(replicas_4)
print(len(replicas_4))

normality_test=stats.shapiro(replicas_4)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f4,density=True)
first_edge, last_edge = np.min(replicas_f4),np.max(replicas_f4)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f4, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma4.eps")
plt.show()

p = nx.spring_layout(GNDA)

nx.draw(GNDA, node_size = 5500, node_color = 'y', node_shape = 's', with_labels=True)
plt.draw()
plt.savefig("GNDA.eps")
plt.show(4)
```

En la figura 4 se puede apreciar el histograma referido al algoritmo que se analiza en esta sección:

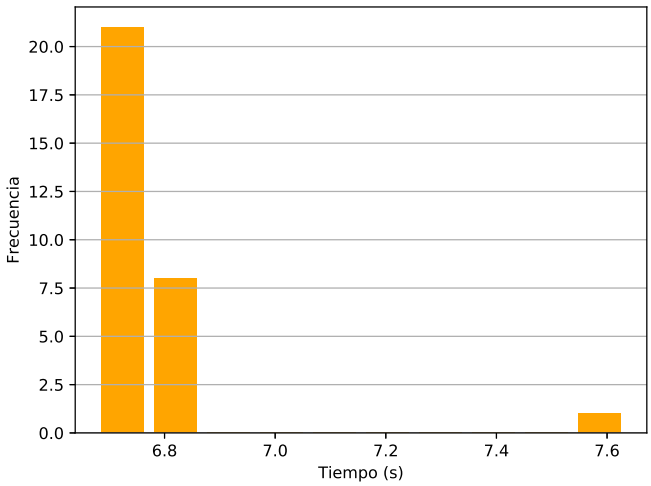

Figura 4: Histograma No.4

5. Algoritmo camarilla máxima

El objetivo que persigue este algoritmo, es encontrar el mayor grupo de nodos en un grafo, de forma tal que todos estén conectados entre sí [2]. Una camarilla en un grafo no dirigido G = (V, E) es un subconjunto del conjunto de vértices C, de manera que por cada dos vértices en C, existe una arista que conecta los dos. La camarilla máxima es aquella que posee el mayor tamaño posible en un grafo dado [5].

El código es:

```
#Grafo 5

MNDR = nx.MultiGraph()

MNDR.add_nodes_from(["1","5"])
MNDR.add_edges_from([("1","2"),("2","3"),("2","4"),("2","5"),("3","5"),("4","5"),("3","5"),
```

```
                    ("4","5"),("5","5")])

color_map = []
for node in MNDR:
    if (node == "5"):
        color_map.append('yellow')
    else:
        color_map.append('red')

replicas_5 = []
for j in range(30):
    start_time = time()
    for i in range (100000):
        cliques = nx.cliques_containing_node(MNDR)
    elapsed_time = time() - start_time
    replicas_5.append(elapsed_time)
print(replicas_5)
print(len(replicas_5))

normality_test=stats.shapiro(replicas_5)
print(normality_test)

hist, bin_edges=np.histogram(replicas_f5,density=True)
first_edge, last_edge = np.min(replicas_f5),np.max(replicas_f5)

n_equal_bins = 10
bin_edges = np.linspace(start=first_edge, stop=last_edge,num=n_equal_bins + 1, endpoint=True)

plt.hist(replicas_f5, bins=bin_edges, rwidth= 0.8, color= 'orange')
plt.ylabel('Frecuencia')
plt.xlabel('Tiempo (s)')
plt.grid(axis='y', alpha= 0.5)
plt.savefig("Histograma5.eps")
plt.show()

p = nx.spectral_layout(MNDR)

nx.draw(MNDR, node_size = 1000, node_color = color_map, node_shape = 'o', with_labels=True)
plt.draw()
plt.savefig("MNDR.eps")
```

A continuación, en la figura siguiente se muestra el histograma:

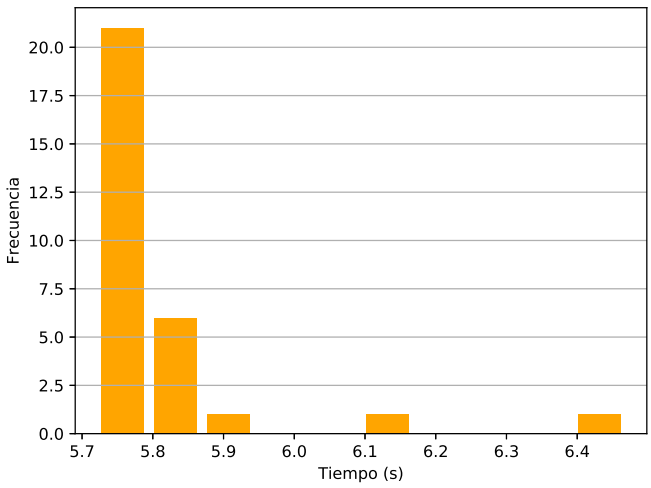

Figura 5: Histograma No.5

Vale la pena recalcar que se realizó la prueba de normalidad Shapiro Wilk, para los datos recolectados del análisis de cada algoritmo por separado, arrojando la misma, que en ninguno de los casos, los datos poseían una distribución normal, por lo cual no se pudo tomar los valores de la media, ni de la desviación estándar como referencia en el análisis.

6. Discusión de conclusiones

A continuación se muestran las dos gráficas de dispersión necesarias para efectuar el análisis del comportamiento de las combinaciones algoritmo-grafo:

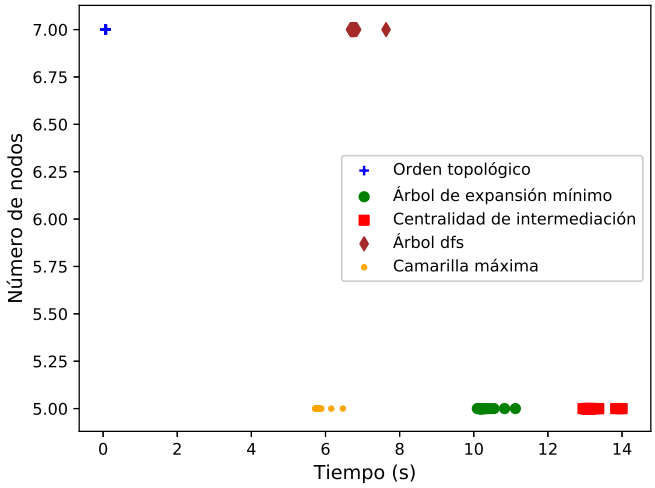

Figura 6: Gráfica No.1

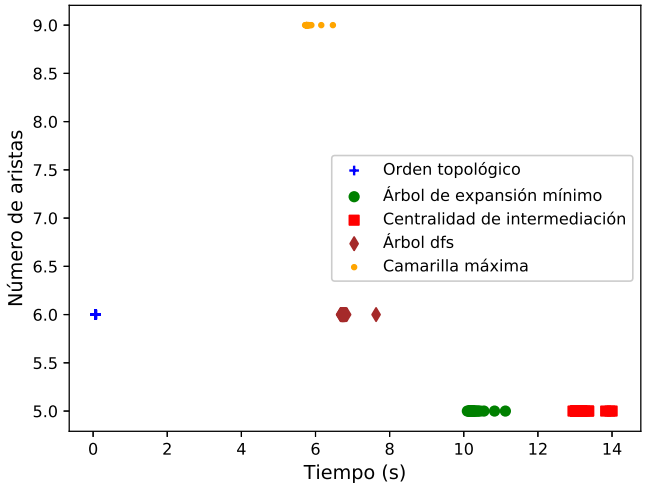

Figura 7: Gráfica No.2

A partir del análisis de las gráficas anteriores, se puede concluir que la combinación algoritmo-grafo que mostró tiempos de ejecución más pequeños, para el número de réplicas realizadas (100000), fue la de orden topológico y la que arrojó mayores tiempos, fue la correspondiente a centralidad de intermediación, en el resto de las combinaciones los tiempos toman valores mayores que 5 y menores a 12 segundos. También se puede apreciar que no existe mucha dispersión entre los tiempos obtenidos para las diferentes combinaciones ya que se encuentran en un rango de valores cercanos.

Referencias

[1] Sunil Kumar Raghavan Unnithan. Balakrishnan Kannan and Madambi Jathavedan. Betweenness centrality in some classes of graphs. Last access in 03-17-2019 to http://www.hindawi.com/journals/ijcom/2014/241723/, 2014.

[2] James MacCaffrey. Algoritmos tabú y el clique máximo. Last access in 03-14-2019 to http://msdn.microsoft.com/es-es/magazine/hh580741.aspx, 2011.

[3] Brad Miller and David Ranum. Problem solving with algorithms and data structures using python. Last access in 03-17-2019 to http://interactivepython.org/runestone/static/pythoned/Graphs/OrdenamientoTopologico.html, 2013.

[4] Networkx. Topological sort. Last access in 03-13-2019 to http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.dag.topologicalsort.html, 2014.

[5] Networkx. Max clique. Last access in 03-17-2019 to http://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.algorithms.approximation.clique.maxclique.html, 2014.

[6] Networkx. Minimum spanning tree. Last access in 03-17-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.mst.minimumspanningtree.html, 2015.

[7] Networkx. Dfs tree. Last access in 03-14-2019 to http://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.traversal.depthfirstsearch.dfstree.html, 2015.

[8] Ace Technosoft Team. Graph traversals. Last access in 03-17-2019 to http://acetechnosoft.com/ds/graphtraversal.php, 2018.

[9] GraphStream Team. Betweenness centrality. Last access in 03-14-2019 to http://graphstream-project.org/doc/Algorithms/Betweenness-Centrality/, 2018.