Tarea 3

5271

19 de marzo de 2019

1. Algoritimos utilizados en las mediciones

Se seleccionaron cinco de los doce algoritmos para ejecutarlos sobre cinco de los grafos de tareas anteriores, midiendo los tiempos de ejecución de cada algoritmo y realizando un análisis de los datos obtenidos. Dado los requerimientos de los algoritmos escogidos para la realización de esta tarea, se tuvo modificar los dichos grafos agregándoles más nodos y se convirtiéndolos a grafos no dirigidos, como se muestra en la figura 1.

Los algoritmos escogidos fueron los siguientes:

- *Make max clique graph* (encuentra las camarillas máximas y las trata como vértices . Los vértices están conectados si tienen miembros comunes en el grafo original) [1].
- *Betweenness centrality* (calcula la centralidad de intermediación de ruta más corta para los vértices . La centralidad de la intermediación es una forma de detectar la cantidad de influencia que un vértice tiene sobre el flujo de información en un grafo) [2].
- *Greedy color* (colorea un grafo usando varias estrategias de coloración codiciosa. Las estrategias se pueden describir como el intento de colorear un grafo con la menor cantidad de colores posibles, donde ningún vecino puede tener el mismo color) [3].
- *Maximal matching* (encuentra una cardinalidad máxima de coincidencia en el grafo. Una coincidencia es un subconjunto de aristas en las que no se produce ningún vértice más de una vez. La cardinalidad de una coincidencia es el número de arcos coincidentes. Se uso este algoritmo en lugar del*min maximal matching* ya que este daba un error en networkx)[4].
- *Dfs tree* (crea un árbol orientado hacia el retorno construido a partir de una fuente en una búsqueda en profundidad) [5].

```python
import random as rnd
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
def Crear_Grafos(nombre, size, int_distancia):

    G=nx.Graph()
    nodes = []
    for i in range(size):
        nodes.append(i)
    print(nodes)
    for i in nodes:
        idx = nodes.index(i) + 1
        print(idx)
        for j in nodes[idx:len(nodes)]:
            if rnd.randint(0,80)==1:
                G.add_edge(i,j, distancia=rnd.randint(1,
    int_distancia))
    print(G.edges)

    df = pd.DataFrame()
    df = nx.to_pandas_adjacency(G, dtype=int, weight='distancia')
    df.to_csv(nombre+".csv", index=None, header=None)
    plt.figure(figsize=(15,15))
    position = nx.spring_layout(G, scale=5,iterations=200)
    nx.draw_networkx_nodes(G, position, node_size=50, node_color="#
    de8919", node_shape="<")
    nx.draw_networkx_edges(G, position, width=0.5, edge_color='
    black' )


    plt.axis("off")
    plt.savefig(nombre + ".png",bbox_inches='tight')
    plt.savefig(nombre + ".eps",bbox_inches='tight')
    plt.show(G)

Crear_Grafos("1NoDirigido",800, 20)
```

Genera_grafos.py

2. Medición de los tiempos de ejecución

Con el objetivo de realizar las mediciones de los tiempos de ejecución de los algoritmos seleccionados se desarrolló el siguiente código.

En primer lugar, se crea una función que lee de un archivo la matriz de adyacencia de un grafo y convierte dicha matriz en un grafo.

```
def Leer_grafos(nomb):
    dr = pd.read_csv( nomb , header = None)
    A = nx.from_pandas_adjacency(dr, create_using = nx.Graph())
    print(A.edges)
    A.name=(i)
    return (A)
```

Corer_Algoritmos.py

Se crea una función para cada algoritmo. En esta se le pasa el grafo leído por parámetros al algoritmo en cuestión, en este proceso se mide el tiempo de ejecución del algoritmo y se repite treinta veces la ejecución, los tiempos son almacenados en una lista. Con las mediciones acumuladas se calcula la media, la mediana y la desviación estándar.

```
def dfs_tree(grafo):
    tiempo=[]
    tiempo_inicial = dt.datetime.now()
    for i in range(30):
        tiempo_inicial = dt.datetime.now()
        for j in range(1000):
            nx.dfs_tree(grafo)
        tiempo_final = dt.datetime.now()
        tiempo_ejecucion = (tiempo_final - tiempo_inicial).
    total_seconds()
        tiempo.append(tiempo_ejecucion)

    media=nup.mean(tiempo)
    desv=nup.std(tiempo)
    mediana=nup.median(tiempo)
    datos["algoritmo"].append("dfs_tree")
    datos["grafo"].append(grafo.name)
    datos["cant_vertice"].append(grafo.number_of_nodes())
    datos["cant_arista"].append(grafo.number_of_edges())
    datos["media"].append(media)
    datos["desv"].append(desv)
    datos["mediana"].append(mediana)
    return datos

def Greedy_color(grafo):
    tiempo=[]
    for i in range(30):
        tiempo_inicial = dt.datetime.now()
```

```
        for j in range(1000):
            nx.greedy_color(grafo)
        tiempo_final = dt.datetime.now()
        tiempo_ejecucion = (tiempo_final - tiempo_inicial).
    total_seconds()
        tiempo.append(tiempo_ejecucion)

    media=nup.mean(tiempo)
    desv=nup.std(tiempo)
    mediana=nup.median(tiempo)
    datos["algoritmo"].append("greedy_color")
    datos["grafo"].append(grafo.name)
    datos["cant_vertice"].append(grafo.number_of_nodes())
    datos["cant_arista"].append(grafo.number_of_edges())
    datos["media"].append(media)
    datos["desv"].append(desv)
    datos["mediana"].append(mediana)
    return datos
```

Corer_Algoritmos.py

En este otro fragmento se muestra donde se ejecutan las funciones y se guarda el archivo con los datos recopilados.

```
listgrafoNoDi=["1NoDirigido.csv","2NoDirigido.csv","3NoDirigido.csv
    ","4NoDirigido.csv","5NoDirigido.csv"]

for i in listgrafoNoDi:
   l=Leer_grafos(i)
   make_max_clique_graph(l)
for i in listgrafoNoDi:
   l=Leer_grafos(i)
   Betweenness_centrality(l)
for i in listgrafoNoDi:
   l=Leer_grafos(i)
   Greedy_color(l)
for i in listgrafoNoDi:
   l=Leer_grafos(i)
   Maximal_matching(l)
for i in listgrafoNoDi:
    l=Leer_grafos(i)
    dfs_tree(l)

df = pd.DataFrame(datos)
df.to_csv("salid.csv", index=None)
```

Corer_Algoritmos.py

3. Resultados del Análisis de los datos

Con los datos recopilado de las ejecuciones de los Algoritmos par los cinco grafos se realizó un histograma, como se muestra en la figura 2 y dos diagramas de dispersión, uno que muestra la relación de la media de los tiempos de ejecución de cada algoritmo con la cantidad de vértices y el otro con la cantidad de aristas, como se muestra en la figura 3 y 4 respectivamente. En estas figuras se puede observar que el algoritmo con los tiempos de ejecución más pequeños es el *Maximal matching*, así como que el de mayores tiempos es el *Betweenness centrality*.

```
dr = pd.read_csv( "salid.csv" )
print(dr["media"][10:15])
fig,axes = plt.subplots(nrows=2, ncols=3, figsize=(12, 6))
axes[0,0].hist(dr["media"][:5],bins=5, color="#932525", alpha=1,
    edgecolor = 'black',  linewidth=1)
axes[0,0].set_title(dr["algoritmo"][1])

axes[0,0].set_ylabel('Frecuencia de ocurrencia')
axes[0,1].hist(dr["media"][5:10],bins=5, color="#ee9110", alpha=1,
    edgecolor = 'black',  linewidth=1)
axes[0,1].set_title(dr["algoritmo"][6])

axes[0,1].set_ylabel('Frecuencia de ocurrencia')
axes[0,2].hist(dr["media"][10:15],bins=5,color="#adcb18", alpha=1,
    edgecolor = 'black',  linewidth=1)
axes[0,2].set_title(dr["algoritmo"][11])

axes[0,2].set_ylabel('Frecuencia de ocurrencia')
axes[1,0].hist(dr["media"][15:20],bins=5,color="#129f10", alpha=1,
    edgecolor = 'black',  linewidth=1)
axes[1,0].set_title(dr["algoritmo"][16])

axes[1,0].set_ylabel('Frecuencia de ocurrencia')
axes[1,1].hist(dr["media"][20:25],bins=5,color="#093ea8", alpha=1,
    edgecolor = 'black',  linewidth=1)
axes[1,1].set_title(dr["algoritmo"][21])

axes[1,1].set_ylabel('Frecuencia de ocurrencia ')
axes[1,2].set_axis_off()


plt.savefig("Histograma.eps",bbox_inches='tight')
plt.savefig("Histograma.png",bbox_inches='tight')
plt.show()
```

Histograma.py

```
size = (25 * dr["cant_arista"][5:10] / dr["cant_vertice"][5:10])
color_names = ["#932525", "#129f10", "#093ea8", "#adcb18", "#ee9110"]
figure, axes = plt.subplots(figsize=(8, 8))
axes.scatter(dr["media"][:5], dr["cant_vertice"][:5],
             s=size, c=color_names, marker="D",
             label="Make max clique graph", alpha=0.8, edgecolors='black')
axes.scatter(dr["media"][5:10], dr["cant_vertice"][5:10],
             s=size, c=color_names, marker="s",
             label="Betweenness centrality", alpha=0.8, edgecolors='black')
axes.scatter(dr["media"][10:15], dr["cant_vertice"][10:15],
             s=size, c=color_names, marker="8",
             label="Greedy color algorithm", alpha=0.8, edgecolors='black')
axes.scatter(dr["media"][15:20], dr["cant_vertice"][15:20],
             s=size, c=color_names, marker=">",
             label="Maximal matching", alpha=0.8, edgecolors='black')
axes.scatter(dr["media"][20:25], dr["cant_vertice"][20:25],
             s=size, c=color_names, marker="*",
             label="Dfs_tree", alpha=0.8, edgecolors='black')

axes.set_ylabel("Vertices ", fontsize=12, fontfamily="arial", fontweight="bold")
axes.set_xlabel("Tiempo de Ejecucion", fontsize=12, fontfamily="arial", fontweight="bold")
plt.ylim((min(dr["cant_vertice"])-30, max(dr["cant_vertice"]) + 30))

axes.legend()
plt.savefig("DiagramVertices.eps",bbox_inches='tight')
```

Histograma.py

Referencias

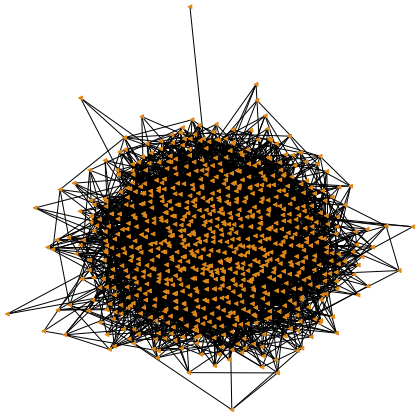
[1] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.clique.make_max_clique_graph.html. Accessed: 18-03-2019.

[2] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html#networkx.algorithms.centrality.betweenness_centrality. Accessed: 18-03-2019.

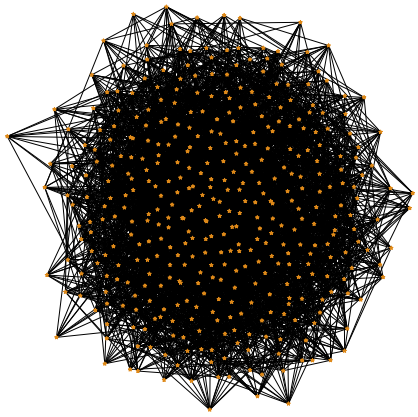
[3] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.coloring.greedy_color.html. Accessed: 18-03-2019.

[4] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.matching.maximal_matching.html. Accessed: 18-03-2019.

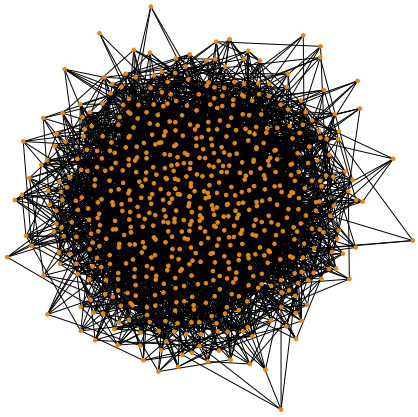
[5] Desarrolladores NetworkX. https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.traversal.depth_first_search.dfs_tree.html#networkx.algorithms.traversal.depth_first_search.dfs_tree. Accessed: 18-03-2019.



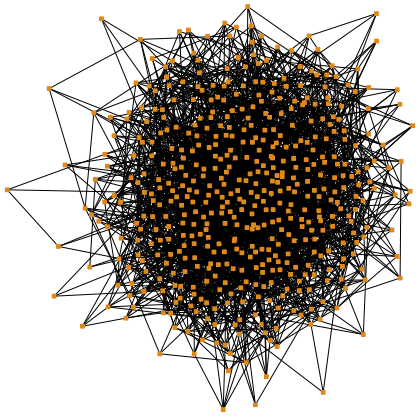
(a) grafo1, 800 vértices, 3966 aristas



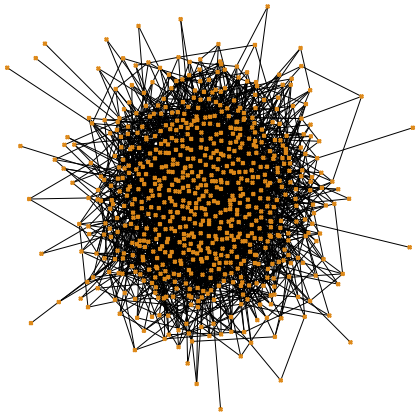
(b) grafo2, 400 vértices, 3780 aristas



(c) grafo3, 640 vértices, 3850 aristas



(d) grafo4, 500 vértices, 2439 aristas



(e) grafo5, 710 vértices, 2478 aristas

Figura 1: Grafos generados para la tarea 3.

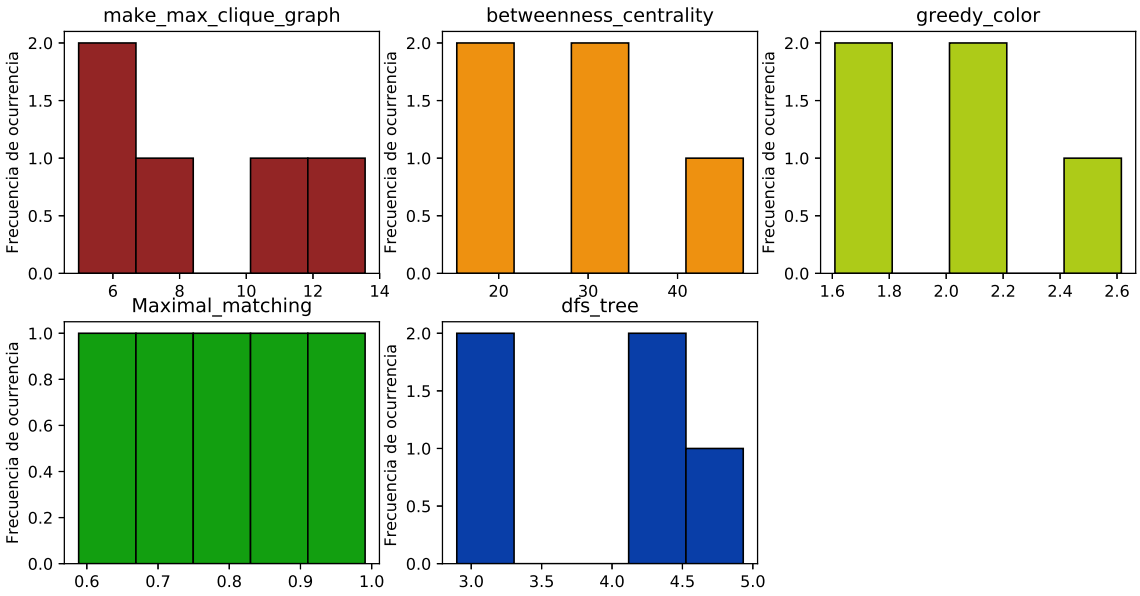


Figura 2: Histograma que de los cinco algoritmos con cinco grafos.

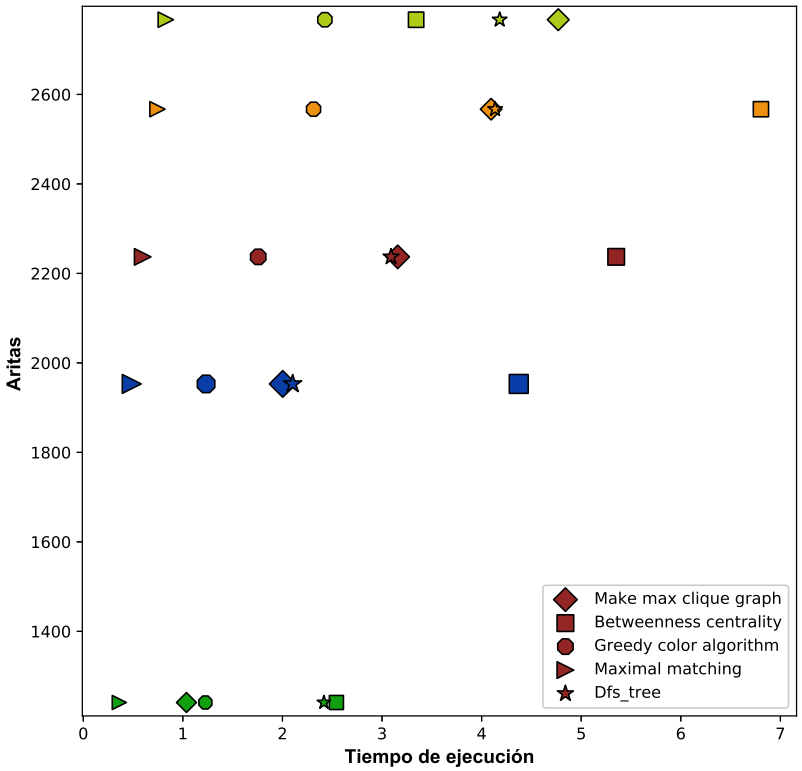


Figura 3: Diagrama de dispersión que muestra la relación entre los tiempos de ejecución y la cantidad de vértices, los diferentes colores indican los cinco grafos.

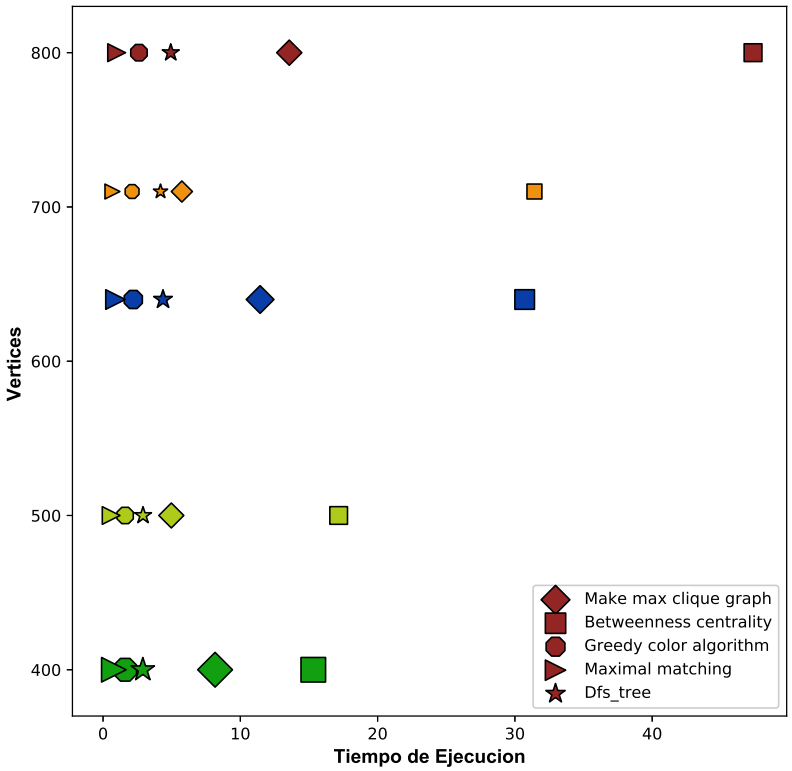


Figura 4: Diagrama de dispersión que muestra la relación entre los tiempos de ejecución y la cantidad de aristas,los diferentes colores indican los cinco grafos.