PRÁCTICA 2

ALGORITMIA

Luis Núñez Fernández y Lucía Chacón Sanabria
Grupo: 2

1. Componentes Conexas:

Escribimos tres funciones:

- *connected(n, e):*
    - Recibe un entero 'n' y una lista de tuplas de enteros 'e' representante de los arcos del grafo con n nodos. La función devuelve la estructura de conjuntos disjuntos resultante.
- *connected_count(p):*
    - Recibe los conjuntos disjuntos de *connected* y devuelve el número de componentes conexas del grafo.
- *connected_sets(p):*
    - Recibe los conjuntos disjuntos de *connected* y devuelve una lista de listas de enteros.

Para comprobar el correcto funcionamiento de estas funciones, usamos el ejemplo que nos viene en la práctica:

```
lst = [(1,4), (3,4), (2,5)]

s = connected(6, lst)
n = connected_count(s)
ccp = connected_sets(s)

print(n)
print(s)
print(ccp)
```

El output que nos sale es el siguiente:

```
drejor@luis-DESKTOPUBUNTU:~/Desktop/universidad/git/Algorit/Practica2$ python3 AEDATA_code.py
3
[-1, -2, -2, 1, 1, 2]
[[0], [1, 3, 4], [2, 5]]
```

Donde "3" nos indica el número de componentes conexas, la siguiente línea nos indica la estructura de conjuntos disjuntos y la última línea la lista de enteros, que cada lista representa una componente conexa del grafo y contiene los nodos que forman parte de esa componente conexa.

2. Algoritmo de Kruskal

Escribimos dos funciones:

- *kruskal(n, E)*:
    - Esta función se encarga de ejecutar el algoritmo de Kruskal. La función devuelve un grafo, donde n es el número de nodos y E' es el conjunto de arcos que forman el árbol.
- *k_weight(n, E)*:
    - Recibe la lista de arcos producidos por la anterior función y devuelve el peso del árbol.

Para comprobar el correcto funcionamiento de estas dos funciones, hemos hecho el siguiente código:

```
E = [(0, 1, 10), (0, 3, 3), (1, 2, 1), (2, 3, 1), (2, 5, 1), (3, 4, 10), (4, 5, 1)]
n = 6

n_k, E_k = kruskal(n, E)

print(E_k)
print(k_weight(n_k, E_k))
```

El output que nos sale es el siguiente:

```
e506301@6B-8-18-8:~/UnidadH/Algoritmia/Practica2 copia/Practica2$ python3 AEDATA_code.py
[(1, 2, 1), (2, 3, 1), (2, 5, 1), (4, 5, 1), (0, 3, 3)]
7
```

Donde la primera línea es el grafo, y "7" es el peso del árbol.

3. *Tiempo de Ejecución*

Escribimos dos funciones:

- *erdos_conn(n, m)*:
  - Recibe los parámetros n y m, donde n es el número de nodos y m son los vecinos de cada nodo y devuelve la lista de arcos.

Para probar esta función, hemos puesto 10 nodos y tres vecinos para cada nodo:

```
print(erdos_conn(10, 3))
```

El output que nos sale es el siguiente:

```
e506301@6B-8-18-8:~/UnidadH/Algoritmia/Practica2 copia/Practica2$ python3 AEDATA_code.py
[(0, 0, 1), (1, 0, 1), (2, 1, 0), (3, 0, 0), (4, 2, 1), (5, 2, 0), (6, 2, 0), (7, 6, 1), (8, 4, 0), (9, 0, 1)]
```

- *time_kruskal(n, m, n_graphs)*:
  - Dados los parámetros n y m, construye n_graphs grafos aleatorios. Luego calcula el tiempo de ejecución de cada uno de ellos formando una media y una varianza.

Para medir el tiempo de ejecución de nuestro algoritmo hemos probado a ejecutar Kruskal con varios tamaños de grafo con la siguiente función:

```
def times_kruskal_erdos(mp, me, mincr, n_graphs):
    # Mide los tiempos de ejecución del algoritmo de Kruskal en varios grafos
   times = []
   for m in range (mp, me, mincr):
       n = 2*m
       times.append((n, time_kruskal(n, m, n_graphs)[0]))
   return times
```

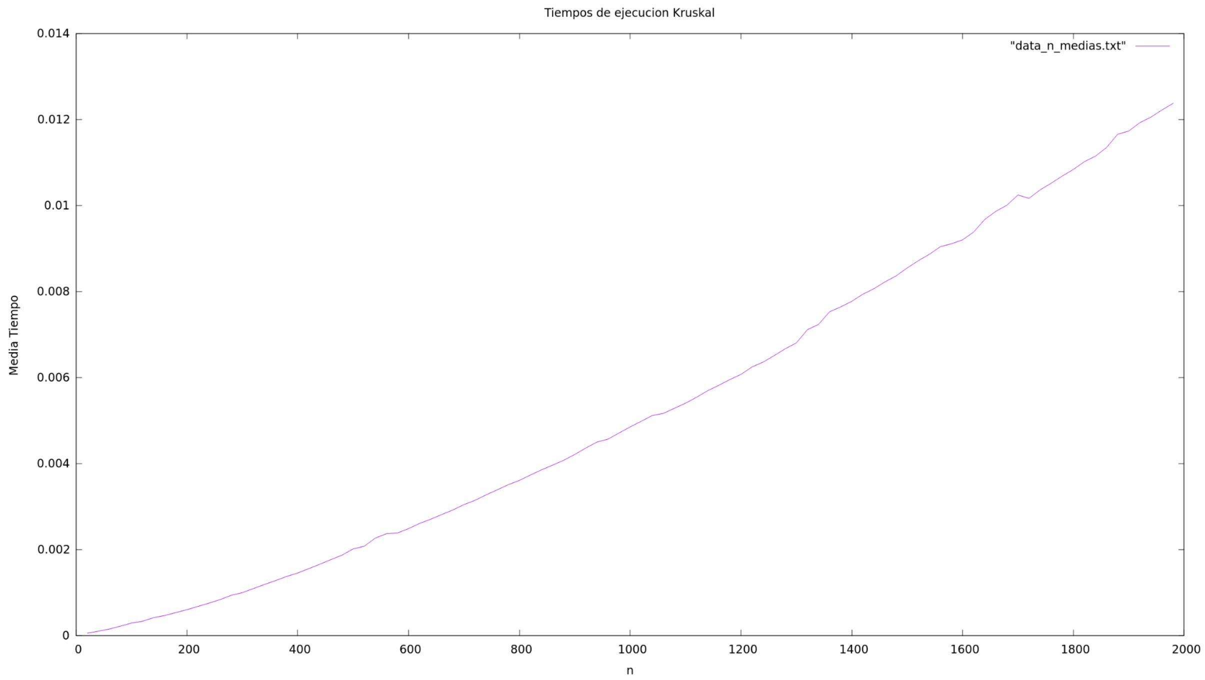
Como en la práctica se nos indica que para que la función *erdos_conn()* funcione correctamente debemos poner m << n, por lo que vamos a poner que m sea siempre 2n.

Ejecutamos la función con los siguientes parámetros y relacionamos n con el tiempo medio de cada ejecución de *time_kruskal()* e imprimimos los resultados con el siguiente fragmento de código:

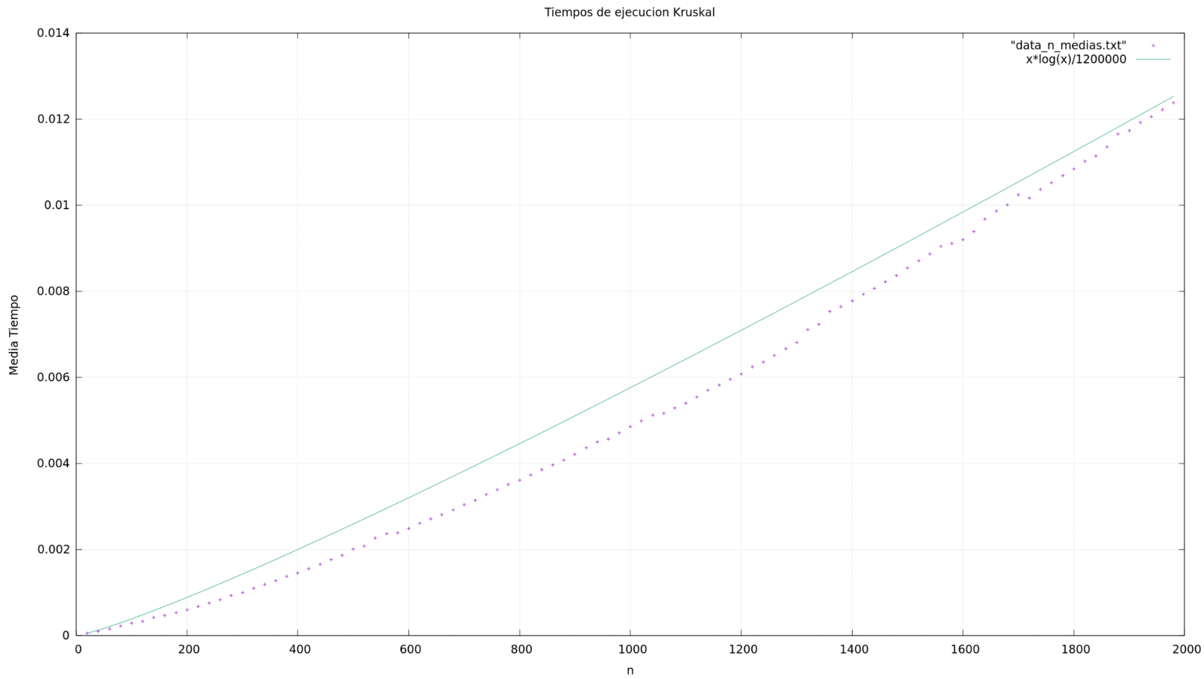
```
times = times_kruskal_erdos(10, 1000, 10, 20)

for i in times:
   print(i)
```

Podemos observar que se va a ejecutar con m desde 10 hasta 1000 con un incremento de 10, lo que significa que la variable *n* irá desde 20 hasta 2000 con un incremento de 20.

Utilizamos gnuplot para graficar los resultados contrastando la n con la media de tiempo de ejecución dando como resultado la siguiente gráfica:



Para hacer un análisis más exhaustivo, vamos a compararlo con el tiempo de ejecución teórico de nuestra función de Kruskal, el cual es O(n log(n)):



Como podemos ver nuestro tiempo de ejecución es muy similar a la gráfica O(n log(n)) lo que verifica la eficacia deseada del algoritmo.

4. El Viajante de Comercio

Escribimos cinco funciones:

- *dist_matrix(n_cities, w_max = 10)*
  - Esta función viene con la práctica y genera una matriz de conexiones del grafo para poder analizar los caminos de forma más eficiente.
- *greedy_tsp(dist_m, node_ini)*
  - Recibe una matriz de distancias de *dist_matrix()* y un nodo inicial y devuelve, a través de un algoritmo codicioso, una lista de valores representando las ciudades a recorrer y volviendo siempre a la de origen.
- *len_circuit(circuit, dist_m)*
  - Recibe el circuito generado por *greedy_tsp()* y la matriz de distancias de *dist_matrix()* y devuelve el coste o longitud del camino.
- *repeated_greedy_tsp(dist_m)*
  - Recibe la matriz de distancias o costes de *dist_matrix()* y ejecuta *greedy_tsp()* con cada nodo inicial posible de la matriz de conexiones y devuelve el circuito óptimo.
- *exhaustive_tsp(dist_m)*
  - Recibe la matriz de distancias o costes de *dist_m()* y prueba todas las permutaciones posibles (añadiendo al final el nodo inicio para convertirlo en circuito) con todos los nodos de la matriz y devuelve el circuito óptimo.

    Ejemplo: Si tenemos una matriz con los nodos (0, 1, 2) esta función probará la función *len_circuit()* de los siguientes circuitos:

    [[0, 1, 2, 0], [0, 2, 1, 0], [1, 0, 2, 1], [1, 2, 0, 1], [2, 0, 1, 2], [2, 1, 0, 2]]

Para comprobar el correcto funcionamiento de estas funciones, hemos hecho el siguiente código, generando una matriz de nueve:

```
m = dist_matrix(9)

print(m)
print()

print(greedy_tsp(m, 0))

print(repeated_greedy_tsp(m))
print(exhaustive_tsp(m))
```

El output es el siguiente:

```
e506301@6B-8-18-8:~/UnidadH/Algoritmia/Practica2 copia/Practica2$ python3 AEDATA_code.py
[[0, 5.795640557760659, 6.391506443221972, 6.879221206200535, 0.6985130434747289, 3.3000244269900536, 5.109503303357202, 3.4460689758712175, 5.285398064246362], [5.795640557760659, 0, 5.7313
80785913061, 2.7407715567473137, 5.504761419906807, 2.4640236385247882, 6.396521027660117, 5.938208971189292, 6.512446112190904], [6.391506443221972, 5.731380785913061, 0, 1.737460279493425,
 6.519641434403268, 3.321089045772888, 3.0995167061792173, 5.6634454631193005, 1.8353220022505445], [6.879221206200535, 2.7407715567473137, 1.737460279493425, 0, 0.7114575750655011, 3.153671
4168864584, 4.906749633116322, 2.6551955710354833, 2.3065801193428195], [0.6985130434747289, 5.504761419906807, 6.519641434403268, 0.7114575750655011, 0, 5.544754530484295, 2.37641505001141,
 1.7705334613351487, 0.7726174911338152], [3.3000244269900536, 2.4640236385247882, 3.321089045772888, 3.1536714168864584, 5.544754530484295, 0, 6.555650930153088, 7.664252907986212, 3.945943
1158452536], [5.109503303357202, 6.396521027660117, 3.0995167061792173, 4.906749633116322, 2.37641505001141, 6.555650930153088, 0, 6.749418779219912, 6.02348800764171], [3.4460689758712175,
5.938208971189292, 5.6634454631193005, 2.6551955710354833, 1.7705334613351487, 7.664252907986212, 6.749418779219912, 0, 6.781804703547442], [5.285398064246362, 6.512446112190904, 1.835322002
2505445, 2.3065801193428195, 0.7726174911338152, 3.9459431158452536, 6.02348800764171, 6.781804703547442, 0]]

[0, 4, 3, 2, 8, 5, 1, 7, 6, 0]
[4, 0, 5, 1, 3, 2, 8, 6, 7, 4]
[0, 4, 6, 2, 8, 5, 1, 3, 7, 0]
```

Donde nos genera la matriz, luego [0, 4, 3, 2, 8, 5, 1, 7, 6, 0] es la lista de valores representando las ciudades a recorrer y volviendo siempre a la de origen, es decir, la función *greedy_tsp*. Luego la lista [4, 0, 5, 1, 3, 2, 8, 6, 7, 4] es el circuito óptimo, es decir, hemos probado la función *repeated_greedy_tsp.* Y por último, la lista [0, 4, 6, 2, 8, 5, 1, 3, 7, 0] es el circuito óptimo, pero esta vez usando la función *exhaustive_tsp*.

5. Permutaciones

Escribimos una función:

- *permute(lst)*:
    - Recibe en entrada una lista y devuelve una lista de listas, donde cada una de esas listas es una permutación de *lst*.

Hemos comprobado que funciona la función para una serie de valores (3, 4, 5):

```
l1 = permute([1, 2, 3])
print("Numero de permutaciones generadas: " + str(len(l1)) +
"\npermutaciones:\n" + str(l1), end="\n\n")
l2 = permute([1, 2, 3, 4])
print("Numero de permutaciones generadas: " + str(len(l2)) +
"\npermutaciones:\n" + str(l2), end="\n\n")
l3 = permute([1, 2, 3, 4, 5])
print("Numero de permutaciones generadas: " + str(len(l3)) +
"\npermutaciones:\n" + str(l3), end="\n\n")
```

El output es el siguiente:

```
drejor@luis-DESKTOPUBUNTU:~/Desktop/universidad/git/Algorit/Practica2$ python3 AEDATA_optional.py
Numero de permutaciones generadas: 6
permutaciones:
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

Numero de permutaciones generadas: 24
permutaciones:
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2,
 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1], [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2], [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]

Numero de permutaciones generadas: 120
permutaciones:
[[1, 2, 3, 4, 5], [1, 2, 3, 5, 4], [1, 2, 4, 3, 5], [1, 2, 4, 5, 3], [1, 2, 5, 3, 4], [1, 2, 5, 4, 3], [1, 3, 2, 4, 5], [1, 3, 2, 5, 4], [1, 3, 4, 2, 5], [1, 3, 4, 5, 2], [1, 3,
5, 2, 4], [1, 3, 5, 4, 2], [1, 4, 2, 3, 5], [1, 4, 2, 5, 3], [1, 4, 3, 2, 5], [1, 4, 3, 5, 2], [1, 4, 5, 2, 3], [1, 4, 5, 3, 2], [1, 5, 2, 3, 4], [1, 5, 2, 4, 3], [1, 5, 3, 2, 4]
, [1, 5, 3, 4, 2], [1, 5, 4, 2, 3], [1, 5, 4, 3, 2], [2, 1, 3, 4, 5], [2, 1, 3, 5, 4], [2, 1, 4, 3, 5], [2, 1, 4, 5, 3], [2, 1, 5, 3, 4], [2, 1, 5, 4, 3], [2, 3, 1, 4, 5], [2, 3,
 1, 5, 4], [2, 3, 4, 1, 5], [2, 3, 4, 5, 1], [2, 3, 5, 1, 4], [2, 3, 5, 4, 1], [2, 4, 1, 3, 5], [2, 4, 1, 5, 3], [2, 4, 3, 1, 5], [2, 4, 3, 5, 1], [2, 4, 5, 1, 3], [2, 4, 5, 3, 1
], [2, 5, 1, 3, 4], [2, 5, 1, 4, 3], [2, 5, 3, 1, 4], [2, 5, 3, 4, 1], [2, 5, 4, 1, 3], [2, 5, 4, 3, 1], [3, 1, 2, 4, 5], [3, 1, 2, 5, 4], [3, 1, 4, 2, 5], [3, 1, 4, 5, 2], [3, 1
, 5, 2, 4], [3, 1, 5, 4, 2], [3, 2, 1, 4, 5], [3, 2, 1, 5, 4], [3, 2, 4, 1, 5], [3, 2, 4, 5, 1], [3, 2, 5, 1, 4], [3, 2, 5, 4, 1], [3, 4, 1, 2, 5], [3, 4, 1, 5, 2], [3, 4, 2, 1,
5], [3, 4, 2, 5, 1], [3, 4, 5, 1, 2], [3, 4, 5, 2, 1], [3, 5, 1, 2, 4], [3, 5, 1, 4, 2], [3, 5, 2, 1, 4], [3, 5, 2, 4, 1], [3, 5, 4, 1, 2], [3, 5, 4, 2, 1], [4, 1, 2, 3, 5], [4,
1, 2, 5, 3], [4, 1, 3, 2, 5], [4, 1, 3, 5, 2], [4, 1, 5, 2, 3], [4, 1, 5, 3, 2], [4, 2, 1, 3, 5], [4, 2, 1, 5, 3], [4, 2, 3, 1, 5], [4, 2, 3, 5, 1], [4, 2, 5, 1, 3], [4, 2, 5, 3,
 1], [4, 3, 1, 2, 5], [4, 3, 1, 5, 2], [4, 3, 2, 1, 5], [4, 3, 2, 5, 1], [4, 3, 5, 1, 2], [4, 3, 5, 2, 1], [4, 5, 1, 2, 3], [4, 5, 1, 3, 2], [4, 5, 2, 1, 3], [4, 5, 2, 3, 1], [4,
 5, 3, 1, 2], [4, 5, 3, 2, 1], [5, 1, 2, 3, 4], [5, 1, 2, 4, 3], [5, 1, 3, 2, 4], [5, 1, 3, 4, 2], [5, 1, 4, 2, 3], [5, 1, 4, 3, 2], [5, 2, 1, 3, 4], [5, 2, 1, 4, 3], [5, 2, 3, 1
, 4], [5, 2, 3, 4, 1], [5, 2, 4, 1, 3], [5, 2, 4, 3, 1], [5, 3, 1, 2, 4], [5, 3, 1, 4, 2], [5, 3, 2, 1, 4], [5, 3, 2, 4, 1], [5, 3, 4, 1, 2], [5, 3, 4, 2, 1], [5, 4, 1, 2, 3], [5
, 4, 1, 3, 2], [5, 4, 2, 1, 3], [5, 4, 2, 3, 1], [5, 4, 3, 1, 2], [5, 4, 3, 2, 1]]
```

Podemos ver que da exactamente el número de permutaciones que debe dar (n!) y que se verifican correctas las permutaciones generadas por estos valores.