Algoritmos

Proyecto Final

Grupo 4

Johan Steeb Rodríguez Alarcón
Santiago Alexander Rodríguez Triana
Harol Achagua Clavijo
Andrés Felipe Moreno Pinzón

2021-II

PROCESO

Primero se descargaron las librerías necesarias entre las que se encuentran: numpy, pandas, networkx, etc. Posteriormente se descargaron y se bajaron los datos (en formato csv) de los tiempos y distancias entre las ciudades de Colombia.

```
#Se obtiene el archivo desde Drive
path = "/content/drive/MyDrive/CVS/Algorithm/co.csv"
second_path = "/content/drive/MyDrive/CVS/Algorithm/ti.csv"

#Dataframe de distancias
kd = pd.read_csv(path)
display(kd)

#Dataframe de tiempos
td = pd.read_csv(second_path)
display(td)

```

lómetros)	Bogotá	Cali	Medellín	Barranquilla	Cartagena	Cúcuta	Bucaramanga	Pereira	Santa Marta	Ibagué	Pasto	Manizales	Neiv
	NaN	463.0	419.0	1009.0	1052.0	599.0	409.0	322.0	968.0	193.0	764.0	317.0	30
	461.0	NaN	440.0	1144.0	1098.0	984.0	785.0	215.0	1231.0	274.0	393.0	268.0	38
	419.0	439.0	NaN	704.0	659.0	592.0	392.0	224.0	795.0	361.0	816.0	209.0	59
	1006.0	1139.0	702.0	NaN	122.0	669.0	642.0	924.0	102.0	991.0	1517.0	910.0	117
	1078.0	1094.0	656.0	124.0	NaN	740.0	713.0	879.0	225.0	1015.0	1471.0	864.0	124
	596.0	982.0	590.0	669.0	741.0	NaN	200.0	767.0	628.0	705.0	1354.0	718.0	89
	407.0	783.0	391.0	643.0	715.0	201.0	NaN	568.0	602.0	506.0	1155.0	519.0	69
	321.0	216.0	226.0	930.0	884.0	770.0	570.0	NaN	1017.0	134.0	593.0	54.0	33
	966.0	1229.0	793.0	101.0	223.0	629.0	602.0	1014.0	NaN	951.0	1600.0	964.0	113
	192.0	275.0	362.0	992.0	1035.0	705.0	505.0	133.0	951.0	NaN	669.0	190.0	20
	762.0	391.0	817.0	1520.0	1475.0	1353.0	1154.0	592.0	1600.0	670.0	NaN	644.0	46
	314.0	267.0	209.0	912.0	867.0	718.0	518.0	52.0	964.0	188.0	644.0	NaN	39
	300.0	381.0	565.0	1179.0	1221.0	891.0	691.0	336.0	1138.0	207.0	465.0	393.0	N
	108.0	559.0	527.0	1117.0	1160.0	707.0	517.0	418.0	1076.0	289.0	860.0	424.0	39
	274.0	188.0	282.0	985.0	940.0	789.0	590.0	53.0	1036.0	87.0	565.0	110.0	28
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
a ciudades:	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N
distancias-entre-ciudades/co/	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	N

1 to 18 o

onducción	Bogotá	Cali	Medellín	Barranquilla	Cartagena	Cúcuta	Bucaramanga	Pereira	Santa Marta	Ibagué	Pasto	Manizales	Neiv
	NaN	6:27	5:45	12:04	13:34	7:18	5:05	4:42	11:30	2:39	11:02	4:18	3:49
	6:27	NaN	5:43	14:23	13:54	12:25	10:10	2:39	15:07	3:54	7:06	3:20	6:25
	5:43	5:45	NaN	8:41	8:12	7:53	5:38	3:09	9:46	5:19	12:38	2:57	7:50
	12:02	14:15	8:32	NaN	1:53	8:51	7:55	11:40	1:15	11:53	21:09	11:28	14:0
	13:43	13:53	8:10	1:50	NaN	10:31	9:36	11:17	3:04	13:27	20:46	11:05	15:5

- Posteriormente se modifican ciertas partes y datos (referente al tipo de dato) de la columna para poder hacer posible las operaciones entre ellas.

```python
import pandas as pd
import numpy as np

pd.options.mode.chained_assignment = None

# Transformar formato de horas del dataframe a float para poderse tratar

def transform(n):
    try:
      if len(n) >= 5:
            return round(float((int(n[0]))*10 +(int(n[1])) + ((int(n[3])*10) + (int(n[4])))/60),3)
      else:
            return round(float(int(n[0]) + ((int(n[2])*10) + (int(n[3])))/60),3)
    except:
        return n

def replace(row):
    for i, item in enumerate(row):
        row[i] = transform(item)
    return row



td = td.apply(lambda row : replace(row))
print('After Applying Function: ')

# Dataframe transformado
display(td)

td = td.apply(lambda col:pd.to_numeric(col, errors='coerce'))


```

```python
#Elimina filas innecesarias
kd.drop([15,16,17], axis=0, inplace=True)
td.drop([15,16,17], axis=0, inplace=True)

#Elimina las columnas innecesarias
kd = kd.drop('Distancia (kilómetros)',1)
td = td.drop('Tiempo de conducción',1)

#Nuevos dataframes
display(kd)
display(td)


```

index	Bogotá	Cali	Medellín	Barranquilla	Cartagena	Cúcuta	Bucaram
0	NaN	6.45	5.75	12.067	13.567	7.3	
1	6.45	NaN	5.717	14.383	13.9	12.417	
2	5.717	5.75	NaN	8.683	8.2	7.883	
3	12.033	14.25	8.533	NaN	1.883	8.85	
4	13.717	13.883	8.167	1.833	NaN	10.517	
5	7.233	12.433	7.717	8.85	10.383	NaN	
6	5.017	10.2	5.5	7.933	9.483	2.3	
7	4.733	2.667	3.15	11.833	11.333	9.867	
8	11.467	15.117	9.617	1.233	3.1	8.267	
9	2.667	3.917	5.3	11.883	13.383	8.617	
10	11.033	7.083	12.617	21.283	20.8	18.1	
11	4.283	3.367	2.95	11.617	11.133	9.15	
12	3.883	6.417	7.883	14.2	15.7	10.933	
13	1.45	7.683	7.183	13.483	15.0	8.733	
14	4.167	2.4	3.817	12.5	12.0	10.167	

Show 25 per page

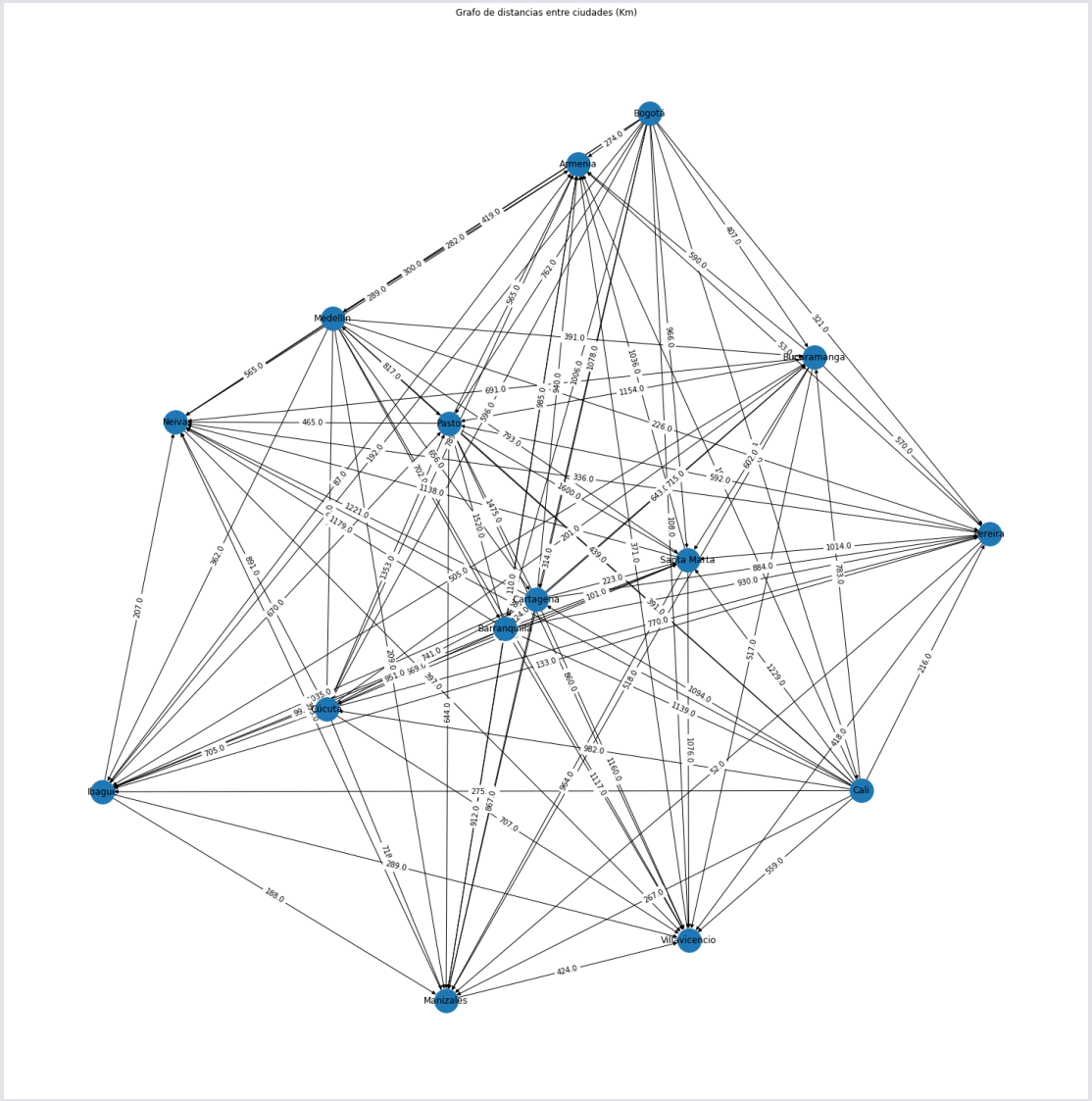
- Se tratan los datos del Dataframe para el grafo, y así de esta manera poder hacer posible su gráfica.

```python
ciudades = ["Bogotá", "Cali", "Medellín", "Barranquilla","Cartagena","Cúcuta", "Bucaramanga", "Pereira", "Santa Marta","Ibagué","Pasto",
                                    "Manizales","Neiva","Villavicencio","Armenia"]
#Convierte los dataframes en grafos de NetworkX
def def_graf(df):
    M = df.to_numpy()
    G = nx.from_numpy_matrix(M)
    dic = {}
    for i in range(15):
        dic[i]=ciudades[i]
    G = nx.relabel_nodes(G, dic) #Se reemplazan los números por las ciudades como nodos

    for j in ciudades:
        G.remove_edge(j,j) #Se eliminan las aristas hacia sí mismos
    return G

#Dibuja el grafo
def draw_graf(G):
    plt.figure(figsize = (20,20))
    pos = nx.fruchterman_reingold_layout(G) #Distribución de los nodos
    nx.draw(G,pos, arrows = True, node_size = 1000, with_labels=True)
    nx.draw_networkx_edge_labels(G,pos, nx.get_edge_attributes(G, 'weight'))

#Se definen los grafos
GD = def_graf(kd)
GT = def_graf(td)
#Pesos grafo distancias
wkd = nx.get_edge_attributes(GD, 'weight')
print(wkd)

#Pesos grafo tiempos
wtd = nx.get_edge_attributes(GT, 'weight')
print(wtd)
```

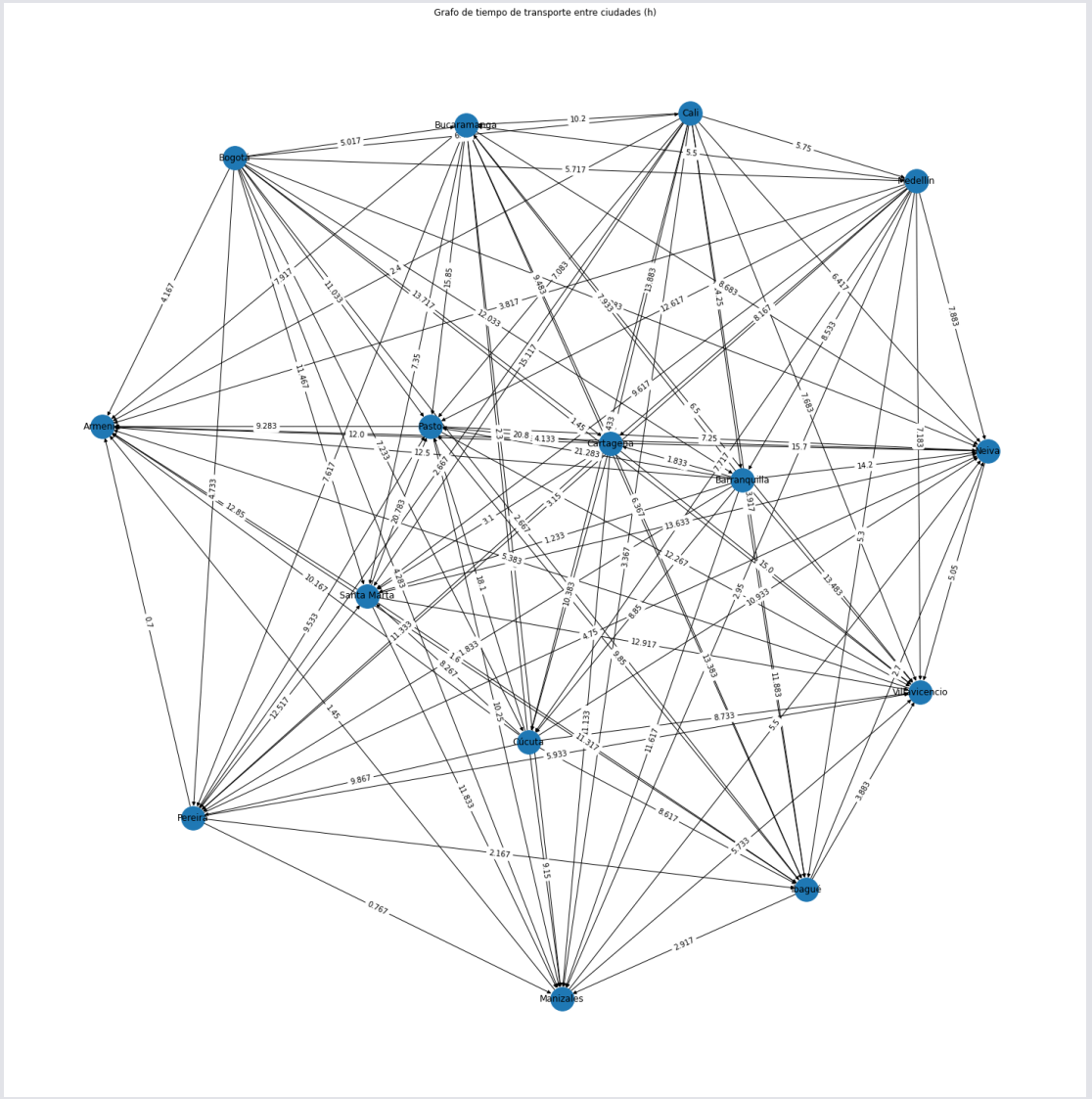
```python
#Se dibuja el de distancias
draw_graf(GD)
plt.title("Grafo de distancias entre ciudades (Km)")
plt.show()
```

```python
#Se dibuja el de tiempos
draw_graf(GT)
plt.title("Grafo de tiempo de transporte entre ciudades (h)")
plt.show()
```

- Gráfico de distancia



- Gráfico de tiempo



- Se usa una API para obtener las coordenada de las ciudades, después se define una función para graficar los nodos de tal manera que, con la ayuda de la librería basemap ubica los nodos de las ciudades en el mapa de Colombia correctamente

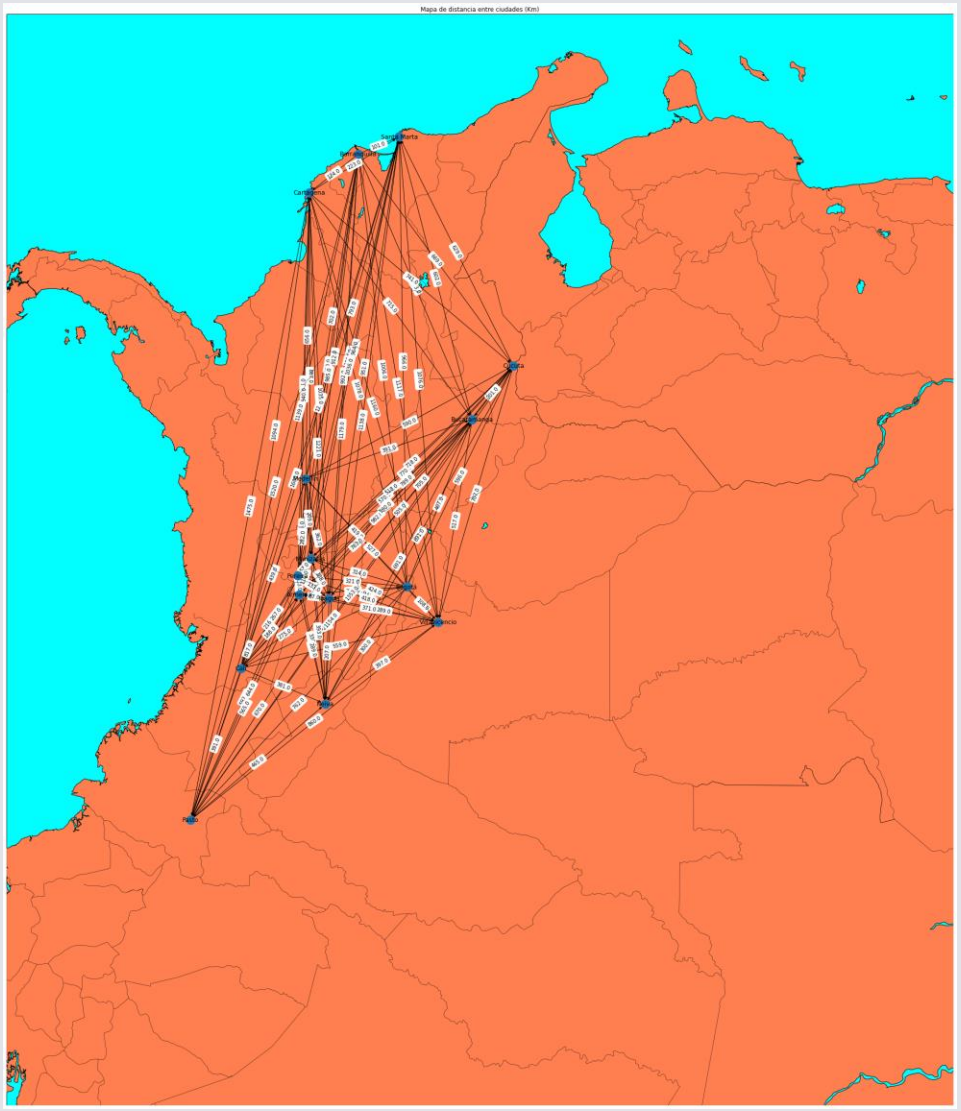
```python
def draw_graf_country(G):
    plt.figure(figsize = (40,40))
    m = Basemap(
            projection='merc', #modelo de mapa
            llcrnrlon=-80, #coordenadas de esquinas
            llcrnrlat=-3,
            urcrnrlon=-66,
            urcrnrlat=13,
            lat_ts=0,
            resolution='h', #Alta resolución
            suppress_ticks=True)
    pos = {}
    for i in ciudades:
        coord = get_coordinates(i) #se le asignan coordenadas a las ciudades
        x, y = m(coord[1], coord[0])
        pos[i] = (x, y)
    #Se dibuja el mapa y grafo
    m.drawcountries()
    m.drawstates()
    m.drawcoastlines()
    m.drawmapboundary(fill_color='aqua')
    m.fillcontinents(color='coral',lake_color='aqua')
    nx.draw(G,pos, arrows = True, node_size = 300, with_labels=True)
    nx.draw_networkx_edge_labels(G,pos, nx.get_edge_attributes(G, 'weight'))
```

```python
#Se usa API para obtener coordenadas de ciudades
def get_coordinates(city):
    geolocator = Nominatim(user_agent="MyApp")
    location = geolocator.geocode(city + ", Colombia")
    return(location.latitude, location.longitude)
```

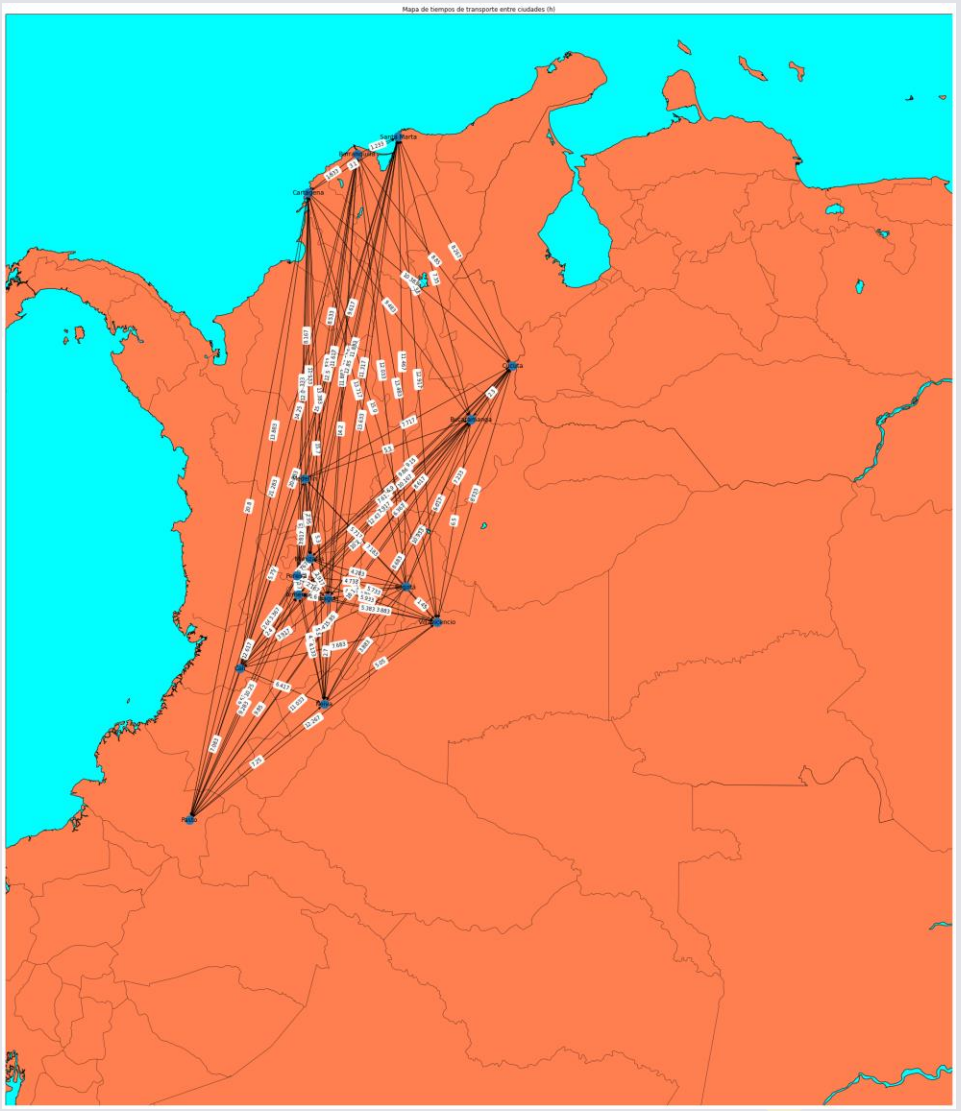
```python
#se dibuja el mapa de distancias
draw_graf_country(GD)
plt.title("Mapa de distancia entre ciudades (Km)")
plt.show()
```

```python
#se dibuja el mapa de tiempos
draw_graf_country(GT)
plt.title("Mapa de tiempos de transporte entre ciudades (h)")
plt.show()
```

- Mapa de distancia entre ciudades (Km)



- Mapa de tiempos de transporte entre ciudades (h)



```python
def dijsktra(Grafo, Nodos):

    grafo = nx.to_dict_of_lists(Grafo)              # Se retorna en una lista todos los nodos adyacentes a cada nodo
    #print(grafo)
    S = []; Queue = [];                             #Se crean la lista camino de los nodos "S" y una cola Queue vacías
    anterior = [0 for i in range(max(grafo)+1)];
    distancia = [0 for i in range(max(grafo)+1)]

    for nodo in grafo:                              #Iteración de los nodos en el grafo
        distancia[nodo] = 10000
        Queue.append(nodo)

    distancia[Nodos[0]] = 0                         #Se pone en distacias en cero

    while not len(Queue) == 0:
        distancia_minima = 10000
        for nodo in Queue:
            if distancia[nodo] < distancia_minima:
                distancia_minima = distancia[nodo]  #La distancia mínima será la distancia actual del nodo en la iteración
                nodo_temporal = nodo
        nodo_distancia_minima = nodo_temporal
        Queue.remove(nodo_distancia_minima)

        for vecino in grafo[nodo_distancia_minima]:
            if distancia[nodo_distancia_minima] == 10000:
                distancia_temporal = 0
            else:
                distancia_temporal = distancia[nodo_distancia_minima]  #Se valida la distancia minima para asignarla en distancia temporal
            distancia_con_peso = distancia_temporal + Grafo[nodo_distancia_minima][vecino]['weight']
            if distancia_con_peso < distancia[vecino]:
                distancia[vecino] = distancia_con_peso
                anterior[vecino] = nodo_distancia_minima

        if nodo_distancia_minima == Nodos[1]:
            if anterior[nodo_distancia_minima] != 0 or nodo_distancia_minima == Nodos[0]:
                while nodo_distancia_minima != 0:
                    S.insert(0, nodo_distancia_minima)
                    nodo_distancia_minima = anterior[nodo_distancia_minima]
                return S

print(f"Camino mas corto en distancia: {dijsktra(GD, (6, 10))}")
print(f"Camino más corto en tiempo: {dijsktra(GT, (6, 10))}")
```

Se implementa una forma el algoritmo Dijkstra para encontrar el camino más corto en términos de la distancia y tiempos de desplazamiento entre dos ciudades en específico, para este caso entre las ciudades de Bucaramanga y Pasto

```
1719.0
Camino mas corto en distancia: ['Bucaramanga', 'Pasto']
25.133
Camino más corto en tiempo: ['Bucaramanga', 'Pasto']
Tiempo de ejecución: 0.0017328262329101562
```

```
import time
first_time = time.time()
def dijsktra(Grafo, Nodos):

    grafo = nx.to_dict_of_lists(Grafo)                                  O(1)
    #print(grafo)
    S = []; Queue = [];
    anterior = [0 for i in range(max(grafo)+1)];                        O(1)
    distancia = [0 for i in range(max(grafo)+1)]                        O(1)

    for nodo in grafo:                                                  O(n)
        distancia[nodo] = 10000                                         O(1)
        Queue.append(nodo)                                              O(n)

    distancia[Nodos[0]] = 0                                             O(1)

    while not len(Queue) == 0:                                          O(n)
        distancia_minima = 10000                                        O(1)
        for nodo in Queue:                                              O(n)
            if distancia[nodo] < distancia_minima:                      O(1)
                distancia_minima = distancia[nodo]                      O(1)
                nodo_temporal = nodo                                    O(1)
        nodo_distancia_minima = nodo_temporal                           O(1)
        Queue.remove(nodo_distancia_minima)                             O(1)

        for vecino in grafo[nodo_distancia_minima]:                     O(n)
            if distancia[nodo_distancia_minima] == 10000:               O(1)
                distancia_temporal = 0                                  O(1)
            else:
                distancia_temporal = distancia[nodo_distancia_minima]                                   O(1)
            distancia_con_peso = distancia_temporal + Grafo[nodo_distancia_minima][vecino]['weight']    O(1)
            if distancia_con_peso < distancia[vecino]:                  O(1)
                distancia[vecino] = distancia_con_peso                  O(1)
                anterior[vecino] = nodo_distancia_minima                O(1)

        if nodo_distancia_minima == Nodos[1]:                           O(1)
            if anterior[nodo_distancia_minima] != 0 or nodo_distancia_minima == Nodos[0]:       O(1)
                while nodo_distancia_minima != 0:                                               O(n)
                    S.insert(0, nodo_distancia_minima)                                          O(1)
                    nodo_distancia_minima = anterior[nodo_distancia_minima]                     O(1)
                return S                                                                        O(1)

print(f"Camino mas corto en distancia: {dijsktra(GD, (6, 10))}")
print(f"Camino más corto en tiempo: {dijsktra(GT, (6, 10))}")

O(dijsktra) = O(n)
```

- Se realiza el cálculo de complejidad del algoritmo Dijkstra previamente desarrollado, para concluir que O(n) es la complejidad total del algoritmo

Usando las herramientas de la librería Networkx

```
first_time = time.time()

print(trans(list(nx.all_shortest_paths(GT, source=6, target=10))[0]))
print(trans(list(nx.all_shortest_paths(GD, source=6, target=10))[0]))

end_time = time.time()
print(f"Tiempo de ejecución: {end_time-first_time}")
```

```
['Bucaramanga', 'Pasto']
['Bucaramanga', 'Pasto']
Tiempo de ejecución: 0.0039038658142089844
```

```
first_time = time.time()

print(trans(list(nx.dijkstra_path(GD, source=6, target=10))))
print(trans(list(nx.dijkstra_path(GT, source=6, target=10))))

end_time = time.time()
print(f"Tiempo de ejecución: {end_time-first_time}")
```

```
['Bucaramanga', 'Pasto']
['Bucaramanga', 'Pasto']
Tiempo de ejecución: 0.005417346954345703
```