Tecnológico de Monterrey

Estructura de Datos
Proyecto final

Emilio Cantón - A01226707
Albert Hassey - A01024639
Isaac Harari - A01024688
Carlos García - A01025948
Yann Le Lorier - A01025977

ESTRUCTURA DE DATOS - PROYECTO FINAL

Índice

1. Introducción

El proyecto en cuestión es el número 3, en el cual tenemos la información de algunas estaciones de servicio de combustible, junto con su latitud y su longitud en grados. El objetivo del proyecto es de realizar una gráfica con base en los datos extraídos de internet, generando de ésta forma:

- La matriz de adyacencia
- El árbol abarcador mínimo
- Las rutas más cortas entre dos estaciones
- Los detalles de cada nodo (las estaciones) al buscarlos por el hash
- Exportación a un csv

2. La matriz de adyacencia

Para generar la matriz de adyacencia era necesario primeramente calcular las distancias relativas entre dos estaciones, pero nosotros ya conocíamos las coordenadas de las estaciones, por lo que sólamente se necesitaba un cálculo para establecer el peso entre dos nodos dados, y generar a partir de esto la matriz de adyacencia. a continuación un extracto de los cálculos realizados para obtener la distancia:

```
1                 latitud = float(datos[datos.__len__() - 3])
2                 # Hay que pasarlo de grados a radianes
3                 latitud_radianes = radians(latitud)
4                 longitud = float(datos[datos.__len__() - 2] )
5                 self.grafo.update({datos[0]: Node(datos[0], latitud *
    110574, longitud * (111320 * cos(latitud_radianes)))})
```

Ya que se tienen las distancias relativas, tomando en consideración que si se trata de una distancia de menos de 10,000 km, se trata de una conectividad completa, se puede generar un hashtable de adyacencia.

3. Árbol abarcador mínimo

Se utilizó el algoritmo de Prim para obtener el árbol abarcador mínimo, por lo que se programó una clase en Python que permite poblar la grafica en cuestión, y generar los nodos que van conectados entre sí.

```
1            with open('abarcador','w') as ab_tabla, open('abarcador.
   csv', 'w') as ab_csv:
2                csv = writer(ab_csv)
3                for values in self.abarcador:
4                    csv.writerow([values[0].group, values[0].id,
   values[1].id, values[2]])
5                    abarcador.append([values[0].group, values[0].id,
   values[1].id, values[2]])
6                ab_tabla.write(tabulate(abarcador, headers=['Cluster'
   ,'Node','Parent','Distance'],tablefmt='simple'))
```

Listing 1. Escritura del árbol abarcador mínimo en un csv

4. Las rutas más cortas

Para calcular las rutas más cortas entre dos nodos, fue necesario implementar un método que calculara la distancia entre dos estaciones, pero para ésto, era necesario que calculáramos la distancia en km entre dos estaciones, ya que conocíamos las coordenadas de cada estación.
Se implementó el algoritmo de Dijkstra, al llamar la función cuando se tomaba en cuenta las distancias más cortas entre dos estaciones, llamando, de ésta manera a las etiquetas que tiene cada uno de los nodos, y actualizándolos cuando ésto era necesario:

```
1        self.set_visited(False)
2        nodes = {x: (1000000, key_a) for x in list(self.grafo.keys())
   }
3        nodes[key_a] = (0, None)
4        stack = [key_a]
5        node_id = ''
6        while stack:
7            node_id = stack.pop(0)
8            if node_id == key_b:
9                break
10           node = self.grafo[node_id]
```

```
11              if node.visited == True:
12                  continue
13              node.visited = True
14              for connection, distance in node.connections.items():
15                  stack.append(connection)
16                  if distance + nodes[node_id][0] < nodes[connection
    ][0]:
17                      nodes[connection] = (distance + nodes[node_id
    ][0], node_id)
```

Listing 2. Extracto de nuestro algoritmo de Dijkstra

5. Detalles desplegados por el hash

Era necesario implementar un hash que permita el despliegue de toda la información necesaria de cada uno de los nodos, para que cada vez que se acceda a algún nodo, se impritma la información requerida por el usuario. Dentro del main, es posible obtener información del hash:

```
1 for key, value in g.grafo.items():
2   print(key, value.group, value.connections)
```

Listing 3. Instrucción para obtener información del registro

Lo cual genera una salida parecida a la siguiente:

```
1 E05553 1 {'E06926': 3067, 'E10209': 4377, 'E10017': 8837, 'E08425':
    6774, 'E10922': 5548, 'E11474': 106635}
2 E06926 1 {'E05553': 3067, 'E10209': 2321, 'E10017': 5921, 'E08425':
    3707, 'E10922': 2956}
3 E10209 1 {'E05553': 4377, 'E06926': 2321, 'E10017': 6524, 'E08425':
    3770, 'E10922': 1351}
4 E11474 2 {'E05553': 106635, 'E03450': 45452}
5 E03450 3 {'E11474': 45452, 'E06153': 22931}
6 E06153 4 {'E04641': 2801, 'E10839': 3849, 'E03450': 22931, 'E08596':
    72544}
```

Listing 4. Extracto del output obtenido

6. Parte del problema que se volvió intratable

No encontramos alguna parte del problema que lo volviera intratable, por lo menos en la implementación o en las pruebas que realizamos.
Por otro lado, encontramos el problema de agregar dos veces un mismo nodo al stack, ya que al llegar a este último, no se había actualizado todavía el booleano de “visitado”.

7. Conclusión

La implementación del algoritmo de Dijkstra en éste caso fue una oportunidad para aplicar lo que fue aprendido en clase de la manera más generalizada posible, se trata de una práctica enriquecedora porque permite visualizar lo que podría estar sucediendo en los algoritmos de aplicaciones de navegación satelital, y también aprendimos sobre la generación desde cero de un mapa hash.