Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Querétaro

Análisis y Diseño de Algoritmos Avanzados (Gpo 602)

Profesora: Ramona Fuentes Valdéz

Actividad Integradora 2: Reporte

Presentando:

Paulina Almada Martínez - A01710029

Miguel Ángel Barrón Sánchez - A01710304

Jesús Alejandro Cedillo Zertuche - A01705442

En esta actividad me comprometo conmigo y mi equipo a asumir un rol activo, honesto y responsable, basado en la confianza y la justicia y a no servirse de medios no autorizados o ilícitos para realizarla.

Noviembre 10, 2024. Santiago de Qro.

Parte 1

La primera parte de esta situación problema nos pide procesar un grafo ponderado no dirigido que representan una serie de colonias y las distancias entre las mismas para calcular la forma óptima de conectar cada colonia. En otras palabras, se nos pide encontrar la distancia más corta entre todos los pares de nodos de un grafo. Considerando esto, implementamos el algoritmo de **Floyd-Warshall**, ya que es un algoritmo que es eficiente para el problema y fácil de implementar en código.

El algoritmo de Floyd-Warshall, en esencia, utiliza programación dinámica para recorrer cada arista que conecta a los nodos y comparar con las conexiones previamente encontradas para identificar la más corta posible. Logra comparar todos las posibles combinaciones a través de realizar sus comparaciones con nodos intermedios, lo que permite considerar si existe un camino más corto que pasar de un nodo a otro directamente. A través de estas iteraciones, se crea una matriz de distancias que contiene las distancias más cortas para todos los pares de nodos, las cuales se actualizan cuando se encuentra una menor.

La complejidad del algoritmo es de $\mathbf{O}(n^3)$, donde n representa el número de nodos en el grafo, porque el algoritmo (y nuestra implementación) incorpora tres bucles anidados para definir los tres nodos que forman parte de la comparación. Estos bucles permiten probar combinaciones para cada nodo inicio, nodo final y nodo intermedio. En nuestra implementación en particular, creamos una función auxiliar llamada **floyd_parte_1()** para regresar los resultados en el formato especificado. Esta función tiene una complejidad de $\mathbf{O}(n^2)$ por lo mismo que la función principal, solo que en este caso, solo se tiene que considerar el nodo inicial y el nodo final. Esto significa que nuestra implementación final de esta primera parte tiene la complejidad mayor de nuestras funciones, por lo que tiene una complejidad total de $\mathbf{O}(n^3)$.

Los resultados de esta primera parte se ven así:

```
Punto 01:

Km de Colonia 1 a Colonia 2: 16
Km de Colonia 1 a Colonia 3: 34
Km de Colonia 1 a Colonia 4: 32

Km de Colonia 2 a Colonia 1: 16
Km de Colonia 2 a Colonia 3: 18
Km de Colonia 2 a Colonia 4: 21

Km de Colonia 3 a Colonia 1: 34
Km de Colonia 3 a Colonia 2: 18
Km de Colonia 3 a Colonia 4: 7

Km de Colonia 4 a Colonia 1: 32
Km de Colonia 4 a Colonia 2: 21
Km de Colonia 4 a Colonia 3: 7
```

```
Punto 01:

Km de Colonia 1 a Colonia 2: 2
Km de Colonia 1 a Colonia 3: 3
Km de Colonia 1 a Colonia 4: 7
Km de Colonia 1 a Colonia 5: 3
Km de Colonia 1 a Colonia 6: 5

Km de Colonia 2 a Colonia 1: 2
Km de Colonia 2 a Colonia 3: 3
Km de Colonia 2 a Colonia 4: 5
Km de Colonia 2 a Colonia 5: 1
Km de Colonia 2 a Colonia 6: 4

Km de Colonia 3 a Colonia 1: 1
Km de Colonia 3 a Colonia 2: 2
Km de Colonia 3 a Colonia 4: 4
Km de Colonia 3 a Colonia 5: 3
Km de Colonia 3 a Colonia 6: 5
```

```
Km de Colonia 4 a Colonia 1: 2
Km de Colonia 4 a Colonia 2: 2
Km de Colonia 4 a Colonia 3: 2
Km de Colonia 4 a Colonia 5: 2
Km de Colonia 4 a Colonia 6: 1

Km de Colonia 5 a Colonia 1: 1
Km de Colonia 5 a Colonia 2: 2
Km de Colonia 5 a Colonia 3: 3
Km de Colonia 5 a Colonia 4: 4
Km de Colonia 5 a Colonia 6: 5

Km de Colonia 6 a Colonia 1: 2
Km de Colonia 6 a Colonia 2: 3
Km de Colonia 6 a Colonia 3: 1
Km de Colonia 6 a Colonia 4: 3
Km de Colonia 6 a Colonia 5: 1
```

```
Punto 01:

Km de Colonia 1 a Colonia 2: 34
Km de Colonia 1 a Colonia 3: 53
Km de Colonia 1 a Colonia 4: 21
Km de Colonia 1 a Colonia 5: 55
Km de Colonia 1 a Colonia 6: 21

Km de Colonia 2 a Colonia 1: 42
Km de Colonia 2 a Colonia 3: 56
Km de Colonia 2 a Colonia 4: 24
Km de Colonia 2 a Colonia 5: 21
Km de Colonia 2 a Colonia 6: 1

Km de Colonia 3 a Colonia 1: 1
Km de Colonia 3 a Colonia 2: 23
Km de Colonia 3 a Colonia 4: 22
Km de Colonia 3 a Colonia 5: 21
Km de Colonia 3 a Colonia 6: 22

Km de Colonia 4 a Colonia 1: 32
Km de Colonia 4 a Colonia 2: 46
Km de Colonia 4 a Colonia 3: 32
Km de Colonia 4 a Colonia 5: 53
Km de Colonia 4 a Colonia 6: 21

Km de Colonia 5 a Colonia 1: 53
Km de Colonia 5 a Colonia 2: 34
Km de Colonia 5 a Colonia 3: 53
Km de Colonia 5 a Colonia 4: 21
Km de Colonia 5 a Colonia 6: 35

Km de Colonia 6 a Colonia 1: 55
Km de Colonia 6 a Colonia 2: 69
Km de Colonia 6 a Colonia 3: 55
Km de Colonia 6 a Colonia 4: 23
Km de Colonia 6 a Colonia 5: 45
```

1.1. Captura de pantalla del archivo txt de salida regresados por el programa.

Parte 2

La segunda parte de esta situación problema nos pide utilizar el grafo de la primera parte para identificar la manera más eficiente de visitar todos las colonias (representadas por los nodos) una vez y regresar a la colonia / el nodo inicial para terminar. En otras palabras, se nos pide resolver una iteración de TSP (Traveling Salesman Problem). Considerando esto, implementamos el algoritmo de **Prim**, ya que, aunque es una aproximación en vez de una solución exacta, es uno de los métodos más eficientes para resolver este problema NP.

El algoritmo de Prim, en esencia, crea un árbol de extensión mínima (MST) para identificar el camino desde origen a destino, sin repetir nodos, con el menor costo. Logra esto a través de asignar claves a los nodos, las cuales pueden ser comparadas para agregar los nodos en orden de menor a mayor y también aseguran que no se repitan aristas en el árbol. El crear el MST reduce significativamente la complejidad del problema, ya que primero se identifican las aristas con los costos mínimos. Después, se realiza un recorrido en preorden del árbol con DFS (búsqueda de profundidad), del cual se construye el camino final basado en el orden en que es visitado cada nodo y el costo asociado de cada vértice del árbol. Esta búsqueda nos permite visitar los nodos en un órden óptimo, lo que asegura que obtenemos un camino que regresa al nodo inicial. Aún así, justamente por el uso del árbol y el recorrido en preorden, el algoritmo de Prim no siempre regresa el camino más óptimo, aunque regresa una aproximación cercana. Este efecto se ve primordialmente en la suma del costo total, en vez del camino en sí, donde la lógica de los árboles MST nos apoyan a tener el camino con los menores costos posibles entre nodos.

La complejidad del algoritmo es de $\mathbf{O}(n^2)$, donde n representa cada nodo del grafo, ya que se debe involucrar crear el árbol MST y después recorrerlo con una búsqueda de profundidad. Esto involucra procesar cada nodo para comparar las claves y después recorrer el árbol para crear el camino y sumar el costo total. Por esto, se deben procesar los nodos del grafo dos

veces, aunque por esto mismo la complejidad depende ligeramente de la implementación de este algoritmo, donde la creación del MST y el procesamiento del mismo puede realizarse de distintas maneras. Aún así, por la lógica que se mantiene consistente sin importar los métodos utilizados, se puede generalizar su complejidad.

La complejidad de nuestra solución depende de las cuatro funciones que ocupamos para implementar este algoritmo. La función principal de **prim()** tiene una complejidad de **O(n^2)** porque se tienen dos bucles, primero para encontrar el vértice con la clave mínima y después para actualizar las claves dependientes del nodo actual. La función auxiliar de **min_clave()** tiene una complejidad de **O(*n*)** porque solo contiene un bucle, por lo que se recorre el árbol una vez en tiempo lineal. La función principal de **recorrido_preorden()** tiene una complejidad de **O(*n* * *m*)** porque recorre cada nodo y procesa cada arista relacionada una vez en tiempo lineal. En nuestra implementación en particular, también creamos la función auxiliar **prim_parte_2()** para regresar los resultados en el formato especificado. Esta función tiene una complejidad de **O(*n*)** por lo mismo que la función principal, solo que en este caso, solo se tiene que considerar el nodo inicial y el nodo final. Esto significa que nuestra implementación final de esta primera parte tiene la complejidad mayor de nuestras funciones, por lo que tiene una complejidad total de **O(n^2)**.

Los resultados de esta segunda parte se ven así. Es importante recordar que, en el caso de los costos, esta es una solución aproximada, por lo que las sumas no siempre son correctas:

```
Punto 02:

El recorrido: 1 -> 2 -> 3 -> 4 -> 1
El costo: 73
```

```
Punto 02:

El recorrido: 1 -> 2 -> 5 -> 4 -> 6 -> 3 -> 1
El costo: 12
```

```
Punto 02:

El recorrido: 1 -> 4 -> 3 -> 5 -> 2 -> 6 -> 1
El costo: 208
```

2.1. Captura de pantalla del archivo txt de salida regresados por el programa.

Parte 3

La tercera parte de esta situación problema nos pide procesar un nuevo grafo. Este es un grafo ponderado dirigido que representa la capacidad máxima de transmisión de datos entre distintas colonias para identificar el flujo máximo de esta red de colonias. En otras palabras, se nos pide encontrar el flujo máximo del grafo. Considerando esto, además del hecho de que

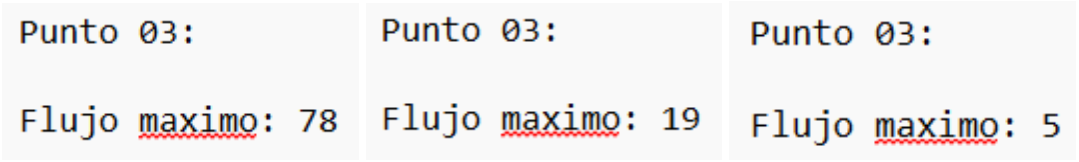
este problema puede tener una gran complejidad fácilmente con grafos más grandes o dispersos, para la resolución de esta tercera parte, decidimos implementar el algoritmo de **Edmonds-Karp**, ya que este algoritmo está diseñado para justamente obtener el flujo máximo de una manera eficiente, incluso con grafos retadores.

El algoritmo de Edmonds-Karp, en esencia, primero procesa el grafo a través de una búsqueda en anchura (**BFS**) para identificar distintos posibles caminos aumentantes, lo que significa que la capacidad máxima del nodo siguiente es mayor a la del nodo anterior. A través de este primer procesamiento, se obtienen los caminos viables con la menor cantidad de nodos, lo que hace más ágil el algoritmo. Después, se puede calcular el flujo máximo de cada camino, el cual está limitado por el nodo con la menor capacidad entre todos los nodos del camino. Mientras se analizan los distintos caminos, se actualizan las capacidades residuales (en otras palabras, las capacidades entre nodos limitadas por la capacidad mínima) entre cada par de nodos que conforman los distintos caminos. Se repite este proceso hasta que se hayan analizado todos los caminos identificados por el BFS. En tal momento, se puede realizar una suma de todo el flujo que pudo pasar por el camino con la mayor capacidad mínima. Tal suma representa el flujo máximo del grafo.

La complejidad del algoritmo es de $\mathbf{O(}n * m^2\mathbf{)}$, donde *n* representa el número de nodos y *m* representa el número de aristas. Esto, porque al determinar los caminos con BFS, cada nodo solamente puede participar O(n) veces, ya que constantemente se ajusta y reduce la capacidad del nodo lo que limita la cantidad de veces que será una posibilidad para el camino. Por asociación, las aristas solo pueden participar O(n) veces. Por ende, en el peor de los casos, se realiza una exploración de caminos de O(n * m). Ya que cada vez que se analiza un nodo, se analiza su arista correspondiente, la complejidad incrementa a $O(n * m^2)$.

Nuestra implementación en conjunto de estos algoritmos dentro de nuestro código tiene una complejidad total de $\mathbf{O(}n * m^2\mathbf{)}$, ya que es la mayor de las dos complejidades. La función **bfs()** tiene una complejidad de **O(*n* + *m*)**, donde *n* representa el número de nodos y *m* representa el número de aristas, porque gracias al uso de una queue, solamente se visitan los nodos del grafo O(n) veces. De manera parecida, ya que se visitan estos nodos en tiempo lineal, igualmente la complejidad de las aristas es de O(m), por lo que complejidad final es de O(n + m). La función **edmonds_karp()** tiene una complejidad de $\mathbf{O(}n * m^2\mathbf{)}$ porque, en el peor de los casos, se llama la función auxiliar de bfs() para cada camino aumentante, por lo que la complejidad inicial de la cantidad de veces que se llama la función es de O(n * m), combinada con la complejidad de bfs(), la cual es O(n + m), la cual se debe considerar para cada una de las llamadas, resulta en una complejidad final de $\mathbf{O(}n * m^2\mathbf{)}$.

Los resultados de esta tercera parte se ven así:



3.1. Captura de pantalla del archivo txt de salida regresados por el programa.

Parte 4

La cuarta y última parte de esta situación problema nos pide que teniendo en cuenta los grafos que teníamos anteriormente y sus distancias respectivas, podamos identificar cuál es la central más cercana a la nueva central que se va a instalar y asimismo poder calcular la distancia de esta de acuerdo a las coordenadas de una nueva central. Por lo que la salida será la distancia más corta indicando el valor de la central más cercana.

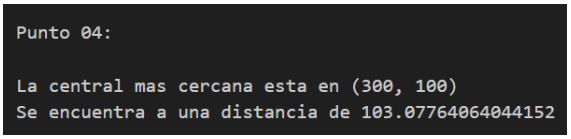
Para resolver esto implementamos un algoritmo usando la librería **KDTree** que es una estructura de datos bastante eficiente para el caso de buscar vecinos más cercanos, esto nos permite determinar qué central está más cerca de qué punto y en general la implementación de KDTree es útil para conjuntos de puntos sobre un plano y encontrar la distancia mínima entre ellos (de allí el nombre K-Dimensional Tree).

La complejidad de este algoritmo implementando KDTree es de **O(log *n*)**, donde *n* es el número de puntos de las centrales existentes, este algoritmo reduce significativamente el orden de complejidad usando fuerza bruta (O(n)) ya que es significativamente más rápida ya qué el KDTree se parte recursivamente a la mitad (similar a un AB) y en cada nivel del árbol selecciona una "dimensión" (alterna entre x o y) y esto decide cómo se dividen los puntos. Finalmente los puntos quedan organizados de tal forma que el árbol permite la búsqueda más eficiente. Una vez construido el KDTree solo es cuestión de descender en el árbol para poder encontrar el punto más cercano. Cabe mencionar que dependiendo de qué tan bien se encuentre distribuido el árbol, en este caso los puntos el nivel de complejidad podría subir en el peor de los casos a un O(n), aunque el promedio queda debajo de esa complejidad por lo que se mantiene que este algoritmo trabaja bajo el orden **O(log *n*)**.

Nuestra implementación del KDTree en nuestro código es la siguiente:

- tree = KDTree(ubicaciones_centrales): construye el árbol de acuerdo a la lista de las centrales existentes al momento.
- tree.query(nueva_central): realiza la búsqueda de la central más cercana y devuelve la distancia mínima y el índice de esa central.

Los resultados de esta parte se ven así:



```
Punto 04:

La central mas cercana esta en (690, 420)
Se encuentra a una distancia de 22.360679774997898
```

```
Punto 04:

La central mas cercana esta en (234, 576)
Se encuentra a una distancia de 354.1934499676695
```

4. Captura de pantalla del archivo txt de salida en el punto 04.

Es muy importante mencionar que para poder correr nuestra solución, primero debe correrse el comando **pip install scipy** para que la librería que importamos funcione.

Referencias

Das, A. (2023, marzo 23). *How to perform Edmonds-Karp on a graph*. DEV Community. https://dev.to/ananddas/how-to-perform-edmonds-karp-on-a-graph-i51

GeeksforGeeks. (2023, June 13). *Search and Insertion in K Dimensional tree*. GeeksforGeeks. https://www.geeksforgeeks.org/search-and-insertion-in-k-dimensional-tree/

GeeksforGeeks. (2024, octubre 26). *Prim's Algorithm for Minimum Spanning Tree (MST)*. GeeksforGeeks. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

KDTree. (s. f.). Scikit-learn. https://scikit-learn.org/1.5/modules/generated/sklearn.neighbors.KDTree.html

Ruedas, I. M. &. L. (2021, diciembre 29). Algoritmo de Floyd-Warshall. Análisis e implementación | by Ingrid Mendoza & Luisa Ruedas | Análisis de algoritmos. *Medium*. https://medium.com/algoritmo-floyd-warshall/algoritmo-de-floyd-warshall-e1fd1a900d8

W3Schools.com. (s.f.). https://www.w3schools.com/dsa/dsa_algo_graphs_edmondskarp.php