Tecnológico de Monterrey

Análisis y diseño de algoritmos avanzados

TC2038.602

Actividad Integradora 2

Diego Araque | A01026037
Fernando Valdeón | A01745186
Marco Torres | A01025334
Uriel Aguilar | A01781698

24 de noviembre 2022

Introducción

Los grafos son estructuras de datos esenciales para la representación de información en distintas áreas ya que nos permiten conocer las relaciones que existen entre los elementos de un sistema. Hay muchas formas de procesar los datos contenidos en un grafo dependiendo del resultado que esperas obtener, podríamos obtener la similitud de dos sistemas, o conocer el camino más cercano de un punto a otro, pero, para este problema tuvimos que buscar cómo conectar todos los nodos de un gráfico de la forma más ligera y eficiente posible.

La problemática empieza con una empresa de servicios de internet que desea llevar conexión cableada a cuatro colonias, pero, se busca que en esta conexión gaste la menor cantidad de cable posible. Para cumplir con este requisito debemos de obtener el MST (Minimum Spanning Tree) de un grafo que modela las relaciones entre las colonias, de esta forma conoceríamos la manera de conectar todas las colonias pasando el cable por los caminos más cortos posibles, y así ahorraríamos la máxima cantidad de cable.

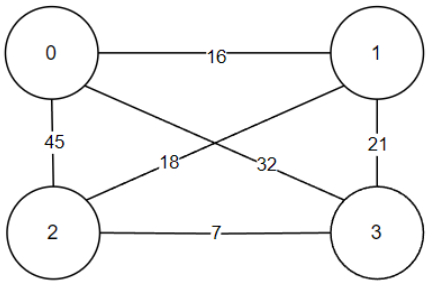
Implementación del Algoritmo de Prim

Para obtener el MST de un grafo se pueden usar varios algoritmos, nosotros decidimos usar el algoritmo de Prim.

Nuestra implementación funciona de la siguiente manera:

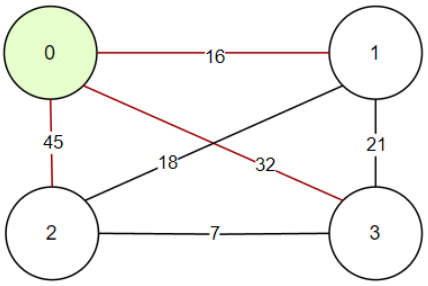
- El algoritmo empieza con un arreglo de tamaño igual al número de vértices (4) con todos los vértices marcados como no visitados, un mapa vacío donde se almacenarán las aristas disponibles y un arreglo vacío para los resultados.

```
visited_nodes = [0, 0, 0, 0]
results = []
available_edges = {}
```



- Se selecciona el vértice inicial y se guardan dentro de available_edges todas las aristas que no vayan hacia el mismo vértice (0) y que tengan un peso mayor a 0, estas aristas servirán para buscar la menor distancia posible que conecte con un vértice nuevo. En el mapa, la llave será una tupla con el vértice origen y destino, y el valor será el peso.

```
visited_nodes = [0, 0, 0, 0]

results = []

available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
}
```

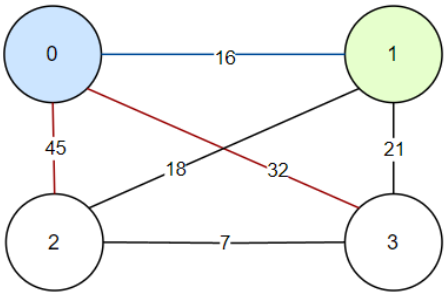


- Ahora, se debe de buscar la arista con menor peso, ya que está será la forma más corta de conectar con un nuevo vértice, para eso, iteramos nuestro mapa con las aristas disponibles, buscando la arista que contenga el menor peso pero que su vértice destino no haya sido visitado aún y que no vaya al mismo vértice que se está procesando. En este caso ningún vértice destino ha sido visitado y tampoco van al mismo vértice (0), entonces el peso menor es 16 y se guarda en resultados de la siguiente forma: [v_origen, v_destino, peso].

```
visited_nodes = [0, 0, 0, 0]

results = [0, 1, 16]

available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
}
```

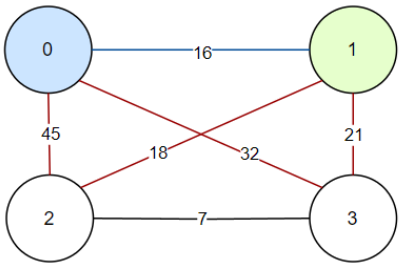
- Para finalizar marcamos el vértice actual como visitado, y nuestro siguiente vértice a analizar será el de la arista mínima encontrada, en este caso sería el vértice 1.

```
visited_nodes = [1, 0, 0, 0]

results = [0, 1, 16]

available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
}
```



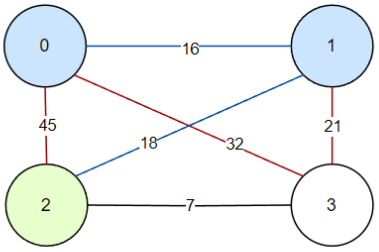
- Ahora repetimos el proceso, obtenemos las aristas del vértice actual y las agregamos a available_edges.

```
visited_nodes = [1, 0, 0, 0]

results = [0, 1, 16]

available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
      (1, 0): 16
      (1, 2): 18
      (1, 3): 21
}
```



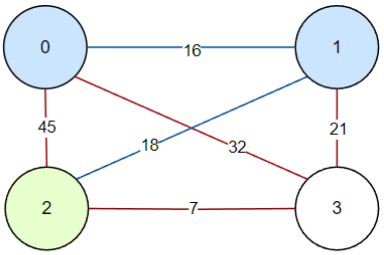
- Iteramos las aristas disponibles buscando el valor mínimo y con un vértice destino que no haya sido visitado aún. El valor mínimo es 16 y se tienen registrados dos sentidos de la arista: (0, 1) y (1, 0), pero, ningúno de los dos registros es válido porque en (0, 1) el vértice destino es el vértice actual (1) y no se debe considerar una arista que va al mismo vértice, y en (1, 0) el vértice destino ya ha sido visitado, por lo tanto se ignoran y el siguiente peso mínimo es 18.

```
visited_nodes = [1, 1, 0, 0]

results = [
      [0, 1, 16]
      [1, 2, 18]
]
available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
      (1, 0): 16
      (1, 2): 18
      (1, 3): 21
}
```



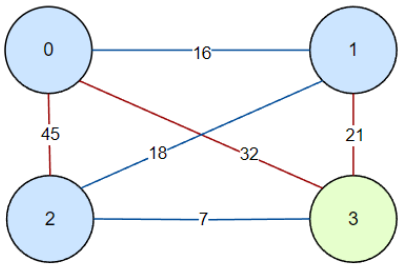
- Nuevamente debemos de guardar las aristas del vértice actual en available_edges, y buscamos la que tenga menor peso y su vértice destino no esté visitado, que es la arista (2, 3).

```
visited_nodes = [1, 1, 0, 0]

results = [
      [0, 1, 16]
      [1, 2, 18]
]

available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
      (1, 0): 16
      (1, 2): 18
      (1, 3): 21
      (2, 0): 45
      (2, 1): 18
      (2, 7): 7
}
```



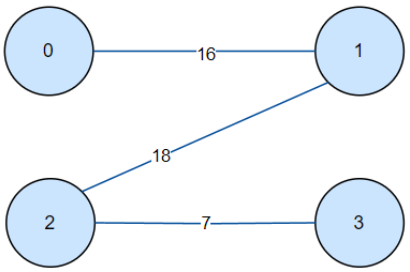
- Guardamos ese resultado, marcamos el vértice 2 como visitado y pasamos a procesar el siguiente y último vértice que es 3.

```
visited_nodes = [1, 1, 1, 0]
results = [
      [0, 1, 16]
      [1, 2, 18]
      [2, 3, 7]
]
available_edges = {
      (0, 1): 16
      (0, 2): 45
      (0, 3): 32
      (1, 0): 16
      (1, 2): 18
      (1, 3): 21
      (2, 0): 45
      (2, 1): 18
      (2, 7): 7
}
```



- Nuestro vértice actual (3) será el último. Se guardan las aristas del vértice y se busca cuál es la menor, pero, al ser el último vértice todas las aristas disponibles serán inválidas porque irán al mismo vértice (3) o a vértices que ya han sido visitados.

- Finalmente podemos ver que todos los vértices destino de las aristas disponibles están visitados, por lo que no habrá más resultados. Al marcar este vértice como visitado termina el algoritmo y obtenemos nuestro MST.



- Los resultados obtenidos son que la forma más eficiente de conectar todas las colonias es: conectar la colonia 0 con 1 (16km), conectar la colonia 1 con 2 (18km) y conectar la colonia 2 con 3 (7km).

```
results: [
      [0, 1, 16]
      [1, 2, 18]
      [2, 3, 7]
]
```

Complejidad del algoritmo

Temporal:

Nuestra implementación tiene complejidad temporal, en el peor de los casos, de O(n^2) donde n es la cantidad de vértices. El peor de los casos sería que cada vértice tenga aristas a todos los demás vértices ya que el algoritmo procesaría por cada vértice todos los demás vértices dando una complejidad cuadrática. Esta complejidad mejoraría si los vértices tienen menos aristas entre sí.

Espacial:

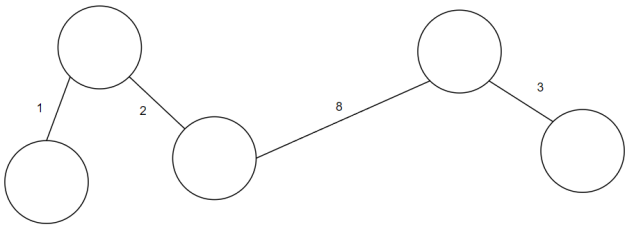
Nuestra implementación tiene complejidad espacial lineal O(n). Esto es debido a que en cada iteración de nuestro algoritmo se agregan aristas al mapa available_edges, y al finalizar todas las aristas estarán ahí.

Aplicaciones del algoritmo

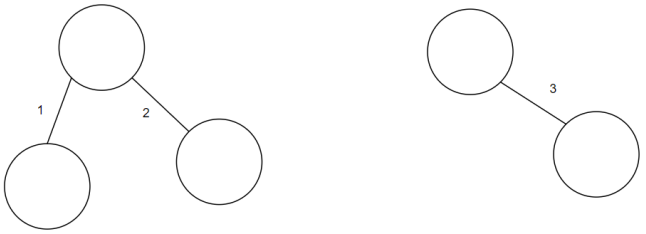
Uno de los usos más comunes que tiene el algoritmo de Prim es en el diseño de redes, como en esta situación problema. Cuando se tiene que optimizar una red, ya sea telefónica, eléctrica, hidráulica o de computadoras, se puede usar el algoritmo para encontrar la forma de conectar todos los nodos de la red eficientemente. Además de este uso, podemos encontrar otras aplicaciones para el algoritmo:

- Se puede usar en el área de transportes y vialidad cuando se quiere encontrar la ruta más corta que visite todos los destinos. Por ejemplo, empresas de paquetería que tienen que realizar varias entregas en diferentes casas usan el algoritmo para encontrar la ruta que conecte todas los destinos en la menor distancia posible.
- Se usa en análisis de datos para crear clusters de datos. Para poder clasificar grupos de datos similares entre sí se puede representar sus valores en un plano como puntos, posteriormente se crearía un grafo que conecte a todos los puntos entre sí y donde el peso de cada arista sea la distancia euclidiana entre los puntos. Al obtener el MST con el algoritmo de Prims de ese grafo obtendremos las distancias más cortas que relacionen a todos los nodos, y de ahí podemos obtener n clusters quitando las n-1 aristas con mayor peso.

  Por ejemplo el siguiente MST representa la relación más corta entre todos los datos en un plano 2D:



Si queremos obtener dos clusters debemos de quitar las n-1 aristas con mayor peso, en este caso debemos de quitar la arista con peso 8 y obtendríamos nuestras dos agrupaciones de datos.



- Se usa este algoritmo para crear el ruteo de una LAN. Cuando se realiza un broadcast en una LAN se puede generar un ciclo infinito de switches pasándose los paquetes entre sí, es por eso que se usa el Spanning Tree Protocol para encontrar una topología de la red que no contenga ciclos y se evite este problema.

Referencias

- Applications of minimum spanning trees. (s.f.). https://personal.utdallas.edu/~besp/teaching/mst-applications.pdf
- Canal Talkisoverrated. (28 sep 2020). Applications of spanning trees. [Archivo de Vídeo]. Youtube. https://www.youtube.com/watch?v=z6RB7WPO50M