ITC-ADA-C1-2023: Examen #1

Luis Ballado
luis.ballado@cinvestav.mx

CINVESTAV UNIDAD TAMAULIPAS — March 8, 2023

1 Realiza una aplicación en el lenguaje de tu preferencia que cumpla con los siguientes requerimientos, sea lo más eficiente posible y permita encontrar las respuestas a las interrogantes planteadas

Pregunta 1

Lee el archivo de entrada "bitacoraGrafos.txt" y almacena los datos en una lista de adyacencia organizada por la dirección IP de origen.

Ejecutar el programa con el archivo make

ver código en github

Command Line

```
$ make
$ ./problema_examen < bitacoraGrafos.txt
```

Pregunta 2

Determina el grado de salida de cada nodo del grafo y almacena en un archivo llamado "gradosIPs.txt" una lista con los pares (IP,grado de lista) en orden decreciente

ver archivo gradosIPs.txt

Pregunta 3

¿En qué dirección IP presumiblemente se encuentra el boot master?

```
boot masters

1   boot master: 73.89.221.25  Grados: 18
2   boot master: 185.109.34.183  Grados: 18
```

Pregunta 4

Si el camino más corto entre el boot master y cualquier otra IP del grafo representa el esfuerzo requerido para infectar dicha IP, ¿Cuál es la dirección IP que presumiblemente requiere más esfuerzo para que el boot master la ataque? Imprima en pantalla su respuesta.

```
boot masters

1     boot master: 73.89.221.25  Grados: 18
2     Camino mas largo:
3     IP - 244.223.133.50 -- 99
4
5
6     boot master: 185.109.34.183  Grados: 18
7     Camino mas largo:
8     IP - 244.223.133.50 -- 11368
9
```

Pregunta 5

Dado el grafo que construyó a partir de la bitácora, construya su MST. ¿Dentro del MST cuál es la arista más larga entre el boot master y sus IPs vecinas?

```
boot masters

1     boot master: 73.89.221.25  Grados: 18
2     Arista mas larga:
3     IP1 - 5.103.188.166 -##-  IP2 - 131.166.58.201 -- 2524
4
5     boot master: 185.109.34.183  Grados: 18
6     Arista mas larga:
7     IP1 - 5.103.188.166 -##-  IP2 - 154.1.79.55 -- 6646
8
```

2 Realiza una reflexión de la importancia y eficiencia del uso grafos en una situación problema de esta naturaleza, discute acerca de la complejidad computacional de cada uno de los métodos/funciones que implementaste, genera un documento PDF con tu reflexión.

El uso de grafos es muy amplio, así como lo he externado en clase. Dentro de mi formación siempre habia tocado el tema, pero no habia llegado a programar uno o intentado.

Me gusta ver aplicaciones más allá de sólo pensar en Redes Sociales. Así por ejemplo: conocer la ruta más corta para llegar de un punto A-B en una aplicación de transportes, donde se puede considerar un peso de arista como el tiempo que toma de viajar de un punto a otro, con el avance tecnológico o con mis conocimientos adquiridos, no es lejano pensar en una implementación de ciudad inteligente y tener representadas las calles y avenidas de la ciudad como un grafo y así controlar la circulación de automóviles y conocer los caminos más largos, vertices (colonias) no conectadas y un sin fin de posibilidades.

Fué extraño al principio armar un grafo apartir de un log de información posiblemente de un webserver y poder ejecutar un análisis de grafos. Pero, partiendo de que una IP representa un vertice y sus conexiones (aristas) con un esfuerzo para llegar a los diferentes vertices del grafo.

En mis anteriores trabajos, no le tomaba la importancia a la eficiencia y en conocer la complejidad del código. Así como lo cité en la primera tarea *El conocer nuestro código y dar respuestas a cuantas tareas máximas puede soportar, es lo mismo que le pregunten a un Ingeniero Civil acerca de las construcciones de cuanto puede soportar tal edificio respecto al peso que pueda soportar.*

Tomando en consideración el programa para la resolución del exámen práctico; el proceso que mayor tiempo toma es la carga y limpiado de información necesaria, donde al conocer el número de vertices e incidencias se itera para almacenar los vertices representados con las IPs en una estructura de tipo **unordered map** que no resultó eficiente, ya que llega a tener una complejidad lineal por su implementación con tablas hash, en su contraparte la estructura **map** cuya complejidad es O(log(n)) al contar con una estructura de tipo Red Black Tree.

Se describen los pasos que conforman la estructura del programa.

1. Tomar la información del archivo **bitacora** para conocer el número de IPs e incidencias Una vez conociendo el número de IPs, creamos la lista de adjacencia. Se itera respecto a ese número para ir creando los objetos de tipo IP guardando la relación del index y el string de la IP, almacenando el objeto en el **unordered map**
2. Al contar con la lista de adjacencia, y el unordered map, vuelvo a iterar, ahora con respecto al número de incidencias y es aquí donde obtengo los datos entre los vértices y los pesos
3. Al iterar respecto a las incidencias cree una función **get_data** con el uso de expresiones regulares que me regresa un vector con los valores importantes leidos: IP1, IP2, y el peso. Todo esto se hace analizando toda la linea del texto.
4. Una vez teniendo respuesta de la información valiosa cuya respuesta viene en un vector, itero nuevamente para obtener la IP1, IP2, peso. Aquí se hace el uso de la búsqueda O(1) del **unordered map** para obtener el index e ir guardando respecto a ese index en la clase grafo IP1, IP2, peso.
5. Itero nuevamente, siendo este proceso muy ineficiente ya que pude usar una iteración anterior para obtener el objeto y obtener los grados respecto a la lista de adjacencia.
   En esta iteración se crea otro vector con IP y grado, para despues aplicarle un ordenamiento **quicksort** para que ordene la lista de mayor a menor.
   Todo esto se pudo evitar haciendo uso correcto de la estructura de datos tipo **map**.
6. Crear el archivo **gradosIPs** iterando la lista previamente creada y guardandola en el archivo **gradosIPs.txt**

7. Generación de resultados
   Al tener dos posibles bootmasters itero nuevamente la lista de grados generada en el paso previo, dentro del ciclo se corren los algoritmos de **dijkstra** para obtener el camino más largo y también el algoritmo de **PRIM** para buscar el árbol recubridor minimo y conocer la arista más larga entre el boot master y sus IP vecinas

Conclusiones:

El programa después de tantas iteraciones no necesarias, hace que el programa tarde en su ejecución y comprobando la gran eficiencia que se puede llegar a obtener creando algoritmos eficientes, tomando en cuenta las complejidades en las operaciones y estructuras de datos que podamos utilizar.

Muchas veces dejamos de lado la eficiencia por **programar más rápido**, pero a la larga y mientras más usuarios ó más datos se tengan que trabajar, hasta ese momento nos preguntamos el ¿por qué es lento? y comenzamos a tomar atención a las complejidades de nuestros programas y tener ese control para no fragelarnos como programadores y hacer parches interminables que se llegan a convertir no mantenibles e ineficientes.