UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Estructura de Datos y Análisis de Algoritmos

Informe Laboratorio 3

Ariel Ignacio Tirado Maturana

Profesor:
Jacqueline Kohler Casasempere
Alejandro Cisterna Villalobos

Ayudante:
Gerardo Zuñiga Leal

Santiago - Chile
2-2016

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

La teoría de grafos es un campo de estudio de las matemáticas y las ciencias de la computación, y tiene un gran impacto a la hora de estudiar ciertos fenómenos en donde se ven envueltos muchos elementos que se conectan entre sí. Una de las aplicaciones más comunes de los grafos es la conexión entre un conjunto de ciudades en un país determinado, donde se dice que cada ciudad sería un *nodo* o un *vértice*, mientras que la distancia entre dos ciudades corresponde a una *arista*. De aquí se puede encontrar, por ejemplo, la distancia entre una ciudad y otra, saber si todas las ciudades están conectadas entre sí, etc.

Para el presente laboratorio del curso de Análisis de Algoritmos y Estructuras de Datos, se les ha solicitado a los alumnos que están cursando el ramo, la implementación de un algoritmo que se encargue de cumplir tres funciones; primero, el algoritmo debe verificar si un grafo es conexo o no. Posteriormente, el algoritmo debe ordenar los vértices del grafo según dos criterios; el primero será ordenar los vértices del grafo de prueba, según su centralidad de grado y luego ordenarlos según la centralidad *Betweenness*. Esto será programado en el lenguaje de programación C, el cual es uno de los mayores referentes de la programación imperativa.

Se espera que al finalizar la presente experiencia, se logre conseguir una mayor comprensión sobre el TDA Grafo, en proyección a buscar su utilidad en el área de la programación y ampliar la cantidad de herramientas disponibles al realizar algoritmos que se caractericen por trabajar con gran cantidad de datos.

A lo largo de la presente entrega, se le presentará al lector el problema actual y como fue diseñada la solución para este de una manera didáctica y precisa.

CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

2.1 ANÁLISIS DEL PROBLEMA

Para el presente laboratorio, se ha propuesto realizar un algoritmo que ejecute tres tareas dado un grafo de entrada, estas tareas son:

- Almacenar el grafo en memoria.
- Verificar si un grafo es conexo o no.
- Ordenar los vértices de un grafo, según la centralidad de grado.
- Ordenar los vértices de un grafo, según la centralidad *Betweenness.*

Para obtener el grafo a estudiar, este será ingresado en un archivo de entrada, el cual tendrá un nombre proporcionado por el usuario. Además se debe tener en consideración que el archivo contendrá en la primera línea la cantidad de vértices, y después viene una línea por cada vértice con todos sus adyacentes, separados por un espacio. Los vértices serán identificados por números enteros que están en el rango *1…n,* siendo *n* la cantidad total de vértices.

2.1.1 Almacenar el grafo en memoria

Debido a que el grafo a estudiar será ingresado en un archivo de entrada, se debe leer este y almacenar en una estructura adecuada para su posterior análisis. En este paso se deberá intentar recoger datos que serán utilizados posteriormente para estudiar las propiedades del grafo. Entre los datos que serán recogidos, se encuentran los vértices adyacentes de cada vértice y el grado de cada uno de estos.

2.1.2 Grafos conexos

Un grafo conexo, en su definición más simple, corresponde a un grafo en los cuales todos sus vértices se encuentran conectados de alguna manera al resto, ya sea directamente por

una arista o mediante la conexión sucesiva con otros vértices. A continuación se muestra una imagen donde se puede diferenciar un grafo conexo de un grafo no conexo.

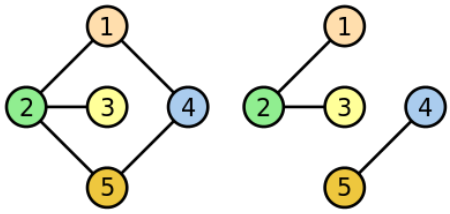

Imagen 2-1: Grafo conexo (izquierda) y un grafo no conexo (derecha).

Para verificar correctamente si un grafo es conexo o no, se debe recorrer el grafo, y ver si al recorrerlo se logró llegar a todos los vértices de este. Luego de realizar este procedimiento, se debe mostrar por pantalla un mensaje indicando si el grafo en estudio es conexo o no.

Realizando el desglose de lo dicho anteriormente, se obtienen los siguientes sub-problemas.

- Se debe realizar un recorrido por todo el grafo, intentando llegar a la mayor cantidad posible de vértices.
- Verificar si todos los vértices fueron visitados.
- En base al resultado del procedimiento, mostrar un mensaje por pantalla, indicando si el grafo es conexo o no.

2.1.3 Centralidad de grado

La centralidad de grado se define por el grado de cada vértice, donde los vértices que tienen un mayor grado, son más centrales. El grado corresponde a la cantidad de vértices adyacentes de un vértice en específico. Visto de una manera más simple, el grado corresponde a la cantidad de vértices a los cuales está conectado por una arista un vértice en específico. La siguiente imagen ejemplifica lo mencionado anteriormente.

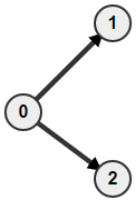

Imagen 2-2: Vértice 0 con dos vértices adyacentes (grado 2).

Dado que el grado de un vértice, corresponde a un número entero entre 0 y n, donde n es la cantidad de vértices del grafo, se debe encontrar la cantidad de vértices adyacentes de cada nodo, luego ordenarlos de mayor a menor, para posteriormente mostrar el resultado por pantalla.

Realizando el desglose de lo dicho anteriormente, se obtienen los siguientes sub-problemas.

- Leer el grafo completamente, y contar cuantos vértices adyacentes tiene cada vértice de este.
- Ordenar de mayor a menor los grados en conjunto a su vértice correspondiente.
- Mostrar el resultado por pantalla, indicando cuales son los vértices con mayor centralidad de grado hasta los menos centrales.

2.1.4 Centralidad *betweenness*

Así como se puede calcular la centralidad de grado de un vértice, también se puede calcular la centralidad betweenness de este. La centralidad betweenness corresponde a una medida que cuantifica la frecuencia o el número de veces que un vértice actúa como un puente a lo largo del camino más corto entre otros dos vértices. De esta definición se deduce también la definición de *camino mínimo*. El camino mínimo entre dos vértices corresponde al camino en que la suma de todos los pesos de las aristas entre dos vértices es el menor posible. Un ejemplo de esto, es encontrar el camino más corto entre dos ciudades.

De las definiciones dadas anteriormente, se expresa la centralidad betweenness en la siguiente fórmula.

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Donde σ_{st} es el número total de caminos mínimos desde que el nodo s al nodo t, y $\sigma_{st}(v)$ corresponde al número de esos caminos mínimos que pasan por el vértice v.

Realizando un desglose de lo mencionado anteriormente, se obtienen los siguientes sub-problemas.

- Obtener el camino mínimo entre todos los pares de vértices en el grafo.
- Comprobar cuántos de esos caminos pasan por el vértice v.
- Realizar la sumatoria de la división todos los caminos que pasan por v y los caminos mínimos entre el nodo s y el nodo t.
- Ordenar de mayor a menor los vértices, según su centralidad betweenness y mostrar el resultado por pantalla.

2.2 DESCRICIÓN DE LA SOLUCIÓN

En base a los sub-problemas obtenidos en la sección anterior, es que se planteó una solución para cada uno de ellos.

2.2.1 Almacenamiento del grafo en memoria

Para esto, se considera en la lectura del archivo tres factores; el primero es encontrar la cantidad de vértices que contiene el grafo, para esto se debe leer la primera línea del archivo, la cual corresponde a un número entero que indica lo que se está solicitando. El segundo paso se hace de manera paralela al tercer paso, y consiste en el almacenamiento de los vértices junto con sus adyacentes en una *lista de adyacencia* mientras se usa un contador, el cual aumenta en uno cada vez que se lee un número en el archivo. Este contador representará posteriormente al grado de cada vértice. El siguiente pseudocódigo muestra el procedimiento que representa lo mencionado anteriormente.

```
Función crearListaAdyacencia(archivo: Archivo donde se contiene el grafo):

        cont = 1
        leer(archivo, N)
        leer(archivo, espacio)
        listaAdyacente = crearNodo(listaAdyacente, N)

        Para i = 0 hasta N - 1, hacer:

                arregloAux = asignarMemoria()
                leer(archivo, vertice)
                arregloAux[0] = vertice
                leer(archivo, espacio)

                Mientras espacio = ' ' y espacio != Fin del archivo y espacio != '\n', hacer:
                        arregloAux = reasignarMemoria(cont + 1)
                        leer(archivo, vertice)
                        arregloAux[cont] = vertice
                        leer(archivo, espacio)
                        cont = cont + 1

                listaAdyacente -> grados[i][0] = cont - 1
                listaAdyacente -> grados[i][1] = i + 1
                listaAdyacente -> verticesAdyacentes[i] = arregloAux
                cont = 1

        Devolver listaAdyacente
```

Imagen 2-3: Pseudocódigo de la función crearListaAdyacencia.

Una cosa a tener en consideración, es que la variable *listaAdyacente* es una variable del tipo *grafo*, el cual corresponde a una estructura definida previamente, y que contiene en su interior los siguientes datos:

- Cantidad total de vértices en el grafo.
- Una matriz, que en el peor de los casos será de tamaño *NxN*, que corresponde a la lista de adyacencia del grafo.
- Una matriz de dimensiones *Nx2*, que contiene en la primera fila el grado de cada vértice y en la segunda fila el vértice asociado.

2.2.2 Grafos conexos

Para verificar si un grafo es conexo, se debe recorrer este, dado un vértice raíz, y ver si este se conecta a todos los demás vértices ya sea por una arista o por la conexión con otros vértices. Para esto, se eligió implementar el algoritmo de búsqueda en profundidad, el cual

realiza el procedimiento mencionado anteriormente, agregándole la particularidad de que al terminar el recorrido, se verifica si todos los vértices fueron visitados o no, transformando así la función a una de tipo booleano. A continuación se muestra la implementación del algoritmo que representa lo mencionado anteriormente.

```
Función busquedaProfunidad(actual: Grafo representado en una lista de adyacencia):

        marcados = asinarMemoria(actual -> cantidadNodos)

        Para i = 0 hasta actual cantidadNodos - 1, hacer:

                marcados[i] = asinarMemoria()
                marcados[i][0] = i + 1
                marcados[i][1] = 0

        raiz = actual -> verticesAdyacentes[0][0]
        vertices = crearPila()
        marcados = visitarVertice(vertices, marcados, actual, raiz)

        Para i = 0 hasta actual -> cantidadNodos - 1, hacer:
                Si marcados[i][1] = 0, hacer:
                        Devolver Falso

        Devolver Verdadero
```

Imagen 2-4: Pseudocódigo del algoritmo de búsqueda en profundidad.

2.2.3 Centralidad de grado

Como se mencionó anteriormente, al realizar la lectura del grafo en el archivo, se almacenó en el tipo de dato *grafo* una variable que contenía una matriz de *Nx2* con los grados de cada vértice. Para realizar el ordenamiento según la centralidad de cada vértice, se utiliza esta matriz, realizando el algoritmo de *BubbleSort* sobre esta matriz, tomando como referencia el grado de cada vértice. A continuación se muestra el pseudocódigo que representa lo mencionado anteriormente.

```
Función ordenarPorGrado(grados, cantidadNodos):

        Para i = 0 hasta cantidadNodos - 2, hacer:

                Para j = cantidadNodos - 1 hasta i, hacer:

                        Si grados[j][0] > grados[j - 1][0], hacer:

                                gradoAux = grados[j][0]
                                verticeAux = grados[j][1]
                                grados[j][0] = grados[j - 1][0]
                                grados[j][1] = grados[j - 1][1]
                                grados[j - 1][0] = gradoAux
                                grados[j - 1][1] = verticeAux

        Devolver grados
```

Imagen 2-5: Pseudocódigo del algoritmo de BubbleSort.

Como resultado, se obtendrá la matriz de *Nx2* ordenada desde el vértice más central hasta el menos central.

2.2.4 Centralidad *betweenness*

Como se mencionó anteriormente, para calcular la centralidad betweenness de un vértice en un grafo es necesario conocer la cantidad de caminos mínimos entre todos los pares de vértices del grafo y saber cuántos de ellos pasan por el vértice en cuestión. Para esto se propone en un principio implementar el algoritmo de *Dijkstra*, el cual estará implementado de tal forma, que el retorno de la función tenga la cantidad de caminos mínimos encontrados desde el nodo raíz hacia los demás nodos, y un arreglo donde cada posición *i* contendrá la cantidad de veces que pasó un camino mínimo por un vértice *i+1*. La función de Dijkstra se ejecutará una cantidad de veces igual a la cantidad de vértices del grafo, realizando así la comparación entre todos los vértices de este. Luego de realizar este procedimiento, se calculará la centralidad betweenness para cada vértice y almacenarlos en una matriz de dimensiones *Nx2*, la cual contendrá en la primera fila los vértices enumerados y la segunda, la centralidad betweenness correspondiente. Finalmente, se ordenará de mayor a menor, según la centralidad obtenida.

CAPÍTULO 3. ANÁLISIS DE RESULTADOS

Luego de realizar completamente el algoritmo, se procede a medir la eficiencia de este, mediante los conceptos adquiridos en clases. A continuación se muestra el cálculo de la función que representa el tiempo de ejecución del algoritmo.

- *crearListaAdyacencia( ):*

$$T_{crearListaAdyacencia}(n) = O(1) + n\left(O(1) + n\big(O(1)\big)\right) + O(1)$$

$$T_{crearListaAdyacencia}(n) = O(1) * n^2 + O(1)$$

- *ordenarPorGrado( ):*

$$T_{ordenarPorGrado}(n) = 1 + 1 + 1 + 1 + n\left(n\big(O(1)\big)\right) + 1$$

$$T_{ordenarPorGrado}(n) = O(1) * n^2 + O(1)$$

- *mostrar( ):*

$$T_{mostrar}(n) = O(n) + 1 + n(1 + 1 + 1) + 1$$

$$T_{mostrar}(n) = O(n) + 2$$

Además, se tiene en consideración que el *T(n)* del procedimiento de búsqueda en profundidad es aproximadamente *O(n).* Finalmente, el tiempo de ejecución del programa es el siguiente.

$$T(n) = T_{crearListaAdyacencia}(n) + T_{ordenarPorGrado}(n) + T_{mostrar}(n) + O(n)$$

$$T(n) = O(1) * n^2 + O(1) + O(1) * n^2 + O(1) + O(n) + 2$$

$$T(n) = 2n^2 + O(n) + O(1) \sim O(n^2)$$

CAPÍTULO 4. CONCLUSIONES

Para concluir con el presente informe, se deben tomar en cuenta dos parámetros. El primero de ellos es la eficacia del algoritmo, o dicho de otra forma, que el algoritmo cumpla con su objetivo. En base a este parámetro, se podría considerar medianamente exitoso, ya que si bien logra verificar que un grafo sea conexo o no y ordenar según la centralidad de grado, no logra cumplir con el objetivo de ordenar según la centralidad betweenness, ya que por falta de conocimiento sobre la teoría de los caminos mínimos, no se supo adaptar un algoritmo que encontrara todos los caminos mínimos entre todos los pares de vértices. El otro parámetro a considerar es la eficiencia del algoritmo, la que se puede cuantificar mediante la función que expresa el tiempo de ejecución, la cual resultó ser de orden $O(n^2)$. Esto puede ser considerado eficiente en entradas con poca cantidad de datos, pero en presencia de una gran cantidad de estos, el tiempo de ejecución aumenta significativamente. Esto puede mostrar un problema en la vida real, ya que si se pretende utilizar el algoritmo con el fin de estudiar diversos grafos, como la interconexión entre ciudades, redes eléctricas, etc. se retrasaría mucho el proceso, debido a que estas aplicaciones de grafos contienen generalmente una gran cantidad de datos.

Se espera para futuras entregas poder seguir buscando nuevos métodos más eficientes que el actual, y poder utilizar los conocimientos adquiridos en la presente entrega a futuro de manera ágil y eficaz.

CAPÍTULO 5. REFERENCIAS

- Enunciado Laboratorio 1 Análisis de Algoritmos y Estructuras de Datos (Septiembre de 2016). UdeSantiagoVirtual. Obtenido de UdeSantiago Virtual Moodle: *http://www.udesantiagovirtual.cl/*
- StackOverflow (Agosto de 2008), creado por Jeff Attwood. Utilizado para dudas varias. *http://stackoverflow.com/*