



**Tecnológico
de Monterrey**

**Programación de Estructuras de Datos y
Algoritmos Fundamentales**

***Act 4.1 - Actividad Integral de Grafos
(Evidencia Competencia)***

Alumno:

Carlos Antonio Buendia López	A01379471
Luis Ángel Terrazas García	A01377440

Profesor(a):

Víctor Adrián Sosa Hernández

04/12/2020

Investigación:

Un grafo consiste en un conjunto de nodos y arcos los cuales establecen una relación entre los nodos. Estos pueden tener diferentes características como puede ser su peso, dirección, su cantidad de aristas u conexiones entre ellos. Los grafos pueden tener una estructura dirigida y no dirigida.

Las conexiones entre los nodos de los grafos pueden tener un peso el cual en la mayoría de los casos se utiliza como medio de referencia para representar el camino que existe desde un nodo hacia otro. Existen algoritmos para buscar el camino con mínimo peso en un grafo.

Los grafos en programación viene de la teoría de los grafos los cuales se fundamentan en las matemáticas discretas. En las ciencias computacionales actuales hay una enorme cantidad de fenómenos que se pueden modelar como grafos y aplicar mecanismos de optimización sobre estos. Las ventajas por lo que se utilizan los grafos son bastantes, con grafos se pueden determinar en un tiempo fijo y constante si un arco pertenece o no al grafo, también es fácil determinar si existe un ciclo en el grafo. Las desventajas de esta estructura es su complejidad debido a que solo el examinar y leer el grafo lleva un tiempo de $BigO(n^2)$.

Lectura y creación de grafo/Impresión del grafo con sus conexiones:

```
21
22 //Si la ip de origen o de destino no existe en el grafo lo agrega
23 if(!ejemploGrafo1->buscarNodo(iporigen)){
24     ejemploGrafo1->insertarNodoGrafo(iporigen);
25 }
26 if(!ejemploGrafo1->buscarNodo(ipdestino)){
27     ejemploGrafo1->insertarNodoGrafo(ipdestino);
28 }
29
30 //Busca la lista de conexiones de iporigen
31 Lista<Conexion *> * conexiones = ejemploGrafo1->buscarNodo(iporigen)->getConexiones();
32 Nodo<Conexion *> * auxiliar= conexiones->getHead();
33 while(auxiliar){
34     if(auxiliar->getValor()->getNombreNodo() == ipdestino){
35         agregar = false; //Si la ipdestino esta en la lista de conexiones del origen no lo agrega.
36     }
37     auxiliar = auxiliar->getSiguiente();
38 }
39 if(agregar){
40     ejemploGrafo1->agregarArco(iporigen,ipdestino,1); //agrega arco al grafo
41 }
42
43 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

2: bash

```
elbar@DESKTOP-E1K5286:~/projects/UltimaActividad$ ./main
Nodo:164.215.215.39 -> 225.76.242.46:1 -> 81.142.27.233:1 -> 232.18.244.210:1 -> 207.121.154.113:1 -> 164.215.215.39:1 -> 121.118.80.161:1 -> 237.142.59.88:1
Nodo:121.118.80.161 -> 81.142.27.233:1 -> 225.76.242.46:1 -> 232.18.244.210:1 -> 237.142.59.88:1 -> 207.121.154.113:1 -> 164.215.215.39:1
Nodo:237.142.59.88 -> 164.215.215.39:1 -> 207.121.154.113:1 -> 232.18.244.210:1 -> 237.142.59.88:1 -> 225.76.242.46:1 -> 121.118.80.161:1 -> 81.142.27.233:1
Nodo:207.121.154.113 -> 225.76.242.46:1 -> 237.142.59.88:1 -> 232.18.244.210:1 -> 207.121.154.113:1 -> 164.215.215.39:1 -> 121.118.80.161:1 -> 81.142.27.233:1
Nodo:81.142.27.233 -> 232.18.244.210:1 -> 164.215.215.39:1 -> 81.142.27.233:1 -> 121.118.80.161:1 -> 225.76.242.46:1 -> 237.142.59.88:1 -> 207.121.154.113:1
Nodo:225.76.242.46 -> 225.76.242.46:1 -> 207.121.154.113:1 -> 81.142.27.233:1 -> 164.215.215.39:1 -> 237.142.59.88:1
Nodo:232.18.244.210 -> 164.215.215.39:1 -> 207.121.154.113:1 -> 121.118.80.161:1 -> 237.142.59.88:1 -> 225.76.242.46:1
```

Complejidad lectura y creación de grafo:

Debido a que para agregar las ips al grafo se tenía que verificar que esta no fuera una conexión repetida se tiene que buscar en el peor de los casos todos los nodos del grafo y todos los elementos de la lista de conexiones de grafo. Por estas razones existe un doble ciclo en esta generación de grafo haciendo que tenga una complejidad de $\text{BigO}(n^2)$.

Realizar un método el cual responda a la pregunta: ¿Cuál es la dirección IP que recibe más fallas? (La IP destino a la que se conectan más veces, pueden ser varias IP)

Complejidad de esta función:

En esta función está conformada por diferentes ciclos while los cuales en el peor de los casos nos darían un $\text{BigO}(n)$, a pesar de que esta función tiene diferentes ciclos while el conjunto de estos ciclos no llegaría a más de $\text{BigO}(n)$, no existe ningún ciclo dentro de otro ciclo en esta función haciendo que tenga una complejidad de $\text{BigO}(n)$.

```
grafoLista.hpp > Grafo > cantFallasRecibidas()
296 //Regresa todos los elementos con la mayor cantidad de fallas recibidas.
297 void cantFallasRecibidas(){
298     int cantidad = 0;
299     Nodo<NodoGrafo*> *elemento=this->nodos->getHead();
300     vector<string> conMayorCant;//Almacenara todos los elementos con el mayor tamaño de fallas recibidas.
301
302     //Busca en cada lista de conexiones si el nombre del valor es igual al nombre del nodo grafo.
303     while(elemento){
304         Nodo<Conexion*> * conexionesTemp = elemento->getValor()->getConexiones()->getHead();
305         while(conexionesTemp){
306             if(conexionesTemp->getValor()->getNombreNodo() == elemento->getValor()->getNombreNodo()){//Si los nombres son iguales le suma en uno la
307                 elemento->getValor()->aumentarFallas();
308             }
309             conexionesTemp= conexionesTemp->getSiguiente();
310         }
311         elemento=elemento->getSiguiente();
312     }
313     elemento=this->nodos->getHead();
314
315     //Revisa los nodos del grafo para buscar los que tienen una mayor cantidad de fallas recibidas.
316     while(elemento){
317         if(elemento->getValor()->getFallasRecibidas()>cantidad){
318             cantidad = elemento->getValor()->getFallasRecibidas();//Actualiza el numero mayor de fallas recibidas encontradas en cada nodo grafo.
319         }
320         elemento=elemento->getSiguiente();
321     }
322     elemento=this->nodos->getHead();
323
324     //Agrega todo elemento que tenga la misma cantidad de fallas al numero mayor de fallas en un vector.
325     while(elemento){
326         if(elemento->getValor()->getFallasRecibidas() == cantidad){
327             conMayorCant.push_back(elemento->getValor()->getNombreNodo());
328         }
329         elemento=elemento->getSiguiente();
330     }
331     return conMayorCant;
332 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

3: bash

Ips que reciben mas fallas:
164.215.215.39/ 237.142.59.88/ 207.121.154.113/ 81.142.27.233/ 225.76.242.46/

Realizar un método el cual responda a la pregunta: ¿Cuál es la dirección IP que genera más fallas? (La IP origen que tiene más arcos, pueden ser varias IP)

Complejidad de función:

Esta función es igual que la función anterior tiene varios ciclos pero no tiene ningún ciclo dentro de otro ciclo, esta función tiene una complejidad de $O(n)$.

```
269 //Regresa todos los elementos con la mayor cantidad de fallas generadas.
270 void cantFallasGeneradas(){
271     int cantidad = 0;
272     vector<string> conMayorCant; //Almacenara todos los elementos con el mayor tamano de conexiones.
273     Nodo<NodoGrafo *> *elemento=this->nodos->getHead();
274     //Revisa el tamano de la lista conexiones de cada elemento y guarda el valor mas grande.
275     while(elemento){
276         if(elemento->getValor()->getConexiones()->getTam() > cantidad){
277             cantidad = elemento->getValor()->getConexiones()->getTam();
278         }
279         elemento=elemento->getSiguiente();
280     }
281     elemento=this->nodos->getHead();
282
283     //Compara si el tamano de la lista conexiones es igual a la mayor cantidad de conexiones encontradas en un nodo del grafo.
284     while(elemento){
285         if(elemento->getValor()->getConexiones()->getTam() == cantidad){
286             conMayorCant.push_back(elemento->getValor()->getNombreNodo());
287         }
288         elemento=elemento->getSiguiente();
289     }
290     //Imprime el vector de elementos para mostrar los elementos con mayor cantidad de fallas generadas.
291     cout << "Ips con mayor fallas generadas: " << endl;
292     for(int i=0; i < conMayorCant.size(); i++)
293         cout << conMayorCant.at(i) << '/';
294 }
295
296 //Regresa todos los elementos con la mayor cantidad de fallas recibidas.
297 void cantFallasRecibidas(){
298     int cantidad = 0;
299     Nodo<NodoGrafo *> *elemento=this->nodos->getHead();
300     vector<string> conMayorCant; //Almacenara todos los elementos con el mayor tamano de fallas recibidas.
301 }
302
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 3: bash

Ips con mayor fallas generadas:
164.215.215.39/237.142.59.88/207.121.154.113/81.142.27.233/

En orden Descendente

Realizar un método que implemente el recorrido en profundidad (Depth First) y mostrar el resultado en pantalla.

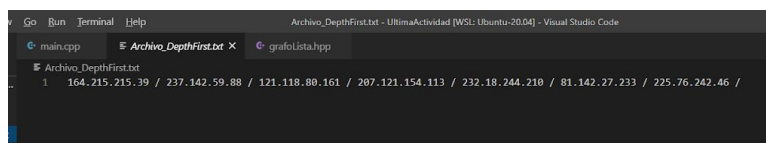
La complejidad del algoritmo de ordenamiento Depth First tiene una complejidad en el peor de los casos de $O(n^2)$.

```
48 //Imprime la o las ips con mayor cantidad de fallas recibidas.
49 ejemploGrafo1->cantFallasRecibidas();
50 cout<< endl;
51
52 //Imprime la o las ips con mayor cantidad de fallas generadas.
53 ejemploGrafo1->cantFallasGeneradas();
54 cout<< endl;
55 cout<< endl;
56
57 //Imprime ips en orden DepthFirst
58 cout<< "Ips en orden DepthFirst:"<< endl;
59 cout<< ejemploGrafo1->DepthFirst(0);
60 cout<< endl;
61
62 //Almacena en un archivo el resultado del recorrido DepthFirst
63 ofstream archivoDepthFirst("Archivo_DepthFirst.txt");
64 |   archivoDepthFirst << ejemploGrafo1->DepthFirst(0);
65 |   archivoDepthFirst.close();
66 |   return 0;
67 |
68 |
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Ips en orden DepthFirst:
164.215.215.39 / 237.142.59.88 / 121.118.80.161 / 207.121.154.113 / 232.18.244.210 / 81.142.27.233 / 225.76.242.46 /

Almacenar en un archivo el resultado del recorrido anterior.



```
Go Run Terminal Help
Archivo_DepthFirst.txt - UltimaActividad [WSL: Ubuntu-20.04] - Visual Studio Code
main.cpp Archivo_DepthFirst.txt grafosLista.hpp
Archivo_DepthFirst.txt
1 164.215.215.39 / 237.142.59.88 / 121.118.80.161 / 207.121.154.113 / 232.18.244.210 / 81.142.27.233 / 225.76.242.46 /
```

Referencias:

Morales Martínez, J. O., & Gómez Rizo, E. A. (2016). *Apareamientos en grafos y su implementación en C++, casos general y bipartito* (Doctoral dissertation).

Montilva, J. A., & Granados, G. (1996). Modelado y Manipulación de Redes de Servicio usando Grafos Espaciales en C++. In *Actas de la Primera Conferencia Internacional sobre Métodos Numéricos en Ingeniería. Mérida, Venezuela.*

Montilva, J. A., & Granados, G. (1996). Modelado y Manipulación de Redes de Servicio usando Grafos Espaciales en C++. In *Actas de la Primera Conferencia Internacional sobre Métodos Numéricos en Ingeniería. Mérida, Venezuela.*

Hinojosa Ramos, E. A. (2014). *Velocidad de respuesta de los algoritmos de búsqueda de datos contenidos en estructuras estáticas y dinámicas. Se encuentra en el sitio web:*
<http://www.revistas.unjbg.edu.pe/index.php/cyd/article/view/733/745>