Actividad 4.3 - Actividad Integral de Grafos (Evidencia Competencia)

Programación de estructuras de datos y algoritmos fundamentales (Gpo 4)

Alumnos:

José Ángel Rentería Campos //A00832436

Santiago Andrés Serrano Vacca //A01734988

Thomas Freund Paternostro //A00831997

Fecha de entrega:

19/11/2021

```
19      ifstream bitacora("bitacora.txt");
20      ofstream output("output.txt");
21      int n, m;
22      bitacora >> n >> m;
23
24      map<string, int> IPtoID;
25      string* IDtoIP = new string[n];
26
27      string currentIP;
28      for(int i = 0; i < n; i++){
29          bitacora >> currentIP;
30          IPtoID.insert({currentIP, i});
31          IDtoIP[i] = currentIP;
32      }
```

La línea 19 a la 26 tiene una complejidad O(1) y de la línea tiene una complejidad O(n).

```
34      list<list<int>> adjList(n);
35
36      string currentLine;
37      string currentOriginIP, currentDestinationIP, unused;
38      list<list<int>>::iterator it = adjList.begin();
39      int currentIndex = 0;
40      while(getline(bitacora, currentLine)){
41          stringstream currLineStream(currentLine);
42          currLineStream >> unused >> unused >> unused >> currentOriginIP >> currentDestinationIP;
43
44          removePort(currentOriginIP);
45          removePort(currentDestinationIP);
46
47          int originID = IPtoID[currentOriginIP];
48          int destID = IPtoID[currentDestinationIP];
49
50          advance(it, originID - currentIndex);
51          currentIndex = originID;
52
53          it->push_back(destID);
54      }
```

La línea 34 a la 39 tiene una complejidad O(1) y de la 40 a la 53 tiene una complejidad O(n).

```
56      map<int, vector<string>> fanOutToIPs;
57      it = adjList.begin();
58      for(int i = 0; i < n; i++){
59          if(fanOutToIPs.find(it->size()) != fanOutToIPs.end()){
60              fanOutToIPs[it->size()].push_back(IDtoIP[i]);
61          } else {
62              fanOutToIPs.insert({it->size(),{IDtoIP[i]}});
63          }
64          advance(it, 1);
65      }
66
67      map<int, vector<string>>::iterator mapIt = fanOutToIPs.end();
68      advance(mapIt, -1);
69
```

La línea 56 a la 57 tiene una complejidad O(1), 40 a la 53 tiene una complejidad O(n) y la 67 y 70 también tiene una complejidad O(1).

```
70      output << "Posibles IP del Boot Master: " << endl;
71      for(string ip : mapIt->second){
72          output << ip << endl;
73      }
74      output << endl;
75
76      mapIt = fanOutToIPs.end();
77      for(int i = 0; i < fanOutToIPs.size(); i++){
78          advance(mapIt, -1);
79          output << "fan-out de " << mapIt->first << ":" << endl;
80          for(string ip : mapIt->second){
81              output << ip << endl;
82          }
83          output << endl;
84      }
85
86      cout << "He terminado. Mirar archivo output.txt." << endl;
87
88      delete[] IDtoIP;
89
90      return 0;
91  }
```

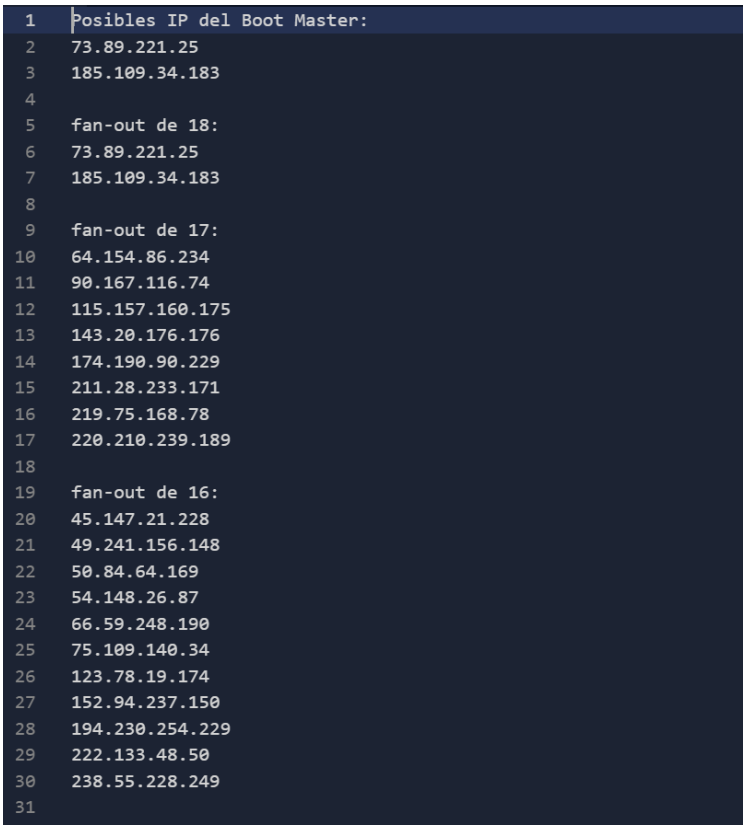
Desde la línea 71 a la 74 el tiempo tiene una complejidad de O(n). De la línea 74 a la 76 se tiene una complejidad de O(1). De las líneas 77 a la 79, además de la línea 83 es de complejidad O(n). Las líneas 80 y 81 tienen un tiempo de complejidad O(n^2).
Las líneas 88, 88 y 90 tienen una complejidad de O(1).

El documento refleja la importancia y eficiencia del uso de grafos en una situación problema de esta naturaleza.

En esta actividad se trabajó con grafos por medio de una lista de adyacencia. Los grafos en de esta naturaleza son importantes dado que este tuvo una cantidad considerable de datos qué tenían que ser analizados. Dónde se requirió analizar más de 13000 diferentes IPs y 90000 artistas se trabajó con una lista de adyacencia particularmente útil dado que permite encontrar la presencia o ausencia de un borde específico entre dos nodos. De igual manera esta estructura de datos nos permitió iterar rápidamente sobre todas las aristas dado que nos permite acceder a cualquier vértice adyacente de manera directa. Por estas razones se considera a esta estructura de datos eficiente e útil para resolver este problema.

Para resolver las preguntas presentando el fan out en orden descendente, se puede encontrar los simples que son llamados con la mayor frecuencia. Al mismo tiempo se pide determinar el IP qué puede ser el boot master sin embargo no sé determino solo uno dado que el bootmaster es el que prende o enciende otras computadoras por lo tanto es un nodo origen y está activa los nodos destino y no al revés. Por lo tanto se estaba considerando sumar también los de destino con los fan in pero no se lo revisó más seguido de que no se encontraba en las indicaciones sino porque la estructura de una lista adyacente no sería eficiente para verificar que ips de origen tienen un ip de destino.

En el screenshot se puede evidenciar los dos IPS dónde cada uno tiene 18 accesos.



```
Posibles IP del Boot Master:
73.89.221.25
185.109.34.183

fan-out de 18:
73.89.221.25
185.109.34.183

fan-out de 17:
64.154.86.234
90.167.116.74
115.157.160.175
143.20.176.176
174.190.90.229
211.28.233.171
219.75.168.78
220.210.239.189

fan-out de 16:
45.147.21.228
49.241.156.148
50.84.64.169
54.148.26.87
66.59.248.190
75.109.140.34
123.78.19.174
152.94.237.150
194.230.254.229
222.133.48.50
238.55.228.249

```

Posiblemente lo que se pudiera hacer para determinar por medio de grafos utilizar un algoritmo BFS dónde se plantea qué cada uno de estos y peso en el origen y se tiene un contador con la cantidad de conexiones directas que se generan y a base de esto se pudiera inferir are bootmaster. Realmente no se pudo determinar esto porque no llegamos a resolver cómo poner el ip como nodo origen pero en este caso hipotético los grafos permitirían encontrar y determinar con exactitud cuál sería el origen de una manera de que otras estructuras de datos no pudieron hacerlo. Cómo civilizaciones de análisis breve el código tiene una complejidad n^2 qué no es lo más eficiente pero el tiempo de procesamiento es casi instantáneo por lo tanto se considera que cumple con lo requerido.