Laboratorio Nro. 1: Implementación de Grafos

Kevin Arley Parra Henao
Universidad EAFIT
Medellín, Colombia
kaparrah@eafit.edu.co

Daniel Alejandro Mesa Arango
Universidad EAFIT
Medellín, Colombia
damesaa@eafit.edu.co

Felipe Olaya Ospina
Universidad EAFIT
Medellín, Colombia
folayao@eafit.edu.co

3) Simulacro de preguntas de sustentación de Proyectos

3.1

Para el 1.4: Se realizó un conjunto de clases para poder guardar los demás parámetros que se nos daba en el archivo de texto. Se lee el texto desde la entrada estándar y se hace un pequeño proceso en el fichero Point1.cpp que se usa como ejemplo de la ejecución de las clases creadas, las cuales fueron Vertex (vértices), Edges (aristas) y DinaGraph (Grafo). El grafo lo implementamos con un map de vectores para las listas y a su vez se tiene un map de Vertex para los vértices.

Para el 1.1: Se usó en la clase DigraphAL un ArrayList de listas enlazadas, las cuales a su vez guardaba parejas que contiene un sucesor y su respectivo peso para cada vértice en el grafo, lo cual vendría siendo una arista. Para la clase DigraphAM se usó una matriz de enteros en la cual la primera posición representa el vértice fuente y la segunda posición representa el vértice destino y el valor que toma en dichas posiciones es el peso que hay en la arista que los une.

3.2

Se podría decir que es más conveniente utilizar una lista de adyacencia en grafos en el que queremos saber los sucesores de un nodo o saber qué aristas existen, además si queremos utilizar menos memoria. También es más natural usarlo si se tienen aristas que unan los mismos vértices, ya que con una matriz se tendría la necesidad de hacer un caso especial. Una matriz de adyacencia sirve si queremos saber si una arista está presente en un grafo en tiempo constante.

3.3

Es mejor utilizar una lista de adyacencia ya que debemos optimizar al máximo la cantidad de memoria que se necesita para la representación.

3.4

Es mejor utilizar lista de adyacencia, debido a que no van a existir muchas aristas por cada nodo; en este caso, por cada persona (que sería un nodo en el grafo) solo hay en promedio 200 amigos (amigos sería una forma de decir que dos nodos se conectan en el grafo para este caso). Entonces no vamos a necesitar tener los demás espacios para la representación de los amigos de cada persona, además si queremos saber los amigos de una persona, sería mejor usar la lista de adyacencia y obtener la lista del nodo que a esa persona representa.

3.5

Es mejor tener una matriz de adyacencia porque igual se tienen que tener todos los vértices conectadas por si en un momento dado se establece la conexión entre dos dispositivos, se pueda saber si dicha conexión existe (si dicha arista existe). Además, el tamaño del grafo casi siempre es constante, no es frecuente que se modifique el tamaño en un enrutador.

3.6

Se utilizaron estructuras como pilas, matrices y dos tipos de lista además de arreglos, por supuesto no todos se usaron en una solución si no que realizamos dos soluciones concretas al problema 2.1, el primero fue representar mediante una matriz el grafo, a continuación el proceso, primero hacemos la lectura de los datos esto mediante el scanner de java, leemos un entero que representa el número de vértices del primer grafo a colorear, luego mediante la ayuda del ciclo "while" comprobamos cuando debe terminar el programa dentro del ciclo creamos una matriz cada vez del tamaño del número de vértices especificado, a continuación leemos el número de aristas y tomamos un ciclo para leer ese número de aristas, a medida que leemos las conexiones vamos llenando el grafo con un "1" esto para indicar si hay o no conexión, luego llamamos a la función esColoreable, a la cual le pasamos el grafo es decir la matriz, esta crea un arreglo y lo inicializa con "-1" en todas sus posiciones (número de vértices) en la posición "0" cambiamos por un "1" luego de esto llamamos una función auxiliar a la cual le pasamos el grafo y el arreglo inicializado con "-1" al cual llamaremos coloreable, en esta función auxiliar creamos una pila de enteros y allí metemos al primer vértice coloreado es decir el "0" luego entramos a un ciclo, estando allí, saco de la pila el número ingresado, verifico que no tenga una conexión a el mismo, si la hay retornamos "false" si todo está bien entramos a un ciclo donde recorreremos todos sus vecinos y verificamos si están despintados (en -1) y si hay una conexión en el grafo (en 1) si esto cumple en el arreglo colorear en el vértice que este en "-1" vamos a colorear restándole a "1" el número del vértice en el que estamos parados actualmente (recordemos que estamos recorriendo los vecinos) esto quiere decir que a cero le habíamos puesto "1" para indicar que ya estaba coloreado y esto le restamos a "1" es decir queda pintado con el numero "0" con esta resta existirán solo dos posibilidades de color, es decir el "1" y el "0" y luego agrego a la pila el vértice que acabe de colorear, y así con todos los vecinos, también se verifica en este mismo ciclo si existe una conexión y a su vez tanto el nodo actual como algún vecino tienen la misma pintura o color si esto pasa el programa termina y retorna "false" cada vez que el while se ejecuta saca un número hasta que la pila este vacía, todo número que deje de estar despintado no vuelve a ser ingresado en la pila, al terminar y si no hubo problema alguno en el ciclo significa que es bicoloreable, al volver al main se verifica "true" o "false" y se imprime lo correspondiente, luego se vuelve a pedir un numero de vértices, aristas y conexiones y todo se repite hasta ingresar en el número de vértices un "0". Cabe resaltar que la implementación con listas del grafo también fue hecha y el procedimiento es el mismo, nos dimos cuenta que también es recursivo pero por falta de tiempo no pudimos implementar una tercer solución, además podemos agregar que la solución más óptima para nosotros en este problema en específico es con listas pues no tenemos que recorrer todo el grafo, solo aquellos vecinos que tenga el nodo, la pila nos permite ingresar en tiempo constante así como sacar al igual que la LinkedList para la lista de vecinos de cada uno de los nodos.

3.7

Hicimos dos soluciones, pero calcularemos solo la complejidad de la solución implementada con matrices.

```
public boolean esColoreable(int [][] grafo)
{

    int [] yaColor = new int[grafo.length];                C
    int i = 0;     C
    while(i < yaColor.length)         n
       {
          yaColor[i] = -3;  C*n
          i++; c*n
       }
    yaColor[0] = 1;   c
    boolean es =  aux(grafo, yaColor);c* O(n^2)
    return es? true : false;    c
```

```
   }
T(n) = c+c+n+c*n+c*n+c+n^2+c
O(c+c+n+c*n+c*n+c+n^2+c)
O(c*n^2)
O(n^2)

   private boolean aux(int [][] grafo, int [] color)
   {
      Stack<Integer> recorridos =  new Stack<>();     c

      recorridos.push(0);                                    c
      while(recorridos.size() != 0)                          n
         {
            int actual = recorridos.pop();                    c*n
            if(grafo[actual][actual] == 1) return false;      c*n
            for(int i = 0; i < grafo.length; i++)             n*n
               {
                  if(grafo[actual][i] == 1 && color[i] == -3)    c*n*n
                     {
                        color[i] = 3-color[actual];               c*n*n
                        recorridos.push(i);                       c*n*n
                     }else if(grafo[actual][i] == 1  && color[actual] == color[i]) return false;    c*n*n
               }
         }
      return true;                  c
   }
   T(n) = c+c+c+c*n+c*n+n*n+c*n*n+c*n*n+c*n*n+c*n*n+c*n*n
O(c+c+c+c*n+c*n+n*n+c*n*n+c*n*n+c*n*n+c*n*n)
O(c*n*n)
O(n^2)
   public static void main(String[] args)
   {
      Ejercicio2 g = new Ejercicio2();                    c
      Scanner consola =  new Scanner(System.in);    c
      int numeroVertices= consola.nextInt();          c
      int contador = 1;                                  c
      int aristas = 0;                             c
      int origen = 0;                                 c
      int destino = 0;                                c
      while(numeroVertices != 0)
         {
            aristas = consola.nextInt();              c
            int [][] grafo = new int[numeroVertices][numeroVertices];       c
            //ArrayList <LinkedList<Integer>> grafo = new ArrayList<>();
            /*for(int j = 0; j < numeroVertices; j++ )
               {
                  grafo.add(new LinkedList<Integer>());
               }*/
            contador = 1;                                  c
            while(contador <= aristas)
```

```
            {
               origen = consola.nextInt();                    c
               destino = consola.nextInt();                   c
               //grafo.get(origen).add(destino);
               grafo[origen][destino] = 1;                    c
               //System.out.println("origen: " + origen+ " " + "destino: " + destino);
               contador++;
            }
         String res = g.esColoreableP(grafo)? "BICOLORABLE" : "NOT BICOLORABLE"; c*n^2
         System.out.println(res); c
         numeroVertices = consola.nextInt();       c
      }

   }
```

T(n) = c+c*n^2
O(c+c*n^2)
O(c*n^2)
O(n^2)

La complejidad de analizar un solo grafo es la anterior, cuando son varios grafos quedaría "n*m + n^2" la m es el número de aristas y considerando que las conexiones nunca son a sí mismas la más grande es "n^2" entonces es para varios grafos a procesar de O(n^2).

3.8

Las variables c, son constantes, la m representa siempre el número de vértices que tiene el grafo en el peor de los casos puede recorrer todos los vértices siendo n*n sin importar si hay conexión o no, ya que para comprobarlo hay que ir por toda la matriz, a su vez m en el método main es el número de aristas que en el peor de los casos es el mismo de vértices, pero puede llegar a ser menor.

4) Simulacro de Parcial

1.

	0	1	2	3	4	5	6	7
0				1	1			
1	1					1		
2					1		1	
3								1
4			1					
5								
6			1					
7								

Nota: entre el uno y el dos no se especifica cual va a cuál.

2.

0->[3, 4]
1->[0 , 5]
2->[4, 6]
3->[7]
4->[2]
5->[]
6->[2]
7->[]

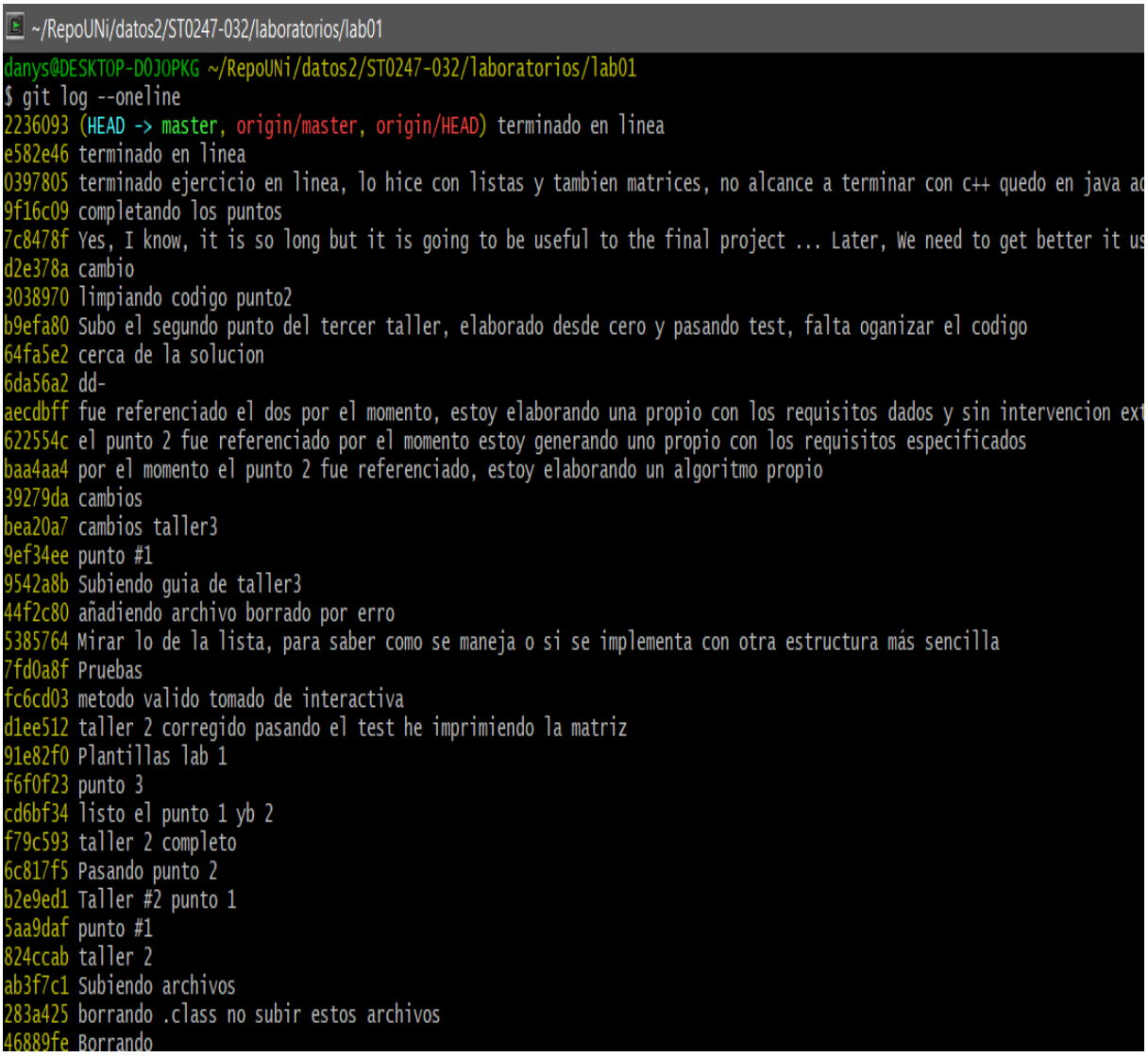
3. b

6) Trabajo en Equipo y Progreso Gradual (Opcional)

a) Actas de reunión

Integrante	Fecha	Hecho	Haciendo	Por hacer
Daniel Mesa	17/02/2018	Punto 2.1 con Felipe	Dos del simulacro y dos del 3	Subir codigo
Kevin Parra	16/02/2018	Comienzo 1	Punto 1	Subir Punto 1 y algunos del 3
Kevin Parra	18/02/2018	Numerales del 1 con Felipe y los correspondientes del 3	subiendo	
Felipe	18/02/2018	Termine el 2.1 algunas cosas y del 3		
Daniel Mesa	18/02/2018	Subir código y guias	Subir todo	

b) El reporte de cambios en el código



c) El reporte de cambios del informe de laboratorio