DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Bouzón, María Belén	128/13	belenbouzon@hotmail.com
Jiménez, Paula	655/10	puly05@gmail.com
Montepagano, Pablo	205/12	pablo@montepagano.com.ar
Rey, Maximiliano	037/13	rey.maximiliano@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Problema 2: Algoritmo Exacto para List Coloring **3**
1.1. Descripción de la problemática . . . 3
1.2. Resolución propuesta y justificación . . . 3
1.3. Análisis de la complejidad . . . 3
1.4. Código fuente . . . 3
1.5. Experimentación . . . 3
1.5.1. Constrastación Empírica de la complejidad . . . 3
1.5.2. Mejor Caso . . . 3
1.5.3. Peor caso . . . 3

2. Problema 3: Heurística Constructiva Golosa **4**
2.1. Descripción de la problemática . . . 4
2.2. Resolución propuesta y justificación . . . 4
2.3. Código fuente . . . 9
2.4. PseudoCódigo y Análisis de la complejidad . . . 11
2.5. Experimentación . . . 11

1. Problema 2: Algoritmo Exacto para List Coloring

1.1. Descripción de la problemática

1.2. Resolución propuesta y justificación

El pseudocódigo de nuestro algoritmo es el siguiente:

1.3. Análisis de la complejidad

1.4. Código fuente

1.5. Experimentación

1.5.1. Constrastación Empírica de la complejidad

1.5.2. Mejor Caso

1.5.3. Peor caso

2. Problema 3: Heurística Constructiva Golosa

2.1. Descripción de la problemática

En esta oportunidad se nos pide colaborar en la asignación de aulas con diversos recursos a un conjunto de materias (con un horario establecido e invariable) que necesitarían hacer uso de ellos durante el dictado de clase.
Como en este caso nos interesa distinguir las aulas de acuerdo a sus recursos, le asignaremos a cada una un color y uniremos con aristas aquellas materias que se han querido reservar para horarios cuya intersección no es vacía.
Dicho esto, nuestra tarea consiste en generar un algoritmo que implemente una heurística cuyo objetivo sea encontrar alguna forma de colorear los nodos (es decir, asignar a cada materia un aula), minimizando en la medida de lo posible la cantidad de conflictos (entendidos estos como instancias que impiden que el coloreo generado sea válido, es decir, que verifique para toda arista que a sus nodos adyacentes se les haya asignado distinto color.)

2.2. Resolución propuesta y justificación

Dos ideas de peso sugieron al proponernos resolver este problema:

- Iterar sobre el grafo (a lo sumo C-1 veces) descartando de cada uno de los nodos uno de sus colores posibles, escogido mediante una función de peso que decidiera cuál de ellos podría generar más conflictos si se utilizara para colorear al nodo examinado.

- Realizar un barrido del grafo mediante BFS utilizando una función de peso similar a la mencionada que permitiera decidir para cada nodo cuál sería el color - potencialmente - de menor riesgo en una vecindad reducida y asignárselo.

Un problema emergió al analizar la primera de las ideas: el hecho de que se realizaran varias iteraciones sobre el mismo grafo hacía que al comenzar cada una de ellas cada nodo tuviese más información del grafo completo que la que podía verificar en la anterior. Dicho de otra forma: en la primera iteración cada nodo tomaba la mejor decisión posible en función de su vecindad de primer nivel, pero en las siguientes ya dicha vecindad - ahora modificada de acuerdo a su propio entorno - le proveía al nodo más datos que los alcanzables hasta entonces. Es decir, no cumplía los requisitos para ser catalogado como “algoritmo goloso”.
Es por esto que se decidió implementar la segunda opción, la cuál se explica a continuación:
Supóngase que se tiene el siguiente grafo, donde los números dentro de cada nodo indican su índice y los que se encuentran fuera enumeran sus colores posibles.

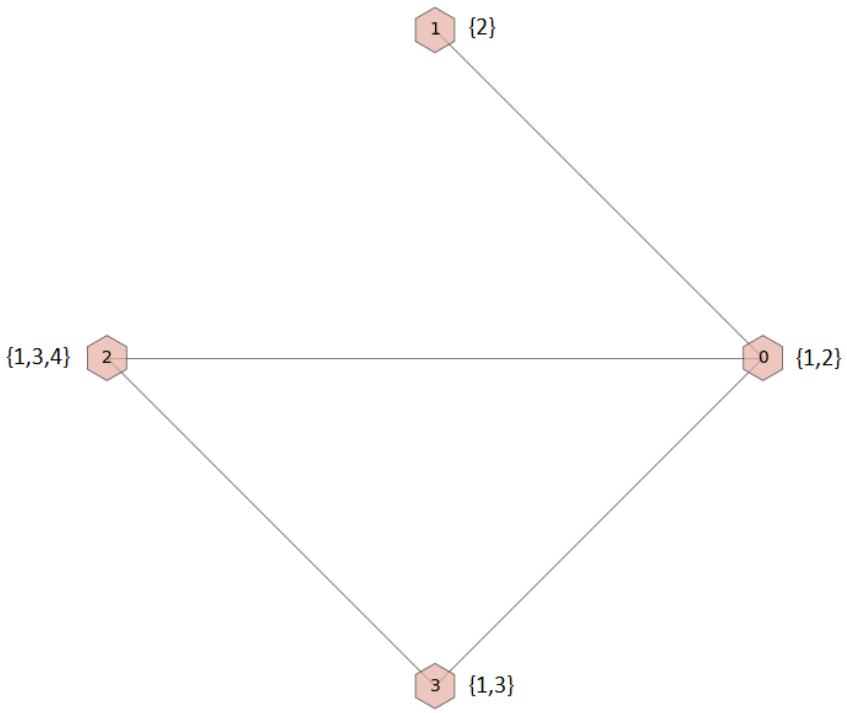

Figura 1: Grafo sin colorear

Defínase Peso(N_c) como la función que determina la magnitud del perjuicio ocasionable al elegir colorear el nodo N de color c (con c ϵ {colores de N}), $X_{(c,p)}$ como el valor relativo X del color c para el nodo p. Entonces en un primer paso, los valores son los siguientes:

- **Nodo 0**
  - Color 1:
    - $\frac{0}{1}(1,1)$
    - $\frac{1}{3}(1,2)$
    - $\frac{1}{2}(1,3)$
  - Color 2:
    - $\frac{1}{1}(2,1)$
    - $\frac{0}{3}(2,2)$
    - $\frac{0}{2}(2,3)$

Por lo tanto,

$\text{Peso}(0_1) = \frac{\frac{1}{2}(1,3)}{2} + \frac{\frac{1}{3}(1,2)}{3} + \frac{\frac{0}{1}(1,1)}{4} \simeq 0.36$

$\text{Peso}(0_2) = \frac{\frac{1}{1}(2,1)}{2} + \frac{\frac{0}{3}(2,2)}{3} + \frac{\frac{0}{2}(2,3)}{4} \simeq 0.5$

Como Peso(0_1) $\leq$ Peso(0_2), el color del que se pintará al nodo es del 1 (tal como se muestra en la figura 2)

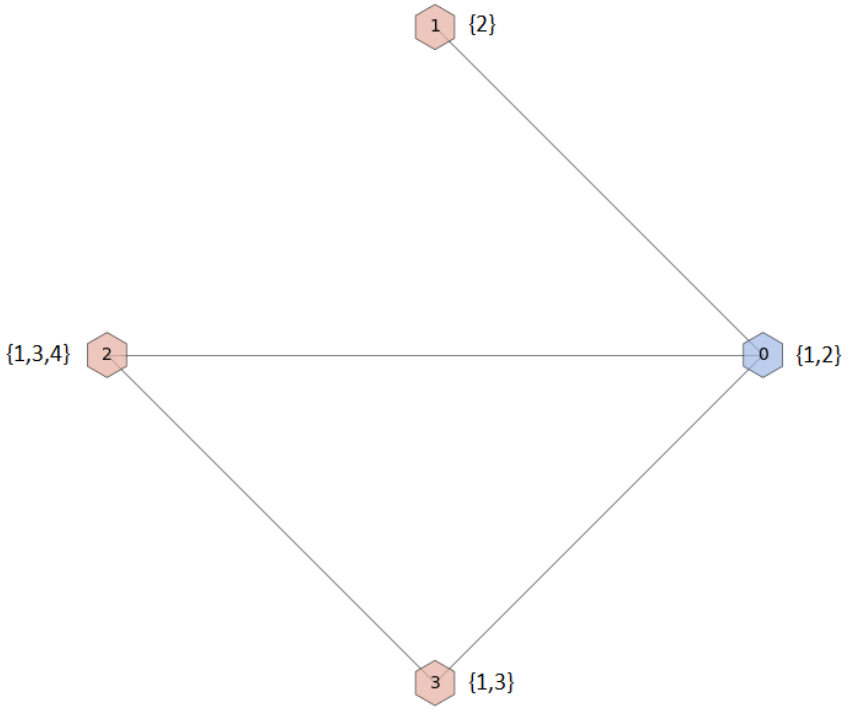

Figura 2: Al nodo 0 se le asigna el color 1

Prosigamos con el nodo 1. Su valor es el siguiente:

- **Nodo 1**
  - Color 2:
    - $\frac{0}{2}_{(2,0)}$

Nótese que el valor $\frac{0}{2}_{(2,0)}$ no es $\frac{1}{2}_{(2,0)}$ porque el nodo 0 ya fue coloreado y de un color distinto al 2. Es por esto que al nodo 0 "no le importa" que su vecino, el nodo 1, se coloree del color 2 y por lo tanto el peso que retorna para dicho color es *0*. Además, este es el único peso que tiene en cuenta el nodo 1, puesto que no tiene más nodos adyacentes ni otros colores disponibles. Por lo tanto, como se muestra en la figura 3, el nodo 1 se pinta del color 2.

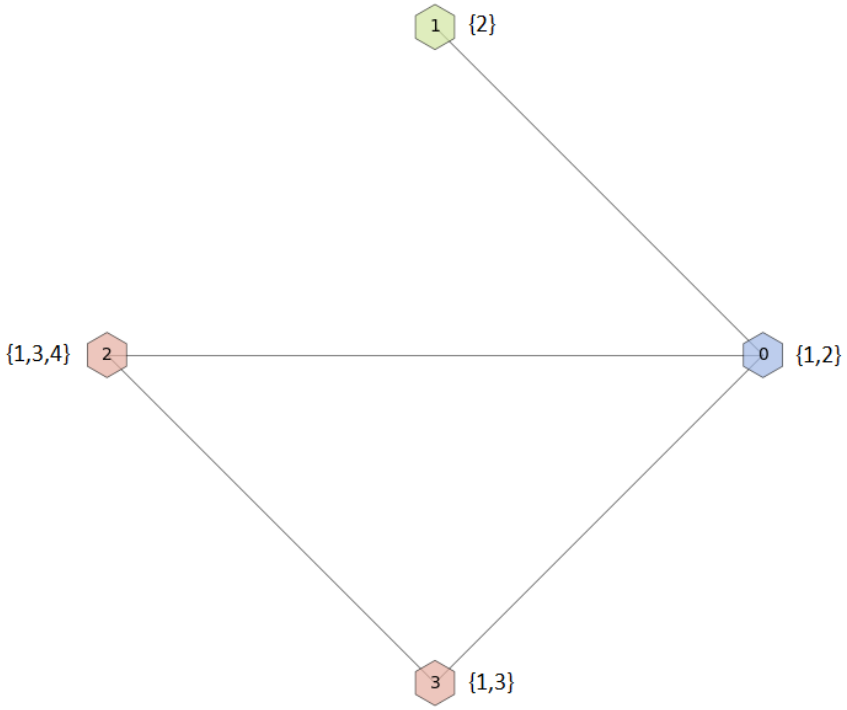

Figura 3: Al nodo 1 se le asigna el color 2

Continuemos el coloreo con el nodo 2:

- **Nodo 2**
  - Color 1:
    - $\frac{0}{2}(1,0)$
    - $\frac{1}{2}(1,3)$
  - Color 3:
    - $\frac{0}{2}(3,0)$
    - $\frac{1}{2}(3,3)$
  - Color 4:
    - $\frac{0}{2}(4,0)$
    - $\frac{0}{2}(4,3)$

Nuevamente, los colores posibles del nodo 0 no tienen injerencia a menos que el valor por el que se pregunte sea equivalente al color con el cual se pretende colorear un nodo adyacente.
Como en este caso Peso$(2_4) \leq$ Peso$(2_3) \leq$ Peso(2_1), se le asigna al nodo 2 el color 4 (ver 4)

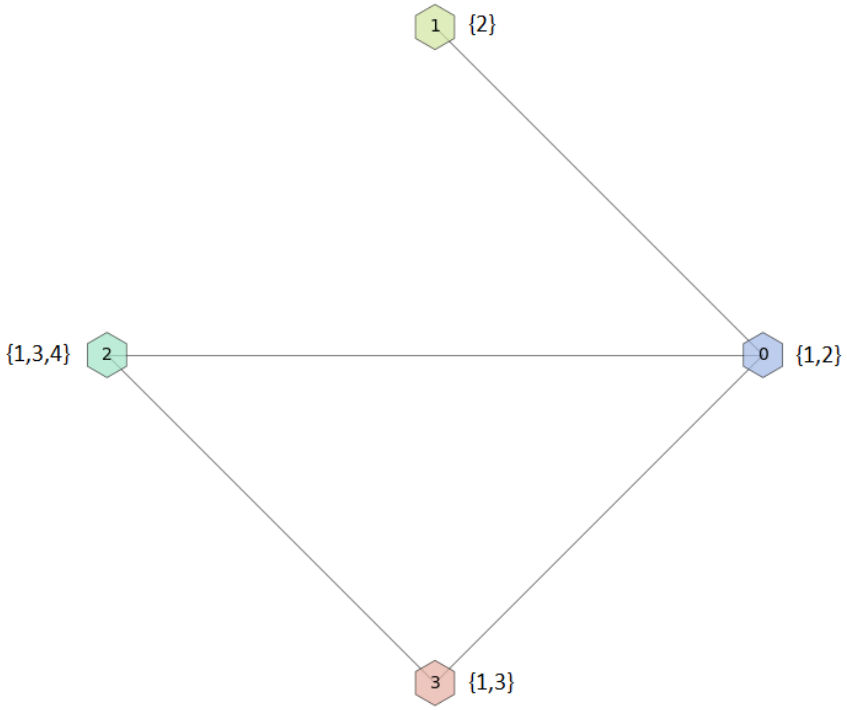

Figura 4: Al nodo 2 se le asigna el color 4

Por último, definamos el coloreo del nodo 3:

- **Nodo 3**
  - Color 1:
    - $\frac{1}{1}(1,0)$
    - $\frac{0}{3}(1,2)$
  - Color 3:
    - $\frac{0}{2}(3,0)$
    - $\frac{0}{3}(3,2)$

Obsérvese que $\frac{1}{1}(1,0)$ es distinto del esperado $\frac{1}{2}(1,0)$. Esto se debe a que ell valor escogido representa una medida de la importancia que se le da al nodo ya coloreado “0”. Como queremos evitar la mayor cantidad de conflictos, ante casos de este estilo se define que el nodo coloreado retorne el valor absoluto “1” (recordemos que, cuanto más grande este valor, mayor es la restricción de pintar un nodo de dicho color).

De esta forma, queda coloreado todo el grafo como se muestra en la figura 5

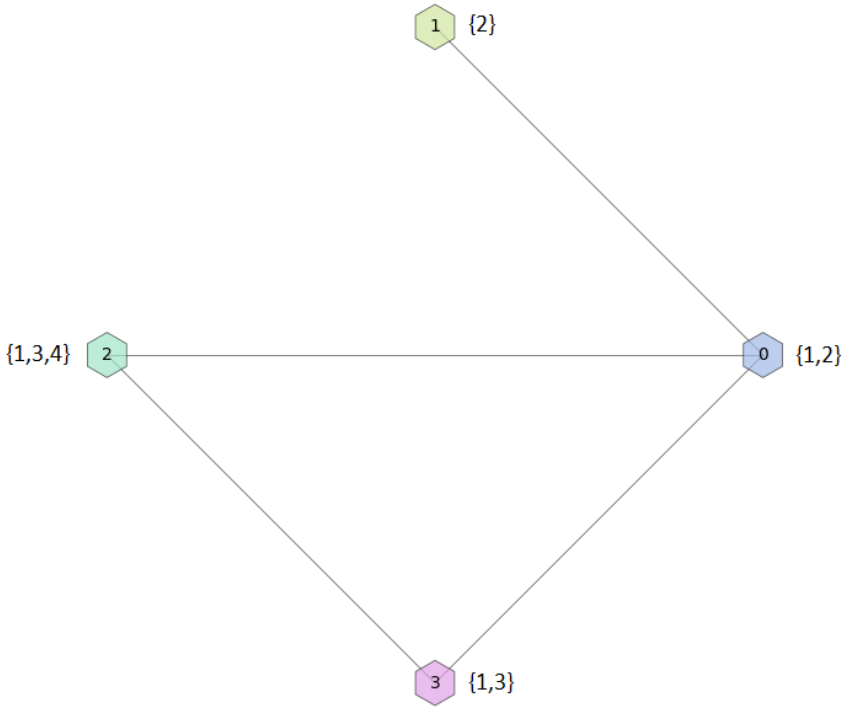

Figura 5: Al nodo 3 se le asigna el color 3

2.3. Código fuente

A continuación se incluyen las partes más relevantes del código generado

La clase *Lector.java* se encarga de tomar los datos del archivo de entrada y procesarlos para construir el grafo mediante el método *MakeGraph()*:

```
1    public Grafo MakeGraph() throws IOException
2    {
3       Grafo grafo = new Grafo();
4       int[] nodosAristasColores = Ej3Utils.ToIntegerArray(this.getArchivo().readLine().split("␣"));
5       grafo.cantidadDeNodos = nodosAristasColores[0];
6       grafo.setCantidadDeAristas(nodosAristasColores[1]);
7       grafo.setCantidadDeColores(nodosAristasColores[2]);
8
9       try { grafo.setNodos(this.ObtenerListaDeNodos(grafo.cantidadDeNodos, grafo.getCantidadDeColores
            ()));}
10      catch (IOException e) { System.out.println("Se␣produjo␣un␣error␣al␣generar␣los␣nodos␣del␣grafo."
            );}
11      boolean[][] matrizDeAdyacencia = GenerarMatrizDeAdyacencia(grafo.getCantidadDeNodos(), grafo.
            getCantidadDeAristas(), false);
12      grafo.setListaDeAdyacencia(Ej3Utils.matrizDeAdyacenciaToListDeAdyacencia(matrizDeAdyacencia,
            grafo.getNodos()));
13
14      return grafo;
15   }
```

La clase *Grafo.java* contiene el método *MakeRainbow()* que colorea el grafo con la heurística propuesta:

```
1   public void MakeRainbow() // O(n^2*c*log(n))
2   {
3      int nodosPintados = 0;
4      LinkedList<Nodo> colaNodos = new LinkedList<Nodo>();
5
6      while(nodosPintados < this.cantidadDeNodos)
7      {
8         colaNodos.add(PicANode(this)); //O(1)
9
10        while (!colaNodos.isEmpty()) //O(n)
11        {
12           Nodo nodoActual = colaNodos.removeFirst(); //O(1)
13
14           if (!nodoActual.isVisitado())
15           {
16              colaNodos.addAll(this.getVecinosDe(nodoActual)); //O(n)
17              LinkedList<Integer> coloresRestantes = nodoActual.getColoresRestantes(); //O(1)
18              PintarNodo(nodoActual, coloresRestantes, this); //O(c*n*log(n))
19              nodosPintados ++;
20              nodoActual.setVisitado(true); //O(1)
21           }
22        }
23     }
24  }
25
26  private static Nodo PicANode(Grafo grafo)
27  {
28     Nodo next = new Nodo();
29     for(Nodo nodo : grafo.getNodos())
30     {
31        if (!nodo.isVisitado())
32        {
33           next = nodo;
34           break;
35        }
36     }
37     return next;
38  }
39
40  private static void PintarNodo(Nodo nodoActual, LinkedList<Integer> coloresRestantes, Grafo grafo)
        //O(c*n*log(n))
41  {
42     int colorAPintar = CalcularColorMenosPerjudicial(nodoActual, grafo); //O(c*n*log(n))
43     nodoActual.setColor(colorAPintar);
44  }
45
46  private static int CalcularColorMenosPerjudicial(Nodo nodoActual, Grafo grafo) //O(c*n*log(n))
47  {
48     Double pesoColor = 1.0;
49     int colorAPintar = -1;
50     for (int color : nodoActual.getColoresRestantes()) //O(c)
51     {
52        Double peso = CalcularPeso(color, grafo.getVecinosDe(nodoActual)); //O(nlog(n))
53        if (peso <= pesoColor)
54        {
55           pesoColor = peso;
56           colorAPintar = color;
57        }
58     }
59     return colorAPintar;
60  }
61
62
63  private static Double CalcularPeso(int color, List<Nodo> vecinos)
64  {
65     ArrayList<Double> pesos = new ArrayList<Double>();
66     for (Nodo nodo : vecinos) //O(n)
67     {
68        if (nodo.LeImportaQueSuVecinoSePinteDelColor(color)) //O(1)
69              pesos.add(nodo.PeligroDePintarUnVecinoDelColor(color));//O(1)(amortizado)
70
71     }
72     Collections.sort(pesos); //O(nlog(n))
73     Double pesoTotal = 0.0;
74     for (int k = 0; k < pesos.size(); k++) //O(n)
75        pesoTotal += pesos.get(k)/(k+2); //O(1)
76
77     return pesoTotal;
78  }
```

2.4. PseudoCódigo y Análisis de la complejidad

A continuación se presenta el pseudocódigo del coloreo implementado:

```
while nodos pintados ¡cantidad de nodos //O(n) do
  Elegir un nodo no visitado del grafo // O(n)
  Encolarlo //O(1)
  while hay elementos en la cola do
    Tomar el primer elemento de la cola //O(1)
    if no fue coloreado //O(1) then
      Agregar a todos sus vecinos a la cola //O(n)
      Pintar nodo //O(n*c*log(n))
    end if
  end while
end while
```

El ciclo principal se ejecuta porque dentro del bucle anidado sólo se colorea una componente conexa por vez. Verificando que el algoritmo funcione hasta que la cantidad de nodos pintados sea efectivamente la cantidad de nodos total, y escogiendo cada vez un nodo de una componente no visitada podemos asegurarnos de que el coloreo se realiza sobre la totalidad del grafo.
El método *PintarNodo()* tiene como cota superior O(n*c*log(n)). Como se puede ver en el código, dicha complejidad la alcanza al calcular el color menos perjudicial (método *CalcularColorMenosPerjudicial(nodo,grafo)*). En él para cada color C del nodo examinado calcula su peso chequeando si C pertenece al conjunto de colores posibles de cada uno de sus nodos adyacentes. Gracias a que cada nodo lleva un array de booleanos que responde a dicha operación en O(1) la cota no es mayor.
Como la condición de que el nodo no haya sido visitado se valida exactamente n veces y cada vez se agregan a lo sumo n-1 elementos a la cola (con un costo O(1) cada uno), dicha parte del algoritmo cuesta $O(n^2)$. Sin embargo un costo mayor se paga al ejecutar el método *PintarNodo()* - de costo O(n*c*log(n)) - n veces. Esto hace que la complejidad final del algoritmo ascienda a $O(n^2*c*\log(n))$.

2.5. Experimentación