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 1: Algoritmo Exacto para 2-List Coloring

1.1. Descripción de la problemática

Este ejericicio está centrado en la resolución de un problema particular de coloreo, 2-listColoring. En este caso, cada nodo se puede colorear solo con uno o dos colores específicos. El algoritmo debe determinar si existe o no un coloreo en donde no existen dos nodos consecutivos del mismo color y, en caso afirmativo, encontrar una solución. Este problema es facilmente reducible a 2-SAT, por lo cual existen algortmos polinomiales que lo resuelven.

1.2. Resolución propuesta y justificación

Como mencionamos antes, el problema del 2-ListColoring es facilmente reducible a un problema de 2-SAT, y en esto se basa el algoritmo implementado. Si existe una coloración posible, entonces el hecho de que un determinado nodo esté pintado de un color implica que ninguno de sus vecinos esta pintado del mismo. Además, si un color tenía dos opciones posibles, el nodo esta pintado de un color A si y solo si no esta pintado del color B. El 2-SAT se representa con un grafo dirigido en donde un nodo A (que representa una premisa) apunta a un nodo B si y solo si A implica B.

Una vez construido el grafo dirigido que representa el problema, lo primero que hace el algoritmo es encontrar componentes fuertemente conexas en el grafo. En este contexto, una componente fuertemente conexa representa un conjunto de propociones de las cuales se puede afirmar que todas son válidas o todas son falsas. Consecuentemente, se construye un grafo dirigido en donde cada nodo representa una componente fuertemente dirigida del grafo anterior; logicamente, este segundo grafo no contiene ninguna componente fuertemente conexa. El algoritmo debe chequear, además, si se incluye en una misma componente conexa una propoción (asignación de un color a un nodo) y su negación, en cuyo caso el nodo que representa dicha componente debe tener un valor de verdad negativo. Además, el algoritmo registra, para cada componente, cual es la componente que representa su negación, es decir, que contiene la negación de las variables de la primera. Finalmente comienzan a fijarse los valores de verdad. Cuando se fija un valor para un nodo determinado, el nodo que contiene los valores contrarios se fija en el valor opuesto. Además, si un nodo se fija en false, todos los nodos que implican al mismo deben fijarse en false (igual recursivamente) y si se fija en true, todos los que son implicados por el mismo deven fijarse en true. Si el algoritmo encuentra una contradicción (es decir, un nodo que debía setearse en false fue seteado en true o viceversa), determina que el problema no teiene una solución válida.

1.3. Análisis de la complejidad

El grafo dirigido contiene a los sumo cuatro veces más nodos que el grafo de entrada, y el grafo compacto tiene a los sumo tantos nodos como el grafo dirigido. Por lo tanto, podemos afirmar que, sea n la cantidad de nodos inicial, n' la cantidad de nodos del grafo dirigido y n" la cantidad de nodos del grafo compacto, $\mathcal{O}(f(n)) = \mathcal{O}(f(n')) = \mathcal{O}(f(n''))$ (si no se conocen datos adicionales para la entrada).

Generar el grafo dirigido a partir del grafo de entrada tiene una complejidad de $\mathcal{O}(n^2 \log n)$, ya que para cada nodo del grafo, es necesario crear los nodos dirigidos que lo representan y definir sus aristas con otros nodos. A medida que se van creando, los nodos se guardan y se buscan en un TreeSet.

El algoritmo utilizado para encontrar las componentes fuertemente conexas no recorre más de dos veces todos los nodos, por lo cual su complejidad es lineal, al igual que el algoritmo que determina el valor de verdad de los nodos compactos.

1.4. Código fuente

A continuación se incluyen las partes más relevantes del código.
La clase lector se encarga de leer la entrada y transforla en un grafo. Como en los test de complejidad es importante no medir el tiempo que se tarda en cargar el archivo en memoria, el Lector posee funciones para cargar la información sin procesar.

```
    private void cargar_y_procesar() throws IOException{
        if(this.is==null){
            throw new IOException("error, lector no vinculado a archivo");
        }

        String parametros = this.leer_palabra();
        String [] parametros_procesados = parametros.split(" ");
        this._cantidad_nodos = Integer.parseInt(parametros_procesados[0]);
        this._cantidad_aristas = Integer.parseInt(parametros_procesados[1]);
        this._cantidad_colores = Integer.parseInt(parametros_procesados[2]);
        this._nodos_del_grafo = new ArrayList<Nodo_Coloreable>(_cantidad_nodos);
        for(int i = 0;i<this._cantidad_nodos;i++){
            String nodo_string = this.leer_palabra();
            String [] nodo_string_procesado = nodo_string.split(" ");
            Nodo_Coloreable nuevo = new Nodo_Coloreable(i);
            nuevo.cantidad_colores = Integer.parseInt(nodo_string_procesado[0]);
            for(int j=1;j<=nuevo.cantidad_colores;j++){
                nuevo.colores.agregar_color(Integer.parseInt(nodo_string_procesado[j]));
            }
            this._nodos_del_grafo.add(nuevo);
        }
        for(int i = 0;i<this._cantidad_aristas;i++){
            String arista_string = this.leer_palabra();
            String [] arista_string_procesada = arista_string.split(" ");
            int nodo_1 = Integer.parseInt(arista_string_procesada[0]);
            int nodo_2 = Integer.parseInt(arista_string_procesada[1]);
            Nodo_Coloreable n_1 = this._nodos_del_grafo.get(nodo_1);
            Nodo_Coloreable n_2 = this._nodos_del_grafo.get(nodo_2);
            n_1.adyacentes.add(n_2);
            n_2.adyacentes.add(n_1);
        }
        this.is.close();
    }

    private void procesar_informacion_cargada(){
        for(int i = 0;i<this._cantidad_nodos;i++){
            String nodo_string = this.nodos_sin_procesar[i];
            String [] nodo_string_procesado = nodo_string.split(" ");
            Nodo_Coloreable nuevo = new Nodo_Coloreable(i);
            nuevo.cantidad_colores = Integer.parseInt(nodo_string_procesado[0]);
            for(int j=1;j<=nuevo.cantidad_colores;j++){
                nuevo.colores.agregar_color(Integer.parseInt(nodo_string_procesado[j]));
            }
            this._nodos_del_grafo.add(nuevo);
        }
        for(int i = 0;i<this._cantidad_aristas;i++){
            String arista_string = this.aristas_sin_procesar[i];
            String [] arista_string_procesada = arista_string.split(" ");
            int nodo_1 = Integer.parseInt(arista_string_procesada[0]);
            int nodo_2 = Integer.parseInt(arista_string_procesada[1]);
            Nodo_Coloreable n_1 = this._nodos_del_grafo.get(nodo_1);
            Nodo_Coloreable n_2 = this._nodos_del_grafo.get(nodo_2);
            n_1.adyacentes.add(n_2);
            n_2.adyacentes.add(n_1);
        }
    }
}
```

Los metodos que construyen el coloreo se encuentran en Calculador_de_Coloracion_Ej1

```
    private void DFS_original(Stack<Nodo_Dirigido_SAT> pila){
        Iterator<Nodo_Dirigido_SAT> it = this.grafo_dirigido.iterator();
        while(it.hasNext()){
            auxiliar_DFS_original(it.next(),pila);
        }
    }
    private void DFS_inverso_auxiliar(Nodo_Dirigido_SAT nodo,TreeSet<Nodo_Dirigido_SAT> conjunto,int
         id_componente_conexa_actual){
        if(nodo.id_componente_conexa==-1){
            conjunto.add(nodo);
            nodo.id_componente_conexa = id_componente_conexa_actual;
            for(Nodo_Dirigido_SAT vecino: nodo.adyacentes_inverso){
                if(vecino.id_componente_conexa==-2){
                    System.out.printf("error! 1\n");
                }
                DFS_inverso_auxiliar(vecino,conjunto,id_componente_conexa_actual);
```

1.5. Experimentación

Casos de test:

1.5.1. El triángulo

En este test existen tres nodos interconextados en donde cada nodo comparte un color con uno de sus vecinos. El algoritmo debería encontrar la única solución posible. Posteriormente, se le añade al triangulo un nodo extra vecino del primero con el color con el cual estaba pintado el primero, con lo cual no existe solución.

1.5.2. El pentagono

En esta oportunidad, se dibuja un k5. Cada nodo tiene como opción un color que no se repite en sus compañeros y otro que es compartido por más de uno. El algoritmo debería encontrar una solución

1.5.3. El señor de los anillos

En este caso se prueba un ciclo de seis nodos. A los pares se les asigna 0 y a los impares uno, y además todos poseen un segundo color que comparten con uno de sus vecinos. Como el grafo es bipartito, el algoritmo debería hallar una solución.

1.5.4. Grafos bipartitos completos

Otro de los casos se basa en generar grafos bipartitos completos. En este caso, se les asigna a todos los nodos 0 y 1. Como un grafo bipartito (en particular, uno completo) puede colorearse con dos colores, el algoritmo deberia encontrar una solución para el mismo. Posteriormente, se le añade al grafico una arista que une dos nodos que no estaban conectados, con lo que el grafo deja de ser bipartito y se necesitan 3 colores para colorearlo. En este caso, el algoritmo debería determinar que no existe solución.

1.5.5. Testeos de complejidad

Para testear la complejidad, se generan grafos al azar en donde todos los nodos poseen como opciones de coloreo a un color que no se repite en los demás nodos y otro que es seleccionado al azar. En una de las estrrategias se deja fija la cantidad de aristas, en otra se fija la cantidad de nodos, y en la otro se incrementan las aristas y los nodos en igual proporción.

1.5.6. Complejidad en grafos completos

En un último caso se generan grafos completos en donde se a incrementando la cantidad de nodos (y, logicamente, de aristas). Debido al consumo de memoria, este último caso se ha testeado con valores inferiores que al resto de los test.

2. Problema 2: Algoritmo Exacto para List Coloring

2.1. Descripción de la problemática

En esta oportunidad se nos pide resolver el problema List Coloring que consiste en (si es posible) colorear un grafo de forma que ningún nodo tenga el mismo color que un nodo adyacente, respetando la lista de colores de cada nodo, lo cual quiere decir que no se le puede asignar un color a un nodo si este no pertenecía a su lista de colores posibles.

2.2. Resolución propuesta y justificación

Para desarrollar un algoritmo exacto para este problema decidimos usar backtracking y de esta forma recorrer todas las opciones de colores y poder encontrar la solución si esta existe.

Como el enunciado pide que al llegar a un caso de 2-List Coloring se utilice el algoritmo del primer punto, decidimos cambiar el enfoque de la resolución para poder adaptarlo a instancias del ejercicio 1. Por lo tanto en lugar de usar el enfoque convencional que sería fijar un color para cada nodo y ver si es solución, optamos por recorrer todos los nodos del grafo fijando dos colores por cada uno y al llegar al último nodo llamar a 2-list Coloring para que chequee si con esa configuración de colores hay solución. Si la respuesta es "X" entonces se selecciona el siguiente color del último nodo y se vuelve a probar. Cuando se acaban los colores del último, se elije el próximo color del ante último y se vuelve a empezar de cero con los colores del siguiente y así hasta que se prueben todos los colores de todos los nodos. De esta forma siempre se llega a un caso base que es resoluble por 2-List Coloring.

Para evitar repetir colores se utilizan dos iteradores, el primero (*it*) siempre arranca desde el primer color de la lista y el segundo (*it2*) arranca desde el índice siguiente al de *it*. De esta forma se generan todas las permutaciones de dos colores sin repeticiones ni omisiones.

2.2.1. Podas

- Una forma de poda sería no llamar a 2-List Coloring con un nodo que tenga los colores $\{0, 1\}$ y luego volverlo a llamar con el mismo nodo con los colores $\{1, 0\}$ por ese motivo se recorren las listas de colores de la forma que explicamos antes.

- También se evitan varios chequeos al llamar directamente a 2-List Coloring si el nodo original tiene un solo color.

- Si se llega a una solución, se corta el backtracking y se devuelve esa. De esta forma en el caso promedio y mejor, no se recorrería todo el árbol de soluciones ya que no necesita encontrar todas las posibles.

2.3. Código fuente

A continuación se incluyen las partes más relevantes del código.
La clase *Main.java* se encarga de hacer el backtracking para recorrer todos los nodos fijando de a dos colores:

```
1    private static boolean listColoring(int id){
2       if (id == cantNodos) {
3          //llamo a 2listColoring
4          solucion = new Calculador_de_Coloracion_Ej1(cantNodos, grafo2colores).obtener_resolucion();
5          return !solucion.equals("X");
6       }
7
8       Nodo_Coloreable_ej2 nodo = grafo.get(id);
9       ListIterator<Integer> it = nodo.colores.listIterator();
10      ColoresPosibles coloresSeleccionados = grafo2colores.get(id).colores;
11      id++; //aumento el id para llamar al proximo nodo.
12
```

```
        if(nodo.cantidad_colores < 2){
            //si hay un solo color directamente llamo a listColoring.
            coloresSeleccionados.set_color(0, it.next());
            return listColoring(id);
        }else{
            while(it.hasNext()){
                int color1 = it.next();
                ListIterator<Integer> it2 = nodo.colores.listIterator(it.nextIndex());
                //Agrego color1 a la lista de colores
                coloresSeleccionados.set_color(0, color1);
                while(it2.hasNext()){
                    int color2 = it2.next();
                    //Agrego color2
                    coloresSeleccionados.set_color(1, color2);
                    if(listColoring(id)){
                        return true;
                    }
                }
            }
        }
        return false;
    }
```

La clase *Lector.java* se encarga de leer el input y transformarlo en dos grafos, uno que contiene todos los colores y el otro que tiene un máximo de dos colores para poder pasarselo como parámetro a *2ListColoring*.

```
    public void inicializar_lector() throws IOException{
        String parametros = this.leer_palabra();
        String [] parametros_procesados = parametros.split("␣");
        this.cantidad_nodos = Integer.parseInt(parametros_procesados[0]);
        this.cantidad_aristas = Integer.parseInt(parametros_procesados[1]);

        this.nodos_del_grafo = new ArrayList<Nodo_Coloreable_ej2>(cantidad_nodos);
        this.grafo_2colores = new ArrayList<Nodo_Coloreable>(cantidad_nodos);

        for(int i = 0; i < this.cantidad_nodos; i++){
            String nodo_string = this.leer_palabra();
            String [] nodo_string_procesado = nodo_string.split("␣");

            //creo nodo con todos los colores
            int cantidad_colores_nodo = Integer.parseInt(nodo_string_procesado[0]);
            Nodo_Coloreable_ej2 nuevo = new Nodo_Coloreable_ej2(i, cantidad_colores_nodo);

            for(int j = 1; j <= nuevo.cantidad_colores; j++){
                nuevo.colores.add(Integer.parseInt(nodo_string_procesado[j]));
            }
            this.nodos_del_grafo.add(nuevo);

            //creo nodo de 2 colores
            Nodo_Coloreable nodo2color = new Nodo_Coloreable(i);
            nodo2color.cantidad_colores = 1;
            nodo2color.colores.agregar_color(nuevo.colores.get(0));
            if (cantidad_colores_nodo > 1) {
                nodo2color.cantidad_colores = 2;
                nodo2color.colores.agregar_color(nuevo.colores.get(1));
            }
            this.grafo_2colores.add(nodo2color);
        }
        for(int i = 0; i < this.cantidad_aristas; i++){
            String arista_string = this.leer_palabra();
            String [] arista_string_procesada = arista_string.split("␣");
            int nodo_1 = Integer.parseInt(arista_string_procesada[0]);
            int nodo_2 = Integer.parseInt(arista_string_procesada[1]);
            Nodo_Coloreable n_1 = this.grafo_2colores.get(nodo_1);
            Nodo_Coloreable n_2 = this.grafo_2colores.get(nodo_2);
            n_1.adyacentes.add(n_2);
            n_2.adyacentes.add(n_1);
        }
    }
}
```

La clase *Nodo_Coloreable_ej2* es muy similar a la clase *Nodo_Coloreable_ej1*, lo único que cambia es que esta acepta una lista de mas de dos colores.

```
    public Nodo_Coloreable_ej2(int id, int cantColores){
        this.identidad = id;
        this.cantidad_colores = cantColores;
```

```
4         this.colores = new ArrayList<Integer>(cantColores);
5     }
```

3. Problema 3: Heurística Constructiva Golosa

3.1. 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.)

3.2. 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.

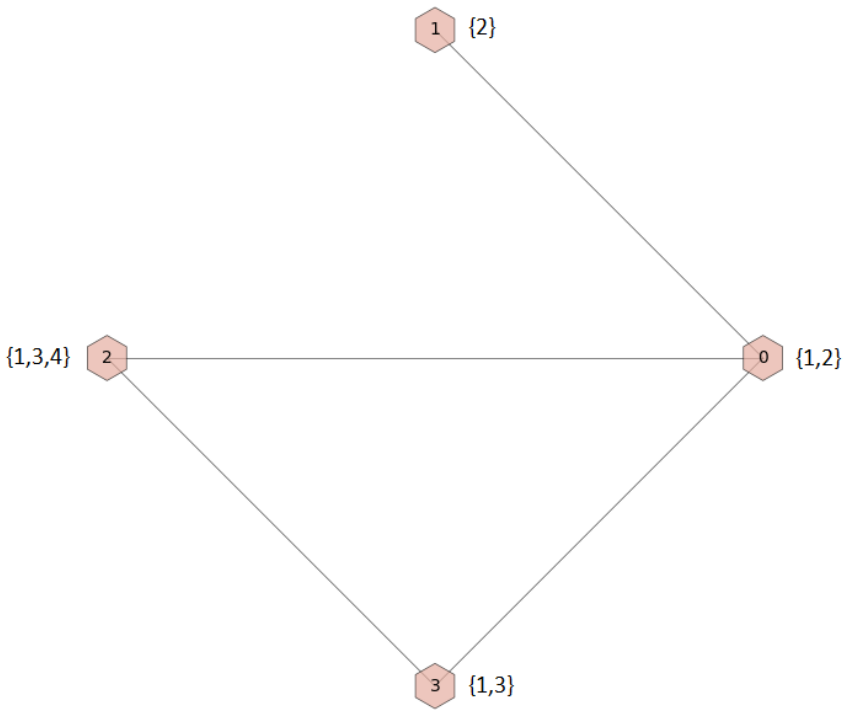

Figura 1: Grafo sin colorear

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)

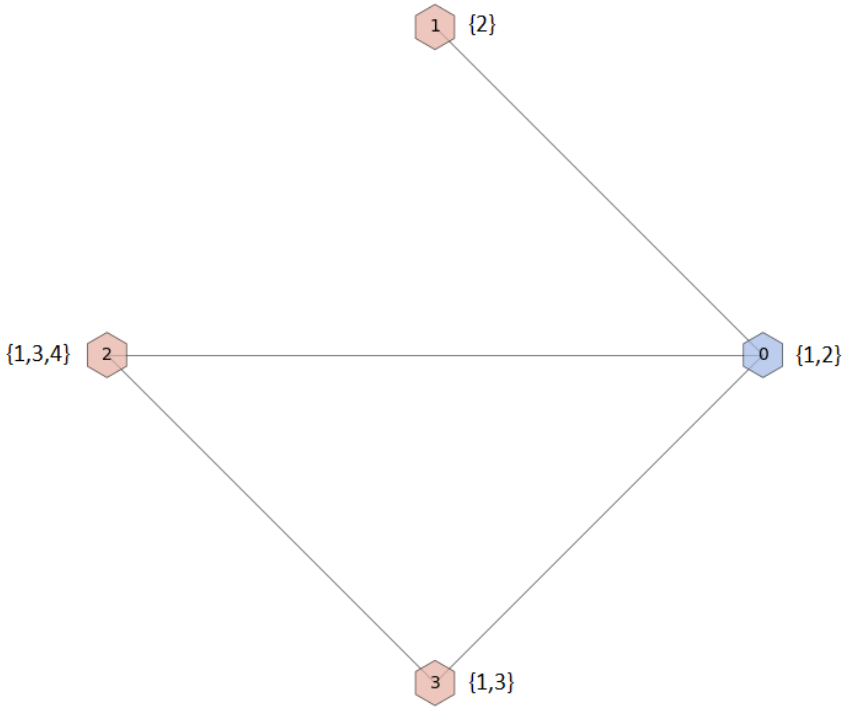

Figura 2: Al nodo 0 se le asigna el color 1

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.

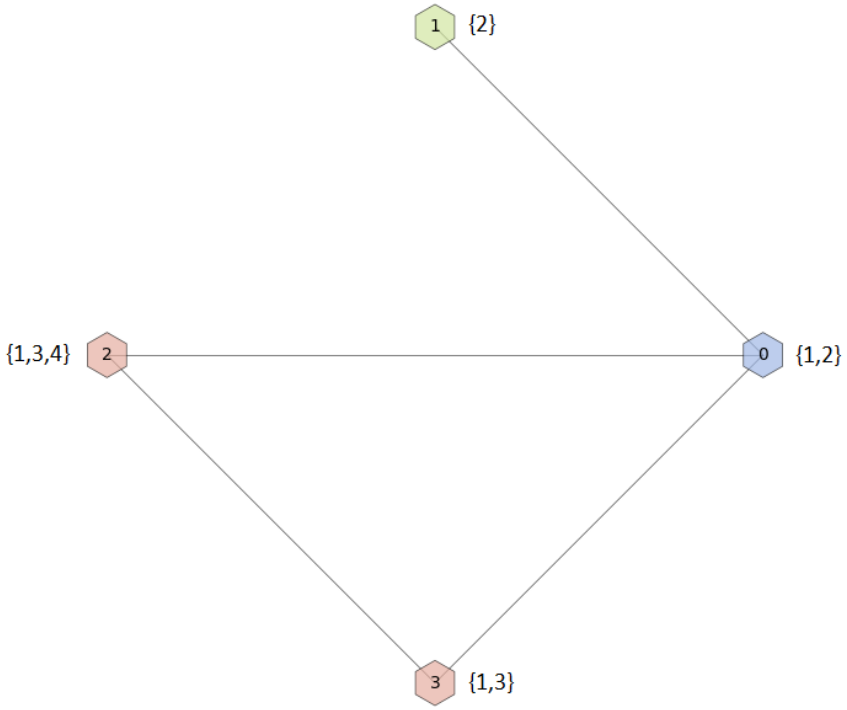

Figura 3: Al nodo 1 se le asigna el color 2

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)

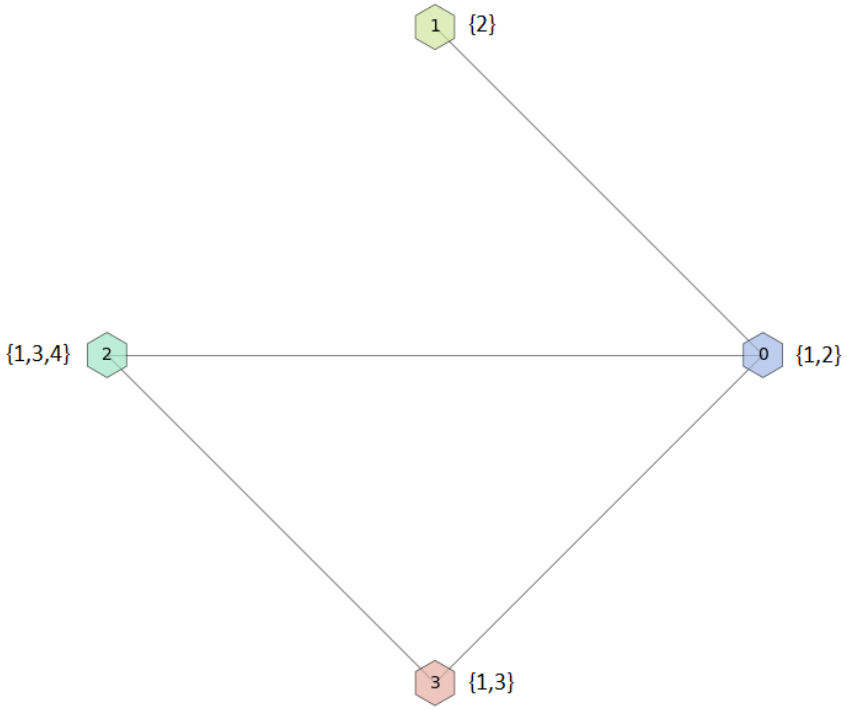

Figura 4: Al nodo 2 se le asigna el color 4

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

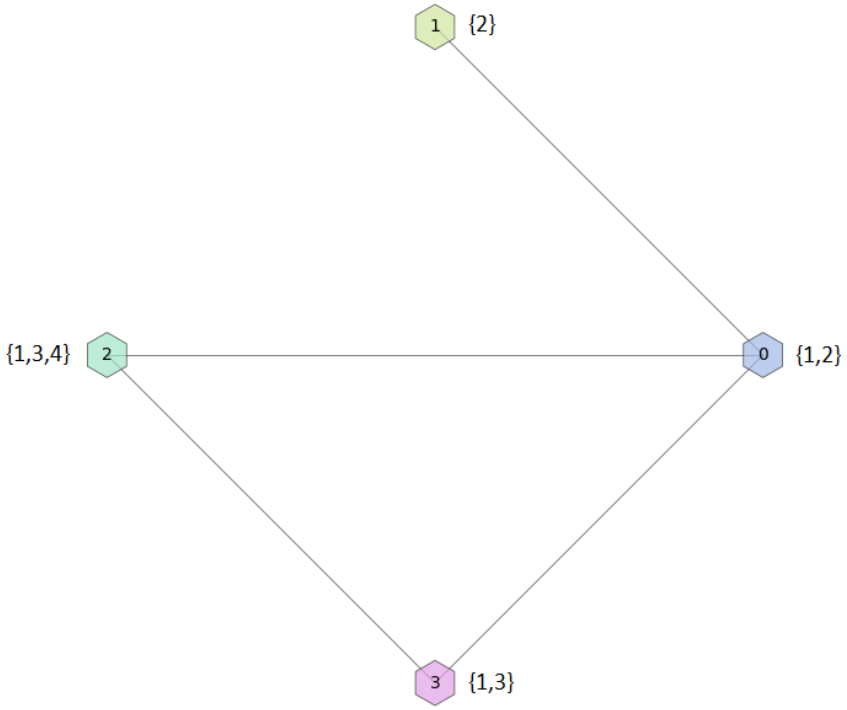

Figura 5: Al nodo 3 se le asigna el color 3

3.3. 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  }
```

3.4. 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 (el externo) tiene razón de ser porque dentro del bucle anidado sólo se colorea una componente conexa por el grafo que recibimos puede ser no conexo, por lo tanto debemos iterar sobre las distintas componentes. 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 V,grafo G)*). Mediante el mismo, se calcula el peso de cada color posible C de V mediante la verificación de la existencia de C entre el conjunto de los colores posibles de cada uno de sus nodos adyacentes a V. 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()*: n veces O(n*c*log(n)) . Esto hace que la complejidad final del algoritmo ascienda a $O(n^2*c*\log(n))$.

3.5. Experimentación

Nos interesa analizar cómo se comporta nuestro algoritmo en función de ciertas características del grafo a ser coloreado: su cantidad de nodos, su cantidad de aristas y el número máximo de colores que puede llegar a tener como opción cada uno de sus nodos.
Es por ello que nuestro primer experimento consistió en evaluar el tiempo necesario para el cómputo de distintos grafos en los cuales se incrementó paulatinamente la cantidad de nodos. Los resultados se pueden observar en la figura 6

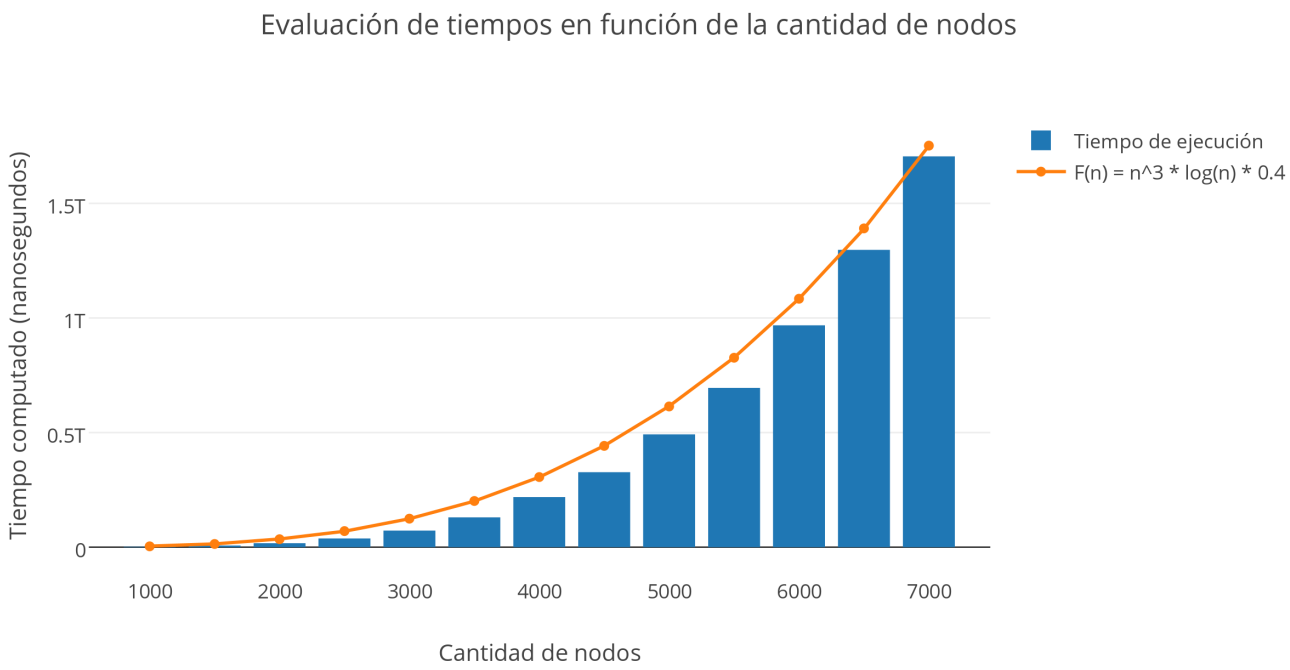

Figura 6: Tiempos de cómputo en función de la cantidad de nodos

Como se puede observar, la función que permite acotar los resultados obtenidos se aproxima asintóticamente a los calculados previamente en forma teórica para estas instancias.
Por otro lado, generamos grafos que diferían entre sí únicamente en su cantidad de aristas. Al medir los tiempos obtuvimos los resultados que se exponen en la figura 7

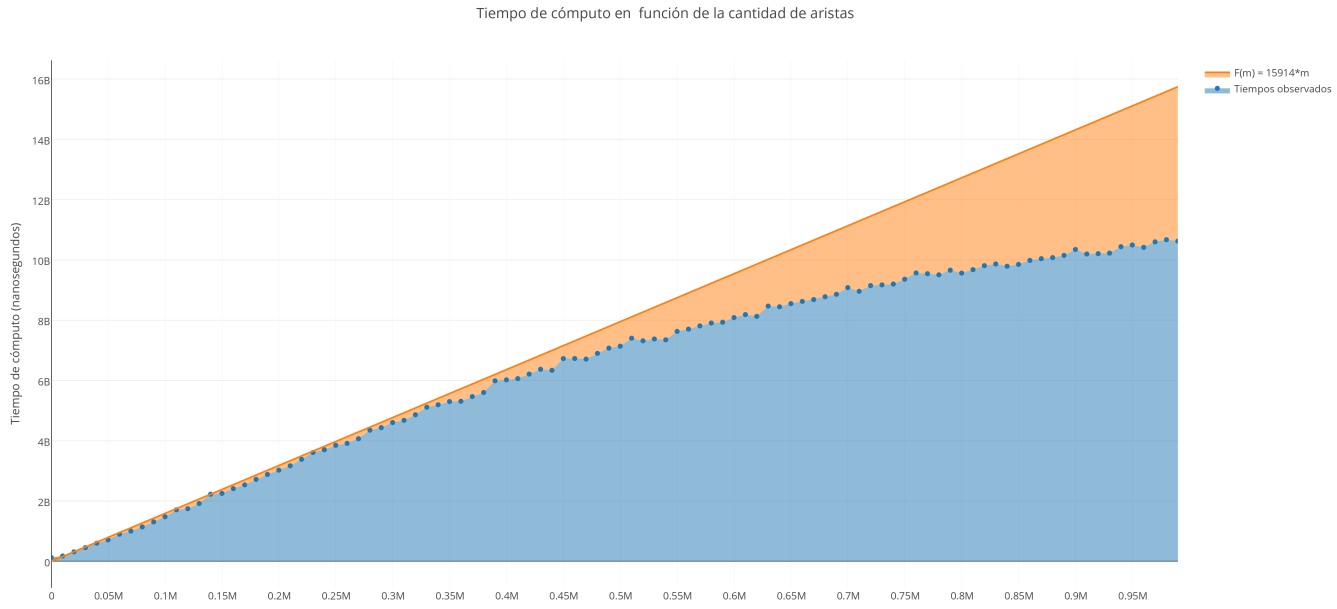

Figura 7: Tiempos de cómputo en función de la cantidad de aristas

Así mismo, realizamos una serie de experimentos en los que modificamos para cada grafo la cantidad

máxima de colores que cada uno de ellos podía tener. Con los resultados obtenidos se generó la ilustración 8

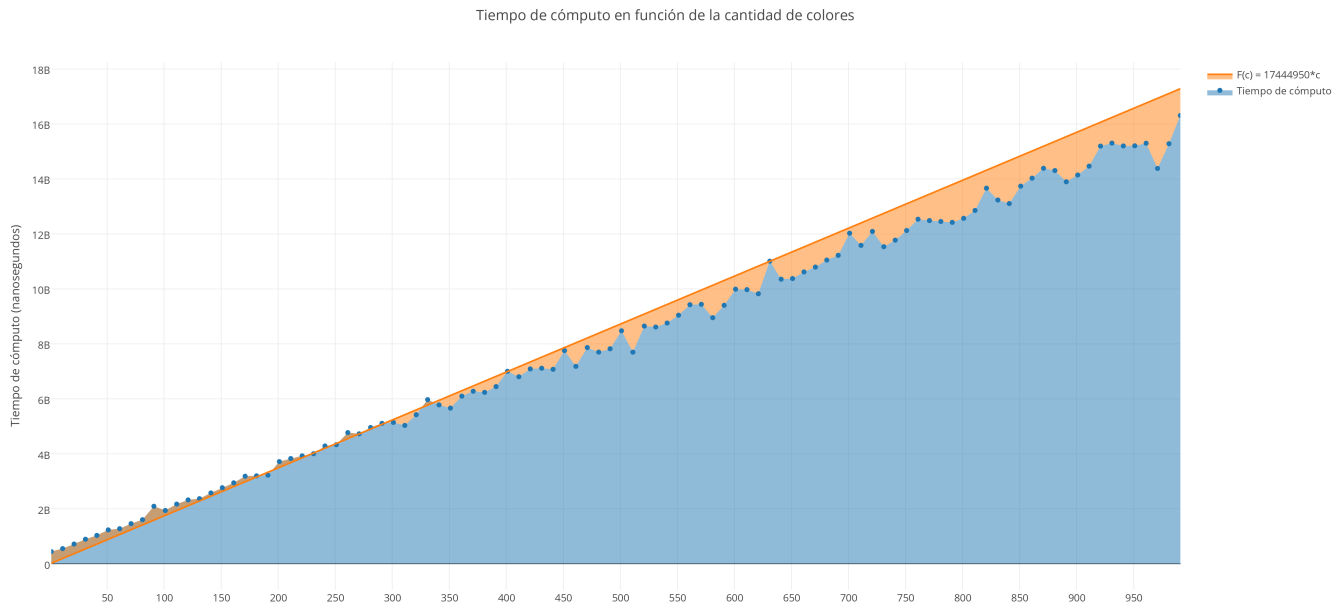

Figura 8: Tiempos de cómputo en función de la cantidad de colores

La misma parece indicar que el tiempo de cómputo dependiera linealmente de la cantidad de colores máxima del grafo. Para entenderlo, hay que considerar que la elección de un color para cada vértice implica haber recorrido cada uno de sus colores. Es decir, es necesario hacer N veces C (a lo sumo). De esta forma, al incrementar la cantidad de colores se realiza una iteración más costosa por cada nodo que no deja de ser lineal.

Por último analizamos qué ocurre con la cantidad de conflictos presentes luego del coloreo en función de la cantidad de colores disponibles para realizarlo. Tal como esperábamos a medida que se incrementa el módulo del conjunto de colores, disminuye la cantidad de conflictos.
En este caso los resultados resultan intuitivos, puesto que si se dispone de un único color entonces todos aquellos nodos que tengan adyacentes generarán conflictos, puesto que todos ellos tienen una única opción para colorearse. En contrapartida, sabemos que cualquier grafo requiere para ser coloreado a lo sumo tantos colores como cantidad de vértices contenga. En un punto intermedio, como cada nodo tiene potencialmente una cantidad de restricciones intermedia, también la cantidad de conflictos se espera que sea media (es decir, si dos nodos adyacentes comparten colores, siempre existe la posibilidad de que elijan uno distinto cada uno)
Los resultados se pueden observar en la imagen 9

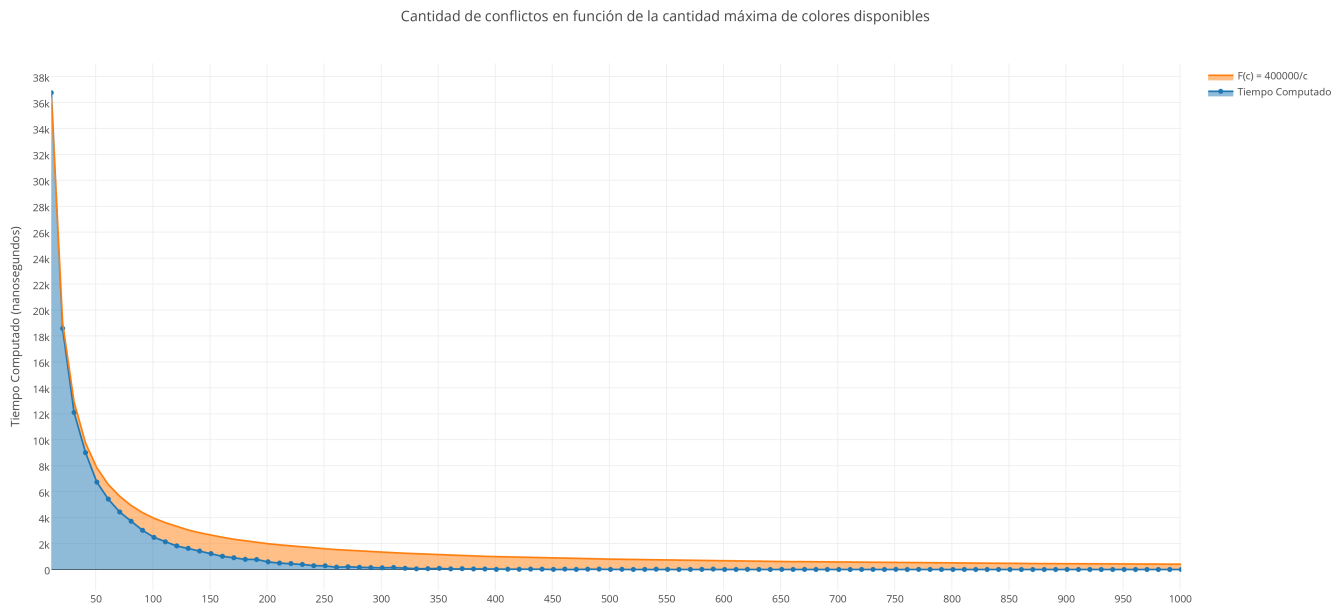

Figura 9: Total de conflictos en función de la cantidad de colores máxima

3.5.1. Peor caso

Considerando la heurística propuesta, el peor caso a considerar es aquel en que todos los nodos tienen n - 1 nodos adyacentes (es decir un grafo completo). En esta situación el método PintarNodo() - que se encarga de decidir qué color de los disponibles parecería el apropiado para cada vértice - cumple la cota teórica propuesta, pues debe hacer tantas operaciones como cantidad de vecinos del nodo que se examina.

Además se debe considerar la relación entre la cantidad de colores y la de nodos, puesto que al estar la complejidad fuertemente influida por ambos valores, el que determine finalmente los tiempos de ejecución en el peor caso será aquel cuya cota inferior sea la mayor entre ambas. En otras palabras, si el valor de C fuese equivalente en cada caso a N^N (un caso irrisorio - dado que cualquier grafo necesita a lo sumo N colores para colorearse - pero posible) entonces la cota superior estaría determinada en última instancia por el valor de C, absorbiendo asintóticamente la complejidad aportada por N.
Por el contrario, si C fuese menor que la cantidad de vértices entonces la cota superior de la heurística propuesta pertenecería a O(N^3 * log (N))

3.5.2. Mejor caso

Por lo antedicho, el mejor caso se constituye tomando un grafo donde todos sus vértices son aislados. De esta manera, todos y cada uno de ellos podría ser coloreado indistintamente con cualquiera de sus colores disponibles. Sin embargo el conjunto de colores seguiría siendo iterado en su totalidad.