Universidad Nacional de Córdoba

Laboratorio Discreta 2016.

Matemática Discreta II

Marro Santiago, Barraco Ramiro

Profesor:
Daniel Penazzi
Fecha de entrega:
27 de septiembre de 2016

Contents

1 Introducción

Realizamos un programa que afronta el problema de encontrar el número cromático de un grafo determinado. Como sabemos por lo visto en la cátedra, no existe algoritmo alguno que resuelva esto en orden polinomial, al menos cuando el grafo no es bipartito ($X(G) > 2$) .

Para ello utilizamos el algoritmo Greedy como principal herramienta de coloreo, combinándolo con una serie aleatoria de ordenaciones de los vértices. Ya establecimos en la cátedra que el algoritmo de Greedy arroja distintos resultados, según el orden de los vértices que se van coloreando y, también, si se ordena de acuerdo a sus colores, el algoritmo de Greedy no puede devolver resultados mayores a la iteración anterior. Aprovechándonos de este hecho, resolvimos un programa donde se ejecuta Greedy más de 1000 veces a partir de un "mejor" coloreo previamente logrado, guardando siempre el menor coloreo como resultado. Todo este proceso será explicado luego en la especificación del main.

Siguiendo la consigna, el proyecto está separado en dos directorios llamados apifiles y dirmain, donde el main se encuentra en dirmain y los archivos fuente en apifiles.

Para compilar el mismo se debe ejecutar lo siguiente:

- gcc -Wall -Wextra -O3 -std=c99 -Iapifiles dirmain/mainBarracoMarro.c apifiles/*.c -o BM

1.1 Funcionamiento

Nuestro programa se comporta de forma tal que, para grafos muy grandes, no demora más de 40 minutos en tiempo. En cuanto a consumo de memoria, solo el grafo de casi 2 millones de vertices logra superar los 300mb, llegando a unos 302mb utilizados a lo largo de todo el programa. El resto de los grafos testeados presentaron un consumo de memoria ampliamente inferior.

A continuación le dejaremos una tabla a modo de guía para los grafos testeados en nuestra computadora.

Grafo	Segundos	KBytes Utilizados	Colores
Gf12345_12111222_457_15	15.71	258804	49
bxb777_999_12	659.75	287936	4
R1999999_10123123_1	957.40	303264	7
myciel5	0.00	1628	6
inithxi2	0.27	1948	31
miles1000	0.14	1676	42
R22_99_15	0.00	1624	7
fpsol2i1	0.26	1720	65
david	0.01	1632	11
myciel4	0.00	1564	5
fpsol2i3	0.17	1816	30
fpsol2i2	0.17	1768	30
miles1500	0.17	1776	73
bxb15_22_10	0.06	1712	4
huck	0.01	1588	11
anna	0.02	1592	11
myciel7	0.06	1648	8
inithxi1	0.46	1960	54
q13	0.00	1316	54
R1789_875431_10	12.48	17700	587

Side note: La computadora utlizada para testear los grafos posee las siguientes caracteristicas:
Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 8gb RAM.

2 Especificación

Explicaremos el programa en diferentes etapas:

- La carga del grafo.
- El ordenamiento de sus vértices.
- Coloreo usando Greedy.

Primero vamos a explicar las estructuras y los archivos utilizados en el programa.

2.1 VerticeSt

```
struct VerticeSt {
    u32 nombreV;
    u32 gradoV;
    u32 colorV;
    u32 capacidad;
    VerticeP *vecinos;
};
```

nombreV : Se guarda el nombre del vértice.
gradoV: Se guarda el grado del vértice.
colorV : Se guarda el color del vértice.
capacidad : Cantidad de vecinos que puede tener.
vecinos : Puntero a un array, donde se encuentran los vecinos del vértice.

2.2 NihmeSt

```
struct NimheSt {
    u32 cantVertices;
    u32 cantLados;
    u32 cantcolor;
    struct VerticeSt *vertices;
    VerticeP *orden;
    bool *vertices_usados;
    VerticeP *orden_natural;
};
```

cantVertices : Cantidad de vértices del grafo.
cantLados : Cantidad de lados en el grafo
cantcolor: Cantidad de colores en el grafo.
vertices: Puntero a un arreglo de vértices.
orden: Arreglo con el orden de los vértices.
vertices_usados: Arreglo para saber si ya creamos el vértice o no.
orden_natural: Arreglo con los vértices ordenados en su orden natural.

2.3 Infraestructura del proyecto

2.3.1 Apifiles

Contiene todos los archivos .c y sus respectivos .h. En ellos están definidas las estructuras y las funciones de ordenamiento y Greedy. Entre ellos se encuentran :

- Cthulhu.h : incluye todos los archivos, declara constantes globales y las estructuras.
- Nimhe.c y Nimhe.h: funciones para crear y cargar grafos, como destruirlos o agregar vertices. Aparte de todas las funciones necesarias para sacar datos del grafo.
- Greedy.c : Funcion de Greedy y Chidos, como las funciones para manejar la cola del Chidos.
- Sort.c : Todas las funciones de ordenamiento y sus funciones auxiliares.
- vSt.c : Estructura de vértices y sus respectivas funciones.

2.3.2 Dirmain

contiene el main del programa.

- mainBarracoMarro.c: main del program llama a Cthulhu.h y llama a las funciones de : NuevoNimhe(), los órdenes, etc.

3 las 3 etapas del programa:

3.1 *La carga del grafo.*

El formato de entrada es una variación de DIMACS. Primero para cargar el grafo se utiliza la función, *NuevoNihme()*. Lee el input descartando la lineas que empiecen con 'c', los comentarios. Hasta que se ingrese una línea que comienza con 'p'. Allí sabemos la cantidad de vértices y lados que va a tener el grafo. En ese momento se reserva la memoria necesaria para guardar el grafo completo. A continuación se observan las líneas que tienen al principio 'e'. Para luego agregar los lados llamando a *AgregarLado()*, se revisa si los vértices están creados o no. Si están creados entonces se agregan como vecinos entre sí con la función *AgregarVecino()*. Si no están agregados se los crea con la función *NuevoVertice()* y luego se los vuelve vecinos.

La función *AgregarLado()* realiza la siguiente tarea. Se le asigna un id provisorio al vértice y a partir del mismo se busca si el vértice ya existe. Si no existe se obtiene el id definitivo y luego se lo guarda en el arreglo *vertices* en la posición del id. En ambos casos se consigue el puntero al vértice y se agregan como vecinos entre sí.

La función *AgregarVecino()* chequea si tiene capacidad para agregar un vecino a un vértice dado. Si ya esta lleno pide mas memoria y luego lo agrega al arreglo vecinos que es parte del vértice. Si no solo lo agrega.

3.2 *El ordenamiento de sus vértices.*

Parte fundamental de los algoritmos de ordenación es la función ***qsort()***, así que le daremos una breve explicación a continuación.

La función *qsort()* toma la información de un arreglo de elementos, su tamaño y cantidad como también una función para comparar cada uno. Si la función que se usa para comparar devuelve 0 no se hace nada, pues lo dos elementos son iguales. Pero si la función devuelve 1 o -1 entonces *qsort()* lo mueve a la derecha o izquierda correspondientemente.

A continuación explicaremos cada uno de los órdenes implementados:

3.2.1 *OrdenNatural()*

Usamos la función *qsort()* con la comparación *CrecienteCompNombre* la cual compara los vértices por su nombre de mayor a menor. Copiamos el orden en el arreglo del grafo que se llama "orden_natural".

3.2.2 *OrdenWelshPowell()*

Parecido con el orden anterior utilizamos la función *qsort()*, esta vez usando la comparación *CompWelshPowell()*. Que compara los grados y pone los vértices de menor grado primero.

3.2.3 *ReordenAleatorioRestringido()*

Esta vez usamos una variable global para guardar un número aleatorio, que luego utilizamos en la función de comparación que usamos en *qsort()*. Para

generar nuestro número aleatorio usamos la función *rand()*. De esta forma ordenamos los vértices eligiendo un color al azar. Hacemos esto múltiples veces para que todos los vértices queden ordenados en grupos de colores elegidos al azar.

3.2.4 *GrandeChico()*

Ordena los vértices poniendo primero los vértices cuyo color sea el más usado a menos usado. Utilizando la función *qsort()* con la comparación *CompGrandeChico()*.

3.2.5 *ChicoGrande()*

Ordena los vértices poniendo primero los vértices cuyo color sea el menos usado a más usado. Utilizando la función *qsort()* con la comparación *CompGrandeChico()*.

3.2.6 *Revierte()*

Como dice el nombre revierte el orden de los vértices. También se utiliza la función *qsort()* y la función *DecreCompColores()*.

3.2.7 *OrdenEspecifico()*

La función toma un arreglo de elementos del mismo tamaño que el grafo. Luego ordena los vértices en su orden natural con *OrdenNatural* si el elemento *orden_natural* está vació. Luego hace un loop para chequear que el i ésimo elemento no sea mas grande que el tamaño del arreglo, también se fija que no esté repetido. Si no hay problemas, guarda en la posición i ésima del arreglo *orden*, lo que se encuentra en el arreglo *orden_natural* en la posición k-ésima(siendo la posición k-ésima el número que corresponde a la i ésima posición de la copia del arreglo).

3.3 *Main y el coloreo de los vértices.*

El programa comienza cargando el grafo, utilizando la función *NuevoNihme()*, la cual realiza el proceso previamente explicado en la sección *3.1.* Luego de cargar correctamente el mismo, se revisa si es bipartito corriendo la función *Chidos*, algoritmo el cual hemos visto y desarrollado en la cátedra, de orden polinomial. Una vez comprobado esto, si el grafo es bipartito, se termina el programa imprimiendo en standard output "Grafo Bipartito". En el caso contrario, se procede a ordenar los vértices del grafo de forma aleatoria y correr el algoritmo de *Greedy()* 10 veces seguidas. Estas ordenaciones son generadas utilizando un array *x*, de largo *n*, el cual tiene números aleatorios del *1* al *-1*. Este array es utilizado como plantilla de ordenación para la función *OrdenEspecifico*, la cual fue explicada anteriormente.

A medida que se realizan las 10 ordenaciones aleatorias, con sus respectivas corridas de *Greedy*, se va guardando el mejor coloreo obtenido (es decir el menor). Para esto decidimos ir guardando el ordenamiento de esa mejor corrida en un array llamado *mejor_orden*. Luego de las 10 corridas, se realiza un más, utilizando el orden *WelshPowell* Una vez completadas las 11 corridas de *Greedy*, se revisa si el mejor coloreo obtenido es 3, caso donde se imprime en standard output *"X(G)=3"* y se termina el programa.
Si el mejor coloreo es mayor a 3, se procede a ordenar el grafo nuevamente, utilizando *mejor_orden* como plantilla de ordenación para *OrdenEspecifico()*. Luego, se comienzan 1001 iteraciones donde, en cada una, se elige un orden determinado con el cual ordenar el grafo, siguiendo la *Table 1*; para luego correr nuevamente *Greedy*. Por cada iteración se va guardando el menor coloreo en la variable *alduin*, el cual será finalmente la solución del programa.

Al terminar las 1001 iteraciones, se devuelve el mejor coloreo y la cantidad de veces que se usó cada orden.

Table 1: Uso de ordenes

ChicoGrande()	50%
GrandeChico()	12,5%
Revierte()	31,25%
ReordenAleatorioRestringido()	6,25%

4 Preguntas Puntuales.

4.1 ¿Como hicieron para resolver el problema de que los vértices pueden ser cualquier u32?

Para resolver este problema decidimos utilizar un array donde guardamos los vértices a medida que se va leyendo el grafo. La primera complicación que tuvimos fue el hecho de saber si un vértice que estábamos leyendo ya lo habíamos leído antes. Recorrer el array no era una opción debido a su complejidad.

Para ello resolvimos utilizar un array de booleanos donde su índice indica el *hash* del nombre del vértice. Si el array en esa posición está seteado en true, quiere decir que el vértice ya fue creado y no se lo crea.

El *hash* de cada vértice se obtiene de la siguiente forma:

1. Se toma el nombre del vértice a guardar y se le aplica la función módulo con respecto a la cantidad de vértices que tiene el grafo.

2. Se revisa en el array *vertices_usados*, parte de la estructura del grafo, en la posición *hash*. Si se encuentra en true, quiere decir que esa posición ya fue tomada por algún vértice, donde lo más probable es que sea el mismo que se quiere agregar. Por lo tanto se revisa si ese vértice posee el mismo nombre al que queremos agregar. Si es así, no se crea un vértice nuevo, ya que ya lo habíamos leído previamente. Si no es el mismo, se procede a sumarle 1 a hash y se realiza el mismo procedimiento. Esto continúa hasta que se encuentre una posición en *vertices_usados* donde sea false ó se encuentre el mismo vértice a agregar.

3. En el caso de que se encuentre una posición en *vertices_usados* donde sea false su valor, se debe crear el vértice, y luego setear esa posición como true.

4.2 ¿Cómo implementaron Gamma(x)?

Resolvimos este problema agregando a cada vértice x, una lista de sus vecinos. Es decir, si tengo el *VerticeSt* x, para ver sus vecinos solo debo hacer: $iesimoVecino = x.vecinos[i]$. Donde i es el i ésimo vecino.

4.3 ¿Cómo implementaron el orden de los vértices?

Simplemente guardamos en la estructura NimheSt un arreglo llamado *orden* de punteros a VerticeSt, en determinado orden. Estos punteros *VerticeP* apuntan a vértices VerticeSt guardados en el array *vertices*, también en la estructura del grafo NimheSt. Cuando el programa recién termina de cargar el grafo, el orden de los vértices en *vertices* y *orden* es el mismo. Luego cuando se quiere ordenar solo se cambia *orden*.

Side note: Para facilitar la función *OrdenEspecifico* también tenemos un array llamado *orden_natural* donde guardamos el orden natural según como lo especifica la consigna.

4.4 ¿Cómo implementaron Greedy?

Para implementar Greedy decidimos utilizar las siguientes variables:

```
u32 n = G->cantVertices; // Cantidad de vertices del grafo
u32 color;               // Variable para guardar color actual del
    vecino
u32 max_color = 0;       //Variable para guardar la cantidad maxima
    de colores
VerticeP vertice = NULL; //Puntero para guardar vertice
bool usado[n+1];         // Array para indicar colores no
    disponibles. n+1 ya que el color 0 no se usa
u32 grado = 0;
```

Antes de arrancar con el coloreo, despintamos todos los vértices del grafo, recorriendo uno por uno y asignandole *0* a su *color*.

Luego obtenemos el primer vértice según, el orden determinado, y lo copiamos en la variable *vertice*. Con el mismo lo coloreamos con el primer color, *1*, y luego empezamos a colorear los siguientes vértices en el orden determinado.

Mientras se va copiando vertice a vertice, se guarda su grado en la variable *grado*, para luego recorrer todos sus vecinos.

Luego, a medida que vemos cada vecino, si está coloreado, asignamos en la posición del color, en el array de *bool usado*, como true. Esto nos servirá para poder elegir el color del *vertice*

Al terminar de recorrer todos los vecinos del *vertice* actual, se busca por el primer color no usado en el array *usado*. Dicho color será el que se le asigne al *vertice*. También vamos guardando en *max_color* el máximo color utilizado hasta el momento.

Para poder seguir con el algoritmo, se resetea el array *usado* y se repite con el proximo *vertice* en el orden especificado.

Para finalizar, se guarda la cantidad de colores utilizados en *G-¿cantcolor* y se devuelve como resultado la misma.

Analizando el algoritmo y su complejidad, vemos que se comporta en *O(n*m)*, donde n es la cantidad de vértices y m la cantidad de lados. Esto se debe a que, por cada vértice (n veces), se recorren todos su k vecinos, donde k, puede ser m.