Proyecto de Matemática Discreta II-2024 Parte 2

Contents

1 Introducción

1.1 Importante: desaprobación automática

Estas funciones estan pensadas para correr con cualquier implementacion de la primera parte que cumpla las especificaciones.

Por lo tanto las funciones definidas aca **deben** usar las funciones definidas en la parte 1 **PERO NO** la estructura interna del grafo.

Pej, si en algun momento necesitan acceder al grado del i-esimo vertice, y uds. usan en esta etapa algo como G->vertice[i].grado , estan automaticamente desaprobados.

Suponer que todo el mundo va a hacer la misma estructura que uds. incluyendo el nombre especifico de los campos es algo que deberia ser obvio que es una estupidez, y sin embargo algunos estudiantes lo suponen.

Ud. deben suponer que han sido contratados para codear estas funciones y todo lo que tienen son las especificaciones de las funciones de la primera parte pero no el código de las mismas, el cual ha sido asignado a otro equipo.

Obviamente para testear estas funciones van a tener que usar las funciones de la 1ra etapa que uds codearon. Se sugiere que ademas de usar sus funciones, traten de usar las funciones de la 1ra etapa de algún otro grupo como otra forma de verificar que no esten programando usando alguna parte de la estructura interna del grafo.

Nosotros testearemos con NUESTRAS funciones de la parte 1.

Estas funciones deben estar en uno o mas archivos .c cada uno de los cuales con un include de un archivo:

API2024Parte2.h

El archivo API2024Parte2.h es una declaracion de estas funciones, con un include de APIG24.h para poder usar las funciones de la parte 1.

2 Greedy()

Prototipo de función:

```
u32 Greedy(Grafo G,u32* Orden);
```

Esta función **asume** que Orden apunta a un sector de memoria con n lugares. Uds no necesitan verificar esto.

Pero, lo primero que debe hacer esta función es verificar que Orden provee efectivamente un orden de los elementos $\{0, 1, ..., n-1\}$. Es decir, que Orden induce una biyección en $\{0, 1, ..., n-1\}$.

Esta función colorea los vértices de G de la forma descripta abajo.

Retorna el número de colores que usa, salvo que haya algún error, (pej al alocar memoria temporal o si Orden no induce una biyección) en cuyo caso retorna $2^{32} - 1$.

Corre greedy en G comenzando con el color 1, iterando sobre los vértices siguiendo el orden dado en el array apuntado por Orden, asumiendo que Orden[i]=k significa que el vértice k será el vértice procesado en el i-esimo lugar.

Es decir, se procesaran los vértices en el orden dado por Orden[0],Orden[1],Orden[2], etc. hasta Orden[n-1] incluido.

Importante: como el nombre lo indica, esta funcion corre GREEDY. Si uds programan una función que da un coloreo propio pero que no es Greedy, no cuenta. Mucho peor, obviamente, si su función ni siquiera da un coloreo propio.

Esta función debe ser O(m).

3 Funciones para crear ordenes

Estas funciones asumen que el grafo G tiene un coloreo propio con colores $\{1, 2, .., r\}$ para algún r.

Tambien asumen que Orden apunta a una region de memoria con n lugares, donde n es el número de vertices de G.

Lo que hacen estas funciones es ordenar los vértices, llenando el array Orden en la forma indicada en cada uno de los casos.

Si todo anduvo bien devuelven el char 0, si no el char 1.

La forma de ordenar los vértices en ambos casos es ordenar vértices por bloques de colores, como en el VIT.

Es decir, ambas funciones lo que harán es ordenar los vértices poniendo primero todos los vértices que tengan un color x_1, luego todos los vértices que tengan un color x_2, etc.

La diferencia entre ambas funciones es quienes son los colores $x_1, x_2, ...$ etc. asi que en cada caso especificaremos esto.

Ambas funciones deberian ser, idealmente, O(n), pero O(nlog n) es aceptable.

3.1 GulDukat()

Prototipo de función:

```
char GulDukat(Grafo G,u32* Orden);
```

La forma de llenar Orden, como se dijo, es poner todos los vértices de color x_1 primero, luego los de color x_2, etc, donde $x_1, x_2, ...$ vienen dados de la siguiente manera:

asumiendo que r es la cantidad de colores usados para colorear G, definimos las siguientes funciones de $\{1, ..., r\}$ a $\mathbb{Z}$, usando las funciones de la parte 1:

$$m(x) = \min\{Grado(i, G) : Color(i, G) = x\}$$
$$M(x) = \max\{Grado(i, G) : Color(i, G) = x\}$$

Entonces debe poner primero todos los colores que sean divisibles por 4, ordenados entre si de acuerdo con $M(x)$ (de mayor a menor), luego todos los colores pares no divisibles por 4, ordenados entre si de acuerdo con $M(x) + m(x)$ (de mayor a menor) y finalmente todos los colores impares, ordenados entre si de acuerdo con $m(x)$. (de mayor a menor).

(motivación para esta función: vertices con alto grado en general tienen mayores problemas para ser coloreados. Asi que poner primero los vertices de mayor grado deberia dar un buen coloreo....pero eso no necesariamente será un ordenamiento por bloque de colores. Asi que debemos ordenar los colores, de acuerdo con los grados de los vertices de ese color. Podriamos tomar el promedio de los grados, o el mayor grado, o el menor grado...aca hicimos una mezcla entre el mayor y el menor grado, con un test de divisibilidad para que sea facil chequear la funcion en el caso de un grafo regular por ejemplo y ademas para mezclar los colores).

3.2 ElimGarak()

Prototipo de función:

```
char ElimGarak(Grafo G,u32* Orden);
```

La forma de llenar Orden como se dijo, es poner todos los vértices de color x_1 primero, luego los de color x_2, etc, donde $x_1, x_2, ...$ vienen dados de la siguiente manera, asumiendo que hay r colores:

$x_r = 1$.

$x_{r-1} = 2$.

x_1 es el color que tiene la menor cantidad de vértices de ese color, excluyendo a los colores 1 y 2.

x_2 es el color que tiene la menor cantidad de vértices de ese color, excluyendo a los colores 1,2 y x_1.

etc.

(motivación para esta función: poniendo los colores 1 y 2 al ultimo, nos aseguramos que los vertices que menos problemas tuvieron para ser coloreados sean coloreados al final. Y los demas colores los ordenamos por cardinalidad, asumiendo que los colores que menos vertices tienen son los mas problematicos, y poniendolos al principio quizas podamos bajar la cantidad de colores).

4 Main

Tambien deben hacer uno o mas mains para testear las funciones. No deben entregarlos excepto por uno solo, como para que veamos que al menos hicieron alguna integracion de las funciones.

El main que pediremos es simple, uds. pueden hacer otros mas complicados si quieren.

El main que entreguen debe empezar corriendo Greedy en 5 ordenes distintos, (especificados mas abajo) imprimiendo la cantidad de colores que se obtiene, y para cada uno de esos ordenes iniciales, luego se hace una iteracion en donde hacen lo siguiente:

1. corren GulDukat y a continuacion Greedy con ese orden
2. Luego corren ElimGarak y Greedy con ese orden.

e iteran estos dos pasos un total de 50 veces, imprimiendo la cantidad de colores obtenidos luego de cada Greedy. (es decir, que para cada uno de los ordenes iniciales, haran un total de 100 reordenamientos y coloreos con Greedy, con lo cual haran en total 500 reordenamientos mas Greedys, mas 5 Greedys iniciales).

Para cada una de esas iteraciones deben guardar la cantidad de colores que usaron en esa iteración, mas el ultimo coloreo que usaron (el cual deberan guardar en algun array apropiado, usando ExtraerColores)

Luego, comparando las 5 corridas, toman el coloreo que les dio la menor cantidad de colores, recolorean los vertices de G con ese coloreo (usando ImportarColores) y luego hacen 500 veces un reordenamiento GulDukat o ElimGarak seguido de un Greedy, impriendo la cantidad de colores obtenida.

El programa debe elegir al azar, con una probabilidad de 50%, cual de los dos ordenes se elije en cada una de las 500 iteraciones.

Al final de todo habran hecho 1000 reordenes por bloque de colores, mas sus correspondientes Greedys, mas algunos Greedys extras.

Los ordenes iniciales son: 1) el orden natural 0,1,...,n-1 de los vertices,
2) el orden n-1,...,2,1,0
3) el orden poniendo primero todos los pares en orden decreciente y luego todos los impares, en orden creciente.
4) ordenar los vértices de acuerdo con su grado, comenzando por el mayor grado y terminando con el menor grado.
y 5) algun orden extra, elegido por ustedes.

4.1 Velocidad

El código debe ser "razonablemente"rápido. Demoras de horas, dias o semanas no son razonables. Deberian poder hacer los 1000 Greedys pedidos junto con los reordenes de vertices en a lo sumo 5 minutos en una maquina razonable aun para los grafos grandes, pero aceptaremos tiempos de aprox. 15 minutos para no ser estrictos.

5 Cosas a prestar atención

5.1 Errores comunes

1. **Usar un algoritmo de ordenación $O(n^2)$ como Bubble Sort es desaprobación automática.**

2. Recordemos que por el VIT si se usa un orden que asegure que vértices del mismo color esten consecutivos en el orden, entonces la cantidad de colores no puede aumentar. Asi que si luego de usar las funciones de ordenamiento y correr Greedy el numero de colores AUMENTA respecto al que se tenia antes, entonces tienen un error. Si ese error viene de que tienen un error en las funciones de orden o en Greedy, es algo que van a tener que descubrir, pero al menos saben que tienen un error.

3. Una forma de testear que las funciones de ordenamiento hacen lo que se pide es, luego de usarlas pero antes de hacer Greedy, hacer un for en i imprimiendo los colores de los vértices en el orden dado y verificando que la salida es lo que se pide en cada caso.

4. Sabemos que Greedy no puede producir mas de $\Delta + 1$ colores, en ningun orden, asi que esto es algo que pueden testear para detectar algún error mayúsculo.

5. Testeen que Greedy de siempre coloreos propios. Por ejemplo, si colorean K_n con menos de n colores tienen un problema grave en algún lado. Como vimos en el teórico, testear que el coloreo es propio es $O(m)$ y lo pueden hacer con una función extra.

6. Un error mas sútil pero que ha ocurrido es que programen Greedy con un error tal que siempre da la misma cantidad de colores para un grafo dado, independientemente del orden de los vertices, o bien que pueda depender del orden, pero que una vez corrido Greedy para un orden inicial, todos los ordenes siguientes den la misma cantidad de colores.

   Hay grafos con los cuales greedy siempre dará la misma cantidad de colores, pej, los completos pero la mayoria no.

7. Eviten un stack overflow en grafos grandes. En general los estudiantes provocan stack overflows haciendo una recursion demasiado profunda, o bien declarando un array demasiado grande.

8. Buffer overflows o comportamiento indefinido. Se evaluará la gravedad de los mismos. Algunos son producto de un error muy sutil pero otros son mas o menos obvios.

9. Presten atención a los memory leak, especialmente si corren Greedy y no liberan memoria, pues Greedy se correrá muchas veces.

10. No usen variables shadows.