Proyecto de Matemática Discreta II-2023 Parte 3

Contents

1 Introducción

1.1 Importante: desaprobación automática

Estas funciones estan pensadas para correr con cualquier implementación de la primera parte que cumpla las especificaciones. Las testearemos usando NUESTRA(s) parte 1.

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 merece una desaprobacion.

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, pero NO DEBEN entregar su parte 1.

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.

1.2 que entregar

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

APIParte3.h

El archivo APIParte3.h debe ser una declaracion de estas funciones, con un include de APIG23.h para poder usar las funciones de la parte 1. Pueden agregar a su APIParte3.h cualquier otra declaracion de funcion auxiliar que necesiten o includes de librerias basicas de C que necesiten. Este archivo debe ser hecho por ustedes y entregado.

Pueden usar otros .h si quieren, los cuales deben ser entregados.

NO DEBEN ENTREGAR NINGUN EJECUTABLE, ni ningun archivo con un main, y SOLO deben entregar las funciones de la parte 3. Entrega de funciones de la parte 1 es desaprobación automática.

1.3 Fecha de entrega

La parte 3 debe ser entregada a mas tardar el dia del examen en el cual rinda el primer integrante del grupo que rinda un examen.

Aun si es entregada antes, la parte 3 NO SE CORRIJE hasta que no se presenta al menos un integrante del grupo.

Si la parte 3 no es aprobada, entonces todos los integrantes del grupo que se hayan presentado en esa fecha tienen desaprobado el examen final de esa fecha y el grupo deberá hacer una parte 4 del proyecto.

Si la parte 3 queda aprobada, queda aprobada para el resto de los examenes hasta marzo 2024 incluido.

1.4 Integrantes

Esta parte 3 debe ser hecha por alumnos que no entregaron la parte 2, o la entregaron y obtuvieron 1(uno) en el proyecto.

No es necesario que el grupo de entrega de la parte 3 sea el mismo que entregó la parte 2.

pej si un grupo tenia integrantes A,B,C ahora puede entregar A una parte 3 por su cuenta, y B,C otra parte 3 por su cuenta. O los 3 entregar partes distintas.

O puede ser que A haga grupo con D que no habia entregado o que formaba parte de otro grupo D,E,F que tambien obtuvo un 1.

Pero el requerimiento de que los grupos no puede tener mas de 3 integrantes sigue valiendo.

Si hay mas de un integrante, deben entregar la parte 3 una sola vez.

Quien entrega debe mandar el mail de entrega con copia a los demas integrantes del grupo, o aclarar que es un solo integrante.

2 GreedyDinamico()

Prototipo de función:

```
u32 GreedyDinamico(Grafo G,u32* Orden,u32* Color,u32 p);
```

Esta función **asume** que Orden es un array de n elementos que provee un orden de los indices, es decir, es una biyección de $\{0, 1, ..., n-1\}$.

Tambien asume que Color apunta a un sector de memoria con al menos n lugares disponibles.

Uds. no necesitan programar dentro de esta función una verificación de esto.

Esta función llena Color[] de la forma descripta abajo, escribiendo en el lugar i de Color[] cual es el color que esta función le asigna al vértice cuyo índice es i en el Orden Natural.

Retorna el número de colores que usa, salvo que haya algún error, en cuyo caso retorna $2^{32} - 1$.

Asumiendo que Orden[i]=k significa que el vértice cuyo índice es k en el Orden Natural será el vértice procesado en el lugar i, corre greedy en G comenzando con el color 0, iterando sobre los vértices siguiendo el orden dado en el array apuntado por Orden, PARA LOS PRIMEROS p lugares de Orden. (Si p=0, se trata como si fuese p=1).

Es decir, se procesaran los vértices en el orden de sus indices dado por Orden[0],Orden[1],Orden[2], etc. hasta Orden[p-1] incluido. (obviamente si p es mayor o igual que el numero de vertices de G, entonces esto ejecuta Greedy normal)

De ahi en adelante, si p<numero de vertices de G, los demas vertices se procesan dinámicamente: luego de los p primeros, para determinar cual será el siguiente vertice a colorear, se calcula para cada vertice que haya quedado sin colorear la siguiente función:

$$NC(v) = \text{numero de colores distintos de los vecinos ya coloreados de } v$$

Pej, si v tiene 7 vecinos coloreados, 3 de ellos de color 4, uno de color 5, 2 de color 11 y uno de color 23, entonces $NC(v) = 4$ pues hay 4 colores: 4,5,11,23 entre los vecinos.

Luego de calcular esto, el siguiente vértice a colorear es el vértice que tiene el MAYOR NC.

Si hay mas de un vertice con el mayor NC, entonces de entre todos ellos se elije el que este primero en Orden[].

Una vez seleccionado el vértice, la forma de colorearlo es la misma que la de Greedy, es decir, esto colorea exactamente igual que Greedy, solo que va determinando el orden en que colorea a medida que va corriendo, en vez de usar un orden fijo.

3 Funciones para crear ordenes

3.1 FirstOrder()

Prototipo de función:

```
char FirstOrder(Grafo G,u32* Orden,u32* Color);
```

La función asume que Color y Orden apuntan a una region de memoria con al menos n lugares, donde n es el numero de vertices de G, y que la imagen de Color[] es un conjunto $\{0, 1, ..., r-1\}$ para algun r.

Ordena indices llenando el array Orden en la forma indicada abajo.

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

La forma de llenar Orden es la siguiente: asumiendo que r es la cantidad de colores que aparecen en Color[], definimos las siguientes funciones de $\{0, 1, ..., r-1\}$ a $\mathbb{Z}$:

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

$$E(x) = \begin{cases} 0\text{x}7\text{fffffff} + m(x) & \text{si } x+2 \text{ es divisible por 3} \\ 0\text{x}1\text{fff} + M(x) & \text{si } x+1 \text{ es divisible por 3} \\ M(x) + m(x) & \text{si } x \text{ es divisible por 3} \end{cases}$$

Entonces se ponen primero los indices i tal que Color[i] sea igual al color x tal que $E(x)$ es el maximo de E, luego los indices i tal que Color[i] es el color x tal que $E(x)$ es el segundo mayor valor de E luego del maximo, etc.

3.2 SecondOrder()

Prototipo de función:

```
char SecondOrder(Grafo G,u32* Orden,u32* Color);
```

La función asume que Color y Orden apuntan a una region de memoria con al menos n lugares, donde n es el numero de vertices de G, y que la imagen de Color[] es un conjunto $\{0, 1, ..., r-1\}$ para algun r.

Ordena indices llenando el array Orden en la forma indicada abajo.

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

La forma de llenar Orden es la siguiente: asumiendo que r es la cantidad de colores que aparecen en Color[], definimos la funcion $S : \{0, 1, ..., r-1\} \mapsto \mathbb{Z}$ dada por:

$$S(x) = \left(\sum_{i:(Color[i]=x) \wedge Grado(i,G)>1} Grado(i, G) \right)$$

Entonces se ponen primero los indices i tal que Grado(i,G)>1 y Color[i] sea igual al color x tal que $S(x)$ es el maximo de S; luego los indices i tal que Grado(i,G)>1 y Color[i] es el color x tal que $S(x)$ es el segundo mayor valor de S luego del maximo, etc, y al final de todo se ponen todos los indices i con Grado(i,G)=1.

3.3 Velocidad

El código debe ser "razonablemente"rápido.

Las funciones de ordenación deben ser eficientes. Se pueden hacer en $O(n)$ pero $O(n \log n)$ es aceptable. $O(n^2)$ en las funciones de ordenación es desaprobación automática.

Funciones de ordenación que no ordenen tambien es desaprobación automática.

GreedyDinamico va a necesariamente ser mas lento que Greedy normal. Con algunos grafos muy grandes puede ser mucho mas lento. No daré una cota de tiempo como di en Greedy, pero el código debe ser razonable, tanto en el uso del tiempo como de la memoria. En particular, si p esta cerca de n, el código debería correr a una velocidad comparable con Greedy normal, y si $p = n$ entonces si deben valer las cotas de tiempo que di para Greedy en la parte 2.