Proyecto de Matemática Discreta II-2023 2da parte

Contents

1 Introducción

1.1 Importante: desaprobacion automatica

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 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 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.

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

APIParte2.h

El archivo APIParte2.h es una declaracion de estas funciones, con un include de APIG23.h para poder usar las funciones de la pagina 1.

1.2 Motivacion de las funciones

Recordemos que la idea general del proyecto es implementar "Greedy iterado", es decir, dado que Greedy en un cierto orden no garantiza encontrar el numero cromatico, pero iterar sobre todos los ordenes si, pero no podemos iterar sobre todos los ordenes, usar ciertas estrategias para explorar mejor el espacio de ordenes posibles.

En el teorico vimos que si se reordenan los vertices por bloques de colores, entonces esta garantizado que Greedy no va a empeorar la cantidad de colores.

Asi que una idea basica seria: correr Greedy con una cantidad modesta (10-20, pej) de diversos ordenes, (pej, el orden natural, el orden natural reverso, orden WelshPowell(por grados de mayor a menor) u ordenes aleatorios), quedarse con el mejor de ellos, y a partir de ahi usar ordenes que vengan de ordenar por bloque de colores dado que sabemos que no empeoraremos.

Entonces obviamente una función que necesitaremos será una función que corra Greedy en algun orden dado, y ademas necesitaremos funciones que generen esos diversos ordenes.

En otros años hemos pedido varias de estas funciones.

Hay diversos ordenes por bloque de colores que funcionan bien, como pej reordenar los colores aleatoriamente, u ordenar poniendo los vertices del mayor color primero, luego los del segundo mayor color, etc, (ordenar de menor a mayor NO funciona, como se ve en el practico 1), o bien ordenar poniendo primero los vertices cuyo color sea el color menos frecuente, luego el segundo menos frecuente, etc.

Varias de estas funciones las hemos pedido otros años asi que para dar algunas funciones no vistas anteriormente daremos dos que no hemos dado antes y tambien funcionan bien, pero uds pueden si asi lo desean programar otras para mejorar su programa, (sin entregarlas) aunque deberia funcionar bastante bien incluso solo con estas dos que pedimos.

2 Greedy()

Prototipo de función:

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

Corre greedy en G comenzando con el color 0, iterando sobre los vértices siguiendo el orden dado en el array apuntado por Orden, asumiendo que Orden[i]=k significa que el vértice cuyo índice es k en el Orden Natural será el vértice procesado en Greedy en el lugar i.

Es decir, Greedy procesará los vértices en el orden de sus indices dado por Orden[0],Orden[1],Orden[2], etc.

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.

La función escribe en el lugar i de Coloreo[] cual es el color que Greedy le asigna al vértice cuyo índice es i en el Orden Natural.

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

3 Funciones para crear ordenes

Solo pediremos funciones que reordenan por bloques de colores.

3.1 OrdenImparPar()

Prototipo de función:

```
char OrdenImparPar(u32 n,u32* Orden,u32* Color);
```

Esta funcion asume que Orden y Color apuntan a un sector de memoria con al menos n lugares. Tambien asume que la imagen de Color[] es un conjunto $\{0, 1, ..., r-1\}$ para algun r.

Ordena indices llenando el array Orden poniendo primero los indices i tal que Color[i] sea igual al mayor impar en $\{0, 1, ..., r-1\}$, luego los indices i tal que Color[i] es igual el segundo mayor impar, etc hasta terminar con los impares. Luego pone los indices i tal que Color[i] es el mayor par, luego el segundo mayor par, etc.

Observacion: Aca no es necesaria la estructura de G, solo n.

Si todo anduvo bien devuelve el char 0, si no el char 1. (razones por la que podria andar mal es que hagan un alloc de memoria auxiliar y el mismo falle)

3.2 OrdenJedi()

Prototipo de función:

```
char OrdenJedi(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 $F : \{0, 1, ..., r-1\} \mapsto \mathbb{Z}$ dada por:

$$F(x) = x. \left(\sum_{i:Color[i]=x} Grado(i, G) \right)$$

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

Idea de porque esta funcion es util: Un par de posibles estrategias de orden de los vertices que suelen funcionar bien:

1. Mientras mas grande sea el grado de un vertice, mas complicado es colorearlo, salvo que este al principio del orden. Dejar los vertices de mayor grado para el final puede complicar colorearlos, por eso seria mejor ordenarlos en orden inverso de sus grados. (este seria el famoso orden WelshPowell). Pero queremos algo que ordene por bloque de colores, no por grados.

2. Si ya coloreamos con Greedy, es razonable suponer que los vertices que tienen el mayor color son los mas dificiles de colorear, por lo tanto conviene ponerlos primero.

Tratando de hacer una estrategia que haga una mescla, sumamos todos los grados de todos los vertices que tienen un color dado para imitar la estrategia 1. (no es del todo adecuada porque podria pasar que la suma sea grande no porque los grados sean grandes sino porque haya muchos vertices de ese color).

Si solo sumaramos los grados, una vez aplicada esta estrategia, volver a aplicarla seguiria pintando todos esos de color 0, asi que casi seguro seguirian teniendo color 0 de ahi en mas (porque la suma de los grados seria la misma o mayor) y quizas necesitamos mesclar un poco. Asi que para tener en cuenta el color, multiplicamos la suma por el color, para mesclar con la estrategia [2] Esto en particular hace que los vertices que vienen coloreados con 0 ahora queden AL ULTIMO, aseguradonos que no tendremos el mismo orden que antes.

4 Cosas a prestar atención

4.1 Errores comunes

1. 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.

2. 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.

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

4. No usen variables shadows.

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

6. 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.

7. 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.

8. Testeen que los ordenes hacen lo que se pide. Una forma de hacer esto para testear las funciones de ordenamiento es, luego de usarlas, sin hacer Greedy, o antes de hacer Greedy, hacer un for en i imprimiendo Color[Orden[i]] y verificando que la salida es lo que se pide en cada caso.

9. 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.

10. Dependiendo como implementen las funciones van a necesitar una funcion de ordenacion. (casi seguro para la segunda, y posiblemente, o no, para la primera).

    Usar un algoritmo de ordenación $O(n^2)$ como Bubble Sort es desaprobación automática. Estando en 3er año, ya deberian haber visto mejores algoritmos.

Pueden usar su algoritmo preferido $O(n \log n)$ o bien, como les habia adelantado para la primera parte, pueden usar la función qsort de C, que funciona muy bien en gcc.

Para la 2da funcion, para ordenar los COLORES de acuerdo con la funcion F, parece a primera vista que si o si hay que usar una funcion de comparacion. Pero para ordenar los INDICES, tanto en la primera como en la 2da funcion, hay varias posibilidades, dependiendo como implementen las funciones. Pueden tambien usar qsort u otro algoritmo de ordenacion por comparacion, pero otra posibilidad es usar un algoritmo $O(n)$ para ordenar **sin comparar**, en el caso de los indices, pues al ordenarlos de acuerdo con sus COLORES, que son una cantidad acotada por $r \leq n$, se puede usar alguno de los algoritmos similares a bucket sort adaptados a este caso.

5 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 correr Greedy al menos en el orden natural (si quieren correr otros ordenes posibles y quedarse con el mejor, esta bien, pero al menos debe hacer el orden natural).

Luego de correr Greedy en el orden natural (o diversos ordenes, quedandose con el mejor) se hace una iteracion en donde hacen lo siguiente:

1. corren OrdenImparPar y a continuacion Greedy con ese orden
2. Paralelamente corren OrdenJedi y Greedy con ese orden.

Obviamente los coloreos obtenidos en [1] y [2] deben ser guardados en distintos arrays.

Se repiten [1] y [2] 16 veces, y luego se intercambian las estrategias: el array donde se guardaba el coloreo de [1] ahora se procesa usando la estrategia [2] y viceversa.

Cada 16 veces se alternan las estrategias.

En total deben correr 500 veces [1] y 500 veces [2] (es decir, que el total de Greedys sin contar los primeros seran 1000).

5.1 Velocidad

El código debe ser "suficientemente"rápido. Greedy en particular debe ser rápido pues será usado múltiples veces.

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 hasta 15-20 minutos para no ser estrictos. Puede ser un poco mas, pero no horas o dias.

Tengan en cuenta que voy a corregir su parte 2 con mi parte 1, que es rapida, asi que si tienen un problema de velocidad porque su parte 1 es lenta, en la correccion de la parte 2 eso no importa. Pero deberia haber hecho una parte 1 suficientemente agil.