Proyecto de Matemática Discreta II-2021 3ra parte

1 Introducción

Esta 3ra parte tiene sólo tres requerimientos:

1. una función.
2. un main que ensambla (casi) todas las funciones dadas hasta ahora.
3. Archivos de resultados de la corrida del main de (2) en algunos grafos seleccionados

2 Función de la parte 3

Al igual que las funciones de la parte 2, la función definida aca debe usar las funciones definidas en la parte 1 PERO NO la estructura interna del grafo. Para testear esta funcion usaré MI parte 1.

Como dijimos en la parte 2,Uds. deben suponer que han sido contratados para codear esto y todo lo que tienen son las especificaciones de las funciones de la primera parte pero no el código de las mismas.

2.1 OrdenPorBloqueDeColores

Prototipo de función:

```
char OrdenPorBloqueDeColores(Grafo G,u32* perm);
```

Verifica que perm sea una permutación de $\{0, 1, ..., r-1\}$ donde r es el número de colores que tiene G. Si no lo es, retorna 0 sin hacer nada mas.

Si perm es una permutación de $\{0, 1, ..., r-1\}$ entonces ordena los vértices de G poniendo primero todos los vértices de color perm[0], luego todos los vértices de color perm[1], etc, hasta poner al último todos los vértices de color perm$[r-1]$ y luego retorna 1.

El orden entre los vértices del mismo color no se especifica, pueden hacerlo como les sea mas conveniente.

La función reordena los vértices PERO NO LES CAMBIA EL COLOR.

Esta función debe ser $O(n)$ (lo mejor, pero requiere un código un poco mas complicado) o bien $O(n \log n)$ también es aceptable.

$O(n^2)$ o similar es desaprobación instantanea de todo el proyecto.

Por supuesto esto depende de que Color() y FijarOrden() sean $O(1)$, que en el caso de mi parte 1 lo son. Si las de ustedes no lo son, van a demorar en testear esta función, pero el código de OrdenPorBloqueDeColores debe ser tal que si Color() y FijarOrden() son $O(1)$ entonces OrdenPorBloqueDeColores sea $O(n)$ u $O(n \log n)$.

Ustedes deben testear con algún archivo de testeo que esta función ordene los vértices cómo se especifica, con distintos reordenes de los colores.

Al testear que perm sea una permutación de $\{0, 1, ..., r-1\}$, pueden asumir que perm es un array de al menos r lugares y que lo que van a testear es que la parte del array de los primeros r lugares es la que es una permutación de $\{0, 1, ..., r-1\}$.

Si bien en mi main siempre doy una perm que usa exactamente r lugares, dejo esta posibilidad abierta por si prefieren usar otro método en su main.

Obviamente si perm no tiene reservado al menos r lugares, se puede producir un segfault, asi que deben ser cuidadosos en cómo usan esta función.

3 Main

Deberán ademas entregar un archivo con un main que cargue un grafo y haga todo lo que se indica a continuación:

1. Para empezar deberá requerir 6 parametros enteros a, b, c, d, e, f. Esto debe ser hecho de forma tal que si compilamos a un ejecutable ejec podamos hacer

   ./ejec a b c d e f <enter>

   y luego cargar a mano el grafo, o bien poder usar el operador de redirección y hacer pej

   ./ejec a b c d e f <q10.txt

   donde q10.txt sea algun archivo con un grafo.

2. Luego de cargar el grafo, imprime el número de vértices, el número de lados y Delta.

3. Luego de eso, corre Bipartito() para ver si el grafo es bipartito o no. Si lo es, se declara eso y se termina la corrida. Si no lo es, se declara eso y se continua con el resto.

4. Para el resto de la corrida se usarán procesos pseudoaleatorios, pej en AleatorizarVertices() uno de los parametros es R que funciona como semilla de aleatoriedad. La fuente "inicial"de aleatoriedad a partir de la cual todas las demas fuentes se vayan obteniendo será el parámetro f.

5. Hace un ordenamiento en el orden natural mas a ordenes aleatorios usando la función AleatorizarVertices() de la parte 2. Luego de cada uno de estos ordenamientos corre la función Greedy() e imprime la cantidad de colores obtenida. Salva la información suficiente como para poder saber cual de todos estos $a + 1$ ordenamientos da el mejor coloreo, y luego, salvo que sea el último de todos los hechos, reordena los vértices con el mismo ordenamiento que da el mejor coloreo y vuelve a correr Greedy con ese ordenamiento.

6. Luego de esto, a partir de ese coloreo que se ha obtenido, se hacen b ordenamientos usando OrdenPorBloqueDeColores() con distintos ordenes pseudoaleatorios de los colores, y corre Greedy() luego de cada uno de ellos, imprimiendo el número de colores obtenido.

   Por el teorema dado en clase, si programaron OrdenPorBloqueDeColores() y Greedy() bien, la cantidad de colores no puede aumentar luego de cadar orden por bloque de colores mas greedy, asi que no hace falta salvar al mejor coloreo porque siempre será el último.

Pero deben, al menos en una versión de uso interno suyo, verificar que el coloreo efectivamente no empeora, pues si lo hace es prueba de que tienen algo mal.(MIS diversos mains de testeo harán este chequeo.)

7. Luego de esto, a partir del último coloreo obtenido, hace c ciclos externos dentro de los cuales habrá d ciclos internos, en donde se ejecuta lo siguiente:

   - Al comienzo de cada uno de los c ciclos se copia el grafo en dos copias extras. (es decir, en total habrá tres grafos: el original y dos copias).
   - Cada una de esos grafos hará d ciclos internos donde cada uno "evolucionará" en paralelo pero por separado de las otras dos copias, con tres estrategias de evolución distinta, todas usando OrdenPorBloqueDeColores() y Greedy() pero cambiando como van ordenando los colores. En cada caso deben imprimir los números de colores obtenidos de forma que quede claro cual color corresponde a cual copia. (pej, poner $G, H, W : 15, 19, 17$ o algo similar).
   - Otra vez, si tienen bien programadas OrdenPorBloqueDeColores() y Greedy(), la evolución dentro de cada rama no debe aumentar el número de colores, pero por supuesto el número de colores en cada rama puede ser distinto. Al termino de los d ciclos por separado, se compara quien de las tres ramas tiene la menor cantidad de colores, se eliminan las otras dos copias, se preserva el grafo con el coloreo óptimo hasta ese momento y se comienza un nuevo ciclo de d evoluciones en paralelo volviendo a copiar ese grafo óptimo y haciendo evolucionar los 3 en paralelo, etc, haciendo esto, como dijimos arriba, c veces.
   - Las tres estrategias de evolución serán:
     (a) Uno de los grafos continuará evolucionando como antes, ordenando los colores aleatoriamente.
     (b) Otro de los grafos ordenará siempre los colores de mayor a menor.
     (c) El tercero ordenará los colores de mayor a menor y luego cambiara aleatoriamente de lugar ALGUNAS de sus entradas solamente. Para ello mirará cada entrada y con probabilidad $\frac{1}{e}$ la intercambiará con otra elegida al azar.

8. Al final de todo, se imprime el número de colores obtenido y el número total de Greedys hecho y se libera toda la memoria usada.

Su Greedy, reordenes y CopiarGrafo deberian ser tales que, en cualquier combinación de a, b, c, d que haga que la cantidad total de Greedys sea aprox. 1000, puedan hacer todo lo anterior en no mas de 15-20 minutos aún para los grafos grandes. El límite no es "hard" (pej, podria ser que para algún grafo demoren, digamos 30 minutos) pero no pueden estar demorando una hora, dos horas, tres dias o un par de meses.

El CopiarGrafo() que usaré será el mio, que es eficiente, asi que si su CopiarGrafo() no lo es, no les afectará la velocidad en mis tests, aunque obviamente impactará en los tests que usen ustedes.

3.1 Corridas de Ejemplo

Deben testear extensivamente sus funciones, con este y otros mains que hagan, pero al menos con este. Deben mandar algunos archivos de corrida de su main con algunos grafos y algunos parametros. Especificaremos esos grafos y parámetros en la página del Aula Virtual.