Proyecto de Matemática Discreta II-2021 2da parte

Contents

1 Funciones de la parte 2

Las funciones definidas aca deben usar las funciones definidas en la parte 1 PERO NO la estructura interna del grafo. Para testear estas funciones podré usar sus funciones de la parte 1 O LAS MIAS. (casi siempre las mias)

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 codigo de las mismas.

Obviamente para testear estas funciones van a tener que usar las funciones de la 1ra etapa, pero si pueden, ademas de usar SUS funciones, traten de usar las funciones de la 1ra etapa de algun otro grupo para estar seguros que estan programando bien.

1.1 Greedy()

Prototipo de función:

```
u32 Greedy(Grafo G);
```

Corre greedy en G ******comenzando con el color 0*****, con el orden interno que debe estar guardado de alguna forma dentro de G.

Devuelve el número de colores que se obtiene.

Si su implementación de Greedy usa algún alloc debe devolver $2^{32} - 1$ si el alloc falla. (este número es demasiado grande para que pueda ser la cantidad de colores de cualquiera de los grafos que usaremos).

Obviamente deberán hacer uso extensivo de la función FijarColor y las funciones para obtener información de los vértices y sus vecinos.

1.2 Bipartito()

Prototipo de función:

```
char Bipartito(Grafo G);
```

Devuelve 1 si G es bipartito, 0 si no.

Ademas, si devuelve 1, colorea G con un coloreo propio de dos colores. Si devuelve 0, debe dejar a G coloreado con algún coloreo propio. (no necesariamente el mismo que tenía al principio).

1.3 AleatorizarVertices()

Prototipo de función:

```
char AleatorizarVertices(Grafo G,u32 R);
```

"Aleatoriza" el orden de los vértices de G, usando como semilla de aleatoridad el número R.

Retorna 0 si todo anduvo bien y 1 si hubo algún problema.

El objetivo de esta función es que ustedes puedan usarla para testear Greedy (ver abajo) con diversos ordenes.

El orden no será en realidad aleatorio sino "pseudoaleatorio"y debe depender determinísticamente de la variable R.

(es decir, correr dos veces esta función con R=4 pej, debe dar los mismos resultados, pero si R=12, debe dar otro resultado, el cual debería ser sustancialmente distinto del anterior).

IMPORTANTE: debe dar el mismo resultado independientemente del coloreo o del orden que tenga en ese momento G. Es decir, si esta función es llamada en dos lugares distintos de un programa con la misma semilla, el orden obtenido debe ser el mismo.

Esto deberian testearlo (es una de las cosas que yo voy a hacer).

Debe cambiar el orden de los vértices PERO NO EL COLOR de los vertices.

Idealmente la función debería ser tal que para cualquier orden posible de los vértices exista algún R tal que AleatorizarVertices(G,R) obtenga ese orden. Sin embargo esto puede ser pedir demasiado y no es el objetivo de la función ser perfectamente aleatoria.

Pero si deberia ser tal que no haya ninguna restricción obvia en los ordenes que se puedan obtener. Pej, una función que, con vértices 4,5,7,10,15 sólo es capaz de producir los ordenes (4,5,7,10,15); (5,7,10,15,4);(7,10,15,4,5);(10,15,4,5,7) y (15,4,5,7,10) no es buena, o una función que, pej, en el 90% de los casos hace que el primer vértice sea 4 tampoco seria una buena idea. De todos modos si bien revisaré que el orden "aleatorio"al menos "luzca"aleatorio, no haré tests estadísticos ni nada de eso.

LO MAS IMPORTANTE DE ESTA FUNCIÓN: es que lo que quede luego de ser ejecutada SIGA SIENDO UN ORDEN DE TODOS LOS VÉRTICES y no que algún vértice "desapareció"o hay algún vértice que aparece 2 veces en el orden.

Idealmente deberian usar un esquema de aleatorización que les asegure que el orden que obtienen es una biyección de los vértices, pero aún si demuestran esto (para ustedes, no hace falta que me manden ninguna prueba) podrían tener algún error, asi que deberían testear esto con algunos ejemplos, verificando que realmente no perdieron ni duplicaron vertices.

2 Cosas a prestar atención

2.1 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.

   Una función donde los alumnos suelen producir stacks overflows por la primera razón es Bipartito.

   Respecto de la segunda, si el tamaño del array depende de una variable que puede ser muy grande, usen el heap, no el stack. Por ejemplo, algo que dependa del número de vertices no debe ir al stack mientras que algo que dependa del número de colores puede ir, pues no habrá grafos que usen mas de a lo sumo una decena de miles de colores.

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 o Bipartito den 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 sutil pero que ha ocurrido es que Greedy de siempre la misma cantidad de colores para un grafo dado, independientemente del orden de los vertices. Hay grafos con los cuales greedy siempre dará la misma cantidad de colores, pej, los completos pero la mayoria no. En mi página del 2020 hay muchos ejemplos de grafos para los cuales Greedy debe dar distinto número de colores dependiendo del orden.

8. Con los grafos que mi pagina del 2020 dice que son bipartitos....chequeen que Bipartito diga que lo son.

9. El caso opuesto de que Bipartito diga que es bipartito un grafo que no es bipartito, sea de mi página o no, el coloreo que les va a dar no es propio asi que eso deberian poder testearlo facilmente.

2.2 Velocidad

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

Deben poder hacer 1000 Greedys con reordenes de vertices en a lo sumo 15-20 minutos en una maquina razonable aun para los grafos grandes. Puede ser un poco mas, pero no horas o dias.

3 Main

Deberán ademas entregar un main, pero no en esta etapa. En esta etapa deberian hacer 2 mains (partes de los cuales serán pedidos luego) En un main deberan hacer Greedy mas reordenes aleatorios una cierta cantidad de veces y quedarse con el mejor de esos coloreos. Como se dice arriba deberian poder hacer 1000 Greedys mas reordenes en no mas de 15-30 minutos, o bien 100 Greedys en aprox. una decima parte de eso, etc. (no seria exactamente una decima parte por el overhead de la construccion del grafo).

Tambien deberán hacer Bipartito antes de hacer eso y al final de todo, y verificar que en ambos casos obtienen la misma respuesta. [esto en realidad es nada mas que un test simple para ver si no cometieron un error grave en el código de Bipartito o en el código de Greedy, en el cual podrían cometer un error sutil que les haga desaparecer o agregar lados, pej (ha pasado en otros años)]

El otro main es similar pero con CopiarGrafo: deben copiar el grafo, hacerles Greedy y reordenes aleatorios a las dos copias por separado una cierta cantidad de veces, luego destruir la copia original, volver a copiar el grafo y repetir, imprimiendo los colores obtenidos en cada Greedy.

El objetivo de este main es simplemente para que verifiquen que no estan haciendo algo super mal con Greedy o CopiarGrafo que provoque pej que en vez de cambiar los colores al grafo original se los esten cambiando a la copia. Si bien este test no es muy exhaustivo que digamos, deberia detectar un error catastrófico.