Proyecto de Matemática Discreta II-2022 2da parte

Contents

1 Introducción

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 o las de otro grupo. (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 código de las mismas.

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:

AlduinPaarthurnaxIrileth.h

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

En la parte 3 deberán entregar un .c que tenga un main que use ESTAS funciones (mas las de la parte 1) y corra Greedy con diversos ordenes. Las especificaciones de ese main las daremos en otro documento (la parte 3), pero si quieren ir programando algo similar a lo que pediremos, y para testear estas funciones, les damos una idea de como será ese main al final de este documento. Por supuesto pueden (y deberian) testear partes especificas de sus funciones con diversos otros mains. (yo uso varios para testearlos a uds).

2 Funciones de coloreo

2.1 Bipartito()

Prototipo de función:

```
u32* Bipartito(Grafo G);
```

Si $\chi(G) \leq 2$, (y como todos nuestros grafos tienen al menos un lado, $\chi(G) \leq 2$ es equivalente a $\chi(G) = 2$) devuelve un puntero a un array de u32s que debe tener longitud n = número de vértices de G, y ser un coloreo propio de G con los colores 1 y 2, con la entrada i del array diciendo que color tiene el vértice cuyo índice es i en el Orden Natural. La reserva de memoria para ese array debe ser hecha dentro de Bipartito, obviamente.

Si $\chi(G) \geq 3$, devuelve un puntero a NULL.

WARNING: muchos grupos a lo largo de los años programan mal esta función pues usan una recursión llamandose a si misma que con algunos grafos crashea el sistema. Puede que hayan quedado enamorados de las recursiones autoreferentes en cursos anteriores, pero les deberían haber enseñado que siempre que se usa ese recurso hay que contemplar si las condiciones del problema hacen que un stack overflow pueda ser un problema. (la respuesta es afirmativa en el caso de algunos grafos de millones de vértices con cierta estructura especial).

2.2 Greedy()

Prototipo de función:

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

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.

En otras palabras, 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 vértices, es decir, es una biyección. Tambien asume que Coloreo 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 $2^{32}-1$.

OBSERVACIÓN: el coloreo de Greedy empieza en 0, el coloreo de Bipartito es con colores 1 y 2. Sheogorath insistió en eso.

3 Función para crear un orden a partir de claves

Esta función es clave para generar los ordenes que luego usará Greedy. En vez de hacer diversas funciones de generaciones de ordenes, este año usaremos sólo esta, que genera un orden a partir de una clave, y lo que haremos será cambiar que claves usaremos. De esta manera sólo tendrán que codificar una vez la parte dificil de crear un orden a partir de otra cosa.

Claves tipicas que usaremos son claves aleatorias, o grados de los vértices, o colores de los vértices.

3.1 OrdenFromKey()

Prototipo de función:

```
char OrdenFromKey(u32 n,u32* key,u32* Orden);
```

A partir del array apuntado por key, el cual se asume que es de longitud n,se llena el array al cual apunta Orden, el cual se asume que tiene suficiente memoria para al menos n entradas. Tambien se asume que el **rango** de key es n o menor, es decir key[i]$\leq n$ para todo i. Lo que NO se asume, y de hecho casi nunca será cierto, es que key sea una biyección.

Debe llenar Orden de forma tal que en el lugar 0 vaya el índice i_0 tal que key[i_0] sea el máximo de todos los valores key[i] (si hay varios is que dan el máximo, se puede tomar cualquiera, no se especifica cual), luego en el lugar 1 va el índice $i_1 \neq i_0$ tal que key[i_1] sea el máximo de todos los valores key[i] excepto por i_0, etc.

Es decir, da un orden de los indices tal que Orden[i]$\neq$Orden[j] si $i \neq j$ y tal que:

key[Orden[0]]$\geq$key[Orden[1]]$\geq$key[Orden[2]]$\geq$....$\geq$ key[Orden[n-1]].

OBSERVACIÓN SOBRE ESTA FUNCIÓN: Esencialmente lo que hace es ordenar las claves de MAYOR a MENOR. Asi que tendrán que usar algún algoritmo de ordenación. **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.

En realidad la especificación de qsort no demanda ninguna complejidad concreta, asi que alguna implementación de qsort en alguna implementación de C podria ser $O(n^2)$ pero con gcc las cosas andan bien. Aparentemente en gcc usan una variación de introsort para qsort, con una complejidad de peor caso de $O(n \log n)$ pero que ademas es considerablemente mas rápida que pej heapsort en la práctica.

De todos modos, hay una posibilidad en este caso incluso mejor que qsort. (en la parte 1 esto que explico a continuación no se puede usar para ordenar los lados porque los vértices eran cualquier u32).

Esta otra posibilidad es usar un algoritmo $O(n)$ para ordenar **sin comparar**. Este tipo de algoritmos puede usarse en este caso pues se asume que el rango de key es n o menor.

En todos los casos que usaremos esta función, eso se cumplirá, asi que pueden programar la función usando uno de los algoritmos $O(n)$ de ordenación que ordena sin comparar cuando el rango es n o menos.

Esta última opción les puede llevar un poco mas de tiempo para programar que usar qsort, pero en los casos de n en millones ganaran apreciablemente en velocidad. Pero si usan qsort tambien es aceptable.

4 Funciones para crear claves especificas

La primera crea claves aleatorias. Las otra dos en realidad crean un recoloreo de los vértices, y la idea es luego pasarle ese recoloreo como clave a la funcion OrdenFromKey, para usar la propiedad del VIT.

4.1 AleatorizarKeys()

Prototipo de función:

```
void AleatorizarKeys(u32 n,u32 R,u32* key);
```

Usando como semilla de pseudoaleatoridad al número R, “aleatoriza” las entradas de key. El array al cual apunta key se asume que tiene al menos n entradas disponibles.

Es decir, en key[i] introduce un número “aleatorio” entre 0 y $n-1$.

La entrada obviamente no será verdaderamente aleatoria sino "pseudoaleatorio", dependiendo 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).

Al ser cada entrada pseudoaleatoria, el array al cual apunta key no necesariamente será una permutación de $\{0, 1, ..., n-1\}$ (de hecho, casi nunca lo será). Esto no es problema pues la función OrdenFromKey detallada arriba luego usa este array y crea un orden.

4.2 PermutarColores

`u32* PermutarColores(u32 n,u32* Coloreo,u32 R);`

La función asume que Coloreo apunta a una region de memoria con al menos n lugares. Tambien asume que las entradas de Coloreo son numeros compatibles con un coloreo de Greedy, es decir, para algún $r > 0$, aparecen r colores y esos colores son números entre 0 y $r - 1$.(Coloreo[i] lo pueden pensar como el color asignado al vértice cuyo índice es i en el Orden Natural., pero como esta función no accede al grafo, esta información le es irrelevante a la función).

Esta función lo que hace es permutar los colores:

Aloca memoria para un array de n lugares, y devuelve un puntero a ese array. (si hay algún error de alocación, devuelve NULL).

Las entradas de ese array son las mismas que las de Coloreo **excepto que se les cambia el nombre a los colores.**

Los nuevos nombres deben seguir siendo $0, 1, ..., r-1$, pero si antes se tenia pej Coloreo[i]=4, y denotamos por ColoreoNuevo al array que hay que devolver, ahora puede ser pej ColoreoNuevo[i]=7, y en ese caso no solo i sino todos los j tal que Coloreo[j]=4 van a tener ahora ColoreoNuevo[j]=7. Por otro lado los indices k tal que Coloreo[k]=7 ahora tendran otro color, pej ColoreoNuevo[k]=1 etc.

El reasignamiento de colores debe ser pseudoaleatorio dependiendo de la semilla de aleatoriedad R.

Ayuda: Hay al menos 2 formas de programar esta función. Una es copiar Coloreo en el nuevo array y luego ir modificando el nuevo array pseudoaleatoriamente, pero teniendo en cuenta siempre de mantener las condiciones arriba mencionadas. Dado que esto debera iterar sobre un array de dimensión n varias veces, y que r puede ser considerablemente menor que n, esta probablemente no sea la mejor forma, pero tiene la ventaja de no usar memoria extra.

La segunda forma, mas fácil, es generar un segundo array temporario digamos PermC, con r entradas, definir PermC[i]=i; y permutando lugares o con alguna otra forma de aleatorizacion que asegure que PermC siempre sea una biyección, terminar con una PermC que es una permutación pseudoaleatoria de $\{0, 1, .., r - 1\}$, es decir, una permutación de los colores. Luego hacer ColoreoNuevo[i]=PermC[Coloreo[i]]; y finalmente obviamente liberar la memoria de PermC.

La ventaja es que solo se trabaja sobre un array de r lugares en vez de n, pero la desventaja es que hay que usar un poco de memoria extra. Pero como r en general será a lo sumo 2000, no deberia ser problema.

4.3 RecoloreoCardinalidadDecrecienteBC

(Cardinalidad Decreciente de los Bloques de Colores)

Esta función tambien cambia el nombre de los colores, como la anterior, pero a diferencia de la anterior, en vez de renombrar los colores aleatoriamente, lo hace siguiendo una regla específica.

Prototipo de función:

`u32* RecoloreoCardinalidadDecrecienteBC(u32 n,u32* Coloreo);`

La función asume que Coloreo apuntan a una region de memoria con al menos n lugares. Tambien asume que las entradas de Coloreo son numeros compatibles con un coloreo que empieza en 0, es decir, números entre 0 y algún número $r - 1$, y que todos los números $0, 1, ..., r - 1$ aparecen al menos una vez en el rango de Coloreo. Coloreo[i] deberia ser el color asignado al vértice cuyo índice es i en el Orden Natural., pero como esta función no accede al grafo, esta información le es irrelevante.

Al igual que la función anterior, esta función aloca memoria para un array de n lugares, y devuelve un puntero a ese array. (si hay algún error de alocación, devuelve NULL).

Especificación de lo que hace:

Si Coloreo tiene r colores $0, 1, .., r - 1$ definimos para cada color c:

IC_c =conjunto de indices i tal que Coloreo[i]=c.

Ordenar los colores en la forma $c_0, c_1, ..., c_{r-1}$ tales que $|IC_{c_0}| \geq |IC_{c_1}| \geq |IC_{c_2}| \geq ... \geq |IC_{c_{r-1}}|$.

Renombra el color c_k con el color k.

(es decir, si el array creado se llama pej NuevoColoreo, entonces Coloreo[i]= c_k $\Rightarrow$NuevoColoreo[i]= k).

5 Cosas a prestar atención

5.1 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 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. 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.

10. Testeen que OrdenFromKey funcione bien mirando que en cuando la usan luego de RecoloreoCardinalidadDecrecienteBC o PermutarColores, si imprimen el array Coloreo siguiendo el orden de los indices dado por Orden, la salida sea como se supone que tiene que ser. Tambien pueden chequear algunos casos como key[i]=i, o llenar key con los grados de los vertices y verificar la salida.

11. 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. Esto significa que si usan como clave para OrdenFromKey un coloreo de los vértices, si el numero de colores AUMENTA luego de Greedy respecto al que se tenia antes, entonces tienen un error. Si ese error viene de que crearon mal el coloreo, o que tienen un error en OrdenFromKey, o tienen un error en Greedy, es algo que van a tener que descubrir, pero al menos saben que tienen un error.

5.2 Main

La idea del main que van a tener que entregar en la parte 3 es generar diversas ramas de exploración de coloreos, usando claves diversas como pej los grados, o claves aleatorias, luego para cada una de esas claves generar un orden usando OrdenFromKey(), correr Greedy con ese orden, y luego iterar Greedy una cierta cantidad de rondas de ahi en mas con claves basadas en el coloreo, para que el VIT nos asegure de no empeorar los coloreos. Luego de hacer esto para cada una de las claves iniciales, nos quedamos con el mejor coloreo que hayamos podido conseguir y a partir de el seguir VIT-iterando.

5.3 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. (en realidad deberian poder hacerlo en menos de 5 minutos en casi todos los grafos, pero no seremos tan estrictos)