Proyecto de Matemática Discreta II-2022-Primera Parte

Contents

1 Introducción

El proyecto puede ser hecho en forma individual o en grupos de 2 o 3 personas.

El proyecto se dividirá en varias etapas. Este primer documento da las especificaciones de las funciones de la primera etapa.

El proyecto tiene una nota entre 0 y 10 y deben obtener al menos un 4 para aprobar.

Si bien el proyecto está dividido en varias etapas, se evalua globalmente, es decir, las distintas etapas no se promedian. Esto es asi porque básicamente la forma de corregir es descontar puntos por errores, asi que aun si una etapa esta perfecta, si hay un error grave en otra el descuento será substancial.

Las funciones de las otras etapas usarán estas funciones como funciones auxiliares.

Los objetivos en este proyecto son:

1. Implementar un tema enseñado en clase en un lenguaje, observando las dificultades de pasar de la teoria a un programa concreto.
2. Prácticar programar funciones adhiriendose a las especificaciones de las mismas.
3. Práctica de testeo de programas.

El lenguaje es C. (C99, i.e., pueden usar comentarios // u otras cosas de C99).

El objetivo global es hacer un programa que corra Greedy iterado como se explica en el teórico.

La idea NO ES presentar un programa único completo que haga esto, sino las funciones detalladas, que luego se pueden ensamblar en uno o mas mains que las use, de acuerdo a distintas estrategias.

La catedra usará sus propios mains para testear las funciones, y ustedes no tendrán acceso a ellos, por lo que deben programar las funciones detalladas siguiendo las especificaciones.

Deben testear la funcionalidad de cada una de las funciones que programan, con programas que testeen si las funciones efectivamente hacen lo que hacen o no.

Programar sin errores es difícil, y algunos errores se les pueden pasar aún siendo cuidadosos y haciendo tests, porque somos humanos, y algunos errores son dificiles de detectar. Pero hay errores que no deberían quedar en el código que entreguen, porque son errores que son fácilmente detectables con un mínimo testeo.

Tengan en cuenta que uds. deben programar esto de acuerdo a las especificaciones porque no saben qué programa va a ser el que llame a sus funciones, ni cómo las van a usar.

El objetivo no es sólo programar lo indicado abajo sino tambien TESTEAR lo programado adecuadamente.

Se descontarán mas puntos por errores de programación que deberían haberse detectado con un testeo razonable que por errores que pueden ser díficiles de detectar con tests.

Ejemplo: un año un grupo "A"entregó una función que estaba programada razonablemente, y funcionaba casi siempre bien, pero habia algunos casos raros en los que no funcionaba y daba resultados atroces como reportar que un grafo con n vertices necesitaba MAS de n colores para ser coloreado. Pero esos casos no eran fáciles de detectar, es decir, no eran pej casos extremos que se testearian normalmente, sino que eran producto de un error sútil en la programación.

Por otro lado, un grupo "B"entregó una función que no hacia lo que se pedia ni siquiera con casos elementales de testeo, y daba el resultado incorrecto casi siempre.

Un grupo "C"entregó una función que no hacia nada. Es decir, "hacia"muchas cosas, declaraba variables temporales, las modificaba, etc, pero al final de toda la función, el estado quedaba igual.

Lo que estamos diciendo es que el grupo "A"tuvo muchisimo menos descuento que el grupo "B"mientras que el grupo "C"tuvo el mayor descuento porque deberia haberse dado cuenta con un mínimo testeo que su función no hacía nada.

1.1 Entrega

Deben entregar vía e-mail a daniel.penazzi arroba unc.edu.ar los archivos que implementan el proyecto.

Las funciones que estan descriptas en esta etapa serán usadas luego por otras funciones que especificaremos mas adelante. En esta etapa las funciones consisten basicamente en las funciones que permiten leer los datos de un grafo y cargarlos en una estructura adecuada, mas funciones que permiten acceder a esos datos.

1.2 Restricciones generales

1. No se puede incluir NINGUNA libreria externa que no sea una de las básicas de C. (eg, stdlib.h, stdio.h, strings.h, stdbool.h, assert.h etc, si, pero otras no. Especificamente, glibc NO).

2. El código debe ser razonablemente portable, aunque no tengo acceso a un sistema de Apple, y en general lo testearé con Linux, puedo tambien testearlo desde Windows.

3. No pueden usar archivos llamados aux.c o aux.h

4. No pueden tener archivos tales que la unica diferencia en su nombre sea diferencia en la capitalización.

5. No pueden usar getline.

6. El uso de macros esta permitido pero como siempre, sean cuidadosos si los usan.

7. Pueden consultar con otros grupos, pero no pueden tener grandes fragmentos de código iguales, o con cambios meramente cosméticos en el código de otro grupo. Consultar por ideas es aceptable, copiar código en bloque no.

8. En años anteriores se les pedia entregar todas las funciones juntas, estas mas algunas de las que pediremos en otras etapas, lo cual hacia que los grupos pudieran usar en algunas funciones la estructura interna del grafo tal como estaba guardado, en vez de hacer un llamado a las funciones auxiliares correspondientes. Algunos grupos tomaban ventaja de esto creando una estructura interna del grafo con campos auxiliares extras que les permitia correr las otras funciones mas rapidamente, y eso estaba bien en esos años. Este año uds. podrán hacer eso con las funciones de esta etapa, pero no con las funciones de las siguientes etapas: las funciones de las siguientes etapas deberán ser programadas de forma tal de usar las funciones de esta primera etapa, llamandolas, pero no pudiendo acceder a la estructura interna del grafo. Nos aseguraremos de esto ademas de obviamente leyendo el código, testeando las funciones de las etapas posteriores con **nuestras** funciones de esta primera etapa, no con las suyas, o bien con estructuras de otro grupo.

1.3 Formato de Entrega

Los archivos del programa deben ser todos archivos .c o .h.

Pueden entregar un sólo .c con todas las funciones si quieren, o separados en varios archivos, pero:

*****No debe haber ningun include de un .c*****

Es usual en C incluir .h pero no deberia en un .c incluirse otro .c !

No debe haber ningun ejecutable.

Los .c que entreguen deben hacer un include de un archivo API llamado AniquilamientoPositronicoIonizanteGravitatorio.h donde se declaran las funciones y, obviamente, de cualquier otro .h que uds necesiten, los cuales deben ser entregados.

AniquilamientoPositronicoIonizanteGravitatorio.h debe tener simplemente la declaración de las funciones, la declaración del tipo de datos 2.3 definida mas abajo y un include de otro .h, EstructuraGrafo.h que es donde pueden poner cosas extras si quieren, incluyendo los dos primeros tipos de datos de la sección 2

Para evitar errores de tipeo con las declaraciones de las funciones, subiremos una copia de AniquilamientoPositronicoIonizanteGravitatorio.h a la página del Aula Virtual.

Para testear deberán hacer por su cuenta uno o mas .c que incluyan un main que les ayude a testear sus funciones, incluyendo funciones nuevas que ustedes quieran usar para testear cosas. (pej, luego de cargar el grafo, imprimir los vertices y sus vecinos para chequear que sus funciones cargaron bien el grafo) Estos archivos NO deben ser entregados.

El mail de entrega debe ser hecho con copia a los demas integrantes del grupo, pero ademas deben adjuntar un archivo ASCII donde conste el nombre, apellido y email de todos los integrantes del grupo.

Esta parte es muy simple, asi que no deberían entregar un montón de archivos complicados. Si lo estan pensando muy complicadamente, probablemente esta mal.

1.4 WARNING

La primera parte de cada grupo será entregada **a otro grupo**. Asi que no soy sólo yo el que leerá su código, sino otro grupo, asi que sean cuidadosos con lo que entregan escrito.

1.5 Compilación

Compilaremos (con mains nuestros) usando gcc, -Wall, -Wextra, -O3, -std=c99 . Tambien haremos -I al directorio donde pondremos los .h

Esas flags seran usadas para testear la velocidad, pero para testear grafos chicos podemos agregar otras flags. Por ejemplo, podemos usar -DNDEBUG si vemos que estan mal usando asserts.

Tambien compilaremos, para testear grafos chicos, con flags que nos permitan ver si hay buffer overflows, shadow variables o comportamientos indefinidos, en particular con -fsanitize=address,undefined. Su programa DEBE compilar y correr correctamente con esa flag aunque para grafos grandes lo correremos con un ejecutable compilado sin esa flag, dado que esa flag provoca una gran demora en la ejecución.

Luego de enviado, se les responderá con un “recibido”. Si no reciben confirmación dentro de las 24hs pregunten si lo recibí.

2 Tipos de datos

Los dos primeros tipos de datos deben ser definidos en un archivo EstructuraGrafo.h.

AniquilamientoPositronicoIonizanteGravitatorio.h debe tener un include de ese .h, si no usan el mismo AniquilamientoPositronicoIonizanteGravitatorio.h que subamos a la página.

EstructuraGrafo.h lo tienen que definir uds de acuerdo con la estructura particular con la cual piensan guardar el grafo.

Tambien puede estar ahi cualquier declaración de funciones auxiliares que necesiten. Esas NO pueden estar en AniquilamientoPositronicoIonizanteGravitatorio.h .

2.1 u32:

Se utilizará el tipo de dato `u32` para especificar un entero de 32 bits sin signo.

Todos los enteros sin signo de 32 bits que aparezcan en la implementación deberán usar este tipo de dato.

Los grafos a colorear tendran una lista de lados cuyos vertices serán todos u32.

Pueden declarar u32 como unsigned int o bien haciendo un include de int.h y declarandolo apropiadamente. u32 NO ES un long unsigned int en computadoras modernas.

2.2 GrafoSt

Es una estructura, la cual contendrá toda la información sobre el grafo necesaria para correr las funciones pedidas.
En particular, la definición interna debe contener como mínimo:

- El número de vertices.
- El número de lados.
- Los nombres y grados de todos los vertices.
- el Delta del grafo (el mayor grado).
- Quienes son los vecinos de cada vértice.

Como se verá en la sección 3, los grafos que se carguen serán no dirigidos.

2.3 Grafo

es un puntero a una estructura de datos *GrafoSt*. Esto estará definido en AniquilamientoPositronicoIonizanteGravitatorio.h .

3 Formato de Entrada

El formato de entrada será una variación de DIMACS, que es un formato estandard para representar grafos, con algunos cambios.

- Las lineas pueden tener una **cantidad arbitraria de caracteres**. (la descripción oficial de Dimacs dice que ninguna linea tendrá mas de 80 caracteres pero hemos visto archivos DIMACS en la web que no cumplen esta especificación y usaremos algunos con lineas de mas de 80 caracteres)
- Al principio habrá cero o mas lineas que empiezan con c las cuales son lineas de comentario y deben ignorarse.
- Luego hay una linea de la forma:

  p edge n m

  donde n y m son dos enteros. Luego de m, y entre n y m, puede haber una cantidad arbitraria de espacios en blancos.

  El primero número (n) representa el número de vértices y el segundo (m) el número de lados.

  Si bien hay ejemplos en la web en donde n es en realidad solo una COTA SUPERIOR del número de vertices y m una cota superior del número de lados, todos los grafos que nosotros usaremos para testear cumplirán que n será el número de vertices exacto y m el número de lados exacto.
- Luego siguen m lineas todas comenzando con e y dos enteros, representando un lado. Es decir, lineas de la forma:

  e v w

  (luego de "w" y entre "v" y "w" puede haber una cantidad arbitraria de espacios en blanco)
- Nunca fijaremos $m = 0$, es decir, siempre habrá al menos un lado. (y por lo tanto, al menos dos vértices).
- Si bien en algunos ejemplos en algunas paginas hay vértices con grado 0, y que por lo tanto no aparecen en ningún lado en nuestros ejemplos no habrá vértices con grado 0: los únicos vértices que cuentan son los vértices que aparecen como extremos de al menos un lado.
- Luego de esas m lineas puede haber una cantidad arbitraria de lineas de cualquier formato las cuales deben ser ignoradas. Es decir, se **debe detener la carga sin leer ninguna otra linea luego de las m lineas**, aún si hay mas lineas. Estas lineas extras pueden tener una forma arbitraria, pueden o no ser comentarios, o extra lados, etc. y deben ser ignoradas.

  Pueden, por ejemplo, tener un SEGUNDO grafo, para que si la función de carga de un grafo se llama dos veces por algún programa, el programa cargue dos grafos.

  Por otro lado, el archivo puede efectivamente terminar en la última de esas lineas, y su código debe poder procesar estos archivos también.

  En un formato válido de entrada habrá al menos m lineas comenzando con e, pero puede haber algún archivo de testeo en el cual no haya al menos m lineas comenzando con e. En ese caso, como se especifica en 4.1, debe detenerse la carga y devolver un puntero a NULL. O por ejemplo tambien podré testea con archivos donde en vez de p edge n m tengan pej p edgee n m.

- En algunos archivos que figuran en la web, en la lista pueden aparecer tanto un lado de la forma

  e 7 9

  como el

  e 9 7

  Los grafos que usaremos nosotros **no son asi**.

  Es decir, si aparece el lado e v w NO aparecerá el lado e w v.

  Ejemplo:

  c FILE: myciel3.col

  c SOURCE: Michael Trick (trick@cmu.edu)

  c DESCRIPTION: Graph based on Mycielski transformation.

  c Triangle free (clique number 2) but increasing

  c coloring number

  p edge 11 20

  e 1 2

  e 1 4

  e 1 7

  e 1 9

  e 2 3

  e 2 6

  e 2 8

  e 3 5

  e 3 7

  e 3 10

  e 4 5

  e 4 6

  e 4 10

  e 5 8

  e 5 9

  e 6 11

  e 7 11

  e 8 11

  e 9 11

  e 10 11

- En el formato DIMACS no parece estar especificado si hay algun limite para los enteros, pero en nuestro caso los limitaremos a enteros de 32 bits sin signo.

- Observar que en el ejemplo y en muchos otros casos en la web los vertices son 1,2,...,n, PERO ESO NO SIEMPRE SERÁ ASI.

  Que un grafo tenga el vértice v no implicará que el grafo tenga el vértice v' con $v' < v$.

  Por ejemplo, los vertices pueden ser solo cinco, y ser $0, 1, 10, 15768, 1000000$.

- El orden de los lados no tiene porqué ser en orden ascendente de los vertices.

  Ejemplo Válido:

  c vertices no consecutivos

  p edge 5 3

  e 1 10

  e 0 15768

  e 1000000 1

4 Funciones De Construcción/Destrucción del grafo

4.1 ConstruccionDelGrafo()

Prototipo de función:

```
Grafo ConstruccionDelGrafo();
```

La función aloca memoria, inicializa lo que haya que inicializar de una estructura GrafoSt ,lee un grafo **desde standard input** en el formato indicado en la sección 3, lo carga en la estructura, y devuelve un puntero a la estructura.

En caso de error, la función devolverá un puntero a `NULL`. (errores posibles pueden ser falla en alocar memoria, pero también que el formato de entrada no sea válido. Por ejemplo, en la sección 3 se dice que en una cierta linea se indicará un número m que indica cuantos lados habrá y a continuación debe haber m lineas cada una de las cuales indica un lado. Si no hay AL MENOS m lineas luego de esa, debe retornar NULL. (si hay mas de m lineas, luego de la primera, sólo debe leer las m primeras).

Dado que esta función debe como mínimo leer todos los lados de los datos de entrada, su complejidad no puede ser inferior a $O(m)$.

Esta función NO PUEDE ser $O(n^2)$ (y MENOS puede ser $O(mn)$) pues en los grafos de testeo habrá grafos con millones de vértices, y un grafo asi con un algoritmo $O(n^2)$ no terminará en un tiempo razonable.

En cuanto a m, puede estar en el orden de millones tambien, y puede ser $m = O(n^2)$, pero sólo para n del orden de miles, mientras que cuando n sea del orden de millones, m no será $O(n^2)$ sino $O(n)$, pues como dijimos arriba esta función no puede tener complejidad menor a $O(m)$ y un m de pej miles de millones haria que demorara mucho.

Asi que deberian pensar una estructura tal que esta función sea, idealmente, $O(m)$ o algo no mucho peor que eso, como $O(mlogm)$.

4.2 DestruccionDelGrafo()

Prototipo de función:

```
void DestruccionDelGrafo(Grafo G);
```

Destruye G y libera la memoria alocada.

Esta función tambien deberia tener una complejidad razonable, no hay razón para que sea mayor a $O(m)$ e incluso puede ser menor, pero $O(m)$ es aceptable.

5 Funciones para extraer información de datos del grafo

Las funciones detalladas en esta sección y la que sigue deben ser todas $O(1)$, pues serán usadas repetidamente por las funciones de la segunda etapa y si no son $O(1)$ no podrán hacer correr las funciones en un tiempo razonable. No debería haber ningún problema con esto, basta con guardar la información en un campo adecuado en la estructura del grafo.

5.1 NumeroDeVertices()

Prototipo de función:

```
u32 NumeroDeVertices(Grafo G);
```

Devuelve el número de vértices de G.

5.2 NumeroDeLados()

Prototipo de función:

```
u32 NumeroDeLados(Grafo G);
```

Devuelve el número de lados de G.

5.3 Delta()

Prototipo de función:

```
u32 Delta(Grafo G);
```

Devuelve $\Delta(G)$, es decir, el mayor grado.

Esta función esta detallada aca para ser usada en algunos casos y no tener que recalcular Δ, asi que si, en vez de hacer el cálculo una vez durante la construcción del grafo y guardar el resultado para que esta función lo pueda leer en $O(1)$, lo que hacen es recalcular Δ cada vez que se llama esta función, tendrán descuento de puntos.

6 Funciones para extraer información de los vertices

Las funciones detalladas en esta sección, como en la anterior deben ser $O(1)$. De hecho, es mucho mas importante que sean $O(1)$ estas, pues las anteriores probablemente sean usadas sólo una o dos veces en cada función, mientras que por ejemplo en Greedy iteraremos sobre los vecinos de un vértice repetidamente, y si la función que permite hacer esto no es $O(1)$ habrá problemas de velocidad.

6.1 Orden Natural e índice en el Orden Natural

Esta no es una función, sino un par de definiciones para que se entiendan las especificaciones de las funciones.

Necesitamos acceder a los datos de los vértices del grafo, pej querriamos saber los grados de los vertices. Pero necesitamos que esto sea hecho de forma independiente de como cada grupo desee hacer la estructura interna. Entonces cuando digamos que queremos saber el grado de un cierto vértice, necesitamos una forma inequivoca de saber quien es el "cierto vértice". Podriamos ciertamente usar el nombre del vértice, pero el nombre es cualquier entero entero 0 y $2^{32}-1$, asi que no podemos guardar todos los datos en un array con 2^{32} elementos. Hay varias formas de poder solucionar esto, y poder seguir refiriendonos a los vértices con su nombre, pero limitarian la forma en que pueden implementar las cosas, y no son las soluciones mas rápidas. Por lo que en vez de eso, le daremos a cada vértice un identificado univoco que en vez de estar entre 0 y $2^{32}-1$, este entre 0 y $n-1$. Lo mejor es usar un orden fijo pero estandard de los vértices, y referirse a los vértices por el índice que tienen en ese orden. Hay varios posibles, nosotros tomaremos el siguiente:

DEFINICIÓN:

El **Orden Natural** de los vértices es el orden de los vértices que se obtiene al ordenarlos de MENOR a MAYOR de acuerdo con sus nombres.

Recordemos que los nombres de los vértices serán enteros sin signo de 32 bits, asi que tiene sentido hablar de ordenarlos de menor a mayor.

Pej, si los vértices son 174391,15,7,4,45,1,95980, entonces el orden natural es:

1,4,7,15,45,95980,174391.

El **indice** de cada vértice en este orden se supone que es a partir de 0, es decir, pej, el índice del vértice 174391 en este orden es 6, el índice del vértice 95980 es 5, el índice del vértice 15 es 3, el índice del vértice 1 es 0, etc.

Observemos que el Orden Natural es un orden fijo, bien definido para cualquier grafo, independiente del estado interno de la estructura que hayan decidido usar.

6.2 Nombre()

Prototipo de Función:

```
u32 Nombre(u32 i,Grafo G);
```

Devuelve el nombre del vértice cuyo índice es i en el Orden Natural.

El nombre es el nombre del vértice con el que figuraba en los datos de entrada.

Pej, en el ejemplo dado arribe Nombre(6,G)=174391, Nombre(2,G)=7, etc.

Dado que el nombre de un vértice puede ser cualquier entero sin signo de 32 bits, esta función no tiene forma de reportar un error (que se produciría si se la llama cuando i es mayor o igual que el número de vértices), asi que debe ser usada con cuidado.

6.3 Grado()

Prototipo de Función:

```
u32 Grado(u32 i,Grafo G);
```

Devuelve el grado del vértice cuyo índice es i en el Orden Natural.

Si i es mayor o igual que el número de vértices, devuelve $2^{32}-1$. (esto nunca puede ser un grado en los grafos que testeeemos, pues para que eso fuese un grado de algún vértice, el grafo deberia tener al menos 2^{32} vertices, lo cual lo haría inmanejable).

6.4 IndiceONVecino()

Prototipo de función:

```
u32 IndiceONVecino(u32 j,u32 k,Grafo G);
```

Es la función que devuelve el índice en el Orden Natural de un vecino de un vértice si j,k estan en los rangos adecuados, y $2^{32}-1$ si no.

Especificamente:

Si k es menor que el número de vértices y j es menor que el grado del vértice cuyo índice es k en el Orden Natural entonces

IndiceONVecino(j,k,Grafo G) es igual a i si y sólo si el vécino j-ésimo del vértice cuyo índice es k en el Orden Natural es el vértice cuyo índice es i en el Orden Natural.

Si k es mayor o igual que el número de vértices o k es menor que el número de vértices pero j es mayor o igual que el grado del vértice cuyo índice es k en el Orden Natural entonces la función devuelve $2^{32} - 1$.

En esta funcion se habla del "vécino j-ésimo".

Con esto nos referimos al vértice que es el j-ésimo vécino del vértice en cuestión donde el orden del cual se habla es el orden en el que ustedes hayan guardados los vecinos de un vértice en G, con el índice 0 indicando el primer vécino, el índice 1 el segundo, etc.

Este orden NO ESTA ESPECIFICADO, y un grupo puede tener un orden y otro grupo otro, asi que el retorno para valores individuales de estas funciones no será el mismo para un grupo que para otro, y no será necesariamente igual al retorno de MIS funciones.

WAIT, WHAT?

Como se puede testear esta funciones entonces? Y para que sirve una función que da valores distintos dependiendo de la implementación?

Esta función existe pues Greedy necesita **iterar sobre todos los vecinos**, y porque si queremos testear si la estructura del grafo esta bien guardada, necesitamos una función que nos permita "ver"cómo estan guardados los vecinos en esa estructura.

Para poder iterar sobre todos los vecinos necesitamos primero saber cuales "son"esos vecinos. Lo mas fácil es una función que nos diga en que lugar en un orden "estandar"está cada vecino, y como estamos usando Orden Natural como orden estandard, por eso pedimos esta función.

Como el uso para el cual esta funcion esta pensada es para poder iterar sobre todos los vecinos de un vértice, el orden interno con el cual son guardados esos vecinos no es relevante., aunque se asume que una vez que terminaron con la función ConstruccionDelGrafo(), este orden queda fijo.

IMPORTANTE: Esta función debe ser $O(1)$.

Como va a ser usada para iterar sobre todos los vecinos, probablemente dentro de un loop que ademas itere sobre todos los vértices, es **CLAVE** que esta función sea $O(1)$ pues de lo contrario nada va a poder funcionar a una velocidad razonable. Asi que la estructura que armen del grafo debe ser tal que esta función sea $O(1)$ y no tenga que hacer una iteración para ser calculada.

7 Consideraciones finales para esta primera etapa

En esta etapa, la mayoria de las funciones son muy fáciles si piensan primero bien la estructura con la cual van a cargar el grafo.

Observar que no hay funciones que modifiquen la estructura interna del grafo, es decir, una vez construida la estructura, queda estática. Las cosas mas dificiles de esta primera etapa son:

- Definir la estructura en forma adecuada para que las funciones de extracción de información sean $O(1)$.
- Programar en forma eficiente la construcción del grafo. Algunos grafos tendrán millones de vértices, por lo tanto una construcción que sea $O(n^2)$ no terminará de cargar el grafo en ningún tiempo razonable. No es necesario que sea hipereficiente, pues la construcción del grafo se hace una sola vez, mientras que la lectura de los datos múltiples veces, pero no puede ser tan ineficiente que demore horas o dias en cargar un grafo.