Proyecto de Matemática Discreta II-2020

Contents

1 Introducción

La idea del proyecto es implementar algo enseñado en clase en un lenguaje, observando las dificultades de pasar de algo teórico a un programa concreto.

Otros objetivos son práctica de testeo de programas y práctica en programar funciones adhiriendose a las especificaciones de las mismas.

El algoritmo elegido es Greedy iterado y el lenguaje es C. (C99, i.e., pueden usar comentarios // u otras cosas de C99).

La idea NO ES presentar un programa completo que haga eso, sino las funciones detallas abajo, que luego se pueden ensamblar en un main que las use.

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. Pero hay errores que no deberían quedar en el código que me entreguen, porque son errores que son fácilmente detectables con un mínimo de sentido común a la hora de testear.

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 va a usar.

Usaremos varios mains para testear, pero uno de ellos cargará el grafo y luego realizará diversos reordenamientos de los vértices usando las funciones de la sección 4 para ir corriendo Greedy luego de cada una con la esperanza de obtener al final un número que se acerce bastante a $\chi(G)$. Su programación de Greedy y los reordenes debe ser tal que podamos hacer 1000 de estas iteraciones en un tiempo razonable. (no mas de 15 minutos en una máquina como las de los labs de Famaf para los grafos mas grandes de testeo, que estarán en mi página).

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

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.

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

1.1 Fechas de Entrega

Hay diversas opciones. En los tres primeros casos sólo deben entregar un subconjunto de las funciones, como se especifica mas abajo en el documento. En el cuarto deben entregar todo.

1. Jueves 23 de Abril de 2020, hasta las 14:00 hs. En este caso la nota del proyecto es entre 0 y 10.

2. Luego de esa fecha hasta el 7 de mayo, con un descuento de medio punto por cada día (contado hasta las 14:00 hs) que entreguen tarde, hasta un máximo de 6 puntos de descuento. Observen que en este último caso cualquier error que tengan provoca que el proyecto sea desaprobado.

3. El dia del examen, si rinden hasta agosto incluido. Aunque entreguen antes el proyecto no se corrige a menos que se presenten al examen. La nota máxima es 4 y cualquier error serio hace que desaprueben el proyecto y el examen.

4. El dia del examen, si rinden luego de agosto. Vale lo mismo que en el item anterior, pero ademas deben entregar TODAS las funciones detalladas en este documento, mas el archivo GrafoSt2020.h y todo otro .h que necesiten.

Si eligen las opciones 1 o 2 y el proyecto no es aprobado, siempre pueden luego optar por las opciones 3 o 4.

1.2 Entrega

Deben entregar vía e-mail a matematicadiscretaii.famaf arroba gmail.com los archivos que implementan el proyecto.

Para evitar demasiados descuentos de puntos en lugares en los que al parecer año tras año los estudiantes cometen errores (en especial con la construcción del grafo), este año sólo pediremos un núcleo reducido de funciones que corregiremos, si

entregan hasta las fechas de examen de agosto 2020 incluidas. (como se dijo antes, si entregan despues de las fechas de agosto, deben entregar todo lo especificado en este documento).

Sólo deben entregar para ser corregidas las funciones descriptas en las secciones 3, 4 y 5.

Para que estas funciones trabajen correctamente, las otras funciones descriptas en este documento son necesarias, pero nosotros usaremos para testear nuestras propias funciones.

Obviamente uds. tambien deberán implementar esas otras funciones, pues de lo contrario les va a ser imposible testear las que entreguen, pero no deben entregarlas y por lo tanto cualquier error que tengan en ellas no les descontará puntos, ademas que pueden poner los comentarios que quieran o nombrar las variables como se les ocurra, pues nosotros no las veremos. Claro que si el error es tal que les oculta un error de las funciones que si deben entregar estarán en problemas.

Otra ventaja de esto es que practicarán programar usando especificaciones: dado que no sabrán cómo estarán programadas nuestras funciones, ustedes deben programar las funciones que deben entregar de forma tal que puedan interactuar con cualquier función de las que no entreguen que cumpla con las especificaciones.

Para minimizar errores de tipeo también estará en la página el archivo veinteveinte.h que tendrá las declaraciones de todas las funciones detalladas en este documento. Ese archivo tendrá un include de otro archivo, GrafoSt2020.h en donde estará detallada la estructura del grafo que nosotros usaremos. Uds. no tendrán acceso a ese archivo ni a esa estructura mas que a través de las funciones.

Claramente uds. deberán construir su propio GrafoSt2020.h, pero no deben entregarlo, salvo que entreguen el proyecto luego de agosto 2020. Deben entregar cualquier .h que sea necesario para que las funciones que uds. entreguen de las secciones 3, 4 y 5 funcionen, pero no es necesario que entreguen los archivos .h que uds. necesiten para hacer que sus versiones de las funciones de las otras secciones funcionen.

1.3 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). (observación obvia: si entregan hasta agosto, pueden usar lo que quieran en las funciones que no entreguen, pero deben estar seguros que las que entreguen funcionen sin esas cosas extras. Lo mejor seria no usarlas para asegurarse que no hay problemas).

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 tener archivos ni directorios que tengan un espacio en el nombre.

6. No pueden usar getline (esta restricción en realidad es sólo para los que entreguen luego de agosto, pues los que entreguen antes pueden usarla en ConstruccionDelGrafo ya que no me voy a enterar).

7. Pueden consultar con otros grupos, pero si vemos copiado de grandes fragmentos de código la van a pasar mal, especialmente si descubrimos intento de engañarnos haciendo cambios cosméticos en el código de otro grupo. Obviamente este año los que entreguen antes de agosto pueden copiar completamente de otro grupo (si ese grupo los deja) las funciones que no deben entregar, pues no las veremos. De hecho, quizas una buena táctica sea hacer una "alianza"entre grupos, donde algunos miembros de todos esos grupos se concentran en codear en conjunto las funciones que no deben entregar, mientras que el resto de los miembros de cada grupo, (por separado) codean las otras funciones.

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

1.4 Formato de Entrega

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

No debe haber ningun ejecutable.

Los .c que entreguen deben hacer un include de veinteveinte.h y, obviamente, de cualquier otro .h que uds necesiten, los cuales deben ser entregados.

Para testear deberán hacer por su cuenta uno o mas .c que incluyan un main que les ayude a testear sus funciones. Estos archivos NO deben ser entregados.

Empaqueten todo lo detallado abajo en formato .tgz o tar.gz y envienlo por mail a:

matematicadiscretaii.famaf arroba gmail.com

La estructura de lo que entreguen debe ser de la siguiente forma:

Debe haber un directorio de primer nivel cuyo nombre consiste en los apellidos de los integrantes.

En ese directorio debe haber un archivo ASCII con los nombres y los mails de contacto de todos los integrantes del grupo.

Tambien pueden poner cualquier información extra que quieran agregar.

El formato de esa información extra debe ser .pdf o un archivo ASCII. Si entregan un .docx el proyecto se devuelve sin corregir y tienen 1 punto de descuento.

Ademas:

- Para los que entreguen hasta agosto: En ese directorio habrá un directorio de segundo nivel, llamado CangrejoEstelar

  En este directorio deben estar todos los archivos que necesiten para implementar las funciones de las secciones 3, 4 y 5, incluidos archivos auxiliares, pero no debe haber ningun main ni ningún archivo con sus definiciones de las funciones que no sean de esas secciones, ni un veinteveinte.h **Todo .c que este en este directorio se considera que debe formar parte de la compilación.** (usaremos *.c en CangrejoEstelar para compilar).

  Compilaremos (con mains nuestros) desde el directorio de primer nivel con gcc, -Wall, -Wextra, -O3, -std=c99 . Tambien haremos -I al directorio donde tengamos nuestra copia de veinteveinte.h y si mandan algún .h en CangrejoEstelar entonces haremos tambien -ICangrejoEstelar .

  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.

- LOS QUE ENTREGUEN LUEGO DE AGOSTO: ademas de CangrejoEstelar (ver lo que deben entregar los que entregan hasta agosto) y lo que debe haber ahi, debe haber otro directorio de segundo nivel, con el nombre que quieran, donde esten todas las demas funciones mas una copia de veinteveinte.h y su versión de GrafoSt2020.h. No debe haber ningún main ni ejecutable.

  Primero compilaremos de la misma forma que haremos con los que entregan hasta agosto, para testear las funciones de CangrejoEstelar con nuestras funciones. Si no detectamos errores ahi, entonces luego compilaremos todas sus funciones (con mains nuestros) desde el directorio de primer nivel con gcc, -Wall, -Wextra, -O3, -std=c99 haciendo -I al directorio donde tengan su copia de veinteveinte.h y GrafoSt2020.h y si mandan algún .h en CangrejoEstelar entonces haremos tambien -ICangrejoEstelar . Al igual que hasta agosto, tambien podemos usar para algunos grafos una compilación con -fsanitize=address,undefined o -DNDEBUG.

Luego de enviado, se les responderá con un "recibido". Si no reciben confirmación dentro de las 24hs pregunten si lo recibí. (eso ahora, en la epoca de entrega o bien en epoca de examenes. Si lo envian luego, pej en el segundo cuatrimestre, puedo tomarme mucho mas, como pej un par de meses para contestar)

Dependiendo de los errores, podemos darles la nota definitiva del proyecto luego de corregido, o bien podemos devolverles el proyecto para que corrijan algunos errores antes de fijar la nota, si es que consideramos estos errores como serios pero no lo suficientemente serios como para desaprobar el proyecto. Si los errores son muy graves, el proyecto queda desaprobado.

2 Tipos de datos

Los dos primeros tipos de datos serán definidos por nosotros en GrafoSt2020.h al cual uds. no tendrán acceso. veinteveinte.h hace un include de ese .h. Obviamente, para que el resto de sus funciones puedan compilar, uds. deberán definirlos para testeo interno, pero no deben entregar el .h donde realicen esas definiciones.

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.

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 contendrá el número de vertices y lados,los nombres, grados y colores de todos los vertices y el Delta del grafo (el mayor grado). Ademas cómo Greedy usa un orden de los vertices, esta estructura tendrá guardado algún orden de los vertices. Cómo vamos a cambiar ese orden repetidamente, será algo que puede ser cambiado. (ver secciones 4 y 10). Tambien estará guardado de alguna forma quienes son los vecinos para cada vertice.

2.3 Grafo

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

3 Funciones de coloreo. (entrega requerida)

Estas funciones cambian el coloreo de los vértices del grafo. Obviamente deberán hacer uso extensivo de la función FijarColor definida en la sección 10 y las funciones para obtener información de los vértices y sus vecinos dadas en las secciones 8 y 9 asi como las de 7

3.1 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).

3.2 Bipartito()

Prototipo de función:

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

Devuelve 1 si W es bipartito, 0 si no.

Ademas, si devuelve 1, colorea W 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).

3.3 SwitchColores()

Prototipo de función:

```
char SwitchColores(Grafo G,u32 i,u32 j);
```

Verifica que $i, j <$número de colores que tiene G en ese momento. Si no es cierto, retorna 1. Si ambos estan en el intervalo permitido, entonces intercambia los colores i, j: todos los vertices que esten coloreados en el coloreo actual con i pasan a tener el color j y los que estan coloreados con j pasan a tener el color i. Retorna 0 si todo se hizo bien.

Esta función permite obtener nuevos reordenamientos en combinación con la 4.2.

4 Funciones de ordenación. (entrega requerida)

Estas funciones cambian el orden interno guardado en G que se usa para correr Greedy.

Obviamente deberán hacer uso extensivo de la función FijarOrden definida en la sección 10 ademas de otras funciones de otras secciones.

Es estas funciones se habla de retornar 1 "si hubo algún problema.". Pej, si allocan memoria extra temporaria para realizar esa función y el alloc falla, deben reportar 1

4.1 WelshPowell()

Prototipo de función:

```
char WelshPowell(Grafo G);
```

Ordena los vertices de W de acuerdo con el orden Welsh-Powell, es decir, con los grados en orden **no creciente**.

Por ejemplo si hacemos un for en i de Grado(i,G) luego de haber corrido WelshPowell(G), deberiamos obtener algo como 10,9,7,7,7,7,5,4,4.

El orden relativo entre vértices que tengan el mismo grado no se especifica, es a conveniencia de ustedes.

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

4.2 RevierteBC()

Prototipo de función:

```
char RevierteBC(Grafo G);
```

Revierte los bloques de colores: Si G esta coloreado con r colores y V_0 son los vertices coloreados con 0, V_1 los vertices coloreados con 1, V_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de V_{r-1}, luego los de V_{r-2}, luego los de V_{r-3}, etc, hasta V_0. En otras palabras, para todo $i = 0, .., r-2$ los vertices de V_i estan todos luego de los vertices de V_{i+1}. (el orden entre los vértices de cada V_i no se especifica).

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

Esta función se puede usar sola o en combinación con 3.3 para obtener nuevos ordenamientos que respeten los bloques de colores.

4.3 ChicoGrandeBC()

Prototipo de función:

```
char ChicoGrandeBC(Grafo G);
```

Ordena por cardinalidad de los bloques de colores, de menor a mayor:

Si G esta coloreado con r colores y V_0 son los vertices coloreados con 0, V_1 los vertices coloreados con 1, etc, entonces esta función ordena los vertices poniendo primero los vertices de V_{j_1}, luego los de V_{j_2} , etc, donde $j_1, j_2, ..., j_r$ son tales que $|V_{j_1}| \leq |V_{j_2}| \leq ... \leq |V_{j_r}|$. (el orden entre los vértices de cada V_i no se especifica).

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

4.4 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.

El orden no será 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.

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

Idealmente la función debe ser tal que para cualquier orden posible de los vértices exista algún R tal que AleatorizarVertices(G,R) obtenga ese orden: No les pido tanto, pero si les pido 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 aceptable. Una función que, pej, en el 90% de los casos hace que el primer vértice sea 4 tampoco es aceptable.

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

5 Función de cálculo de número de componentes conexas. (entrega requerida)

Tenemos una sola funcion en esta seccion:

5.1 NumCCs()

Prototipo de función:

```
u32 NumCCs(Grafo G);
```

Devuelve el número de componentes conexas de G.

==========

Fin de funciones requeridas para entregas hasta agosto 2020

==========

Las siguientes funciones serán implementadas por nosotros y su entrega no es requerida, salvo que entreguen el proyecto luego de agosto 2020. Obviamente deberán implementarlas para testear las funciones anteriores, pero cualquier error en su implementación no será visto por nosotros.

6 Funciones De Construcción/Destrucción/Copia del grafo

6.1 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 11, lo carga en la estructura, incluyendo algún orden de los vertices, corre Greedy con ese orden para dejar todos los vertices coloreados 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 11 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.

6.2 DestruccionDelGrafo()

Prototipo de función:

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

Destruye G y libera la memoria alocada.

6.3 CopiarGrafo()

Prototipo de función:

```
Grafo CopiarGrafo(Grafo G);
```

La función aloca memoria suficiente para copiar todos los datos guardados en G, hace una copia de G en esa memoria y devuelve un puntero a esa memoria.

En caso de no poder alocarse suficiente memoria, la función devolverá un puntero a `NULL`.

Esta función se puede usar (y la usaremos asi en un main) para realizar una o mas copias de G, intentar diversas estrategias de coloreo por cada copia, y quedarse con la que de el mejor coloreo.

7 Funciones para extraer información de datos del grafo

Las funciones detalladas en esta seccion serán $O(1)$.

7.1 NumeroDeVertices()

Prototipo de función:

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

Devuelve el número de vértices de G.

7.2 NumeroDeLados()

Prototipo de función:

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

Devuelve el número de lados de G.

7.3 Delta()

Prototipo de función:

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

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

8 Funciones para extraer información de los vertices

Las funciones detalladas en esta sección serán $O(1)$. (las que programo yo son $O(1)$ y si ustedes entregan luego de agosto, la funciónes que entreguen deben ser $O(1)$. Si entregan antes y las funciones que uds programen no son $O(1)$ yo no me voy a enterar, pero a ustedes los tests les van a demorar mas).

8.1 Nombre()

Prototipo de Función:

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

Devuelve el nombre real del vértice número i en el orden guardado en ese momento en G, (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

Esta función no tiene forma de reportar un error (que se produciría si i es mayor o igual que el número de vértices), asi que debe ser usada con cuidado.

8.2 Color()

Prototipo de Función:

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

Devuelve el color con el que está coloreado el vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc).

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

8.3 Grado()

Prototipo de Función:

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

Devuelve el grado del vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc).

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 hace inmanejable).

9 Funciones para extraer información de los vecinos de un vértice

En las funciones de esta sección se habla del "vécino número j". Con esto nos referimos al vértice que es el j-ésimo vécino del vértice en cuestión donde el orden que usamos es el orden en el que esten guardados los vecinos en G, con el índice 0 indicando el primer vécino, el índice 1 el segundo, etc. Este orden NO ESTA ESPECIFICADO y es irrelevante. Estas funciones estan pensadas para poder iterar sobre TODOS los vecinos de un vértice y por eso el orden interno de esos vecinos no es relevante.

9.1 ColorVecino()

Prototipo de función:

`u32 ColorVecino(u32 j,u32 i,Grafo G);`

Devuelve el color del vécino número j del vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

El orden de LOS VERTICES guardado en G es el orden en que supuestamente vamos a correr Greedy, pero como dijimos arriba, el orden DE LOS VECINOS es irrelevante, lo importante es que de esta forma se puede iterar sobre los vecinos para correr Greedy o realizar tests. (por ejemplo, para saber si el coloreo es o no propio).

Para que quede claro: si el orden en el que vamos a correr Greedy es 4,10,7,15,100 y los vecinos de 7 son 10,100,4,15 (en ese orden) y los de 15 son 4,7,100 (en ese orden) y el color de 4 es 1, el de 7 es 2, el de 10 es 3, el de 15 es 4 y el de 100 es 0, entonces:

- ColorVecino(0,2,G)=3 (pues 10 es el primer vécino de 7)
- ColorVecino(2,2,G)=1 (pues 4 es el tercer vécino de 7)
- ColorVecino(0,3,G)=1 (pues 4 es el primer vécino de 15)
- ColorVecino(1,3,G)=2 (pues 7 es el segundo vécino de 15)

Si i es mayor o igual que el número de vértices o j es mayor o igual que el número de vecinos del vértice i, devuelve $2^{32}-1$.

9.2 NombreVecino()

Prototipo de función:

```
u32 NombreVecino(u32 j,u32 i,Grafo G);
```

Devuelve el nombre del vécino número j del vértice número i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

Esta función no tiene forma de reportar un error (que se produciría si i es mayor o igual que el número de vértices o j es mayor o igual que el número de vecinos de i), asi que debe ser usada con cuidado.

9.3 OrdenVecino()

Prototipo de función:

```
u32 OrdenVecino(u32 j,u32 i,Grafo G);
```

OrdenVecino(j,i,Grafo G) es igual a k si y sólo si el vécino j-ésimo del i-ésimo vértice de G en el orden interno es el k-ésimo vértice del orden interno de G.

10 Funciones para modificar datos de los vértices

10.1 FijarColor()

Prototipo de función:

```
char FijarColor(u32 x,u32 i,Grafo G);
```

Si i es menor que el número de vértices, le asigna el color x al vértice número i en el orden guardado en ese momento en G y retorna 0.

De lo contrario, retorna 1.

Esta función debe ser usada con cuidado pues puede provocar que queden colores no asignados a ningún vértice (pej, dando un coloreo con colores 0,1,4,7)

10.2 FijarOrden()

Prototipo de función:

```
char FijarOrden(u32 i,Grafo G,u32 N);
```

Si i y N son menores que el número de vértices, fija que el vértice i en el orden guardado en G será el N-ésimo vértice del orden "natural"(el que se obtiene al ordenar los vértices en orden creciente de sus "nombres" reales) y retorna 0. De lo contrario retorna 1.

Esta función debe ser usada con cuidado pues puede provocar que quede un orden interno que no sea un orden de todos los vértices o que haya vértices repetidos en el orden.

11 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).
- 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) en teoría representa el número de vértices y el segundo (m) el número de lados, pero 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. Sin embargo, **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)

- 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 esta función 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. De todos modos, este año si tienen errores aca no los veré salvo que entreguen luego de agosto.

  En un formato válido de entrada habrá al menos m lineas comenzando con e, pero puede haber algún archivo de testeo (que usaré sólo para los que entreguen luego de agosto) en el cual no haya al menos m lineas comenzando con e. En ese caso, como se especifica en 6.1, debe detenerse la carga y devolver un puntero a NULL. O por ejemplo tambien testearé para los que entreguen luego de agosto con archivos donde en vez de p edge n m tengan pej p edgee n m.

  Ejemplo tomado de la web:

  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 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.

- Nunca fijaremos $m = 0$, es decir, siempre habrá al menos un lado. (y por lo tanto, al menos dos vértices).

- 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

12 Algunos Errores que pueden ocurrir y sus consecuencias

12.1 Cosas que causan desaprobación automática

Las siguientes cosas provocan desaprobación automática no sólo por los errores en si, sino porque son cosas fácilmente chequeables, asi que no se justifica entregar un proyecto sin haberlas chequeado. Esto no quiere decir que si no ocurre ninguna esten aprobados, pueden desaprobar por otras cosas, en particular todos los años hay algún grupo que encuentra una nueva forma de meter la pata en forma espectacular de una forma que no se me habia ocurrido.

1. En mi página habrá muchos grafos de ejemplos. Un subconjunto de ellos serán declarados de testeo obligatorio, es decir, su programa debe si o si procesarlos bien, sin producir segmentation faults, stacks overflows o alguna otra cosa rara. Si algo asi ocurre, quedan desaprobados.

2. No es capaz de leer un grafo y hacer 1000 reordenamientos y Greedys en un tiempo "razonable": alrededor de 15 minutos en una máquina como las de Famaf, mas o menos, para los grafos mas grandes de la página. Mi programa puede hacer 3000 en menos de 5 minutos, pero no pido tanto. Pero un par de horas no es razonable y una semana, menos. En particular he tenido alumnos que incluso con grafos con sólo un par de miles de lados demoraban mas de una hora en hacer UN Greedy. El grafo de entrada puede tener millones de vertices y lados, ver mi pagina para ejemplos.

3. En los reordenes por bloque de colores, la cantidad de colores que se obtiene al correr Greedy luego del reorden aumenta respecto de la cantidad de colores que tenia antes de ese reordenamiento. En clase demostramos que esto no puede pasar, y es algo fácilmente testeable. Es un error común, lo cual muestra que muchos grupos no testean adecuadamente.

4. Greedy o Bipartito dan coloreos no propios. Por ejemplo, en años anteriores hubo grupos que coloreaban K_n con menos de n colores. Esto es fácilmente testeable si hacen una función que testee si el coloreo que tiene el grafo es propio o no. (hubo un grupo que coloreaba K_n con MAS de n colores. En ese caso el coloreo era propio pero habia colores que no coloreaban a ningún vértice. Esto último no es desaprobación automática, pero si gran descuento de puntos).

5. Greedy da 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 otros no. En mi página hay varios ejemplos de grafos para los cuales Greedy debe dar distinto número de colores dependiendo del orden).

6. Bipartito dice que alguno de los grafos bipartitos de mi página no es bipartito.

   Bipartito puede tener un error sutil que haga que diga que todos los grafos bipartitos de mi página son bipartitos pero que con algún otro grafo bipartito falle en decir que es bipartito. En este caso no es desaprobación automática pues es mas dificil de detectar.

   Observar que en el caso opuesto de que Bipartito diga que es bipartito un grafo que no es bipartito, sea de mi página o no, entonces es desaprobación por el punto anterior, dado que el coloreo que les va a dar no es propio.

7. Su programa funciona bien con mains que no usen CopiarGrafo, pero no con mains que usen CopiarGrafo. Pej, si usan una variable global en este programa es muy probable que sean unos fucking idiots, dada la existencia de la función CopiarGrafo, pero muchos insisten en usar una, especialmente con las funciones de reorden. Puede pasar que se les ocurra una forma de usar una variable global que funcione bien aun si hacen múltiples instancias de CopiarGrafo pero dudo que sea la mejor forma de programar esas funciones y lo mas probable es que haga que el programa crashee.

8. Dada la existencia de una función que debe calcular el número de componentes conexas, asumir que el grafo es conexo es una fucking idiocy y si su programa funciona bien con grafos conexos pero no para grafos desconexos están desaprobados.

9. El programa no compila, aunque en este caso pueden aprobar (con gran descuento de puntos) dependiendo de cual es la razón por la cual no compile y si puedo reparar el error rápidamente. Pero en principio estan desaprobados.

10. AleatorizarVertices no da una biyección. Es decir, luego de correr AleatorizarVertices, el "orden"que queda no es un orden de todos los vértices. (pej, aparece dos veces un vértice y otro no aparece).

11. Si entregan luego de agosto, desaprueban si su ConstruccionDelGrafo no es capaz de leer archivos en el formato establecido. Debe poder leer si se carga un grafo a mano por stdin pero tambien usando el operador de redirección "<". Pej, si el archivo ejecutable luego de compilar es "ejec"entonces algo como ./ejec <KC debe funcionar si KC es un archivo.

12.2 Cosas que no causan desaprobación automática pero si descuento de puntos y/o devolución del proyecto

1. El programa anda bien con grafos chicos y medianos, pero produce un stack overflow en algunos grafos grandes que no son los de testeo obligatorio. 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. Bipartito funciona bien con los grafos de mi página pero tiene algún error que hace que para otros grafos bipartitos diga que no lo son.

3. Retorna mal el número de componentes conexas.

4. AleatorizarVertices da una biyección pero la aleatoriedad no es buena.

12.3 Cosas que pueden o no causar desaprobación del proyecto

Pero ciertamente causarán descuento de puntos.

1. Los ordenes no ordenan correctamente. Pej, que WelshPowell ordene al revés de lo que se pide, no va a causar desaprobación pero si descuento, por no testearlo. En el caso de RevierteBC y ChicoGrandeBC puede haber dos tipos de errores: que los vértices queden ordenados por bloque de colores, pero el orden de esos bloques esté mal, o bien que directamente los vértices no esten ordenados por bloque de colores. El primer caso no causa desaprobación pero el segundo muy probablemente si, porque es muy posible que cause aumento del número de colores.

2. Buffer overflows o comportamiento indefinido. Se evaluará la gravedad de los mismos.

3. Un memory leak. Puede no provocar desaprobación automática si no es muy grave pero hubo un grupo que para grafos grandes agregaba algo asi como 10MB de RAM por cada corrida de Greedy y no la liberaba nunca.

4. Hay alguna interacción defectuosa entre las funciones que hace que con algunos mains salten errores y con otros no. Por ejemplo, en años anteriores algunos grupo entregaron proyectos en donde Bipartito funcionaba de forma distinta dependiendo si se hacia antes o despues de WelshPowell. Este tipo de errores será evaluado caso por caso.

5. El código es complicado y difícil de leer. Recuerden KISS: Keep It Simple (and) Stupid. Si hacen algo complicado deberian agregar una cantidad suficiente de comentarios. Para entregas luego de agosto, deben comentar adecuadamente las estructuras que usen en GrafoSt2020.h. Si un proyecto se me hace dificil de entender, pasa al fondo de la pila de correcciones y puedo descontar puntos o desaprobarlo por inentendible.

6. Los nombres de las variables o funciones auxiliares violan el sentido común. Pej, una función que se llame OrdenarVertices pero en realidad ordene lados o colores. O, para denotar el número de vértices, en vez de usar algo descriptivo como n, nv, nvertices o incluso algo neutral como "Superman", usan algo que induce a error como nlados, ncolores o m.

7. Usan "demasiada"RAM: no diré un límite definitivo y como este año la construcción del grafo será hecha con mi propia función, no preveo mayores dificultades en este sentido pero pej, podrian querer hacer una estructura auxiliar para los reordenes que sea demasiado grande. Si lo que hacen es una desmesura de GBs se les devolverá el proyecto para que lo corrijan, y si es demasiado extravagante, los puedo desaprobar directamente por no tener sentido común. Pej, un grupo necesitaba mas de un TByte de memoria para grafos grandes.

8. Variables shadows. (pueden no provocar desaprobación si es algo que no causa problemas).