Proyecto de Matemática Discreta II-2019

1 Introducción

Deberán implementar lo que se detalla a continuación en C (C99, i.e., pueden usar comentarios // u otras cosas de C99).
El proyecto puede ser hecho en forma individual o en grupos de 2 personas.

Contents

1.1 Restricciones generales

El código debe ser razonablemente portable. Ademas:
1) No pueden usar getline
2) No pueden usar archivos llamados aux.c o aux.h
3) No pueden tener archivos tales que la unica diferencia en su nombre sea diferencia en la capitalización.
4) No pueden llamar a ninguna libreria que no sean las del estandard de C99.
5) No pueden tener archivos ni directorios que tengan un espacio en el nombre.

Pueden consultar con otros grupos, pero si vemos copiado de grandes fragmentos de código la van a pasar mal. Y la van a pasar peor si descubrimos intento de engañarnos haciendo cambios cosméticos en el código de otro grupo.

2 Entrega

Deben entregar vía e-mail los archivos que implementan el proyecto. Los archivos del programa deben ser todos archivos .c o .h. No debe haber ningun ejecutable. Si entregan un ejecutable el proyecto se devuelve automáticamente sin corregir.

2.1 Fecha de Entrega

Martes 23 de abril de 2018, hasta las 09:00 hs. (9AM).

2.2 Archivos del programa

Deben entregar un archivo Rii.h en el cual se hagan las llamadas que uds. consideren convenientes a librerias generales de C, ademas de las especificaciones de los tipos de datos y funciones descriptas en este documento. Tengan cuidado de poner los nombres correctos, pues ese será el archivo que incluiremos en nuestros algoritmos de testeo, el cual usará los nombres dados acá, asi que si los nombres no son los correctos, el programá no compilará y serán desaprobados.

No es necesario que todo este definido en Rii.h, pero si definen algo en otros archivos auxiliares la llamada a esos archivos auxiliares debe estar dentro de Rii.h. En Rii.h debe figurar (comentados, obvio) los nombres y los mails de contacto de todos los integrantes del grupo.

Junto con Rii.h obviamente deben entregar uno o mas archivos .c que implementen las funciones indicadas en este documento, pero pueden nombrarlos como quieran.

Deben empaquetar todo de la siguiente forma:

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

En ese directorio deben poner cualquier información extra que quieran agregar. Pueden no poner nada, pero si lo hacen el formato debe ser .pdf o un archivo ASCII. Si entregan un .docx el proyecto se devuelve sin corregir.

Ademas, en ese directorio habrá un directorio de segundo nivel, llamado Wahlaan

En este directorio deben estar Rii.h y todos los otros archivos que necesiten, incluidos archivos auxiliares, pero no debe haber ningun main. **Todo .c que este en este directorio se considera que debe formar parte de la compilación.** (usaremos *.c en Wahlaan para compilar).

(Rii significa “Esencia”, Wahl es un verbo que significa “Crear/Construir”y Wahlaan es la forma pasada de ese verbo “Construido/Creado”).

Empaqueten todo en formato .tgz y envienlo por mail a:

matematicadiscretaii.famaf arroba gmail.com

Compilaremos (con mains nuestros) desde el directorio de primer nivel con gcc, -IWahlaan , -Wall, -Wextra, -O3, -std=c99 .

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.

2.3 Protocolo

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. Si lo envian luego, pej en el segundo cuatrimestre, puedo tomarme mucho mas para contestar)

2.4 Nota del Proyecto, Errores del programa y retorno

El Proyecto será evaluado con una nota entre 0 y 10, la cual sera promediada por medio de un promedio pesado con las notas del Teórico y del Práctico para obtener la nota final. Las tres partes deben aprobarse por separado, asi que si no aprueban el proyecto no aprueban el final.

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. Toda devolución de proyecto implica descuento de puntos.

Si los errores son muy graves, el proyecto queda desaprobado.

2.5 Que pasa si el proyecto no es aprobado

Si el proyecto no es aprobado (o no lo entregan) tienen un "recuperatorio":pueden entregar el proyecto hasta el mismo dia del examen que quieran rendir, pero las notas posibles son solo 0,1,2,3 si desaprueban esa versión o 4 si la aprueban. El proyecto no se corrige a menos que efectivamente se presenten al examen y si no aprueban esa version no aprueban el final. Esta opción no esta disponible luego de la primera fecha de febrero de 2020. (es decir, para la segunda fecha de febrero o las fechas de marzo en adelante, ya no tienen esta opción, pero en la primera de febrero si). Si aprueban el proyecto pero no el examen (pues no aprueban el teórico o el práctico) el proyecto queda aprobado, no es necesario que lo reentreguen cuando vuelvan a rendir.

Para las fechas de diciembre y febrero, deberán ademas entregar un archivo .c con un main que se especificará mas adelante. (ver mi pagina en agosto, luego de las fechas de examen de julio-agosto)

2.6 Duración de la aprobación

En principio, la aprobación del proyecto, ya sea durante la entrega normal o aprobado durante un examen, caduca en abril del 2021, pero es posible que la extienda a abril del 2022.

3 Tipos de datos

3.1 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 seran todos u32.

3.2 GrafoSt

Es una estructura, la cual debe contener toda la información sobre el grafo necesaria para correr su implementación. La definición interna de la esta estructura es a elección de ustedes y deberá soportar los métodos que se describirán más adelante, más los métodos que ustedes consideren necesarios para implementar los algoritmos que esten implementando.

Entre los parametros debe haber como mínimo los necesarios para guardar los datos de un grafo (vertices y lados) pero ademas los necesarios para guardar el coloreo que se tiene hasta ese momento en el grafo y cualquier información requerida en los algoritmos a implementar.

IMPORTANTE: Cómo Greedy usa un orden de los vertices, en esta estructura tiene que estar guardado algún orden de los vertices, y cómo vamos a cambiar ese orden repetidamente, debe ser algo que pueda ser cambiado.

El coloreo siempre debe cumplir que si es un coloreo con q colores **entonces los colores son** $0, 1, ..., q-1$. (en años anteriores pediamos colores entre 1 y la cantidad de colores, pero dado que los arrays en C comienzan en 0, algunos alumnos cometieron errores. Para minimizar la posibilidad de errores pedimos de esta forma este año).

3.3 Grafo

es un puntero a una estructura de datos *GrafoSt*.

4 Funciones De Construcción/Destrucción/Copia 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 9, 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`, liberando cualquier memoria que haya alocado. Errores posibles pueden ser fallas en alocar memoria, pero también que el formato de entrada no sea válido. En la sección 9 se especifica el formato. Entre otras cosas la primera linea que no sea comentario empieza con "p edge"y luego tiene indicados el número n de vértices y el número m de lados. Si la primera linea que no sea comentario no es asi debe imprimirse:

error en primera linea sin comentario

y retornar `NULL`.

A continuación de esa linea debe haber m lineas cada una de las cuales indica un lado. Si no hay AL MENOS m lineas de lados luego de esa, o si alguna de esas lineas no tiene el formato indicado en 9 o hay algún problema con la lectura de una de esas lineas, se debe imprimir

error de lectura en lado L

donde L es el número de linea donde se encuentra el error (en particular, si sólo hay ℓ lineas de lados, con $\ell < m$, ahi debe imprimirse "error de lectura en lado $\ell + 1$"(pues en realidad el lado $\ell + 1$ no existe).

Luego de ese mensaje de error, retornar `NULL`.

Si el total de vértices extraidos de esas m lineas no es n, debe imprimir

cantidad de vertices leidos no es la declarada

y luego devolver `NULL`.

Pueden imprimir otros mensajes de error para algún caso no contemplado acá, por ejemplo por la forma de su estructura interna, pero **si el formato es el correcto no deben imprimir nada**.

4.2 DestruccionDelGrafo()

Prototipo de función:

`void DestruccionDelGrafo(Grafo G);`

Destruye G y libera la memoria alocada.

4.3 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.

5 Funciones de coloreo

5.1 Greedy()

Prototipo de función:

`u32 Greedy(Grafo G);`

Corre greedy en G con el orden interno que debe estar guardado de alguna forma dentro de G. Devuelve el numero de colores que se obtiene.

5.2 Bipartito()

Prototipo de función:

`int Bipartito(Grafo G);`

Devuelve 1 si G es bipartito, 0 si no.

Si es bipartito, deja a G coloreado en forma propia con los colores 0 y 1.

Si devuelve 0, antes debe colorear a G con Greedy, en algún orden.

6 Funciones para extraer información de datos del grafo

Las tres funciones detalladas en esta seccion deben ser $O(1)$.

6.1 NumeroDeVertices()

Prototipo de función:

`u32 NumeroDeVertices(Grafo G);`

Devuelve el número de vértices de G.

6.2 NumeroDeLados()

Prototipo de función:

`u32 NumeroDeLados(Grafo G);`

Devuelve el número de lados de G.

6.3 NumeroDeColores()

Prototipo de función:

`u32 NumeroDeColores(Grafo G);`

Devuelve la cantidad de colores usados en el coloreo que tiene en ese momento G. (luego de ConstruccionDelGrafo, G siempre debe tener algún coloreo propio, salvo en el medio de la corrida de Bipartito o de Greedy)

7 Funciones de los vertices

Las cinco funciones detalladas en esta seccion deben ser $O(1)$.

7.1 NombreDelVertice()

Prototipo de Función:

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

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.

7.2 ColorDelVertice()

Prototipo de Función:

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

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, el grafo deberia tener al menos esa cantidad de vertices, lo cual lo hace inmanejable).

7.3 GradoDelVertice()

Prototipo de Función:

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

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).

7.4 ColorJotaesimoVecino()

Prototipo de función:

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

Devuelve el color del vécino numero j (en el orden en que lo tengan guardado uds. en G, con el índice 0 indicando el primer vécino, el índice 1 el segundo, etc)) 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)

Como en las otras funciones, el orden de LOS VERTICES guardado en G es el orden en que supuestamente vamos a correr Greedy, pero el orden DE LOS VECINOS que usen ustedes es irrelevante, lo importante es que de esta forma podemos iterar sobre los vecinos para 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:

- ColorJotaesimoVecino(G,2,0)=3 (pues 10 es el primer vécino de 7)
- ColorJotaesimoVecino(G,2,2)=1 (pues 4 es el tercer vécino de 7)
- ColorJotaesimoVecino(G,3,0)=1 (pues 4 es el primer vécino de 15)
- ColorJotaesimoVecino(G,3,1)=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 $2^{32}-1$. (esto nunca puede ser un color en los grafos que testeeemos, pues para que eso fuese un color, el grafo deberia tener al menos esa cantidad de vertices, lo cual lo hace inmanejable).

7.5 NombreJotaesimoVecino()

Prototipo de función:

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

Devuelve el nombre del vécino numero j (en el orden en que lo tengan guardado uds. en G, con el índice 0 indicando el primer vécino, el índice 1 el segundo, etc)) 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.

8 Funciones de ordenación

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

8.1 OrdenNatural()

Prototipo de función:

`char OrdenNatural(Grafo G);`

Ordena los vertices en orden creciente de sus "nombres" reales. (recordar que el nombre real de un vértice es un u32)
retorna 0 si todo anduvo bien, 1 si hubo algún problema.

8.2 OrdenWelshPowell()

Prototipo de función:

`char OrdenWelshPowell(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 GradoDelVertice(G,i) luego de haber corrido OrdenWelshPowell(G), deberiamos obtener algo como 10,9,7,7,7,7,5,4,4.

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

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

8.3 SwitchVertices()

Prototipo de función:

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

Verifica que $i, j <$número de vértices. Si no es cierto, retorna 1. Si ambos estan en el intervalo permitido, entonces intercambia las posiciones de los vertices en los lugares i y j del orden interno de G que G tenga en el momento que se llama esta función. (numerados desde 0), y retorna 0.

Por ejemplo, si el orden interno en ese momento es 10,15,7,22,25,17,4 entonces SwitchVertices(G,4,2) deja el orden interno 10,15,25,22,7,17,4.

8.4 RMBCnormal()

(el nombre son las iniciales de ReordenManteniendoBloqueColores, normal) Prototipo de función:

`char RMBCnormal(Grafo G);`

Si G esta coloreado con r colores y VC_1 son los vertices coloreados con 1, VC_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de VC_1, luego los de VC_2, etc, hasta VC_{r-1}. En otras palabras, para todo $i = 0, .., r-2$ los vertices de VC_i estan todos antes de los vertices de VC_{i+1}.

Retorna 0 si todo anduvo bien, 1 si hubo algún problema. (pej, si allocan memoria extra temporaria para realizar esta función y el alloc falla, deben reportar 1).

A diferencia de las dos funciones que siguen, esta función por si sola es inútil para bajar la cantidad de colores de algo coloreado con Greedy, pero se usará en conjunción con SwitchColores (ver 8.7)

8.5 RMBCrevierte()

(el nombre son las iniciales de ReordenManteniendoBloqueColores, revierte) Prototipo de función:

`char RMBCrevierte(Grafo G);`

Si G esta coloreado con r colores y VC_1 son los vertices coloreados con 1, VC_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de VC_{r-1}, luego los de VC_{r-2}, luego los de VC_{r-3}, etc, hasta VC_1. En otras palabras, para todo $i = 0, .., r-2$ los vertices de VC_i estan todos luego de los vertices de VC_{i+1}.

Retorna 0 si todo anduvo bien, 1 si hubo algún problema. (pej, si allocan memoria extra temporaria para realizar esta función y el alloc falla, deben reportar 1).

8.6 RMBCchicogrande()

(el nombre son las iniciales de ReordenManteniendoBloqueColores, chico-grande) Prototipo de función:

`char RMBCchicogrande(Grafo G);`

Si G esta coloreado con r colores y VC_1 son los vertices coloreados con 1, VC_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de VC_{j_1}, luego los de VC_{j_2} , etc, donde $j_1, j_2, ..., j_r$ son tales que $|VC_{j_1}| \leq |VC_{j_2}| \leq ... \leq |VC_{j_r}|$

Retorna 0 si todo anduvo bien, 1 si hubo algún problema. (pej, si se alloca memoria extra temporaria para realizar esta función y el alloc falla, deben reportar 1).

8.7 SwitchColores()

Esta función no reordena los vertices, pero permite obtener nuevos reordenamientos en combinación con las anteriores.

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 en el nuevo coloreo y los que estan coloreados con j en el coloreo actual pasan a tener el color i en el nuevo coloreo. Los demas colores quedan como están. Retorna 0 si todo se hizo bien.

9 Formato de Entrada

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

La descripción **oficial** de DIMACS es asi:

- Ninguna linea tiene mas de 80 caracteres. PERO hemos visto archivos DIMACS en la web que NO cumplen esta especificación, asi que NO la pediremos.

  Su código debe poder procesar lineas con una cantidad arbitraria de 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 primer 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. En nuestro caso, pediremos que n sea el número EXACTO de vértices. Todos los grafos "válidos"que nosotros usaremos para testear cumplirán que n será el número de vertices exacto, pero también usaremos algunos grafos "no válidos"para testear si su programa detecta que el formato no es correcto y devuelve un puntero a `NULL` en ese caso. (no cumplir con este requisito de detectar fallas no es motivo de desaprobación, ver 10.5).

- 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 deben **detener la carga sin leer ninguna otra linea**, 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. (en particular en otros años hubo grupos que hicieron código que no funcionaba si la ultima linea no era una linea en blanco. Esto es un error.)

  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.

  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, ustedes pueden asumir en su programa que si aparece el lado e v w NO aparecerá el lado e w v. No es necesario que detecten si esto se cumple o no: no testearemos archivos que tengan lados e v w y 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 $0, 1, 10, 15768, 1000000$. Ustedes deben pensar bien como van a resolver este problema.

- 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

10 Final Warnings

10.1 Advertencias generales

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

- Debe haber MUCHO COMENTARIO, las cosas deben entenderse.

  Si un proyecto se me hace dificil de entender, pasa al fondo de la pila de correcciones. Si se me hace muy díficil de entender, puede haber descuento de puntos. Si es muy inentendible, puede ser devuelto para que se corrija, o incluso ser desaprobado.

  Respecto de los nombres de las variables, sólo pido sentido común. Por ejemplo, para denotar el número de vértices pueden simplemente usar n, o bien una variable mas significativa como nvertices o si quieren usar un nombre como

"Superman", no me opongo. Pero si al número de vértices le llaman nlados, nl o ncolores entonces tendrán un gran descuento de puntos. Idem con los nombres de las funciones. (pej, una función que se llame OrdenarVertices pero en realidad ordene lados)

- 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).
- Respecto de la RAM, no voy a poner este año un limite definitivo, pero si lo que hacen es una desmesura carente de sentido común se les devolverá el proyecto para que lo corrijan.
- BE CAREFUL de no producir un stack overflow. En general los estudiantes provocan stack overflows haciendo una recursion demasiado profunda, o bien declarando un array demasiado grande. 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 (ver el siguiente punto) mientras que algo que dependa del número de colores es posible que si puede ir al stack, pues no habrá grafos que usen mas de a lo sumo una decena de miles de colores. Sin embargo, testeen su uso, y recuerden que el tamaño del stack no es igual en Windows que en Linux y creo que en Mac tambien es distinto a los dos.
- El grafo de entrada puede tener miles, cientos de miles de vertices e incluso millones de vertices y lados. Testeen para casos grandes. En mi página hay algunos grafos de ejemplos y varios en otras paginas, y deberían hacer sus propios ejemplos.
- En general, usar linked lists es un error en este proyecto, pues suele producir demoras significativas para grafos grandes. Especialmente, grupos que usan una linked list para la lista de vecinos suelen encontrarse con problemas, dado que Greedy se usará muchas veces y para cada vértice debe recorrer al menos parcialmente la lista de vecinos. Si bien no esta prohibido usar linked lists, si las usan y ven que tienen problemas de tiempo, esto puede ser una causa.
- Recuerden KISS: Keep It Simple, Stupid. Intentar estructuras estravagantes lo mas probable es que les haga perder tiempo de programación sin beneficio claro. Por ejemplo, un par de grupos en años anteriores intentaron usar bitmaps. Para qué, no me quedó claro. Lo único que lograron fue complicar el código y aumentar la probabilidad de cometer errores, algo que efectivamente les pasó.
- Para hacer OrdenNatural, WelshPowell y los RMBC van a necesitar, obviamente, alguna función de ordenación. También pueden necesitar una función de ordenación en la construcción del grafo. (esto no siempre es necesario, depende de cómo implementen la lectura de los datos).

  Recuerden que n puede ser del orden de millones, asi que una función $O(n^2)$ como bubble sort es un grave error. Al parecer no todos saben que existe una función en C llamada qsort (que NO ES quick sort y al menos en gcc es suficientemente rápida: en realidad el estandard de C no demanda nada sobre la complejidad de qsort, asi que algún compilador podría hacer que qsort fuese $O(n^2)$, pero con gcc las cosas parecen andar bien). Quizás les puede ser útil en vez de programar ustedes su propia función de ordenación.

  WelshPowell,RMBCnormal y RMBCrevierte pueden ser implementados con un sort $O(n)$. Si usan qsort es mas lento pero no mucho mas lento y es aceptable. (si lo programan bien, obvio)

  RMBCchicogrande también puede ser implementado con un sort $O(n)$ si se ordenan adecuadamente los colores primero, y esto último puede ser hecho con qsort. Si se usa directamente qsort sobre los vértices en vez de los colores, es mas lento pero no mucho mas lento y es aceptable.

  OrdenNatural no puede ser implementado con un **sort** $O(n)$ pero puede ser implementado en $O(n)$ si la estructura del grafo es construida adecuadamente y se hace un ordenamiento durante la construcción del grafo. Pero si no se hace esto, usar qsort para OrdenNatural también funciona adecuadamente.
- Testeen. Luego testeen un poco mas. Finalmente, testeen.
- Para el punto anterior deberán hacer por su cuenta uno o mas .c que incluyan un main que les ayude a testear sus funciones. (no deben entregar estos archivos)
- 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. Asumir que serán usadas de una forma que no esté en las especificaciones es un error grave.

10.2 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. El programa no compila. En particular, presten atención a los nombres de las funciones. Usen copypaste desde este documento para asegurarse que esten bien. Even better, ustedes van a tener que hacer uno o varios archivos test.c (o algún otro nombre) con un main para testear su código, y otros grupos tambien lo harán. Intercambien sus archivos test.c con los de otros grupos y verifiquen que su proyecto con su Rii.h compila con el test.c (o como se llame) de otro grupo.

2. No es capaz de leer archivos en el formato establecido. En particular, debe poder leer si se carga un grafo a mano por stdin pero tambien usando el operador de redirección "<". Por ejemplo, si al compilar el archivo ejecutable es "ejec"entonces algo como ./ejec <KC debe funcionar si KC es un archivo. Si ejec tiene opciones de entrada, como por ejemplo el número de veces que se realiza Greedy, o una semilla de aleatoriedad, tambien debe andar, pej, algo como ./ejec 1000 154367 <KC debe funcionar.

3. Greedy da coloreos no propios. Por ejemplo, en años anteriores hubo grupos que coloreaban K_n con menos de n colores.

4. Greedy da colores propios, pero al testear si el coloreo es propio o no la respuesta es que no porque tienen mal implementadas las funciones ColorDelVertice y ColorJotaesimoVecino. Tengan en cuenta que mis archivos de testeo no pueden acceder a su estructura interna, asi que la única forma que tengo para testear si un coloreo es propio o no es usar esas funciones. Si la función de testeo de si un coloreo es propio me dice que no es propio, no me voy a poner a ver si el problema está en Greedy o en las funciones ColorDelVertice y ColorJotaesimoVecino. (en otros años si lo hacía, diferenciaba entre los dos casos, y le avisaba al grupo, pero es su responsabilidad testear bien asi que este año no diferenciaré entre esos casos).

5. Bipartito da un coloreo no propio, diciendo que un grafo es bipartito cuando no lo es.

6. Greedy da siempre la misma cantidad de colores para un grafo dado, independientemente del orden de los vertices. (hay grafos para los cuales greedy siempre colorea con la misma cantidad de colores, por ejemplo, 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).

7. Durante cualquiera de los RMBCs o SwitchColores seguido de algún RMBC, la cantidad de colores 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.

8. No es capaz de leer un grafo y hacer 1000 reordenamientos y Greedys en un tiempo razonable (razonable es alrededor de 15 minutos en una máquina como las de Famaf, mas o menos, para los grafos mas grandes de la página. (pueden hacerse 3000 ordenamientos y Greedys en 5 minutos, asi que pedir 15 minutos para 1000 me parece razonable). 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. En este caso estan sumamente desaprobados).

9. Bipartito dice que alguno de los grafos bipartitos de mi página no es bipartito. Idem si no funciona bien con un grafo que tenga 4 vertices y dos lados alimentado directamente por stdin.

10. Buffer overflows, comportamiento indefinido o variables shadows.

11. Un memory leak. Puede no provocar desaprobación automática si no es muy grave pero si es suficientemente grande provocará desaprobación automática. Por ejemplo, hubo un año que un grupo entregó un proyecto que para grafos grandes agregaba algo asi como 10MB de RAM por cada corrida de Greedy y no la liberaba nunca.

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

10.3 Cosas que pueden o no causar desaprobación del proyecto

1. Hay alguna interacción defectuosa entre las funciones. Por ejemplo, un grupo entregó un Bipartito que daba resultados incorrectos si se lo llamaba despues de OrdenWelshPowell, pero daba resultados correctos si se lo llamaba antes. El problema era que ellos tenian una estructura interna compleja y poco transparente y el que programó WelshPowell hizo que tocara una parte de esa estructura que el que programó Bipartito creia que no se iba a tocar despues del inicio. Este tipo de errores será evaluado caso por caso.

10.4 Cosas que no causan desaprobación automática pero si devolución del proyecto

1. Bipartito funciona bien con los grafos de mi página pero tiene algún error que hace que para algunos grafos bipartitos diga que no lo son.
2. Los RMBCs ordenan por bloque de colores, pero no en el orden pedido. Por ejemplo, RMBCchicogrande ordena de grande a chico en vez de chico a grande o RMBCrevierte hace lo que debería hacer RMBCnormal y viceversa.
3. OrdenNatural no ordena correctamente.
4. OrdenWelshPowell no ordena correctamente.
5. No procesa bien algunos grafos debido a un error en el procesamiento de las lineas de comentarios del grafo, pero anda bien si se eliminan los comentarios.
6. 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, ver el último item de la enumeración de cosas que desaprueban.
7. Bipartito dice bien si un grafo es o no bipartito, pero retorna mal el número de componentes conexas.

10.5 No causa desaprobación ni devolución del proyecto pero si algún descuento en la nota

1. El programa anda bien con archivos que cumplan las especificaciones, pero falla en devolver `NULL` si el formato de entrada no es el correcto. Especificamente testearemos con archivos que declaren un m y luego no tengan al menos m lineas, o declaren un n y el número de vertices no sea n. Tambien con algunos en donde la primera linea que no sea comentario no empieza con “p edge”y cuando alguna de las lineas de lados no empieza con “e”.