Proyecto de Matemática Discreta II-2016 (primera parte)

Contents

1 Introducción

Deberán implementar lo que se detalla a continuación en C (de hecho, C99, i.e., pueden usar comentarios // u otras cosas de C99).

Este año el proyecto se divide en dos partes: la primera parte (y principal) se detalla en este documento. La segunda se les dirá mas adelante, pero será considerablemente mas fácil que la primera. (y usará lo hecho en esta primera parte).

Este año el proyecto iba a ser individual pero dada la cantidad de alumnos, se permiten grupos de 2 personas.

El objetivo es cargar un grafo y dar un coloreo propio de sus vertices, usando repetidamente Greedy. El proyecto llevará una nota entre 0 y 10, la cual será usada con las notas del práctico y del teórico del final para obtener la nota final de la materia. No pueden rendir el final mientras el proyecto no este aprobado por la cátedra, o si la nota del proyecto es menor a 4.

La aprobación o no del proyecto **sólo depende de esta primera parte**. La segunda parte es sólo para incrementar o bajar la nota. (pero nunca por debajo de cuatro si aprobaron esta primera parte). El proyecto será revisado por la cátedra, y si no se lo considera aprobado será devuelto para ser corregido. Cada devolución implica un descuento de puntos de la nota final máxima que puede tener el proyecto. Cuando el total de puntos descontado sume mas de 6 (lo cual puede suceder ya en la primera entrega, si los errores son muy graves), no pueden volver a entregar el proyecto y no pueden rendir, debiendo recursar la materia.

2 Entrega

Deben entregar vía e-mail los archivos que implementan el proyecto y un informe. Al final de este documento se detalla lo que debe ir en el informe. Los archivos del programa deben ser todos archivos .c o .h No debe haber ningun ejecutable. Entrega de un ejecutable implica desaprobación instantánea.

2.1 Archivos del programa

Deben entregar un archivo Cthulhu.h en el cual se hagan las llamadas que uds. consideren convenientes a librerias generales, ademas de las especificaciones de los tipos de datos y funciones descriptas abajo.

El programa que la cátedra usará para testear su programa solo tendrá includes de stdio.h, stdlib.h y Cthulhu.h por lo cual, por ejemplo, si van a usar string.h o stdint.h el include lo deben escribir uds. en Cthulhu.h. No es necesario (ni conveniente) que todo este definido en Cthulhu.h, pero si definen algo en otros archivos auxiliares, ademas de (obviamente) entregarlos, la llamada a esos archivos auxiliares debe estar dentro de Cthulhu.h. En Cthulhu.h debe figurar (comentados, obvio) los nombres y los mails de contacto de todos los integrantes del grupo.

Junto con Cthulhu.h obviamente deben entregar algunos archivos .c que implementen las especificaciones indicadas abajo, pero pueden nombrarlos como quieran.

Deben empaquetar todo de la siguiente forma: Debe haber un directorio de primer nivel cuyo nombre consiste en su nombre y apellido (si es un solo integrante) o sus dos apellidos. (si son dos integrantes)

En ese directorio deben poner el informe, ya sea en README o en .pdf (o ambos, si hacen los dos).

Ademas, en ese directorio habrá dos directorios de segundo nivel, uno llamado apifiles y otro llamado dirmain.

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

En dirmain debe haber un solo archivo, que se llamará mainPParker.c si su nombre es Peter Parker. (reemplazar PParker por la inicial de su nombre y su apellido si es un solo integrante, o por los dos apellidos si son dos integrantes). Nosotros luego agregaremos otros .c con distintos mains, que usen sus archivos de apifiles en ese directorio. Mas abajo especificamos que debe hacer mainPParker.c.

Para la **segunda parte** del proyecto, se les pedirá un nuevo main, con otras especificaciones, que tambien incluiremos en este directorio. Ese otro main usará las cosas especificadas en Cthulhu.h y sus otros archivos auxiliares, pero ustedes no sabrán lo que se requiere hasta mas tarde.

Empaqueten todo en formato .tgz.

No hace falta un make. Compilaremos desde el directorio de primer nivel con gcc, -Wall, -Wextra, -O3, -std=c99 y -Iapifiles. Es decir, algo como:

```
gcc -Wall -Wextra -O3 -std=c99 -Iapifiles dirmain/mainPParker.c apifiles/*.c -o PP
```

Sin embargo, nuestro compilador de gcc puede no ser el mismo que el suyo, traten de hacer las cosas lo mas portable posible.

2.2 Protocolo

Luego de enviado, se les responderá con un "recibido". Si no reciben confirmación dentro de las 24hs pregunten si lo recibí.Excepción: para los fines de semana pueden tener que esperar 48hs.

Se les puede requerir que corran su programa en su máquina para algunos grafos que les mandaremos. Algunas razones por las cuales pediremos esto es que su algoritmo corra demasiado lento con grafos grandes, o que obtenga resultados distintos dependiendo del compilador que se use o en la maquina que se use. Traten de hacer el código lo mas portable posible.

Una vez entregado el proyecto, lo podemos aprobar, ya sea con nota 10 o con alguna nota menor con descuento de puntos por errores diversos. Dependiendo de los errores podemos considerar devolver el programa para que lo corrijan. Esto implica un descuento de al menos 2 puntos, pero pueden ser mas. Excepción al descuento automatico de 2 puntos es si los errores son muy leves y fácilmente corregibles,en cuyo caso el descuento puede ser incluso menor a 1 punto.

Pueden entregarlo tantas veces como quieran, pero cada vez suma puntos de descuento.

Una vez que se hayan descontado 6 puntos, no pueden volver a entregar, el proyecto queda rechazado definitivamente y deben recursar la materia

2.3 Fecha de Entrega

El 5 de Mayo a las 10:00AM o antes deben entregar el proyecto. Un par de dias despues se entregarán las especificaciones para la segunda parte, que deberá ser entregada a fecha a confirmar, pero alrededor de fines de Mayo/principios de Junio.

Todos los que entreguen en esa fecha a esa hora o antes, salvo que el proyecto esté mal y sea devuelto, tienen asegurado que los proyectos completos serán corregidos a tiempo para que puedan rendir en la primera fecha, si asi lo desean, y que si quedan regulares en la materia la regularidad será inscripta en la planilla de regulares.

2.3.1 Entrega Alternativa

Si no lo entregan en esa fecha o lo entregan y se los devolvemos, quedan LIBRES (pero ver abajo*) y no tienen asegurada la rendida en la primera fecha de exámenes. Se corregirá primero a todos los que entreguen en esa fecha. Sólo se empezará a corregir a los otros luego de haber terminado de corregir a todos los que entregaron en esa fecha. Los que entreguen luego de esa fecha serán puestos en una cola y los que entreguen primero tendrán mas probabilidades de ser corregidos a tiempo para la primera fecha, pero dependerá del numero de proyectos entregados, y, otra vez, no tienen ninguna garantía.

Ademas, todos los que no entreguen en esa fecha tendran una penalización (chica, a determinar) en la nota del proyecto.

(*) Quedan libres en el sentido que cuando entregue la planilla de regularidad, figurarán como "libres", pero si aprobaron los parciales rinden como regulares. Algunos afortundados que entreguen

luego de esa fecha pueden tener suerte que los corrijamos a tiempo para la planilla de regularidad y en ese caso quedarían regulares, pero el default es que si no entregan a tiempo, quedan libres.

Para rendir en la segunda fecha de examen deben entregar (ahi si ya todo el proyecto completo, primera y segunda parte) para el 21 de Junio a las 1000AM. Para rendir en la tercera fecha de examen deben entregar el primer lunes despues de las vacaciones de julio a las 1000 AM. (en este momento no se exactamente cuando serán las vacaciones de Julio).

Para rendir en Noviembre-Diciembre-Febrero-Marzo tienen que entregarlo como máximo el 30 de Septiembre. Luego de esa fecha no se aceptan más proyectos.

A continuación, las especificaciones

3 Formato de Entrada

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

3.1 Basics

La descripción oficial de DIMACS es asi:

1) Ninguna linea tiene mas de 80 caracteres.

2) Al principio habrá cero o mas lineas que empiezan con c las cuales son lineas de comentario y deben ignorarse.

3)Luego hay una linea de la forma:

p edge n m

donde n y m son dos enteros.

n representa el número de vértices y m el número de lados.

4) Luego siguen m lineas todas comenzando con e y dos enteros, representando un lado. Luego de esas m lineas deben detener la carga.

3.2 Ejemplo

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
```

3.3 Further Especificaciones, Observaciones y Refinamientos

5) Cada lado vw aparecerá exactamente una vez en alguna linea de la forma e v w, o bien en una linea e w v. Si aparece una no aparecerá la otra. En muchos ejemplos (como el anterior) se usa e v w con "v" menor que "w", pero eso no esta especificado y no pueden asumir que la entrada será asi.

6) 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.

7) Observar que en el ejemplo y en muchos otros 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. (reservar espacio para 2^{32} posibles entradas no es una buena idea)

8) 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

9) Si se recorren todos los lados, leyendo los vertices, habrá un número n^* de vértices entre ellos. Si el formato de entrada esta bien hecho, n^* debe ser menor o igual que n, y los vértices que no aparecen en los lados serían vértices aislados, y se debería especificar de alguna forma cuales son.

Para este proyecto, todos los archivos de entrada válidos serán sin vértices aislados, es decir, n^* será siempre igual a n. De todos modos, uds. deberán verificar que esa condición se cumple y si no detener el programa.

A continuación se detalla la API, y la documentación que se debe entregar junto al código. Su programa DEBE RESPONDER a estas especificaciones.

4 Tipos de datos

4.1 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.

4.2 VerticeSt

Será una estructura definida por ustedes que representará un vértice.

La información mínima que esta estructura debe guardar es al menos:

1. El "nombre" real del vértice
2. El grado del vértice
3. El color del vértice

4.3 NimheSt:

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. 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. (dependiendo la estructura que usen, puede ser conveniente tener DOS ordenes guardados: uno fijo de referencia, y otro mutable).

Basicamente, debe tener lo que uds. consideren necesario para poder implementar las funciones descriptas abajo, de la forma que a uds. les parezca mas conveniente.

Una forma posible de definir la estructura es por ejemplo:

```
typedef struct Grafoplus{
    //aquí van los parámetros específicos de cada equipo
} NimheSt;
```

El coloreo siempre debe cumplir que si es un coloreo con j colores entonces los colores son $1, 2, ..., j$. A los vértices no coloreados se le asigna el color "0" para indicar esto.

4.4 NimheP:

es un puntero a una estructura de datos *NimheSt*. Esto se define de la siguiente forma:

```
typedef NimheSt *NimheP;
```

Las funciones que deberan implementar son en general de la forma Funcion(NimheP G), para poder cambiar la estructura interna del grafo.

5 Funciones

Se deben implementar las siguientes funciones.

5.1 Funciones De Construcción/Destrucción del grafo

5.1.1 NuevoNimhe()

Prototipo de función:

```
NimheP NuevoNimhe();
```

La función aloca memoria, inicializa lo que haya que inicializar de una estructura NimheSt y devuelve un puntero a ésta. Ademas lee un grafo **desde standard input** en el formato especificado al principio de este documento y llena esa estructura NimheSt con esos datos. Observación: como estan haciendo estas cosas al mismo tiempo, al leer la entrada ya sabrán cuantos vértices y lados tiene el grafo antes de tener que leer todos los lados, por lo que pueden eficientar la inicialización de la estructura.

Ademas de cargar el grafo, asigna el "color" 0 a cada vertice para indicar que están todos no coloreados.

Como dijimos, la función debe leer desde standard input. En general le daremos de comer una serie de lineas en ese formato, a mano, o bien usaremos en standard input el redireccionador "<" para redireccionar a un archivo que sea de esa forma.

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, que la primera linea sin comentarios no empieze con p o que $n^* \neq n$ (ver en el item de formato de entrada que significa esto)

5.1.2 DestruirNimhe()

Prototipo de función:

```
int DestruirNimhe(NimheP G);
```

Destruye G y libera la memoria alocada. Retorna 1 si todo anduvo bien y 0 si no.

5.2 Funciones de los vertices

5.2.1 ColorDelVertice()

Prototipo de Función:

```
u32 ColorDelVertice(VerticeSt x);
```

Devuelve el color con el que está coloreado el vértice x.

Si el vértice no esta coloreado, devuelve 0.

5.2.2 GradoDelVertice()

Prototipo de Función:
`u32 GradoDelVertice(VerticeSt x);`
Devuelve el grado del vértice x.

5.2.3 NombreDelVertice()

Prototipo de Función:
`u32 NombreDelVertice(VerticeSt x);`
Devuelve el nombre real del vértice x.

5.2.4 ImprimirVecinosDelVertice()

Prototipo de Función:
`void ImprimirVecinosDelVertice(VerticeSt x,NimheP G);`
Imprime en standard output una lista de los vecinos de x. La lista debe estar formada por los nombres reales de los vertices, separados por comas, con un punto al final de la lista.
Ejemplo:
VecinosDelVertice(x,G);
debe producir:
1,7,10,333,819092.
(asumiendo que esos son los vecinos de x). No es necesario que la lista esté ordenada en ningún orden particular.
Nota: observar que las tres primeras funciones sólo dependen de x mientras que la ultima requiere x y G. Esto es porque estoy asumiendo que las tres primeras son tan locales a x que no requieren ningun conocimiento de la estructura de G. Esta última función en cambio puede (o no) necesitar acceso a la estructura general de G para ejecutarse, dependiendo de como implementen la estructura, y no quiero restringirlos de ninguna manera. Si ustedes consideran que tienen una estructura tal que las implementaciónes de las tres primeras funciones también funcionarían mejor si fuesen de la forma (x,G) en vez de (x), haganmelo saber, pero tienen que tener buenas razones.

5.3 Funciones para extraer información de datos del grafo

5.3.1 NumeroDeVertices()

Prototipo de función:
`u32 NumeroDeVertices(NimheP G);` Devuelve el número de vértices del grafo G.

5.3.2 NumeroDeLados()

Prototipo de función:
`u32 NumeroDeLados(NimheP G);` Devuelve el número de lados del grafo G.

5.3.3 NumeroVerticesDeColor()

Prototipo de función:
`u32 NumeroVerticesDeColor(NimheP G,u32 i);`
Retorna el número de vértices de color i. (si $i = 0$ devuelve el número de vertices no coloreados).

5.3.4 ImprimirVerticesDeColor()

Prototipo de función:
`u32 ImprimirVerticesDeColor(NimheP G,u32 i);`
Imprime una linea que dice "Vertices de Color i:" y a continuacion una lista de los vertices que tienen el color i, separados por comas y con un punto luego del último color. (si $i = 0$ esta lista será la lista de vertices no coloreados) Por ejemplo:
Vertices de Color 3: 15,17,1.

Los vertices no tienen porque estar ordenados.
Si no hay vertices de color i debe imprimir "No hay vertices de color i"
Retorna el número de vertices de color i.

5.3.5 CantidadDeColores()

Prototipo de función:
`u32 CantidadDeColores(NimheP G);`
Devuelve la cantidad de colores usados en el coloreo de G.

5.3.6 IesimoVerticeEnElOrden()

Prototipo de función:
`VerticeSt IesimoVerticeEnElOrden(NimheP G,u32 i);`
Devuelve el vértice numero i en el orden guardado en ese momento en G. (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

5.3.7 IesimoVecino()

Prototipo de función:
`VerticeSt IesimoVecino(NimheP G,VerticeSt x,u32 i);`
Devuelve el vécino numero i de x en el orden en que lo tengan guardado uds. en G. (el orden es irrelevante, lo importante es que de esta forma podemos pedir externamente la lista de vecinos) (el índice 0 indica el primer vértice, el índice 1 el segundo, etc)

Nota acerca de la eficiencia: Obviamente si deciden guardar la estructura del orden de G o la de los vécinos en un array, estas últimas dos funciones serán mas eficientes que si deciden usar por ejemplo pointers al "siguiente" vécino o vértice en el orden, o alguna otra estructura para la cual para determinar el $i$ésimo elemento haya que recorrer toda la estructura desde el principio, pero para estas dos funciones NO se tienen que preocupar por la eficiencia, pues estan sólo para testear.

Es decir, usen la estructura interna que mas les parezca conveniente para correr Greedy, ordenar repetidamente, etc, y no se preocupen si estas dos últimas funciones demoran mas o menos, porque sólo la usaremos en mains para testear que su estructura interna este guardando bien los vecinos y el orden, usando grafos relativamente chicos.

Nota II medio irrelevante: esta función la podría haber puesto en la subsección anterior, pero la puse aqui porque es la función que me permite extraer los lados de G, junto con la función anterior a esta. Como no les estoy pidiendo una estructura "Lados" ni que guarden los lados, no les puedo pedir que me den el $i$ésimo lado (aunque algunos proyectos podrían hacerlo pues quizas usen una lista de lados) pero hagan lo que hagan, deben tener una forma de buscar los vecinos de un vértice o no podrían correr Greedy, por eso pido esta función.

5.4 Funciones de coloreo

5.4.1 Chidos()

Prototipo de función:
`int Chidos(NimheP G);`
Devuelve 1 si $\chi(G) = 2$, 0 si no.

5.4.2 Greedy()

Prototipo de función:
`u32 Greedy(NimheP 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.

5.5 Funciones de ordenación

5.5.1 OrdenNatural()

Prototipo de función:

`void OrdenNatural(NimheP G);`

Ordena los vertices en orden creciente de sus "nombres" reales. (recordar que el nombre real de un vértice es un u32)

5.5.2 OrdenWelshPowell()

Prototipo de función:

`void OrdenWelshPowell(NimheP G);`

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

5.5.3 ReordenAleatorioRestringido()

Prototipo de función:

`void ReordenAleatorioRestringido(NimheP G);`

Si G esta coloreado con r colores y W_1 son los vertices coloreados con 1, W_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de W_{j_1} (en algún orden) luego los de W_{j_2} (en algún orden), etc, donde $j_1, j_2, ...$ son aleatorios (pero distintos).

5.5.4 GrandeChico()

Prototipo de función:

`void GrandeChico(NimheP G);`

Si G esta coloreado con r colores y W_1 son los vertices coloreados con 1, W_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de W_{j_1} (en algún orden) luego los de W_{j_2} (en algún orden), etc, donde $j_1, j_2, ...$ son tales que $|W_{j_1}| \geq |W_{j_2}| \geq ... \geq |W_{j_r}|$.

5.5.5 ChicoGrande()

Prototipo de función:

`void ChicoGrande(NimheP G);`

Idem que el anterior excepto que ahora el orden es tal que $|W_{j_1}| \leq |W_{j_2}| \leq ... \leq |W_{j_r}|$.

5.5.6 Revierte()

Prototipo de función:

`void Revierte(NimheP G);`

Si G esta coloreado con r colores y W_1 son los vertices coloreados con 1, W_2 los coloreados con 2, etc, entonces esta función ordena los vertices poniendo primero los vertices de W_r (en algún orden) luego los de W_{r-1} (en algún orden), etc.

5.5.7 OrdenEspecifico()

Prototipo de función:

`void OrdenEspecifico(NimheP G,u32* x);`

La imagen de x deben ser todos los numeros entre 0 y $n-1$, sin repetir. La función primero chequea que x cumpla esa condición. Si no lo hace, la función deja el orden de G como estaba.

Si x cumple la condición, entonces la función ordena los vertices con OrdenNatural(), luego lee el orden dado en la string x y los ordena de acuerdo a lo que lee. Es decir, si luego de OrdenNatural los vertices quedaron ordenados como, por ejemplo, V[0],V[1],...,V[n-1], luego de OrdenEspecifico el orden debe ser V[x[0]],V[x[1]],...,V[x[n-1]].

Ejemplo:

Supongamos que los vertices son 1, 10, 22, 5, 64, 7, 1001. OrdenNatural() los ordena como 1,5,7,10,22,64,1001. (i.e. V[0]=1,V[1]=5, etc).

Supongamos que x=4,0,2,6,5,3,1. Entonces OrdenEspecifico(G,x) ordenará los vértices como V[4],V[0],V[2],etc, es decir, 22,1,7,1001,64,10,5.

6 Main

El archivo mainPParker.c (o su equivalente con su nombre) debe tener una función main, y llamadas solamente a stdio.h, stdlib.h y Cthulhu.h.

Puede tener funciones definidas dentro de mainPParker.c que no esten definidas en Cthulhu.c, pero lo que no puede hacer es usar la estructura INTERNA de su implementación. (esto es algo tan obvio que me da verguenza tener que aclararselo, pero todos los años parece que alguien no entiende estas cosas básicas de las APIs). **El mainPParker.c debe funcionar con el Cthulhu.h de Clark Kent.** Por ejemplo, si van a leer el color o el grado de un vértice, deben usar ColorDelVertice() o GradoDelVertice() en este archivo. (Sin embargo, en los archivos de apifiles esto NO ES NECESARIO NI CONVENIENTE. Es decir, si tienen por ejemplo, un campo en la estructura VerticeSt que sea "color", entonces para un vértice x, en los archivos de apifiles es probablemente mas eficiente usar x.color y no ColorDelVertice(x). Pero en el mainPParker.c no pueden usar x.color.

mainPParker.c debe usarse para leer un grafo desde standard input, donde la entrada será como es descripta arriba y que haga lo siguiente:

1. Carga el grafo. Si el formato de entrada esta mal imprime una linea que dice "Error en formato de entrada" y para. (por ejemplo, si hay una linea que empieza con "d", o si la primera linea que no empieza con "c" no empieza con "p", o si el número de vertices que se extrae de los lados no es n, etc).

2. Si $\chi(G) = 2$ debe imprimir en standard output "Grafo Bipartito" (sin las comillas) y parar.

3. Si $\chi(G) > 2$ debe imprimir en una linea "Grafo No Bipartito" y luego crear 10 ordenes aleatorios mas Welsh-Powell y correr Greedy con cada uno de esos ordenes y decir cuantos colores obtiene en cada caso, en 11 lineas separadas.

   En el caso de las corridas aleatorias cada linea será de la forma

   "coloreo aleatorio numero k: r colores" (sin las comillas)

   donde r será el número de colores que obtuvieron y k el numero de corrida aleatoria.

   En el caso del orden WelshPowell la linea será

   " coloreo con Greedy en WelshPowell:r colores" (sin las comillas)

   (observar el espacio en blanco al principio de la linea). Ademas, antes de esa linea habrá una linea en blanco que la separe de las lineas anteriores de las corridas aleatorias.

4. Si estan en este paso es porque el grafo no es bipartito, por lo tanto si alguno de esos coloreos es con tres colores, ya saben que $\chi(G) = 3$ asi que deben imprimir una linea que diga "X(G)=3" (sin las comillas) y parar.

5. Si no obtuvieron 3 colores con ninguno de los coloreos, deben imprimir una linea en blanco, luego una linea que diga:

   "====Comenzando Greedy Iterado 1001 veces====" (sin las comillas)

   y luego otra linea en blanco.

   Luego deben tomar el mejor coloreo de los primeros 11 que hicieron y correr Greedy 1000 veces cambiando el orden de los colores, siguiendo el siguiente patrón:

   - 50% ChicoGrande
   - 12,5% GrandeChico
   - 31,25% Revierte
   - 6,25% ReordenAleatorioRestringido

   (nota: esos porcentajes son 8/16, 2/16, 5/16 y 1/16 respectivamente). Luego de estas 1000 iteraciones, deben hacer una iteración final con Revierte, e imprimir :

   Mejor coloreo con Greedy iterado 1001 veces: k colores

   (a CG,b GC, c R, d RAR)

donde k es el menor número de colores que hayan obtenido en las 1001 iteraciones, y a,b,c,d son la cantidad de veces que corrieron Chico Grande,GrandeChico, Revierte y ReordenAleatorioRestringido, respectivamente.

Estas condiciones no significan que exactamente un 12,5% van a usar GrandeChico, exactamente un 50% van a usar ChicoGrande, etc, sino que en cada iteración, cuando el algoritmo va a decidir cual orden usará, elegirá GrandeChico con una probabilidad del 12,5%, ChicoGrande con una probabilidad del 50%, etc.

(*) Nota: deben empezar Greedy iterado a partir del mejor coloreo que obtuvieron, asi que salvo que tengan suerte con WP (lo cual en realidad sucede en muchos casos, por eso lo puse al último) deberán reconstruir el coloreo que tenian en esa corrida. Esto lo pueden hacer de diversas maneras: una es guardando en un array el coloreo, otra es guardando el orden de los vertices que dio origen a ese coloreo, cargarlo y correr Greedy otra vez, otra es guardando la semilla que dio origen al orden (si es que esto es facil de hacer en su algoritmo de aleatoriedad), etc. Lo que vamos a considerar mal es si comienzan Greedy iterado desde el lugar equivocado, lo demas es su elección.

7 Informe

Deberán entregar un informe sobre el proyecto. El informe puede ser un .pdf o un README que sea leible con cualquier lector de textos ASCII (asi que NO usen Word o FreeOffice o cualquier editor no ASCII). Pueden hacer tanto un .pdf como un README, por ejemplo, en el .pdf poner un informe detallado, y en el README algo mas corto.

El informe debe ser claro, y debe mostrar que ustedes entienden su propio proyecto y no lo copiaron de algun lado. (esta permitido tomar ideas de otros lados, e incluso fragmentos de código si lo desean, pero debe quedar claro que las/los entienden y no que solo hicieron cut and paste). Algunos detalles del código nos pueden hacer sospechar que lo copiaron, en ese caso podemos requerir una exposición oral intensiva y si no nos convencen el proyecto se devuelve con 5 puntos de descuento.

En el informe deben incluir los nombres y apellidos de los integrantes. Ademas, el informe debe proveer una especificación del proyecto, tal que alguien, leyendo el informe, pudiera hacer una implementación "black room" de su proyecto sin tener que leer el código. Es decir, deben explicar que hace su algoritmo en forma resumida. Copiar en bloque el código no es una especificación. No hace falta que expliquen que es Greedy pero si hace falta que expliquen lo mas claro posible como funciona su implementación y cómo resolvieron las dificultades. A continuación damos explicitamente una lista de preguntas que deben estar respondidas en el informe. (estas preguntas deben estar todas respondidas, lo cual no significa que el informe sea sólo respuestas a estas preguntas)

7.1 Preguntas que deben responder en el informe

1. ¿Cómo hicieron para resolver el problema de que los vertices pueden ser cualquier u32? Dependiendo de cómo resuelvan este problema, deben responder otras preguntas.
   (a) Si usan hash tables: ¿Que tipo de hash table usaron?
   (b) Si usan un array de tamaño n, llenandolo a medida que van leyendo los vértices: ¿Cómo solucionaron el problema de saber si un vértice que acaban de leer ya lo habian leido antes?
   (c) Si, para no tener que solucionar el problema del item anterior, usan un array de tamaño 2^{32} entonces la pregunta que deben responder es: ¿Porqué son tan idiotas y que hacen estudiando en esta facultad? (vale triple si usan una matriz de adyacencia $2^{32} \times 2^{32}$).
   (d) Si usan otra cosa, describanla bien y expliquen porqué prefirieron usar esa estructura.
2. ¿Cómo implementaron $\Gamma(x)$?(si es que lo hicieron, y si no, ¿cómo hicieron para buscar vécinos? Es decir, guardaron los vecinos en una lista u otra estructura que representa $\Gamma(x)$, o los buscan cada vez que los necesitan, o que?) Nota: usar una matriz de adyacencia $n \times n$ es posible para n chicos, pero con 100K vértices, bueno, hagan la cuenta. Una posibilidad, ya que saben n al comienzo, es tener DOS estructuras, una para usar cuando n es chico y otra cuando n es grande, pero creo que es una complicación innecesaria a la cual no le veo la utilidad en este

momento y peligrosa, pero si les gusta jugar con fuego empapados en petroleo al lado de un volcán en erupción, well, go ahead.

3. ¿Cómo implementaron el orden de los vértices? (puede ser una simple lista, o pueden ser que la estructura "VerticeSt" tenga entre sus campos un puntero al "siguiente" vécino, o alguna otra cosa. Expliquen bien que usaron)

4. ¿Cómo implementaron Greedy? ¿Qué estrategia usaron para lograr reducir la velocidad? (Greedy será probablemente el cuello de botella de su velocidad, dado que lo tendran que ejecutar 1000 veces, aunque los algoritmos de ordenación tambien pueden ser un cuello de botella si los hacen mal).

5. ¿Cómo implementaron Revierte en forma eficiente?

6. ¿Cómo implementaron GrandeChico y ChicoGrande en forma eficiente? (Deberian ser bastante similares entre los dos, basta que expliquen uno y luego las diferencias menores). Estos pueden darle muchos dolores de cabeza si no los implementan eficientemente.

8 Final Warnings

8.1 Cosas para tener en cuenta

- El uso de macros esta permitido pero como siempre, sean cuidadosos si los usan.
- Debe haber MUCHO COMENTARIO, las cosas deben entenderse. En particular, en cada funcion o macro que usen debe haber una buena explicación, entendible, de lo que hace. Ademas cada variable que se use debe tener un comentario al lado indicando su significado y para que se usará. (por ejemplo:

  ```
  int i,j; //indices
  int nv; //numero de vertices
  bool Stop; //una booleana que sera 0 si...  y 1 si...
  ```

  etc;

  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 quedan automáticamente desaprobados. (esto me ocurrió con un proyecto hace unos años. En aquel momento no lo desaprobé pero me pareció muy estúpido).
- El código debe estar bien modularizado.
- Presten atencion al formato especificado, no solo al de entrada sino al de salida: uds. deben asumir que otro grupo de personas usará la salida de su programa como input de algun programa propio, de ahi las especificaciones de salida.
- Como se dijo varias veces, la entrada y la salida son estandards. Imprimir a un archivo implica devolución automática del proyecto.
- No se pueden usar librerias externas que no sean las básicas de C. (eg, stdlib.h, stdio.h, strings.h, stdbool.h, assert.h etc, si, pero otras no. Especificamente, glibc NO). Pueden tomar ideas de ellas, pero su código debe ser lo mas portable posible, algunas librerias externas no siempre lo son, o son demasiado grandes. La experiencia de años anteriores indica que nos crea problemas.
- Su programa no debe usar nunca mas de 256 MB de memoria RAM.
- El grafo de entrada puede tener miles o incluso cientos de miles de vertices y lados. Testeen para casos grandes. (nunca va a haber millones de vertices pero puede haber millones de lados)

- Su programa debe ser razonablemente rápido. Algunos lo harán mas rápido que otros, pero no deberia demorar horas (y menos aun dias) en terminar. En particular, con todos los grafos de ejemplo que se me ocurran deben demorar menos de una hora.

8.2 ErroresFrecuentes

En general la experiencia demuestra que algunos alumnos son muy creativos a la hora de cometer errores, asi que es imposible para nosotros listar todos los errores posibles que puedan cometer.

La nota por default del proyecto es 10, es decir, no juzgaremos calidades distintas de proyectos. Sin embargo, a partir del 10, descontaremos puntos por fallas en el proyecto, en algunos casos devolviendo el proyecto.

Si juzgamos que los comentarios no son los suficientes o no lo suficientemente claros, les descontaremos entre 0,5 puntos y 1 punto, y podemos devolver el proyecto requiriendo que agreguen mas comentarios.

Idem si consideramos que el informe no es claro.

SI MANDAN EL INFORME EN UN FORMATO QUE NO ES EL QUE ESPECIFICAMOS LES DESCONTAREMOS 4 PUNTOS.

La siguiente es una lista de errores de programación que hemos visto en otros años. Obviamente, seguramente habrá errores que no hemos pensado.

- No compila. A veces no compila con sus archivos de apifiles y su main, otras veces no compila cuando lo intentamos con nuestro main y sus apifiles (o al reves). Esto ocurre si no respetan las especificaciones. Por ejemplo, no tipean bien las funciones de la API, que sus funciones usen mas argumentos, menos argumentos o argumentos de distinto tipo que los especificados o que retornen valores que no son los especificados.
- Hay memory leaks.
- Runtime error.
- Errores de impresión de la salida.
- No acepta inputs válidos.
- El algoritmo dice que un grafo es bipartito cuando no lo es o viceversa.
- Errores de carga en los datos: continua cargando mas alla de lo que deberia o carga mal algunos lados, o directamente no los carga, o inicializa mal algo.
- Colorean con un coloreo que no es propio.
- Uno o mas de las implementaciones no son realmente implementaciones de lo que se pide. Por ejemplo Greedy no es en realidad Greedy, o WelshPowell no es en realidad WelshPowell, etc.