Facultad de Matemática, Astronomía, Física y Computación

Alejandro Jose Nigrelli y Alejandro Ismael Silva

Informe Proyecto de Matématica Discreta II

Parte I

Profesor:
Daniel Penazzi

Contents

1 Creando a Cthulhu

En una tarde de marzo, en un fin de semana largo por semana santa, comenzamos el proyecto. En esos dias definimos la primera estructura del proyecto, los TADs, y codificamos hasta que el codigo compiló, aunque aún no andaba.

Luego de ese primer intento, pasamos por muchas ideas, modelos y estructuras hasta que llegamos a este modelo final.

En la primera estructura usamos un arreglo de punteros a los vertices para el orden, una tabla hash para la carga y listas para los vecinos. Decidimos quitar la tabla hash por que nos estaba trayendo muchos problemas y pusimos provisoriamente una busqueda lineal.

Cuando el proyecto empezo a funcionar, nos dimos cuenta que la carga era muy lenta con grafos grandes, entonces cambiamos la busqueda lineal por binaria, esto mejoro sólo un poco por lo que volvimos de nuevo a la tabla hash.

Para el reordenamiento implementamos unos algoritmos lineales que no terminaban de andar bien, al final los cambiamos por *qsort* de la libreria estandar de C.

Cambiamos las listas de vecinos por arreglos porque consumia mucha memoria. Aunque durante un tiempo las mantuvimos solo para la carga, pero con ciertos grafos la memoria no se reducia luego de crear el grafo a pesar de hacer *free* de todos los nodos, por lo que quitamos las listas y guardamos todos los lados en una matriz.

A mitad de la anterior refactorización, por culpa de esa memoria liberaba que no se veia reflejada en *htop*, pusimos todos los vertices en un arreglo y para el orden un arreglo de numeros, en un intento de reducir la memoria usada.

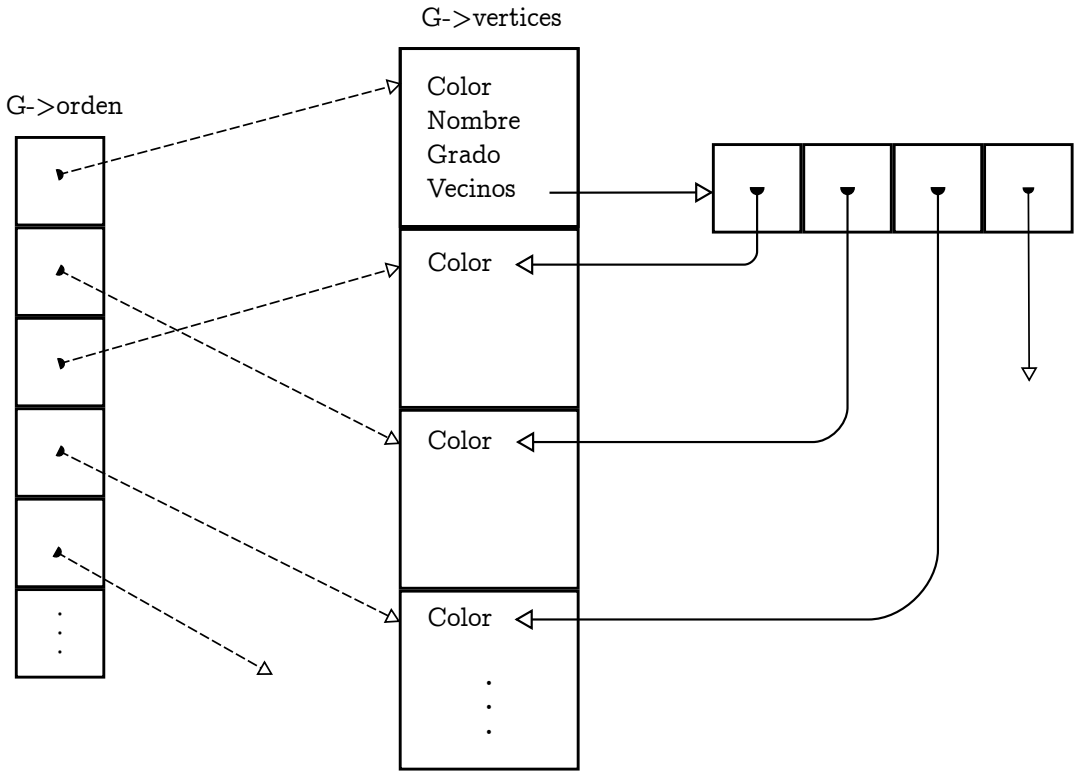
Desde el principio nos planteamos la idea de ordenar los vecinos de un vértice para no tener que revisarlos a todos cuando corra Greedy. Nos llevó unos meses convencernos de que el costo de ordenarlos era mayor con la estructura que teniamos.

Al final, seguimos el estilo de nomenclatura *(*St, *P)* para todas las estructuras internas para mantener el mismo estilo de codigo.

Pasamos mucho tiempo leyendo como escribir codigo eficiente en C, y fue una de las cosas que mas aprendimos con este proyecto, desde usar una mascara para hacer modulo con una potencia de dos, principios de la caché, hasta medir el *User time* y usar *Perf*.

2 Mirada a la telaraña

La estructura general del proyecto tiene la siguiente forma:



Implementamos el orden con un arreglo de *u32* (*G->orden*), que va a indexar a un arreglo del mismo tamaño de estructuras *VerticeSt* (*G->vertices*). Implementamos $\Gamma(x)$ con un arreglo de punteros. Notamos una mejora al hacer que en este arreglo se apunte sólo al color del vértice vecino y no a toda la estructura. Para poder acceder a toda la estructura del vecino, el campo color es el primero, así podemos castear el puntero y acceder a la estructura.

Ademas, con *G->vertices* tenemos menos llamadas al sistema durante la carga para alocar memoria y nos ahorramos memoria en maquinas de 64 bit, ya que un puntero ocupan el doble de espacio que un *u32*.

3 Las patas de la araña

3.1 Estructura VerticeSt

- **Color:** Es el color del vértice, 0 si no está coloreado.
- **Nombre:** Nombre unico del vértice.
- **Grado:** *d(x)*. Es la cantidad de vertices conecados por lados.
- **Vecinos:** Γ*(x)*. Es un arreglo de punteros que van a apuntar al color de los vecinos del vértice.

3.2 Funciones implementadas

- *crearVertice(u32 nombre, VerticeP v):* Llena a *v* con el nombre e inicializa el grado en 1 ya que estaba en la lista de lados, y por lo tanto, ya está conectado a un vertices.
- *agregarVecinos(VerticeP v, VerticeP w):* Pone en el arreglo de vecinos de v, un puntero al color de w. Y pone un puntero al color de v en el arreglo de vecinos de w. Usamos el campo *color* como variable temporal para ir llenando el arreglo de vecinos.
- *destruirVertice(VerticeP v):* Solo destruye el arreglo de vecinos, ya que la estructura del vértice será liberada cuando se destruya *G->vertices*.

En la función *ImprimirVecinosDelVertice(VerticeP x)* nos aprovechamos de que podemos castear el puntero al color del vecino, acceder a su nombre e imprimirlo.

4 Las entrañas de la araña

4.1 Estructura NimheSt

- **vertices:** Es el arreglo que contiene todos los vertices del grafo.
- **orden[i]:** Guarda el i-esimo vértice según el orden.
- **colorV[i]:** Guarda la cantidad de vertices coloreados con el color i. Se llena mientras se colorea.
- **colorN:** Guarda el mejor coloreo hasta el momento. Se actualiza al finalizar greedy.
- **totalV:** Guarda la cantidad total de vertices.
- **totalL:** Guarda la cantidad total de lados.
- **deltaM:** Guarda el valor Δ

4.2 Funciones implementadas

- **Nueva Araña:** Decidimos dividir la carga en 3 funciones porque era muy largo y habia muchas variables que se dejaban de usar.
  - *crearGrafo():* Alocamos la estructura del grafo, parseamos el principio del input buscando la linea edge y creamos el arreglo para los vertices y el orden.
  - *cargarLados(NimheP G):* Para cargar los lados del grafo nos ayudamos con una tabla hash para saber que vertices ya fueron creados y cuales no. Guardamos los lados en un arreglo temporal para primero calcular el grado de cada vértice, luego alocar el arreglo de vecinos y recien despues llenarlos.

– *NuevoNimhe():* Despues de llamar a *crearGrafo()* y *cargarLados()*, calculamos el delta mayus, llenamos el arreglo de orden, y alocamos espacio para el coloreo.

Mas adelante contaremos porque necesitamos 2 variables globales que las seteamos en la carga.

- **Colores de las patas:** En *ImprimirVerticesDeColor(NimheP G, u32 i)* lo unico destacable es la guarda del *for*. Como *G->colorV* se actualiza a medida que coloreamos, sabemos que vamos a encontrar el *total* de vertices de color *i* sin pasarnos del tamaño del arreglo *orden*.

- **i-esima pata:** Para indexar un vértice en orden se debe indexar en el arreglo *G->vertices* la posición del numero que se encuentra en la posición *i* del arreglo *G->orden*.

- **Destruir Araña:** Como habiamos dicho, *destruirVertice(VerticeP v)* solo libera el arreglo de vecinos, por lo que luego liberamos *G->vertices* que contiene todas las estructuras.

  Despues liberamos el resto de arreglos y siempre retornamos 1 ya que *free* no falla, y si fallaría, el programa terminaría por *segfault*.

5 Ordenando las patas

En todas las funciones, menos OrdenEspecifico, usamos *qsort* que trae la libreria standard de C y para usarla hay que implementar una función auxiliar, la cual tiene el prototipo *int comparar(void *a, void *b)*, usaremos la resta entre dos campos (que dependen del orden que queremos) de los vertices para devolver el resultado de la comparación. Como vamos a llamar a *qsort* con *G->orden*, las funciones auxiliares no tienen acceso a la estructura de los vertices, por lo tanto, definimos dos variables globales: Una para las estructura de los vertices y otra para *G->colorV*.

5.1 OrdenNatural

Llamamos qsort con la función *compararVertices()* que usa el campos nombre de la estructura VerticeSt.

5.2 OrdenWelshPowell

Usamos qsort con la función *compararGrados()* que usa el campo grado de la estructura VerticeSt.

5.3 GrandeChico

Qsort usa la función *compararColoresGra* que utiliza el campo color de la estructura de VerticeSt y luego utiliza *G->colorV* para saber cuantos vertices estan coloreados con ese color.

5.4 ChicoGrande

Es lo mismo que *GrandeChico()* solo que hacemos la resta al revés.

5.5 Revierte

Qsort utiliza la función *compararColoresRev* que usa el campo color y hace la resta al revés para que sea decreciente.

5.6 ReordenAleatorioRestringido

Generamos un orden aleatorio sobre el arrelgo *colorV* y luego llamamos *qsort* con la función *compararColoresChi()* porque esta función depende de *colorV* y agrupa todos los vertices de un color.

5.7 OrdenEspecifico

Primero chequeamos que el orden solicitado sea correcto, es decir, lo recorremos chequeando que todos los numeros esten entre 0 y *totalV*, y los vamos marcando en un arreglo de booleanos para buscar repeticiones. Ordenamos con *ordenNatural()* y luego usamos un nuevo arreglo para definir las posiciones del nuevo orden.

6 Otras cosas en la telaraña

6.1 Hash

La tabla hash sólo la usamos durante la carga del grafo, la que utilizamos se llama **Hash lineal**.

Decidimos que el tamaño del arreglo de la tabla sea el doble de la cantidad de vertices para evitar colisiones sin sacrificar mucho espacio.

El hash elegido fue multiplicar por un numero primo y tomar resto de la división por el tamaño del arreglo.

Para tratar las colisiones, este tipo de tabla hash trabaja de la siguiente manera:

- **Guardar pata:** Calculamos la posición en la que debería ir ubicado aplicandole la hash al nombre del vértice. Si no está ocupado se guarda en ese lugar, si lo está entonces recorremos linealmente el arreglo hacia delante hasta encontrar un espacio libre y lo guardamos ahí.
- **Buscar pata:** Calculamos la posición donde debería estar. Como los nombres de los vertices son unicos, recorremos linealmente el arreglo hasta encontrar un vértice con el mismo nombre o un espacio vacio.

6.2 Cola

Utilizamos dos estructuras la estructura *Nodo* y la estructura *Cola*.

En la estructura *Cola* pusimos los siguientes campos:

- **primero:** Este campo se utiliza para el principio de la cola y sirve para decolar.
- **ultimo:** Este campo apunta al ultimo nodo y es por una cuestion de eficiencia ya que teniendo este campo no hace falta recorrer toda la cola para encolar

La estructura *Nodo* contiene los siguientes campos:

- **dato:** Es un puntero a un vértice.
- **sig:** Es un puntero al siguiente Nodo.

Este es el TAD cola mas común entonces explicaremos las funciones mas importantes:

- **Encolar:** Encolamos un elemento nuevo en la cola.
- **Decolar:** Decolamos un elemento del principio de la cola y lo devolvemos.
- **Destruir cola:** Libera los nodos de la cola, no libera el *dato* ya que es un vértice y libera la estructura *Cola*.

7 Araña colorida

7.1 Chidos

Para chidos seguimos el pseudo código dado por la catedra. Para encontrar un vértice sin colorear hacemos una busqueda lineal, y usamos el TAD cola ya explicado.

La unica diferencia es que hacemos la comprobación final mientras coloreamos preguntando si los vecinos ya coloreados tienen el mismo color.

7.2 Greedy

Para colorear con Greedy tenemos que limpiar los colores de los vertices y *G->colorV* porque son datos del coloreo viejo. Ademas llevaremos la cuenta del color más grande usado con una variable *n*.

La primera opimizacón es colorear al primer vértice con 1.

Con el resto de los vertices preguntamos si solo tiene un vecino, en ese caso lo coloreamos con dos si su vecino tiene color uno y lo coloreamos con uno si su vecino tiene otro color.

Si tiene mas de un vecino preguntamos si esta conectado con todos, en ese caso lo coloreamos con k.

Si no es ninguno de los dos casos llamamos a *colorearVertice()*.

Para colorear un vértice usamos bitwise porque podemos chequear 32 colores al mismo tiempo. Iremos marcando los colores de los vecinos en el arreglo *G->colorA*. Luego buscaremos una posición en el arreglo con un bit libre y hacemos una busqueda lineal para encontrar ese bit y al color que se refiere. Coloreamos al vértice.

Ya con el vértice coloreado actualizamos *G->colorV* y *k* si es necesario. Al terminar con todos los vertices actualizamos *G->colorN* con *n*.

8 Araña veloz

Algunos algoritmos que pensamos para optimizar el proyecto.

8.1 Generar orden aleatorio

Este algoritmo está en nuestro *main.c* y en *orden.c* con ligeras modificaciones sobre incluir o no el cero dentro del orden generado.

La idea de fondo es generar un arreglo con un orden creciente y luego hacer *swaps* aleatorios para mezclarlo. Así nos aseguramos que el orden aleatorio generado no tiene repeticiones ya que es una permutacion de un orden creciente.

Para mejorar la idea, hacemos los dos pasos al mismo tiempo. Ya que *calloc* nos inicializa el arreglo en cero, cuando nos encontremos con una posición *i* que tiene el valor cero, sabemos que deberia tener el valor *i*.

Como la primera posición tendría un cero, esto traería problemas. Entonces recorremos todo el arreglo a partir del uno, y al final *swapeamos* (intercambiamos) el cero con alguna posición aleatoria.

Así todas las posiciones habrán pasado por la mezcla.

8.2 Elegir orden aleatorio

En el *seleccionarOrden()* usamos las fracciones para cumplir con las probabilidades pedidas porque un **and 15** es mas eficiente que un **mod 100** o **mod 16**.

Ademas, para mejorar un poco más la eficiencia, pusimos los ordenamientos con mas probabilidad primero.