Laboratorio 2 de Diseño y análisis de algoritmos

Leandro Rabindranath León

1. Introducción

El fin de este laboratorio es aprehender los principios básicos y fundamentales de los recorridos sobre grafos. Para ello, tú mismo programarás diversos recorridos sobre grafos, por nodos, por arcos y los arquetipos en profundidad y en amplitud.

Este laboratorio es evaluado y la calificación que obtengas cuenta para tu calificación definitiva.

Para desarrollar exitosamente este laboratorio requieres tener instalada la biblioteca $\aleph_\omega$ (Aleph-ω); especialmente debes estar familiarizado con el tipo `List_Graph` y sus distintos métodos.

La instalación y uso de estos programas ya se te debe haber presentado y explicado en la sesión de laboratorio anterior (no evaluada).

A efectos del ahorro de memoria, en este laboratorio emplearemos el tipo `Array_Graph`, cuya interfaz es igual a `List_Graph`. La diferencia estriba en que `Array_Graph` usa arreglos en lugar de listas enlazadas para representar sus listas de adyacencia. Consiguientemente, este tipo ocupa menos memoria, pero es más lento para otras operaciones; notablemente las eliminaciones.

En este laboratorio requieres ordenar secuencias según criterios que serán dados. Revisa las interfaces de ordenamiento de la biblioteca y/o repasa algún método de ordenamiento de tu preferencia.

2. Entrada y salida

2.1. Los fuentes

Los fuentes de este laboratorio están compuestos por lo siguientes archivos:

1. `net.H`: contiene la definición de la red.

   Este es el primer archivo a estudiar.

   ¡No lo debes modificar!

2. `find.H`: contiene las declaraciones e implementaciones parciales de las funciones que deberás modificar.

   Tu entrega consiste en este archivo. Coloca los nombres, apellidos y cédulas de identidad de la pareja que está sometiendo el archivo.

3. `gen-net.[HC]`: contiene un generador de redes aleatorias. La sintaxis es la siguiente:

   ```
   ./gen-net V prob min-w max-w seed
   ```

   `V` es el número de nodos del grafo. `prob` es la probabilidad de que un par de nodos cualquiera posea un arco. `min-w` es el peso mínimo de un arco y `max-w` es el peso máximo. `seed` es la semilla de generación de números aleatorios.

4. `rand-paths.C`: contiene una batería de pruebas para las funciones en `find.H`.

   El ejecutable `rand-paths` recibe como entrada un grafo del siguiente modo:

   ```
   ./rand-paths n < grafo
   ```

   `n` es el número de pruebas que se desea hacer y grafo es un stream.

   Puede usarse en combinación con `gen-net`:

   ```
   ./gen-net 100 .01 -100 100 | ./rand-paths 10
   ```

   En este ejemplo se genera un grafo aleatorio de 100 nodos con probabilidad 0.01 y pesos comprendidos entre -100 y 100. Este grafo es pasado a `rand-paths` para efectuar 10 pruebas sobre cada función.

5. `to-dot.C`: lee un grafo y genera su visualización en `graphviz`.

   La sintaxis es;

   ```
   ./to-dot < grafo
   ```

   Donde `grafo` es un stream que contiene el grafo. La salida se da por consola con el fuente graphviz del grafo leído. Por ejemplo, la siguiente línea:

   ```
   ./to-dot < g40-10.txt | dot -T svg > g40-10.svg
   ```

   Genera una visualización en formato `svg` del grafo contenido en `g40-10.txt`. La visualización es recibida por `dot` y su salida es reenviada a `g40-10.svg`.

   Una visualización puede serte útil para corroborar casos de prueba pequeños y para depurar.

2.2. La red

2.2.1. Los nodos

El grafo conque trabajarás modeliza una hipotética red social cibernética. Los miembros de esta red están definidos de la siguiente forma:

```
struct Friend
{
  unsigned long id;
  NetColor color;

  // ..
};
```

`id` representa un identificador único de un nodo de esta red y `color` una simple clasificación.

Un color es una máscara de bits que se define del siguiente modo:

```
const NetColor Red    =  1;
const NetColor Blue   =  2;
const NetColor Yellow =  4;
const NetColor Orange =  8;
const NetColor Green  = 16;
const NetColor Black  = 32;
const NetColor White  = 64;
const NetColor Gray   = 128;
```

Nota que el color es una "máscara de bits"; es decir, cada bit representa un color. La ventaja de este enfoque es que permite fácilmente especificar que un nodo tenga varios colores. También en las búsquedas se puede especificar mediantes la operación binaria `|` (or) distintos colores. Por ejemplo, si quisiéramos expresar que un objeto de tipo `Friend` llamado `f` tenga el color rojo y negro a la vez, entonces lo haríamos del siguiente modo:

```
(f & (Red | Black)) != (Red | Black)
```

Si quisiéramos especificar que sea rojo, negro o azul entonces sería:

```
(f & (Red | Black | Blue)) != 0
```

2.2.2. Los arcos

A cada relación dentro de este grafo se le asocian los atributos definidos por la siguiente clase:

```
struct FriendShip
{
  NetColor color;
  int cost;
  // ...
};
```

`color` es un color de arco y `cost` es un coste asociado a la relación.

3. Laboratorio

Edita el archivo `find.H`, el cual contiene implementaciones parciales de las rutinas que debes programar para este laboratorio.

No olvides escribir en las primeras líneas del fuente tus apellidos, nombres y número de cédula de identidad.

3.1. Conteo de nodos según su máscara de color

Implementa la rutina:

```
size_t count_node_color(const SocialNet & net, NetColor color)
```

La cual retorna la cantidad de nodos que satisfacen la máscara de color `color`. Por ejemplo:

```
count_node_color(net, Red | Blue)
```

Retorna la cantidad de nodos que tienen colores rojo o azul.

Este problema los puedes resolver mediante un iterador de nodos del grafo o mediante el método `foldl()`.

En ambos casos no olvides que independientemente de la topología es necesario recorrer todos los nodos del grafo.

3.2. Conteo de arcos según su máscara de color

Implementa la rutina:

```
size_t count_arc_color(const SocialNet & net, NetColor color)
```

La cual es similar a la anterior pero para los arcos.

3.3. Listas de nodos por color

Implementa

```
DynList<std::pair<NetColor, size_t>> count_node_color(const Social_Net & net)
```

La cual retorna una lista ordenada por color de pares donde `first` es el color y `second` es la cantidad de nodos con el color `first`.

Para esta rutina puedes usar un iterador sobre los nodos del grafo o una función de alto orden aplicada sobre el operador `for_each_node()`; también podrías usar `foldl()`.

Nota que dos valores de color distintos representan colores distintos (no es tan obvio)). Por ejemplo, `Red | Blue | Yellow` es distinto a `Red | Blue`. En este sentido, pueden haber hasta 255 colores distintos.

Es importante detectar cuando un color es visto por primera vez, así como llevar la cuenta por cada color. Para ello, debes mantener alguna especie de mapeo entre

el color y la cantidad de nodos. Una tabla hash o un árbol binario son las opciones escalables, pero en este caso, dado que sólo hay 255 combinaciones, quizá un arreglo te sea más fácil.

Para tablas hash puedes usar el tipo `DynMapHash<Key,Data>`, contenido `tpl_dynSetHash.H`. Para árboles binarios puedes usar `DynMapTree<Key,Data>`, contenido en `tpl_dynSetTree.H`. Usa el foro para discutir la interfaz o la implementación del mapeo con arreglos. En caso de que uses una tabla hash, no olvides que el resultado debe estar ordenado por color.

3.4. Listas de arcos por color

Implementa;

```
DynList<std::pair<NetColor, size_t>> count_arc_color(const Social_Net & net)
```

La cual es similar a la de la sección anterior pero esta vez para contar los arcos.

3.5. Búsqueda de nodos según máscara de color

Implementa:

```
DynList<SocialNet::Node*> nodes(const SocialNet & net, NetColor color)
```

La cual retorna una lista inversamente ordenada por `id` de nodos cuyos colores se corresponden con la máscara `color`.

Nota que aquí es por máscara. Por ejemplo:

```
nodes(net, Red | Blue)
```

Retorna los nodos que tienen el bit de color `Red` o `Blue` en uno.

Recuerda que el resultado debe estar ordenado inversamente por `id` del nodo.

Ayuda 1: la función `sort()` de `ahSort.H` ordena una lista o arreglo.

Ayuda 2: el método `filter()` recibe una función de primer orden o lambda que retorna `true` si el elemento debe ser filtrado. Con esta función y `sort()` podrías escribir esta búsqueda en una sola instrucción.

Si estas ayudas te son enredadas, entonces no te enrolles y programa proceduralmente usando iteradores.

3.6. Búsqueda de arcos según máscara de color

Programa:

```
DynList<SocialNet::Arc*> arcs(const SocialNet & net, NetColor color)
```

La cual retorna una lista de arcos ordenada por costo cuyos colores se corresponden con la máscara `color`. En caso de empate con los costes, desempata con los colores de los arcos. Si se te presenta un doble empate, entonces desempata con los `id` s de los nodos resultantes de la llamada a `get_src_node()` sobre el arco proporciona

el desempate. Y si resulta que aquí también hay empate, entonces el desempate se hace con `get_tgt_node()`[1].

Aunque la implementación es similar al problema anterior ten en cuenta que el ordenamiento es por el costo del arco.

3.7. Búsqueda de caminos en profundidad

Hasta el presente todos los programas anteriores se pueden resolver con iteraciones sobre los nodos o arcos. En este problema y otros siguientes deberás recorrer el grafo topológicamente; es decir, según las relaciones de conexión.

Programa la función:

```
Path<SocialNet> find_path_depth_first(const SocialNet & net,
                                      const unsigned long start_id,
                                      const unsigned long end_id,
                                      const NetColor node_colors,
                                      const NetColor arc_colors)
```

La cual retorna un camino en profundidad desde el nodo con `id start_id` hasta el nodo con `id end_id`. A la excepción del origen y destino, el camino debe estar compuesto por nodos que se correspondan con la máscara de color `node_colors`. Similarmente, los colores de los arcos del camino deben corresponderse con la máscara de color `arc_colors`.

Para asegurar la homogeneidad en los resultados, el orden de visita de los arcos debe hacerse según el grado del nodo destino (número de arcos del nodos destino). Es decir, el nodo con mayor grado se visita de primero. Los empates se deciden por el `id`

3.7.1. Recomendaciones para este y el siguiente problema

La dificultad de este problema subyace en dos obstáculos. El primero es el filtrado de nodos y arcos según los colores. Hay varias maneras de hacerlo. Puedes usar iteradores filtro, en cuyo caso debes programar genéricamente los functors de filtro. Si lo haces así, entonces podrías basarte en el código de la biblioteca para instrumentar tus recorridos.

El segundo obstáculo es el requerimiento de orden de visita de arcos según el grado del nodo destino.

Así las cosas, el enfoque recomendado es obtener una secuencia de arcos a recorrer. Esto lo puedes hacer muy fácilmente con el método `arcs(src)`, el cual devuelve una lista de arcos relacionados con el nodo `src`. Una vez que tengas esta lista, entonces puedes filtrar por colores y luego ordenarla por los grados de los nodos destino.

Los recorridos de $\aleph_\omega$ (Aleph-ω) mantienen el camino en línea, en una variable de tipo `Path`. Este enfoque es costoso en tiempo y espacio, especialmente si el recorrido es en amplitud.

[1] Puesto que un arco está definido por nodos, en un grafo no se concibe, porque no es necesario, un `id` para arcos. La meta de este requerimiento de ordenamiento es que entiendas esta observación.

Una manera más económica consiste en almacenar en el `cookie` de cada nodo `p` "su padre" en el recorrido; es decir, el nodo desde el cual `p` fue descubierto. Este truco lo puedes hacer antes de recursionar o de meter en la pila (si la implementación de tu recorrido es iterativa). Con este truco, una vez que encuentres el nodo destino los `cookie` s contienen el camino invertido. Así que lo que te restaría es una última fase en la cual construyes el camino a partir de los `cookies` s.

Recuerda que los nodos y arcos tienen bits de control y contadores. Úsalos porque es más sencillo con ellos reconocer que se ha visitado y que no. No olvides reiniciar los nodos y arcos antes del recorrido.

Estudia el recorrido en amplitud, pues aparte de ser requerimiento en el siguiente problema, puedes economizar código y pensamiento si implementas el recorrido en profundidad iterativamente, con la misma estructura del recorrido en amplitud, pero en lugar de usar una cola usas una pila.

3.8. Búsqueda de caminos en amplitud

Programa

```
Path<SocialNet> find_path_breadth_first(const SocialNet & net,
                                        const unsigned long start_id,
                                        const unsigned long end_id,
                                        const NetColor node_colors,
                                        const NetColor arc_colors)
```

La cual retorna un camino en amplitud desde el nodo con `id` `start_id` hasta el nodo con `id` `end_id`. Los nodos y arcos son filtrados y ordenados de la misma forma que en problema anterior.

3.9. Cálculo del coste de un camino

Programa:

```
long long cost(Path<SocialNet> & path)
```

La cual retorna el coste total del camino `path`.

3.10. Búsqueda genérica de caminos (opcional)

Las búsquedas de caminos sobre grafos son estructuralmente similares. La diferencia estriba en el modo en que se priorizan las siguientes visitas, lo cual puede determinarse por una estructura de dato orientada hacia el flujo: una pila o cola según sea el caso.

Si entiendes bien lo anterior, o si implementaste el recorrido en profundidad iterativo, entonces puede interesarte programar

```
template <template <typename T> class Q>
Path<SocialNet> find_path(SocialNet & net,
   const unsigned long start_id,
```

```
const unsigned long end_id,
const NetColor node_colors,
const NetColor arc_colors)
```

Cuya interfaz y funcionalidad es similar a los recorridos anteriores, con la diferencia de que el tipo de recorrido está determinado por la estructura `Q` (que es un parámetro tipo).

4. Evaluación

La fecha de entrega de este laboratorio es el jueves 6 de octubre hasta la 11:30 am. Puedes entregar antes, recibir corrección e intentar hasta un máximo de tres intentos. Tu nota será la otorgada en el último envío.

Para evaluarte debes enviar el archivo `find.H` a la dirección:

`alejandro.j.mujic4@gmail.com`

Antes de la fecha y hora de expiración. El "subject" debe **obligatoriamente** contener el texto **AYDA-LAB-02**. En el mensaje y en el inicio del fuente del programa debes colocar los nombres, apellidos y números de cédula de la pareja que somete el laboratorio. **Por favor, no uses otros medios para enviar la solución ni escribas otros comentarios adicionales en el email**.

La evaluación esta dividida en dos partes. La primera parte es ejecutada automáticamente por un programa evaluador compilado con tu fuente. El programa genera un grafo aleatorio con una semilla dada, por cada una de tus rutinas ejecuta una serie de prueba (aproximadamente 10) y las compara con las pruebas de referencia. La más mínima desavenencia implica que el test particular tiene 0 puntos. Por cada test que falle, se te reportará la entrada que causó la falla; de este modo, aunado a la semilla aleatoria, podrás reconstruir el grafo sobre el cual se ejecutaron las pruebas y con él examinar, verificar y eventualmente corregir tu programa. La evaluación de la primera parte representa aproximadamente 94 % de la calificación.

Atención: si tu programa no compila, entonces el evaluador no compila. Si una de tus rutinas se cae, entonces el evaluador se cae. Si una de tus rutinas cae en un lazo infinito o demora demasiado, entonces el evaluador cae en un lazo infinito o se demora demasiado. Exactamente sucedería lo mismo si tu programa fuese parte de un sistema de aviónica: se estrella el avión. Por esa razón, en todos estos casos será imposible darte una nota ... o simplemente tienes cero. Si una de tus rutinas dispara una excepción inesperada, el evaluador hará lo posible por capturarla y reportártela; sin embargo, esto no es un compromiso; podría estrellarse el avión. En todo caso, si el evaluador no logra capturar la excepción, no se te dará nota pero se te reportará la excepción.

La segunda parte es subjetiva y manual. Consiste en revisar tu fuente con la intención de evaluar tu estilo de codificación. Es recomendable que sigas las reglas de estilo comentadas en clase, pues estas armonizan con la persona que subjetivamente te evaluará. Esta parte aporta aproximadamente el 6 % de la nota.

Una suite de casos de prueba se adjunta con este laboratorio. Si los pasas, entonces es muy probable que pases los casos conque te evaluaremos.

5. Recomendaciones

Una dificultad con los grafos es el diseño de casos de prueba generales.

Los grafos de prueba fueron generados al azar. Como verás en la distribución, los nombre contienen dos números. El primero indica la cantidad de nodos, el segundo la probabilidad de conexión entre cualquier par de nodos. Por ejemplo, `g100-10.txt` indica que el grafo tiene 100 nodos con una probabilidad de conexión entre cualquier par de nodos de 0.1; o sea un 10 % de densidad.

Escoge un grafo sencillo, por ejemplo `g40-10.txt`, el cual es dibujable con `graphviz` y diseña tus casos de prueba y soluciones manualmente. Luego, verifica que tus rutinas den la solución esperada. Una vez que cumplas este paso, entonces compara con la salida de `paths`. Finalmente, prueba con grafos más grandes y diseña otros casos de prueba.