Laboratorio 3 de Diseño y análisis de algoritmos

Leandro Rabindranath León / Alejandro Mujica

Fecha de entrega: viernes 4 de noviembre de 2016

1. Introducción

El fin de este laboratorio es aprehender reforzar los principios básicos y fundamentales de la conectividad sobre grafos euclidianos, así como reforzar y experienciar el cálculo de caminos mínimos.

Para desarrollar exitosamente este laboratorio requieres tener instalado:

1. La suite clang
2. 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.

Puedes implementar tu propios algoritmos, emplear algunos de los de $\aleph_\omega$ (Aleph-ω) o adaptarlos o mejorarlos según tus preferencias.

1.1. Puntos en un plano

En este laboratorio construirás grafos euclidianos. Considera n puntos en un plano definidos mediante el siguiente sinónimo:

```
using Point = tuple<double,double>;
```

El primer elemento representa, por mera convención, la coordenada x de un punto en un plano, mientras que el segundo la coordenada y.

Para identificar unívocamente los puntos y poder verificar tus soluciones, asociaremos a cada punto un `id` único. De este modo, definimos el siguiente sinónimo:

```
using City = tuple<Point, unsigned long>;
```

O sea, una dupla cuyo primer elemento es un punto en el plano y segundo elemento un `id` único.

La dupla es llamada `City`, sólo por facilitar el discurso, pues en cierta forma, una ciudad se representa mediante un punto (latitud y longitud) en un plano esférico y su nombre (el `id`).

1.1.1. Generando puntos al azar

La siguiente rutina, provista, genera una lista al azar de `num_points` ciudades:

```
DynList<City> cities(const size_t num_points,
                     const double xsize,
                     const double ysize,
                     unsigned long seed)
```

`xsize` y `ysize` son el ancho y altura del plano en dónde se pondrán los puntos. `seed` es la semilla del generador de números aleatorios.

1.1.2. Grafo euclidiano de puntos

Las conexiones entre las ciudades pueden definirse mediante un grafo euclidiano de la siguiente manera:

```
using Gmap = Array_Graph<Graph_Anode<City>, Graph_Aarc<double>>
```

Los nodos contienen ciudades y los arcos la distancia euclidiana entre ellas.

2. Laboratorio

Tu trabajo en este laboratorio será implementar la clase `Points` incluida en `min-dist.H`. Básicamente, `Points` almacena un conjuntos de puntos en un plano y brinda soporte para calcular grafos euclidianos según una distancia de cobertura.

Lee y comprende bien los métodos de `Points` antes de diseñar su estado interno.

Lee los archivos dados con este laboratorio. Se te adjunta un probador sencillo.

No se provee `Makefile`.

2.1. Constructor

Programa

```
Points(const DynList<City> & l)
```

`l` es una lista de "ciudades".

Piensa en las oportunidades que presenta este constructor para "construir" estado interno que luego sirva a futuros cálculos en pos de alto desempeño. Sin embargo ten cuidado con la eficiencia. El tiempo de construcción no sólo es evaluado, sino que es parte en el límite de tiempo del evaluador.

2.2. Construcción de grafo euclidiano

Programa el método:

```
Gmap build_gmap(const double span_dist)
```

El cual retorna un grafo euclidiano de los puntos almacenados en `this` que se encuentran separados por una "distancia de cobertura" `span_dist`. Dadas dos ciudades `c1` y `c2`, ellas están conectadas si y sólo si su separación es menor o igual a `span_dist`.

Los nodos del grafo resultante deben contener exactamente las ciudades dadas en el constructor.

La precisión contada para los arcos será de $\epsilon = 10^{-6}$.

2.3. Obtención de punto según `id`

Programa

```
const Point & get_point(const unsigned long id) const
```

La cual retorna un punto asociado a un `id`.

2.3.1. Número de componentes conexos

Programa

```
static size_t num_blocks(Gmap & g)
```

La cual retorna la cantidad de componentes conexos de `g`.

Nota que la rutina es estática, lo que se traduce a decir que no requiere `this` para ser ejecutada; sólo requiere el grafo.

Puedes usar el soporte de $\aleph_\omega$ (Aleph-ω), pero recuerda el factor de desempeño de este laboratorio.

2.4. Cálculo de componentes conexos

Programa

```
static DynList<DynList<Gmap::Node*>> blocks(Gmap & g)
```

El cual retorna una lista de los componentes conexos del grafo.

Cada componente conexo está representado por una lista ordenada por `id` de punteros a los nodos que lo conforman.

A su vez, la lista de componentes (o bloques) está ordenada por el menor `id` del componente.

2.5. Cálculo de camino mínimo entre dos ciudades

Programa

```
static
Path<Gmap> min_path(const Gmap & g, unsigned long src_id, unsigned long tgt_id)
```

La rutina retorna un camino mínimo desde `src_id` hasta `tgt_id`. Si tal camino no existe, entonces se retorna un camino vacío.

Si unos de los `id` es inválido, entonces se debe generar la excepción `domain_error(''node id not found'')`,

En caso de que prefiera usar $\aleph_\omega$ (Aleph-ω), concretamente el algoritmo de Dijkstra, para calcular caminos mínimos, por favor usa la versión de `Dijkstra.H` que se incluye en el tar de este laboratorio. La versión actual de $\aleph_\omega$ (Aleph-ω) (y probablemente tampoco las anteriores) no te compilará.

2.6. Cálculo de la mínima distancia de cobertura

Programa

```
double min_connectivity_distance()
```

El cual retorna la mínima distancia que cubre a todos los puntos. En otras palabras, el mínimo valor de `span_dist` tal que el grafo construido mediante `build_gmap(span_dist)` es conexo.

Por simplicidad, el valor de mínima distancia de cobertura es un entero. O sea el mínimo valor entero de distancia tal que se cubran todos los puntos.

Ayuda: Piensa en el principio de la búsqueda binaria y el algoritmo de Kruskal.

Advertencia: si no prestas atención a la algorítmica y complejidad, tu tiempo de ejecución será $\mathcal{O}(V^3)$ o peor y el evaluador se bloqueará ejecutando esta rutina. Si no logras una complejidad de a lo sumo $\mathcal{O}(V^2 \lg V)$ o mejor, entonces no hagas esta rutina.

En $\aleph_\omega$ (Aleph-ω) existe una clase denominada `Fixed_Relation` (`tpl_union.H`) que podría serte útil. *Grosso modo* su interfaz es la siguiente:

- `Fixed_Relation(size_t n)`: construye una relación de equivalencia (reflexiva, transitiva y simétrica) vacía, sin pares, de `n` elementos. Lo fijo (`Fixed`) consiste en que los elementos son enteros entre 0 y $n-1$.
- `bool are_connected(size_t i, size_t j)`: retorna `true` si i está relacionado, directa o transitivamente, con j.
- `void join(size_t i, size_t j)`: relaciona i con j[1].

[1] En caso de que te preguntes por qué no usar el término `union`, cual es más afín al sentido de la operación y a nuestro lenguaje sin perder el carácter anglosajón de nombramiento de la biblioteca, la respuesta es que `union` es una palabra reservada de `C ++`

- `size_t get_num_blocks()`: retorna el número de bloques de la relación.

  Inicialmente, la relación vacía `r` contiene n bloques, pues no existe ninguna conexión.

  La primera vez que se relaciona algún par (i, j) el número de componentes desciende a $n-1$, pues i, j conforman un componente. Si efectuamos `r.join(i,k)` entonces el número de componentes desciende a $n-2$. Igual habría ocurrido si se hubiese ejecutado `r.join(k,i)` o `r.join(j,k)` o `r.join(k,j)`, pues recuerda que la relación es simétrica.

  Si ejecutamos `r.join(x,y)` el número de componentes desciende a $n-3$, pero nota que (x, y) estaría disyunto respecto a la clase de equivalencia $\{i, j, k\}$

  Como se explicó en clase, una relación de equivalencia es un grafo dirigido en el cual cada clase de equivalencia es un componente inconexo.

  Ahora bien, cuando el número de bloques es uno, entonces podemos concluir que el grafo que define a la relación es conexo.

Esta clase tiene dos ventajas grandiosas:

1. Es mucho más simple que un grafo de $\aleph_\omega$ (Aleph-ω).
2. Sus operaciones son $\mathcal{O}(1)$ en tiempo amortizado. O sea ...

3. Evaluación

La fecha de entrega de este laboratorio es el viernes 4 de noviembre de 2016 hasta la 6pm. Puedes entregar antes, recibir corrección y enviar hasta un máximo de 3 intentos supeditado a un intento por día.

Para evaluarte debes enviar el archivo `min-dist.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-03**. 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**.

El programa genera un grafo aleatorio con una semilla dada, por cada una de tus rutinas ejecuta una serie de pruebas (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.

Atención: si tu programa no compila, entonces el evaluador no compila. Por favor, **no asumas que una compilación exitosa de tu parte implica una exitosa del evaluador**. 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.

4. Recomendaciones

1. Esmérate primero en asegurarte de que tu implementación esté correcta. Luego, sólo si tienes suficiente tiempo, intenta acelerar los tiempos de respuesta. No te enrolles demasiado por la rapidez, pues nuestra implementación de referencia transó por la simplicidad y correctitud más que por la eficiencia.

2. Diseña tus casos de prueba. Piensa en condiciones límites; por ejemplos, un grafo sin arcos, o un camino (inexistente) entre dos nodos inconexos.

3. Ve por fases, desde las más simples hasta la más complejas. No pases a una siguiente fase hasta que no hayas probado rigurosamente que todo está bien.

4. Ten sumo cuidado con `min_connectivity_distance()`, tanto por su correctitud como por su desempeño. Especialmente, ten cuidado con no caer en un tiempo cúbico o peor porque si no el evaluador no terminará y no se te evaluará (ni corregirá).

5. La manera en que construyas en grafo puede ser determinante para los tiempos de ejecución. Recuerda la trigonometría.

6. $\aleph_\omega$ (Aleph-ω) posee soporte parcial o completo para implementar muchas de las rutinas que se te piden. Esta es pues una vía expedita para hacer este laboratorio y sus rendimientos son aceptables.

7. Es probable que programes recorridos en profundidad recursivos o uses rutinas parecidas de $\aleph_\omega$ (Aleph-ω). En ese caso, grafos muy grandes pueden causar un desborde de la pila del sistema del proceso. Usa la rutina de $\aleph_\omega$ (Aleph-ω) `resize_process_stack()` para agrandar el tamaño de la pila del sistema.

8. **¿Usa el Piazza!**; pide ayuda públicamente cuándo la necesites, pero plantea bien la índole de tu problema. Ayuda a tus compañeros participando en la discusión o aclarando dudas.