Proyecto 4 - Algoritmo de Kruskal

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Leonardo Rodríguez, Diego Dubois, Santiago Ávalos, Gonzalo Peralta, Jorge Rafael.

1. Objetivo

Este proyecto tiene como finalidad implementar el *algoritmo de Kruskal* (http://en.wikipedia.org/wiki/Kruskal%27s_algorithm) para encontrar *árboles generadores de costo mínimo* (MST) (http://en.wikipedia.org/wiki/Minimum_spanning_tree).

Para ello se deberán implementar los TAD's indicados más abajo, utilizando las técnicas aplicadas en los proyectos anteriores. Para cada TAD se describe el conjunto mínimo de funciones que debe proveer.

Como se vio en el teórico, el algoritmo Kruskal tiene orden $\mathcal{O}(n.log(n))$ donde n es la cantidad de aristas del grafo. Las implementaciones de los TAD's deben garantizar esta complejidad.

2. Descripción

En la figura 1, se muestra un diagrama de los tipos abstractos de datos opacos necesarios para la resolución de este proyecto. Como en los otros proyectos, los módulos resaltados con **verde** son provistos por la cátedra (grafo, arista, vértice), y los resaltados en **rojo** deberán ser implementados por los alumnos con la técnica de puntero a estructura, para ocultar los detalles de implementaciones. Se provee con este enunciado un esqueleto de código con esos archivos.

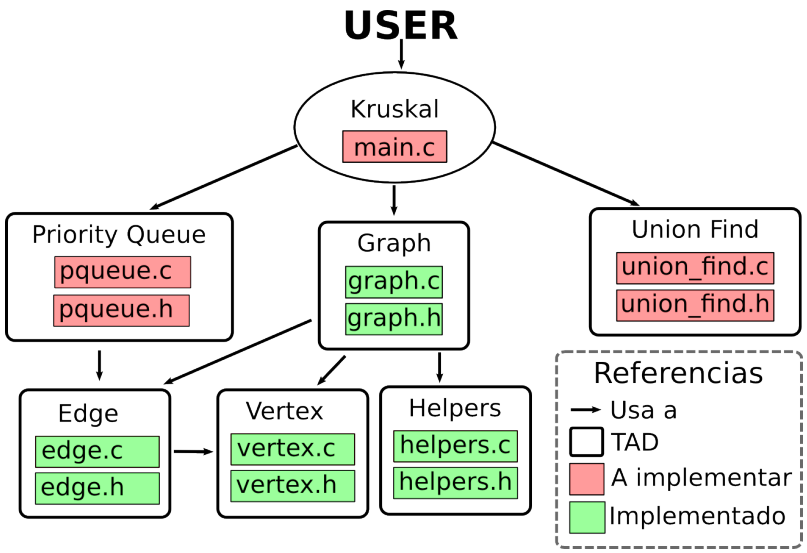


Figura 1: Diagrama de TADs

2.1. Main

En el archivo `main.c` se debe crear una función con la siguiente signatura:

`graph_t kruskal(graph_t g)`

Esta función devuelve **una copia** del grafo g pero con las aristas que forman parte del MST marcadas como primarias, y las aristas que no formar parte del MST marcadas como secundarias. Para marcar una arista del grafo como primaria o secundaria usar la función `edge_set_primary` provista por el tipo `edge_t`.

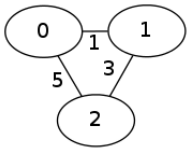
Como guía para implementar el algoritmo se puede utilizar el siguiente pseudocódigo:

```
fun kruskal (grafo g) ret: grafo
  V   := {0, ... , N} := vértices de g.
  E   := {e_0, ... , e_M} := aristas de g.
  C   := {{0}, ... , {N}} (conjunto de conjuntos de vértices).
  Q   := cola de prioridades con todas las aristas E.
  g'  := grafo vacío.
  do Q no vacía y |C| > 1 →
      e  := primera arista de la cola Q (arista en Q con menor peso).
      (l, r) := vértices de e.
      L := conjunto de C que contiene a l.
      R := conjunto de C que contiene a r.
      if L ≠ R →
          marcar e como primaria.
          C := C - {L, R}.
          C := C ∪ {L ∪ R}.
      else if L = R →
          marcar e como secundaria.
      end if.
      agregar e en g'.
      decolar en Q.
  od
  agregar en g' las aristas que queden en Q (si es que quedan) marcadas
  como secundarias.
  devolver g'.
end fun
Postcondición:
- g y g' tienen las mismas aristas.
- las aristas primarias de g' forman un MST de g'.
```

Como se deduce del pseudocódigo, el algoritmo analiza las aristas del grafo una por una, y decide si deben formar parte o no del MST. La cola de prioridades (TAD `pqueue_t`) se utiliza para elegir primero las aristas con menor peso, y así garantizar que el MST sea en efecto de costo mínimo.

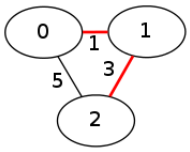
El conjunto de conjuntos `C` que menciona el algoritmo se utiliza para ir llevando un registro de cuáles son los vértices que hasta el momento están unidos por aristas del MST (aristas primarias). El TAD `union_find_t` se utiliza para implementar esa funcionalidad.

Como un pequeño ejemplo, supongamos que aplicamos el algoritmo al siguiente grafo:



Al comienzo tenemos `C = {{0},{1},{2}}`. La arista con menor peso es `0--1`, y como esa arista conecta vértices de conjuntos distintos, la elegimos para formar parte del MST, y actualizamos `C = {{0,1},{2}}`. La siguiente arista es `1--2`, que nuevamente conecta vértices de conjuntos distintos y formará parte del MST;

actualizamos entonces `C = {{0,1,2}}`. La siguiente arista es `0--2`, pero tanto 0 como 2 están en el mismo conjunto, asi la marcamos como secundaria. En este punto el algoritmo finaliza y devuelve el siguiente grafo:



Las aristas en rojo son las aristas primarias, que forman el MST.

En `main.c` proveer también la siguiente función

```
unsigned int mst_total_weight(graph_t mst)
```

que devuelve el costo total del MST, que se obtiene simplemente sumando los pesos de las aristas primarias.

2.2. Priority Queue

Este TAD permitirá ir obteniendo las aristas priorizándolas de acuerdo a su peso. En el archivo `pqueue.h` se provee la interfaz a implementar. Se recomienda utilizar como guía el pseudocódigo del apunte teórico, pero tener **cuidado** de que en este proyecto implementarán una cola de prioridades por **mínimo** (aristas de menor peso tienen mayor prioridad) por lo tanto es necesario realizar algunas modificaciones a ese pseudocódigo.

(ver http://en.wikipedia.org/wiki/Priority_queue)

2.3. Union Find

Este TAD *union-find* (también conocido como *disjoint-set*) debe proveer operaciones para hacer de forma eficiente las uniones de conjuntos y la búsqueda del conjunto al que pertenece un vértice.

La implementación que deben realizar es el "Último intento" que se explica en el apunte teórico. Esta implementación es la más eficiente ya que efectua una *compresión de caminos*, y cuando realiza una unión elige siempre el representante con mayor cantidad de elementos.

(ver http://en.wikipedia.org/wiki/Disjoint-set_data_structure)

3. Formato de los archivos que describen grafos

3.1. Archivo de entrada para cargar grafos

El módulo `graph` provee una función para cargar automáticamente, en memoria, grafos descriptos en un archivo con una sintaxis específica. Esta función es:

```
graph_t graph_from_file(FILE *fd);
```

El archivo que lee esta función debe comenzar con una linea de texto que indicará la cantidad de vértices y de aristas que contiene el grafo, precedido del símbolo #.

```
# <cantidad de vértices> <cantidad de aristas>
```

Por ejemplo para un grafo de 7 vértices y 10 aristas el archivo debe comenzar con:

```
# 7 10
```

A continuación deberá ir la palabra reservada `graph` seguida por un nombre del grafo y el símbolo `{`, por ejemplo:

```
# 7 10
graph G {
```

En las líneas siguientes se listan las aristas separadas por el símbolo `;`, y al final se cierra la definición con `}`. Por ejemplo, el grafo de la figura 2 se puede escribir como se muestra a continuación:

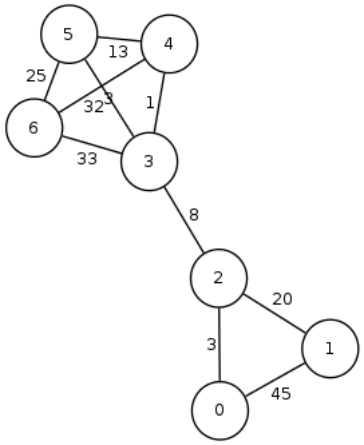


Figura 2: Grafo ejemplo

```
# 7 10

graph G {
  0 -- 1 [label=45];
  0 -- 2 [label=3];
  2 -- 3 [label=8];
  2 -- 1 [label=20];
  4 -- 5 [label=13];
  4 -- 3 [label=1];
  5 -- 3 [label=32];
  6 -- 3 [label=33];
  6 -- 4 [label=3];
  6 -- 5 [label=25];
}
```

Notar que los números a cada lado de "--" son las claves de los vértices y donde dice `[label=<w>]` el `<w>` es el peso de la arista que se usa en el algoritmo.

Los vértices estarán numerados desde el 0 hasta N - 1, donde N es la cantidad de vértices del grafo.

Para obtener un archivo de imagen PNG de un grafo descripto por el formato especificado, se puede correr el siguiente comando (suponiendo que el archivo que define el grafo se llama `test.dot`):

```
$ neato -Tpng -o output.png test.dot
```

Esto creará un archivo `output.png` con el dibujo del grafo.

3.2. Archivo de salida

El formato de salida será similar al de entrada, con la particularidad de que se deberá distinguir las aristas que pertenecen al árbol generador de las que no. Por ejemplo, para el árbol generador de costo mínimo del ejemplo anterior obtendríamos algo como lo que está en la figura 3:

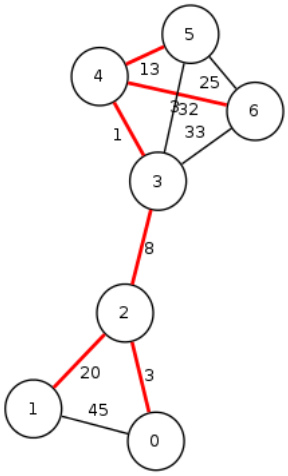


Figura 3: Grafo de salida

que se codifica en un archivo de la siguiente forma:

```
# 7 10

graph G {
        4 -- 3 [label=1, color=red, style=bold];
        0 -- 2 [label=3, color=red, style=bold];
        6 -- 4 [label=3, color=red, style=bold];
        2 -- 3 [label=8, color=red, style=bold];
        4 -- 5 [label=13, color=red, style=bold];
        2 -- 1 [label=20, color=red, style=bold];
        6 -- 5 [label=25];
        5 -- 3 [label=32];
        6 -- 3 [label=33];
        0 -- 1 [label=45];
}

# MST : 48
```

Notar que la última linea tiene el costo total del MST.

3.3. Ejecutar Kruskal

Una vez que uno tiene compilado el ejecutable `kruskal`, se puede pasar el contenido de un archivo en disco, por entrada estándar, para que el algoritmo lo procese, haciendo:

```
$./kruskal < input/test.dot
```

Asimismo, usando el operador > junto con lo anterior, se puede redirigir lo que se imprima en la salida estándar para que se guarde directamente en un archivo (y así luego poder generar el `.png`).

En resumen, con el siguiente comando se puede correr el algoritmo de Kruskal, usando como archivo de entrada `input/test.dot` y guardando la salida en `result.dot`, y luego generando la imagen en un archivo `output.png`.

```
$ valgrind --leak-check=full --show-reachable=yes ./kruskal < input/test.dot > result.dot
$ neato -Tpng -o output.png result.dot
```

Si se desea se pueden agregar otras opciones para dibujar el grafo (ver man page de neato).

3.4. Recomendaciones, antes de empezar a escribir código

- Estudiar el apunte del teórico de union-find y cola de prioridades.
- Entender el objetivo y funcionamiento del algoritmo de Kruskal.
- Revisar y entender la interfaz provista por `graph.h`.
- Como parte del código inicial, se incluyen varios grafos de entrada en el directorio `input/`. Probarlos y verificar el resultado! Producir nuevos casos de ejemplo.

3.5. Requerimientos adicionales

En este proyecto deberán escribir un Makefile. Todos los archivos .c deben pasar el test de estilo.

4. Estrellas (sólo cuando terminen el básico!)

4.1. Ciclos y componentes conexas

Crear dos funciones extra en `main.c`:

```
bool graph_has_cycle(graph_t g)
unsigned int graph_connected_components(graph_t g)
```

La primera devuelve `true` si el grafo de entrada tiene un ciclo (y `false` en caso contrario). La segunda devuelve la cantidad de componentes conexas que tiene el grafo de entrada.

Adicionar a la salida del programa la cantidad de componentes conexas del grafo, y si tiene ciclos o no. Por ejemplo, al correr `./kruskal < input/test12.dot` debería imprimirse en pantalla, además del grafo, la siguiente información:

```
# Has a cycle: YES
# Connected components: 3
```

4.2. Estrella supernova: Implementación en Java

Implementar el algoritmo de Kruskal en Java, respetando el formato de archivo descripto en este enunciado. La implementación es libre y se pueden usar los TADs ya programados por Robert Sedgewick (entre otros):
http://algs4.cs.princeton.edu/24pq/MinPQ.java.html
http://algs4.cs.princeton.edu/15uf/UF.java.html

Proveer instrucciones para compilar (Makefile) y para testear con los archivos de ejemplo.

5. Entrega y evaluación

- Fecha de entrega del código y parcialito: Martes 16 de Junio (sin posibilidad de cambiar la fecha).
- Fecha de recuperatorio del LAB: Jueves 18 de Junio (sin posibilidad de cambiar la fecha).
- El programa resultante **no** debe tener *memory leaks* **ni** accesos (read o write) inválidos a la memoria.
- En este proyecto no se permite usar recursión ni variables globales.