Proyecto 4 - Algoritmo de Kruskal

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Natalia Bidart, Matías Bordese, Diego Dubois, Leonardo Rodríguez.

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* (`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. Algoritmo Kruskal

Para implementar el algoritmo se utilizarán varios TADs. Algunos son provistos por la cátedra, a saber:

- Los tipos vértice (`vertex_t`), arista (`edge_t`), y grafo (`graph_t`).

junto con una serie de funciones que permiten manipular las respectivas instancias de estos TADs (ver las bibliotecas `graph` y `helpers`). Y los TADs que deben implementar los alumnos:

- Heap de aristas de grafo, implementado con un arreglo, como se vio en el teórico (`heap_t`).
- Cola de prioridades, implementada usando el heap del punto anterior. Este TAD será un sinónimo del TAD `heap` (`priority_queue_t`).
- Conjunto de conjuntos de vértices (también conocido como conjunto de forestas, *disjoint-set*, o *union-find*). Este TAD debe estar implementado de forma tal que las uniones de conjuntos y la búsqueda del conjunto al que pertenece un vértice sea eficiente, como se estudia en el teórico (`union_find_t`).
- Stack de aristas, para guardar en memoria los resultados intermedios (`stack_t`).

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, y los resaltados en **rojo** deberán ser implementados con la técnica de puntero a estructura, para ocultar los detalles de implementaciones.

Es importante notar que se deberá proveer, además de los `*.c` y `*.h`, un archivo Makefile para compilar el proyecto.

Entonces, con estos TAD's se puede construir el algoritmo Kruskal como se muestra en **Algoritmo 1** (pag. 2). Tener en cuenta que lo siguiente es pseudocódigo, aunque la implementación final en C debería ser similar, por lo cual hay que tener especial atención a mapear las construcciones de control (**do**, **if**, etc) de manera adecuada a C.

Algoritmo 1 Algoritmo Kruskal en pseudocódigo

```
⟦ Var pq : priority_queue;
       uf : union_find;
       mst_edges, unused_edges : stack;
       graph, mst : graph;
       e : edge;
       class1, class2 : union_find_class
  ⋮
  leer grafo de la entrada estándard y almacenarlo en graph
  ⋮
  construir los demás TAD's (cola de prioridades, union find)
  ⋮
  do graph_has_unprocessed_edges(graph) →
     e := next_unprocessed_edge(graph)
     pq := enqueue(pq, e)
     uf := add(uf, left_vertex(e))
     uf := add(uf, right_vertex(e))
  od

  do length(uf) > 1 ∧ ¬is_empty(pq) →
     e := first(pq)
     pq := dequeue(pq)
     class1 := find_class(uf, left_vertex(e))
     class2 := find_class(uf, right_vertex(e))
     if class1 ≠ class2 →
        uf := union(uf, class1, class2)
        mst_edges := push(mst_edges, e)
     □ class1 = class2 →
        unused_edges := push(unused_edges, e)
     fi
  od
  ⋮
  contruir el grafo mst usando las aristas de mst_edges
  ⋮
  agregar a mst las aristas no usadas, y marcarlas como secundarias
  ⋮
  hacer el dump de mst a la salida estándar
  ⋮
  destruir todos los TAD's
  ⋮
⟧
```

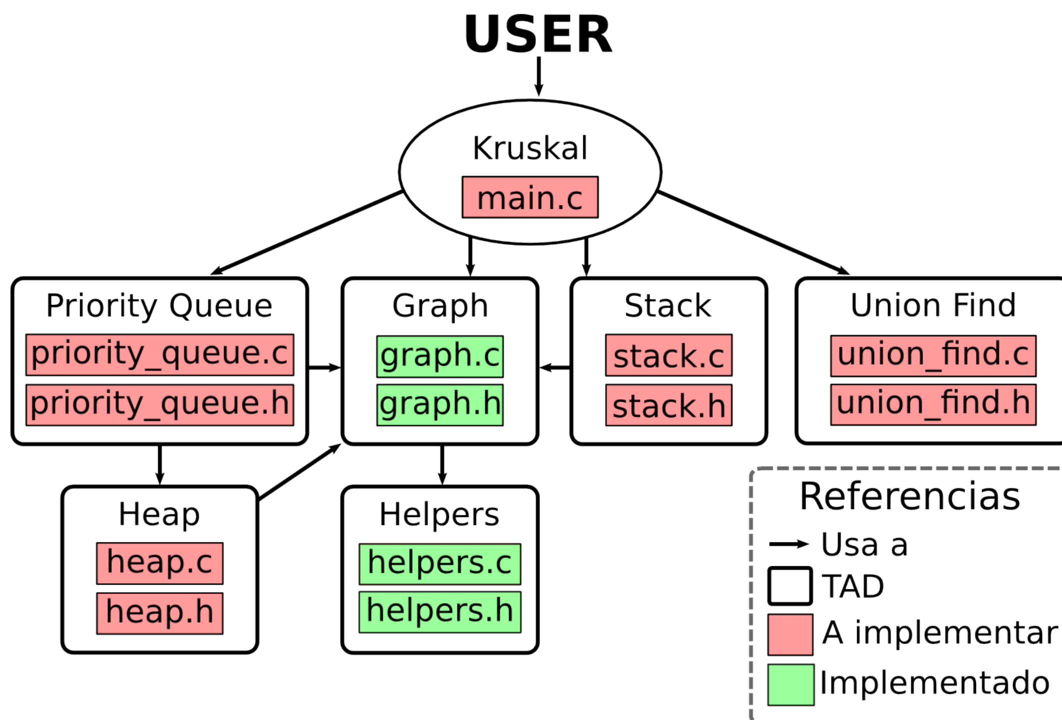

Figura 1: Diagrama de TADs

3. Detalles de los TADs

3.1. Heap

Necesario para implementar la cola de prioridades. Debe ser un heap **por mínimo**, ya que nos interesa poder recorrer las aristas, según su peso, de menor a mayor. A continuación la interfaz a implementar:

```
#ifndef _HEAP_H
#define _HEAP_H

#include <stdbool.h>

#include "graph.h"


/* Heap element abstract data type */

typedef edge_t heap_elem_t;

#define heap_elem_lt edge_lt
#define heap_elem_lte edge_lte
#define heap_elem_copy edge_copy
#define heap_elem_destroy edge_destroy
#define heap_elem_dump edge_dump


/* Heap abstract data type */

typedef struct _heap_t *heap_t;
/* A (min-)heap (http://en.wikipedia.org/wiki/Heap_%28data_structure%29).
```

```c
 *
 * A heap is a specialized tree-based data structure that satisfies the heap
 * property: if B is a child node of A, then A <= B.
 *
 */

heap_t heap_empty(unsigned int max_size);
/*
 * Return a newly created heap that can hold up to 'max_size' elements.
 *
 * The resources allocated for the heap returned from this call must be freed
 * by calling 'heap_destroy' once the result is no longer needed.
 *
 * POST: the result is not NULL and heap_is_empty(result) holds.
 *
 */

heap_t heap_destroy(heap_t heap);
/*
 * Free all the resources used by 'heap'.
 *
 * PRE: 'heap' must be not NULL, and must have been returned by a previous
 * call to 'heap_empty'.
 */

heap_t heap_insert(heap_t heap, heap_elem_t elem);
/*
 * Insert the element 'elem' in the heap 'heap'.
 * The heap owns 'elem' after this, consider passing a copy if you
 * plan to use 'elem' somewhere else.
 *
 * PRE: 'heap' must be not NULL and the heap should not be full.
 *
 * POST: the size of the resulting heap is the same as the size of 'heap'
 * plus one, and it as the same elements as the given 'heap' plus 'elem'.
 */

heap_t heap_remove_minimum(heap_t heap);
/*
 * Remove and destroy the minumum value in the heap 'heap'.
 *
 * PRE: 'heap' must be not NULL and 'heap' can not be empty.
 *
 * POST: the size of the resulting heap is the same as the size of 'heap'
 * minus one, and it as the same elements as the given 'heap' but the minimum.
 */

heap_elem_t heap_minimum(heap_t heap);
/*
 * Return a copy of the minimum element of the heap 'heap'.
 *
 * PRE: 'heap' must be not NULL and 'heap' can not be empty.
 *
 * POST: the result is the minimum value of the heap 'heap'.
```

```
 */

bool heap_is_empty(heap_t heap);
/*
 * Return whether 'heap' is an empty heap.
 *
 * PRE: 'heap' must not be NULL.
 */

bool heap_is_full(heap_t heap);
/*
 * Return whether 'heap' is full.
 *
 * PRE: 'heap' must not be NULL.
 */

unsigned int heap_size(heap_t heap);
/*
 * Return the size of the heap 'heap'.
 *
 * PRE: 'heap' must not be NULL.
 */

void heap_dump(heap_t heap, FILE * fd);
/*
 * Dump the heap 'heap' to the given file descriptor.
 *
 * PRE: 'heap' must not be NULL, and 'fd' must be a valid file descriptor.
 */

#endif
```

3.2. Priority Queue

Se implementa usando el heap del punto anterior. Permitirá ir obteniendo las aristas priorizándolas de acuerdo a su peso (aristas de menor peso tienen más prioridad). A continuación la interfaz a implementar:

```
#ifndef _PRIORITY_QUEUE_H
#define _PRIORITY_QUEUE_H

#include <stdbool.h>

#include "graph.h"
#include "heap.h"


typedef heap_t priority_queue_t;
/* The priority_queue (http://en.wikipedia.org/wiki/Priority_queue).
 *
 * A priority_queue is an abstract data type which is like a regular queue
 * or stack data structure, but additionally, each element is associated
 * with a "priority".
 */
```

```c
typedef edge_t priority_queue_elem_t;
/* The type of the elements to store in a priority_queue */

priority_queue_t priority_queue_empty(unsigned int max_size);
/*
 * Return a new empty priority queue.
 * It can hold up to max_size elems.
 *
 * The caller must call priority_queue_destroy when done using the resulting
 * queue, so the resources allocated by this call are freed.
 *
 * POST: the result is a well-defined priority queue.
 */

priority_queue_t priority_queue_destroy(priority_queue_t pq);
/*
 * Free the resources allocated for the given priority_queue,
 * and set it to NULL.
 *
 * PRE: 'pq' must be a well-defined priority queue.
 */

priority_queue_t priority_queue_enqueue(priority_queue_t pq,
                                        priority_queue_elem_t elem);
/*
 * Enqueue the given element, according to its priority.
 * The queue owns 'elem' after this, consider passing a copy if you
 * plan to use 'elem' somewhere else.
 *
 * PRE: 'pq' must a well-defined priority queue, 'elem' must also be a
 * well-defined priority queue element. The given queue should not be full.
 *
 * POST: the size of the resulting queue is the same as the size of 'pq'
 * plus one, and it as the same elements as the given 'pq' plus 'elem'.
 */

priority_queue_elem_t priority_queue_first(priority_queue_t pq);
/*
 * Return a copy of the minimum element (according to the priority of the
 * elements) of the queue 'pq'.
 * This means that the caller owns the resources used by the result, and must
 * destroy it once is no longer used.
 *
 * PRE: 'pq' must be a well-defined priority queue and can not be empty.
 *
 * POST: the result is the minimum value of the queue 'pq'.
 */

priority_queue_t priority_queue_dequeue(priority_queue_t pq);
/*
 * Remove and destroy the minimum value in the queue 'pq'.
 *
 * PRE: 'pq' must be a well-defined priority queue and can not be empty.
```

```
 *
 * POST: the size of the resulting queue is the same as the size of 'pq'
 * minus one, and it as the same elements as the given 'pq' but the minimum.
 */

bool priority_queue_is_empty(priority_queue_t pq);
/*
 * Return whether 'pq' is an empty queue.
 *
 * PRE: 'pq' must be a well-defined priority queue.
 */

bool priority_queue_is_full(priority_queue_t pq);
/*
 * Return whether 'pq' is full.
 *
 * PRE: 'pq' must be a well-defined priority queue.
 */


unsigned int priority_queue_size(priority_queue_t pq);
/*
 * Return the current size of the priority queue 'pq'.
 *
 * PRE: 'pq' must be a well-defined priority queue.
 */

void priority_queue_dump(priority_queue_t pq, FILE * fd);
/*
 * Dump the priority queue 'pq' to the given file descriptor.
 *
 * PRE: 'pq' must be a well-defined priority queue, and 'fd' must be a
 * valid file descriptor.
 */

#endif
```

3.3. Union Find

Este TAD 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 tiene la menor complejidad es *Union Find* con *compresión de caminos*, que es la que se debe hacer para garantizar el orden del algoritmo. A continuación la interfaz a implementar:

```
#ifndef _UNION_FIND_H
#define _UNION_FIND_H


typedef unsigned int union_find_elem_t;
typedef unsigned int union_find_class_t;
typedef struct _union_find_t *union_find_t;
```

```c
union_find_t union_find_create(unsigned int max_size);
/*
 * Return a new disjoint set. It can hold up to max_size initial classes.
 *
 * The resources allocated for the set returned from this call must be freed
 * by calling 'union_find_destroy' once the result is no longer needed.
 *
 * POST: the result is not NULL.
 */

union_find_t union_find_destroy(union_find_t uf);
/*
 * Free all the resources used by 'uf'.
 *
 * PRE: 'uf' must be not NULL.
 */

union_find_t union_find_push(union_find_t uf, union_find_elem_t elem);
/*
 * Update the set 'uf' by enabling the initial class for 'elem'.
 *
 * Is worth noting that this method has to be idempotent, in the sense that
 * if is called more than one time with an element already in it, nothing
 * should happen (ie, the count of classes should not change).
 *
 * PRE: 'uf' must be not NULL, and 'elem' must be a well-defined union find
 * element.
 *
 * POST: the result is not NULL, and is equal to 'uf' except that is updated
 * to hold a new initial class for 'elem'.
 */

union_find_t union_find_union(union_find_t uf,
                              union_find_class_t class1,
                              union_find_class_t class2);
/*
 * Update the set 'uf' by merging tha classes 'class1' with 'class2'.
 *
 * PRE: 'uf' must be not NULL, and 'class1' and 'class2' should be
 * valid (existing) classes for 'uf'.
 *
 * POST: the result is not NULL, and is equal to 'uf' except that the given
 * classes were merged into one (thus the count of classes of the result is
 * the same as the one from 'uf' minus one).
 */

union_find_class_t union_find_find(union_find_t uf, union_find_elem_t elem);
/*
 * Return the class for the given element 'elem' in the set 'uf'.
 *
 * PRE: 'uf' must not be NULL, and 'elem' must be a valid set element that has
 * been previously pushed to 'uf'.
 */
```

```
unsigned int union_find_class_count(union_find_t uf);
/*
 * Return the amount of classes that the given set 'uf' has.
 *
 * PRE: 'uf' must not be NULL.
 */

#endif
```

3.4. Stack

Necesario para ir acumulando aristas dentro del algoritmo principal (es decir, será usado por el módulo que provee un `main`). Se utilizan dos instancias, una para las aristas que forman parte del árbol generador de costo mínimo, y otra para las aristas que quedan afuera. A continuación la interfaz a implementar:

```
#ifndef _STACK_H
#define _STACK_H

#include <stdbool.h>

#include "graph.h"


/* Stack element abstract data type */

typedef edge_t stack_elem_t;

#define stack_elem_copy edge_copy
#define stack_elem_destroy edge_destroy


/* Stack abstract data type */

typedef struct _node_t *stack_t;

stack_t stack_empty(void);
/*
 * Return an empty,  newly created stack.
 *
 * The resources allocated for the stack returned from this call must be freed
 * by calling 'stack_destroy' once the result is no longer needed.
 *
 * POST: the result is a well-defined stack and stack_is_empty(result) holds.
 *
 */

stack_t stack_destroy(stack_t stack);
/*
 * Free all the resources used by 'stack'.
 *
 * PRE: the given 'stack' must be a well-defined stack.
 */
```

```c
stack_t stack_push(stack_t stack, stack_elem_t elem);
/*
 * Push 'elem' to the top of the given 'stack'.
 * The stack owns 'elem' after this, consider passing a copy if you
 * plan to use 'elem' somewhere else.
 *
 * PRE: 'elem' must be a well-defined stack element.
 *
 * POST: the resulting stack will have the same elements as 'stack' plus 'elem'
 * in the top of the stack.
 */

stack_t stack_pop(stack_t stack);
/*
 * Pop and destroy the element at the top of the given 'stack'.
 *
 * PRE: 'stack' must not be empty.
 *
 * POST: the resulting stack will have the same elements as 'stack' minus the
 * element in the top of the stack.
 */

stack_elem_t stack_top(stack_t stack);
/*
 * Return a copy of the element at the top of the given 'stack'.
 *
 * PRE: 'stack' must not be empty.
 *
 * POST: the given 'stack' does not change, and the resulting element is a
 * valid stack element and is the one at the top of the stack.
 */

bool stack_is_empty(stack_t stack);
/*
 * Return whether the given 'stack' is empty or not.
 */

#endif
```

4. Formato de los archivos que describen grafos

4.1. 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:

```c
graph_t read_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:

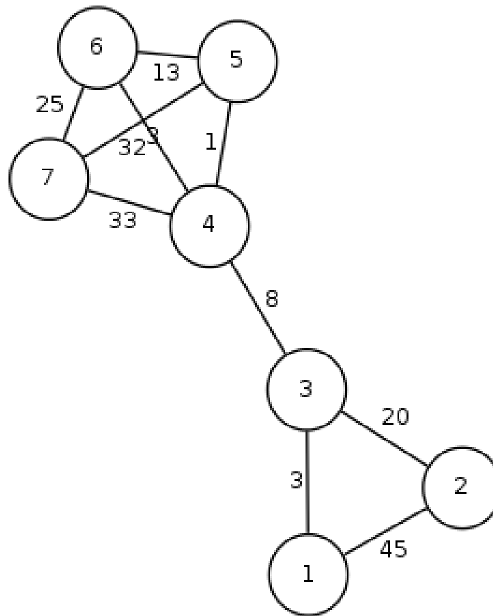

Figura 2: Grafo ejemplo

```
# 7 10

graph G {
  edge [fontsize=10];
  node [fontsize=10,shape=circle,width = "0.10000",height = "0.100000"];
  1 -- 2 [label=45];
  1 -- 3 [label=3];
  3 -- 4 [label=8];
  3 -- 2 [label=20];
  5 -- 6 [label=13];
  5 -- 4 [label=1];
  6 -- 4 [label=32];
  7 -- 4 [label=33];
  7 -- 5 [label=3];
  7 -- 6 [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.

Dependiendo de la implementación del conjunto de conjunto de vértices (union-find) quizás convenga utilizar numeros de vértices que formen un intervalo del 1 a la cantidad de aristas para simplificar la implementación.

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. Para visualizarlo se puede correr:

```
$ xdg-open output.png
```

Ver más detalles del archivo de entrada en la interfaz `graph.h`.

4.2. 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:

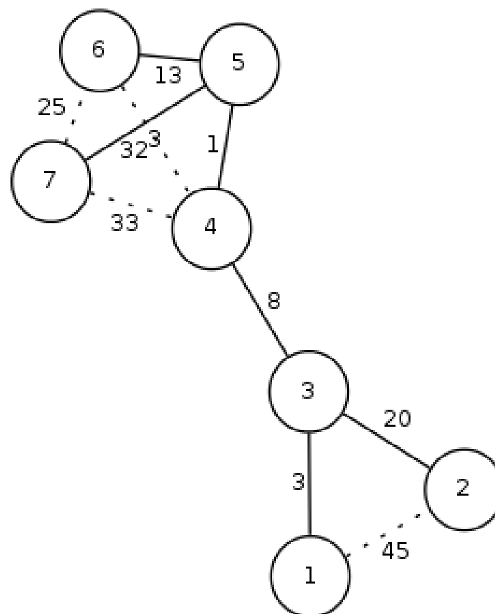

Figura 3: Grafo de salida

que se codifica en un archivo de la siguiente forma:

```
# 7 10

graph G {
  1 -- 2 [label=45,style=dotted];
  1 -- 3 [label=3];
  3 -- 4 [label=8];
  3 -- 2 [label=20];
  5 -- 6 [label=13];
  5 -- 4 [label=1];
  6 -- 4 [label=32,style=dotted];
  7 -- 4 [label=33,style=dotted];
  7 -- 5 [label=3];
  7 -- 6 [label=25,style=dotted];
}
```

Para lograr lo de arriba, ello el TAD edge_t provee una función para indicar que una arista es secundaria; las aristas secundarias, al momento de graficar el grafo, se representan por una línea de puntos (más específicamente, a estas aristas se les define la propiedad $style$ al hacer el dump). De esta manera, las líneas solidas representan las aristas del árbol generador.

4.3. Ejecutar Kruskal pasando el grafo por `stdin`

El pseudocódigo del Kruskal (algoritmo 1) dice que hay que leer el grafo de la entrada estándar, y almacenarlo en una instancia de grafo. Para ello, como se dijo en la sección anterior, deben usar la función:

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

Como se vio en los proyectos anteriores, el tipo `FILE *`, en C, representa un archivo abierto. Y también se vio que `stdout` y `stdin` son archivos abiertos que pueden usarse como `FILE *`. Por ende, para cargar un grafo de la entrada estándar, basta con llamar:

```
graph_t graph = NULL;
graph = read_graph_from_file(stdin);
```

Entonces, una vez que uno tiene compilado el ejecutable `kruskal`, se puede pasar el contenido de un archivo en disco, por entrada estándar, al algoritmo para que 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
$ xdg-open output.png
```

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

4.4. Recomendaciones, antes de empezar a escribir código

- Revisar y entender la interfaz provista por `graph.h`.
- Entender el objetivo y funcionamiento del algoritmo de Kruskal.
- 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.

5. Puntos $\star$

Recomendación usual: no empezar con los puntos estrella hasta que no se tengan absolutamente resueltos los puntos obligatorios, incluyendo la corroboración de ausencia de *memory leaks* y de accesos inválidos a memoria.

1. Cantidad de componente conexas (nivel de dificultad: bajo)

Modificar el algoritmo y los TAD's para que en el caso que el grafo de entrada no sea conexo, se vuelque en la salida la cantidad de componentes conexas.

2. Mejora al Union Find (nivel de dificultad: medio)

Al implementar el *Union Find* hay que utilizar la compresión de caminos. Otra técnica que aumenta la eficiencia es mantener cierto balanceo de la cantidad de los subárboles de cada raíz. Esto se logra, cuando se hacen las uniones, insertando el árbol con menos nodos en el que tiene más.

Se puede implementar esto en orden constante y sin aumentar la cantidad de espacio usado en memoria? Si se puede, hágalo.

3. Nodos del grafo con labels no numéricos ni consecutivos (nivel de dificultad: alto)

Hacer las modificaciones necesarias a los TADs para que en el archivo de entrada los vértices puedan tener nombres arbitrarios.

Si se implementó el conjunto de conjunto de vértices con un arreglo, modificarlo para que en el archivo de entrada los vértices puedan no formar un intervalo entre 1 y la cantidad de vértices (o sea que se puedan saltear números) pero sin desperdiciar memoria. Hacerlo también para que los vértices esten identificados con nombres arbitrarios, por ejemplo con cadenas de texto.

6. Entrega y evaluación

- Fecha de entrega del código:
  - Desde el Lunes 9 de Junio a las 00:00 hrs hasta las 23:59 hrs del mismo día.
  - Obligatorio hacerlo a través de Jaime, habrá una tarea específica para subir el código (separada de la tarea para probar los proyectos). Los archivos a entregar son:

    ```
    heap.c
    heap.h
    priority_queue.c
    priority_queue.h
    stack.c
    stack.h
    union_find.c
    union_find.h
    main.c
    Makefile
    ```
- El parcialito se rinde el Martes 10 de Junio a las 14. Llegar a horario pero no ocupar las máquinas del lab. No hace falta traer ningún archivo ni pendrive.
- El programa resultante **no** debe tener *memory leaks* **ni** accesos (read o write) inválidos a la memoria.