Proyecto 4
Algoritmo Kruskal

Algoritmos y Estructuras de Datos II

4 de junio de 2009

Este proyecto tiene como finalidad implementar el *algoritmo de Kruskal* para encontrar *árboles generadores de costo mínimo*.

Para hacer el algoritmo se tendrán que implementar TAD's indicados utilizando las técnicas para construirlos en C vistas en los proyectos anteriores. Esto quiere decir que se deberá construirlos de forma tal que sean lo más paramétricos, modulares e independientes de su implementación que el lenguaje C lo permita.

En este enunciado se mencionarán una serie de TAD's junto con el conjunto de funciones que deben proveer cada uno. Para aprobar el proyecto se deberán implementar todas esta funciones como mínimo.

Como se vio en el teórico el algoritmo Kruskal tiene orden $\mathcal{O}(e.log(e))$ con e la cantidad de aristas del grafo. Por lo cual se deberán construir las implementaciones de los TAD's que garanticen esta complejidad.

Índice

1. Algoritmo Kruskal **1**

2. TAD's **2**
- 2.1. Cola de prioridades . . . 2
- 2.2. Conjunto de conjuntos de vertices . . . 4
- 2.3. Cinta de lectura de aristas . . . 5
  - 2.3.1. Formato del archivo de entrada . . . 5
- 2.4. Cinta de escritura de aristas . . . 7
  - 2.4.1. Formato del archivo de salida . . . 7

3. Recomendaciones generales **8**

4. Puntos estrella **9**

1. Algoritmo Kruskal

Para implementar el algoritmo se utilizarán los siguientes TAD's:

- Una cola de prioridades que contiene las aristas del grafo.

- Un conjunto de conjuntos de vértices. 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.
- El tipo conjunto que devuelve la búsqueda mencionada en el ítem anterior.
- Una cinta de lectura de aristas para poder leer el grafo de un archivo.
- Una cinta de escritura de aristas para poder escribir a un archivo el resultado del algoritmo.
- Los tipos nodo y arista. Este último debe contener el peso de la misma.
- Dos stacks de aristas para guardar en memoria los resultados intermedios.

Con estos TAD's se puede construir el algoritmo Kruskal como se muestra en **Algoritmo 1** (pag.3). En este algoritmo se pusieron nombres esquemáticos a las funciones de los TAD's para facilitar su comprensión. Las signaturas de estos TAD's tendrán otros nombres.

2. TAD's

2.1. Cola de prioridades

La cola de prioridades se utiliza en el algoritmo para obtener la mínima arista del grafo. También se usa para almacenarlas.

Las operaciones que hay que implementar (o sea la signatura en C que hay que hacer) son:

```
colap
colap_create(const size_t max); /* constructor */
                  /* max es la máxima cantidad de elementos */

void
colap_encolar(colap c, const tcpalpha elem); /* encolar */

tcpalpha
colap_primero(const colap c); /* primero en la cola */

void
colap_pop(colap c); /* saca primero */

bool
colap_vacia(const colap c); /* cola vacia? */

bool
colap_llena(const colap c); /* cola llena? */

tcpalpha
colap_saca(colap c); /* saca y obtiene
                        cualquier elemento */
colap
colap_destroy(colap c); /* destructor */
```

Algoritmo 1 Algoritmo Kruskal (con TAD's)

```
⟦ Var cp : cola_prioridad; ccv : conjunto_de_conjuntos_de_vertices
      cr : cinta_lectura_arista; cw : cinta_escritura_arista
      s1, s2 : stack; e : arista
      cv1, cv2 : conjunto_de_vertices
  ⋮
  construyo todos los TAD's
  ⋮
  do ¬esfin(cr) →
    e := elecorr(cr)
    cp := encolo(cp, e)
    ccv := agrego_singulete(ccv, vertice1(e)); ccv := agrego_singulete(ccv, vertice2(e))
    avanzo(cr)
  od

  do cantidad_conjuntos(ccv) > 1 ∧ ¬vacia(cp) →
    e := primero(cp); cp := saco_primero(cp)
    cv1 := busco_conjunto(ccv, vertice1(e)); cv2 := busco_conjunto(ccv, vertice2(e))
    if cv1 ≠ cv2 →
      ccv := union(ccv, cv1, cv2)
      s1 := push(s1, e)
    □ cv1 = cv2 →
      s2 := push(s2, e)
    fi
  od

  do ¬vacia(s1) →
    e := top(s1); s1 := pop(s1)
    escribo_arbol_min(cw, e)
    e := destruyo(e)
  od
  do ¬vacia(s2) →
    e := top(s2); s2 := pop(s2)
    escribo_grafo_restante(cw, e)
    e := destruyo(e)
  od
  do ¬vacia(cp) →
    (e, cp) := saco_cualquiera(cp)
    escribo_grafo_restante(cw, e)
    e := destruyo(e)
  od
  ⋮
  destruyo todos los TAD's
  ⋮
⟧
```

donde `tcpalpha` es tipo de los elementos que se guardan en la cola (en nuestro caso aristas).

El argumento del constructor es el tamaño máximo de la cola.

La función `colap_saca` obtiene y elimina algún elemento de la cola (no necesariamente el primero). En el **Algoritmo 1** (pag.3) esta nombrada como `saco_cualquiera`.

Además hay que implementar una función sobre el tipo `tcpalpha` que me devuelva la prioridad que tiene (en nuestro caso el peso de la arista):

```
tcpprior
alfa_prior(const tcpalpha elem);
```

Ver que esta función devuelve la prioridad como un tipo nuevo `tcpprior`. Sobre este tipo hay que definir un orden que se debe implementar en la siguiente función:

```
bool
prior_ord(const tcpprior x, const tcpprior y);
```

Esto permite utilizar la cola de forma más o menos paramétrica. Además redefiniendo la última función puedo por ejemplo implementar colas descendentes o ascendentes.

Estas últimas funciones son de uso en la implementación de la cola, no son para el que va a utilizar el TAD (no están en el **Algoritmo 1**). Ver en que archivos hay que declararlas y escribirlas.

Una implementación de cola de prioridades que es muy eficiente es con un *heap*. Implementar la cola con esta estructura.

2.2. Conjunto de conjuntos de vertices

Según el **Algoritmo 1** (pag.3) las operaciones que debe realizar este TAD de forma eficiente son 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*.

La signatura que hay que hacer para este TAD es:

```
union_find
uf_create(const size_t max); /* constructor */

void
uf_add_singulete(union_find uf, const tufalpha el); /* agrega singulete */

tufset
uf_find(union_find uf, const tufalpha el); /* buzca conjunto a partir
                                      de un elemento que le pertenece */
void
uf_union(union_find uf, const tufset s1, const tufset s2);/* union
                                                  de los conjuntos */

bool
uf_oneset(const union_find uf); /* hay solo un conjunto? */

union_find
uf_destroy(union_find uf); /* destructor */
```

El constructor tiene como parámetro la máxima cantidad de elementos que pueden contener los conjuntos (igual que con la cola de prioridades).

La función `uf_oneset` devuelve `TRUE` si el conjunto tiene un solo conjunto.

2.3. Cinta de lectura de aristas

La cinta de lectura de aristas debe proveer las siguientes funciones:

```
cintrar
cra_create(void); /* constructor */

void
cra_arr(cintrar cra); /* arrancar */

void
cra_av(cintrar cra); /* avanzar */

tarista
cra_elec(const cintrar cra); /* elemento corriente */

bool
cra_fin(const cintrar cra); /* fin cinta */

size_t
cra_cantvert(const cintrar cra); /* devuelve la cantidad de vertices */

size_t
cra_cantaris(const cintrar cra); /* devuelve la cantidad de aristas */

cintrar
cra_destroy(cintrar cra); /* destructor */
```

Esta es una cinta de lectura como la vista en otros proyectos (con arrancar, avanzar, etc.). Se agregan las funciones `cra_cantvert` y `cra_cantaris` que devuelven la cantidad de vertices y de aristas totales respectivamente. Estas magnitudes sirven de parametros de los constructores de la cola de prioridades y del conjunto de conjunto de vértices.

Esta cinta debe leer las aristas desde la entrada estándar (`stdin`). Con ello el programa se podrá ejecutar de la siguiente forma:

```
[lopez@hal lopez]# ./kruskal < grafo.dot
```

Donde `grafo.dot` será un archivo de texto con el formato que se declarará a continuación.

2.3.1. Formato del archivo de entrada

El archivo 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>
```

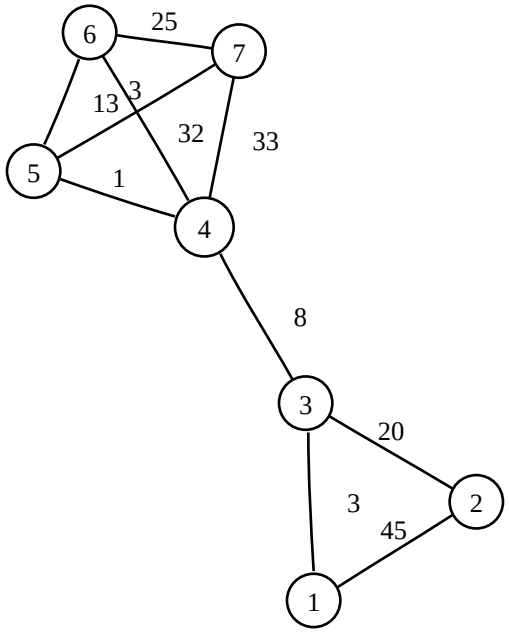

Figura 1: Grafo ejemplo

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 `{`

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

Seguidamente irán una lista de aristas separadas por el símbolo `;`, y al final se cierra la definición con `}`. Por el ejemplo, el grafo de la figura 1 (pag. 6) se puede escribir como:

```
# 7 10
graph G{
  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 que rodean las palabras "`--`" 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 quizás convenga utilizar numeros de vértices que formen un intervalo del 0 a la cantidad de aristas menos uno para reducir la memoria utilizada.

Este formato sirve como entrada al programa `neato` que dibuja grafos. Así, por ejemplo, si el archivo que define el grafo se llama `G.dot`, ejecutando el comando:

```
[lopez@hal lopez]# neato -Tps -o G.ps G.dot
```

se creará un archivo postscript llamado `G.ps` con el dibujo del grafo.

2.4. Cinta de escritura de aristas

La cinta de escritura de aristas debe proveer las siguientes funciones:

```
cintwar
cwa_create(void); /* constructor */

void
cwa_arr(cintwar cwa); /* arrancar */

void
cwa_insarbol(cintwar cwa, tarista ar); /* escribe una arista
                                          del arbol generador */
void
cw_ainsrest(cintwar cwa, tarista ar); /* escribe una arista
                                         del grafo restante */
cintwar
cwa_destroy(cintwar cwa); /* destructor */
```

Esta cinta debe escribir las aristas a la salida estándar (`stdout`). Con ello el programa se podrá ejecutar de la siguiente forma:

```
[lopez@hal lopez]# ./kruskal < grafoin.dot > grafoout.dot
```

Donde `grafoin.dot` es un archivo que define el grafo de entrada (sección 2.3, pag. (pag. 5)) y `grafoout.dot` será un archivo de texto con el formato que se declarará a continuación.

2.4.1. Formato del archivo de salida

El formato de salida será muy similar al de entrada. En este caso no será necesario agregar la línea con la cantidad de vértices y aristas.

Lo que si se tendrá que tener en cuenta es, alguna forma de indicar las aristas que pertenecen al árbol generador de las que no. Una forma de hacerlo es agregando al peso (`[label=<w>]`) algo que indique el estilo de linea con que se dibujará el grafo, con por ejemplo `[label=<w>, style=dotted]`. De esta forma el grafo de la figura 2 (pag. 8) se podrá escribir como:

```
graph G {
  1--2 [label=45,style=dotted];
  1 -- 3 [label=3];
```

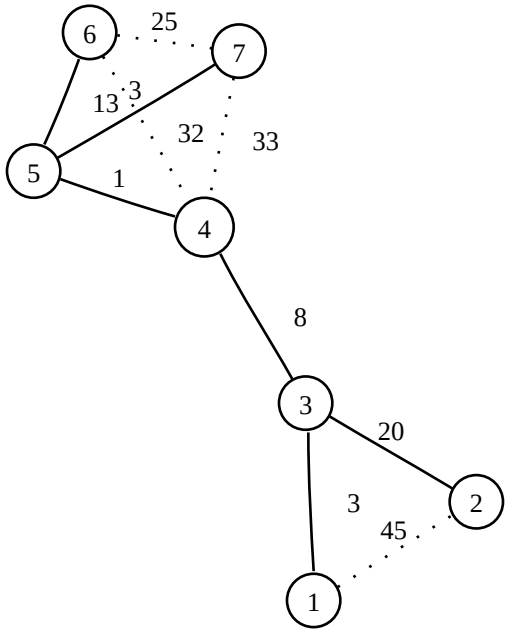

Figura 2: Grafo de salida

```
  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];
}
```

Así, las líneas solidas representan las aristas del árbol generador.

Con este formato también se puede visualizar el grafo resultado con el comando `neato`. Si se desea se pueden agregar otras opciones para dibujar el grafo (ver man page).

3. Recomendaciones generales

1. Construir los TAD's lo mas paramétricos, modulares e independientes de su implementación posible, utilizando las técnicas vistas en los proyectos anteriores.

2. Si se decide cambiar alguna implementación de los TAD's, tener en cuenta que el algoritmo debe tener complejidad menor o igual a $\mathcal{O}(e.log(e))$

3. No usar ninguna variable global.

4. Liberar toda la memoria asignada (con las funciones "`destroy`").

5. Recordar que todos los `.h` deben tener su cerradura `ifndef`.

6. Utilizar `Makefile`.

7. Utilizar todos los parámetros de *gcc* para detectar posibles errores a tiempo de compilación.

8. Para obtener nota B el programa debe funcionar sin errores y debe cumplir con todos los puntos anteriores.

4. Puntos estrella

Los siguiente son puntos opcionales que se pueden hacer en el proyecto y suben la nota final.

1. 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 en memoria? Si se puede hágalo.

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

3. Modificar los TAD's para que en el archivo de entrada los vértices puedan tener nombres arbitrarios.

4. Este es un punto super estrella!!: ¿Hay algún algoritmo que sea mejor que el de Kruskal? Si hay impleméntelo y compárelo.