Analista Universitario en Sistemas - 2023

TALLER DE PROGRAMACIÓN II

TRABAJO FINAL

KRUSKAL

ALUMNO:
Ezequiel Spolli

INDICE

OBJETIVO:. . . . . Pág. 3

ELEMENTOS DE C: . . . . . Pág. 4

FUNCIONAMIENTO DEL PROGRAMA: . . . . . Pág. 4

- MAIN . . . . . Pág. 4
- FUNCIONES. . . . . .Pág. 5

CONJUNTO_CE + EJEMPLO: . . . . . Pág. 7

EJECUCIÓN. . . . . .Pág. 11

MEJORAS A FUTURO. . . . . .Pág. 12

OBJETIVO:

El algoritmo de Kruskal es un algoritmo utilizado para obtener como resultado árboles abarcadores de costo mínimo.

Este algoritmo tiene como idea la siguiente: el árbol, para que sea mínimo, debe estar formado por los lados menos costosos por lo que, se van agregando lados desde los más económicos hasta los más costosos mientras no formen circuito hasta llegar a *n-1* (siendo *n* la cantidad de vértices del grafo).

El objetivo del ejercicio es desarrollar el algoritmo de kruskal en el lenguaje de programación C planteando la complejidad y estructuras necesarias, y teniendo en cuenta la sugerencia de los siguientes prototipos y operaciones:

```
typedef int tipo_nombre ;
typedef int tipo_elemento ;
typedef int vertice ;

#define VERTICES 5

typedef struct _ARISTA {
      vertice u ;
      vertice v ;
      int costo ;
 } arista;

 typedef struct _RAMA {
      arista a;
      rama * sig ;
} rama;
```

```
typedef struct _ENCABEZADO {
      int cuenta ;
      int primer_elemento ;
} encabezado ;

typedef struct _NOMBRE {
      tipo_nombre nombre_conjunto ;
      int siguiente_elemento ;
} nombre ;

typedef struct _CONJUNTO_CE {
      nombre nombres [ VERTICES ];
      encabezado encabezamientos_conjunto [ VERTICES ];
} conjunto_CE ;
```

```
void inicial (tipo_nombre , tipo_elemento , conjunto_CE *) ;

void combina (tipo_nombre , tipo_nombre , conjunto_CE *) ;

tipo_nombre encuentra (int , conjunto_CE *) ;

void kruskal (rama *) ;
```

Conjunto_CE (Conjunto con operaciones COMBINA y ENCUENTRA): empieza con una colección de objetos (vértices), cada uno de ellos contenido en un conjunto, luego se combinan los conjuntos bajo algún orden dado, y de vez en cuando es necesario preguntar en que conjunto se encuentra algún elemento en particular.

Las operaciones que se aplican son:

1. COMBINA(A, B, C) para combinar los componentes A y B en C y llamar al resultado A o B arbitrariamente.

2. ENCUENTRA (v, C) para devolver el nombre del componente de C, del cual el vértice v es miembro. Esta operación se utilizará para determinar si los dos vértices de una arista se encuentran en el mismo o en distintos componentes.

3. INICIAL (A, v, C) para que A sea el nombre de un componente que pertenece a C y que inicialmente contiene el vértice v.

Todas estas operaciones forman parte de una estructura de datos abstracta que llamaremos COMBINA ENCUENTRA.

ELEMENTOS DE C:

En este ejercicio se emplean los siguientes elementos de programación C:

- Directivas de preprocesador: Macros
- Estructuras iterativas – Bucles (For, While)
- Estructuras condicionales (If - else)
- Variables compuestas – Arrays multidimensionales (Matrices)
- Memoria dinámica y estructuras (TDA)
- Funciones - Recursividad

El programa consiste en 3 archivos

- **kruscal.c** – Programa en C (main)
- **libKruskal.c** – Programa en C (Implementaciones)
- **libKruskal.h** – Librería (Declaraciones)

FUNCIONAMIENTO DEL PROGRAMA:

MAIN

- Define e inicializa variables y parámetros a utilizar.
- Da la bienvenida al programa.
- Solicita a través de teclado el ingreso de los valores relacionados al costo de cada una de las aristas del grafo a consultar siendo “0” en el caso que la arista no exista.
- Almacena los datos obtenidos en una matriz *M_Costos [VERTICES][VERTICES].*
- A través de la función **inserta** genera una cola con prioridad conformada con las aristas ingresadas ordenadas por su costo en forma ascendente.
- Una vez obtenida esta cola con prioridad llamada *arbol*, el programa llama a la función **kruskal** para emplear el algoritmo y así obtener el arbol abarcador de costo mínimo.
- Luego, el programa consulta al jugador si desea ingresar un nuevo grafo a consultar y chequea que la opción ingresada sea correcta.
    - Si la respuesta del usuario es afirmativa el programa vuelve a comenzar solicitando el ingreso de los datos correspondiente al nuevo grafo.

- En caso contrario, de ser negativa la respuesta, el programa finaliza.

FUNCIONES

void **inserta** (int, int, int, rama **);

Solicita la creación de una "nueva rama" con el valor de una arista en su interior a través de la función **obtieneRama.**

Luego inserta esta "nueva rama" dentro de un arbol dado en la posición de orden que corresponde según el costo de la arista en su interior, creando así una cola de prioridad con todas las aristas del grafo ingresado.

rama ***obtieneRama** (int , int , int);

Reserva en la memoria dinámica el espacio correspondiente a un tipo de dato rama (**sizeof** rama) y le asigna los valores correspondiente a la nueva rama solicitada.

rama ***suprimeRama** (rama *)

Al pasarle una "rama anterior" a la que se desea eliminar, esta función elimina y libera el espacio en la memoria dinámica correspondiente a esta rama.

void **muestraRama** (rama *);

Como su nombre lo indica, esta función muestra por pantalla, la cola de prioridad conformada por todas las aristas de un arbol dado.

void **kruskal** (rama*);

- Crea una estructura de tipo **conjunto_CE** y luego la inicializa con los valores correspondientes a través de la función **inicial.**
- Luego recorre la cola con prioridad llamada *arbol* (con todas las aristas ordenadas en su interior) y opera sobre el arbol y el conjunto_CE siguiendo los siguientes pasos:
  - Toma la primera rama con la arista de menor costo en su interior.
  - Consulta en el conjunto_CE por medio de la función **encuentra** si los vértices de la arista en el interior de la rama están en el mismo conjunto.
    - De estar en conjuntos diferentes, combina ambos conjuntos a través de la función **combina**.
    - De estar en conjuntos iguales, el agregado de esta arista al grafo abarcador implicaría la creación de un ciclo por lo cual se elimina esta rama del arbol a través de la función **suprimeRama.**
  - Toma la siguiente rama de *arbol* y vuelve a repetir el proceso hasta llegar al final de la cola *arbol* o hasta conseguir la cantidad de *n-1* aristas en el grafo abarcador.
- Luego de recorrer la cola *arbol:*
  - Si **la cantidad total de aristas es inferior a *n-1*** esto significa que el grafo que se ingreso a consultar en un inicio NO era un GRAFO CONEXO por lo cual no se puede obtener un arbol abarcador.

La función informa por pantalla este resultado, libera el espacio de memoria dinámica correspondiente a la cola *arbol* con la función **liberaArbol** y retorna al programa principal.

- Si **la cantidad total de aristas es igual a *n-1,*** el grafo abarcador de costo mínimo ya fue obtenido, por lo cual, la función elimina y libera el restante de aristas de la cola *arbol* que no forman parte.
  Luego muestra por pantalla el grafo abarcador de costo mínimo obtenido a través de la función **muestraRama**, libera el espacio de memoria dinámica correspondiente a la cola *arbol* con la función **liberaArbol** y retorna al programa principal.

void **inicial** (tipo_nombre, tipo_elemento, conjunto_CE*);

Esta función inicializa el conjunto_CE de la siguiente manera:

Conjunto_CE

INICIO

NOMBRES

	0	1	2	3	4
nombre_connjunto	0	1	2	3	4
siguiente_elemento	-1	-1	-1	-1	-1

ENCABEZAMIENTOS_CONJUNTOS

	0	1	2	3	4
cuenta	1	1	1	1	1
primer_elemento	0	1	2	3	4

Encabezamiento_Conjuntos:

- **cuenta = 1:** Porque cada conjunto está conformado por un solo elemento o vértice.
- **primer_elemento = vértice:** El primer y único elemento que tiene cada conjunto en su interior es el vértice.

Nombres:

- **nombre_conjunto = vertice:** Porque el nombre de cada conjunto coincide con el nombre del primer elemento o vértice del conjunto.
- **Siguiente_elemento = -1:** Al tener cada conjunto un único elemento, los siguientes elementos de cada conjunto son nulos **(-1)**, es decir, no existe un próximo elemento.

tipo_nombre **encuentra** (int, conjunto_CE*);

Dado un vértice y el conjunto_CE, esta función retorna el nombre del conjunto del cual forma parte dicho vértice.

void **combina** (tipo_nombre, tipo_nombre, conjunto_CE*);

Dados los nombre de dos conjuntos distintos, esta función los combina siguiendo los siguientes pasos:

- Consulta de ambos conjuntos la cantidad de vértices que contienen en su interior y determina cual es el conjunto mayor y cual el menor.
- Luego procede a combinar el conjunto menor dentro el conjunto mayor.
  Para realizar esta operación:

- Recorre el array de *nombres_conjuntos* dentro del conjunto_CE, buscando el final del conjunto mayor y una vez encontrado, lo anexa al inicio del conjunto menor.
- Luego, continua recorriendo el array de *nombres_conjuntos*, actualizando los nombres de los elementos del conjunto menor por el nombre del conjunto mayor del cual ahora forman parte.
- Una vez actualizado el array de *nombres_conjuntos*, procede a actualizar el array de *encabezamiento_conjuntos* (también dentro del conjunto_CE) siguiendo los siguientes pasos:
    - Primero actualiza la *cuenta* de elementos del conjunto mayor por la suma total de los elementos de ambos conjuntos recientemente combinados.
    - Y luego actualiza la *cuenta* de elementos del conjunto menor por "0" y el *primer_elemento* del conjunto menor por "-1", ya que ahora este conjunto no posee vértices.

void **muestraCe** (conjunto_CE *);

Como su nombre lo indica, esta función muestra por pantalla el estado actual de la estructura Conjunto_CE al momento de llamarla.

Esta función no es requerida durante el transcurso del programa, pero es una herramienta de gran ayuda e importancia en el trabajo de mantenimiento del programa.

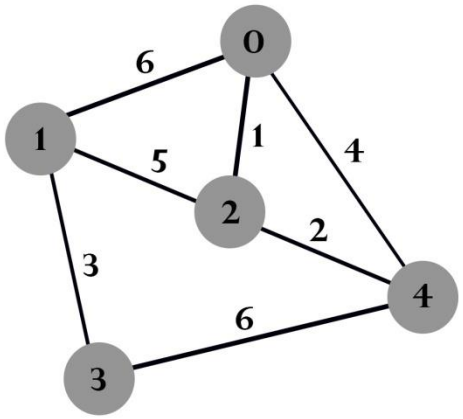
void **liberaArbo**l(rama *);

Dada una cola empleada en memoria dinámica, esta función la recorre liberando y eliminando cada uno de los elementos de la misma, devolviendo al sistema el espacio de memoria dinámica antes reservado y utilizado.

Conjunto_CE + EJEMPLO

A continuación vamos a analizar el funcionamiento de nuestro programa visualizando el recorrido de los estados del conjunto_CE.

Para esto vamos a utilizar el siguiente grafo como ejemplo.



Al principio el conjunto_CE comienza cargando con los valores iníciales.

Conjunto_CE

INICIO

NOMBRES

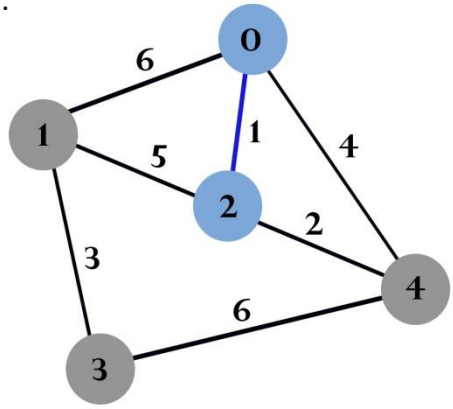
	0	1	2	3	4
nombre_connjunto	0	1	2	3	4
siguiente_elemento	-1	-1	-1	-1	-1

ENCABEZAMIENTOS_CONJUNTOS

	0	1	2	3	4
cuenta	1	1	1	1	1
primer_elemento	0	1	2	3	4

PASO 1:

Tomamos la arista mas económica en todo el grafo dado. En este caso es la arista entre los vértices 0 y 2 de costo 1.



En este caso obsevamos que el vértice 0 está en el conjunto de nombre 0 y el vértice 2 está en el conjunto de nombre 2. Por lo tanto se combinan los grupos 0 y 2.

Conjunto_CE

1 PASO

NOMBRES

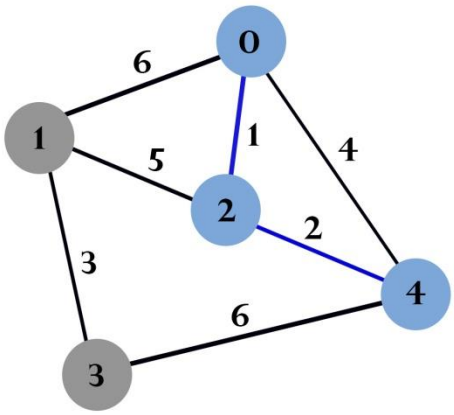
	0	1	2	3	4
nombre_connjunto	2	1	2	3	4
siguiente_elemento	-1	-1	0	-1	-1

ENCABEZAMIENTOS_CONJUNTOS

	0	1	2	3	4
cuenta	0	1	2	1	1
primer_elemento	-1	1	2	3	4

PASO 2:

Tomamos la segunda arista más económica del grafo ingresado. En este caso es la arista entre los vértices 2 y 4 de costo 2.



En esta oportunidad, observamos que el vértice 2 está en el conjunto de nombre 2 y el vértice 4 está en el conjunto de nombre 4. Por lo tanto a través de la función **combina** se unen los grupos 2 y 4 quedando el conjunto_CE de la siguiente manera.

Conjunto_CE

2 PASO

NOMBRES

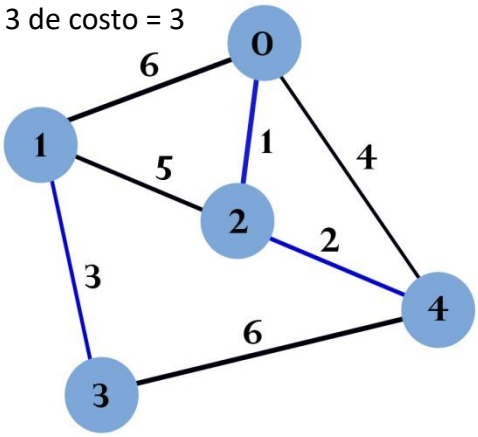
	0	1	2	3	4
nombre_connjunto	2	1	2	3	2
siguiente_elemento	4	-1	0	-1	-1

ENCABEZAMIENTOS_CONJUNTOS

	0	1	2	3	4
cuenta	0	1	3	1	0
primer_elemento	-1	1	2	3	-1

PASO 3:

Nuevamente tomamos la arista más económica entre las restantes. En este caso es la arista entre los vértices 1 y 3 de costo = 3



Ahora observamos que el vértice 1 está en el conjunto de nombre 1 y el vértice 3 está en el conjunto de nombre 3. Por lo tanto combinamos los conjuntos 1 y 3 quedando el conjunto_CE de la siguiente manera.

Conjunto_CE

3 PASO

NOMBRES

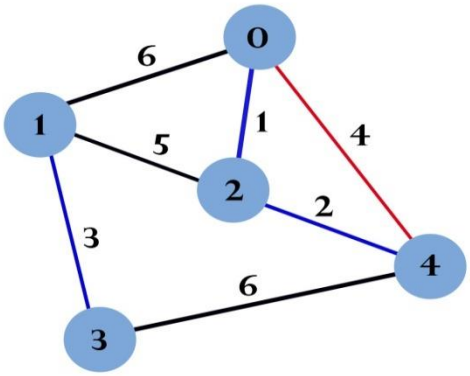
	0	1	2	3	4
nombre_connjunto	2	1	2	1	2
siguiente_elemento	4	3	0	-1	-1

ENCABEZAMIENTOS_CONJUNTOS

	0	1	2	3	4
cuenta	0	2	3	0	0
primer_elemento	-1	1	2	-1	-1

PASO 4:

Buscando la arista más económica entre las restantes encontramos que en este caso es la arista entre los vértices 0 y 4 de costo = 4

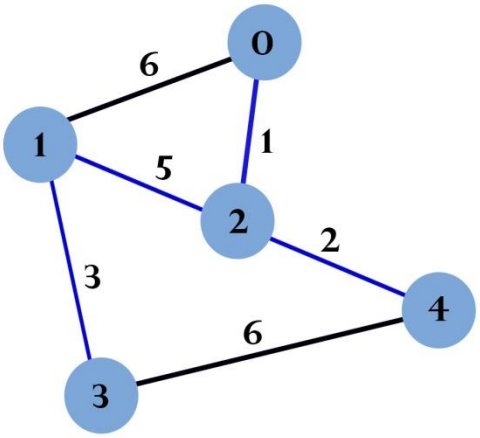


En esta ocasión podemos ver que el vértice 0 está en el conjunto de nombre 2 y el vértice 4 está en el conjunto del mismo nombre. Por lo tanto, si agregáramos esta arista a nuestro grafo abarcador, ésta generaría un ciclo. Razón por la cual, esta arista no forma parte de nuestro grafo abarcador de costo mínimo y debe ser eliminada.

Nuestro conjunto_CE no sufre modificación alguna.

PASO 5:

La siguiente arista más económica entre las restantes es la arista entre los vértices 1 y 2 de costo = 5



Podemos ver que el vértice 1 está en el conjunto de nombre 1 y el vértice 2 a está en el conjunto de nombre 2. Entonces a través de la función **combina** unimos ambos conjuntos y una vez más el conjunto_CE de la siguiente manera.

Conjunto_CE

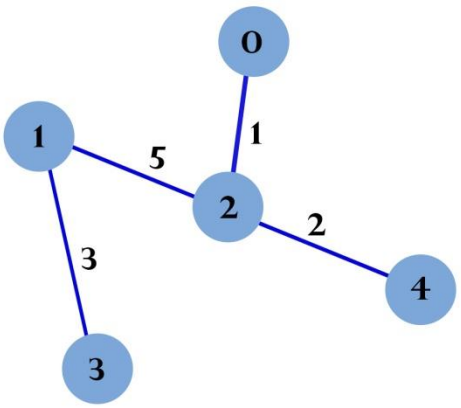
5 PASO

NOMBRES	0	1	2	3	4
nombre_connjunto	2	2	2	2	2
siguiente_elemento	4	3	0	-1	1

ENCABEZAMIENTOS_CONJUNTOS	0	1	2	3	4
cuenta	0	0	5	0	0
primer_elemento	-1	-1	2	-1	-1

Como podemos observar, todos los vértices forman parte de un mismo conjunto y la cantidad de aristas agregadas son equivalentes a *n-1*, por lo cual hemos encontrado nuestro grafo abarcador de costo mínimo y solo resta eliminar las aristas restantes que no forman parte de este grafo.

Nuestro grafo abarcador de costo mínimo quedaría de la siguiente manera:



EJECUCIÓN:

- El programa da la bienvenida, y le solicita al usuario ingresar el valor de costo de cada una de las distintas aristas. En este caso ingresamos el ejemplo utilizado anteriormente.
- Una vez ingresada la información de todas las aristas, el programa nos muestra una lista de todas las aristas que conforman el grafo abarcador de costo mínimo obtenido.

```
oem@smeli-linux:~/Desktop/khruskal$ ./a.out

                  ÁRBOLES ABARCADORES DE COSTO MÍNIMO

Para encontrar el árbol abarcador de costo mínimo de un grafo de 5 vertices:

Ingrese el COSTO DE LA ARISTA entre:

   - Los vertices 0 y 1:  6

   - Los vertices 0 y 2:  1

   - Los vertices 0 y 3:  0

   - Los vertices 0 y 4:  4

   - Los vertices 1 y 2:  5

   - Los vertices 1 y 3:  3

   - Los vertices 1 y 4:  0

   - Los vertices 2 y 3:  0

   - Los vertices 2 y 4:  2

   - Los vertices 3 y 4:  6


El grafo abarcador de MENOR COSTO esta conformado por LAS ARISTAS entre:

   - Los VERTICES 0 y 2, de COSTO 1

   - Los VERTICES 2 y 4, de COSTO 2

   - Los VERTICES 1 y 3, de COSTO 3

   - Los VERTICES 1 y 2, de COSTO 5

¿Desea ingresar un nuevo grafo? ('S'/'N')
```

- Por último, nos consulta por si tenemos la necesidad de ingresar un nuevo grafo a consultar.
- En esta oportunidad, ingresamos un nuevo grafo pero esta vez se trata de un grafo que no es conexo.

```
¿Desea ingresar un nuevo grafo? ('S'/'N')

S

Para encontrar el árbol abarcador de costo mínimo de un grafo de 5 vertices:

Ingrese el COSTO DE LA ARISTA entre:

   - Los vertices 0 y 1:  0

   - Los vertices 0 y 2:  1

   - Los vertices 0 y 3:  0

   - Los vertices 0 y 4:  4

   - Los vertices 1 y 2:  0

   - Los vertices 1 y 3:  3

   - Los vertices 1 y 4:  0

   - Los vertices 2 y 3:  0

   - Los vertices 2 y 4:  2

   - Los vertices 3 y 4:  0


El grafo ingresado NO ES CONEXO

¿Desea ingresar un nuevo grafo? ('S'/'N')

N

FIN de la operación

oem@smeli-linux:~/Desktop/khruskal$
```

- El programa nos advierte que se trata de un grafo que no es conexo y nos vuelve a consultar por si tenemos la necesidad de ingresar un nuevo grafo a consultar. Pero en esta ocasión, ingresamos un “No” como respuesta y el programa termina.

MEJORAS A FUTURO:

Algunas mejoras a futuro podrían ser:

- Al inicio del programa consultar por la cantidad de vértices que tiene el grafo a consultar y que esta cantidad pueda ser variable en las distintas ejecuciones dependiendo de las necesidades del usuario.
- Validar y controlar que los datos sean ingresados de manera adecuada. Por ejemplo que sean números positivos los valores de costo de cada arista.