Trabajo Final de Taller de Programación II

Mesa 16/12/2022

Algoritmo de Kruskal

Documentación

TRABAJO REALIZADO POR MARINA CARRACEDO

El algoritmo de Kruskal es un proceso que permite unir todos los vértices de un grafo formando un árbol, tomando en cuenta el peso de las aristas y cuyo coste total es el mínimo posible, o sea un grafo recubridor mínimo en un grafo conexo y ponderado.

El programa desarrollado busca obtener ese resultado, a partir del ingreso de los costos de las aristas (en caso de ingresar costo 0, se considera que no hay arista conectando esos vértices).

Para esto se tomó en cuenta las siguientes estructuras básicas:

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

typedef struct _RAMA
{
    struct _ARISTA a;
    struct _RAMA *sig;
} rama;

typedef struct Grafo
{
    int cant_ramas;
    rama *cabezas[VERTICES];
    int costoTotal;
    int cant_aristas;
} grafo;
```

En primera instancia, a partir de la cantidad de vértices, se ingresan los costos y se genera una lista de aristas, cuyo nodo llamo rama. Los vértices se declaran como enteros (u y v).

Luego, se recorre esa lista seleccionando el nodo de menor costo, y quitándolo de la lista original para colocarlo en el grafo resultado en caso de que este este vacío o alguno de sus vértices no esté ya cubierto. En caso de que los dos vértices estén ya incluidos en el grafo resultado, se lo manda a una lista llamada papelera.

Cada vez que se agrega una rama, se incrementan los valores de los costos y cantidad de aristas.

Sobre la ubicación de la nueva arista en el grafo:

Caso 1: ninguno de los vértices de la arista está presente en el grafo. Se incrementa el conteo de ramas en el grafo y se inicia una nueva lista, ubicando esa rama como cabeza.

Caso 2: uno de los vértices está incluido y el otro no. Se agrega la nueva rama a la misma lista que tiene un vértice en común.

Caso 3: Los dos vértices están incluidos en la misma lista del grafo resultado. Esa rama es descartada.

Caso 4: Los dos vértices están incluidos en el grafo en diferentes listas. Esa rama va a papelera (lista de ramas)

Una vez recorrida la lista original, y reubicadas todas sus ramas, ya tenemos un grafo recubridor como resultado, pero falta constatar si el grafo resultado es conexo o cuenta con más de una lista (o subgrafos)

En caso de que cuente con más de una lista, recorremos la papelera, buscando la rama con la arista de menor peso y evaluamos la ubicación de sus dos vértices. Si forman parte de la misma lista, la rama es definitivamente eliminada y su memoria liberada. En el caso de que sus vértices se ubiquen en diferentes listas, se toma la lista de menor posición, se le agrega la nueva rama y se agrega a esa lista la lista donde está el otro vértice, unificando la lista, bajando el conteo de ramas del grafo, hasta que el grafo resultado tenga una sola lista de ramas.

En caso de obtener ese resultado (grafo recubridor conexo) el resultado es impreso en un archivo .txt.

En caso de no obtener ese resultado con un solo subgrafo, la consola informará el resultado con sus diferentes listas o subgrafos.

En cualquier caso la papelera también se guarda en un .txt

El programa admite como input la información de grafos vacíos, e inconexos.

Resumen de las funciones utilizadas:

```
void inserta(int, int, int, rama **);
// La uso solamente para el árbol original // Llama a crearRama que genera el nodo

rama *crearRama(int i, int j, int micosto);
// La uso para insertar rama en lista

void insertaRamaEnLista(rama *nuevaRama, rama **arbol);
// La uso para cuando saco la rama del árbol y la pongo en el Kruskal o en la papelera

rama *sacar_min(rama **arbol);
// Recorre todo el árbol y devuelve la arista de costo mínimo para después procesarla

void correr(rama **arbol, grafo *kruskal, rama **papelera);
// Llama a sacar_min mientras haya arbol y si se termina y es necesario saca de
papelera // Llama a procesar
```

```c
void procesar(rama *nuevaRama, rama **arbol, grafo *kruskal, rama **papelera);
// Elimina del viejo y combina en nuevo // Llama a eliminar y a combinar

void eliminarRama(rama *miRama, rama **arbol);
// Quita de la lista sin liberar memoria, hace que anterior->sig apunte a miRama->sig

void combina(rama *miRama, grafo *arbol, rama **papelera);
// Agrega si no hay nada, si hay un solo vértice en común agrega y manda a papelera si están los dos vértices

void buscarEnPapelera(grafo *kruskal, rama **papelera);
// Trae de la papelera los minimos que habian sido descartados porque algún vértice se repetía en caso de que en la primera vuelta haya como resultado un grafo inconexo

int encuentra(int *i, rama **arbol);
// Si encuentra = 1 es true busca un valor en ambos vértices de cada puntero que recorre

int encuentraEnGrafo(int *u, grafo *kruskal);
// Me da 1 por si y 0 por no buscando el vértice en el grafo // Llama a encuentra

int verificoAmbosVerices(int *vertice, rama *puntero);
// Recorre los punteros de la lista verificando si el nro de vértice esta en cualquiera de los dos

int encuentraLugarEnGrafo(int *u, grafo *kruskal);
// Me devuelve la posición de la lista que tiene un vértice en común con el vértice que agrego

int buscarIntMin(int a, int b);
// buscarIntMin y buscarIntMax son para los casos en que traigo de la papelera una arista con dos vértices en listas de diferentes posiciones en el grafo y empalma la lista de menor pos con la arista y la arista con la lista de mayor posición, buscando que cuando termine el proceso, la lista completa este en pos [0]
int buscarIntMax(int a, int b);

void imprimirArbol(rama **arbol);
// Muestra la lista en consola
void imprimirGrafo(grafo migrafo);
// Imprime en consola los subgrafos si el resultado es inconexo

void printTXT(rama **lista, char nombreArchi[9]);
// Imprime la lista en un archivo .txt // Se usa para imprimir papelera y resultado si hay camino Kruskal
```