Heurísticas voraces

17 de enero de 2023

Analisis y Diseño de Algoritmos

Herrera Guadarrama Juan Pablo

0. Heurísticas voraces

En ciencias de la computación, un algoritmo voraz (también conocido como goloso, ávido, devorador o greedy) es una estrategia de búsqueda por la cual se sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima

1. Análisis y diseño

Kruskal: Árbol de recubrimiento mínimo

En seudocódigo:
Método Kruskal (Grafo)

- Se inicializa el árbol de expansión mínima vacío
- Se inicializa una estructura de unión-búsqueda
- Se ordenan las aristas de menor a mayor peso
- Para cada arista a que une 2 vértices (u, v)
    - Si u y v no están en la misma componente
        - Se añade la arista a al árbol de expansión mínima.
    - Se unen las componentes de u y v

  Fin Si
  Fin Para
- Fin Método Kruskal

Después de realizar el análisis del código, para el algoritmo de Kruskal, el orden de complejidad computacional temporal es de **O (a log n).** Siendo n el número de vértices y a el número de aristas del grafo. Este orden de complejidad es el obtenido al realizar la ordenación de las aristas de menor a mayor peso.

2. Código fuente

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cabecera.h"

#define VERTICES 5
/* run this program using the console pauser or add your own getch, system("pause") or
input loop */

	int main(int argc, char *argv[]) {
		int M_Costos[VERTICES][VERTICES];
		grafo kruskal;
		kruskal.cabezas[0] = NULL;
		kruskal.cant_aristas = 0;
		kruskal.cant_ramas = 0;
		kruskal.costoTotal = 0;
		 rama *arbol = NULL;
		rama *papelera = NULL;
		int i, j;

		printf("\n\t Algoritmo de Kruskal \n");
```

```
            printf("\nPara %d nodos, donde el primero es 0 y el ultimo es %d
\n\n",VERTICES, VERTICES-1);
            for (i=0; i < VERTICES / 2; i++)
                {
                    kruskal.cabezas[i] = NULL;
                }

            for (i = 0; i <= VERTICES - 1; i++)
                  for (j = i + 1; j <= VERTICES - 1; j++)
                    {
                        printf("Ingrese costo(peso) entre los vertices %d y %d: ", i,
j);
                        scanf(" %d", &M_Costos[i][j]);
                    }

            for (i = 0; i <= VERTICES - 1; i++) // la mitad inf. de diagonal de matriz
                  for (j = i + 1; j <= VERTICES - 1; j++)
                        if (M_Costos[i][j] != 0)
                              inserta(i, j, M_Costos[i][j], &arbol); // inserto en cola
prior .

            imprimirArbol(&arbol);
            correr(&arbol, &kruskal, &papelera);
            printf("\n IMPRIMO KRUSKAL:\n");

            imprimirGrafo(kruskal); // Imprime todo el grafo de resultado, aun si da
inconexo
      return 0;
      }
```

Funciones.c

```
#include "cabecera.h"
#define VERTICES 5


void inserta(int i, int j, int micosto, rama **arbol) // Agrega solamente desde el
ingreso por teclado y crea el nodo
{
    rama *nuevaRama = crearRama(i, j, micosto);
    if (*arbol == NULL)
    {
        (*arbol) = nuevaRama;
    }
    else if (*arbol)
    {
        rama *puntero;
        puntero = *arbol;
        while (puntero->sig)
        {
            puntero = puntero->sig;
        }
        puntero->sig = nuevaRama;
    }
}

void insertaRamaEnLista(rama *nuevaRama, rama **arbol) //
{
    (*nuevaRama).sig = NULL;
    if (*arbol == NULL)
    {
        (*arbol) = nuevaRama;
    }
    else if (*arbol)
    {
```

```
        rama *puntero;
        puntero = *arbol;
        while (puntero->sig)
        {
            puntero = puntero->sig;
        }
        puntero->sig = nuevaRama;
    }
}

rama *crearRama(int i, int j, int micosto)
{
    rama *nuevaRama = (rama *)malloc(sizeof(rama));
    nuevaRama->a.u = i;
    nuevaRama->a.v = j;
    nuevaRama->a.costo = micosto;
    nuevaRama->sig = NULL;
    return nuevaRama;
}

void imprimirArbol(rama **arbol)
{
    int costoTotal = 0;
    contArista = 0;
    if (*arbol != NULL)
    {
        contArista = 1;
        rama *puntero;
        puntero = *arbol;
        printf("Arista %d tiene vertices u %d y v %d con costo de %d\n", contArista,
puntero->a.u, puntero->a.v, puntero->a.costo);
        costoTotal = costoTotal + puntero->a.costo;
        while (puntero->sig)
        {
            puntero = puntero->sig;
            contArista++;
            costoTotal = costoTotal + puntero->a.costo;
            printf("Arista %d tiene vertices u %d y v %d con costo de %d\n", contArista,
puntero->a.u, puntero->a.v, puntero->a.costo);
        }
        printf("El costo total del arbol es: %d\n", costoTotal);
    }
    else
    {
        printf("Arbol Vacio\n");
    }
}

rama *sacar_min(rama **arbol)
{
    if (arbol)
    {
        rama *ramaMin;
        rama *puntero;
        puntero = *arbol;
        ramaMin = puntero;
        int min = puntero->a.costo;
        while (puntero->sig)
        {
            if (puntero->a.costo < min)
            {
                ramaMin = puntero;
                min = puntero->a.costo;
            }
            puntero = puntero->sig;
        }
```

```
        if (puntero->a.costo < min)
        {
            ramaMin = puntero;
            puntero = puntero->sig;
        }

        return ramaMin;
    }
    else if (!arbol)
    {
        return NULL;
    }
}

void combina(rama *miRama, grafo *arbol, rama **papelera)
{
    miRama->sig = NULL;
    int u = miRama->a.u;
    int v = miRama->a.v;
    int eU = encuentraEnGrafo(&u, arbol);
    int eV = encuentraEnGrafo(&v, arbol);

    if (arbol->cant_aristas == 0)
    {
        int i = arbol->cant_ramas;
        insertaRamaEnLista(miRama, &(arbol->cabezas)[i]);
        arbol->cant_aristas++;
        arbol->costoTotal += miRama->a.costo;
    }

    else if (arbol->cant_aristas != 0)
    {

        if (eU == 0) // Si el primer vertice no esta, inserta
        {
            if (eV == 0)
            {
                arbol->cant_ramas++;
                int i = arbol->cant_ramas;
                insertaRamaEnLista(miRama, &(arbol->cabezas)[i]);
            }
            else if (eV == 1)
            {
                int i = encuentraLugarEnGrafo(&v, arbol);
                insertaRamaEnLista(miRama, &(arbol->cabezas)[i]);
            }
            arbol->cant_aristas++;
            arbol->costoTotal = (arbol->costoTotal) + miRama->a.costo;
        }
        else if (eU == 1)
        {
            if (eV == 0) // Si el segundo vertice no esta, inserta
            {
                int lugarU = encuentraLugarEnGrafo(&u, arbol);
                insertaRamaEnLista(miRama, &(arbol->cabezas)[lugarU]); //// ESTOY ACA
                arbol->cant_aristas++;
                arbol->costoTotal = (arbol->costoTotal) + miRama->a.costo;
            }
            else if (eV == 1)
            {
                int lugarU = encuentraLugarEnGrafo(&u, arbol);
                int lugarV = encuentraLugarEnGrafo(&u, arbol);
                //printf("Ambos vertices estan en el arbol va a papelera\n");
                insertaRamaEnLista(miRama, papelera);
            }
        }
```

```
    }
    EXIT_SUCCESS;
}

void eliminarRama(rama *miRama, rama **arbol)
{
    if (*arbol != NULL)
    {
        rama *anterior;
        if (miRama == (*arbol))
        {
            if (miRama->sig == NULL)
            {
                *arbol = (*arbol)->sig; // Elimino el primero
                *arbol = NULL;
            }
            else if ((miRama->sig != NULL) && ((*arbol)->sig))
            {
                rama *aux = (*miRama).sig;
                (*miRama).sig = NULL;
                *arbol = aux;
            }
        }

        else if (miRama != (*arbol)) // Si no es el primero
        {
            anterior = *arbol; // Ubico el anterior en cima de la lista
            if (miRama == anterior->sig) // Si es el segundo
            {
                rama *aux = (*miRama).sig;
                (*miRama).sig = NULL;
                anterior->sig = aux;
            }
            else if (miRama != anterior->sig)
            { // Si no es el segundo
                while (anterior->sig != NULL)
                {
                    anterior = anterior->sig;
                    if (miRama == anterior->sig) // Si es el siguiente
                    {
                        rama *aux = (*miRama).sig;
                        (*miRama).sig = NULL;
                        anterior->sig = aux;
                        return;
                    }
                    else if (miRama != anterior->sig)
                    {
                        // printf("Termino de buscar\n");
                    }
                }

                if (miRama != anterior->sig)
                {
                    // printf("Termino de buscar\n");
                }
            }
        }
    }
    else
    {
        printf("No hay nada para eliminar\n");
    }
}

void procesar(rama *ramaMin, rama **arbol, grafo *kruskal, rama **papelera)
{
```

```
    eliminarRama(ramaMin, arbol);
    combina(ramaMin, kruskal, papelera);
    EXIT_SUCCESS;
}

void correr(rama **arbol, grafo *kruskal, rama **papelera)
{
    if (*arbol != NULL)
    {
        while (*arbol != NULL)
        {
            rama *ramaMin = sacar_min(arbol);
            procesar(ramaMin, arbol, kruskal, papelera);
        }
        if (kruskal->cant_ramas > 0)
        {
            buscarEnPapelera(kruskal, papelera);
        }
        else
        {
            printTXT(&(kruskal->cabezas)[0], "Kruskal.txt");
           // printf("El grafo esta impreso en kruskal.txt en tu carpeta\n");
        }
    }
    else if (*arbol == NULL)
    {
        printf("No hay mas arbolito para jugar\n");
        EXIT_SUCCESS;
    }
}

int encuentra(int *i, rama **arbol) // Si encuentra = 1 es true busca i en ambos
vertices del puntero recorriendo toda la lista
{
    int verificacion = 0;
    if (*arbol)
    {
        rama *puntero;
        puntero = *arbol;
        verificacion = verificoAmbosVerices(i, puntero);
        if (verificacion == 0)
        {
            while (puntero->sig)
            {
                verificacion = verificoAmbosVerices(i, puntero);
                if (verificacion == 1)
                {
                    return verificacion;
                }
                else
                {
                    puntero = puntero->sig;
                }
            }
            verificacion = verificoAmbosVerices(i, puntero);
            return verificacion;
        }
        else
        {
            verificacion = verificoAmbosVerices(i, puntero);
            return verificacion;
        }
    }
    else
    {
       return 0;
```

```
    }
}

int encuentraEnGrafo(int *u, grafo *kruskal)
{
    int resultado = 0;
    int i;
    if (kruskal->cabezas[0] != NULL)
    {
        for (i = 0; i < VERTICES / 2; i++)
        {
            if ((encuentra(u, (kruskal->cabezas) + i)) == 1)
            {
                resultado = 1;
                 return resultado;
            }
            else if ((encuentra(u, (kruskal->cabezas) + i)) == 0)
            {
            }
        }
        return resultado;
    }
    else
    {
        resultado = 0;
        return resultado;
    }
    return resultado;
}

int encuentraLugarEnGrafo(int *u, grafo *kruskal)
{
      int i;      // printf("Entra a encuentra Lugar en Grafo\n");
    if (kruskal->cabezas != NULL)
    {
        int resultado = 0; ///
        while (resultado < kruskal->cant_ramas)
        {
            for (i = 0; i <= kruskal->cant_ramas; i++)
            {
                if (encuentra(u, (kruskal->cabezas) + i) == 1)
                {
                    resultado = 1;
                    return i;
                }
            }
            resultado++;
        }
        return kruskal->cant_ramas;
    }
    else
    {
        printf("No hay que buscar lugar si no esta en el grafo\n");
        EXIT_SUCCESS;
    }
}

int verificoAmbosVerices(int *vertice, rama *puntero)
{
    if (puntero->a.u == *vertice)
    {
        return 1;
    }
    else if (puntero->a.v == *vertice)
    {
        return 1;
```

```
    }
      else
            return 0;

}

int buscarIntMin(int a, int b)
{
    if (a < b)
    {
        return a;
    }
    else
        return b;
}

int buscarIntMax(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
        return b;
}

void imprimirGrafo(grafo miGrafo)
{
      int i;
    int contArista = 1;
    rama *puntero;
    for (i = 0; i <= miGrafo.cant_ramas; i++)
    {
        printf("Subgrafo %d: \n", i);
        imprimirArbol((miGrafo.cabezas) + i);
    }
    printf("\tCosto total del grafo: %d\n", miGrafo.costoTotal);
    if (miGrafo.cant_ramas > 0)
    {
        printf("Faltaron aristas para poder formar un grafo conexo\n");
    }
    EXIT_SUCCESS;
}

void buscarEnPapelera(grafo *kruskal, rama **papelera)
{
    if (*papelera != NULL)
    {
      //  printf("Hay para agregar en papelera..\n");
        if (kruskal->cant_ramas > 0)
        {

            rama *ramaMin = sacar_min(papelera);
            eliminarRama(ramaMin, papelera);
            int u = ramaMin->a.u;
            int v = ramaMin->a.v;
            int lugarU = encuentraLugarEnGrafo(&u, kruskal);
            int lugarV = encuentraLugarEnGrafo(&v, kruskal);
            int minPos = buscarIntMin(lugarU, lugarV);
            int maxPos = buscarIntMax(lugarU, lugarV);

            if (maxPos == minPos)
            {

                free(ramaMin);
                    buscarEnPapelera(kruskal, papelera);
```

```
            }

            else if (maxPos != minPos)
            {

                rama *puntero;
                puntero = *(&((*kruskal).cabezas)[minPos]);

                while (puntero->sig != NULL)
                {
                    puntero = puntero->sig;
                }
                puntero->sig = ramaMin;
                ramaMin->sig = *(&(kruskal->cabezas)[maxPos]);
                (kruskal->cabezas)[maxPos] = NULL;
                kruskal->cant_ramas--;
                kruskal->costoTotal += ramaMin->a.costo;
                kruskal->cant_aristas++;
                if (kruskal->cant_ramas > 0)
                {
                    buscarEnPapelera(kruskal, papelera);
                }
                else
                {
                    printTXT(&(kruskal->cabezas)[0], "Kruskal.txt");
                   EXIT_SUCCESS;
                }
            }
        }
        else
        {
            printTXT(&(kruskal->cabezas)[0], "Kruskal.txt");
           EXIT_SUCCESS;
        }
    }
    else if (kruskal->cant_ramas > 0)
    {
        printf("La papelera esta vacia y el grafo no es conexo\n");
        EXIT_FAILURE;
    }
}

void printTXT(rama **lista, char nombreArchi[12])
{

    int costoTotal = 0;
    contArista = 0;
    FILE *archi = fopen(nombreArchi, "w");
    rama *puntero;
    if (*lista != NULL)
    {
        puntero = *lista;
        fprintf(archi, "La arista %d tiene vertices u %d y v %d de costo %d\n",
contArista, puntero->a.u, puntero->a.v, puntero->a.costo);
        costoTotal += puntero->a.costo;
        while (puntero->sig)
        {
            puntero = puntero->sig;
            contArista++;
            fprintf(archi, "La arista %d tiene vertices u %d y v %d de costo %d\n",
contArista, puntero->a.u, puntero->a.v, puntero->a.costo);
            costoTotal += puntero->a.costo;
        }
        fprintf(archi, "El costo total es de %d\n", costoTotal);
    }
    else
```

```
    {
        fprintf(archi, "No hay nada en la lista");
        fclose(archi);
        printf("Cierro archi\n");
        printf("La lista se imprimio en el archivo %s\n", nombreArchi);
    }
}
```

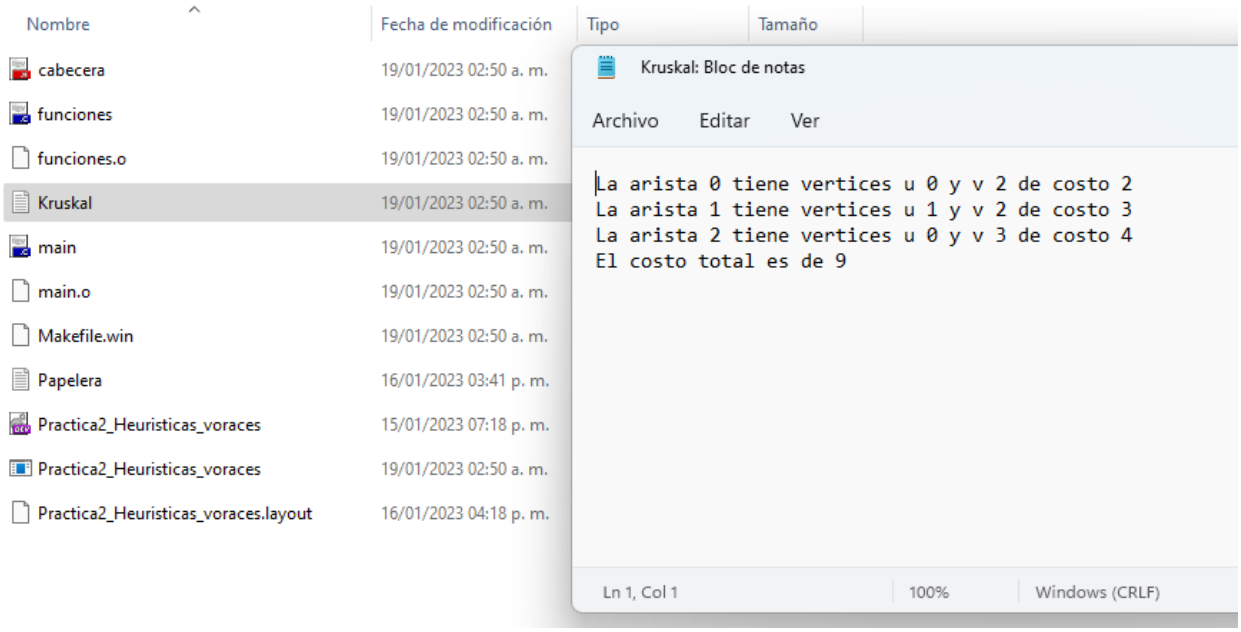
Cabecera.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define VERTICES 5

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;

void inserta(int, int, int, rama **);                                   // La uso
solamente para el arbol 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 arbol y la pongo en el Kruskal o en la papelera
rama *sacar_min(rama **arbol);                                          //
Recorre todo el arbol y devuelve la arista de costo minimo para despues 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
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 vertice en comun agrega y manda a papelera si estan los
dos vertices
void buscarEnPapelera(grafo *kruskal, rama **papelera);                 // Trae
de la papelera los minimos que habian sido descartados porque algun vertice se repetia
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 vertices de cada puntero que recorre
int encuentraEnGrafo(int *u, grafo *kruskal);                           // Me da
1 por si y 0 por no buscando el vertice en el grafo // Llama a encuentra
```

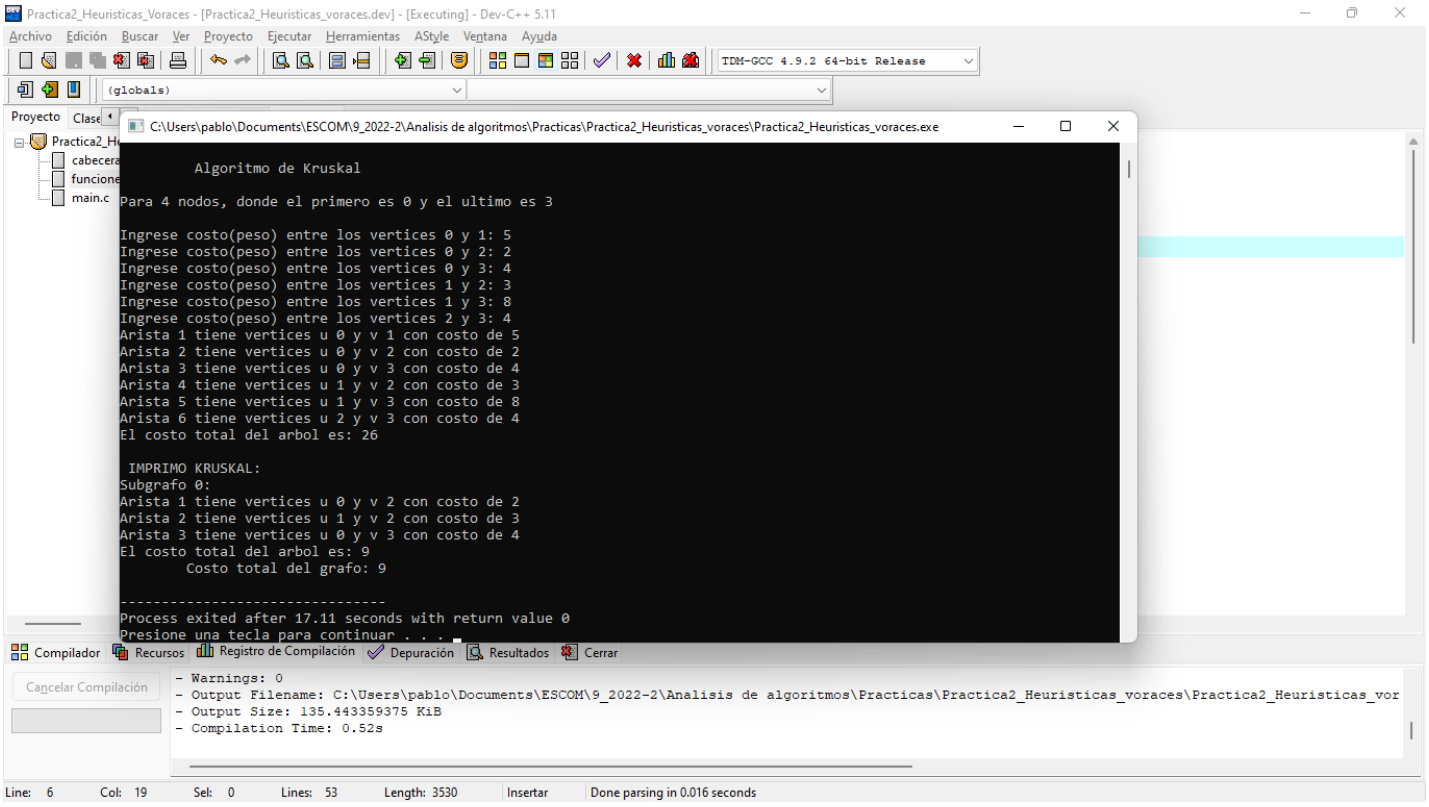
```
int verificoAmbosVerices(int *vertice, rama *puntero);                          //
Recorre los punteros de la lista verificando si el nro de vertice esta en cualquiera de
los dos
int encuentraLugarEnGrafo(int *u, grafo *kruskal);                              // Me
devuelve la posicion de la lista que tiene un vertice en comun con el vertice 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 vertices 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 posicion, 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

int contArista;
```

3. Evidencias

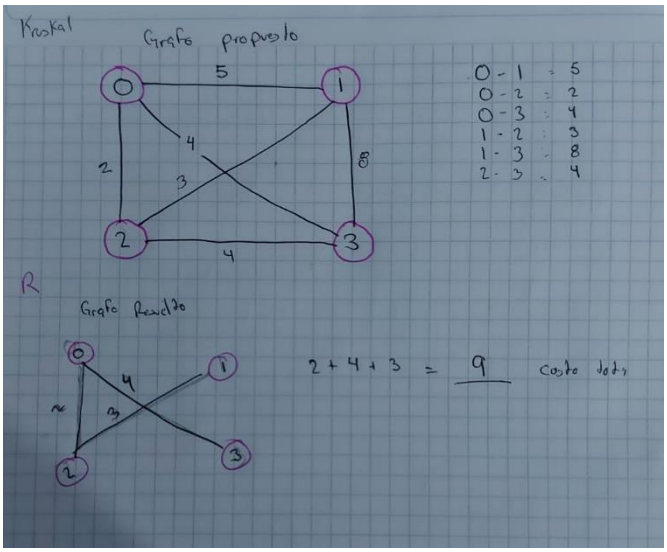
Al ingresar los datos se crea el archivo con el kruscal ya hecho y el costo total



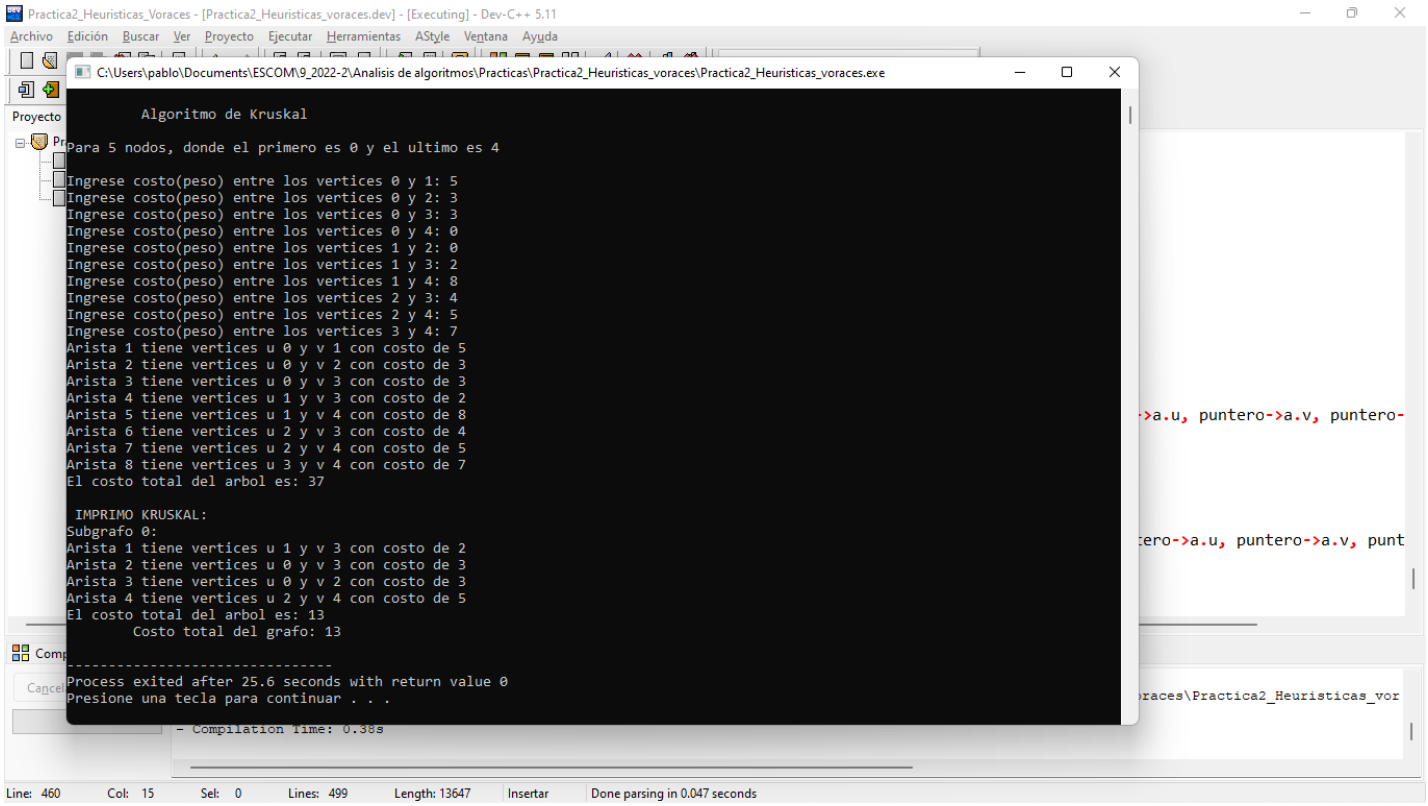
Aquí lo va haciendo un poco mas detallado



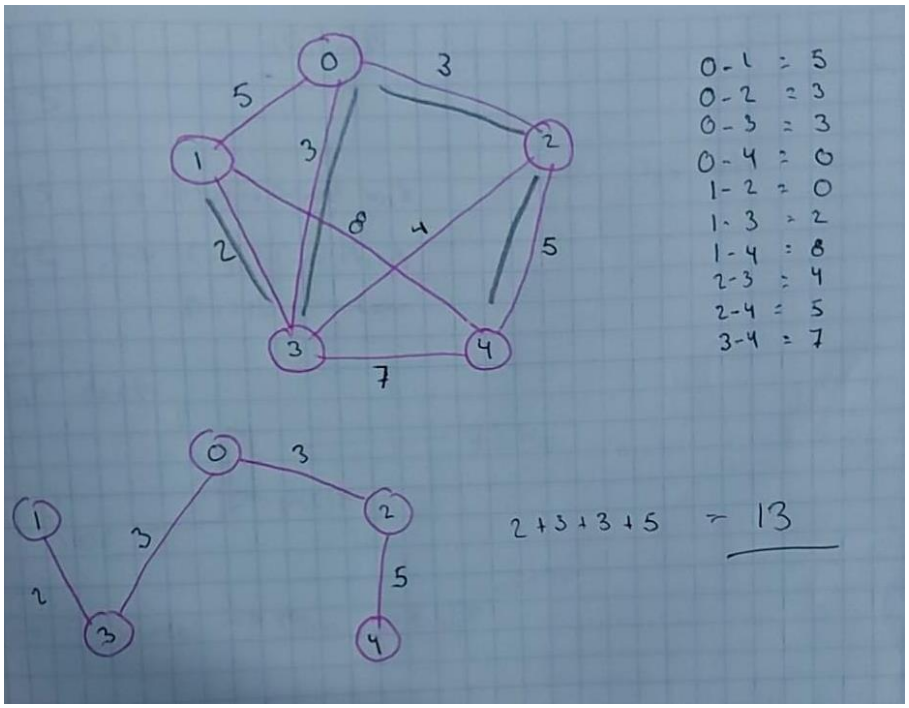
La comprobación con una prueba de escritorio



Segunda prueba



Prueba de escritorio para comprobar resultados



4. Referencias

https://github.com/mcarraced012/Kruskal/blob/master/Documentacion.pdf