Entrega #6: Proyecto final

Grupo 3

David Bernal Gómez

Xuban Barberena Apezetxea

Sara Martín Aramburu

Asignatura de Diseño de Algoritmos
(22/05/2022)

Modelo A
Hecho en Python
IDE: Anaconda Spyder

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Índice

1. Introducción

El departamento de tráfico de una gran ciudad ha decidido llevar a cabo obras de mantenimiento en varias calles a la vez. Evidentemente esto implica que, mientras duren las obras, no se podrá circular por algunos tramos. Disponemos de un mapa de la ciudad con todas las calles y sus correspondientes cruces. Conocemos también cuál es el volumen de tráfico habitual entre cada par de cruces adyacentes y suponemos que todos los tramos son bidireccionales. Queremos saber qué tramos hay que cortar (todos los que se puedan) de manera que ningún cruce quede aislado y maximizando la suma del volumen de tráfico de los tramos que quedan abiertos.

En este trabajo trataremos de abordar este problema mediante dos algoritmos: Kruskal y Prim. Estos algoritmos tienen como objetivo obtener el árbol de recubrimiento mínimo, es decir, obtener las aristas que recorran el grafo con el menor coste posible. Deberemos adaptar estos algoritmos para obtener la solución del problema que se nos pide.

También se han desarrollado una serie de ficheros de prueba, que nos servirá para comprobar la implementación y analizar su coste.

2. Algoritmos

Utilizaremos los algoritmos Prim y Kruskal, pero con algunas modificaciones:

- Estos algoritmos devuelven el árbol de recubrimiento mínimo, pero nosotros queremos un árbol de recubrimiento máximo; obtener las aristas que recorran el grafo con el mayor coste (tráfico) posible.
- No queremos devolver el árbol como tal, sino aquellos tramos que se puedan cortar, es decir, las aristas que no pertenezcan a éste. Es por ello que comenzamos añadiendo todas las aristas a la solución, y vamos eliminando aquellas que pertenecen al árbol.

2.1. Kruskal

El algoritmo de Kruskal se ha realizado de forma similar al presentado en clase, pero con algunas pequeñas modificaciones:

- A la hora de realizar la selección voraz, elegimos las aristas de mayor peso y no las de menor peso.
- En la solución metemos todas las aristas del grafo, y eliminamos las pertenecientes al árbol, para que las resultantes sean las que hay que cortar.

También añadimos un dato: tráfico, que sirve para calcular el peso de las aristas que quedan abiertas.

Pseudocódigo:

```
Func Kruskal (G=<N,A>) return ConjuntoAristas
        Solución <- listaAristas(G)
        trafico<-0
        numNodos<-0
        P<- N.size()          //Número de nodos.
        Candidato<-A          //Al inicio, todas las aristas son candidatas.
        Ordenar(Candidato)  //Por peso descendente.
        Inicializar(partición[1..P])        //Cada nodo en un conjunto distinto, con
                                            altura 0.
        While Solución.size<P-1 loop
                (a,b)<-Candidato.removeFirst() //Arista con mayor peso.
                Ra<- buscar(partición,a)  //Devuelve etiqueta del conjunto al que
                                            pertenece a.
                Rb<- buscar(partición,b)
                If Ra!=Rb then              //Si están en conjuntos distintos no hay
                                            ciclo.
                        Unir (particón, Ra, Rb)
                        Solución<- Solución.remove(a,b)//Eliminar la arista de la
                        numNodos<-numNodos+1            solución.
                        trafico<-trafico+peso(a,b)
                End if
        End loop
        Return trafico, solucion.size, solución
end
```

2.2. Prim

El siguiente algoritmo es el de Prim, que se basa en el diseño de un árbol en el que se incluyen los nodos que pertenecen a la solución.

- AR es el conjunto de nodos que están en el árbol.
- C es el conjunto de nodos que NO están en el árbol.

Al igual que en el algoritmo de Kruskal, distinguimos varios elementos:

- **Candidatos**: Los nodos que todavía no están en el árbol.
- **Pesos:** Lista que guarda los pesos de cada nodo candidato.
- **Array padre**: Array que guarda el padre de cada nodo.

Las únicas modificaciones respecto al algoritmo visto en clase, es que se parte de una solución con todas las aristas. Se van eliminando las aristas de mayor peso, hasta eliminar todo el árbol, quedando así las aristas que hay que cortar como solución.

Al igual que en el algoritmo anterior, añadimos un dato: tráfico, que sirve para calcular el peso de las aristas que quedan abiertas.

Pseudocódigo:

```
Func Prim (G=<N,A>) return ConjuntoAristas
   inicializarACero(D[1..N.size])    //Array de pesos.
   inicializarATrue(Candidatos[1..N.size])     //Al principio, todos los nodos son
                                               candidatos.
   Solución<- listaAristas(G)
   D[1]<- inf   //Al primer nodo le asignamos el mayor peso.
   Padre[1]<- -1 //El primer nodo es el padre.
   Cont<-0
   trafico<-0
   While cont!= N.size loop
           U<- buscarMax(candidatos, D)        //Buscar la arista de mayor peso entre
                                               los candidatos.
           Candidatos[u]<- false               //Ya no es candidato.
           If padre[u]!= -1 then               //Caso: es de otro árbol.
                  Solución.remove(padre[u], u)
                  trafico<- trafico+u
           End if
           Foreach v e adyacentes(u) loop
                  If Candidatos[v] and peso(u,v)> D[v] then   //El peso de esa arista
                                                              es mayor.
                         D[v]<- peso(u,v)
                         Padre[v]<-u
                  End if
           End loop
           Cont<- cont+1
   End loop
   Return (trafico, solución.size, solucion)
End
```

3.Funciones utilizadas

3.1. Funciones utilizadas en los algoritmos

Se han usado diferentes funciones para poder manejar la estructura *Partición*:

- **unir (particion[1..N], a, b):** Dada una partición y dos nodos, une ambos nodos en un único conjunto.
- **buscar(partición [1..N], i):** Dada una partición un nodo, devuelve la etiqueta del conjunto al que pertenece el nodo.

De la misma forma, hemos hecho uso de otras funciones complementarias:

- **buscarMaximo(candidato, D):** Dada una lista de nodos y una lista de pesos, devuelve el nodo candidato con mayor peso.
- **inicializarATrue(lista, N):** Dada una lista y su tamaño, inicializa todos sus valores a *True*.
- **inicializarACero(lista, N):** Dada una lista y su tamaño, inicializa todos sus valores a 0.
- **inicializarAMenosInfinito(lista, N):** Dada una lista y su tamaño, inicializa todos sus valores a menos infinito.
- **adyacentes(G, n):** Dado un grafo y un nodo, devuelve una lista que incluye todos los nodos adyacentes al nodo inicial.

3.2. Funciones utilizadas para hallar la solución

Las siguientes funciones se han utilizado para escribir en el formato correcto el resultado.

- **añadirNodos(G):** Dado un grafo, crea una lista con todos sus nodos.
- **añadirAristas(G):** Dado un grafo, crea una lista con todas sus aristas y sus respectivos pesos.
- **almacenarGrafo(dirección):** Lee un fichero que contiene un grafo y guarda su información (grafo, nodos y aristass)
- **escribirEnFichero(tráficos, tramosCerrados, S, dirección):** Dada la información de la solución al completo, la escribe en un fichero de salida con el formato indicado.

3.3. Funciones para realizar pruebas

Estas son funciones que se han utilizado para hacer pruebas y comprobar el correcto funcionamiento del programa.

- **calcularTrafico(G):** Dado un grafo, calcula el peso total de todas las aristas.
- **generarGrafo(nodos, aristas)**: Dado un número de nodos y de aristas, genera un grafo conexo aleatorio, con pesos aleatorios entre 1 y 50.

4. Coste temporal

4.1. Kruskal

Para calcular el coste temporal del algoritmo de Kruskal, tenemos que tener en cuenta todas las funciones a las que se le llama.

Para empezar, se ordenan los candidatos en función del peso de las aristas (en orden descendente). La función usada para ello ha sido sorted(), teniendo un coste de de O(A*logA), indicado en la documentación del lenguaje, siendo A el número de aristas del grafo.

El siguiente paso es inicializar la partición, que tiene un coste de O(N), siendo N el número de nodos del grafo.

Por último, se recorre un bucle N veces, en el que se les llama a las funciones buscar y unir, las cuales tienen un coste de O(logN) y O(1) respectivamente. Por lo tanto, este paso tiene un coste de (N*logN).

De este modo, el coste temporal total del algoritmo Kruskal es O(N*logN) + O(A*logA), que equivale a O(N*logN)

4.2. Prim

El algoritmo empieza con la inicialización de los parámetros necesarios para el algoritmo. Se llama a las funciones InicializarATrue, InicializarACero e inicializarAMenosInfinito una vez, y éstas tienen un coste de O(N), siendo N el número de nodos del grafo. De este modo, el coste antes de entrar al bucle es O(3N).

Por otro lado, el bucle se repite N veces, y se usan la funciones buscarMaximo y adyacentes, siendo O(N) el coste de cada una de ellas. Sin embargo, el resultado de adyacentes se recorre en un bucle for, por lo que en el caso peor el coste sería O(N^2). Teniendo en cuenta que esto se repite en el bucle while N veces, el coste del bucle en el caso peor sería O(N^3).

De este modo, el coste temporal total del algoritmo es $O(N^3)$.

5. Análisis experimental

Se han realizado varias ejecuciones del algoritmo con un total de 17 ficheros de prueba, 10 de ellos creados por nosotros, y los otros 7 proporcionados por la profesora.

Se han tomado dos mediciones distintas, la primera mide solo la ejecución del algoritmo, mientras que la otra mide la ejecución del algoritmo y el tiempo que tarda en escribir la respuesta en los ficheros. Como el tiempo de escritura es tan bajo en comparación al tiempo de ejecución del algoritmo, para la elaboración de las gráficas no se ha tenido en cuenta.

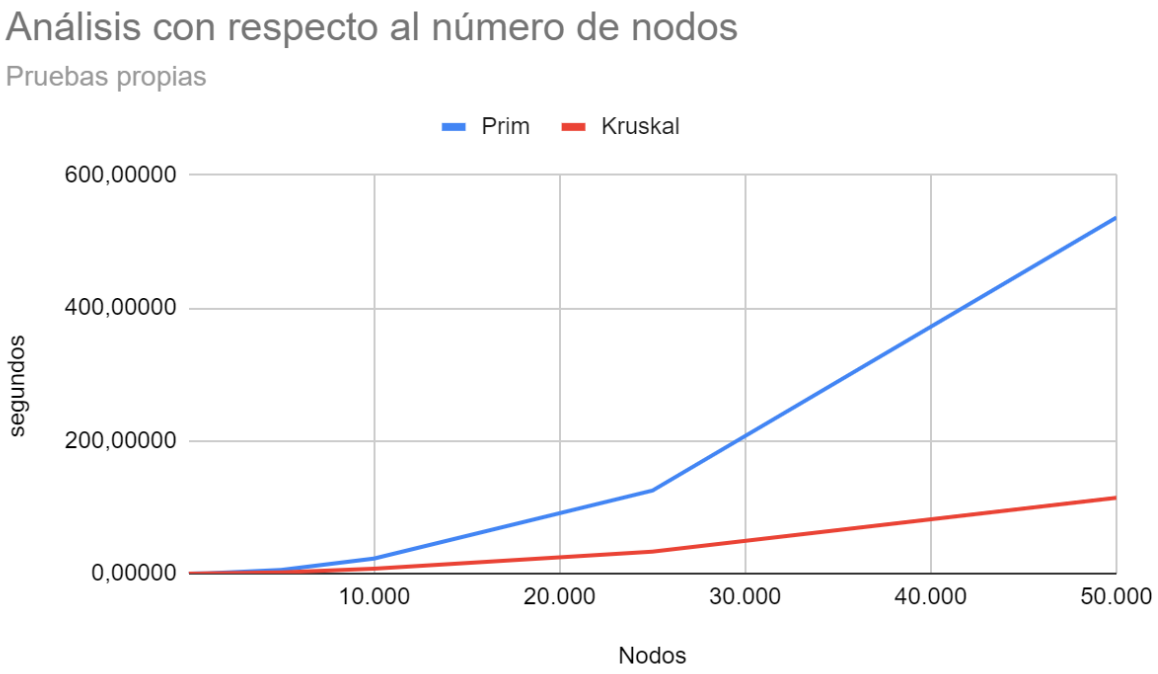
Por otra parte, el tiempo de preparación de los datos, esto es, leer el fichero y almacenar la información en variables, también es muy bajo, por lo que se ha descartado.

			PRIM (en segundos)		KRUSKAL (en segundos)	
	Nodos	Aristas	Sin escribir en fichero	Escribiendo en fichero	Sin escribir en fichero	Escribiendo en fichero
Grafo 1	5	6	0,00582	0,00725	0,00473	0,00547
Grafo 2	15	20	0,00735	0,01136	0,00118	0,00190
Grafo 3	30	35	0,00634	0,01478	0,00533	0,00584
Grafo 4	100	500	0,01216	0,01299	0,01453	0,01852
Grafo 5	500	1.000	0,04320	0,04619	0,02213	0,02711
Grafo 6	1.000	2.000	0,17787	0,18186	0,05107	0,05805
Grafo 7	5.000	20.000	5,52193	5,57578	1,83454	1,89437
Grafo 8	10.000	40.000	22,67734	22,78805	7,19431	7,31703
Grafo 9	25.000	60.000	125,01990	125,15254	33,01794	33,15458
Grafo 10	50.000	100.000	536,01244	536,58744	113,96711	114,29307
em 1	7	16	0,01396	0,01473	0,00147	0,00220
em 2	9	36	0,00540	0,00639	0,00227	0,00238
em 3	100	1.000	0,01136	0,01735	0,01439	0,01842
em 4	200	396	0,01694	0,01794	0,01125	0,01225
em 5	250	1.273	0,02756	0,03351	0,01600	0,02100
em 6	1.000	8.433	0,35798	0,39287	0,13309	0,16486
em 7	10.000	61.731	29,11961	29,34001	9,69948	9,92185

Las líneas de nombre “Grafo x” corresponden a nuestros casos de prueba, mientras que las líneas de nombre “em x” corresponden a casos de prueba proporcionados por la profesora.

Hemos decidido analizar el tiempo de ejecución en función de las aristas y de los nodos por separado:

Pruebas propias



Pruebas eGela

Como se puede ver, tanto en las pruebas propias como en las proporcionadas, el coste de ejecución del algoritmo en relación al número de nodos/aristas del grafo cumple con el análisis teórico comentado previamente. Esta diferencia entre algoritmos puede apreciarse mejor en las pruebas propias, ya que en ellas hay grafos de tamaños mayores. El algoritmo de Prim tiene un crecimiento potencial, del orden de $O(n^3)$, mientras que el crecimiento del algoritmo Kruskal es más lento, $O(n * \log n)$.