DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

7 de mayo de 2014 Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Chapresto, Matías	201/12	matiaschapresto@gmail.com
Dato, Nicolás	676/12	nico_dato@hotmail.com
Fattori, Ezequiel	280/11	ezequieltori@hotmail.com
Vileriño, Silvio	106/12	svilerino@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Índice

Resumen

1. Ejercicio 1

1.1. Descripción del problema

1.2. Ideas para la resolución

1.2.1. Algoritmo

1.3. Justificación del procedimiento

1.4. Cota de complejidad

1.5. Casos de prueba y resultado del programa

1.6. Mediciones de performance

1.7. Concluciones

2. Ejercicio 2

2.1. Descripción del problema

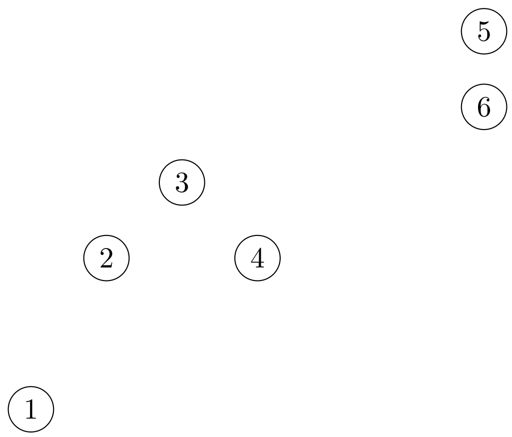
El problema se trata de un conjunto de ciudades hubicadas a cierta distancia entre ellas, las cuales todas deben de ser provistas de gas. Para ésto, tenemos una cantidad k de centrales distribuidoras de gas, y tuberías para conectar ciudades. Una ciudad tiene gas si hay un camino de tuberías que llegue hasta una central distribuidora, es decir, si una ciudad está conectada a otra ciudad por una tubería y a su vez ésta está conectada a otra ciudad por medio de una tubería la cual tiene una central distribuidora, entonces las 3 ciudades tienen gas. Se pide lograr que todas las ciudades tengan gas, pero que la longitud de la tubería más larga de la solución debe ser la más corta posible. La longitud de una tubería es igual a la distancia entre las 2 ciudades.

El algoritmo tiene que tener una complejidad de $O(n^2)$, con n la cantidad de ciudades.

De a partir de ahora, k va a ser siempre la cantidad de centrales distribuidoras disponibles, y n la cantidad de ciudades.

2.1.1. Ejemplo

Como entrada podemos tener 6 ciudades y 2 centrales distribuidoras, como en la Figura 1, y la solución que se busca es como indica la Figura 2.



Se tienen 2 centrales disponibles

Figura 1: La entrada del problema, con las 5 ciudades y la cantidad de centrales

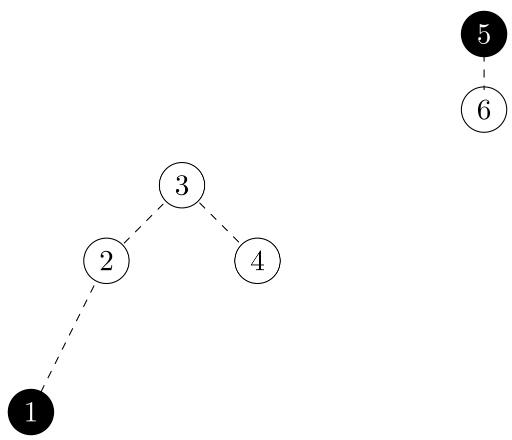


Figura 2: La solución al problema de la Figura 1, los nodos negros son los que contienen las centrales de distribución

2.2. Ideas para la resolución

Para la resolución del problema, se puede penzar en un *grafo*, donde las ciudades son nodos y las tuberías las aristas, cada arista va a tener una distancia asociada que es la distancia entre los nodos (ciudades) que conecta.

Como cada ciudad tiene gas si hay un camino hasta una central distribuidora, entonces todas las ciudades que se conectan a una misma central pertenecen a una misma componente conexa. Como tenemos un máximo de k centrales, la solución tiene que tener un máximo de k componentes conexas y las centrales se colocarán cada una en una componente conexa y en cualquier ciudad dentro de la componente conexa, ya que la distancia de las aristas dentro de una componente conexa no se verá modificada si se cambia de nodo la central dentro de la misma componente conexa.

Entonces, la solución que se pide es que el grafo tenga a lo sumo k componentes conexas, y que la distancia de la arista mas larga, sea la más corta posible.

Una idea que se propone es comenzar con todos los nodos sueltos, sin ninguna arista conectada, y calcular todas las distancias entre todos los nodos, es decir, calcular todas las tuberías posibles con sus respectivas distancias. Luego se ordenarán las aristas (tuberías) de menor a mayor de acuerdo a sus longitudes. Se calcula si la cantidad de componentes conexas es menor o igual a k, si es así, entonces el algoritmo termina, sino coloca la tubería más corta que no genere un ciclo y continúa preguntando sobre la cantidad de componentes conexas e iterando sobre las aristas siguiendo el orden de menor a mayor. Es parecido al algoritmo de *Kruskal*.

Para mejorar la complejidad del algoritmo, en vez de calcular la distancia de todos las aristas e iterar sobre todas las aristas, requiriendo ordenar por peso *todas* las aristas, primero creamos un árbol generador mínimo (con las aristas más cortas posibles), con una idea como el algoritmo de Prim y que tenga de cota $O(n^2)$, quedándonos $n-1$ aristas, y luego, al igual que antes, se ordenan y se recorren esas $n-1$ aristas de menor a mayor.

En la sección 2.2.1 se propone un pseudocódigo, en la sección 2.2.2 se verá un ejemplo de ejecución del algoritmo, en la sección 2.3 se justificará la correctitud, y en la sección 2.4 se hará el cálculo de la complejidad del algoritmo.

2.2.1. Algoritmo

Algorithm 1 minimizarTuberias

Require: *centrales*: cantidad de centrales disponibles, mayor que 0
Require: *ciudades*: las ciudades con sus posiciones
Require: n: cantidad de ciudades en el parámetro *ciudades*

Ensure: Retorna el grafo tal que hay tanas componentes conexas como *centrales*, y que la arista más larga es la más corta posible

```
procedure minimizarTuberias(Entero: centrales, Array: ciudades, Entero: n)→ Grafo
    Grafo g ← NuevoGrafo(n)                              ▷ Creo un grafo con n nodos, sin aristas
    ¡bool agregado, entero distancia, entero nodo¿nodos[n]    ▷ La distancia es desde n (índice del array) hasta nodos[n].nodo
    ¡nodo1, nodo2, distancia¿aristas[n - 1]

    nodos[0] ← ¡true, 0, 0¿
    for i ← 1; i < n; i++ do
        nodos[i] ← ¡false, Distancia(ciudades[i], ciudades[0]), 0¿
    end for

    for agregados ← 0; agregados ¡n - 1; agregados++ do
        distancia_minima ← ∞
        nodo_minimo ← 0
```

```
12:         for i ← 0; i < n; i++ do                ▷ Busco el nodo mas cercano al árbol que ya tenemos
13:             if nodos[i].agregado = false then
14:                 if nodos[i].distancia < distancia_minima then
15:                     distancia_minima ← nodos[i].distancia
16:                     nodo_minimo ← i
17:                 end if
18:             end if
19:         end for

20:         nodos[nodo_minimo].agregado ← true
21:         aristas[agregados] ← ¡nodo_minimo, nodo[nodo_minimo].nodo, distancia_minima¿ ▷ Agrego
    el nodo encontrado

22:         for i ← 0; i < n; i++ do                ▷ Actualizo la distancia de los nodos no agregados aún
23:             if nodos[i].agregado = false then
24:                 if nodos[i].distancia > Distancia(ciudades[i], ciudades[nodo_minimo]) then
25:                     nodo[i].distancia ← Distancia(ciudades[i], ciudades[nodo_minimo)
26:                     nodo[i].nodo ← nodo_minimo
27:                 end if
28:             end if
29:         end for
30:     end for

31:     Ordenar(aristas)                                      ▷ Ordeno las aristas por la distancia

32:     for componentes ← n; componentes > centrales; componentes−− do
33:         AgregarArista(g, aristas[n - componentes].nodo1, aristas[n - componentes].nodo2)
34:     end for
35:     retornar ← g
36: end procedure
```

2.2.2. Ejemplo de ejecución

2.3. Justificación del procedimiento

2.4. Cota de complejidad

Se analiza la complejidad de algoritmo 1, para ésto se supone que el cálculo de la distancia entre ciudades (el peso de las aristas/tuberías) se realiza en $O(1)$

La cantidad de nodos se representará como n, y la cantidad de centrales como k.

- La inicialización de unas variables se realiza de la línea 5 a 8, y se realizan n iteraciones calculando la distancia y guardando en un vector, quedando $O(n)$
- Para la creación del AGM entre las líneas 9 a 30 realiza $n-1$ iteraciones
  - Líneas 12 a 19, busca el nodo más cercano, realizando n iteraciones de comparaciones y asignaciones que son $O(1)$, quedandonos $O(n)$
  - Las líneas 20 y 21 son $O(1)$
  - La actualización de las distancias, entre las líneas 22 y 29 realiza n iteraciones de comparaciones y asignaciones que son $O(1)$, quedandonos $O(n)$
- Ordenar las aristas en la línea 31, y se realizan sobre las aristas agregadas en la iteración anterior, la cual agrega $n-1$ aristas, entonces se ordenan $n-1$ elementos. Al n no estar acotado, se puede realizar con algoritmos como *MergeSort* o *HeapSort* en $O(n \log n)$

- Entre las líneas 32 y 34 se realizan $n-k$ iteraciones, implementando el grafo g en una *matriz de adyacencia*, *AgregarArista* se realiza en $O(1)$, quedandonos una complejidad de $O(n-k)$. Notar que si $k > n$, no se realiza ninguna iteración.

Bajo éste análisis, la complejidad nos queda (1)

$$
\begin{aligned}
O(n) + (n-1) * (O(n) + O(1) + O(n)) + O(n \log n) + O(n-k) \\
= O(n) + O(n^2) + O(n) + O(n^2) + O(n \log n) + O(n-k) \\
= O(2n) + O(2n^2) + O(n \log n) + O(n-k) \\
= O(n^2)
\end{aligned}
\tag{1}
$$

Cumpliendo así la complejidad requerida.

2.5. Casos de prueba y resultado del programa

2.6. Mediciones de performance

2.7. Concluciones

3. Ejercicio 3

3.1. Descripción del problema

3.2. Ideas para la resolución

3.2.1. Algoritmo

3.3. Justificación del procedimiento

3.4. Cota de complejidad

3.5. Casos de prueba y resultado del programa

3.6. Mediciones de performance

3.7. Concluciones

4. Apéndice: Generación de casos de prueba

5. Apéndice: Código fuente relevante

5.1. Ejercicio 1

5.2. Ejercicio 2

5.3. Ejercicio 3

6. Apéndice: Entregable e instrucciones de compilacion y testing