Práctica 1 - AMC

Adrián Rodríguez Rodríguez
Eduard Catalin Neacsa
Sergio García Macías

Índice:

Introducción

En la práctica se nos plantea resolver el siguiente problema:

Encontrar el punto más cercano a otros dos dado un conjunto de puntos. Ya sea: generado de forma aleatoria o leído de un fichero.

Para ello se van a implementar 3 algoritmos en Java:

1. Exhaustivo,
2. Divide Y Vencerás,
3. Dijkstra.

Además también se realizarán unas implementaciones de divide y vencerás en otros lenguajes, para poder comparar rendimientos.

CÓDIGO

JAVA

Algoritmo Exhaustivo

https://github.com/ElPsyKoongroo/3-GII/blob/main/AMC/Practica1/App/src/Algoritmos/Exhaustivo.java

El algoritmo analiza *todas** las posibles combinaciones para determinar cual tiene la menor distancia y devolver el trío de puntos correspondiente.

* El algoritmo del código no analiza todas las posibles combinaciones ya que analizar:

    a -> b -> c

Sería lo mismo que analizar:

    c -> b -> a

Por eso el bucle for de la línea 26 empieza en i+1 y no en 0.

Algoritmo Divide Y Vencerás

https://github.com/ElPsyKoongroo/3-GII/blob/main/AMC/Practica1/App/src/Algoritmos/DyV.java

En este caso el algoritmo se divide en dos partes:

Parte recursiva:

> El algoritmo va dividiendo el mapa de puntos por la mitad recursivamente hasta que encuentra un grupo de 6 puntos o menos, al llegar a esto de manera exhaustiva se buscan los 3 puntos más cercanos y se guardan tanto los puntos como la distancia.

Parte expansiva:

> Con los puntos obtenidos en la fase anterior el algoritmo aumenta el rango de los puntos a calcular con la distancia mínima encontrada previamente.

Todo eso se repite hasta encontrar la mejor solución

100_000 puntos: Timeout (Tiempo de espera agotado +120s)

Algoritmo Dijkstra

https://github.com/ElPsyKoongroo/3-GII/blob/main/AMC/Practica1/App/src/Algoritmos/Dijkstra.java

La idea en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen

hasta el resto de los vértices que componen el grafo, el algoritmo se detiene.

IMPLEMENTACIÓN DE C++

https://github.com/ElPsyKoongroo/3-GII/blob/main/AMC/cpp_version/src/main.cpp

En la implementación de C++ podemos observar como el tiempo de ejecución es mucho menor con la misma cantidad de puntos aun teniendo prácticamente el mismo código. Esto se debe a que C++ es un lenguaje compilado que no necesita de GC ni de VM para poder ejecutarse y que todas sus instrucciones son directamente ejecutadas por la CPU.

100_000 puntos: 3,2s

IMPLEMENTACIÓN EN RUST

https://github.com/ElPsyKoongroo/rust_points/blob/main/src/main.rs

El comportamiento del código en la implementación en Rust difiere un poco con los códigos anteriores de C++ y Java. Rust también es un lenguaje compilado por lo que cabe esperar un rendimiento mayor al de Java. Debido a la forma en la que Rust nos obliga a escribir el código el tiempo de ejecución (con la misma cantidad de puntos) es incluso inferior a C++.

100_000 puntos: 2,3s

Los benchmarks fueron realizados en un i7-12700K, Windows.

Análisis Complejidad

Algoritmo Exhaustivo

El Bucle for cumple lo que se llama Variación sin repetición y sin simetría, es decir los 3 puntos tienen que ser diferentes y el último punto tiene que tener un índice mayor al primero. En nuestro caso la complejidad del bucle es de n*(n-1)*(n-2)/2 cuya cota superior es $O(n^3)$

Algoritmo Divide Y Vencerás

$$T \rightarrow \begin{cases} n^3 & \text{si } n<=6 \\ 2T(n/2) + 5n & \text{si } n>6 \end{cases}$$

Al aplicar teorema maestro y la teoría que hemos dado en clase a 2T(n/2) + 5n, resulta que la complejidad es n*log(n) debido a que la llamada recursiva divide el tamaño del array entre 2 lo que da como resultado un algoritmo log(n) y además en cada una de estas llamadas recursivas también se utilizan funciones con complejidad O(n) por lo que el algoritmo resultante es n*log(n). Ejemplo de caso genérico:

$T(n) = 2T\left(\frac{n}{2}\right) + 10n$

Como podemos ver en la fórmula de arriba, las variables tienen los siguientes valores:

$a = 2$
$b = 2$
$c = 1$
$f(n) = 10n$
$f(n) = \Theta\left(n^c \log^k n\right)$ donde $c = 1, k = 0$

Luego, nos fijamos que cumpla la condición del caso 2:

$\log_b a = \log_2 2 = 1$,luego, se cumple que $c = \log_b a$

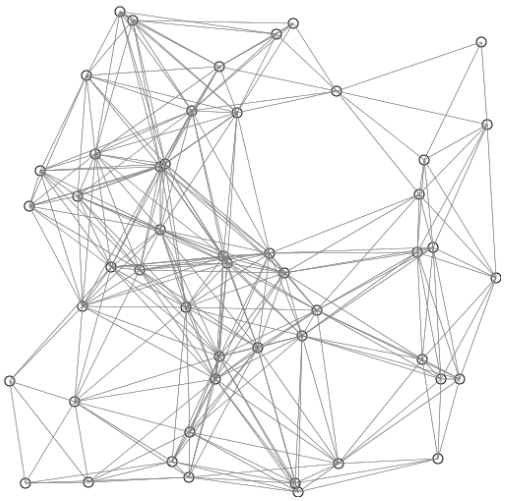
Entonces por el segundo caso del teorema maestro:

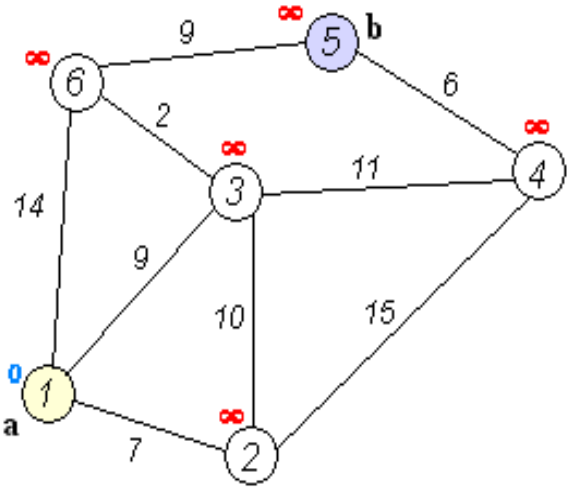
$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right) = \Theta\left(n^1 \log^1 n\right) = \Theta\left(n \log n\right)$

Dando de esa manera que la relación de recurrencia de *T*(*n*) es Θ(*n* log *n*).

Algoritmo Dijkstra

El algoritmo de Dijkstra tarda O(n) en visitar los n vértices del grafo y O(n) como máximo en procesar un vértice, dándonos por tanto una complejidad de O(n²).





Conclusión

Aunque el propósito de la práctica nos parezca interesante, no estamos de acuerdo con que esta se realice en el lenguaje Java. La finalidad de la práctica es analizar la complejidad del algoritmo y para ello necesitamos contar las operaciones elementales que este realiza. Al hacer uso de Java, un lenguaje interpretado*, no podemos realmente contarlas por lo que tenemos que ir suponiendo que “el acceso a una ArrayList” es una operación elemental (cuando no lo es) o que la instanciación de un nuevo objeto también lo es, cuando tampoco lo es. Por suponer, podríamos hasta suponer que buscar donde está un elemento dado en una ArrayList es una operación elemental porque solo llamamos a un método, y no es así.

Pensamos que la práctica debería hacerse en C++, aunque esta solo diera resultados por consola debido a que hacer interfaces gráficas en C++ es mucho más complejo que hacerlas en Java. Y que después se implementara la parte gráfica en Java.

* Java es un lenguaje que compila a bytecode y después la JVM interpreta ese bytecode.