Universidad Zaragoza

Trabajo de fin de Grado:

Problema de todos los pares de caminos óptimos en un grafo

Facultad de Ciencias
Universidad Zaragoza

Lorenzo Sánchez Fabián

Tutor: Alfredo García Olaverri

Universidad de Zaragoza

Julio de 2018

Summary

The aim of this project is to see that the All-Pairs Shortest Paths problem can be solved in less than $O(n^3)$ time. To do that, we divide the project into three chapters.

The first chapter, called **Computational complexity of an algorithm**, is an introduction to show a list of basic concepts such as algorithms and computational complexity, and to give a notation, the Θ, O and Ω notation, in order to delimit the size of a problem depending on the input n.

Knowing these concepts, we can continue with the next chapter called **Graph definition. All-Pairs shortest paths problem in a graph**. In the second chapter we introduce a graph definition, its different characteristics and particularities such as directed and non-directed graphs, or the shortest-path problem. Besides, we explain different ways to show graphs with their respectives advantages and disadvantages.

We describe the All-Pairs Shortest Paths problem and we analyse deeply the classic Floyd-Warshall Algorithm explaining what this algorithm is and studying its complexity, which is $O(n^3)$.

To end this chapter we introduce the matrix formulation for the All-Pairs Shortest Paths problem. In order to do that we present a special algebraic structure called *closed semiring* and the *closure* operation. Also we give a particular closed semiring called S_2 because we are going to use it throughout the project. Afterwards we introduce an algorithm which has a computational complexity of $O(n^3)$ time, and, if we compute that in a particular closed semiring, it solves the All-Pairs Shortest Paths problem.

We will see that this algorithm is equivalent to compute the closure operation of a matrix, which is a composition of operations with matrix, and by this way we transform the All-Pairs Shortest Paths problem in a matrix problem.

The classic Floyd-Warshall Algorithm is the most useful and the most known algorithm, and in practice is the most used algorithm to solve the All-Pairs Shortest Paths problem. However, knowing that the aim of this project is to show that we can solve the All-Pairs Shortest Paths problem in less time, we are going to give a new algorithm, created by Timothy M. Chan, which solves that problem in $O(\frac{n^3}{logn})$ time.

Finally, in the third chapter called **New algorithms. The Chan's method** we explain the Chan's method to solve the All-Pairs Shortest Paths problem, which consists on computing the same operations that we computed with the previous algorithm but in a different way, being the computational complexity $O(\frac{n^3}{logn})$, which is better than $O(n^3)$.

Therefore, since we have seen before that those operations are equals to the All-Pairs Shortest Paths problem, hence we can conclude that the All-Pairs Shortest Paths problem can be solved in $O(\frac{n^3}{logn})$ time.

Firstly, to do this, we reduce the problem to a geometric subproblem. We add the *dominating pair* concept for different sets of points. Then, we describe an algorithm based on the "divide and conquer"

technique and we give a recurrence to delimit the time complexity of that algorithm. Due to this recurrence, we can give the time complexity of that algorithm. After that, we see that we can apply this algorithm to solve the matrix product in the S_2 closed semiring. That is why we can give its time complexity. As we have seen before, this equals to the All-Pairs Shortest Paths problem so we can give a better time complexity to solve this problem.

To end this chapter, we add some interesting information such as similarities between this algorithm and previous algorithms. We also add some interesting historical facts such as this time complexity keeps for boolean matrix or Fredman's studies to solve this problem in subcubic time.

Due to all this facts, we can say that we have achieved the purpose of this project.

To conclude, the relevant bibliography used in this project has been added at the end.

Índice general

Summary III

1. Complejidad computacional de un algoritmo **1**
1.1. Definición de complejidad computacional de un algoritmo en el caso más desfavorable. 1
1.2. Tamaño de problemas. Notación Θ, O y Ω. 2

2. Definición de grafo. Problema de caminos más cortos en un grafo. **5**
2.1. Representación de grafos. 6
2.2. Algoritmo clásico de Floyd-Warshall y análisis de su complejidad. 6
2.3. Planteamiento del problema de forma matricial. 9

3. Nuevos algoritmos. El método de Chan **13**
3.1. Un subproblema geométrico 13
3.2. El algoritmo de todos los pares de caminos óptimos en un grafo 16
3.3. Discusión 19

Bibliografía **21**

Capítulo 1

Complejidad computacional de un algoritmo

Este capítulo es un compendio de los métodos y herramientas que utilizaremos para analizar algoritmos. Esta parte del trabajo también define con precisión varias notaciones asintóticas, como por ejemplo la Θ-notación. El resto del capítulo es básicamente una presentación de notación matemática.

Definición. Un **algoritmo** es cualquier procedimiento que funcione paso a paso para resolver un problema, donde cada paso se puede describir sin ambigüedad y sin hacer referencia a una computadora en particular, y además tiene un límite fijo en cuanto a la cantidad de datos que se pueden leer o escribir en un solo paso. Esta amplia definición abarca tanto a algoritmos prácticos como aquellos que solo funcionan en teoría con números de precisión infinita.

La **entrada** son los datos que se le suministran al algoritmo antes de comenzar.

Un algoritmo funciona en tiempo discretizado -paso a paso-, definiendo así una secuencia de estados computacionales por cada entrada válida.

1.1. Definición de complejidad computacional de un algoritmo en el caso más desfavorable.

El primer paso para resolver un problema informático es encontrar un algoritmo que lo resuelva. Pero una vez resuelto el problema, nos surgen las siguientes dudas: ¿Existen algoritmos ”mejores” para resolver el problema?

Una de las cosas más importantes a tener en cuenta a la hora de seleccionar un algoritmo es el tiempo que va a tardar en obtener una salida. En general se atiende a razones de economía en cuanto al número de operaciones que necesita el algoritmo para resolver el problema. Un algoritmo, será mejor que otro en tiempo si, actuando sobre los mismos datos, el tiempo de ejecución es menor, y será mejor que otro en espacio si, actuando sobre los mismos datos, la memoria, principal o secundaria, que utiliza es menor. A pesar de que los ordenadores de hoy en día son capaces de realizar millones de operaciones en un segundo, es muy fácil encontar ejemplos de problemas y soluciones que tardarán años en terminar. Los ordenadores de hoy en día tienen distintas capacidades en cuanto a velocidad de procesamiento. Saber con exactitud el tiempo que va a tardar un programa en dar una salida es una tarea difícil.

La **complejidad computacional de un algoritmo** es una medida del coste que supone su ejecución tanto en tiempo (o número de operaciones ”elementales” que ha de realizar) como en espacio (o unidades de memoria requeridas para almacenar y manipular los datos a lo largo de la ejecución) y vendrá dada en función del número de datos de entrada n.

En lugar de calcular el tiempo exacto que va a tardar nuestro algoritmo, se aproxima la cantidad de operaciones que realiza. Por lo general, para estimar el tiempo de ejecución, esta función de crecimiento

del tiempo de ejecución se multiplica por una constante c que representa una estimación del tiempo que el ordenador tarda en realizar una operación, $cf(n)$.

El tiempo exacto de ejecución de un algoritmo también depende en gran medida de los detalles de implementación. Normalmente se habla de dos tipos de análisis: Análisis del peor caso y análisis del caso promedio.

- El análisis del peor caso es calcular la función de crecimiento del tiempo de ejecución dándole la peor de todas las entradas posibles.
- El análisis del caso promedio es el promedio de cuánto tardaría nuestro algoritmo en cada una de las entradas posibles.

De los dos análisis, el del peor caso es el más fácil de realizar. Partiendo de la base de que es casi imposible probar todas las entradas posibles, los análisis de complejidad en promedio son en general muy complicados.

Para analizar el coste computacional de un algoritmo, básicamente se siguen tres pasos:

1. Suponer el peor de los casos, en el que se ejecutan más instrucciones (A veces encontar el peor caso no es sencillo).
2. Asignar un costo a cada instrucción de código.
3. Ver cuantas veces va a ser ejecutada cada instrucción de código.

Cuando se analizan algoritmos, sólo importa el término de la función que crece más rápido y se eliminan las constantes. El cálculo de la complejidad tan sólo nos da una aproximación del tiempo. Los detalles de implementación son un factor importante, aunque la complejidad sea la misma.

Si n es el tamaño del conjunto de datos de entrada de un determinado algoritmo, sus complejidades en tiempo y en espacio son funciones positivas de n y, en cuanto a su análisis, se está especialmente interesado en su comportamiento asintótico, es decir, en saber cómo se comportan para valores grandes de n.

Por ejemplo, para el algoritmo de la suma usual de enteros, si queremos sumar dos números de n cifras cada uno, la complejidad del algoritmo es de orden n pues sumamos las cifras una por una. En cambio para el algoritmo del producto usual de dos enteros de n cifras, la complejidad es de orden n^2, pues cada cifra la multiplicamos por las n cifras del segundo número.

1.2. Tamaño de problemas. Notación Θ, O y Ω.

El orden de crecimiento del tiempo que tarda en terminar un algoritmo, da una caracterización simple de la eficacia del algoritmo, y también permite que comparemos el rendimiento relativo de algoritmos alternativos.

Aunque a veces podemos determinar exactamente el tiempo de funcionamiento de un algoritmo, la precisión extra no suele valer el esfuerzo de calcularlo. Para entradas bastante grandes, los términos que dan la precisión extra al tiempo de funcionamiento de un algoritmo, están dominados por el propio tamaño de entrada del algoritmo

Cuando el tamaño de la entrada es lo suficientemente grande queremos conocer su influencia en el tiempo de funcionamiento del algoritmo, es decir, queremos estudiar como aumenta el tiempo de funcionamiento de un algoritmo en función del tamaño de la entrada cuando esta aumenta ilimitadamente.

Este capítulo da varios métodos estándar para simplificar el análisis asintótico de algoritmos. La siguiente sección comienza definiendo varios tipos de ”notación asintótica”. Se presentan varios convenios de notación utilizados a lo largo de este trabajo, y finalmente repasamos el comportamiento de las funciones que comunmente surgen en el análisis de algoritmos.

Notación asintótica

Las notaciones que utilizamos para describir el tiempo de funcionamiento de un algoritmo, se definen en términos de funciones sobre el conjunto de los números naturales $\mathbb{N} = \{0, 1, 2, ...\}$. Tales notaciones son adecuadas para describir la función de tiempo de funcionamiento en el caso más desfavorable de un algoritmo $T(n)$, que se define sólo para tamaños enteros de la entrada.

Definición. Dada una función $g(n)$, denotamos por $\Theta(g(n))$ al conjunto de funciones

$\Theta(g(n)) = \{f(n)$: Existen constantes positivas c_1, c_2 y n_0 tales que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$.

Aunque $\Theta(g(n))$ es un conjunto, escribimos "$f(n) = \Theta(g(n))$" ó "$f(n) \in \Theta(g(n))$" para indicar que $f(n)$ es un miembro de $\Theta(g(n))$.

La definición de $\Theta(g(n))$ requiere que cada miembro de $\Theta(g(n))$ sea asintóticamente no negativo, es decir, que $f(n)$ sea no negativo cuando n es suficientemente grande. Por tanto, la función $g(n)$ debe ser asintóticamente no negativa, o bien el conjunto $\Theta(g(n))$ está vacío.

Así la Θ-notación limita una función salvo por un factor constante.

Ejemplo 1. Sea $f(n) = an^2 + bn + c$ una función cuadrática, donde a, b y c son constantes y $a > 0$. Desechando los términos de orden inferior e ignorando los resultados constantes, vemos que $f(n) \in \Theta(n^2)$. Formalmente, para demostrar lo mismo, tomamos las constantes $c_1 = a/4$, $c_2 = 7a/4$ y $n_0 = 2\,\text{máx}\{| b | /a, \sqrt{| c | /a}\}$. Podemos comprobar que $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ para todo $n \geq n_0$.

En general, para cualquier polinomio $p(n) = \sum_{i=0}^{d} a_i n^i$, donde los a_i son constantes y $a_d > 0$, tenemos que $p(n) \in \Theta(n^d)$.

Puesto que cualquier constante es un polinomio de grado 0, podemos expresar cualquier función constante como $\Theta(1)$.

La Θ-notación limita asintóticamente una función por arriba y por abajo. Cuando sólo tenemos un límite superior asintótico utilizamos la O-notación, y si sólo tenemos un límite inferior asintótico, la Ω-notación.

Definición. Dada una función $g(n)$, denotamos por $O(g(n))$ al conjunto de funciones

$O(g(n)) = \{f(n)$: Existen constantes positivas c y n_0 tales que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0\}$.

Utilizamos la O-notación para dar un límite superior de una función, salvo un factor constante.

Aunque $O(g(n))$ es un conjunto, escribimos "$f(n) = O(g(n))$" ó "$f(n) \in O(g(n))$" para indicar que $f(n)$ es un miembro de $O(g(n))$. Notar que $f(n) = \Theta(g(n))$ implica que $f(n) = O(g(n))$,y por tanto que $\Theta(g(n)) \subseteq O(g(n))$.

Por tanto nuestra prueba de que cualquier función cuadrática $f(n) = an^2 + bn + c$, donde $a > 0$, está en $\Theta(n^2)$ también muestra que cualquier función cuadrática está en $O(n^2)$. Además cualquier función lineal $an + b$ está en $O(n^2)$, lo cual es facilmente verificable tomando $c = a + | b |$ y $n_0 = 1$.

Puesto que la O-notación describe un límite superior, cuando la utilizamos para limitar el peor caso de tiempo de ejecución de un algoritmo, implícitamente también limitamos el tiempo de ejecución del algoritmo en entradas arbitrarias.

Definición. Dada una función $g(n)$, denotamos por $\Omega(g(n))$ al conjunto de funciones

$\Omega(g(n)) = \{f(n)$: Existen constantes positivas c y n_0 tales que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0\}$.

Para todos los valores n a la derecha de n_0, el valor de $f(n)$ está en ó por encima de $g(n)$. La Ω-notación proporciona un límite inferior asintótico para una función.

Como la Ω-notación da una cota inferior, la O-notación da una cota superior y la Θ-notación da una cota superior e inferior, podemos enunciar el siguiente teorema:

Teorema 1.1. *Dadas dos funciones cualesquiera $f(n)$ y $g(n)$, $f(n) = \Theta(g(n))$ sí y sólo sí $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$.*

Capítulo 2

Definición de grafo. Problema de caminos más cortos en un grafo.

Definición. Un grafo dirigido G es un par ordenado $G = (V, E)$ donde:

- $V \neq \emptyset$, es un conjunto de vértices o nodos.
- $E \subseteq \{(a,b) \in V \times V : a \neq b\}$ es un conjunto de pares ordenados de elementos de V, denominados aristas o arcos.

Dada una arista (a,b), a es su nodo inicial y b su nodo final. Vamos a ver sólo grafos con V finito. Se llama orden del grafo G a su número de vértices, $\mid V \mid$.

Definición. Un grafo no dirigido G es un par no ordenado $G = (V, E)$ donde:

- $V \neq \emptyset$ es un conjunto de vértices o nodos.
- $E \subseteq \{(a,b) \in V \times V : a \neq b\}$ es un conjunto de pares no ordenados de elementos de V, denominados aristas o arcos, pero que no tienen un sentido definido.

Definición. Llamamos camino p de un grafo G a una secuencia de vértices dentro del grafo tal que exista una arista entre cada vértice y el siguiente.

Si un camino empieza y termina en el mismo vértice se llama ciclo.

Las aristas tanto para los grafos dirigidos como para los no dirigidos pueden tener un costo, es decir, un valor asignado a cada arista. Los costos de las aristas son utilizados para representar tiempo, costes, sanciones, pérdidas o cualquier otra cantidad que se acumule linealmente a lo largo de un camino y que se desea minimizar.

Definición. Dado un grafo $G = (V, E)$ dirigido, con costos dados por la función de costo $\omega : E \longrightarrow \mathbb{R}$, el costo del camino $p = \langle v_0, v_1, ..., v_k \rangle$ es la suma de los costos de sus ejes constituyentes:

$$\omega(p) = \sum_{i=1}^{k} \omega(v_{i-1}, v_i)$$

Llamamos camino más corto de u a v a un camino p de u a v con costo mínimo, es decir, con costo:

$$\delta(u,v) = \text{mín}\{\omega(p) : u \xrightarrow{p} v\}$$

si hay un camino de u a v. Si no existe un camino de u a v definimos $\delta(u,v) = \infty$.

En este capítulo, vamos a estudiar el problema de encontar los caminos más cortos entre todos los pares de vértices en un grafo. Suponer que tenemos un grafo $G = (V, E)$ dirigido, con costos dados por una función $\omega : E \longrightarrow \mathbb{R}$, que asigna a cada arista (i, j) un costo $\omega(i, j)$. Deseamos encontrar, para cada par de vértices $u, v \in V$, el camino más corto de u a v.

2.1. Representación de grafos.

Hay dos formas estándar de representar un grafo $G = (V,E)$: Como una colección de listas de adyacencia o como una matriz de adyacencia.

La representación mediante lista de adyacencia es normalmente utilizada, porque proporciona una manera compacta de representar grafos para los cuales $\mid E \mid$ es mucho menor que $\mid V \mid^2$.

Una representación mediante matriz de adyacencia puede ser utilizada cuando $\mid E \mid$ está cerca de $\mid V \mid^2$, o cuando necesitamos saber rápidamente si hay una arista que conecta dos vértices dados.

La **matriz de adyacencia** A de un grafo $G = (V,E)$ tiene $\mid V \mid^2$ elementos y se define como:

$$a_{i,j} = \begin{cases} 1 & \text{si (i,j)} \in E \\ 0 & \text{en otro caso} \end{cases}$$

Esta forma de representar un grafo tiene las siguientes ventajas:

- Se puede determinar en un tiempo fijo y constante si una arista pertenece o no al grafo, sólo se debe posicionar en la matriz.
- Es fácil determinar si existe un ciclo en el grafo, basta multiplicar la matriz por sí misma n veces hasta obtener la matriz nula(no hay ciclos) o bien una matriz no nula(hay ciclos).

Esta forma de representar un grafo tiene los siguientes inconvenientes:

- Se requiere un almacenamiento $\mid V \mid^2$, es decir, $\Theta(n^2)$.
- Sólo el leer o examinar la matriz lleva un tiempo de $\Theta(n^2)$.

La **lista de adyacencia** para un vértice v es una lista de todos los vértices u adyacentes a v. Un grafo puede ser representado por $\mid V \mid$ listas de adyacencia, una para cada vértice.

Esta forma de representar un grafo tiene las siguientes ventajas:

- Requiere un espacio proporcional a la suma del número de vértices más el número de aristas. Hace buen uso de la memoria.
- Se utiliza bastante cuando el número de aristas es mucho menor que $\Theta(n^2)$.

Esta forma de representar un grafo tiene los siguientes inconvenientes:

- Puede llevar un tiempo $O(n)$ determinar si existe una arista del vértice u al v, ya que puede haber $O(n)$ vértices en la lista de adyacencia para el vértice u.

Aunque la lista de adyacencia es al menos tan asintóticamente eficiente como la matriz de adyacencia, la matriz de adyacencia puede ser preferible cuando los grafos son pequeños.

2.2. Algoritmo clásico de Floyd-Warshall y análisis de su complejidad.

Algunos algoritmos de caminos más cortos, como el algoritmo de Dijkstra, resuelven el problema de caminos más cortos desde un vértice inicial $s \in V$ a todos los demás vértices $v \in V$ de un grafo $G = (V,E)$ dirigido y con costos dados por la función de costo $\omega : E \longrightarrow \mathbb{R}$, para el caso en que todos los costos de las aristas sean no negativos.

Otros, como el algoritmo de Bellman-Ford, permiten aristas de costo negativo en el grafo $G = (V,E)$ y producen una respuesta correcta siempre y cuando no se alcancen los ciclos de costo negativo del

vértice inicial. Si hay tal ciclo de peso negativo, el algoritmo detecta su existencia, indicando que no existe solución.

Podemos solucionar un problema de caminos más cortos ejecutando cualquiera de estos algoritmos de caminos más cortos de un único vértice inicial $\mid V \mid$ veces, cada vez tomando como vértice inicial cada uno de los distintos $\mid V \mid$ vértices. En este capítulo veremos como hacerlo mejor.

El algoritmo de Floyd-Warshall

Vamos a utilizar una formulación de programación dinámica diferente para resolver el problema de todos los pares de caminos más cortos en un grafo dirigido $G = (V, E)$. El algoritmo resultante, conocido como **algoritmo de Floyd-Warshall**, se ejecuta en un tiempo $\Theta(\mid V \mid^3)$. Además, puede haber aristas de costo negativo, pero supondremos que no hay ciclos de costo negativo.

Primero analizaremos la estructura de un camino más corto. El algoritmo utiliza los vértices intermedios de un camino más corto.

Un vértice intermedio de un camino simple $p = \langle v_1, v_2, ..., v_l \rangle$, es cualquier vértice de p distinto de v_1 ó v_l, es decir, cualquier vértice del conjunto $\{v_2, v_3, ..., v_{l-1}\}$.

El algoritmo de Floyd-Warshall se basa en lo siguiente. Sea $V = \{1, ..., n\}$ el conjunto de los vértices de G y consideramos el subconjunto de vértices $\{1, ..., k\}$ de V para algún k. Para cualquier par de vértices $i, j \in V$, consideramos todos los caminos de i a j cuyos vértices intermedios pertenecen al conjunto $\{1, ..., k\}$, y que p sea un camino de costo mínimo entre ellos (El camino p es simple pues hemos supuesto que G no contiene ciclos de costo negativo). El algoritmo de Floyd-Warshall analiza una relación entre el camino p y los caminos más cortos de i a j con todos los vértices intermedios en el conjunto $\{1, ..., k-1\}$. las relaciones dependen de si k es o no un vértice intermedio del camino p.

- Si k no es un vértice intermedio del camino p, entonces todos los vértices intermedios de p están en el conjunto $\{1, ..., k-1\}$. Por tanto, el camino más corto de i a j con todos los vértices intermedios en el conjunto $\{1, ..., k-1\}$ es también el camino más corto de i a j con todos los vértices intermedios en el conjunto $\{1, ..., k\}$.

- Si k es un vértice intermedio del camino p, descomponemos p como $i \xrightarrow{p_1} k \xrightarrow{p_2} j$. Así p_1 es un camino más corto de i a k con todos los vértices intermedios en $\{1, ..., k\}$. En particular, el vértice k no es un vértice intermedio del camino p_1 y por tanto p_1 es un camino más corto de i a k con todos los vértices intermedios en el conjunto $\{1, ..., k-1\}$. Análogamente, p_2 es un camino más corto del vértice k al vértice j con todos los vértices intermedios en el conjunto $\{1, ..., k-1\}$.

Ahora vamos a definir una solución recursiva para el problema de todos los pares de caminos óptimos en un grafo dirigido con costos dados por la función de costo $\omega : E \longrightarrow \mathbb{R}$.

Sea $d_{i,j}^{(k)}$ el costo del camino más corto del vértice i al vértice j, con todos los vértices intermedios en el conjunto $\{1, ..., k\}$. Si $k = 0$, un camino más corto del vértice i al vértice j sin vértice intermedio numerado mayor que 0 quiere decir que no tiene vértices intermedios lo que implica que tiene como mucho una arista y así $d_{i,j}^{(0)} = \omega_{i,j}$. Una definición recursiva viene dada por:

$$d_{i,j}^{(k)} = \begin{cases} \omega_{i,j} & \text{si } k = 0 \\ \text{mín}\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} & \text{si } k \geq 1 \end{cases}$$

donde $\omega_{i,j}$ viene dado por

$$\omega_{i,j} = \begin{cases} 0 & \text{si } i = j \\ \text{costo } \omega(i,j) \text{ de la arista dirigida } (i,j) & \text{si } i \neq j \text{ y } (i,j) \in E \\ \infty & \text{si } i \neq j \text{ y } (i,j) \notin E \end{cases}$$

La matriz $D^{(n)} = (d_{i,j}^{(n)})$ da una respuesta final porque todos los vértices intermedios están en el conjunto $V = \{1,...,n\}$ y además $d_{i,j}^{(n)} = \delta(i,j)$ para todo i, $j \in V$.

Veamos ahora como construir todos los pares de caminos más cortos a partir del algoritmo de Floyd-Warshall.

Una forma es calcular la matriz D de los costos de los caminos más cortos y a partir de ella construir la matriz predecesora Π, que es la matriz que da los predecesores de cada vértice en los caminos más cortos. Vamos a apoyarnos en un ejemplo para entender mejor la definición de la matriz predecesora. Sea un grafo con 5 vértices. Si la matriz predecesora es

$$\Pi = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

y queremos hallar el camino más corto del vértice 1 al vértice 2 se calcula de la siguiente manera: Nos posicionamos en el elemento $\pi_{1,2}$ de la matriz y vemos que ese elemento es un 3. Esto significa que el vértice predecesor del 2 en el camino más corto de 1 a 2 es el 3. Por tanto ahora queremos calcular el camino más corto del vértice 1 al 3 y realizamos el mismo procedimiento, es decir, miramos el elemento $\pi_{1,3}$ de la matriz que es un 4. Volvemos a hacer lo mismo repetidamente hasta que el vértice predecesor sea el vértice inicial, en este caso el 1. Nos situamos en el elemento $\pi_{1,3}$ de la matriz el cual es un 4. Ahora nos situamos en el elemento $\pi_{1,4}$ de la matriz que es un 5. Finalmente nos situamos en el elemento $\pi_{1,5}$ de la matriz que, ahora sí, es el 1. Por tanto el camino más corto del vértice 1 al vértice 2 es $1 \to 5 \to 4 \to 3 \to 2$.

Este método puede ser implementado en un tiempo $O(n^3)$.

Podemos calcular la matriz predecesora Π tal como el algoritmo de Floyd-Warshall calcula las matrices $D^{(k)}$. Más concretamente, calculamos una secuencia de matrices $\Pi^{(0)}$, $\Pi^{(1)}$, ... ,$\Pi^{(n)}$, donde $\Pi = \Pi^{(n)}$ y $\pi_{i,j}^{(k)}$ se define como el predecesor del vértice j en un camino más corto desde el vértice i al j con todos los vértices intermedios en el conjunto $\{1,...,k\}$.

Podemos dar una fórmula recursiva de $\pi_{i,j}^{(k)}$. Cuando $k = 0$, un camino más corto de i a j no tiene vértices intermedios, por tanto

$$\pi_{i,j}^{(0)} = \begin{cases} \text{NIL} & \text{si } i = j \text{ ó } \omega_{i,j} = \infty \\ i & \text{si } i \neq j \text{ y } \omega_{i,j} < \infty \end{cases}$$

Para $k \geq 1$, tomando el camino $i \to k \to j$, tenemos que el predecesor de j que elegimos, es el mismo que el que elegiríamos en un camino más corto desde k con todos los vértices intermedios en el conjunto $\{1,...,k-1\}$. De lo contario, elegimos el mismo predecesor de j que elegiríamos en un camino más corto desde i con todos los vértices intermedios en el conjunto $\{1,...,k-1\}$. Más formalmente

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} \leq d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \\ \pi_{k,j}^{(k-1)} & \text{si } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \end{cases}$$

El siguiente procedimiento puede utlilizarse para calcular los valores $d_{i,j}^{(k)}$, con el propósito de aumentar los valores de k. Su entrada es una matriz $n \times n$, W dada por los costos de las aristas del grafo dado, es decir, por $\omega_{i,j}$.

El procedimiento devuelve la matriz D con los costos de los caminos más cortos y la matriz Π con los predecesores de cada arista en los caminos más cortos.

FLOYD-WARSHALL(W)

1. $n \leftarrow rows[W]$

2. $D \leftarrow W$ **and** $\Pi \leftarrow \Pi^{(0)}$
3. **for** $k \leftarrow 1$ **to** n
4. **do for** $i \leftarrow 1$ **to** n
5. **do for** $j \leftarrow 1$ **to** n
6. $\mathrm{d}_{i,j} \leftarrow \min\{d_{i,j}, d_{i,k} + d_{k,j}\}$
7. **if** $d_{i,j} > d_{i,k} + d_{k,j}$
8. **then** $\pi_{i,j} \leftarrow \pi_{k,j}$
9. **else** $\pi_{i,j} \leftarrow \pi_{i,j}$
10. **end if**
11. **end do**
12. **end do**
13. **end for**
14. **return** D **and** Π

El algoritmo de Floyd-Warshall es bastante práctico incluso para los grafos cuya entrada tiene un tamaño moderado.

Finalmente podemos comprobar fácilmente que la complejidad en tiempo del algoritmo de Floyd-Warshall es $\Theta(n^3)$ pues el algoritmo consta de tres bucles encajados. Por otro lado, la complejidad en espacio de memoria es $\Theta(n^2)$, pues el algoritmo realiza el proceso n veces y devuelve en cada iteración dos matrices $n \times n$, D y Π. Además actúa recursivamente guardando cada salida encima de la salida de la iteración anterior, por tanto el orden es $\Theta(n^2)$ y así devuelve sólo la matriz de la última iteración. Podría guardar en matrices diferentes las matrices resultantes en cada iteración y así el orden de memoria sería $\Theta(n^3)$, pero evidentemente esto es menos óptimo que guardar cada salida encima de la anterior.

2.3. Planteamiento del problema de forma matricial.

Primero vamos a presentar una estructura algebraica especial:

Definición. Un **semianillo cerrado** es un conjunto $(S, +, \cdot, 0, 1)$ donde S es un conjunto de elementos y $+$ y $\cdot$ son operaciones binarias en S que satisfacen las siguientes propiedades:

1. $(S, +, 0)$ es un monoide, es decir, la operación $+$ es una operación interna y asociativa, y 0 es el elemento neutro para la operación $+$. De la misma forma $(S, \cdot, 1)$ también es un monóide y 0 es un aniquilador, es decir, para todo $a \in S$, se tiene que $a \cdot 0 = 0 \cdot a = 0$.
2. + es conmutativa e idempotente, es decir, para todo $a \in S$, se tiene que $a + a = a$.
3. $\cdot$ es distributiva sobre +.
4. Si $a_1, a_2, ..., a_i, ...$ es una secuencia contable de elementos de S, entonces $a_1 + a_2 + ... + a_i + ...$ existe y es único.
5. $\cdot$ es distributiva sobre operaciones + finitas y sobre operaciones + infinitas contables. Por tanto las propiedades 4 y 5 implican que

$$(\sum_i a_i) \cdot (\sum_j b_j) = \sum_{i,j} (a_i \cdot b_j) = \sum_i (\sum_j (a_i \cdot b_j)).$$

Ejemplo 2. El siguiente sistema es un semianillo cerrado: $S_2 = (R, MIN, +, +\infty, 0)$ donde R es el conjunto de reales no negativos incluyendo a $+\infty$. Es fácil comprobar que $+\infty$ es el elemento neutro de la operación *MIN* y 0 el de la operación $+$. A lo largo de este trabajo vamos a trabajar principalmente con este semianillo S_2.

Definición. La operación llamada **cierre** es una operación singular denotada $*$, que es central para nuestro análisis de semianillos cerrados.

Si $(S, +, \cdot, 0, 1)$ es un semianillo cerrado y $a \in S$, definimos $a* = \sum_{i=0}^{\infty} a^i$ donde $a^0 = 1$ y $a^i = a \cdot a^{i-1}$. Es decir, $a*$ es la suma infinita $1 + a + a \cdot a + a \cdot a \cdot a + ...$ La propiedad 4 de la definición de semianillo cerrado asegura que $a* \in S$. Las propiedades 4 y 5 implican que $a* = 1 + a \cdot a*$. Notar que $0* = 1* = 1$.

Para S_2, $a* = 0$ para todo $a \in R$ pues en este caso $a* = MIN\{0, a, a+a, a+a+a, ...\}$.

Sea ahora un grafo dirigido $G = (V, E)$ donde el costo de cada arista es un elemento del semianillo cerrado $(S, +, \cdot, 0, 1)$. Definimos la **etiqueta de un camino** como el producto $(\cdot)$ de los costos de las aristas que forman el camino tomadas en orden. En particular, la etiqueta de un camino de longitud 0 es 1(la identidad para la operación $\cdot$). Para cada par de vértices (v, w) definimos **costo** de ir de v a w como la suma de las etiquetas de todos los caminos entre v y w, y se denota por $c(v, w)$. Por convenio si no existe ningún camino de v a w tomamos que el costo es 0 (la identidad para la $+$ del semianillo). Si G tiene ciclos puede haber infinitos caminos de v a w, pero los axiomas de un semianillo cerrado aseguran que $c(v, w)$ está bien definido. Notar que aunque parezca que esta definición de costo es diferente a la dada anteriormente, esto no es así ya que la definición es la misma pero como estamos es un semianillo cerrado distinto, la operación $\cdot$ toma el papel de la operación $+$ de la definición anterior, y la operación $+$ toma el papel de la operación *MIN* de la definición anterior.

Vamos a dar un algoritmo para calcular $c(v, w)$ para todos los pares de vértices.

Algoritmo: Cálculo de costos entre vértices

- INPUT-ENTRADA: Un grafo dirigido $G = (V, E)$ y una función de etiquetado $l : V \times V \longrightarrow S$, donde $(S, +, \cdot, 0, 1)$ es un semianillo cerrado. Tomamos $l(v_i, v_j) = 0$ si (v_i, v_j) no está en E.
- OUTPUT-SALIDA: Para todo i y j entre 1 y n, el elemento $c(v_i, v_j)$ de S.
- MÉTODO: Calculamos $C_{i,j}^k$ para todo $1 \leq i \leq n$, $1 \leq j \leq n$ y $0 \leq i \leq n$. Definimos $C_{i,j}^k$ como la suma de las etiquetas de todos los caminos de v_i a v_j tal que todos los vértices del camino, excepto los extremos están en el conjunto $\{v_1, v_2, ..., v_k\}$.

CÁLCULO DE COSTOS ENTRE VÉRTICES

1. **begin**
2. **for** $i \leftarrow 1$ **until** n **do** $C_{i,i}^0 \leftarrow 1 + l(v_i, v_i)$;
3. **for** $1 \leq i, j \leq n$ **and** $i \neq j$ **do** $C_{i,j}^0 \leftarrow l(v_i, v_j)$;
4. **for** $k \leftarrow 1$ **until** n **do**
5. **for** $1 \leq i, j \leq n$ **do**
6. $C_{i,j}^k \leftarrow C_{i,j}^{k-1} + C_{i,k}^{k-1} \cdot (C_{k,k}^{k-1}) * \cdot C_{k,j}^{k-1}$;
7. **for** $1 \leq i, j \leq n$ **do** $c(v_i, v_j) \leftarrow C_{i,j}^n$
8. **end**

Teorema 2.1. *El algoritmo anterior utiliza $O(n^3)$ operaciones $+, \cdot$ y $*$ del semianillo y calcula $c(v_i, v_j)$ para $1 \leq i, j \leq n$.*

Algoritmo de camino más corto

Para calcular caminos más cortos, utilizaremos el semianillo cerrado $S_2 = (R, MIN, +, +\infty, 0)$, donde ya hemos visto que $a* = 0$ para todo $a \in R$, así que podemos eliminar la operación $*$ en la línea 6 del algoritmo anterior, sustituyendo esa línea por

$$C_{i,j}^{k} \leftarrow MIN\{C_{i,j}^{k-1}, C_{i,k}^{k-1} + C_{k,j}^{k-1}\}.$$

Tomando $l(v_i, v_j)$ como el costo de la arista (v_i, v_j) si esta existe, y $+\infty$ en otro caso, el valor de $c(v_i, v_j)$ producido por el algoritmo será el costo mínimo para ir de v_i a v_j, pues, gracias al algoritmo de Floyd-Warshall, sabemos que esta orden sirve para resolver el problema de todos los pares de caminos óptimos en un grafo. Más concretamente, esta orden significa que el camino más corto de v_i a v_j que no pasa a través de ningún vertice superior a v_k es el más corto entre:

1. El camino más corto de v_i a v_j que no pasa a través de ningún vertice superior a v_{k-1}
2. El camino más corto de v_i a v_k y luego va a v_j sin pasar a través de ningún vertice superior a v_{k-1} entre estos puntos.

Por tanto podemos concluir que el problema de todos los pares de caminos óptimos en un grafo es equivalente al problema matricial del cálculo de costos entre vértices, cuyo tiempo de ejecución es de orden $O(n^3)$.

Capítulo 3

Nuevos algoritmos. El método de Chan

Vamos a describir un algoritmo con un tiempo de ejecución del orden de $O(n^3/logn)$ para el problema de todos los pares de caminos óptimos en un grafo.

Consideraremos el caso general donde la entrada es un grafo dirigido con costos reales arbitrarios. El problema consiste en calcular la distancia del camino más corto entre cada par de vértices y una representación de estos caminos más cortos, pero en un tiempo menor al planteado por Floyd-Warshall, que es del orden de $O(n^3)$ para un grafo con n vértices.

Durante mucho tiempo el mejor resultado conocido tenía un tiempo de ejecución del orden de $O(n^2logn+mn)$, donde m indicaba el número de aristas del grafo, pero recientemente se había logrado rebajar los límites de tiempo a $O(n^2loglogn+mn)$ y $O(mnlog\alpha(m,n))$, siendo α una función real que dependía exclusivamente de m y de n, para grafos dirigidos y no dirigidos respectivamente pero utilizando técnicas muy complicadas.

Un conjunto de estudios en la última década se ha centrado en el caso de grafos con costos enteros pequeños y, en particular, para grafos sin costos, donde se han podido desarrollar un conjunto de algoritmos subcúbicos utilizando distintos métodos de multiplicación de matrices sobre anillos. Actualmente, los mejores algoritmos para el problema de todos los pares de caminos óptimos en un grafo funcionan en tiempos $O(n^{2,376}M)$ y $O(n^{2,575}M^{0,681})$, donde M es el máximo costo de las aristas en valor absoluto. Estos tiempos de funcionamiento son subcúbicos sólamente cuando $M << n^{0,634}$. No se sabe si estos métodos de multiplicación de matrices pueden ayudar a generar algoritmos para resolver el problema de todos los pares de caminos óptimos en un grafo en el caso de pesos reales, o en este caso, costos enteros del conjunto $\{0,1,...,n\}$. Incluso si la respuesta a esta pregunta fuese afirmativa, los algoritmos que implican la multiplicación usual de matrices no necesariamente son atractivos desde un punto de vista práctico. Feder y Motwani describieron un algoritmo cuyo tiempo de ejecución era del orden $O(n^3/logn)$ que evita la multiplicación usual de matrices, pero dicho algoritmo sólamente funciona para grafos no dirigidos sin costos.

Nuestro resultado sólo representa una pequeña mejora sobre algoritmos anteriores levemente subcúbicos en el caso general de costos reales. Además es interesante porque es muy simple conceptualmete y enfrenta el problema con un enfoque notablemente distinto. Nuestro enfoque está inspirado en la técnica "divide y vencerás", que se suele utilizar en geometría computacional, y que consiste en dividir el problema original en varios problemas más sencillos de resolver y así poder solucionar el problema original.

3.1. Un subproblema geométrico

Empezamos planteando un problema de geometría computacional. El problema consiste en encontrar todos los pares dominantes entre un conjunto de puntos rojos fijos y un conjunto de puntos azules fijos, pero, a diferencia del análisis tradicional en geometría computacional, queremos estudiar el caso en el que la dimensión no es constante, es decir, para una dimensión cualquiera.

Definición. Sea $p = (p_1, ..., p_d) \in \mathbb{R}^d$ un punto rojo y sea $q = (q_1, ..., q_d) \in \mathbb{R}^d$ un punto azul. Se dice que forman un **par dominante** si $p_k \leq q_k$ para todo $k = 1, ..., d$.

Lema 1. *Dados n puntos azules o rojos en $\mathbb{R}^d$, podemos dar todos los pares dominantes en un tiempo $O(c_\varepsilon^d n^{1+\varepsilon} + k)$ para cualquier constante $\varepsilon \in (0,1)$, donde $c_\varepsilon := \frac{2^\varepsilon}{2^\varepsilon - 1}$, y k es el número de pares obtenidos.*

Demostración. Vamos a describir un algoritmo basado en la técnica "divide y vencerás".

Si $n = 1$ sólo hay un punto, por tanto paramos.

Si $d = 0$, podemos interpretar que todos los puntos son el mismo, pues están contenidos en el mismo espacio de dimensión 0, por tanto todos los azules dominan a todos los rojos pues la condición de par dominante no es una desigualdad estricta.

Si $n \neq 1$ y $d \neq 0$, calculamos la mediana z de las coordenadas d-ésimas de todos los puntos de tal forma que podemos definir:

- $P_{left,blue} = \{$puntos azules cuya coordenada d-ésima es menor o igual que la mediana $z\}$
- $P_{left,red} = \{$puntos rojos cuya coordenada d-ésima es menor o igual que la mediana $z\}$
- $P_{right,blue} = \{$puntos azules cuya coordenada d-ésima es mayor o igual que la mediana $z\}$
- $P_{right,red} = \{$puntos rojos cuya coordenada d-ésima es mayor o igual que la mediana $z\}$

Así los puntos azules de $P_{right,blue}$ dominan a los rojos de $P_{left,red}$ en la coordenada i-ésima.

Resolvemos recursivamente el problema con los puntos de los conjuntos $P_{left,blue} \cup P_{left,red}$ y $P_{right,blue} \cup P_{right,red}$

Excluyendo el coste de salida, el tiempo de ejecución puede acotarse mediante la siguiente recurrencia:

$$T_d(n) \leq 2T_d(\frac{n}{2}) + T_{d-1}(n) + O(n)$$

con $T_d(1) = O(1)$ y $T_0(n) = O(n)$. Esta cota recursiva para el tiempo de ejecución surge de lo siguiente:

- $2T_d(\frac{n}{2})$ es el coste de realizar el mismo procedimiento con los 2 conjuntos $P_{left,blue} \cup P_{left,red}$ y $P_{right,blue} \cup P_{right,red}$, los cuales tienen exactamente la mitad de puntos que los que teníamos al principio, pues los hemos separado en distintos conjuntos en función de si eran mayores o menores que la mediana z de la coordenada d-ésima.
- $T_{d-1}(\mid P_{right,blue} \cup P_{left,red} \mid)$ es el coste de realizar el mismo procedimiento en la coordenada $(d-1)$-ésima del conjunto de puntos $P_{right,blue} \cup P_{left,red}$, pues los puntos de ese conjunto son aquellos tales que la coordenada d-ésima de los azules es mayor o igual que la coordenada d-ésima de los rojos, por ello ahora realizamos el mismo procedimiento pero mirando en la coordenada $(d-1)$-ésima. Este tiempo es menor o igual que $T_{d-1}(n)$.
- $O(n)$ es el coste de calcular la mediana.

El coste de salida está limitado por $O(k)$, pues cada par se da una vez. Por inducción sobre d se puede establecer que $T_d(n) = O(nlog^d n)$, dando un algoritmo de tiempo $O(nlog^d n + k)$.

Por otro lado, vamos a hacer un análisis alternativo de la recurrencia que es mejor para valores no constantes de d. Hacemos un cambio de variable: fijamos un parámetro b y definimos:

$$T'(N) := \max_{b^d n \leq N} \{T_d(n)\}$$

y nos queda

$$T'(N) \leq 2T'(\frac{N}{2}) + T'(\frac{N}{b}) + cN$$

para alguna constante c.

Veamos como hemos llegado a esta conclusión:

$$\max_{b^d n \leq N} \{T_d(n)\} \leq \max_{b^d n \leq N} \{2T_d(\frac{n}{2}) + T_{d-1}(n) + O(n)\}$$

y como

$$\max_{b^d n \leq N} \{2T_d(\frac{n}{2}) + T_{d-1}(n) + O(n)\} \leq \max_{b^d n \leq N} \{2T_d(\frac{n}{2})\} + \max_{b^d n \leq N} \{T_{d-1}(n)\} + \max_{b^d n \leq N} \{O(n)\}$$

entonces

$$\max_{b^d n \leq N} \{T_d(n)\} \leq \max_{b^d n \leq N} \{2T_d(\frac{n}{2})\} + \max_{b^d n \leq N} \{T_{d-1}(n)\} + \max_{b^d n \leq N} \{O(n)\}.$$

Además

$$\max_{b^d n \leq N} \{O(n)\} = O(\frac{N}{b^d}) \leq cN$$

para alguna constante c.Por tanto

$$\max_{b^d n \leq N} \{T_d(n)\} \leq \max_{b^d n \leq N} \{2T_d(\frac{n}{2})\} + \max_{b^d n \leq N} \{T_{d-1}(n)\} + cN =$$

$$= 2 \max_{b^d n \leq N} \{T_d(\frac{n}{2})\} + \max_{b^d n \leq N} \{T_{d-1}(n)\} + cN =$$

$$= 2 \max_{\frac{b^d n}{2} \leq \frac{N}{2}} \{T_d(\frac{n}{2})\} + \max_{b^d n \leq N} \{T_{d-1}(n)\} + cN =$$

$$= 2 \max_{\frac{b^d n}{2} \leq \frac{N}{2}} \{T_d(\frac{n}{2})\} + \max_{b^{d-1} n \leq \frac{N}{b}} \{T_{d-1}(n)\} + cN$$

Así nos queda

$$T^{'}(N) \leq 2T^{'}(\frac{N}{2}) + T^{'}(\frac{N}{b}) + cN$$

para alguna constante c.

Ahora queremos ver que

$$T^{'}(N) \leq c^{'}[N^{1+\varepsilon} - N],$$

para algún $\varepsilon \in (0,1)$ fijo.Si aplicamos esta condición a la desigualdad anterior nos queda:

$$T^{'}(N) \leq 2c^{'}[(\frac{N}{2})^{1+\varepsilon} - \frac{N}{2}] + c^{'}[(\frac{N}{b})^{1+\varepsilon} - \frac{N}{b}] + cN$$

y así la tesis se cumple siempre que la constante $c^{'}$ sea lo suficientemente grande y se cumpla la igualdad

$$\frac{2}{2^{1+\varepsilon}} + \frac{1}{b^{1+\varepsilon}} = 1$$

la cual se cumple tomando

$$b = c_{\varepsilon}^{\frac{1}{1+\varepsilon}} = (\frac{2^{\varepsilon}}{2^{\varepsilon} - 1})^{\frac{1}{1+\varepsilon}}.$$

Así, simplificando y sustituyendo este valor de b en la desigualdad anterior, tenemos:

$$T^{'}(N) \leq 2c^{'}[(\frac{N}{2})^{1+\varepsilon} - \frac{N}{2}] + c^{'}[(\frac{N}{b})^{1+\varepsilon} - \frac{N}{b}] + cN =$$

$$= 2c^{'}[\frac{N}{2}(\frac{N}{2})^{\varepsilon} - \frac{N}{2}] + c^{'}[NN^{\varepsilon}(\frac{1}{b})^{1+\varepsilon} - \frac{N}{b}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon} - 1] + Nc^{'}[N^{\varepsilon}(\frac{1}{b})^{1+\varepsilon} - \frac{1}{b}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon} - 1 + N^{\varepsilon}(\frac{1}{b})^{1+\varepsilon} - \frac{1}{b}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon} - 1 + N^{\varepsilon}(\frac{2^{\varepsilon}-1}{2^{\varepsilon}}) - (\frac{2^{\varepsilon}-1}{2^{\varepsilon}})^{\frac{1}{1+\varepsilon}}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon} - 1 + N^{\varepsilon}(\frac{2^{\varepsilon}-1}{2^{\varepsilon}}) - (\frac{(2^{\varepsilon}-1)2}{2^{\varepsilon}2})^{\frac{1}{1+\varepsilon}}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon}(1+2^{\varepsilon}-1) - 1 - (\frac{(2^{\varepsilon}-1)2}{2^{\varepsilon}2})^{\frac{1}{1+\varepsilon}}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon}(1+2^{\varepsilon}-1) - 1 - \frac{1}{2}(2^{\varepsilon+1}-2)^{\frac{1}{1+\varepsilon}}] + cN =$$

$$= Nc^{'}[(\frac{N}{2})^{\varepsilon}2^{\varepsilon} - 1 - \frac{1}{2}(2^{\varepsilon+1}-2)^{\frac{1}{1+\varepsilon}}] + cN =$$

$$= Nc^{'}[N^{\varepsilon} - 1 - \frac{1}{2}(2^{\varepsilon+1}-2)^{\frac{1}{1+\varepsilon}}] + cN =$$

$$= Nc^{'}[N^{\varepsilon} - (1 + \frac{1}{2}(2^{\varepsilon+1}-2)^{\frac{1}{1+\varepsilon}})] + cN$$

Sea ahora la función $f(t) = \frac{1}{2}(2^{t+1}-2)^{\frac{1}{1+t}}$. Esta función es positiva y estrictamente creciente en $(0,1)$. Como ε es fijo, existe un $\delta > 0$ fijo tal que $\delta = f(\varepsilon)$. De esta forma podemos afirmar que $1 + \frac{1}{2}(2^{\varepsilon+1}-2)^{\frac{1}{1+\varepsilon}} = 1 + \delta$. Así nos queda:

$$T^{'}(N) \leq Nc^{'}[N^{\varepsilon} - (1 + \frac{1}{2}(2^{\varepsilon+1}-2)^{\frac{1}{1+\varepsilon}})] + cN = Nc^{'}[N^{\varepsilon}] - Nc^{'}(1+\delta) + cN =$$

$$= Nc^{'}[N^{\varepsilon}] - N(c^{'}\delta + c^{'} - c)$$

Tomamos $c^{'}$ tal que $c^{'}\delta + c^{'} - c \geq c^{'}$ que es lo mismo que $c^{'} \geq \frac{c}{\delta}$. Así concluimos que:

$$T^{'}(N) \leq Nc^{'}[N^{\varepsilon}] - N(c^{'}\delta + c^{'} - c) \leq Nc^{'}[N^{\varepsilon}] - Nc^{'} = c^{'}[N^{1+\varepsilon} - N]$$

lo cual implica que $T^{'}(N) = O(N^{\varepsilon+1}) \Rightarrow T_d(n) = O((b^d n)^{\varepsilon+1}) = O(c_{\varepsilon}^d n^{1+\varepsilon})$ y el lema queda probado, ya que como el coste de salida está limitado por $O(k)$, el problema puede resolverse en un tiempo $O(c_{\varepsilon}^d n^{1+\varepsilon} + k)$.

□

3.2. El algoritmo de todos los pares de caminos óptimos en un grafo

En esta sección presentaremos el nuevo algoritmo para el problema de todos los pares de caminos óptimos en un grafo. Para ello simplificaremos el problema reduciendolo al cálculo del producto de dos matrices en el semianillo cerrado S_2 definido como sigue: Dadas dos matrices $A = \langle a_{i,k} \rangle$, $n \times d$,$i = 1,...,n$, $k = 1,...,d$ y $B = \langle b_{k,j} \rangle$, $d \times m$, $k = 1,...,d$, $j = 1,...,m$, el resultado de esta multiplicación se define como la matriz $C = \langle c_{i,j} \rangle, i = 1,...,n, j = 1,...,m$ con $c_{i,j} = \text{mín}_k\{a_{i,k} + b_{k,j}\}$. Para simplificar la notación, suponemos que el término mínimo de la expresión $\text{mín}_k\{a_{i,k} + b_{k,j}\}$ es único. Veremos que nuestro algoritmo identifica automáticamente el índice k donde se logra el mínimo para cada $c_{i,j}$. Esta propiedad es necesaria porque así el algoritmo también puede determinar el camino más corto.

En el siguiente lema observamos la conexión clave entre los anteriores problemas geométricos y el producto de matrices rectangulares en el semianillo cerrado S_2.

Lema 2. *Podemos calcular el producto de una matriz A $n \times d$ y de una matriz B $d \times n$ en el semianillo cerrado S_2 en un tiempo $O(dc_{\varepsilon}^d n^{1+\varepsilon} + n^2)$.*

Demostración. El esquema del algoritmo es simple: Para cada $k = 1,...,d$ calculamos el conjunto de pares $X_k = \{(i,j) \mid \text{para todo } k' = 1,...,d,\ a_{i,k} + b_{k,j} \leq a_{i,k'} + b_{k',j}\}$. A continuación tomamos $c_{i,j} = a_{i,k} + b_{k,j}$ para cada $(i,j) \in X_k$.

Seguimos con una observación obvia: $a_{i,k} + b_{k,j} \leq a_{i,k'} + b_{k',j}$ es equivalente a $a_{i,k} - a_{i,k'} \leq b_{k',j} - b_{k,j}$. Así nos queda $X_k = \{(i,j) \mid \text{para todo } k' = 1,...,d,\ a_{i,k} - a_{i,k'} \leq b_{k',j} - b_{k,j}\}$, es decir, el conjunto X_k es el conjunto de los pares (i,j) que cumplen la ecuaciones siguientes:

$$\begin{cases} a_{i,k} - a_{i,1} \leq b_{1,j} - b_{k,j} \\ a_{i,k} - a_{i,2} \leq b_{2,j} - b_{k,j} \\ \vdots \\ a_{i,k} - a_{i,k} \leq b_{k,j} - b_{k,j} \\ \vdots \\ a_{i,k} - a_{i,d} \leq b_{d,j} - b_{k,j} \end{cases}$$

Ahora hacemos otra observación: Calcular X_k para un k fijo, equivale a calcular todos los pares dominantes entre dos conjuntos de puntos d-dimensionales, los cuales son $\{(a_{i,k} - a_{i,1}, a_{i,k} - a_{i,2}, ..., a_{i,k} - a_{i,k}, ..., a_{i,k} - a_{i,d})$ para todo $i = 1,...,n\}$ y $\{(b_{1,j} - b_{k,j}, b_{2,j} - b_{k,j}, ..., b_{k,j} - b_{k,j}, ..., b_{d,j} - b_{k,j})$ para todo $j = 1,...,n\}$. En realidad, la dimensión es $d-1$ ya que todas las coordenadas k-ésimas de dichos conjuntos son 0, pues esas coordenadas son $a_{i,k} - a_{i,k}$ y $b_{k,j} - b_{k,j}$ respectivamente.

El número de pares dominantes es el número de elementos de X_k pues los pares (i,j) del conjunto X_k cumplen la condición de dominancia. Podríamos llegar a pensar que hay pares dominantes de puntos dentro de cada uno de los dos conjuntos, pero nos interesa saber sólamente los pares dominantes que se forman con un punto de cada uno de los dos conjuntos que tenemos. Así como en el Lema 2 podíamos diferenciar los conjuntos de puntos como rojos o azules, ahora tambien podemos diferenciar los dos conjuntos y de esta forma podemos aplicar el Lema 2, el cual nos dice que para realizar este cálculo se requiere un tiempo $O(c_\varepsilon^d n^{\varepsilon+1} + \mid X_k \mid)$.

Veamos ahora que $\sum_{k=1}^{d} \mid X_k \mid = n^2$: La matriz C es $n \times n$, por tanto tiene n^2 elementos. Cada elemento $c_{i,j}$ corresponderá a unas coordenadas (i,j) que estarán en un conjunto X_k para algún k y no en ningún otro, pues ya hemos dicho que podemos suponer que el elemento mínimo es único. Así si contamos todos los elementos de los conjuntos X_k para todo k tenemos n^2 elementos que son las coordenadas correspondientes a cada elemento de la matriz C.

Así, como este cálculo hay que realizarlo con cada $k = 1,...,d$ y $\sum_{k=1}^{d} \mid X_k \mid = n^2$, entonces podemos concluir que calcular el producto de una matriz A $n \times d$ y de una matriz B $d \times n$ en el semianillo cerado S_2 requiere un tiempo $O(dc_\varepsilon^d n^{1+\varepsilon} + n^2)$.

□

Para obtener un algoritmo subcúbico para el producto de matrices cuadradas en el semianillo cerrado S_2, y consecuentemente para el problema de todos los pares de caminos óptimos en un grafo, lo único que nos queda es elegir un valor apropiado para el parámetro dimensional d.

Teorema 3.1. *Podemos calcular el producto de dos matrices cuadradas $n \times n$ en el semianillo cerrado S_2 en un tiempo $O(\frac{n^3}{\log n})$*

Demostración. Sean A y B dos matrices cuadradas $n \times n$. Dividimos la primera matriz en $\frac{n}{d}$ matrices $A_1,...,A_{\frac{n}{d}}$ de dimensión $n \times d$ y la segunda matriz en $\frac{n}{d}$ matrices $B_1,...,B_{\frac{n}{d}}$ de dimensión $d \times n$. Calculamos el producto de A_l y B_l en el semianillo cerrado S_2 para cada $l = 1,...,\frac{n}{d}$ y devolvemos una matriz $n \times n$ cuyas coordenadas son la mínimas de entre todas las correspondientes coordenadas de esas matrices $\frac{n}{d}$ de dimensión $n \times n$. Veamos esto más detalladamente:

Primero calculamos el producto de A y B en el semianillo cerrado S_2 y luego veremos que el resultado es el mismo que si hacemos el algoritmo descrito previamente:

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,n} \end{pmatrix} = \begin{pmatrix} \min_{1\le i\le n}\{a_{1,i}+b_{i,1}\} & \cdots & \min_{1\le i\le n}\{a_{1,i}+b_{i,n}\} \\ \vdots & \ddots & \vdots \\ \min_{1\le i\le n}\{a_{n,i}+b_{i,1}\} & \cdots & \min_{1\le i\le n}\{a_{n,i}+b_{i,n}\} \end{pmatrix}$$

Ahora calculamos el producto de las distintas matrices A_l y B_l en el semianillo cerrado S_2:

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,d} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,d} \end{pmatrix} \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{d,1} & \cdots & b_{d,n} \end{pmatrix} = \begin{pmatrix} \min_{1\le i\le d}\{a_{1,i}+b_{i,1}\} & \cdots & \min_{1\le i\le d}\{a_{1,i}+b_{i,n}\} \\ \vdots & \ddots & \vdots \\ \min_{1\le i\le d}\{a_{n,i}+b_{i,1}\} & \cdots & \min_{1\le i\le d}\{a_{n,i}+b_{i,n}\} \end{pmatrix}$$

$$\vdots$$

$$\begin{pmatrix} a_{1,n-d+1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,n-d+1} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} b_{n-d+1,1} & \cdots & b_{n-d+1,n} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,n} \end{pmatrix} =$$

$$= \begin{pmatrix} \min_{n-d+1\le i\le n}\{a_{1,i}+b_{i,1}\} & \cdots & \min_{n-d+1\le i\le n}\{a_{1,i}+b_{i,n}\} \\ \vdots & \ddots & \vdots \\ \min_{n-d+1\le i\le n}\{a_{n,i}+b_{i,1}\} & \cdots & \min_{n-d+1\le i\le n}\{a_{n,i}+b_{i,n}\} \end{pmatrix}$$

y como

$$\begin{pmatrix} \min\{\min_{1\le i\le d}\{a_{1,i}+b_{i,1}\},...,\min_{n-d+1\le i\le n}\{a_{1,i}+b_{i,1}\}\} & \cdots & \cdots \\ \vdots & \vdots & \vdots \\ \min\{\min_{1\le i\le d}\{a_{n,i}+b_{i,1}\},...,\min_{n-d+1\le i\le n}\{a_{n,i}+b_{i,1}\}\} & \cdots & \cdots \end{pmatrix}$$

$$\begin{pmatrix} \cdots & \cdots & \min\{\min_{1\le i\le d}\{a_{1,i}+b_{i,n}\},...,\min_{n-d+1\le i\le n}\{a_{1,i}+b_{i,n}\}\} \\ \vdots & \vdots & \vdots \\ \cdots & \cdots & \min\{\min_{1\le i\le d}\{a_{n,i}+b_{i,n}\},...,\min_{n-d+1\le i\le n}\{a_{n,i}+b_{i,n}\}\} \end{pmatrix} =$$

$$= \begin{pmatrix} \min_{1\le i\le n}\{a_{1,i}+b_{i,1}\} & \cdots & \min_{1\le i\le n}\{a_{1,i}+b_{i,n}\} \\ \vdots & \ddots & \vdots \\ \min_{1\le i\le n}\{a_{n,i}+b_{i,1}\} & \cdots & \min_{1\le i\le n}\{a_{n,i}+b_{i,n}\} \end{pmatrix}$$

se cumple lo que queriamos ver. Así el tiempo total es como mínimo el tiempo límite del Lema 3 multiplicado por $\frac{n}{d}$, es decir,

$$O(c_\varepsilon^d n^{2+\varepsilon} + \frac{n^3}{d})$$

El teorema queda probado eligiendo d como $logn$ por una constante C lo suficientemente pequeña dependiendo de $\varepsilon \in (0,1)$. Así, tomando $d = Clogn$, veamos que se cumple la desigualdad

$$c_\varepsilon^{Clogn} n^{2+\varepsilon} < \frac{n^3}{Clogn}$$

Tomando logaritmos en ambos lados nos queda

$$Clognlogc_\varepsilon + (2+\varepsilon)logn < 3logn - log(Clogn)$$

$$(Clogc_\varepsilon)logn + (2+\varepsilon)logn + log(Clogn) < 3logn$$

$$(Clogc_\varepsilon)logn + (2+\varepsilon)logn + logC + log(logn) < 3logn$$

lo cual es trivial debido a las constantes que hemos tomado. Finalmente podemos afirmar que

$$O(c_\varepsilon^d n^{2+\varepsilon} + \frac{n^3}{d}) = O(c_\varepsilon^{Clogn} n^{2+\varepsilon} + \frac{n^3}{Clogn}) = O(\frac{n^3}{logn})$$

A modo de ejemplo, podemos tomar $\varepsilon \sim 0{,}38$, lo que implica que $c_\varepsilon \sim 4{,}32$, y $d \sim 0{,}42 logn$ para minimizar el factor constante en el término dominante.

□

Corolario 1. *Podemos resolver el problema de todos los pares de caminos óptimos en un grafo en un tiempo* $O(\frac{n^3}{logn})$.

Este corolario es la consecuencia de que el problema de todos los pares de caminos óptimos en un grafo es equivalente al cálculo del producto de dos matrices en el semianillo cerrado S_2, lo cual hemos visto en el Capítulo 2.

3.3. Discusión

Aunque hemos tomado un enfoque geométrico, el algoritmo resultante comparte algunas similitudes con algoritmos anteriores como la comparación de valores de los costos de los distintos caminos entre los mismos vértices y tomando el mínimo, el paso a través de cada índice k y el cálculo del mismo conjunto X_k de pares de índices.

El tiempo $O(\frac{n^3}{logn})$ parece bastante natural. En cualquier caso, reducir el tiempo de funcionamiento más allá de un factor logarítmico será difícil, y no merece la pena hacer todos los cálculos necesarios para reducirlo más, pues la precisión extra no será significativa. Incluso para multiplicar matrices *booleanas* el mejor algoritmo conocido sin técnicas algebraicas se ejecuta en un tiempo $O(n^3/logn)$ y no ha podido mejorarse durante más de tres décadas.

Aunque nuestro algoritmo es muy sencillo para implementarlo, el interés principal es teórico. Algunos experimentos preliminares parecen indicar que incluso para un n alrededor de 1000 (donde el tamaño del grafo es del orden de un millón), la mejor elección de d sigue siendo 2. Notar que el factor $logn$ sólo es relevante cuando el tamaño de la entrada es muy grande.

Finalmente remarcamos que Fredman encontró una forma no algorítmica de resolver el problema todos los pares de caminos óptimos en un grafo utilizando $O(n^{2,5})$ operaciones. Estas operaciones consisten en comparar los costos de los diferentes caminos entre el mismo par de vértices (i, j). Más tarde utilizó esto para su propio algoritmo subcúbico. Todavía se sigue estudiando este procedimiento para intentar acotar el número de comparaciones necesarias de la mejor forma posible.

Bibliografía

[1] ALFRED V. AHO, JOHN E. HOPCROFT, JEFFREY D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] TIMOTHY M. CHAN, *All-Pairs Shortest Paths with Real Weights in $O(n^3/logn)$ Time*, 10 de Mayo, 2005.

[3] THOMAS H. CORMEN, CHARLES E. LEISERSON, RONALD L. RIVEST, *Introduction to Algorithms*, 16ª impresión, McGraw-Hill, 1996.

[4] ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS, *Geometría Computacional*, Universidad Politécnica de Madrid, `http://www.dma.fi.upm.es/recursos/aplicaciones/geometria_computacional_y_grafos/web/shortest_path/html/Geometria_Computacional.htm#Top`.

[5] INSTINTO LÓGICO, *Complejidad Computacional*, disponible en `http://instintologico.com/complejidad-computacional/`.

[6] JOEMMANUEL PONCE G., *Introducción a los Algoritmos. Complejidad Computacional*, disponible en `https://es.slideshare.net/joemmanuel/complejidad-computacional`.