Técnicas Fundamentales

Programación Dinámica II

Segundo semestre 2022

IIC2283

Prof. Nicolás Van Sint Jan

Recordatorio: Programación dinámica

Al igual que dividir para conquistar, la técnica de **programación dinámica** resuelve un problema dividiéndolo en sub-problemas más pequeños.

Pero a diferencia de dividir para conquistar, en este caso se espera que **los sub-problemas estén traslapados**.

De esta forma se reduce el número de sub-problemas a resolver, de hecho se espera que este número sea pequeño (al menos polinomial).

Recordatorio: Contando el número de caminos en un grafo

Sea $G = (V, E)$ un **grafo dirigido**.

Recordar que una secuencia v_1, ..., v_ℓ de elementos en N es un **camino** en G si:

1. $\ell \geq 2$
2. $(v_i, v_{i+1}) \in E$ para cada $i \in \{1, \ldots, \ell - 1\}$

Decimos que un camino v_1, ..., v_ℓ va desde v_1 a v_ℓ, y definimos su largo como (ℓ - 1), vale decir, el número de aristas en el camino.

Ejemplo

$$\begin{tikzcd}
a \arrow[dr] & b \arrow[l] & e \arrow[l] \arrow[d] \\
& d \arrow[ur] & f \arrow[l]
\end{tikzcd}$$

b, a, d, e, f

Recordatorio: Contando el número de caminos en un grafo

Dado un grafo $G = (V, E)$, un par de nodos v_i, v_f en V y un número ℓ, queremos desarrollar un algoritmo que cuente el **número de caminos** desde v_i a v_f en G cuyo largo es igual a ℓ

Suponemos que $V = \{1, \ldots, n\}$, $1 \leq \ell \leq n$ y representamos G a través de su **matriz de adyacencia** M tal que:

Si $(i, j) \in E$, entonces $M[i, j] = 1$, en caso contrario $M[i, j] = 0$.

Ejemplo

$$
\begin{tikzcd}
1 \arrow[dr] & 2 \arrow[l] & 4 \arrow[l] \arrow[d] \\
 & 3 \arrow[ur] & 5 \arrow[l]
\end{tikzcd}
\qquad
M = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0
\end{bmatrix}
$$

Recordatorio: Una primera definición de **ContarCaminos**

Queremos entonces definir la función **ContarCaminos**$(M,\ v_i,\ v_f,\ \ell)$.

ContarCaminos$(M[1 \dots n][1 \dots n],\ v_i,\ v_f,\ \ell)$
    if $\ell = 1$ **then return** $M[v_i, v_f]$
    else
        $aux := 0$
        for $v_j := 1$ **to** n **do**
            $aux \mathrel{+}= M[v_i, v_j] \cdot$ **ContarCaminos**$(M,\ v_j,\ v_f,\ \ell - 1)$
        return aux

Observe que usamos la notación $C[1 \dots m][1 \dots n]$ para indicar que la matriz C tiene m filas y n columnas.

¿Se puede mejorar este algoritmo?

Recordatorio: Una segunda definición de **ContarCaminos**

Podemos reducir el número de llamadas recursivas:

ContarCaminos($M[1 \ldots n][1 \ldots n]$, v_i, v_f, ℓ)
    if $\ell = 1$ **then return** $M[v_i, v_f]$
    else
        $aux := 0$
        for $v_j := 1$ **to** n **do**
            if $M[v_i, v_j] = 1$ **then**
                aux += **ContarCaminos**(M, v_j, v_f, $\ell - 1$)
        return aux

¿Hay algún **problema** con este algoritmo?

Outline

Programación dinámica: Grafos

Programación dinámica: Palabras

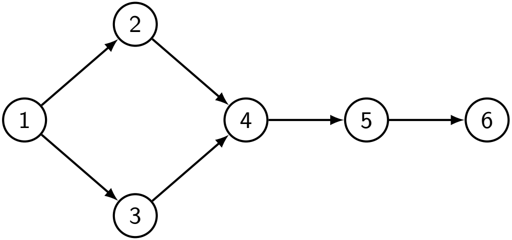
Outline

Programación dinámica: Grafos

Programación dinámica: Palabras

Llamadas repetidas en **ContarCaminos**

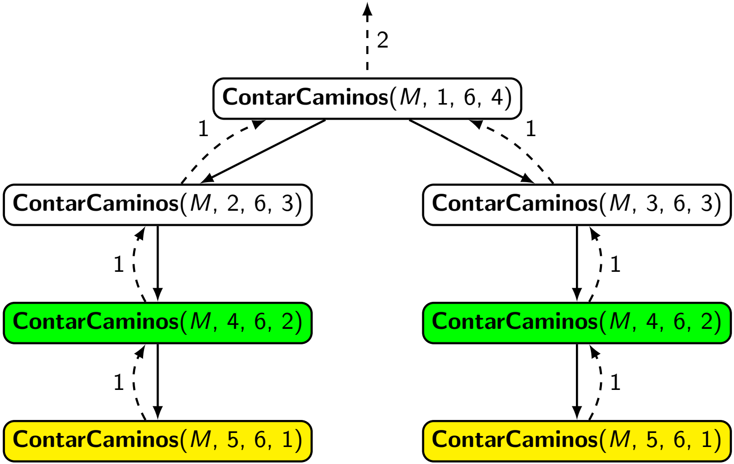
Considere el siguiente grafo G (representado por una matriz M):



Suponga que queremos contar el número de caminos en G desde el nodo 1 al nodo 6 y cuyo largo sea 4

Llamadas repetidas en **ContarCaminos**

Tenemos las siguientes llamadas recursivas:



Llamadas repetidas en **ContarCaminos**

Ejercicio

Demuestre que en el peor caso se deben realizar $\dfrac{n^\ell - 1}{n - 1}$ llamadas al procedimiento **ContarCaminos**, suponiendo que la entrada es $M[1 \dots n][1 \dots n]$, v_i, v_f, ℓ con $n \geq 2$.

- ¿Para qué grafos obtenemos el peor caso?

El algoritmo realiza **un número exponencial de llamadas repetidas**. Dado que sólo podemos tener $n^2 \cdot \ell$ llamadas distintas en la ejecución de **ContarCaminos**$(M[1 \dots n][1 \dots n], v_i, v_f, \ell)$.

Puesto que tenemos **llamadas traslapadas** y un **espacio pequeño de sub-problemas** este es un problema adecuado para **programación dinámica**.

Una tercera definición de **ContarCaminos**

Queremos calcular el número de caminos de largo ℓ desde un nodo v_i a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)

Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** $H_1, \ldots, H_\ell$ tales que:

$$H_k[v_i, v_j] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de largo } k$$

La secuencia $H_1, \ldots, H_\ell$ se puede definir recursivamente de la siguiente forma:

1. $H_1 = M$
2. $H_{k+1} = M \cdot H_k$ para $k \in \{1, \ldots, \ell - 1\}$

¿Dónde estaría entonces la respuesta a la pregunta original?

R: en $H_\ell[v_i, v_f]$

Una tercera definición de **ContarCaminos**

La implementación recursiva de la idea descrita:

ContarTodosCaminos($M[1 \ldots n][1 \ldots n]$, ℓ)
    if $\ell = 1$ **then return** M
    else
        $H :=$ **ContarTodosCaminos**(M, $\ell - 1$)
        return $M \cdot H$

ContarCaminos($M[1 \ldots n][1 \ldots n]$, v_i, v_f, ℓ)
    $H :=$ **ContarTodosCaminos**(M, ℓ)
    return $H[v_i, v_f]$

La complejidad de **ContarCaminos**

Ejercicio

Demuestre que **ContarCaminos** en el peor caso es $O(\ell \cdot n^3)$

- ¿Cuál es el tamaño de la entrada para **ContarCaminos**?
- ¿Qué operaciones básicas debemos considerar en el análisis de **ContarCaminos**?
- Es necesario utilizar un algoritmo para multiplicar mátrices para obtener este resultado

Outline

Programación dinámica: un segundo ingrediente

En general, programación dinámica es usada para resolver **problemas de optimización**.

Para que un problema de optimización pueda ser resuelto con esta técnica se debe cumplir el siguiente **principio de optimalidad para los sub-problemas:**

Una solución óptima para un problema contiene soluciones óptimas para sus sub-problemas.

Vamos a ver un ejemplo de este principio que enfatiza otra característica de programación dinámica: en general los problemas deben ser resueltos de forma **bottom-up**.

Midiendo la distancia entre dos palabras

Sea $\Sigma = \{a, b, \ldots, z\}$ el alfabeto español, el cual contiene 27 símbolos.

Vamos a considerar las palabras $w \in \Sigma^*$ (strings) sobre el alfabeto Σ.

Vamos a utilizar la **distancia de Levenshtein** para medir cuán similares son dos palabras.

- Esta es una de las medidas de similitud de palabras más populares por lo que usualmente es llamada **edit distance**.

Dadas dos palabras $w_1, w_2 \in \Sigma^*$, utilizamos la notación $\mathrm{ed}(w_1, w_2)$ para la edit distance entre w_1 y w_2

- Tenemos que $\mathrm{ed} : \Sigma^* \times \Sigma^* \to \mathbb{N}$

Tres operadores sobre palabras

Sea $w \in \Sigma^*$ con $w = a_1 \cdots a_n$ y $n \geq 0$

- Si $n = 0$ entonces $w = \varepsilon$

Para $i \in \{1, \ldots, n\}$ y $b \in \Sigma$ tenemos que:

$$\begin{array}{rcl} \text{eliminar}(w, i) & = & a_1 \cdots a_{i-1} a_{i+1} \cdots a_n \\ \text{agregar}(w, i, b) & = & a_1 \cdots a_i b a_{i+1} \cdots a_n \\ \text{cambiar}(w, i, b) & = & a_1 \cdots a_{i-1} b a_{i+1} \cdots a_n \end{array}$$

Además, tenemos que agregar$(w, 0, b) = bw$.

Ejemplo

$$\begin{array}{rclcrcl} \text{eliminar}(\texttt{hola},1) & = & \texttt{ola} & & \text{eliminar}(\texttt{hola},3) & = & \texttt{hoa} \\ \text{agregar}(\texttt{hola},0,\text{y}) & = & \texttt{yhola} & & \text{agregar}(\texttt{hola},3,\text{z}) & = & \texttt{holza} \\ \text{cambiar}(\texttt{hola},2,\texttt{x}) & = & \texttt{hxla} & & & & \end{array}$$

La distancia de Levenshtein

Definición

Dadas palabras $w_1, w_2 \in \Sigma^*$, definimos la **edit-distance** entre w_1 y w_2: $\text{ed}(w_1, w_2)$ como el menor número de operaciones eliminar, agregar y cambiar que aplicadas desde w_1 generan w_2

Ejemplo

Tenemos que $\text{ed}(\texttt{casa}, \texttt{asado}) = 3$ puesto que:

$$\begin{aligned}
\text{agregar}(\texttt{casa}, 4, \texttt{d}) &= \texttt{casad} \\
\text{eliminar}(\texttt{casad}, 1) &= \texttt{asad} \\
\text{agregar}(\texttt{asad}, 4, \texttt{o}) &= \texttt{asado}
\end{aligned}$$

La distancia de Levenshtein

Ejercicios

1. Demuestre que $\mathrm{ed}(w_1, w_2) \leq |w_1| + |w_2|$

2. Demuestre que ed es una función de distancia, vale decir, muestre lo siguiente:

   2.1 $\mathrm{ed}(w_1, w_2) = 0$ si y sólo si $w_1 = w_2$
   2.2 $\mathrm{ed}(w_1, w_2) = \mathrm{ed}(w_2, w_1)$
   2.3 $\mathrm{ed}(w_1, w_2) \leq \mathrm{ed}(w_1, w_3) + \mathrm{ed}(w_3, w_2)$

3. Dadas dos palabras w_1, w_2 del mismo largo, la distancia de Hamming entre w_1 y w_2 es definida como el número de posiciones en las que tienen distintos símbolos. Denote esta distancia como $\mathrm{hd}(w_1, w_2)$

   3.1 Demuestre que $\mathrm{ed}(w_1, w_2) \leq \mathrm{hd}(w_1, w_2)$
   3.2 Encuentre palabras w_3 y w_4 tales que $\mathrm{ed}(w_3, w_4) < \mathrm{hd}(w_3, w_4)$

Calculando la distancia de Levenshtein

Para calcular $\mathrm{ed}(w_1, w_2)$ no podemos considerar todas las posibles secuencias de operaciones que aplicadas desde w_1 generan w_2.

- Incluso si consideramos las secuencias de largo a los más $|w_1| + |w_2|$ vamos a tener demasiadas posibilidades.

Para calcular $\mathrm{ed}(w_1, w_2)$ utilizamos **programación dinámica**.

Notación preliminar

Dado $w \in \Sigma^*$ tal que $|w| = n$, y dado $i \in \{1, \ldots, n\}$, definimos $w[i]$ como el símbolo de w en la posición i.

- Tenemos que $w = w[1] \cdots w[n]$.

Además, definimos el **infijo** (*substring*) de w entre las posiciones i y j como:

$$w[i, j] \quad = \quad \begin{cases} w[i] \cdots w[j] & 1 \leq i \leq j \leq n \\ \varepsilon & \text{en caso contrario} \end{cases}$$

La distancia y un principio optimalidad para sub-secuencias

Sean $w_1, w_2 \in \Sigma^*$.

Suponga que o_1, ..., o_k es una secuencia óptima de operaciones que aplicadas desde w_1 generan w_2 con $k \geq 1$.

- Tenemos que $\mathrm{ed}(w_1, w_2) = k$.

Considere $0 \leq i \leq |w_1|$ y suponga que o_1, ..., o_ℓ es la sub-secuencia de las operaciones en o_1, ..., o_k que son aplicadas a $w_1[1, i]$ con $1 \leq \ell \leq k$.

Estamos suponiendo que las operaciones sobre $w_1[1, i]$ son las **primeras en ser aplicadas**.

¿Por qué podemos hacer este supuesto?