Técnicas Fundamentales

Programación Dinámica III

Segundo semestre 2022

IIC2283

Prof. Nicolás Van Sint Jan

Recordatorio: Programación dinámica

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**.

Recordatorio: 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 $\mathrm{ed} : \Sigma^* \times \Sigma^* \to \mathbb{N}$

Recordatorio: 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:

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

Además, tenemos que $\text{agregar}(w, 0, b) = bw$.

Ejemplo

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

Recordatorio: La distancia de Levenshtein

Definición

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

Ejemplo

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

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

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:

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

Outline

Programación dinámica: Palabras

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 $\text{ed}(w_1, w_2) = k$.

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?

La distancia y un principio optimalidad para sub-secuencias

Podemos realizar el supuesto por la siguiente razón: si para $s \in \{1, \ldots, \ell - 1\}$ tenemos que o_{s+1} es una operación sobre $w_1[1, i]$ pero o_s no lo es, entonces podemos **permutar** estas dos operaciones para generar $o_{s+1}, o_s^\star$ tal que:

- si o_{s+1} **cambia** un símbolo por otro, entonces $o_s^\star = o_s$.
- si o_{s+1} **agrega** un símbolo, entonces $o_s^\star$ se obtiene desde o_s cambiando la posición $t \in \{1, \ldots, n\}$ mencionada en o_s por $t + 1$.
- si o_{s+1} **elimina** un símbolo, entonces $o_s^\star$ se obtiene desde o_s cambiando la posición $t \in \{1, \ldots, n\}$ mencionada en o_s por $t - 1$.

Finalmente, suponga que la aplicación de o_1, …, o_ℓ desde $w_1[1, i]$ genera $w_2[1, j]$ con $0 \leq j \leq |w_2|$

La distancia y un principio optimalidad para sub-secuencias

Entonces la sub-secuencia $o_1, \ldots, o_\ell$ debe ser óptima para la generación de $w_2[1, j]$ a partir de $w_1[1, i]$.

- Vale decir, $\text{ed}(w_1[1, i], w_2[1, j]) = \ell$.

Demostración

Suponga que lo anterior no es cierto, y suponga que $o'_1, \ldots, o'_m$ es una secuencia de operaciones con $m < \ell$ que genera $w_2[1, j]$ desde $w_1[1, i]$

En la secuencia $o_1, \ldots, o_k$ que genera w_2 desde w_1 podemos reemplazar $o_1, \ldots, o_\ell$ por $o'_1, \ldots, o'_m$

- La secuencia resultante se puede utilizar para generar w_2 desde w_1

Concluimos que $\text{ed}(w_1, w_2) \leq (k - \ell) + m = k - (\ell - m) < k$, lo que contradice el supuesto inicial. $\square$

Una definición recursiva de la distancia de Levenshtein

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Dados $i \in \{0, \ldots, m\}$ y $j \in \{0, \ldots, n\}$, definimos

$$\mathrm{ed}(i, j) \quad = \quad \mathrm{ed}(w_1[1, i], w_2[1, j])$$

Observe que $\mathrm{ed}(w_1, w_2) = \mathrm{ed}(m, n)$

Además, definimos el valor $\mathrm{dif}(i, j)$ como 0 si $w_1[i] = w_2[j]$, y como 1 en caso contrario.

Una definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub-secuencias obtenemos la siguiente **definición recursiva** para la función ed:

$$\mathrm{ed}(i,j) \quad = \quad \begin{cases} \max\{i,j\} & i = 0 \text{ o } j = 0 \\ & \\ \min\{1 + \mathrm{ed}(i-1,j), & \\ \qquad 1 + \mathrm{ed}(i,j-1), & i > 0 \text{ y } j > 0 \\ \qquad \mathrm{dif}(i,j) + \mathrm{ed}(i-1,j-1)\} & \end{cases}$$

Tenemos entonces una forma de calcular la función ed que se basa en resolver sub-problemas más pequeños

- Estos sub-problemas están traslapados y se tiene un número polinomial de ellos, podemos aplicar entonces programación dinámica

Una implementación recursiva de la distancia de Levenshtein

EditDistance(w_1, i, w_2, j)
  if $i = 0$ **then return** j
  else if $j = 0$ **then return** i
  else
    $r :=$ **EditDistance**(w_1, $i - 1$, w_2, j)
    $s :=$ **EditDistance**(w_1, i, w_2, $j - 1$)
    $t :=$ **EditDistance**(w_1, $i - 1$, w_2, $j - 1$)
    if $w_1[i] = w_2[j]$ **then** $d := 0$
    else $d := 1$
    return $\min\{1 + r, 1 + s, d + t\}$

¿Es esta una buena implementación de **EditDistance**?

R: No porque tenemos muchas llamadas recursivas repetidas, es mejor evaluar esta función utilizando un enfoque bottom-up.

Una evaluación bottom-up de la distancia de Levenshtein

Para determinar los valores de la función ed construimos una tabla siguiendo un orden lexicográfico para los pares (i, j):

$$(i_1, j_1) < (i_2, j_2) \quad \text{si y sólo si} \quad i_1 < i_2 \text{ o } (i_1 = i_2 \text{ y } j_1 < j_2)$$

Por ejemplo, para determinar el valor de $\mathrm{ed}(\texttt{casa}, \texttt{asado})$ construimos la siguiente tabla:

		a	s	a	d	o
	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	2	3	4
s	3	2	1	2	3	4
a	4	3	2	1	2	**3**

¿ Cómo obtener o_1, o_2, o_3 tal que al aplicarlas sobre casa se obtiene asado?

Una evaluación bottom-up de la distancia de Levenshtein

A partir de la tabla podemos obtener una secuencia óptima de operaciones para transformar `casa` en `asado`:

		a	s	a	d	o
	0	1	2	3	4	5
c	**1**	1	2	3	4	5
a	2	**1**	2	2	3	4
s	3	2	**1**	2	3	4
a	4	3	2	**1**	**2**	**3**

Estas operaciones son las siguientes:

$$\begin{aligned}\text{eliminar}(\texttt{casa}, 1) &= \texttt{asa}\\ \text{agregar}(\texttt{asa}, 3, \texttt{d}) &= \texttt{asad}\\ \text{agregar}(\texttt{asad}, 4, \texttt{o}) &= \texttt{asado}\end{aligned}$$

¿Qué operaciones básicas deberíamos contar para analizar la complejidad de **EditDistance**?

El costo de calcular la distancia de Levenshtein

Consideramos como operaciones básicas:

- Acceder a una celda de la tabla (para leer o escribir).
- Realizar operaciones con elementos de la tabla (sumar o comparar).
- Comparar elementos de las palabras de entrada.

Corolario

Utilizando programación dinámica es posible construir un algoritmo para calcular $\text{ed}(w_1, w_2)$ que en el peor caso es $\Theta(|w_1| \cdot |w_2|)$