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 $\text{ed}(w_1, w_2)$ para la edit distance entre w_1 y w_2

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

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

eliminar(`hola`,1)	=	`ola`	eliminar(`hola`,3)	=	`hoa`
agregar(`hola`,0,y)	=	`yhola`	agregar(`hola`,3,z)	=	`holza`
cambiar(`hola`,2,`x`)	=	`hxla`			

Recordatorio: 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:

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

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

Algoritmos codiciosos: Codificaciones

Outline

La distancia y un principio optimalidad para sub-secuencias

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

Suponga que $o_1, \ldots, o_k$ 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 $o_1, \ldots, o_\ell$ es la sub-secuencia de las operaciones en $o_1, \ldots, o_k$ 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. □

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

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

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

Además, definimos el valor $\text{dif}(i, j)$ 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:

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

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 ed(casa, 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**

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 debemos contar al analizar la complejidad del algoritmo discutido en las transparencias anteriores?

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) y 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 $\Theta(|w_1| \cdot |w_2|)$

Outline

Algoritmos codiciosos

Los **algoritmos codiciosos** son usados, en general, para resolver **problemas de optimización**.

El principio fundamental de este tipo de algoritmos es lograr un **óptimo global** tomando **decisiones locales que son óptimas**.

Existen dos componentes fundamentales de un algoritmo codicioso:

1. Una **función objetivo** a minimizar o maximizar.
2. Una **función de selección** usada para tomar decisiones locales óptimas.

¿Siempre se puede usar un **algoritmo codicioso** para resolver un problema de optimización?

Algoritmos codiciosos

Lamentablemente en muchos casos los algoritmos codiciosos **NO** encuentran un óptimo global.

- Sin embargo, en muchos casos pueden ser usados como **heurísticas** para encontrar valores de la función objetivo cercanos al óptimo.

Una segunda dificultad con los algoritmos codiciosos es que, en general, se necesita de una demostración formal para asegurar que encuentran el óptimo global.

- Aunque pueden ser algoritmos simples, en algunos casos no es obvio por qué encuentran el óptimo global.

Vamos a ver un ejemplo clásico de algoritmos codiciosos.

Almacenamiento de datos

Sea Σ un alfabeto. Dada una palabra $w \in \Sigma^*$ suponga que queremos **almacenar** w **utilizando los símbolos 0 y 1**.

- Esta palabra w puede ser pensada como un **documento** si Σ incluye las letras del alfabeto español, símbolos de puntuación y el espacio, por lo tanto puede ser muy larga.
- El uso de 0 y 1 corresponde a la idea de almacenar el texto en un computador.

Definimos entonces una función $\tau : \Sigma \to \{0,1\}^*$ que asigna a cada símbolo en $a \in \Sigma$ una palabra en $\tau(a) \in \{0,1\}^*$ con $\tau(a) \neq \varepsilon$

- Vamos a almacenar w reemplazando cada símbolo $a \in \Sigma$ que aparece en w por $\tau(a)$.
- Llamamos a τ una **Σ-codificación**.

Almacenamiento de datos

La **extensión** $\hat{\tau}$ de una Σ-codificación τ a todas las palabras $w \in \Sigma^*$ se define como:

$$\hat{\tau}(w) = \begin{cases} \varepsilon & w = \varepsilon \\ \tau(a_1) \cdots \tau(a_n) & w = a_1 \cdots a_n \text{ con } n \geq 1 \end{cases}$$

Vamos a almacenar w como $\hat{\tau}(w)$

- Si la Σ-codificación τ está fija (como el código ASCII) entonces no es necesario almacenarla
- Si τ no está fija entonces debemos almacenarla junto con $\hat{\tau}(w)$
  - En general $|w|$ es mucho más grande que $|\Sigma|$, por lo que el costo de almacenar τ es despreciable

Almacenamiento de datos

La función $\hat{\tau}$ debe especificar una traducción no ambigua:

$$\forall w_1, w_2 \in \Sigma^*. w_1 \neq w_2 \;\Rightarrow\; \hat{\tau}(w_1) \neq \hat{\tau}(w_2)$$

De esta forma podemos reconstruir el texto original dada su traducción. **(¿Por qué?)**.

Vale decir, $\hat{\tau}$ debe ser una **función inyectiva**.

¿Cómo podemos lograr esta propiedad?

Codificaciones de largo fijo

Es claro que para lograr una traducción no ambigua necesitamos que $\tau(a) \neq \tau(b)$ para cada $a, b \in \Sigma$ tal que $a \neq b$

Para obtener la misma propiedad para $\hat{\tau}$ podemos imponer la siguiente condición:

$$\forall a, b \in \Sigma.\ |\tau(a)| = |\tau(b)|$$

Si se cumple esta condición entonces diremos que τ es una **Σ-codificación de largo fijo**.

Ejercicio

Muestre que para tener una Σ-codificación de largo fijo basta con asignar a cada $a \in \Sigma$ una palabra $\tau(a) \in \{0,1\}^*$ tal que $|\tau(a)| = \lceil \log_2(|\Sigma|) \rceil$

Codificaciones de largo variable

Por otro lado, decimos que τ es una **Σ-codificación de largo variable** si

$$a, b \in \Sigma.\ |\tau(a)| \neq |\tau(b)|$$

¿Por qué nos conviene utilizar **codificaciones de largo variable**?

R: Podemos obtener representaciones más cortas para la palabra que queremos almacenar

Codificaciones de largo variable

Ejemplo

Suponga que $w = aabaacaab$.

- Para $\tau_1(a) = 00$, $\tau_1(b) = 01$ y $\tau_1(c) = 10$ tenemos que:

$$\hat{\tau}_1(w) \quad = \quad 000001000010000001$$

- Para $\tau_2(a) = 0$, $\tau_2(b) = 10$ y $\tau_2(c) = 11$ tenemos que:

$$\hat{\tau}_2(w) \quad = \quad 001000110010$$

Por lo tanto $|\hat{\tau}_2(w)| = 12 < 18 = |\hat{\tau}_1(w)|$

¿Por qué $\hat{\tau}_2$ es **inyectiva**?

Codificaciones de largo variable

Lema

Si existen $w_1, w_2 \in \Sigma^*$ tales que $w_1 \neq w_2$ y $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, entonces existen $a, b \in \Sigma$ tales que $a \neq b$ y $\tau(a)$ es un prefijo de $\tau(b)$

Demostración

Suponga que $w_1 \neq w_2$, $\hat{\tau}(w_1) = \hat{\tau}(w_2)$ y

$$\begin{array}{lcll} w_1 & = & a_1 \dots a_m & m \geq 1 \\ w_2 & = & b_1 \dots b_n & n \geq 1 \end{array}$$

Además, sin perdida de generalidad suponga que $m \leq n$.

Si w_1 es **prefijo propio** de w_2 entonces $\hat{\tau}(w_1)$ es prefijo propio de $\hat{\tau}(w_2)$

- Puesto que $\tau(a) \neq \varepsilon$ para cada $a \in \Sigma$.

Dado que $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, tenemos entonces que w_1 no es prefijo propio de w_2.

Codificaciones de largo variable

Demostración (continuación)

Sea $k = \min_{1 \leq i \leq m} a_i \neq b_i$

k está bien definido puesto que $w_1 \neq w_2$ y w_1 no es prefijo propio de w_2.

Dado que $\hat{\tau}(a_1 \cdots a_{k-1}) = \hat{\tau}(b_1 \cdots b_{k-1})$ y $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, concluimos que $\hat{\tau}(a_k \cdots a_m) = \hat{\tau}(b_k \cdots b_n)$.

Tenemos entonces que $\tau(a_k) \cdots \tau(a_m) = \tau(b_k) \cdots \tau(b_n)$, de lo cual se deduce que $\tau(a_k)$ es un prefijo de $\tau(b_k)$ o $\tau(b_k)$ es un prefijo de $\tau(a_k)$.

Lo cual concluye la demostración puesto que $a_k \neq b_k$. □

Codificaciones libres de prefijos

Decimos que una Σ-codificación τ es **libre de prefijos** si para cada $a, b \in \Sigma$ tales que $a \neq b$ se tiene que $\tau(a)$ no es un prefijo de $\tau(b)$

Ejemplo

Para $\Sigma = \{a, b, c\}$ y $\tau(a) = 0$, $\tau(b) = 10$, $\tau(c) = 11$, se tiene que τ es libre de prefijos.

Corolario

Si τ es una codificación libre de prefijos, entonces $\hat{\tau}$ es una **función inyectiva**.