Programación Dinámica

Integrantes

- Christian Echeverría 221441
- Gustavo Cruz 22779
- Josué Say 22801
- Mathew Cordero 22982
- Pedro Guzmán 22111

1. Modificación del Rod-Cutting Problem (1)

Descrpción

Suponga que al rod-cutting problem agregamos un límite piezas que se pueden vender o cortar para cada posible tamaño. Es decir que, para $0 < i \leq n, l_i \in N$ es el número máximo de piezas de tamaño i que podemos usar en la solución. Demuestre que con esta nueva restricción el problema ya no exhibe una subestructura óptima. Hint: ¿qué característica de los subproblemas se evaluó para determinar la ausencia de subestructura óptima en el unweighted longest simple path problem?

Solución

En un problema normal de Rod Cutting definimos $C(n)$ como el costo mínimo para cortar una barra de longitud n:

$$C(n) = \min_{1 \leq i \leq n} (C(i) + C(n-i))$$

Pero con la nueva restricción en la cantidad de piezas, debemos verificar que la elección de una pieza de tamaño i respete la restricción de cantidad de piezas l_i.

Ahora debemos buscar un contraejemplo donde la solución óptima de un subproblema **no se pueda reutilizar** para construir la solución óptima del problema general.

Base de la Inducción

Si $n = 0$, el costo es 0. Si $n = 1$, el costo es simplemente el costo de la pieza de tamaño 1 (si está disponible bajo la restricción l_1).

Si suponemos que para cualquier longitud menor que n, la restricción en la cantidad de piezas no afecta la subestructura óptima y se puede resolver óptimamente cada subproblema.

$$C(n) = \min_{1 \leq i < n} (C(i) + C(n-i))$$

Pero ahora, la restricción en la cantidad de piezas **rompe la independencia de los subproblemas.** Porque si en una solución óptima para una barra de longitud menor se usó el máximo permitido de piezas de un tamaño particular, esa misma solución **no puede ser utilizada nuevamente** cuando se resuelve el problema para una longitud mayor.

Ejemplo

Si que quiere cortar una barra de tamaño $n = 5$ y que podemos usar un máximo de:

- $l_2 = 2$ (como máximo 2 piezas de tamaño 2)
- $l_3 = 1$ (como máximo 1 pieza de tamaño 3)

Se puede construir a partir de:

Usar 2 piezas de tamaño 2 y 1 pieza de tamaño 1

Pero si ahora se aplica el mismo enfoque con una barra con mayor tamaño, por ejemplo $n = 8$, la solución óptima **podría haber reutilizado las piezas de tamaño 2**, pero como ya hemos alcanzado el límite en una instancia previa, esta solución deja de ser válida.

Como en el caso del problema unweighted longest simple path, donde la mejor solución a un subproblema puede volverse inválida al expandirse a un problema mayor, aquí sucede lo mismo con la restricción en la cantidad de piezas permitidas.

2. Modificación del Rod-Cutting Problem (2)

Descripción

Volvamos al rod-cutting problem original. Resulta que no tenemos ni las herramientas ni las habilidades necesarias para cortar tubos, por lo que subcontratamos este servicio. Cada corte que queramos realizar costará una cantidad fija c. ¿Qué modificación se le tiene que hacer al algoritmo ya provisto para adaptarse a esta nueva condición?

Solución

En el problema rod-cutting, buscamos maximizar el beneficio al vender piezas de una barra de longitud n. La ecuación de recurrencia estándar es:

$$r(n) = \max_{1 \leq i \leq n} \{p_i + r(n-i)\}$$

donde p_i representa el precio de una pieza de longitud i.

Modificación por Costo de Corte Si cada corte tiene un costo fijo c, debemos considerar este gasto adicional. Cada vez que realizamos un corte, pagamos c, lo que modifica la ecuación de recurrencia:

$$r(n) = \max_{1 \leq i \leq n} \{p_i + r(n-i) - c\}$$

donde:

- El término $-c$ se aplica únicamente si efectivamente realizamos un corte, es decir, cuando $i < n$.
- Si no se hacen cortes (es decir, si se vende la barra completa), no se paga el costo c.

3. Cálculo de la Distancia de Edición mediante Programación Dinámica

Descripción

Tenemos dos strings X e Y cuyos tamaños son m y n respectivamente. Para transformar X en Y le aplicamos una secuencia de operaciones cuyos resultados se van almacenando en un string Z. La transformación se lleva con índices i y j sobre X y Z respectivamente, iniciando con $i = j = 1$. Cuando la transformación concluye se debe tener que $i = m + 1, j = n$ y $Z = Y$. Las operaciones permitidas para la transformación son las siguientes:

- **Copy**: copia un carácter de X a Z haciendo $Z[j] = X[i]$, y luego incrementa tanto i como j.
- **Replace**: reemplaza un carácter de X con algún otro carácter c dado, haciendo $Z[j] = c$; y luego incrementa tanto i como j.
- **Delete**: ignora un carácter de X al incrementar i sin incrementar j.
- **Insert**: inserta en Z un carácter c dado, haciendo $Z[j] = c$, y luego incrementa j sin incrementar i.
- **Twiddle**: intercambia los siguientes dos caracteres de X copiándolos en orden inverso. Para lograrlo hace $Z[j] = X[i+1]$ y $Z[j+1] = X[i]$, y luego incrementa tanto i como j dos unidades (i.e., $i = i + 2, j = j + 2$).
- **Kill**: ignora el resto de X haciendo $i = m + 1$. Esta operación, si se usa, debe ser la última.

Cada operación tiene un costo constante propio, pero los costos de **Copy** y **Replace** son, cada uno, menores al costo de hacer **Delete** seguido de un **Insert.** Considere el siguiente ejemplo ilustrativo que convierte la palabra algorithm en altruistic:

Operation	x	z
initial strings	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruís_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

Para este ejercicio deberá desarrollar un algoritmo apoyado en el acercamiento **bottom-up** de programación dinámica que permita encontrar la secuencia de operaciones que convierte un string en otro incurriendo en el menor costo posible (este costo mínimo es llamado la **edit distance**). Para realizar el ejercicio siga los siguientes pasos:

a. Identifique la subestructura óptima siguiendo el procedimiento enseñado en clase.
   Hint: para identificar los subproblemas, considere una secuencia de operaciones $(o_1, ..., o_k)$ dada como óptima. Habiéndose aplicado alguna de las operaciones, ¿qué podemos decir que tiene que haber sucedido antes de aplicarse dicha operación? ¿Cuál de los pasos del ejemplo hace la conversión más sencilla (caso base)?

b. Esboce una tabla T con m filas y n columnas. Esta tabla debe llenarse durante la ejecución de su solución bottom-up. ¿Cuál es el significado del contenido de la celda $T[i, j]$?

c. Apoyándose en el inciso anterior, escriba la ecuación recurrente que computa el valor de la celda $T[i, j]$ tomando en cuenta las condiciones que restringen el uso de cada operación. Puede describir el costo de una operación como **costo(nombre de operación).**

Solución

Se puede ver que:

a. Subestructura Óptima Los subproblemas son los siguientes:

Identificamos las operaciones que son:

- **Copy** (cuando los caracteres son iguales).
- **Replace** (reemplazar un carácter por otro).
- **Delete** (eliminar un carácter de X).
- **Insert** (agregar un carácter a Y).
- **Twiddle** (intercambiar dos caracteres consecutivos).
- **Kill** (eliminar toda la cadena restante, solo si es la última operación).

Caso Base

- Si X está vacío, el costo es simplemente insertar todos los caracteres de Y.
- Si Y está vacío, el costo es eliminar todos los caracteres de X.
- La copia de X y Y si son iguales.

b. Tabla T

$T[i][j]$ representa el costo mínimo de transformar los primeros i caracteres de X en los primeros j caracteres de Y.

Para iniciar la tabla tenemos:

- $T[0][j] = j$ (convertir una cadena vacía a los primeros j caracteres).
- $T[i][0] = i$ (convertir los primeros i de X en eliminaciones).

c. Ecuación de Recurrencia

- Si $X[i]$ y $Y[j]$ son iguales:

$$T[i][j] = T[i-1][j-1]$$

  (costo de la copia)

- Si $X[i]$ y $Y[j]$ no son iguales:

$$T[i][j] = \min \begin{cases} T[i-1][j-1] + \text{costo de Reemplazar} \\ T[i-1][j] + \text{costo de Eliminar} \\ T[i][j-1] + \text{costo de Insertar} \end{cases}$$

En caso de **Intercambio**, si y solo si:

$$X[i] = Y[j-1] \text{ o } X[i-1] = Y[j]$$

Ejemplo del Algoritmo en Pseudocódigo

```
FUNCION edit_distance(X, Y, cost_replace, cost_delete, cost_insert):
    m ← longitud(X)
    n ← longitud(Y)

    T ← matriz de tamaño (m+1) x (n+1) con valores 0

    PARA i DESDE 0 HASTA m:
        T[i][0] ← i * cost_delete

    PARA j DESDE 0 HASTA n:
        T[0][j] ← j * cost_insert

    PARA i DESDE 1 HASTA m:
        PARA j DESDE 1 HASTA n:
            SI X[i-1] == Y[j-1]:
                T[i][j] ← T[i-1][j-1]
            SINO:
                T[i][j] ← mínimo(
                    T[i-1][j-1] + cost_replace,
                    T[i-1][j] + cost_delete,
                    T[i][j-1] + cost_insert
                )

            SI i > 1 Y j > 1 Y X[i-1] == Y[j-2] Y X[i-2] == Y[j-1]:
                T[i][j] ← mínimo(T[i][j], T[i-2][j-2] + 1)

    RETORNAR T[m][n]

//Ejemplo de uso
X ← "algorithm"
Y ← "altruistic"
IMPRIMIR "Distancia de edición:", edit_distance(X, Y, 1, 1, 1)
```

4. Algoritmo Bellman-Ford

Descripción

Considere el siguiente problema: dado un grafo dirigido y ponderado $G = (V, E)$, una función de peso $w : E \to \mathbb{R}^+$ y un vértice origen $s \in V$, encontrar el camino más corto desde s hasta cualquier otro vértice. Este problema es resuelto por el **algoritmo de Bellman-Ford**, presentado a continuación. En el algoritmo, d es el arreglo de soluciones que se llena con las distancias más cortas desde el vértice origen s hasta cada uno de los demás vértices en el grafo. π es el arreglo que contiene, para un vértice v dado, el vértice que le precede en el camino más corto de s hacia v.

Bellman-Ford algorithm

```
d[s] ← 0
for each v   V − {s}
    do d[v] ← ϖ

for i ← 1 to |V| − 1 do
    for each edge (u, v)   E do
        if d[v] > d[u] + w(u, v) then
            d[v] ← d[u] + w(u, v)
             [v] ← u
```

Busque o desarrolle un ejemplo de aplicación de este algoritmo por pasos. Apoyándose en el algoritmo, identifique la subestructura óptima y los subproblemas traslapados del problema. Luego pruebe y explique la recurrencia que calcula el valor de la solución óptima. Observe que, aunque d es un arreglo, todos los valores de d se actualizan varias veces conforme avanza el ciclo externo.

Solución

Consideremos el siguiente grafo:

- **Vértices:** $V = \{A, B, C, D\}$
- **Aristas y pesos:**
  - $A \to B$ con peso 1
  - $A \to C$ con peso 4
  - $B \to C$ con peso 2
  - $B \to D$ con peso 6
  - $C \to D$ con peso 1
- **Vértice origen:** $s = A$

El objetivo es hallar la **distancia mínima** desde A hasta cada vértice utilizando el algoritmo de Bellman-Ford.

1. Inicialización Definimos la tabla $d[k, v]$ donde k indica el número de iteraciones (o el máximo número de aristas permitidas).

Para $k = 0$:

- $d[0, A] = 0$ (origen)
- $d[0, B] = +\infty$
- $d[0, C] = +\infty$
- $d[0, D] = +\infty$

k	d(A)	d(B)	d(C)	d(D)
0	0	∞	∞	∞

2. Primera Iteración ($k = 1$) Se relajan todas las aristas usando los valores de $d[0, \cdot]$:

1. **Arista $A \to B$:**
   $d[1, B] = \min(d[0, B],\ d[0, A] + 1) = \min(+\infty,\ 0 + 1) = 1.$
2. **Arista $A \to C$:**
   $d[1, C] = \min(d[0, C],\ d[0, A] + 4) = \min(+\infty,\ 0 + 4) = 4.$
3. **Arista $B \to C$:**
   Como $d[0, B] = +\infty$, no mejora $d[1, C]$.
4. **Arista $B \to D$:**
   No se actualiza, pues $d[0, B] = +\infty$.
5. **Arista $C \to D$:**
   No se actualiza, pues $d[0, C] = +\infty$.

La tabla tras la primera iteración es:

k	d(A)	d(B)	d(C)	d(D)
0	0	∞	∞	∞
1	0	1	4	∞

3. Segunda Iteración ($k = 2$) Usamos los valores de $d[1, \cdot]$ para relajar nuevamente:

1. **Arista $A \to B$:**
   $d[2, B] = \min(d[1, B],\ d[1, A] + 1) = \min(1,\ 0 + 1) = 1.$

2. **Arista** $A \to C$**:**
   $d[2, C] = \min(d[1, C],\ d[1, A] + 4) = \min(4,\ 0 + 4) = 4.$

3. **Arista** $B \to C$**:**
   $d[2, C] = \min(4,\ d[1, B] + 2) = \min(4,\ 1 + 2) = 3.$ (Se mejora $d(C)$ de 4 a 3.)

4. **Arista** $B \to D$**:**
   $d[2, D] = \min(+\infty,\ d[1, B] + 6) = \min(+\infty,\ 1 + 6) = 7.$

5. **Arista** $C \to D$**:**
   $d[2, D] = \min(7,\ d[1, C] + 1) = \min(7,\ 4 + 1) = 5.$ (Se mejora $d(D)$ de 7 a 5.)

La tabla tras la segunda iteración es:

k	d(A)	d(B)	d(C)	d(D)
1	0	1	4	∞
2	0	1	3	5

4. Tercera Iteración ($k = 3$**)** Finalmente, se relajan nuevamente las aristas usando $d[2, \cdot]$:

1. **Arista** $A \to B$**:**
   $d[3, B] = \min(d[2, B],\ d[2, A] + 1) = \min(1,\ 0 + 1) = 1.$

2. **Arista** $A \to C$**:**
   $d[3, C] = \min(d[2, C],\ d[2, A] + 4) = \min(3,\ 0 + 4) = 3.$

3. **Arista** $B \to C$**:**
   $d[3, C] = \min(3,\ d[2, B] + 2) = \min(3,\ 1 + 2) = 3.$

4. **Arista** $B \to D$**:**
   $d[3, D] = \min(d[2, D],\ d[2, B] + 6) = \min(5,\ 1 + 6) = 5.$

5. **Arista** $C \to D$**:**
   $d[3, D] = \min(5,\ d[2, C] + 1) = \min(5,\ 3 + 1) = 4.$ (Se mejora $d(D)$ de 5 a 4.)

La tabla final queda:

k	d(A)	d(B)	d(C)	d(D)
2	0	1	3	5
3	0	1	3	4

Conclusión Después de $|V| - 1 = 3$ iteraciones, las **distancias mínimas** desde el vértice A son:

- $d(A) = 0$

- $d(B) = 1$
- $d(C) = 3$
- $d(D) = 4$

Subestructura optima y subproblemas traslapdos

En este algoritmo el problema principal es encontrar un camino mas corto entre los vertices S y v.

Los subproblemas en los que se puede dividir son:

- Encontrar el camino más corto entre v y cada i (1 ≤ i ≤ |V⊢1), usando i aristas.
- Por ejemplo, en el camino A→B→C→D, el camino más corto a D depende del camino más corto a C, que a su vez depende del camino más corto a B.

Subproblemas traslapados Los subproblemas se superponen porque el cálculo de $d[v]$ puede depender de $d[u]$ para múltiples vértices u.

Ejemplo: $d[D]$ se actualiza primero usando B y luego usando C.

Recurrencia

Relación de Recurrencia:

La relación de recurrencia que define el valor óptimo d[v] es:

$$d[v] = \min_{(u,v)\in E}(d[u] + w(u,v))$$

- Esto significa que la distancia más corta a v es el mínimo de las distancias a sus predecesores u más el peso de la arista (u, v).