Algoritmos y Estructuras de Datos

Programación dinámica

Camino mínimo

Programación dinámica

Camino mínimo

Programación dinámica

La *programación dinámica* es una técnica de diseño de algoritmos.

Es útil para **evitar la repetición de cómputos**.
Se puede aplicar cuando el problema tiene ciertas características.

Principio de optimalidad

Un problema cumple con el **principio de optimalidad** si:

> Dada una instancia P del problema y P' una subinstancia:
> la solución óptima para P
> se construye a partir de soluciones óptimas para P'.

Ejemplos — Principio de optimalidad

- Si queremos encontrar la ruta más corta de Ushuaia a La Quiaca, y la ruta pasa por Río Cuarto, la solución al problema incluye también la ruta más corta de Ushuaia a Río Cuarto.
  (Cumple con el P.O.)
- Si tenemos un arreglo A de enteros (posiblemente negativos) y queremos encontrar el fragmento de suma máxima, la solución al problema **no** incluye el fragmento de suma máxima de los subarreglos de A.
  (**No** cumple con el P.O.)

Sucesión de Fibonacci

La sucesión de Fibonacci F se define así:

$$F(0) = 1 \qquad F(1) = 1 \qquad F(n) = F(n-2) + F(n-1) \text{ si } n \geq 2$$

El algoritmo ingenuo para calcularla es exponencial:

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

Cumple con el **principio de optimalidad**:

- Para calcular $F(n)$ es necesario calcular $F(i)$ para todos los valores de $i < n$.

Sucesión de Fibonacci

El algoritmo se puede hacer lineal con **memoización**:

```
TABLA = {}
def fib(n):
    if n in TABLA:
        return TABLA[n]
    if n <= 1:
        res = 1
    else:
        res = fib(n - 2) + fib(n - 1)
    TABLA[n] = res
    return res
```

Sucesión de Fibonacci

O se puede dar un algoritmo "directo":

```
def fib(n):
    a = 1
    b = 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Subsecuencia común más larga

Si $w = x_0\, x_1\, \ldots x_{n-1}$ es una palabra, una **subsecuencia** resulta de borrar cero, uno, o muchos de sus símbolos.

Dadas dos palabras $v, w \in \Sigma^*$, queremos hallar la longitud de la palabra más larga que es subsecuencia de v y de w.

Por ejemplo, la subsecuencia común más larga de:

$$\begin{aligned} v &= \text{“abracadabra” “abracadabra”} \\ w &= \text{“carburan” “carburan”} \end{aligned}$$

es “cabra” y:

$$\mathrm{lcs}(v, w) = 5$$

Aplicaciones

- *Diffs* de archivos de texto.
- Detección de plagios.
- Análisis de secuencias de ADN.

Subsecuencia común más larga

Principio de optimalidad

El valor de $\text{lcs}(x_0 \dots x_{i-1}x_i,\ y_0 \dots y_{j-1}y_j)$
se puede conocer en función de:

- $\text{lcs}(x_0 \dots x_{i-1},\ y_0 \dots y_{j-1})$
- $\text{lcs}(x_0 \dots x_{i-1}x_i,\ y_0 \dots y_{j-1})$
- $\text{lcs}(x_0 \dots x_{i-1},\ y_0 \dots y_{j-1}y_j)$

Subsecuencia común más larga

Ejemplo

$\mathsf{lcs}(\texttt{od}, \texttt{hol}) = 1$	$\mathsf{lcs}(\texttt{oda}, \texttt{hol}) = 1$
$\mathsf{lcs}(\texttt{od}, \texttt{hola}) = 1$	$\mathsf{lcs}(\texttt{oda}, \texttt{hola}) = 2$

Otro ejemplo

$\mathsf{lcs}(\texttt{oh}, \texttt{hol}) = 1$	$\mathsf{lcs}(\texttt{oho}, \texttt{hol}) = 2$
$\mathsf{lcs}(\texttt{oh}, \texttt{hola}) = 1$	$\mathsf{lcs}(\texttt{oho}, \texttt{hola}) = 2$

Subsecuencia común más larga

Entrada: dos palabras v, w de tamaños $|v| = n$ y $|w| = m$.
Salida: $\text{lcs}(v, w)$

- Armamos una tabla L de $(n+1) \times (m+1)$ enteros.
  Queremos calcular en $L[i, j]$ el valor de $\text{lcs}(v[0..i), w[0..j))$.
- Inicializamos:
  - $L[i, 0] := 0$ para todo $0 \leq i \leq n$
  - $L[0, j] := 0$ para todo $0 \leq j \leq m$
- Para cada i desde 1 hasta n, y para cada j desde 1 hasta m:

$$
\begin{aligned}
L[i,j] := \max\{ & \\
& L[i-1, j], \\
& L[i, j-1], \\
& L[i-1, j-1] + \begin{cases} 1 & \text{si } v[i-1] = w[j-1] \\ 0 & \text{si no} \end{cases} \\
\}&
\end{aligned}
$$

- El valor de $\text{lcs}(v, w)$ se halla en $L[n][m]$.

Subsecuencia común más larga

		a	**b**	**r**	**a**	**c**	**a**	**d**	**a**	**b**	**r**	**a**
	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2	2
r	0	1	1	2	2	2	2	2	2	2	3	3
b	0	1	2	2	2	2	2	2	2	3	3	3
u	0	1	2	2	2	2	2	2	2	3	3	3
r	0	1	2	3	3	3	3	3	3	3	4	4
a	0	1	2	3	4	4	4	4	4	4	4	5
n	0	1	2	3	4	4	4	4	4	4	4	5

Subsecuencia común más larga en Python

```
def lcs(v, w):
    n = len(v)
    m = len(w)
    L = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            b = 1 if v[i - 1] == w[j - 1] else 0
            L[i][j] = max(L[i - 1][j],
                          L[i][j-1],
                          L[i - 1][j - 1] + b)
    return L[n][m]
```

Problema del cambio

Tenemos un arreglo $D = [d_1, \ldots, d_k]$ de enteros positivos, sin repetidos, que corresponden a denominaciones de billetes. Dado un entero n, queremos determinar cuál es la menor cantidad de billetes que se pueden usar para sumar n.

Asumimos que tenemos una cantidad ilimitada de billetes de cada denominación.

Por ejemplo, si:

$$D = [5, 10, 25, 50] \qquad n = 45$$

la respuesta es 3 (dada por $25 + 10 + 10$).

Camino mínimo

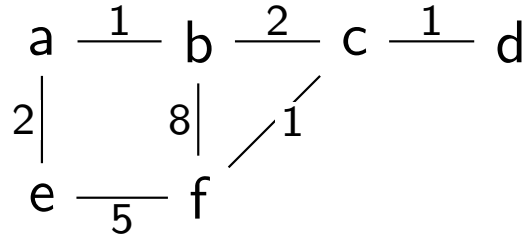
Camino mínimo en grafos

Descripción del problema

Tenemos un grafo no dirigido $G = (V, E)$ y una función $w : E \to \mathbb{R}_{>0}$ que le asigna un **peso** positivo a cada arista.

Dados dos vértices $v, w \in V$, queremos hallar el camino más corto (*i.e.* el de **menor peso**) que conecta v con w.

Ejemplo



El camino más corto de a a f es:

a — b — c — f

Algoritmo de Dijkstra

Entrada: un grafo $G = (V, E)$ con pesos y dos vértices $v, w \in V$.
Los pesos de las aristas deben ser positivos.
Salida: el peso del camino más corto de v a w.

- Inicializar una cola de prioridad con $(0, v)$.
  La primera componente representa la distancia desde v.
  Un elemento (d, x) tiene mayor prioridad cuando d es menor.
- Inicializar un conjunto de vértices visitados como $W := \emptyset$.
- Mientras la cola de prioridad no esté vacía:
  - Sacar el par más prioritario (d, x) de la cola de prioridad.
    Nota: x es el vértice más cercano todavía no visitado.
    La distancia de v a x es d.
  - Si $x = w$, devolver d.
  - Marcar x como visitado.
  - Para cada vecino y de x que no haya sido visitado:
    - Sea p el peso de la arista x — y.
    - Agregar $(d + p, y)$ en la cola de prioridad.
- Llegado este punto, no hay camino de v a w.

Algoritmo de Dijkstra en Python

La complejidad es $O((n + m) \log n)$.

Algoritmo de Dijkstra en Python

```
def dijkstra(grafo, v, w):
    cola = []
    visitados = set()
    heapq.heappush(cola, (0, v))
    while len(cola) > 0:
        (dist, x) = heapq.heappop(cola)
        if x == w:
            return dist
        visitados.add(x)
        for y in grafo.vecinos[x]:
            if y in visitados:
                continue
            peso = grafo.peso[(x, y)]
            heapq.heappush(cola, (dist + peso, y))
    return "no hay camino"
```

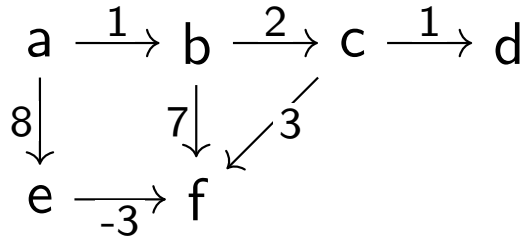
Nota. Sería más eficiente reemplazar la prioridad cuando el vértice ya se encontraba en la cola.

Camino mínimo con pesos negativos

Descripción del problema

Tenemos un grafo **dirigido** $G = (V, E)$ y una función $w : E \to \mathbb{R}$ que le asigna un **peso** posiblemente negativo a cada arista.

Ejemplo



El camino más corto de a a f es:

$$a \longrightarrow e \longrightarrow f$$

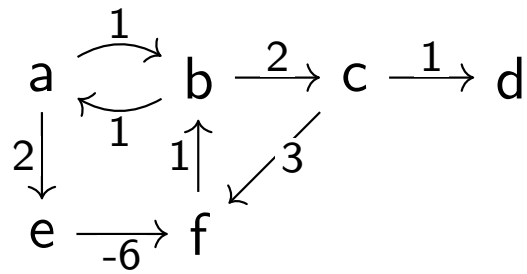
El algoritmo de Dijkstra encuentra **incorrectamente** el camino:

$$a \longrightarrow b \longrightarrow c \longrightarrow f$$

Camino mínimo con pesos negativos

Problema: ciclos negativos

Si un grafo tiene **ciclos negativos**, no siempre existe el camino mínimo:



El ciclo:

$$a \longrightarrow e \longrightarrow f \longrightarrow b \longrightarrow a$$

tiene peso -2.

Hay caminos de peso arbitrariamente chico desde a hasta c.

Algoritmo de Bellman–Ford

Entrada: un digrafo $G = (V, E)$ con pesos y un vértice $v \in V$.
Se asume que el grafo no tiene ciclos negativos.
Salida: el diccionario de distancias a cada vértice $w \in V$.

- Inicializar un diccionario D de tal modo que:
  - $D[v] = 0$
  - $D[w] = +\infty$ para todo vértice $w \neq v$
- Repetir n veces, una para cada vértice del grafo:
  - Para cada arista $(w, w') \in E$:
    - $D[w'] = \min\{D[w'], D[w] + \text{peso}(w, w')\}$
- Devolver D.

Complejidad: $O(n \cdot m)$.