UNIVERSIDAD AUTÓNOMA DE BAJA CALIFORNIA

Facultad de Ciencias de la Ingeniería y Tecnología

Unidad Valle de las Palmas

Análisis de algoritmos

Meta 5.1 Programación Dinámica

José Humberto Moreno Mejía

Noviembre 2024

La programación dinámica es una técnica de optimización para resolver problemas complejos dividiéndolos en subproblemas más simples, reutilizando los resultados de estos subproblemas para mejorar la eficiencia. Este enfoque es útil en problemas que cumplen dos propiedades:

Subestructura óptima: La solución óptima del problema puede formarse a partir de las soluciones de sus subproblemas.

Superposición de subproblemas: Los mismos subproblemas se resuelven repetidamente, y sus resultados pueden almacenarse para evitar cálculos innecesarios.

Ventajas

Ahorro de recursos: Al evitar cálculos repetitivos, reduce el tiempo y la complejidad computacional.
Aplicación en problemas variados: Se utiliza en áreas como economía, planificación y optimización logística.
Soluciones óptimas: Asegura encontrar la mejor solución posible en problemas de optimización.

Ejemplos

Entre los más famosos ejemplos de problemas resueltos con programación dinámica se encuentran:

- **Problema de la Mochila**: Selección de elementos para maximizar el beneficio sin exceder un límite de peso.
- **Subsecuencia Común Más Larga (LCS):** Encontrar la subsecuencia común más larga entre dos secuencias.
- **Problema del Viajero (TSP):** Minimizar la distancia total en una ruta de múltiples ciudades.

Solución del problema propuesto

Analizando el problema nos damos cuenta que se encuentran los 3 algoritmos con el que se resuelven los ejemplos anteriores.

Antes de resolver debemos declarar los datos del problema:

```
productos = ["laptop", "camara", "libro", "zapatos", "ropa"]
pesos = [3, 2, 1, 4, 5]
valores = [1000, 500, 150, 800, 200]

pedido_cliente = ["laptop", "camara", "zapatos", "telefono"]
capacidad_vehiculo = 100
```

1. Optimización de la Carga del Vehículo

Debemos maximizar el valor de los productos que el vehículo puede cargar sin exceder su límite de peso (100 kg).

Función mochila: Esta función se utiliza para llenar una tabla “dp” que guarda el valor máximo que se puede obtener con los productos disponibles y con un peso límite dado.

```
def mochila(pesos, valores, capacidad):
    n = len(pesos)
    dp = [[0 for _ in range(capacidad + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacidad + 1):
            if pesos[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], valores[i-1] + dp[i-1][w - pesos[i-1]])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacidad]
```

La tabla se llena iterando sobre los productos y capacidades, eligiendo el beneficio máximo entre incluir o excluir cada producto

2. Verificación entre Inventario y Pedido

Debemos comparar el pedido del cliente con el inventario para identificar los productos que están en ambos y evitar errores en la entrega.

Se utiliza una función “lcs” para encontrar la subsecuencia común más larga entre el pedido del cliente y el inventario.

```python
def lcs(pedido, inventario):
    m, n = len(pedido), len(inventario)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if pedido[i-1] == inventario[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    subsecuencia = []
    i, j = m, n
    while i > 0 and j > 0:
        if pedido[i-1] == inventario[j-1]:
            subsecuencia.append(pedido[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

    subsecuencia.reverse()
    return subsecuencia
```

Primero se construye una tabla dp para almacenar la longitud de la subsecuencia común más larga. Luego, se reconstruye la subsecuencia usando la tabla dp, permitiendo identificar los productos que coinciden entre el inventario y el pedido del cliente.

3. Ruta de Entrega Más Corta

Debemos encontrar la ruta más corta para entregar los productos en los puntos A, B, C y D. Para esto, estructuramos los datos de las distancias A, B, C y D.

La función “ruta_mas_corta” utiliza el módulo itertools para probar todas las permutaciones posibles de las rutas y encontrar la de menor distancia total.

```python
import itertools

distancias = {
    'A': {'A': 0, 'B': 3, 'C': 9, 'D': 8},
    'B': {'A': 3, 'B': 0, 'C': 7, 'D': 5},
    'C': {'A': 9, 'B': 7, 'C': 0, 'D': 6},
    'D': {'A': 8, 'B': 5, 'C': 6, 'D': 0}
}

def ruta_mas_corta(puntos, distancias):
    mejor_ruta = None
    menor_distancia = float('inf')

    for ruta in itertools.permutations(puntos):
        distancia_total = 0
        for i in range(len(ruta) - 1):
            distancia_total += distancias[ruta[i]][ruta[i + 1]]
        distancia_total += distancias[ruta[-1]][ruta[0]]
        if distancia_total < menor_distancia:
            menor_distancia = distancia_total
            mejor_ruta = ruta

    return mejor_ruta, menor_distancia
```

La función usa todas las posibles permutaciones de las rutas entre los puntos para calcular la distancia total. Al final registra la ruta con la distancia más corta, considerando que el vehículo debe regresar al punto inicial al final de la ruta.

Al final debemos llamar a las funciones y dados los datos que le entregamos debería darnos un resultado aproximadamente así:

```
Beneficio maximo que puede cargar el vehiculo: 2650
Productos coincidentes en inventario y pedido: ['laptop', 'camara', 'zapatos']
Ruta mas corta: ('A', 'B', 'D', 'C')
Distancia total de la ruta: 23
```

De esta forma se optimizan la carga del vehículo, la coincidencia entre inventario y pedido, y la ruta de entrega en una interfaz de logística eficiente.

Referencias consultadas

- *Bellman, R. (1957). Programación dinámica. Princeton University Press.*
- *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introducción a los algoritmos (3.ª ed.). MIT Press.*
- *Vasquez, M., & Hao, J. K. (2001). Un enfoque híbrido para el problema de la mochila multidimensional 0-1. Actas del Simposio Internacional de Optimización Combinatoria.*
- *Erickson, J. (2019). Algoritmos. Universidad de Illinois.*
- *Papadimitriou, C. H., & Steiglitz, K. (1998). Optimización combinatoria: algoritmos y complejidad. Dover Publications.*
- *Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). Diseño y análisis de algoritmos de computadora. Addison-Wesley.*