INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO

2024

SESIÓN DE LABORATORIO 2: INTRODUCCIÓN A LA PROGRAMACIÓN DINÁMICA

ALGORITMOS BIOINSPIRADOS

Elaborado por: Flores Estopier Rodrigo

Profesor: Rosas Trigueros Jorge Luis

Grupo: 6CV2

Fecha de realización: 14/09/2024

Fecha de entrega: 17/09/2024

Semestre: 25-1

Contenido

1. MARCO TEÓRICO

La programación dinámica es una técnica de optimización que permite resolver problemas complejos dividiéndolos en subproblemas más simples. La idea central es evitar el cálculo repetido de subproblemas, almacenando sus resultados intermedios en una tabla o matriz para ser reutilizados cuando sea necesario. Este enfoque es particularmente útil en problemas que exhiben una propiedad llamada subestructura óptima, donde la solución global puede construirse a partir de soluciones óptimas a subproblemas más pequeños, y la superposición de subproblemas, donde los mismos subproblemas se repiten múltiples veces a lo largo del cálculo.

Principios de la Programación Dinámica

La programación dinámica se basa en dos principios clave:

1. **Descomposición en subproblemas**: El problema original se descompone en subproblemas que son más fáciles de resolver.
2. **Almacenamiento de soluciones**: Las soluciones de los subproblemas se almacenan para evitar recomputarlas, generalmente en una estructura de datos como una matriz.

Este enfoque puede aplicarse de manera bottom-up, donde los subproblemas más pequeños se resuelven primero y se utilizan para construir la solución a problemas más grandes, o de manera top-down mediante memoización, donde se resuelve el problema original de forma recursiva y se almacenan los resultados intermedios.

Problema de la Mochila 0/1

Uno de los problemas clásicos que se resuelve con programación dinámica es el problema de la mochila 0/1. Este problema puede plantearse de la siguiente forma: se tiene una mochila con una capacidad máxima de peso y una serie de objetos, cada uno con un peso y un valor asociado. El objetivo es determinar qué objetos se deben seleccionar para maximizar el valor total sin exceder la capacidad de la mochila.

El problema recibe el nombre de "0/1" porque para cada objeto solo se tienen dos opciones: incluirlo o no incluirlo. Este problema se resuelve eficazmente con programación dinámica, utilizando una tabla para almacenar las soluciones parciales. Cada entrada de la tabla representa la mejor solución para una capacidad dada y un

subconjunto de los objetos. De este modo, se construye la solución óptima en función de los resultados de subproblemas más pequeños.

Algoritmo para el Problema de la Mochila 0/1

Para resolver el problema de la mochila, se utiliza una tabla bidimensional donde las filas representan los objetos y las columnas las diferentes capacidades de la mochila. La entrada en la posición T[i][j] representa el valor máximo que se puede obtener con los primeros i objetos y una capacidad de j. El algoritmo sigue una estructura recursiva:

- Si el objeto actual i no se incluye en la mochila, el valor óptimo es igual al valor óptimo obtenido con i−1objetos.
- Si se incluye el objeto i, se suma su valor al valor óptimo obtenido para los objetos restantes con la capacidad reducida por el peso del objeto i.

Problema del Cambio

Otro problema resuelto eficientemente con programación dinámica es el problema del cambio. En este caso, dado un conjunto de monedas de diferentes denominaciones, se debe determinar el número mínimo de monedas necesarias para hacer un cambio exacto de una cantidad C. Este problema también exhibe una estructura óptima: la solución para una cantidad C puede construirse a partir de la solución para cantidades menores.

Algoritmo para el Problema del Cambio

El enfoque de programación dinámica utiliza una tabla unidimensional en la que cada entrada T[i] almacena el número mínimo de monedas necesarias para formar la cantidad i. Para calcular la entrada T[i], se evalúan todas las monedas disponibles y se selecciona la moneda que minimiza el número de monedas totales.

El algoritmo sigue la siguiente lógica:

1. Se inicializa una tabla con valores infinitos, excepto para el caso base donde no se necesita ninguna moneda para hacer el cambio de 0.
2. Para cada cantidad iii, se prueba cada moneda disponible y se actualiza la tabla con el número mínimo de monedas requerido para hacer el cambio.

La programación dinámica no solo es útil en los problemas mencionados, sino que también se aplica en áreas como la teoría de grafos, la optimización combinatoria y el procesamiento de lenguaje natural. Al permitir resolver problemas grandes de manera

eficiente, reduce la complejidad temporal en casos que de otra manera serían intratables con técnicas de fuerza bruta.

2. DESARROLLO

Se desarrollarán dos algoritmos utilizando las técnicas de la programación dinámica para encontrar la solución optima de los problemas de la mochila, y el problema del cambio. De nuevo se utilizará Python en el entorno de desarrollo de Google Colab.

PROBLEMA DE LA MOCHILA 0/1

El algoritmo que se utilizara para resolver este problema es el siguiente.
Se definirá a la matriz m[i,w] como el valor máximo que se puede obtener con un peso menor o igual a w utilizando elementos hasta i(primeros i elementos).
Se consideran los siguientes casos para asignar los valores de la matriz:

1. m[0,w] = 0; m[i,0] = 0
2. m[i,w] = m[i-1,w] si wi > w
3. m[i,w] = max(m[i-1,w], m[i-1, w-wi] + vi)

Se consideraron las siguientes variables que representan cada uno de los elementos del problema.
-v: Este es el conjunto de valores de los objetos, donde cada valor representa el beneficio asociado a incluir un objeto en la mochila.
-w: Esta es la lista de pesos de los objetos, donde cada peso corresponde al peso de un objeto.
- W: Es el peso máximo que la mochila puede soportar, es decir, la capacidad total disponible de la mochila.
- N = len(v): Es el número total de objetos disponibles (en este caso, 3).

Se crea una matriz m de tamaño (N+1) x (W+1), donde las filas representan los objetos disponibles (cada fila corresponde a un objeto) y las columnas las diferentes capacidades de la mochila, desde 0 hasta W (5 en este caso).

La matriz se inicializa con ceros, y cada celda almacenará el valor máximo que se puede obtener para un número dado de objetos y una capacidad de la mochila.

```
import numpy as np

v = (6,10,12)
w = (1,2,3)
W = 5

N= len(v)

m= np.zeros((N+1, W+1))

for row in range(1, N+1):
    for col in range(1, W+1):
      if w[row-1] > col:
        m[row][col]= m[row-1][col]
      else:
        m[row][col] = max(m[row-1][col], m[row-1][col -w[row-1]]+v[row-1])
```

El bucle for nos permite rellena la matriz siguiendo las siguientes reglas:

1. Se itera sobre todos los objetos disponibles (row) y sobre todas las capacidades posibles de la mochila (col).
2. Se compara el peso del objeto actual (w[row-1]) con la capacidad actual de la mochila (col).
    a. Si el peso del objeto actual es mayor que la capacidad de la mochila (w[row-1] > col):
        i. El objeto no se puede incluir. Por lo tanto, el valor óptimo en esa celda es igual al valor óptimo obtenido sin ese objeto:
    b. Si el objeto puede incluirse (w[row-1] <= col). Aquí se decide si es mejor incluir el objeto o no incluirlo:
        i. No incluirlo: El valor máximo será el mismo que el de la fila anterior (m[row-1][col]).
        ii. Incluirlo: Si el objeto se incluye, el valor se calcula como el valor del objeto actual (v[row-1]) más el valor óptimo que queda al reducir la capacidad de la mochila (m[row-1][col - w[row-1]]).

La decisión se toma seleccionando el valor máximo entre incluir o no el objeto.

Al final del algoritmo, la entrada m[N][W] contendrá el valor máximo que se puede obtener con los (N) objetos y una capacidad de mochila (W).

Posteriormente, se realizó un código para determinar cuales fueron los objetos que se tomaron en la mochila, a partir de la matriz construida en el código anterior.

```
i = N
j = W
objetos = []

while i > 0 and j > 0:
  if m[i][j] != m[i-1][j]:
    objetos.append(i-1)
    j = j - w[i-1]
  i = i - 1

print("Objetos seleccionados:", objetos)
```

El código utiliza un bucle while que recorre la matriz desde el último objeto y la capacidad máxima hacia el principio para identificar qué objetos se seleccionaron.
Este bucle continuará hasta que se hayan evaluado todos los objetos (i>0) o hasta que la capacidad de la mochila restante sea 0 (j>0).
La condición if verifica si el valor en la celda actual (m[i][j]) es diferente del valor en la celda correspondiente de la fila anterior (m[i−1][j]).

- Si los valores son iguales, significa que el objeto i-ésimo no fue incluido en la solución óptima para esa capacidad.
- Si los valores son diferentes, significa que el objeto i-ésimo fue incluido. Esto se debe a que, al incluir el objeto, el valor de la solución óptima se incrementa (el objeto añade valor).

Si el objeto fue seleccionado, se añade su índice (i−1) a la lista objetos. Esto indica que ese objeto fue incluido en la solución óptima.
Luego, se reduce la capacidad restante de la mochila en función del peso del objeto i-ésimo. Esto se debe a que, al incluir el objeto, ocupamos parte de la capacidad total, y ahora debemos calcular el valor óptimo para la capacidad restante.

Independientemente de si el objeto fue seleccionado o no, se decrementa el valor de i, lo que significa que se pasa a evaluar el siguiente objeto hacia atrás (del último al primero).

Ejecución 1

Valores de variables:

v = (6,10,12)

w = (1,2,3)

W = 5

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  6.,  6.,  6.,  6.,  6.],
       [ 0.,  6., 10., 16., 16., 16.],
       [ 0.,  6., 10., 16., 18., 22.]])
```

```
Objetos seleccionados: [2, 1]
```

Ilustración 1. Ejecución del algoritmo del problema de la mochila.

En la ejecución podemos visualizar la matriz con los valores asignados, donde la última celda nos da el valor máximo 22, posteriormente, nos muestra los índices de los objetos que fueron tomados, en este caso el índice 2 y 1, es decir el objeto de 10 de valor y 2 de peso, y el objeto de 12 de valor y 3 de peso.

Ejecución 2

Valores de variables:

v = (2,3,5,5,6)

w = (2,4,4,5,7)

W = 9

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 0.,  0.,  2.,  2.,  3.,  3.,  5.,  5.,  5.,  5.],
       [ 0.,  0.,  2.,  2.,  5.,  5.,  7.,  7.,  8.,  8.],
       [ 0.,  0.,  2.,  2.,  5.,  5.,  7.,  7.,  8., 10.],
       [ 0.,  0.,  2.,  2.,  5.,  5.,  7.,  7.,  8., 10.]])
```

```
Objetos seleccionados: [3, 2]
```

Ilustración 2. Segunda ejecución del algoritmo del problema de la mochila.

En la ejecución podemos visualizar la matriz con los valores asignados, donde la última celda nos da el valor máximo 10, posteriormente, nos muestra los índices de los objetos que fueron tomados, en este caso el índice 3 y 2, es decir el objeto de 5 de valor y 4 de peso, y el objeto de 5 de valor y 5 de peso.

PROBLEMA DEL CAMBIO

El algoritmo que se utilizara para resolver este problema es el siguiente. Considerando los siguientes casos.

1. m[1,N] = N;
2. m[i,N] = m[i-1,N] si di > N
3. m[i,N] = min(m[i-1,N], m[i, N-di] + 1)

Donde, además, la lista de las denominaciones debe tener las siguientes características:

- Deben estar en orden ascendente.
- La primer denominación debe ser 1.
- Existe una cantidad ilimitada de monedas.

Se tomaron en cuenta las siguientes variables:

- d: Estas son las denominaciones de las monedas disponibles.
- N: Es la cantidad total de dinero que se desea cambiar.
- W = len(d): Es el número de denominaciones de monedas disponibles.

Se crea una matriz de tamaño (W x N), donde las filas representan las diferentes denominaciones de monedas y las columnas representan las cantidades de dinero desde 1 hasta N.

```
d = [1,3,5]
N = 6

W= len(d)

m= np.zeros((W, N))

#Inicializar la primera fila
for col in range(N):
  m[0][col] = col + 1

#Inicializar la primera columna
for row in range(1, W):
  m[row][0] = 1

for row in range(1, W):
  for col in range(1, N):
    if d[row] > col+1:
      m[row][col]= m[row-1][col]
    else:
      m[row][col] = min(m[row-1][col], m[row][col -d[row]]+1)
```

La matriz m[row][col] almacenará el número mínimo de monedas necesarias para formar una cantidad exacta de dinero utilizando las monedas hasta la denominación representada por la fila (row).

En la primera fila, que corresponde a la moneda de denominación 1, el número de monedas necesario para formar cualquier cantidad es igual a esa cantidad, ya que solo se pueden usar monedas de 1 para formar todas las cantidades.

Por ejemplo, para obtener 3 unidades de dinero con solo monedas de 1, se necesitan exactamente 3 monedas. Así que la primera fila simplemente almacena el valor (col + 1) en cada celda.

Mientras que en la primera columna, donde la cantidad a formar es 1 unidad de dinero, siempre se necesitará exactamente 1 moneda, independientemente de la denominación, ya que la moneda de menor denominación (1) siempre puede formar 1 unidad.

Después, se utiliza un bucle que itera sobre todas las denominaciones de monedas y todas las cantidades de dinero para llenar la matriz con el número mínimo de monedas necesario. El algoritmo funciona de la siguiente manera:

1. Si el valor de la moneda actual (d[row]) es mayor que la cantidad que se está tratando de formar (col + 1), la moneda no puede incluirse. Por lo tanto, el número mínimo de monedas necesario será el mismo que el de la fila anterior (sin incluir esa moneda).

3. Si el valor de la moneda es menor o igual a la cantidad que se desea formar (d[row] <= col + 1), entonces se evalúan dos opciones:
    a. No incluir la moneda actual, ya que el número mínimo de monedas será el mismo que el de la fila anterior (m[row-1][col]).
    b. Incluir la moneda actual, ya que el número mínimo de monedas será igual al número mínimo de monedas para formar la cantidad (col - d[row]) (restando el valor de la moneda), más una moneda adicional.

La solución se toma eligiendo el mínimo de estas dos opciones.

Por ejemplo, si estamos tratando de formar 6 unidades de dinero (N = 6) utilizando las monedas de denominaciones d = [1, 3, 5]:
- En la primera fila (usando solo monedas de 1), necesitaríamos 6 monedas para formar 6 unidades.
- En la segunda fila (añadiendo monedas de 3), el algoritmo verifica si es mejor usar monedas de 3 o seguir usando solo monedas de 1.
- En la tercera fila (añadiendo monedas de 5), el algoritmo decide si conviene usar la moneda de 5 o una combinación de monedas de 1 y 3.

Al final del algoritmo, la celda m[W-1][N-1] contendrá el número mínimo de monedas necesarias para formar la cantidad N utilizando las monedas disponibles.

Posteriormente, se realizó un código para determinar cuántas monedas de cada denominación fueron tomadas para pagar, a partir de la matriz construida en el código anterior.

```
i = W-1
j = N-1
monedas = []

while i > 0 and j > 0:
  if m[i][j] != m[i-1][j]:
    monedas.append(i)
    j = j - d[i]
  else:
    i = i - 1

# Contar la cantidad de monedas de cada denominación
cantidad_monedas = {}
for moneda in monedas:
  if moneda in cantidad_monedas:
    cantidad_monedas[moneda] += 1
  else:
    cantidad_monedas[moneda] = 1

# Imprimir la cantidad de monedas de cada denominación
print("Cantidad de monedas de cada denominación:")
for moneda, cantidad in cantidad_monedas.items():
  print(f"Denominación {d[moneda]}: {cantidad}")
```

Este bucle itera a través de la matriz m para determinar qué monedas se utilizaron para formar el cambio:

- Si el valor en la celda actual m[i][j] es diferente del valor en la fila anterior m[i−1][j], significa que la moneda de denominación d[i] fue seleccionada. Esto se debe a que incluir la moneda modifica el valor de la solución óptima.

- Si la moneda se seleccionó, se añade su índice i a la lista monedas y despues, se actualiza la capacidad restante del cambio al restar el valor de la moneda seleccionada.
- Si el valor en m[i][j] es igual al de la fila anterior, significa que la moneda de denominación d[i] no fue utilizada, por lo que se pasa a la fila anterior.

Para contar la cantidad de monedas seleccionadas de cada denominación se toma en cuenta lo siguiente:

- Después de que se haya determinado qué monedas se seleccionaron, este bloque cuenta cuántas monedas de cada denominación se utilizaron.
- Se recorre la lista monedas, que contiene los índices de las denominaciones de monedas seleccionadas.
- Si una moneda ya está en el diccionario cantidad_monedas, se incrementa su valor (es decir, se añade una más de esa denominación); si no, se inicializa el conteo con 1.

Finalmente, se imprime cuántas monedas de cada denominación se usaron para formar el cambio.

Se recorre el diccionario cantidad_monedas y, para cada denominación de moneda, se imprime la cantidad correspondiente.

Ejecución 1

Valores de variables:

d = [1,3,5]

N = 6

```
array([[1., 2., 3., 4., 5., 6.],
       [1., 2., 1., 2., 3., 2.],
       [1., 2., 1., 2., 1., 2.]])
```

```
Cantidad de monedas de cada denominación:
Denominación 3: 2
```

Ilustración 3. Ejecución del algoritmo del problema del cambio.

En la ejecución podemos visualizar la matriz con los valores asignados, donde la última celda nos da el valor mínimo de monedas 2, posteriormente, la denominación de las monedas tomadas, y la cantidad de cada una de ellas, en este caso se tomaron 2 monedas de 3 para pagar 6. Podemos observar que la solución en la fila de arriba corresponde a lo presentado por el programa, la cual debería decir que debemos tomar una moneda de 5 y otra de 1. Sin embargo, la forma en la que esta hecho el algoritmo, se nos muestra la primera solución que se encuentra.

Ejecución 2

Valores de variables:

d = [1,2,5]

N = 7

```
array([[1., 2., 3., 4., 5., 6., 7.],
       [1., 1., 2., 2., 3., 3., 4.],
       [1., 1., 2., 2., 1., 2., 2.]])
```

```
Cantidad de monedas de cada denominación:
Denominación 5: 1
Denominación 2: 1
```

Ilustración 4. Segunda ejecución del algoritmo del problema del cambio.

En la ejecución, de nuevo podemos visualizar la matriz con los valores asignados, donde la última celda nos da el valor mínimo de monedas 2, posteriormente, la denominación de las monedas tomadas, y la cantidad de cada una de ellas, en este caso se tomaron una moneda de 5 y una de 2.

3. CONCLUSIONES

Durante esta práctica, se abordaron dos problemas clásicos de optimización utilizando programación dinámica: el problema de la mochila 0/1 y el problema del cambio. El desarrollo de ambos algoritmos permitió comprender a fondo cómo funciona la descomposición en subproblemas y la reutilización de soluciones intermedias, lo que resulta en una reducción significativa de la complejidad temporal en comparación con enfoques de fuerza bruta.

Una observación relevante es la importancia de la estructura de la matriz utilizada en la programación dinámica, ya que nos permite ver claramente la evolución de las soluciones parciales hacia la solución óptima. Además, el código para determinar qué objetos o monedas fueron seleccionados es útil para verificar la validez de los resultados y comprender el proceso de selección de los elementos óptimos.

Un experimento adicional realizado fue la modificación de las denominaciones de las monedas en el problema del cambio. Al cambiar el conjunto de denominaciones y ajustar el valor de N, se pudo observar cómo la matriz de programación dinámica se ajustaba para ofrecer la solución óptima. Este ejercicio demostró la flexibilidad del enfoque de programación dinámica, permitiendo soluciones eficientes para diversos problemas.

4. BIBLIOGRAFÍA

Eiben, A. E., & Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.