INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

EVOLUTIONARY COMPUTING

PRÁCTICA 2: PROGRAMACIÓN DINÁMICA

ALUMNO: ORTEGA VICTORIANO IVAN

PROF.: JORGE LUIS ROSAS TRIGUEROS

FECHA DE REALIZACIÓN DE LA PRÁCTICA: 06/03/2018

FECHA DE ENTREGA DEL REPORTE: 14/03/2018

MARCO TEÓRICO.

La programación dinámica se utiliza para mejorar el rendimiento de algunos algoritmos. En ciertos casos, cuando se divide un problema en varios subproblemas, resulta ser que éstos no son independientes entre sí. Cuando se resuelve un subproblema traslapado con otros, se repite una parte de la solución que ya se encontró al resolver otro subproblema. Cuando se pretende subdividir un problema con la técnica divide y vencerás y los subproblemas no son independientes entre sí, es muy probable que el tiempo para hallar la solución crezca de forma exponencial con el tamaño de la entrada. Es decir, si no hay manera de dividir el problema en un pequeño número de subproblemas independientes entre sí, tendremos que resolver el subproblema varias veces, lo que quizá produzca un algoritmo de tiempo exponencial. [1]

Es obvio que solucionar el mismo problema varias veces no es eficiente. La programación dinámica consiste en conservar la solución a cada subproblema que ya se ha resuelto en una tabla (puede ser un arreglo o una matriz), para tomarla cuando ésta se requiera. Esto reduce el tiempo necesario para obtener el resultado final, pues evita repetir algunos de los cálculos. Entonces, una característica del paradigma de la programación dinámica es que se utilizan tablas como una estructura auxiliar para evitar el solapamiento, es decir, evitar calcular varias veces un mismo resultado. [1]

“En la programación dinámica normalmente se empieza por los subcasos más pequeños, y por tanto más sencillos. Combinando sus soluciones, obtenemos las respuestas para subcasos de tamaños cada vez mayores, hasta que finalmente llegamos a la solución del caso original”.[2]

El principio de subestructuras óptimas significa que la solución a un problema puede ser construida usando soluciones óptimas para subproblemas. El algoritmo sería: [3]

- Divide el problema en pequeños subproblemas.
- Resuelve los subproblemas utilizando este algoritmo de 3 pasos recursivamente hasta que las soluciones sean triviales.
- Usa las soluciones a los subproblemas para construir la solución al problema original.

Un problema con subproblemas superpuestos utiliza la solución de un subproblema para llegar a la solución de múltiples problemas mayores. [3]

Dos aproximaciones pueden ser utilizadas:

- Top-Down: Dado un problema, dividirlo en subproblemas encontrando las soluciones y guardándolas para su uso a futuro. Combina recursión y memorización.
- Bottom-up: Todos los posibles subproblemas son resueltos y las soluciones son usadas para construir soluciones a problemas más grandes.

Principio de Bellman

Dada una secuencia de decisiones óptimas, cada subsecuencia es óptima también. [3]

La programación dinámica es útil cuando la subdivisión de un problema lleva a:

- Una gran cantidad de problemas
- Problemas con soluciones que se superponen.

MATERIAL Y EQUIPO.

- Equipo de cómputo (PC o laptop).
- Python2 o Python3

DESARROLLO DE LA PRÁCTICA.

Para esta práctica se nos dejó implementar los algoritmos para resolver el *Knapsack Problem 0/1* y el *Coin Changing Problem* utilizando programación dinámica.

En el caso del primero, para construir la tabla con las soluciones el algoritmo dados el número de objetos con su respectivo peso w, y la capacidad máxima de peso de la mochila W, fue el siguiente:

1. `Crear una matriz M de tamaño (número de objetos +1)*(W+1)`
2. `Llenar la primera fila y la primera columna con ceros`
3. `Para cada i desde 1 hasta (número de objetos +1), hacer:`

3.1. Para cada j desde 1 hasta (W +1)

3.1.1. Si j - w[i-1] < 0 (El objeto no cabe en la mochila), hacer:

3.1.1.1. M[i][j] = M[i-1][j] (No hay cambios con respecto al valor anterior)

3.1.2. En caso contrario

3.1.2.1. M[i][j] = máximo (a, b)
Donde:
a = M[i-1][j], El valor óptimo anterior
b = M[i-1][j-w[i-1]]+v[i-1], El valor óptimo para un W = W – w[i-1], sumándole el valor que aporta el i-ésimo elemento

Definido el algoritmo para la construcción de la tabla, se implementó además un algoritmo para saber qué elementos fueron los que se utilizaron en la solución óptima. El cuál es el siguiente:

1. Creamos una lista vacía que contendrá los elementos usados en la solución.
2. Hacemos i = tamaño de M -1, j = W
3. Mientras i, j sean mayores a 0, hacer:

3.1. Si M[i][j] es distinto a M[i-1][j] (El elemento de la fila i aporta un valor mayor que el anterior), hacer:

3.1.1. Agregar el elemento a la lista de elementos para la solución.

3.1.2. Hacer j = j – w[i-1]

3.1.3. Hacer i = i – 1

3.2. En caso contrario

3.2.1. Hace i = i – 1

Para el caso del Coin Changing Problem dados el número de monedas, las denominaciones de las monedas (d) y el valor N para el cambio, el algoritmo utilizado fue el siguiente:

1. Crear una matriz M de tamaño (número de monedas +1)*(N+1)
2. Llenar la primera fila con un valor muy grande (infinito) y la primera columna con ceros
3. Para cada i desde 1 hasta (número de monedas +1), hacer:

3.1. Para cada j desde 1 hasta (N +1), hacer:

3.1.1. Si d[i-1] == j, hacer:

3.1.1.1. M[i][j] = 1 (La denominación de la moneda es igual al valor a dar de cambio)

3.1.2. De lo contrario, si d[i-1] > j, hacer:

3.1.2.1. M[i][j] = M[i-1][j] (No se puede agregar la moneda, y no hay aportación)

3.1.3. En caso contrario

3.1.3.1. M[i][j] = mínimo (a, b)
Donde:
a = M[i-1][j], El valor óptimo anterior
b = M[i-1][j-d[i-1]]+1, El número de monedas usadas para un N=N–d[i-1], sumándole una moneda más (la moneda i-ésima).

Por último, para la saber que monedas se utilizaron se empleó el siguiente algoritmo:

```
1. Creamos un diccionario vacío que contendrá los elementos usados en la solución.
2. Inicializamos el diccionario con sus entradas igual a la denominación de las monedas
   y su valor en cero.
3. Hacer i = tamaño de M -1
4. Hacer j = N
5. Mientras i, j sean mayores a 0, hacer:
   5.1. Si M[i][j] es menor a M[i-1][j], hacer:
      5.1.1. Agregar el elemento al diccionario de elementos para la solución.
      5.1.2. Hacer j = j – d[i-1], (El elemento de la fila i contribuye a usar menos
           monedas)
   5.2. En caso contrario
      5.2.1. Hace i = i – 1
```

Ya implementados nuestros algoritmos en Python, se realizan las siguientes pruebas con los ejemplos vistos en clase.

Para el KP 0/1 se utilizaron los siguientes valores:

- w = [2,3,4,4,5]
- v = [3,3,3,5,6]
- W = 10

Para el Coin Changing Problem los siguientes valores:

- d = [1, 2, 3]
- N = 10

Los resultados de las pruebas se muestran en la figura 1.

```
Knapsack Problem 0/1
Solucion:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 3, 3, 3, 3, 3, 3, 3, 3]
[0, 0, 3, 3, 3, 6, 6, 6, 6, 6]
[0, 0, 3, 3, 3, 6, 6, 6, 6, 9, 9]
[0, 0, 3, 3, 5, 6, 8, 8, 8, 11, 11]
[0, 0, 3, 3, 5, 6, 8, 9, 9, 11, 12]
Se utilizaron los siguientes elementos:
[5, 2, 1]
Coin Change Problem
Solucion:
[9223372036854775807, 9223372036854775807, 9223372036854775807, 9223372036854775807, 9223372036854775807, 9223372036854775807,
9223372036854775807, 9223372036854775807, 9223372036854775807, 9223372036854775807, 9223372036854775807]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
[0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4]
Monedas utilizadas (moneda , # de monedas usadas):
1 0
2 2
3 2
```

Figura 1. Resultados de la ejecución del script de Python.

CONCLUSIONES.

Al inicio me costó trabajo el dominar como se construían las tablas visualizándolo como instrucciones para un programa, ya que de forma intuitiva resulta más sencillo, y me resultaba aún más difícil en reconstruir la posible solución óptima. De hecho, muchas páginas donde explican este tipo de problemas y resolviéndolos con programación dinámica omiten este paso de reconstruir la solución, que creo yo es lo más importante. Analizando y maldiciendo por un buen rato, logré dar con las soluciones para la reconstrucción, y en lo personal, esta práctica me fue de utilidad para resolver esas dudas que llegué a tener en su momento sobre programación dinámica en la materia de Análisis de Algoritmos.

Referencias

[1] M. d. C. Gómez Fuentes y J. Cervantes Ojeda, Introducción al análisis y al diseño de algoritmos, Ciudad de México: Universidad Auntónoma Metropolitana, 2014.

[2] G. Brassard y P. Bratley, Fundamentos de Algoritmia, Prentice Hall, 2008.

[3] J. L. R. Trigueros, *Apuntes de clase: "Dynamic Programming",* 2018.