Practica 3. “Algoritmos voraces y programación dinámica”

Angel Zait Hernández López
No. Boleta: 2014080682
Análisis de Algoritmos, Grupo: 3CM4

16 de marzo de 2020

Resumen

Un algoritmo voraz (también conocido como ávido, devorador o goloso) es aquel que, para resolver un determinado problema, sigue una meta-heurística, consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima [2]. La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas [3].

Palabras clave: Algoritmo, voraz, programación, dinámica.

1. Introducción

Los algoritmos voraces suelen ser muy eficientes, sin embargo, existen problemas dónde no encuentran una solución óptima. Una alternativa es el uso de técnicas de programación dinámica.

1.1. Problema de devolver el cambio.

El problema consiste en desarrollar un algoritmo para pagar una cierta cantidad de dinero a un cliente, empleando el menor número posible de monedas (Algoritmo 1).

Solución con programación dinámica. Utlizamos una tabla de $C[1, ..., n], [0, ..., N]$ $(C[i, j])$, donde i (filas) representa la denominación de las monedas y /o billetes y j (columnas), define la cantidad por pagar. El valor $c[i, j]$ registra el mínimo de monedas necesarias para pagar una cantidad de j unidades, con $0 \leq j \leq N$, empleando monedas de las denominaciones desde 1 hasta i con $1 \leq i \leq n$. Sea $c[i, 0] = 0$ para cualquier i. La estrategia que sigue la programación dinámica para determinar una función que nos permita llenar la tabla, es de la siguiente:

$$c[i, j] = 1 + c[i, j - d_i] \tag{1}$$

para seleccionar la mejor alternativa:

$$c[i, j] = min(c[i - 1, j], 1 + c[i, j - d_i]) \tag{2}$$

Al utilizar la función anterior exiten posiciones fuera de la tabla, por ejemplo, cuando $i = 1^j < d_i$, para seleccionarlo, asignaremos el valor de $+\infty$.

1.2. El problema de la mochila fraccionaria

Nos dan n objetos y una mochila. Para $i = 1, 2, ..., n$, el objeto i tiene un peso positivo w_i y valor positivo v_i. La mochila puede llevar un peso que no sobrepase W. Nuestro objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de capacidad de la mochila. En una versión del problema, se puede suponer que se puede tomar un trozo de los objetos, de manera que podamos decidir llevar solo una fracción x de estos (la versión del problema no permite fraccionar objetos, es más compleja). En este caso, el objeto i contribuye en x, w, al peso total de la mochila, y en x, v el valor de la carma. Matemáticamente, el problema está definido por:

$$Maximizar \sum_{i=1}^{n} x_i v_i \tag{3}$$

sujeto a:

$$\sum_{i=1}^{n} x_i w_i \leq W \tag{4}$$

Dónde:
$v_i > 0$,
$w_i > 0$ y
$0 \leq x_i \leq 1$ para $1 \leq i \leq n$.

2. Marco Teórico

2.1. Algoritmo voraz

En los algoritmos voraces, tenemos que resolver algún problema de forma óptima. Para construir la solución de nuestro problema, disponemos de un conjunto de candidatos. A medida que avanza el algoritmo, vamos acumulando dos conjuntos. Uno contiene candidatos que ya han sido considerados y seleccionados, mientras que el otro contiene candidatos que han sido considerados y rechazados.
Existe una función que comprueba si un cierto conjunto de candidatos constituye una solución de nuestro problema, ignorando si es o no óptima por el momento. Hay una segunda función que comprueba si un cierto conjunto de candidatos es factible, esto es, si es posible o no completar el conjunto añadiendo otros candidatos para obtener al menos una solución de nuestro problema.
Hay otra función, la función de selección, que indica en cualquier momento cuál es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados. Por ultimo está una función objetivo, que da el valor de la solución que hemos encontrado. Para resolver nuestro problema, buscamos un conjunto de candidatos que constituya una solución, y que optimice el valor de la función objetivo.
Los algoritmos voraces avanza paso a paso. Inicialmente, el conjunto de elementos seleccionados está vacío. Entonces, en cada paso se considera añadir a este conjunto el mejor candidato sin considerar los restantes, estando guiada nuestra elecci ón por la función de selección. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible, rechazamos el candidato que estamos considerando en ese momento [1].

2.2. Programación dinámica

En informática, la programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas, como se describe a continuación. Una sub estructura óptima significa que se pueden usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto. Por ejemplo, el camino más corto entre dos vértices de un grafo se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
3. Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.
Decir que un problema tiene subproblemas superpuestos, es decir, que se usa un mismo subproblema para resolver diferentes problemas mayores. Por ejemplo, en la sucesión de Fibonacci ($F_3 = F_1 + F2$ y $F_4 = F_2 + F_3$) calcular cada término supone calcular F_2. Como para calcular F_5 hacen falta tanto F_3 como F_4, una mala implementación para calcular $F5$ acabará calculando $F2$ dos o más veces. Esto sucede siempre que haya subproblemas superpuestos: una mala implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a problemas que ya han sido resueltos anteriormente.
Esto se puede evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de

la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se llama memoización. Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que de antemano sabemos que vamos a necesitar [4].

3. Desarrollo de la práctica

El lenguaje que se utilizó para el desarrollo de esta práctica fue C++, ya que su velocidad es mayor a comparación de otros lenguajes, como lo es Python ó Java. Esto se tomó a consideración por el tipo de problema que se nos presenta, porque puede que exista un problema muy grande y se quiera aproximar a la mejor solución de una manera rápida. Además de que se optó usar clases, ya que ambos tipos de algoritmos usan las mismas entradas, por lo que es más fácil inicializar un objeto con sus atributos correspondientes.

3.1. Problema de devolver el cambio

Para los datos de entrada, se utiliza un archivo de texto, el cual contiene los valores de las monedas a utilizar para dar el cambio. Estos datos están en varias filas, es decir, que cada una de las monedas debe estar en una linea diferente para poder leerlas y así poder ejecutar el programa, no importa si están ordenados o no los datos, ya que el programa lo hará por su cuenta.

3.1.1. Algoritmo voraz

Para la solución de este problema, se utilizó el algoritmo 1, el cual ingresará un conjunto de datos, los cuales serán el tipo de moneda que se manejará para dar el cambio exacto o aproximarlo y además la cantidad que se desea que cambie (n). Se tendrán dos variables extras. S que representa un conjunto en el cual se estarán almacenando los valores de la moneda que se necesiten para dar el cambio, y s que representa la suma de las monedas de cambio del conjunto S, es decir, que tanto estamos dando de cambio al usuario.
Se iniciará un ciclo el cual no se dentendrá mientras que la suma (s) sea diferente de cero. se seleccionará el valor máximo de D (conjunto de monedas de entrada), tal que la adición de x (valor mayor del conjunto D) más la suma sea menor a la cantidad del que queremos el cambio. De esta forma, se recorrerá todo el conjunto si es necesario para poder dar el cambio exácto.
Si no existe el elemento en el conjunto, o bien, ya no hay elementos por revisar dentro del conjunto, el algoritmo mandará un mensaje al usuario de que no encontró una solución (cambio exacto) al problema; de otro modo, al conjunto de soluciones se le agregará el valor tomado de la moneda x, y se acumulará en el valor de s. Todo esto se repetirá hasta que la suma sea igual a cero o bien, se cumpla la condición de la linea 7.

3.1.2. Programación dinámica

Para resolver el problema del cambio con programación dinámica, se utiliza el algoritmo 2, donde en lugar de devolver una arreglo, dará en s lugar una matriz, donde se almacenarán las posibles combinaciones para dar el mejor cambio al usuario.
Para esto primero se llenará el todas las filas del valor cero, y el valor de las columnas será el valor de la columna, es decir, el valor de la unidad, hasta llegar al valor de n, el cambio al que se desea entregar.
Después de eso, se va recorriendo toda la matriz dato por dato, guiándonos del algoritmo propuesto, podemos decir que j es una cantidad que también queremos dar en cambio; como vemos en la linea 10 del algoritmo 2.
Comparamos si este dato es menor a algún tipo de moneda del conjunto de iniciación, ya que de ser así, el valor que se tomará para esa celda es el valor de la celda anterior a ella, puesto que al pertenecer o no a ese conjunto, puede tener la misma la cantidad de veces a repetir de dicho valor. Con lo anterior podemos obtener el menor cambio posible, veamos un ejemplo comparando el algoritmo voráz y programación dinámica.
Supongamos que tenemos un conjunto de denominación de monedas $D = \{5, 4, 1\}$ y nosotros queremos que nos de un cambio de una moneda con valor $n = 8$.
El cambio devuelto por el algoritmo voraz sera de un conjunto de solución $S = \{5, 1, 1, 1\}$, por lo que tendremos una cantidad de 4 monedas, en la vida real, nosotros buscamos dar el cambio con una m cantidad de monedas a dar, aquí es donde la programación dinámica, ya que al momento de revisar todas las monedas del conjunto D, esto nos dará como resultado un conjunto de solución

$S = \{4, 4\}$, y como podemos notar, la cantidad de monedas a devolver es 2, menor al del algoritmo voraz. En la sección 4 se puede observar algunas pruebas como la que se acaba de explicar.

3.2. Problema de la mochila

Al igual que en el problema anterior del cambio, se utiliza un archivo de texto para guardar los pesos y los valores de dichos pesos, solo que esta vez irán dos valores por línea, dónde el primer valor representa el peso y el segundo son los valores.

3.2.1. Mochila fraccionaria (algoritmo voraz)

Para esta parte de la practica se utilizó el algoritmo 3, el cual consiste en guardar el mejor valor en una mochila de capacidad W. El algoritmo recibirá un arreglo con valores de pesos w, que son requeridos guardar en la mochila, y un arreglo con valores v que nos van a decir que tan recomendable es guardar esos pesos.
Todos los valores se guardarán en otro arreglo que tendrá el tamaño de la longitud del arreglo de pesos. También tenemos una variable auxiliar que es $peso$, y es un acumulador para saber si ya nos pasamos o no del peso de la mochila.
Para elegir un peso adecuado, primero debemos saber el mejor objeto restante que hay en el arreglo, el cual podemos obtener tres casos:

1. Minimizar w: Esto significa que escogeremos del arreglo w el valor mínimo para poder almacenarlo en la mochila, una vez elegido este, se descartará y se escogerá el siguiente peso más bajo.

2. Maximizar v: Es decir, que en caso contrario de punto anterior, se tomará un peso, pero esto se hará con base al valor positivo de dicho peso, por lo que dependiendo quien tenga ese valor mayor al de los demás, se elegirá ese peso.

3. Maximizar $\frac{v}{w}$: Igual que el punto anterior, tomar el valor máximo resultante de la división del valor entre el peso, así tomando el valor máximo y elegir el peso de dicha posición.

De los casos anterior, se tomará la posición en la que se encuentra ese valor que nos conviene.
Si el valor del peso acumulado más el peso elegido es menor al peso de la mochila, entonces el valor de solución en esa posición será cero y se adicionará el peso del valor que se eligió, de caso contrario, la solución en esa posición será el peso de la mochila menos el peso acumulado entre el peso tomado del arreglo y el peso acumulado tomará el peso de la mochila. Todo esto se repetirá las veces necesarias hasta que el peso acumulado sea mayor o igual al peso de la mochila.

3.2.2. Mochila entera (programación dinámica)

La resolución de este problema es similar al de la sección 3.1.2, la parte que cambia es la línea 7 del algoritmo 2 que en lugar de tener los valores de i se tiene el valor de 1 (algoritmo 4), ya que solamente puede caber ese primer objeto solo una vez. Como sigue, este algoritmo hace una tabla en la cual va viendo cual es la mejor opción para poder guardar los objetos en la mochila sin excederce de ella, como podemos observar, revisa los valores anteriores para poder saber si le conviene o no tomar la decisión, y de ser así cual le conviene más.

4. Pruebas

En esta sección se hicieron un par de pruebas con datos específicos, aquí podremos ver y comparar los algoritmos voraces y la programación dinámica en los problemas a resolver. Para mayor información de ejecución, se anexa a la práctica un archivo "README.txt"donde se puede leer las especificaciones de como compilar y correr el programa.

4.1. Problema de devolver cambio

Para este problema se tomaron los conjuntos $D = \{100, 25, 10, 5, 1\}$ y $D = \{6, 4, 1\}$ para devolver el cambio. Para poder ver mejor la competencia de ambos algoritmos, se omitieron todas las impresiones de las soluciones, y solamente tomamos el tiempo del algoritmo completo.
Como podemos notar, no hay mucha diferencia en tiempos para el primer conjunto, comparando la taba 2 y la tabla 3, cuando el cambio es pequeño, es más rápido, como lo podemos ver en el valor 8, pero no podemos ver a

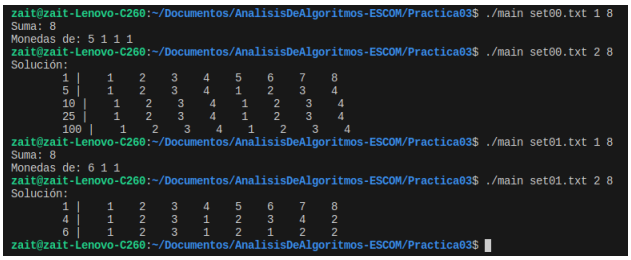

Figura 1: Prueba del problema devolver cambio.

detalle que método es más rápido.
La diferencia que hay entre la programación voraz entre la programación dinámica es la implementación, y con ella se puede observar que los valores de la denominación para la programación voraz de estar de mayor a menor, para poder devolver el mejor cambio, caso contrario de la programación dinámica, que debe tener la denominación mínima primero para así poder trabajar con las combinaciones posibles y dar el mejor cambio.
Observemos la figura 1, Se hicieron pruebas con el ejemplo de devolver en cambio el valor de 8 con las distintas denominaciones, podemos ver que en la primera parte, se devuelve un cambio completo con los valores de $\{5, 1, 1, 1\}$, pero con la programación dinámica, podemos observar en la que la mejor opción es devolver 8 monedas de valor 1, pero en la vida real no nos conviene eso.
Ahora veremos el siguiente conjunto, con el algoritmo voraz nos da un conjunto de solución de $\{6, 1, 1\}$ el cual no es malo dado al conjunto de monedas que le dimos, pero veamos la programación dinámica, en este caso, la mejor solución es dar dos monedas de valor 4 el cual es mucho más conveniente si lo que buscamos es dar el cambio con el menor número de monedas.

4.2. Problema de la mochila

Para este problema, veremos como es que funcionan ambos métodos con un solo conjunto de pruebas, el cual, se puede apreciar en la tabla 1:

Recordemos el algoritmo voraz, vamos a ver cual de todos nos conviene dependiendo el problema deseado, en este caso, el programa nos imprime tres diferentes arreglos o listas (figura 2). A continuación, enlistamos en orden el significado de cada linea para el algoritmo voraz:

$W = 11$					
w	1	2	5	6	7
v	1	6	18	22	28
$\frac{v}{w}$	1	3	3.6	3.66	4

Tabla 1: Conjunto de prueba para el problema de la mochila

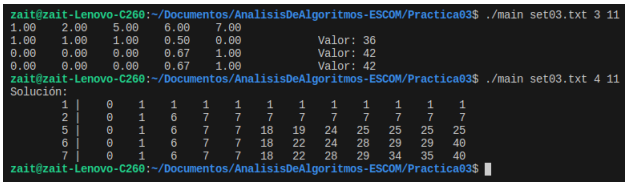

Figura 2: Prueba del problema devolver cambio.

1. Pesos del conjunto de prueba.
2. Minimización de pesos
3. Maximización de valores
4. Maximización de $\frac{v}{w}$

Como recordaremos, se le dice que es una mochila fraccionaria, ya que el valor de los pesos se puede dar por partes, es decir, que podemos guardar la mitad o una n-ésima parte de un objeto, por ello es que podemos notar números decimales, en este caso en particular, debemos fijarnos que nos conviene más, pero siempre guiándonos con el valor que se va acumulando, y por conveniencia, tomando el mayor.
Notemos que al minimizar los pesos obtenemos un valor de 36 y logramos guardar tres objetos y la mitad de otro. Si bien se guardan bastantes objetos no podemos decir que sea el mejor resultados. Ahora bien, si maximizamos los valores y la división de $\frac{v}{w}$ estos tienen el mismo valor que es 42, pero solamente guardamos un objeto y dos terceras partes (aproximadamente) de otro objeto, a consecuencia, el valor de estos objetos es mayor y convendría más tener estos objetos en la mochila que si minimizamos los pesos.

Ahora, con respecto a la programación dinámica, es similar al del cambio, este va ir viendo peso por peso hasta llegar al limite de la mochila, que tanto conviene guardar

ese peso tomando en cuenta su valor, en el caso anterior, podemos notar que en el algoritmo 3 solamente utilizamos el valor del peso, mientras que en la mochila entera, nos apoyamos del valor de este, así también teniendo valores más exactos y guardando los objetos de una manera completa y no por partes como es el algoritmo voraz.

5. Conclusiones

Podemos notar que dependiendo del problema a resolver, se puede optar por cualquier método, la diferencia es la implenentación y la visión de los multiples resultados que podemos tomar, como es el caso de la programación dinámica, que en ambos problemas hace una tabla para ver cual es el mejor camino para poder llegar a la solución, y claro , el tiempo de ejecución no es mucho, que es una de las grandes ventajas y que siempre todos buscamos para poder resolver un problema.
Se podría decir que la mejor manear de resolver un problema es con programación dinámica, por su flexibilidad de soluciones y que trata de acercarse a lo más real posible, y que si se necesitan resultados que se necesitan obtener más de una vez, este ya los tiene almacenados para volverlos a utilizar.
Pero como siempre, es cuestión de gustos y que debemos saber en qué situaciones debemos usar un método u otro.

Algorithm 1: Algoritmo voráz de devolver el cambio

```
Data: D : denominaciones de las monedas y/o
      billetes, n : Valor que se desea dar cambio
1  begin
2      DevolverCambio(D,n)
3      S ← ∅, es el conjunto que obtendrá la solución.
4      s ← 0, s es la suma de los elementos en S.
5      while s ≠ 0 do
6          x ← el mayor elemento de D tal que
           x + s ≤ n
7          if no existe el elemento then
8              devolver "NO ENCUENTRO UNA
               SOLUCIÓN
9          else
10             S∪ moneda de valor x
11             s ← s + x
12     Devolver S
13 end
```

Algorithm 2: Programación dinámica de devolver el cambio

```
Data: D : denominaciones de las monedas y/o
      billetes. n : Valor que se desea dar cambio
1  begin
2      DevolverCambio(D,n)
3      c[m][n], matriz de tamaño m × n, donde m es la
       cantidad de monedas de cambio.
4      for i ← 0 to (m + 1) do
5          c[i][0] ← 0
6      for i ← 1 to (n + 1) do
7          c[0][i] ← i
8      for i ← 1 to (m + 1) do
9          for j ← 1 to (n + 1) do
10             if j < D[i] then
11                 c[i][j] = c[i − 1][j]
12             else
13                 c[i][j] =
                   min(c[i − 1][j], 1 + c[i][j − D[i]])
14     Devolver c
15 end
```

Algorithm 3: Algoritmo voráz de la mochila

```
Data: w : lista de pesos positivos, v : lista de valores
      positivos, W : peso máximo
1  begin
2      Mochila(w,v,W)
3      s ← 0, es una lista de los pesos que tomará la
       solución del problema.
4      peso ← 0, suma de los pesos w
5      i ← 0
6      while peso < W do
7          i ← el mejor objeto restante
8          if (peso + w[i] ≤ W) then
9              s[i] ← 1
10             peso ← peso + w[i]
11         else
12             s[i] ← (W − peso) / w[i]
13             peso ← W
14     Devolver s
15 end
```

Algorithm 4: Programación dinámica de la mochila

```
Data: w : lista de pesos positivos, v : lista de valores
      positivos, W : peso máximo
1  begin
2      Mochila(w,v,W)
3      c[m][n], matriz de tamaño m × n, donde m es la
       cantidad de monedas de cambio.
4      for i ← 0 to (m + 1) do
5          c[i][0] ← 0
6      for i ← 1 to (n + 1) do
7          c[0][i] ← 1
8      for i ← 1 to (m + 1) do
9          for j ← 1 to (n + 1) do
10             if j < w[i] then
11                 g[i][j] = g[i − 1][j]
12             else
13                 if
                   g[i−1][j] ≥ (g[i−1][j−w[i]]+v[i])
                   then
14                     g[i][j] = g[i − 1][j]
15                 else
16                     g[i][j] = g[i − 1][j − w[i]] + v[i];
17     Devolver c
18 end
```

Denominaciones	**Devolver**					
	8	125	47	642	1025	356
$D = \{100, 25, 10, 5, 1\}$	0.006	0.005	0.004	0.005	0.006	0.005
$D = \{6, 4, 1\}$	0.006	0.004	0.005	0.005	0.005	0.006

Tabla 2: Tiempo en segundos del algoritmo voráz para devolver cambio

Denominaciones	**Devolver**					
	8	125	47	642	1025	356
$D = \{100, 25, 10, 5, 1\}$	0.005	0.005	0.006	0.006	0.006	0.006
$D = \{6, 4, 1\}$	0.005	0.006	0.005	0.004	0.005	0.005

Tabla 3: Tiempo en segundos de programación dinámica para devolver cambio

Referencias

[1] BRASSARD, G., AND BRATLEY, P. *Fundamentos de Algoritmia*. Départament d'informatique et de recherche opérationelle. Université de Montréal, 1997.

[2] ECURED. Algoritmos voraz, https://www.ecured.cu/algoritmos_voraz.

[3] ECURED. Programación dinámica, https://www.ecured.cu/programación_dinámica.

[4] WIKIPEDIA, L. E. L. Programación dinámica, https://es.wikipedia.org/wiki/programación_dinámica.