DEPARTAMENTO
INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA UNIVERSIDAD DE CONCEPCIÓN

Análisis de Algoritmos: Programación Dinámica y Algoritmos Aleatorizados

Alumnos: Fabián Cid Escobar
Rodolfo Fariña Reisenegger

Profesora: Cecilia Hernández Rivas

12 de julio del 2020, Concepción

Introducción

Dentro de los problemas que son resueltos por los algoritmos, algunos de ellos llegan a escalar de una manera que no permite que sean resueltos por código más tradicional. En estos casos, se recurre a la programación dinámica y los algoritmos aproximados.

La programación dinámica permite resolver problemas que tengan gran overlap de subproblemas, así evitando calcular una y otra vez la misma parte. Mientras que, los algoritmos aproximados sacrifican precisión a favor del tiempo de ejecución.

En este breve proyecto, nos enfrentamos al famoso problema de la mochila, en particular, la mochila 0/1, llamado así porque los objetos van o no van dentro de esta.

Este problema consiste en una mochila con una capacidad máxima, y una serie de objetos, con el objetivo de maximizar el valor de la suma de los objetos que uno mete en esta mochila.

a) Describa y proporcione una solución usando programación dinámica y establezca su complejidad asintótica del tiempo de ejecución

Primero comenzaremos con una breve descripción del problema de la mochila. El problema de la mochila consiste en, dado un cierto conjunto de ítems, cada uno con un peso w_i, valor v_i, y la mochila con una capacidad de peso máximo C, qué subconjunto de ítems meter en la mochila para maximizar el valor de los objetos que están dentro de ella. Notar que a la mochila se le pueden agregar más dimensiones, como por ejemplo el volumen, complicando el problema aún más. En este caso, nos limitaremos a trabajar con el peso y valor de los objetos y la capacidad máxima de la mochila.

Con este tipo de problema, la cantidad de combinaciones posibles es increíblemente alta y resolverlo mediante fuerza bruta tiene complejidad 2^n debido a que hay subproblemas que deben calcularse más de una vez, lo cual incrementa por mucho el tiempo de ejecución.

Para ello utilizaremos programación dinámica, en donde calculamos cada subproblema sólo una vez y lo guardamos en una tabla, luego si el subproblema aparece otra vez, sólo accedemos a la tabla para tomar el resultado ya guardado. Esto disminuye el tiempo de ejecución pero a su vez, el uso de memoria es mayor.

Al momento de crear la tabla, consideramos que el “eje X” es la capacidad restante de la mochila (desde 0 hasta C) y el “eje Y” la cantidad de objetos que disponemos para escoger (desde 0 hasta n), donde en cada par (x,y) de la tabla guardaremos el mayor valor posible de entre los objetos considerando si debemos tomar o no el objeto y_i.

Para visualizar mejor, se presenta un ejemplo:

Si disponemos de una mochila que puede almacenar 8 kg y tenemos para escoger 4 objetos cuyos pesos y valores se muestran a continuación

Objeto →	**1**	**2**	**3**	**4**
Peso (kg) →	2	3	4	5
Valor →	1	2	5	6

Luego, creamos la siguiente tabla

Objetos	**Capacidad de Mochila**								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
1 y 2	0	0	1	2	2	3	3	3	3
1, 2 y 3	0	0	1	2	5	5	6	7	7
1, 2, 3 y 4	0	0	1	2	5	6	6	7	8

Entonces, nuestra tabla tendrá nC casillas y para cada una de estas habrá que analizarla como un subproblema. En cada una de estas casillas sólo se usará uno de los tres casos, los cuales son:

1. Si $i = 0$ o w_i = 0, en este caso, solo estamos iniciando la casilla y se anota un 0. Inicializamos la primera fila de la tabla con ceros.
2. Si $w_i > w$, entonces el objeto no cabe, y solo optamos por la solución óptima anterior. Esto es como “tomar la solución de la casilla de arriba”. Lo cual corresponde a $tabla[i-1][j]$.
3. Si $w_i \leq w$ y $i \neq 0$, el objeto cabe en la mochila, y se toma el valor máximo entre la solución óptima anterior y tomar el objeto más la solución óptima con el peso restante. Lo cual se puede llevar a una expresión matemática como: $max(tabla[i-1][j],\ v[i-1] + tabla[i-1][j - pesos[i-1]])$. Aquí se “toma el máximo entre la casilla de arriba o el valor del objeto i-ésimo más la solución óptima con la capacidad restante”.

Cualquiera de estos tres casos toma tiempo constante, puesto que una función max es trivial, y obtener el valor anterior de la tabla también es un cálculo constante.

Para la implementación de este problema, proponemos el siguiente pseudocódigo:

Capacity: Capacidad de la mochila **(C)**
Pesos: Arreglo con los pesos de los objetos a escoger
Valores: Arreglo con los valores de los objetos a escoger
n: Cantidad de objetos a escoger

Function:

```
Mochila0_1(capacity, pesos, valores, n)
    Crear matriz tabla[n+1][Capacity+1]
    FOR j = 0 to capacity                  // O(C)
        tabla[0][j] = 0                    // Fila 0 se llena con 0's
    END FOR

    FOR i = 1 to n+1                       // Por cada ítem propuesto  // O(n*C)
        FOR j = 0 to capacity              // Para cada capacidad restante

            IF pesos[i-1] > j              // Si item no cabe
                tabla[i][j] = tabla[i-1][j]

            ELSE                           // Si ítem cabe
                // Tomar objeto y agregar a la tabla
                IF v[i-1] + tabla[i-1][j-pesos[i-1]] > tabla[i-1][j]
                    tabla[i][j] = v[i-1] + tabla[i-1][j-pesos[i-1]
                // No tomar objeto y tomar valor de fila anterior
                ELSE
                    tabla[i][j] = tabla[i-1][j]
        END FOR
    END FOR

    // Escoger que valores van en la mochila (Bottom-Top)
    j = capacity
    FOR n down to 1                        // O(n)
        IF tabla[i][j] > tabla[i-1][j]
            Agregar Item i-1 a la mochila
            j = j - w[i-1]
    END FOR
```

Notar que la segunda sección de código es para obtener la solución, puesto que primero formulamos la tabla y luego con los valores calculados la obtenemos.

Del pseudocódigo nos damos cuenta que podemos definir la complejidad que la complejidad más grande viene de los dos ciclos *for* anidados. Anteriormente habíamos analizado los tres casos que pueden ocurrir en cada casilla. Cada uno de estos es de tiempo constante. El primer caso ocurre fuera del doble for. Por esto, como se tiene que revisar ***n*** filas y ***C*** columnas, cada una tomando tiempo constante, la complejidad temporal se podría estimar como ***nC***.

Por lo tanto, podemos decir que la solución es polinomial con complejidad **Θ(nC)**, aunque en el siguiente punto esto será analizado con mayor detención.

b) ¿Qué puede decir respecto al tiempo de ejecución? ¿Observa alguna diferencia a la forma en la cual hemos analizado el tiempo de ejecución en unidades anteriores? Establezca de qué manera la complejidad obtenida para el problema puede incidir en la complejidad en término de tiempo de ejecución. La complejidad asociada a la solución obtenida normalmente se conoce como pseudo polinomial. Investigue y comente por que la solución para la mochila 0-1 se dice que es pseudo polinomial.

La diferencia en la complejidad de este algoritmo respecto a los que se habían visto con anterioridad en el curso, es que para este caso ya no sólo depende de la cantidad de elementos ingresados, si no que también depende de la capacidad ***C*** de la mochila. En caso de tener un valor ***C*** muy grande, la complejidad se verá amplificada **C** veces. Por ejemplo, para ***C*** = 1000 la complejidad es $\Theta(10^3 n)$, para ***C*** = 100.000 la complejidad es $\Theta(10^5 n)$ cuyo comportamiento es exponencial en *función de los dígitos de C*, pero a valores no tan grandes tiene comportamientos polinomiales, es por ello que la complejidad de este problema se dice pseudo polinomial.

c) Implemente el algoritmo usando enfoque bottom-up mediante tabulación.

El archivo *"01 knap.cpp"* está adjunto y contiene la solución a esta pregunta.

d) Ahora, resuelva el problema de programación dinámica usando tabulación, pero esta vez en lugar de usar el peso en las columnas use los valores de los objetos. Describa la solución usando programación dinámica para este enfoque de solución.

Usar los valores de los objetos en las columnas cambia drásticamente el problema, esto porque primero tenemos que encontrar una manera de organizar los valores de los objetos en las columnas, y además todos los valores generados por ellos.

Para lograr esto, se suman todos los valores v_i desde el objeto 0 hasta n y se obtiene el valor total V. Con esto, crearemos las columnas de la tabla desde 0 hasta V. Esto garantiza que se tengan todos los valores posibles en la tabla.

Posteriormente, en las filas tendremos los objetos con su peso w_i y valor v_i asociados. En cada celda de la tabla, se anotará el peso con el cual se puede generar dicho valor. Este valor puede ser obtenido con uno o más objetos. Al igual que en el problema anterior, en la fila i solo se podrán usar los objetos $0, 1, ..., i$.

Usando los mismos objetos del ejemplo anterior en la pregunta a):

Objeto →	**1**	**2**	**3**	**4**
Peso (kg) →	2	3	4	5
Valor →	1	2	5	6

A continuación vamos a generar la tabla y explicaremos cómo se va llenando.

De esto generamos la tabla con los valores desde 0 hasta V . En las filas tendremos los objetos. Notamos que la primera fila se inicializa con valores infinitos, porque estamos minimizando y queremos obtener el peso mínimo para generar un valor dado.

En la tabla se buscará generar un valor dado j con el menor peso posible. Este valor j viene de la columna en la que nos encontremos. Por ejemplo, nosotros sabemos que con los objetos 1 y 2, podemos generar un valor 3 con un peso 5. Entonces, esto debe estar reflejado en la tabla, al ubicar el peso 5 en la columna 3, con los dos primeros objetos en uso.

A medida que vamos avanzando y se van incluyendo más objetos, se podrá generar valores con un menor peso, o generar más valores porque se podrá incluir más objetos. Cabe decir que no se podrá registrar ningún peso que supere la capacidad de la mochila.

Objeto	**Valor**														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
NULL	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
P2V1	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-
P3V2	-	2	3	5	-	-	-	-	-	-	-	-	-	-	-
P4V5	-	2	3	5	-	4	6	7	-	-	-	-	-	-	-
P5V6	-	2	3	5	-	4	5	7	8	-	-	-	-	-	-

Si con el objeto actual no se puede generar un cierto valor, se copia la solución óptima anterior. A continuación se tiene el pseudocódigo.

El algoritmo funciona de manera similar al anterior, donde va revisando hasta la casilla j, hasta ese punto copia lo anterior. Luego ve si puede usar el objeto i actual junto a la solución óptima con el valor restante. Si este valor restante no es infinito, entonces es una solución y la usa. Después revisa si la capacidad no es excedida. Finalmente, compara los pesos y valores obtenidos para ver si es conveniente y efectivamente mejora con la nueva solución obtenida.

La implementación de este algoritmo está en el archivo “valKnap.cpp”.

```
valKnap01 (n, capacity, objects)
   totalValue = calcular la suma de todos los valores
   Crear una matriz[n+1][totalValue+1]
   llenar la primera fila y columna con valor infinito (llamado INF)
   FOR (i = 1; i <= n ; ++i)
        FOR (j = 1; j <= totalValue; ++j)
                peso = peso del objeto actual
                valor = valor del objeto actual
                IF (j < valor)
                        copiamos el valor óptimo anterior
                        table[i][j] = table[i-1][j]
                ELSE
                        IF (table[i-1][j-valor] != INF)
                                pesoGen = table[i-1][j-valor]
                                valGen = j-valor
                        ELSE
                                pesoGen = 0
                                valGen = 0
                        ENDIF
                        valAux = valGen + valor
                        pesoAux = pesoGen + peso
                        IF (pesoAux > capacity)
                                //Si sobrepasamos el peso copiamos la solución anterior
                                table[i][j] = table[i-1][j]
                                continue
                        ENDIF
                        IF (valAux == j)
                                if (pesoAux <= table[i-1][j])
                                        //Si valAux coincide con j y es mejor que la
                                        //solución anterior, lo guardamos
                                         table[i][j] = pesoAux
                                ELSE
                                        //solución óptima previa
                                         table[i][j] = table[i-1][j]
                                ENDIF
                        ELSE
                                //solución óptima previa
                                table[i][j] = table[i-1][j]
                        ENDIF
                ENDIF
        ENDFOR
   ENDFOR
```

e) Considere el mismo algoritmo anterior, pero ahora asuma que puede aceptar algo de error y no requiere obtener el óptimo. En este caso analice el caso en el cual los objetos se puede agrupar por valor $\lfloor \frac{v_i}{x} \rfloor$, donde $x = (1 - \beta)\frac{V}{n}$ y $0 < \beta < 1$, y $\sum_{i}^{n} v_i$. Esta solución conlleva a un algoritmo aproximado. Para este caso, implemente el algoritmo, analice el tiempo de ejecución asintótica del algoritmo en el peor caso, el error absoluto máximo y el factor de aproximación $\rho(n) = \frac{V^*}{V_a}$, donde V^* es la solución óptima y V_a la solución aproximada.

El algoritmo implementado está adjunto a este documento, de nombre "*aproxKnap.cpp*", cuyo input a recibir cumple el siguiente estándar: $< n > \ < C > \ < \beta > \ < lista\ p_i\ v_i >$. Ejemplo: 3 6 0.5 1 2 3 4 5 6

Cuyo pseudocódigo es:

FUNCTION: aproxKnap(n, C, β , weights, values)

```
totalValues ← 0
FOR 0 to i                                  // Sumar valores iniciales
    totalValues ← totalValues + values[i]

x ← (1- □) * (totalValues) / n              // Obtener x de tipo double
FOR 0 to n                                  // Obtenemos valores aproximados
    values[i] ← values[i] / x               // Casteados como int para "floor"

valKnap(n, C, weights, values)              // Ejecutamos función anterior
                                            // con los valores aproximados
```

END FUNCTION

Respecto al tiempo de ejecución asintótica, al obtener nuevos valores aproximados "hacia abajo" y se obtiene mediante el siguiente desarrollo:

$$Si\ V = \sum_{i=0}^{n} v_i \ , \ x = (1 - \beta) * \frac{V}{n} \ y \ v_i' = \lfloor \frac{v_i}{x} \rfloor \ entonces:$$

$$v_i' = \lfloor \frac{v_i}{(1-\beta)\frac{V}{n}} \rfloor = \lfloor \frac{n}{(1-\beta)} \frac{v_i}{V} \rfloor \qquad Si\ \varepsilon = (1-\beta)\ nos\ queda:$$

$$v_i' = \lfloor \frac{n}{\varepsilon} \frac{v_i}{V} \rfloor$$

Luego, la cantidad de columnas que tendrá la tabla será

$$V' = \sum_{i=0}^{n} v_i' = \sum_{i=0}^{n} \lfloor \frac{n}{\varepsilon} \frac{v_i}{V} \rfloor$$

De esto podemos aproximar:

$$\sum_{i=0}^{n} \lfloor \frac{n}{\varepsilon} \frac{v_i}{V} \rfloor \leq \sum_{i=0}^{n} \frac{n}{\varepsilon} \frac{v_i}{V} = \frac{n}{\varepsilon V} \sum_{i=0}^{n} v_i$$

Pero recordar que $V = \sum_{i=0}^{n} v_i$, reemplazamos:

$$\frac{n}{\varepsilon V} \sum_{i=0}^{n} v_i = \frac{nV}{\varepsilon V} = \frac{n}{\varepsilon}$$

Con esa aproximación de columnas y con ***n*** filas, podemos concluir que la complejidad del algoritmo es:

$$O(nV') = O(\frac{n^2}{\varepsilon})$$

Para obtener el error absoluto máximo, tomamos V_i, $V_j \in valores[] \cap S_k$, con $S_k = \{ \forall v \in valores[] : \lfloor \frac{n}{\varepsilon} \frac{v}{V} \rfloor = V_K' \}$, es decir, el conjunto de todos los valores que caen en la misma aproximación.

Luego, para obtener el error absoluto máximo, sabemos que:

$$|V_i - V_j| \leq (1-\beta)\frac{V}{n}$$

debido a que $(1-\beta)\frac{V}{n}$ es la condición que los hace caer en el mismo S_k
Desarrollamos:

$$|V_i - V_j| \leq (1-\beta)\frac{V}{n} < (1-\beta)V$$

Siendo $|V_i - V_j| = ERROR$ el error absoluto entre 2 elementos que caen al mismo conjunto, se deduce que el error absoluto máximo para cada situación se puede aproximar:

$$ERROR\ ABSOLUTO\ MÁXIMO\ < (1 - \beta)V$$

Conclusión

Previamente a este proyecto nuestra comprensión de la programación dinámica era algo débil, nos había costado seguir la materia y comprender correctamente cómo funcionaban y la utilidad de estos. Fue necesario complementar con contenido externo al del curso. Sin embargo, ahora se tiene una comprensión mucho mayor de estos, de su utilidad y las ventajas de la resolución de subproblemas y de la subestructura óptima.

El problema de la mochila es algo que personalmente me he preguntado desde hace mucho tiempo, y fue bastante interesante ver una resolucion a este problema. Estas preguntas vienen de hace muchos años, gracias a videojuegos con inventarios de tamaño y volumen limitado. En estos casos se necesitaría una mochila de aún más dimensiones, pero queda claro cómo se podría aplicar el algoritmo para esto, entonces, es interesante ver cómo funciona este algoritmo y aún más saber que hay maneras de acelerar los procesos permitiendo un margen de error con los algoritmos aproximados.