Enunciado 2, Problema de Programación Dinámica

Análisis e Implementación

Dany Efraín Rubiano Jiménez
Departamento de Ingeniería Informática
Universidad de Santiago de Chile
Santiago de Chile, Chile
dany.rubiano@usach.cl

***Abstract*— En el presente documento se analiza la aplicación del método de programación dinámica, con el fin de resolver una problemática dada aplicando su definición y explicando cómo se contextualiza en el desarrollo de esta.**

Index Terms—Programación Dinámica, Algoritmo, C.

I. Introducción

A pesar de que existen incontables algoritmos, se han logrado identificar un número de técnicas de diseño suficiente mente generales como para poder aplicarse a un gran número de problemas. Estas técnicas dan como resultado algoritmos eficientes (a veces...) y que son de muy diversos dominios de a plicación.

Para esta ocasión se aborda un problema dado, para el cual se aplicará la técnica de resolución de la programación dinámica, buscando encontrar la solución óptima a partir de la resolución de problemas mediante una secuencia de decisiones.

II. Descripción del Problema

El problema dado al cual se aplicará la técnica de resolución de la programación dinámica, se enmarca en la obtención de la sub matriz de m x p dimensiones cuya suma interna sea la mayor dentro de las posibles sub matrices de m x p, formadas a partir de la matriz inicial n x n dimensiones, que contiene números positivos, negativos y el cero, se pide encontrar

Para ello se entrega en un archivo de texto llamado “entrada.in”, las dimensiones de la sub-matriz, acompañado de la matriz principal, a la cual se le aplicará la operación., un ejemplo de ello, se encuentra en la Tabla 1.

TABLE I. Tabla Complejidad Algorítmica

Sub-matriz	*Filas: 3*		*Columnas: 2*	
	j_0	j_1	j_2	j_3
i_0	1	-2	-7	2
i_1	9	2	-7	2
i_2	-4	3	-4	0
i_3	-1	8	-3	-2

Para el resultado, se debe generar un archivo de nombre “salida.out” donde se escriba la sub matriz de m x p que tenga la mayor suma interna, seguida por la suma asociada a dicha sub matriz. Esta sería la representación de la solución óptima de la asignación de las tareas en función del tiempo y habilidades de cada persona. Un ejemplo para la salida se representa en la Tabla II.

TABLE II. Tabla Complejidad Algorítmica

	j_0	j_1	j_2
i_0	1	-2	-7
i_1	9	2	-7
i_2	-4	3	-4
Suma = 17			

III. Explicación de la Solución

En el problema dado, se requiere encontrar la sub matriz de m x p dimensiones cuya suma interna sea mayor a partir de la matriz inicial n x n dimensiones. Para lo anterior, se debe en primer lugar obtener las dimensiones de la submatriz m x p, luego, obtener las dimensiones de la matriz n x n, para después obtener los datos de esta y realizar la operación pertinente.

La técnica requerida en la resolución del problema es la de programación dinámica. Siguiendo este método, se debe dividir el problema en subproblemas más pequeños, es decir, dividir la matriz en submatrices más pequeñas que la submatriz requerida, para luego resolver estos problemas de manera óptima buscando encontrar la mayor suma, y por último usar estas soluciones óptimas para construir una solución óptima al problema encontrando la suma mayor.

IV. Método Utilizado

Para la resolución del problema, tal como se mencionó anteriormente se utiliza la técnica de programación dinámica, este método se utiliza para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas. Este método fue inventado por el matemático Richard Bellman en 1953.

Hay dos condiciones que se deben cumplir antes de comenzar a pensar en una solución a un problema de optimización este método. Primero, debe tener sub-estructuras óptimas, o sea, cuando la solución óptima a un problema se puede componer a partir de soluciones óptimas de sus sub-problemas. Y segundo debe tener una superposición de problemas, o sea, que para el cálculo de la solución óptima implique el resolver muchas veces un mismo sub-problema.

La programación dinámica toma normalmente uno de los dos siguientes enfoques:

- Top-down: El problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente. Es una combinación de memoización y recursión.
- Bottom-up: Todos los problemas que puedan ser necesarios se resuelven de antemano y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Si se analiza el problema a resolver, se puede encontrar con que este tiene sub-estructuras óptimas, ya que la matriz se puede dividir en múltiples submatrices; y ocurre una superposición de problemas, ya que para encontrar la submatriz con la mayor suma, implicará resolver varias veces las submatrices más pequeñas.

V. Descripción de la Solución

Ya se pudo comprobar que a este problema se le puede aplicar perfectamente el método de programación dinámica, para dar paso a establecer la solución del problema, primero se debe obtener las dimensiones de la submatriz m x p, para ello, se debe leer la primera línea del archivo y pasarlo a un arreglo que guarde estas.

Posteriormente a esto, se debe obtener las dimensiones de la matriz n x n, para ello, ya que es una matriz cuadrada, basta con encontrar una de las dimensionas, entonces, se cuentan el número de líneas que posee el archivo pasando este número a una variable, y restándole un uno a esta, por la primera línea.

Ya obtenidas las dimensiones de la matriz del archivo, se crea una matriz con las mismas dimensionas que contengan los datos del archivo, y así tener estos datos disponibles para operarlos. Basta entonces, con pasar estos a la matriz creada.

Como ya tenemos todos los datos necesarios para operar, procedemos a tratar de encontrar la solución óptima, para ello, siguiendo el método establecido, con la estrategia de Bottom – up. De esta manera

Las funciones implementadas para llevar a cabo esta solución son:

- *struct cam*
  Estructura para guardar la suma y el camino de las sub-matrices en una tabla, para poder aplicar el método.

- *void getFilasYColumnasSubMatriz(int arr[])*
  Función que lee el archivo de texto, leyendo la primera línea y obteniendo las dimensiones de la sub-matriz.

- *int getFilasYColumnasMatriz( )*
  Función que lee el archivo y cuenta el número de líneas, retornando este número disminuido en uno.

- *void lecturaArchivo(int m, int matriz[m][m])*
  Función que lee el archivo y pasa los datos a la matriz.

- *void camino1(int a, int s1, int s2, int m, int matriz[m][m], struct cam tabla[a][m])*
  Si el número de filas de la sub-matriz es mayor que el número de columnas, escogemos dividir la matriz según las columnas para realizar menos operaciones. Si es así, se utiliza esta función, que guarda los datos en una tabla, utilizando así el método de Bottom-up.

- *void camino2(int a, int s1, int s2, int m, int matriz[m][m], struct cam tabla[m][a])*
  Si el número de filas de la sub-matriz es menor que el número de columnas, escogemos dividir la matriz según las filas, para realizar menos operaciones Si es así, se utiliza esta función, que guarda los datos en una tabla, utilizando así el método de Bottom-up.

- *int getSubmatriz1(int a, int s1, int s2, int m, struct cam tabla[a][m], int arr[])*
  Función que a partir de la tabla construida en la función camino1, encuentra la sub-matriz óptima.

- *int getSubmatriz2(int a, int s1, int s2, int m, struct cam tabla[m][a], int arr[])*
  Función que a partir de la tabla construida en la función camino1, encuentra la sub-matriz óptima.

- *main( )*
  Función principal en la que se van llamando las funciones anteriores.

VI. Complejidad Algorítmica

La complejidad algorítmica representa la cantidad de recursos (temporales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo. En esta ocasión, se representa el orden de cada una de las funciones utilizadas, calculado por el número

de ciclos. El cálculo del orden para cada una de las funciones establecidas en la resolución del problema se encuentran en la Tabla III.

TABLE III. TABLA COMPLEJIDAD ALGORÍTMICA

Función	Orden
getFilasYColumnasSubMatriz	O (n^2)
getFilasYColumnasMatriz	O (n)
lecturaArchivo	O (n^2)
camino1	O (n^3)
camino2	O (n^3)
getSubmatriz1	O (n^2)
getSubmatriz2	O (n^2)
main	O (n^3)

VII. TRAZA

En la Tabla IV, se muestra una traza simple recortada en base a la entrada propuesta en la Tabla I.

TABLE IV. TABLA TRAZA SIMPLE

Función	Resultado - Traza
getFilasYColumnasSubMatriz(int arr[])	3 2
getFilasYColumnasMatriz ( )	4
lecturaArchivo(int m, int matriz[m][m])	1 -2 -7 2 9 2 -7 2 -4 3 -4 0 -1 8 -3 -2
camino1(int a, int s1, int s2, int m, int matriz[m][m], struct cam tabla[a][m]) o camino2(int a, int s1, int s2, int m, int matriz[m][m], struct cam tabla[m][a])	//Donde los elementos de la tabla se separan por los corchetes a modo de vista. El primer elemento representa la suma, el segundo representa la fila de inicio, el tercero representa la fila de término, la cuarta representa la columna de inicio y la quinta representa la columna de término. [6, 0, 3, 0, 0] [3, 0, 3, 1, 1] [-18, 0, 3, 2, 2] [4, 0, 3, 3, 3] [4, 1, 4, 0, 0] [13, 1, 4, 1, 1] [-14, 1, 4, 2, 2] [0, 1, 4, 3, 3]
getSubmatriz1(int a, int s1, int s2, int m, struct cam tabla[a][m], int arr[]) o getSubmatriz2(int a, int s1, int s2, int m, struct cam tabla[m][a], int arr[])	7//Donde el primer elemento representa la suma mayor, el segundo representa la fila de inicio, el tercero representa la fila de término, la cuarta representa la columna de inicio y la quinta representa la columna de término. 17, 1, 4, 0, 1
main( )	9 2 -4 3 -1 8 Suma: 17

VIII. EFICIENCIA

La eficiencia de cada una de las funciones empleadas en la resolución del problema, es de alguna manera aceptable, ya que en su solución entregan el resultado esperado, y en un tiempo mínimo, que no alcanza a ser ni siquiera un segundo.

De por si, por el hecho de utilizar la el método de programación dinámica, da una eficacia en la resolución del problema.

IX. CONCLUSIONES

Como conclusiones se puede afirmar que se logró de alguna manera llegar a los resultados correctos, si lo probamos con los ejemplos que están en el enunciado del laboratorio.

Las estructuras de datos utilizadas se usaron en base a los algoritmos planteados y cumplen la función para lo que fueron declaradas.

Las técnicas de desarrollo de algoritmos nos permiten encontrar solución a los problemas que se nos presentan y deben ser solucionados por el computador, estas técnicas están orientadas para utilizarse en cada uno de los niveles de complejidad y variedad o alternativas para las cuales se aplican los algoritmos, en el caso de la fuerza bruta, para la mayoría de problemas reales, el número de soluciones candidatas es prohibitivamente elevado, por lo cual es ineficiente ante casos muy grandes. El tiempo de ejecución y la eficiencia de la solución con este método, depende del tamaño del problema.

La programación dinámica encuentra la solución óptima de un problema con n variables descomponiéndolo en n etapas siendo cada etapa u sub-problema de una sola variable, sin embargo como la naturaleza de la etapa, difiere de acuerdo con el problema. La programación dinámica no proporciona los detalles de cómputo para optimizar cada etapa siendo esta una de las principales desventajas.

Si se compara el método de programación dinámica con el de la fuerza bruta o el de goloso, obviamente, este tiene una mayor eficiencia en la utilización de recursos y reduce notoriamente el tiempo de ejecución, es por esto, que esta técnica es ampliamente utilizada.

REFERENCIAS

[1] www.lcc.uma.es/~av/Libro/CAP5.pdf

[2] www.cs.upc.edu/~iea/TranspasJavier/Pr_**dinamica**2.ppt

[3] https://es.wikipedia.org/wiki/**Programación_dinámica**