TAREA SEMANAL 2 – 4

PROGRAMACIÓN DINÁMICA

Integrantes:
Andrés Monetta
Alejandro Clara
Sebastián Daloia

Contenido

3. Preguntas

1. ¿Cómo se generan las soluciones de problemas utilizando la técnica de Programación Dinámica?

Para problemas de optimalidad se demuestra que se puede (o no) resolver evaluando el conjunto de decisiones que llevan a la solución óptima. Es decir, mirando si el conjunto de soluciones óptimas cumple con el Principio de Optimalidad.

En general se propone una recurrencia para obtener las soluciones intermedias y la final. Estas soluciones son almacenadas en tablas, al final del algoritmo se deduce que la solución se debe encontrar en cierta posición de la tabla.

2. Mencione diferencias entre Programación Dinámica y D&C.

Por lo general en Divide And Conquer las soluciones son implementadas por recurrencia.
En cambio, en Programación Dinámica generalmente las soluciones son implementadas por iteración.

En Divide And Conquer los subproblemas se pueden resolver independientemente.
En Programación Dinámica los subproblemas no son independentes. En ella se guardan las respuestas para solucionar problemas mayores.

Divide And Conquer es útil para problemas de gran tamaño que pueden dividirse en subproblemas de la misma clase que el problema inicial.
En Programación Dinámica se resuelven todos los subproblemas una sola vez, almacenando sus respuestas para utilizarlas más adelante.

Las propuestas por recurrencia de Divide And Conquer pueden tener un aumento en el orden del tiempo de ejecución debido a repetición de operaciones a causa de las divisiones del problema en subproblemas.
En Programación Dinámica, por lo general un problema se resuelve usando algoritmos iterativos, y aprovechando la información de resultados o estados anteriores, almacenándolos en tablas.
Programación Dinámica tiene la ventaja de evitar repetición de operaciones.

3. ¿Para qué es útil el principio de optimalidad?

Para determinar si un problema de optimización puede ser resuelto usando Programación Dinámica, para luego proponer la recurrencia que determinará cómo almacenar los resultados intermedios, hasta el resultado final.

4. Problema

1. Juegos de datos para contraejemplos

Atributo Fecha de Inicio

Charla 1 Fecha de inicio: 9 Fecha de fin:10 Asistentes: 0	Charla 2 Fecha de inicio: 20 Fecha de fin:22 Asistentes:55	Charla 3 Fecha de inicio: 12 Fecha de fin:15 Asistentes:2	Charla 4 Fecha de inicio: 21 Fecha de fin:23 Asistentes:50

Ordenamiento por Fecha de Inicio descendente {Charla 4, Charla 2, Charla 3, Charla 1}

Se toma la charla 4 como primer elemento de la solución. Luego se procede a evaluar la compatibilidad de la charla 2 con la charla 4, como no son compatibles, no se la agrega.
Se pasa a evaluar la compatibilidad de la charla 3 con la 4 y como es compatible con ella, se agrega. Mirando la charla 1 se ve que no hay problemas de compatibilidad. Por lo tanto la solución en este caso será <4,3,1>, que garantiza 52 asistentes. Sin embargo la solución óptima es <2,3,1>, que da 57 asistentes.

Atributo Fecha de Fin

Charla 1 Fecha de inicio: 19 Fecha de fin:23 Asistentes: 11	Charla 2 Fecha de inicio: 15 Fecha de fin:17 Asistentes: 0	Charla 3 Fecha de inicio: 20 Fecha de fin:22 Asistentes: 58	Charla 4 Fecha de inicio: 10 Fecha de fin:13 Asistentes: 5

Ordenamiento por Fecha de fin descendente {Charla 1, Charla 3, Charla 2, Charla 4}

Se toma la charla 1 como primer elemento de la solución, y luego se evalúa la compatibilidad con la charla 3. Claramente no son compatibles, por lo tanto se descarta la 3.
Mirando la charla 2 se puede ver que es compatible con la 1, por lo tanto se la agrega a la solución. La charla 4 también es compatible con las que ya están en la solución. Entonces se la incluye. La solución propuesta es <1,2,4> que da un total de 16 asistentes. Sin embargo, la solución óptima es <3,2,4>, que garantiza 63 asistentes.

Atributo Duración

Charla 1 Fecha de inicio: 10 Fecha de fin:12 Duración:2 Asistentes: 80	Charla 2 Fecha de inicio: 13 Fecha de fin:15 Duración:2 Asistentes: 80	Charla 3 Fecha de inicio: 9 Fecha de fin:19 Duración:10 Asistentes: 5	Charla 4 Fecha de inicio: 17 Fecha de fin:19 Duración:2 Asistentes: 200

Ordenamiento por Duración descendente {Charla 3, Charla 1, Charla 2, Charla 4}

Se agrega la charla 3 a la solución, y al mirar las demás se ve claramente que ninguna es compatible con ella. La solución propuesta es <3> que da lugar a 5 asistentes, sin embargo la solución óptima es <1,2,4>, dando lugar a 285 asistentes.

Atributo Asistentes

Charla 1 Fecha de inicio: 10 Fecha de fin:16 Asistentes: 45	Charla 2 Fecha de inicio: 17 Fecha de fin:19 Asistentes: 45	Charla 3 Fecha de inicio: 15 Fecha de fin:18 Asistentes: 80

Ordenamiento por Asistentes descendente {Charla 3, Charla 1, Charla 2}

Se incluye la charla 3 a la solución, y se procede a mirar la charla 1. Ésta no es compatible con la 3, por lo que se la descarta. La charla 2 tampoco es compatible, y por lo tanto la solución alcanzada es <3> que permite tener 80 asistentes. Sin embargo la solución óptima es <1,2> que garantiza 90 de ellos.

2.

I- Probar que se puede usar Programación Dinámica demostrando que se cumple el principio de optimalidad.

Sea $\{x_{0,}\, x_{1,}\ldots, x_{k-1}\}\ 0 \le k < n$ el conjunto de charlas que maximiza la cantidad de asistentes, supongamos ordenadas por fecha de finalización, siendo n la cantidad total de charlas.

Se deduce que $\{x_{1,}\ldots, x_{k-1}\}$ debe ser solución óptima para el intervalo de tiempo que abarcan, de no ser así existirían $\{r_1, \ldots, r_j\}$ $1 \le j < n$ solución óptima para el intervalo de tiempo que abarcan $\{x_{1,}\ldots, x_{k-1}\}$.

Entonces $\{x_{0,} r_1, \ldots, r_j\}$ maximizaría la cantidad de asistentes pues x_0 es compatible con cada elemento del conjunto $\{r_1, \ldots, r_j\}$ lo cual es absurdo pues $\{x_{0,}\, x_{1,}\ldots, x_{k-1}\}$ es solución óptima.

II- Escriba la recurrencia.

Supuesto el arreglo charlas[1..n] ordenado de forma creciente por fecha de fin.
Sean **j** la charla ubicada en la j-ésima posición $0 \le j < n$ y **T** el tiempo de finalización de la última charla.
Definimos $f_j(t)$ para todo t, $0 \le t \le T$ como la función que devuelve la cantidad máxima de asistentes para el tiempo t y las charlas del arreglo comprendidas desde 1 hasta j.

Se define la siguiente recurrencia:

$f_j(t)$ = máximo ({ $charlas[j].asistentes + f_{uj}(charlas[j].fechainicio)$, $f_{j-1}(t)$ })

Si charlas[j] es solución óptima entonces se suma su valor junto con la solución óptima para el tiempo charlas[j].fechainicio desde 1..ui que son el conjunto de charlas que comprenden todas las charlas compatibles con charlas[j].

Si charlas[j] no es solución entonces la solución óptima saldrá de no considerar a j, para el tiempo t.

3. Escriba el seudocódigo correspondiente a esta recurrencia

charlas[1..n] Arreglo de charlas ordenadas en forma creciente por atributo Fecha de fin
cantidadObjetos cantidad de charlas del arreglo
inicioSala Horario de inicio de la sala
finSala Hora de fin de la sala

La matriz g[cantidadObjetos + 1][cantHoras +1] almacena los estados de las soluciones óptimas para cada caso.
Se recorren las columnas desde la última fila de g, hasta la primera.
La ultima fila representa la solución para cero charlas.
La anteúltima representa las soluciones para una charla, siendo ésta la del primer elemento del arreglo charlas[1...n]
La primer fila viene a representar las soluciones considerando a todas las charlas del arreglo.

```
//La primera fila de la matriz g corresponde a los estados considerando el ultimo elemento del arreglo charlas
int max_asistentes(int *charlas, int cantidadObjetos , int inicioSala, int finSala)
{
	//Inicialización de tabla
	int cantHoras=0;
	int ui=0;
	for(int i = inicio; i < fin; i++)
		cantHoras++;
	int g[cantidadObjetos + 1][cantHoras +1]

	//Inicialización ultima fila
	for(int c=0; c<=cantHoras; c++)
		g[cantidadObjetos][c]=0;
	//Recorrida de filas desde la (n – 1)-ésimas filas de la matriz, columna a columna
	for(int  j = cantidadObjetos -1; j>=0; j--){
		ui=siguienteCompatible( charlas, (cantidadObjetos  - 1)- j, cantidadObjetos);
		for(int c=0; c <= cantHoras; c++){
			if(charlas[(cantidadObjetos  - 1) - j].fechafin <= c + inicioSala){

				g[j][c]=max(charlas[(cantidadObjetos  - 1) - j].asistentes +
				    g[ui][charlas[(cantidadObjetos  - 1) - j].fechainicial, g[j+1][c] )

			}else
				g[j][c]=g[j+1][c]
		}
	}
	return g[0][cantHoras]
}
```

```
//La función devuelve el índice de la fila en la tabla de estados de la charla anterior compatible mas próxima a la charla j.
int siguienteCompatible( int charlas; int j; int cantidadObjetos){
        indice=(cantidadObjetos - 1 - j)
        for(int  h = j-1; h>=0;  h--){
                if(charla[h].fechafinal <=  charlas[j].fechainicial )
                        return indice + (j-h);
        }
        return cantidadObjetos;
}
```

4. Modifiquelo para que además devuelva la solución óptima

Se define la matriz p[cantidadCharlas + 1][cantidadHoras + 1]
Con cantidadHoras siendo el intervalo de horas que está disponible la sala.

Siendo la entrada (i,j) $0 \leq i < cantidadCharlas$ $0 \leq j < cantidadHoras$
Y siendo
$f_i(j)$ = máximo ({ $charlas[i].asistentes + f_{ui}(charlas[i].fechainicio)$, $f_{i-1}(j)$ })
en (i,j) se guarda o el índice i si se incluye en la solución o el índice óptimo para $f_{i-1}(j)$.
Es decir la matriz p almacena en sus entradas el índice de la ultima charla de la tupla solución óptima para el tiempo j, para el arreglo charlas[1...i].

De esta manera al final en la entrada (0, cantidadHoras) de la matriz p se obtendrá la última charla agregada a la solución.
Luego buscando el índice de charla óptimo para ui el compatible más próximo ui, y el tiempo charlas[i].fechainicio se obtiene el siguiente índice que forma parte de la solución.
Procedemos así hasta que no haya más compatibles en los que buscar.

```
//Calculo de la matriz p para obtener valores optimos
//int optimo[cantidadObjetos + 1][cantHoras +1]
int max_asistentes(int *charlas, int cantidadObjetos , int inicioSala, int finSala, int &optimo)
{
	//Inicialización de tabla
	int cantHoras=0;
	int ui=0;
	for(int i = inicio; i <= fin; i++)
		cantHoras++;
	int g[cantidadObjetos + 1][cantHoras +1]
	int optimo[cantidadObjetos + 1][cantHoras +1]

	//Inicialización ultima fila
	for(int c=0; c<=cantHoras; c++){
		g[cantidadObjetos][c]=0;
		optimo[cantidadObjetos][c]=0;
	}

	//Recorrida de filas desde la (n – 1)-ésimas filas de la matriz, columna a columna
	for(int  j = cantidadObjetos -1; j>=0; j--){
		ui=siguienteCompatible( charlas, j, cantidadObjetos);
		for(int c=0; c <= cantHoras; c++){
			if(charlas[(cantidadObjetos  - 1)) - j].fechafin < c + inicioSala){
				if(charlas[(cantidadObjetos  - 1) - j].asistentes + g[ui]
[(cantidadObjetos  - 1) - j].fechainicial >= g[j+1][c])
				{
					g[j][c]=charlas[(cantidadObjetos  - 1) - j].asistentes + g[ui]
[charlas[(cantidadObjetos  - 1) - j].fechainicial;
					optimo[j][c]=j;
```

```
                    }else{
                        g[j][c]=g[j+1][c];
                        optimo[j][c]=optimo[j+1][c];
                    }
                }
                else{
                    g[j][c]=g[j+1][c];
                    optimo[j][c]=optimo[j+1][c];
                }
            }
        }
        return g[0][cantHoras];
}
```