eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
informatika fakultatea
facultad de informática

Practica programación III

Diseño de algoritmos

20/05/2016

Gorka Bravo
Facultad de Informática
Universidad del País Vasco, UPV/EHU

Indice

1. Presentación del problema--------pag 1.
2. Presentación del código----------pag 1.
3. Estudio experimental-------------pag 8.
4. Limite del problema para
   la programación dinámica----------pag 11.
5. Conclusiones-------------------pag 12.

1. Problema transportar miles de litros de agua:

Una agencia de viajes ha planeado un paquete turistico especial a la Antártida, que sólo podrá realizarse una vez en este a~no, y está ofertando las P plazas del único vuelo que va a poder realizar. Se ha hecho pública la oferta del viaje y ha tenido tanto éxito que ya tiene una gran cantidad de personas (muchas más de P) solicitando una plaza. Para obtener el máximo beneficio posible, la agencia ha decidido realizar una especie de subasta para asignar las plazas y ha pedido a las personas solicitantes que propongan un par de datos cada una:
1. El precio en euros que está dispuesta a pagar para conseguir una plaza y
2. El máximo peso en kg. que llevara en el viaje.
La agencia sabe que no puede transportar en el avión más de T kilos y desea obtener el máximo beneficio.

2. Presentación del código:

Aquí podemos ver un ejemplo de backtrack para resolver este problema.
Para ello utilizaremos dos algoritmos de backtrack diferentes:

1) Algoritmo backtrack que se vale de un árbol binario para resolver el problema.
2) Algoritmo backtrack que se vale de un árbol n-ario para resolver el problema.

2.1 Árbol binario:

Primero tendremos el caso base de la recursión donde el backtrack para de recorrer el árbol cuando ya ha comprobado a todos los viajeros dispuestos a viajar (N numero de viajeros apuntados al viaje). Después vendrá la recursión principal, donde escogeremos entre añadir el pasajero "k" o no añadirlo al array solución (sol). Esta parte se ejecutara siempre y cuando queden viajeros por comprobar, Ademas se le añaden dos condiciones más. Si añadiendo el siguiente viajero se pasa del peso establecido (Tmax carga máxima permitida en el avión) se deja de analizar la parte de añadir viajeros en ese caso y todas las demás combinaciones con esos viajeros. Si no, y mientras no esta lleno el array sol, se sigue comprobando las combinaciones añadiendo y sin añadir el viajero k. Además se suma un entero count que guardará el numero de casos

analizados al finalizar el backtrack.

```
public void backtrack_if(int k, int[] sol, int sitios, int peso, int g) {
		if (k == N) {
			if (g > ganancias & peso <= Tmax) {
				ganancias = g;
				Plazas = sitios;
				asigResult(sol);
			}
		} else {
			if(peso + pesos[k] > Tmax) {
				count++;
				backtrack_if(k+1, sol, sitios, peso, g);
			} else if(!lleno(sol)){
				int pos = anadir(sol, k);
				count++;
				backtrack_if(k+1, sol, sitios+1, peso + pesos[k], g +
pagos[k]);
				quitar(sol, -1, pos);
				count++;
				backtrack_if(k+1, sol, sitios, peso, g);
			}
		}
	}
```

(Imagen 2.1)

2.1.1 Variables de entrada:

→ int k: Entero que recorre los viajeros que quieren ir a la Antártida, este recorrido se hace de uno en uno en cada llamada recursiva, con el valor inicial a 0.

→int[] sol: Array de enteros que contiene los viajeros que viajarán, se añaden o no los viajeros a este array, al final contendrá el resultado de con los viajeros con el numero de cada viajero dentro del array, el valor contiene todo el array con -1.

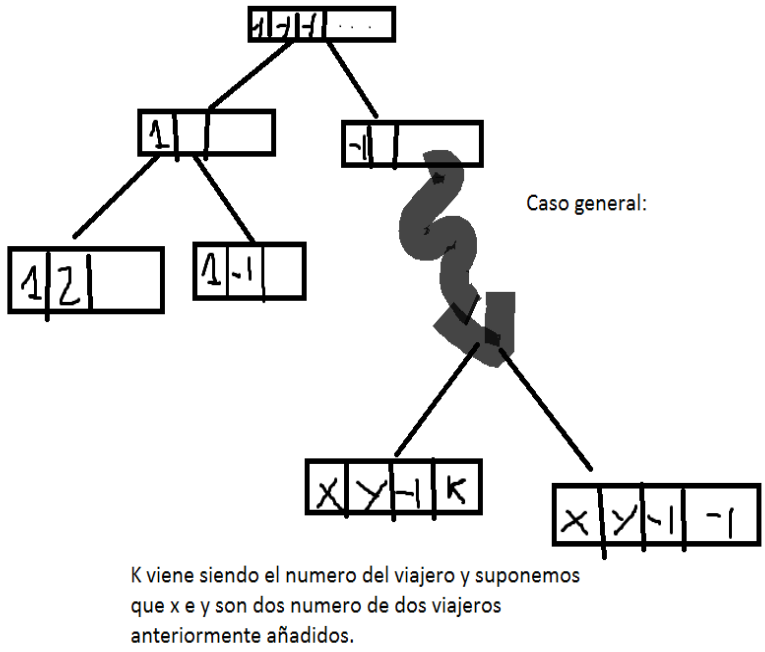
→int sitios: cantidad de sitios que se van llenando según se añade un viajero a la lista, el valor inicial es 0.

→ int peso: Numero de kg que se van añadiendo según se añaden personas, inicialmente esta a 0.

→ int peso: Cantidad que esta dispuesto a pagar cada cliente que se van añadiendo según se añaden personas, inicialmente esta a 0.

2.1.2Explicación del código:

Este código se basa en el método de backtrack para comprobar todas las opciones de un árbol en el que se encuentran todas soluciones para el problema y al recorrerlo se encuentra la solución optima. El árbol lo creamos añadiendo o no viajeros, se acorta el recorrido evitando ramas que se sabe que no contendrán la solución como se explica en en el apartado 2 de la página 3. El árbol quedaría así:



(Imagen 2.2)

2.2 Árbol n-ario:

Este algoritmo es ligeramente diferente al otro, primero tenemos un caso donde cada vez que un resultado sea Aceptable lo valoraremos y si es mejor que el mejor resultado hasta el momento lo sustituiremos por este. Después tenemos un if-else que simplemente nos comprueba en que viajero estamos y se lo asigna a una variable auxiliar que utilizaremos para empezar a comprobar los viajeros a partir de esa variable para no recorrer viajeros ya añadidos en el array cuando hagamos el "for" (recorrido de todos los viajeros hasta el final, desde el viajero que coincida con la variable auxiliar).

Dentro del for realizaremos la recursión, añadiremos los viajeros al array y por cada vez volveremos a ejecutar el mismo procedimiento desde el principio para el siguiente viajero. Al igual que en el código anterior el count nos servirá para saber el numero de casos estudiados del árbol.

```
public void backtrack_n(int k, int[] sol, int sitios, int peso, int g) {
		if(peso <= Tmax && g > ganancias) {
			ganancias = g;
			Plazas = sitios;
			asigResult(sol);
		}
		int aux;
		if(k == 0)
			aux = 0;
		else
			aux = sol[k-1];

		for(int i=aux+1; i < N;i++)
			if(peso+pesos[i] <= Tmax && !lleno(sol)) {
				int pos = anadir(sol, i);
				count++;
				backtrack_n(k+1, sol, sitios+1, peso+pesos[i], g+pagos[i]);
				quitar(sol, -1, pos);
			}
	}
```

(Imagen 2.3)

2.2.1 Variables de entrada:

(Las variables de entrada son las mismas que en el algoritmo anterior. Se puede obviar este apartado.)

→ int k: Entero que recorre los viajeros que quieren ir a la Antártida, este recorrido se hace de uno en uno en cada llamada recursiva, con el valor inicial a 0.

→int[] sol: Array de enteros que contiene los viajeros que viajarán, se añaden o no los viajeros a este array, al final contendrá el resultado de con los viajeros con el numero de cada viajero dentro del array, el valor contiene todo el array con -1.

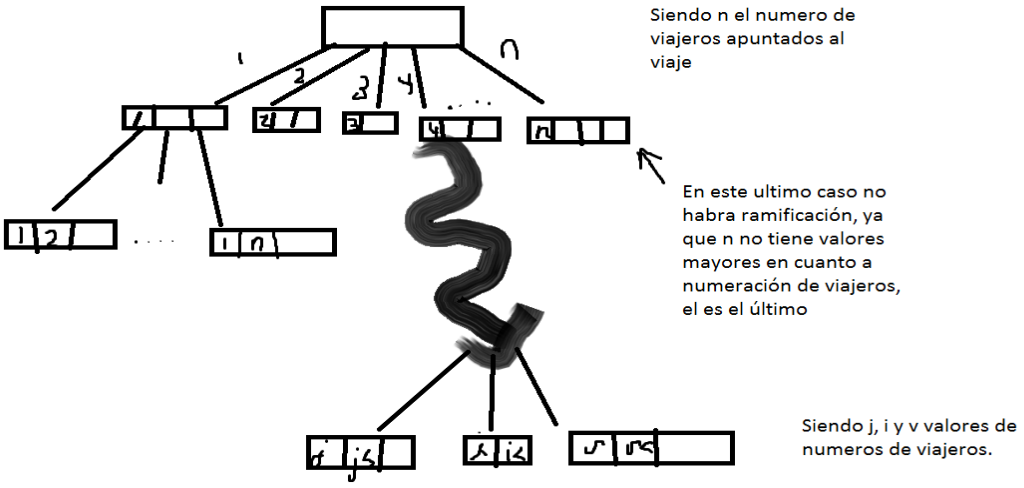
→int sitios: cantidad de sitios que se van llenando según se añade un viajero a la lista, el valor inicial es 0.

→ int peso: Numero de kg que se van añadiendo según se añaden personas, inicialmente esta a 0.

→ int peso: Cantidad que esta dispuesto a pagar cada cliente que se van añadiendo según se añaden personas, inicialmente esta a 0.

2.2.2 Explicación:

Este código se basa en el método de backtrack para comprobar todas las opciones de un árbol en el que se encuentran todas soluciones para el problema y al recorrerlo se encuentra la solución optima. El árbol se crea a partir de que para cada viajero se añaden, siempre y cuando no se supere el peso máximo, todos los viajeros que estén por encima de este en cuanto a numeración se refiere. Si los viajeros que hay en el array son aceptables se comparan con la solución actual y de ser mejor se actualiza. Este árbol quedará en una forma algo singular que tendrá las ramas del lado derecho mas largas y a medida que se va creando el árbol las ramas serán más pequeñas.
El árbol quedaría así:



(Imagen 2.4)

3. Estudio experimental

Para la realización del estudio experimental se ha tenido en cuenta los tres factores de los datos de entrada; Numero de clientes apuntado, numero de plazas disponibles y la cantidad de peso máxima que podía llevar el avión. Se ha jugado con estas 3 variables de manera que podamos ver diferencias al combinar las variables y sacar en claro que variable afecta más a que algoritmo, sin contemplar tanto cuestiones como cuanto duran los algoritmos, como resultado quedara un algoritmo "vencedor" (el más rápido en tiempo de ejecución de los dos) y las variables que más afectan a los algoritmos.

Árbol 0/1	Clientes apuntados	Plazas disponibles	Carga máxima	Tiempo en minutos
1	300	200	60	7 min 05 seg
2	300	170	60	8 min 14 seg
3	300	200	70	11 min 32 seg
4	310	200	60	12 min

(Tabla 3.1)

Árbol n-ario	Clientes apuntados	Plazas disponibles	Carga máxima	Tiempo en minutos
1	300	200	60	6 min 10 seg
2	300	170	60	4 min 21 seg
3	300	200	70	9 min
4	310	200	60	9 min 07 seg

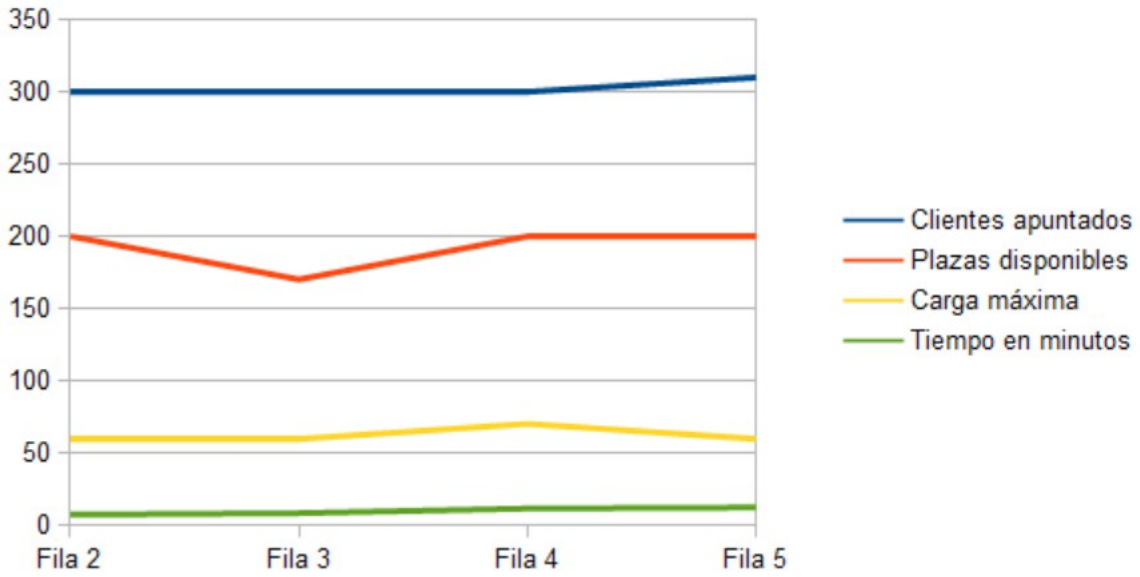
(Tabla 3.2)

Los resultados de estas tablas han sido una media de 6 ejecuciones en diferentes casos del mismo tamaño.

4.1 Comentario datos

Árbol binario:

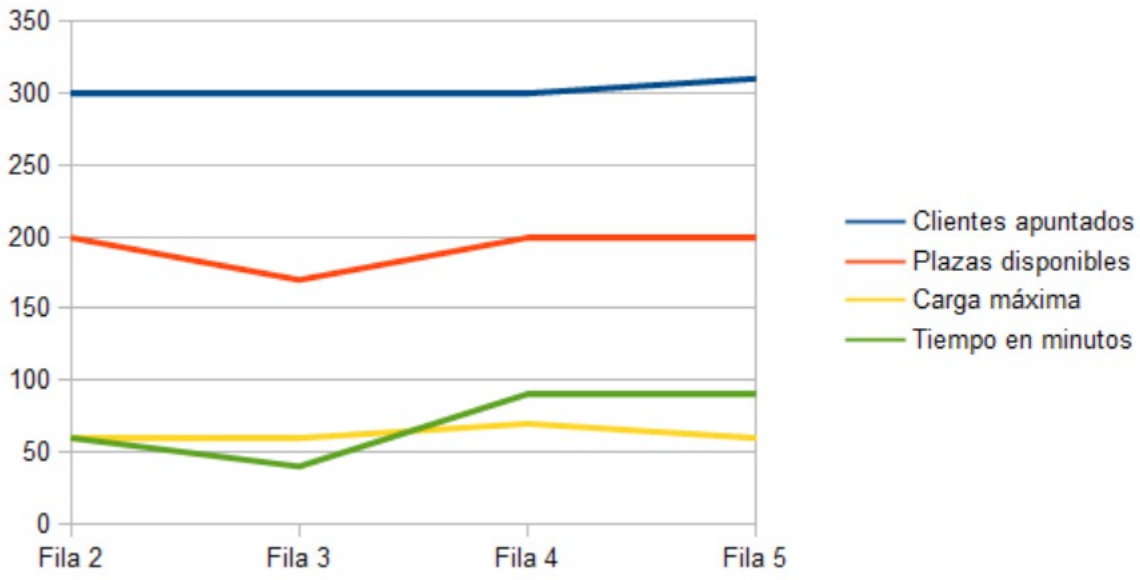
En la Tabla 3.1 tenemos los datos de el algoritmo con el árbol binario. En estos datos compararemos el tiempo de ejecución a partir de la primera fila. Podemos ver que al reducir el numero de plazas disponibles respecto a la primera fila el tiempo aumenta, aunque la reducción no sea demasiado grande se empieza a ver el aumento del tiempo. Aun así esta reducción es mayor que el aumento del peso permitido, en cambio la diferencia de tiempos entre la tercera fila y la primera es más significativa. Lo que nos lleva a deducir que el peso máximo permitido es más determinante para este algoritmo en cuanto al tiempo se refiere. De todas maneras si nos fijamos en la fila 4, podemos apreciar que reduciendo el peso máximo una "cantidad considerable" y aumentamos la cantidad de clientes apuntados muy poco el tiempo crece considerablemente en diferencia con la primera fila. En conclusión queda claro que el factor más determinante son la cantidad de cliente apuntados. Para la mejor visualización del progreso de los datos tenemos la tabla de arriba donde podemos ver la evolución de los mismo en el orden que están en la tabla (Imagen 3.3).



(Imagen 3.3)

Árbol n-ario:

En la Tabla 3.2 tenemos los mismos datos que teníamos para el algoritmo del árbol binario, pero en este caso para el árbol n-ario. Nada más empezar a ver los datos salta a la vista que los tiempos en este algoritmo son considerablemente menores. Sinceramente no sabría explicar el porque a ciencia cierta, aunque puedo intuir que la poda realizada en este árbol es mucho mayor que en el anterior lo nos permite rebajar el tiempo de ejecución. En este caso al reducir el tamaño de las plazas el problema reduce el tiempo, esto se debe a que en este algoritmo habría que rellenar un array menor. Con la carga máxima sucede igual que en el algoritmo anterior si subimos la carga el tiempo aumenta considerablemente, es más, podemos ver que el tiempo que aumenta en el primer algoritmo y en el segundo pueden llegar a ser proporcionales en cuanto la carga se refiere. En la última fila el cambio no es tan significativo, aunque si la distancia se aumenta más ser aprecia mejor, yo no lo he podido hacer para que los datos no se desvíen demasiado y se aprecien bien en el gráfico. Así pues queda claro que el factor determinante es la cantidad de clientes apuntados en los dos algoritmos. Además podemos ver que este ultimo algoritmo es considerablemente mejor. Como en el anterior están los datos en el gráfico para verlos mejor en la Imagen3.4.



(Imagen 3.4)

4. Limite del problema para la programación dinámica.

En este apartado se buscara el limite de datos para el caso de la programación dinámica. Este limite se refiere a encontrar datos para los cuales tarda "demasiado" como para ser resuelto el problema. Para ello contaremos con la tabla te tiempos para cada valor y analizaremos el gráfico que nos de. Se intentarán encontrar los limites para cada variable.

Prog. Dinámica	Clientes apuntados	Plazas disponibles	Carga máxima	Tiempo en segundos
1	1000	500	900	Falla por cuestiones de memoria
2	1000	500	70	1 seg
3	1000	20	900	1 seg
4	50	40	900	1 seg

(Tabla 3.3)

Nos encontramos con que los datos utilizados para los algoritmos de backtrack son muy fáciles de resolver en cuanto a tiempo se refiere para este algoritmo dinámico. En cambio sabemos que los algoritmos de backtrack llegaran a la solución por más que tarden, aquí se ve que el punto flaco de este problema recae en la memoria utilizada. A pesar de ser un algoritmo muy rápido respecto a los de backtrack tiene otras limitaciones, la memoria, así se ve reflejado en la tabla 3.3 al compararla con las tablas 3.1 y 3.2.
Por último vemos que a este algoritmo le afecta tanto el cambio de la variable de carga máxima como el de clientes apuntados o plazas disponibles. Se pueden apreciar los tiempos en la tabla 3.3 para ver como al cambiar los valores no le afecta demasiado y lo que de verdad afecta son las tres variables en general. Esto a simple vista puede pasar desapercibido, pero si lo pensamos mejor podemos caer en que la memoria utilizada para este algoritmo de programación dinámica puede ser un cubo o array tridimensional. Esto no significa nada porque no demuestra nada ni asegura que sea ningún array tridimensional, pero no se descarta que pueda ser una opción.

5. Conclusiones.

Como conclusión, podemos ver las dos caras una misma moneda en cuanto a las diferencias entre los dos algoritmos de backtrack. Puede que a veces diferentes técnicas de implementar un mismo algoritmo nos permita cortar más ramas o ramificar menos en el caso del backtrack. Por otra parte vemos la gran diferencia entre backtrack y prog. Dinámica, donde el primero encontrara la solución pero perderá en tiempo, mientras que este último a pesar de ganar con gran diferencia en tiempo tiene limitaciones en memoria que le llevan a no logran solución para el problema.