Práctica 6 AED

Ismael Caamano Figueiras, Mateo Díaz Pérez

Resumen

Ramificación y Poda supone una optimización con respecto al algoritmo de Backtraking en términos de números de nodos generados, ya que desciende significativamente, sobre todo cuando se utiliza el cálculo de podas precisas, donde a veces la solución llega a ser voraz.

Palabras Clave: *Ramificación y Poda, Backtraking, árbol de soluciones, problema de asignación, matriz de beneficios, asignación voraz, cota de poda, nivel, solución parcial, raíz, nodo*

1. Introduccóon

En el siguiente informe se analizará el funcionamiento del algoritmo de **Backtracking** y su correpondiente optimización (mediante las podas) para un problema de asignación sin repeticiń de N galeones a N ciudades, en donde se busca maximizar el beneficio obtenido de realizar dichos ataques.

Para la primera parte analizaremos el funcionamiento del algoritmo de Backtracking y el coste computacional de algunas de sus funciones principales. Mientras que en la segunda mitad mediante la estrategia de **Ramificación y poda** veremos como se puede mejorar la complejidad exponencial de Backtracking.

2. Backtracking

Este algoritmo se basa en la exploración de todo el árbol de soluciones, haciéndolo por tanto muy costoso, pero verificando que siempre se encuentra la solución óptima. Este árbol para el problema de asignación que se nos presenta tiene un estructura permutacional, es decir para cada nivel del árbol cada nodo tiene exactamente un hijo menos que el nodo padre del nivel anterior.

2.1. Coste computacional del algoritmo

Sin embargo, en esta implementación Backtracking utiliza la función **Generar()** para realizar todas las posibles combinaciones N, por tanto se ejecuta 3^N veces más las tres primeras asignaciones que son siempre válidas. Ya que al inicializar el vector a `{-1}` se generan 3 estados (o soluciones parciales, pueden entenderse como nodos, para generalizar luego a Ramificación y Poda) donde se asigna cada una de las N Ciudades al primer Galeón, en el caso de $N = 3$.

Por tanto, el coste computacional del algoritmo de Backtracking en esta implementación es de $O(N^n)$ donde N es el número de ciudades o galeones a asignarse entre si (la dimensión horizontal/vertical de la Matriz), que en este caso solo trataremos para $N = 3$, $N = 6$.

2.2. Interpretación de funciones e implementación

2.2.1. Que representa el vector solución

Para implementar Backtracking lo primero es definir como se representa la **solución**, en este caso es un array de enteros $[c_1, c_2, \ldots, c_n]$ donde c_i representa la ciudad asignada al Galeón i (pensando en i como el índice para acceder a esa posición del array) y es un número positivo $<= N$ (excepto en la inicialización por lo que se comentó anteriormente).

2.2.2. Qué función realiza la asignación de ciudades a galeones

Para realizar la asignación de galeones a ciudades se utiliza la función `Generar(nivel, s)` que para el galeón Nº nivel le asigna la siguiente ciudad disponible, es decir incrementa el valor de ciudad de $s[nivel]$, asociando así el galeón correpondiente con una nueva ciudad.

2.2.3. Qué comprueba la función MasHermanos()

Para saber poder explorar todo el árbol de soluciones en nuestra implementación de Backtraking, cada vez que la tupla `s` sea una solución parcial correcta del problema (es decir no rompa la restricción de no repetir asignaciones) se pasa al siguiente nivel para explorar primero el arbol en profundidad.

Por tanto, a la hora de volver a un nivel anterior interesa saber si un nodo es el último valor posible de la permutación en su nivel, esto lo comprueba en nuestro código la función **MasHermanos()**, que simplemente verifica que la asignación de un determinado Galeón (puede pensarse como un nivel del árbol), no supera al número de ciudades.

```
return solucion[galeon] <N-1
```

Figura 1: Implementacion **MasHermanos()**

En este caso comprueba que sea menor que $N-1$ ya como el array se accede desde la posición 0, pues las ciudades también se enumararon de 0 hasta $N-1$, para N ciudades. Por tanto si vale $N-1$ es porque en ese nivel se ejecutó N veces la función **Generar()**, o lo que es lo mismo, el nodo correspondiente no tiene más hermanos que puedan ser generados.

2.2.4. Objetivo de la función Retroceder()

Como ya se indicó antes esta implementación explora primero los niveles más profundos del árbol de soluciones, ya que cada vez que una solución parcial sea factible (esto lo comprueba la función **Criterio()**) y no estemos en el nivel más bajo, el nivel se ve incrementado.

Por tanto desde el nivel inferior del árbol, cuando ya probásemos todas las posibilidades (**MasHermanos()**, con `s[N-1] = N-1` devuelve falso entonces según la Figura 1) entonces debemos volver al nivel anterior, para eso se utiliza la función **Retroceder()**.

Además, esta función también actualiza el beneficio actual de desasignar ese ataque del último galeón a la última ciudad y reduce el nivel, en caso de utilizar el vector usadas, que veremos más adelante para que sirve, se marcaría la ciudad en cuestión como libre para ser atacada por otros galeones.

Este proceso se repite, ya que la función se encuentra dentro de un bucle while hasta que nos encontremos en un nivel donde ese galeón no esté asociado a la última ciudad posible, excepto cuando estemos en el nivel 0, donde no se puede retroceder a la raiz.

2.3. Experimentación

A continuación se muestran los resultados de nodos generados para una asignación de N galeones a ciudades, es decir una matriz de tamaño $N \times N$ con resultados de el número de iteraciones que se han ejecutado las correspondientes funciones del algoritmo y si se ha optimizado utilizando el vector de ciudades usadas o no (ver Tabla 1).

Sobre todo cabe destacar la destacable difeerencia de pasos en **Criterio()**, cuya complejidad es $O(n)$ y por tanto se han contado el número de iteraciones del bucle por el número de ejecuciones de la función y que más adelante analizaremos como afecta la utilización de un vector auxiliar para identificar si una ciudad ha sido o no usada, sacrificando tan solo un aumento de memoria a cambio de una complejidad temporal $O(1)$.

Se pueden ver a continuación el máximo beneficio obtenido para cada tamaño de N, así como el vector solución:

- N = 3 **Mejor beneficio:** 42 **Mejor solución:** s = [1, 0, 2]

- N = 6 **Mejor beneficio:** 111 **Mejor solución:** s = [3, 5, 0, 2, 1, 4]

n	Vector usadas	Nodos visitados	Pasos Criterio	Pasos Generar	Pasos Solución	Pasos Más Hermanos	Pasos Retroceder
3	No	16	69	30	30	40	10
3	Sí	16	48	30	30	40	10
6	No	1957	37446	7422	7422	8659	1237
6	Sí	1957	11742	7422	7422	8659	1237

Tabla 1: Análisis de bactracking

2.4. Conclusiones

2.4.1. Análisis teórico

El algoritmo de backtracking implementado sin la utilización del vector usadas, presenta un orden de complejidad computacional de $O(n!)$, debido a que en el peor caso se tendrían que evaluar todas las posibles permutaciones. El vector usadas ayuda a reducir la cantidad de pasos al evitar realizar operaciones redundantes, haciéndolo más eficiente.

2.4.2. Análisis práctico

- Por una parte, es apreciable la diferencia de pasos criterio realizados en cada paso, no tanto en N= 3 donde sin usadas el número de pasos Criterio es 69 y con el vector ya bajaría a 48, pero es a partir de N= 6 donde es mucho más notoria la diferencia, pasando de 37446 a 11742.
- De esta forma, aunque solo se ahorren pasos Criterio, al tratarse de una diferencia tan grande a partir de valores de N relativamente pequeños, lo hace mejor para tratar problemas de gran tamaño.

3. Ramificación y Poda

La implementación de este algoritmo constituye una mejora con respecto a backtracking a la hora de explorar nodos, reduciendo significativamente esta cantidad. En este apartado se analizarán dos estrategias de este algoritmo utilizando dos estimaciones distintas para las cotas así como los cambios necesarios para cambiar de un problema de maximización a uno de minimización

3.1. Resultados de ambos algoritmos

El tratar de resolver el mismo problema, los resultados coinciden exactamente con los devueltos por backtracking, sin embargo el número de nodos explorados es mucho menor:

- **Mejor beneficio:** 42 **Mejor solución:** 2 1 3
- **Mejor beneficio:** 111 **Mejor solución:** 4 6 1 3 2 5

N	Nodos explorados
3	9
6	1248

Tabla 2: Asignación trivial

N	Nodos explorados
3	4
6	105

Tabla 3: Asignación precisa

3.2. Relación entre complejidad y reducción del número de nodos

- Al realizar las estimaciones **triviales**, la complejidad no resulta demasiado alta al realizar operaciones más simples, lo que conlleva a una estimación más general y que, por tanto, el número de nodos explorados no se reduzca tanto debido a que no se hacen tantas podas y es más factible que un nodo entre en la LNV.
- Por otra parte, las estimaciones **precisas**, como su nombre indica, permiten obtener unas cotas mucho mejores al utilizar un algoritmo de asignación voraz. De esta forma, el calculo de estas cotas conlleva un coste computacional más alto debido a estas operaciones

3.3. Comparación con backtracking

Para vólumenes grandes de datos, en nuestro problema para valores cada vez más grandes de N, el ahorro de explorar menos nodos es cada vez mayor con respecto al gasto de hacer el cálculo extra de cálcular las cotas.

Para problemas pequeños como con $N = 3$ conincide que la asignación de galeones a ciudades es directamente voraz (por eso resultan 4 nodos explorados como se ve en la tabla 3, es una única rama de nivel 4), no suele ser una diferencia muy significativa en el total de nodos generados, por lo que Backtraking podría ser una alternativa interesante.

Igualmente podemos ver que pasamos de explorar 16 nodos con Backtraking a 9 con Ramificación y Poda con cotas tribiales (las cuales no requieren más cálculo que encontrar el máximo de la matriz e ir realizando sumas y acumulando valores), por tanto la única desventaja de Ramificación y Poda podría ser que requiere más espacio en memoria para almacenar todas las cotas de cada Nodo, sus vectores `usadas[N]` y sus `tuplas[N]` individuales, cuando en Backtraking con un vector N elementos nos bastaba.

En problemas de talla mayor (como $N = 6$) asignaciones triviales no resulta tan interesante, ya que la cantidad de nodos que se pueden podar es muy baja, por tanto probablemente el hecho de duplicar (o incluso más) la reserva de memoria no salga tan a cuenta. Sin embargo, con asignaciones precisas hemos generado $\approx 5\,\%$ de los nodos que generarmos con Backtraking, mientras que el consumo de memoria sigue siendo el mismo que en cotas tribiales, únicamente aumenta el coste computacional de explorar cada nodo.

3.4. Modificaciones para el problema de minimización

Para poder adaptar el algoritmo para pasar de tratar un problema de maximización a uno de minimización es necesario introducir una serie de cambios:

- En primer lugar, en cuanto a la inicialización de los valores, para maximización se establece a $-\infty$ o un valor muy bajo para que cualquier valor encontrado en la primera iteración ya sea mayor, de modo que en minimización se busca lo contrario, que el primer valor siempres sea menor, por lo que se debe inicializar el beneficio inicial a $+\infty$ o un valor muy alto inalcanzable en el problema dado.
- Por otra parte, en maximización C se actualiza cada vez que una cota inferior supera ese valoro si se encuentra una solución mayor, y en minimización lo que busacamos es que sea lo más pequeña, por lo que C deberá atualizar como $min(C, Valor(y))$
- En cuanto a la estrategia de poda, lo que conviene en minimización es quedarnos con aquellos nodos con los que se pueda obtener un menor valor, de modo que la poda deberá hacerse comparando la cota inferior, si se cumple que $CI >= C$
- Otro cambio necesario es en la estrategia de ramificación, ya que en maximización se escoge aquel nodo de la LNV con mayor beneficio estimado, así que se tiene que cambiar la estrategia de MB-LIFO por LC-LIFO
- Finalmente, para el calculo de las cotas, se utilizarán cálculos que reflejen los valores más bajos y losa mejores costos posibles desde el nodo en expansión.

4. Conclusión General

Finalmente, ambos algoritmos aseguran encontrar la solución óptima que buscamos (si existe), por tanto dependiendo del **contexto** escogeremos una implementación u otra, priorizando Ramificación y Poda en problemas de tamaño mayor, donde sus posibles combinaciones de elementos se disparan, para acotar el número de soluciones parciales que hemos de tener en cuenta, y donde **calcular las cotas** sea más sencillo.

En problemas pequeños y sencillos Backtraking puede ofrecer una solución, sencilla, **fácil de implementar** y menos costosa en **consumo de memoria** siempre que el número de soluciones posibles del problema no sea muy elevado. Además su adaptación al problema de minimización es más sencilla. Utilizar un **vector de asignaciones** ya realizadas sale realmente rentable ya que si bien el consumo de memoria asciende de $O(1)$ a $O(N)$, el consumo computacional del algoritmo desciende drásticamente de $O(N)$ a $O(1)$, para la función **Criterio()**, la que más veces se ejecuta en nuestra implementación de Backtraking.