Algoritmos y Estructuras de Datos

Práctica 9

Problema de asignación: resolución mediante estrategia de vuelta atrás

Xiana Carrera Alonso

xiana.carrera@rai.usc.es

Diciembre, 2020

Índice

1. Introducción

En esta práctica se plantea el estudio del problema de asignación de un número n de tareas a n personas, cuyo rendimiento varía en función de la tarea que realicen.

La cuestión central es la siguiente: ¿cuál es la combinación de personas y tareas que resultará en el mayor beneficio total? Para resolverla, se ha optado por emplear la estrategia algorítmica de vuelta atrás, también llamada *backtracking* o método de retroceso. Así, la obtención de la solución óptima pasará por una búsqueda exhaustiva y sistemática entre todas las soluciones posibles.

Los datos de los beneficios se darán en forma de una matriz B de tamaño $n\ x\ n$, donde cada fila corresponderá a una persona y cada columna, a una tarea. Por lo tanto, dado un elemento $B[i, j]$, este representará el rendimiento asociado a la persona i cuando realiza la tarea j. Se tendrán que realizar n asignaciones de este tipo, una por persona (o tarea), con la restricción de que los trabajos no se pueden repetir. Es decir, siempre se cumple que dos individuos distintos llevan a cabo cometidos distintos. Luego tendremos que maximizar:

$$\sum_{i=1}^{n} B[p_i, t_i] \tag{1}$$

donde $p_i \neq p_j \implies t_i \neq t_j,\ \ \forall i \neq j,\ \ i, j \in 1, ..., n.$

Tras un análisis inicial teórico de las funciones y estructuras que utiliza esta estrategia, se mostrará la implementación del algoritmo que se ha llevado a cabo y, finalmente, se estudiará cuál es la complejidad del mismo.

1.1. Optimización

Para utilizar *backtracking*, será necesario contar con una función Criterio que asegure si una determinada combinación de personas y tareas es válida. Para este caso, el único requisito será asegurar la no repetición de tareas en la solución. No obstante, esto resulta en una comprobación que llega a ser lineal.

Para reducir el número de operaciones, se puede optar por registrar en un array de enteros de longitud n, usada, qué trabajos se han ido seleccionando. Para cada tarea, se guardará el número de veces que aparece en la solución que se ha ido construyendo. Si una solución es válida, usada contendrá un 1 en todas sus posiciones, porque todos los trabajos tendrán que haber sido asignados. Si alguna posición contiene un entero mayor que 1, el trabajo correspondiente aparecerá repetido, lo que implica que se debe descartar la asignación.

En consecuencia, Criterio se simplifica a una única comprobación, con lo que pasa a tener orden constante.

En los distintos apartados presentados a continuación se hará una contraposición entre las dos elecciones, indicando qué se tendría que cambiar en el programa. Uno de los objetivos del análisis de eficiencia será además determinar cuál es la implementación preferible.

2. Cuestiones iniciales

2.1. Representación de la solución mediante un vector s

Para expresar las n combinaciones correspondientes a una solución s, se utilizará un vector de longitud n dado desde el punto de vista de las personas, esto es:

$$s = (t_1, t_2, ..., t_i, ..., t_n) \tag{2}$$

con $i \in 1, ..., n$ y $t_i \neq t_j \ \ \forall i \neq j, \ \ j \in 1, ..., n$.

De este modo, cada índice i del vector se corresponde a una persona, y el elemento correspondiente, t_i, es su tarea.

Nótese que en C los arrays comienzan en 0, de modo que habría que adaptar la notación:

$$s = (t_0, t_1, ..., t_i, ..., t_{n-1}) \tag{3}$$

con $i \in 0, ..., n-1$ y $t_i \neq t_j \ \ \forall i \neq j, \ \ j \in 0, ..., n-1$.

Entonces, cada individuo vendría representado por un índice que varía entre 0 y $n-1$. Del mismo modo, los trabajos se numerarían de 0 a $n-1$, y a una persona arbitraria i se le asignaría una tarea t_i, distinta de todas las que realizan el resto de trabajadores.

Esta elección dará lugar a recorridos por un árbol permutacional, adecuado para representar problemas de permutaciones, como es el caso. Si numeramos los niveles desde -1 (la raíz) hasta $n-1$ (las hojas), en cada uno el número de nodos vendrá dado por

$$\frac{n!}{(n - nivel - 1)!} \tag{4}$$

Se puede comprobar que en cada nivel se tiene un número de nodos 1, n, $n(n-1)$, ..., $n!$. El -1 en el denominador se utiliza para ajustar los índices a los empleados en C.

Con esta numeración, hay una correspondencia biunívoca entre los índices de cada nivel, cada persona y cada posición del array.

Cabe destacar que en la implementación del problema, el vector s se irá actualizando para cada nodo visitado, independientemente de si representa una posible solución o no. Por consiguiente, es necesario definir cómo se va a indicar si a una persona aún no se le ha asignado tarea. Como se utiliza un vector de enteros, se ha optado por utilizar el valor -1 en esta situación (veremos el sentido de tal decisión cuando desarrollemos el cometido de la función Generar).

Otros posibles métodos de representación de la solución serían con un vector dado desde el punto de vista de las tareas (análogo a la forma aquí explicada, pero los índices del vector son los trabajos y sus elementos asociados, las personas que los realizan) o con una matriz binaria de asignaciones.

2.2. Asignación de tareas

Una de las claves del problema es establecer cuál será el procedimiento a la hora de asignar un trabajo a un individuo, esto es, determinar cómo quedará reflejada esa acción. Precisaremos entonces usar la función Generar.

Supongamos que nos encontramos en un punto de la resolución en el que queremos decidir qué tarea dar a una persona. La información de la que disponemos es el nivel del árbol permutacional en el que nos encontramos y la solución parcial s que hemos construido hasta ese momento. Si llamamos con ambos a Generar, la función debe, por un lado, proponer una tarea (que puede ser válida o no) para el trabajador y, por otro lado, actualizar el beneficio acumulado consecuentemente.

Para lo primero, lo que hará Generar es incrementar en una unidad el elemento del vector s cuyo índice es igual al nivel actual. Esto significa que para la persona en cuestión, se probará la siguiente tarea en orden de numeración de las mismas. Como Generar realiza un incremento de uno en uno, garantizamos que se probará con todos los trabajos posibles.

Al haber seleccionado -1 como el identificador de que todavía no ha habido una asignación para un individuo, llamar a Generar entonces supondrá darle la primera tarea (aquella de índice 0).

Otras funciones (en concreto, MasHermanos) se encargarán de asegurar que no se continúe por caminos donde se utilice un cometido inexistente (que el elemento del vector s llegue como máximo a $n-1$, el último posible) o donde este se repita en el vector (función Criterio).

Para lo segundo, se sumará al beneficio acumulado el rendimiento correspondiente al par (persona, tarea) elegido. En caso de que ya hubiera un trabajo asignado anteriormente, es decir, s[nivel] era distinto de 0, habrá que eliminar ese rastro: se tendría que restar el beneficio correspondiente a B[nivel][s[nivel] - 1].

Bajo estas premisas, se ha desarrollado la función Generar como sigue:

```
// Da lugar a una tupla candidata a solución a partir de un nuevo nodo del nivel actual.
void Generar(int B[][N], int nivel, int s[], int *bact){
  s[nivel]++;          // Se toma la siguiente tarea posible (cuyo índice está entre 0 y N - 1)
  if (!s[nivel])       // Previamente, en la posición no se estaba usando ninguna tarea
    // Se suma el beneficio correspondiente a la persona del nivel y a la tarea s[nivel]
    *bact = *bact + B[nivel][s[nivel]];
  else                 // s[nivel] > 0, lo que implica que (s[nivel] - 1) > -1, es decir, había una tarea anteriormente
    // Además de sumar el nuevo beneficio, se resta el remanente correspondiente a la anterior asignación
    *bact = *bact + B[nivel][s[nivel]] - B[nivel][s[nivel] - 1];
}
```

2.2.1. Modificación de Generar con usada

Si optamos por la propuesta de utilizar un vector con las tareas usadas para simplificar la función Criterio, esta pasaría de la siguiente implementación:

```
// Comprueba si la tupla de s cumple los requisitos necesarios para que pueda llegar a ser una solución
int Criterio(int nivel, int s[], int *cumplenCriterio, int *numPasos){
  int i;

  // Se comprueba si en cualquier posición anterior de s hay la misma tarea que la última asignada (s[nivel])
  for (i = 0; i < nivel; i++){
    // El contador del número de pasos se actualiza en cada iteración por si el bucle se termina antes de tiempo
    (*numPasos)++;
    if (s[nivel] == s[i]) return 0;      // No se cumple el criterio
  }
  (*cumplenCriterio)++;    // Se actualiza el número de nodos que pasan Criterio
  return 1;                // Se cumple el criterio
}
```

a esta:

```
// Comprueba si la tupla de s cumple los requisitos necesarios para que pueda llegar a ser una solución
int CriterioUsada(int usada[], int nivel, int s[], int *cumplenCriterio, int *numPasos){
  if (usada[s[nivel]] == 1)   // La tarea solo se usa una vez
    (*cumplenCriterio)++;       // Si se cumple el criterio, se incrementa en 1 el contador de nodos válidos

```

```
  // Se aumenta el contador del número de pasos (no se incluyen aquellas
    operaciones relativas a cumplenCriterio)
  (*numPasos) += 1;

  // Devuelve 1 si la tarea del nivel actual solo se usó una vez, y 0 si aparece más veces en la tupla solución
  return usada[s[nivel]] == 1;
}
```

Otras funciones también pasarían por cambios. Entre ellas, Generar debería adaptarse para actualizar de forma consistente el array usada, de acuerdo al uso de tareas.

Si el individuo del nivel actual ya tenía un cometido, s[nivel], este dejará de ser empleado a favor de su siguiente. Por lo tanto, se tendrá que disminuir usada[s[nivel]] en 1. Después, se le dará a la persona el nuevo trabajo, s[nivel] + 1, y, entonces, habrá que registrar su reparto en usada.

La función modificada quedaría como sigue:

```
// Da lugar a una tupla candidata a solución a partir de un nuevo nodo del nivel actual.
void GenerarUsada(int usada[], int B[][N], int nivel, int s[], int *bact){
  if (s[nivel] != -1)     // La persona tenía ya asignada una tarea, por lo que usada[s[nivel]] era mayor que 0
    usada[s[nivel]]--;  // Se desasignará la tarea, por lo que se usa una vez menos en s
  s[nivel]++;           // Se toma la siguiente tarea posible (cuyo índice está entre 0 y N - 1)
  usada[s[nivel]]++;    // Se registra el uso de esa nueva tarea (aparece una vez más en s)

  if (!s[nivel])  // Previamente, en la posición no se estaba usando ninguna tarea
    // Se suma el beneficio correspondiente a la persona del nivel y a la tarea s[nivel]
    *bact = *bact + B[nivel][s[nivel]];
  else              // s[nivel] > 0, lo que implica que (s[nivel] - 1) > -1, es decir, había una tarea anteriormente
    // Además de sumar el nuevo beneficio, se resta el remanente correspondiente a la anterior asignación
    *bact = *bact + B[nivel][s[nivel]] - B[nivel][s[nivel] - 1];
}
```

2.3. Función MasHermanos

El rango de valores válidos para una posición dada de s es $[0, n-1]$, donde cada valor posible representa una de las n tareas. Por consiguiente, no podemos permitir que se alcancen valores iguales o superiores a n. Este es el cometido de MasHermanos:

```
// Comprueba si quedan nodos sin estudiar que tengan el mismo padre que el nodo actual. Es decir, comprueba si quedan combinaciones por probar manteniendo fijos los nivel elementos anteriores en s.
int MasHermanos(int nivel, int s[]){
  // Las tareas tienen asignados índices en el rango [0, N). De un nodo padre salen N tareas, desde 0 hasta N - 1
  return s[nivel] < N - 1;
}
```

Como $s[nivel] < n-1$, quedan nodos sin visitar con el mismo padre, por lo que aún es preciso explorar el nivel actual y los siguientes.

Por tanto, MasHermanos controlará las llamadas a Retroceder (además de $nivel > -1$, para evitar un bucle infinito cuando no quedan más ramas por explorar). Si MasHermanos no se cumple, los descendientes del nodo padre se han estudiado por completo, y se debe pasar a otras zonas del árbol.

MasHermanos no sufre cambios si se decide optimizar Criterio con el vector usadas.

2.4. Función Retroceder

Mientras Criterio y $nivel < n$ se cumplan, el esquema algorítmico provocará que siempre se explore hacia niveles mayores del árbol, es decir, hacia las hojas. Sin embargo, debemos establecer una forma de volver atrás, para explorar caminos con posibles soluciones que se dejaron abiertos.

Entra en juego entonces la función Retroceder, que deshace los cambios provocados por la última asignación, y devuelve el control al nivel anterior al actual en el árbol. Así, se resta de bact el beneficio añadido por la última generación, y se cambia el valor correspondiente a la misma en el vector s a -1, dejando constancia de que para el siguiente subárbol que se explore del nodo padre del padre, este nivel aún no ha sido utilizado. Por último, se resta una unidad de nivel.

Siguiendo este esquema, la función Retroceder en C quedaría como:

```
// Deshace los cambios provocados por la última asignación. Se vuelve al nivel
    anterior del árbol.
void Retroceder(int B[][N], int *nivel, int s[], int *bact){
  *bact = *bact - B[*nivel][s[*nivel]];           // Se elimina el beneficio sumado
     en la última asignación
  s[*nivel] = -1;     // La persona del nivel actual pasa a no tener tarea asignada
  (*nivel)--;         // Se vuelve al nivel anterior
}
```

2.4.1. Modificación de Retroceder con usada

Si se utiliza un vector de tareas usadas para reducir la complejidad de Criterio, Retroceder también tendrá que ser modificada acordemente.

Al poner el valor correspondiente al nivel actual en s a -1, se explicita que la tarea que se había asignado a la persona que representa el nivel deja de usarse. El cambio tendrá que reflejarse en el vector usada, de forma que Retroceder quedaría como:

```
// Deshace los cambios provocados por la última asignación. Se vuelve al nivel
    anterior del árbol.
void RetrocederUsada(int usada[], int B[][N], int *nivel, int s[], int *bact){
  *bact = *bact - B[*nivel][s[*nivel]];     // Se elimina el beneficio sumado en la
    última asignación
  usada[s[*nivel]]--;     // Se deja de usar la tarea que tenía el nivel, que
    aparecerá una vez menos en s
  s[*nivel] = -1;     // La persona del nivel actual pasa a no tener tarea asignada
  (*nivel)--;         // Se vuelve al nivel anterior
}
```

3. Esquema algorítmico

Suponiendo una inicialización correcta de las variables, el algoritmo de *backtracking* en sí tiene la siguiente estructura:

```
while (nivel > -1) {     // Condición de parada
  Generar(B, nivel, s, &bact);        // Se genera una nueva combinación
  if (Solucion(nivel, s, &cumplenCriterio, &numPasos) && bact > voa) {
    // SolucionCriterio llama a CriterioUsada. cumplenCriterio se actualiza si es
    necesario
    // Se registra el incremento en pasos totales
    // Se comprueba si la combinación generada es solución y si esta es mejor que
    la óptima actual

    voa = bact;              // Se actualiza el valor óptimo
    for (i = 0; i < N; i++)
      soa[i] = s[i];       // Se guarda la nueva solución
  }

  if (nivel < N - 1 && Criterio(nivel, s, &cumplenCriterio, &numPasos)) {
    // cumplenCriterio se actualiza en la propia función Criterio, si es necesario
    // Se pasa antes nivel < N - 1 para que cumplenCriterio solo se incremente para
     los nodos internos
    // Se actualiza el número de pasos totales
    // Si el nodo cumple el criterio y no es un nodo hoja, se pasa al siguiente
    nivel del árbol
    nivel++;
  }

  while (!MasHermanos(nivel, s) && nivel > -1)     // No quedan más hermanos, el
    nivel es válido
    Retroceder(B, &nivel, s, &bact);             // Se retrocede en el árbol
}
```

Optimizar Criterio no supondrá cambios en el esquema algorítmico, más allá de tomar las funciones con la nueva implementación:

```
while (nivel > -1) {             // Condición de parada
  GenerarUsada(usada, B, nivel, s, &bact);        // Se genera una nueva combinació
    n
  if (SolucionUsada(usada, nivel, s, &cumplenCriterio, &numPasos) && bact > voa) {
    // SolucionUsada llama a CriterioUsada. cumplenCriterio se actualiza si es
    necesario
    // Se registra el incremento en pasos totales
    // Se comprueba si la combinación generada es solución y si esta es mejor que
    la óptima actual

    voa = bact;                  // Se actualiza el valor óptimo
    for (i = 0; i < N; i++)
      soa[i] = s[i];           // Se guarda la nueva solución
  }
  if (nivel < N - 1 && CriterioUsada(usada, nivel, s, &cumplenCriterio, &numPasos))
     {
    // cumplenCriterio se actualiza en la propia función Criterio, si es necesario
    // Se pasa antes nivel < N - 1 para que cumplenCriterio solo se incremente para
     los nodos internos
    // Se actualiza también el número de pasos totales
    // Si el nodo cumple el criterio y no es un nodo hoja, se pasa al siguiente
    nivel del árbol
    nivel++;
  }
  while (!MasHermanos(nivel, s) && nivel > -1)          // No quedan más hermanos,
    el nivel es válido
    RetrocederUsada(usada, B, &nivel, s, &bact);      // Se retrocede en el árbol
}
```

4. Análisis de eficiencia

4.1. Complejidad del algoritmo

Para medir la complejidad del algoritmo de backtracking, debemos decidir en primer lugar qué utilizaremos como comparativa.

Podemos determinar el tiempo de ejecución del algoritmo multiplicando el número de nodos del árbol por el tiempo de ejecución de cada nodo, puesto que para cada uno se realizan operaciones equivalentes. Así, suponiendo constante dicho tiempo, podemos tomar como el parámetro para el que realizaremos el estudio el número de nodos.

No obstante, será necesario descartar aquellos nodos no válidos que no se incluirían en el árbol permutacional. Se trata de aquellos nodos que no pasen la función Criterio, de modo que el árbol nunca se llega a explorar en profundidad a partir de esos puntos.

De esto se deduce que lo que tenemos que medir para una ejecución es el total de veces que Criterio devuelve 1. Es decir, colocaremos un contador en la función que se incrementará en 1 si se cumplen las restricciones.

De forma teórica, podemos servirnos de la fórmula 4 para determinar el total esperado. Nótese que no debemos incluir un hipotético nivel -1, correspondiente a la raíz, pues únicamente sirve como punto de partida para la representación del árbol y, en realidad, "no existe" para el algoritmo.

Se tendrá entonces que:

$$
\begin{aligned}
n = 3 \implies numNodos &= \sum_{i=0}^{n-1} \frac{n!}{(n-i-1)!} = \sum_{i=0}^{2} \frac{3!}{(2-i)!} = \frac{3!}{2!} + \frac{3!}{1!} + \frac{3!}{0!} = 3 + 6 + 6 = 15 \\
n = 6 \implies numNodos &= \sum_{i=0}^{n-1} \frac{n!}{(n-i-1)!} = \sum_{i=0}^{2} \frac{6!}{(5-i)!} = \frac{6!}{5!} + \frac{6!}{4!} + \frac{6!}{3!} + \frac{6!}{2!} + \frac{6!}{1!} + \frac{6!}{0!} = \\
&= 6 + 30 + 120 + 360 + 720 + 720 = 1956
\end{aligned}
\tag{5}
$$

Veamos que los resultados en la práctica coinciden.

Debemos notar que en el esquema algorítmico utilizado hay dos puntos en los que se llama a la función Criterio. El primero es desde Solucion:

```
// Determina si la asignación de tareas a personas realizada es una solución al problema
// Aunque tiene la misma implementación que Solucion, se distinguen entre si llaman a Criterio o CriterioUsada
int SolucionCriterio(int usada[], int nivel, int s[], int *cumplenCriterio, int *numPasos){
  /*
  * Se comprueba que el nivel actual sea N - 1, el de la última persona, esto es, que se haya llegado a los nodos hoja. Además, se llama a Criterio para saber si la tupla s es válida. Esto es necesario porque en el esquema algorítmico se llama a Solucion antes que a Criterio, por lo que no se ha determinado todavía si la última generación cumple los requisitos establecidos.
  */

  // Como se comprueba antes nivel == N - 1 que Criterio, cumplenCriterio solo se actualiza para los nodos hoja
  return (nivel == N - 1 && CriterioUsada(usada, nivel, s, cumplenCriterio, numPasos));
}
```

Con esto, ya tenemos garantizado que se contarán los nodos hoja que verifiquen Criterio.

El segundo punto es desde la comprobación para avanzar de nivel, pero con otra condición: $nivel < n - 1$. Por lo tanto, aquí solo se comprobarán los nodos internos.

```
if (nivel < N - 1 && Criterio(nivel, s, &cumplenCriterio, &numPasos)) {
  // cumplenCriterio se actualiza en la propia función Criterio, si es necesario
  // Se pasa antes nivel < N - 1 para que cumplenCriterio solo se incremente para los nodos internos
  // Se actualiza el número de pasos totales
  // Si el nodo cumple el criterio y no es un nodo hoja, se pasa al siguiente nivel del árbol
  nivel++;
}
```

En conclusión, el primer método añadirá el último sumando en 5, mientras que el segundo sumará los restantes. Y, efectivamente, se obtienen los resultados esperados:

```
---------- RESULTADOS ----------
Valor de la solución óptima: 21
Solución óptima:
Persona 0 --> Tarea 1
Persona 1 --> Tarea 0
Persona 2 --> Tarea 2
Número de nodos que cumplen el criterio: 15
```

```
---------- RESULTADOS ----------
Valor de la solución óptima: 111
Solución óptima:
Persona 0 --> Tarea 3
Persona 1 --> Tarea 5
Persona 2 --> Tarea 0
Persona 3 --> Tarea 2
Persona 4 --> Tarea 1
Persona 5 --> Tarea 4
Número de nodos que cumplen el criterio: 1956
```

Figura 1: Resultados prácticos para n = 3 y n = 6, respectivamente

Si realizamos mediciones con otros tamaños, obtenemos:

n	Nodos que cumplen criterio
2	4
3	15
4	64
5	325
6	1956
7	13699
8	109600
9	986409

Cuadro 1: Complejidad

Busquemos ahora una generalización. Otra forma de expresar las fórmulas de 5 es

$$numNodos = n + n * (n-1) + n * (n-1) * (n-2) * ... * n! \tag{6}$$

Ahora bien, los dos últimos sumandos serán $\frac{n!}{1!}$ y $\frac{n!}{0!}$, de modo que coinciden. El resto de sumandos serán siempre de menor grado y pueden ser despreciados. Luego el total de nodos viables será 2n! + términos de menor grado, de modo que concluimos que la complejidad será O(n!).

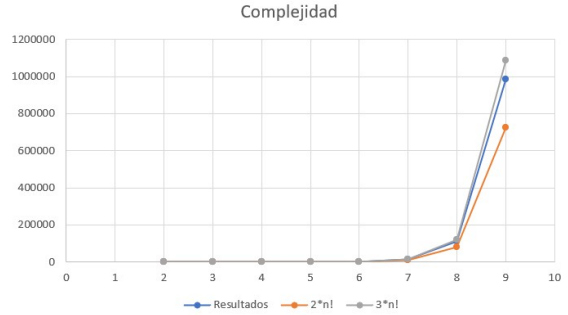

Figura 2: Comparativa entre los resultados y las gráficas de 2*n! y 3*n!

Como se puede ver en la figura, los datos obtenidos se ajustan bien a un orden factorial, concretamente, entre $2 * n!$ y $3 * n!$

Además, la complejidad es independiente de si usamos la implementación con el array usada o no, puesto que no se cambia el algoritmo (únicamente la forma de llevarlo a cabo), por lo que el árbol permutacional que se genera no sufre cambios.

4.2. Número de pasos

Para empezar, estudiemos el número de pasos de las funciones genéricas, sin el vector usada.

Generar, MasHermanos y Retroceder tienen todas orden constante, O(1). Ninguna de ellas tiene bucles de ningún tipo, y realizan accesos a memoria y otras operaciones que se pueden considerar instantáneas. Entonces, Generar tendría 3 pasos; MasHermanos, 1, y Retroceder, 3.

Por otro lado, Criterio es una función (casi) lineal: contiene un bucle cuyo número de iteraciones depende de nivel que, si bien no es la unidad de medida, n, sí va aumentando progresivamente en la ejecución del programa, hasta alcanzar un valor de n - 1. Es cierto que cuando analiza un nodo no válido el bucle se terminará antes de tiempo y no se llegará a las nivel iteraciones, pero si queremos restringirnos al peor caso no debemos tener en cuenta estas situaciones.

Por consiguiente, se podría considerar que Criterio llega a ser lineal en algunas de sus llamadas. Asimismo, Solución utiliza Criterio y efectúa una comparación más, de forma que su número de pasos será el mismo que los de Criterio + 1. Como además esta llamada siempre se realizará para los nodos hoja, podemos asegurar que en ese caso el bucle tendrá $n - 1$ iteraciones, es decir, que es lineal.

Para intentar aumentar la eficiencia, empleábamos el array usada, que lleva un registro permanentemente actualizado del número de veces que ha sido asignada cada tarea, simplificando Criterio.

Con esta nueva implementación, Criterio pasa a tener también orden constante, realizando una sola comparación, luego lo mismo ocurre para Solución, que realizará dos comparaciones en total.

MasHermanos no sufrirá cambios, pero sí Retroceder, que pasa a efectuar 4 pasos, y Generar, que pasa a efectuar 6. Aun así, al seguir manteniendo orden constante, comprobaremos que este cambio es favorable en cuanto a rendimiento. La otra desventaja sería el mayor gasto de memoria, pero al tratarse de un único array, es un aspecto despreciable.

Ejecutando el programa para n = 3 y n = 6, se obtiene:

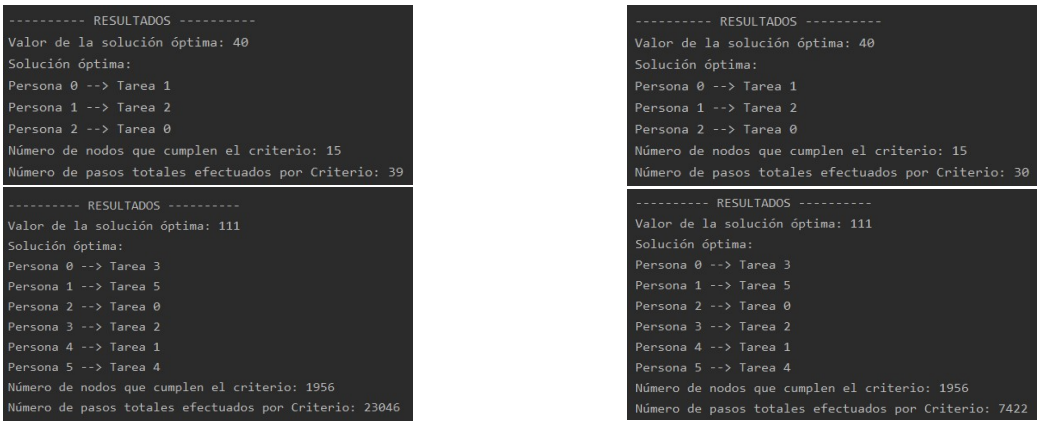

Figura 3: Número de pasos para n = 3 (sin usada, con usada) y n = 6 (sin usada, con usada), respectivamente

La diferencia no es demasiado grande si n es pequeño, porque el número de iteraciones que habrá que hacer en el bucle de Criterio no alcanza grandes magnitudes. No obstante, a medida que incrementemos n, nivel podrá alcanzar valores cada vez mayores, de forma que empezará a notarse la linealidad en contraposición al orden constante.

Si probamos el programa para más tamaños, obtenemos:

n	Pasos sin usada	Pasos con usada
2	4	6
3	39	30
4	316	164
5	2605	1030
6	23046	7422
7	221935	60620
8	2329720	554248
9	26579241	5611770

Cuadro 2: Número de pasos

El mayor número de pasos con usada que sin usada para n = 2 se explica porque en el primer caso, se tienen que comprobar los 6 nodos que habría en todo el árbol, incluyendo los no válidos, y que corresponderían al nivel 0 y al 1. En cambio, con la segunda opción únicamente se entra en el bucle para los nodos hoja (nivel 1), que son 4.

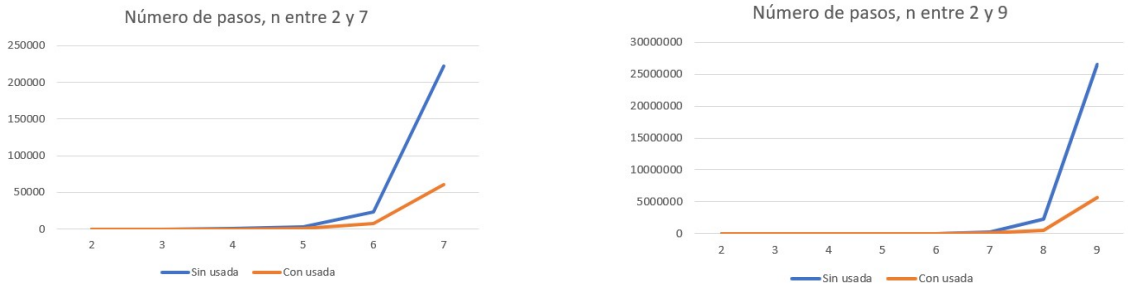

Figura 4: Número de pasos para n = 3 (sin usada, con usada) y n = 6 (sin usada, con usada), respectivamente

En las gráficas se puede apreciar lo significativa que es la mejora con el array usada, y lo rápido que se incrementa el número de pasos sin él. Cabe destacar que, con usada, los resultados son superiores al número de nodos válidos generados, medidos en el apartado anterior, puesto que hay que comprobar nodos no válidos. No obstante, los pasos son inferiores al número de nodos totales, $\sum_{i=1}^{n} n^i$, porque nunca se llegan a comprobar los descendientes de aquellos nodos que no pasan Criterio.

4.3. Conclusiones

A la vista de los resultados obtenidos, destaca el hecho de que la complejidad del algoritmo es factorial e independiente de la implementación. No obstante, esta se puede optimizar simplificando la función Criterio, con el objetivo de reducir su orden de lineal a simple, lo que disminuye significativamente el número de operaciones a realizar.