Algoritmos y Estructuras de Datos

Práctica 10

Problema de asignación: resolución mediante estrategia de ramificación y poda

Xiana Carrera Alonso

xiana.carrera@rai.usc.es

Diciembre, 2020

Índice

1. Introducción

En este documento se explican las características de la estrategia algorítmica de ramificación y poda, aplicada al problema de asignación de n tareas a n personas.

Los principales objetivos de este informe serán comparar la eficacia resultante de utilizar estimaciones triviales o precisas para la poda y explicar los cambios que habría que realizar sobre la implementación para adaptarla a un caso de minimización. Además, se comprobará si se producen cambios significativos con respecto al uso de la estrategia algorítmica de vuelta atrás para resolver el mismo planteamiento.

Se utilizará una notación análoga a la práctica de backtracking: B[i,j] representa la matriz de beneficios con filas para las personas y columnas para las tareas; s será un array solución representado desde el punto de vista de las personas, donde dos personas distintas no pueden tener la misma tarea; n será el orden de la matriz, coincidente con el número de personas y tareas; la numeración se adapta al lenguaje de programación C (los índices de personas y tareas van de 0 a $n-1$); etc.

2. Implementación

2.1. Nodos

Para empezar, se tendrá que definir una representación de los nodos del árbol de soluciones. En ellos se guardará un array de enteros con la solución parcial que se ha construido hasta ese punto, su nivel y el beneficio acumulado, es decir, la suma de los beneficios de cada una de las asignaciones persona-tarea que se están considerando en la solución parcial.

Para cada nodo se registrarán también la cota inferior (CI), la cota superior (CS) y el beneficio estimado (BE). Una vez determinados, ya no se vuelven a calcular.

Aunque es un dato opcional, se ha decidido guardar también un array con la información de las tareas que ya han sido utilizadas. Esto permite ahorrar comprobaciones que pueden llegar a ser lineales en los peores casos, a costa de un uso mayor de memoria. No obstante, la reducción en el tiempo es lo suficientemente significativa en comparación con el gasto en memoria como para justificar la elección.

Tal array, usadas[ ], contendrá en cada posición un 0 si la tarea correspondiente aún no ha sido usada, y un 1 si ya forma parte de la solución parcial del nodo.

Bajo estas condiciones, la implementación final será:

```
1 typedef struct{
2     int tupla[N];           // Solución parcial correspondiente al nodo
3     int nivel;              // Nivel del nodo
4     int bact;               // Beneficio acumulado hasta ese punto
5     float CI, BE, CS;       // Cota inferior, beneficio estimado, cota superior
6     int usadas[N];          // Vector de tareas usadas (cada posición tiene 0 o 1)
7 } Nodo;
```

2.2. Estimaciones

Será necesario desarrollar una forma de estimar el beneficio de un nodo dado del árbol de soluciones. Se han considerado dos opciones: una versión "trivial", más ligera computacionalmente, pero donde un gran número de nodos no serán descartados en la poda; y una versión "precisa", más costosa en cuanto a cálculos, pero con la que menos nodos pasarán la poda.

El beneficio estimado para cada nodo se obtendrá en ambos casos como la media entre su cota inferior y su cota inferior.

2.2.1. Estimaciones triviales

En el caso de las estimaciones triviales, la cota inferior del beneficio de las soluciones que descienden de un nodo será igual al beneficio acumulado del mismo.

```
1 // Calcula la cota inferior trivial de un nodo como su beneficio acumulado
2 void CI_trivial(tipoelem *x) {
3     x->CI = x->bact;
4 }
```

La cota superior se determinará sumándole a este beneficio acumulado el máximo elemento de la matriz por cada uno de los restantes niveles. Tal máximo habrá sido obtenido de antemano, pues exige comparar todos los elementos de la matriz, de modo que se trata de una operación de orden cuadrático.

```
1 // Calcula la cota superior trivial de un nodo
2 void CS_trivial(tipoelem *x, int maxB){
3     // A bact se le suma el máximo de la matriz, previamente calculado, por el nú
       mero de niveles no recorridos
4     x->CS = x->bact + (N - 1 - x->nivel) * maxB;
5 }
```

2.2.2. Estimaciones precisas

La cota inferior se determinará sumándole al beneficio acumulado el resultado de ejecutar una asignación voraz a partir del nodo. En ella, para cada persona se tomará la tarea que produce el mayor beneficio, con la condición de que no se repitan con ninguna de las elegidas hasta el momento (es decir, con las que dieron lugar al beneficio acumulado y con las que correspondan a la asignación voraz).

```
1 // Calcula la cota inferior precisa de un nodo
2 void CI_precisa(tipoelem *x, int B[][N]){
3     // Se suma el beneficio correspondiente a una asignación voraz a partir de x con
        su beneficio acumulado
4     x->CI = x->bact + _AsignacionVoraz(*x, B);
5 }
```

La cota superior se calculará sumando al beneficio acumulado el resultado de la operación MáximosTareas. Esta funciona de manera similar a la asignación voraz, con la diferencia de que ahora las tareas elegidas en MáximosTareas sí se pueden repetir entre sí (aunque no con las de la solución parcial del nodo, esto es, las que dieron lugar al beneficio acumulado).

```
1 // Calcula la cota superior precisa de un nodo
2 void CS_precisa(tipoelem *x, int B[][N]){
3     // Se suma el beneficio obtenido por el cálculo de _MaximosTareas con el
       beneficio acumulado de x
4     x->CS = x->bact + _MaximosTareas(*x, B);
5 }
```

2.3. Estrategia de poda

En todo momento se mantendrá actualizada la variable de poda C, que será igual al máximo entre las cotas inferiores de los nodos generados hasta el momento y el valor de cada una de las soluciones a las que se ha llegado.

Cuando se considere un nuevo nodo, este se descartará si su cota superior es menor o igual que C, pues esto indica que en ningún caso podrá dar lugar a una solución mejor que otra de las posibilidades consideradas hasta entonces y, por consiguiente, no merece la pena explorar sus descendientes.

2.4. Estrategia de ramificación

Otra de las variables a mantener será una lista de nodos vivos (LNV), que guardará todos aquellos nodos que no hayan sido podados y tengan descendientes por explorar.

Después de generar un nodo, se elegirá el siguiente de entre los que pertenezcan a la LNV. Luego habrá que decidir un criterio para elegir cuál tomar de todos ellos.

La estrategia de ramificación seleccionada es MB-LIFO. Será preferible tomar siempre los nodos más prometedores (aquellos con mayor beneficio estimado), pues si dan lugar a condiciones más exigentes para la variable de poda, contribuirán a reducir el espacio de búsqueda.

Por otro lado, en caso de que varios nodos coincidan en su beneficio estimado, se desempatará considerando LIFO: "last in, first out". Es decir, se cogerá el último nodo que fue introducido en la LNV de entre los que empataron.

Con el objetivo de no tener que recorrer la lista en busca del nodo con mayor BE cada vez que se quiera seleccionar uno, se ha optado por introducirlos de forma ordenada. Entonces, se colocará cada nodo justo antes de todos los que tengan BE menor o igual que él. Esto permitirá ir sacando los elementos en orden decreciente de beneficio y, en caso de empate, tomando primero aquellos que se hayan introducido de últimos:

```
1  // Inserta un nodo en la lista de nodos vivos de forma ordenada. La lista va de
       mayor a menor beneficio
2  // Se emplea en AsignacionRyP_trivial y en AsignacionRyP_precisa
3  void _inserta_LIFO(lista *LNV, tipoelem m){
4     tipoelem n;
5     posicion p = primero(*LNV);
6
7     while (p != fin(*LNV)) {          // Se recorre la lista
8        recupera(*LNV, p, &n);        // Se extrae un nodo
9
10       // Se está usando una estrategia LIFO. Por consiguiente, el nodo tiene que
       quedar colocado después de todos los que tengan un beneficio estimado superior
       a él, pero antes de todos los que tengan uno igual o menor.
11
12       if (n.BE <= m.BE) break;      // Se ha encontrado la posición de inserción
13       p = siguiente(*LNV, p);       // Se sigue recorriendo la lista
14    }
15
16    inserta(LNV, p, m);            // Se inserta el nodo antes del primer elemento con
       BE menor o igual que él
17 }
```

2.5. Esquema algorítmico

El esquema algorítmico resultante para las estimaciones triviales será el siguiente:

```
1  while(!esvacia(LNV)){              // Condición de parada
2     x = Seleccionar(LNV);                  // Se saca de LNV un nodo siguiendo la
3                                            // estrategia MB-LIFO
4     suprime(&LNV, primero(LNV));           // El nodo se elimina de la lista
5     if (x.CS > C){                 // Estrategia de poda. Si no se cumple, no se
6                                    // generan los hijos del nodo
7        y.nivel = x.nivel + 1;      // Se puede realizar fuera del for. Los hijos
8                                    // tendrán un nivel más que su padre
9        for (i = 0; i < N; i++){              // Se generan los N posibles hijos, uno
10                                            // por tarea
11          for (j = 0; j < N; j++) {        // Se copian las tuplas con la solución
12                                           // parcial y tareas usadas del padre
13             y.tupla[j] = x.tupla[j];
14             y.usadas[j] = x.usadas[j];
15          }
16
17          if (!x.usadas[i]){             // El nodo es válido. La tarea no fue usada
18                                         // anteriormente
19             y.tupla[y.nivel] = i;          // A la persona correspondiente al nivel se
20                                         // le asigna la tarea i
21             y.usadas[i] = 1;
22             y.bact = x.bact + B[y.nivel][i];         // Se suma el beneficio
23                                         // correspondiente a la asignación
24          } else {
25             continue;          // No es necesario realizar más operaciones. Se
26                                // comprueba la siguiente tarea
27          }
28
29          // El nodo es válido. Se calcula su cota inferior, cota superior y
        beneficio estimado
30          CI_trivial(&y);
31          CS_trivial(&y, maxB);
32          BE(&y);
33
34          (*nodosExplorados)++;      // Se han calculado las estadísticas del nodo
35
36          if (Solucion(y) && y.bact > sBact){           // El nodo es una solución
37                                                       // mejor que la óptima actual
38             for (j = 0; j < N; j++){
39                s[j] = y.tupla[j];                    // Se guarda la nueva solución
40             }
41
42             sBact = y.bact;                           // Se actualiza el valor óptimo
43                                                       // para las soluciones
44             C = _max(C, sBact);                       // Si sBact supera a C, esta se
45                                                       // actualiza a sBact
46
47          } else if (!Solucion(y) && y.CS > C){         // El nodo no es solución,
48                                                       // pero es prometedor
49             _inserta_LIFO(&LNV, y);                   // Se guarda
50                                                       // en la lista de nodos vivos
51             C = _max(C, y.CI);                        // Se actualiza C a la CI del
52                                                       // nodo si es necesario
53          }
54       }
55    }
56 }
```

Con las estimaciones precisas, se introduce un cambio. Cabe la posibilidad de que en un nodo coincidan la cota inferior y la superior, es decir, que la solución está perfectamente acotada. En ese caso, se realiza una asignación voraz que determina el valor final, en lugar de generar los descendientes de la forma normal. Esta situación no se podría dar con estimaciones triviales, porque el beneficio acumulado, la cota inferior, solo puede ser igual a la cota superior si el nodo es una hoja. En tal caso, la dificultad ya está resuelta.

```
while(!esvacia(LNV)){           // Condición de parada
   x = Seleccionar(LNV);                  // Se saca de LNV un nodo siguiendo la
                                          // estrategia MB-LIFO
   suprime(&LNV, primero(LNV));           // El nodo se elimina de la lista
   if (x.CS > C){                 // Estrategia de poda. Si no se cumple, no se
                                  // generan los hijos del nodo
      y.nivel = x.nivel + 1;      // Se puede realizar fuera del for. Los hijos
                                  // tendrán un nivel más que su padre
      for (i = 0; i < N; i++){     // Se generan los N posibles hijos, uno por tarea
         for (j = 0; j < N; j++) {      // Se copian las tuplas con la solución
                                        // parcial y tareas usadas del padre
            y.tupla[j] = x.tupla[j];
            y.usadas[j] = x.usadas[j];
         }

         if (!x.usadas[i]){           // El nodo es válido. La tarea no fue usada
                                      // anteriormente
            y.tupla[y.nivel] = i;        // A la persona correspondiente al nivel se
                                         // le asigna la tarea i
            y.usadas[i] = 1;
            y.bact = x.bact + B[y.nivel][i];         // Se suma el beneficio
                                                     // correspondiente a la
                                                     // asignación
         } else {
            continue;        // No es necesario realizar más operaciones. Se
                             // comprueba la siguiente tarea
         }

         // El nodo es válido. Se calcula su cota inferior, cota superior y
    beneficio estimado
         CI_precisa(&y, B);
         CS_precisa(&y, B);
         BE(&y);

         (*nodosExplorados)++;    // Se han calculado las estadísticas del nodo

         if (!Solucion(y) && y.CS >= C && y.CS == y.CI) {
            // Se obtiene inmediatamente el valor de la solución mediante una
    asignación voraz. Aunque se cumpla y.CS == y.CI, si y.CS < C, el nodo es podado
     igualmente. Además, si y es solucion, tampoco se entra en el if, porque la
    llamada a _SolAsignacionVoraz no es necesaria.

            // Este caso no es posible en AsignacionRyP_trivial, porque si el
    beneficio acumulado (la cota inferior) es igual al máximo de la matriz * N (la
    cota superior), entonces necesariamente se está en un nodo hoja. Esto es debido
     a que, en el mejor caso, que sería aquel en el que todas las asignaciones
    realizadas hasta el nivel actual dieron beneficio igual al máximo de la matriz,
     bact sería igual a dicho valor * el nivel actual + 1. Para que coincida con la
     cota superior, necesariamente el nivel es igual a N - 1, el último del árbol.


            (*nodosExplorados) += N - 1 - y.nivel;       // Niveles restantes
            y = _SolAsignacionVoraz(y, B);
         }

         if (Solucion(y) && y.bact > sBact) {          // El nodo es una solución
                                                     // mejor que la óptima actual
            for (j = 0; j < N; j++) {
               s[j] = y.tupla[j];                  // Se guarda la nueva solución
            }

            sBact = y.bact;     // Se actualiza el valor óptimo para las soluciones
            C = _max(C, sBact);  // Si sBact supera a C, esta se actualiza a sBact
         } else if (!Solucion(y) && y.CS > C){        // El nodo no es solución,
                                                      // pero es prometedor
            _inserta_LIFO(&LNV, y);    // Se guarda en la lista de nodos vivos
            C = _max(C, y.CI);  // Se actualiza C a la CI del nodo si es necesario
         }
      }
   }
}
```

3. Conclusiones sobre los efectos de la poda

Para determinar la eficiencia de la poda, se contará el número de nodos explorados. A efectos prácticos, se considera que un nodo ha sido explorado si se ha llegado a calcular su cota inferior, su cota superior y su beneficio estimado, pues son las funciones que llegan a suponer el mayor esfuerzo computacional. Por lo tanto, generar un nodo implica que será contado, independientemente de si posteriormente es podado o no, o si una solución válida o no.

Si introducimos las matrices:

$$A = \begin{pmatrix} 4 & 9 & 1 \\ 7 & 2 & 3 \\ 6 & 3 & 5 \end{pmatrix}$$

$$B = \begin{pmatrix} 11 & 17 & 8 & 16 & 20 & 14 \\ 9 & 7 & 6 & 12 & 15 & 18 \\ 13 & 15 & 16 & 12 & 16 & 18 \\ 21 & 24 & 28 & 17 & 26 & 20 \\ 10 & 14 & 12 & 11 & 15 & 13 \\ 12 & 20 & 19 & 13 & 22 & 17 \end{pmatrix}$$

obtenemos:

Figura 1: Resultados para A y B, respectivamente

En el caso de A, tenemos tamaño n = 3, y la diferencia entre el número de nodos de estimaciones triviales y precisas es tan solo de 2. En el caso de B, donde la matriz ya tiene un orden considerable, n = 6, se aprecia mejor el cambio: de estimaciones triviales a precisas se reduce el número de nodos en 1143.

Estos resultados son ya de por sí indicativos de la tendencia general: que la elección de las estimaciones precisas compensa con tamaños mayores, mientras que las triviales son preferibles si estos son pequeños.

Recordemos que el número de nodos válidos, como se comprobó en la práctica de vuelta atrás, tiene un orden factorial. Esto provocará una cantidad de operaciones inmanejable a poco que se incremente n. Por consiguiente, lo primordial será disminuir el espacio de búsqueda, aunque esto suponga operaciones de mayor coste. La importancia de lo anterior se ve reflejado ya con n = 6, pues una diferencia de más de 1000 nodos es muy importante.

Aun así, no debemos ignorar la complejidad de las estimaciones. En el caso de las triviales, puesto que el beneficio acumulado se guarda en el nodo y el máximo de la matriz se ha calculado de antemano, el cálculo de CI, CS y BE tiene orden constante, con una sola operación en cada caso.

En contraposición, para las estimaciones precisas se tiene que analizar la estructura de las funciones AsignaciónVoraz y MáximosTareas. En ambos casos, el peso recae sobre un bucle cuyo

número de iteraciones depende del nivel del nodo a estudiar. Será $n - 1 - nivel$, donde el -1 es necesario para adaptar la notación a la de los arrays en C. Considerando todos los nodos, tendremos $O(n(n - 1 - nivel))$. Por tanto, la complejidad será cuadrática.

El peor caso se presentará para el nodo raíz, de nivel -1, para el que se realizarán n iteraciones.

Teniendo en cuenta esto, podemos concluir que es preferible usar operaciones más complejas para tratar de compensar la gran cantidad de candidatos. En concreto, se puede ver que las estimaciones precisas para $n = 6$ devuelven 105 nodos, cantidad muy inferior incluso a $6! = 720$.

Si, en cambio, tenemos matrices relativamente pequeñas, como es el caso de $n = 3$, es quizás preferible utilizar estimaciones triviales, pues el número total de nodos todavía es manejable incluso con poca poda, de modo que se puede priorizar la rapidez en el cálculo. No obstante, dado que no se necesitarán realizar demasiadas operaciones, la diferencia en el tiempo de ejecución probablemente sea prácticamente imperceptible.

Si realizamos algunas ejecuciones más, con datos introducidos al azar:

n	**Número de nodos con estimaciones triviales**	**Número de nodos con estimaciones precisas**
2	4	5
3	9	7
4	23	10
5	217	24
6	1248	105
7	5300	68
8	74325	97
9	351585	4596

Cuadro 1: Número de nodos explorados

Estos resultados no deben tomarse como referencia, pues el comportamiento del algoritmo depende directamente de los datos introducidos. Aun así, nos dan una indicación sobre la magnitud que alcanza la dimensión del árbol de soluciones, y la mejora en eficacia que supone una buena poda.

Figura 2: Comparativa

En comparación, estos fueron los resultados para vuelta atrás:

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

Cuadro 2: Nodos explorados con vuelta atrás

Se puede ver que los resultados son peores que con ramificación y poda, llegando casi a triplicar los de estimaciones triviales.

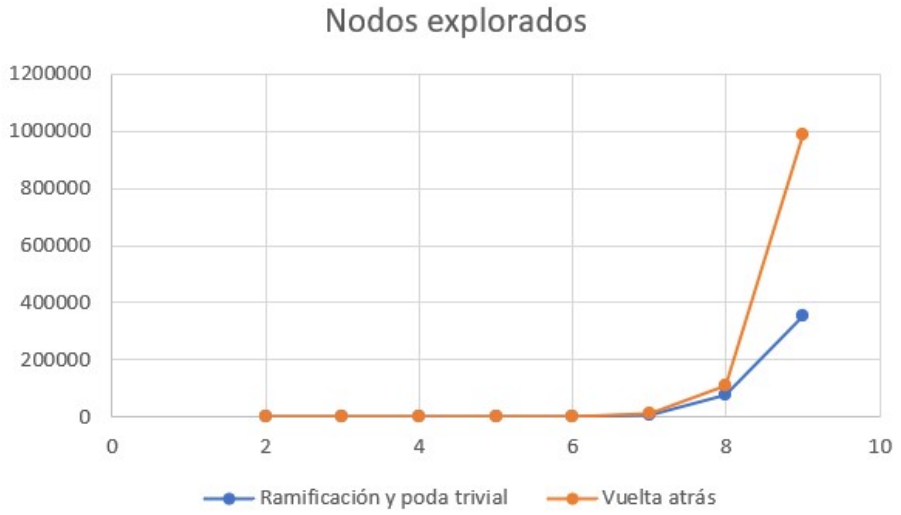

Figura 3: Comparativa entre vuelta atrás y ramificación y poda

En la práctica anterior se había comprobado que, con un vector para guardar las tareas usadas, la estrategia de *backtracking* usaba únicamente operaciones constantes. Sin embargo, basta compararla con el caso trivial de ramificación y poda, que también utiliza funciones de orden constante, para ver que el método tiene un amplio margen de mejora. Entre estas dos opciones, ramificación y poda es claramente la preferible, al ser la que más reduce el número de comprobaciones a llevar a cabo.

Entre vuelta atrás y ramificación y poda con estimaciones precisas, el razonamiento es análogo al que contraponía estas con las triviales. Si bien vuelta atrás es menos costosa computacionalmente, no merece la pena cuando el árbol de soluciones alcanza una gran escala. Y si ya se establecía que en esa situación, utilizar estimaciones precisas era preferible a las triviales, tanto más con respecto a vuelta atrás. Únicamente cabría contemplar la última opción cuando se tiene un n pequeño, pero en tal caso cualquier método será suficientemente rápido.

4. Minimización

Veamos qué habría que modificar si se deseara cambiar el problema a uno de minimización. Los elementos de la matriz pasarían de representar beneficio a representar coste, y las soluciones serán mejores cuanto menor sea la suma de su coste asociado total.

4.1. Condición de poda y cálculo de C

La variable de poda tendrá que ser modificada. Ahora, su función será la de acotar superiormente las soluciones. A medida que encuentren caminos más y más prometedores, C se irá reduciendo y, así, se podrán descartar aquellos caminos que no puedan dar lugar a costes totales bajos.

La definición de C es análoga a la del problema de maximización, pero invirtiendo los papeles de mínimos y máximos. Luego tomaremos:

$$C = min(\{CS(j), \forall j\ generado\}, \{Valor(s), \forall s\ solucion\ final\}) \tag{1}$$

Así, dado un valor de C, sabemos que o bien existe un nodo que da una solución con valor C o menor, o bien existe una solución que ha dado exactamente valor C. Se trata, pues, del menor valor consistente que permitirá descartar posibilidades. Si, por ejemplo, se actualiza C a algo menor estrictamente a todas las CS y al valor de todas las soluciones, podría suceder que todos los nodos diesen lugar a resultados cuyo valor coincida con su CS, de forma que las comparaciones se efectuarían con una cota incorrecta, al ser demasiado exigente.

Para minimización, un nodo x será podado si se cumple $x.CI \geq C$, pues esto indica que nunca generará una solución con menor valor que otros caminos. Que se cumpliera $x.CI = C$ querría decir que, como mucho, sus subárboles pueden dar soluciones igual de buenas que las que ya están en consideración. Por tanto, tampoco valdría la pena estudiar sus descendientes.

4.2. Estrategia de ramificación y valores iniciales

La estrategia de ramificación también tendrá que ser modificada. Ya no se estará buscando el mayor beneficio, sino el menor coste (LC). Como en maximización, se podrá tomar LIFO o FIFO, dependiendo de si se quieren hacer desempates tomando el último elemento introducido en la LNV o el primero, respectivamente.

Si se quiere seguir tomando siempre el primer elemento de la LNV, y que este sea el más prometedor, se tendrá que modificar el orden de inserción. Ahora, los nodos se introducirán en orden creciente de coste estimado, en lugar de decreciente, de modo que el primer nodo de la LNV es siempre el de coste mínimo.

Si se decide tomar LIFO para deshacer empates, al insertar un nodo se recorrerá la lista hasta encontrar alguno con beneficio estimado mayor o igual a este, y se colocará justo antes que él. De ese modo, será el primero de entre todos aquellos que tengan su mismo beneficio estimado, y anterior a los que tengan uno mayor.

Si se decide tomar FIFO para deshacer empates, se recorrerá la lista hasta encontrar un nodo con beneficio estimado estrictamente mayor, y se colocará el nuevo elemento justo antes que él. Por consiguiente, estará después de todos los nodos con beneficio estimado menor o igual, y antes de los que tengan uno mayor. Así, se mantiene la prioridad de llegada con sus iguales, pues todos los que ya estaban en la lista saldrán antes.

Las únicas modificaciones de los valores iniciales de las variables se producen en C y en sCact, variable análoga a sBact y que almacenaría el coste acumulado de la solución óptima actual.

C ahora se tendrá que inicializar a la cota superior de la raíz del árbol de soluciones, debido a su nueva definición (1).

Coste estimado	3	4	5	5	8	11	11	12
Posiciones	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Cuadro 3: Ejemplo de lista LNV para minimización. Si se quiere insertar un nodo con coste estimado 5, utilizando FIFO iría en la posición 4 (y los elementos de la posición 4 en adelante se desplazarían una posición a la derecha). Con LIFO iría en la posición 2 (y también se desplazarían los elementos a partir de dicho índice).

En cuanto a sCact, se debe tener en cuenta que ahora se actualizará la solución s a una nueva si el coste total de la solución que se está considerando es menor que el de s. Por tanto, sCact tendrá que empezar como un valor mayor a cualquiera posible para una solución (en caso contrario, s podría no modificarse correctamente).

Para representar este "infinito", se puede aprovechar que se tendrá que encontrar el elemento máximo de la matriz para el cálculo de las cotas. Sea max tal elemento. Entonces, si $sCact = max * n + 1$, garantizamos que es mayor que cualquier asignación de personas y tareas.

4.3. Cambios en las estimaciones de cotas superior e inferior

En el caso de las estimaciones triviales, se podrían mantener las funciones para maximización, sin cambios. El coste mínimo de cada camino a explorar sigue siendo igual al coste acumulado (cualquier nodo a mayores sumará coste extra), y el coste máximo se puede calcular como el coste acumulado sumado al máximo coste de la matriz multiplicado por el número de niveles no visitados, esto es, coste acumulado + max * (n - 1 - nivel).

No obstante, se podría pensar en mejorar el cálculo de la siguiente forma: si además de obtener el valor máximo de la matriz de antemano, se obtiene también el menor, la cota inferior se podría ajustar al coste acumulado sumado al mínimo de la matriz por el número de niveles no visitados, es decir, coste acumulado + min * (n - 1 - nivel). Sabemos que esta cota es válida, porque todos los elementos cogidos a partir del nodo actual son mayores o iguales que min.

La cota superior trivial no se podría concretar mejor, si se desea obtener en orden constante.

En cambio, sí que es obligatorio alterar el modo de calcular las cotas precisas. Se invertirán los papeles de MáximosTareas y AsignaciónVoraz. Ahora se empleará MínimosTareas en la cota inferior precisa, y AsignaciónVoraz en la superior.

En MínimosTareas el objetivo será darle a cada persona restante la tarea de menor coste que no haya sido elegida anteriormente. Por tanto, de cada fila se elige la tarea con menor beneficio que no esté usada en el array usadas[ ] del nodo, y este no se actualiza. Para hacer la comparación entre los elementos similar a la de MáximosTareas, se puede utilizar una variable, minCol, que tendrá que ser inicializada a "infinito", esto es, un valor mayor que cualquiera de la matriz (podría ser el máximo de esta + 1, por ejemplo).

La cota inferior en sí se obtendrá sumando al coste acumulado del nodo el resultado de MínimosTareas.

Esto resulta en una subestimación de las soluciones, justificando que la condición de poda sea CI >= C. Si la cota inferior de un nodo es mayor o igual que la variable de poda, en ningún caso se podrá mejorar lo ya estudiado.

En AsignaciónVoraz el objetivo será darle a cada persona restante la tarea de menor coste, sin repetir ninguna tarea.

Esto es una solución válida del problema, de modo que sabemos que o bien es la de coste mínimo, o bien existe una asignación mejor partiendo del nodo actual.

Las modificaciones en AsignaciónVoraz pasarían por obtener, para cada nivel, el menor elemento de la fila de la matriz. De nuevo, se puede inicializar una variable, minCol, a un valor superior a todos los de la matriz, y utilizarla para iniciar la comparación. Para cada elemento se debe comprobar que no se haya repetido con ninguno de los ya usados. Una vez obtenido el elemento mínimo, se añadirá su índice al registro de las tareas usadas.

El coste estimado se calculará para estimaciones triviales y precisas como la media entre la cota inferior y la superior.