

Algoritmia y Complejidad

Tema 5. Recursión intensiva (Backtracking)

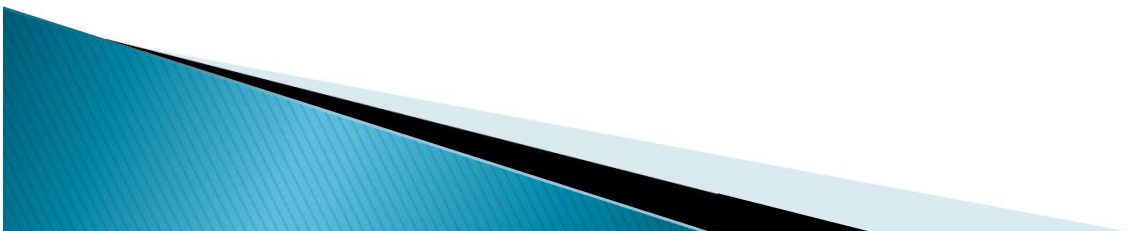
1.- Introducción.

- ▶ Existe una gran cantidad de problemas que se pueden formular en términos de grafos.
- ▶ A veces entendemos por grafo, una estructura de datos en la memoria de un ordenador, que representan nodos y aristas en dicha memoria y sobre el que se realizan una serie de operaciones bastante concretas.
- ▶ En estos casos, si se dispone de métodos algorítmicos que permitan resolver el problema (como los que hemos visto en temas anteriores), dichos algoritmos exigen la visita de todos los nodos y/o todas las aristas y un cierto orden en los recorridos.
- ▶ Otras veces esto no es así.



1.- Introducción.

- ▶ En otros casos el grafo solo existe de forma implícita, debido a su gran tamaño. Es el caso de grafos de juegos como el ajedrez. Los nodos serían las situaciones posibles y las aristas las jugadas válidas.
- ▶ La mayor parte del tiempo, lo único que se tiene es una representación de la situación actual (nodo actual que visitamos) y un pequeño número de situaciones diferentes.
- ▶ En cualquier caso, las técnicas que usamos para recorrerlos son las mismas.



2.- Búsqueda exhaustiva.

- ▶ Si un problema no tiene un método algorítmico que lo resuelva, haremos una búsqueda exhaustiva de soluciones entre todas las posibilidades. Este método se conoce como **fuerza bruta**: se generan todos los casos posibles y se testean uno a uno hasta encontrar las soluciones necesarias (a veces basta con encontrar una, otras veces hay que encontrar todas, y otras la mejor).
- ▶ Muchos de estos problemas pueden resolverse por etapas. Paso a paso se comprueba si se está creando una solución o si se han tomado decisiones que no conseguirán resolver el problema. En cada etapa se estudian las propiedades cuya validez ha de examinarse con objeto de seleccionar las adecuadas para proseguir con el siguiente paso.

2.– Búsqueda exhaustiva.

- ▶ La gestión de las etapas, las posibles decisiones que se toman y las relaciones entre ellas, suponen la generación de un árbol de decisiones. En los nodos se encuentran las soluciones parciales, y los enlaces entre ellos son las decisiones tomadas para cambiar de etapa.
- ▶ El avance a lo largo del árbol se detiene cuando se llega a una situación en que no puede tomarse ninguna decisión que permita obtener una solución, o cuando efectivamente se llega a resolver el problema.
- ▶ En estos casos se recupera una solución parcial anterior y se continua buscando si es necesario.



3.- Backtracking

- ▶ El backtracking es un método recursivo para buscar de soluciones por etapas de manera exhaustiva, usando prueba y error para obtener la solución completa del problema añadiendo decisiones a las soluciones parciales.
- ▶ Utiliza como árbol de decisiones la propia organización de la recursión. Cuando se avanza de etapa se realiza una llamada recursiva, y cuando se retrocede lo que se hace es terminar el correspondiente proceso recursivo, con lo que efectivamente se vuelve al estado anterior por la pila de entornos creada en la recursión.
- ▶ El árbol de decisiones no es una estructura almacenada en memoria, sino un árbol implícito y que se habitualmente se denomina árbol de expansión o espacio de búsqueda.



3.- Backtracking

```
proc Backtracking(E/S sol: vector; E k: natural) {1}  
  var hijos: lista de decisiones  
  hijos = crear_nivel(k) {2}  
  Desde i = 1 hasta total(hijos) Hacer  
    sol[k] = hijos(i) {3}  
    si es_solución(sol,k) entonces {3a}  
      tratar_solución(sol)  
    si no  
      si es_completable(sol,k) entonces {3b}  
        Backtracking(sol,k+1)  
      fsi  
    fsi  
  fdesde  
fproc
```

La llamada inicial al procedimiento Backtracking recibiría como parámetros una solución vacía y valor de k igual a 1.

3.– Backtracking

Este esquema se corresponde con las siguientes ideas:

1. Al llegar a un nodo (al entrar en una llamada recursiva) se tiene una solución parcial formada por $k - 1$ etapas.
2. Se generan todas las decisiones posibles a partir de la solución parcial actual.
3. Para cada una de esas decisiones, se incorpora a la solución parcial (por lo que ahora tendrá k etapas) y se comprueba si se ha conseguido una solución total
 - a. Si es una solución del problema, se gestiona: se guarda en una estructura de soluciones, se compara con otras anteriores, se muestra al usuario... y termina la búsqueda con esas decisiones.
 - b. Si no es una solución, si se detecta que la solución parcial es completable (en caso de tener una función capaz de detectar esta situación) se hace una nueva llamada recursiva con la nueva solución parcial y el nuevo nivel.

3.– Backtracking

- ▶ A este esquema básico pueden incorporarse muchas componentes, en general en forma de parámetros que se añaden en la llamada recursiva:
 - Si lo único que buscábamos era una solución al problema, el algoritmo debe terminar en el punto (3a). Para ello, puede usarse un parámetro booleano de salida que se marque a TRUE en este punto, y que se use con una condicional en (3) para ver si hay que seguir añadiendo decisiones o no.
 - Si lo que se quería eran todas las soluciones, el procedimiento recursivo necesitará otro parámetro de entrada / salida que será la estructura donde se almacenen todas las soluciones encontradas, por ejemplo una lista que se actualiza en (3a).



3.– Backtracking

- Si se buscaba la mejor solución al problema (como en los problemas de optimización), el procedimiento necesitará otro parámetro de entrada / salida que guarde la mejor solución conocida hasta el momento. Al encontrar una nueva solución en (3a) se compara con la que se pasa por parámetro, actualizándolo si es necesario, y se sigue con la búsqueda.
- Si la creación de los hijos en el punto (2) puede ser costosa, y el conjunto de decisiones que se pueden tomar en cada etapa k es siempre el mismo, se añade un nuevo parámetro al método para llevar un control de las decisiones que se pueden tomar en cada momento, por ejemplo mediante un conjunto o vector. Cuando se toma una decisión en el apartado (3), se elimina del conjunto para que los hijos no puedan tomarla de nuevo.

3.- Backtracking

COSTES Y EFICIENCIA EN BACKTRACKING

- ▶ La forma de generar y escoger entre las distintas posibilidades determina la forma del árbol, la cantidad de descendientes de un nodo, la profundidad del árbol, etc., y determina la cantidad de nodos del árbol y por tanto posiblemente la eficiencia del algoritmo, pues el tiempo de ejecución depende en gran medida del número de nodos que se generen y se visiten. Habitualmente, tendremos como máximo tantos niveles como valores tenga una secuencia solución.
- ▶ Suponiendo que hay n etapas y m posibles decisiones en cada una de ellas (normalmente m es mayor que n) el orden de eficiencia de Backtracking está en $O(m^n)$; aun cuando se pueda tener $m = n$ se consigue el orden potencial $O(n^n)$.

3.– Backtracking

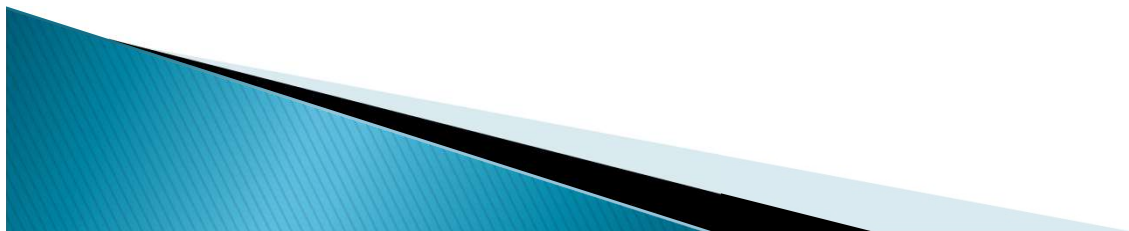
- ▶ Si se reduce a un solo nodo generado, el método Backtracking se convierte en una solución voraz: hay un total en $O(n)$ nodos a generar en total.
- ▶ Si cada nodo tiene todas las decisiones posibles, pero de forma que no se puedan repetir en distintos niveles, el coste se acerca a $O(n!)$.
- ▶ Si es posible, intentar reordenar las selecciones de forma que la cantidad de decisiones C_i que pueden tomarse en el nivel i sea creciente, es decir, $C_1 < C_2 < \dots < C_n$. La idea es que en los primeros casos haya menos posibilidades a evaluar para que el árbol no se expanda demasiado pronto.

3.- Backtracking

PROBLEMAS

Al tener una gran carga de recursión y debido al alto paso de parámetros entre las llamadas, hay que tener en cuenta una serie de consideraciones:

- ▶ El paso de parámetros únicamente como datos de entrada consume memoria y tiempo, ya que se realizan copias locales para cada uno de los entornos. Puede evitarse con el uso de parámetros por referencia, pero hay que tener cuidado con las modificaciones ya que afectarán en todos los niveles.



4.- El problema de las ocho reinas.

En un tablero de ajedrez (tamaño 8x8) hay que colocar ocho reinas sin que se amenacen unas a otras. En el juego de ajedrez, la reina amenaza a aquellas fichas que se encuentran en su misma fila, columna, o diagonal.

La forma más evidente de resolver el problema consiste en probar sistemáticamente todas las maneras de colocar ocho reinas en un tablero usando las coordenadas (x, y) ; para cada reina se implementaría con un doble bucle Desde, y solo en crear esos entornos (sin comprobar las soluciones) tendríamos un total de 64^8 posibilidades.

Más de 281 billones (millones de millones) de casos.

4.– El problema de las ocho reinas.

Siguiendo sin siquiera comprobar si hay amenazas entre reinas, entre todos esos casos se permitiría colocar distintas reinas en la misma casilla, lo que obviamente no es correcto.

La primera mejora sería evitar repeticiones y cuidar el orden de colocación desde la casilla superior izquierda hasta la inferior derecha; así el numero de situaciones a examinar sería:

$$\binom{64}{8} = 4.426.165.368$$

pero siguen siendo muchas y aún no se comprueban los requisitos.

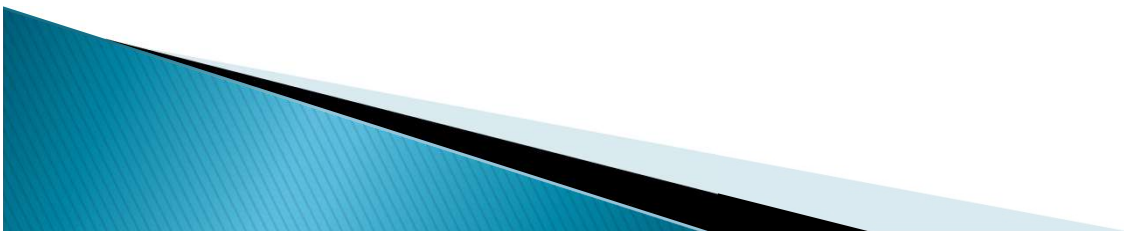
4.– El problema de las ocho reinas.

Una primera mejora de este último método explicado consistiría en no poner nunca más de una reina en una fila.

Esto reduce la representación del tablero a un vector de ocho elementos, cada uno de los cuales da la posición de la reina dentro de la fila correspondiente (la reina K estará en la fila K y columna $v[K]$).

Con esta situación se tendría 8 posibilidades para cada reina, es decir $8^8 = 16.777.216$.

No solo hemos reducido casos, es que además hemos conseguido avanzar hacia la solución al evitar una situación de amenaza.



4.– El problema de las ocho reinas.

Ahora evitaremos que dos reinas se encuentren en la misma columna $v[K]$. Una vez que nos damos cuenta que con la representación del tablero mediante un vector nos impide intentar poner dos reinas en la misma fila, es natural darnos cuenta que también lo podemos hacer con las columnas, por tanto ahora representaremos el tablero mediante un vector formado por ocho números diferentes entre 1 y 8, mediante una permutación de los ocho primeros números enteros:

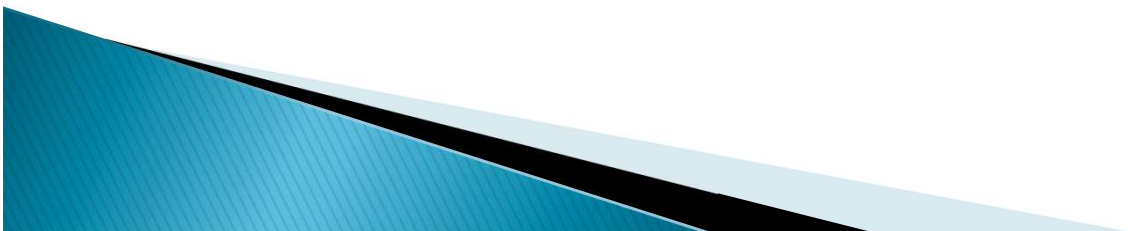
- para la primera hay 8 posiciones (columnas) posibles,
- para la segunda ya solo hay 7 posiciones, etc.

La cantidad de casos es de $8! = 40.320$ posibilidades.



4.– El problema de las ocho reinas.

- ▶ Es en este momento cuando podemos empezar a pensar en Backtracking.
- ▶ En vez de crear esas 40.320 posibilidades y testear, vamos poniendo una reina cada vez: si se producen amenazas no seguimos, y si no hay amenazas continuamos con la recursión.
- ▶ Así, evitamos generar los casos como los que empiezan por (1, 2, x, x, x, x, x, x) porque la reina 1 y la reina 2 comparten la misma diagonal principal y se amenazan entre sí; no tiene sentido seguir.



4.– El problema de las ocho reinas.

Usaremos las siguientes estructuras:

- ▶ Un conjunto *col* que representa las columnas ocupadas por las reinas usadas hasta el momento, con valores entre 1 y 8. Tiene los mismos valores que la solución del problema.
- ▶ Un conjunto *diag45* que representa las diagonales de 45° del tablero. Una diagonal de 45° tiene la propiedad de que la suma de sus coordenadas es constante, por ejemplo las casillas (2, 5) y (4, 3) están en la misma diagonal 45° ya que las dos coordenadas suman 7. El conjunto *diag45* toma los valores entre 2 y 16.
- ▶ Un conjunto *diag135* que representa las diagonales de 135° del tablero. Estas diagonales tienen la propiedad de que la diferencia de la primera coordenada menos la segunda es constante; las casillas (2, 4) y (5, 7) están en la misma diagonal 135° . Los valores del conjunto están entre -7 y 7.

4.– El problema de las ocho reinas.

```
proc reinas(E k: natural, E col, diag45, diag135: conjunto, E/S sol:
vector)
  si (k > 8)
    Escribir(sol)
  si no
    Desde j = 1 hasta 8 Hacer
      si (j  $\notin$  col) y (j+k  $\notin$  diag45) y (j-k  $\notin$  diag135)
        sol[k]= j
        reinas(k+1,col U {j},diag45 U {j+k},diag135 U {j-k},sol)
      Fsi
    fdesde
  Fsi
fproc
```

y la llamada inicial es:

reinas(1, \emptyset , \emptyset , \emptyset , sol_vací).

4.– El problema de las ocho reinas.

Por la forma de pasar las estructuras de entorno no es necesario deshacer acciones, y la copia local se realiza en la ejecución de la recursión.

El procedimiento anterior necesita 2.057 nodos, y basta con explorar 114 para encontrar una primera solución.



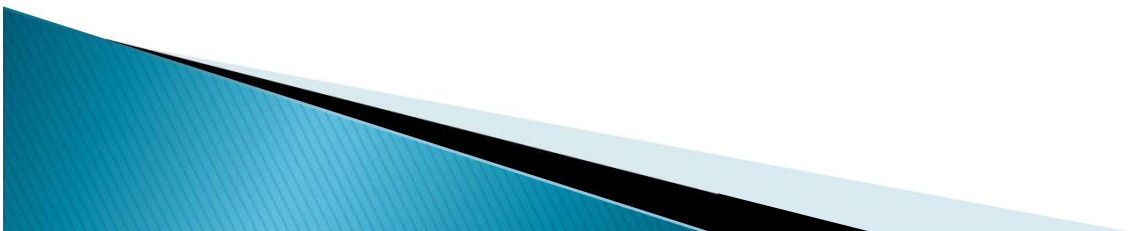
5.- El problema de la mochila.

Tenemos n tipos de objetos y un número adecuado de objetos de cada tipo.

Para $i=1,2,\dots,n$ un objeto de tipo i tiene un peso w_i y un valor positivo v_i . La mochila puede llevar un peso que no exceda de W .

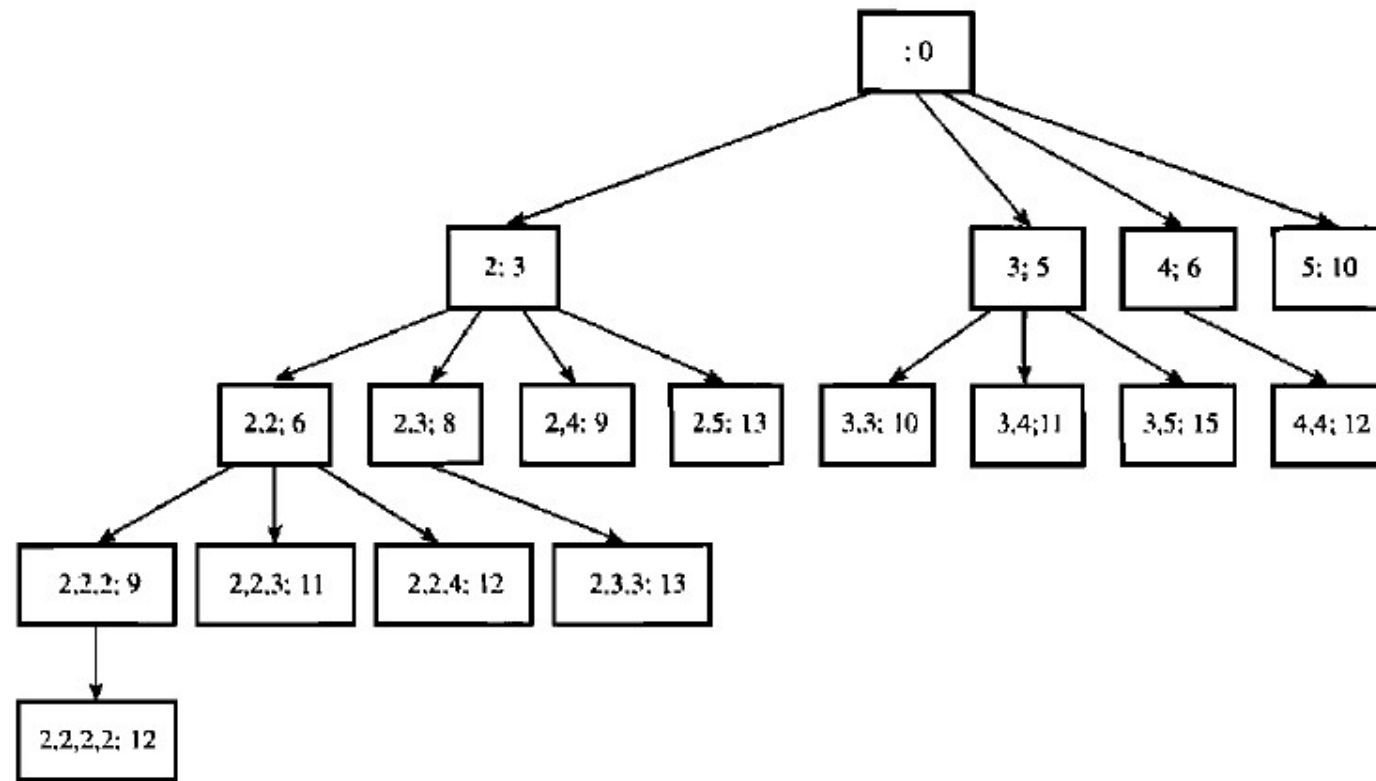
Nuestro objetivo es llenar la mochila maximizando el valor de los objetos incluidos, sin sobrepasar su capacidad.

Podemos tomar un objeto o prescindir de él, pero no podemos tomar una cierta parte de un objeto.



5.- El problema de la mochila.

Supongamos 4 tipos de objetos de pesos 2,3,4 y 5, cuyos valores son respectivamente 3,5,6 y 10. El valor de W es 8. Explorando el árbol implícito:

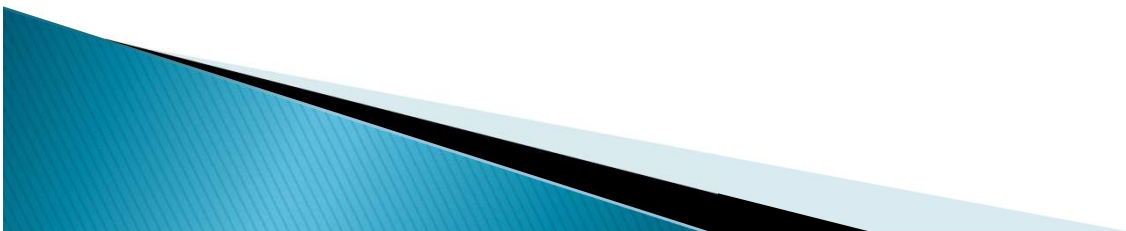


5.- El problema de la mochila.

El algoritmo de backtracking explora el árbol como un recorrido en profundidad, construyendo nodos y soluciones parciales a medida que avanza.

En el ejemplo, el primer nodo que se visita es el $(2;3)$, el siguiente es $(2,2;6)$, el tercero es $(2,2,2;9)$ y el cuarto $(2,2,2,2;12)$. A medida que se visita cada nodo nuevo, se extiende la solución parcial.

Después de visitar estos 4 nodos, se bloquea el recorrido: el nodo $(2,2,2,2;12)$ no tiene sucesores no visitados, puesto que añadir más elementos a esta solución parcial violaría la restricción de capacidad. Además, como podría ser la solución óptima de nuestro caso, la memorizamos.



5.– El problema de la mochila.

El algoritmo vuelve atrás ahora en busca de otras soluciones. En cada paso retrocedido por el árbol, se elimina el elemento correspondiente de la solución parcial.

Vuelve primero a $(2,2,2;9)$ que también carece de sucesores.

Vuelve a $(2,2;6)$, que tiene dos sucesores: $(2,2,3;11)$ y $(2,2,4;12)$ que no mejoran la solución memorizada antes.

Vuelve otro paso más, y así sucesivamente.

En $(2,3,3;13)$ encontramos una solución mejor que la que ya tenemos, y después $(3,5;15)$ es aún mejor, y acabará siendo la solución óptima del caso.

5.- El problema de la mochila.

El algoritmo es el siguiente, suponiendo que los datos n y W , y los vectores $w[1..n]$ y $v[1..n]$ son variables globales:

función *mochilava*(i, r)

{Calcula el valor de la mejor carga que se puede construir empleando elementos de los tipos i a n y cuyo peso total no sobrepase r }

$b \leftarrow 0$

{Se prueban por turno las clase de objetos admisibles }

para $k \leftarrow i$ **hasta** n **hacer**

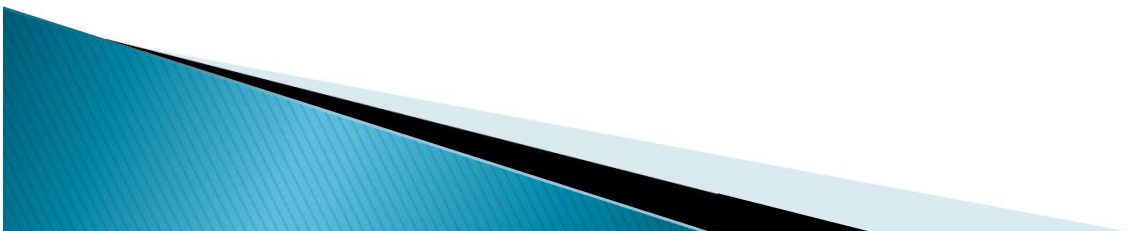
si $w[k] \leq r$ **entonces**

$b \leftarrow \text{máx}(b, v[k] + \text{mochilava}(k, r - w[k]))$

devolver b

6.– Ramificación y poda.

- ▶ Al igual que la vuelta atrás, la ramificación y poda es una técnica para explorar un grafo dirigido implícito, que normalmente es acíclico (o incluso un árbol).
- ▶ Aquí se trata de buscar la solución óptima de algún problema entre todas las soluciones posibles.
- ▶ En cada nodo, calculamos una cota del posible valor de aquellas soluciones que pudieran encontrarse más adelante en el grafo. Si esa cota muestra que cualquiera de estas soluciones es peor que la mejor hallada hasta el momento, no seguiremos explorando esa parte del grafo.



6.– Ramificación y poda.

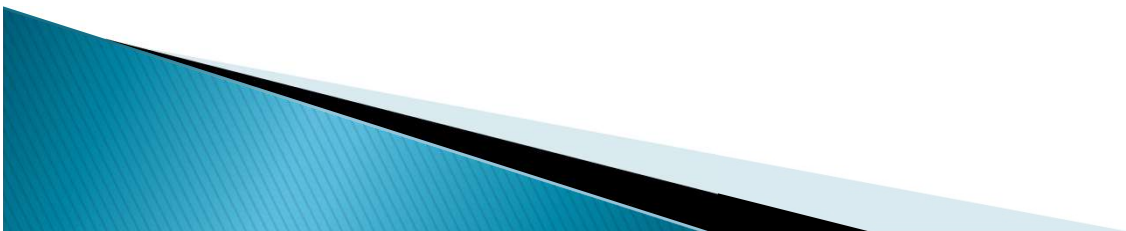
Diferencia fundamental con el backtracking:

En backtracking :

- ▶ Tan pronto como se genera un nuevo hijo del nodo en curso, este hijo pasa a ser el nodo en curso. Por lo tanto, los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso.

Ramificación y acotación:

- ▶ Se generan todos los hijos del nodo en curso antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso. Por lo tanto, puede haber más nodos vivos y se deben almacenar en una estructura de datos auxiliar: lista de nodos vivos.



6.– El problema de la asignación.

- ▶ El problema de la asignación consiste en asignar N tareas a N agentes, de manera que cada tarea se realice una única vez y cada agente tenga asociada una única tarea.
- ▶ Cada asignación tiene un coste $M[i, j]$, donde i es el agente y j la tarea, así que de las $N!$ posibles asignaciones válidas hay que determinar la que tenga un coste total lo menor posible.
- ▶ Supongamos que la matriz de costes es la siguiente:

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

6.– El problema de la asignación.

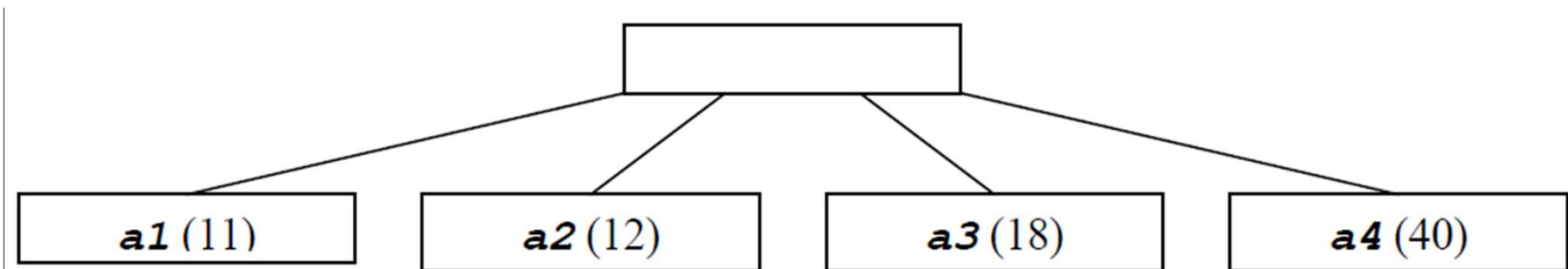
- ▶ Empezamos creando las cotas: vamos a obtener una cota real, basada en soluciones conocidas y que nos permita podar candidatos no deseables, y otra estimada que sirva para terminar rápidamente el problema si se encuentran candidatos con valores próximos a ella.
- ▶ Para este ejemplo, usaremos como cota real cualquier asignación válida que sea fácil de encontrar; por ejemplo la solución a_1, b_2, c_3, d_4 , que tiene coste $11 + 15 + 19 + 28 = 73$.
- ▶ Cualquier candidato parcial que tenga un coste superior a 73 será rechazado (podado) ya que en vez de seguir generándolo nos podríamos quedar con la solución completa ya encontrada.

6.– El problema de la asignación.

- ▶ Como cota estimada consideraremos, por ejemplo, el coste mínimo que se tendría por parte de los agentes.
- ▶ Los mínimos de las filas serían $11 (a1) + 13 (b3) + 11 (c1) + 14 (d2)$.
- ▶ Hay que observar que esto solo es una estimación de lo mejor que podría pasar, pero que no es una solución real (la tarea 1 se asignaría dos veces), y tendría un coste de 49.
- ▶ Pero también podríamos haber considerado el coste mínimo de realizar las tareas, es decir, los mínimos de las columnas $11 (a1) + 12 (a2) + 13 (b3) + 22 (b4) = 58$.

6.– El problema de la asignación.

- ▶ Considerando ambas estimaciones, la cota estimada será $\max\{49, 58\} = 58$. Así, el rango de valores válidos para explorar será $[58, 73]$.
- ▶ Para realizar la ramificación y poda recorreremos un árbol implícito en el que los nodos se corresponden con asignaciones parciales de agentes a tareas. En la raíz del árbol no hay asignación. En cada etapa (nivel del árbol) se asigna un agente más. Las posibilidades para la primera asignación serían:



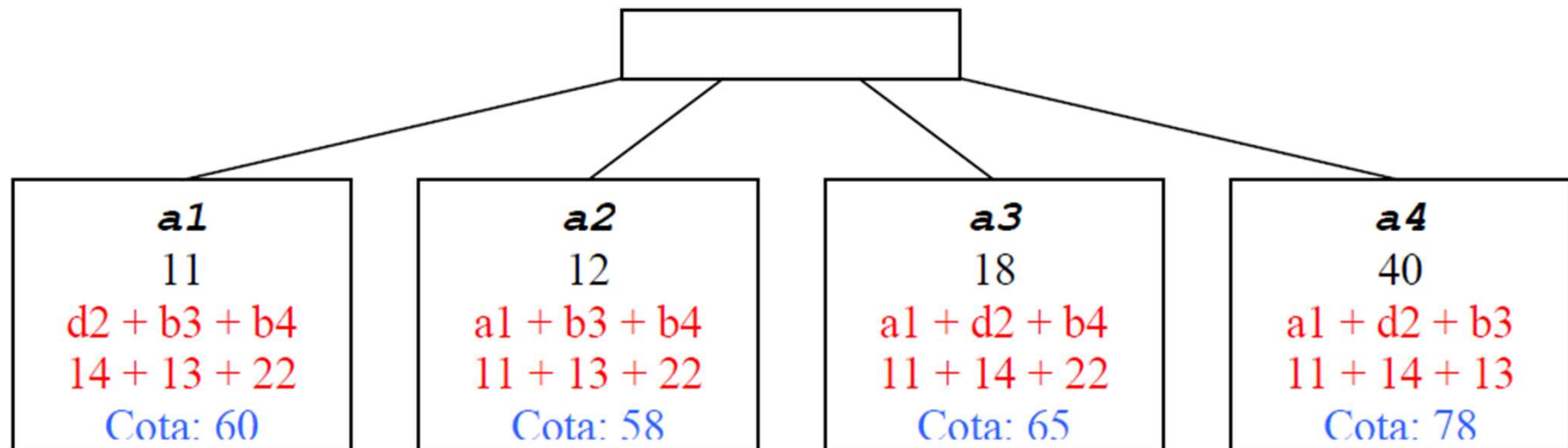
6.– El problema de la asignación.

- ▶ Para poder decidir por cuál de los candidatos sigue la exploración del árbol, en cada nodo se calcula una cota de la mejor solución que se podría conseguir a partir de ese nodo.
- ▶ Consideraremos el mínimo de los costes necesarios para realizar las tareas que aún faltan por asignar; por ejemplo, tras la asignación a1 estimamos el mejor coste de hacer la tarea 2 (que sería d2, coste 14), la tarea 3 (b3, coste 13) y la tarea 4 (b4, coste 22).
- ▶ Una vez más, esto es una estimación y no forma una solución completa posible.

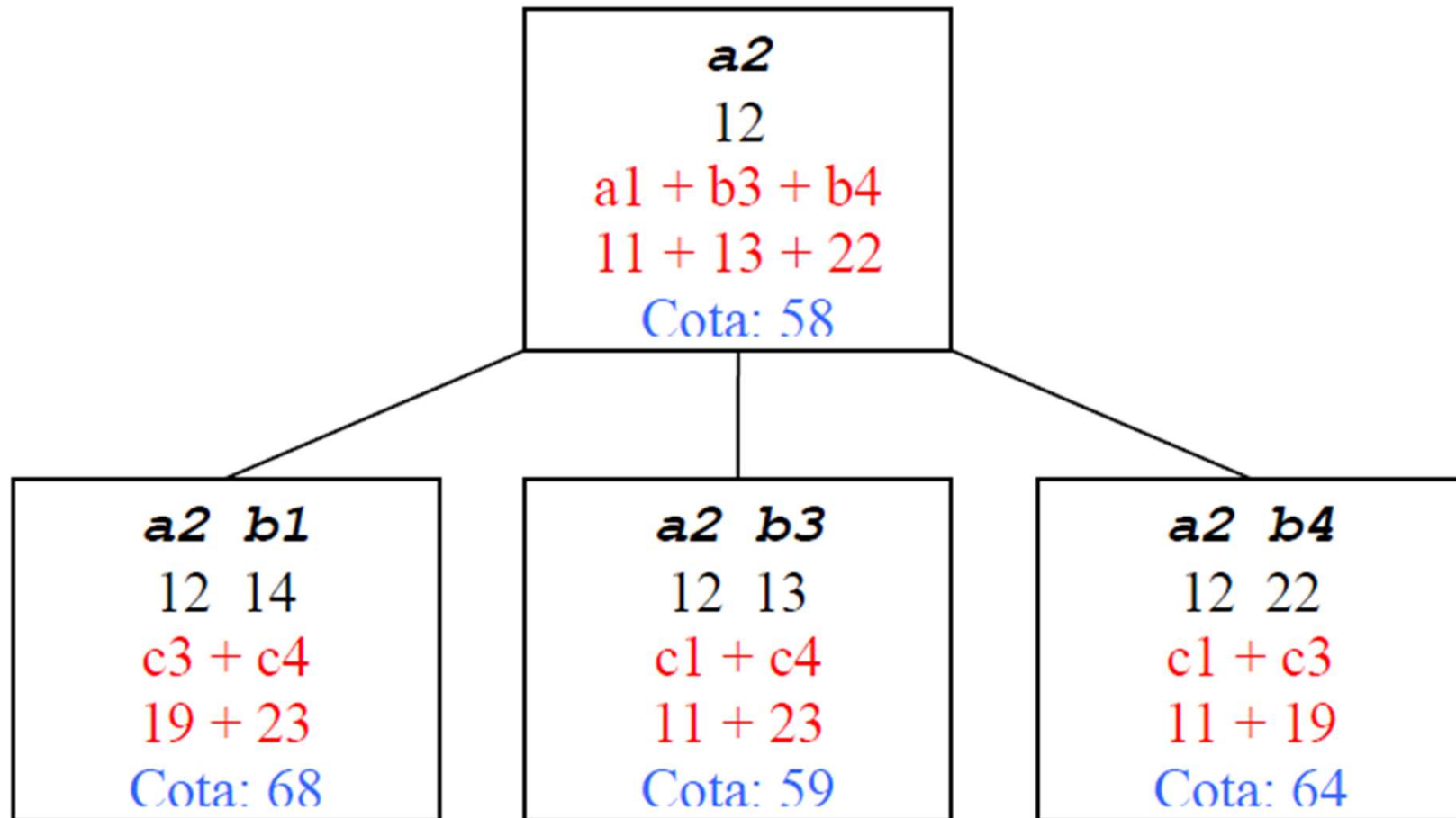


6.- El problema de la asignación.

- ▶ Añadimos esta información a los nodos (en negro, las decisiones tomadas; en rojo, las estimaciones; en azul, la cota obtenida):

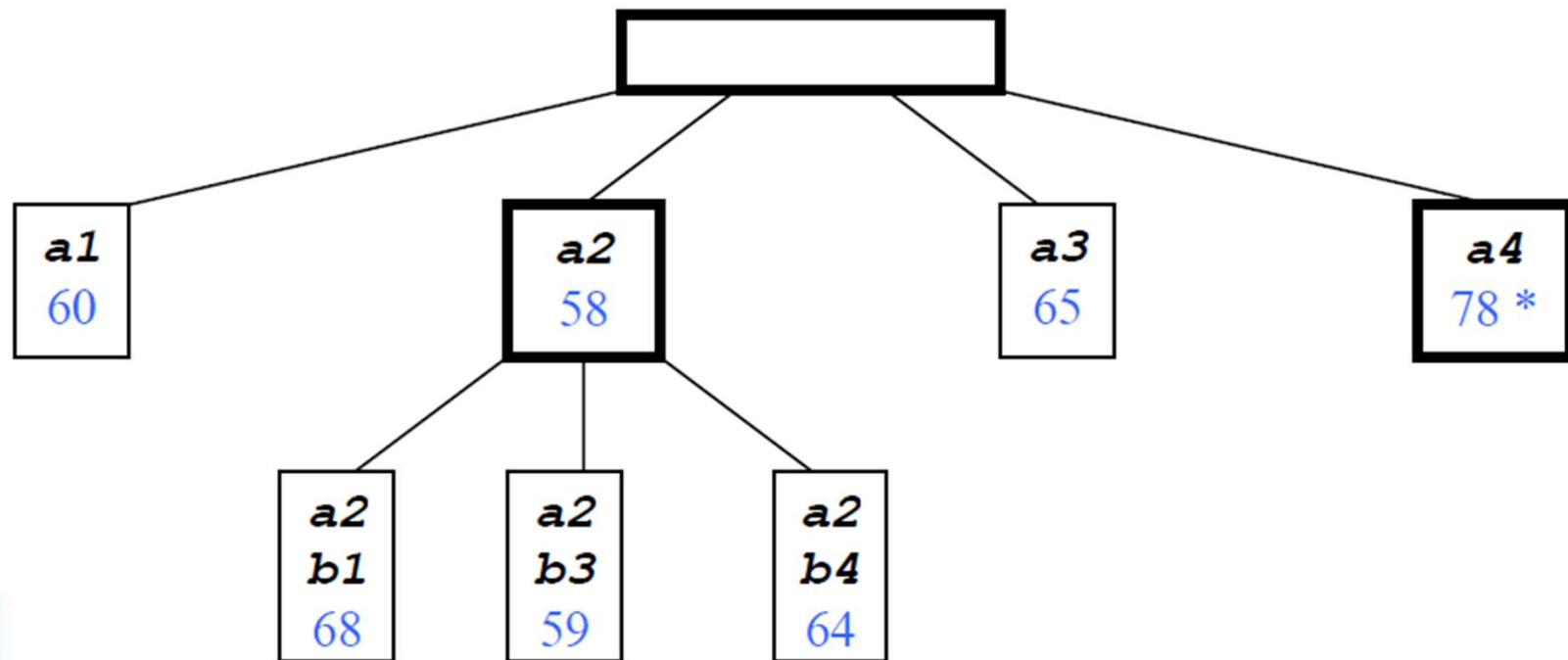


6.- El problema de la asignación.



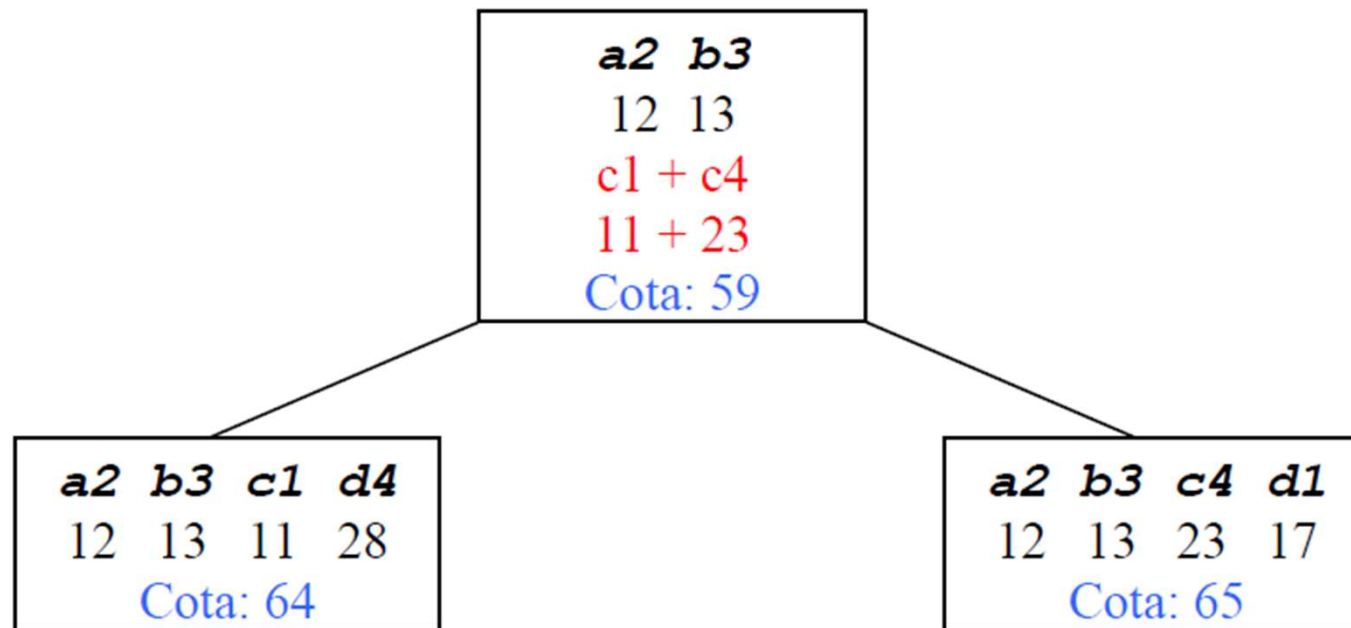
6.- El problema de la asignación.

- ▶ Ahora la estructura simplificada del árbol (recordar que solo se guardan los nodos que aún no han sido explorados) sería como sigue, con los nodos podados marcados con * y los que ya no están en la cola de prioridad con borde oscuro.



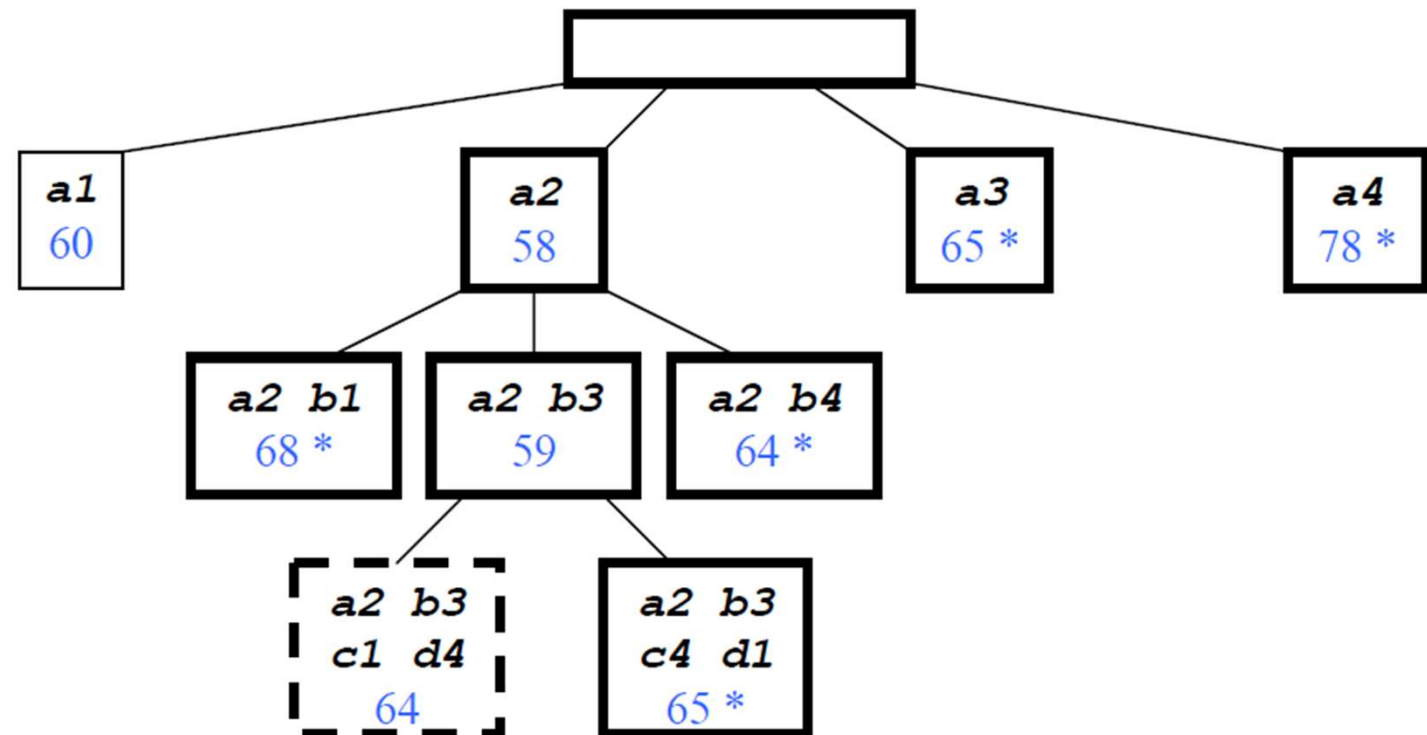
6.- El problema de la asignación.

- ▶ De los nodos vivos se coge el que tenga la cota más pequeña y se añade otro agente, y como solo quedaría una tarea por realizar la estimación es realmente una solución completa.



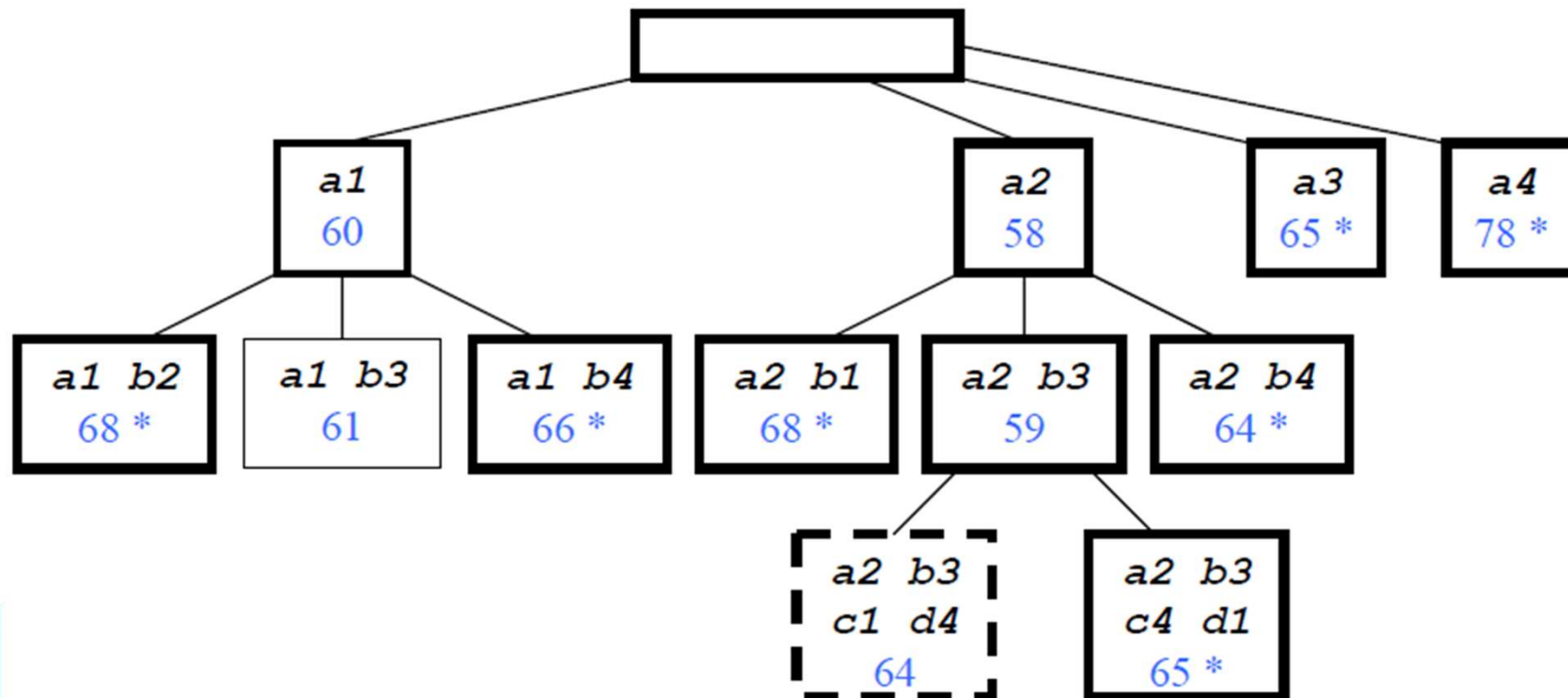
6.- El problema de la asignación.

Al haber encontrado una solución mejor que la conocida, la cota superior se actualiza a 64, por lo que el rango de valores admisibles es $[58, 64]$; esto hace que haya más nodos que se podan automáticamente, y el árbol quedaría (punteada la solución óptima hasta ahora):



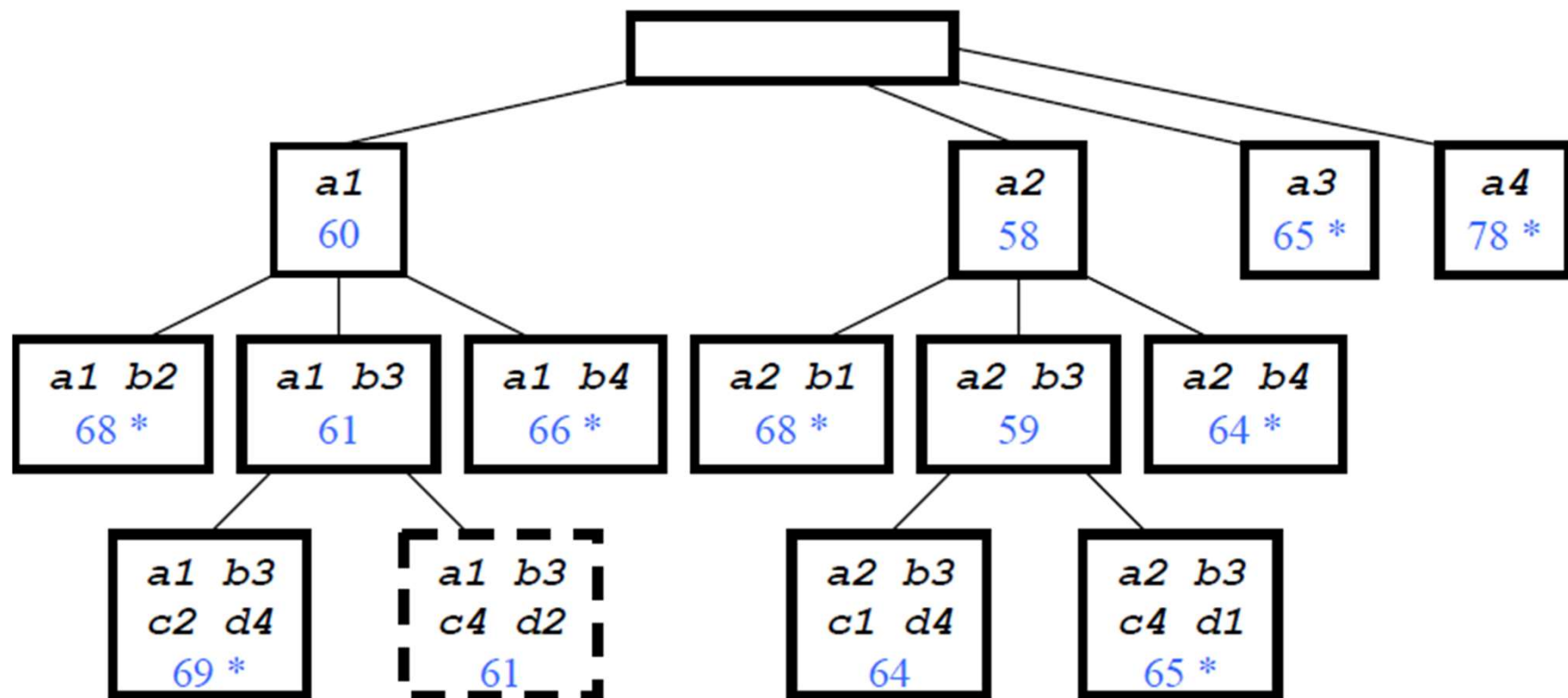
6.- El problema de la asignación.

- ▶ Puede verse que al haber encontrado esta solución a2 b3 c1 d4 el árbol se ha visto muy reducido a la hora de explorar nodos vivos, quedando solo uno.
- ▶ Acabamos el ejemplo con los dos últimos árboles implícitos desarrollados tras a1...



6.- El problema de la asignación.

- ▶ y tras desarrollar a1 b3 se encuentra la que finalmente será la solución óptima:



6.– El problema de la asignación.

- ▶ Si se hubiese hecho una búsqueda completa se habría necesitado recorrer una cantidad de nodos igual a $4 + 4*3 + 4*3*2 = 40$ nodos, mientras que con la ramificación y poda solo ha hecho falta crear 14 nodos.

