Análisis y Diseño de Algoritmos

Tema 7: Ramificación y Poda

Titulación: Grado en Ingeniería Informática

E.T.S. INGENIERÍA INFORMÁTICA

Contenido

- Introducción
- Técnicas de Ramificación y Poda
  – Conceptos generales
  – El problema de la asignación de tareas
  – El problema del viajante de comercio
  – El problema de la mochila: el método relajado
- Referencias

Introducción

- Características del Backtracking
  - Explora el espacio de estados en busca de una solución que satisfaga ciertas propiedades.
  - El espacio de estados es un árbol que la técnica recorre utilizando la estrategia “depth-first”.
  - Cuando se alcanza un estado (una solución parcial) que se sabe que no conduce a una solución del problema se poda la rama (no se explora) que cuelga de ese estado.
  - Si se desea encontrar una solución al problema, en cuanto se llega a ella la búsqueda acaba.
  - Si es un problema de optimización, entonces es necesario recorrer todo el espacio de estados para decidir cual es la mejor solución de entre todas las posibles, aunque esto no siempre es posible si el espacio de estados es muy grande.

Introducción

- La técnica de ramificación y poda (branch and bound) se aplican a problemas de optimización.
- Como el backtracking, explora el espacio de estados, pero intentan hacer más eficiente este recorrido utilizando funciones que permiten saber "lo bueno" que es un estado:
  - Cada estado se ramifica, considerando todas las posibles continuaciones del mismo (como en backtracking)
  - Se evalúan todos los nuevos estados, y se descartan aquellos que no llevan a una solución del problema (como en backtracking)
  - De entre todos los nuevos estados, para continuar la búsqueda, se escoge el estado que en ese momento parece más prometedor *(es más probable que lleve a una solución óptima)*
    - Esta selección significa que el árbol de estados no tiene que recorrerse utilizando la técnica depth-first

Introducción

- Para decidir cuál de entre todos los estados es el mejor (a priori) se utiliza una función de estimación *f*.
  - Dada una solución parcial *p* (un estado del árbol), *f(p)* es una estimación optimista de la calidad que tendrá una solución al problema obtenida a partir de *p*.
    - *f(p)* debe ser una cota inferior de la calidad de las soluciones que se derivan de *p*, si la mejor solución es un *mínimo.*
    - *f(p)* debe ser una cota superior de la calidad de las soluciones que se derivan de *p*, si la mejor solución es un *máximo.*

Introducción

- Evidentemente, *f* depende del problema que se está resolviendo.
- Sea *sol* la mejor solución obtenida hasta ahora, su calidad es *f(sol)= C* entonces, si la mejor solución es un mínimo:
  - Si $f(p) < C$, *p* es una solución parcial prometedora
  - Si $f(p) \geq C$, *p* no va a permitir mejorar la solución conocida, y puede podarse su rama. Esto incluye el caso en que *p* no lleve a una solución del problema.
- Si *sol* es una solución mejor* que *sol* (la solución óptima hasta ahora encontrada), entonces se actualiza la solución a *sol* y se actualiza su calidad* $f(sol^*) = C^* < C$.

Estrategia best-first

- La selección del siguiente estado a explorar no tiene por qué seguir la estrategia depth-first. Lo más natural es seleccionar el estado más prometedor (no el que está más a la izquierda del árbol) utilizando la técnica denominada best-first.
- Para implementar este comportamiento no puede utilizarse la recursión (que internamente hace uso de una pila) sino que hay que usar una estructura de datos intermedia para almacenar temporalmente los estados no explorados, ordenados según la estimación de su calidad. Una cola de prioridades o un montículo son estructuras apropiadas.
- Cada vez que se modifica la solución y la calidad: *C, C*,* etc. hay que actualizar el montículo, podando las ramas que ya no son prometedoras.

Estrategia best-first

- La estrategia best-first se basa en la siguiente idea:
  - Si C^* es la calidad de la mejor solución encontrada,
    - En caso de que la mejor solución sea la mínima, cualquier subproblema p tal que $f(p)<C^*$ debe explorarse.
    - En caso de que la mejor solución sea la máxima, cualquier subproblema p tal que $f(p)>C^*$ debe explorarse.
  - Cuando se sigue la estrategia best-first
    - Si se busca la solución mínima, no se explora ningún subproblema p con $f(p) > C^*$.
    - Si se busca la solución máxima, no se explora ningún subproblema p con $f(p) < C^*$.

Ramificación y Poda: esquema general

```
TSolución BranchBound(TSubproblema p){ //SE BUSCA UN MÍNIMO
   Heap h = new Heap();
   h.insertar(p); // se introduce el subproblema inicial p en h
   float c = ∞;
   Tsolucion s = null;
   while (!h.vacio()){
      Tsubproblema act = h.extraer();
      if (esSolucion(act)){
         if (cota(act)<c){           //SE BUSCA UN MÍNIMO
            c = cota(act);
            s = act;
            h.actualizar(c);
         }
      } else {
        List<TSubproblema> hijos = act.ramificar();
        for (TSubproblema hijo:hijos){
           if (cota(hijo)<c) h.insertar(hijo);
        }
      }
   }
   return s;
}
```

La función de cota f

- Normalmente la función de cota para un subproblema *p*, *f(p)* , se calcula como la suma del coste hasta *p* (*g(p)*) , más una estimación del coste de las soluciones que pueden obtenerse como continuación de *p* (*h(p)*).

$$f(p) = g(p) + h(p)$$

  – Para problemas de minimización
    – $f(p) = g(p) + \min\{coste(cont):cont \in continuacion(p)\}$
  – Para problemas de maximización
    – $f(p) = g(p) + \max\{coste(cont):cont \in continuacion(p)\}$

Calcular la función de cota f

$$f(p) = g(p) + h(p)$$

- *g(p)* es fácil: es el coste, gasto o esfuerzo ya realizado.
- *h(p)* es difícil: es un heurístico, una estimación del esfuerzo futuro. Una estimación optimista
- Para problemas de minimización
  - *h(p) = min{coste(cont):cont ∈continuacion(p)}*
- Para problemas de maximización
  - *h(p) = max{coste(cont):cont ∈continuacion(p)}*
    - *Usualmente en maximización el coste es un "beneficio"; g(p) es el beneficio ya obtenido y h(p) es una estimación a máximos del beneficio que espero obtener.*

El problema de la asignación de tareas

- Suponemos que tenemos *n* tareas, $t_1,\ldots,t_n$, que realizar, y debemos asignárselas a *n* personas $p_1,\ldots,p_n$ de manera que el coste total de la realización de las tareas sea mínimo.
- Tenemos una tabla $C = (c_{ij})$ (un array nxn) con los costes de las tareas, de manera que c_{ij} es lo que cobra la persona p_i por realizar la tarea t_j
- El problema se resuelve seleccionando *un elemento en cada fila* de la matriz, de manera que *no haya dos elementos en la misma columna* y que la suma total sea mínima.

El problema de la asignación de tareas

- Suponemos que tenemos *n* tareas, $t_1,\ldots,t_n$, que realizar, y debemos asignárselas a *n* personas $p_1,\ldots,p_n$ de manera que el coste total de la realización de las tareas sea mínimo.
- El primer análisis informal es ver el problema desde el punto de vista del que paga, que quiere gastar lo mínimo. Y dice: "tengo que pagar a n personas; miro lo mínimo que puede cobrar cada persona y hago el cálculo de que esa suma es lo que pagaré (lo más optimista)".
- Alguien sensato le dice: "Pero si hay un trabajo fácil, y dos personas cobran poco, y se lo asignas a las dos, eso es imposible y dejarás alguna tarea sin hacer"
- Respuesta del pagador "no importa, esto es un heurístico que iré actualizando según vaya asignando tareas"

El problema de la asignación de tareas

- Ejemplo de matriz *C*

	Tarea 1	Tarea 2	Tarea 3	Tarea 4
Persona 1	9	2	7	8
Persona 2	6	4	3	7
Persona 3	5	8	1	8
Persona 4	7	6	9	4

- una cota inferior de la solución óptima se calcula con la suma del mínimo que cobra cada persona
  - 2+3+1+4 = 10
- aunque está claro que este valor no corresponde a ninguna solución del problema porque significaría asignar la Tarea 3 a dos personas distintas

El problema de la asignación de tareas

- Si suponemos que las soluciones parciales son listas $[a_1,\ldots,a_i]$, donde a_i es la tarea asignada a la persona p_i, podemos aplicar el mismo criterio para calcular una estimación del coste de las soluciones parciales

	Tarea 1	Tarea 2	Tarea 3	Tarea 4
Persona 1	9	2	7	8
Persona 2	6	4	3	7
Persona 3	5	8	1	8
Persona 4	7	6	9	4

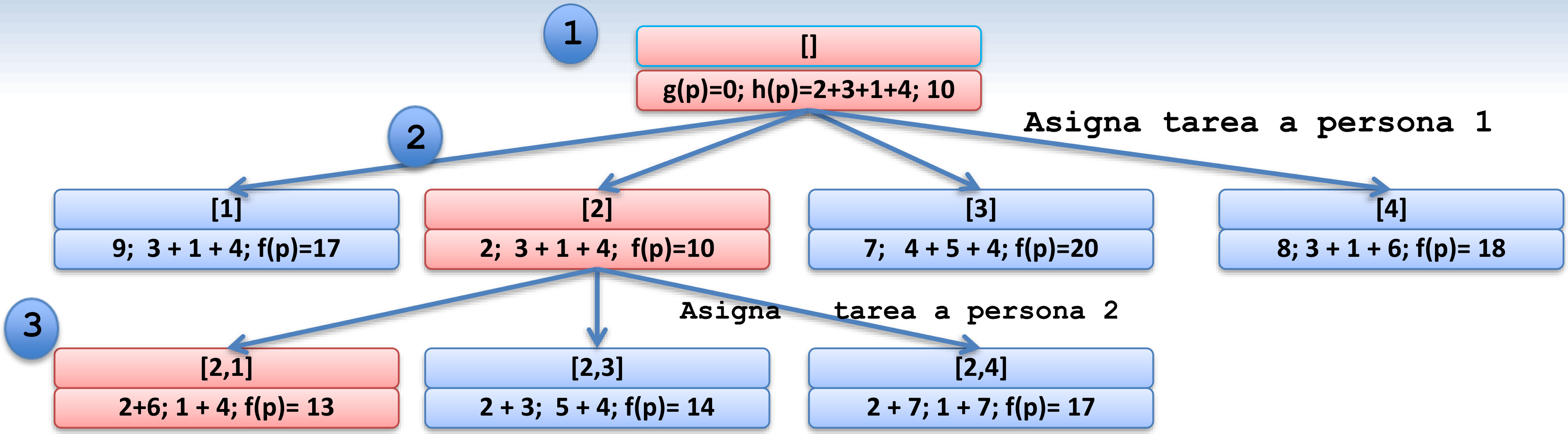
- Por ejemplo, si una solución parcial asigna la tarea 1 a la primera persona *[1]*, una cota interior del coste de cualquier solución que complete esta asignación sería $f([1]) = \underbrace{9}_{\text{Coste de [1]}}+\underbrace{3+1+4}_{\text{Cota inferior de cualquier continuación de [1]}} = 17$

El problema de la asignación de tareas

	Tarea 1	Tarea 2	Tarea 3	Tarea 4
Persona 1	9	2	7	8
Persona 2	6	4	3	7
Persona 3	5	8	1	8
Persona 4	7	6	9	4

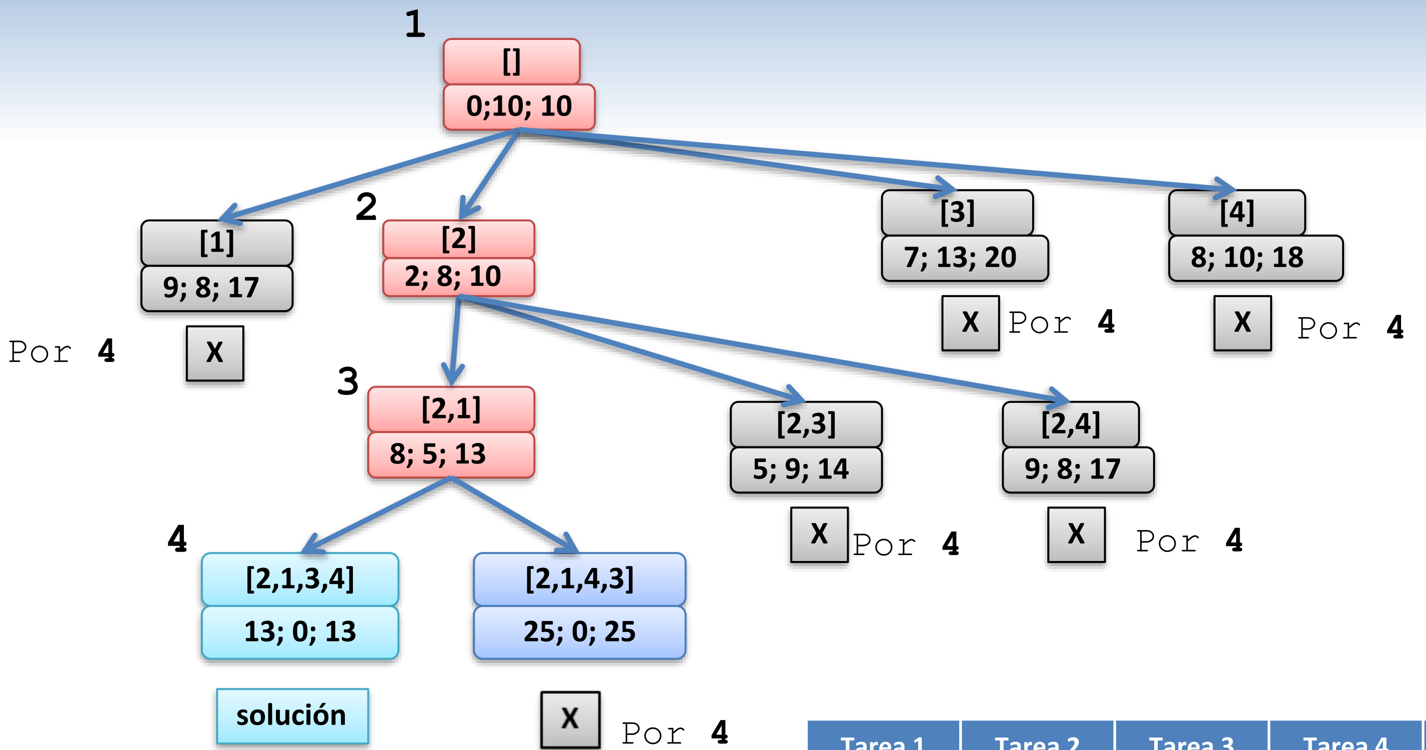
- En el árbol que vamos a desarrollar aparecerá en cada nodo la tarea asignada a cada persona por niveles.
- En el nivel 0, raíz, no hay tarea asignada, g(p)=0 (aún no hemos pagado nada), h(p)= 2+3+1+4 = 10, f(p) =g(p)+h(p) = 10. Ponemos las tres cantidades en ese orden separadas por "punto y coma".
- En el nivel 1 se asigna, a la persona 1, una tarea que puede ser 1, 2, 3, 4. Y en cada caso cambia el valor de g(p), cambia el valor de h(p) y cambia la suma f(p) = g(p) + h(p).
- Por ejemplo, el primer nodo asigna la tarea 1 a la primera persona *[1]*, una cota interior del coste de cualquier solución que complete esta asignación sería *g([1]) = 9; h([1]) = 3+1+4 = 8; f([1]) = g([1]) + h([1]) = 9 + 8 = 17*

El problema de la asignación de tareas



	Tarea 1	Tarea 2	Tarea 3	Tarea 4
Persona 1	9	2	7	8
Persona 2	6	4	3	7
Persona 3	5	8	1	8
Persona 4	7	6	9	4

El problema de la asignación de tareas



Tarea 1	Tarea 2	Tarea 3	Tarea 4	
9	2	7	8	Persona 1
6	4	3	7	Persona 2
5	8	1	8	Persona 3
7	6	9	4	Persona 4

El problema de la asignación de tareas

- Expresión formal de las funciones de gasto ya realizado, *g(p)* y heurístico a futuro, *h(p)*.

Sea $p_i = [a_1, \cdots, a_i]$ entonces $f(p_i) = g(p_i) + h(p_i)$, donde

1. $g(p_i) = \sum_{j=1}^{i} C_{j,a_j}$ (coste de las asignaciones ya realizadas)
2. $h(p_i) = \sum_{i=j+1}^{n} min\{C_{j,k} | 1 \leq k \leq n, \forall 1 \leq r \leq i.k \neq a_r\}$

El problema de la asignación de tareas

```
public  static List<Integer> asignacion(int[][] costes){
    List<List<Integer>> heap = new ArrayList<List<Integer>>();
    List<Integer> act = new ArrayList<Integer>();
    int mejor = Integer.MAX_VALUE;
    List<Integer> mejorSolucion = null;
    heap.add(act);
    while (!heap.isEmpty()){
        act = menorCoste(heap,costes);
        heap.remove(act);
        if (act.size() == costes.length){
            int m = coste(act,costes);
            if (m<mejor){
                mejor = m;
                mejorSolucion = act;
                actualizar(heap,m,costes,0);
            }
        } else
        // continúa  …
    }
}
```

El montículo se implementa como una lista, para simplificar la implementación

El problema de la asignación de tareas

```
public  static List<Integer> asignacion(int[][] costes){
        ……
        //continúa
        } else {
             for (int i = 0; i<costes.length;i++){
                  if (!act.contains(i)){
                       List<Integer> l = new ArrayList<Integer>(act);
                       l.add(i);
                       if (coste(l,costes)<mejor)    heap.add(l);
                  }
             }
        }
    }  // del while
   return mejorSolucion;
}
```

El problema del viajante de comercio

- Enunciado del problema
  - Dadas *n* ciudades conectadas entre ellas, encontrar *el camino más corto* que pasa por todas las ciudades, sin que se visite ninguna más de una vez.
  - Una instancia del problema queda definida por una matriz $D_{nxn} = \{d_{ij}\}$ con las distancias entre las ciudades.

El problema del viajante de comercio

– Supongamos que llamamos *tour* a un camino que pasa por todas las ciudades.
– Un tour puede representarse mediante una lista $[c_1,\ldots,c_n]$ con el orden en el que se visitan las ciudades
– La longitud del tour puede calcularse a partir de la matriz de distancias D, como

$$g([c_1,\ldots,c_n]) = d_{12} + d_{23}+\ldots+d_{n-1n}+d_{n1}$$

El problema del viajante de comercio

- Para aplicar la técnica de Ramificación y Poda y encontrar el menor tour, hay que buscar una cota inferior. Entonces, debemos pensar en un heurístico. Ese heurístico será el valor de f en el nodo raíz (*g([]) =0; h([])= un mínimo optimista; f([]) = g([]) + h([]) = 0+ h([]) = h([])*).
- Sabemos que cualquier circuito pasa por cada ciudad con dos arcos, uno de entrada (desde la ciudad anterior) y uno de salida (hacia la ciudad siguiente).
- Un heurístico optimista es pensar que, para cada nodo, voy a usar los dos arcos con menor peso.
- Si para cada nodo cuento dos arcos, al sumar esas cantidades estoy contando el doble de lo necesario.
- Entonces esa suma debo dividirla por 2.

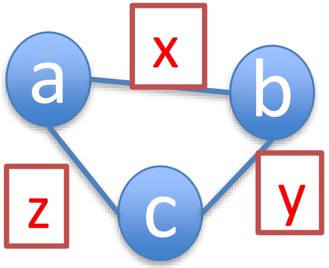
El problema del viajante de comercio

- Dicho ahora de otra forma, mirando ahora a partir de un tour completo ya construido.
- Para un tour ya hecho (solo hay g($[c_1,\ldots,c_n]$), pues h($[c_1,\ldots,c_n]$) =0), tenemos que $g([c_1,\ldots,c_n])$ es igual que a la mitad de la suma de los caminos adyacentes a cada nodo en el tour

$$g([c_1,\ldots,c_n]) =$$
$$= d_{12} + d_{23}+\ldots+d_{n-1n}+d_{n1} =$$
$$= \tfrac{1}{2}((d_{n1} + d_{12}) + (d_{12} + d_{23})+\ldots+(d_{n-1n}+d_{n1}))$$

Por ejemplo,

$$g([a,b,c]) = x + y + z =$$
$$= \tfrac{1}{2}((z+x)+(x+y)+(y+z))$$



El problema del viajante de comercio

– Entonces un heurístico optimista se puede calcular como la suma de los dos caminos más cortos que salen de cada ciudad, dividida por 2.

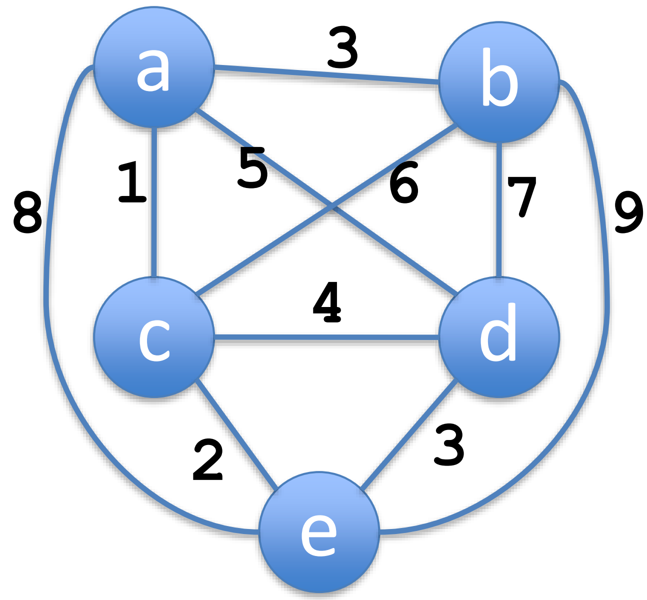
$g([c_1,\dots,c_n]) =$

$= \frac{1}{2}((d_{n1} + d_{12}) + (d_{12} + d_{23}) + \dots + (d_{n-1n} + d_{n1})) \geq$

$\geq \frac{1}{2}($*suma dos aristas más cortas que salen de* c_1 $+$ *suma dos aristas más cortas que salen de* c_2 $+ \dots +$ *suma dos aristas más cortas que salen de* c_n $)$

El problema del viajante de comercio

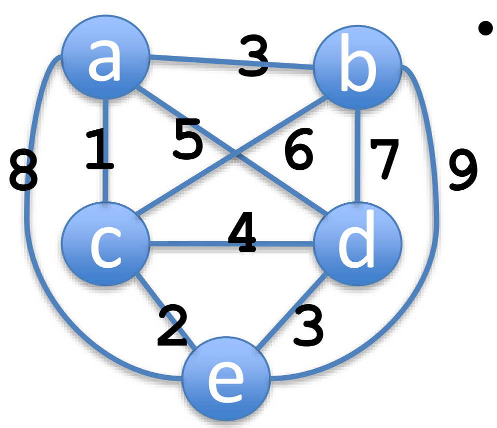
- Por ejemplo,



- *$h([]) = \lceil((1+3) + (3 + 6) + (1+2) + (3+4) + (2+3))/2\rceil = 14$*
- Se toma el valor por exceso porque estamos considerando números enteros; entonces un recorrido real no puede tener decimales y no se puede quedar por debajo de la cota optimista (que ocurriría si tomamos el valor por defecto).

El problema del viajante de comercio

- Partiendo de una ciudad cualquiera c_1, una solución parcial del problema viene dada por una lista $[c_1,\ldots,c_i]$ con el orden en el que se visitan las ciudades desde la ciudad c_1 hasta la ciudad c_i.
- La cota inferior de una solución parcial que incluye una arista *(a,d)* entre las ciudades *a* y *d* se calcula incluyendo en la expresión de *f()* las distancias d_{ad} y d_{da}: (esa es la parte *g()*, y el resto es la parte *h()*)

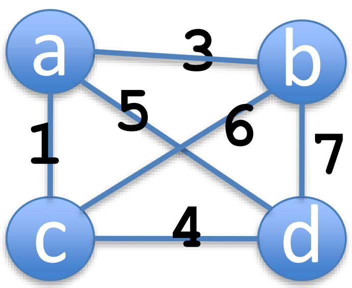


- Por ejemplo, cualquier tour que incluye el camino *(a,d)* tiene como cota inferior de su longitud a la expresión:

$f() = \lceil (1+5) + (3 + 6) + (1+2) + (3+5) + (2+3)/2 \rceil = =16$

El problema del viajante de comercio

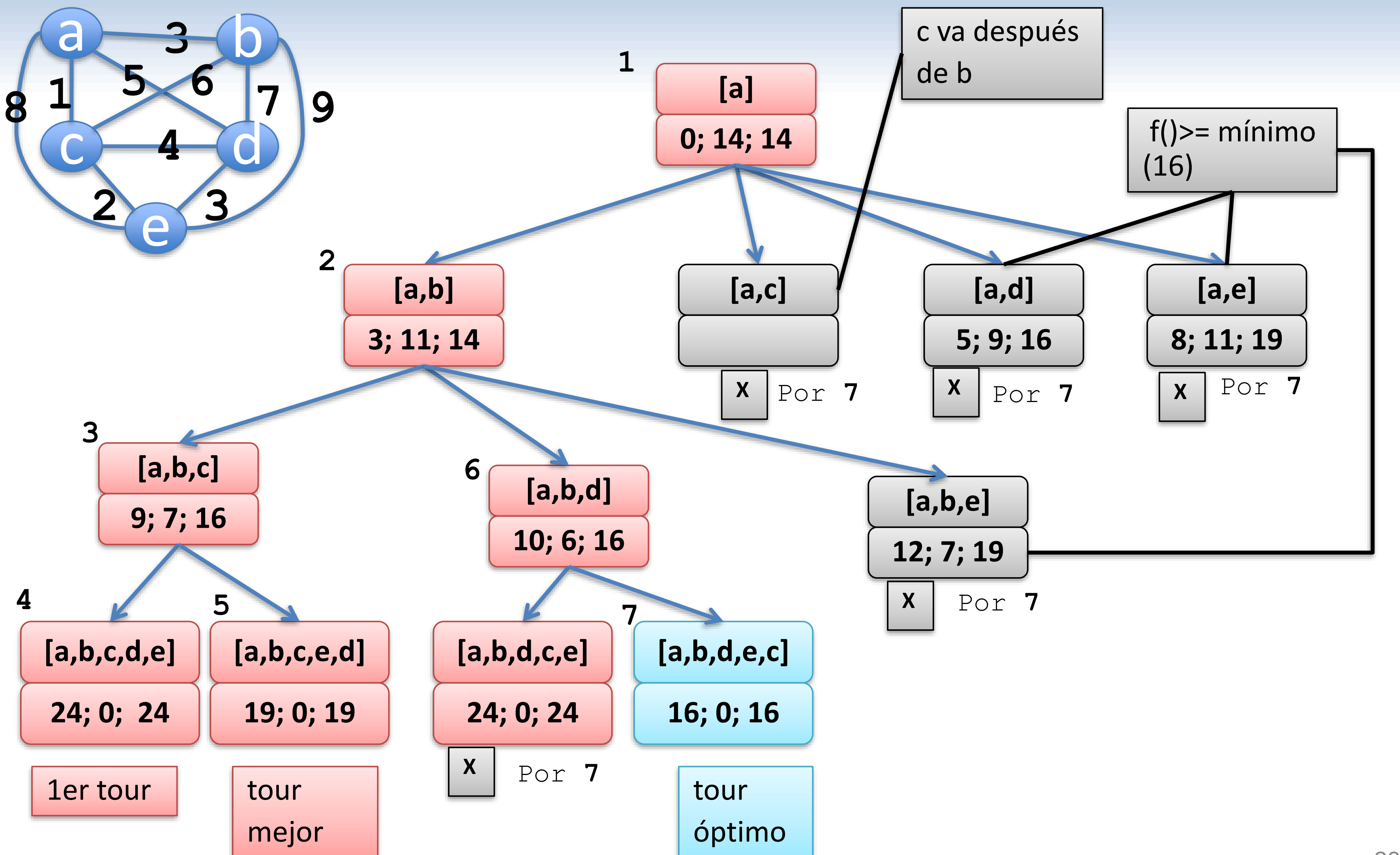
- Para resolver TSP es conveniente tener en cuenta dos aspectos:
  - Como se busca un ciclo en el grafo, *la ciudad inicial no es relevante*
    - Si observamos el grafo del dibujo, tenemos 6 tours posibles:



- [a,b,c,d] = 3 + 6 + 4 + 5 = 18
- [a,b,d,c] = 3 + 7 + 4 + 1 = 15 *óptima
- [a,c,b,d] = 1 + 5 + 7 + 6 = 19
- [a,c,d,b] = 1 + 4 + 7 + 3 = 15 *óptima
- [a,d,c,b] = 5 + 4 + 6 + 3 = 18
- [a,d,b,c] = 5 + 7 + 6 + 1 = 19

  - Cada pareja corresponde al mismo tour en sentido inverso, por lo tanto, si *fijamos un orden para dos ciudades cualesquiera*, (por ejemplo, *b* precede a *c*) dividimos por la mitad el número de tours posibles.

El problema del viajante de comercio



El problema de la mochila

- **Enunciado**: Dada una mochila de capacidad W, n objetos de pesos $w_1,\dots,w_n$ y valor $v_1,\dots,v_n$, encontrar la distribución de objetos en la mochila que hacen que tenga *el valor máximo*.
- Ordenamos los objetos según la razón v/w de manera que
  - $v_1/w_1 \geq v_2/w_2 \geq \dots \geq v_n/w_n$

El problema de la mochila

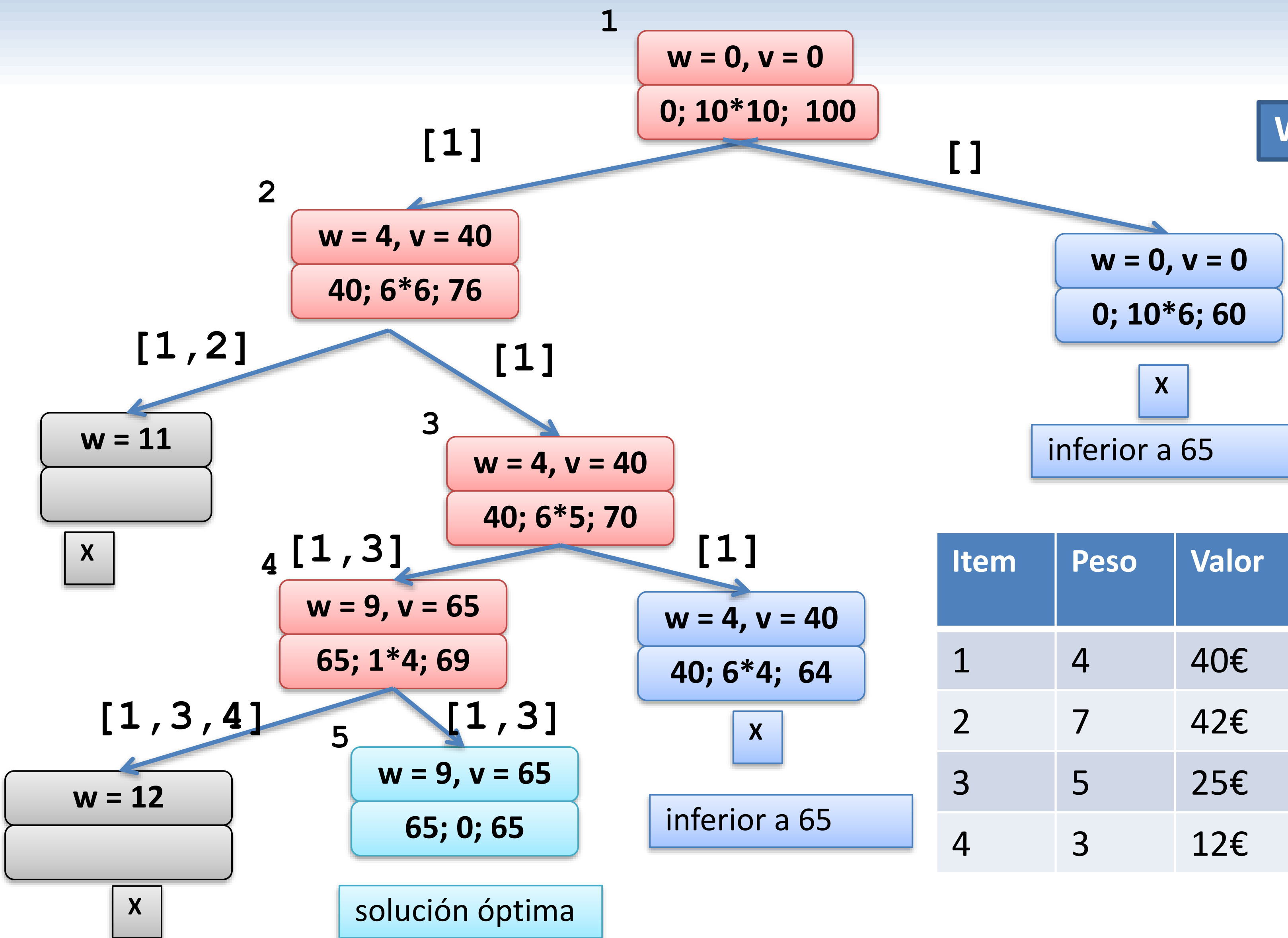
- El árbol de estados puede estructurarse como un árbol binario en el que cada nivel *i* del árbol representa todos los subconjuntos de los *n* items que tienen una configuración específica para los elementos *1* a *i*. Esta configuración está unívocamente determinada por el camino que va de la raíz hasta el nodo:
  - En el i-ésimo nivel, *la rama izquierda indica que el item i+1 se incluye en la mochila*, y *la rama derecha indica que el item i+1 se excluye de la mochila*
  - En cada nodo, se guarda el peso y el valor de la mochila en ese momento, junto con una *cota superior* del valor de cualquier mochila que incluya los items del nodo.

El problema de la mochila

- *Sea p la solución parcial en el nodo actual, que ya ha ocupado una parte del peso de la mochila* w_p *.*
- *Sabemos que la función completa es f(p)= g(p) + h(p)*
- *g(p) será la suma de los valores de los objetos que ya están en la mochila, digamos* v_p *.*
- El heurístico *h(p)* debe ser optimista, una cota superior.
- Si estamos en el nivel i, lo más que podemos es incluir el objeto *i+1* y suponer que TODO lo que falta por rellenar de la mochila, lo podemos hacer con ese objeto (que es el que mejor relación valor/peso tiene, pues previamente los hemos ordenado).
- Entonces *h(p)* será el producto de la capacidad restante de la mochila ($W-w_p$) por la mejor razón *v/w* de los restantes items (v_{i+1}/w_{i+1})
- En resumen:

$$f(p) = g(p) + h(p) = v_p + (W-w_p)\, v_{i+1}/w_{i+1}$$

El problema de la mochila



W = 10

Item	Peso	Valor	Valor/ Peso
1	4	40€	10
2	7	42€	6
3	5	25€	5
4	3	12€	4

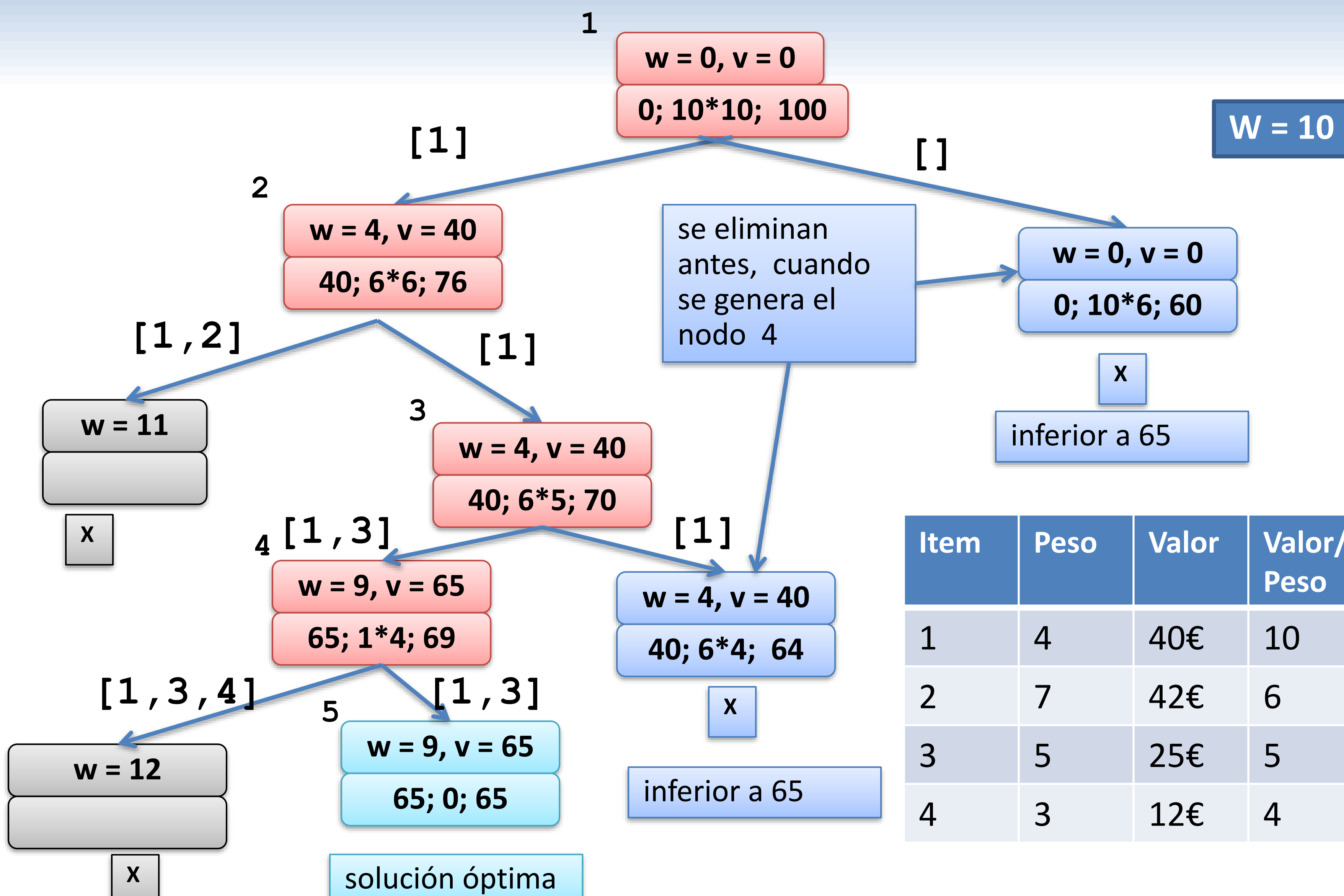
El problema de la mochila

- La solución al problema de la mochila con Ramificación y Poda tiene una característica poco habitual.
- Normalmente, los nodos interiores del árbol corresponden a soluciones parciales del problema, porque todavía hay algunas componentes indeterminadas.
- Sin embargo, en el problema de la mochila, cada nodo interior contiene un subconjunto de items, y puede ser, por lo tanto, una solución del problema.
- Podemos utilizar este hecho para actualizar la información sobre el mejor subconjunto visto hasta ese momento.

El problema de la mochila: optimización

```
TSolución BranchBound(TSubproblema p){
   Heap h = new Heap();
   h.insertar(p); // se introduce el subproblema inicial p en h
   float c = 0;
    Tsolucion s = null;
    while (!h.vacio()){
       Tsubproblema act = h.extraer();
       if (esSolucion(act)){
          if (cota(act)>c){
             c = cota(act);
             s = act;
              h.actualizar(c);
          }
       } else {
         List<TSubproblema> hijos = act.ramificar();
         for (TSubproblema hijo:hijos){
            if (cota(hijo)>c) {
              h.insertar(hijo);
              c = valor(hijo);
          }
       }
   }  return s;
}
```

El problema de la mochila



Item	Peso	Valor	Valor/ Peso
1	4	40€	10
2	7	42€	6
3	5	25€	5
4	3	12€	4

Referencias

- *Introduction to The Design & Analysis of Algorithms*. A. Levitin. Ed. Adison-Wesley
- *Introduction to Algorithms*. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Ed. The MIT Press