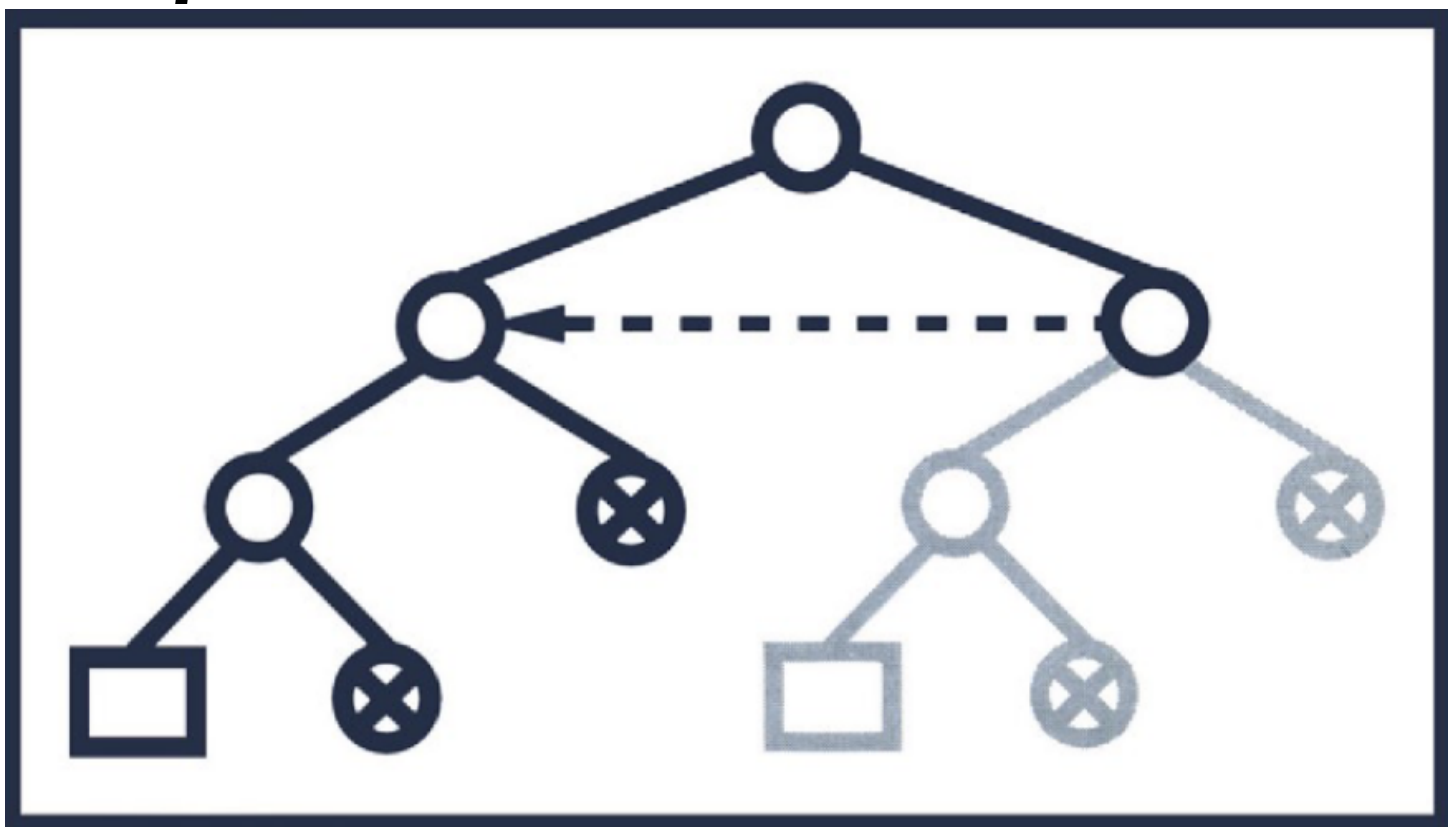
بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

Chapter 6: Branch-and-Bound



Recall that the central idea of backtracking, discussed in the previous chapter, is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution.

This idea can be strengthened further if we deal with an optimization problem. An optimization problem seeks to minimize or maximize some objective function, usually subject to some constraints.

Note that in the standard terminology of optimization problems, a feasible solution is a point in the problem's search space that satisfies all the problem's constraints (e.g., a Hamiltonian circuit in the traveling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an optimal solution is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Compared to backtracking, branch-and-bound requires two additional items:

* *a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node*
* *the value of the best solution seen so far*

نکتهٔ مهم:

The bound should be a lower bound for a minimization problem and an upper bound for a maximization problem.

If this information is available, we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is "pruned"). Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

- *The value of the node's bound is not better than the value of the best solution seen so far.*
- *The node represents no feasible solutions because the constraints of the problem are already violated.*
- *The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.*

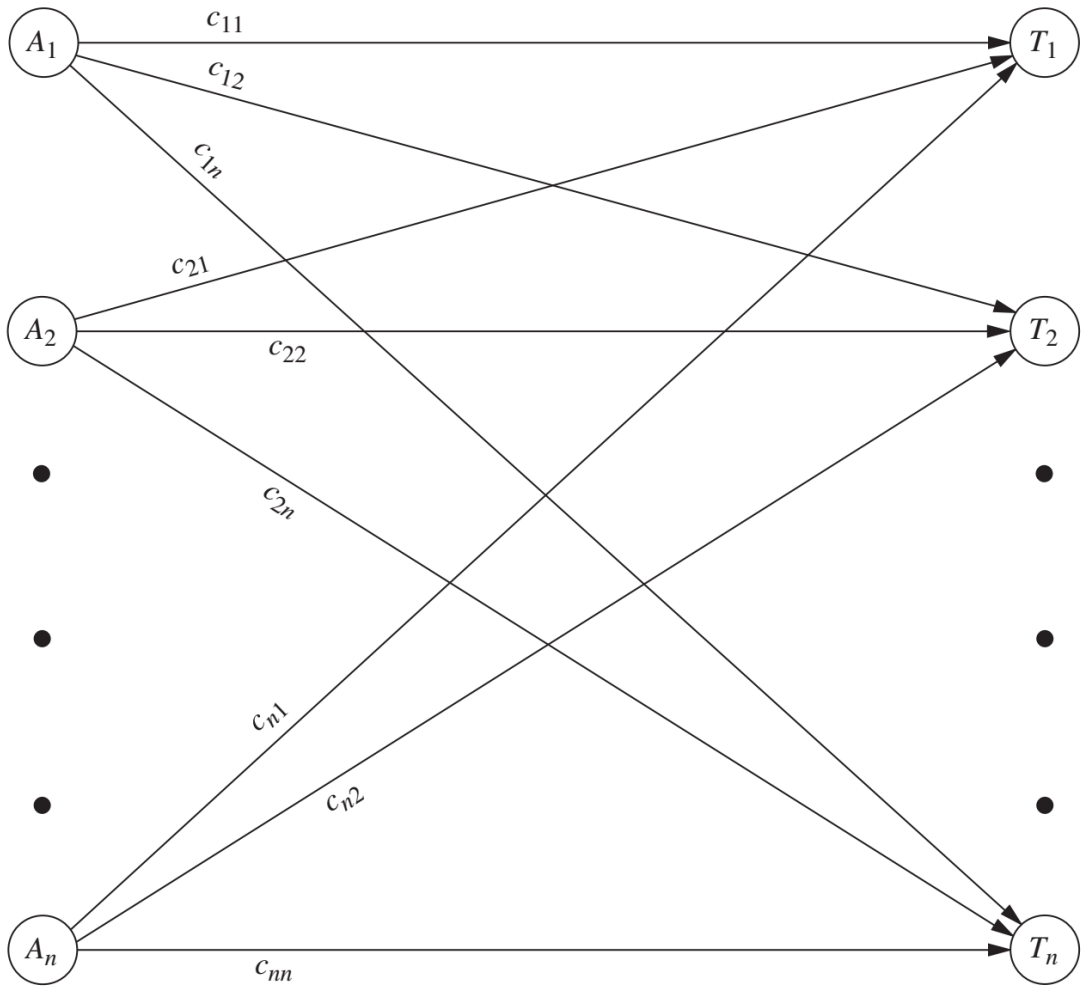
Assignment Problem

☞ *Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible.*

☞ *An instance of the assignment problem is specified by an $n \times n$ cost matrix C so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.*

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

The assignment problem can be depicted in a very similar way, as shown in the following figure. The first column now lists the n assignees and the second column the n tasks.



* *How can we find a lower bound on the cost of an optimal selection without actually solving the problem? We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is* $2 + 3 + 1 + 4 = 10$.

* *It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection.*

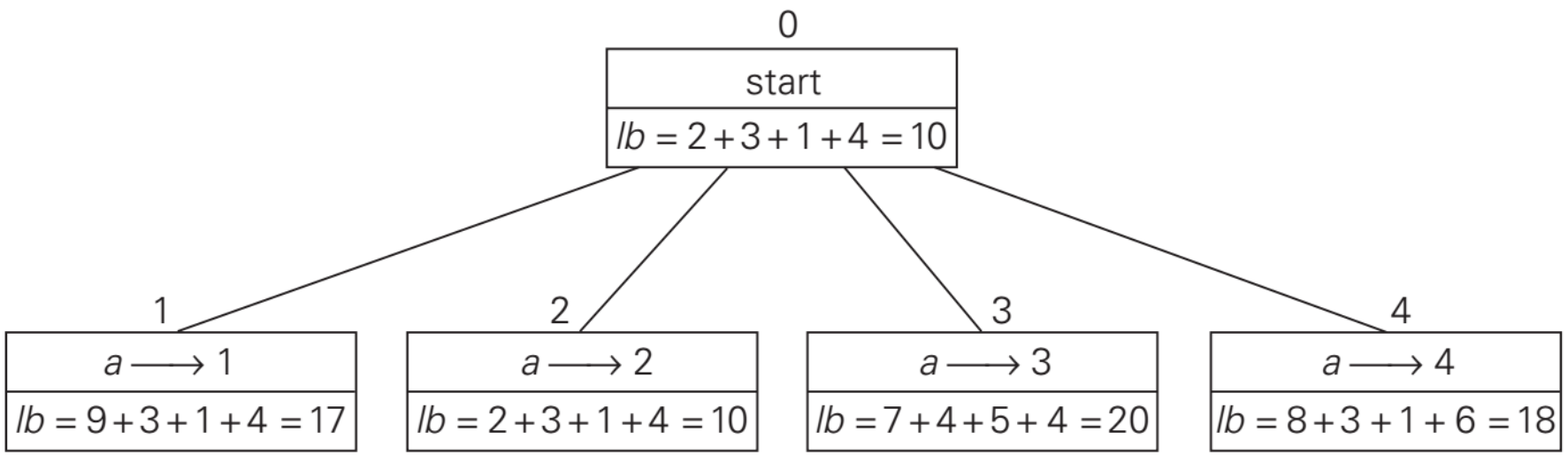
* *We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects* 9 *from the first row, the lower bound will be* $9 + 3 + 1 + 4 = 17$.

Best-first branch-and-bound

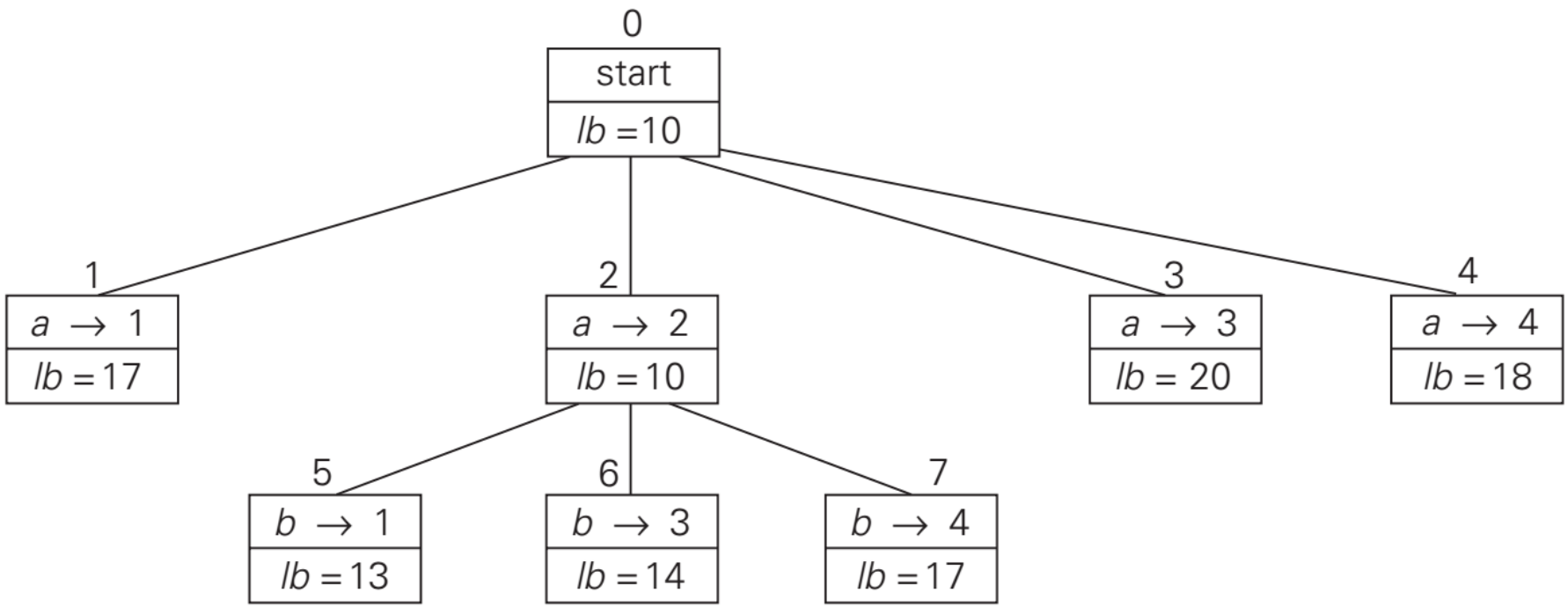
Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among nonterminated leaves in the current tree. (Nonterminated, i.e., still promising, leaves are also called live.)

How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the best-first branch-and-bound.

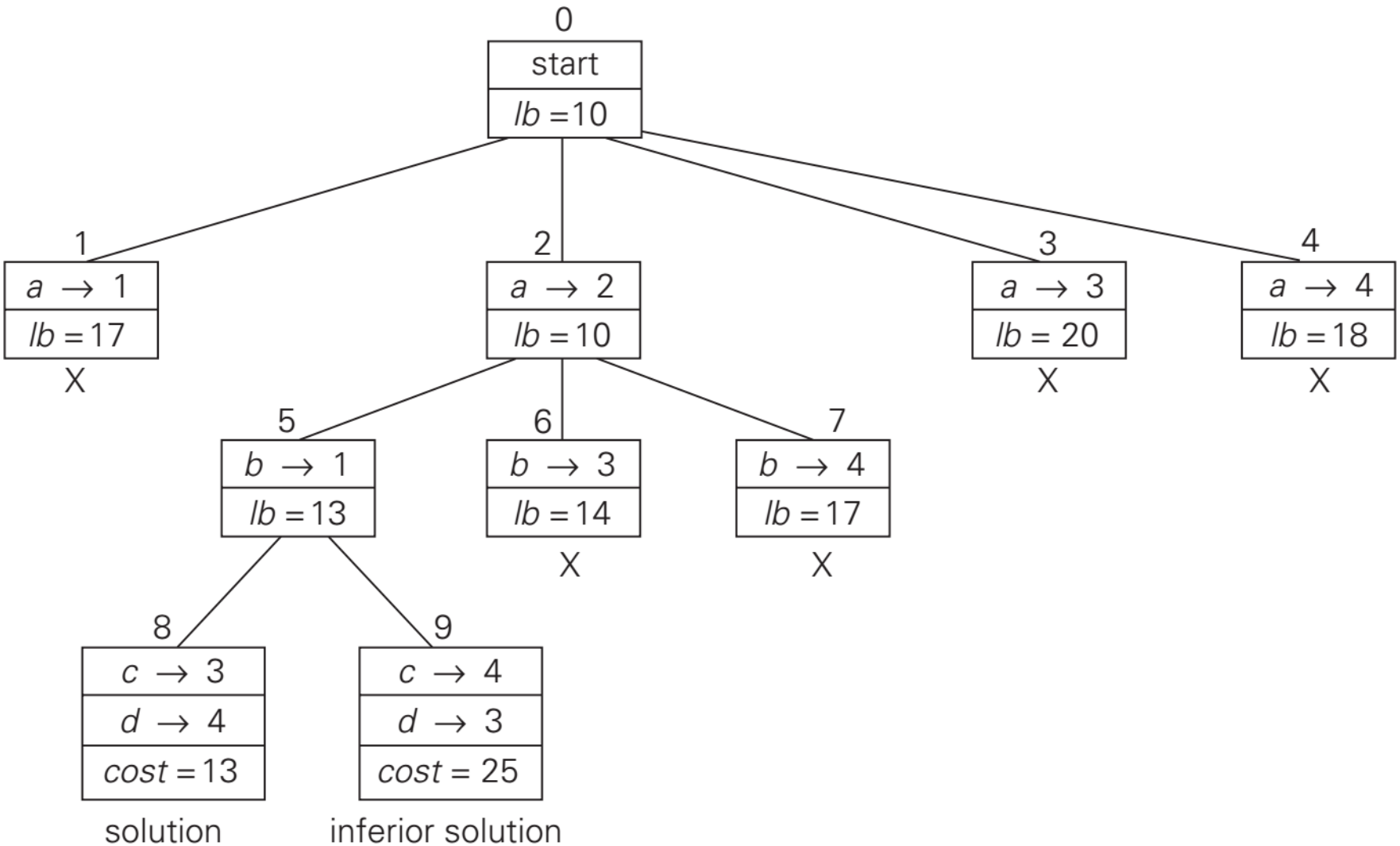
Returning to the instance of the assignment problem given earlier, we start with the root that corresponds to no elements selected from the cost matrix. As we already discussed, the lower-bound value for the root, denoted lb, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a. So we have four live leaves—nodes 1 through 4—that may contain an optimal solution.



The most promising of them is node 2 because it has the smallest lower-bound value. ***Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person b.***



Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from c's row (i.e., assigning person c to job 3); this leaves us with no choice but to select the element from the fourth column of d's row (assigning person d to job 4). This yields leaf 8, which corresponds to the feasible solution $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 3, d \rightarrow 4\}$ *with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution* $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 4, d \rightarrow 3\}$ *with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)*


0
start
lb = 10
1
a → 1
lb = 17
X
2
a → 2
lb = 10
3
a → 3
lb = 20
X
4
a → 4
lb = 18
X
5
b → 1
lb = 13
6
b → 3
lb = 14
X
7
b → 4
lb = 17
X
8
c → 3
d → 4
cost = 13
solution
9
c → 4
d → 3
cost = 25
inferior solution

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in the figure—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

Before we leave the assignment problem, we have to remind ourselves again that, unlike for our next examples, there is a polynomial-time algorithm for this problem called the Hungarian method. In the light of this efficient algorithm, solving the assignment problem by branch-and-bound should be considered a convenient educational device rather than a practical recommendation.

مجدداً بیان میکنیم که:

The branch-and-bound design strategy is very similar to back-tracking in that a state space tree is used to solve a problem. The differences are that the branch-and-bound method (1) does not limit us to any particular way of traversing the tree and (2) is used only for optimization problems.

A branch-and-bound algorithm computes a number (bound) at a node to determine whether the node is promising. The number is a bound on the value of the solution that could be obtained by expanding beyond the node. If that bound is no better than the value of the best solution found so far, the node is nonpromising. Otherwise, it is promising. Because the optimal value is a minimum in some problems and a maximum in others, by "better" we mean smaller or larger, depending on the problem. As is the case for backtracking algorithms, branch-and-bound algorithms are ordinarily exponential-time (or worse) in the worst case. However, they can be very efficient for many large instances.

آیا کران، بغیر از تعیین امیدبخش بودن یا غیرامیدبخش بودن یک نود، کاربرد دیگری هم دارد؟ بله!

Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound. In this way we often can arrive at an optimal solution faster than we would by methodically visiting the nodes in some predetermined order (such as a depth-first search). This approach is called best-first search with branch-and-bound pruning.

Knapsack Problem

Given** n **items of known weights** w_i **and values** v_i, $i = 1, 2, \ldots, n$, **and a knapsack of capacity** W, **find the most valuable subset of the items that fit in the knapsack.

It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows. Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.

یک روش ساده برای محاسبهٔ کران در هر نود

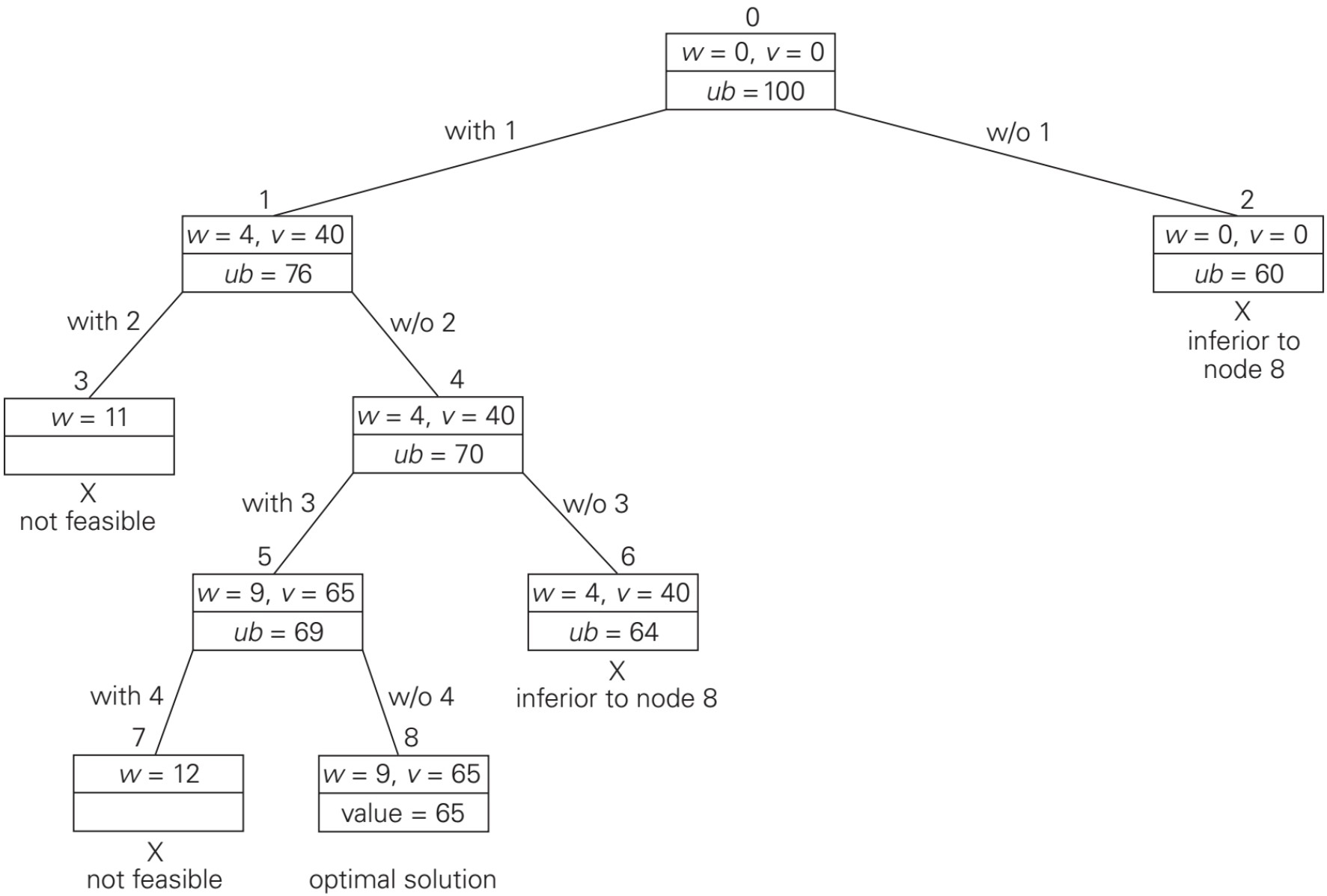
We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is $\frac{v_{i+1}}{w_{i+1}}$:

$$ub = v + (W - w)\left(\frac{v_{i+1}}{w_{i+1}}\right).$$

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.


0
w = 0, v = 0
ub = 100
with 1
w/o 1
1
w = 4, v = 40
ub = 76
2
w = 0, v = 0
ub = 60
X
inferior to
node 8
with 2
w/o 2
3
w = 11
X
not feasible
4
w = 4, v = 40
ub = 70
with 3
w/o 3
5
w = 9, v = 65
ub = 69
6
w = 4, v = 40
ub = 64
X
inferior to node 8
with 4
w/o 4
7
w = 12
X
not feasible
8
w = 9, v = 65
value = 65
optimal solution

Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 *and* $\$40$*, respectively; the value of the upper bound is* $40+(10-4)*6 = \$76$*. Node 2 represents the subsets that do not include item 1. Accordingly,* $w = 0$*,* $v = \$0$*, and* $ub = 0 + (10 - 0) * 6 = \$60$*.*

Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first.

*Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively. Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and v as its parent; the upper bound ub is equal to $40 + (10 - 4) * 5 = \$70$. Selecting node 4 over node 2 for the next branching (why?), we get nodes 5 and 6 by respectively including and excluding item 3.*

Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset $\{1, 3\}$ of value $\$65$. The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset $\{1, 3\}$ of node 8 the optimal solution to the problem.

روش سادهٔ فوق برای محاسبهٔ کران، در کتاب ***Levitin*** مطرح شده است. در کتاب ***Neapolitan***، یک روش قویتر برای محاسبهٔ کران معرفی شده است.

وقتی چند استراتژی برای تعیین کران در دست داریم:

در یک مسئلهٔ بیشینه‌سازی، هرچه کران بالایی که در یک نود محاسبه می‌شود کمتر باشد، هرس کردن بهتر واقع می‌شود. پس اگر در یک نود از درخت فضای حالت به دو روش مختلف کران بالا را محاسبه کردیم، یکی ub_1 و دیگری ub_2، که $ub_1 < ub_2$، آنگاه قطعاً ub_1 کران بهتری است نسبت به ub_2. چرا؟ چون اگر بهترین جواب شدنی دردست ما برابر با α باشد، آنگاه احتمال اینکه ub_1 کمتر از α باشد بیشتر از آن است که ub_2 کمتر از α باشد. پس استفاده از ub_1 ممکن است به هرس شدن نود بیانجامد اما در مورد ub_2 شاید چنین نباشد. (متناظر با این، برای مسائل کمینه‌سازی، هرچه کران نظیر نود بیشتر باشد امکان هرس شدن نود بیشتر است.)

اکنون به دنبال ارائهٔ یک الگوریتم شاخه و کران بهتر برای مسئلهٔ کوله پشتی ۰ـ۱ هستیم.