(a) A live node in $AS(x)$ can kill nodes in $CH(x)$.

(b) Nodes in $CH(x)$ can kill nodes in $AS(x)$.

(c), (d) The upper bound $U(x)$ can kill nodes in $AS(x)$ or $CH(x)$.

Again there is a trade-off in the choice of what is to be implemented: the time taken to test $CH(x)$ against $AS(x)$ may or may not be worthwhile in any particular problem.

Fourth, there is the choice at each branching step of which node to branch from. The usual alternatives are least-lower-bound-next, last-in-first-out, or first-in-first-out.

Still another choice must be made at the start of the algorithm. It is often practical to generate an initial solution by some heuristic construction, such as those described in Chapters 17 or 19. This gives us an initial upper bound $U < \infty$ and may be very useful for killing nodes early in the algorithm. As usual, however, we must trade off the time required for the heuristic against possible benefit.

Finally, we should mention that the branch-and-bound algorithm is often terminated before optimality is reached, either by design or necessity. In such a case we have a complete solution with cost U, and the lowest lower bound L of any live node provides a lower bound on the optimal cost. We are therefore within a ratio of $(U - L)/L$ of optimal.

It should be clear by now that the branch-and-bound idea is not one specific algorithm, but rather a very wide class. Its effective use is dependent on the design of a strategy for the particular problem at hand, and at this time is as much art as science. We conclude the discussion of branch-and-bound with an example of its application to a scheduling problem.

18.5 Application to a Flowshop Scheduling Problem

We now describe something of a "case history" of the application of branch-and-bound to a scheduling problem of some general interest—the two-machine flowshop scheduling problem with a sum-finishing-time criterion, defined as follows.

Definition 18.3

We are given a set of n jobs, J_i, $i = 1, \ldots, n$. Each job has two tasks, each to be performed on one of two machines. Job J_i requires a processing time τ_{ij} on machine i, and each task must complete processing on machine 1 before starting on machine 2. Let F_{ji} be the time at which job i finishes on machine j. The *sum finishing time* is defined to be the sum of the times that all the jobs finish processing on machine 2:

$$f = \sum_{i=1}^{n} F_{2i}$$

The sum-finishing-time problem (SFTP) is the problem of determining the order in which to assign the tasks to machines so f is minimum. □

Example 18.5

A common example of a situation in which a problem like SFTP may arise is a computer that executes one program at a time. We can identify the jobs with individual computer programs, machine 1 with the central processor, and machine 2 with the printer. We then assume that we are given a set of jobs with known execution and printing times and wish to schedule them so that the sum (or, equivalently, the average) finishing time is as small as possible. □

We now quote two important facts about SFTP. The first can be found in Conway, Maxwell, and Miller [CMM] and allows us to restrict our search for a single permutation that determines a complete schedule.

Theorem 18.1 *There is an optimal schedule for SFTP in which both machines process the jobs in the same order with no unnecessary idle time between jobs. (These are called* permutation *schedules.)*

The second, more recent, result is due to Garey, Johnson, and Sethi [GJS] and justifies the serious pursuit of a branch-and-bound algorithm.

Theorem 18.2 *The problem SFTP is NP-complete. (We mean, of course, the* yes-no *problem corresponding to SFTP with a solution of cost less than or equal to some L.)*

Example 18.6

Consider the following numerical example with 3 jobs.

τ_{ij}	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

Figure 18-9 shows all 6 possible permutation schedules, among which the optimal schedule must lie, by Theorem 18.1. The unique optimum has cost 18. □

```
Machine 1   1 1 2 2 2 3 3
Machine 2       1     2   3 3 3        f = 19
                 ↑     ↑       ↑

Machine 1   1 1 3 3 2 2 2
Machine 2       1   3 3 3 2            f = 18
                 ↑       ↑ ↑

Machine 1   2 2 2 1 1 3 3
Machine 2         2   1   3 3 3        f = 20
                   ↑   ↑       ↑

Machine 1   2 2 2 3 3 1 1
Machine 2         2   3 3 3 1          f = 21
                   ↑       ↑ ↑

Machine 1   3 3 1 1 2 2 2
Machine 2       3 3 3 1   2            f = 19
                     ↑ ↑   ↑

Machine 1   3 3 2 2 2 1 1
Machine 2       3 3 3 2   1            f = 19
                     ↑ ↑   ↑
```

Figure 18-9 The six possible permutation schedules in Example 18-5, and their associated costs. The arrows indicate finishing times on machine 2.

This problem is, with the help of Theorem 18.1, a problem of finding one permutation of n objects, and the natural way to branch is to choose the first job to be scheduled at the first level of the branching tree, the second job at the next level, and so on. What we need next is a lower-bound function.

Ignall and Schrage [IS] describe a very effective lower bound, which we derive here. Suppose we are at a node at which the jobs in the set $M \subseteq \{1, \ldots, n\}$ have been scheduled, where $|M| = r$. Let t_k, $k = 1, \ldots, n$, be the index of the kth job under any schedule which is a descendant of the node under consideration. The cost of this schedule, which we wish to bound, is

$$f = \sum_{i \in M} F_{2i} + \sum_{i \notin M} F_{2i} \qquad (18.5)$$

Now if every job could start its processing on machine 2 immediately after completing its processing on machine 1, the second sum in Eq. 18.5 would become

$$S_1 = \sum_{k=r+1}^{n} [F_{1t_r} + (n - k + 1)\tau_{1t_k} + \tau_{2t_k}] \qquad (18.6)$$

(see Problem 3). If that is not possible, S_1 can only increase, so

$$\sum_{i \notin M} F_{2i} \geq S_1 \qquad (18.7)$$

Similarly, if every job can start on machine 2 immediately after the preceding job finishes on machine 2, the second sum in Eq. 18.5 would become

$$S_2 = \sum_{k=r+1}^{n} [\max(F_{2t_r}, F_{1t_r} + \min_{i \notin M} \tau_{1i}) + (n - k + 1)\tau_{2t_k}] \qquad (18.8)$$

[illegible]

Again, this is a lower bound:

$$\sum_{i \notin M} F_{2i} \geq S_2 \qquad (18.9)$$

Therefore

$$f \geq \sum_{i \in M} F_{2i} + \max(S_1, S_2) \qquad (18.10)$$

The bound depends on the way the remaining jobs are scheduled, through t_k. This dependence can be eliminated by noting that S_1 is minimized by choosing t_k so that the tasks of length τ_{1t_k} are in ascending order, and that S_2 is minimized by choosing t_k so that the tasks of length τ_{2t_k} are likewise in ascending order. Call the resulting minimum values $\hat{S}_1$ and $\hat{S}_2$. Then

$$f \geq \sum_{i \in M} F_{2i} + \max(\hat{S}_1, \hat{S}_2) \qquad (18.11)$$

is an easily computed lower bound.

Example 18.6 (Continued)

At the first branching step, Eq. 18.11 yields the lower bounds

$$f = \begin{cases} 18 & \text{if job 1 is scheduled first} \\ 20 & \text{if job 2 is scheduled first} \\ 18 & \text{if job 3 is scheduled first} \end{cases}$$

From the results in Figure 18-9, we see that the first two of these are as low as possible. Figure 18-10 shows a complete search tree in which the least lower bound is branched from first, from left to right in case of ties. The optimal solution (1, 3, 2) kills all the others when it is obtained. ☐

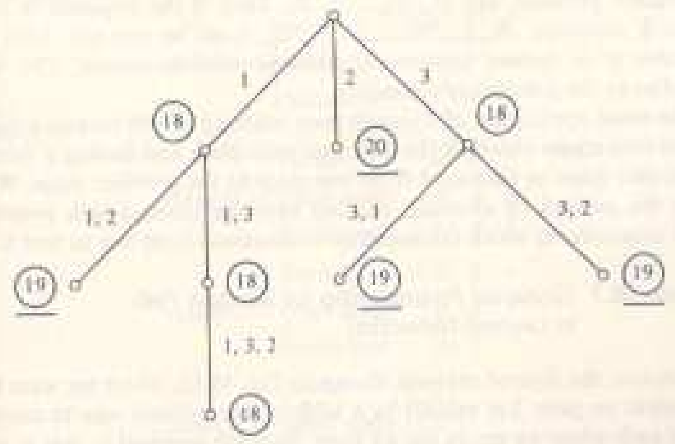

Figure 18-10 The search tree for Example 18.6.

Finally, we describe a natural dominance relation also given by Ignall and Schrage [IS]. Suppose we have two nodes t and u representing partial assignments of the same set of jobs, M. Let the kth scheduled job be t_k and u_k, $k = 1, \ldots, r$, under partial schedules t and u, respectively. Then if

$$F_{2t_r} \leq F_{2u_r} \tag{18.12}$$

(the set of jobs in M finishes no later on machine 2 under the partial schedule t), and if the accumulated cost under partial schedule t is no more than that under u,

$$\sum_{i \in M} F_{2i}|_{\text{schedule } t} \leq \sum_{i \in M} F_{2i}|_{\text{schedule } u} \tag{18.13}$$

then the best completion of schedule t is at least as good as the best completion of u. Equations 18.12 and 18.13 therefore define a dominance relation of t over u.

Example 18.6 (Continued)

Consider the nodes $t = (1, 2)$ and $u = (2, 1)$ (not generated in Fig. 18-10). Then t dominates u. On the other hand, $t = (1, 3)$ does not dominate $u = (3, 1)$. This instance is too small to show the real power of a dominance relation. □

18.6 Dynamic Programming

Dynamic programming is related to branch-and-bound in the sense that it performs an intelligent enumeration of all the feasible points of a problem, but it does so in a different way. The idea is to work backwards from the last decisions to the earlier ones.

Suppose we need to make a sequence of n decisions to solve a combinatorial optimization problem, say $D_1, D_2, \ldots, D_n$. Then if the sequence is optimal, the last k decisions, $D_{n-k+1}, D_{n-k+2}, \ldots, D_n$, must be optimal. That is, the *completion of an optimal sequence of decisions must be optimal*. This is often referred to as the *principle of optimality*.

The usual application of dynamic programming entails breaking down the problem into stages at which the decisions take place and finding a recurrence relation that takes us backward from one stage to the previous stage. We shall explain the method by example, starting with the shortest-path problem for layered networks, in which the sequence of decisions from last to first is clear.

Example 18.7 (Dynamic Programming for Shortest Path in Layered Networks)

Consider the layered network shown in Fig. 18.11, where we want to find the shortest s-t path. Let $table(i)$ be a table of the optimal way to continue a shortest path when we are in the ith layer from the terminal t; that is, $table(i)$ contains the best decision for each node in the layer i arcs from t. Thus the first

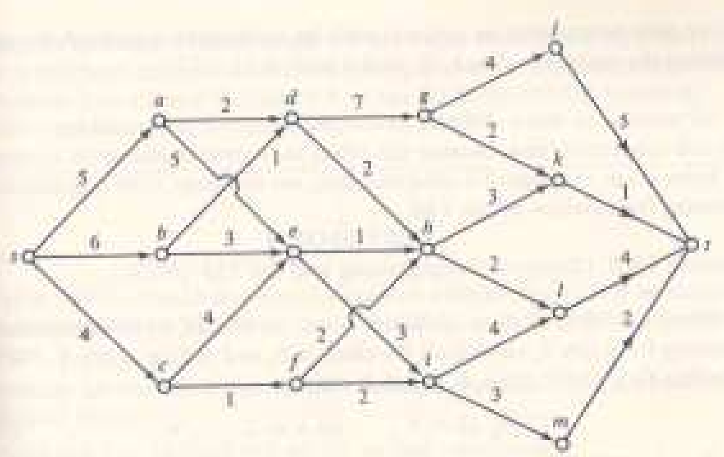

Figure 18-11 A shortest-path problem in a layered network.

table is simply

$$table(1) = \begin{cases} \text{node} & j \quad k \quad l \quad m \\ \text{next node} & t \quad t \quad t \quad t \\ \text{total cost} & 5 \quad 1 \quad 4 \quad 2 \end{cases}$$

Now consider the construction of $table(2)$. At node g we can reach j or k. We know the cost of the optimal completion from j or k by $table(1)$ and thus can find the best decision at node g by comparing the cost of arc (g, j) plus the cost of completion from j, with the cost of arc (g, k) plus the cost of completion from k. Continuing for nodes h and i, $table(2)$ is

$$table(2) = \begin{cases} \text{node} & g \quad h \quad i \\ \text{next node} & k \quad k \quad m \\ \text{total cost} & 3 \quad 4 \quad 5 \end{cases}$$

Next we find

$$table(3) = \begin{cases} \text{node} & d \quad e \quad f \\ \text{next node} & h \quad h \quad h \\ \text{total cost} & 6 \quad 5 \quad 6 \end{cases}$$

$$table(4) = \begin{cases} \text{node} & a \quad b \quad c \\ \text{next node} & d \quad d \quad f \\ \text{total cost} & 8 \quad 7 \quad 7 \end{cases}$$

$$table(5) = \begin{cases} \text{node} & s \\ \text{next node} & c \\ \text{total cost} & 11 \end{cases}$$