UNIT-V

State the subset-sum problem and Complete state-space tree of the backtracking algorithm applied to the instance A={3, 5, 6, 7} and d=15 of the subset-sum problem.[M-16]

The ***subset-sum problem*** finds a subset of a given set $A = \{a1, \ldots, an\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set‘s elements in increasing order. So, we will assume that $a1 < a2 < \ldots < an$. For Example,

A = {3, 5, 6, 7} and d = 15 of the subset-sum problem. The number inside a node is the sum

of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

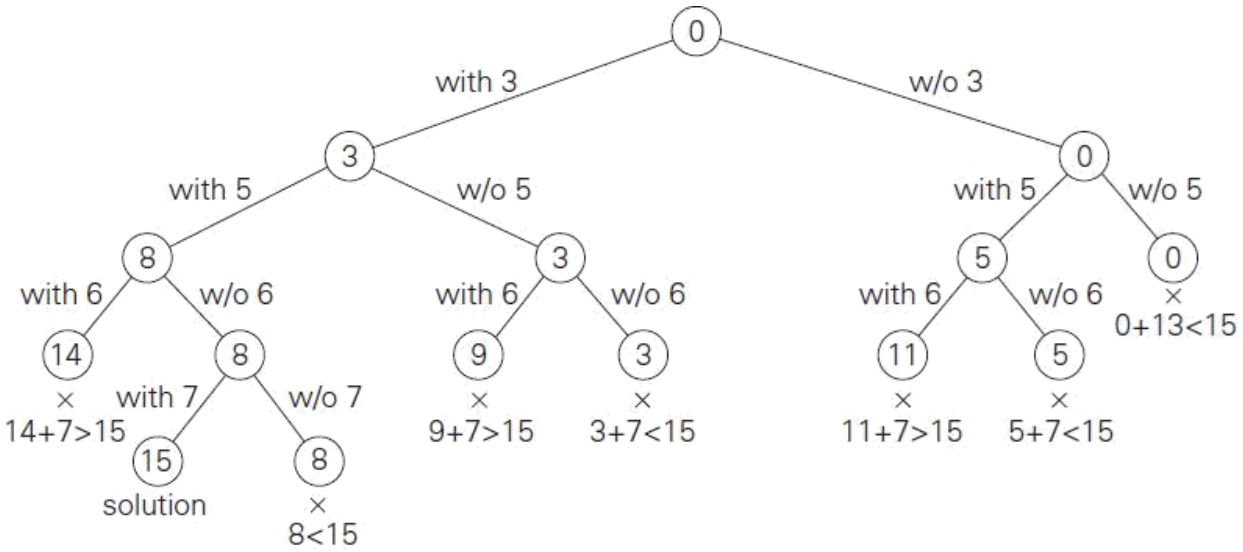


FIGURE Complete state-space tree of the backtracking algorithm applied to the instance

- The state-space tree can be constructed as a binary tree like that in the instance $A = \{3, 5, 6, 7\}$ and $d = 15$.

- The root of the tree represents the starting point, with no decisions about the given elements made as yet.
- Its left and right children represent, respectively, inclusion and exclusion of $a1$ in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of $a2$ while going to the right corresponds to its exclusion, and so on.
- Thus, a path from the root to a node on the ith level of the tree indicates which of the first I numbers have been included in the subsets represented by that node. We record the value of s, the sum of these numbers, in the node.
- If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node‘s parent.
- If s is not equal to d, we can terminate the node as non-promising if either of the following two inequalities holds:

  + i+1 > [the sum s is too large], s +

  = +1 j < d [the sum s is too small].

APPROXIMATION ALGORITHM FOR NP HARD PROBLEMS

A polynomial-time approximation algorithm is said to be a approximation algorithm, where c ≥ 1, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question: r(sa) ≤ c.

KNAPSACK PROBLEM

The knapsack problem, given n items of known weights w1, . . . , wn and values v1, . . . , vn and a knapsack of weight capacity W, find the most valuable subset of the items that fits into the knapsack.

Greedy algorithm for the discrete knapsack problem

Step 1 Compute the value-to-weight ratios ri= vi/wi, i = 1, . . . , n, for the items given.

Step 2 Sort the items in non-increasing order of the ratios computed in Step 1.

Step 3 Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack and proceed to the next item; otherwise, just proceed to the next item.

EXAMPLE 5 Let us consider the instance of the knapsack problem with the knapsack capacity 10 and the item information as follows:

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

Computing the value-to-weight ratios and sorting the items in nonincreasing order of these efficiency ratios yields

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

K-A PPROXIMATION SCHEME

Approximation Schemes We now return to the discrete version of the knapsack problem. For this problem, unlike the traveling salesman problem, there exist polynomial-time ***approximation schemes***, which are parametric families of algorithms that allow us to get approximations *s(k) a* with any predefined accuracy level:

$$\frac{f(s^*)}{f(s_a^{(k)})} \leq 1 + 1/k \quad \text{for any instance of size } n,$$

where k is an integer parameter in the range $0 \leq k < n$.

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	1	$ 4	4

capacity $W = 10$

(a)

subset	added items	value
∅	1, 3, 4	$69
{1}	3, 4	$69
{2}	4	$46
{3}	1, 4	$69
{4}	1, 3	$69
{1, 2}	not feasible	
{1, 3}	4	$69
{1, 4}	3	$69
{2, 3}	not feasible	
{2, 4}		$46
{3, 4}	1	$69

(b)

FIGURE 12.16 Example of applying Sahni's approximation scheme for $k = 2$. (a) Instance. (b) Subsets generated by the algorithm.

TRAVELING SALESMAN PROBLEM

Greedy Algorithms for the TSP

The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique..

Nearest-neighbor algorithm

The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic: always go next to the nearest unvisited city.

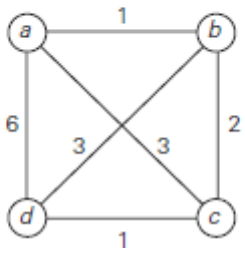
Step 1 Choose an arbitrary city as the start.

Step 2 Repeat the following operation until all the cities have been visited:go to the unvisited city nearest the
one visited last (ties can be broken arbitrarily).

Step 3 Return to the starting city.

For Example,

EXAMPLE 1 For the instance represented by the graph in Figure 12.10, with a as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) s_a: $a - b - c - d - a$ of length 10.



The optimal solution, as can be easily checked by exhaustive search, is the tour s^*: $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(i.e., tour s_a is 25% longer than the optimal tour s^*). ■

Multifragment-heuristic algorithm

Step 1 Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.
Step 2 Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.
Step 3 Return the set of tour edges.

As an example, applying the algorithm to the above graph yields $\{(a, b), (c, d), (b, c), (a, d)\}$. This set of edges forms the same tour as the one produced by the nearest-neighbor algorithm. In general, the multifragment-heuristic algorithm tends to produce significantly better tours than the nearest-neighbor algorithm, as we are going to see from the experimental data quoted at the end of this section. But the performance ratio of the multifragment-heuristic algorithm is also unbounded, of course.

Minimum-Spanning-Tree–Based Algorithms

There are approximation algorithms for the traveling salesman problem that exploit a connection between Hamiltonian circuits and spanning trees of the same graph. Since removing an edge from a Hamiltonian circuit yields a spanning tree, we can expect that the structure of a minimum spanning tree provides a good basis for constructing a shortest tour approximation. Here is an algorithm that implements this idea in a rather straightforward fashion.

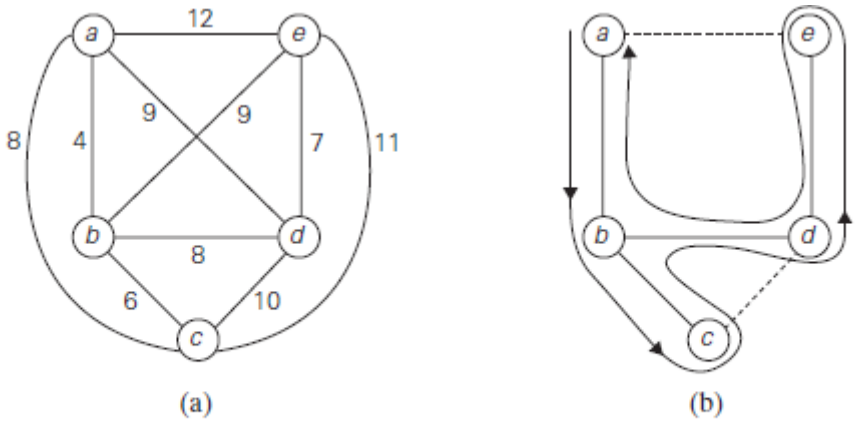
Twice-around-the-tree algorithm

Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

Step 2 Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)
Step 3 Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the

algorithm.

EXAMPLE 2 Let us apply this algorithm to the graph in Figure 12.11a. The minimum spanning tree of this graph is made up of edges *(a, b)*, *(b, c)*, *(b, d)*, and *(d, e)* (Figure 12.11b). A twice-around-the-tree walk that starts and ends at *a* is *a, b, c, b, d, e, d, b, a*. Eliminating the second *b* (a shortcut from *c* to *d*), the second *d*, and the third *b* (ashortcut from *e* to *a*) yields the Hamiltonian circuit*a, b, c, d, e, a* of length 39.



IGURE 12.11 Illustration of the twice-around-the-tree algorithm. (a) Graph. (b) Walk around the minimum spanning tree with the shortcuts.

3. Using an example prove that, satisfiability of Boolean formula in 3-conjunctive normal form is NP-Complete. Or Briefly explain NP-Hard and NP-Completeness with example. Or Write short notes on deterministic and non-deterministic algorithms.

Class P: An algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to *O(p(n))* where *p(n)* is a polynomial of the problem‘s input size *n*. (Note that since we are using big-oh notation here, problems solvable in, say, logarithmic time are solvable in polynomial time as well.) Problems that can

be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

- Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial. (Class P)
- Examples: Searching, Element uniqueness, primality test, graph acyclicity

Class NP: A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following. Nondeterministic (―guessing‖) stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (but may be complete gibberish as well).

- Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial.
- Examples: TSP, AP, Graph coloring problem, partition problem, Hamiltonian circuit proble

Class- NP Complete: A decision problem D1 is said to be polynomially reducible to a decision problem D2, if there exists a function t that transforms instances of D1 to instances of D2 such that:

1. t maps all yes instances of D1 to yes instances of D2 and all no instances of D1 to no instances of D2
2. t is computable by a polynomial time algorithm

1. it belongs to class NP
2. every problem in NP is polynomially reducible to D

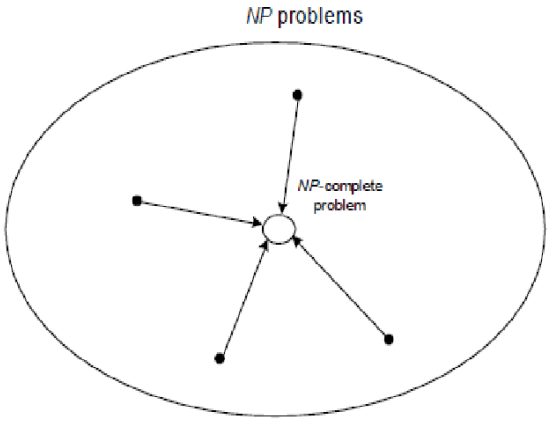


FIGURE 5.6 Polynomial-time reductions of *NP* problems to an *NP*-complete problem

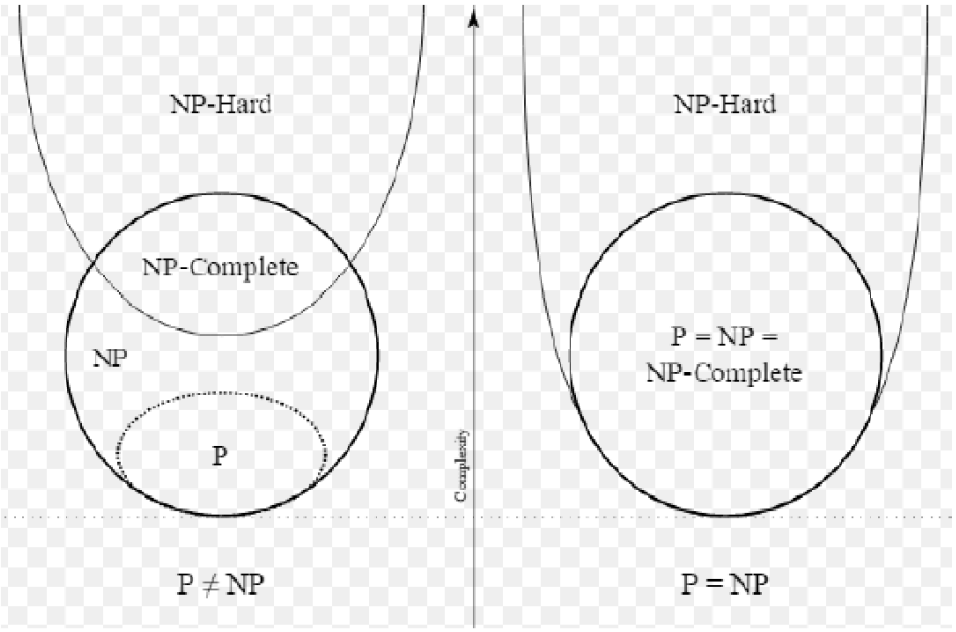


FIGURE 5.8 Relation among P, NP, NP-hard and NP Complete problems

4. Solve n-Queens problem. Or Explain 8-Queens problem with an algorithm. Explain why backtracking is the default procedure for solving problems. Or Explain 8 Queens problem with example.[M-14][N-13] [N-14]

The n-queens problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For *n* = 1, the problem has **a trivial solution**.

Q

For n = 2, it is easy to see that there is **no solution** to place 2 queens in 2 × 2 chessboard.

Q	

For *n* = 3, it is easy to see that there is **no solution** to place 3 queens in 3 × 3 chessboard.

	1	2	3	
1	Q			← queen 1
2			Q	← queen 2
3				

Or

	1	2	3	
1	Q			← queen 1
2				
3		Q		← queen 2

Or

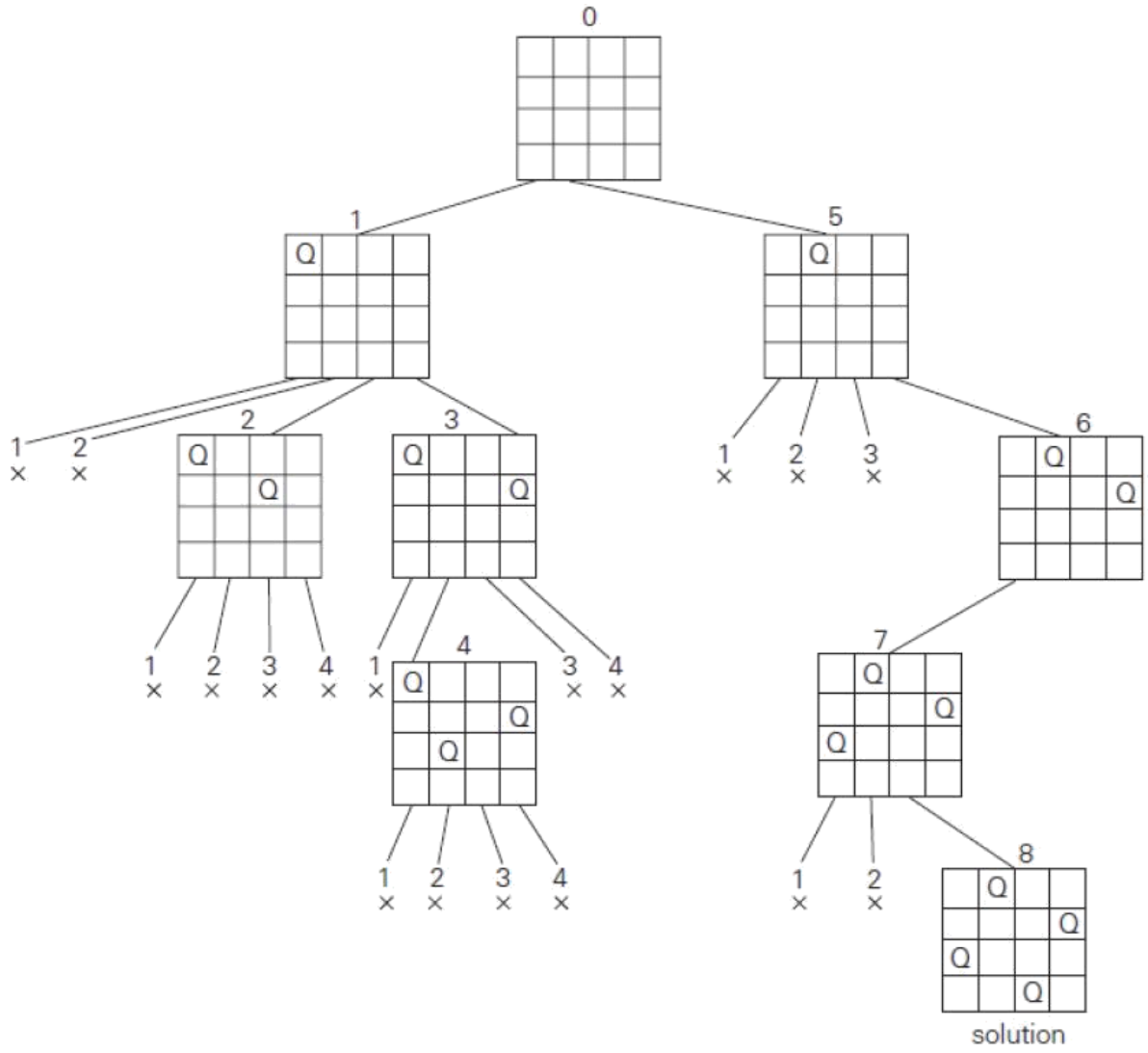
	1	2	3	
1		Q		← queen 1
2				
3	Q			← queen 2

For *n* = 4, There is **solution** to place 4 queens in 4 × 4 chessboard. the four-queens problem solved by the backtracking technique.

	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

For *n* = 8, There is **solution** to place 8 queens in 8 × 8 chessboard.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4			Q					
5	Q							
6							Q	
7					Q			
8		Q						



State space tree for 4-queen problem. Similar to other queen problem also.

```
Algorithm place(k,I)
{
for j := 1 to k-1 do
if(x[j]=I) or(abs(x[j]-I)=abs(j-k))) then return false;
return true;
```

```
}
Algorithm Nqueens(k,n)
{
for I:= 1 to n do
{
if( place(k,I) then
{
x[k]:= I;
if(k=n) then write(x[1:n]);
else
Nqueens(k+1,n) } } }
```

Hamiltonian Circuit Problem

As our next example, let us consider the problem of finding a Hamiltonian circuit in the graph in followingFigure

Without loss of generality, we can assume that if a Hamiltonian circuit exists,it starts at vertex *a*. Accordingly, we make vertex *a* the root of the state-space The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian circuit to be constructed. Using the alphabet order to break the three-way tie among the vertices adjacent to *a,* we select vertex *b*. From *b*, the algorithm proceeds to *c*, then to *d*, then to *e*, and finally to *f*, which proves to be a dead end. So the algorithm backtracks from *f* to *e*, then to *d*, and then to *c*, which provides the first alternative for the algorithm to pursue. Going from *c* to *e* eventually proves useless, and the algorithm has to backtrack from *e* to *c* and then to *b*. From there, it goes to the vertices *f* , *e*, *c*, and *d*, from which it can legitimately return to *a*, yielding the Hamiltonian circuit *a*, *b*,*f* , *e*, *c*, *d*, *a*. If we wanted to find another Hamiltonian circuit, we could continue
this process by backtracking from the leaf of the solution found.

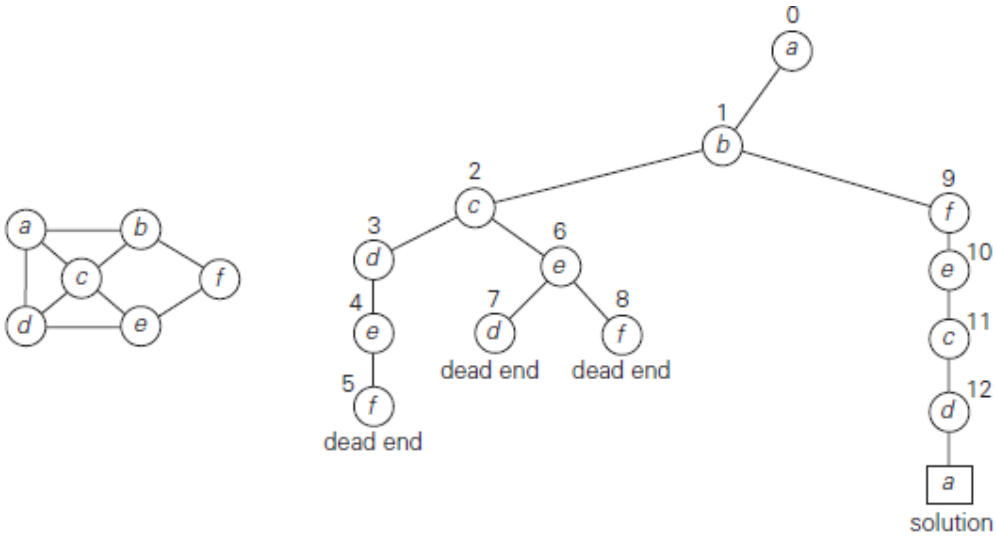


FIGURE 12.3 (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

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 smallas possible. We introduced this problem in Section 3.4, where we solved it by exhaustive search. Recall that 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.

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

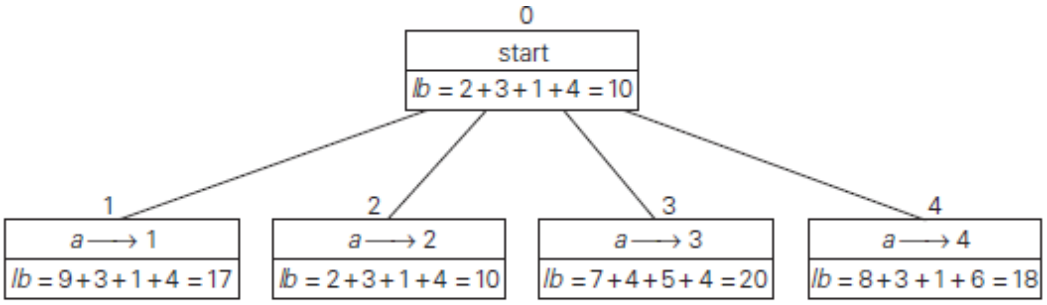


FIGURE 12.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

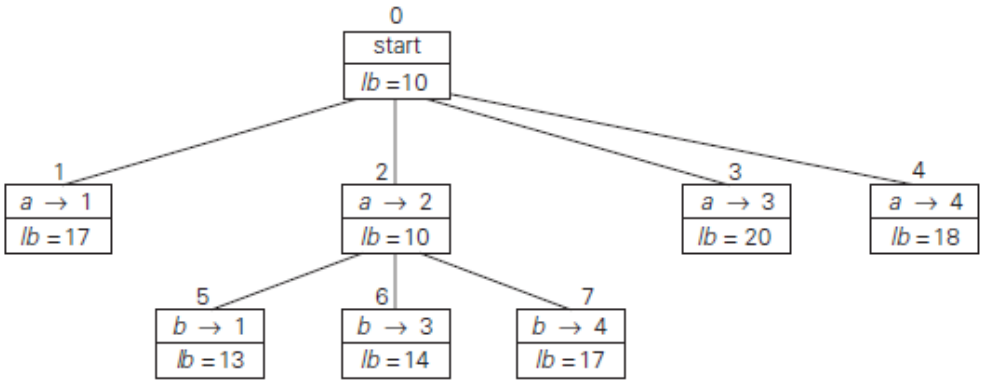


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

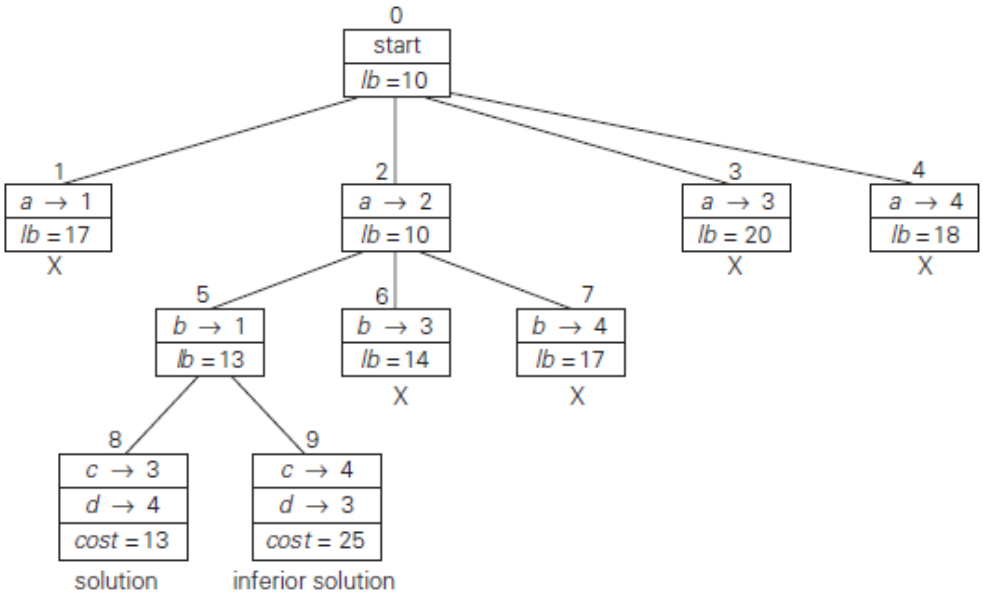


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

How can we find a lower bound on the cost of an optimal selection withoutactually 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 be9 + 3 + 1+ 4 = 17.

Knapsack Problem

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 v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

Consider the following problem,

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

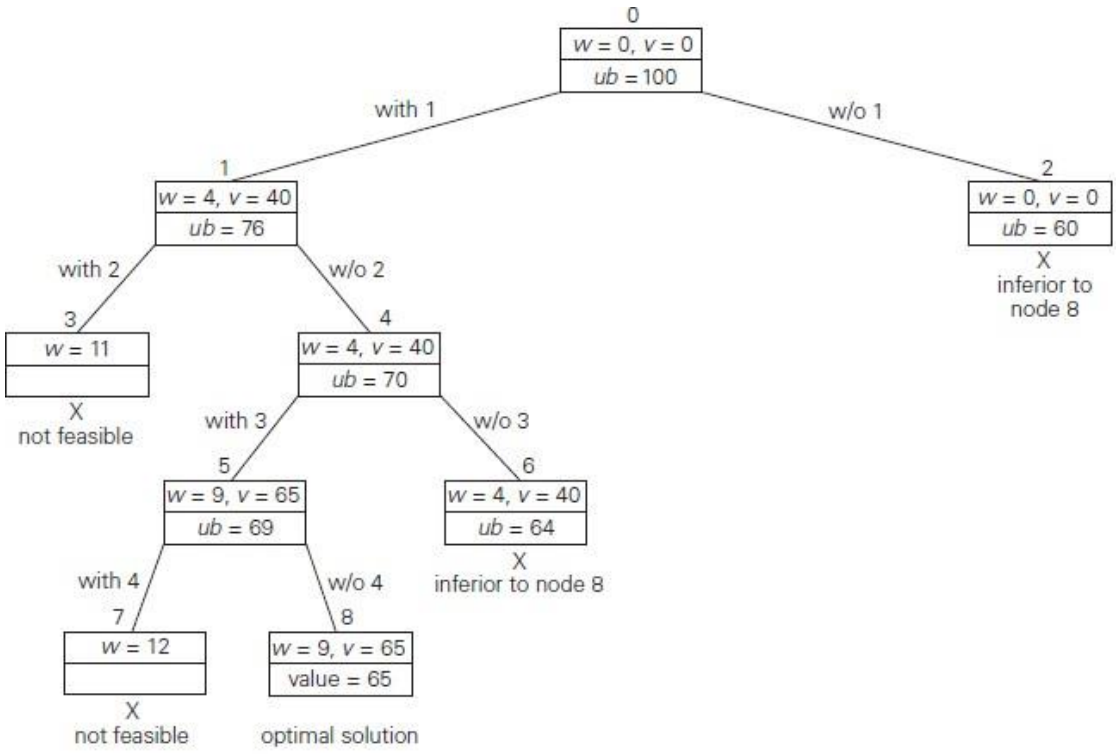


FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

The solution is the items 1 and 3 are included with maximum profit value of 65

Traveling Salesman Problem

For each city *i*, 1≤ *i* ≤ *n*, find the sum *si* of the distances from city *i* to the two nearest cities; compute the sum *s* of these *n* numbers, divide the result by 2, and, if all the distances are integers,round up the result to the nearest integer:

$$lb = \lceil s/2 \rceil \quad \textbf{(12.2)}$$

For example, for the instance in Figure 12.9a, formula (12.2) yields $lb = \lceil (1+ 3) + (3 + 6) + (1+ 2) + (3 + 4) + (2 + 3) \rceil /2 \rceil = 14$.

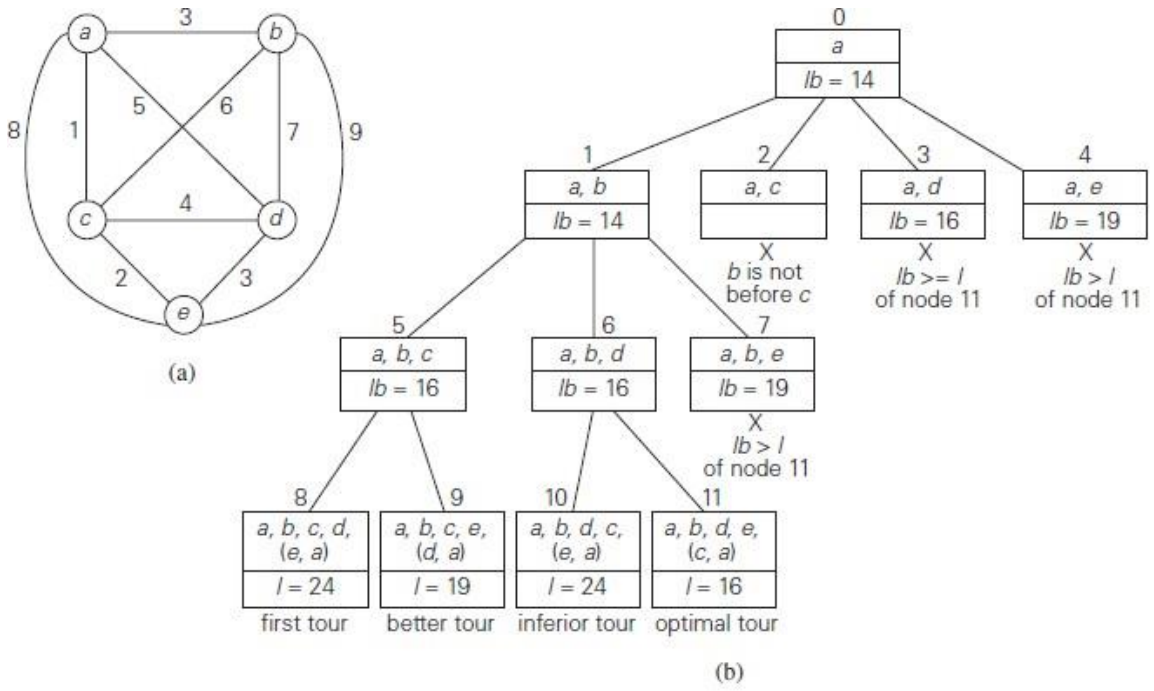


FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

To reduce the amount of potential work, we take advantage of twoobservations made in Section 3.4. First, without loss of generality, we can consider only tours that start at *a*. Second, because our graph is undirected, we can generate only tours in which *b* is visited before *c*. In addition, after visiting $n - 1 = 4$ cities,a tour has no choice but to visit the remaining unvisited city and return to the starting one.

The optimal solution is the tour with a,b,d,e,c,a with tour length of 16.