MODULE-5
BACKTRACKING

Contents

Backtracking

Some problems can be solved, by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.

Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm **backtracks** to replace the last component of the partially constructed solution with its next option.

It is convenient to implement this kind of processing by constructing a tree of choices being made, called the **state-space tree**. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second

component, and so on. A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called **non-promising**. Leaves represent either non-promising dead ends or complete solutions found by the algorithm.

In the majority of cases, a statespace tree for a backtracking algorithm is constructed in the manner of depth-first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

General method

In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$, where the x_i are chosen from some finite set S_i.

Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P. The *brute force approach* would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (sometimes called

bounding functions) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realized that the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

Common examples of explicit constraints are

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} & S_i & = & \{0, 1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. □

General Algorithm (Recursive)

```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k − 1 values
// x[1], x[2], ..., x[k − 1] of the solution vector
// x[1 : n] have been assigned. x[ ] and n are global.
{
    for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
    {
        if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
        {
            if (x[1], x[2], ..., x[k] is a path to an answer node)
                then write (x[1 : k]);
            if (k < n) then Backtrack(k + 1);
        }
    }
}
```

General Algorithm (Iterative)

```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1 : n] and printed
// as soon as they are determined.
{
    k := 1;
    while (k ≠ 0) do
    {
        if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
            x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
        {
                if (x[1], ..., x[k] is a path to an answer node)
                    then write (x[1 : k]);
                k := k + 1; // Consider the next set.
        }
        else k := k − 1; // Backtrack to the previous set.
    }
}
```

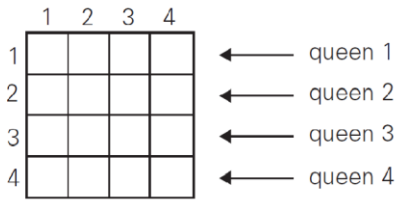
General Algorithm for backtracking (From textbook T1)

```
ALGORITHM Backtrack(X[1..i])
    //Gives a template of a generic backtracking algorithm
    //Input: X[1..i] specifies first i promising components of a solution
    //Output: All the tuples representing the problem's solutions
    if X[1..i] is a solution write X[1..i]
    else      //see Problem 9 in this section's exercises
        for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
            X[i + 1] ← x
            Backtrack(X[1..i + 1])
```

N-Queens problem

The 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.

So let us consider the **four-queens problem** and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in figure.



We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row . Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in figure.

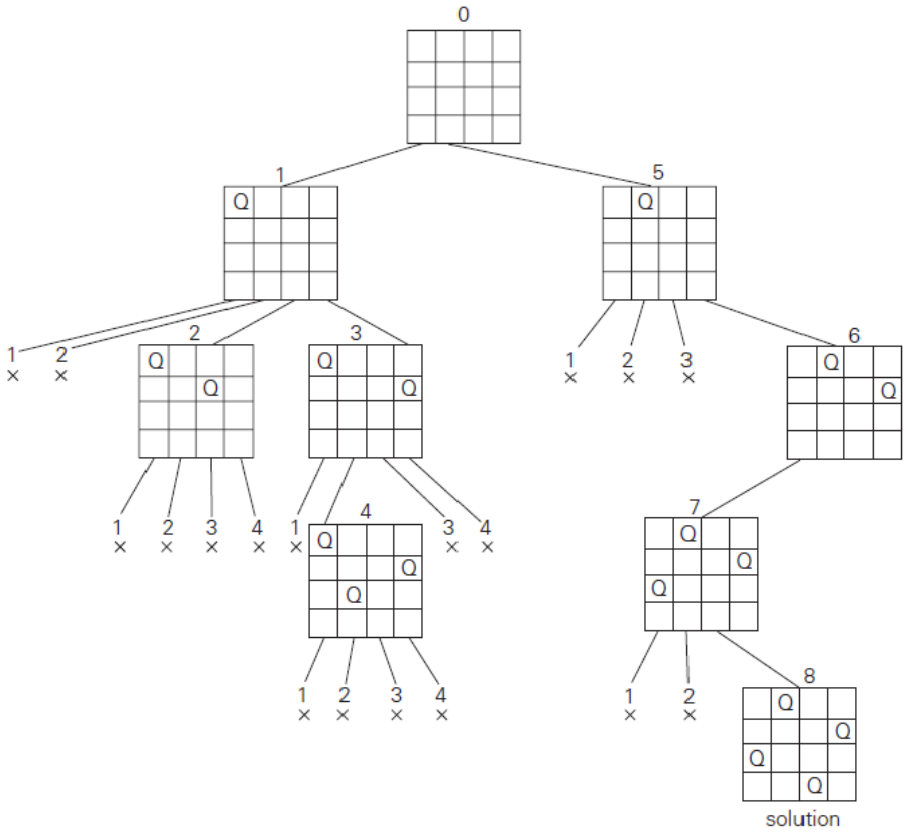


Figure: State-space tree of solving the four-queens problem by backtracking.
× denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

If other solutions need to be found, the algorithm can simply resume its operations at the leaf at which it stopped. Alternatively, we can use the board's symmetry for this purpose.

Finally, it should be pointed out that a single solution to the n-queens problem for any $n \geq 4$ can be found in **linear time**.

Note: The algorithm NQueens() is not in the syllabus. It is given here for interested learners. The algorithm is referred from textbook T2.

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    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);
        }
    }
}
```

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Sum of subsets problem

Problem definition: Find a subset of a given set A = {a_1, . . . , a_n } of n positive integers whose sum is equal to a given positive integer d.

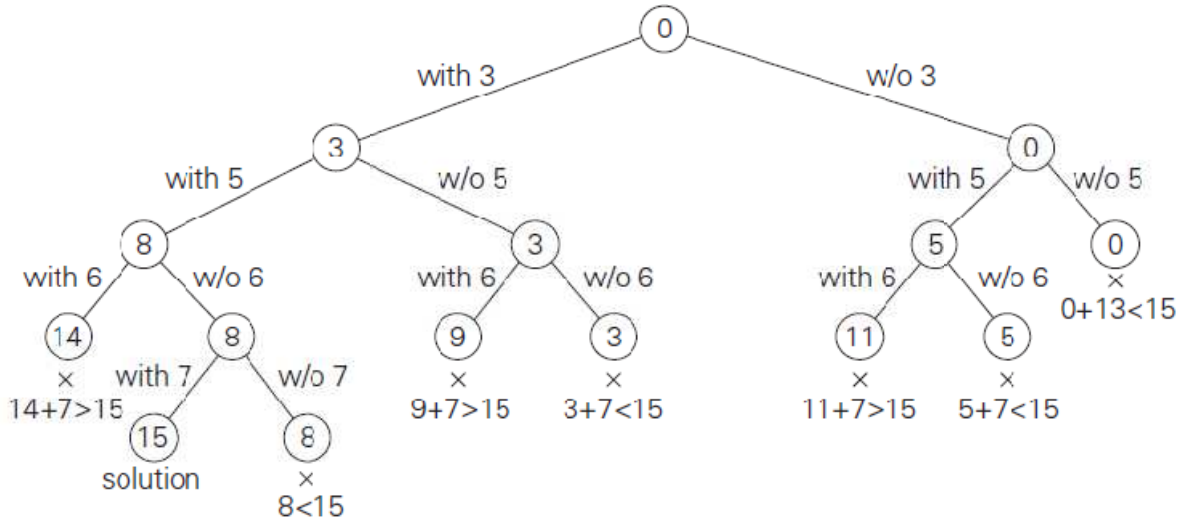
For example, for A = {1, 2, 5, 6, 8} and d = 9, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that

$$a_1 < a_2 < \ldots < a_n.$$

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

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.



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 a_1 in a set being sought.

Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the i^{th} 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:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

Example: Apply backtracking to solve the following instance of the subset sum problem: A = {1, 3, 4, 5} and d = 11.

Graph coloring

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the *m-colorability decision* problem Note that if d is the degree of the given graph, then it can be colored with $d + 1$

colors. The *m-colorability optimization* problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the *chromatic number* of the graph. For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

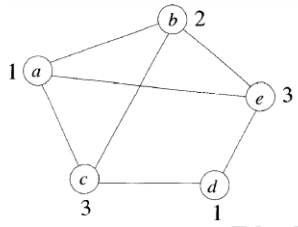


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other. A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question: given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge. Figure 7.12 shows a map with five regions and its corresponding graph. This map requires four colors. For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section we consider not only graphs that are produced from maps but all graphs. We are interested in determining all the different ways in which a given graph can be colored using at most m colors.

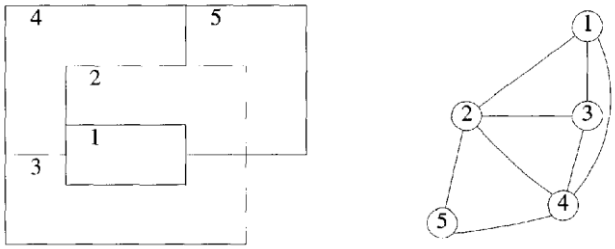


Figure 7.12 A map and its planar graph representation

Suppose we represent a graph by its adjacency matrix $G[1:n, 1:n]$, where $G[i, j] = 1$ if (i, j) is an edge of G, and $G[i, j] = 0$ otherwise. The colors are represented by the integers $1, 2, \ldots, m$ and the solutions are given by the n-tuple $(x_1, \ldots, x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is mColoring (Algorithm 7.7). The underlying state space tree used is a level $n + 1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.

Algorithm 7.7 Finding all m-colorings of a graph

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then     // At most m colors have been
                            // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}

Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

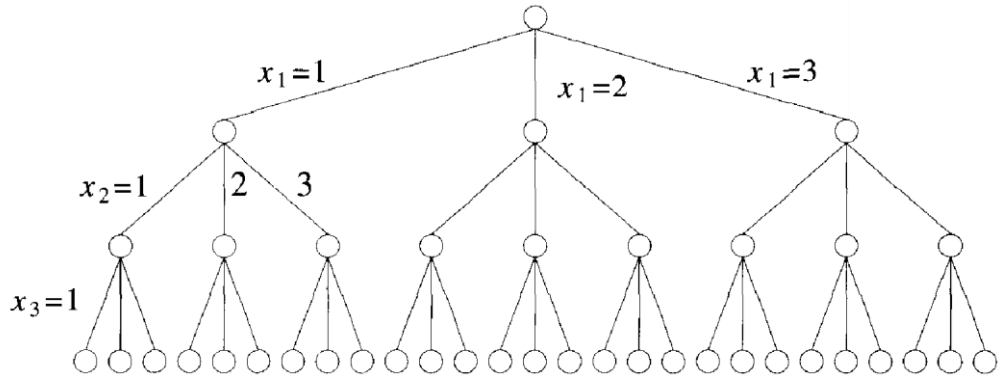


Figure 7.13 State space tree for mColoring when n =3 and m = 3

Function mColoring is begun by first assigning the graph to its adjacency matrix, *setting the array x[ ] to zero*, and then invoking the statement mColoring(1);.

recursive backtracking schema of Algorithm 7.1. Function NextValue (Algorithm 7.8) produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of mColoring repeatedly picks an element from the set of possibilities, assigns it to x_k, and then calls mColoring recursively. For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by mColoring. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.

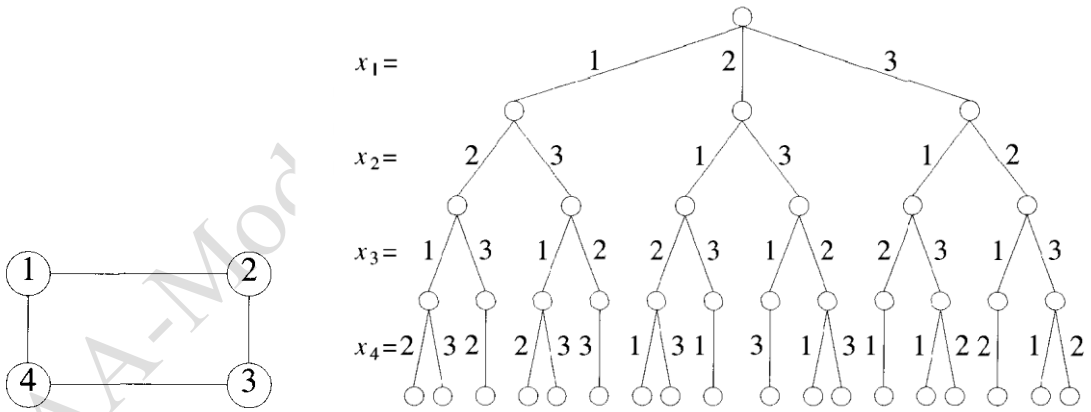


Figure 7.14 A 4-node graph and all possible 3-colorings

Analysis

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m - 1) = O(nm^n)$.

Hamiltonian cycles

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

The graph $G1$ of Figure 7.15 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph $G2$ of Figure 7.15 contains no Hamiltonian cycle. There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.

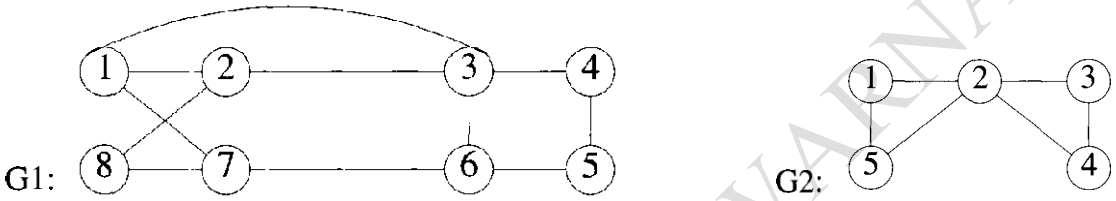


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

Hamiltonian cycle. We now look at a backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

The backtracking solution vector $(x_1, \ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle. Now all we need do is determine how to compute the set of possible vertices for x_k if $x_1, \ldots, x_{k-1}$ have already been chosen. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to x_{k-1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1. We begin by presenting function NextValue(k) which determines a possible next vertex for the proposed cycle.

Using NextValue we can particularize the recursive backtracking schema to find all Hamiltonian cycles (Algorithm 7.10). This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

Recall from the traveling salesperson problem which asked for a tour that has minimum cost. This tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists. If the common edge cost is c, the cost of a tour is cn since there are n edges in a Hamiltonian cycle.

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}

Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                                // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then return;
        }
    } until (false);
}
```

Branch and Bound

Recall that the central idea of backtracking, discussed in the previous section, is to cut off a branch of the problem's stat -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 (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints. 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:

1. 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
2. the **value of the best solution** seen so far

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:

1. The value of the node's bound is not better than the value of the best solution seen so far.
2. The node represents no feasible solutions because the constraints of the problem are already violated.
3. 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 × 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. We will demonstrate how this problem can be solved using the branch-and-bound technique by considering the small instance of the problem. Consider the data given below.

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

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. 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.

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 non-terminated 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**.

We start with the root that corresponds to no elements selected from the cost matrix. 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. See the figure given below.

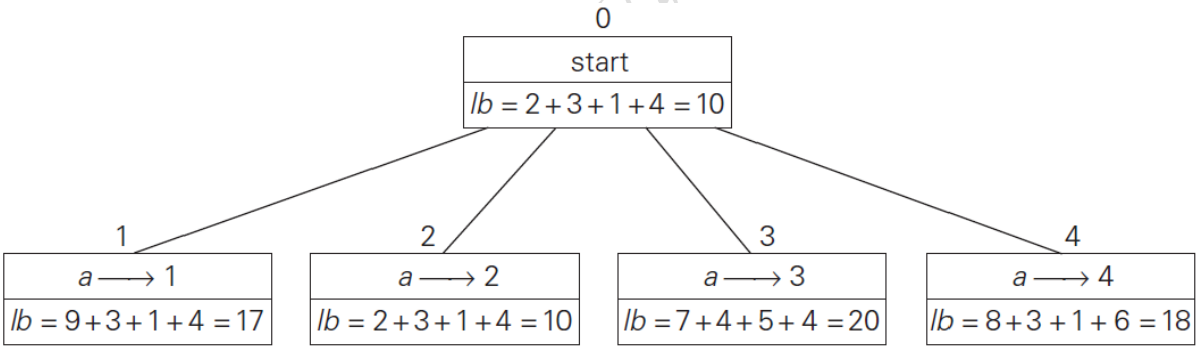


Figure: Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound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

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 lowerbound 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. See the figure given below (Fig 12.7).

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 (Figure 12.7), which corresponds to the feasible solution {a→2, b→1, c→3, d →4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {a→2, b→1, c→4, d →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.)

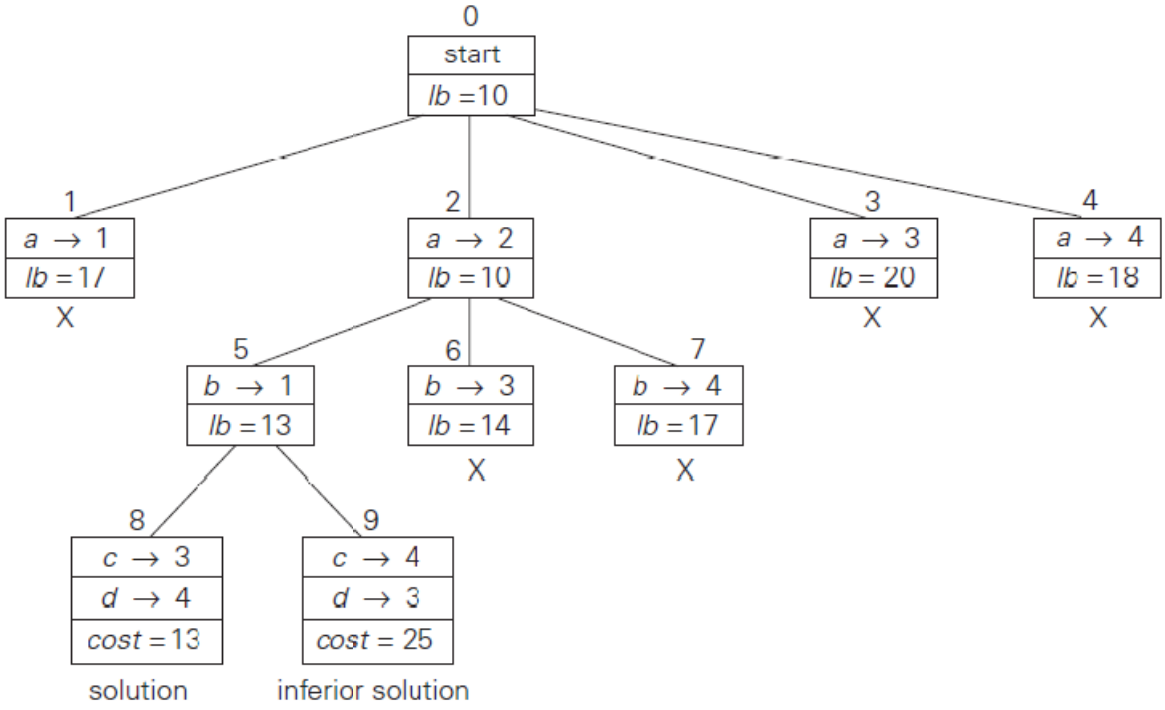


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure 12.7—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.

Travelling Sales Person problem

We will be able to apply the branch-and-bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n.

But there is a less obvious and more informative lower bound for instances with symmetric matrix D, which does not require a lot of work to compute. We can compute a lower bound on the length l of any tour as follows. For each city i, $1 \le i \le n$, find the sum s_i 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:

$$lb = \lceil s/2 \rceil \quad \text{... (1)}$$

For example, for the instance in Figure 2.2a, formula (1) yields

$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound (formula 1) accordingly. For example, for all the Hamiltonian circuits of the graph in Figure 2.2a that must include edge (a, d), we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges (a, d) and (d, a):

$$\lceil [(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2 \rceil = 16.$$

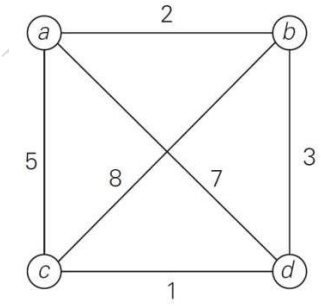
We now apply the branch-and-bound algorithm, with the bounding function given by formula-1, to find the shortest Hamiltonian circuit for the graph in Figure 2.2a.

To reduce the amount of potential work, we take advantage of two observations.

1. First, without loss of generality, we can consider only tours that start at a.
2. Second, because our graph is undirected, we can generate only tours in which b is visited before c. (Refer ***Note*** at the end of section 2.2 for more details)

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 state-space tree tracing the algorithm's application is given in Figure 2.2b.

Note: An inspection of graph with 4 nodes (figure given below) reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, b and c, and then consider only permutations in which b precedes c. (This trick implicitly defines a tour's direction.)



Tour	Length	
a —> b —> c —> d —> a	l = 2 + 8 + 1 + 7 = 18	
a —> b —> d —> c —> a	l = 2 + 3 + 1 + 5 = 11	optimal
a —> c —> b —> d —> a	l = 5 + 8 + 3 + 7 = 23	
a —> c —> d —> b —> a	l = 5 + 1 + 3 + 2 = 11	optimal
a —> d —> b —> c —> a	l = 7 + 3 + 8 + 5 = 23	
a —> d —> c —> b —> a	l = 7 + 1 + 8 + 2 = 18	

Figure: Solution to a small instance of the traveling salesman problem by exhaustive search.

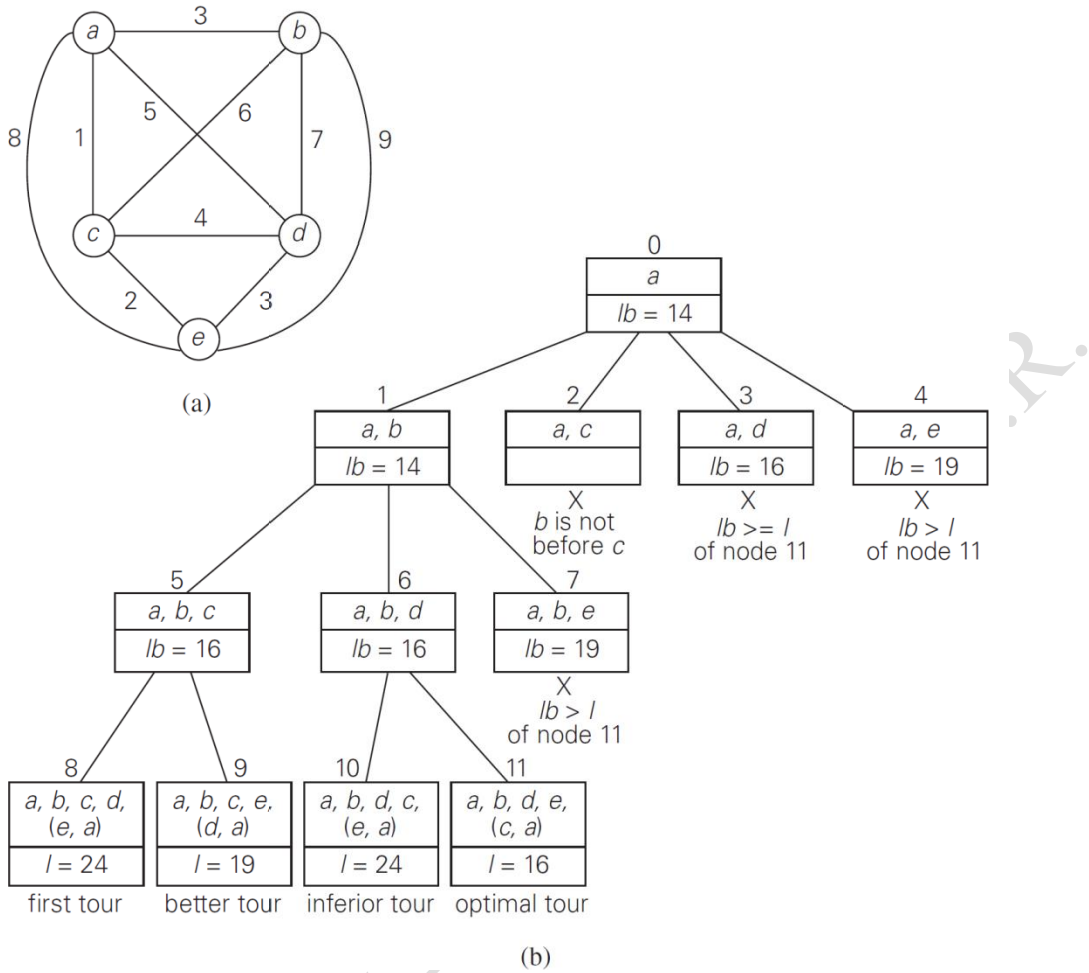


Figure 2.2 (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

Discussion

The strengths and weaknesses of backtracking are applicable to branch-and-bound as well. The state-space tree technique enables us to solve many large instances of difficult combinatorial problems. As a rule, however, it is virtually impossible to predict which instances will be solvable in a realistic amount of time and which will not.

In contrast to backtracking, solving a problem by branch-and-bound has both the challenge and opportunity of choosing the order of node generation and finding a good bounding function. Though the best-first rule we used above is a sensible approach, it may or may not lead to a solution faster than other strategies. (Artificial intelligence researchers are particularly interested in different strategies for developing state-space trees.)

Finding a good bounding function is usually not a simple task. On the one hand, we want this function to be easy to compute. On the other hand, it cannot be too simplistic - otherwise, it would fail in its principal task to prune as many branches of a state-space tree as soon as possible. Striking a proper balance between these two competing requirements may require intensive experimentation with a wide variety of instances of the problem in question.

0/1 Knapsack problem

Note: For this topic as per the syllabus both textbooks T1 & T2 are suggested. Here we discuss the concepts from T1 first and then that of from T2.

Topic form T1 (Levitin)

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. Given n items of known weights w_i and values v_i , i = 1, 2, . . . , n, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

$$\sum_{1 \le i \le n} w_i x_i \le W \text{ and } \sum_{1 \le i \le n} p_i x_i \text{ is maximized, where } x_i = 0 \text{ or } 1$$

It is convenient to order the items of a given instance in descending order by their value-to-weight ratios.

$$v_1/w_1 \ge v_2/w_2 \ge \cdots \ge v_n/w_n$$

Each node on the i^{th} level of state space tree, $0 \le i \le n$, 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.

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

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

Example: Consider the following problem. The items are already ordered in descending order of their value-to-weight ratios.

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.

Let us apply the branch-and-bound algorithm. At the root of the state-space tree (see Figure 12.8), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal to 0. The value of the upper bound is 100.

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 (Due to better ub), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes.

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.

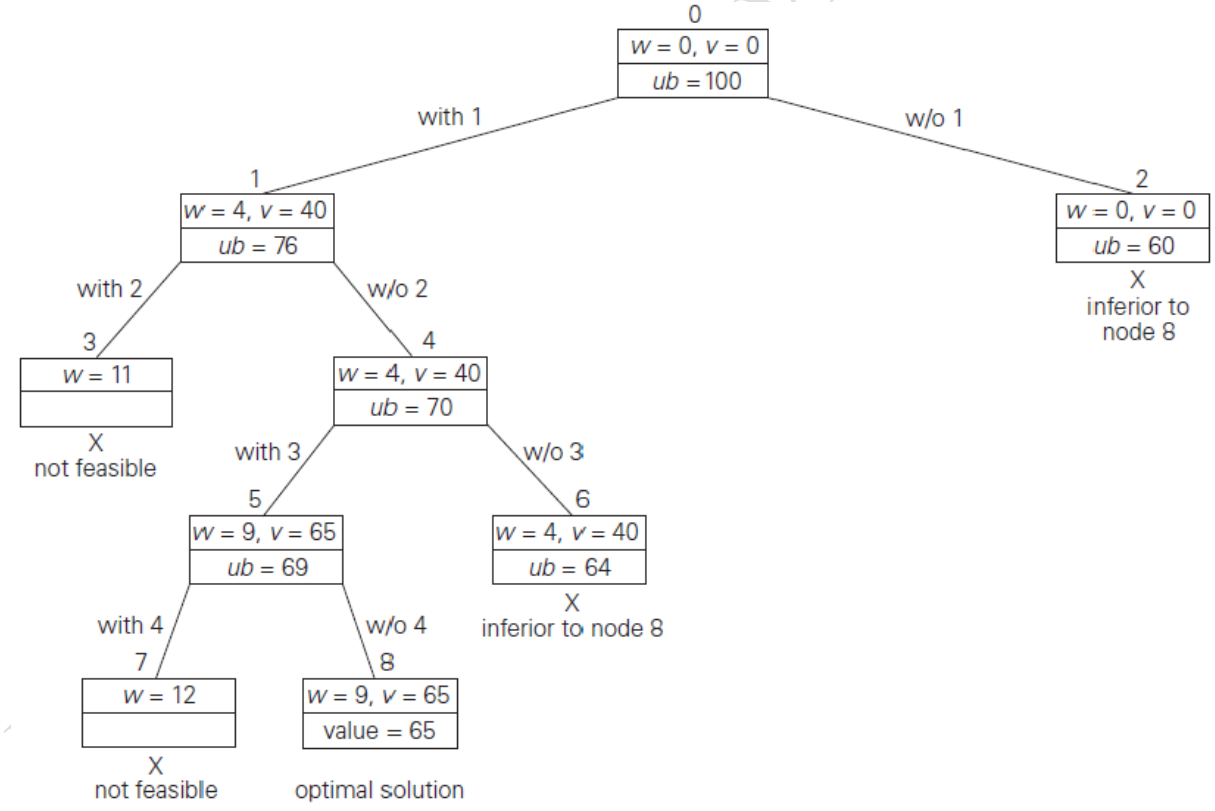


FIGURE 12.8 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic. Typically, internal nodes of a state-space tree do not define a point of the problem's search space, because some of the solution's components remain undefined. (See, for example, the branch-and-bound tree for the assignment problem discussed in the

preceding subsection.) For the knapsack problem, however, every node of the tree represents a subset of the items given. We can use this fact to update the information about the best subset seen so far after generating each new node in the tree. If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value 65 of node 5.

Concepts form textbook T2 (Horowitz)

Let us understand some of the **terminologies used in backtracking & branch and bound**.

→ ***Live node*** - a node which has been generated and all of whose children are not yet been generated.

→ ***E-node*** - is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

→ ***Dead node*** - a node that is either not to be expanded further, or for which all of its children have been generated

→ ***Bounding Function*** - will be used to kill live nodes without generating all their children.

→ ***Backtracking*** - is depth first node generation with bounding functions.

→ ***Branch-And-Bound*** is a method in which E-node remains E-node until it is dead.

→ ***Breadth-First-Search:*** Branch-and Bound with each new node placed in a queue. The front of the queen becomes the new E-node.

→ ***Depth-Search (D-Search):*** New nodes are placed in to a stack. The last node added is the first to be explored.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function.

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(\cdot)$ such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any nondecreasing function.

By using $f(\cdot) \not\equiv 0$, we can force the search algorithm to favor a node z close to the root over a node w which is many levels below z. This would reduce the possibility of deep and fruitless searches into the tree.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with bounding functions is called an LC branch-and-bound search.

0/1 Knapsack problem - Branch and Bound based solution

As the technique discussed here is applicable for minimization problems, let us convert the knapsack problem (maximizing the profit) into minimization problem by negating the objective function

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i \quad \text{subject to } \sum_{i=1}^{n} w_i x_i \le m \qquad x_i = 0 \text{ or } 1, \quad 1 \le i \le n$$

Every leaf node in the state space tree representing an assignment for which $\sum_{1 \le i \le n} w_i x_i \le m$ is an answer (or solution) node. All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution, we need to define $c(x) = -\sum_{1 < i < n} p_i x_i$ for every answer node x. The cost $c(x) = \infty$ for infeasible leaf nodes. For nonleaf nodes, $c(x)$ is recursively defined to be min $\{c(lchild(x)), c(rchild(x))\}$.

We now need two functions $\hat{c}(x)$ and $u(x)$ such that $\hat{c}(x) \le c(x) \le u(x)$ for every node x. The cost $\hat{c}(\cdot)$ and $u(\cdot)$ satisfying this requirement may be obtained as follows. Let x be a node at level j, $1 \le j \le n+1$. At node x assignments have already been made to x_i, $1 \le i < j$. The cost of these assignments is $-\sum_{1 \le i < j} p_i x_i$. So, $c(x) \le -\sum_{1 \le i < j} p_i x_i$ and we may use $u(x) = -\sum_{1 \le i < j} p_i x_i$. If $q = -\sum_{1 \le i < j} p_i x_i$, then an improved upper bound function $u(x)$ is $u(x) = \text{UBound}(q, \sum_{1 \le i < j} w_i x_i, j-1, m)$, where UBound is defined in Algorithm 8.2.

Algorithm 8.2 Function $u(\cdot)$ for knapsack problem

```
Algorithm UBound(cp, cw, k, m)
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
//                    w[i] and p[i] are respectively
// the weight and profit of the ith object.
{
     b := cp; c := cw;
     for i := k + 1 to n do
     {
          if (c + w[i] ≤ m) then
          {
               c := c + w[i]; b := b - p[i];
          }
     }
     return b;
}
```

LC (Least Cost) Branch and Bound solution

To use LCBB to solve the knapsack problem, we need to specify (1) the structure of nodes in the state space tree being searched, (2) how to generate the children of a given node, (3) how to recognize a solution node, and (4) a representation of the list of live nodes and a mechanism for adding a node into the list as well as identifying the least-cost node. The node structure needed depends on which of the two formulations for the state space tree is being used. Let us continue with a fixed size tuple formulation. Each node x that is generated and put onto the list of live nodes must have a *parent* field. In addition, as noted in Example 8.2, each node should have a one bit *tag* field. This field is needed to output the x_i values corresponding to an optimal solution. To generate x's children, we need to know the level of node x in the state space tree. For this we shall use a field *level*. The left child of x is chosen by setting $x_{level(x)} = 1$ and the right child by setting $x_{level(x)} = 0$. To determine the feasibility of the left child, we need to know the amount of knapsack space available at node x. This can be determined either by following the path from node x to the root or by explicitly retaining this value in the node. Say we choose to retain this value in a field *cu* (capacity unused). The evaluation of $\hat{c}(x)$ and $u(x)$ requires knowledge of the profit $\sum_{1 \le i < level(x)} p_i x_i$ earned by the filling corresponding to node x. This can be computed by following the path from x to the root. Alternatively, this value can be explicitly retained in a field *pe*. Finally, in order to determine the live node with least $\hat{c}$ value or to insert nodes properly into the list of live nodes, we need to know $\hat{c}(x)$. Again, we have a choice. The value $\hat{c}(x)$ may be stored explicitly in a field *ub* or may be computed when needed. Assuming all information is kept explicitly, we need nodes with six fields each: *parent*, *level*, *tag*, *cu*, *pe*, and *ub*.

Using this six-field node structure, the children of any live node x can be easily determined. The left child y is feasible iff $cu(x) \ge w_{level(x)}$. In this case, $parent(y) = x$, $level(y) = level(x) + 1$, $cu(y) = cu(x) - w_{level(x)}$, $pe(y) = pe(x) + p_{level(x)}$, $tag(y) = 1$, and $ub(y) = ub(x)$. The right child can be generated similarly. Solution nodes are easily recognized too. Node x is a solution node iff $level(x) = n + 1$.

We are now left with the task of specifying the representation of the list of live nodes. The functions we wish to perform on this list are (1) test if the list is empty, (2) add nodes, and (3) delete a node with least *ub*. We have seen a data structure that allows us to perform these three functions efficiently: a min-heap. If there are m live nodes, then function (1) can be carried out in $\Theta(1)$ time, whereas functions (2) and (3) require only $O(\log m)$ time.

Example 8.2 [LCBB] Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

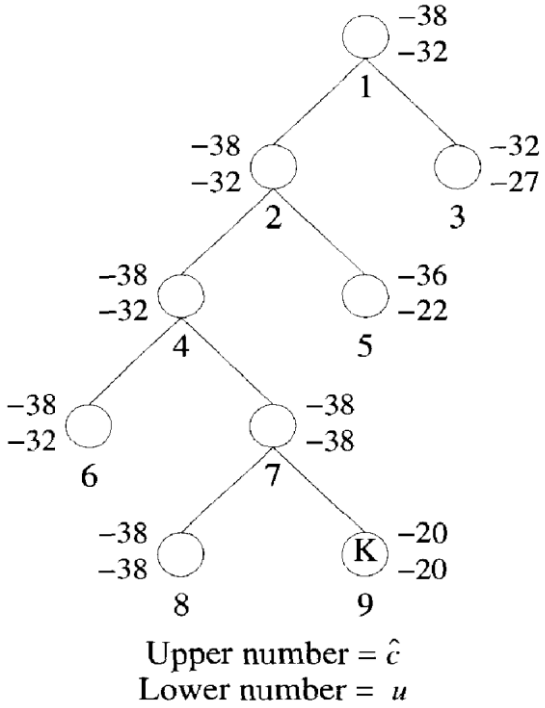


Figure 8.8 LC branch-and-bound tree for Example 8.2

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value UBound$(0, 0, 0, 15)$. UBound scans through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned. In Function UBound, c and b start with a value of zero. For $i = 1, 2$, and 3, c gets incremented by $2, 4$, and 6, respectively. The variable b also gets decremented by $10, 10$, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, LCBB sets $ans = 0$ and $upper = -32$ (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node

4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and *upper* is updated to -38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then *upper* is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > upper$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq upper$ and the search terminates with node 8 the answer node. At this time, the value -38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i = upper$. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, *tag*. The sequence of *tag* bits from the answer node to the root give the x_i values. Thus, we have $tag(2) = tag(4) = tag(6) = tag(8) = 1$ and $tag(3) = tag(5) = tag(7) = tag(9) = 0$. The *tag* sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1, x_3 = 0, x_2 = 1$, and $x_1 = 1$. □

FIFO Branch and Bound solution

:e to branch-and-bound strategies. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (**F**irst **I**n **F**irst **O**ut) search as the list of live nodes is a first-in-first-out list (or queue).

Example 8.3 Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in Example 8.2. Initially the root node, node 1 of Figure 8.9, is the E-node and the queue of live nodes is empty. Since this is not a solution node, *upper* is initialized to $u(1) = -32$.

We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of *upper* remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue. Node 3, the next E-node, is expanded. Its children nodes are generated. Node 6 gets added to the queue. Node 7 is immediately killed as $\hat{c}(7) > upper$. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then *upper* is updated to $u(9) = -38$. Nodes 5 and 6 are the next two nodes to become E-nodes. Neither is expanded as for each, $\hat{c}() > upper$. Node 8 is the next E-node. Nodes 10 and 11 are generated. Node 10 is infeasible and so killed. Node 11 has $\hat{c}(11) > upper$ and so is also killed. Node 9 is expanded next. When node 12 is generated, *upper* and *ans* are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as $\hat{c}(13) > upper$. The only remaining live node is node 12. It has no children and the search terminates. The value of *upper* and the path from node 12 to the root is output. As in the case of Example 8.2, additional information is needed to determine the x_i values on this path. □

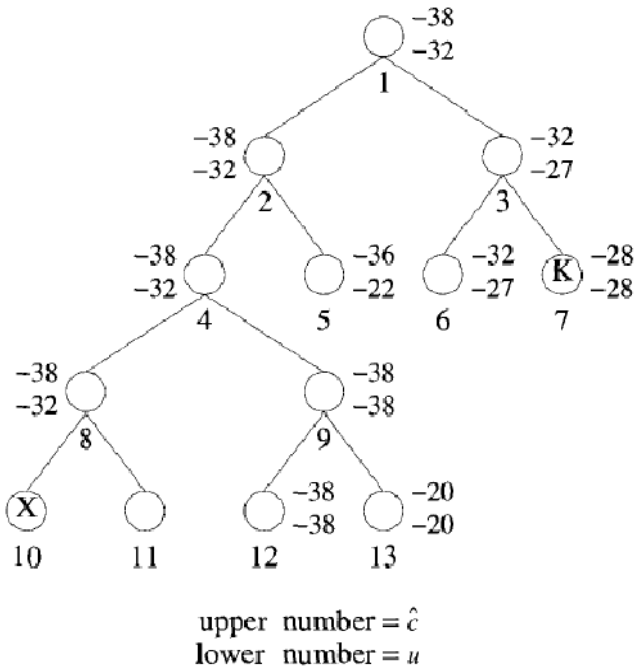


Figure 8.9 FIFO branch-and-bound tree for Example 8.3

Conclusion

At first we may be tempted to discard FIFOBB in favor of LCBB in solving knapsack. Our intuition leads us to believe that LCBB will examine fewer nodes in its quest for an optimal solution. However, we should keep in mind that insertions into and deletions form a heap are far more expensive (proportional to the logarithm of the heap size) than the corresponding operations on a queue ($\Theta(1)$). Consequently, the work done for each E-node is more in LCBB than in FIFOBB. Unless LCBB uses far fewer E-nodes than FIFOBB, FIFOBB will outperform (in terms of real computation time) LCBB.

NP-Complete and NP-Hard problems

Basic concepts

For many of the problems we know and study, the best algorithms for their solution have computing times can be clustered into two groups;

1. Solutions are bounded by the **polynomial**- Examples include Binary search O(log n), Linear search O(n), sorting algorithms like merge sort O(n log n), Bubble sort $O(n^2)$ & matrix multiplication $O(n^3)$ or in general $O(n^k)$ where k is a constant.
2. Solutions are bounded by a non-polynomial - Examples include travelling salesman problem $O(n^2 2^n)$ & knapsack problem $O(2^{n/2})$. As the time increases exponentially, even moderate size problems cannot be solved.

So far, no one has been able to device an algorithm which is bounded by the polynomial for the problems belonging to the non-polynomial. However impossibility of such an algorithm is not proved.

Non deterministic algorithms

We also need the idea of two models of computer (Turing machine): deterministic and non-deterministic. A deterministic computer is the regular computer we always thinking of; a non-deterministic computer is one that is just like we're used to except that is has unlimited parallelism, so that any time you come to a branch, you spawn a new "process" and examine both sides.

When the result of every operation is uniquely defined then it is called **deterministic algorithm**.

When the outcome is not uniquely defined but is limited to a specific set of possibilities, we call it **non deterministic** algorithm.

We use new statements to specify such **non deterministic** h algorithms.

- **choice(S)** - arbitrarily choose one of the elements of set S
- **failure** - signals an unsuccessful completion
- **success** - signals a successful completion

The assignment ***X = choice(1:n)*** could result in X being assigned any value from the integer ***range[1..n]***. There is no rule specifying how this value is chosen.

"The nondeterministic algorithms terminates unsuccessfully iff there is no set of choices which leads to the successful signal".

Example-1: Searching an element **x** in a given set of elements A(1:n). We are required to determine an index j such that A(j) = x or j = 0 if x is not present.

```
j := choice(1:n)
if A(j) = x then print(j); success endif
```

print('0'); failure

Example-2: Checking whether n integers are sorted or not

```
procedure NSORT(A,n);
//sort n positive integers//
var integer A(n), B(n), n, i, j;
begin
        B := 0; //B is initialized to zero//
        for i := 1 to n do
        begin
                j := choice(1:n);
                if B(j) <> 0 then failure;
                B(j) := A(j);
        end;

        for i := 1 to n-1 do //verify order//
                if B(i) > B(i+1) then failure;
        print(B);
        success;
end.
```

"A nondeterministic machine does not make any copies of an algorithm every time a choice is to be made. Instead it has the ability to correctly choose an element from the given set".

A deterministic interpretation of the nondeterministic algorithm can be done by making unbounded parallelism in the computation. Each time a choice is to be made, the algorithm makes several copies of itself, one copy is made for each of the possible choices.

Decision vs Optimization algorithms

An optimization problem tries to find an optimal solution.

A decision problem tries to answer a yes/no question. Most of the problems can be specified in decision and optimization versions.

For example, Traveling salesman problem can be stated as two ways

- Optimization - find hamiltonian cycle of minimum weight,
- Decision - is there a hamiltonian cycle of weight ≤ k?

For graph coloring problem,

- Optimization – find the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color
- Decision - whether there exists such a coloring of the graph's vertices with no more than m colors?

Many optimization problems can be recast in to decision problems with the property that the decision algorithm can be solved in polynomial time if and only if the corresponding optimization problem.

P, NP, NP-Complete and NP-Hard classes

NP stands for Non-deterministic Polynomial time.

Definition: P is a set of all decision problems solvable by a deterministic algorithm in polynomial time.

Definition: NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time. This also implies P ⊆ NP

Problems known to be in P are trivially in NP — the nondeterministic machine just never troubles itself to fork another process, and acts just like a deterministic one. One example of a problem not in P but in NP is ***Integer Factorization***.

But there are some problems which are known to be in NP but don't know if they're in P. The traditional example is the decision-problem version of the Travelling Salesman Problem (***decision-TSP***). It's not known whether decision-TSP is in P: there's no known poly-time solution, but there's no proof such a solution doesn't exist.

There are problems that are known to be neither in P nor NP; a simple example is to enumerate all the bit vectors of length n. No matter what, that takes 2^n steps.

Now, one more concept: given decision problems P and Q, if an algorithm can transform a solution for P into a solution for Q in polynomial time, it's said that Q is ***poly-time reducible*** (or just reducible) to P.

The most famous unsolved problem in computer science is "whether P=NP or P≠NP? "

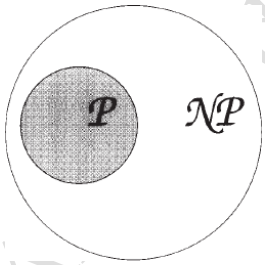


Figure: Commonly believed relationship between P and NP

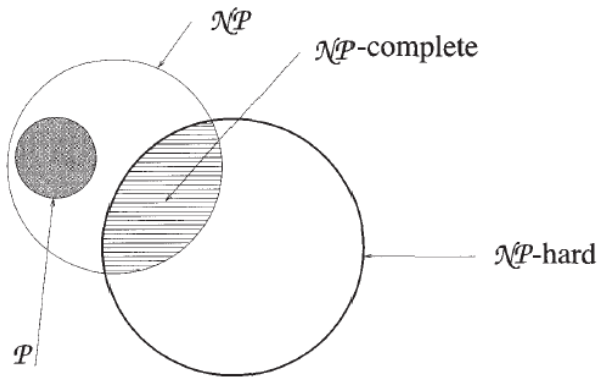


Figure: Commonly believed relationship between P, NP, NP-Complete and NP-hard problems

Definition: A decision problem D is said to be **NP-complete** if:

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

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

NP-Complete problems have the property that it can be solved in polynomial time if all other NP-Complete problems can be solved in polynomial time. i.e if anyone ever finds a poly-time solution to one NP-complete problem, they've automatically got one for ***all*** the NP-complete problems; that will also mean that P=NP.

Example for NP-complete is **CNF-satisfiability problem**. The CNF-satisfiability problem deals with boolean expressions. This is given by Cook in 1971. The CNF-satisfiability problem asks whether or not one can assign values true and false to variables of a given boolean expression in its CNF form to make the entire expression true.

Over the years many problems in NP have been proved to be in P (like Primality Testing). Still, there are many problems in NP not proved to be in P. i.e. the question still remains whether P=NP? NP Complete Problems helps in solving this question. They are a subset of NP problems with the property that all other NP problems can be reduced to any of them in polynomial time. So, they are the hardest problems in NP, in terms of running time. If it can be showed that any NP-Complete problem is in P, then all problems in NP will be in P (because of NP-Complete definition), and hence P=NP=NPC.

NP Hard Problems - These problems need not have any bound on their running time. If any NP-Complete Problem is polynomial time reducible to a problem X, that problem X belongs to NP-Hard class. Hence, all NP-Complete problems are also NP-Hard. In other words if a NP-Hard problem is non-deterministic polynomial time solvable, it is a NP-Complete problem. Example of a NP problem that is not NPC is Halting Problem.

If a NP-Hard problem can be solved in polynomial time then all NP-Complete can be solved in polynomial time.

"All NP-Complete problems are NP-Hard but not all NP-Hard problems are not NP-Complete." NP-Complete problems are subclass of NP-Hard

The more conventional optimization version of Traveling Salesman Problem for finding the shortest route is NP-hard, not strictly NP-complete.

*****