# MODULE V

# BACK TRACKING

## Contents

# 1. BACK TRACKING

## 1.1 n-Queens Problem

- The ***n-queens problem*** is to place $n$ queens on an $n \times n$ 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, and it is easy to see that there is no solution for $n = 2$ and $n = 3$.
- 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 the following figure

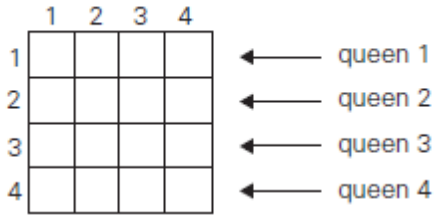


***Figure:*** Board for the four-queens problem.

- 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 1. 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.
- If other solutions need to be found (how many of them are there for the four queens problem?), 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.

- In fact, over the last 150 years mathematicians have discovered several alternative formulas for non attacking positions of *n* queens. Such positions can also be found by applying some general algorithm design strategies
- The state-space tree of this search is shown in Figure

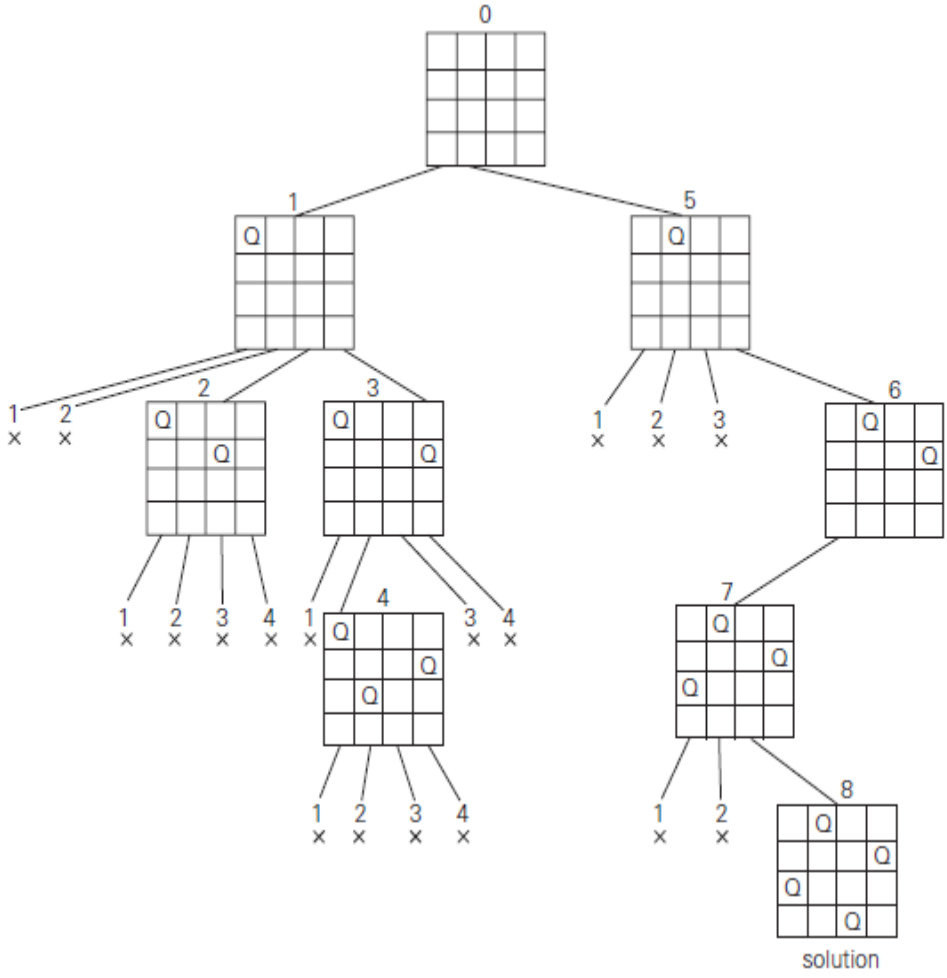


**Figure:** State-space tree of solving the four-queens problem by backtracking.× denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

## 1.2 Subset-Sum Problem

- We consider the ***subset-sum problem***: find a subset of a given set $A = \{a_1, \ldots, a_n\}$ 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 for 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 $a_1$ 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 nonpromising if either of the following two inequalities holds:

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

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

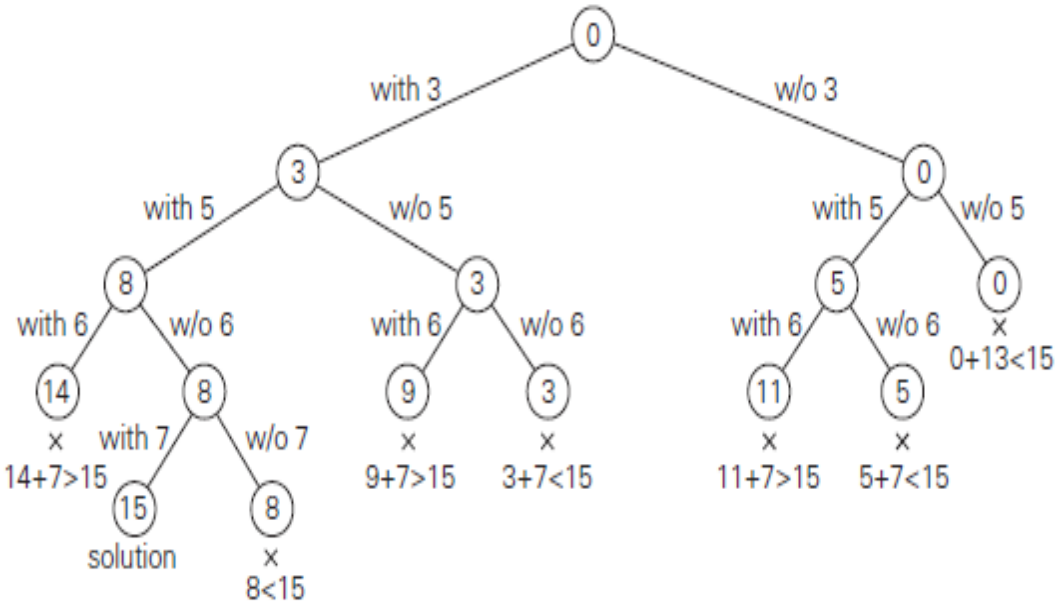


**Figure :** Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

## 2. BRANCH AND BOUND

- The central idea of backtracking, discussed in the previous section, is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution.
- This idea can be strengthened further if we deal with an optimization problem.
- An optimization problem seeks to minimize or maximizesome objective function (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints.
- Note that in the standard terminology of optimization problems, a ***feasible solution*** is a point in the problem's search space that satisfies all the problem's constraints (e.g., a Hamiltonian circuit in the traveling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an ***optimal solution*** is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuablesubset of items that fit the knapsack).
- Compared to backtracking, branch-and-bound requires two additional items:

- ✓ a way to provide, for every node of a state-space tree, a bound on the best value of the objective function1 on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
- ✓ the value of the best solution seen so far

- If this information is available, we can compare a node's bound value with the value of the best solution seen so far.
- If the bound value is not better than the value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is "pruned").
- Indeed, no solution obtained from it can yield a better solution than the one already available.
- This is the principal idea of the branch-and-bound technique.
- In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
  - ✓ The value of the node's bound is not better than the value of the best solution seen so far.
  - ✓ The node represents no feasible solutions because the constraints of the problem are already violated.
  - ✓ The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of thebest solution seen so far and update the latter with the former if the new solution is better.

**2.1 Assignment Problem**

- Let us illustrate the branch-and-bound approach by applying it to the problem of assigning $n$ people to $n$ jobs so that the total cost of the assignment is as small as possible.
- An instance of the assignment problem is specified by an $n \times n$ cost matrix $C$ so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.
- We will demonstrate how this problem can be solved using the branch-and-bound technique by considering the small instance of the problem

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

- We can solve 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.
- For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.
- One more comment is in order before we embark on constructing the problem's state-space tree. It deals with the order in which the tree nodes will be generated.
- 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***.)
- This variation of the strategy is called the ***best-first branch-and-bound***.
- So, returning to the instance of the assignment problem given earlier, we start with the root that corresponds to no elements selected from the cost matrix.
- As we already discussed, the lower-bound value for the root, denoted *lb*, is 10.
- The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a* as shown in figure.

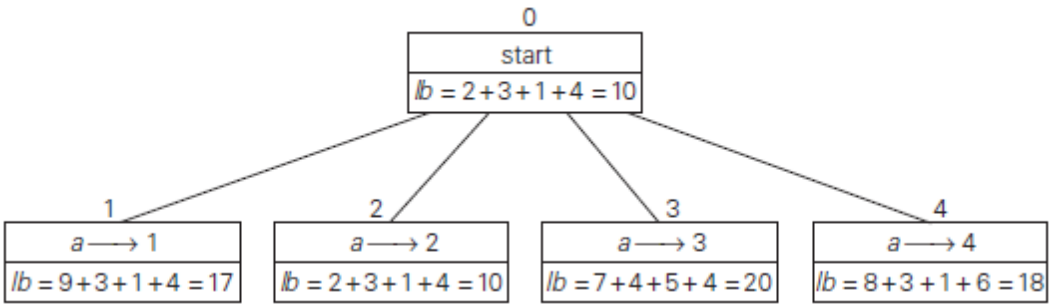

**Figure :** Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person *a* and the lower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

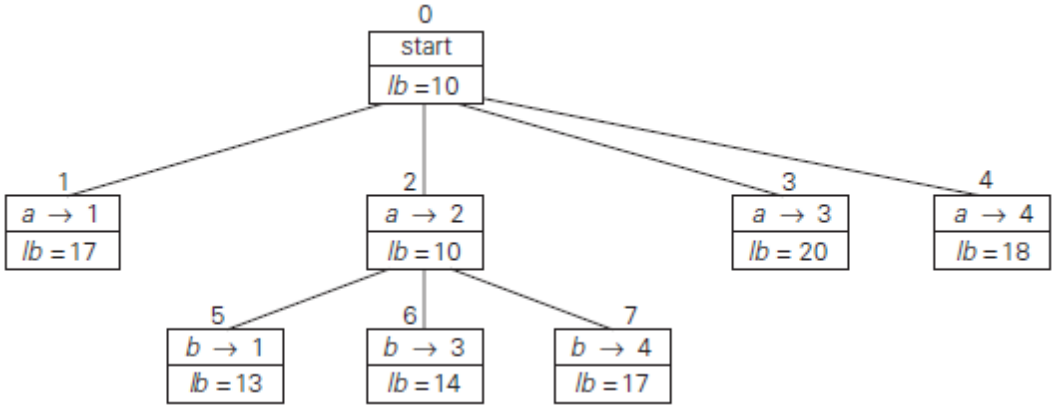


**Figure :** Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

- 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 below), 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.)

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

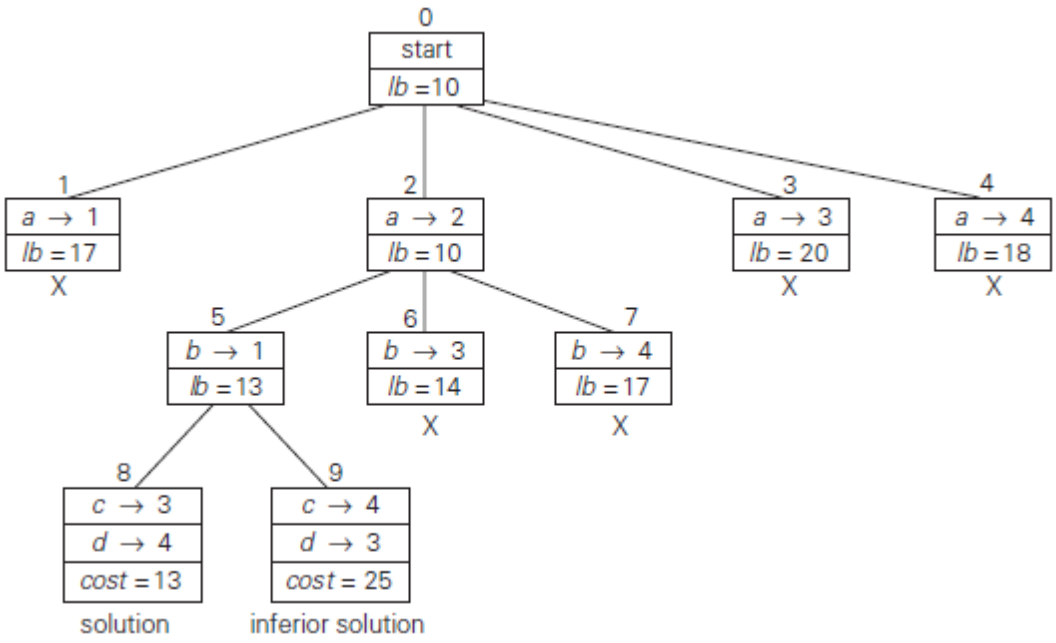


**Figure :** Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

## 2.2 Knapsack Problem

- Here applying the branch-and-bound technique for solving the knapsack problem.
- Problem states that given $n$ items of known weights $w_i$ and values $v_i$ , $i = 1, 2, \ldots, n$, and a knapsack of capacity $W$, to find the most valuable subset of the items that fit in the knapsack.
- It is convenient to order the items of a given instance in descending order by their value-to-weight ratios.
- Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

- It is natural to structure the state-space tree for this problem as a binary tree constructed as follows.
- Each node on the *i*th level of this tree, $0 \leq i \leq n$, represents all the subsets of $n$ items that include a particular selection made from the first $i$ ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.
- 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 $v_{i+1}/w_{i+1}$:

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

- As a specific example, let us apply the branch-and-bound algorithm to the instance of the knapsack problem. (We reorder the items in descending order of their value-to-weight ratios, though.)

| 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.

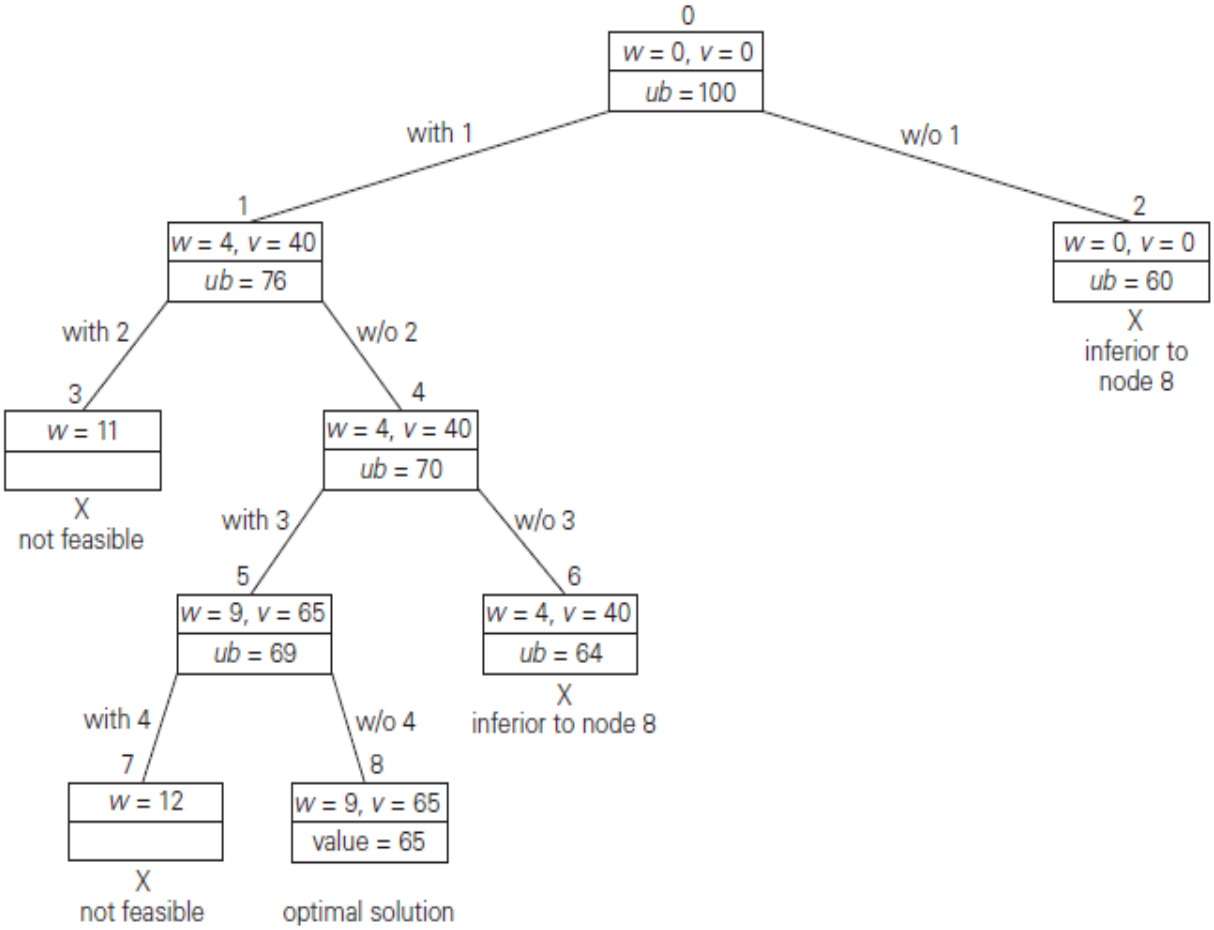


**Figure :** State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

- At the root of the state-space tree (see above figure), 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 computed by formula (12.1) 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, 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.

### 2.3 Traveling SalesMan 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.
- It is not difficult to show that we can compute a lower bound on the length *l* of any tour as follows. For each city *i*, 1≤ *i* ≤ *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:

$$lb = \lceil s/2 \rceil.$$

- For example, for the instance in Figure (a), formula 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 accordingly.
- For example, for all the Hamiltonian circuits of the graph in Figure (a) 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, to find the shortest Hamiltonian circuit for the graph in Figure (a).

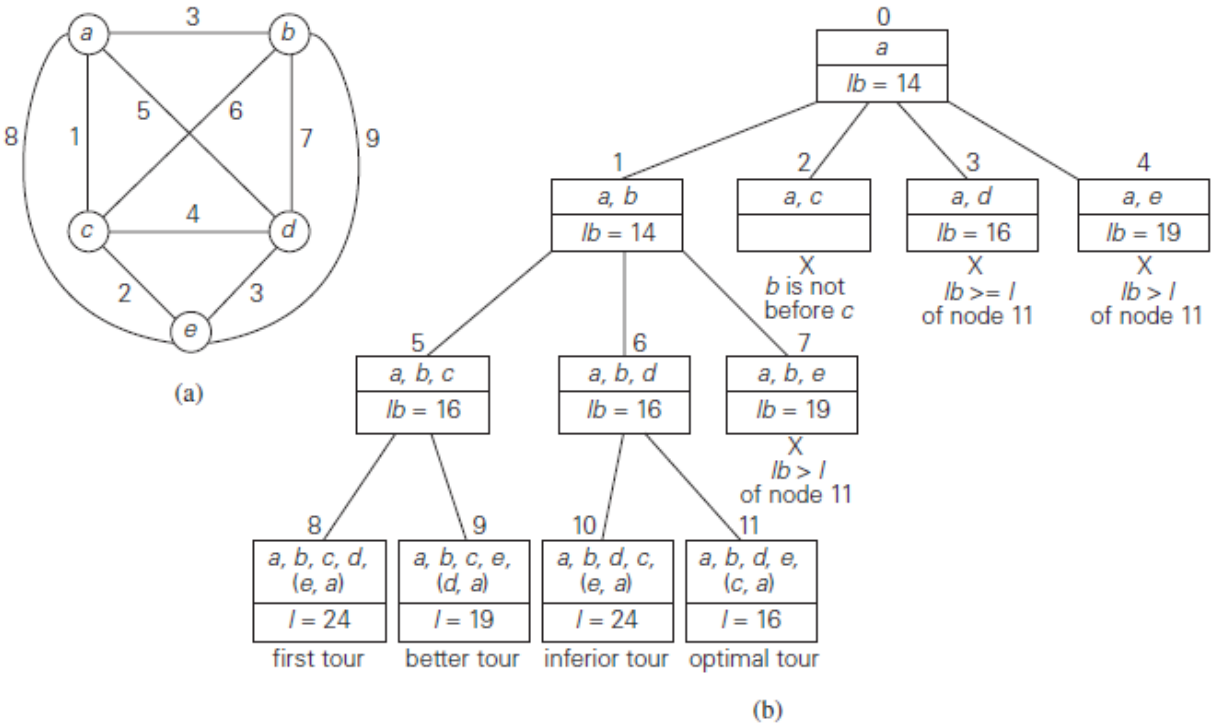


**Figure :** (a)Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

- To reduce the amount of potential work, we take advantage of two observations
  - ✓ 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 state-space tree tracing the algorithm's application is given in Figure(b)

## 3. NP-COMPLETE AND NP-HARD PROBLEMS

### 3.1 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;

- ✓ 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.
- ✓ 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.

### 3.2 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** 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 leads to 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.

### 3.3 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 optimization problem.

### 3.4 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 $P \subseteq NP$

- 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? "

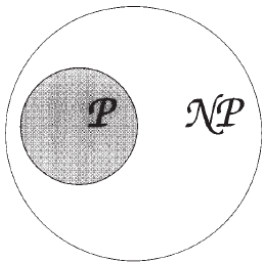


**Figure:** Commonly believed relationship between P and NP

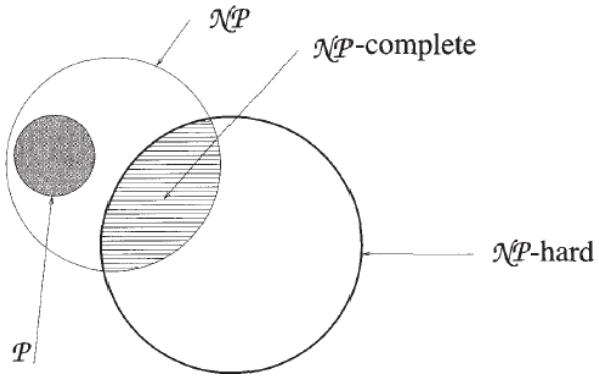


**Figure:** Commonly believed relationship between P, NP, NP-Complete and NP-hard problems

**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.