UNIT V COPING WITH THE LIMITATIONS OF ALGORITHM POWER

Limitations of Algorithm Power-Lower-Bound Arguments-Decision Trees-P, NP and NP-Complete Problems--Coping with the Limitations - Backtracking – n-Queens problem – Hamiltonian Circuit Problem – Subset Sum Problem-Branch and Bound – Assignment problem – Knapsack Problem – Traveling Salesman Problem- Approximation Algorithms for NP – Hard Problems – Traveling Salesman problem – Knapsack problem.

5.1 LIMITATIONS OF ALGORITHM POWER

Some problems cannot be solved by any algorithm. Other problems can be solved algorithmically but not in polynomial time. And even when a problem can be solved in polynomial time by some algorithms, there are usually lower bounds on their efficiency.

Lower bounds: It estimates on a minimum amount of work needed to solve a problem. In general, obtaining a nontrivial lower bound even for a simple-sounding problem is a very difficult task. As opposed to ascertaining the efficiency of a particular algorithm, the task here is to establish a limit on the efficiency of *any* algorithm, known or unknown. This also necessitates a careful description of the operations such algorithms are allowed to perform.

Decision trees: This technique allows us, among other applications, to establish lower bounds on the efficiency of comparison-based algorithms for sorting and for searching in sorted arrays. As a result, we will be able to answer such questions as whether it is possible to invent a faster sorting algorithm than merge sort and whether binary search is the fastest algorithm for searching in a sorted array.

Question of intractability: Which problems can and cannot be solved in polynomial time. This well-developed area of theoretical computer science is called computational complexity theory. The basic elements of this theory includes the fundamental notions as *P*, *NP*, and *NP*-complete problems, including the most important unresolved question of theoretical computer science about the relationship between *P* and *NP* problems.

Numerical analysis: This branch of computer science concerns algorithms for solving problems of ―continuous‖ mathematics—solving equations and systems of equations, evaluating such functions as sin *x* and ln *x*, computing integrals, and so on. The nature of such problems imposes two types of limitations. First, most cannot be solved exactly. Second, solving them even approximately requires dealing with numbers that can be represented in a digital computer with only a limited level of precision. Manipulating approximate numbers without proper care can lead to very inaccurate results.

5.1.1 Lower-Bound Arguments

We will first look at *lower bounds*, which estimate the minimum amount of work needed to solve a given problem.

Once we have established a lower bound, we know that no algorithm can exist without performing work equivalent to at least that of the upper bound.

Some examples:

- The number of comparisons needed to find the largest element in a set of n numbers
- Number of comparisons needed to sort an array of size n
- Number of comparisons necessary for searching in a sorted array of n numbers
- The number of comparisons needed to determine if all elements of an array of n elements are unique
- Number of multiplications needed to multiply two n × n matrices

Lower bounds may be exact counts or efficiency classes (big □). A lower bound is *tight* if there exists an algorithm with the same efficiency as the lower bound.

Some lower bound examples:

- sorting: lower bound (n log n), tight
- searching in a sorted array: lower bound (log n), tight
- determine element uniqueness: lower bound (n log n), tight
- n-digit integer multiplication: lower bound (n), tightness unknown
- multiplication of n × n matrices: lower bound (n2), tightness

unknown There are a number of methods that can be used to establish lower bounds:

- Trivial lower bounds
- Information-theoretic arguments (decision trees)
- Adversary arguments
- Problem reduction

Trivial Lower Bounds

Trivial lower bounds are based on counting the number of items that **must** be processed in input and generated as output to solve a problem.

Some examples:

- Generating all permutations of a set of n elements has a trivial lower bound of (n!) since all n! permutations must be generated. This lower bound is tight since we have algorithms to do this that operate in □(n!).

• Evaluating a polynomial

$p(x) = a_n x_n + a_{n-1} x_{n-1} + \cdot \cdot \cdot + a_0$

requires that each of the n ai's need to be processed, leading to a lower bound of
Ω(n). Again, we have linear algorithms for this, so the bound is tight.

• Computing the product of two n × n matrices requires that each of the 2n2 numbers be multiplied at some point, leading to a lower bound of Ω(n2). No known algorithm can meet this bound, and its tightness is unknown.

• A trivial lower bound for the traveling salesman problem can be obtained as Ω(n2) based on the number of cities and inter-city distances, but this is not a useful result, as no algorithm comes anywhere near this lower bound.

Information-Theoretic Arguments

Rather than the number of inputs or outputs to process, an *information-theoretic lower bound* is based on the amount of information an algorithm needs to produce to achieve its solution.

A binary search fits here – we are trying to find the location of a given value in a sorted array. Since we know the array is sorted, we can, with each guess, eliminate half of the possible locations of the goal, resulting in a lower bound (worst case) of log n steps.

Decision trees

Decision trees are a model of an algorithm's operation that can help us analyze algorithms such as search and sort that work by comparisons. In a decision tree, internal nodes represent comparisons and leaves represent outcomes. The tree branches based on whether the comparison is true or false.

Adversary Arguments

Another approach to finding lower bounds is the *adversary argument*. This method depends on a ―adversary‖ that makes the algorithm work the hardest by adjusting the input.

For example, when playing a guessing game to determine a number between 1 and n using yes/no questions (*e.g.,* ―is the number less than x?‖), the adversary puts the number in the larger of the two subsets generated by last question. (Yes, it cheats.)

The text also provides an adversary argument to show the lower bound on the number of comparisons needed to perform a merge of two sorted n-element lists into a single 2n-element list (as in merge sort).

Problem Reduction

A key idea in the analysis of algorithms is *problem reduction*. If we can come up with a way to convert a problem we wish to solve to an instance of a different problem to which we already have a solution, this produces a solution to the original problem.

Suppose you wrote a program solving some problem A. A few days later, you find out a program needs to be written to solve a similar problem B. To avoid writing too much new code, you might try to come up with a way to solve B using your implementation of A. So given your input to problem B, you would need to have a procedure to transform this input into corresponding input to an instance of problem A. Then solve the instance of problem A (which you already knew how to do). Then you need to transform the output of A back to the corresponding solution to B.

As a very simple example, suppose you have written a procedure to draw an ellipse. draw_ellipse(double horiz, double vert, double x, double y)

This procedure deals with trigonometry and works at a low-level with a graphics library. But it works. If you are later asked to write a procedure to draw a circle. Hopefully you would quickly realize that you could make use of your solution to the problem of drawing an ellipse.

```
draw_circle(double radius, double x,
        double y) { draw_ellipse(2*radius,
        2*radius, x, y);
        }
```

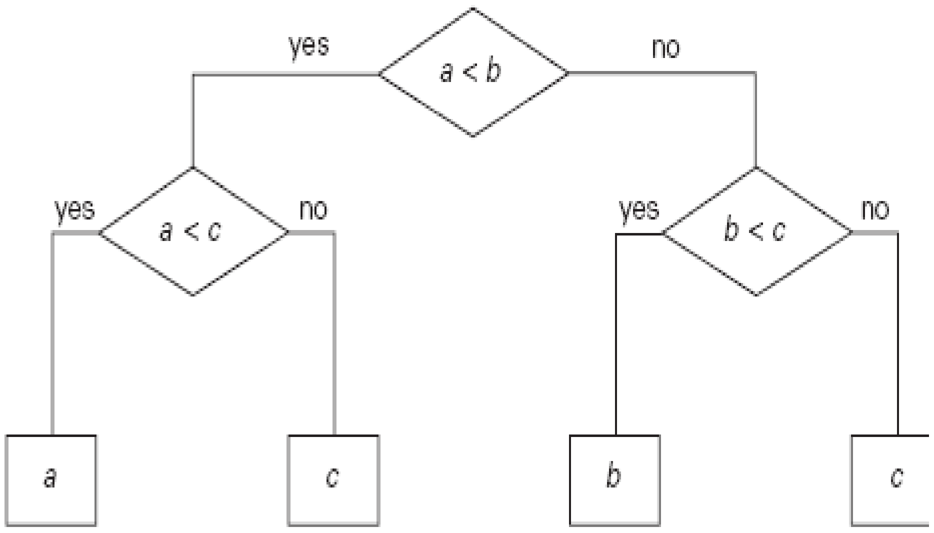
So we have *transformed* or *reduced* the problem of drawing a circle to the problem of drawing an ellipse.

A problem reduction can be used to show a lower bound.

• If problem A is **at least as hard as** problem B, then a lower bound for B is also a lower bound for A.

• Hence, we wish to find a problem B with a known lower bound that can be reduced to the problem A.

5.1.2 Decision Trees



Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., $k < k$. The node's left subtree contains the information about subsequent comparisons made if $k < k$, and its right subtree does the same for the case of $k > k$. Each leaf represents a possible outcome of the algorithm's run on some input of size *n*.

Note that the number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons An important point is that the number of leaves must be at least as large as the number of possible outcomes.

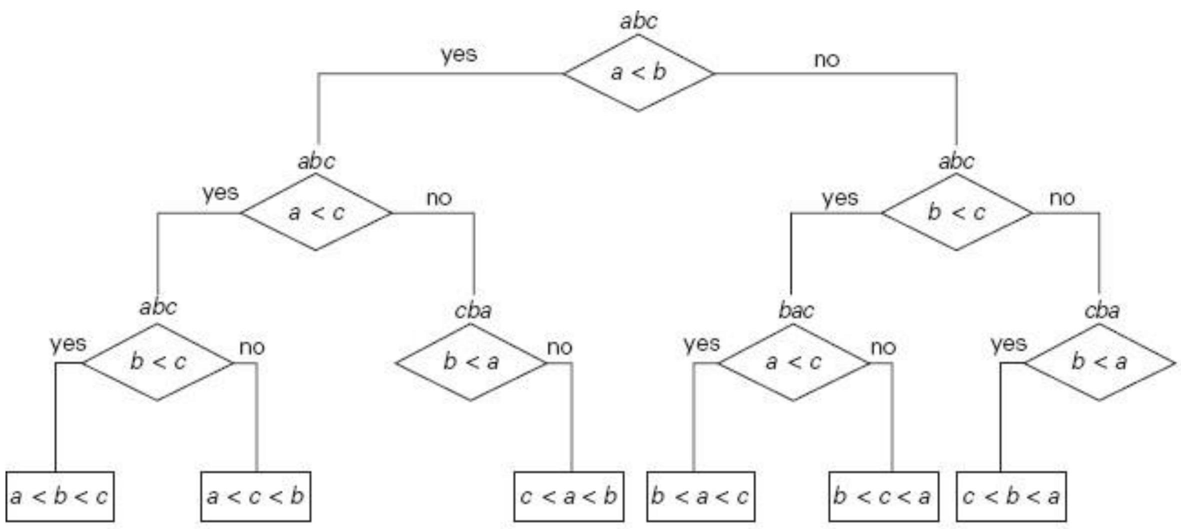
Decision Trees for Sorting

Most sorting algorithms are comparison based, i.e., they work by comparing elements in a list to be sorted. By studying properties of decision trees for such algorithms, we can derive important lower bounds on their time efficiencies.

We can interpret an outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list's elements in ascending order.

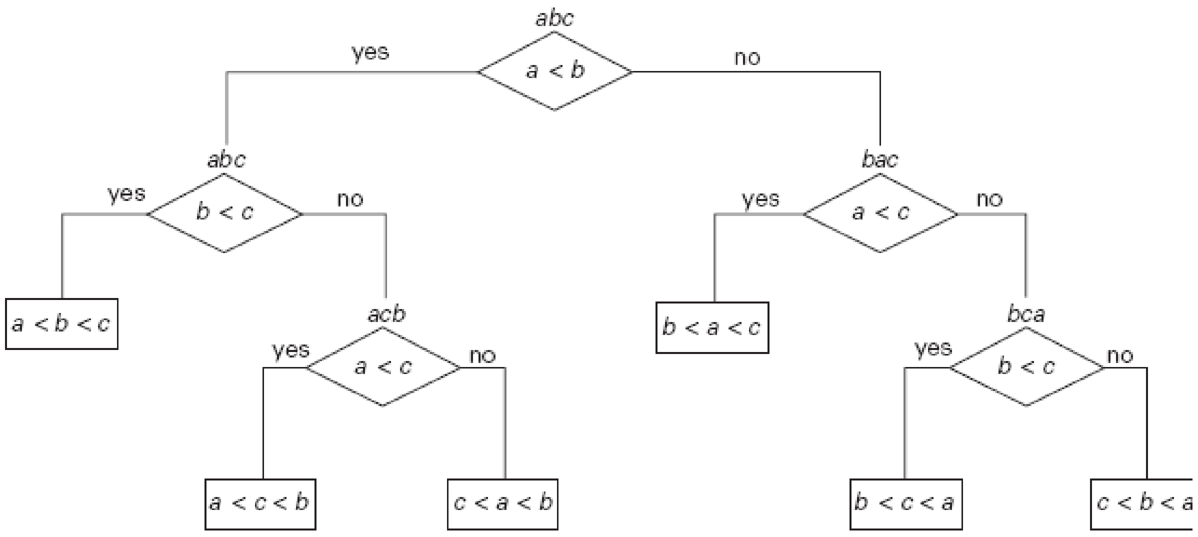
Consider, as an example, a three-element list *a, b, c* of orderable items such as real numbers or strings.

For the outcome $a < c<b$ obtained by sorting this list (see Figure 11.2), the permutation in question is 1, 3, 2. In general, the number of possible outcomes for sorting an arbitrary *n*-element list is equal to *n*!.



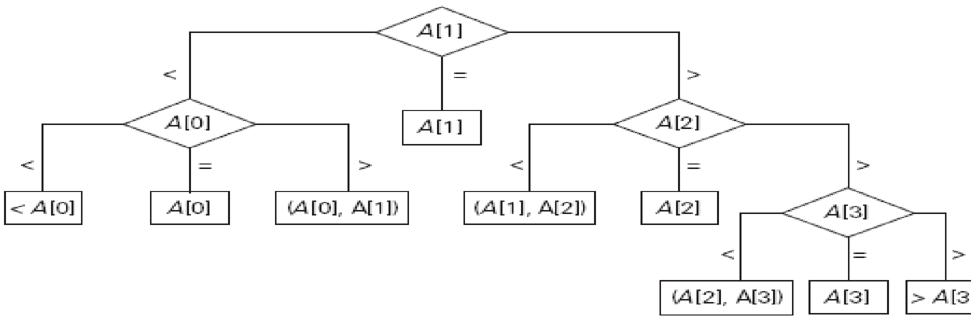
Inequality implies that the height of a binary decision tree for any comparison- based sorting algorithm and hence the worst-case number of comparisons made by such an algorithm cannot be less than _log2 *n*!_:

Cworst(n) ≥ _log2 *n*!_. ≈ *n* log2 *n.*



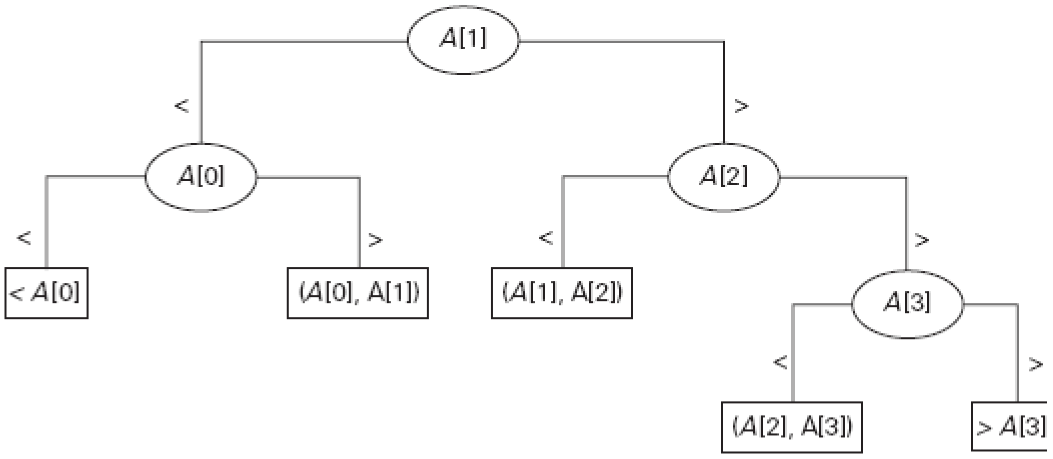
Number of comparisons is *(*2 + 3 + 3 + 2 + 3 + 3*)/*6 = 2 2/3 .

Decision Trees for Searching a Sorted Array



we get the following lower bound on the number of worst-case comparisons:

Cworst(n) ≥ _log3*(2n* + 1*)*_.



Cworst(n) ≥ _log2*(n* + 1*)*_.

5.2 P, NP AND NP COMPLETE PROBLEMS

DEFINITION 1 We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to *O(p(n))* where *p(n)* is a polynomial of the problem's input size *n.* Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

***P* and *NP* Problems**

Informally, we can think about problems that can be solved in polynomial time as the set that computer science theoreticians call *P*. A more formal definition includes in *P* only ***decision problems***, which are problems with yes/no answers.

DEFINITION 2 Class *P* is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called ***polynomial***.

Here is just a small sample of some of the best-known problems that fall into this category:

Hamiltonian circuit problem Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once. ***Traveling salesman problem*** Find the shortest tour through *n* cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).

Knapsack problem Find the most valuable subset of *n* items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

DEFINITION 3 A ***nondeterministic algorithm*** is a two-stage procedure that takes as its input an instance *I* of a decision problem and does the following.

Nondeterministic (―guessing‖) stage: An arbitrary string *S* is generated that can be thought of as a candidate solution to the given instance *I* (but may be complete gibberish as well) Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

Now we can define the class of *NP* problems.

DEFINITION 4 Class *NP* is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.

Most decision problems are in *NP*. First of all, this class includes all the problems in *P*:

$$P \subseteq NP.$$

***NP*-Complete Problems**

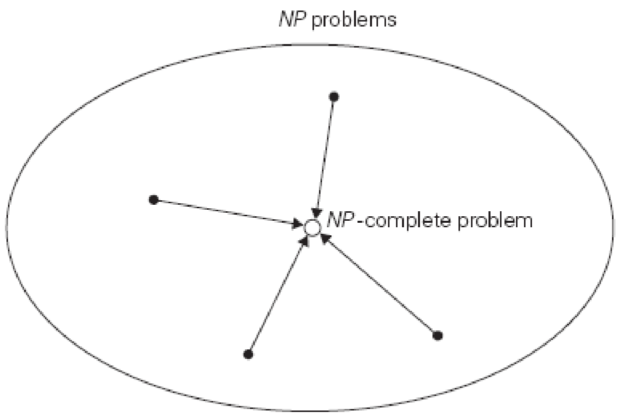
Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class because, by definition, any other problem in *NP* can be reduced to it in

polynomial time (shown symbolically in Figure 11.6). Here are more formal definitions of these concepts.

DEFINITION 5 A decision problem *D*1 is said to be ***polynomially reducible*** to a decision problem *D*2, if there exists a function *t* that transforms instances of *D*1 to instances of *D*2 such that:

1. *t* maps all yes instances of *D*1 to yes instances of *D*2 and all no instances of *D*1 to no instances of *D*2

2. *t* is computable by a polynomial time algorithm



This definition immediately implies that if a problem *D*1 is polynomially reducible to some problem*D*2 that can be solved in polynomial time, then problem *D*1 can also be solved in polynomial time (why?).

DEFINITION 6 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*

5.3 COPING WITH THE LIMITATIONS

• There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):

• Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time

• Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

Exact Solution Strategies

• *exhaustive search* (brute force)

  – useful only for small instances

• *dynamic programming*

  – applicable to some problems (e.g., the knapsack problem)

• *backtracking*

  – eliminates some unnecessary cases from consideration

  – yields solutions in reasonable time for many instances but worst case is still exponential

• *branch-and-bound*

  – further refines the backtracking idea for optimization problems

5.4 BACKTRACKING

• Suppose you have to make a series of *decisions,* among various *choices,* where

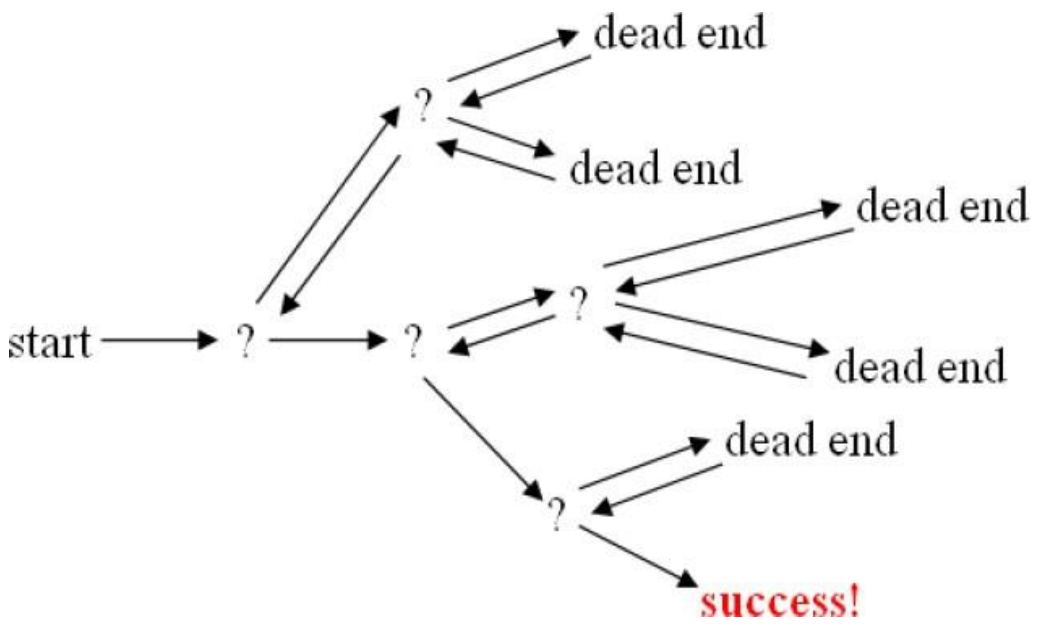
  – You don't have enough information to know what to choose

  – Each decision leads to a new set of choices

  – Some sequence of choices (possibly more than one) may be a solution to your problem

• Backtracking is a methodical way of trying out various sequences of decisions, until you find one that –works‖

Backtracking : A Scenario



Backtracking can be thought of as searching a tree for a particular ―goal‖ leaf node

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent

The backtracking algorithm

- Backtracking is really quite simple--we ―explore‖ each node, as follows:
- To ―explore‖ node N:

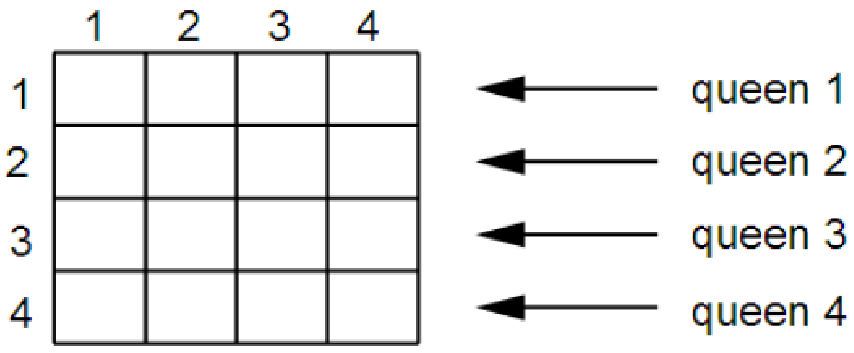
```
1. If N is a goal node, return ―success‖
2. If N is a leaf node, return ―failure‖
3. For each child C of N,
    3.1. Explore C
        3.1.1. If C was successful, return ―success‖
4. Return ―failure‖
```

- Construct the *state-space tree*
  - – nodes: partial solutions
  - – edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- ―Prune‖ *nonpromising nodes*
  - – DFS stops exploring subtrees rooted at nodes that cannot lead to a solution and

backtracks to such a node's parent to continue the search

Example: *n*-Queens Problem

Place *n* queens on an *n*-by-*n* chess board so that no two of them are in the same row, column, or Diagonal



State-Space Tree of the 4-Queens Problem

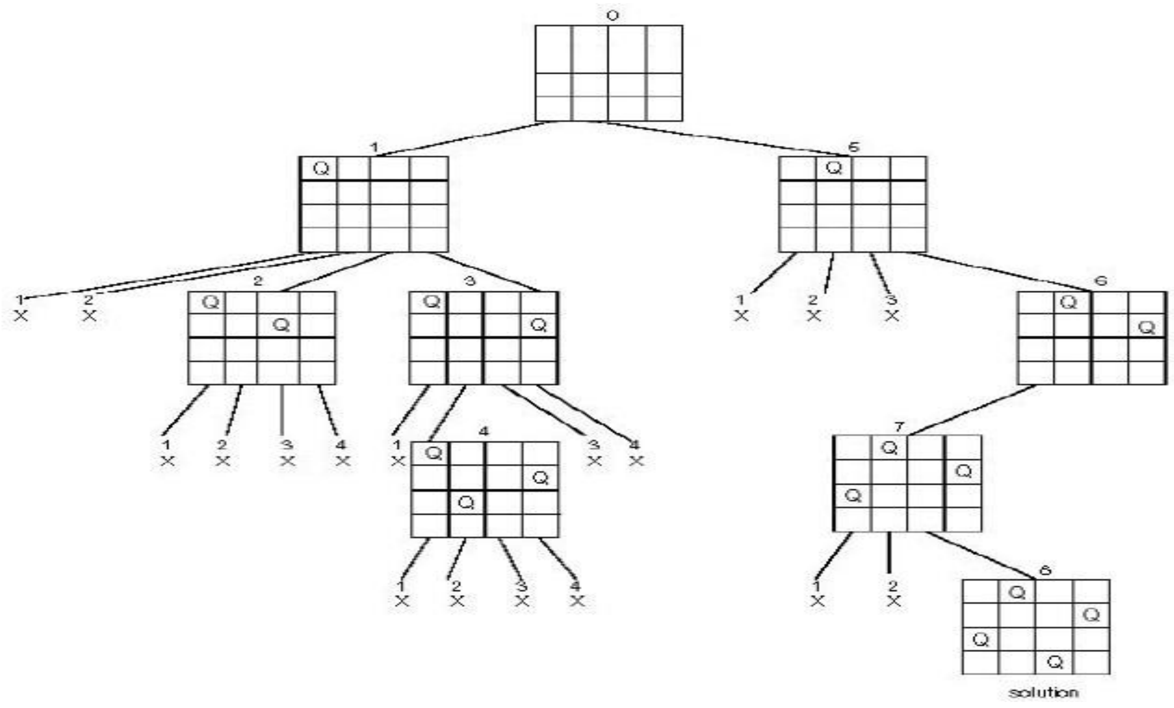
5.4.1 N-Queens Problem:

• The object is to place queens on a chess board in such as way as no queen can capture another one in a single move

– Recall that a queen can move horz, vert, or diagonally an infinite distance

• This implies that no two queens can be on the same row, col, or diagonal

We usually want to know how many different placements there are



4-Queens

- Lets take a look at the simple problem of placing queens 4 queens on a 4x4 board
- The brute-force solution is to place the first queen, then the second, third, and forth
  - – After all are placed we determine if they are placed legally
- There are 16 spots for the first queen, 15 for the second, etc.
  - – Leading to 16*15*14*13 = 43,680 different combinations
- Obviously this isn't a good way to solve the problem

5-First lets use the fact that no two queens can be in the same col to help us

- That means we get to place a queen in each col

6-So we can place the first queen into the first col, the second into the second, etc.

7-This cuts down on the amount of work

- Now there are 4 spots for the first queen, 4 spots for the second, etc.

8-4*4*4*4 = 256 different combinations

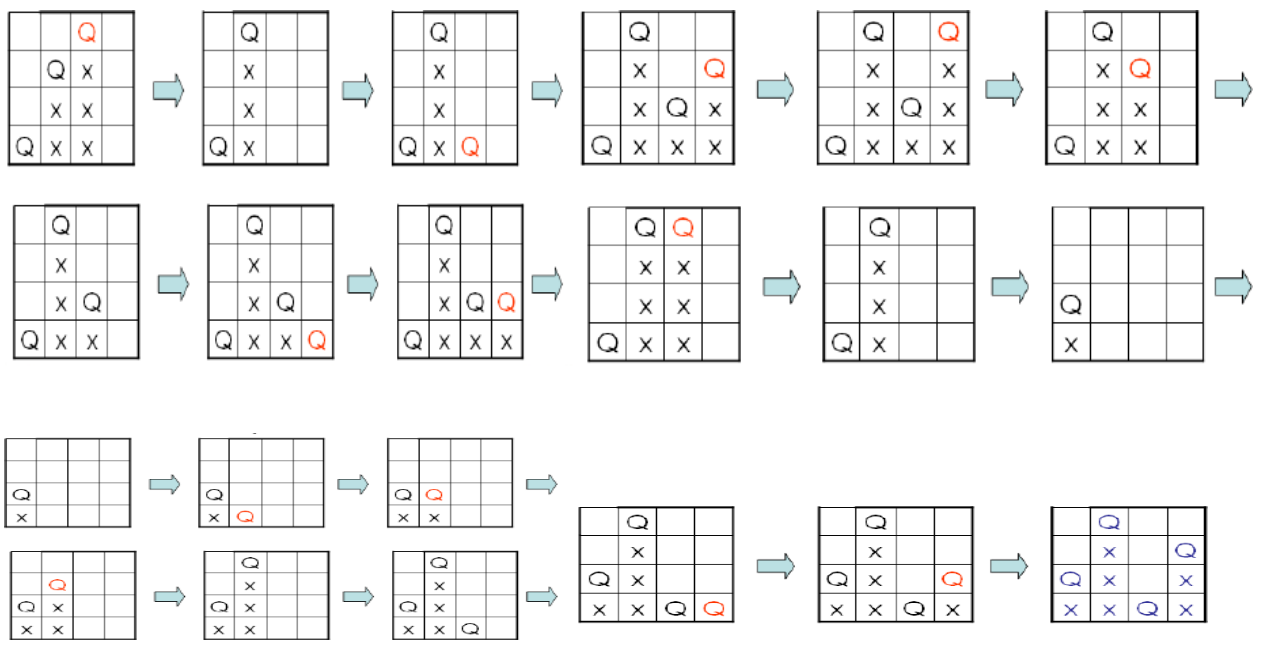
9-However, we can still do better because as we place each queen we can look at the previous queens we have placed to make sure our new queen is not in the same row or diagonal as a previously place queen

10- Then we could use a Greedy-like strategy to select the next valid position for each column

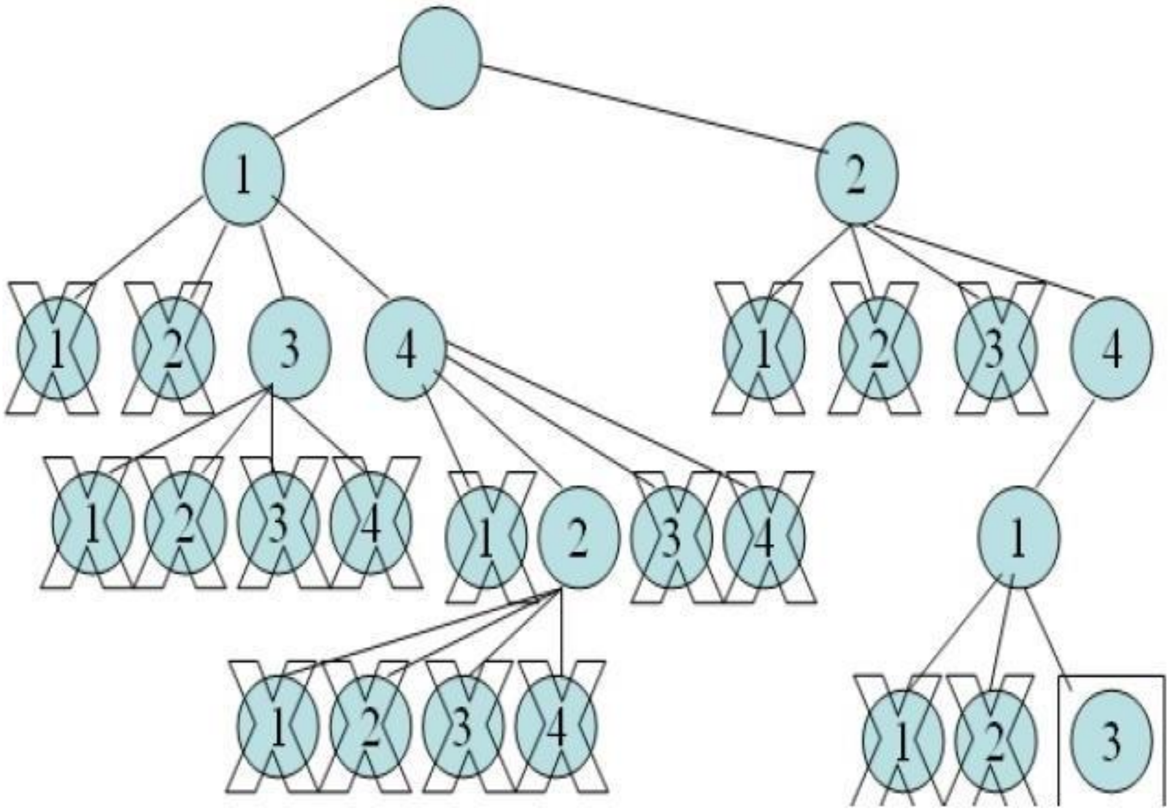
- If one of your choices leads to a dead end, you need to back up to the last choice you made and take a different route

11- That is, you need to change one of your earlier selections

- Eventually you will find your way out of the maze



- **This type of problem is often viewed as a state-space tree**
  - – A tree of all the states that the problem can be in
- **We start with an empty board state at the root and try to work our way down to a leaf node** – Leaf nodes are completed boards



Eight Queen Problem

- The solution is a vector of length 8 (a(1), a(2), a(3), ... , a(8)).

a(i) corresponds to the column where we should place the i-th queen.

- The solution is to build a partial solution element by element until it is complete.
- We should backtrack in case we reach to a partial solution of length k, that we couldn't expand any more.

Eight Queen Problem: Algorithm

```
putQueen(row) {
        for every position col on the same
        row if position col is available
            place the next queen in
        position col if (row<8)
            putQueen(row+
        1); else success;
        remove the queen from position col
```

}

Eight Queen Problem: Implementation

• Define an 8 by 8 array of 1s and 0s to represent the chessboard

• The array is initialized to 1s, and when a queen is put in a position (c,r), board[r][c] is set to zero

• Note that the search space is very huge: 16,772, 216 possibilities.

• Is there a way to reduce search space? Yes Search Pruning.

• We know that for queens:

each row will have exactly one queen each column will have exactly one queen each diagonal will have at most one queen

• This will help us to model the chessboard not as a 2-D array, but as a set of rows, columns and diagonals.

5.4.2 Hamiltonian Circuit Problem

• Hamiltonian Cycle:

– a cycle that contains every node exactly once

• Problem:

– Given a graph, does it have a Hamiltonian cycle?

Background

• **NP-complete** problem:

– Most difficult problems in NP (non- deterministic polynomial time)

• A decision problem *D* is NP-complete if it is complete for NP, meaning that:

– it is in NP

– it is NP-hard (every other problem in NP is reducible to it.)

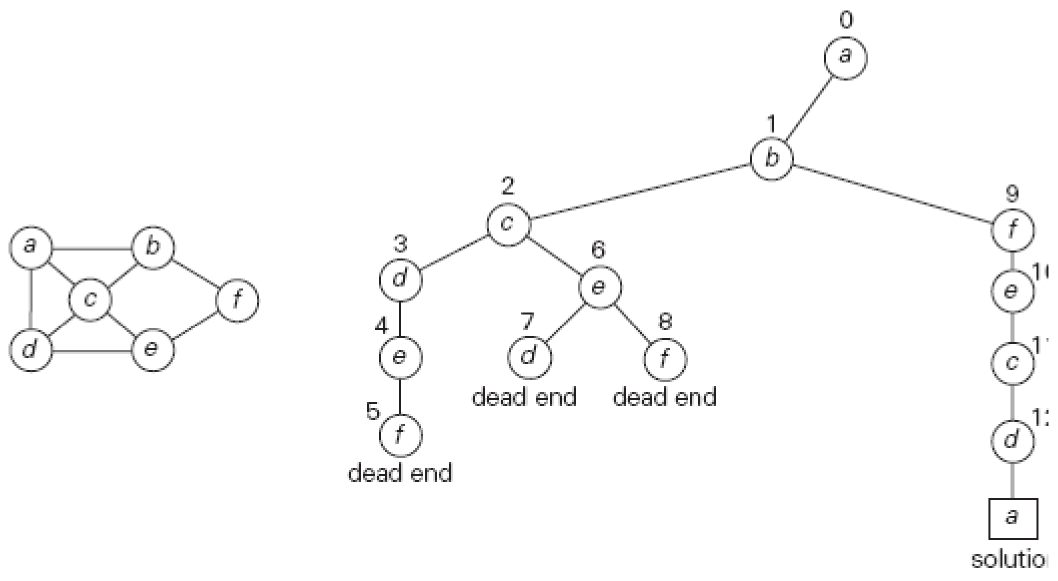
• As they grow large, we are not able to solve them in a reasonable time (polynomial time)

Alternative Definition

• . **NP Problem such as Hamiltonian Cycle:**

– Cannot be solved in Poly-time

– Given a solution, easy to verify in poly-time



(a) (b)

Using the alphabet order to break the three-way tie among the vertices adjacent to *a*, we select vertex *b*. From *b*, the algorithm proceeds to *c*, then to *d*, then to *e*, and finally to *f*, which proves to be a dead end. So the algorithm backtracks from *f* to *e*, then to *d*, and then to *c*, which provides the first alternative for the algorithm to pursue. Going from *c* to *e* eventually proves useless, and the algorithm has to backtrack from *e* to *c* and then to *b*. From there, it goes to the vertices *f* , *e*, *c*, and *d*, from which it can legitimately return to *a*, yielding the Hamiltonian circuit *a*, *b*, *f* , *e*, *c*, *d*, *a*. If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

5.4.3 Subset Sum Problem

• Problem: Given *n* positive integers *w*1, ... *wn* and a positive integer S. Find all subsets of *w*1, ... *wn* that sum to S.

• Example: n=3, S=6, and w1=2, w2=4, w3=6

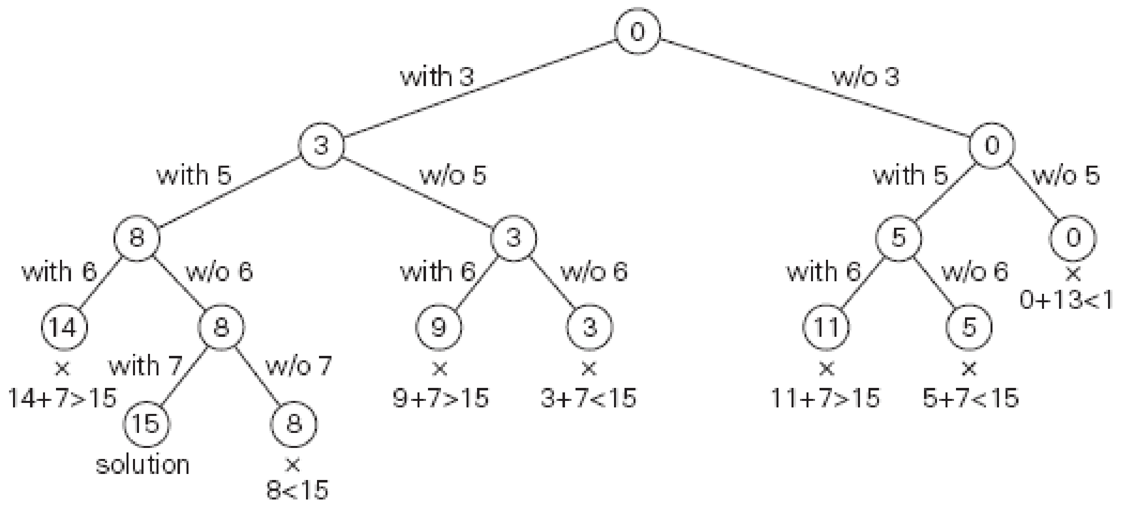
  • Solutions: {2,4} and {6}

The state-space tree can be constructed as a binary tree like that in Figure 12.4 for the instance *A*= {3, 5, 6, 7} and *d* = 15.

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:

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

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



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

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

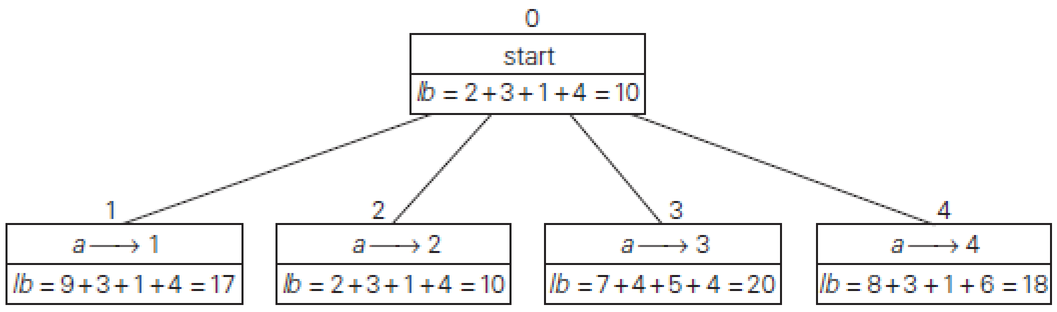
5.5.1 Assignment Problem

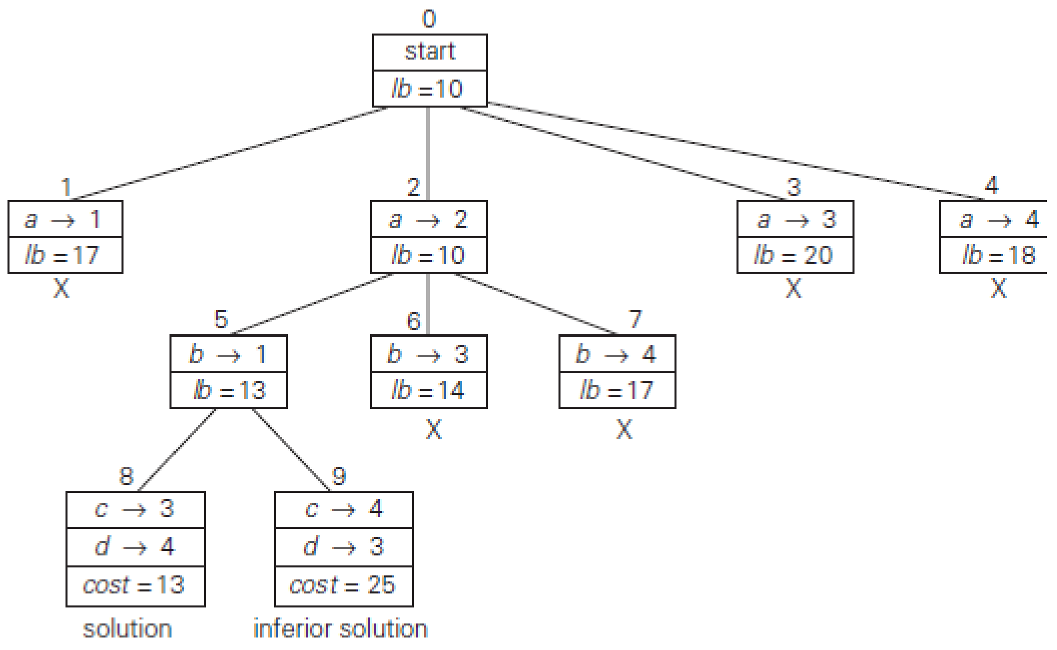
Let us illustrate the branch-and-bound approach by applying it to the problem of assigning *n* people to *n* jobs so that the total cost of the assignment is as small as possible.

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

For the instance here, this sum is 2 + 3+ 1+ 4 = 10. It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.

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 nonterminated leaves in the current tree. 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***.





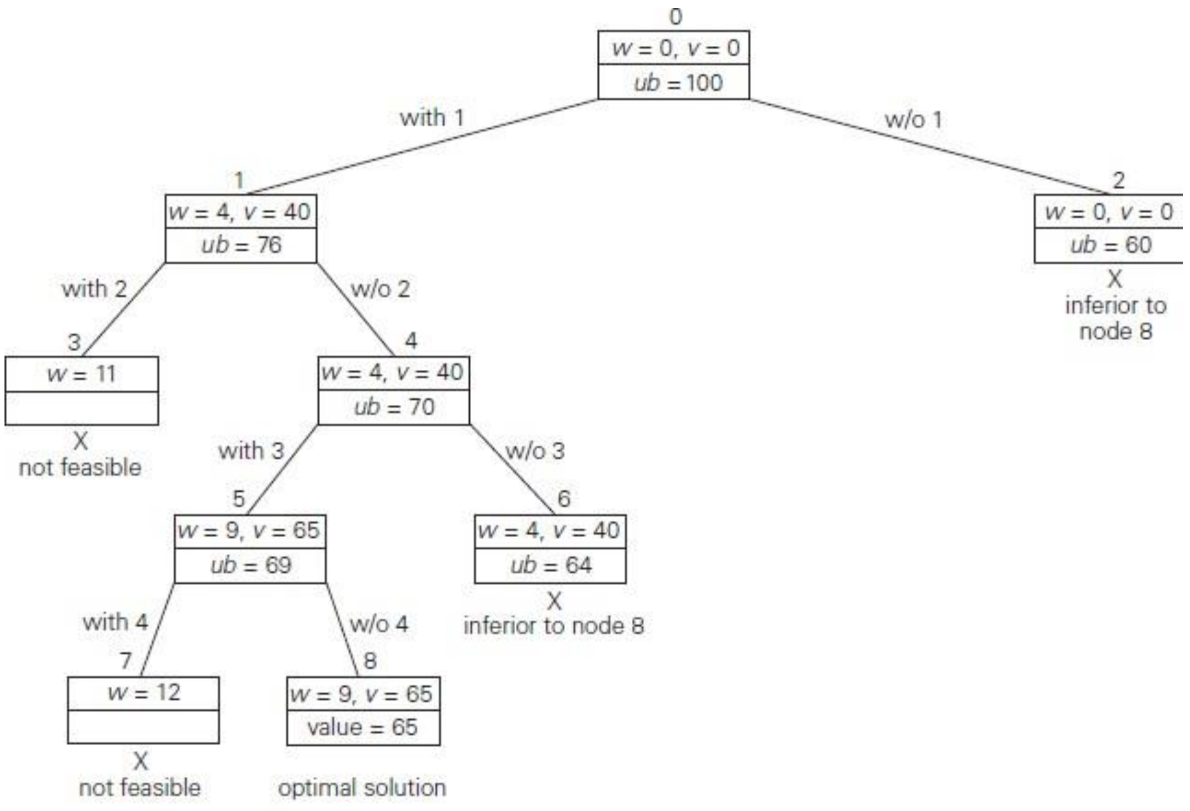
5.5.1 Knapsack Problem

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. A simple way to compute the upper bound *ub* is to add to *v*, the total value of the items already selected, the product of the remaining capacity of the knapsack *W* − *w* and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

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

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

The knapsack's capacity W is 10.

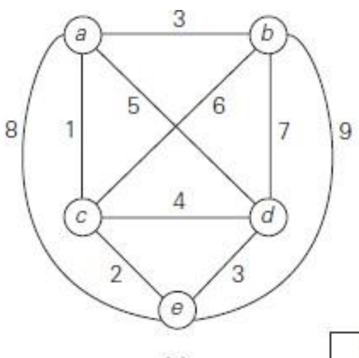


At the root of the state-space tree (see Figure 12.8), no items have been selected as yet. Hence, both the total weight of the items already selected *w* and their total value *v* are equal to 0. The value of the upper bound computed by formula (12.1) is $100. Node 1, the left child of the root, represents 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.

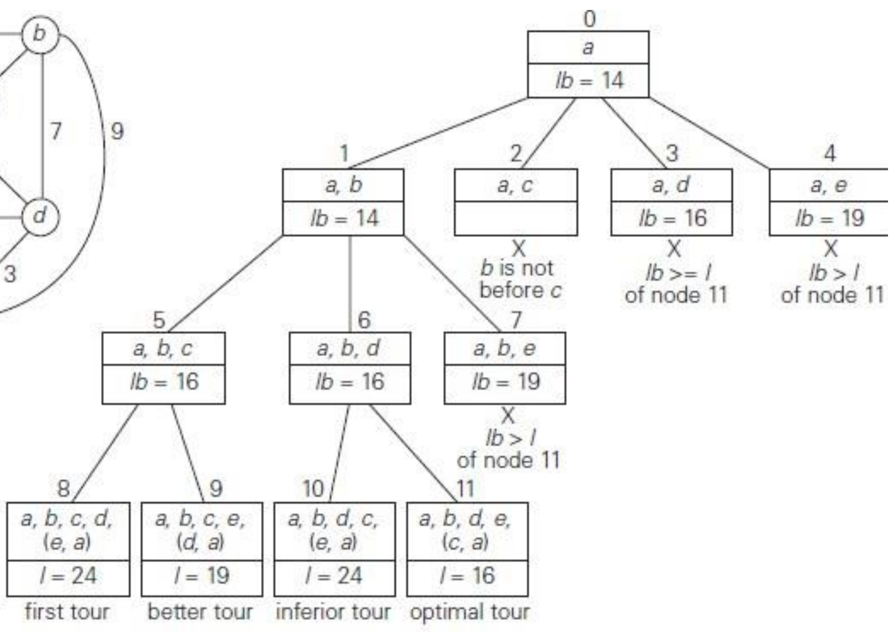
Node 2 represents the subsets that do not include item 1. Accordingly, *w* = 0, *v* = $0, and *ub* = 0 + (10 − 0) * 6 = $60. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children—nodes 3 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 (why?), 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.

5.5.2 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.



(a)



(b)

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

lb = _*s/*2_

For example, for the instance in Figure 12.9a, formula (12.2) yields *lb* = _[*(*1+ 3*)* + *(*3 + 6*)* + *(*1+ 2*)* + *(*3 + 4*)* + *(*2 + 3*)*]*/*2_ = 14.

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

5.6 APPROXIMATION ALGORITHMS FOR *NP*-HARD PROBLEMS

DEFINITION A polynomial-time approximation algorithm is said to be a ***approximation algorithm***, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed *c* for any instance of the problem in question: $r(sa) \leq c$.The best (i.e., the smallest) value of *c* for which inequality (12.3) holds for all instances of the problem is called the ***performance ratio*** of the algorithm and denoted *RA*.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with *RA* as close to 1 as possible. Unfortunately, as we shall see, some approximation algorithms have infinitely large performance ratios $(RA = \infty)$. This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

Brute-force algorithms.

- Develop clever enumeration strategies.
- Guaranteed to find optimal solution.
- No guarantees on running time.

Heuristics.

Develop intuitive algorithms. Guaranteed to run in polynomial time. No guarantees on quality of solution.

Approximation algorithms.

- Guaranteed to run in polynomial time.
- Guaranteed to find "high quality" solution, say within 1% of optimum.

Obstacle: need to prove a solution's value is close to optimum, without even knowing what optimum value is!

An approximation algorithm is bounded by $\rho(n)$ if, for all input of size *n*, the cost *c* of the solution obtained by the algorithm is within a factor $\rho(n)$ of the c^* of an optimal solution Approximation algorithms find an algorithm which return solutions that are guaranteed to be close to an optimal solution.

5.6.1 Approximation Algorithms for the Traveling Salesman Problem

Consider G be an arbitrary undirected graph with *n* vertices Length function

$$l(e) = \{ 1 \qquad \text{if } e \text{ is an edge in } G$$

2 Otherwise for K_n

G has a Hamiltonian cycle then there is an Hamiltonian cycle in K_n whose length is exactly *n* *Traveling salesman problem* is NP hard even if all the edge lengths are 1 or 2 Due to polynomial time reduction from Hamiltonian cycle to this type of Traveling salesman problem

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

Nearest-neighbor algorithm

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

Step 1 Choose an arbitrary city as the start.

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

Step 3 Return to the starting city.

Algorithm Approx-TSP(G, c);

1. Choose a vertex v 2 V .
2. Construct a minimum spanning tree T for G rooted in v (use, e.g., MST-Prim algorithm).
3. Construct the pre-order traversal W of T.
4. Construct a Hamilton cycle that visits the vertices in order W.

Theorem 1 The Approx-TSP is a polynomial-time 2-approximation algorithm for the TSP problem with the triangle inequality.

Theorem 2 Let p ≥1. If P≠NP, then there is no polynomial-time p-approximation algorithm for the TSP problem.

5.6.2 Approximation Algorithms for the Knapsack Problem

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

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

where *k* is an integer parameter in the range $0 \le k < n$. The first approximation scheme was suggested by S. Sahni in 1975. This algorithm generates all subsets of *k* items or less, and for each one that fits into the knapsack it adds the remaining items as the greedy algorithm would do (i.e., in nonincreasing order of their value-to-weight ratios). The subset of the highest value obtained in this fashion is returned as the algorithm's output.

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

capacity $W = 10$

(a)

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

(b)