KALINGA UNIVERSITY, NAYA RAIPUR

Faculty of Computer Science & IT

Programme- Bachelor of Computer Applications (AIML)

Course Name- Design and Analysis of Algorithm

Course Code- BCAAIML504 **5th Semester**

Unit-5

LIMITATIONS OF ALGORITHM POWER

There are many algorithms for solving a variety of different problems. They are very powerful instruments, especially when they are executed by modern computers.

The power of algorithms is limited because of the following reasons:

- There are some problems cannot be solved by any algorithm.
- There are some problems can be solved algorithmically but not in polynomial time.
- There are some problems can be solved in polynomial time by some algorithms, but they are usually lower bounds on their efficiency.

Algorithms limits are identified by the following:

- Lower-Bound Arguments
- Decision Trees
- P, NP and NP-Complete Problems

LOWER-BOUND ARGUMENTS

We can look at the efficiency of an algorithm two ways. We can establish its **asymptotic efficiency class** (say, for the worst case) and see where this class stands with respect to the **hierarchy of efficiency classes**.

For example, selection sort, whose efficiency is quadratic, is a reasonably fast algorithm, whereas the algorithm for the Tower of Hanoi problem is very slow because its efficiency is exponential.

Lower bounds means estimating the minimum amount of work needed to solve the problem. We present several methods for establishing lower bounds and illustrate them with specific examples.

1. Trivial Lower Bounds
2. Information-Theoretic Arguments
3. Adversary Arguments
4. Problem Reduction

In analyzing the efficiency of specific algorithms in the preceding, we should distinguish between a lower-bound class and a minimum number of times a particular operation needs to be executed.

Trivial Lower Bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's **input** that must be **processed** and the number of **output** items that need to be **produced**.

Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a **trivial lower bound**.

For example, any algorithm for generating all permutations of n distinct items must be in $\Omega(n!)$ because the size of the output is n!. And this bound is **tight** because good algorithms for generating permutations spend a constant time on each of them except the initial one.

Consider the problem of **evaluating a polynomial of degree n** at a given point x, given its coefficients $a_n, a_{n-1}, \ldots, a_0$. $p(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_0$. All the coefficients have to be processed by any polynomial-evaluation algorithm. i.e **$\Omega(n)$.** This is tight lower bound.

Similarly, a trivial lower bound for computing **the product of two n × n matrices** is **$\Omega(n^2)$** because any such algorithm has to process $2n^2$ elements in the input matrices and generate n^2 elements of the product. It is still unknown, however, whether this bound is tight.

The trivial bound for the **traveling salesman problem** is $\Omega(n^2)$, because its input is *n(n-1)/2* intercity distances and its output is a list of n + 1 cities making up an optimal tour. But this bound is useless because there is no known algorithm with the running time being a polynomial function.

Determining the lower bound lies in **which part of an input must be processed** by any algorithm solving the problem. For example, searching for an element of a given value in a sorted array does not require processing all its elements.

Information-Theoretic Arguments

The information-theoretical approach seeks to establish a lower bound based on **the amount of information it has to produce** by algorithm.

Consider an example "**Game of guessing number**", the well-known game of deducing a positive integer between 1 and n selected by somebody by asking that person questions with yes/no answers. The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by $]\log_2 n]$.

The number of bits needed to specify a particular number among the n possibilities. Each answer to the question gives information about each bit.

1. Is the first bit zero? → No→ first bit is 1
2. Is the second bit zero? → Yes→ second bit is 0
3. Is the third bit zero? → Yes→ third bit is 0
4. Is the forth bit zero? → Yes→ forth bit is 0

The number in binary is 1000, i.e.8 in decimal value.

The above approach is called the ***information-theoretic argument*** because of its connection to information theory. This is useful for finding ***information-theoretic lower bounds*** for many problems involving comparisons, including sorting and searching.

Its underlying idea can be realized the mechanism of ***decision trees***. Because

Adversary Arguments

Adversary Argument is a method of proving by **playing a role of adversary (opponent)** in which algorithm has to work more for **adjusting input** consistently.

Consider the Game of guessing number between positive integer 1 and n by asking a person (Adversary) with yes/no type answers for questions. After each question at least one-half of the numbers reduced. If an algorithm stops before the size of the set is reduced to 1, the adversary can exhibit a number.

Any algorithm needs $]\log_2 n]$ iterations to shrink an n-element set to a one-element set by halving and rounding up the size of the remaining set. Hence, at least $]\log_2 n]$ questions need to be asked by any algorithm in the worst case. This example illustrates the ***adversary method*** for establishing lower bounds.

Consider the problem of **merging two sorted lists** of size n $a_1 < a_2 < . . . < a_n$ and $b_1 < b_2 < . . . < b_n$ into a single sorted list of size 2n. For simplicity, we assume that all the a's and b's are distinct, which gives the problem a unique solution.

Merging is done by repeatedly comparing the first elements in the remaining lists and outputting the smaller among them. The number of key comparisons (lower bound) in the worst case for this algorithm for merging is **2n − 1**.

Problem Reduction

Problem reduction is a method in which a difficult unsolvable problem P is reduced to another solvable problem B which can be solved by a known algorithm.

A similar reduction idea can be used for finding a lower bound. To show that problem P is at least as hard as another problem Q with a known lower bound, we need to reduce Q to P (not P to Q!). In other words, we should show that an arbitrary instance of problem Q can be transformed to an instance of problem P, so any algorithm solving P would solve Q as well. Then a lower bound for Q will be a lower bound for P. Table 5.1 lists several important problems that are often used for this purpose.

TABLE 5.1 Problems often used for establishing lower bounds by problem reduction

Problem	Lower bound	Tightness
Sorting	Ω(n log n)	yes
searching in a sorted array	Ω (log n)	yes
element uniqueness problem	Ω (n log n)	yes
multiplication of n-digit integers	Ω(n)	unknown
multiplication of n × n matrices	Ω(n2)	unknown

Consider the Euclidean minimum spanning tree problem as an example of establishing a lower bound by reduction:

Given n points in the Cartesian plane, construct a tree of minimum total length whose vertices are the given points. As a problem with a known lower bound, we use the element uniqueness problem.

We can transform any set x1, x2, . . . , xn of n real numbers into a set of n points in the Cartesian plane by simply adding 0 as the points' y coordinate: (x1, 0), (x2, 0), . . . , (xn, 0). Let T be a minimum spanning tree found for this set of points. Since T must contain a shortest edge, checking whether T contains a zero length edge will answer the question about uniqueness of the given numbers. This reduction implies that Ω (n log n) is a lower bound for the Euclidean minimum spanning tree problem.

P, NP AND NP-COMPLETE PROBLEMS

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

There are several **reasons for intractability**.

- **First**, we **cannot solve** arbitrary instances of intractable problems in a reasonable amount of time unless such **instances are very small**.
- **Second**, although there might be a huge difference between the running times in *O(p(n))* for polynomials of **drastically different degrees**. where p(n) is a polynomial of the problem's input size n.
- **Third**, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are **always polynomials too**.
- **Fourth**, the choice of this class has led to a development of an extensive theory called ***computational complexity***.

Definition: Class *P* is a class of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called ***polynomial class***.

- 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.
- The class of decision problems that are solvable in O($p(n)$) **polynomial time**, where $p(n)$ is a polynomial of problem's input size n

  Examples:
  - Searching
  - Element uniqueness
  - Graph connectivity
  - Graph acyclicity
  - Primality testing (finally proved in 2002)
- **The restriction of P** to decision problems can be justified by the following reasons.
  - First, it is sensible to **exclude problems not solvable in polynomial time** because of their exponentially large output. e.g., generating subsets of a given set or all the permutations of n distinct items.
  - Second, **many important problems that are not decision problems** in their most natural formulation can be reduced to a series of decision problems that are

easier to study. For example, instead of asking about the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color. Coloring of the graph's vertices with no more than m colors for m = 1, 2, (The latter is called the **m-coloring problem**.)

- So, every decision problem can not be solved in polynomial time. Some **decision** problems cannot be solved at all by any algorithm. Such problems are called **undecidable**, as opposed to **decidable** problems that can be solved by an algorithm (**Halting problem**).

- **Non polynomial-time algorithm:** There are many important problems, however, for which no polynomial-time algorithm has been found.
    - ***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.
    - ***Partition problem*:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
    - ***Bin-packing problem*:** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.
    - ***Graph-coloring problem*:** For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.
    - ***Integer linear programming problem*:** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

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

1. **Nondeterministic ("guessing") stage:** An arbitrary string S is generated that can be thought of as a candidate solution to the given instance.
2. **Deterministic ("verification") stage:** A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. (If S is not a solution to

instance I , the algorithm either returns no or is allowed not to halt at all.) Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

Definition: 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 ⊆ NP

This is true because, if a problem is in P, we can use the deterministic polynomial time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string S generated in its nondeterministic ("guessing") stage. But NP also contains the Hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in NP.

Note that P = NP would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm.

Definition: 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 D1 is polynomially reducible to some problemD2 that can be solved in polynomial time, then problem D1 can also be solved in polynomial time

Definition: A decision problem *D* is said to be ***NP-complete*** if it is hard as any problem in NP.

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, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

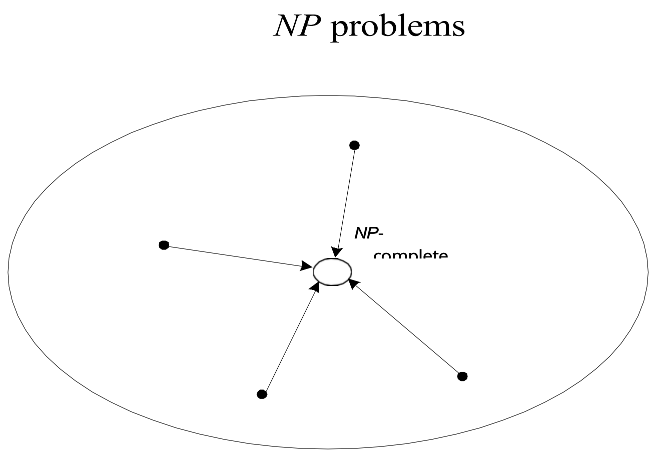


FIGURE 5.6 Polynomial-time reductions of *NP* problems to an *NP*-complete problem

Theorem: A decision problem is said to be *NP-complete* if it is hard as any problem in NP.

Proof: Let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G′ representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G. As the upper bound m on the Hamiltonian circuit length, we take $m = n$, where n is the number of vertices in G (and G′). Obviously, this transformation can be done in polynomial time.

Let G be a yes instance of the Hamiltonian circuit problem. Then G has a Hamiltonian circuit, and its image in G′ will have length n, making the image a yes instance of the decision traveling salesman problem.

Conversely, if we have a Hamiltonian circuit of the length not larger than n in G′, then its length must be exactly n and hence the circuit must be made up of edges present in G, making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the Hamiltonian circuit problem.

This completes the proof.

Theorem: State and prove Cook's theorem.

Prove that CNF-sat is *NP*-complete.

Satisfiability of boolean formula for three conjuctive normal form is NP-Complete.

NP problems obtained by polynomial-time reductions from a *NP*-complete problem

Proof: The notion of *NP*-completeness requires, however, polynomial reducibility of *all* problems in *NP,* both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that specific examples of *NP*-complete problems have been actually found.

Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the former Soviet Union. In his 1971 paper, Cook [Coo71] showed that the so-called ***CNF-satisfiability problem*** is *NP*complete.

x_1	x_2	x_3	$\bar{x}_1$	$\bar{x}_2$	$\bar{x}_3$	$x_1 \vee \bar{x}_2 \vee \bar{x}_3$	$\bar{x}_1 \vee x_2$	$\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$	$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \mathrm{A} \quad (\bar{x}_1 \vee x_2) \mathrm{A} \quad (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
T	T	T	F	F	F	T	T	F	F
T	**T**	**F**	F	F	T	T	T	T	**T**
T	F	T	F	T	F	T	F	T	F
T	F	F	F	T	T	T	F	T	F
F	T	T	T	F	F	F	T	T	F
F	T	F	T	F	T	T	T	T	T
F	F	T	T	T	F	T	T	T	T
F	F	F	T	T	T	T	T	T	T

The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2, and x_3 and their negations denoted $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$ respectively:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \& (\bar{x}_1 \vee x_2) \& (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

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*. (It is easy to see that this can be done for the above formula: if $x_1 =$ *true*, $x_2 =$ *true*, and $x_3 =$ *false*, the entire expression is *true*.)

Since the Cook-Levin discovery of the first known *NP*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well-known problems (or their decision versions) mentioned above—Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *NP*-complete. It is known, however, that if *P* != *NP* there must exist *NP* problems that neither are in *P* nor are *NP*-complete.

Showing that a decision problem is *NP*-complete can be done in two steps.

1. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.
2. The second step is to show that every problem in NP is reducible to the problem in questionin polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial time.

The definition of *NP*-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one *NP*-complete problem, then every problem in *NP* can be solved in polynomial time by a deterministic algorithm, and hence *P* = *NP*.

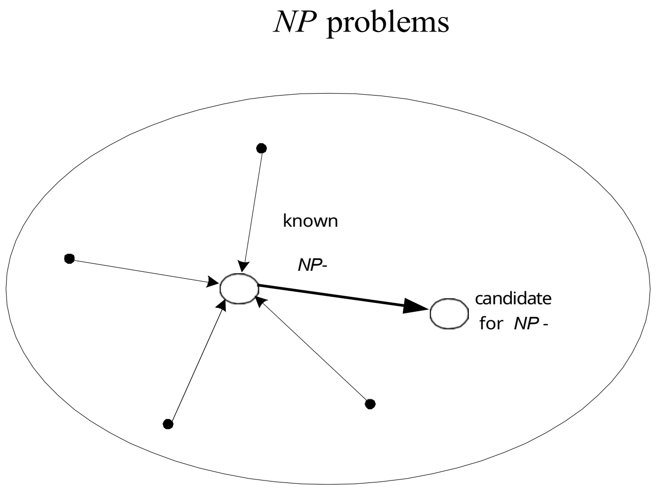


FIGURE 5.7 NP-completeness by reduction

Examples: TSP, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature P = NP would imply that every problem in NP, including all NP-complete problems, could be solved in polynomial time If a polynomial-time algorithm for just one NP-complete problem is discovered, then every problem in NP can be solved in polynomial time, i.e. P = NP Most but not all researchers believe that P != NP , i.e. P is a proper subset of NP. If P != NP, then the NP-complete problems are not in P, although many of them are very useful in practice.

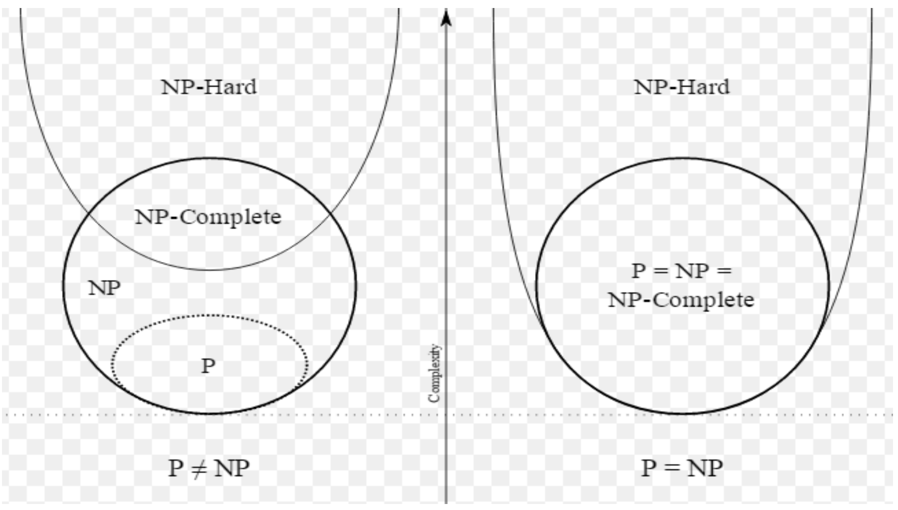


FIGURE 5.8 Relation among P, NP, NP-hard and NP Complete problems

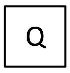
BACKTRACKING TECHNIQUE OVERVIEW

- Backtracking is a more intelligent variation 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 ***nonpromising***.
- Leaves represent either nonpromising 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 depthfirst 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 nonpromising, 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.
- Backtracking techniques are applied to solve the following problems
    - *n*-Queens Problem
    - Hamiltonian Circuit Problem
    - Subset-Sum Problem

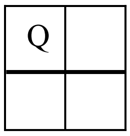
N-QUEENS PROBLEM

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.

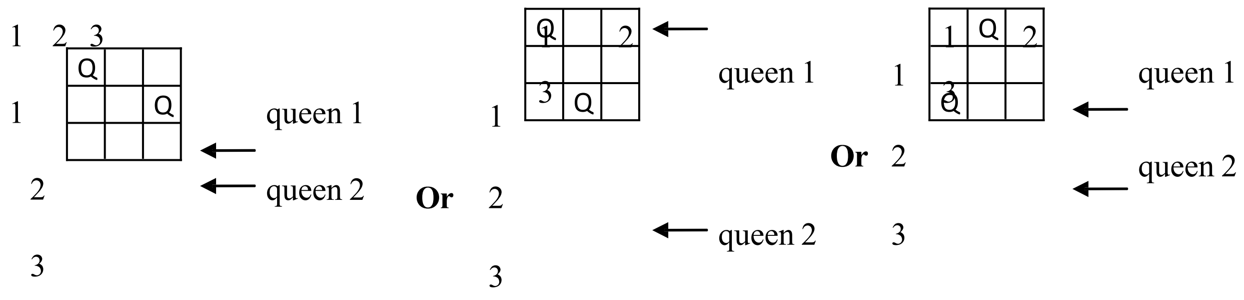
For *n* = 1, the problem has **a trivial solution**.



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



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

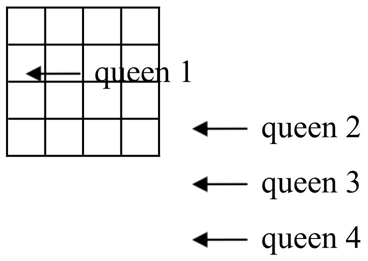


For *n* = 4, There is **solution** to place 4 queens in 4 × 4 chessboard. the four-queens problem solved by the backtracking technique.

Step 1: Start with the empty board

1 2 3 4

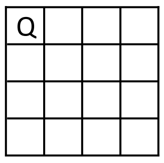
1



Step 2: Place queen 1 in the first possible position of its row, which is in column 1 of row 1.

1 2 3 4

1



2

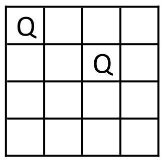
3

4

Step 3: 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.

1 2 3 4

1



2

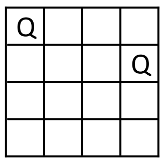
3

4

Step 4: 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).

1 2 3 4

1



2

3

4

Step 5: Then queen 3 is placed at (3, 2), which proves to be another dead end.

1 2 3 4

1

Q			
			Q
	Q		

2

3

4

Step 6: The algorithm then backtracks all the way to queen 1 and moves it to (1, 2).

1 2 3 4

1

	Q		

2

3

4

Step 7: The queen 2 goes to (2, 4).

1 2 3 4

1

	Q		
			Q

2

3

4

Step 8: The queen 3 goes to (3, 1).

1 2 3 4

1

	Q		
			Q
Q			

2

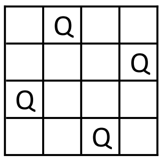
3

4

Step 9: The queen 3 goes to (4, 3). This is a solution to the problem.

1 2 3 4

1



2

3

4

FIGURE 5.9 Solution four-queens problem in 4x4 Board.

The state-space tree of this search is shown in Figure 12.2

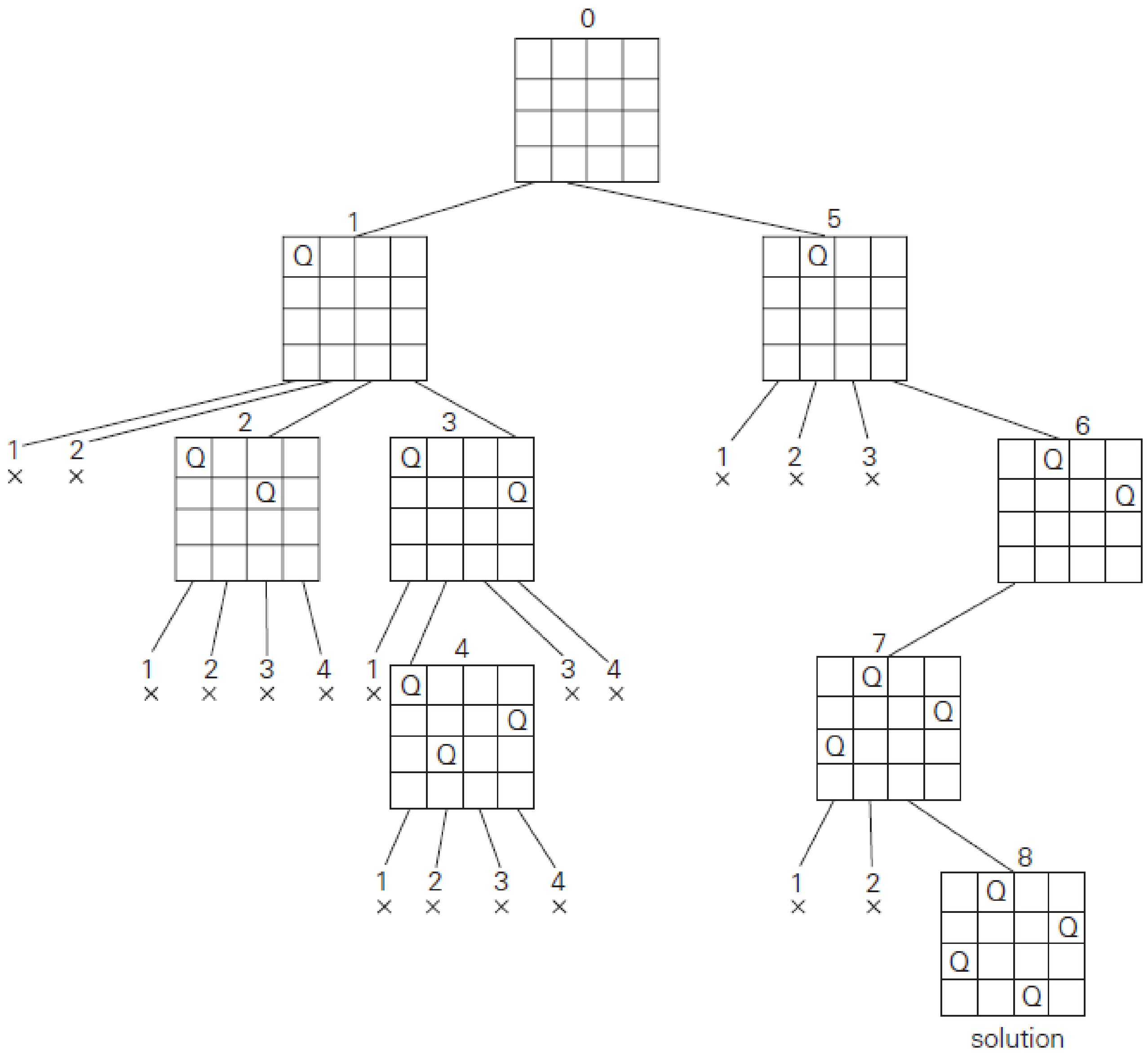


FIGURE 5.10 State-space tree of solving the four-queens problem by backtracking. × denotes an

unsuccessful attempt to place a queen.

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

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

FIGURE 5.11 Solution 8-queens problem in 8x8 Board.

HAMILTONIAN CIRCUIT PROBLEM

A **Hamiltonian circuit** (also called a **Hamiltonian cycle**, **Hamilton cycle**, or **Hamilton circuit)** is a graph **cycle** (i.e., closed loop) through a graph that visits each node exactly once. A graph possessing a **Hamiltonian cycle** is said to be a **Hamiltonian** graph.

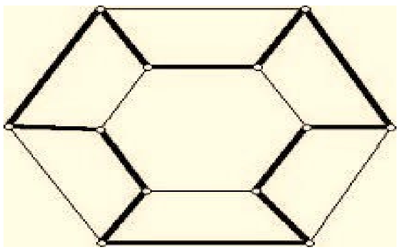


FIGURE 5.12 Graph contains Hamiltonian circuit

Let us consider the problem of finding a Hamiltonian circuit in the graph in Figure 5.13.

Example: Find Hamiltonian circuit starts at vertex *a.*

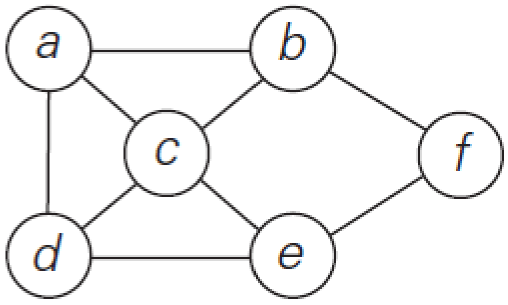


FIGURE 5. 13 Graph.

Solution:

- Assume that if a Hamiltonian circuit exists, it starts at vertex *a*. accordingly, we make vertex *a* the root of the state-space tree as in Figure 5.14.
- In a Graph G, Hamiltonian cycle begins at some vertex $V_1 \in G$, and the vertices are visited only once in the order $V_1, V_2, \ldots, V_n$. (V_i are distinct except for V_1 and V_{n+1} which are equal).
- The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian circuit to be constructed. Using the alphabet order to break the three-way tie among the vertices adjacent to *a,* we
- Select vertex *b*. From *b*, the algorithm proceeds to *c*, then to *d*, then to *e*, and finally to *f,* which proves to be a dead end.
- So the algorithm backtracks from *f* to *e*, then to *d*, and then to *c*, which provides the first alternative for the algorithm to pursue.
- Going from *c* to *e* eventually proves useless, and the algorithm has to backtrack from *e* to *c* and then to *b*. From there, it goes to the vertices *f* , *e*, *c*, and *d*, from which it can legitimately return to *a*, yielding the Hamiltonian circuit *a*, *b*, *f* , *e*, *c*, *d*, *a*. If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

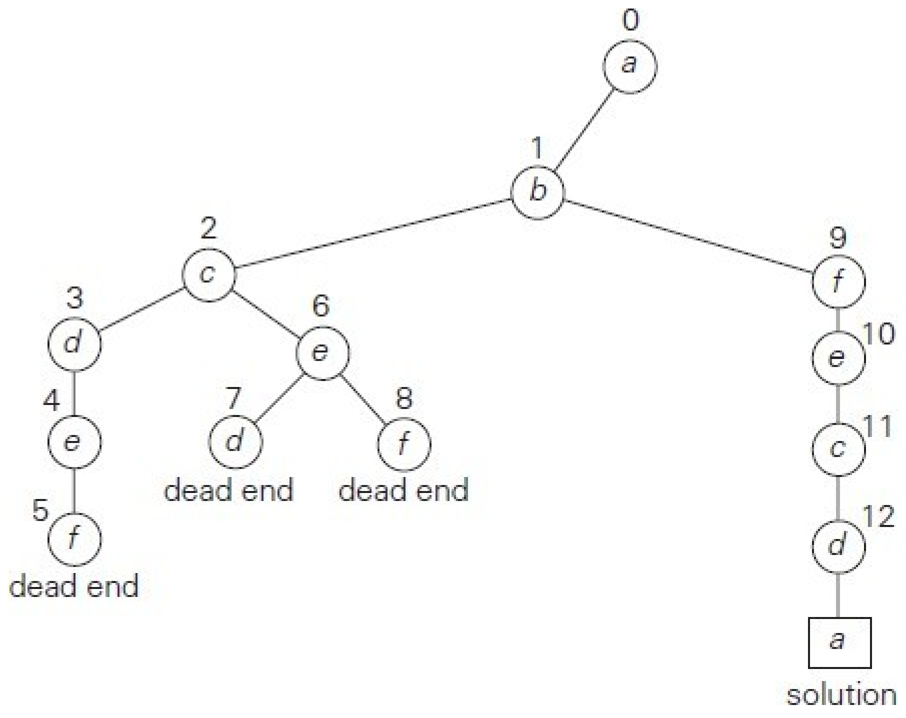


FIGURE 5.14 State-space tree for finding a Hamiltonian circuit.

SUBSET SUM PROBLEM

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

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

A = {3, 5, 6, 7} and d = 15 of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

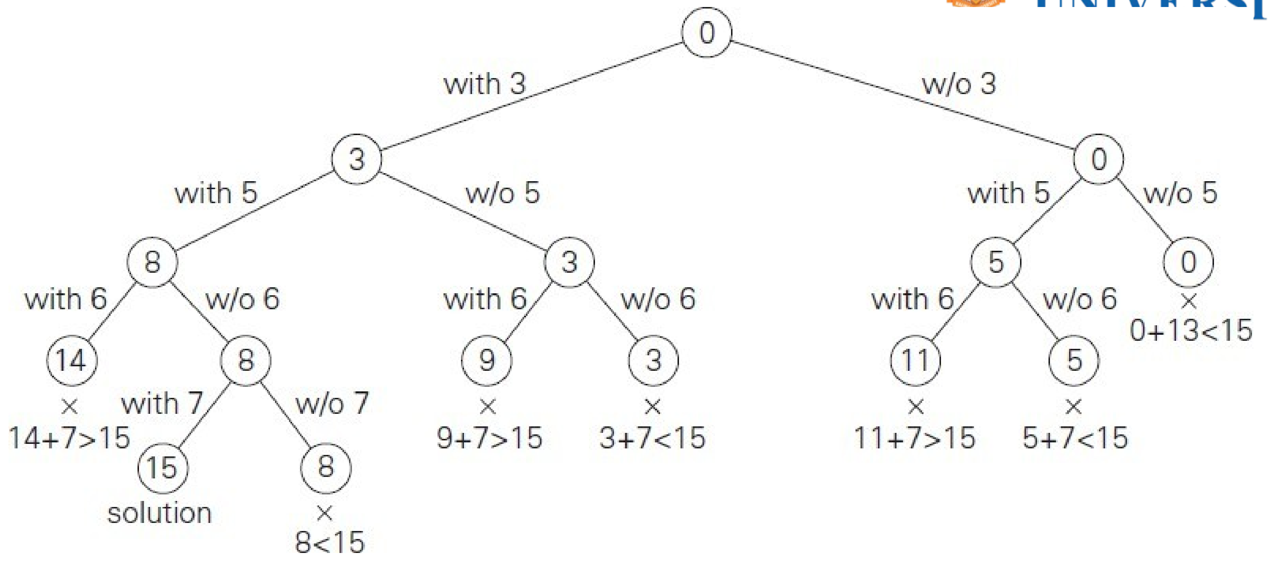


FIGURE 5.15 Complete state-space tree of the backtracking algorithm applied to the instance

Example:

- The state-space tree can be constructed as a binary tree like that in Figure 5.15 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:

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

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

General Remarks

From a more general perspective, most backtracking algorithms fit the following escription. An output of a backtracking algorithm can be thought of as an *n*-tuple $(x_1, x_2, \ldots, x_n)$ where each coordinate *xi* is an element of some finite lin early ordered set *Si* . For example, for the *n*-queens problem, each *Si* is the set of integers (column numbers) 1 through *n.*

A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partially constructed tuples with the first i coordinates defined by the earlier actions of the algorithm. If such a tuple $(x_1, x_2, \ldots, x_i)$ is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of $((x_1, x_2, \ldots, x_i)$ and the problem's constraints, and adds it to the tuple as its (i + 1)st coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of xi, and so on.

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 this section
    for each element $x \in Si+1$ consistent with $X[1..i]$ and the constraints do
        $X[i + 1] \leftarrow x$
        Backtrack($X[1..i + 1]$)
```

BRANCH AND BOUND TECHNIQUE

An optimization problem seeks to minimize or maximize some objective function, 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 travelling 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 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 function1 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

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:

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.

Some problems can be solved by Branch-and-Bound are:

1. Assignment Problem
2. Knapsack Problem

3. Traveling Salesman Problem

TRAVELING SALESMAN PROBLEM(Branch and Bound)

We will be able to apply the branch-and-bound technique to instances of the travelling 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 (Problem 8 in this section's exercises) that 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 *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:

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

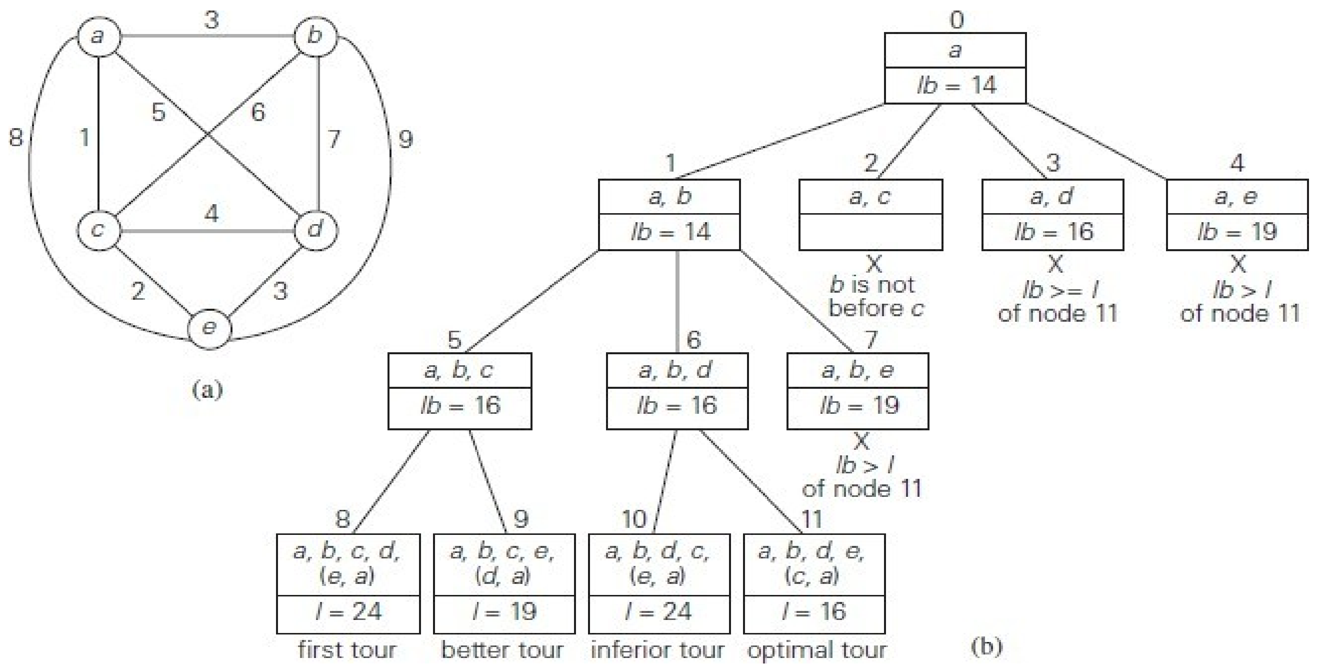


FIGURE 5.19 (a)Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

For example, for the instance in Figure and above formula yields

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

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 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)*:

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

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 5.19a. To reduce the amount of potential work. 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 5.19b.

APPROXIMATION ALGORITHMS FOR NP HARD PROBLEMS

Now we are going to discuss a different approach to handling difficult problems of combinatorial optimization, such as the **travelling salesman problem and the knapsack problem.** The decision versions of these problems are *NP*-complete. Their optimization versions fall in the class of ***NP-hard problems***—problems that are at least as hard as *NP*-complete problems. Hence, there are no known polynomial-time algorithms for these problems, and there are serious theoretical reasons to believe that such algorithms do not exist.

Approximation algorithms run a gamut in level of sophistication; most of them are based on some problem-specific heuristic. A ***heuristic*** is a common-sense rule drawn from experience rather than from a mathematically proved assertion. For example, going to the nearest unvisited city in the travelling salesman problem is a good illustration of this notion.

Of course, if we use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution s_a to a problem of ***minimizing*** some function ***f*** by the size of the relative error (***re***) of this approximation,

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)}$$

where s^* is an exact solution to the problem. Alternatively, $re(sa) = f(sa)/f(s^*) - 1$, we can simply use the ***accuracy ratio***

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

as a measure of accuracy of s_a. Note that for the sake of scale uniformity, the accuracy ratio of approximate solutions to ***maximization*** problems is usually computed as

$$r(s_a) = f(s^*)$$

$f(s_a)$

to make this ratio greater than or equal to 1, as it is for minimization problems. Obviously, the closer $r(s_a)$ is to 1, the better the approximate solution is. For most instances, however, we cannot compute the accuracy ratio, because we typically do not know $f(s^*)$, the true optimal value of the objective function. Therefore, our hope should lie in obtaining a good upper bound on the values of $r(s_a)$. This leads to the following definitions.

A polynomial-time approximation algorithm is said to be a ***c 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: $r(s_a) \leq c$.

The best (i.e., the smallest) value of c for which inequality holds for all instances of the problem is called the ***performance ratio*** of the algorithm and denoted R_A.

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 = ∞). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

Approximation Algorithms for NP Hard Problems are:

- Traveling salesman problem (tsp)
- Knapsack problem