WWW.KTUNOTES.IN

MODULE 6

***BackTracking*:** The Control Abstraction – The N Queen's Problem, 0/1 Knapsack Problem

Branch and Bound: Travelling Salesman Problem

Introduction to Complexity Theory: Tractable and Intractable Problems – The P and NP Classes – Polynomial Time Reductions – The NP-Hard and NP-Complete Classes

BackTracking

Problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using backtracking. In backtracking, the desired solution is expressible as an n-tuple (x_1, ..., x_n), where the x_i are chosen from some finite set S_i .Often the problem to be solved calls for finding one vector that maximizes(or minimizes or satisfies) a criterion function $P(x_1, ..., x_n)$.

Most of the problems solved using backtracking require that all the solutions satisfy a complex set of constraints. These constraints can be divided into two categories: explicit and implicit.

Explicit constraints are rules that restrict each x_i to take on values only from a given set. Common examples of explicit constraints are

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

Implicit constraints are rules that determine which of the tuples in the solution space I satisfy the criteria function.

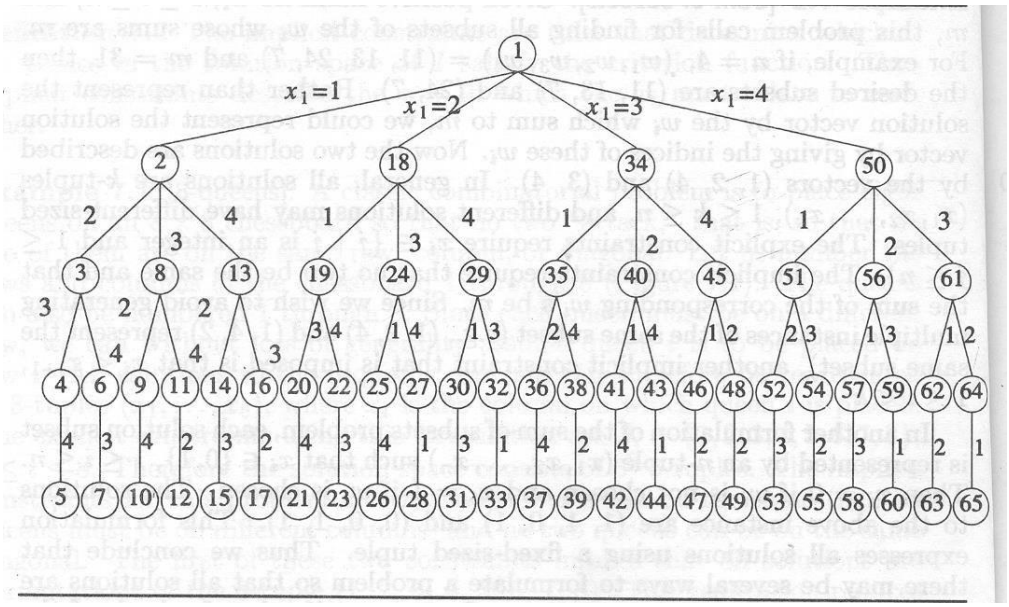
Example:

N-Queens Problem: N queens are to be placed on an nxn chessboard so that no two of them are in attacking position. i.e., no two queens are in same row or same row or same diagonal.

Let us assume that queen i is placed in row i. So the problem can be represented as n-tuple (x1….. xn) where each xi represents the column in which we have to place queen i. Hence the explicit constraint is Si = {1, 2... n}. No two xi can be same and no two queens can be in same diagonal is the implicit constraint.

Consider the case where n=4. A permutation tree or state space tree shows all possible elements in solution space. Edge from level i to level i+1 specify the value of xi. Hence there are 4! = 24 leaves for the tree.

Here nodes are labeled in Depth First Search order.



State Space Tree:

The tree organization of the solution space is referred to as the **state space tree**. Each node in state space tree defines a problem state. All paths from root to other nodes define state space of problem.

Solution states are those problem states for which the path from root to s defines a tuple in solution space.

Answer states are those solution states s for which path from root to s defines a tuple that is member of solution (satisfies implicit constraints). The tree representation of solution space is called state space tree.

Live node:

A node which has been generated and all of whose children are not yet been generated is called a live node.

E node:

A live node whose children are currently been generated is called an E-node (node being expanded).

Dead node:

A generated node with all its children expanded is called a dead node.

There are two different ways to generate problem state

1. Given a list of live nodes, as soon as new child C of current E-node R is generated, the child becomes new E-node. When sub tree C got fully explored, R becomes the next E-node. Hence it will result in a depth first generation.

2. Given a list of live nodes, E-node remains as E-node until it is dead. (Breadth first generation)

In both the cases, bounding functions are used to kill live nodes without generating all their children. Depth first generation with bounding function result in backtracking and the Breadth first generation with bounding function result in Branch and bound method.

Backtracking: General principle

Backtracking find all answer nodes, not just one case. Let (x1... x_i) be a path from root to a node in the state space tree. T(x1... x_i) be the set of all possible values for xi+1 such that (x1,..., xi,xi+1) is also a path to problem state. Let Bi+1 be a bounding function such that if Bi+1(x1, xi,xi+1) is false for the path (x1, ..., xi,xi+1) from the root to the problem state, then path cannot be extended to answer node. Then candidates for position i+1 are those generated by the T satisfying Bi+1.

Control Abstraction

```
Algorithm Backtrack (k)                    //Recursive backtracking algorithm
{
        for (each x[k] such that x[k] □ T(x[1],…,x[k-1]))
        {
                if (Bk(x[1],…,x[k-1])!=0)
                {
                        if ((x[1],…,x[k]) is a path to an answer node) then
                        write(x[1: k]);
                        if (k<n) then Backtrack (k+1);
                }
        }
}
```

An iterative backtracking method is shown below


```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1]) and Bk(x[1], ..., x[k]) is true) then
11          {
12                  if (x[1], ..., x[k] is a path to an answer node)
13                      then write (x[1 : k]);
14                  k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

N Queens Problem

Problem: Given an nxn chessboard, place n queens in non-attacking position i.e., no two queens are in same row or same column or same diagonal.

Let us assume that queen i is placed in row i. So the problem can be represented as n tuple (x1, ..., xn) where each xi represents the column in which we have to place queen i. Hence the explicit constraints is Si ={1,2,..,n}. No two xi can be same and no two queens can be in same diagonal is the implicit constraint. Since we fixed the row number the solution space reduce from n^n to n!.

To check whether they are on the same diagonal, let chessboard be represented by an array a[1..n][1..n]. Every element with same diagonal that runs from upper left to lower right has same "row-column" value. E.g., consider the element a[4][2]. Elements a[3][1], a[5][3], a[6][4], a[7][5] and a[8][6] have row-column value 2. Similarly every element from same diagonal that goes from upper right to lower left has same "row+column" value. The elements a[1][5], a[2][4], a[3][3], a[5][1] have same "row+column" value as that of element a[4][2] which is 6.

Hence two queens placed at (i,j) and (k,l) have same diagonal iff

i – j = k – l or i + j = k + l

i.e., j – l = i – k or j – l = k – i

|j – l| = |i – k|

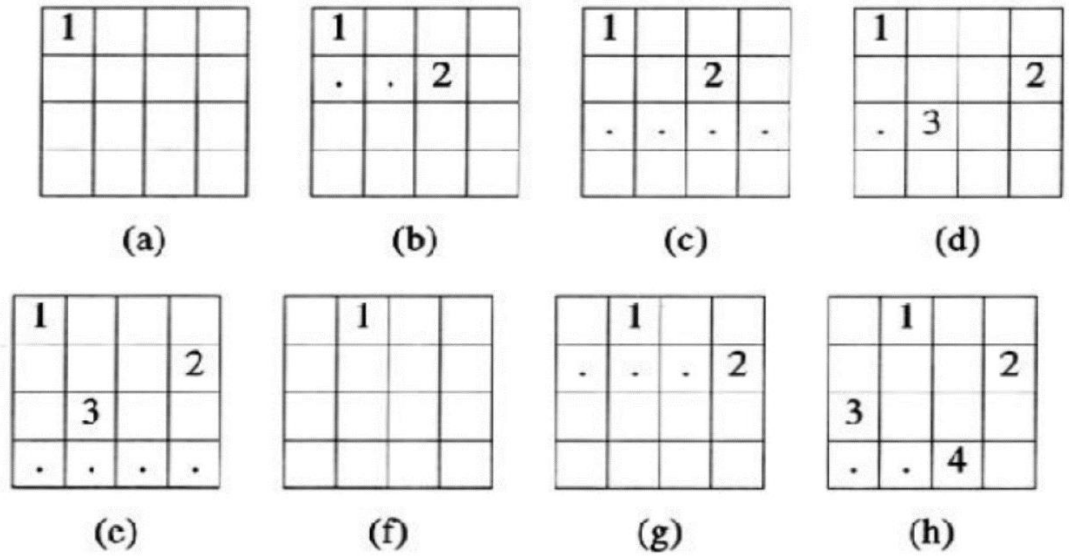
Algorithm


```
Algorithm Place (k, i)
{
        // Returns true if queen can be placed on kth row, ith column. Otherwise return
        false. Assume that first k-1 elements of x is set.
        for j=1 to k-1 do
        if ((x[j] == i) || (abs(x[j]-i) == abs(j-k))) //same column or same diagonal
        {
                return false;
        }
return true;
}

Algorithm NQueens( k, n)
{
// Prints all possible permutations to place n queens on an nxn chess board so that
// none of them are in attacking position.
        for i=1to n do
        {
                if (Place(k,i)) then
                {
                        x[k] = i;
                        if (k==n) then write(x[1..n];
                        else NQueens(k+1, n);
```

```
        }
}
```

In **4 queens problem** where n=4.



Path is (2, 4, 1, 3)

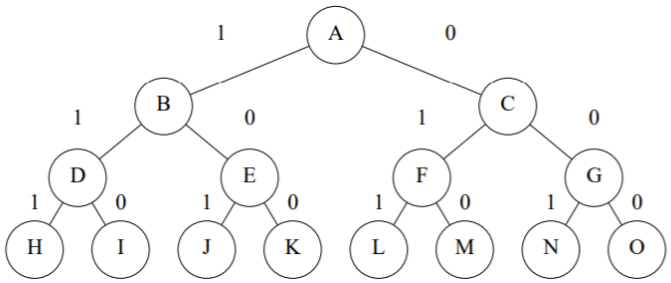
0/1 Knapsack Problem

We are given n objects and a knapsack or bag. Object i has weight w_i and the knapsack has a capacity m. The objective is to fill the knapsack in a way that maximizes total profit earned. Since the knapsack capacity is m, the total weight of all objects chosen must be atmost m. Formally, the problem can be stated as

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized}$$

Where x_i is either 0 or 1.

The solution space for this problem consists of the 2^n ways to assign 0 or 1 values to the x_i's. For n=3, the state space tree is shown below:



Example:

Consider the knapsack instance n = 3, w = [20, 15, 15], p = [40, 25, 25], and c = 30. We search the tree shown above, beginning at the root. The root is the only live node at this time. It is also the E-node. From here we can move to either B or C. Suppose we move to B. The live nodes now are A and B. B is the current E-node. At node B the remaining capacity r is 10, and the profit earned cp is 40. From B we can move to either D or E. The move to D is infeasible, as the capacity needed to move there is w2 = 15. The move to E is feasible, as no capacity is used in this move. E becomes the new E-node. The live nodes at this time are A, B, and E. At node E, r = 10 and cp = 40. From E we have two possible moves (i.e., to nodes J and K). The move to node J is infeasible, while that to K is not. Node K becomes the new E-node. Since K is a leaf, we have a feasible solution. This solution has profit value cp = 40. The values of x are determined by the path from the root to K. This path is (A, B, E, K). Since we cannot expand K further, this node dies and we back up to E. Since we cannot expand E further, it dies too.

Next we back up to B, which also dies, and A becomes the E-node again. It can be expanded further, and node C is reached. Now r = 30 and cp = 0. From C we can move to either F or G. Suppose we move to F. F becomes the new E-node, or the live nodes are A, C, and F. At F, r = 15 and cp = 25. From F we can move to either L or M. Suppose we move to L. Now r = 0 and cp = 50. Since L is a leaf and it represents a better feasible solution than the best found so far (i.e., the one at node K), we remember this feasible solution as the best solution. Node L dies, and we back up to node F. Continuing in this way, we search the entire tree. The best solution found during the search is the optimal one.

Bounding functions are needed to help kill live nodes without expanding them. A good bounding function is obtained by using upper bound on the value of the best feasible solution obtained by expanding the given live node and any of it's descendants. If this upper bound is not higher than the value of the solution determined so far, then that live node can be killed.

Functon Bound(cp,cw,k) determines an upper bound. For this we relax the $x_i = 0$ or 1 to $0<=x_i<=1$. The object weights and profits are w[i] and p[i]. It is assumed that p[i]/w[i] >= p[i+1]/w[i+1], 1<=i=n.

```
1   Algorithm Bound(cp, cw, k)
2   // cp is the current profit total, cw is the current
3   // weight total; k is the index of the last removed
4   // item; and m is the knapsack size.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           c := c + w[i];
9           if (c < m) then b := b + p[i];
10          else return b + (1 − (c − m)/w[i]) ∗ p[i];
11      }
12      return b;
13  }
```

The algorithm below is invoked by BKnap(1,0,0) and f_p is initially set to -1.

```
1   Algorithm BKnap(k, cp, cw)
2   // m is the size of the knapsack; n is the number of weights
3   // and profits. w[ ] and p[ ] are the weights and profits.
4   // p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
5   // knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
6   // is not in the knapsack; else x[k] = 1.
7   {
8       // Generate left child.
9       if (cw + w[k] ≤ m) then
10      {
11          y[k] := 1;
12          if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
13          if ((cp + p[k] > fp) and (k = n)) then
14          {
15              fp := cp + p[k]; fw := cw + w[k];
16              for j := 1 to k do x[j] := y[j];
17          }
18      }
19      // Generate right child.
20      if (Bound(cp, cw, k) ≥ fp) then
21      {
22          y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
23          if ((cp > fp) and (k = n)) then
24          {
25              fp := cp; fw := cw;
26              for j := 1 to k do x[j] := y[j];
27          }
28      }
29  }
```


Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

Branch and Bound

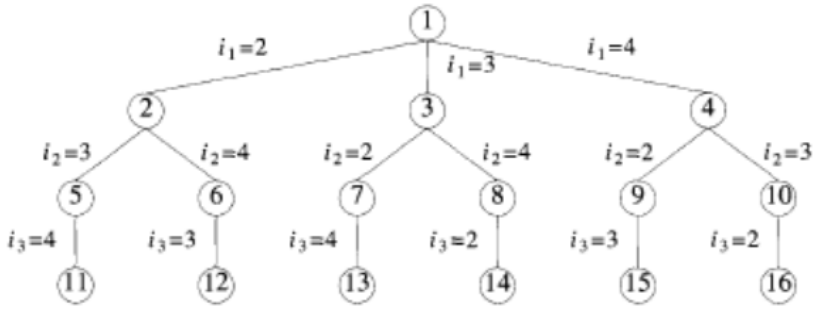
Branch and bound is a general algorithm for finding optimal solutions of various optimization problems. It is a state space search method in which all children of E-node is generated before any other lower node can become the E-node. Here, a BFS like state space search in which the live nodes are maintained using a queue is called **FIFO** and a D-Search like state space search in which the live nodes are maintained using a stack is called **LIFO**.

As in the case of backtracking, bounding functions are used to avoid the generation of subtrees that do not contain an answer node.

Travelling Salesman Problem(TSP)

Given a directed graph G=(V,E), the TSP problem is to find a tour or cycle that begins from a node, covers all the nodes of the graph exactly once and returns back to that node.

Suppose we have a graph with 4 nodes ie, n=4 then the different possibilities is shown in the state space tree S below:



Each leaf node L is a solution node and represents the tour defined by the path from root to L. Node 14 represents the tour i_0=1, i_1=3, i_2=4, i_3=2 and i_4=1.

Let the nodes be numbered from 1 to n. c_{ij}=cost of edge (i,j) and c_{ij}=∞ if (i,j) ∉E. Let |V|=n. We assume that every tour begins and ends at 1.

So the solution space S is given by S={1,π,1 / π=permutation of (2,3,….n). Then size of S = (n-1)!

The size of S can be reduced if (1,i_1,i_2,….i_{n-1},1) Є E iff <1$_j$,1$_{j+1}$> ЄE ,0<=j<=n-1 and i_0=i_n=1.

Using branch and bound method we try to reduce the search space. For this we find the lower bound of the travelling salesman cycle and at each level choose the node with least lower bound. To calculate the lower bound we try t find the reduced cost matrix.

A row or column is said to be reduced iff it contains atleast one zero and all remaining entries are non-negative. A matrix is reduced iff every row and column is reduced.

Example:

Suppose that we have an adjacency or cost matrix of a graph with 5 vertices ie, n=5.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

If t is chosen to be the minimum entry in row i or column j, then subtracting it from all entries in row i and column j introduces a zero into row i or column j. The total amount subtracted from the columns and rows is a lower bound on the length of a minimum cost tour and can be used as the ĉ value for the root of the state space tree.

Subtracting 10, 2, 2, 3, 4, 1and 3 from rows 1, 2, 3, 4 and 5 and columns 1 and 3 respectively of the matrix shown above yields the reduced matrix shown below. The total amount subtracted is 25. Hence al tours in the original graph have a length atleast 25.

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

We can have reduced cost matrix for each node in the TSP state space tree.

Let A be the reduced cost matrix for node R. Let S be a child of R, such that <R,S>=<i,j> ЄE. If S is not leaf then reduced cost matrix for S is given by

(1) Change all entries in row i and column j to ∞.
(2) Set A(j,1)= ∞
(3) Reduce all row and columns in the resulting matrix except for rows and columns containing only ∞.
Then ĉ(S)= ĉ(R)+A(i,j)+r where r=total subtracted in step (3) and (i,j)=matrix at stage R.
Applying **LCCB algorithm(Least cost branch bound)** the reduced cost matrix for each node and it's state space tree is shown below

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

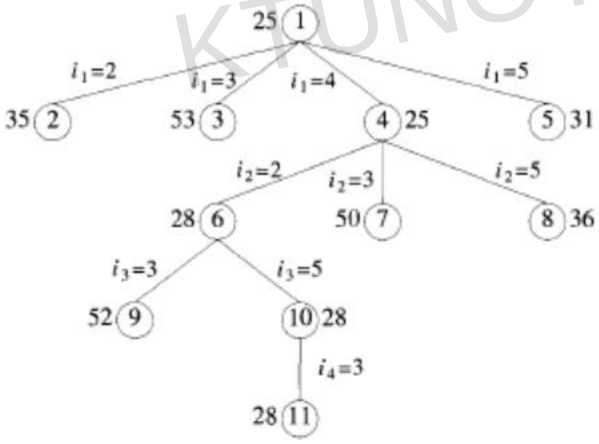
(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10



Numbers outside the node are $\hat{c}$ values

The LCCB algorithm terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour and the tour length is 28.

The worst case complexity of TSP problem is $O(n^2 2^n)$.

Tractable and Intractable problems

Almost all the algorithms we have studied thus far have been ***polynomial-time algorithms***,

On inputs of size n, their worst-case running time is $O(n^k)$ for some constant k. Whether *all* problems can be solved in polynomial time. The answer is no. For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time we allow. There are also problems that can be solved, but not in time $O(n^k)$ for any constant k. Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

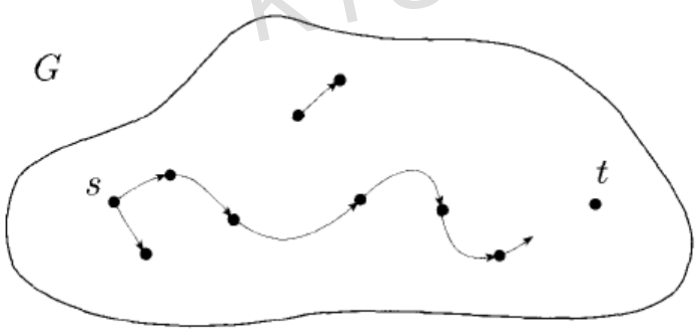
P and NP classes

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

EXAMPLES OF PROBLEMS IN P

When we analyze an algorithm to show that it runs in polynomial time, we need to do two things. First, we have to give a polynomial upper bound on the number of stages or steps that the algorithm uses when it runs on an input of length *n.* Then, we have to examine the individual stages of the algorithm to be sure that each can be implemented in polynomial time. When both tasks have been completed, we can conclude that the algorithm runs in polynomial time.

Problem 1: A directed graph G contains nodes *s* and t, as shown in the following figure. The *PATH* problem is to determine whether a directed path exists from *s* to t.
Let *PATH* = { *(G, s,* t) } *G* is a directed graph that has a directed path from s to t}.



To get a polynomial time algorithm for *PATH* we must do something that avoids brute force. One way is to use a graph-searching method such as breadth first search. Here, we successively mark all nodes in *G* that are reachable from *s* by directed paths of length 1, then 2, then 3, through m.

A polynomial time algorithm *M* for *PATH* operates as follows.
M = "On input *(G, s, t)* where G is a directed graph with nodes s and t:
1. Place a mark on node *s.*
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, *b)* is found going from a marked node a to an unmarked node *b*, mark node *b.*
4. If t is marked, *accept.* Otherwise, *reject."*

Now we analyze this algorithm to show that it runs in polynomial time. Obviously, stages 1 and 4 are executed only once. Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus the total number of stages used is at most 1 + 1 + m, giving a polynomial in the size of G. Hence *M* is a polynomial time algorithm for *PATH.*

Problem 2: Given two numbers, *RELPRIME* is the problem of testing whether two numbers are relatively prime. Thus
RELPRIME = {(x, y) |x and y are relatively prime}.

Two numbers are ***relatively prime*** if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself, whereas 10 and 22 are not relatively prime because both are divisible by 2.

We solve this problem with an ancient numerical procedure, called the ***Euclidean algorithm,*** for computing the greatest common divisor. The ***greatest common divisor*** of natural numbers x and y, written gcd(x, y), is the largest integer that evenly divides both x and y. For example, ged(18, 24) = 6. Obviously, x and y are relatively prime iff gcd(x, y) = 1. We describe the Euclidean algorithm as algorithm *E.* It uses the mod function, where x mod y is the remainder after the integer division of x by y.
The Euclidean algorithm *E* is as follows.

E = "On input (x, y), where x and y are natural numbers:
1. Repeat until y = 0:
2. Assign x ← *x* mod y.
3. Exchange x and y.
4. Output x."

Algorithm *R* solves *RELPRJME,* using *E* as a subroutine.

R = "On input (x, y), where x and y are natural numbers:
1. Run E on (x, y).
2. If the result is 1, *accept.* Otherwise, *reject."*

Clearly, if *E* runs correctly in polynomial time, so does *R.*

The values of x and y are exchanged every time stage 3 of E is executed, so each of the original values of *x* and y are reduced by at least half every other time through the loop. Thus the maximum number of times that stages 2 and 3 are executed is the lesser of 2 1og 2 *x* and 2 1og2 y. These logarithms are proportional to the lengths of the representations, giving the number of stages executed as O(n). Each stage of E uses only polynomial time, so the total running time is polynomial.

The class NP consists of those problems that are “verifiable” in polynomial time. What do we mean by a problem being verifiable? If we were somehow given a “certificate” of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the

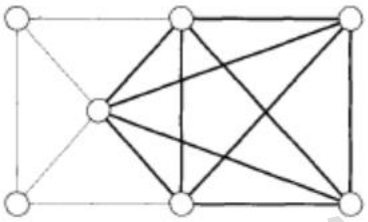
problem. For example, in the Hamiltonian cycle problem, given a directed graph G =(V,E), a certificate would be a sequence (v1, v2, v3,….v_n) of n vertices. We could easily check in polynomial time that (vi, vi+1) Є E for i = 1, 2, 3, …..n and that (v_n,v_1)Є E as well.

More generally, a problem is said to be in NP if there exists a verifier V for the problem. Given any instance I of problem P, where the answer is "yes", there must exist a certificate C such that, given the ordered pair (I,C) as input, V returns the answer "yes" in polynomial time. Furthermore, if the answer to I is "no", the verifier will return "no" with input (I,C) for all possible C. Note that V could return the answer "No" even if the answer to I is "yes", if C is not a valid witness.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate.

EXAMPLES OF PROBLEMS IN NP

Problem 1: *A clique* in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A *k-clique* is a clique that contains *k* nodes. Figure below illustrates a graph having a 5-clique.



The clique problem is to determine whether a graph contains a clique of a specified size. Let

CLIQUE {(G, k) |G is an undirected graph with a k-clique}.

The following is a verifier V for *CLIQUE.* The clique is the certificate.

V = "On input *((G, k),* c):
1. Test whether *c* is a set of k nodes in *G*
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept;* otherwise, *reject."*

Problem 2: Next we consider the *SUBSET-SUM* problem concerning integer arithmetic. In this problem we have a collection of numbers x**1…..x***k* and a target number t. We want to determine whether the collection contains a subcollection that adds up to t. Thus

SUBSET-SUM {(S, t)| S = {x1 ,..., Xk} and for some *{y1, .. ,Yl }* ⊆ {Xi, ... *,Xk },* we have ∑y_i= *t}*

For example, *({4,* 11,16,21, 27}! 25) E *SUBSET-SUM* because 4 + 21 = 25.

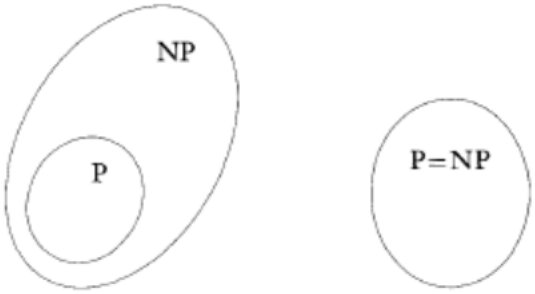
The following is a verifier V for SUBSET-SUM. The subset is the certificate.

V = "On input ((S, t), c):
1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.

3. If both pass, accept; otherwise, reject."

THE P VERSUS NP QUESTION

The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science. If these classes were equal, any polynomially verifiable problem would be polynomially solvable. Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success. Researchers also have tried proving that the classes are unequal, but that would entail showing that no fast algorithm exists to replace brute-force search. Doing so is presently beyond scientific reach. The following figure shows the two possibilities



POLYNOMIAL TIME REDUCIBILITY

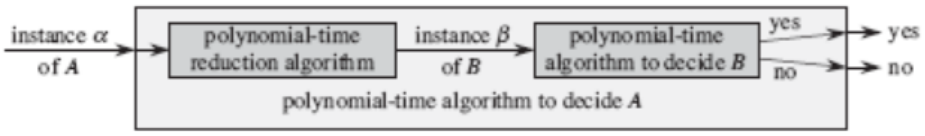
Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an ***instance*** of that problem. Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

(1) The transformation takes polynomial time.

(2) The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

We call such a procedure a polynomial-time ***reduction algorithm*** and, as the figure below shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for α as the answer for β.



By "reducing" solving problem A to solving problem B, we use the "easiness" of B to prove the "easiness" of A.

NP-completeness is about showing how hard a problem is rather than how easy it is, we use polynomial-time reductions in the opposite way to show that a problem is NP-complete. We could use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B. Suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist. Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B. Now we can use a simple proof by contradiction to show that no polynomial time algorithm can exist for B.

NP Complete Classes

If a language L1 is polynomial time reducible to another language L2, then it is denoted as $L1 \leq_p L2$.
A language L is ***NP-complete*** if
1. L Є NP, and
2. $L' \leq_p L$ for every L' Є NP

If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard.***

EXAMPLE NP-COMPLETE PROBLEMS and NP HARD PROBLEMS

Problem 1: *CLIQUE* is NP-complete.

Clique in an undirected graph G = (V, E) is a subset V' ⊆ V of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph.
CLIQUE = {(G, K) | G is a graph containing a clique of size k}

To show that CLIQUE Є NP, for a given graph G = (V, E), we use the set V '⊆ V of vertices in the clique as a certificate for G. We can check whether v' is a clique in polynomial time by checking whether, for each pair u,v Є V ', the edge (u,v) belongs to E.

We next prove that $3\text{-CNF-SAT} \leq_p \text{CLIQUE}$, which shows that the clique problem is NP-hard. The reduction algorithm begins with an instance of 3-CNF-SAT. Let Φ= C1 ^ C2 ^ ……^ Ck be a boolean formula in 3-CNF with k clauses. For r = 1, 2,….., k, each clause Cr has exactly three distinct literals $l 1$, l_2 , and l3. We shall construct a graph G such that Φ is satisfiable if and only if G has a clique of size k.

Consider an example, if we have
Φ = (x1 v ¬x2 v ¬x3) ^ (¬x1 v x2 v x3) ^ (x1 v x2 v x3)
then G is the graph shown in Figure below

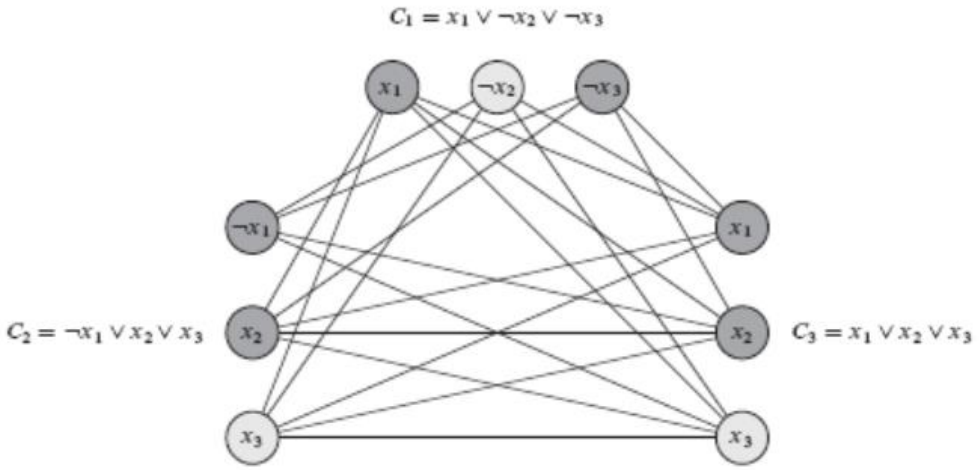
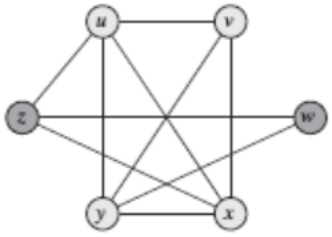


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

Suppose that Φ has a satisfying assignment. Then each clause Cr contains at least one literal l i that is assigned 1, and each such literal corresponds to a vertex v'$_i$. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique.
In the example of Figure 34.14, a satisfying assignment of Φ has x2 = 0 and x3 = 1. A corresponding clique of size k = 3 consists of the vertices corresponding to ¬x2 from the first clause, x3 from the second clause, and x3 from the third clause. Because the clique contains no vertices corresponding to either x1 or ¬x1, we can set x1 to either 0 or 1 in this satisfying assignment.
Hence Clique problem is NP Complete.

Problem 2: The Vertex Cover problem is NP Complete.
A ***vertex cover*** of an undirected graph G = (V, E) is a subset V '⊆ V such that if (u,v) Є E, then u Є V' or v Є V' (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The ***size*** of a vertex cover is the number of vertices in it.
For example, the graph in Figure below has a vertex cover {w, z} of size 2.



The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given graph.

VERTEX-COVER = {(G, k): graph G has a vertex cover of size k}

We first show that VERTEX-COVER Є NP. Suppose we are given a graph G = (V, E) and an integer k. The certificate we choose is the vertex cover V' ⊆ V itself. The verification algorithm affirms that |V'| = k, and then it checks, for each edge (u, v) Є E, that u Є V' or v Є V'. We can easily verify the certificate in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER. This reduction relies on the notion of the "complement" of a graph. Given an undirected graph G = (V, E), we define the ***complement*** of G as G' = (V, E'), where E' = {u,v): u, v ЄV, u ≠ v and (u,v) ∉ E}. In other words, G' is the graph containing exactly those edges that are not in G. Figure below shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

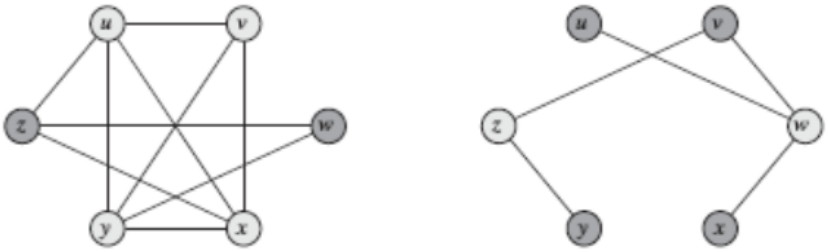


Figure 34.15 Reducing CLIQUE to VERTEX-COVER. **(a)** An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$. **(b)** The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$.

The reduction algorithm takes as input an instance (G, k) of the clique problem.
It computes the complement G', which we can easily do in polynomial time. The output of the reduction algorithm is the instance (G', |V|- k) of the vertex-cover problem.

Suppose that G has a clique V' ⊆ V with |V| = k. We claim that V – V' is a vertex cover in G'. Let (u,v) be any edge in E'. Then, (u,v) ∉ E, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, at least one of u or v is in V –V', which means that edge (u, v) is covered by V – V'. Since (u,v) was chosen arbitrarily from E, every edge of E is covered by a vertex in V- V'. Hence, the set V- V', which has size |V |- k, forms a vertex cover for G'.
Hence Vertex Cover problem is NP Complete.

Note: Clique and Vertex Cover Problems are also NP Hard. Here we need not prove that they are NP problems