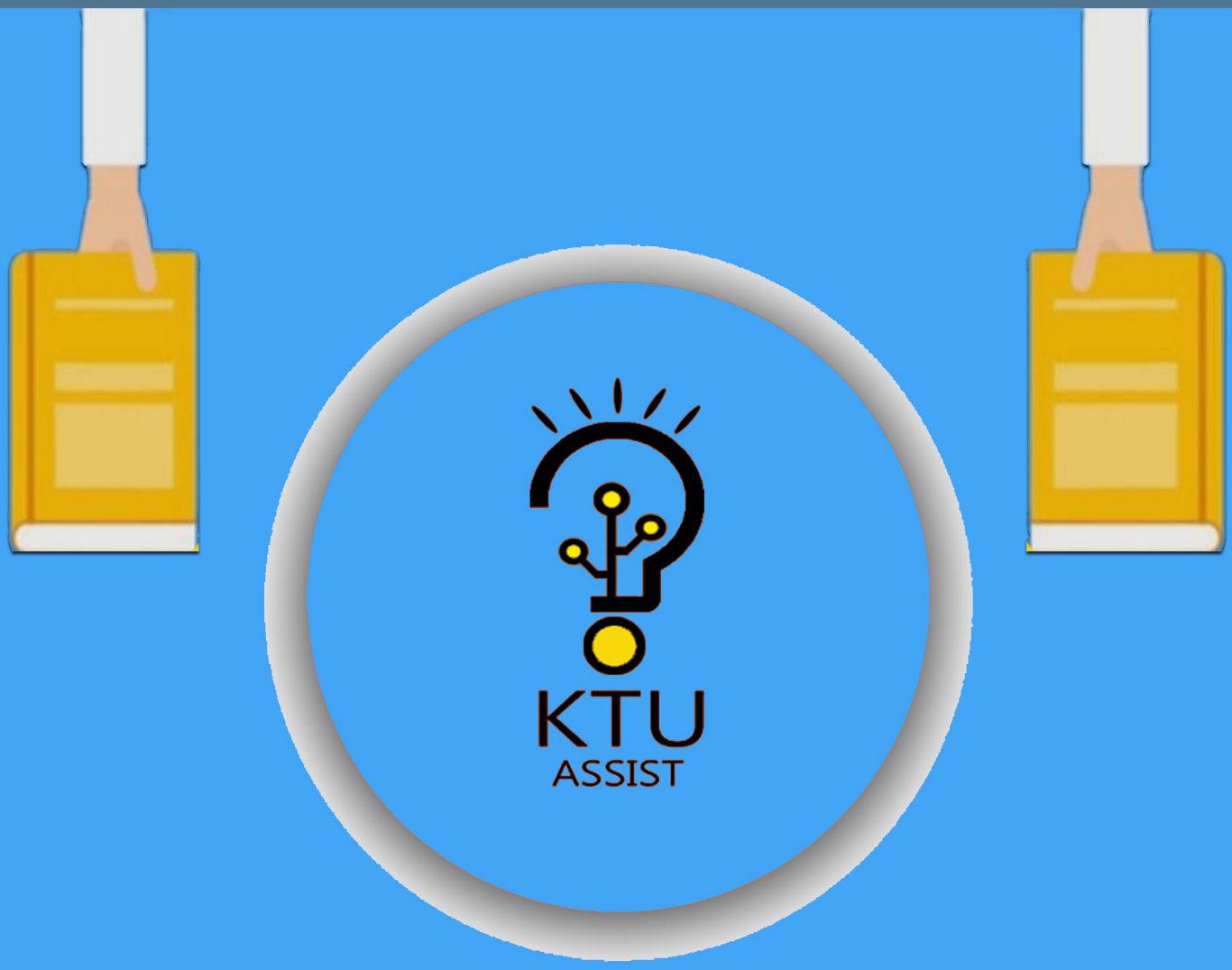
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



Module 6

Back Tracking, Branch and Bound & Complexity Theory

BACKTRACKING

Ques 1) What is backtracking? What are the different applications of backtracking?

Ans: Backtracking
Backtracking is a general algorithm for finding all (or some) solutions to some computational problem that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

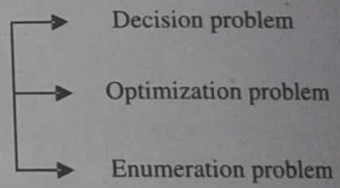
Backtracking can be applied only for problems which admit the concept of a **"partial candidate solution"** and a relatively quick test of whether it can possibly be completed to a valid solution.

Backtracking is an important tool for solving **constraint satisfaction problems**, such as crosswords, verbal arithmetic, and many other puzzles. It is often the most convenient (if not the most efficient) technique for parsing, for the knapsack problem and other combinatorial optimization problems. It is also the basis of the so-called logic programming languages such as Icon, Planner and Prolog.

The backtracking method in which the desired solution is expressed as an n-tuple $(x_1, x_2 \ldots x_n)$ is chosen from finite set S. The solution obtained, i.e., $(x_1, x_2 \ldots x_n)$ can either minimize or maximize or satisfy the criteria function. Let us denote criteria function by C.

In backtracking, the problem can be categorized into three categories:

1) **Decision Problem:** In decision problem we find "whether there is any feasible solution?"
2) **Optimization Problem:** In optimization problems we find "whether there exists any best solution?"
3) **Enumeration Problem:** In enumeration problem we find – all the possible feasible solutions.



The basic idea of backtracking is build-up a vector, one component at a time and to test whether the vector being formed has any chance of success.

Applications of Backtracking
Backtracking method have many applications in solving some problems which are:

1) Eight Queens Problem
2) Sum-of-Subsets
3) Graph Coloring
4) Hamiltonian Cycles
5) Knapsack Problem

Ques 2) Explain the control abstraction of backtracking.

Ans: Control Abstraction of Backtracking
The backtracking method in which the desired solution is expressed as an n-tuple $(x_1, x_2 \ldots x_n)$ is chosen from finite set S. The solution obtained, i.e., $(x_1, x_2 \ldots x_n)$ can either minimize or maximize or satisfy the criteria function. Let us denote criteria function by C.

The **basic idea of backtracking** is build-up a vector, one component at a time and to test whether the vector being formed has any chance of success.

Pseudocode for Backtracking
A general pseudocode for backtracking is given below:

Pseudocode: Backtrack (a, T, B)
//Problem Description: This is recursive backtracking algorithm
//Input : a [k] is a solution vector. On entering (k – 1)
//remaining next values can be computed.
//T (a_1, a_2, …a_k) be the set of all values for $a_{(i+1)}$, such that
// (a_1, a_2, …a_{i+1}) is a path to a problem state.
// B_{i+1} is a bounding function such that if B_{i+1} (a_1, a_2, …a_{i+1}) is

false for a path
//Output: (a_1, a_2, …a_{i+1}) from root node to a problem state then path cannot be extended to reach an answer node
for (each a_k that belongs to T (a_1, a_2, …a_{k-1})
{
if (B_k (a_1, a_2, … a_k) = true) then // Generation of feasible solutions
{
if ((a_1, a_2, … a_k) is a path to answer node then
write (a[1], a[2], …a[k]); //
Printing the answer nodes as solution to given problem
if (k<n) then
Backtrack (k+1); // find the next set.
}
}

Ques 3) Explain the terminologies related to backtracking.

Ans: Terminologies Related to Backtracking

1) **Solution Space:** Backtracking algorithm determines the solution by systematically searching the solution space. The solution space means set of all possible feasible solutions, for a given problem. In backtracking some bounding functions is required when the solution is search. The search made in backtracking is generally of depth first search manner. The solutions obtained in backtracking need to satisfy some constraints. The constraints are of two types:
   i) **Explicit Constraints:** These are the set of rules which restrict each vector element to be chosen from a given set.
   ii) **Implicit Constraints:** These are the set of rules that determine which of the tuples in solution space satisfies the criterion functions.

2) **Exhaustive Search:** It is basic and trivial search routine in which all possible solutions to a given problem are produced. In this search, it is checked if the problem is solved or not. And the search is continued until a correct solution is generated.

   The exhaustive search algorithm produces the entire solution space for the problem.

3) **State Space Tree:** In backtracking technique while solving a given problem, a tree is constructed based on choices made. Such a tree with all possible solutions is called state-space tree.

4) **Promising and Non-Promising Nodes:** A node in a state space tree is said to be promising if it corresponds to a partially constructed solution from which a complete solution can be obtained. The nodes which are not promising for solution in a state space tree are called non-promising nodes.

Ques 4) What are the procedural parameters of backtracking? Write the algorithm for backtracking.

Ans: Procedural Parameters of Backtracking
In order to apply backtracking to a specific class of problems, one must provide the **data P** for the particular instance of the problem that is to be solved, and six procedural parameters, root, reject, accept, first, next, and output. These procedures should take the instance data P as a parameter and should do the following:

1) **root (P):** Return the partial candidate at the root of the search tree. The **reject procedure** should be a Boolean-valued function that returns true only if it is certain that no possible extension of c is a valid solution for P. If the procedure cannot reach a definite conclusion, it should return false. An incorrect true result may cause the bt procedure to miss some valid solutions. The procedure may assume that reject (P,t) returned false for every ancestor t of c in the search tree.

   On the other hand, the **efficiency** of the backtracking algorithm depends on reject returning true for candidates that are as close to the root as possible. If reject always returns false, the algorithm will still find all solutions, but it will be equivalent to a brute-force search.

2) **reject (P, c):** Return true only if the partial candidate c is not worth completing.

3) **accept (P, c):** Return true if c is a solution of P, and false otherwise. The **accept procedure** should return true if c is a complete and valid solution for the problem instance P, and false otherwise. It may assume that the partial candidate c and all its ancestors in the tree have passed the reject test.

   The general pseudo-code above does not assume that the valid solutions are always leaves of the potential search tree. In other words, it admits the possibility that a valid solution for P can be further extended to yield other valid solutions.

4) **first (P, c, s):** Generate the first extension of candidate c.

5) **next (P,s):** Generate the next alternative extension of a candidate, after the extension s.

6) **output (P, c):** Use the solution c of P, as appropriate to the application.

The backtracking algorithm reduces then to the call bt (root (P)), where bt is the following recursive procedure:

Algorithm: bt(c)

Step 1: if reject(P, c) then return

Step 2: if accept(P, c) then output(P, c)

Step 3: s ← first(P, c)

Step 4: while s ≠ Λ do

Step 5: bt(s)

Step 6: s ← next(P,s)

The first and next procedures are used by the backtracking algorithm to enumerate the children of a node c of the tree, that is, the candidates that differ from c by a single extension step. The call first (P, c) should yield the first child of c, in some order; and the call next (P, s) should return the next sibling of node s, in that order. Both functions should return a distinctive "null" candidate, denoted here by 'Λ', if the requested child does not exist.

Together, the root, first, and next functions define the set of partial candidates and the potential search tree. They should be chosen so that every solution of P occurs somewhere in the tree, and no partial candidate occurs more than once. Moreover, they should admit an efficient and effective reject predicate.

Ques 5) Discuss N Queen's problem using backtracking.

Or

Explain the eight queen's problem using backtracking approach.

Ans: N Queen's Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. By attacking, one means that no two queens can be on the same row, column or diagonal. This problem can be solved using a stack and backtracking logic. The general approach to solve this problem is to place a queen on the board, and then analyse all the other positions to check whether there is any queen that could attack the new queen. If such queen exists, we try another position. If an appropriate position is found, we push that position onto the stack.

To understand this approach, consider the 8 queen's problem in which the aim is to place eight queens on a 8 × 8 chessboard.

Eight Queens Problem

In backtracking, the classic problem of placing eight queens on a chessboard in such a way that none of them threatens any of the others. A queen threatens the squares in the same row, in the same column or on the same diagonals.

The most obvious way to solve this problem consists of trying systematically all the ways of placing eight queens on a chessboard, checking each time to see whether a solution has been obtained. This approach is of no practical use, even with a computer, since the number of positions we would have to check is $\binom{64}{8} = 4426165368$.

The first improvement consists of never putting more than one queen on any given row. This reduces the computer representation of the chessboard to a vector of eight elements, each giving the position of the queen in the corresponding row.

For example, the vector (3, 1, 6, 2, 8, 6, 4, 7) represents the position where the queen on row 1 is in column 3, the queen on row 2 is in column 1 and so on. This particular position is not a solution to our problem since the queens in rows 3 and 6 are in the same column and also two pair of queens lie on the same diagonal. Using this representation, one can write an algorithm using eight nested loops.

Program queens1

Step 1: for $i_1 \leftarrow 1$ to 8 do

Step 2: for $i_2 \leftarrow 1$ to 8 do

Step 3: …

Step 4: for $i_8 \leftarrow 1$ to 8 do

Step 5: sol ← $[i_1, i_2, \ldots, i_8]$

Step 6: if solution(sol) then write sol

Step 7: stop

Step 8: write "there is no solution"

The number of positions to be considered is reduced to 8^8 = 16777216, although in fact the algorithm finds a solution and stops after considering only 1299852 positions.

Representing the chessboard by a vector prevents ever trying to put two queens in the same row. Once we have realized this, it is natural to be equally systematic in our use of the columns. Hence now represent the board by a vector of eight different numbers between 1 and 8, i.e., by a permutation of the first eight integers. This yields the following algorithm.

Program queens2

Step 1: sol ← initial – permutation

Step 2: while sol ≠ final-permutation and not solution (sol) do

Step 3: sol ← next-permutation

Step 4: if solution (sol) then write sol

Step 5: else write "there is no solution"

Backtracking Approach in Eight Queens Problem

As a first step, one reformulates the eight queens problem as a tree searching problem.

A vector V[1..k] of integers between 1 and 8 is said to be **k-promising,** for $0 \le k \le 8$, if none of the k queens placed in positions (1, V[1]), (2, V[2]),…,(k, V[k]) threatens any of the others. Mathematically, a vector V is k-promising if,

for every pair of integers i and j between 1 and k with $i \neq j$, one have $V[i]-V[j] \notin \{i-j, 0, j-i\}$. For $k \leq 1$, any vector V is k-promising. Solutions to the eight queens problem correspond to vectors that are 8-promising.

Let N be the set of **k-promising vectors**, $0 \leq k \leq 8$. Let G = (N, A) be the directed graph such that $(U, V) \in A$ if and only if there exists an integer k, $0 \leq k < 8$, such that

1) U is k-promising,
2) V is (k + 1)-promising, and
3) U[i]= V[i] for every $i \in [1..k]$.

This graph is a tree. Its root is the empty vector corresponding to k = 0. Its leaves are either solution (k = 8) or they are dead ends (k < 8) such as [1, 4, 2, 5, 8]: in such a position it is impossible to place a queen in the next row without threatening atleast one of the queens already on the board. The solutions to the eight queen's problem can be obtained by exploring this tree. Depth-first search is the obvious method to use, particularly if one requires only one solution.

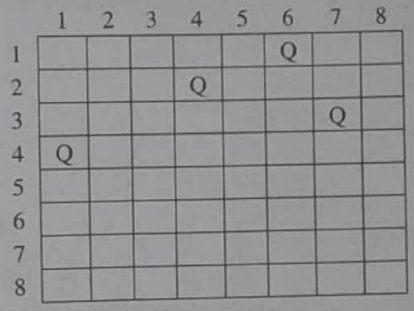
To print all the solutions to the eight queens problem, call queens (0, ∅, ∅, ∅).

Algorithm: queens (k, col, diag45, diag135)

Step 1: {sol[1..k] is k-promising,
Step 2: col = {sol[i]|1 ≤ i ≤ k},
Step 3: diag45 = {sol[i] – i + 1|1 ≤ i ≤ k}, and
Step 4: diag135 = {sol[i] + i – 1|1 ≤ i ≤ k}}
Step 5: if k = 8 then {an 8-promising vector is a solution}
Step 6: write sol
Step 7: else {explore (k + 1)-promising extension of sol}
Step 8: for j ← 1 to 8 do
Step 9: if j ∉ col and j – k ∉ diag45 and j + k ∉ diag 135
Step 10: then sol [k + 1] ← j
Step 11: {sol[1 .. k + 1] is (k + 1)-promising}
Step 12: queens (k + 1, col ∪ {j},
Step 13: diag45 ∪ {j – k}, diag 135 ∪ {j + k})

Ques 6) Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1).

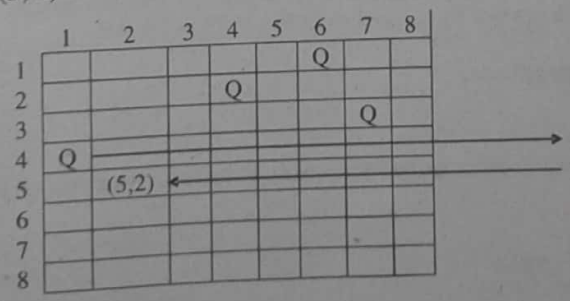
Ans: As the feasible sequence is given, we will place the queen's accordingly and then try out the other remaining places.



The diagonal conflicts can be checked by following formula:

Let $P_1 = (i, j)$ and $P_2 = (k, 1)$ are two positions. Then P_1 and P_2 are the positions that are on the same diagonal, if $i + j = k + 1$ or $i - j = k - 1$

Now, if next queen is placed on (5, 2) then,



It can be summarized below:

1	2	3	4	5	6	7	8	
6	4	7	1					Start
6	4	7	1	2				As 4 – 1 = 5 – 2 conflict occurs.
6	4	7	1	3				5 – 3 ≠ 4 – 1 or 5 + 3 ≠ 4 + 1 ∴ feasible
6	4	7	1	3	2			As 5 + 3 = 6 + 2. It is not feasible
6	4	7	1	3	5			Feasible
6	4	7	1	3	5	2		Feasible
6	4	7	1	3	5	2	8	Lists ends and we have got feasible sequence.

The 8-queen's on 8 × 8 board with this sequence is:

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

8-queen with feasible solution (6, 4, 7, 1, 3, 5, 2, 8)

Ques 7) Discuss 4 queen's problem.

Ans: 4-Queen's Problem

The n-queens problem is a generalisation of the 8-queens problem. Now n queens are to be placed on an n × n chessboard so that no two attack; that is, no two queens are on the same row, column, or diagonal. The solution space consists of all n! permutations of the n-tuple (1, 2,... ,n).

Figure 6.1 shows a possible tree organisation for the case n = 4. A tree such as this is called a permutation tree. The edges are labelled by possible values of x_i. Edges from level 1 to level 2 nodes specify the values for x_1. Thus, the leftmost subtree contains all solutions with $x_1 = 1$ and $x_2 = 2$, and so on. Edges from level i to level i +1 are labelled with the values of x_i. The solution space is defined by all paths from the root node to a leaf node. There are 4! = 24 leaf nodes in the tree of **figure 6.1.**

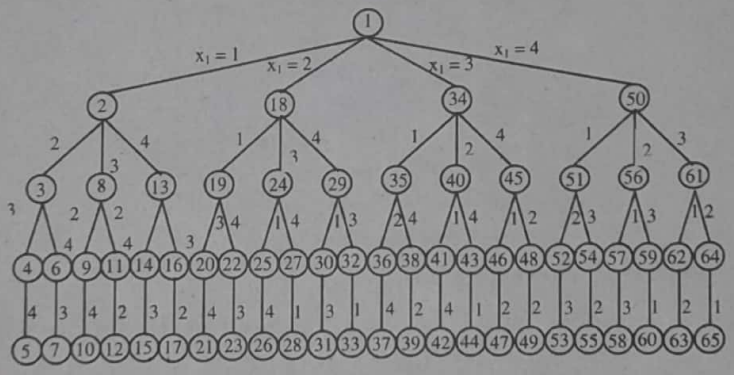


Figure 6.1: Tree Organisation of the 4-Queens Solution Space. Nodes are Numbered as In Depth First Search

Ques 8) Explain the sum of subset problems.

Ans: Sum of Subsets Problems

Given a set of non-negative integers, and a value **sum**, determine if there is a **subset** of the given set with **sum** equal to given **sum**.

For example, set[] = {3, 34, 4, 12, 5, 2}, sum = 9

Output: True //There is a subset (4, 5) with sum 9.

Figures 6.2 and **6.3** show a possible tree organisation for each of these formulations for the case n = 4. The tree of **figure 6.2** corresponds to the variable tuple size formulation. The edges are labelled such that an edge from a level i node to a level i + 1 node represents a value for x_i. At each node, the solution space is partitioned into sub-solution spaces. The solution space is defined by all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints. The possible paths are () (this corresponds to the empty path from the root to itself), (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the leftmost subtree defines all subsets containing w\, the next subtree defines all subsets containing w_2, but not w_1, and so on.

The tree of **figure 6.3** corresponds to the fixed tuple size formulation. Edges from level i nodes to level i + 1 nodes are labelled with the value of x_i, which is either zero or one. All paths from the root to a leaf node define the solution space. The left subtree of the root defines all subsets containing w_1 the right subtree defines all subsets not containing w_1, and so on. Now there are 2^4 leaf nodes which represent 16 possible tuples.

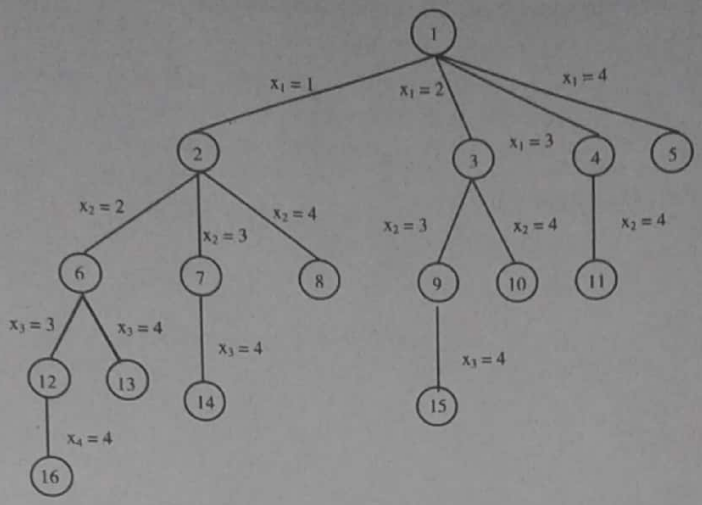


Figure 6.2: Possible Solution Space Organisation for the Sum of Subsets Problems. Nodes are Numbered as in Breadth-First Search

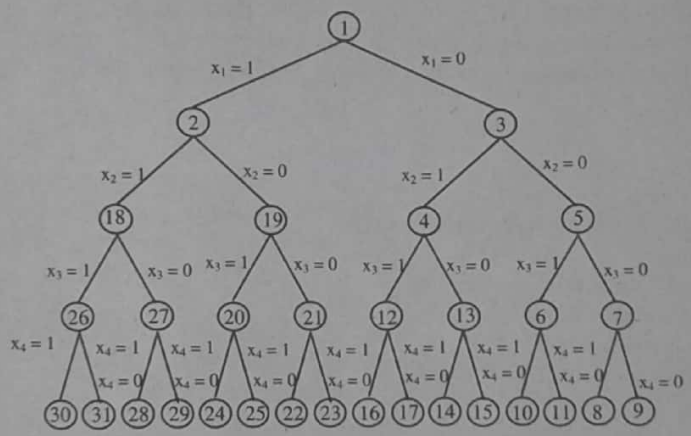


Figure 6.3: Another Possible Organisation for the Sum of Subsets Problems. Nodes are Numbered as in D-Search

Ques 9) Discuss the 0/1 knapsack problem using backtracking approach. Explain with an example.

Ans: 0/1 Knapsack Problem Using Backtracking Approach

Let consider the 0/1 knapsack optimisation problem. Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that,

$$\sum_{1 \le i \le n} w_i x_i \le m \text{ and } \sum_{1 \le i \le n} p_i x_i \text{ is maximised} \qquad \ldots\ldots(1)$$

The x_i's constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's. Thus the solution space is the same as that for the sum of subsets problem. Two possible tree organisations are possible. One corresponds to the fixed tuple size formulation (**figure 6.3**) and the other to the variable tuple size formulation (**figure 6.2**). Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees. Regardless of which is used, bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far, then that live node can be killed.

If at node Z the values of x_i, $1 \le i \le k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \le x_i \le 1$ for $k+1 \le i \le n$ and using the greedy algorithm to solve the relaxed problem. Function Bound(cp, cw, k) (**Algorithm 1**) determines an upper bound on the best solution obtainable by expanding any node Z at level k + 1 of the state space tree. The object weights and profits are w[i] and p[i]. It is assumed that $p[i]/w[i] \ge p[i+1]/w[i+1]$, $1 \le i < n$.

Algorithm 1: Bounding Function

```
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];
10) if(c<m) then b := b + p[i];
11) else return b + (1 – (c – m)/w[i]) × p[i];
12) }
13) return b;
14) }
```

From Bound it follows that the bound for a feasible left child of a node Z is the same as that for Z. Hence, the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node. The resulting algorithm is BKnap (**Algorithm 2**). It was obtained from the recursive backtracking schema. Initially set fp: = –1. This algorithm is invoked as,

```
BKnap(1, 0, 0);
```

Algorithm 2: Backtracking Solution to the 0/1 Knapsack Problem

```
1)  Algorithm BKnap (k, cp, cw)
2)  // m is the size of the knapsack; n is the number of weights
3)  // and profits. w[] amd 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) }
```

When $fp \neq -1$, $x[i]$, $1 \leq i \leq n$, is such that $\sum_{i=1}^{n} p[i]x[i] = fp$. In lines 8 to 18 left children are generated. In line 20, Bound is used to test whether a right child should bo generated. The path $y[i]$, $1 \leq i \leq k$, is the path to the current node. The current weight $cw = \sum_{i=1}^{k-1} w[i]y[i]$ and $cp = \sum_{i=1}^{k-1} p[i]y[i]$. In lines 13 to 17 and 23 to 27 the solution vector is updated if need be.

So far, all our backtracking algorithms have worked on a static state space tree. We now see how a dynamic state space tree can be used to the knapsack problem. One method for dynamically partitioning the solution space is based on trying to obtain an optimal solution using the greedy algorithm. We first replace the integer constraint $x_i = 0$ or 1 by the constraint $0 \leq x_i \leq 1$. This yields the relaxed problem

$$\max \sum_{1 \leq i \leq n} p_i x_i \text{ subject to } \sum_{1 \leq i \leq n} w_i x_i \leq m \qquad \ldots..(2)$$

$$0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

If the solution generated by the greedy method has all x_i's equal to zero or one, then it is also an optimal solution to the original 0/1 knapsack problem. If this is not the case, then exactly one x_i will be such that $0 < x_i < 1$. We partition the solution space of equation (1) into two subspaces. In one $x_i = 0$ and in the other $x_i = 1$. Thus the left subtree of the state space tree will correspond to $x_i = 0$ and the right to $x_i = 1$. In general, at each node Z of the state space tree the greedy algorithm is used to solve equation (2) under the added restrictions corresponding to the assignments already made along the path from the root to this node. In case the solution is all integer, then an optimal solution for this node has been found. If not, then there is exactly one x_i such that $0 < x_i < 1$. The left child of Z corresponds to $x_i = 0$, and the right to $x_i = 1$.

The justification for this partitioning scheme is that the non-integer x_i is what, prevents the greedy solution from being a feasible solution to the 0/1 knapsack problem. So, we would expect to reach a feasible greedy solution quickly by forcing this x_i to be integer. Choosing left, branches to correspond to $x_i = 0$ rather than $x_i = 1$ is also justifiable. Since the greedy algorithm requires $P_j / w_j \geq p_{j+1} / w_{j+1}$, we would expect most objects with low index (i.e., small j and hence high density) to be in an optimal filling of the knapsack. When x_i is set to zero, we are not preventing the greedy algorithm from using any of the objects with $j < i$ (unless x_j has already been set to zero). On the other hand, when x_i is set to one, some of the x_j's with $j < i$ will not be able to get into the knapsack- Therefore we expect to arrive at an optimal solution with $x_i = 0$. So we wish the backtracking algorithm to try this alternative first. Hence the left subtree corresponds to $x_i = 0$.

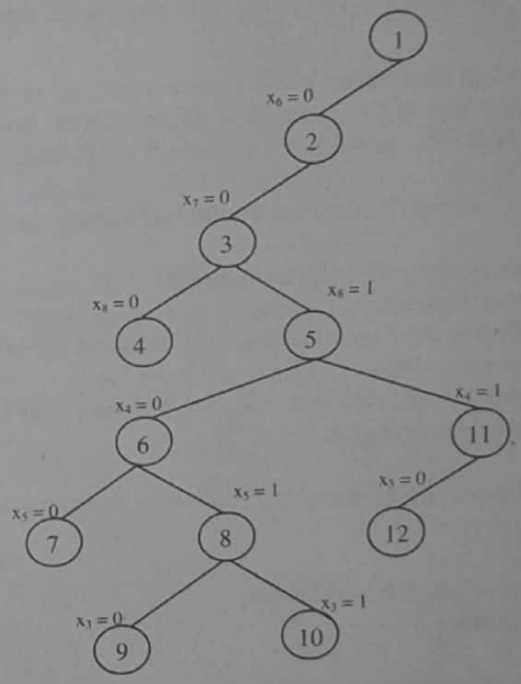


Figure 6.4: Part of the Dynamic State Space Tree Generated

Example: Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: p = {11 ,21, 31, 33, 43, 53, 55, 65}, w = {1, 11, 21, 23, 33, 43, 45, 55} m = 110, and n = 8. The greedy solution corresponding to the root node (i.e., Equation (2)) is x = {1, 1, 1, 1, 1, 21/45 0, 0}. Its value is 164.88. The two subtrees of the root correspond to $x_6 = 0$ and $x_6 = 1$, respectively (**Figure 6.4**).

The greedy solution at node 2 is x = {1, 1, 1, 1, 1, 0, 21/45, 0}. Its value is 164.66. The solution space at node 2 is partitioned using $x_7 = 0$ and $x_7 = 1$. The next E-node is node 3. The solution here has $x_8 = 21/55$. The partitioning now is with $x_8 = 0$ and $x_8 = 1$. The solution at node 4 is all integer so there is no need to expand this node further. The best solution found so far has value 139 and x = {1, 1, 1, 1, 1, 0, 0, 0,}. Node 5 is the next E-node. The greedy solution for this node is x = {1, 1, 1, 22/23, 0, 0, 0, 1}. Its value is 159.56. The partitioning is now with $x_4 = 0$ and $x_4 = 1$. The greedy solution at node 6 has value 156.66 ad $x_5 = 2/3$. Next, node 7 becomes the E-node. The solution here is {1, 1, 1, 0, 0, 0, 0, 1}. Its value is 128. Node 7 is not expanded as the greedy solution here is all integer. At node 8 the greedy solution has value 157.71 and $x_3 = 4/7$. The solution at node 9 is all integer and has value 140. The greedy solution at node 10 is {1, 0, 1, 0, 1, 0, 0, 1}. Its value is 150. The next E-node is 11. Its value is 159.52 and $x_3 = 20/21$. The partitioning is now on $x_3 = 0$ and $x_3 = 1$. The remainder of backtracking process on this knapsack instance is left as an exercise.

BRANCH-AND-BOUND

Ques 10) What is branch and bound? What are the basic terminologies of branch and bound?

Ans: Branch and Bound
Branch and Bound is a general algorithmic method for finding optional solutions of various optimization problems. Branch and bounding is a general optimization technique that applies where the greedy method and dynamic programming fail.

Branch and Bound is a state space search method in which all the children of a node are generated before expanding any of its children.

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. Two graph of such strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalize to branch-and-bound strategies.

Branch-and- Bound Terminology

1) **Live Node:** A node which has been generated and all of whose children have not yet been generated is called a live node. The live node whose children are currently being generated is called the E-node.
2) **E-Node:** It is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3) **Dead Node:** A dead node is a generated node which is not to be expanded further or all of whose children has been generated.
4) **Bounding Functions:** Bounding functions are used to kill live nodes without generating all their children. This is done carefully that at the conclusion of the process atleast one answer node is always generated or all answer nodes are generated if the problem requires finding all solutions.

Ques 11) Discuss the general method of branch and bound technique with example.

Ans: General Method of Branch and Bound
In branch-and-bound:

1) A BFS-like state space search will be called FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue).
2) A D-search-like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack).

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

Pseudocode: Branch and Bound

```
BranchBound()
{
 //E is a node Pointer;
E<- new(node);  //this is the root node which is the dummy start node
/* H is heap for all the live nodes. H is the min-heap for minimization problems. H is the max-heap for maximization problems. */

while(true)
{
        if (E is a final leaf) then
        {
                //E is an optional solution
                print (path from E to the root);
                return;
        }
        Expand(E);
        If (H is empty) then // if no element is present in heap
        {
```

```
            print (there is no solution);
            return;
        }
        E<- delete_top(H);}}
```

Following is an algorithm named Expand is for generating state space tree:

```
Expand(E)
{
        Generate all the children of E;
        Compute the approximate cost value of each child;
        Insert each child into the heap H;
}
```

For example, [4-queens] A FIFO branch-and-bound algorithm would search the state space tree for the 4-queens probability.

1) Initially, there is only one live node, node 1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node.
2) It is expanded and its children, nodes 2, 18, 34, and 50, are generated. These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively.
3) The only live nodes now are nodes 2, 18, 34 and 50. If the nodes are generated in this order, then the next E-node is node 2. It is expanded and nodes 3, 8, and 13 are generated.
4) Node 3 is immediately killed.
5) Nodes 2 and 18 are added to the queue of live nodes.
6) Node 18 becomes the next E-node.
7) Nodes 19, 24, and 29 are generated.
8) Nodes 19 and 24 are killed as a result of the bounding functions.
9) Node 29 is added to the queue of live nodes.
10) The E-node is node 34.

Figure 6.5 shows the portion of the tree that is generated by a FIFO branch-and-bound search.

i) Nodes that are killed as a result of the bounding functions have a "B" under them.
ii) Numbers inside the nodes correspond to the numbers.
iii) Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound.
iv) At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54.

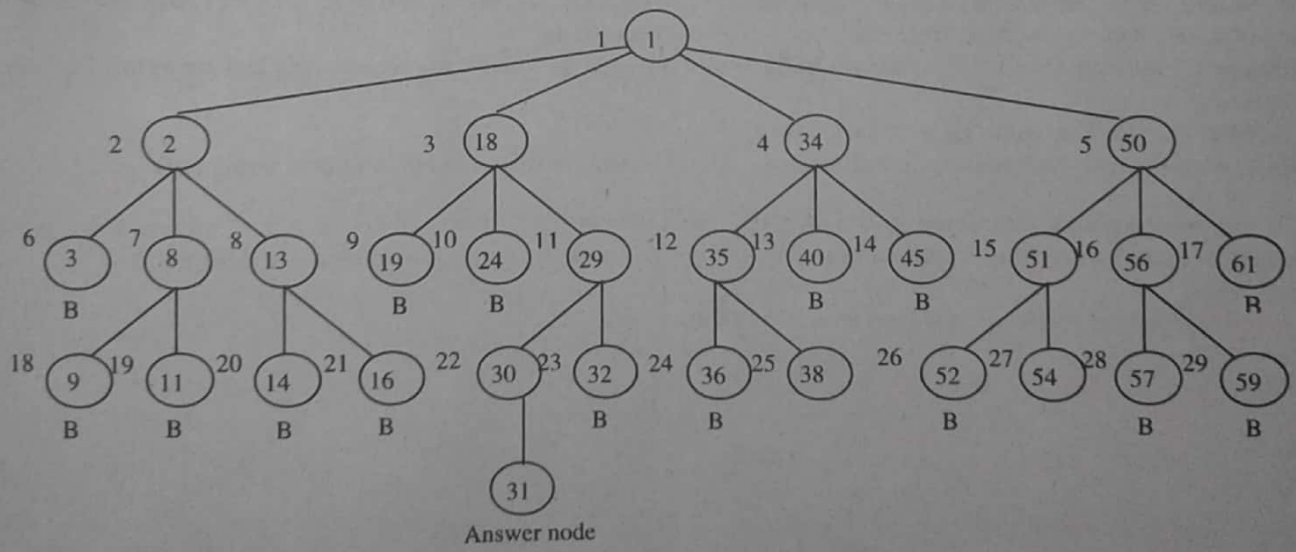


Figure 6.5

Ques 12) Explain the travelling salesman problem using branch and bound. Also explain how to find the Reduced Cost Matrix.

Ans: Traveling Salesman using Branch and Bound

Let G = (V, E) be a directed graph defining an instance of the traveling salesman problem. Let m_{ij} be the cost of the edge (i, j). Assuming $m_{ij} = \infty$ and $|V| = N$ and assume that the tour starts from some designated city (called city 1) and the salesman visits all the other cities once and no city is visited twice and returns to the designated city. So the solution space S consists of $|P|$ points, where P is the set of permutation of (2, 3, 4, …, N).

Solution Space of Traveling Salesman

The size of S is obviously (N – 1)!. The solution space is represented by a state space tree. Three cost functions c, I and u are defined where $l(i) \le c(i) \le u(i)$ for all nodes i of the state space tree. The cost c is such that the solution node with the least value of c corresponds to a shortest tour in G. One possible choice for c is

$$c(i) = \begin{cases} \text{Length of tour defined by the path from the root to i, it i is a leaf} \\ \text{Cost of a minimum cost leaf in the subtree i, if i is not a leaf} \end{cases}$$

I(i) = The length of the path defined at node i.

A better value of I can be obtained by using the reduced cost matrix corresponding to G. A row (column) is said to be reduced if it contains atleast one zero and all the remaining entries are non-negative. A matrix is reduced if every column is reduced. Let us consider the graph of **figure 6.6** and its cost matrix.

Since every tour on this graph includes exactly one edge (i, j) with i = k, $1 \le k \le 4$ and exactly one edge (i, j) with j = k, $1 \le k \le 4$, subtracting a constant θ from every entry in a column or row of the cost matrix reduces the length of every tour by exactly θ.

A minimum cost tour remains a minimum cost tour following the reduction. If θ is chosen to be the minimum entry in row i (column j), then subtracting it from all the entries in row i (column j) introduces a zero into row i (column j). Repeating this as often as needed, the cost matrix can be reduced. 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 I-value for the root of the state space tree.

	A	B	C	D
A	∞	12	5	7
B	11	∞	13	6
C	4	9	∞	18
D	10	3	2	∞

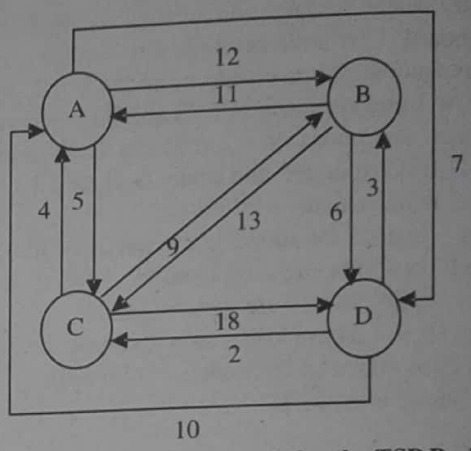


Figure 6.6: Example Graph for the TSP Problem

Reduced Cost Matrix

A reduced cost matrix is associated with every node in the traveling salesman state space tree. Let M be the reduced cost matrix for node A and let B be a child of A such that the tree edge (A, B) corresponds to inclusion of edge (i, j) in the tour. If B is not a leaf, then the reduced cost matrix for B may be obtained as:

1) Change all the entries in row i, column j of M to ∞. This excludes use of any more edges leaving vertex i or entering vertex j.
2) Set M(j, 1) to ∞. This excludes use of the edge (j, 1).
3) Reduce all the rows and columns in the resulting matrix except for rows and columns containing only ∞.

If T is the total amount subtracted in the step (3), then I(B) = I(A) + M(i, j) + T. For leaf nodes, I = c is easily computed as each leaf defines a unique tour. For the upper bound u, we assume u(i) = ∞ for all nodes i.

The search tree for the graph of **figure 6.6** is shown in **figure 6.7.**

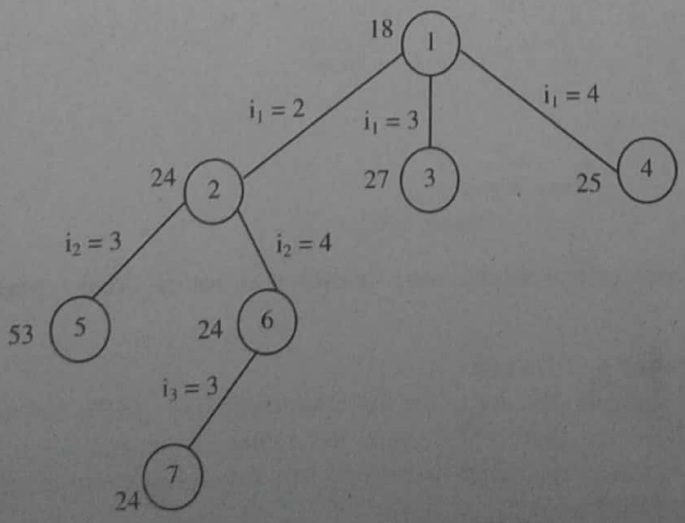


Figure 6.7: Search Tree for the TSP Problem

The root of the search tree (node A) of the graph has the minimum cost 18. This is found by reducing the cost matrix M by subtracting 5 from row 1, 6 from row 2, 4 from row 3, 2 from row 4 and subtracting 1 from column 2. The total amount subtracted is 18. The minimum cost of the tour is atleast 18 and the reduced cost matrix is:

	A	B	C	D
A	∞	6	0	2
B	5	∞	7	0
C	0	4	∞	14
D	8	0	0	∞

The reduced cost matrix at node 2 of the search tree following path (A, B) is

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	7	0
C	0	∞	∞	14
D	8	∞	0	∞

The minimum cost I(2) = I(1) + M(1, 2) + T = 18 + 6 + 0 = 24. The reduced cost matrix at node 3 of the search tree following the path (A, C) is:

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	0
C	∞	0	∞	10
D	3	0	∞	∞

The minimum cost I(3) = I(1) + M(1, 3) + T = 18 + 2 + 5 = 25. The reduced cost matrix at node 4 in the search tree following the path (A, D) is:

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	2	∞
C	0	4	∞	∞
D	∞	0	0	∞

The minimum cost I(4) = I(1) + M(1, 4) + T = 18 + 9 + 0 = 27. At this stage, find that the path (A, B) is the most promising with the minimum cost 24 and so we expand the node 2.

The node 5 is generated following the path (A, B, C). The reduced cost matrix at node 5 is:

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	∞	∞	0
D	0	∞	∞	∞

The minimum cost I(5) = I(2) + M(2, 3) + T = 24 + 7 + 22 = 53. The reduced cost matrix at node 6 in the search tree following the path (A, B, D) is:

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	∞

The minimum cost I(6) = I(2) + M(2, 4) + T = 24 + 0 + 0 = 24. The most promising node to expand now is node 6 and we generate node 7 following the path (A, B, C, D). The reduced cost matrix at node 7 is:

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	0	∞	∞	∞
D	∞	∞	∞	∞

The minimum cost I(7) = I(8) + M(4, 3) + T = 24 + 0 + 0 = 24. So the optimal tour of the traveling salesman will be A to B, B to D, D to C and C to A costing 24 as shown in **figure 6.8**.

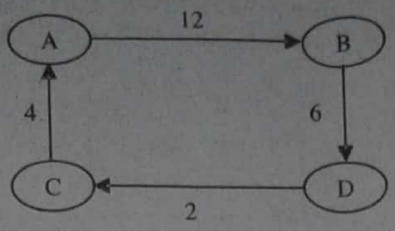


Figure 6.8: Optimal Tour of the TSP

Ques 13) What is difference between backtracking and branch and bound technique?

Ans: Comparison of Backtracking and Branch-and-Bound Technique
The **table 6.1** describes the difference between backtracking and Branch and Bound:

Table 6.1

Backtracking	Branch and Bound
Solution for backtracking is traced using depth first search.	In this method, it is not necessary to use depth first search for obtaining the solution, even the breadth first search, best first search can be applied.
Typically decision problems can be solved using backtracking.	Typically optimization problems can be solved using branch and bound.
While finding the solution to the problem bad choices can be made.	It proceeds on better solutions. So there cannot be a bad solution.
The state space tree is searched until the solution is obtained.	The state space tree needs to be searched completely as there may be chances of being an optimum solution anywhere in state space tree.
Application of backtracking is – Knapsack, sum of subset.	Applications of branch and bound are – Job sequencing, TSP.
In this method, all possible solutions are tried. If solution is not satisfying the constraint then backtracks and tries another solution.	This method is based on search using bounding values, i.e., either upper bound or lower bound is computed, additionally.

COMPLEXITY THEORY

Ques 14) Give the introduction of complexity theory.
Or
What is complexity theory? What is the goal of complexity theory?

Ans: Complexity Theory
Solving problem using various models of computation, existed much before the computer was born. The developments in the field of computer science can be broadly categorised, into the pre and the post computer era.

In the pre computer era, the mathematicians worked very actively to formalise and the study the notion of algorithms. During those periods, much of the emphasis was on the computation of a function. The primary focus in those era was on – what can be computed and what cannot be computed. This field of study was called 'Computability Theory'. The example of a problem that cannot be computed is 'halting problem'. Though the computability theory has significant importance in computer science, solving a problem theoretically does not imply the solvability of the same practically too.

Suppose we develop a chess playing program considering all feasible moves of the chess pieces against the opponent's all possible moves, then it may take several years to run. So, there are many problems which can be solved, but require huge amount of space and time in practical.

Hence, it is clear that the time and space requirements of a program are important. The theory that deals with this is called, the 'Complexity Theory'. The primary focus of complexity theory is to find out how well a problem can be computed?

Complexity theory is the study of the resources (especially computation time and memory) required by algorithms.

The **goal of complexity theory** is to determine the amount of computational resources needed to perform certain computational tasks. The efforts to achieve these goals have only been partly successful.

Ques 15) What are tractable and intractable problems?

Ans: Tractable and Intractable Problems

Most of the algorithms discussed so far or are known to us require polynomial time with time complexity Θ (n), Θ (n^2), Θ (nlgn), Θ (n^3), Θ (n^k), Θ (2^n). Thus at most the time taken by an algorithm which are solvable can be $\theta(n^k)$, which is very much polynomial for some constant k. These problems are called **tractable problems**.

Some of the problems like Turing machines halting problem cannot be solved at all even if we provide it to run on computer for years. Such problems which are insolvable or require super polynomial time is called **intractable**.

Complexity theory seeks to classify problems according to their computational complexity. The problems that can and cannot be solved in polynomial time are **tractable** and **intractable**, respectively. Complexity theory concentrates on decision problems, which are problems with yes-or-no answers.

Ques 16) What is polynomially bounded algorithm? Discuss the P and NP class problem.

Ans: Complexity Class P and NP

An algorithm is said to be **polynomially bounded** if its worst-case complexity is bounded by a polynomial function of the input size. A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it.

Class P

P is the class of all decision problems that are polynomially bounded. The implication is that a decision problem $X \in P$ can be solved in polynomial time on a deterministic computation model (such as a deterministic Turing machine).

Class NP

NP represents the class of decision problems which can be solved in polynomial time by a non-deterministic model of computation. That is, a decision problem $X \in NP$ can be solved in polynomial-time on a non-deterministic computation model (such as a non-deterministic Turing machine). A non-deterministic model can make the right guesses on every move and race towards the solution much faster than a deterministic model.

Ques 17) What is the relationship between P and NP Classes?

Ans: Relationship between P and NP Classes

A deterministic machine, at each point in time, executes an instruction. Depending on the outcome of executing the instruction, it then executes some next instruction, which is unique. A non-deterministic machine on the other hand, has a choice of next steps. It is free to choose any that it wishes.

Figure 6.9 shows four possibilities for relationships among complexity classes. In each diagram, one region enclosing indicates a proper-subset relation. (a) P = NP = co-NP. Most researchers regard this possibility as the most unlikely. (b) If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. (c) P = NP $\cap$ co-NP, but NP is not closed under complement. (d) NP $\neq$ co=NP and P $\neq$ NP $\cap$ co-NP. Most researchers regard this possibility as the most likely.

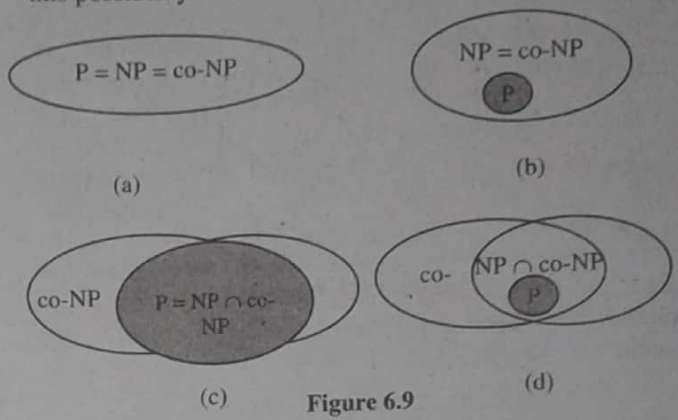


Figure 6.9

For example, it can always choose a next step that leads to the best solution for the problem. A non-deterministic machine thus has the power of extremely good, optimal guessing.

Ques 18) Discuss about the polynomial time reduction?

Ans: Polynomial Time Reduction

A polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem (that is, a reduction), that uses polynomial time excluding the time within the subroutine.

There are several different types of polynomial-time reduction, depending on the details of how the subroutine is used. Intuitively, a polynomial-time reduction proves that the first problem is no more difficult than the second one, because whenever an efficient algorithm exists for the second problem, one exists for the first problem as well.

Polynomial-time reductions are frequently used in complexity theory for defining both complexity classes and complete problems for those classes.

Types of Reduction

The three most common types of polynomial-time reduction, from the most to the least restrictive, are:

1) **Polynomial-Time Many-One Reductions:** A polynomial-time many-one reduction from a problem A to a problem B (both of which are usually required to be decision problems) is a polynomial-time algorithm for transforming inputs to problem A into inputs to problem B, such that the transformed problem has the same output as the original problem. An instance x of problem A can be solved by applying

this transformation to produce an instance y of problem B, giving y as the input to an algorithm for problem B, and returning its output.

Polynomial-time many-one reductions may also be known as polynomial transformations or Karp reductions, named after **Richard Karp**. A reduction of this type may be denoted by the expression:

$$A \leq_m^P B$$

2) **Polynomial-Time Truth-Table Reductions:** A polynomial-time truth-table reduction from a problem A to a problem B (both decision problems) is a polynomial time algorithm for transforming inputs to problem A into a fixed number of inputs to problem B, such that the output for the original problem can be expressed as a function of the outputs for B. The function that maps outputs for B into the output for A must be the same for all inputs, so that it can be expressed by a truth table. A reduction of this type may be denoted by the expression:

$$A \leq_{tt}^P B$$

3) **Polynomial-Time Turing Reductions:** A polynomial-time Turing reduction from a problem A to a problem B is an algorithm that solves problem A using a polynomial number of calls to a subroutine for problem B, and polynomial time outside of those subroutine calls. Polynomial-time Turing reductions are also known as **Cook reductions**, named after **Stephen Cook**. A reduction of this type may be denoted by the expression:

$$A \leq_T^P B.$$

The most frequently used of these are the many-one reductions, and in some cases the phrase "polynomial-time reduction" may be used to mean a polynomial-time many-one reduction.

Ques 19) What are the reasons that Polynomial-time algorithms can be considered tractable?

Ans: Polynomial-time algorithms can be considered tractable for the following **reasons**:

1) Although a problem which has a running time of say $O(n^{20})$ or $O(n^{100})$ can be called intractable, there are very few practical problems with such orders of polynomial complexity.
2) For reasonable models of computation, a problem that can be solved in polynomial time in one model can also be solved in polynomial time on another.
3) Class of polynomial-time solvable problems has nice closure properties (since polynomials are closed under addition, multiplication, etc.).

Ques 20) What is NP-Hard and NP-Complete Class?

Ans: NP-Hard and NP-Complete

A decision problem Y is said to be **NP-hard,** if $X \leq_p Y \forall X \in NP$.

A NP-hard problem Y is said to be **NP-complete**, if $Y \in NP$. NPC is the standard notation for the class of all NP-complete problems.

Informally, an NP-hard problem is a problem that is atleast as hard as any problem in NP. If, further, the problem also belongs to NP, it would become NP-complete.

It can be easily proved that if any NP-complete problem is in P, then NP = P. Similarly, if any problem in NP is not polynomial-time solvable, then no NP-complete problem will be polynomial-time solvable. Thus, NP completeness is at the crux of deciding whether or not NP = P.

Using the definition of NP-completeness to show that a given decision problem, say Y, is NP-complete will call for proving polynomial reducibility of each problem in NP to the problem Y.

This is impractical since the class NP already has a large number of member problems and will continuously grow as researchers discover new members of NP.

A much more practical way of proving NP-completeness of a decision problem Y is to discover a problem $X \in NPC$ such that $Y \leq_p X$.

Since X is NP-complete and $\leq_p$ is a transitive relationship, the above would mean that $Z \leq_p Y \forall Z \in NP$. Furthermore if $Y \in NP$, then Y is **NP-complete**.

The above is the standard technique used for showing the NP-hardness or NP-completeness of a given decision problem. **For example,** all the decision problems can be shown to be NP-complete by the above technique.

The first member of NPC found by **Stephen Cook** who showed the NP-completeness of the problem 3-SAT by directly proving that $X \leq_p 3\text{-SAT} \forall X \in NP$.

If a problem is NP-complete, it does not mean that all hopes are lost. If the actual input sizes are small, an algorithm with, say, exponential running time may be acceptable.

On the other hand, it may still be possible to obtain near-optimal solutions in polynomial-time. Such an algorithm that returns near-optimal solutions (in polynomial time) is called an approximation algorithm. Using **Kruskal's algorithm** to obtain a suboptimal solution to the TSP is an example of this.

Ques 21) What is Hamiltonian cycle? Also discuss about the Clique problem.

Ans: Hamiltonian Cycle

A Hamiltonian cycle of an undirected graph G= (V, E) is a simple cycle that contains each vertex in V. A graph that contains a Hamiltonian cycle is said to be **Hamiltonian**, otherwise it is said to be **non-Hamiltonian.**

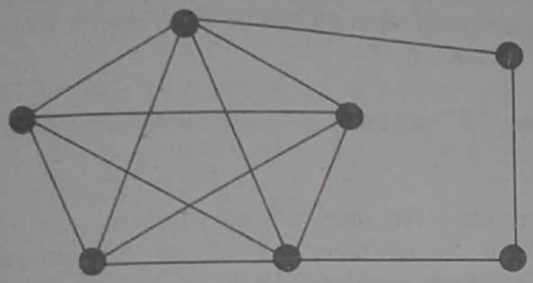
Clique Problem

A 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. As a decision problem, we ask simply whether a clique of a given size k exists in the graph. The formal definition is
CLIQUE = {⟨G, k⟩ : G is a graph with a clique of size k}.

Instance: A graph G = (V, E) and integer j ≤ v.



Does the graph contain a clique of j vertices, i.e., is there a subset of v of size j such that every pair of vertices in the subset defines an edge of G?

For example, this graph contains a clique of size 5.

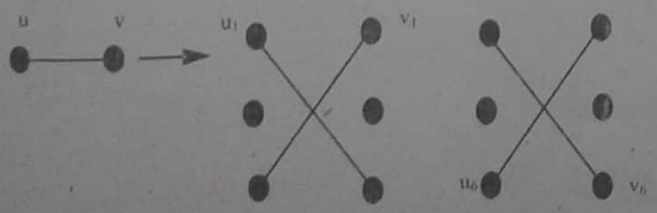
When talking about graph problems, it is most natural to work from a graph problem - the only NP-complete one we have is vertex cover!

Ques 22) Prove the theorem that 'Hamiltonian Circuit (HC) is NP-complete'.

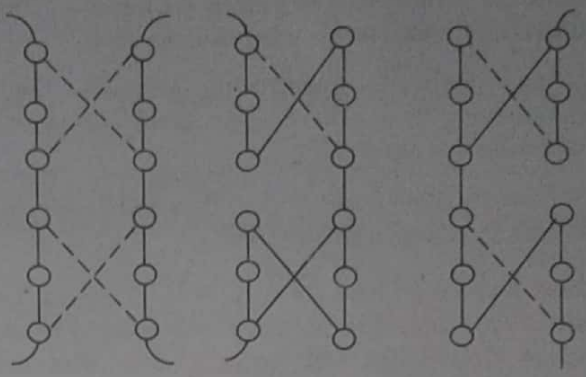
Ans: Proof

Clearly Hamiltonian Circuit is in NP. To show it is complete, use vertex cover. A vertex cover instance consists of a graph and a constant k, the minimum size of an acceptable cover.

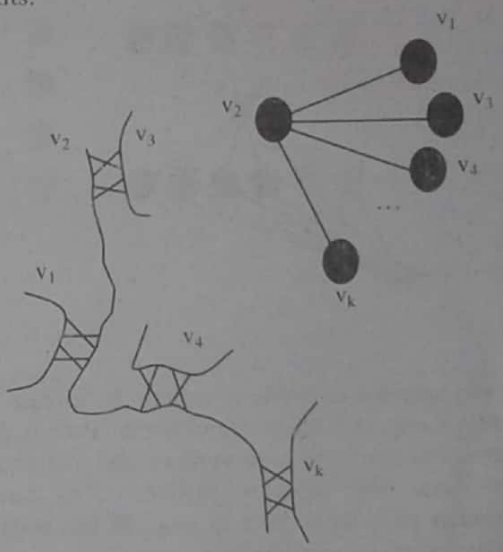
We must construct another graph. Each edge in the initial graph will be represented by the following component:



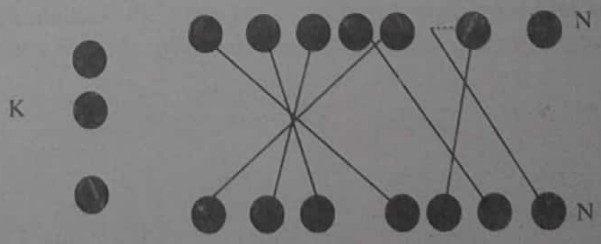
All further connections to this gadget will be through vertices v_1, v_6, u_1 and u_6. The key observation about this gadget is that there are only three ways to traverse all the vertices:



In each case, exit out the same side entered. Each side of each edge gadget is associated with a vertex. Assuming some arbitrary order to the edges incident on a particular vertex, one can link successive gadgets by edges forming a chain of gadgets. Doing this for all vertices in the original graph creates n intertwined chains with n entry points and n exits.



Thus one has encoded the information about the initial graph. One set up k additional vertices and connects each of these to the n start points and n end points of each chain.



Total size of new graph: GE +k vertices and 12E+2kN +2E edges → construction is polynomial in size and time. This graph has a HC iff G has a VC of size k.

1) Suppose $\{v_1, v_2, \ldots\ldots, v_n\}$ is a HC.

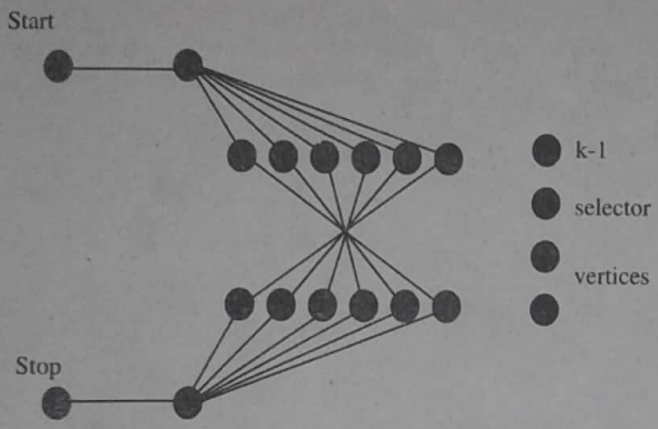
   Assume it starts at one of the k selector vertices. It must then go through one of the chains of gadgets until it reaches a different selector vertex.

Since the tour is a HC, all gadgets are traversed. The k chains correspond to the vertices in the cover.

If both vertices associated with an edge are in the cover, the gadget will be traversal in two pieces - otherwise one chain suffices.

To avoid visiting a vertex more than once, each chain is associated with a selector vertex.

2) Now suppose we have a vertex cover of size ≤ k. One can always add more vertices to the cover to bring it up to size k. For each vertex in the cover, start traversing the chain. At each entry point to a gadget, check if the other vertex is in the cover and traverse he gadget accordingly.

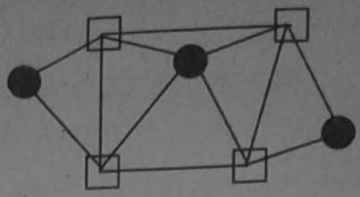


Select the selector edges to complete the circuit. To show that Longest Path or Hamiltonian Path is NP-complete, add start and stop vertices and distinguish the first and last selector vertices. This has a Hamiltonian path from start to stop iff the original graph has a vertex cover of size k.

Ques 23) Prove the theorem that 'Clique is NP-complete'.

Ans: Proof

If you take a graph and find its vertex cover, the remaining vertices form an independent set, meaning there are no edges between any two vertices in the independent set, for if there were such an edge the rest of the vertices could not be a vertex cover.



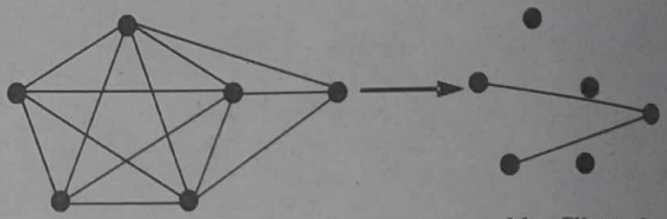
Vertex in cover

Vertex in independent set

Clearly the smallest vertex cover gives the biggest independent set, and so the problems are equivalent – Delete the subset of vertices in one from the total set of vertices to get the order!

Thus finding the maximum independent set must be NP-complete.

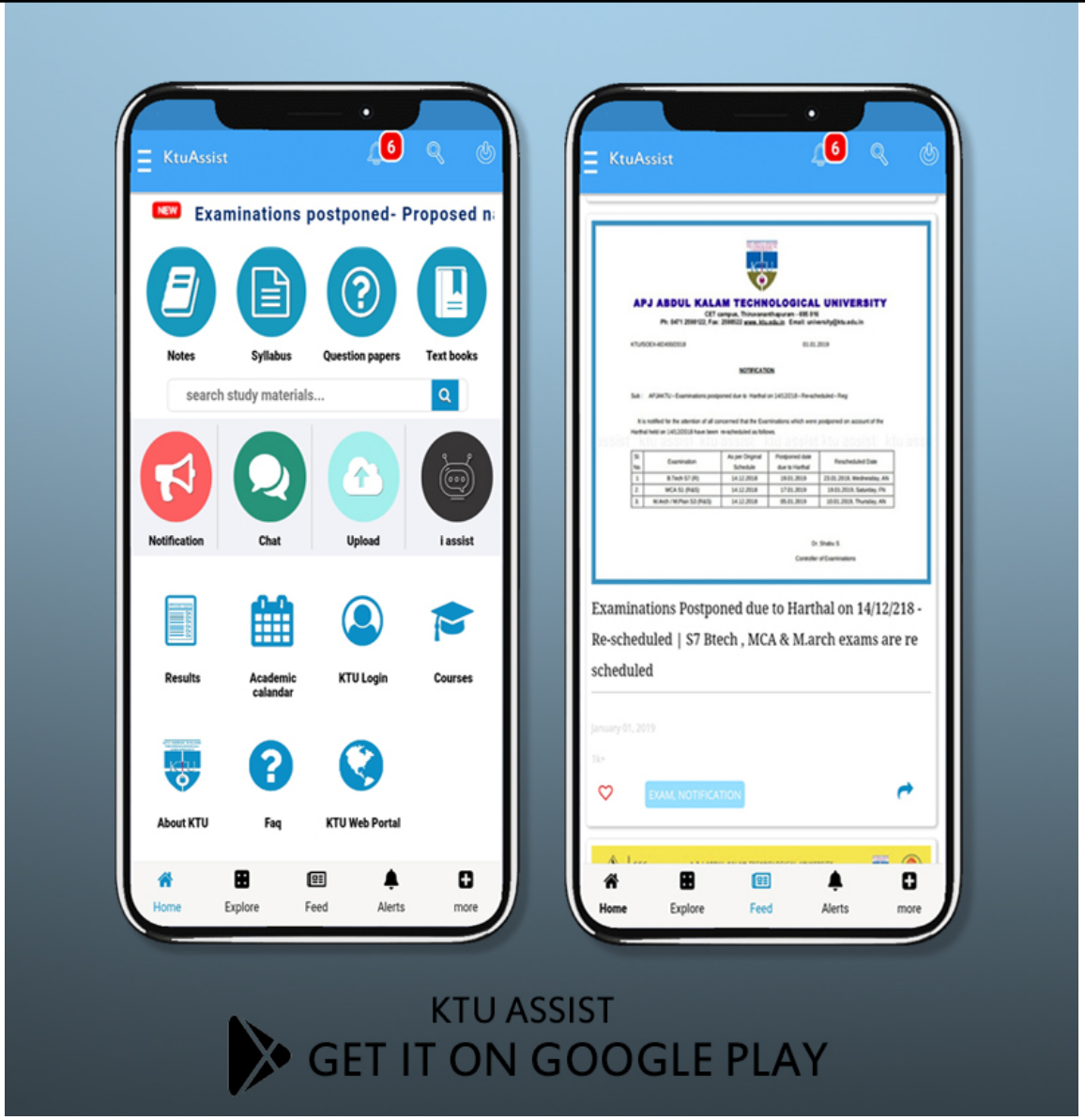
In an independent set, there are no edges between two vertices. In a clique, there are always between two vertices. Thus if we complement a graph (have an edge in there was no edge in the original graph), a clique becomes an independent set and an independent set becomes a clique.



Max Clique =5
Max IS =2

Max Clique =2
Max IS =5

Thus finding the largest clique is NP-complete: If VC is a vertex cover in G, then V – VC is a clique in G'. If C is a clique in G, V – C is a vertex cover in G'.



END

facebook.com/ktuassist

instagram.com/ktu_assist