UNIT - VI

COMPUTATIONAL COMPLEXITY AND PARALLEL ALGORITHMS

6.1 ALGORITHMS AND THEIR CLASSES

We have solved many problems using different algorithms. For each algorithm we also computed its run time. The algorithms are classified into two groups depending on their computing time :

1. **P Class :** This group consists of all the algorithms whose computing times are polynomial time, that is, there computing time is bounded by polynomials of small degree. For example, insertion sort requires $O(n^2)$ time. Merge sort and quick sort require O(n log n) time. Binary search tree requires O(log n) time for searching. Polynomial evaluation requires O(n) time. All these algorithms have polynomial computing time.
2. **NP Class :** This group consists of all the algorithms whose computing times are non-deterministic polynomial time. For example, computing time of the travelling salesperson problem is $O(n^2 2^n)$ using dynamic programming. Computing time of the knapsack problem takes $O(2^{n/2})$ time. These are called non-polynomial algorithms.

Generally, problems which can be solved using polynomial time algorithms are called as **tractable (easy)**. Problems that can be solved using super polynomial time algorithms are called **intractable (hard)**. A problem which requires exponential time are termed as intractable. But in reality, the exponential function works better for smaller values than the polynomial functions.

The NP class problems again can be classified into two groups :

1. NP-Complete (Non-deterministic Polynomial time complete) problems.
2. NP- hard problems

No NP-complete or NP-hard problem is polynomially solvable. All NP-Complete problems are NP-hard but some NP-hard problems are not NP-Complete.

The relationship between P and NP is shown in Fig. 6.1.

Since deterministic algorithms are special cases of non-deterministic algorithms. $P \subseteq NP$. Also all the NP-complete problems are NP-hard problems, but some NP-hard problems are not NP-complete. For example, the halting problem is an NP-hard problem, but not NP-complete. Given a deterministic algorithm A and input I, the halting problem is to determine that if algorithm A is run with input I, whether it will stop or enter in an infinite loop.

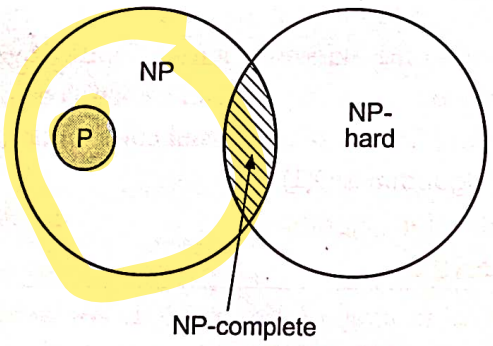


Fig. 6.1 : Relation between P and NP

6.2 DEFINITIONS

Decision Problem : Any problem having the answer either zero or one is called a decision problem. Only a decision problem can be NP-complete.

Decision Algorithm : It is an algorithm which solves a decision problem. It gives output 1 on successful completion. On unsuccessful termination, it gives output 0.

- **Optimization Problem :** Any problem that involves the identification of an optimal value for a given cost function is known as an optimization problem. The optimal value can be either maximum or minimum. An optimization problem may be NP-hard.
- **Optimization Algorithm :** It is an algorithm which solves an optimization problem.
- **Polynomial Complexity :** Generally the complexity of an algorithm is measured in terms of input size. If there exists a polynomial p(n) such that the computing time of A is O(p(n)) for every input of length n, then the corresponding algorithm is said to have polynomial complexity.
- **Satisfiability Problem :** The satisfiability problem is to find out whether a given expression is true for some assignment of truth values to the variable in that expression.
- **Clique :** It is a maximal complete subgraph for a given graph. The size of the clique is the number of vertices in it.
- **Reducibility :** Let L_1 and L_2 are two problems. Problem L_2 can be solved in polynomial time using a deterministic algorithm. If the same algorithm can solve problem L_1 in deterministic polynomial time, then it is termed as L_1 reduces to L_2 ($L_1 \propto L_2$).
- **NP-Hard Problem :** A problem L is said to be NP-hard iff satisfiability $\propto$ L (satisfiability reduces to L)
- **NP-Complete Problem :** A problem L is said to be NP-complete iff L is NP-hard and L $\in$ NP.

6.3 NON-DETERMINISTIC POLYNOMIAL TIME (NP) DECISION PROBLEMS

- To specify non-deterministic algorithms, we can use the following functions :
  - ➢ **Choice (Set S) :** It returns one of the values in set S arbitrarily.
  - ➢ **Success ( ) :** It denotes a successful completion of an algorithm.
  - ➢ **Failure ( ) :** It denotes an unsuccessful termination of an algorithm.
- All the above functions have O(1) computing time.
- The Choice function can return any value in the set S in any order. Hence depending on the order in which values fro input are selected affects the successful termination of a non-deterministic algorithm. If there exists at least one set choices which result in successful completion and if it is made, then the algorithm completes successfully. If there is set of choices which can result unsuccessful completion of an algorithm, then the algorithm always terminat unsuccessfully. A non-deterministic machine can execute a non-deterministic algorithm. But practically such nc deterministic machines do not exist.
- Let us study some non-deterministic decision algorithms. A decision problem gives answer either 0 or 1. A nc deterministic decision algorithm does a successful completion iff it gives output 1. On unsuccessful termination, gives output 0. Remember that this non-deterministic machine is not present practically. Before we proceed, let define the complexity of a non-deterministic algorithm having input of size n as the minimum number of ste required for successful completion if there exists a sequence of choices leading to such completion. If the input is size $n \geq n_0$, then the complexity is O(f(n)) in case of successful completion (n_0 is a constant). If no such sequence choices exists, then the complexity of algorithm is O(1).
- Let us study some non-deterministic decision algorithms :

6.3.1 Non-Deterministic Search Algorithm

- Suppose unordered input list L[1 : n] is an array of size n, n ≥ 1. We want to search an element x in array L [ ]. element x is found, its position should be output, otherwise output is 0. The algorithm is as given below :

Algorithm: NSearch (L, n, x)

```
begin
        p = Choice (1, n) // Returns index between 1 to n.
    // if x is found at position p, print p, else print 0.
    if (L[p]=x)
    then
            print p;
```

```
        Success ( );
    end if
    print 0;
    Failure ( );
end
```

- The computing time of this algorithm is O(1) in both the cases of successful search and unsuccessful search. Any deterministic search algorithm requires O(n) time.

6.3.2 Non-Deterministic 0/1 Knapsack Algorithm

- In the 0/1 knapsack decision problem, array P[1 : n] stores profits and array w[1 : n] stores weights of n input objects. M is the maximum capacity knapsack and R is the minimum profit required. P_i's, w_i's, M and R all the are non-negative numbers. We have to find array x[i] = 0/1 such that $\sum w_i x_i \leq M$ and $\sum p_i x_i \geq R$, for $1 \leq i \leq n$.
- The algorithm computes set of choices in array x[1 : n].

Algorithm: NKnapsack (n, p, w, M, R, x)

```
begin
    // Initialize total profit TP and total weight TW to zero.
    TP = 0
    TW = 0
    for i = 1 to n
    begin
        x[i] = Choice (0, 1);
        TW = TW + w[i] * x [i]
        TP = TP + p[i] * x[i]
    end for
    if ((p<R) OR (w >M))
    then
        Failure ( );
    else
        print x [1 : n]
        Success ( );
    end if
end
```

- If any set of choices can lead to successful completion, then only array x[ ] is printed. The computing time of this algorithm is O(n).

6.3.3 Non-Deterministic Clique Problem

- A clique is a maximal complete sub-graph of a given graph G(V,E). The size of the clique is the number of vertices in it. The clique decision problem is to find out whether graph G contains a clique of size at least x for $x \leq n$.
- The algorithm takes as input: Graph G, total number of vertices n and value x. It first computes a set S having x distinct vertices. Then it checks whether the set S forms a complete subgraph of G. If not, then the algorithm terminates unsuccessfully, otherwise it completes successfully.

Algorithm: NClique (G, n, x)

```
Initialize set S to empty set
•
Input set S
```

```
    for i = 1 to x
    begin
        y = Choice (1, n)
        if (y ∈ S) then Failure
        //otherwise add y to set S
        S = S U {y}
    end for
    //check whether S forms clique
    for each pair (i, j) of vertices in S such that i∈S, j∈S, i ≠ j do
    if edge <i, j> ∈E,
        then Failure( );
    end for
    Success ( );
end
```

- The time complexity of this algorithm = $O(n + x^2) = O(n^2)$.

6.3.4 Non-deterministic Sorting Algorithm

- Let P[1 : n] is an input array to be sorted in ascending order. The sequence in which Choice (1, n) returns indic used to create output array Q[ ] from P[ ]. Finally if Q[ ] is sorted, then algorithm completes successfully, otherwi terminates unsuccessfully,

Algorithm: NSort (p, n)

```
begin
    //Initialize array Q to 0
    for i = 1 to n
        Q[ i] = 0

    for i = 1 to n
    begin
    x = Choice (1, n)
        if Q[x] = 0
        then
        Q[x] = P[i]
        else
            Failure( );
        end if
    end for
// check whether Q is sorted
    for i = 1 to n - 1
    if (Q[i] > Q[i+1])
        Failure ( );
        end if
    end for
    Success ( );
end
```

- The computing time of this algorithm is O(n + n + n) = O(n). Where as the deterministic algorithms have minimu complexity O(n log n).

6.3.5 Non-Deterministic Satisfiability Problem

- Let $x_1, x_2, ..., x_n$ represent boolean variables which can take value true or false. Negation of x_i is represented as $\bar{x}_i$. Using variables and/or their negations, we can construct an expression in propositional calculus using Boolean AND, OR operations. The satisfiablilty problem is to find out whether an expression is true for some assignment of truth values to the variables in that expression.
- Let $E(x_1, x_2, ..., x_n)$ is an expression. The following algorithm uses a set of choices for all the variables $x_1, ..., x_n$. The algorithm completes successfully, if an expression E evaluates to true, otherwise algorithm terminates unsuccessfully.

Algorithm: NSatisfiability (E, n)

```
begin
  //Let x1, x2, ..., xn are variables used in E.
  for i = 1 to n
      x[i] = Choice(true, false);
  end for
  //Evaluate expression E for these xi's
  if E (x1, --- , xn) is true
  then
      Success ( )
  else
      Failure ( )
  end if
end
```

- For loop computes set of choices for x_i's in O(1) time. The expression E can be evaluated in time O(f(n)). Then computing time of algorithm is O(n) + O(f(n)). This algorithm tells that satisfiability is in NP.

6.4 COOK'S THEOREM

- It states that "Satisfiabilty is in P iff P = NP".
- Algorithm in section 7.3.5 tells that satisfiabilty is in NP. To prove that satisfiabilty is in P, we have to obtain a formula Q(A, I) where Q is satisfiable iff non-deterministic decision algorithm A completes successfully for input I. Let size of I is n, computing time of A is p(n), hence length of A is $O(p^3(n) \log n) = O(p^4(n))$. The time required to construct Q is $O(p^4(n))$.
- Assumptions on our non-deterministic machine and form of algorithm A are as given below :
  - The machine is word oriented. Each basic operation using one word operand requires one unit of time. Hence computing of operation depends on the number of words in operands.
  - All variables in A are of type boolean or integer.
  - Algorithm A contains no read or write instructions.
  - Algorithm A contains no constants. Constants are replaced by new variables.
  - A simple expression contains at most one operator and all operands are simple variables (not array). It uses following syntax :
    - (i) <simple var> = <simple expr>
    - (ii) <simple var> = <array var>
    - (iii) <array var> = <simple var>
    - (iv) <simple var> = Choice (S)
    - (v) <simple var> = Choice<*l*, u>

    where *l* and u denote lower and upper bounds of a range *l* to u.

- A contains simple assignment statement. It can contain one of the following statements.
  (i) Success( ), Failure( )
  (ii) declaration and dimension statement
  (iii) goto k
  (iv) "if c then goto b". c is boolean variable and b is an instruction number.
- A contains 1 to l instructions serially.
- A requires time not more than p(n) to process any input of length n, where p(n) denotes a polynomial. Hence by assumption (1), A can operate on p(n) words of memory only. Let A uses memory words numbered 1, 2, ..., p(n).

Q uses boolean variables of the following semantics :

(i) B (i, j, t) :
i denotes memory word.
j denotes bit number in word i.
t denotes number of steps executed.
$1 \le i \le p(n)$ as A uses p(n) memory words.
$1 \le j \le w$, if w is length of each memory word.
$0 \le t \le p(n)$.
B(i, j, t) is true iff bit j of memory word i has value 1 after successful execution of t steps on input I.

(ii) S(j, t) :
j denotes instruction number.
t denotes time.
$1 \le j \le l$ as A contains l instructions.
$1 \le t \le p(n)$ as A requires computing time p(n).
Hence S(j, t) denotes the instruction to be executed at time t. S(j, t) is true iff A executes instruction j at time t.

Suppose Q is made up of six sub-formulae, as follows :

C : Initial status of p(n) words represents input I. All non-input variables are zero.

D : Instruction 1 is the first instruction to execute.

E : At the end of the i^{th} step, only one instruction will execute next. At a time, only one S(j, i) will be true

F : Let S(j, i) is true.
(i) If j is Success ( ) or Failure( ), then S(j, i+1) is true.
(ii) If j is an assignment statement, then S(j+1, i+1) is true.
(iii) If j is goto k, then S(k, i + 1) is true.
(iv) If j is "if c then goto b", and c is true, then S(b, i + 1) is true.
(v) If j is "if c then goto b" and c is false, then S(j+1, i + 1) is true.

G : (i) If the instruction executed at step t is not an assignment statement, then B(i, j, t)'s are unchanged.
(ii) If it is an assignment statement and the variable of L.H.S. is x, then only x may change depending on the R.H.S

H : Success ( ) instruction is executed at time p(n), hence the computation terminates successfully.

$Q = C \wedge D \wedge E \wedge F \wedge G \wedge H$ is satisfiable iff A successfully completes using input I.

Formulae for C to H are as follows :

1. Formula C describes input I.

$$C = \bigwedge_{\substack{1 \le i \le p(n) \\ 1 \le j \le w}} T(i, j, 0)$$

If there is no input then

$$C = \bigwedge_{\substack{1 \le i \le p(n) \\ 1 \le j \le w}} \bar{B}(i, j, 0)$$

$$D = S(1, 1) \wedge \bar{S}(2, 1) \wedge \bar{S}(3, 1) \wedge \ldots \wedge \bar{S}(l, 1)$$

D is true iff instruction 1 is executed first i.e. S(1, 1) is true and S(2, 1), S(3, 1), ..., S(l , 1) are false. Then D is also satisfiable.

$$E = \bigwedge_{1 < t \le p(n)} E_t.$$

E_t will ensure that there is an unique instruction for step t.

$$E_t = S(1, t) \vee (S(2, t) \vee \ldots \vee S(l, t) \wedge \bigwedge_{\substack{1 \le j \le l \\ j \ne k}} (\bar{S}(j, t) \vee \bar{S}(k, t))$$

E_t is true iff exactly one S(j, t) is true.

$$F = \bigwedge_{\substack{1 \le i \le l \\ 1 \le t < p(n)}} F_{i,t}$$

Each $F_{i,t}$ ensures that either instruction i is not the one to be executed at time t, or if it is, then instruction i correctly determines instruction to be executed at time t + 1.

$$G = \bigwedge_{\substack{1 \le i \le l \\ 1 \le t < p(n)}} G_{i,t}$$

Each $G_{i,t}$ ensure that at time t either instruction i is executed or not, and the status of p(n) words after step t is correct w.r.t. the status before t and the changes resulting from instruction i.

6. Let the successful statements in A have numbers $i_1, i_2, \ldots, i_k$.

$$H = S(i_1, p(n)) \vee S(i_2, p(n)) \vee \ldots \vee S(i_k, p(n))$$

It can be verified that Q = C ∧ D ∧ E ∧ F ∧ G ∧ H is satisfiable iff the algorithm A completes successfully using input I. Observe that all C, D, E, F, G, H are in CNF, and if not, can be transformed into CNF. Formula C, D, E, F, G, H contain wp(n), $O(l^2p(n))$, $O(lp(n))$ $O(lwp^3(n))$, at most l literals respectively. Hence total number of literals in Q = $O(lwp^3(n)) = O(p^3(n))$ as w is constant. Each literal in Q can be written using $O(\log (wp^2(n) + lp(n))) = O(\log n)$ bits, because there are $O(wp^2(n) + lp(n))$ distinct literals. As p(n) is at least equal to n, the total length of Q = $O(p^3(n) \log n) = O(p^4(n))$. The time required to construct Q is $O(p^3(n) \log n)$ using algorithm A with input I.

This shows that every problem in NP reduces to satisfibility and also to CNF- satisfiability. Hence, if either of these two problems is in P, then NP ⊆ P and hence P = NP.

6.5 NP-COMPLETE PROBLEMS

To be NP-Complete, a decision problem must belong to NP and it must be possible to polynomially reduce any other problem in NP to that problem.

Definition :

A decision problem X is NP-Complete if

1. X ∈ NP and
2. Y ≤ pX for every problem Y ∈ NP.

If we have already shown that some problems are NP-Complete, and we want to show other problems to be NP-Complete, then we can use the following theorem.

Theorem 6.5.1 : Let X be an NP-Complete problem. Consider a decision problem Z ∈ NP such that $X \le_p Z$, then Z is also NP-Complete.

≤ denotes "less than or equal to". $X \le_p Z$ denotes that X is less harder than Z by a polynomial factor p.

6.5.1 Satisfiability Problem

Let $x_1, x_2, \ldots, x_n$ be boolean variables. Negation of x_i is $\bar{x}_i$. A literal is a variable or its negation. An expression consists of one or more literals. A boolean expression is said to be in conjunctive normal form (CNF) if it is the product of sums of literals. In prepositional calculus, ∧ is termed as conjunction and ∨ is termed as disjunction. For example, $(x_1 + \bar{x}_2)$

$(x_2 + \bar{x}_1)$ is in CNF, which is represented as $(x_1 + \bar{x}_2) \wedge (x_2 + \bar{x}_1)$ in prepositional calculus. A Boolean expression is in k-CNF for some positive integer k if it is composed of clauses, each of which contains at most k literals. For CNF, a clause is either a literal or a disjunction of literal. For example,

$(p + \bar{q})\ (p + q + \bar{r})\ \bar{p}\ r$ is in 3-CNF because second clause contains maximum 3 literals.

$(p + qr)\ (\bar{p} + \bar{q}\ (p + \bar{r}))$ is not in CNF.

Definition 6.5.2 : A boolean expression is satisfiable if it is true for some assignment of truth values to its variables. The problem of deciding whether a given boolean expression is satisfiable or not, is denoted as SAT.

For example, consider a boolean expression $(a \vee b) \Rightarrow (a \wedge b)$. If we assign a = b = true, then the given expression becomes true. Hence it is satisfiable. The boolean expression $(a \vee b) \wedge \bar{a} \wedge \bar{b}$ is not satisfiable because for all possible assignments of truth values to a and b, the given expression remains false.

Whenever we want to find out whether given boolean expression is satisfiable or not, we can try all the possible combinations of input variables. But it is not practical if number of input variables n is large enough, because for n input variables, there are 2^n possible truth values combinations. To solve this problem no efficient algorithm is available.

In the previous section, we have seen by Cook's theorem, that every problem in NP reduces to satisfiabiltiy and also to CNF-satisfiabilty. Also since satisfiability is in NP, and the construction of Q as CNF formula shows that satisfiabilty $\propto$ CNF-satisfiabiltiy. Also CNF-satisfiability is NP. This all together implies that CNF-satisfiability is NP-Complete.

Theorem 6.5.2 : CNF-sastisfiability (SAT-CNF) is NP-Complete.

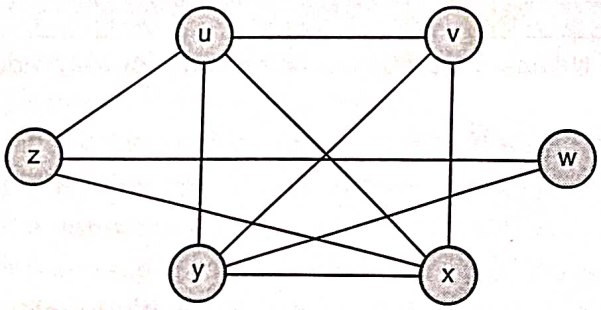
We can apply theorem 7.5.1 to prove the NP-completeness of other problems.

Theorem 6.5.3 : SAT is NP-complete,

Proof : We know that SAT is in NP. If we show that SAT-CNF $\leq_p$ SAT, then we can apply theorem 6.5.1. We know that boolean expression in CNF is a special case of general boolean expression. Given a boolean expression, we can tell whether it is in CNF or not. Hence any algorithm which can solve SAT efficiently can be used to solve SAT-CNF. Hence SAT is also NP-complete.

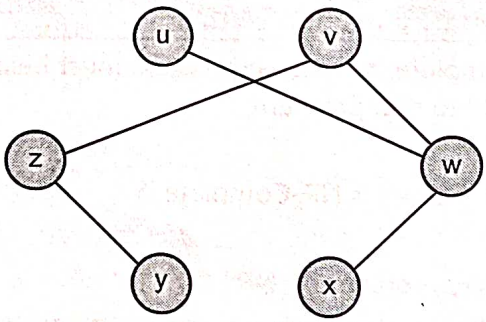
6.5.2 Vertex Cover Problem/Node Cover Decision Problem (NCDP)

Definition : Consider an undirected graph G (V, E). A vertex cover of G is a subset of vertices that covers all the edges in E. In other words, a vertex cover is a subset $V^1 \subseteq V$ such that if edge $< u, v > \in E$, then $u \in V^1$ or $v \in V^1$ (or both), i.e. each vertex covers all the edges in E. The size of a vertex cover is the number of vertices in it. For example, consider the following graph :



Graph G

Fig. 6.2 (a)



A vertex cover {w, z} of size 2

Fig. 6.2 (b)

The vertex cover problem is to find a vertex cover of minimum size in a given graph. As a decision problem, it checks whether a graph has a vertex cover of a given size k.

Theorem 6.5.4 : The vertex-cover problem is NP-complete.

Proof :

(Part I) : Let us show first that VERTEX-COVER $\in$ NP. Consider a graph G (V, E) and an integer k. Let there is a vertex cover $V^1 \subseteq V$. First check that $|V^1| = k$. Then for each edge $<x, y> \in E$, check that either $x \in V^1$ or $y \in V^1$. This checking can be done by an algorithm in polynomial time.

(Part II) : If we prove that CLIQUE $\leq_p$ VERTEX-COVER, then we can say that a vertex cover problem is NP-Complete by using theorem 6.5.1. Let us find the complement $\bar{G}(V, \bar{E})$ of a given graph G(V, E) where $\bar{E}$ consists of edge <x, y> if <x, y> $\notin$ E. Consider the following graph in Fig. 6.3 (a).

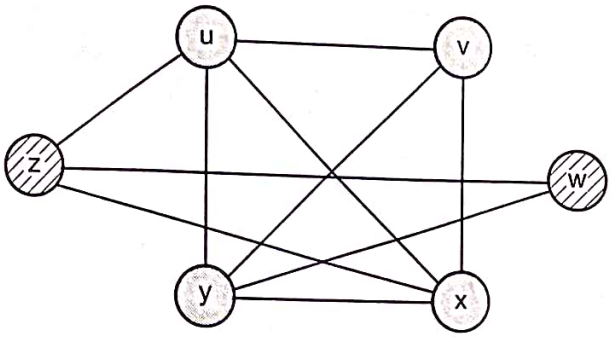


Fig. 6.3 (a) : Graph G with clique V^1 = {u, v, x, y}

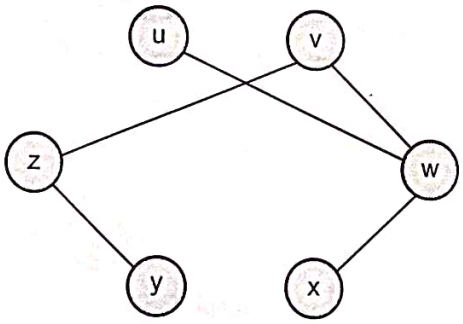


Fig. 6.3 (b) : $\bar{G}$ with vertex cover = $V - V^1$ = {w, z}

- When the graph in Fig. 6.3 (a) is reduced, we obtained its complement $\bar{G}$ in Fig. 6.3 (b) having vertex cover $V - V^1$ = {w, z}. If (G, k) given, then within a polynomial time, the reduction algorithm can find its complement $\bar{G}$ and produces as output ($\bar{G}$, |V| – k) a vertex cover.
- The graph G has a clique of size k iff the graph $\bar{G}$ has a vertex cover of size |V| – k. Assume that G has a clique $V^1 \subseteq V$ with $|V^1|$ = k. Let edge <x, y> $\in \bar{E}$, hence <x, y> $\notin$ E. It means that either $x \notin V^1$ or $y \notin V^1$, that is, either $x \in (V - V^1)$ or $y \in (V - V^1)$. It denotes that edge <x, y> is covered by $V - V^1$. Similarly we can show that each edge in $\bar{E}$ is covered by a vertex in $V - V^1$. Hence the set $V - V^1$ of size |V| – k forms a vertex cover for $\bar{G}$.
- Conversely, if we assume that $\bar{G}$ has a vertex cover $V^1 \subseteq V$, where $|V^1| = |V| - k$. Then for all x, y $\in$ V, if edge <x, y> $\in \bar{E}$, then either $x \in V^1$ or $y \in V^1$ or both. The contrapositive of this implication is that for all x, y $\in$ V, if $x \notin V^1$ and $y \notin V^1$, then <x, y> $\in$ E. That is, $V - V^1$ is a clique, and it has size $|V| - |V^1| = k$.

Conclusion : Form part II, CLIQUE $\leq_p$ VERTEX-COVER. Hence by theorem 6.5.1, the vertex cover problem is NP-Complete.

Example 6.1 : *Consider the following graph G in Fig. 6.4 (a).*

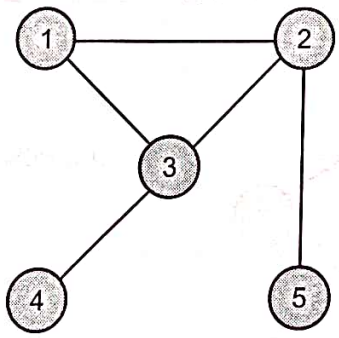


Fig. 6.4 (a) : Graph G

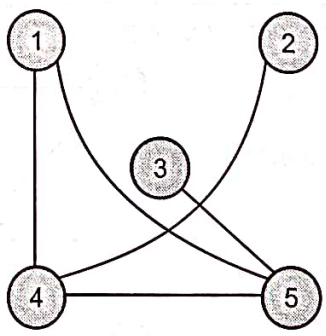


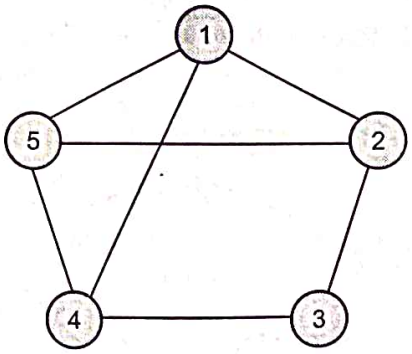
Fig. 6.4 (b) : Graph $\bar{G}$: Complement of G

Since every edge in $\bar{G}$ is incident either on vertex 4 or vertex 5, if has vertex cover = {4, 5}. Graph G has a clique of size 5 – 2 = 3.

Clique = {1, 2, 3}.

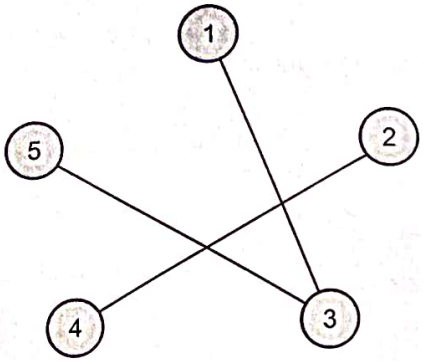
Example 6.2 : *The following graph G in Fig. 6.5 (a) has cliques {1, 2, 5} and {1, 4, 5} of size 3. $\bar{G}$ has corresponding vertex covers {3, 4}, {2, 3} of size 2. Also G has cliques {2, 3} and {3, 4} of size 2. $\bar{G}$ has the corresponding vertex covers {1, 4, 5} and {1, 2, 5} of size 3.*

Solution:



Graph G

Fig. 6.5 (a)



Complement $\bar{G}$

Fig. 6.5 (b)

6.6 NP-HARD PROBLEMS

- A problem X is NP-hard if there is an NP-hard problem Y that can be polynomially reduced to it, that is $Y \leq_p X$. T show that X is NP-hard, apply following steps :
  - ➢ Select some problem Y such that Y is NP-hard.
  - ➢ Using a polynomial time algorithm, p wrt form transformation of X instance into Y instance. It means that $Y \leq_p X$ [i other symbols $Y \propto X$].
  - ➢ Steps (i) and (ii) lead to the conclusion that X is NP-hard.
- There are many reasons to study NP-hardness instead of NP-Completeness. The notion of NP-hardness is importan for decision problems. Let us study some NP-hard as well as NP-Complete problems.

6.7 NP-HARD GRAPH PROBLEMS

- We have already seen two problems which are NP-hard as well as NP-complete. They are clique decision proble (CDP) and vertex cover problem i.e. Node Cover decision problem (NCDP). Let us study few more problems.

6.7.1 Chromatic Number Decision Problem (CNDP)

- A coloring of a graph G(V, E) is a function $f : V \rightarrow \{1, 2, ..., K\}$ defined for all $i \in V$. If edge $<u, v> \in E$, then $f(u) \neq f(v)$, tha is, if vertices u and v are adjacent, then they can not have same colour. The chromatic number decision problem is t find out whether a graph G has a coloring for a given k.

Example 6.3 : *Consider the following graph G :*

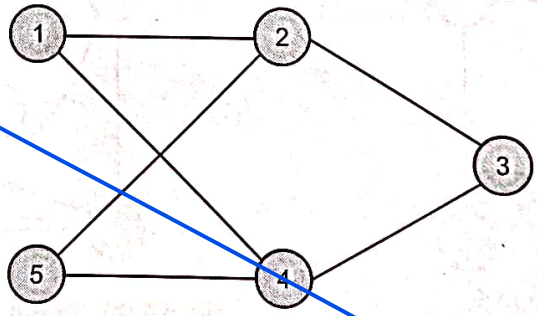


Fig. 6.6 : Graph G

- For the graph G in Fig. 6.6, 1-coloring is not possible. 2-coloring is possible with f(1) = f(3) = f(5) = 1 and f(2) = f(4) 2. That is, vertices 1, 3, 5 have color 1 and vertices 2, 4 have color 2.

Theorem : 3-CNF-Satisfiablility is polynomially transformable to the chromatic number decision problem (CNDP). Henc CNDP is NP complete.

Proof : Let F be a 3-CNF Boolean expression. So each clause of F has at most 3 literals.

Let $F = \wedge_{1 \leq i \leq r} c_i$. Let there are n variables $x_1, x_2, ..., x_n$ used in expression F. Assume $n \geq 4$. The graph G(V, E) is defined by

$$V = \{x_1, x_2, ..., x_n\} \cup \{\bar{x}_1, \bar{x}_2, ..., \bar{x}_n\} \cup \{y_1, y_2, ..., y_n\} \cup \{c_1, c_2, ..., c_r\}$$

where y_i's are new variables.

$$E = \{(x_i, \bar{x}_i),\ 1 \le i \le n\} \cup \{(y_i, \bar{y}_i) \mid i \ne j\} \cup \{(y_i, x_j) \mid i \ne j\} \cup \{y_i, \bar{x}_j) \mid i \ne j\}$$
$$\cup \{(x_i, c_j) \mid x_i \notin c_j\} \cup \{(\bar{x}_i, c_j) \mid \bar{x}_i \notin c_j\}$$

We can construct such (n + 1)-colorable graph from G iff F is satisfiable. Observe that y_i's form a complete subgraph of n vertices. Hence each vertex y_i is adjacent to all the remaining vertices in that subgraph, and therefore each vertex y_i takes different colour i. But y_i is connected to x_j and $\bar{x}_j$ except x_i and $\bar{x}_i$, hence x_i and $\bar{x}_i$ can have colour i. But $\langle x_i, \bar{x}_i \rangle \in E$, hence i can be colour of one vertex to which we will call as true vertex.

Now colour (n + 1) is required for the other vertex between x_i and $\bar{x}_i$, to which we will call as false vertex. There is only one way to colour G using n+1 colours where colour (n + 1) is assigned to one of the vertices x_i and $\bar{x}_i$, for $1 \le i \le n$.

We have assumed that $n \ge 4$. Also each clause has at most 3 literals. So each factor c_i is adjacent to a pair of vertices $x_j, \bar{x}_j$ for at least one j. Hence no c_i can be assigned colour n +1. No c_i can be assigned a colour of x_j or $\bar{x}_j$ not in clause c_i. Hence the only colour c_i can take is the colour of x_j and $\bar{x}_j$ that are in clause c_i and also are true vertices. Hence G is (n + 1) - colorable iff there is a true vertex corresponding to each c_i. Hence G is (n + 1) - colorable iff F is satisfiable. Hence CNDP is NP-complete.

7.2 Directed graph Hamiltonian Cycle (DHC)

A hamiltonian cycle in a directed graph G(V, E) is a directed cycle of length n = |V|. It visits each vertex only once. The DHC problem is find out whether a directed graph G has any hamiltonian cycle.

ample 6.4 : *Consider the following directed graph G :*

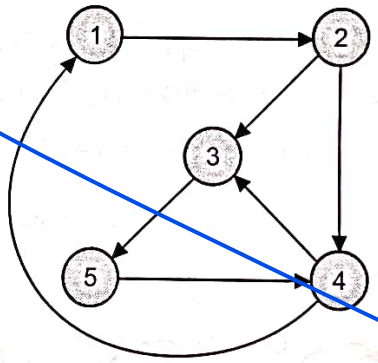


Fig. 6.7 : Graph G

graph G has hamiltonian cycle 1, 2, 3, 5, 4, 1. If any of these edges is deleted, then G will not have any hamiltonian le.

eorem 6.7.2 : CNF-SAT is polynomially **transformable** to DHC, hence DHC is NP-Complete

of :

t I : Let us first show that DHC ∈ NP. Given a graph G(V, E) and |V| = n, then all the n vertices belong to hamiltonian le exactly once. Only first vertex is repeated at the end to complete a cycle. It is necessary to check that each pair of nsecutive vertices in a cycle are connected by an edge. This checking can be done in polynomial time.

t II : We have to prove that CNF-SAT $\le_p$ DHC, which shows that DHC is NP-Complete.

F is a Boolean expression in CNF. We have to construct a directed graph G such that F is satisfiable iff G has a directed miltonian cycle.

Let

$$F = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \text{ where}$$

$$C_1 = x_1 \vee \bar{x}_2 \vee x_4 \vee \bar{x}_5$$

$$C_2 = \bar{x}_1 \vee x_2 \vee x_3$$

$$C_3 = \bar{x}_1 \vee \bar{x}_3 \vee x_5$$

$$C_4 = \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4 \vee \bar{x}_5$$

Let $F = \wedge_{1 \le i \le r} C_i$ and it uses n variables $x_1, x_2, \ldots, x_n$. Construct a graph using the following steps :

- Draw an array with r rows and 2n columns. Row i denotes clause C_i. Each variable x_i can take two forms : x_i and $\bar{x}_i$. Hence for n variables, there are 2n columns, one for each x_i and one for each $\bar{x}_i$.
- Insert ⊙ into column x_i and row C_j iff $x_i \in x_j$.
- Insert ⊙ into column $\bar{x}_i$ and row C_j iff $\bar{x}_i \in C_j$.
- Between each pair of columns x_i and $\bar{x}_i$, insert vertex u_i at the top of column and vertex v_i at the bottom of column.
- For each i, draw a chain of edges upward from v_i to u_i to connect together all ⊙s in column x_i.
- For each i, draw another chain of edges upward form v_i to u_i to connect together all ⊙s in column $\bar{x}_i$.
- Draw edges $< u_i, v_{i+1} >, 1 \le i < n$.
- Insert a box $\boxed{i}$ at the right end of each row C_i, $1 \le i \le r$.
- Draw the edges $< u_n, \boxed{1} >$ and $< \boxed{r}, V_1 >$.
- Draw edges $< \boxed{i}, \boxed{i+1} >, 1 \le i < r$.

For the above example, the resulting array structure is as shown in Fig. 6.8.

Using this array structure, we have to construct a graph by replacing each ⊙ and $\boxed{i}$ by a subgraph. Replace each ⊙ by sub-graph in Fig. 6.9 (a). Then subgraph of columns x_3 and $\bar{x}_3$ will be as shown in Fig. 6.9 (b).

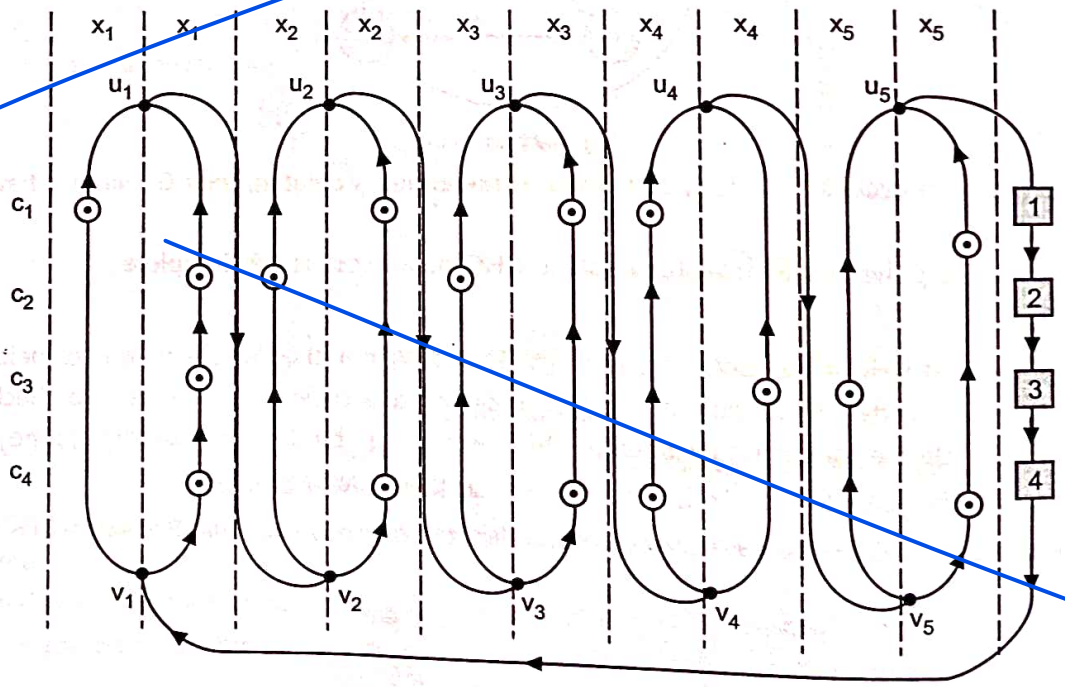


Fig. 6.8 : Array structure for given boolean expression F

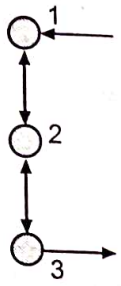


Fig. 6.9 (a)

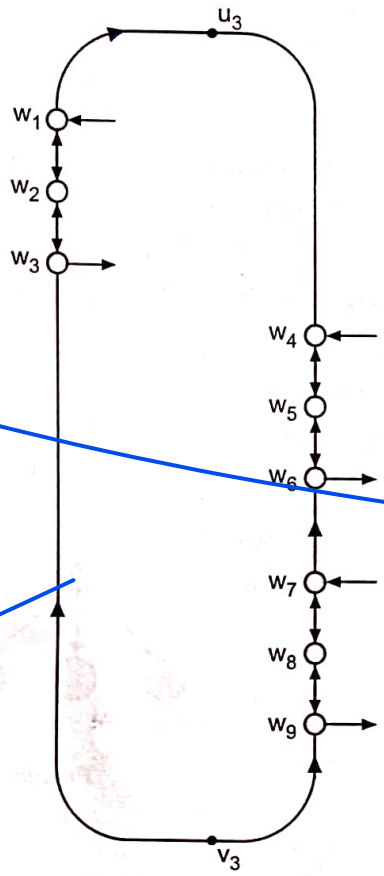


Fig. 6.9 (b)

- Each box [i] is also replaced by other subgraph, where A_i and B_i denote entrance vertex and exit vertex. In this subgraph, each edge

$R_i, a \longrightarrow (*) \longrightarrow R_j, a + 1$

indicates a connection to ⊙ subgraph in row C_i.

$R_{i,a}$ is connected to vertex 1 0f ⊙ and $R_{i,a+1}$ is entered from vertex 3 of ⊙. This completes the construction of graph F. This construction can be carried out in time polynomial in size of F.

We have to show that if G has a directed Hamiltonian cycle, then F is satisfiable. Start a hamiltonian cycle from V_1. It goes through either column x_i or column $\bar{x}_i$ for each pair $(x_i, \bar{x}_i)$ .Cycle also goes through at least one ⊙ subgraph in each row. Hence columns used in going from v_i to u_i, $1 \le i \le n$, define a truth assignment for which F is true. We conclude that F is satisfiable iff G has a hamiltonian cycle.

This can be done in polynomial time. Hence it follows that CNF-SAT $\le_p$ DHC.

7.3 Travelling Salesperson Problem (TSP)

In traveling salesperson problem, a salesman must visit n cities where cities are represented as graph vertices, and the links between cities are represented as edges in graph. A salesman wants to make a tour (hamiltonian cycle) such that each city is visited exactly once and he finishes tour at the city where he started. Each edge <i, j> is associated a cost c(i, j). Salesman is interested in a minimum cost tour.

A traveling salesperson decision problem is to determine whether a graph G has a tour of cost at most k.

Theorem 6.7.3 : DHC is polynomially transformable to TSP, hence TSP is NP-Complete.

Proof :

Part I : First we have to show that TSP∈ NP. Given a graph G having |V| = n, and a sequence of edges, then an algorithm can check whether each vertex is visited exactly once, add costs of all the edges in sequence and check whether the total cost is at most k. This can be done in polynomial time. Hence TSP ∈ NP.

Part II : To prove that TSP is NP-hard, we have to show that DHC $\le_p$ TSP. Let G(V, E) be an instance of DHC. Let us construct an instance of TSP as follows : Form the complete graph $G^1 (V, E^1)$ where $E^1 = \{ (i, j) \mid i, j \in V \text{ and } i \ne j\}$ and define a cost function c as

$$c(i, j) = \begin{cases} 0 & ; \text{if } (i, j) \in E \\ 1 & ; \text{if } (i, j) \notin E \end{cases}$$

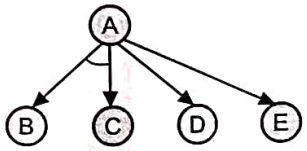
We that G is undirected graph. Hence it has no self-loops and c(u, u) = 1 for all u ∈ V. Then the instance of TSP is , k, 0) which is constructed in polynomial time.

- We have to show that graph G has a hamiltonial cycle iff graph G^1 has a tour of cost at most 0. Assume that there is a hamiltonian-cycle h in graph G. Each edge in h belongs to E and therefore has cost 0 in G^1. Thus h has cost 0 in G^1. Conversely, assume that graph G^1 has a tour h^1 of cost at most 0. Since the cost of edges in E^1 is 0 or 1, the cost of tour h^1 is exactly 0 and each edge on the tour h^1 must have cost 0. Therefore h^1 contains edges in E only. So we can conclude that h^1 is a hamiltonian cycle in graph G.
- From part I, TSP $\in$ NP and from part II, DHC $\leq_p$ TSP. Hence TSP is NP-hard.

6.7.4 AND/OR Graph Decision Problem (AOG)

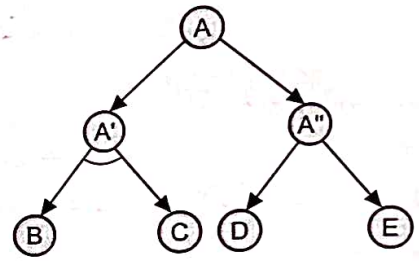
- When a task/problem is very complex, it is better to divide it into subtasks. Again a subtask can be divided into subtasks, until each smallest task is not enough simple to execute. This division of task into subtasks can be represented as a directed graph. Root node represents main task. Subtasks are represented as children.

Example 6.5 : *Consider a problem shown in the following graphs G.*



Problem A

Fig. 6.10 (a)



AOG for A

Fig. 6.10 (b)

- Graph in Fig. 6.10 (a) represents a problem A that can be solved by solving either both subproblems B and C, or the single subproblems D or E. Edges <A, B> and <A, C> are connected by an arc, which means that subproblems B and C must be solved. In Fig. 6.10 (b), dummy nodes A' and A'' are inserted. Node A' requires all its descendents to be solved. It is called **AND Node**. Node A'' requires only one of its descendents to be solved. It is called **OR Node**. The AND node have an arc among all the edges leaving the node. **Terminal Nodes** represent primitive problems. They are marked either solvable or unsolvable.
- Dividing a problem into several subproblems is known as **problem reduction**. Problem reduction is used in many applications like analysis of industrial schedules, theorem proving, etc. A **solution graph** is a subgraph of solvable nodes that shows the problem is solved.

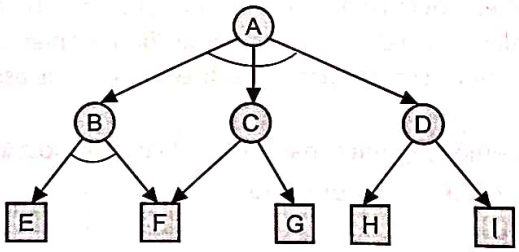


Fig. 6.11 (a) : Problem Graph G_1

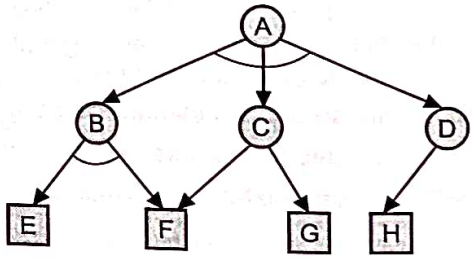


Fig. 6.11 (b) : Solution Graph of G_1

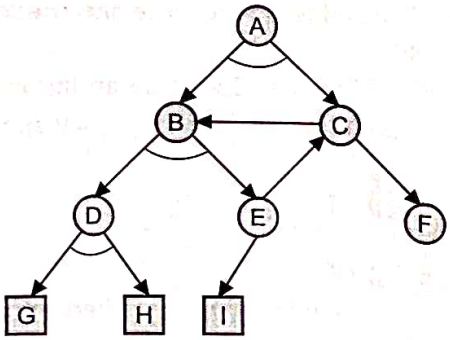


Fig. 6.11 (c) : Problem Graph G_2

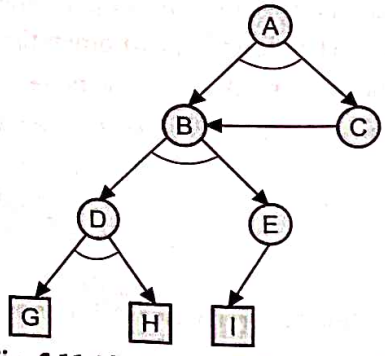


Fig. 6.11 (d) : Solution Graph for G_2

- In AOG, a cost is associated with each edge. The cost of a solution graph H of an AND/OR graph G is the sum of the costs of the edges in H. The AND/OR graph decision problem (AOG) is to find out whether G has a solution graph of cost at most k, for given input.

Example 6.6 : *Consider the following AND / OR graph in Fig. 7.12.*

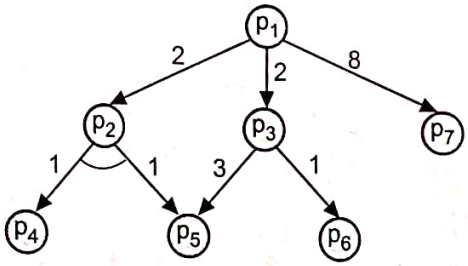


Fig. 6.12

Here p_2 is the only AND node, that is, p_4 and p_5 must be solved. p_3, p_7, p_1 are OR-nodes. Cost to solve p_1 is 2, 2 or 8. depending on whether p_2, p_3 or p_7 is solved. If p_2 is solved, total cost is 2 + 1 + 1 = 4. If p_3 is solved, total cost for p_1 is 2 + 1 = 3. If p_7 is solved total cost is 8. Hence optimal solution of p_1 is to solve p_6, then p_3, then p_1 having total cost 3.

Theorem 6.7.4 : CNF-SAT is polynomially transformable to AOG. Hence AOG is NP-hard.

Proof :

Part I : First we have to show that AOG $\in$ NP. Given a solution graph of cost at most k for a graph G, an algorithm can check whether a solution graph is a subgraph of G, and check whether addition of cost of all the edges in solution graph is at most k. This process requires polynomial time only. Hence AOG $\in$ NP.

Part II : We have to show that CNF-SAT $\leq_p$ AOG. Let P is a boolean expression in CNF. $P = \wedge_{1 \leq i \leq k} c_i$, and $c_i = V\ l_j$ Expression P has k clauses. Each clause is a disjunction of literals *lj*'s. Let V(P) denotes variables used in expression P and V(P) = $\{x_1, x_2, ..., x_n\}$. We have to transform CNF expression P into an AND/OR graph such that AND/OR graph obtained has a certain minimum cost solution iff P is satisfiable.

The AND/OR graph consist of following nodes :

- Root S represents problem to be solved.
- Root S is an AND-node with descendent nodes P, $x_1, x_2, ..., x_n$.
- Each node x_i denotes variable x_i in expression P. Each x_i is an OR-node with two descendents Tx_i and Fx_i. If Tx_i is solved, variable x_i takes true value. If Fx_i is solved, variable x_i takes value false.
- Node is an AND-node having k descendents $c_1, c_2, ..., c_k$. Each node c_i is an OR-node.
- Each node Tx_i or Fx_i has only one descendent, which is terminal. All terminal nodes are named $v_1, v_2, ..., v_{2n}$.

Now nodes can be constructed using above steps, but we have to connect them using the following steps :

- For each clause c_i, if it uses x_i, add an edge $< c_i, Tx_i >$.
- For each clause c_i, if it uses $\bar{x}_i$, add an edge $< c_i, Fx_i >$.
- Repeat steps (i) or (ii) for each variable in each clause.
- Each node c_i is an OR-node.
- Edges $<Tx_i, vj\geq$ or $<Fx_i, vj>$ have cost 1.
- All the remaining edges have cost 0.

Now graph for problem S is ready. To solve S, each of its descendents P, $x_1, x_2, ..., x_n$ must be solved. For each x_i, either subproblem Tx_i or Fx_i is solved, hence cost of each x_i is 1. Total cost of solving $x_1, x_2, ..., x_n$ variables is n. As P is an AND-node, all nodes $c_1, c_2, ..., c_k$ must be solved. Cost of solving each c_i is at most 1 as c_i is an OR-node. Hence node S can be solved at cost n if at least one literal x_i or $\bar{x}_i$ in each clause has true value, that is, each clause results in true value, that is, boolean expression P results in true value, that is, P is satisfiable. If P is not satisfiable, then cost is more than n. Part I tells that we can construct such AND/OR graph having solution of cost n in polynomial time. Part II tells that CNF-SAT $\leq_p$ AOG. Hence AOG is NP-hard.

Example 6.7 : *Construct AND/OR graph for the following CNF expression :*

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2)$$

$$V(P) = x_1, x_2, x_3 \text{ and } n = 3.$$

Solution :

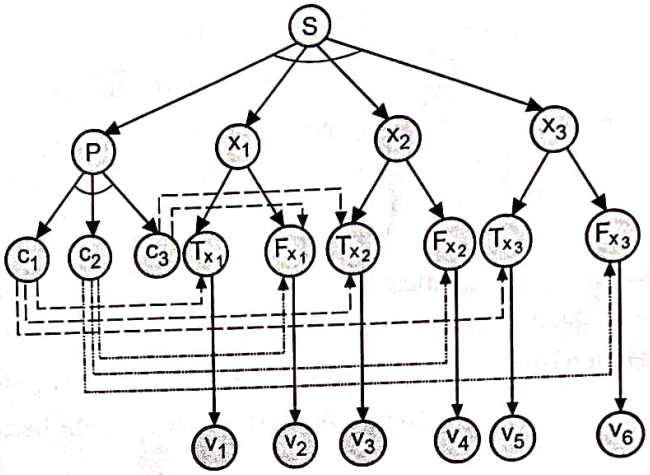


Fig. 6.13

S, P represent AND nodes.

Remaining nodes represent OR nodes.

6.8 NP-HARD SCHEDULING PROBLEMS

- The partition problem is to decide whether a given multiset $A = \{a_1, a_2, ..., a_n\}$ of n positive integers has a partition P such that $\Sigma_{i \in p} a_i = \Sigma_{i \notin p} a_i$.
- The sum of subsets problem is to find out whether a given set $A = (a_1, a_2, ..., a_n)$ has s subset S that sums to a given integer M. We know that the sum of subsets problem is NP-hard. If we show that sum of subsets $\leq_p$ partition, then partition problem is also NP-hard.

Theorem 6.8.1 : Sum of subsets $\leq_p$ Partition

Proof : Let set $A = \{a_1, a_2, ..., a_n\}$ and M define an instance of the sum of subsets problem. Construct the set $B = \{b_1, b_2, ..., b_{n+2}\}$

where,

$$b_i = a_i \text{ for } 1 \leq i \leq n,$$

$$b_{n+1} = M + 1,$$

$$b_{n+2} = (\Sigma_{1 \leq i \leq n} a_i) + 1 - M.$$

Set B has a partition iff set A has a subset with sum of M. If is possible to construct B in polynomial time. Hence sum of subsets $\leq_p$ partition.

6.8.1 Scheduling Identical Processors

- Assume that there are m identical processors P_i, $1 \leq i \leq m$. There are n jobs J_i, $1 \leq i \leq n$. Time required to process each job J_i is t_i. As there are many jobs and many processors, it is important to schedule the activity. Let S is a schedule to allocate processors to jobs. For each job J_i, the time available for job and the processors allocated to it are stored in S. One job can be processed by only one processor at a time. Let job J_i is completed at time f_i and w_i is a weight associated with each job J_i. Also T_i denotes the time at which processor P_i completes all its jobs. We can define the following terms now :
- Mean Finish Time of schedule S :

$$MFT(S) = \frac{1}{n} \sum_{1 \leq i \leq n} f_i$$

- Weighted Mean Finish Time of schedule S :

$$WMFT(S) = \frac{1}{n} \sum_{1 \le i \le n} w_i f_i$$

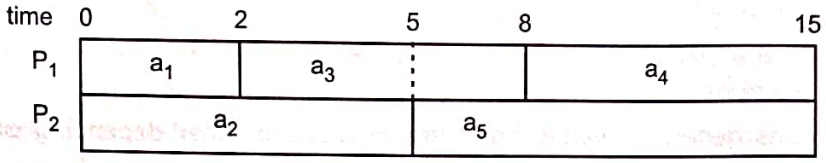
- Finish Time of schedule S :

$$FT(S) = \max_{1 \le i \le m} [Ti]$$

- In preemptive schedule, each job may not be continuously processed from start to finish. In non-preemptive schedule, once started each job is processed until it finishes.

Theorem 6.8.2 : Partition $\le_p$ minimum finish time non-preemptive schedule.

Example 6.8 : *Consider the following jobs instance : $a_1 = 2$, $a_2 = 5$, $a_3 = 6$, $a_4 = 7$, $a_5 = 10$. The minimum finish time schedule S using two processors P_1 and P_2 is as shown below :*



- This is non-preemptive schedule. Processor P_1 processes jobs a_1, a_3, a_4. Processor P_2 processes jobs a_2, a_5. The finish time of this schedule is 15. It leads to a partition problem. $\{a_1, a_3, a_4\}$ and $\{a_2, a_5\}$ are the partitions.

Theorem 6.8.3 : Partition $\le_p$ Minimum WMFT non-preemptive schedule, hence minimum WMFT non-preemptive schedule is NP-hard.

6.8.2 Flow Shop Scheduling

Theorem 6.8.4 : Partition $\le_p$ the minimum finish time preemptive flow shop schedule ($m > 2$). Hence for $m > 2$, obtaining a minimum flow shop schedule is NP-hard.

Proof : Let $m = 3$. Let $A = \{a_1, a_2, ..., a_n\}$ define an instance of the partition problem. Let us construct the preemptive flow shop instance FS as follows : $m = 3$ processors, number of jobs $= n + 2$, at most 2 non-zero tasks per job.

$t_{1,i} = a_i$	$t_{2,i} = 0$	$t_{3,i} = a_i,\ 1 \le i \le n$
$t_{1,n+1} = T/2$	$t_{1,n+1} = T$	$t_{3,n+1} = 0$
$t_{1,n+2} = 0$	$t_{1,n+2} = T$	$t_{3,n+2} = T/2$

where, $T = \sum_1^n a_i$.

- We have to show that the given flow shop instance has a preemptive schedule with finish time at most 2T iff set A has a partition.
  - ➢ If set A has a partition u, then there is a preemptive schedule with finish time 2T.
  - ➢ If set A has no partition, then all preemptive schedules for FS must have a finish time > 2T. This can be shown by contradiction.
- Suppose there is a preemptive schedule for FS having finish time at most 2T, then
  - ➢ Task $t_{1, n+1}$ must finish at time T, as $t_{2, n+1} = T$ and $t_{2, n+1}$ cannot start before $t_{1, n+1}$ finishes.
  - ➢ Task $t_{3, n+2}$ must process for T time, and it cannot start before $t_{2, n+2}$ finishes at T.
- From (i) (a) it is clear that on processor one only T / 2 of the first T time units are unused. Let V be the set of indices of tasks completed on processor 1 by time T excluding task $t_{1, n+1}$. Then we can say that $\sum_{i \in V} t_{1,i} < T/2$ is true because set A has no partition.

Hence, $\sum_{\substack{i \notin V \\ 1 \le i \le n}} t_{3,i} > T/2$ is true.

- The processing of jobs not included in V cannot start on processor 3 until after time T, because processing of these jobs is not completed on processor 1 until time T. Hence this and step (ii) (b) imply that total amount of processing left for processor 3 at time T is $t_{3, n+2} + \sum_{\substack{i \in V \\ 1 \le i \le n}} t_{3,i} > T$.
- Hence the schedule must have length > 2T. Hence for m = 3, the minimum finish time preemptive flow shop schedule is NP-hard.

6.9 NP-HARD CODE GENERATION PROBLEMS

6.9.1 Code Generation

- In computer, different translation programs are used. Assembler is a translator which translates assembly language program into machine language program. The generated code is dependent on the machine for which it is generated. A complier is a program which translates higher-level program into assembly or machine language program.

Definition 6.9.1 : Translation of an expression E, into machine or assembly language of a given machine is optimal iff it has minimum number of instructions.

- The number of instructions generated varies from one machine to other depending on the number of registers available in it. Machines can have only one special register called accumulator, or only one simple register, or only two registers etc.
- Let us see different instruction formats for different machines :

I. For machine A having only special register accumulator :

Instruction	Meaning
a) Load X	Load data from location X to accumulator
b) Store X	Store data from accumulator to location X
c) OP X	OP is Add, sub, Mul, Div.

II. For machines B having number of registers n ≥ 1 :

Instruction	Meaning
a) Load M, R	$M \rightarrow R$
b) Store R, M	$R \rightarrow M$
c) OP R_1, M, R_2	$R_1 + M \rightarrow R_2$
d) OP R_1, R_2, R_3	$R_1 + R_2 \rightarrow R_3$

Example 6.9 : *Generate assembly language code for the instruction (x – y)/(w * z).*

Solution : For machine A having one accumulator :

Way I for machine A	Way II for machine A
Load x	Load w
Sub y	Mul z
Store t_1	Store t_1
Load w	Load x
Mul z	Sub y
Store t_2	Div t_1
Load t_1	
Div t_2	

Way II is better because it generates lesser instructions as compared to way I on machine A.

Now let us generate code for machine B with number of registers N = 1 and N = 2 using way II approach.

N = 1 for machine B	N = 2 for machine B
Load w, R_1	Load w, R_1
Mul R_1, z, R_1	Mul R_1, z, R_1
Store R_1, t_1	Load x, R_2
Laod x, R_1	Sub R_2, y, R_2
Sub R_1, y, R_1	Sub R_1, R_2, R_1
Div R_1, t_1 R_1	

If we observe the instructions carefully, then we will understand that as the number of registers in a machine increase, the overhead of Store-Load decreases. Hence it is important to know that what is the minimum number of registers required to evaluate an expression E without any store instruction. This is the NP-hard problem.

6.9.2 Code Generation for Common Sub-Expressions

- Compiler uses different representations for expressions like triple, quadruple, three address code, postfix expression, trees, dag. If tree is used to represent an expression, then each binary operator represents an internal node. Left and right children represent left and right operands respectively. In general, all the operators are internal nodes and all the operands are leaves of tree. Each node has only one parent.
- If a common subexpression is present in a given expression, then a node may have more than one parents. The resulting structure is called **a directed acyclic graph (dag)**.

Example 6.10 : *Expression tree and a dag for (a + b)* (c – b) is as follows :*

Solution:

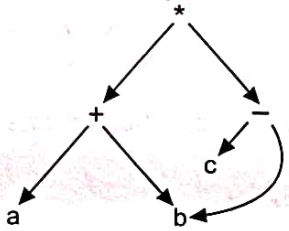


Fig. 6.14: Dag

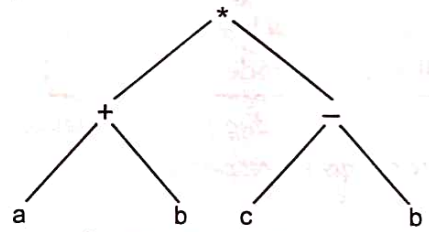


Fig. 6.15: Expression tree

Example 6.11 : *Construct dag for the expression a + (b + c) * (b + c)*

Solution :

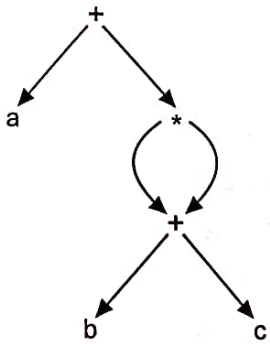


Fig. 6.16: Dag for example 6.11

Example 6.12 : *Construct a dag for a – (b + a/c).*

Solution :

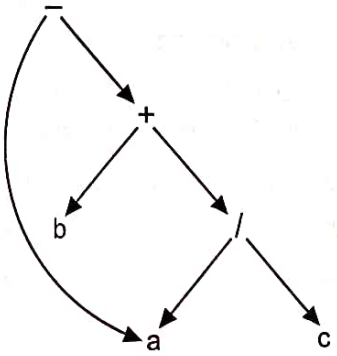


Fig. 6.17: Dag for example 6.12

Example 6.13 : *Construct a dag for (a + b) – (a + b – c).*

Solution :

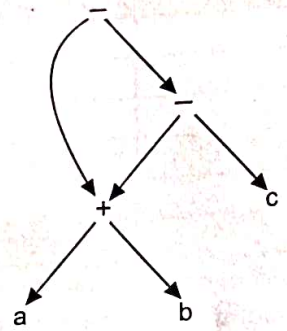


Fig. 6.18: Dag for example 6.13

Some of the definitions related with dag are as follows :

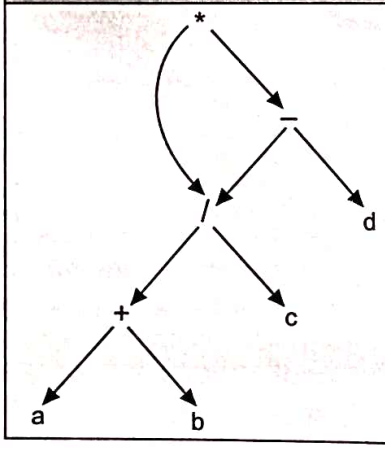
- A **leaf node** in a dag has no children.
- A **level-one node** in a dag has both the children as leaves.
- A **shared node** in a dag has more than one parents.
- A **leaf dag** is a dag in which all shared nodes are leaves.
- A **level-one dag** is a dag in which all shared nodes are level-one nodes.

Examples 7.10 and 7.12 show leaf dags. Example 7.11 and example 7.13 show level-one dags. In example 7.13, + is a level-one node as well as shared node. In example 7.12 '/' is a level-one node while 'a' is a shared node.

Example 6.14 : *Construct a dag for the expression (a + b) /c * ((a + b) / c – d). Also generate optimal code for machines having one and two registers respectively.*

Solution :

Dag	Optimal Code for One Register Machine	Optimal Code for Two Registers Machine

	Load a, R_1
	Add R_1, b, R_1
	Div R_1, c, R_1
	Store R_1, t_1
	Sub R_1, d, R_1
	Mul R_1, t_1, R_1
		Load a, R_1
		Add R_1, b, R_1
		Div R_1, c, R_1
		Sub R_1, d, R_2
		Mul R_1, R_2, R_1

- The optimal code generation problem for a level-one dag is NP-hard even if only one register machine is used. The problem to determine minimum number of registers needed to evaluate a dag with no store instructions is NP-hard. To prove this, we will use the feedback node set (FNS) problem which is NP-hard.

Definition : For a directed graph G(V, E) and an integer K, the feedback node set (FNS) problem is to determine whether there exists a subset of vertices $V^1 \subseteq V$ and $|V^1| \leq K$ such that the graph $H = (V - V^1, E - \{<u, v> \mid u \in V^1 \text{ or } V \in V^1\})$ obtained from G by deleting all vertices in V^1 and all edges incident to a vertex in V' contains no directed cycles.

Theorem 6.9.1 :

FNS $\leq_p$ optimal code generation for level-one dags on a one register machine.

6.9.3 Parallel Assignment Instruction

- A parallel assignment instruction has the format $(v_1, v_2, ..., v_n) = (e_1, e_2, ..., e_n)$ where, the v'_is are distinct variable names and e_i's denote expressions. The value of v_i is updated to the value of expression e_i, $1 \le i \le n$. Expression e_i is executed using the values of variables before execution of e_i.

Example 6.15 : *Parallel assignment instruction*

$$(P, Q) = (Q, R)$$

is equivalent to following two instructions

Solution:

P = Q, Q = R

Refers to new value of Q

Refers to old value of Q.

As Q is used on LHS as well as RHS of parallel assignment instruction it is called a **referenced variable**. Some times it is necessary to copy referenced variables in temporary variables. It is not necessary in all the cases as shown in the following examples :

Example 6.16 : *Parallel assignment instruction*

$$(X, Y) = (Y, X)$$

Solution :

Is equivalent to

t1 = X, X = Y, Y = t1

Example 6.17 : *Parallel assignment instruction*

$(P, Q) = (P + Q, P - Q)$ *is equivalent to*

Solution 1 :

$t_1 = P$

$t_2 = Q$

$P = t_1 + t_2$

$Q = t_1 - t_2$

OR

Solution 2 :

$t_1 = P$

$P = t_1 + Q$

$Q = t_1 - Q$

Consider example 7.17 having

Instruction 1 : P + Q

Instruction 2 : P – Q.

These instructions can be executed in 2 sequences :

- Instruction 1, 2 using solution 1.
- Instruction 2, 1 using solution 2.

Solution 1 uses two temporary variables t_1, t_2. Solution 2 uses only one temporary variable t_1. Depending on the number of instructions n, the different solutions can be obtained.

Let R = (T(1), T(2), ..., T(n)) be a permutation of (1, 2, ..., n). where T (i) denotes one solution. R is called a **realization** of an assignment statement. It specifies the order in which statements appear in an implementation of a parallel assignment statement. The minimum number of instructions of type $t_i = v_j$ for any given realization is called **cost of the realization** C(R).

Example 6.18 : *Refer example 6.15,*

Solution : Cost of the realization

R	C(R)
1, 2	0
2, 1	1

Example 6.19 : *Refer example 6.16,*

Solution : Cost of the realization

R	C(R)
1, 2	1
2, 1	1

Example 6.20 : *Refer example 6.17,*

Solution : Cost of the realization

R	C(R)	
1, 2	1	(optimal)
2, 1	1	(optimal)

Example 6.21 : *Determine all realizations and cost of realization for the parallel assignment statements :*

$$(A, B, C) = (D, A + B, A - B)$$

Solution : Number of variables n = 3. Hence 3! = 6 different realizations are possible as follows :

R	C(R)
1, 2, 3	2
1, 3, 2	2
2, 1, 3	2
2, 3, 1	1
3, 1, 2	1
3, 2, 1	0

As realization 3, 2, 1 i.e.

$$C = A - B, B = A + B, A = D$$

has cost of realization C(R) = 0, hence it requires no temporary variables.

An optimal realization for a parallel assignment statements has minimum cost. Finding an optimal realization requires O(time when all the expressions e_i are all constants or variables. But if e_i's denote expressions, then the problem to determi optimal realization is NP-hard.

Theorem 6.9.2 : FNS $\leq_p$ minimum cost realization.

6.10 SOME NP-HARD PROBLEMS

Some problems are known to be NP-hard or NP-complete. The theorem is as follows :

Theorem 6.10.1 : The following decision problems are in NP :

1. Statisfiability : Is a boolean expression satisfiable ?
2. Clique : Does an undirected graph have a clique of size K ?
3. Vertex cover : Does an undirected graph have a vertex cover of size k ?
4. Hamiltonian circuit : Does an undirected graph have a hamiltonian circuit ?
5. Colorability : Is an undirected graph k-colorable ?
6. Directed Hamiltonian circuit : Does a directed graph have a directed hamiltonian circuit ?

7. Set cover : Given a group of sets $A_1, A_2, ..., A_n$. Does there exist a subgrpup of k sets $A_{i1}, A_{i2}, ..., A_{ik}$ such that

$$\bigcup_{j=1}^{k} S_{ij} = \bigcup_{j=1}^{n} S_j ?$$

8. Exact cover : Given a group of sets $A_1, A_2, ..., A_n$. Does there exist a set cover consisting of a group of pairwise disjoint sets ?

9. Feedback edge set : Does a directed graph have a feedback edge set with cardinality k ?

10. Feedback vertex set : Does a directed graph have a feedback vertex set of cardinality k ?

6.11 PARALLEL ALGORITHMS

- Now-a-days many applications in day-to-day life are such that they demand the operations to be done in a short period of time. In a single processor machine, even though CPU and I/O activities are processed parallely, we can not speedup the operations beyond certain limit.
- So the only solution is to use multiprocessor systems. Such multiprocessor computers can perform parallel computations. Algorithms designed for single-processor machines are called Sequential Algorithms. Algorithms designed for multiprocessor computers are called Parallel Algorithms.

Example: Suppose in a class, there are 5 rows of benches and 10 benches in each row. Let 50 students have occupied 50 benches. If a teacher wants to distribute cards to all the students individually, then it will take 50 seconds. But if 10 cards are given to all first benchers, and are told to distribute them in their row, it hardly requires 10 seconds to distribute 50 cards. This is the case with multiprocessor computers. If there are p processors, then they speedup the work by a factor of p.

6.11.1 Speedup

- Let A be a given problem of size n. Let the best-known sequential algorithm requires a runtime of S'(n) to solve this problem. And a best-known parallel algorithm requires T'(n, p) time on p processor machine. The speedup of parallel algorithm is defined as $\frac{S'(n)}{T'(n, p)}$.
- If both S(n) and T(n, p) are asymptotic, then the speedup is also asymptotic. If the resulting speedup is $\theta(p)$, then it is called Linear Speedup.

Example: For the teachers problem, sequential (individual) distribution requires 50 seconds and parallel distribution requires only 10 seconds. Hence speedup of this parallel distribution process = $\frac{50}{10} = 5$.

6.11.2 Work and Efficiency

- A sequential algorithm performs a task in say T seconds. A parallel algorithm performs more work on 5 processor system in T seconds. A parallel algorithm working on 10 processors performs more tasks as compared to previous in T seconds. Hence, total work done by an algorithm can be expressed as a product of pT(n, p), where p is number of processors and T(n, p) is time to solve problem of size n on p processor machine.
- If a best known sequential algorithm has asymptotic run time S(n) and a p-processor parallel algorithm requires pT(n, p) time, then efficiency of the parallel algorithm is defined as $\left[\frac{S(n)}{pT(n\ \ p)}\right]$. And the parallel algorithm is optimal if pT(n, p) = O(S (n)). A parallel algorithm is said to be work-optimal iff it has linear speedup and the efficiency of such algorithm is $\theta(1)$. If a fraction f of an algorithm has to be run serially, then the maximum speed-up is limited by f. This is stated by Amdahl's law :

$$\text{Maximum speedup} = \frac{1}{f + \frac{1-f}{p}}$$

6.12 COMPUTATIONAL MODEL FOR PARALLEL ALGORITHM

- The basic model of sequential computation is called a random access machine (RAM). In RAM, one operation is performed at a time in one unit of time. But in parallel computation, each processor is given some task. When all the processors finish their task, they have to collect the result of all. So the communication among the processors is very important. Parallel models can be categorized as :
  - ➢ Fixed Connection Machine.
  - ➢ Shared Memory Machine.
- A fixed connection machine represents a graph where nodes denote processors and edges denote direct connection among the processors. If two processors are connected directly, then they can communicate in one step, otherwise data can be send from one processor to other through two or more communication links. Generally, the degree of each node is fixed. The examples of fixed connection machines are shown in Fig. 6.19.

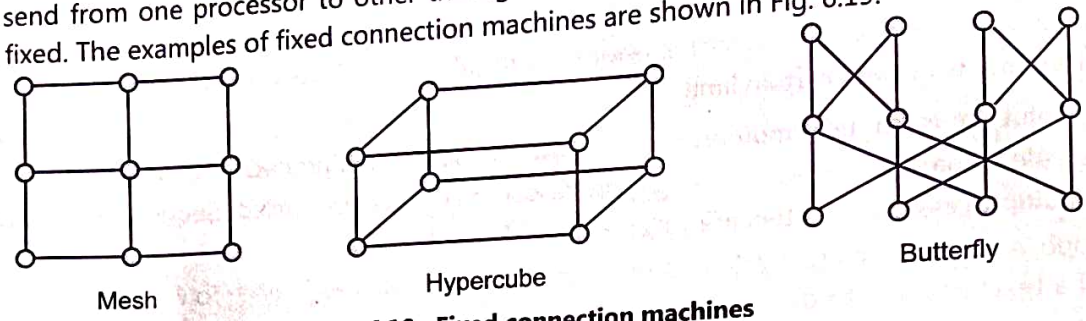


Fig. 6.19 : Fixed connection machines

- In this chapter, we are going to deal with parallel model called PRAM (Parallel Random Access Machine). It has following two properties :
  - ➢ In the PRAM model, p sequential processors share a global memory. The processors communicate through this shared memory in two steps : One processor writes data and other processor reads data from shared memory.
  - ➢ Each processor has its own set of arithmetic and logical instructions. Each processor is actually a random access machine and has its own local memory also.

6.12.1 Read and Write Techniques in PRAM

- Global memory is divided into cells. It is possible that two processors try to access the same cell and conflicts occur. There are different ways to resolve these read and write memory conflicts. Accordingly PRAM models may vary depending on the read-write technique. Different PRAM models are :

1. **EREW (Exclusive Read and Exclusive Write) PRAM**

   In this model, the global memory is shared. No two processors can perform concurrent read or concurrent write on the same memory cell. Obviously two processors can read or write concurrently two different cells.

2. **CREW (Concurrent Read and Exclusive Write) PRAM**

   In this model, two processors cannot write concurrently to the same cell. But they can read same memory cell simultaneously. Obviously they will read same data.

3. **CRCW (Concurrent Read and Concurrent Write) PRAM**

   In this model, two processors can read or write the same memory cell simultaneously. When they are writing, then which processor's data should be written has to be decided. According to way of decision, CRCW can be categorized into the following models :

   - ➢ **Common CRCW Model :** It allows many processors to write the same cell concurrently if all processors want to write same data.
   - ➢ **Arbitrary CRCW Model :** If many processors want to write to the same cell, then only one processor is allowed to write and this processor is selected arbitrarily.
   - ➢ **Priority CRCW Model :** In this model, each processor is assigned a fixed priority statistically. Whenever write conflict occurs, then only the highest priority processor is allowed to write the memory cell.

➢ **Concurrent Read Operation :** Consider the following instructions for reading :

Processor P (in parallel for $1 \le k \le 5$) do Read M[20];

- This instruction denotes that 5 processors want to read memory cell 20 parallelly. If PRAM model is EREW, then concurrent read is not allowed. So 5 processors will perform read sequentially one by one. If PRAM model is CRCW or CREW, then all processors will perform reading simultaneously.
- **Concurrent Write Operation :** Consider the following instruction for writing :

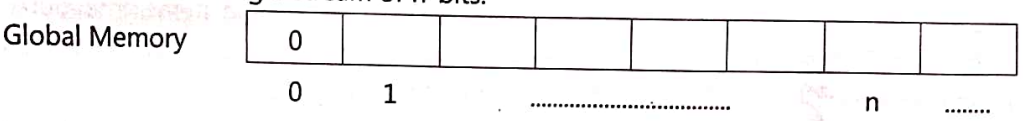
Processor P (in parallel $1 \le k \le 5$) do Write M[20];

- This instruction denotes that 5 processors want to write memory cell 20 simultaneously. If PRAM model is EREW or CREW, then concurrent write is not allowed. If PRAM model is common CRCW, and all the processors want to write the same data, then concurrent write is allowed. If PRAM model is arbitrary CRCW, then only one arbitrarily selected processor is allowed to write. If PRAM model is priority CRCW, then only the highest priority processor is allowed to write memory cell 20.

6.12.2 Boolean Oring in Parallel

Theorem 6.5 : The Boolean OR of n bits can be computed in O(1) time on common CRCW PRAM having n processors.

The theorem can be illustrated considering a stream of n-bits.



Let n-bits are stored in memory cells, 1, 2, 3, ... , n respectively. Let us store answer of logical ORing in cell 0. We can write instructions to do this operation as shown below :

Processor P (in parallel for $1 \le k \le n$) do

if (cell [k] = 1) then cell [0] = 1.

Assume that initial value of cell [0] is 0. As the PRAM model is common CRCW, all the processors which found cell data = 1 will try to write 1 to cell 0 and the concurrent write is allowed because they are writing same value 1.

Slow-Down Lemma 6.6 : Any parallel algorithm that runs on a p_1-processor machine in time t_1 can be run on a p_2 - processor machine in time $O((p_1 \times t_1)/p_2)$ where $p_2 < p_1$.

6.13 BASIC TECHNIQUES AND ALGORITHMS

In this section, we will study some basic techniques and parallel algorithms.

6.13.1 Complete Binary Tree

- When we want to perform same operation on n data items, then we can use complete binary tree for computation. In a complete binary tree, n data items are represented as n leaves and it has (n – 1) internal nodes. Each internal node has two children. Let the tree is stored in an array B[ ] from 1 to 2n – 1. Each node B[x] has its left child at B[2x] and right child at B[2x + 1].
- Suppose we want to multiply n integers. These n integers can be stored from B[n] to B[2n + 1] locations which represent leaves of a tree. Let $n = 2^k$, hence k = log n. Then the complete binary tree has k + 1 levels

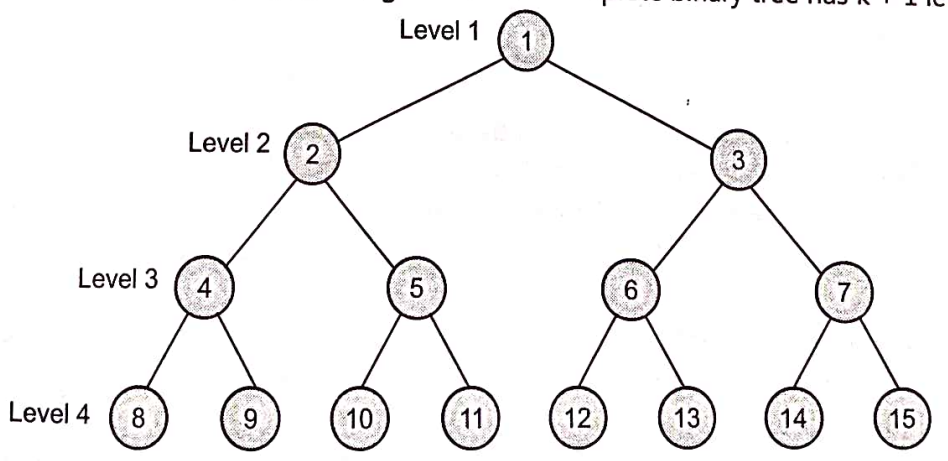


Fig. 6.20 : Complete binary tree

- The multiplication can be carried out parallely at each level from level k + 1 to level 1. n integers at level k + 1 can be multiplied by n/2 parallel processors and will be stored at level k internal nodes. The process continues up to root giving final result. In general, B[2x] and B[2x + 1] will be multiplied and result will be stored at B[x]. Final result of root node will be B[1] at root node B[1]. The algorithm for parallel multiplication is given below:

Function ParallelMult (T, n)

```
begin
    for y = (log n – 1) to 1 step -1 do
        for x = 2^y to (2^(y+1) – 1) in parallel do
            B[x] = B[2x] * B[2x + 1]
        end for
    end for
    return B[1]
end
```

- Here we are assuming that each multiplication requires constant time to read 2 numbers, multiply them and store the result. Maximum number of processors required is n/2 as there are n data items. The outer loop of algorithm is executed log n times, hence time complexity of algorithm is θ(log n). We can apply the same algorithm to add n numbers, to find maximum of n numbers, to find minimum of n numbers etc.

  For example, multiplication of numbers 2, 3, 5, 4 using complete binary tree is as shown below in Fig. 6.9. The root value gives result.

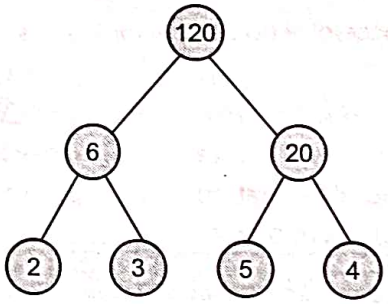
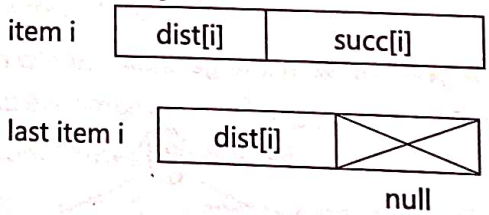


Fig. 6.21 : Multiplication using complete binary tree

6.13.2 Pointer Doubling

- Pointer doubling algorithm applies to items represented in the form of a linked list. Consider the following linked list of n items where each item i has pointer to its successor succ[i]. If i is a last item in the list, then succ[i] points to null pointer. Suppose nodes in a linked list have following fields :



- Suppose we want to find distance of each item from the end of list. We will store distance of item i in dist[i] field. As shown in the above Fig. 6.10 we can initialize dist[i] field to 1 for all the items except last for which dist[i] = 0. Then as shown in iteration 1 of Fig. 6.10, we can double the pointers of each node so that succ[i] points to a node two places away from it. But before doubling pointers, we will add distances of item i and its successor, that is, dist[i] and dist[succ[i]].
- In iteration 2, again the process is repeated and pointers are doubled. So succ[i] points to a node four places away from it in the original list. As there are n items in list and if $n = 2^k$ almost, then we will repeat the process log n times. Obviously, the assumption is that for each item i, a separate processor is assigned. Hence at the end of last iteration, we have dist[i] of each item calculated. Here the original list is destroyed. If we want to retain it, then operations can be performed on a copy of original list.

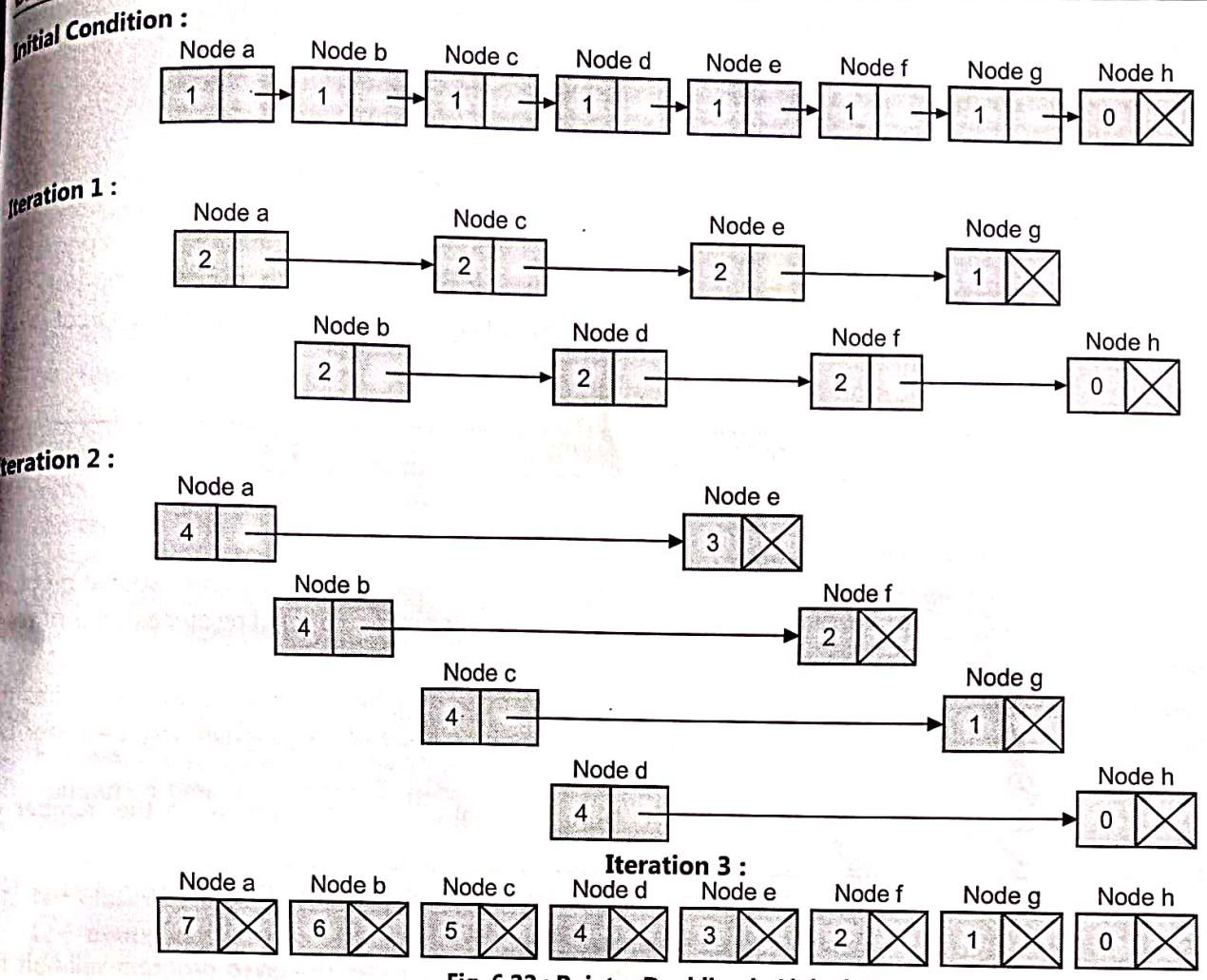


Fig. 6.22 : Pointer Doubling in Linked Lists

- The most important thing here is that as all the processors are working parallely on the same instruction, there must be proper synchronization. For example, in iteration 1

  dist[1] = dist[1] + dist[2], and dist[2] = dist[2] + dist[3]

- Here dist[2] is used in both the instructions. But dist[2] on R.H.S. means the old value and dist[2] on L.H.S. refers to new value. So both the instructions should be synchronized properly such that first instruction reads dist[2] location before it gets modified by second instruction. Let us write the algorithm to find distance of each item in a list using pointer doubling :

Algorithm: ParallelDistance (List)

```
begin
  //Initialize dist[i] for all items
  for each item i ∈ List in parallel do
      if succ[i] = null
      then dist[i] = 0
      else dist[i] = 1
  end for
  //Pointer doubling
  repeat ⌈ log n ⌉ times
  begin
  for each item i ∈ List in parallel do
      if succ[i] ≠ null
      then
          dist[i] = dist[i] + dist[succ[i]]
```

```
            succ[i] = succ[succ[i]]
        end if
    end for
end
end
```

- Analysis of algorithm is very simple. Assuming that each item is assigned one processor and each memory access requires a constant time, the computing time is $\theta(\log n)$ as it requires log n iterations to complete the task. This technique is useful in many applications. For example, to add all consecutive 4 items, or to find multiplication of all the descendant items in list etc.

SOLVED EXERCISE

I. Question Answers :

1. Write a short note on : computational complexity.

Ans. : The term computational complexity refers to the cost of computation required to solve a computational problem. The measurement of computational complexity involves measurement of different computational resources. Two of them are very important : space complexity and time complexity.

- Space complexity specifies the amount of memory required to execute a program. Time complexity specifies the amount of time required to execute a program. There is always space time trade-off. A program may be faster but requires more memory or it may use less memory but it is slower.
- If an algorithm is designed for parallel execution, then its computational complexity depends on the number of parallel processors also.

2. State halting problem and prove that it is undecidable. Which category does it belong to (P or NP-complete or NP-hard or none of these) ?

Ans. : Given a program and an input string, the halting problem is to determine whether the given program will halt for the given input, or not.

- We will consider a halting problem as a decision problem. So if program P halts on input I, then output is 1, otherwise output is 0.

Proof : We will prove the statement by contradiction. Assume that there exists a turing machine TM. So TM (P, I) halts iff P halts on input I.

Consider the following procedure A executed on TM :

Procedure A(B)

```
begin
    if TM (B, B) then
        loop forever
    else
        Halt
    end if
end
```

Let us see how procedure A behaves if the code of procedure A is given as an input to itself. There are only two possibilities :

1. Assume that procedure A halts on input A. Hence by the correctness of turing machine, TM (A, A) returns true. Hence control will execute "if statement" and procedure A loops forever on input A. This is contradictory to our assumption.
2. Assume that procedure A does not halt on input A. By the correctness of turing machine TM (A, A) returns false, hence procedure A executes "else statement" and halts. But again this is contradictory to our assumption.

Hence halting problem is undecidable. It is NP-hard but not NP-complete.

3. Define the terms : deterministic algorithm and non-deterministic algorithms.

Ans. : Algorithms in which the result of every operation is uniquely defined are called deterministic algorithms. Machines which execute deterministic algorithms are called deterministic machines. Such machines exist practically.

- Algorithms which contain certain operations whose results are not uniquely defined but are limited to specified set of possibilities are called non-deterministic algorithms. Machines to execute non-deterministic algorithms do not exist practically.
- To specify non-deterministic algorithms, we can use the following functions :
  - **Choice (Set S) :** It returns one of the values in set S arbitrarily.
  - **Success ( ) :** It denotes a successful completion of an algorithm.
  - **Failure ( ) :** It denotes an unsuccessful termination of an algorithm.
- All the above functions have O(1) computing time.
- The Choice function can return any value in the set S in any order. Hence depending on the order in which values from input are selected affects the successful termination of a non-deterministic algorithm. If there exists at least one set of choices which result in successful completion and if it is made, then the algorithm completes successfully. If there is no set of choices which can result unsuccessful completion of an algorithm, then the algorithm always terminates unsuccessfully. A non-deterministic machine can execute a non-deterministic algorithm. But practically such non-deterministic machines do not exist.

4. Explain the complexity classes : P and NP. Is P = NP ?

Ans. : The algorithms are classified into two groups depending on their computing time :

1. **P Class :** This group consists of all the algorithms whose computing times are polynomial time, that is, there computing time is bounded by polynomials of small degree. For example, insertion sort requires $O(n^2)$ time. Merge sort and quick sort require O(n log n) time. Binary search tree require O(log n) time for searching. Polynomial evaluation requires O(n) time. All these algorithms have polynomial computing time.
2. **NP Class :** This group consists of all the algorithms whose computing times are non-deterministic polynomial time. For example, computing time of the travelling salesperson problem is $O(n^2 2^n)$ using dynamic programming. Computing time of the knapsack problem takes $O(2^{n/2})$ time. These are called non-polynomial algorithms.

   The NP class problems again can be classified into two groups :
   - *NP-Complete (Non-deterministic Polynomial time complete) problems.*
   - *NP- hard problems*

- *No NP-complete or NP-hard problem is polynomially solvable. All NP-Complete problems are NP-hard but some NP-hard problems are not NP-Complete.*
- *The relationship between P and NP is shown in following Fig. 6.23*

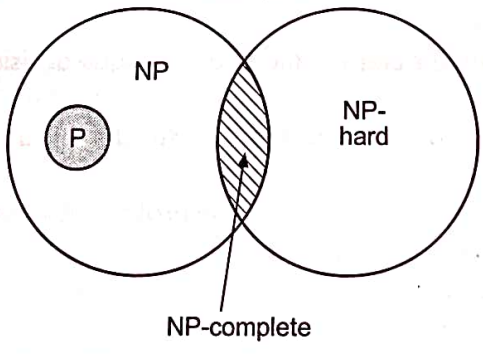


Fig. 6.23 : Relation between P and NP

- Since deterministic algorithms are special cases of non-deterministic algorithms. $P \subseteq NP$. Also all the NP-complete problems are NP-hard problems, but some NP-hard problems are not NP-complete. For example, the halting problem is an NP-hard problem, but not NP-complete. Given a deterministic algorithm A and input I, the halting problem is to determine that if algorithm A is run with input I, whether it will stop or enter in an infinite loop.

5. Define the terms : decision algorithm and optimization algorithm.

Ans. : Decision problem : Any problem having the answer either zero or one is called a decision problem. Only a decision problem can be NP-complete.

- **Decision Algorithm :** It is an algorithm which solves a decision problem. It gives output 1 on successful completion. On unsuccessful termination, it gives output 0.
- **Optimization Problem :** Any problem that involves the identification of an optimal value for a given cost function is known as an optimization problem. The optimal value can be either maximum or minimum. An optimization problem may be NP-hard.
- **Optimization Algorithm :** It is an algorithm which solves an optimization problem.

EXERCISE

1. State Cook's theorem and prove it.
2. Write a short note on :
   (a) P complexity class
   (b) NP complexity class
3. Explain the following terms :
   (a) Computational complexity
   (b) Optimization problem
   (c) Decision problem
   (d) Stable algorithm
   (e) Tractability and intractability
4. Explain deterministic and non-deterministic algorithms.

UNIVERSITY QUESTION BANK

1. (a) Show that knapsack decision problem is NP-Complete.
   (b) Prove that Maxclique problem can be solved in polynomial time iff the clique decision problem can be solved in polynomial time.
   (c) Show that the clique optimization problem reduces to the clique decision problem.
2. Write short notes on :
   (a) NP complete problem,
   (b) Cook's theorem.
3. (a) Show that the max clique problem reduces to clique decision problem.
   (b) Show that knapsack decision problem is NP-complete.
   (c) What is Cook's theorem ? How can it be used to establish whether P = NP ? or P ≠ NP ?
4. What do you mean by
   (i) NP-hard
   (ii) NP-complete problem ?
5. (a) State what is meant by the following classes of problems : NP-Hard and NP-complete. Give example of each.
   (c) Comment on the following : "A deterministic interpretation of a non-deterministic algorithm can be made by allowing unbounded parallelism in computation."
6. Show that the clique optimization problem reduces to the clique decision problem.
7. (a) Show that the job sequencing with deadlines problem is NP-hard.
   (b) Show that the clique optimization problem reduces to the clique decision problem.
   (c) Describe in brief 'Cooks theorem'.
8. (a) A certain problem can be solved by an algorithm whose running time is in $O(n^{\log_2 n})$. Which of the following assertions is true ?
   (i) The problem is tractable
   (ii) The problem is intractable
   (iii) None of the above. Justify your answer.
9. Describe with example following class
   (i) P
   (ii) NP
10. A sorting method is stable if equal elements remain in the same relative order in the sorted sequence as they were originally, which of the following sorting methods are stable ? Justify in brief.
    (i) Bubble sort
    (ii) Merge sort
    (iii) Heap sort
    (iv) Quick sort.
11. State Halting problem and prove that it is unavoidable. Which category does it belong to ? (P or NP complete or NP hard or none of these).
12. State Satisfiability problem and show how it is NP.

✠ ✠ ✠

MODEL QUESTION PAPERS

Mid Sem. Examination

Time: 1 Hour **Max. Marks: 30**

Instructions to the candidates:

(1) Answer Q. 1 or Q.2, Q. 3 or Q. 4.

(2) Figures to the right side indicate full marks.

(3) Neat diagrams must be drawn wherever necessary.

(4) Mere reproduction from IS code as answer, will not be given full credit.

(5) Assume suitable data, if necessary.

1. **(a)** What is an algorithm? Explain it in detail. **(5)**

   (b) Write essential properties and the performance measures of an algorithm. **(5)**

   (c) Write short note on space and time tradeoff in programs. **(5)**

OR

2. **(a)** Write short note on Big 'O' notation. **(5)**

   (b) What are the factors that determine execution speed of a program ? **(5)**

   (c) Compare the growth rate of the following functions. Which one is having the highest rate ?

   $n!, 2^n, n^2, n \log n, n^3$. **(5)**

3. **(a)** Write a pseudo 'C' algorithm for Kruskal's algorithm. **(5)**

   (b) Explain the greedy strategy. **(5)**

   (c) Explain divide and conquer strategy. **(5)**

OR

4. **(a)** Apply quick sort and merge sort :

   10, 30, 70, 80, 40, 60, 50, 20 **(5)**

   (b) Describe in brief : Time complexity of quick sort algorithm. **(5)**

   (c) Using divide-conquer, prove that the computation time for multiplication of two polynomials of type $a_0 x^n + a_1 x^{n-1} + \ldots a_{n-1} x + a_n$ is of order $O(n^{1.51})$. **(5)**

End Sem. Examination

Time : 3 Hours **Max. Marks : 70**

Instruction to candidates :

(1) Answer Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8.
(2) Figures to the right indicate full marks.
(3) Neat diagrams should be drawn wherever necessary.
(4) Assume suitable data, if necessary.

1. (a) Discuss following :
(i) Performance analysis of algorithm and (ii) Asymptotic notation. (6)

(b) Analyse the following :
(i) For loops. (ii) Recursive calls. (iii) While loops. (6)

(c) Explain with an example relationship of data structure and algorithms in the process of problem solving. (6)

OR

2. (a) Explain with neat sketch about job sequencing with deadline. (6)

(b) Find feasible solutions for the following knapsack instance :
Let n = 5, M = 100, w = {10, 20, 30, 40, 50}, P = {20, 30, 66, 40, 60} (6)

(c) Explain in detail about Prim's Algorithm. (6)

3. (a) Explain dynamic programming approach. (6)

(b) State the principle of optimality and explain it in brief. (6)

(c) Generate sets S_i, $0 \le i \le 4$ when weights (w1, w2, w3, w4) = (1, 11, 21, 23) and profits (p1, p2, p3, p4) = (11, 21, 31, 33). (6)

OR

4. (a) Compare Greedy and Dynamic strategies. (6)

(b) Write algorithm for OBST (Optimal Binary Search Tree). (6)

(c) Write dynamic based algorithm for computing minimum cost from source s to destination t in multistage graph. (6)

5. (a) Explain backtracking strategy in detail. (6)

(b) Let P = {12, 22, 32, 34, 44, 54, 56, 66} and W = {2, 12, 22, 24, 34, 44, 46, 56}.
Draw the state space tree for n = 8 and M = 110. (6)

(c) Discuss in brief :
(i) Huffman's codes.
(ii) 8-queens problem (5)

OR

6. (a) Write backtracking algorithm for graph coloring problem. (6)

(b) Write control abstraction for LC search. (6)

(c) Define reduced cost matrix. Explain how we can obtain reduced cost matrix. (5)

7. (a) Devise LC branch and bound algorithm for traveling salesman problem. (6)

(b) Show that knapsack decision problem is NP-Complete. (6)

(c) State what is meant by the following classes of problems : NP-Hard and NP-complete. Give example of each. (5)

OR

8. (a) Show that the clique optimization problem reduces to the clique decision problem. (6)

(b) State Satisfiability problem and show how it is NP. (6)

(c) Explain deterministic and non-deterministic algorithms. (5)