CHAPTER - 2

ON INPUT OUTPUT REPRESENTATION OF THE GRAPH ALGORITHMS
AND THEIR DESIGN AND ANALYSIS APPROACH

2.1. Introduction :

Fundamental to any graph algorithm design is the identification of a graph in a list of many graphs by a computer. There are alternative unique representations for the same graph and one representation may require for an algorithm comparatively less searching effort and space than the other. Clearly, our ultimate goal is to represent the graph in a computer uniquely and at the same time for a particular algorithm it must help give a simple nonexpensive solution. There is no systematic procedure to guide us in the matter. The process is to start with a standard representation and improve upon it depending upon the experience gained in attempts to design an algorithm with the graph as input. The experience allows us to recognise the occurrence of simple notions like symmetries or useful sequence of operations that may lead us quickly towards solution. Thus we must first tell in this chapter what are the standard ways of representing a graph in a computer and in what sense the representation can be called economical.

The selection and generation of some test graphs are discussed next. The criterion is to start with graphs for which the solution space for a particular problem is known. For example Moon Moser graph contains known number of cliques, complete graphs contain exponential number of trees etc. These graphs of our interest must be generated by a computer for large test inputs.
Finally, we have discussed in this chapter the approaches that are adopted to design algorithms in this thesis and also the analysis approach of the algorithms.

2.2. Representation of Graphs in a Computer *(N-1, E-3)

A graph $G$ consists of two sets called the vertices $V$ and edges $E$. $V$ is a finite nonempty set of vertices (sometimes called nodes) usually numbered 1, 2, ..., $n$ and $E$ is a finite set of pairs of vertices. Each pair in $E$ is an edge of $G$. In this thesis we are only interested in undirected simple connected graphs. In an undirected graph we say that the vertex $i$ is adjacent to the vertex $j$ if the edge $(i, j)$ exists. The degree of a vertex is the number of its adjacent vertices.

There are two common ways to represent graphs in a computer, sequential and linked.

2.2.1. Sequential representation *(E-6, N-1)

We are considering a graph $G$ of $n$ number of vertices and $e$ number of edges.

a) Adjacency matrix: The adjacency matrix of graph $G$ is the $n \times n$ matrix $A = A(G)$ whose entries $a_{ij}$ are given by

$$
a_{ij} = \begin{cases} 
1 & \text{if } i \text{ and } j \text{ are adjacent} \\
0 & \text{otherwise}.
\end{cases}
$$

*References are indicated within the first bracket.*
For the graphs we are concerned with, the adjacency matrix is symmetric, since
the edges are undirected, and its diagonal terms are zero, since loops are
forbidden. Thus $A$ is a real symmetric matrix with zero trace.

Since the rows and columns of $A$ correspond to an arbitrary labelling
of the vertices of $G$ it is clear that we are interested in those properties of
adjacency matrix which are invariant under the permutation of rows and columns.

Obviously, the storage space of $A$ in computer requires $n^2$ words of
sequential location stored in row major or column major order. For undirected
loop free graphs it would suffice to keep only an upper triangular matrix, or
$n(n-1)/2$ elements. Note that the main diagonal need not be stored as
$(a_{ii}) = 0$. A saving of almost 50% results but computational time may increase
slightly because any reference to $a_{ij}$ must be replaced by "if $i > j$ then $a_{ji}$
else $a_{ij}$".

This does not necessarily mean that we must set aside this number of
computer words to store a single graph. Since elements of adjacency matrix
are 0 and 1 and the storage requirements are reduced by a factor that depends
on the length of the computer word.

Before begining any computation on a graph we will normally have to
initialise an adjacency matrix so that it contains the graph we are going to
operate on. This step will typically require at least $O(n^2)$ operations. Thus
the computing time of most algorithms using this form of representation will
require at least $O(n^2)$ operations even if the graph has only $O(n)$ edges.
(b) **Incidence matrix**: The incidence matrix $D$ of a graph $G$ with $n$ vertices and $e$ edges is a $n \times e$ matrix $(d_{ij})$ whose entries are

$$d_{ij} = \begin{cases} 1 & \text{if edge } e_j \text{ is incident with vertex } V_i \\ 0 & \text{otherwise.} \end{cases}$$

The rows of incidence matrix correspond to the vertices of $G$ and its columns correspond to the edges of $G$. Each column of $D$ has two nonzero entries $+1$, $+1$. The number of 1's in each row gives the degree of the corresponding vertex.

An incidence matrix requires $n \cdot e$ words of sequential memory location.

If the word length is large several entries can be packed in a word for space saving. Obviously, if $e$ is large compared to $n$ adjacency matrix requires less storage. However, for some algorithms like tree and circuit generation the incidence matrix is computationally advantageous. Thus whenever number of edges of a graph is small compared to the vertices, the incidence matrix requires less storage but the selection of representation is also dictated by the algorithm we are interested in.

(c) **List of edges**: If a graph is sparse i.e., $e = O(n)$, it may be advantageous to list the edges present in the graph as pairs of vertices. This representation can be implemented with two arrays $g = (g_1, g_2, \ldots, g_n)$ and $h = (h_1, h_2, \ldots, h_n)$. Each entry is a vertex label, the $i$th edge in the graph going from vertex $g_i$ to vertex $h_i$. The number of words required to label each vertex is $K = \lceil \log_2 n \rceil$. We need at least $2e \cdot k$ words for storing the list. Obviously if $2ek < n^2$ the edge listing is advantageous over adjacency matrix in respect of storage requirement.
2.2.2. **Linked representation**:

(a) **Adjacency list**: If a graph $G$ is sparse and insertion and deletion are required often for an algorithm on such graphs, we have sufficient motive for linked representation of $G$. The adjacency list of $G$ consists of $n$ lists one for each vertex $i$. Two vertices are neighbours if an edge interconnects them. The list for vertex $i$ contains neighbours of $i$. Because we often need to access the adjacent vertices of a random vertex we insist that the heads of the lists are stored sequentially. But the list of a vertex's neighbours may be linked together.

Since each edge will require 1 additional word for linking the total word requirement are $n + 2e$ locations.

(b) **Incidence list**: This is a list in which for each vertex $i$ we list all the edges incident on $i$ along with $i$ as a linked list. This representation has some advantage in algorithms requiring nodes and edges to be reached simultaneously (Fig. 2.1).

**Illustration**:

All representations of the graph $G$ of Fig. 2.1 are listed below.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency Matrix             Incidence Matrix
2.3. Test Graphs:

For the four problems discussed in the thesis we have chosen different graphs as inputs. The choice of the graphs was motivated by the demand of the particular algorithm. Thus complete or near complete graphs were chosen for tree generation and circuit generation algorithms, Moon Moser graph for clique testing algorithms, Grotzsch graph for node coloring algorithm etc. Random graphs of various density was chosen as input for all algorithms for time efficiency estimation.

2.3.1. Some special graphs with exponential number of trees: \((n-1, n-2)\)

(a) Complete graphs (cliques): A simple symmetric graph \(G\) is complete if there exists an edge between every pair of vertices. Thus for \(n\) node graph \(G\), the number of edges is \(n(n-1)/2\) and degree of each node is \((n-1)\). The number of trees of \(G\) is

\[
K(G) = n^{n-2}
\]

Generation of complete graph is simple as the adjacency matrix consists of all 1 excepting the diagonals (Fig. 2.2).
(b) Let $G$ be a graph constructed by removing $q$ disjoint edges from a complete graph of $n$ vertices ($n \geq 2q$), then (3.1)

$$K(G) = n^{n-2}(1 - 2/n)^q$$

(c) Let $G$ be a graph constructed by removing $p$ edges from a single node then, (4.1)

$$K(G) = n^{p-1} (n-p) (n+1)^{n-1-p}$$

(d) Let $G$ is a connected regular graph having $n$ vertices of degree $K$, then

$$K(G) \leq \frac{1}{n} \left( \frac{nK}{n - 1} \right)^{n-1}$$

Thus if it is complete graph then $K = n-1$ or $K(G) = n^{n-2}$, the formula in (a).

We can generate any of the graphs of special structures in the four categories above for testing tree generation algorithms. These graphs contain a large number of trees and can help in time efficiency estimations also.
2.3.2. Moon-Moser graph; \((E-4)\)

Moon-Moser graphs have the most cliques possible for a graph with a given number of vertices. Also the number of cliques for given class of Moon-Moser graphs is known. A graph with \(n\) vertices can have at most \(f(n)\) cliques, where

\[
\begin{align*}
    f(n) &= \begin{cases} 
        3^{n/3} & n \equiv 0 \pmod{3} \\
        4 \cdot 3^{(n-4)/3} & n \equiv 1 \pmod{3} \\
        2 \cdot 3^{(n-2)/3} & n \equiv 2 \pmod{3}
    \end{cases}
\]

Obviously such a graph can be used as test inputs to clique generation algorithms. The problem is how to generate these graphs in a computer.

The construction shown here is for one of the three cases cited above.

Let \(M_k\) be a Moon-Moser graph with \(3K\) vertices \((1, 2, \ldots, 3K)\), in which the vertices are partitioned into trials \(\{1, 2, 3\}, \{4, 5, 6\}, \ldots, \{3K-2, 3K-1, 3K\}\); \(M_k\) has no edge within any trial, but, aside from that, each vertex is connected to every other vertex. \(M_1\), \(M_2\) and \(M_3\) are shown in Fig. (2.3).

**Illustration:**

\[
\begin{bmatrix}
    1 & 2 & 3 \\
    1 & 0 & 0 & 0 \\
    2 & 0 & 0 & 0 \\
    3 & 0 & 0 & 0
\end{bmatrix}
\]

Adjacency matrix of Fig. 2.3(a)
Algorithm 2.6: This Algorithm generates Moon-Moser graph in adjacency matrix form in the two dimensional array NADJ. It requires as input number of Triads of vertices K only.

Step 1: Read K
Step 2: Let $N = 3K$
Step 3: Initialise whole $N^2$ elements of NADJ $(N, N)$ by 1
Step 4: Let $M = 1$
Step 5: $MM = M + 2$
Step 6: Put all 0 in the submatrix with row and column heads from $M$ to $MM$
Step 7: If $M > N$, then Go To 9 otherwise continue
Step 8: $M = M + 3$ Go To to Step 5
Step 9: Stop.

Adjacency matrix of Fig. 2.3(b)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 1 & 1 & 1 \\
3 & 0 & 0 & 0 & 1 & 1 & 1 \\
4 & 1 & 1 & 1 & 0 & 0 & 0 \\
5 & 1 & 1 & 1 & 0 & 0 & 0 \\
6 & 1 & 1 & 1 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Adjacency matrix of Fig. 2.3(c)

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
3 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
4 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
5 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
6 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
7 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
8 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
9 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{bmatrix}
\]
Theorem 2.1: If \( N = 3^K \) then the Moon Moser graph generated by algorithm 2.0 contains exactly \( 3^K \) cliques.

Proof: By construction, theorem is true for \( K = 1, 2, 3 \).

Let it is true for any \( (K-1) \). Thus \( M_{K-1} \) graph contains \( 3^{K-1} \) cliques each containing \( K-1 \) vertices, then each of the three vertices added to form \( M_K \) form a clique with each of the \( 3^{K-1} \) cliques of \( M_{K-1} \); since they are obviously the only new cliques formed \( M_K \) has \( 3 \cdot 3^{K-1} = 3^K \) cliques each containing \( K \) vertices.

\( M_K \) thus has a number of cliques that is exponential in the number of vertices.

2.3.3. Critical graphs: (J-2)

2.3.3.1. \( K \) critical graph:

When dealing with node coloring algorithm, it is helpful to study the properties of a special class of graphs called critical graphs. We are to develop an algorithm to generate such critical graphs by computer. At the outset, we must define and illustrate the meaning of \( K \)-critical graphs.

A \( K \)-vertex coloring of \( G \) is an assignment of \( K \)-colors, \( 1, 2, \ldots, K \) to the vertices of \( G \); the coloring is proper if no two adjacent vertices have the same color. By \( K \)-coloring of a graph \( G \) we mean proper \( K \)-coloring of \( G \). The chromatic number \( \chi(G) \) of \( G \) is the minimum \( K \) for which \( G \) is \( K \)-colorable. If \( \chi(G) = K \) then \( G \) is \( K \)-chromatic.
A graph $G$ is critical if $X(H) < X(G)$ for every proper subgraph $H$ of $G$. A $K$-critical graph is $K$-chromatic and critical; every $K$-chromatic graph has a $K$-critical subgraph.

2.3.3.2 Triangle free $K$-critical graphs : (J.2)

In any coloring of a graph $G$, the vertices in a clique (contained in $G$) must all be assigned different colors. Thus a graph with a large clique necessarily has a high chromatic number. What is perhaps surprising is that there exists triangle free graphs with arbitrarily high chromatic numbers. A recursive construction of such graphs with an algorithm for these construction will be given here.

**Theorem 2.2**: For any positive integer $K$ there exists a $K$-chromatic graph containing no triangle.

**Proof**: For $K = 1$ and $K = 2$, $K_1$ and $K_2$ (complete graphs of 1 and 2 vertices) have the required property (Fig. 2.4). We proceed by induction on $K$. Suppose that we have already constructed a triangle-free graph $G_k$ with chromatic number $K \geq 2$. Let the vertices $G_k$ be $v_1, v_2, \ldots, v_n$. Form a new graph $G_{k+1}$ from $G_k$ as follows: add $n+1$ new vertices $u_1, u_2, u_3, \ldots, u_n, v$ and then, for $1 \leq i \leq n$ join $v_i$ to the neighbours of $v_i$ and to $v$ (Fig. 2.4).

The graph $G_{k+1}$ clearly has no triangles. For, since $\{u_1, u_2, \ldots, u_n\}$ is an independent set in $G_{k+1}$, no triangles can contain more than one $u_i$, and if $u_i v_j v_k u_i$ were a triangle in $G_{k+1}$, the $v_i v_j v_k v_i$ would be a triangle in $G_k$, contrary to assumption.
We now show that $G_{k+1}$ is $(K+1)$-chromatic. Note first that $G_{k+1}$ is $(K+1)$-colorable. Now we must show that $G_{k+1}$ is not $K$-colorable. If possible consider a $K$-coloring of $G_{k+1}$ in which, without loss of generality, $v$ is assigned color $K$. Clearly, no $u_i$ can also have color $K$. Now reorder each vertex $v_i$ of color $K$ with color assigned to $u_i$. This results in a $(K-1)$ coloring of the $K$-chromatic graph $G_k$. Therefore $G_{k+1}$ is indeed $(K+1)$-chromatic.

The theorem follows from the principle of induction.

Algorithm 2.3: This algorithm generates $K$-chromatic triangle free graph, in adjacency matrix $NADJ$.

1. Generate $A(G_k)$ of $K$ vertices.
2. Form upper half of $A(G_{k+1})$ by repeating $A(G_k)$ in the $K$ columns starting from $K+1$th column and adding $0$'s in the rightmost column.
3. Form low left half by repeating $A(G_k)$ and putting zeros in $K$ columns of last row.
4. Form lower right half by putting zeros in $K \times K$ matrix except last $K$ columns and last $K$ rows.
5. Put 1's in the remaining entries.

2.3.3.3. Computer generation of triangle free $K$-critical graphs:

By starting with the 2-chromatic graph $G_2$ we can construct for all $K \geq 2$ a triangle free $K$-chromatic graph of $3.2^{K-1} - 1$ vertices. For computer generation, we start with adjacency matrix of $G_2$, i.e., $A(G_2)$. We observe that $A(G_3)$ can be constructed as follows:
(17)

\[
A(G_3) = \begin{bmatrix}
A(G_2) & A(G_2) & 0 \\
A(G_2) & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Similarly,

\[
A(G_4) = \begin{bmatrix}
A(G_3) & A(G_3) & 0 \\
A(G_3) & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

In algorithm 2.3 we have generated the K-chromatic triangle free graphs of
vertices 2, 5, 11, 23, ... \((3 \cdot 2^{K-1})\) recursively by starting with adjacency
matrix of \(G_2\).

**Adjacency Matrices:**

\[
\begin{array}{c|c|c}
1 & 2 & \\
1 & 0 & 1 \\
2 & 1 & 0 \\
A(G_2) & & \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
1 & 2 & 3 & 4 & 5 & \\
1 & 0 & 1 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 1 & 1 & 0 \\
A(G_2) & & & & & \\
\end{array}
\]
### RANDOM graph generation: (D.1, D.2)

For time complexity estimation of an algorithm the test graphs chosen are often random graphs. The procedure is to compare two algorithms a and b (say) by feeding both with a large number random graphs and computing average time of execution.

We define density D of edges in a graph G as the quotient $\frac{\text{ISUM}}{n(n-1)}$ where ISUM is the sum of degrees of all nodes of G and n is the number of nodes. The logic for such erratic definition is that for simple symmetric graph
G of n-nodes the complete graph of n nodes have $B_{3TJM} = n(n-1)$ and that is the largest $B_{3TJM}$ possible. Thus we do some sort of normalization and preclude that the $D$ is maximum when it is 1. Sometimes, random graphs are classified to be of low, medium and high density depending on the values of $D$.

2.3.4.1. Random graphs are generated as follows:

(a) The number of nodes is fixed as well as a number $p (0 \leq p \leq 1)$ we call threshold value.

(b) We consider consecutively all nodes $i$; for each pair $(i, j)$ a random number $r \in [0, 1]$ is generated (with uniform distribution); edges $(i, j)$ and $(j, i)$ are created if $(r < p)$.

Algorithm 2.4: This algorithm generates adjacency matrix of a random graph in $NADJ(I, J)$ array. Input to the algorithm are threshold value $F$ and the number of nodes $N$.

Step 1: $I = 1, J = 1$.
Step 2: Generate a random number $r (0 \leq r \leq 1)$.
Step 3: IF $r < F$ then Go To Step 5.
Step 4: $NADJ(I, J) = NADJ(J, I) = 0$ Go To Step 6.
Step 5: $NADJ(I, J) = NADJ(J, I) = 1$.
Step 6: Increment $I$ and $J$.
Step 7: If $I = J = N$ then Stop else Go To 2.

To calculate density we can keep a degree count variable which is incremented by 2 whenever an edge is introduced in the graph. Now division by $N(N-1)$ is trivial.
2.4. Approaches Adopted to Design Gr. Algorithms ; (E-6, N-5, D-1 )

2.4.1. Problem reduction approach :

Most pragmatic approach to solve a large problem is to reduce it to subproblems of manageable sizes, solution of which would provide a solution to the original problem. This is the familiar divide-and-conquer strategy. In general terms, the strategy suggests splitting the n number of inputs into K distinct subsets, 1 < K ≤ n yielding K subproblems. The subproblems can be solved by any method whatsoever. The subsolutions then must be combined into a solution of the whole. If the subproblems are of same type as the original problem then we can reason backward from the problem to be solved, establishing subproblems and subproblems until, finally, the original problem is reduced to a set of trivial primitive problems.

2.4.2. Nondeterministic and parallel algorithm approach :

Informally, we define the state of an algorithm as the combination of the location of the instruction currently being executed and the values of all variables. An algorithm is deterministic if for a single input string there is a single valid 'next state' for any given present state. Such algorithms agree with the way programs are executed on a computer. In a theoretical framework we can remove such restriction of the outcome of every operation. We can allow algorithms to contain operations whose outcome is not uniquely defined but is limited to a specified set of possibilities. The machine executing such operations is allowed to choose any one of these outcomes subject to some testing condition. This leads to the concept of nondeterministic
algorithm. Consider an algorithm that does its calculations until it reaches a point at which a choice must be made among several alternatives. A deterministic algorithm will explore a single alternative and then return to explore the remaining. A nondeterministic algorithm can explore all the alternatives simultaneously, essentially creating a copy of itself for each alternative. Thus a deterministic interpretation of nondeterministic algorithm can be made by allowing unbounded parallelism in computation.

By parallel computation we refer to a situation in which many processor machine is available. A parallel algorithm is an algorithm which is run on a parallel-processor, i.e., a machine which permits more than one processor to function on the same problem at the same time. Some of the algorithms are convertible to be amenable to many processor machine. On the other hand, many problems are essentially sequential and only fit for uniprocessor machine. The complexity of a parallel algorithm is the worst case number of time periods needed for an algorithm to complete. Since at each time period K processors can be computing, the complexity of a parallel algorithm is usually less than that for a uniprocessor machine.

2.4.3. Search methods:

Given a problem modelled as a graph we may have to search all or some of its nodes for its solution. The natural question is how to search the graph systematically to obtain the desired goal under a given cost constrain. Depending upon the algorithm and the computing machine one search method may be advantageous over the other. In general, there are two principal searching
methods for graph searching namely Breadth First Search and Depth First Search which we will describe below. Sometimes a variation like Breadth depth search or K depth search (where K is a given constant) are also adopted. It is worth mentioning here that in the worst case all these search methods require exhaustive search of the graph.

2.4.3.1. Breadth first search:

The Breadth First Search (BFS) method expands nodes in the order in which they are generated. The breadth of a node is the number of nodes adjacent to it. The search process starts with a given node n. n is said to be explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from n are visited next. Since n is explored now all nodes adjacent to n are new unexplored nodes. We keep the nodes in a queue. The first one in the queue is the next node to be explored.

BFS method becomes advantageous with some graph algorithm using problem reduction or partition approach. For a K-processor machine we can safely divide the graph up to K partitions and allow the processors to work in parallel.

2.4.3.2. Depth first search:

A very powerful technique widely used in serial graph algorithms is Depth First Search. In this pattern of search, each time an edge to a new vertex is discovered, the search is continued from the new vertex and is not renewed at the old vertex until all edges from the new vertex are exhausted.
This involves an ordering on the edges which is destroyed if we allow more than one node at a time to be reached or more than one edge from a node to be looked at. The DFS is inherently serial.

2.4.3.3. Backtracking and branch and bound:

Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance.

In backtracking the desired solution must be expressible as an \( n \)-tuple \((x_1, x_2, ..., x_n)\) where \( x_i \) are chosen from some finite set \( S_i \). Often the problem to be solved calls for finding one vector which satisfies a criterion function \( P(x_1, ..., x_n) \). Suppose \( m_1 \) is the size of set \( S_1 \). Then there are \( m = m_1 m_2 ... m_n \) \( n \)-tuples which are possible candidates for satisfying the function \( P \). The brute force approach would be to form all of these \( n \)-tuples and evaluate each one with \( P \), saving those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than \( m \) trials. Its basic idea is to build up the same vector one component at a time and to use modified criterion \( P_1(x_1, x_2, ..., x_i) \) (called bounding function) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realised that the partial vector \((x_1, x_2, ..., x_i)\) can in no way lead to an optimal solution then \( m_{i+1} ... m_n \) possible test vectors may be ignored. Alternatively we can say depth first node generation with bounding functions is called backtracking. Similarly BFS with bounding is called Branch and Bound.
Incidental to the Backtracking method is another method called Sieve. In sieve methods we sieve the nonsolutions until left with only solutions. The classical sieve of Eratosthenese finds primes less than some number. The sieve enumerates the composite (nonprime) numbers, eliminates them and only primes are left over.

2.4.4. Heuristic search and approximation algorithms:

When it is obvious that the problems of our interest cannot be solved in polynomial time, it is practical to solve large instances of the problems in a reasonable amount of time. For example, Tarjan and Trojanowski have obtained an $O(c^{n/3})$ algorithm for max-clique, max-independent set and minimum node cover (E-12). The discovery of a subexponential algorithm for NP problems increases the maximum problem that can be actually solved. However, for large problem instances, even an $O(n^4)$ algorithm requires too much computational effort. Clearly what is required is an algorithm of low polynomial complexity.

The Heuristics is the mainstay for the solution of large problems of polynomial and/or exponential complexity. Heuristics means tending to discover. It is a rule of thumb originated by intuition and global nature of the problem that guides us in general to pull the search towards our goal. A Heuristic search is one that use heuristics to bound, be it DFS or BFS or any other variation. Obviously, a heuristic does not work equally effectively on all problem instances. Exponential time algorithms, even coupled with heuristics will still show exponential behavior on some set of inputs.
For practical purposes we may relax the meaning of solve. Thus by solution of a problem \( p \) we mean a feasible solution with very close to the value of exact solution.

2.4.5. Remark:

The individual personality of the approaches does not necessarily mean that for a given graph algorithm only a single approach will meet our efficiency requirement. In fact, in very rare occasions we do apply a single approach to design an algorithm.

For example, often we partition large graph in subgraphs applying a heuristic that may lead us quickly to the trivial primitive subgraphs. In some algorithms we use both sieve and backtracking to find tighter bound for the solution. In search methods, pure breadth-first or depth first method is seldom used.

The elaborate discussions on these approaches are postponed till we encounter actual design problems in chapters to come where we have discussed individual case studies along with the methodology.

2.5. Classification of Graph Algorithms (E-4)

Graph algorithms are classified according to the order of magnitude of the number of operations expressed as a constant power of the number of vertices \( n \) or edges \( e \). The lower the value of the power the better the algorithm is.
The algorithms can be classified as follows:

1. Problems for which there are algorithms requiring proportional to \(n\) or \(e\) operation.

   Examples: A Spanning tree, Connectedness and component, etc.

2. Problems for which the known algorithms require \(f(n,e)\) operations for some nonlinear but polynomially bounded function \(f\).

   Examples: Minimal spanning tree, Fundamental circuits and cutsets, shortest path between vertex pairs etc.

3. Problems for which no polynomial bound algorithm exist, and nonexistence of such algorithms is not also proved. Heuristics are mainstay of their solutions.

   Examples: Travelling salesperson problem, Clique decision problem, Chromatic number problem etc.

4. Problems that inherently require a number of operations that are exponential in size of the input because they involve the generation of an exponential number of subgraphs.

   Examples: Generating all cliques, Circuits, Cutsets, Trees etc.

2.6. **Analysis of Graph Algorithms**:

   Analysis of an algorithm refers to the process of determining how much computing time and storage an algorithm will require.
Given an algorithm to be analyzed, the first task is to determine which operations are employed and what their relative costs are. In graph algorithms, we take as our data unit a single vertex or an edge. Other data objects like paths, circuits or graphs are composed of these basic data units. The space bounds for any algorithm is given in terms of these data units. Any operation performed on an edge, between two edges, or between two vertices etc. takes a unit of time. Thus testing the presence of an edge between two vertices, adding an edge to a partly constructed tree, finding an edge adjacent to a node requires a unit of time. All other operations performed in the actual execution of an algorithm are assumed to take no time. The time bound of the graph algorithms are generally given in terms of these time units.

The second task is to determine sufficient number of data sets which cause the algorithm to exhibit all possible patterns of behavior. This is one of the important and creative tasks of algorithm analysis. It requires us to understand the working of the algorithm well enough to concoct the data configurations which produce the best or worst or typical behavior. In producing a complete analysis of the computing time of an algorithm we distinguish between two phases: a priori analysis and a posteriori testing. In a priori analysis we obtain a function (of some relevant parameters) which bounds the algorithm's computing time. In a posteriori testing we collect actual statistics about the algorithm's computing time and space while it is executing. This performance profiling of an algorithm tells us how good an algorithm is.

In graph algorithms, performance profiling task is typically difficult. Normally, the number of time units consumed by some of these algorithms depends
intricately on the structure of the graph. Thus if $T_A$ and $T_B$ are upper bounds on the time required by two algorithms $A$ and $B$, respectively and if $T_A < T_B$, it cannot be said with certainty that algorithm $A$ is faster than $B$. The algorithms $A$ and $B$ are often directly compared with each other to make a statement to the effect that $A$ is faster than $B$ on any given graph. In general, the worst-case graphs for two algorithms are not the same.

Thus choice of graphs to test an algorithm or a number of algorithms is interesting. For an approximate algorithm we can design graphs with known result and then show how closely the algorithm generally performs as desired. For algorithms in which we are minimizing the search space we can give a measure of efficiency by comparing with exhaustive search. In these cases we can also take some graphs of known solution space and give a measure of nonsolution testing. Actual run time efficiency, however, requires testing with arbitrary graphs, i.e., graphs generated randomly. Choice of these graphs and their computer generation are pivotal to analysing any graph algorithm.
Computer Programs
and
Sample Results
C ****** MOON MOSER GRAPH
   WRITE(5,3)
   DIMENSION NADJ(5,5)
   DO 10 K=1,5
      N=3*K
   C ******** INITIALIZE WHOLE MATRIX BY ******
   DO 1 I=1,N
      DO 1 J=1,N
         NADJ(I,J)=1
   DO 2 M=1,N+2
      MM=M+2
      DO 2 I=M,MM
         DO 2 J=M,MM
            NADJ(I,J)=0
   DO 20 I=1,N
      WRITE(5,200)N
   DO 20 N = 1, N
      WRITE(5,100)(NADJ(I,J),J=1,N)
   10 CONTINUE
   200 FORMAT( /2DX,'MOON MOSER GRAPH OF V=',I4)
C  ***** K-CHROMATIC GRAPH WITH NO TANGLE *****
DIMENSION NADJ(50,50)
WRITE(5,300)
C  ***** INITIALIZE
NADJ(1,1)=0
NADJ(1,2)=1
NADJ(2,1)=1
NADJ(2,2)=0
C  ***** FORM UPPER HALF *****
DO 5 K=2,5
K2=K-2
N=3*2**K2-1
N1=N-1
M=N1/2
M1=M+1
DO 6 I=1,M
DO 7 J = M1, N1
   L = J - M
7   NA DJ(I, J) = NA DJ(I, L)
   NA DJ(N, I) = 0
6   NA DJ(I, N) = 1
C  ***** FORM LOWER HALF ******
DO 9 I = M1, N1
   DO 8 J = 1, M
      L = I - M
8      NA DJ(I, J) = NA DJ(L, J)
   DO 10 JJ = M1, N1
10     NA DJ(I, JJ) = 0
      NA DJ(N, I) = 1
9      NA DJ(I, N) = 1
      NA DJ(N, N) = 0
   WRITE(5, 200) N
   DO 20 I = 1, N
20    WRITE(5, 100) (NA DJ(I, J), J = 1, N)
5    CONTINUE
100C  CONTINUE
200 FORMAT (/20X,'K- CHROMATIC TRL FREE G OF V='I4)
100 FORMAT (20X,35I2)
300 FORMAT (1I1,://30X,'CRITICAL GRAPHS',/)
STOP
END
MOON MOSER GRAPHS

MOON MOSER GRAPH OF V = 3
0 0 0
0 0 0
0 0 0

MOON MOSER GRAPH OF V = 6
0 0 0 1 1 1
0 0 0 1 1 1
0 0 0 1 1 1
1 1 1 0 0 0
1 1 1 0 0 0
1 1 1 0 0 0

MOON MOSER GRAPH OF V = 9
0 0 0 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1
1 1 1 0 0 0 1 1 1
1 1 1 0 0 0 1 1 1
1 1 1 1 1 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 1 0 0 0

MOON MOSER GRAPH OF V = 12
0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1
1 1 1 1 1 1 0 0 0 1 1 1
1 1 1 1 1 1 0 0 0 1 1 1
1 1 1 1 1 1 0 0 0 1 1 1
1 1 1 1 1 1 0 0 0 1 1 1
1 1 1 1 1 1 0 0 0 1 1 1

MOON MOSER GRAPH OF V = 15
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1
CRITICAL GRAPHS

K- CHROMATIC TRL FREE G OF V = 2
0 1
1 0

K- CHROMATIC TRL FREE G OF V = 5
0 1 0 1 0
1 0 1 0 1
0 1 0 0 1
1 0 0 1 0
0 0 1 1 0

K- CHROMATIC TRL FREE G OF V = 11
0 1 0 1 0 0 1 0 1 0 0
1 0 1 0 0 1 0 1 0 0 0
0 1 0 0 1 1 0 0 1 0 0
0 1 1 0 0 0 1 1 0 0 0
0 1 1 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0 0 0
0 0 1 1 0 0 0 0 3 0 0
0 0 0 1 1 1 1 1 1 1 1

K- CHROMATIC TRL FREE G OF V = 23
0 1 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0
1 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0
0 1 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 0 0
1 1 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 0 0
0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 3 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 3 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

/
RANDOM GRAPHS

THRESHOLD VALUE = 0.59

DENSITY OF G = 0.500 10.000

DENSITY OF G = 0.466 14.000

DENSITY OF G = 0.428 18.000

DENSITY OF G = 0.535 32.000

DENSITY OF G = 0.611 44.000
Fig. 2.1

Fig. 2.2

Fig. 2.3 © M₁

Fig. 2.3 Moon Moser Graphs

© M₂

© M₃
Fig 2.4 Triangle free $K$-critical graphs.