UNIT 2 GRAPH ALGORITHMS

Structure **Page Nos.**

2.0 INTRODUCTION

A number of problems and games like chess, tic-tac-toe etc. can be formulated and solved with the help of graphical notations. The wide variety of problems that can be solved by using graphs range from searching particular information to finding a good or bad move in a game. In this Unit, we discuss a number of problem-solving techniques based on graphic notations, including the ones involving searches of graphs and application of these techniques in solving game and sorting problems.

2.1 OBJECTIVES

After going through this Unit, you should be able to:

- explain and apply various graph search techniques, viz Depth-First Search (DFS), Breadth-First-Search (BFS), Best-First Search, and Minimax Principle;
- discuss relative merits and demerits of these search techniques, and
- apply graph-based problem-solving techniques to solve sorting problems and to games.

2.2 EXAMPLES

To begin with, we discuss the applicability of graphs to a popular game known as NIM.

2.2.1 NIM/Marienbad Game

The game of nim is very simple to play and has an interesting mathematical structure.

Nim is a game for 2 players, in which the players take turns alternately. Initially the players are given a position consisting of several piles, each pile having a finite number of tokens. On each turn, a player chooses one of the piles and then removes at least one token from that pile. The player who picks up the last token wins.

Assuming your opponent plays optimally, there may be some positions/situations in which the player having the current move cannot win. Such positions are called

"**losing** positions" (for the player whose turn it is to move next). The positions which are not losing ones are called **winning**.

Marienbad is a variant of a **nim** game and it is played with matches. The rules of this game are similar to the nim and are given below:

(1) It is a two-player game

(2) It starts with n matches (n must be greater or equal to 2 i.e., n >= 2)

(3) The winner of the game is one who takes the last match, whosoever is left with no sticks, loses the game.

(4) On the very first turn, up to n –1 matches can be taken by the player having the very first move.

(5) On the subsequent turns, one must remove at least one match and at most twice the number of matches picked up by the opponent in the last move.

Before going into detailed discussion through an example, let us explain what may be the possible states which may indicate different stages in the game. At any stage, the following two numbers are significant:

(i) The total number of match sticks available, after picking up by the players so far.
(ii) The number of match sticks that the player having the move can pick up.

We call the ordered pair a **state** in the game, where

i : the number of sticks available
j : the number of sticks that can be picked, by the player having the move, according to the rules.

For example:

(i) Initially if n is the number of sticks, then the state is (n, n−1), because the players must leave at least one stick.

(ii) While in the state (i, j), if the player having the move picks up k sticks then the state after this move, is (i − k, min (2k, i −k)),
which means

(a) the total number of available sticks is (i − k)

(b) the player, next to pick up, can not pick up more than the double of the number of sticks picked up in the previous move by the opponent and also clearly the player can not pick up more than the number of sticks available, i.e., (i −k)

(iii) We can not have the choice of picking up zero match sticks, unless no match stick is available for picking. Therefore the state (i, 0) implies the state (0,0),

After discussing some of possible states, we elaborate the game described above through the following example.

Example 2.2.1:

Let the initial number of matches be 6, and let player A take the chance to move first. What should be A's strategy to win, for his first move. Generally, A will consider all possible moves and choose the best one as follow:

- if A takes 5 matches, that leaves just one for B, then B will take it and win the game;
- if A takes 4 matches, that leaves 2 for B, then B will take it and win;
- if A takes 3 matches, that leaves 3 for B, then B will take it and win;

- if A takes 2 match, that leaves 4 for B, then B will take it and win;
- if A takes 1 match, that leaves 5 for B. In the next step, B can take 1 or 2(*recall that B* can *take at most the twice of the number what A just took*) and B will go to either of the states (4,2) or (3,3) both of which are winning moves for A, B's move will lead to because A can take all the available stick and hence there will be not more sticks for B to pick up. Taking a look at this reasoning process, it is for sure that the best move for A is just taking one match stick.

The above process can be expressed by a directed graph, where each node corresponds to a *position* (*state*) and each edge corresponds a *move* between two positions, with each node expressed by a pair of numbers $< i,j >$, $0 <= j <= i$, and
i: the number of the matches left;
j : the upper limit of number of matches which can be removed in the next move, that is, any number of matches between 1 and *j* can be taken in the next move.

As mentioned earlier, we have
The initial node: $< n,n-1 >$.
Edges leaving the node $< i,j >$ can lead to the node $< i-k, \min(2k, j-k) >$,
with $0 < k \leq i$.

In the directed graph shown below, *rectangular* nodes denote *losing* nodes and *oval* nodes denote *winning* nodes:

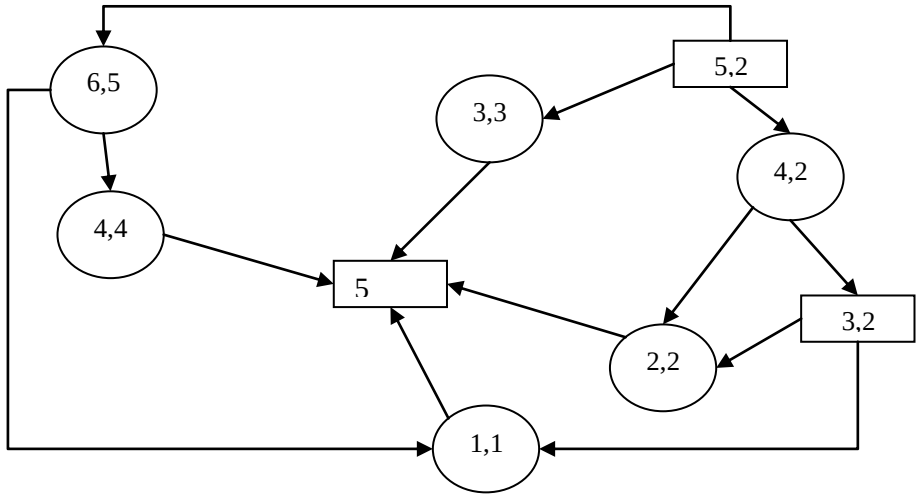


Figure 1

- a terminal node $< 0, 0 >$, from which there is no legal move. It is a *losing* position.
- a nonterminal node is a winning node (denoted by a circle), if **at least one** of its successors is a losing node, because the player currently having the move is can leave his opponent in losing position.
- a nonterminal node is a losing node (denoted by a square) if **all** of its successors are wining nodes. Again, because the player currently having the move cannot **avoid** leaving his opponent in one of these winning positions.

How to determine the wining nodes and losing nodes in a directed graph?
Intuitively, we can starting at the losing node $< 0, 0 >$, and work back according to the definition of the winning node and losing node. A node is a **losing node,** for the current player, if the move takes to a state such that the opponent can make at least one move which forces the current player to lose. On the other hand, a node is a **winning node,** if after making the move, current player will leave the opponent in a state, from which opponent can not win. For instance, in any of nodes $< 1, 1 >$,

< 2, 2 >, < 3, 3 > and < 4, 4 >, a player can make a move and leave his opponent to be in the position < 0, 0 >, thus these 4 nodes are wining nodes. From position < 3, 2 >, two moves are possible but both these moves take the opponent to a winning position so it is a losing node. The initial position < 6, 5 > has one move which takes the opponent to a losing position so it is a winning node. Keeping the process of going in the backward direction, we can mark the types of nodes in a graph. A recursive C program for the purpose, can be implemented as follows:

2.2.2 Function for Computing Winning Nodes

function *recwin(i, j)*
{ Return *true* if and only if node < *i, j* > is winning,
we assume 0 <= *j*< = *i* }
for *k* = 1 to *j* do
{ if not *recwin*(*i* - *k*, min(2*k*, *i* − *k*))
then return *true*
}
return *false*

Ex.1) Draw a directed graph for a game of Marienbad when the number of match sticks, initially, is 5.

2.3 TRAVERSING TREES

Traversing a tree means exploring all the nodes in the tree, starting wsith the root and exploring the nodes in some order. We are already aware that in the case of binary trees, three well-known tree-traversal techniques used are preorder, postorder and inorder. In **preorder** traversal, we first visit the node, then all the nodes in its left subtree and then all nodes in the right subtree. In **postorder** traversal, we first visit the left subtree, then all the nodes in the right subtree and the root is traversed in the last. In i**norder** traversal, the order of traversal is to first visit all the nodes in the leftsubtree, then to visit the node and then all the nodes in its right subtree. Postorder and preorder can be generalized to nonbinary trees. These three techniques explore the nodes in the tree from left to right.

Preconditioning

Consider a scenario in which problem might have many similar situations or instances, which are required to be solved. In such a situation, it it might be useful to spend some time and energy in calculating the auxiliary solutions (i.e., attaching some extra information to the problem space) that can be used afterwards to fasten the process of finding the solution of each of these situations. This is known as **preconditioning**. Although some time has to be spent in calculating / finding the auxiliary solutions yet it has been seen that in the final tradeoff, the benefit achieved in terms of speeding up of the process of finding the solution of the problem will be much more than the additional cost incurred in finding auxiliary/additional information.

In other words, let x be the time taken to solve the problem without preconditioning, y be the time taken to solve the problem with the help of some auxiliary results (i.e., after preconditioning) and let t be the time taken in preconditioning the problem space i.e., time taken in calculating the additional/auxiliary information. **Then to solve n typical instances, provided that y < x , preconditioning will be beneficial only when** ,

$$nx > t + ny$$
$$\text{i.e., } nx - ny > t$$
$$\text{or } n > t / (x - y)$$

Preconditioning is also useful when only a few instances of a problem need to be solved. Suppose we need a solution to a particular instance of a problem, and we need it in quick time. One way is to solve all the relevant instances in advance and store their solutions so that they can be provided quickly whenever needed. But this is a very inefficient and impractical approach,– i.e., to find solutions of all instances when solution of only one is needed. On the other hand, a popular alternative could be to calculate and attach some additional information to the problem space which will be useful to speedup the process of finding the solution of any given instance that is encountered.

For an **example**, let us consider the problem of finding the ancestor of any given node in a rooted tree (which may be a binary or a general tree).

In any rooted tree, node u will be an **ancestor** of node v, if node u lies on the path from root to v. Also we must note that every node is an ancestor of itself and root is an ancestor of all nodes in the tree including itself. Let us suppose, we are given a pair of nodes (u,v) and we are to find whether u is an ancestor or v or not. If the tree contains n nodes, then any given instance can take $\Omega(n)$ time in the worst case. But, if we attach some relevant information to each of the nodes of the tree, then after spending $\Omega(n)$ time in preconditioning, we can find the ancestor of any given node in constant time.

Now to precondition the tree, we first traverse the tree in preorder and calculate the precedence of each node in this order, similarly, we traverse the tree in postorder and calculate the precedence of each node. For a node u, let **precedepre[u]** be its precedence in preorder and let **precedepost[u]** be its precedence in postorder.

Let u and v be the two given nodes. Then according to the rules of preorder and postorder traversal, we can see that :

In ***preorder traversal,*** as the root is visited first before the left subtree and the right subtree, so,

If precedepre[u] <= precedepre[v], then
u is an ancestor of v or u is to the left of v in the tree.

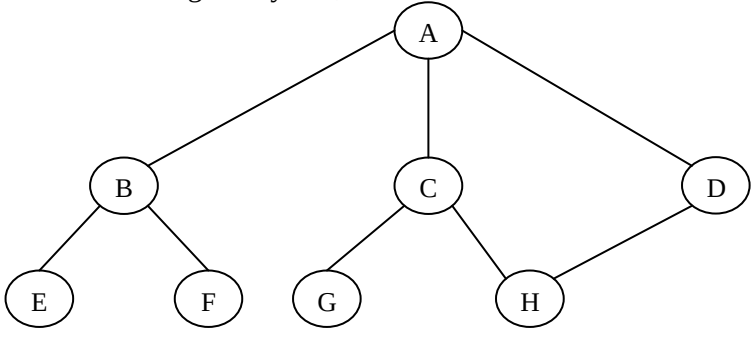
In ***postorder traversal,*** as the root is visited last, because, first we visit leftsubtree, then right subtree and in the last we visit root so,

If precedepost[u] >= precedepost[v], then
u is an ancestor of v or u is to the right of v in the tree.

So for u to be an ancestor of v, both the following conditions have to be satisfied:
precedepre[u] <= precede[v] and precedepost[u] >= precedepost[v].

Thus, we can see that after spending some time in calculating preorder and postorder precedence of each node in the tree, the ancestor of any node can be found in constant time.

Ex. 2) For the following binary tree, in which A is the root:



2.4 DEPTH-FIRST SEARCH

The depth-first search is a search strategy in which the examination of a given vertex u, is delayed when a new vertex say v is reached and examination of v is delayed when new vertex say w is reached and so on. When a leaf is reached (i.e., a node which does not have a successor node), the examination of the leaf is carried out. And then the immediate ancestor of the leaf is examined. The process of examination is carried out in reverse order of reaching the nodes.

In depth first search for any given vertex u, we find or explore or discover the first adjacent vertex v (in its adjacency list), not already discovered. Then, in stead of exploring other nodes adjacent to u, the search starts from vertex v which finds its first adjacent vertex not already known or discovered. The whole process is repeat for each newly discovered node. When a vertex adjacent to v is explored down to the leaf, we back track to explore the remaining adjacent vertices of v. So we search farther or deeper in the graph whenever possible. This process continues until we discover all the vertices reachable from the given source vertex. If still any undiscovered vertices remain then a next source is selected and the same search process is repeated. This whole process goes on until all the vertices of the graph are discovered.

The vertices have three adjacent different statuses during the process of traversal or searching, the status being: *unknown, discovered and visited*. Initially all the vertices have their status termed as '*unknown*', after being explored the status of the vertex is changed to '*discovered*' and after all vertices adjacent to a given vertex are discovered its status is termed as '*visited*'. This technique ensures that in the depth first forest, at a time each vertex belong to only one depth-first tree so these trees are disjoint.

Because we leave partially visited vertices and move ahead, to backtrack later, stack will be required as the underlying data structure to hold vertices. In the recursive version of the algorithm given below, the stack will be implemented implicitly, however, if we write a non-recursive version of the algorithm, the stack operation have to be specified explicitly.

In the algorithm, we assume that the graph is represented using adjacency list representation. To store the parent or predecessor of a vertex in the depth-first search, we use an array parent[]. Status of a 'vertex' i.e., unknown, discovered, or visited is stored in the array status. The variable *time* is taken as a global variable. V is the vertex set of the graph G.

In depth-first search algorithm, we also timestamp each vertex. So the vertex u has two times associated with it, the discovering time *d[u]* and the termination time *t[u]*. The *discovery time* corresponds to the status change of a vertex from unknown to discovered, and *termination time* corresponds to status change from discovered to visited. For the initial input graph when all vertices are unknown, time is initialized to 0. When we start from the source vertex, time is taken as 1 and with each new discovery or termination of a vertex, the time is incremented by 1. Although DFS algorithm can be written without time stamping the vertices, time stamping of vertices helps us in a better understanding of this algorithm. However, one drawback of time stamping is that the storage requirement increases.

Also in the algorithm we can see that for any given node u, its discovering time will be less than its termination time i.e., d[u] < t[u].

The algorithm is:

Program

DFS(G)
//This fragment of algorithm performs initializing
//and starts the depth first search process

```
1 for all vertices u ∈ V
2   {     status[u] = unknown;
3         parent[u] = NULL;

4          time = 0   }
5 for each vertex u ∈ V
6         {if status[u] == unknown
7                   VISIT(u)
```

```
VISIT(U)
       1 status[u] = discovered;
2 time = time + 1;
3 d[u] =time;}
4 for each Vertex v ∈  V adjacent to u
5         {if status[v] == unknown
6                   parent[v] = u;
7                   VISIT(V);

       8 time = time + 1;
       9 t[u] = time;
      10 status[u] = visited;}
```

In the procedure DFS, the first *for-loop* initializes the status of each vertex to unknown and parent or predecessor vertex to NULL. Then it creates a global variable *time* and initializes it to 0. In the second *for-loop* belonging to this procedure, for each node in the graph if that node is still unknown, the VISIT(u) procedure is called. Now we can see that every time the VISIT (u) procedure will be called, the vertex u it will become the root of a new tree in the forest of depth first search.

Whenever the procedure VISIT(u) will be called with parameter u, the vertex u will be unknown. So in the procedure VISIT(u), first the status of vertex u is changed to '*discovered*', time is incremented by 1 and it is stored as discovery time of vertex u in d[u].

When the VISIT procedure will be called for the first time, d[u] will be 1. In the *for-loop* for each given vertex u, every unknown vertex adjacent to u is visited recursively and the parent[] array is updated. When the for-loop concludes, i.e., when every vertex adjacent to u is discovered, the time is increment by 1 and is stored as the termination time of u i.e. t[u] and the status of vertex u is changed to '*visited*'.

Analysis of Depth-First Search

In procedure DFS(), each for loop takes time O($|V|$), where $|V|$ is the number of vertices in V. The procedure VISIT is called once for every vertex of the graph. In the procedure visit for each of the *for-loop* is executed equal to the number of edges emerging from that node and yet not traversed. Considering the adjacency list of all nodes to total number of edges traversed are O($|E|$), where $|E|$, is the number of edges in E. So the running time of DFS is, therefore, O ($|V|+|E|$).

Example 2.4.1:

For the graph given in Figure 2.4.1.1. Use DFS to visit various vertices. The vertex D is taken as the starting vertex and, if there are more than one vertices adjacent to a vertex, then the adjacent vertices are visited in lexicographic order.

In the following,

(i) the label i/ indicates that the corresponding vertex is the ith discovered vertex.

(ii) the label i/j indicates that the corresponding vertex is the ith discovered vertex and jth in the combined sequence of discovered and visited.

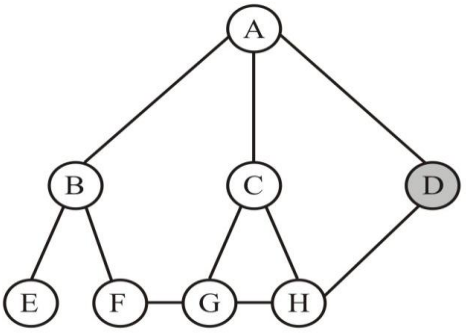


Figure 2.4.1.1: Status of D changes to discovered, d[D] = 1

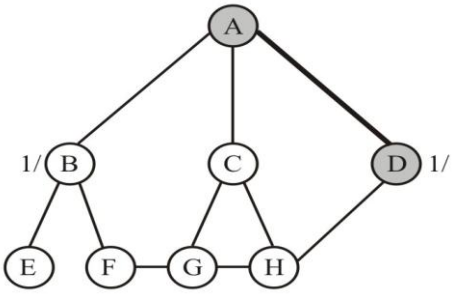


Figure 2.4.1.2: D has two neighbors by convention A is visited first i.e., the status of A changes to discovered, d[A] = 2

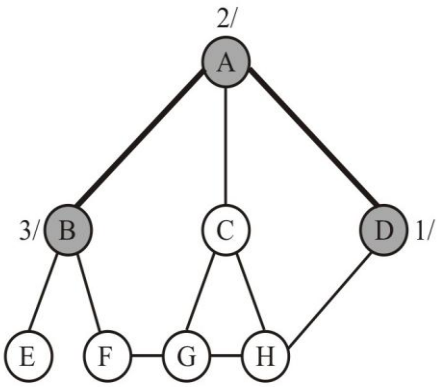


Figure 2.4.1.3: A has two unknown neighbors B and C, so status of B changes to 'discovered', i.e., d[B] = 3

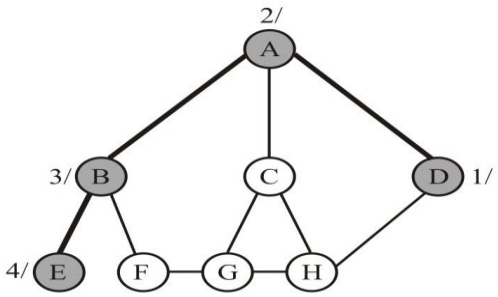


Figure 2.4.1.4: Similarly vertex E is discovered and d[E] = 4

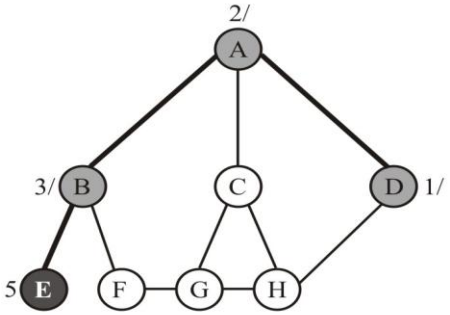


Figure 2.4.1.5: All of E's neighbors are discovered so status of vertex E is changed to 'visited' and t[E] = 5

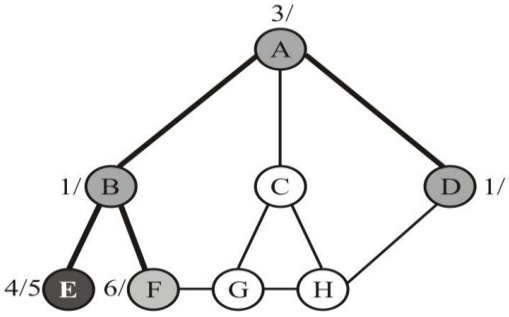


Figure 2.4.1.6: The nearest unknown neighbor ok B is F, so we change status of F to 'discovered', d[F] = 6

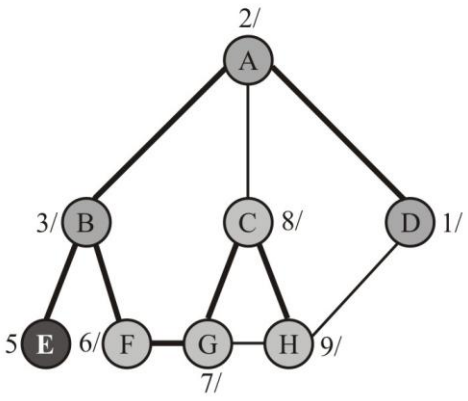


Figure 2.4.1.7: Similarly vertices G, E and H are discovered respectively with d[G] = 7, d[C] = 8 and d[H] = 9

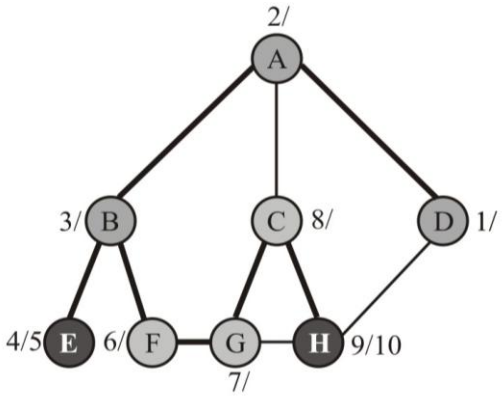


Figure 2.4.1.8: Now as all the neighbors of H are already discovered we backtrack, to C and stores its termination time as t[H] = 10

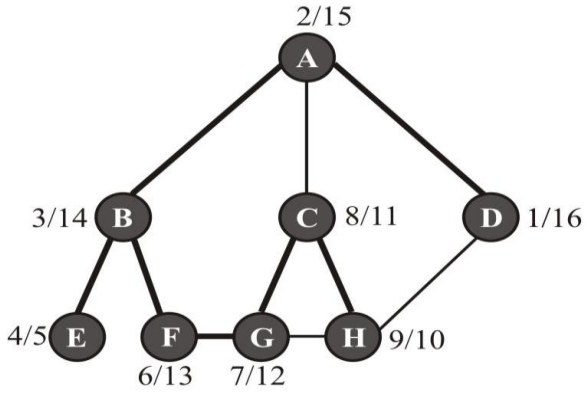


Figure 2.4.1.9: We find the termination time of remaining nodes in reverse order, backtracking along the original path ending with D.

The resultant parent pointer tree has its root at D, since this is the first node visited. Each new node visited becomes the child of the most recently visited node. Also we can see that while D is the first node to be '*discovered*', it is the last node *terminated.* This is due to recursion because each of D's neighbors must be discovered and terminated before D can be terminated. Also, all the edges of the graph, which are not used in the traversal, are between a node and its ancestor. This property of depth-first search differentiates it from breadth-first search tree.

Also we can see that the maximum termination time for any vertex is 16, which is twice the number of vertices in the graph because time is incremented only when a vertex is discovered or terminated and each vertex is discovered once and terminated once.

Properties of Depth-first search Algorithm

(1) **Parenthesis Structure**

In a graph G, if u and v are two vertices such that u is discovered before v then the following cases may occur:

(a) If v is discovered before u is terminated, then v will be finished before u (i.e., the vertex which is being discovered later will be terminated first). This property exists because of the recursion stack, as the vertex v which is discovered after u will be pushed on the stack at a time when u is already on the stack, so v will be popped out of the stack first i.e., v will be terminated first.

Also the interval [d[v],t[v]] is contained in the interval [d[u],t[u]], we Say the v is the proper descendant of u.

(b) If u is terminated before v is discovered then in this case t[u] < d[v] so the two intervals are disjoint.

In this case t[u] < t[v] so the two intervals are disjoint.

Note: We should remember that in depth-first search the third case of overlapping intervals is not possible i.e., situation given below is not, possible because of recursion.

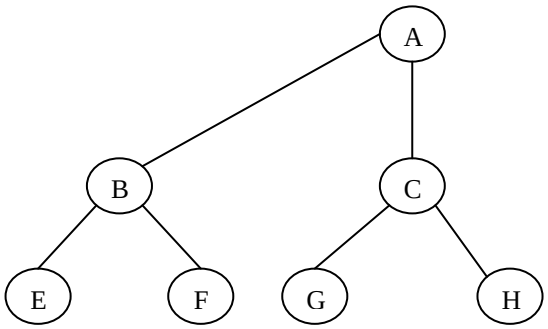
(2) Another important property of depth-first search (sometimes called white path property) is that v is a descendant of u if and only if at the time of discovery of u, there is at least one path from u to v contains only unknown vertices (i.e., white vertices or vertices not yet found or discovered).

(3) *Depth-First Search can be used to find connected components in a given graph:* One useful aspect of depth first search algorithm is that it traverses connected component one at a time and then it can be used to identify the connected components in a given graph.

(4) *Depth-first search can also be used to find cycles in an undirected graph:* we know that an undirected graph has a cycle if and only if at some particular point during the traversal, when u is already discovered, one of the neighbors v of u is also already discovered and is not parent or predecessor of u.

We can prove this property by the argument that if we discover v and find that u is already discovered but u is not parent of v then u must be an ancestor of v and since we traveled u to v via a different route, there is a cycle in the graph.

Ex.3) Trace how DFS traverses (i.e., discover and visits) the graph given below when starting node/vertex is B.



Depth First Search in *Directed* Graphs

The earlier discussion of the Depth-First search was with respect to undirected graphs. Next, we discuss Depth-First strategy with respect to Directed Graph. In a directed graph the relation of '*binary adjacent to*' is not symmetric, where the relation of '*being adjacent to*' is symmetric for undirected graphs.

To perform depth first search in *directed* graphs, the algorithm given above can be used with minor modifications. The main difference exists in the interpretation of an "adjacent vertex". In a directed graph vertex v is adjacent to vertex u if there is a directed edge from u to v. If a directed edge exists from u to v but not from v to u, then v is adjacent to u but u is not adjacent to v.

Because of this change, the algorithm behaves differently. Some of the previously given properties may no longer be necessarily applicable in this new situation.

Edge Classification

Another interesting property of depth first search is that search can be used to classify different type of edges of the directed graph G(V,E). This edge classification gives us some more information about the graph.

The different edges are:

(a) **Tree Edge:** An edge to a still 'unknown' vertex i.e., edge (u,v) is a *tree edge* if it is used to discover v for the first time.

(b) **Back edge:** An edge to an already 'discovered' or ancestor vertex i.e., edge (u,v) is a back edge if it connects to a vertex v which is already discovered which means v is an ancestor of u.

(c) **Forward edge:** An edge to an already 'visited' descendant (not possible in undirected graph) i.e., edge (u, v) is a forward edge if v is a descendant of u in the depth first tree. Also, we can see that d[u] < d[v].

(d) **Cross edge:** An edge to an already 'visited' neighbor, which is not a descendant. As long as one vertex is not descendant of the other, cross edge can go between vertices in the same depth first tree or between vertices in different depth first trees.

Note: In an undirected graph, every edge is either a tree edge or back edge, i.e., forward edges or cross edges are not possible.

Example 2.4.2:

In the following directed graph, we consider the adjacent nodes in the increasing alphabetic order and let starting vertex be.

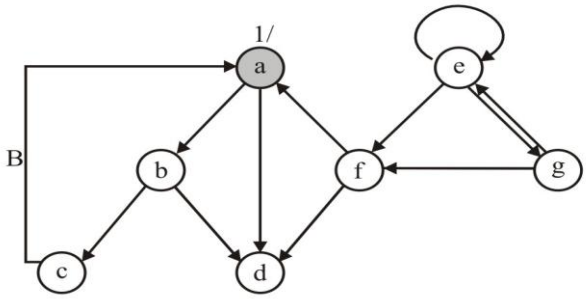


Figure 2.4.2.1: Status of a changed to discovered, d[a] = 1

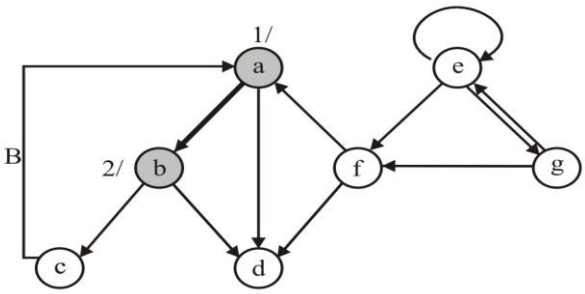


Figure 2.4.2.2: a has unknown two neighbors a and d, by convention b is visited first, i.e the status of b changes to discovered, d[a] =2

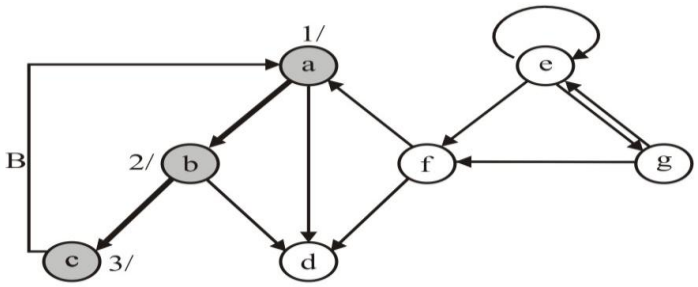


Figure 2.4.2.3: b has two unknown neighbors c and d, by convention c is discovered first i.e., d[c] = 3

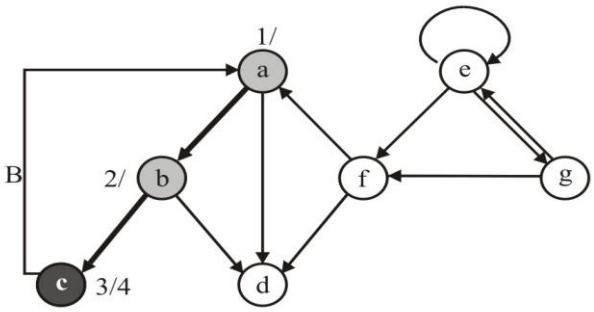


Figure 2.4.2.4: c has only a single neighbor a which is already discovered so c is terminated i.e., t[c] = 5

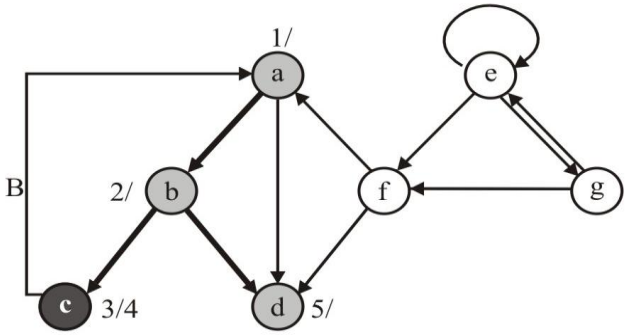


Figure 2.4.2.5: The algorithm backtracks recursively to b, the next unknown neighbor is d, whose status is change to discovered i.e., d[d] = 5

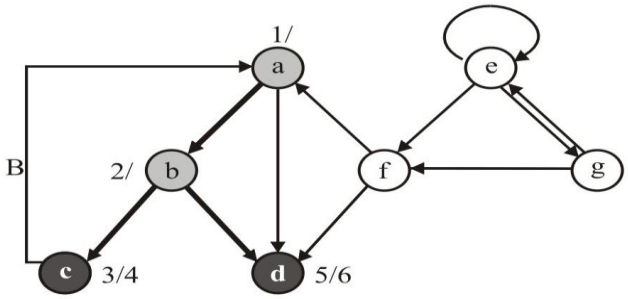


Figure 2.4.2.6: d has no neighbor, so d terminates, t[d] = 6

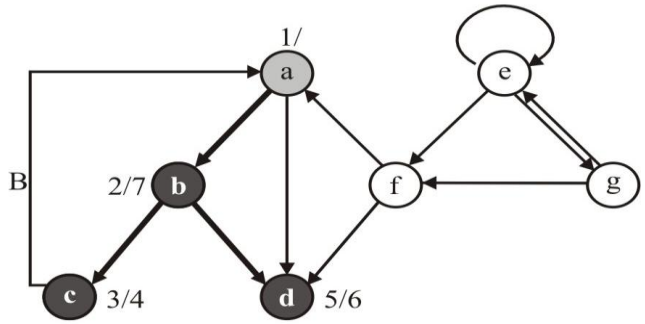


Figure 2.4.2.7: The algorithm backtracks recursively to b, which has no unknown neighbors, so b(terminated) is visited i.e., t[b] = 7

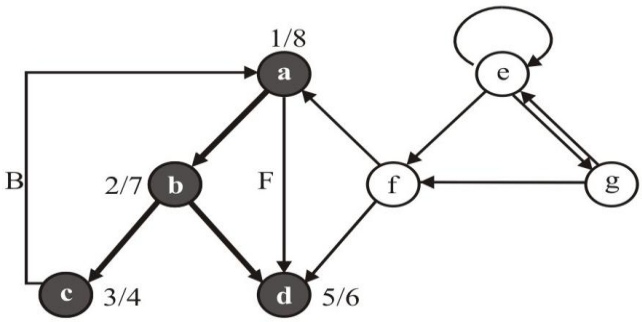


Figure 2.4.2. 8: The algorithm backtracks to a which has no unknown neighbors so a is visited i.e., t[a] = 8.

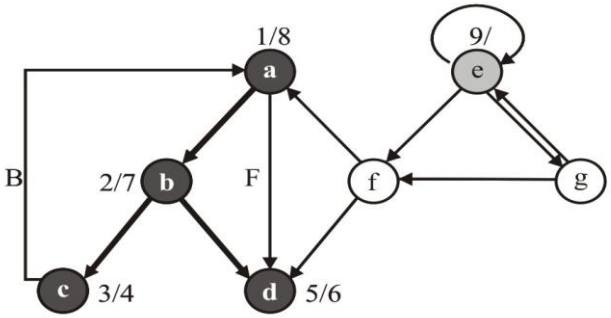


Figure 2.4.2. 9: The connected component is visited so the algorithm moves to next component starting from e (because we are moving in increasing alphabetic order) so e is 'discovered' i.e. , d[e] = 9

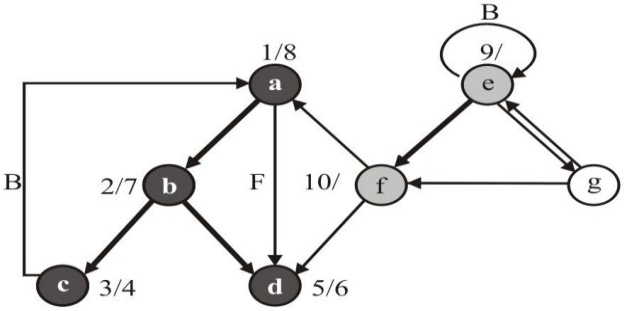


Figure 2.4.2. 10: e has two unknown neighbors f and g, by convention we discover f i.e., d[f] = 10

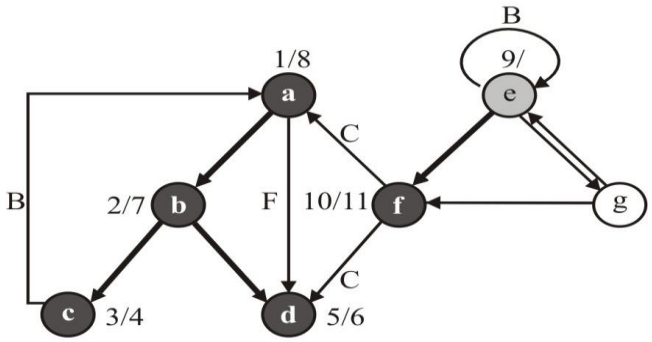


Figure 2.4.2.11: f has no unknown neighbors so f (terminates) is 'visited' i.e., t[f] = 11

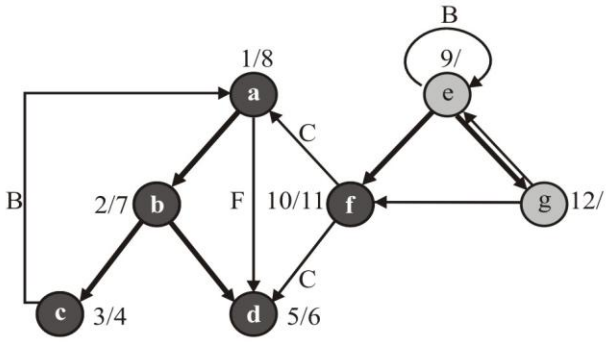


Figure 2.4.2. 12: The algorithm backtracks to e, which has g as the next 'unknown' neighbor, g is 'discovered' i.e., d[g] = 12

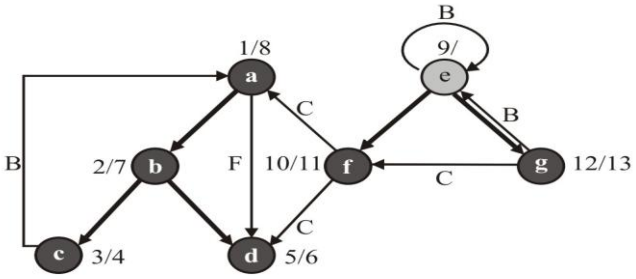


Figure 2.4.2. 13: The only neighbor of g is e, which is already discovered, so g(terminates) is 'visited' i.e., t[g] = 12

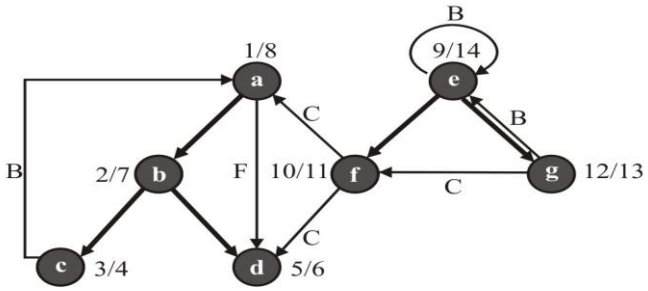


Figure 2.4.2. 14: The algorithm backtracks to e, which has no unknown neighbors left so e (terminates) is visit i.e., t[e] = 14

Some more properties of Depth first search (in directed graph)

(1) Given a directed graph, depth first search can be used to determine whether it contains cycle.

(2) Cross-edges go from a vertex of higher discovery time to a vertex of lower discovery time. Also a forward edge goes from a vertex of lower discovery time to a vertex of higher discovery time.

(3) Tree edges, forward edges and cross edges all go from a vertex of higher termination time to a vertex of lower finish time whereas back edges go from a vertex of lower termination time to a vertex of higher termination time.

(4) A graph is acyclic if and only if f any depth first search forest of graph G yields no back edges. This fact can be realized from property 3 explained above, that if there are no back edges then all edges will go from a vertex of higher termination time to a vertex of lower termination time. So there will be no cycles. So the property which checks cycles in a directed graph can be verified by ensuring there are no back edges.

2.5 BREADTH-FIRST SEARCH

Breadth first search as the name suggests first discovers all vertices adjacent to a given vertex before moving to the vertices far ahead in the search graph. If G(V,E) is a graph having vertex set V and edge set E and a particular source vertex s, breadth first search find or discovers every vertex that is reachable from s. First it discovers every vertex adjacent to s, then systematically for each of those vertices it finds the all the vertices adjacent to them and so on. In doing so, it computes the distance and the shortest path in terms of fewest numbers of edges from the source node s to each of the reachable vertex. Breadth-first Search also produces a breadth-first tree with root vertex s in the process of searching or traversing the graph.

For recording the status of each vertex, whether it is still unknown, whether it has been discovered (or found) and whether all of its adjacent vertices have also been discovered. The vertices are termed as **unknown**, **discovered** and **visited** respectively. So if $(u,v) \in E$ and u is visited then v will be either discovered or visited i.e., either v has just been discovered or vertices adjacent to v have also been found or visited.

As breadth first search forms a breadth first tree, so if in the edge (u,v) vertex v is discovered in adjacency list of an already discovered vertex u then we say that u is the **parent or predecessor** vertex of V. Each vertex is discovered once only.

The data structure we use in this algorithm is a queue to hold vertices. In this algorithm we assume that the graph is represented using adjacency list representation. *front[u]* is used to represent the element at the front of the queue. *Empty()* procedure returns true if queue is empty otherwise it returns false. Queue is represented as Q. Procedure *enqueue()* and *dequeue()* are used to insert and delete an element from the queue respectively. The data structure Status[]is used to store the status of each vertex as unknown or discovered or visited.

2.5.1 Algorithm of Breadth First Search

```
1 for each vertex u ∈ V – {s}
2         status[u] = unknown
3 status[s]=discovered
4 enqueue(Q,s)
5 while(empty[Q] != false)
6         u = front[Q]
7         for each vertex v ∈ Adjacent to u
8                 if status[v] = unknown
```

```
9                       status[v] = discovered
10                      parent (v) = u
11                     end for
12                      enqueue(Q,v);
13      dequeue(Q)
14      status[u] = visited
15      print "u is visited"
16      end while
```

The algorithm works as follows. Lines 1-2 initialize each vertex to 'unknown'. Because we have to start searching from vertex s, line 3 gives the status 'discovered' to vertex s. Line 4 inserts the initial vertex s in the queue. The while loop contains statements from line 5 to end of the algorithm. The while loop runs as long as there remains 'discovered' vertices in the queue. And we can see that queue will only contain 'discovered' vertices. Line 6 takes an element u at the front of the queue and in lines 7 to 10 12 the adjacency list of vertex u is traversed and each unknown vertex v in the adjacency list of u, its status is marked as discovered, its parent is marked as u and then it is inserted in the queue. In the line 13, vertex u is removed from the queue. In line 14-15, when there are no more elements in adjacency list of u, vertex u is removed from the queue its status is changed to 'visited' and is also printed as visited.

The algorithm given above can also be improved by storing the distance of each vertex u from the source vertex s using an array distance[] and also by permanently recording the predecessor or parent of each discovered vertex in the array parent[]. In fact, the distance of each reachable vertex from the source vertex as calculated by the BFS is the shortest distance in terms of the number of edges traversed. So next we present the modified algorithm for breadth first search.

2.5.2 Modified Algorithm

Program BFS(G,s)

```
1 for each vertex u ∈ s v – {s}
2       status[u] = unknown
3       parent[u] = NULL
4       distance[u] = infinity
5 status[s] = discovered
6 distance[s] = 0
7 parent[s] = NULL
8 enqueue(Q,s)
9 while empty(Q) != false
10      u = front[Q]
11      for each vertex v adjacent to u
12              if status[v] = unknown
13                      status[v] = discovered
14                      parent[v] = u
15                      distance[v]  = distance[u] + 1
16                      enqueue(Q,v)
17      dequeue(Q)
18      status[u] = visited
19      print "u is visited"
```

In the above algorithm the newly inserted line 3 initializes the parent of each vertex to NULL, line 4 initializes the distance of each vertex from the source vertex to infinity, line 6 initializes the distance of source vertex s to 0, line 7 initializes the parent of source vertex s NULL, line 14 records the parent of v as u, line 15 calculates the shortest distance of v from the source vertex s, as distance of u plus 1.

Example 2.5.3:

In the figure given below, we can see the graph given initially, in which only source s is discovered.

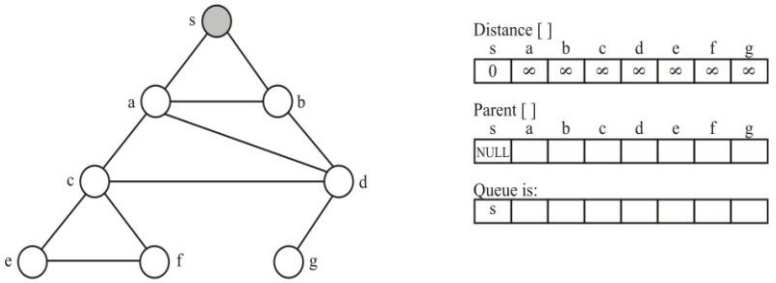


Figure 2.5.1.1: Initial Input Graph

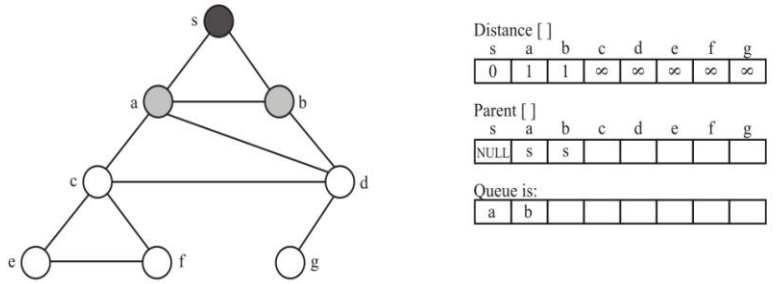


Figure 2.5.1.2: After we visit s

We take unknown (i.e., undiscovered) adjacent vertex of s and insert them in queue, first a and then b. The values of the data structures are modified as given below:

Next, after completing the visit of a we get the figure and the data structures as given below:

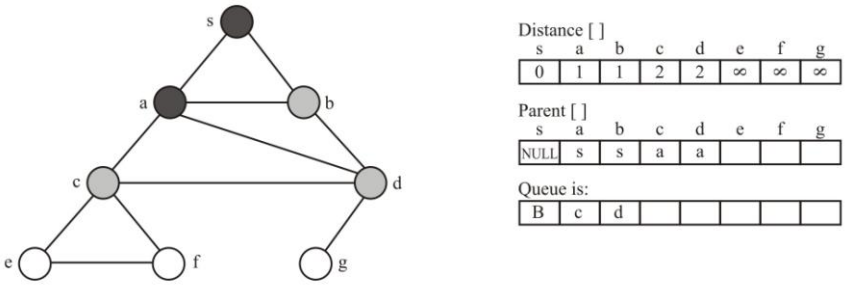


Figure 2.5.1.3: After we visit a

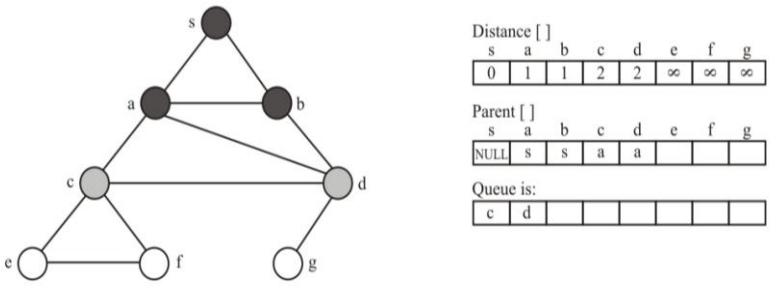


Figure 2.5.1.4: After we visit b

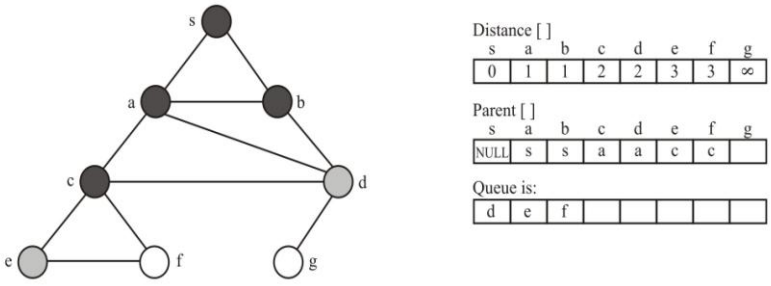


Figure 2.5.1.5: After we visit c

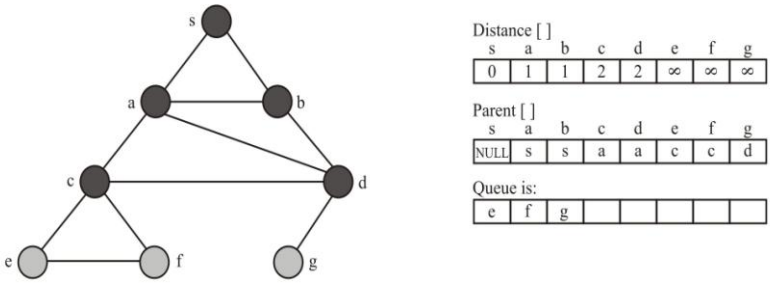


Figure 2.5.1.6: After we visit d

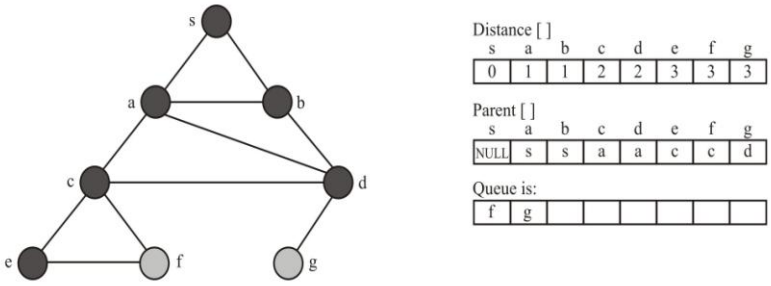


Figure 2.5.1.7: After we visit e

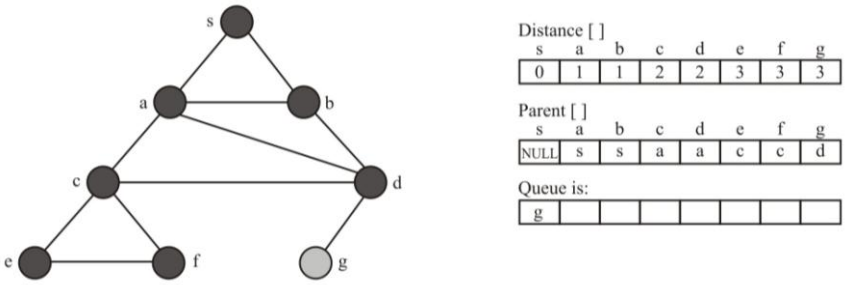


Figure 2.5.1.8: After we visit f

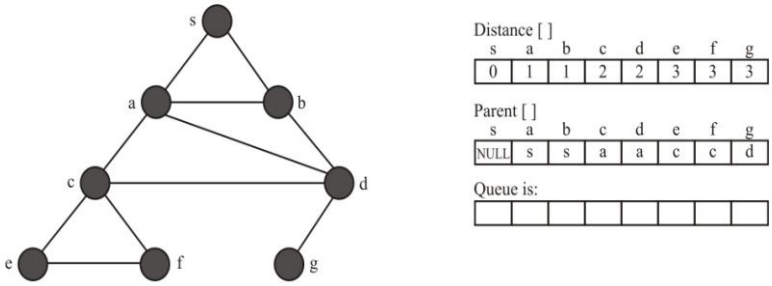


Figure 2.5.1.9: After we visit g

Figure 1: Initial Input Graph

Figure 2: We take unknown (i.e., undiscovered) adjacent vertices of s and insert them in the queue.

Figure 3: Now the gray vertices in the adjacency list of u are b, c and d, and we can visit any of them depending upon which vertex is inserted in the queue first. As in this example, we have inserted b first which is now at the front of the queue, so next we will visit b.

Figure 4: As there is no undiscovered vertex adjacent to b so no new vertex will be inserted in the queue, only the vertex b will be removed from the queue.

Figure 5: Vertices e and f are discovered as adjacent vertices of c, so they are inserted in the queue and then c is removed from the queue and is visited.

Figure 6: Vertex g is discovered as the adjacent vertex of d an after that d is removed from the queue and its status is changed to visited.

Figure 7: No undiscovered vertex adjacent to e is found so e is removed from the queue and its status is changed to visited.

Figure 8: No undiscovered vertex adjacent to f is found so f is removed from the queue and its status is changed to visited.

Figure 9: No undiscovered vertex adjacent to g is found so g is removed from the queue and its status is changed to visited. Now as queue becomes empty so the while loop stops.

2.6 BEST FIRST SEARCH & MINIMAX PRINCIPLE

Best First Search

In the two basic search algorithms we have studied before i.e., depth first search and breadth first search we proceed in a systematic way by discovering/finding/exploring nodes in a predetermined order. In these algorithms at each step during the process of searching there is no assessment of which way to go because the method of moving is fixed at the outset.

The best first search belongs to a branch of search algorithms known as **heuristic search algorithms**. The basic idea of heuristic search is that, rather than trying all possible search paths at each step, we try to find which paths seem to be getting us nearer to our goal state. Of course, we can't be sure that we are really near to our goal state. It could be that we are really near to our goal state. It could be that we have to take some really complicated and circuitous sequence of steps to get there. But we might be able to make a good guess. Heuristics are used to help us make that guess.

To use any heuristic search we need an evaluation function that scores a node in the search tree according to how close to the goal or target node it seems to be. It will just be an estimate but it should still be useful. But the estimate should always be on the lower side to find the optimal or the lowest cost path. For example, to find the optimal path/route between Delhi and Jaipur, an estimate could be straight arial distance between the two cities.

There are a whole batch of heuristic search algorithms e.g., Hill Climbing, best first search, A* ,AO* etc. But here we will be focussing on best first search.

Best First Search combines the benefits of both depth first and breadth first search by moving along a single path at a time but change paths whenever some other path looks more promising than the current path.

At each step in the depth first search, we first generate the successors of the current node and then apply a heuristic function to find the most promising child/successor. We then expand/visit (i.e., find its successors) the chosen successor i.e., find its unknown successors. If one of the successors is a goal node we stop. If not then all these nodes are added to the list of nodes generated or discovered so fart. During this process of generating successors a bit of depth search is performed but ultimately if the solution i.e., goal node is not found then at some point the newly found/discovered/generated node will have a less promising heuristic value than one of the top level nodes which were ignored previously. If this is the case then we backtrack to the previously ignored but currently the most promising node and we expand /visit that node. But when we back track, we do not forget the older branch from where we have come. Its last node remains in the list of nodes which have been discovered but not yet expanded/ visited . The search can always return to it if at some stage during the search process it again becomes the most promising node to move ahead.

Choosing the most appropriate heuristic function for a particular search problem is not easy and it also incurs some cost. One of the simplest heuristic functions is an estimate of the cost of getting to a solution from a given node this cost could be in terms of the number of expected edges or hops to be traversed to reach the goal node.

We should always remember that in best first search although one path might be selected at a time but others are not thrown so that they can be revisited in future if the selected path becomes less promising.

Although the example we have given below shows the best first search of a tree, it is sometimes important to search a graph instead of a tree so we have to take care that the duplicate paths are not pursued. To perform this job, an algorithm will work by searching a directed graph in which a node represents a point in the problem space. Each node, in addition to describing the problem space and the heuristic value associated with it, will also contain a link or pointer to its best parent and points to its successor node. Once the goal node is found, the parent link will allow us to trace the path from source node to the goal node. The list of successors will allow it to pass the improvement down to its successors if any of them are already existing.

In the algorithm given below, we assume two different list of nodes:

- **OPEN list →** is the list of nodes which have been found but yet not expanded i.e., the nodes which have been discovered /generated but whose children/successors are yet not discovered. Open list can be implemented in the form of a queue in which the nodes will be arranged in the order of decreasing prioity from the front i.e., the node with the most promising heuristic value (i.e., the highest priority node) will be at the first place in the list.
- **CLOSED list →** contains expanded/visited nodes i.e., the anodes whose successors are also genereated. We require to keep the nodes in memory I we want to search a graph rather than a tree, since whenever a new node is generated we need to check if it has been generated before.

The algorithm can be written as:

Best First Search

1. Place the start node on the OPEN list.
2. Create a list called CLOSED i.e., initially empty.
3. If the OPEN list is empty search ends unsuccessfully.
4. Remove the first node on OPEN list and put this node on CLOSED list.
5. If this is a goal node, search ends successfully.
6. Generate successors of this node:
   For each successor :
   (a) If it has bot been discovered / generated before i.e., it is not on OPEN, evaluate this node by applying the heuristic function, add it to the OPEN and record its parent.
   (b) If it has been discovered / generated before, change the parent if the new path is better than the previous one. In that case update the cost of getting to this node and to any successors that this node may already have.
7. Reorder the list OPEN, according to the heuristic merit.
8. Go to step 3.

Example

In this example, each node has a heuristic value showing the estimated cost of getting to a solution from this node. The example shows part of the search process using best first search.

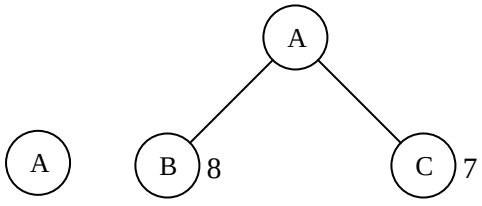


Figure 1

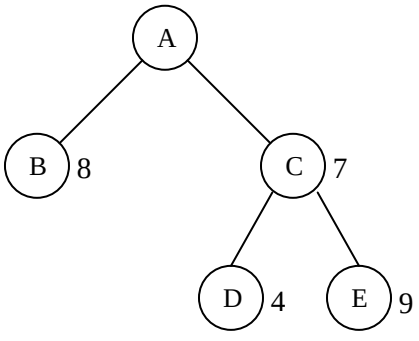


Figure 2

Figure 3

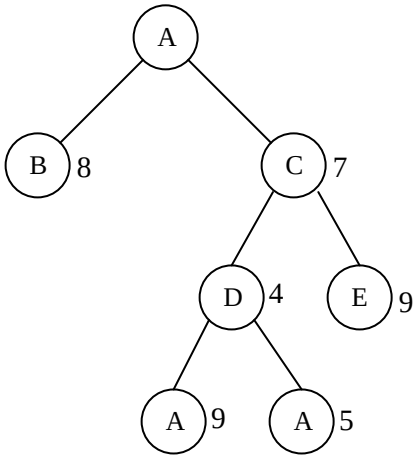


Figure 4

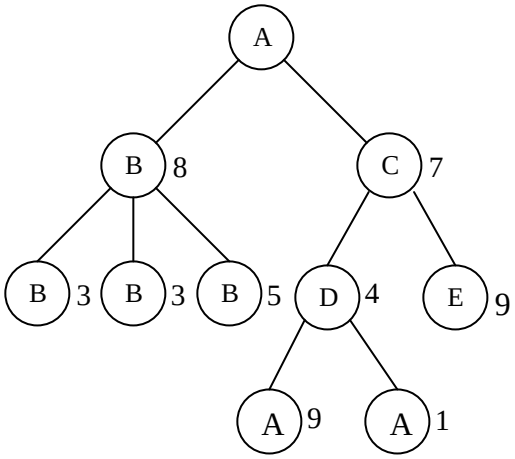


Figure 5

Figure 1: A is the starting node

Figure 2: Generate its successors B and C

Figure 3: As the estimated goal distance of C is less so expand C to find its successors d and e.

Figure 4: Now D has lesser estimated goal distance i.e., 4 , so expand D to generate F and G with distance 9 and 11 respectively.

Figure 5: Now among all the nodes which have been discovered but yet not expanded B has the smallest estimated goal distance i.e., 8, so now backtrack and expand B and so on.

Best first searches will always find good paths to a goal node if there is any. But it requires a good heuristic function for better estimation of the distance to a goal node.

The Minimax Principle

Whichever search technique we may use, it can be seen that many graph problems including game problems, complete searching of the associated graph is not possible. The alternative is to perform a partial search or what we call a limited horizon search from the current position. This is the principle behind the minimax procedure.

Minimax is a method in decision theory for minimizing the expected maximum loss. It is applied in two players games such as tic-tac-toe, or chess where the two players take alternate moves. It has also been extended to more complex games which require general decision making in the presence of increased uncertainty. All these games have a common property that they are logic games. This means that these games can be described by a set of rules and premises. So it is possible to know at a given point of time, what are the next available moves. We can also call them full information games as each player has complete knowledge about possible moves of the adversary.

In the subsequent discussion of games, the two players are named as MAX and MIN. We are using an assumption that MAX moves first and after that the two players will move alternatively. The extent of search before each move will depend o the play depth – the amount of lookahead measured in terms of pairs of alternating moves for MAX and MIN.

As I have already specified, complete search of most game graphs is computationally infeasible. It can be seen that for a game like chess it might take centuries to generate the complete search graph even in an environment where a successor could be

generated in a few nanoseconds. Therefore, for many complex games, we must accept the fact that search to termination is impossible instead we must use partial searching techniques.

For searching we can use either breadth first, depth first or heuristic methods except that the termination conditions must now be specified. Several artificial termination conditions can be specified based on factors such as time limit, storage space and the depth of the deepest node in the search tree.

In a two player game, the first step is to define a **static evaluation function efun()**, which attaches a value to each position or state of the game. This value indicates how good it would be for a player to reach that position. So after the search terminates, we must extract from the search tree an estimate of the best first move by applying a static evaluation function efun() to the leaf nodes of the search tree. The evaluation function measures the worth of the leaf node position. For example, in chess a simple static evaluation function might attach one point for each pawn, four points for each rook and eight points for queen and so on. But this static evaluation is too easy to be of any real use. Sometimes we might have to sacrifice queen to prevent the opponent from a winning move and to gain advantage in future so the key lies in the amount of lookahead. The more number of moves we are able to lookahead before evaluating a move, the better will be the choice.

In analyzing game trees, we follow a convention that the value of the evaluation function will increase as the position becomes favourable to player MAX , so **the positive values will indicate position that favours MAX** whereas for the **positions favourable to player MIN are represented by the static evaluation function having negative values** and values near zero correspond to game positions not favourable to either MAX or MIN. In a terminal position, the static evaluation function returns either positive infinity or negative infinity where as positive infinity represents a win for player MAX and negative infinity represents a win for the player MIN and a value zero represents a draw.

In the algorithm, we give ahead, the search tree is generated starting with the current game position upto the end game position or lookahead limit is reached. Increasing the lookahead limit increases search time but results in better choice. The final game position is evaluated from the MAX's point of view. The nodes that belong to the player MAX receive the maximum value of its children. The nodes for the player MIN will select the minimum value of its children.

In the algorithm, lookahead limit represents the lookahead factor in terms of number of steps, u and v represent game states or nodes, maxmove() and minmove() are functions to describe the steps taken by player MAX or player MIN to choose a move, efun() is the static evaluation function which attaches a positive or negative integer value to a node (i.e., a game state), value is a simple variable.

Now to move number of steps equal to the lookahead limit from a given game state u, MAX should move to the game state v given by the following code :

```
maxval = -
    for each game state w that is a successor of u
      val = minmove(w,lookaheadlimit)
        if (val >= maxval)
            maxval = val
            v = w              // move to the state v
```

The minmove() function is as follows :

```
minmove(w, lookaheadlimit)
{
  if(lookaheadlimit = = 0 or w has no successor)
```

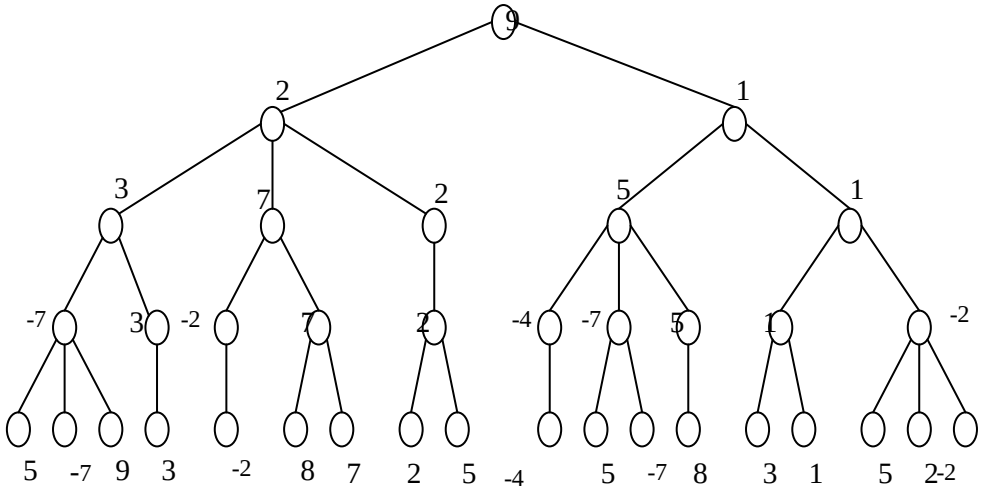
```
    return efun(w)
  else
      minval = +
      for each successor x of w
            val = maxmove(x,lookaheadlimit – 1)
            if (minval > val)
                  minval = val
      return(minval)
}
```

The maxmove() function is as follows :

```
maxmove(w, lookaheadlimit)
{
   if (lookaheadlimit = =  0 or w has no successor)
         return efun(w)
   else
         maxval = -
         for each successor x of w
               val = minmove(x,lookaheadlimit – 1)
               if (maxval < val)
                     maxval = val
         return(maxval)
}
```

We can see that in the minimax technique, player MIN tries to minimize te advantage he allows to player MAX, and on the other hand player MAX tries to maximize the advantage he obtains after each move.

Let us suppose the graph given below shows part of the game. The values of leaf nodes are given using efun() procedure for a particular game then the value of nodes above can be calculated using minimax principle.Suppose the lookahead limit is 4 and it is MAX's turn.

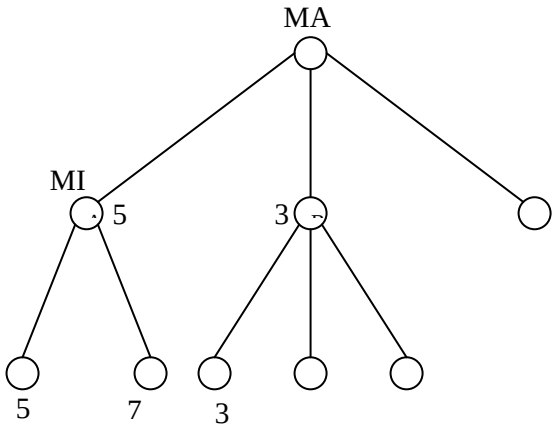


Speeding up the minimax algorithm using Alpha-Beta Pruning

We can take few steps to reduce the search time in minimax procedure. In the figure given below, the value for node A is 5 and the first found value for the subtree starting at node B is 3. So since the B node is at the MIN player level, we know that the selected value for the B node must be less than or equal to 3. But we also know that the A node has the value 5 and both A and B nodes share the same parent at the MAX level immediately above. This means that the game path starting at the B node can never be selected because 5 is better than 3 for the MAX node. So it is not worth

spending time to search for children of the B node and so we can safely ignore all the remaining children of B.

This shows that **the search on same paths can sometimes be aborted** (i.e., it is not required to explore all paths) because we find out that the search subtree will not take us to any viable answer.



This optimization is known as alpha beta pruning/procedure and the values, below which search need not be carried out are known as alpha beta cutoffs.

A general algorithm for alpha beta procedure is as follows:

1. Have two values passed around the tree nodes:

   The alpha value - which holds best MAX value found at the MAX level
   The beta value - which holds best Min value found at the MIN level

2. At MAX player level, before evaluating each child path, compare the returned value of the previous path with the beta value. If the returned value is greater then abort the search for the current node.

3. At Min player level,before evaluating eachh child path, compare the returned value of the previouss path with the alpha value. If the value is lesser then abort the searh for the current node.

We should note that:

- The alpha values of MAX nodes (including the start value) can never decrease.
- The beta value of MIN nodes can never increase.

So we can see that remarkable reductions in the amount of search needed to evaluate a good move are possible by using alpha beta pruning / procedure.

Analysis of BFS Algorithm

In the algorithm BFS, let us analyse the running time taken by the algorithm on a graph G. We can see that each vertex is inserted n the queue exactly once and also deleted from the queue exactly once. So for each the insertion and deletion from the queue costs O(1) time therefore for all vertices queue insertion and deletion would cost O(V) time. Because graph is represented using adjacency list and adjacency list of each vertex is scanned at most once. We can see that the total length of all adjacency list is no. of edge E in the graph G. So a total of O(E) time to spent in scanning all adjacency lists. The initialization portion costs O(V). So the total running time of BFS is O(V+E).

2.7 TOPOLOGICAL SORT

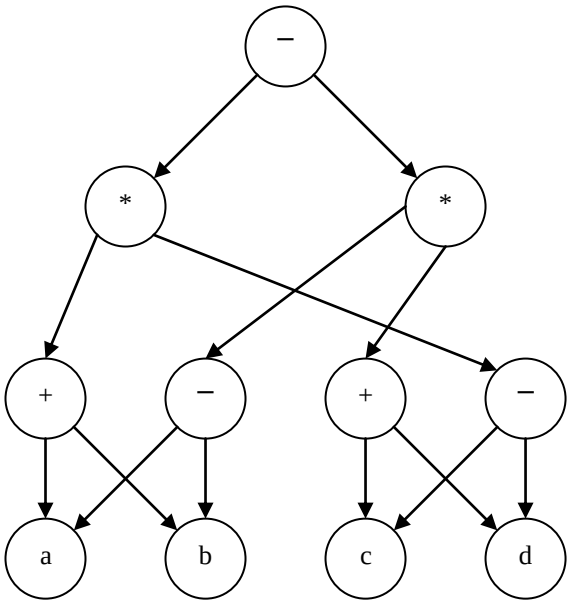
In many applications we are required to indicate precedences or dependencies among various events. Using directed graphs we can easily represent these dependencies. Let a directed graph G with vertex set V and edge set E. An edge from a vertex u to vertex v in the directed graph will then mean that v is dependent on u or v precedes v. Also there cannot be any cycles in these dependency graphs as it can be seen from the following simple argument. Suppose that u is vertex and there is an edge from u to u, i.e., there is a single node cycle. But the graph is dependency graph; this would mean that vertex u is dependent on vertex u which means u must be processed before u which is impossible.

A directed graph that does not have any cycles is known as directed acyclic graph. Hence, dependencies or precedences among events can be represented by using directed acyclic graphs.

There are many problems in which we can easily tell which event follows or precedes a given event, but we can't easily work out in which order all the events are held. For example, it is easy to specify/look up prerequisite relationships between modules in a course, but it may be hard to find an order to take all the modules so that all prerequisite material is covered before the modules that depend on it. Same is the case with a compiler evaluating sub-expressions of an expression like the following:

$$(a + b)(c - d) - (a - b)(c + d)$$

Both of these problems are essentially equivalent. The data of both problems can be represented by directed acyclic graph (See figure below). In the first each node is a module; in the second example each node is an operator or an operand. Directed edges occur when one node depends on the other, because of prerequisite relationships among courses or the parenthesis order of the expression. The problem in both is to find an acceptable ordering of the nodes satisfying the dependencies. This is referred to as a topological ordering. More formally it is defined below.



Directed Acyclic Graph

A **topological sort** is a linear ordering of vertices in a **directed acyclic graph** (normally called **dag)** such that, if there is path from node **u** to node **v**, then **v** appears

after **u** in the ordering. Therefore, a cyclic graph cannot have a topological order. A **topological sort** of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go in one direction.

The term topological sort comes from the study of partial orders and is sometimes called a *topological* order or *linear order*.

ALGORITHM FOR TOPOLOGICAL SORT

Our algorithm picks vertices from a DAG in a sequence such that there are no other vertices preceding them. That is, if a vertex has in-degree 0 then it can be next in the topological order. We remove this vertex and look for another vertex of in-degree 0 in the resulting DAG. We repeat until all vertices have been added to the topological order.

The algorithm given below assumes that the direceted acyclic graph is represented using adjacency lists. Each node of the adjacency list contains a variable indeg which stores the indegree of the given vertex. Adj is an array of |V| lists, one for each vertex in V.

```
Topological-Sort(G)
1 for each vertex u ε G
2         do indeg[u] = in-degree of vertex u
3         if indeg[u] = 0
4                   then enqueue(Q,u)
5 while Q != 0
6         do u = dequeue(Q)
7               print u
8               for each v ε Adj[u]
9                   do indeg[v] = indeg[v] – 1
10                        if indeg[v] = 0
11                              then enqueue(Q,v)
```

The for loop of lines 1-3 calculates the indegree of each node and if the indegree of any node is found to be 0, then it is immediately enqueeud. The while loop of lines 5-11 works as follows. We dequeue a vertex from the queue. Its indegree will be zero(Why?). It then output the vertex, and decrement the in degree of each vertex adjacent to u. If in the process, the in degree of any vertex adjacent to u becomes 0, then it is also enqueued.

TOPOLOGICAL SORT – ANOTHER APPROACH

We can also use Depth First Search Traversal for topologically sorting a directed acyclic graph. DFS algorithm can be slightly changed or used as it is to find the topological ordering. We simply run DFS on the input directed acyclic graph and insert the vertices of a node in a linked list or simply print the vertices in decreasing order of the termination time.

To see why this approach work, suppose that DFS is run on a given dag G = (V,E) to determine the finishing times for its vertices. Let u, v ε V, if there is an edge in G from u to v, then termination time of v will be less than termination time of u i.e., t[v] < t[u]. Since, we output the vertices in decreasing order of termination time, the vertex with least number of dependencies will be outputted first.

ALGORITHM

1. Run the DFS algorithm on graph G. In doing so compute the termination time of each vertex.
2. Whenever a vertex is terminated (i.e. visited), insert it in the front o f a list.

3. Output the list.

RUNNING TIME

Let n is the number of vertices (or nodes, or activities) and m is the number of edges (constraints). As each vertex is discovered only once, and for each vertex we loop over all its outgoing edges once. Therefore, total running time is O(n+m).

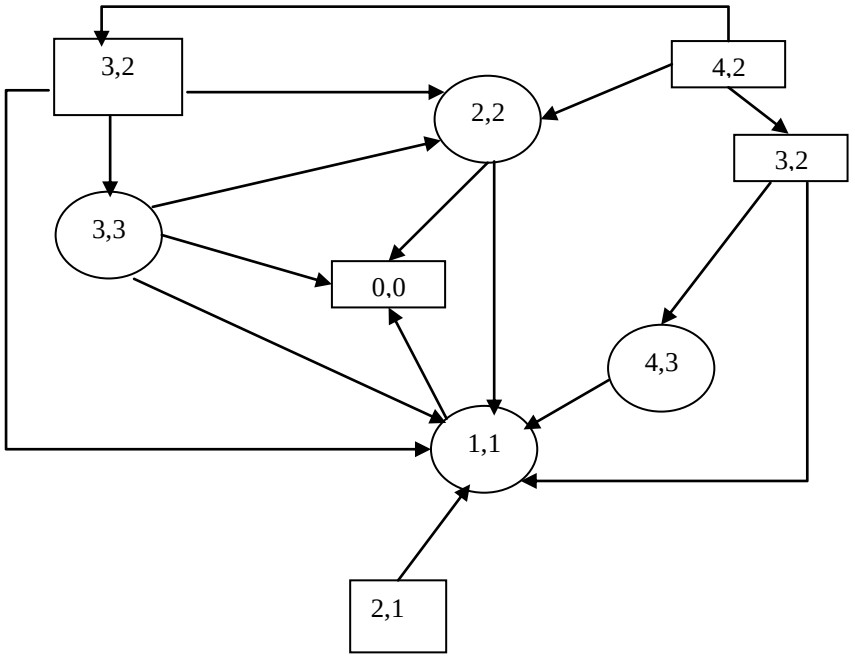
2.8 SUMMARY

This unit discusses some searching and sorting techniques for sorting those problems each of which can be efficiently represented in the form of a graph. In a graphical representation of a problem, generally, a node represents a state of the problem, and an arrow/arc represents a move between a pair of states.

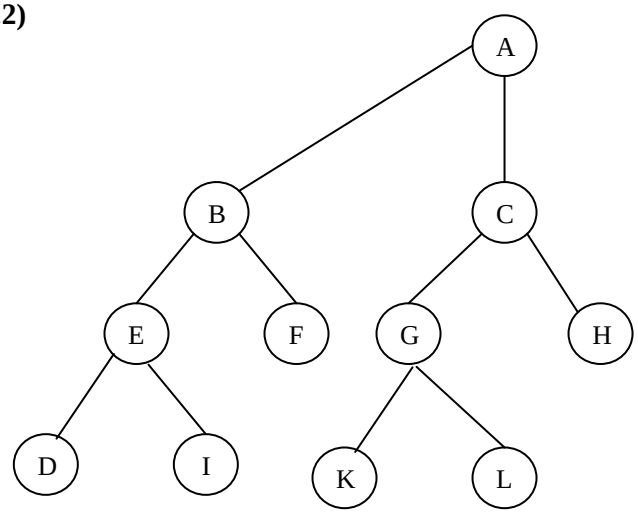
Graph representation of a problem is introduced through the example of a game of NIM/ Marienbad. Then a number of search algorithms viz., Depth-First, Breadth-First, Best-First, and Minimax principal are discussed. Next, a sorting algorithm viz., Topological sort is discussed.

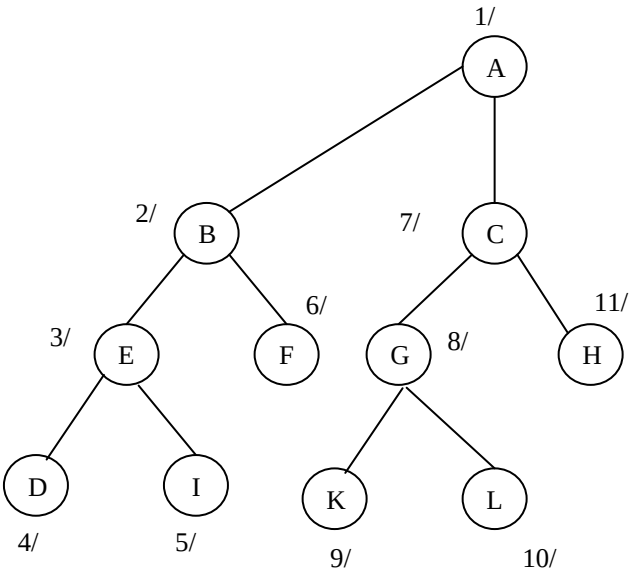
2.9 SOLUTIONS/ANSWERS

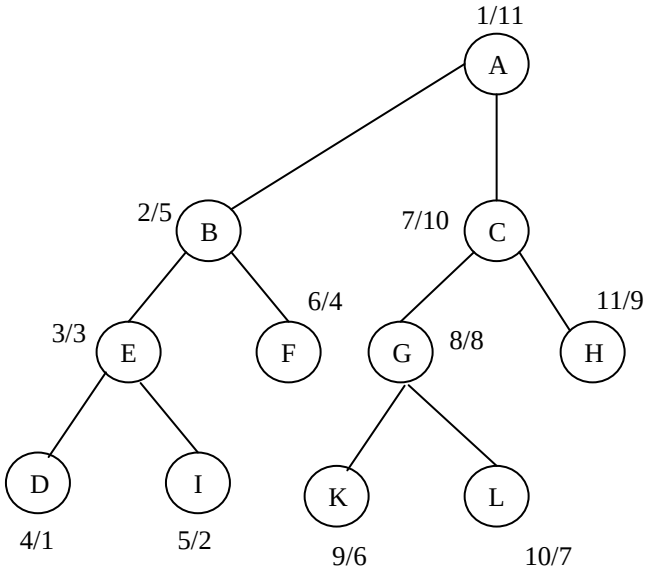
Ex. 1)



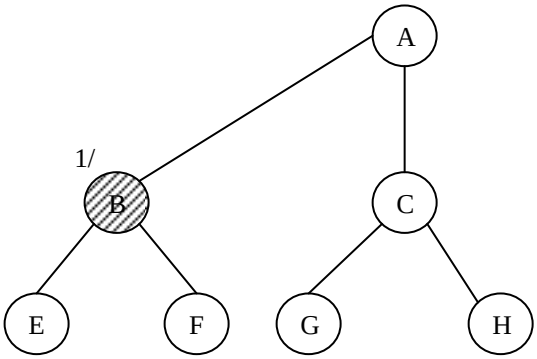
Ex.2)

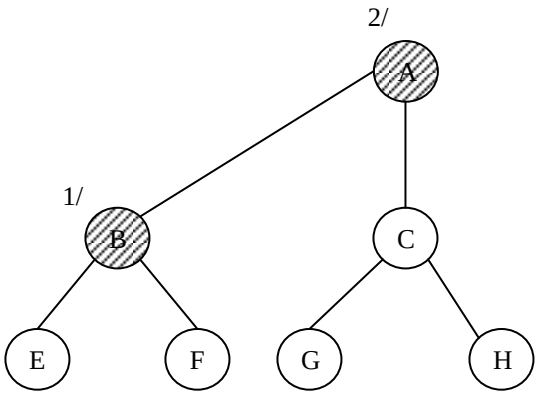


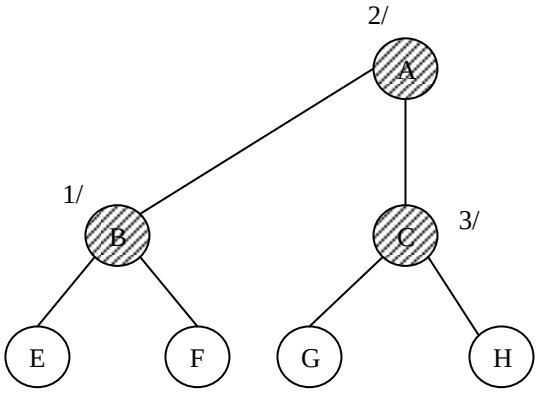

1/
A
2/
B
7/
C
3/
E
6/
F
G
8/
11/
H
D
I
K
L
4/
5/
9/
10/


1/11
A
2/5
B
7/10
C
3/3
E
6/4
F
G
8/8
11/9
H
D
I
K
L
4/1
5/2
9/6
10/7

Ex.3)


A
1/
B
C
E
F
G
H





2.10 FURTHER READINGS

1. *Discrete Mathematics and Its Applications* (*Fifth Edition*) K.N. Rosen: Tata McGraw-Hill (2003).

2. *Introduction to Algorithms* (*Second Edition*), T.H. Coremen, C.E. Leiserson & C. Stein: Prentice-Hall of India (2002).