Design and Analysis of Algorithms

Unit - V

Dr. R. Bhuvaneswari
Assistant Professor
Department of Computer Science
Periyar Govt. Arts College, Cuddalore.

Periyar Govt. Arts College
Cuddalore

Syllabus

UNIT-V

Graph Traversals – Connected Components – Spanning Trees – Biconnected components – Branch and Bound: General Methods (FIFO & LC) – 0/1 Knapsack problems – Introduction to NP-Hard and NP-Completeness.

Text Book:

Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms C++, Second Edition, Universities Press, 2007. (For Units II to V)

Graph Traversals

- The graph is a non-linear data structure. It consists of some nodes and their connected edges. The edges may be directed or undirected.
- A Graph G is a pair (V, E), where V is a finite set of elements called vertices or nodes and E is a set of pairs of elements of V called edges or arcs.
- A graph in which every edge is directed is called **directed graph** or **digraph**.
- A graph in which edges are undirected are called **undirected graph**.
- A graph which contains parallel edges is called **multi-graph**.
- A graph which does not contain parallel edges are called **simple graph**.
- Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way.
- The graph has two types of traversal algorithms.
  - **Depth First Search or Traversal**
  - **Breadth First Search or Traversal**

Depth First Search (DFS)

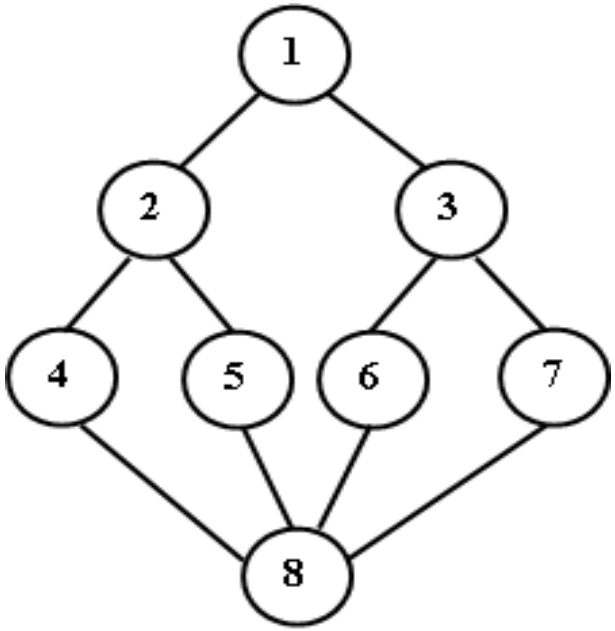
DFS follows the following rules:

1. Select an unvisited node s, visit it, and treat it as the current node.
2. Find an unvisited neighbor of the current node, visit it, and make it the current new node.
3. If the current node has no unvisited neighbors, backtrack to its parent and make that the new current node.

   Repeat the steps 2 and 3 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from step 1.

Depth First Search (DFS)

```
DFS(v)
{
    visited[v] = 1;
    for each vertex w adjacent from v do
    {
        if (visited[w] = 0) then DFS(w);
    }
}
```



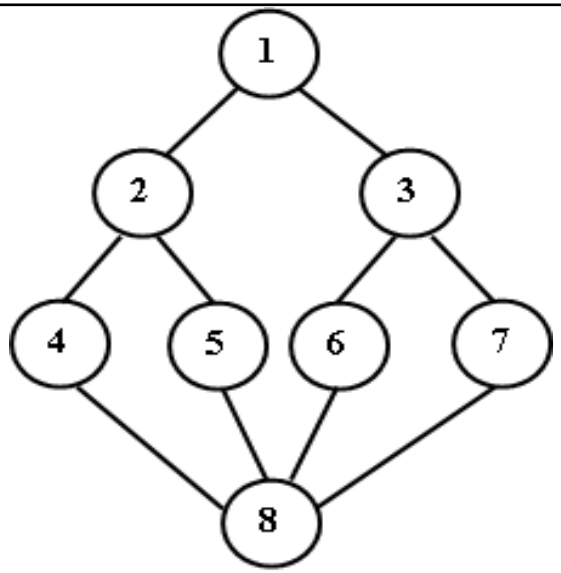
1, 2, 4, 8, 5, 6, 3, 7

Breadth First Search (BFS)

- The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph.
- In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.
- After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.
- This method can be implemented using a queue.
- A Boolean array is used to ensure that a vertex is visited only once.
  - ✓ Add the starting vertex to the queue.
  - ✓ Repeat the following until the queue is empty.
    - ✓ Remove the vertex at the front of the queue, call it v.
    - ✓ Visit v.
    - ✓ Add the vertices adjust to v to the queue, that were never visited.

Breadth First Search (BFS)

```
BFT(G, n)
{
   for i = 1 to n do
      visited[i] = 0;
   for i = 1 to n do
     if(visited[i] = 0) then BFS(i)
}
```

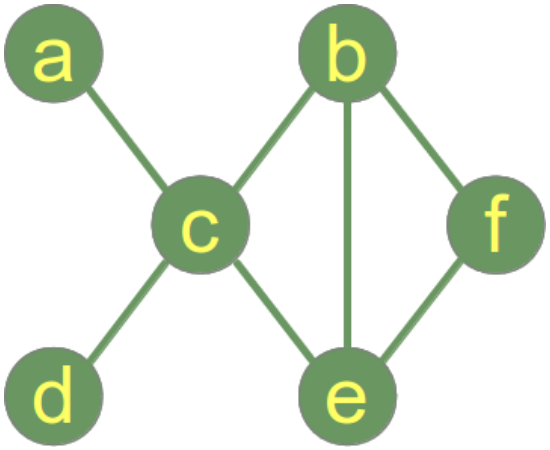


1, 2, 3, 4, 5, 6, 7, 8

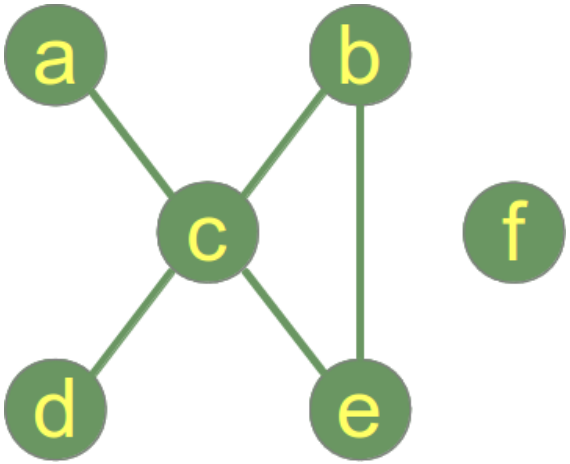
```
BFS(v)
// q is a queue of unexplored vertices
{
   u = v;
   visited[v] = 1;
   repeat
   {
      for all vertices w adjacent from u do
      {
         if(visited[w] = 0) then
         {
            add w to q;
            visited[w] = 1;
         }
      }
      if q is empty then return;
      delete u from q;
   } until(false);
}
```

Connected Components

- A connected component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.
- The connected components of an undirected graph $G = (V, E)$ are the maximal disjoint sets $C_1, C_2, ..., C_k$ such that $V = C_1 \cup C_2 \cup ... \cup C_k$, and $u, v \in C_i$ if and only if u is reachable from v and v is reachable from u.
- The connected components of a graph can be found by performing a depth-first traversal on the graph.



The above is a **connected graph**

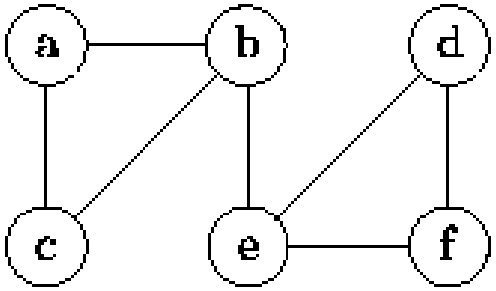


The above graph is **not connected** and has 2 connected components

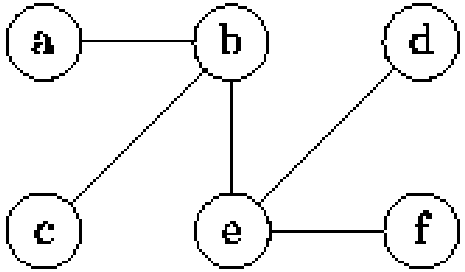
Connected Component 1: {a,b,c,d,e}
Connected Component 2: {f}

Spanning Tree

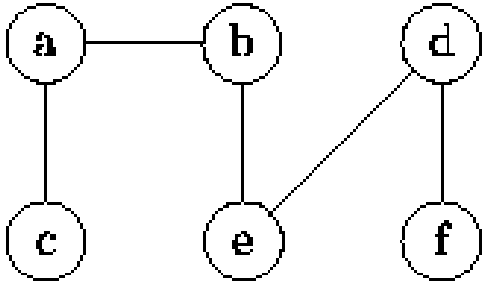
- In a tree there is always **exactly one path** from each vertex in the graph to any other vertex in the graph.
- A **spanning tree** for a graph is a subgraph that includes every vertex of the original, and is a tree.
- A spanning tree that has minimum total weight is called a **minimum spanning tree** for the graph.



(a) Graph G



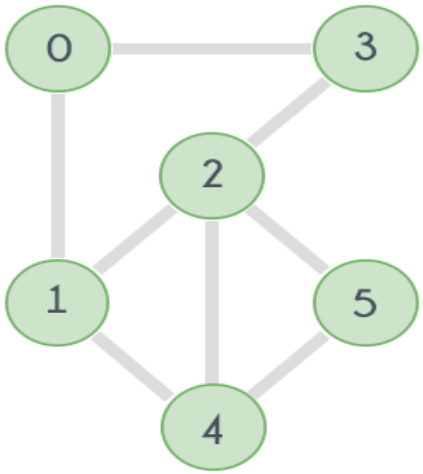
(b) Breadth-first spanning tree of G rooted at b



(c) Depth-first spanning tree of G rooted at c

Biconnected components and DFS

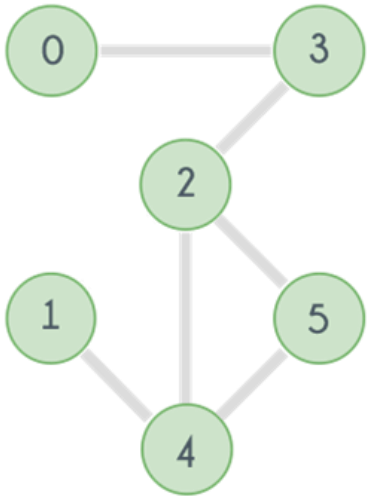
- A graph is said to be Biconnected if:
  1. It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
  2. Even after removing any vertex the graph remains connected.



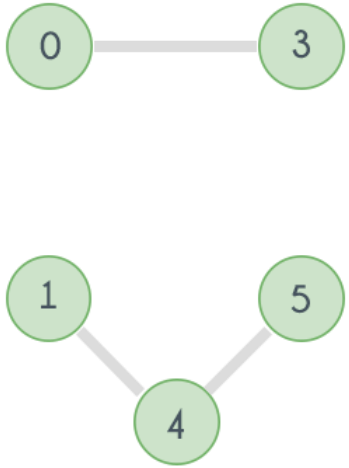
- The given graph is clearly connected. Removing any of the vertices does not increase the number of connected components. So the given graph is Biconnected.

Biconnected components and DFS

Consider the following graph



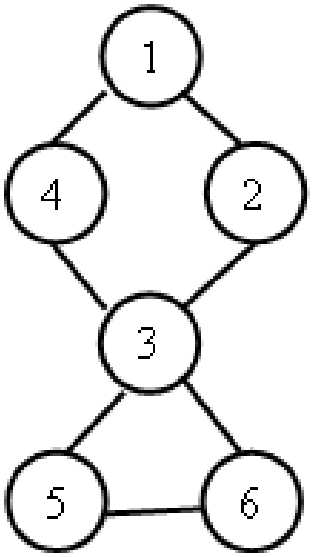
if the vertex 2 is removed,



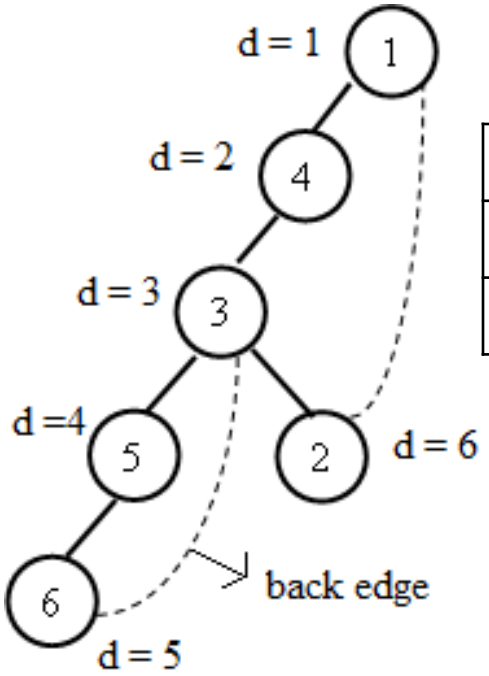
- Similarly, if vertex 3 is removed there will be no path left to reach vertex 0 from any of the vertices 1, 2, 4 or 5.
- Removing vertex 4 will disconnect 1 from all other vertices 0, 2, 3 and 4. So the graph is not Biconnected.
- A graph is Biconnected if it has no vertex such that its removal increases the number of connected components in the graph.

Biconnected components and DFS

- A vertex whose removal increases the number of connected components is called an Articulation Point.
- A vertex v in a connected graph G is an articulation point if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.
- A graph G is biconnected if and only if it contains no articulation points.



1, 4, 3, 5, 6, 2



	1	2	3	4	5	6
d	1	6	3	2	4	5
L	1	1	1	1	3	3

Biconnected components and DFS

- Let (u, v) be a set of vertices, where u is the parent of v.
- If L[v] ≥ d[u] then u is an articulation point, except root.
- The root of an DFS tree is an articulation point if and only if it has 2 children.

Example:

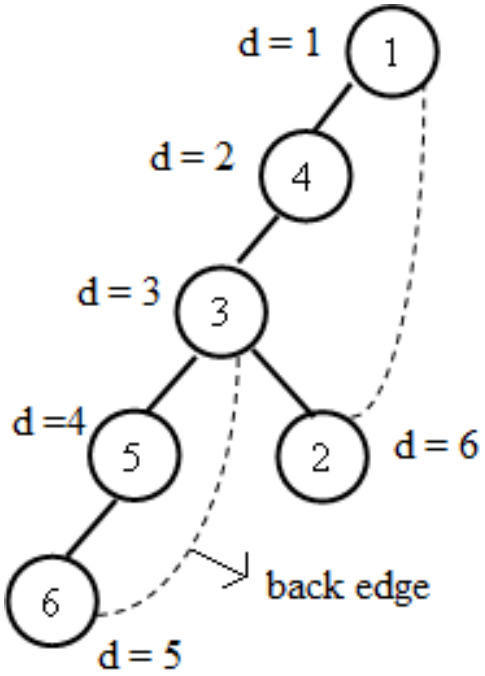
(4, 3)

L[3] ≥ d[4] = 1 ≥ 2 is not true

(3, 5)

L[5] ≥ d[3] = 3 ≥ 3 is true

3 is an articulation point



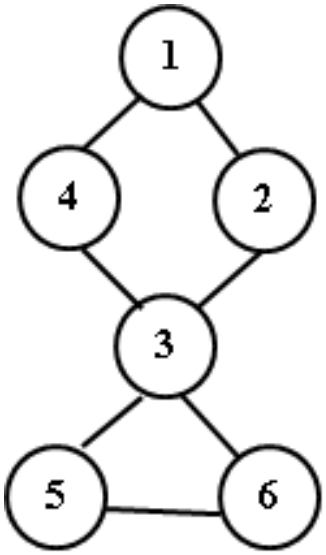
Biconnected components and DFS

```
Algorithm Art(u, v)
//u is a start vertex. V is its parent if any in DFS spanning tree. dfn is
//initialized to zero and num is initialized to 1.
{
   dfn[u] = num; L[u] = num; num = num+1;
   for each vertex w adjacent from u do
  {
     if (dfn[w] = 0) then
    {
       Art(w,u)
       L[u] = min(L[u],L[w]);
    }
    else if (w≠v) then
          L[u] = min(L[u], dfn[w]);
  }
}
```

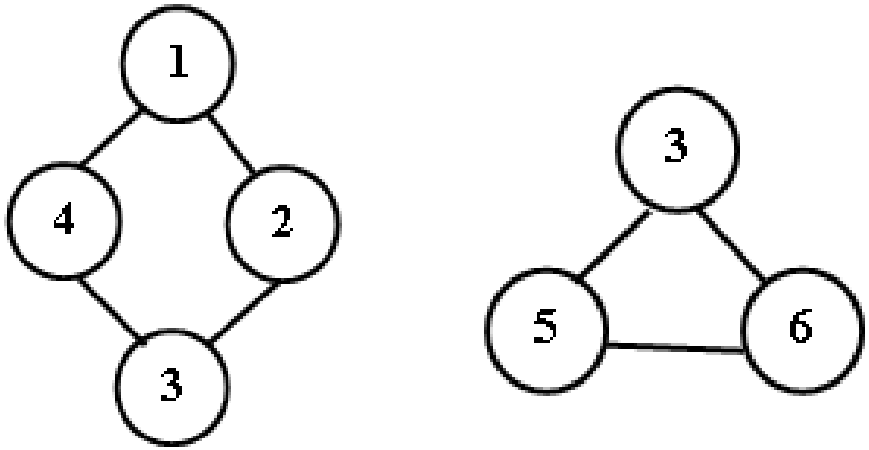
Biconnected components and DFS

Construction of biconnected graph:

- Check the given graph if it is biconnected or not.
- If the given graph is not biconnected, then identify the articulation points.
- If articulation point exists, determine the set of edges whose inclusion makes the graph connected.
- Two biconnected components can have atmost one vertex in common and that vertex is an articulation point.



Given graph G



Biconnected components of G

Biconnected components and DFS

```
Algorithm Bicomp(u, v)
//u is a start vertex for DFS. v is its
//parent if any in the depth first
//spanning tree. It is assumed that the
//global array dfn is initially 0. num→1.
{
 dfn[u] = num; L[u] = num;
 num = num+1;
 for each vertex w adjacent from u do
 {
  if((v ≠ w) and (dfn[w] < dfn[u])) then
     add(u, w) to the top of a stack s;
  if(dfn[w] = 0) then
  {
    Bicomp(w, u);
    L[u] = min(L[u], L[w]);
  if(L[w] ≥ dfn[u]) then
  {
   write("New bicomponent");
   repeat
   {
   delete an edge from the top of stack s;
   let this edge be (x, y);
   write(x, y);
   }until (((x, y) = (u, w)) or
                        ((x, y) = (w, u)));
  }
 }
 else if (w ≠ v) then
       L[u] = min(L[u], dfn[w]);
 }
}
```

Branch and Bound

- Branch and Bound refers to state space search methods in which all children of the E-Node are generated before any other live node becomes the E-Node.
- Branch and Bound is the generalization of both graph search strategies, **BFS** and **D-search**.
  - A BFS like state space search is called as FIFO (First in first out) search, as the list of live nodes are first in first out list (queue).
  - A D-search like state space search is called as LIFO (last in first out) search, as the list of live nodes are last in first out list (stack).
- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound

Three types of search strategies in branch and bound:

- FIFO (First In First Out) search
- LIFO (Last In First Out) search
- LC (Least Cost) search

FIFO (First In First Out) Branch and Bound

- In FIFO search queue data structure is used.
- Initially node 1 is taken as the E-node.
- The child nodes of node 1 are generated. All these live nodes are placed in a queue.
- Next the first element in the queue is deleted, i.e. node 2, the child nodes of node 2 are generated and placed in the queue.
- This continues until the answer node is found.

Branch and Bound

Example:

Job sequencing with deadlines problem

Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3};

d = {1, 2, 1, 2}

Node 1 is the E-node. Child nodes of node 1 are generated and placed in the queue.

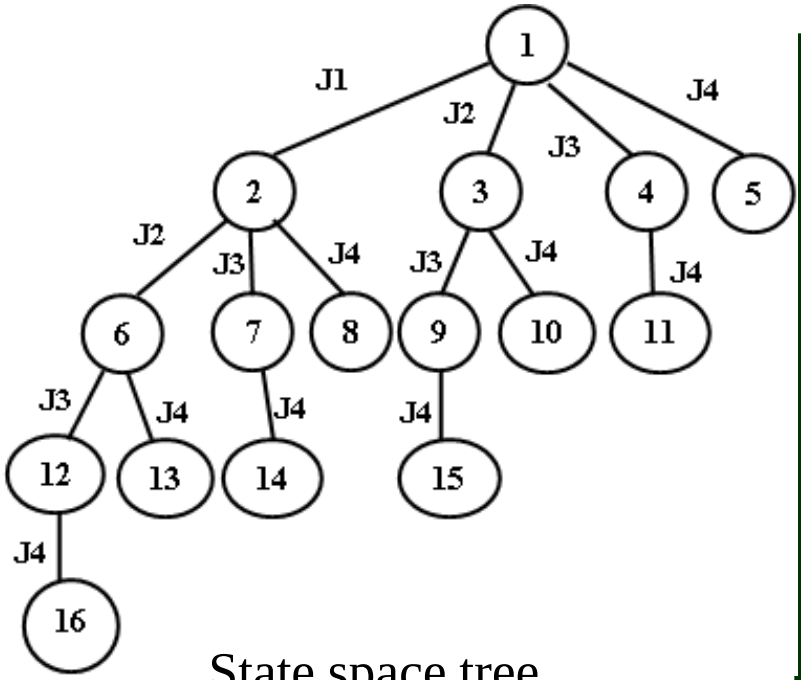
2	3	4	5		

First element in the queue is deleted, ie., 2 is deleted and its child nodes are generated.

3	4	5	6	7	8

Similarly the next element is deleted, ie., 3 and its child nodes are generated and placed in the queue.

This is continued until an answer node is reached.



State space tree

Branch and Bound

LIFO (Last In First Out) Branch and Bound

- In LIFO search stack data structure is used.
- Initially node 1 is taken as the E-node.
- The child nodes of node 1 are generated. All these live nodes are placed in a stack.
- Next the first element in the stack is deleted, i.e. node 5, the child nodes of node 5 are generated and placed in the stack.
- This continues until the answer node is found.

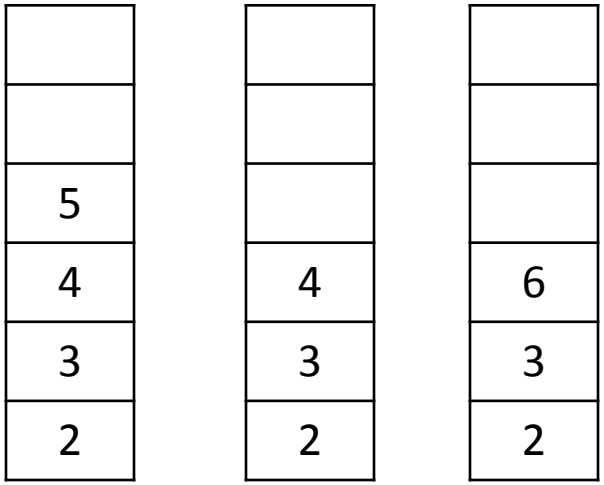
Example:

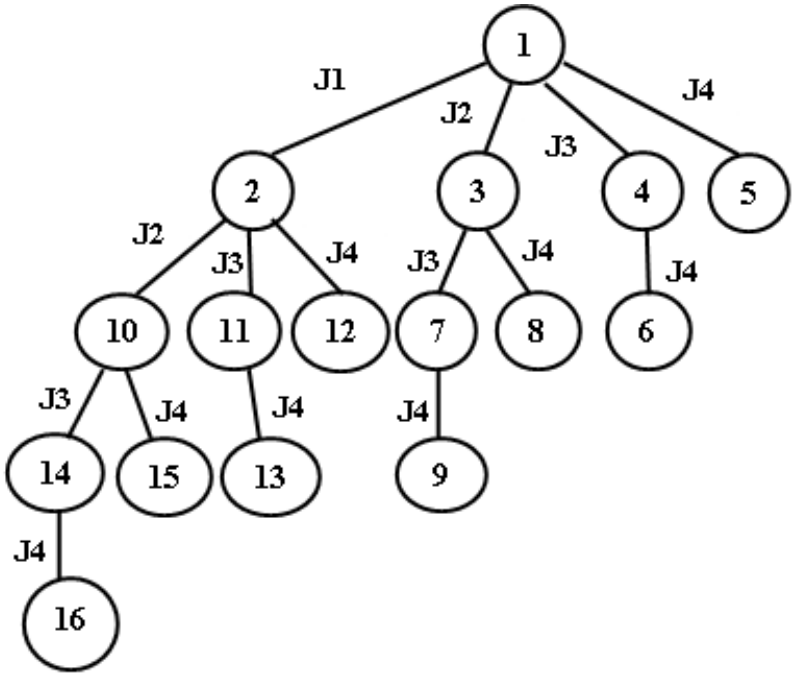
Job sequencing with deadlines problem

Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}

Node 1 is the E-node. Child nodes of node 1 are generated and placed in the stack.

Branch and Bound





State space tree

First element in the stack is deleted, ie., 5 is deleted and its child nodes are generated. Similarly the next element is deleted, ie., 4 and its child nodes are generated and placed in the stack.
This is continued until an answer node is reached.

Branch and Bound

Least Cost Branch and Bound

- In both FIFO and LIFO Branch and Bound the selection rules for the next E-node in rigid and blind.
- The selection rule for the next E-node does not give any preferences to a node that has a very good chance of getting the search to an answer node quickly.
- In this method ranking function or cost function is used.
- The child nodes of the E-node are generated, among these live nodes; a node which has minimum cost is selected. By using ranking function the cost of each node is calculated.

Example:

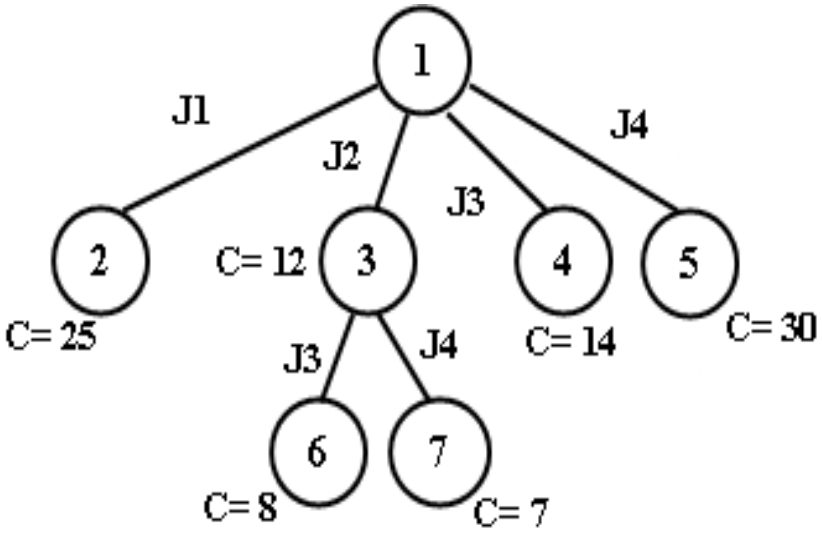
Job sequencing with deadlines problem

Jobs = {J1, J2, J3, J4}; P = {10, 5, 8, 3}; d = {1, 2, 1, 2}

Branch and Bound

Initially we will take node 1 as E-node. Generate children of node 1, the children are 2, 3, 4, 5. By using ranking function we will calculate the cost of 2, 3, 4, 5 nodes is ĉ =25, ĉ =12, ĉ =14, ĉ =30 respectively.

Now we will select a node which has minimum cost i.e., node 3. For node 3, the children are 6, 7.



State space tree

Branch and Bound

Control Abstraction for LC-Search

- Let t be a state space tree and c() a cost function for the nodes in t.
- If x is a node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.
- LC search uses ĉ to find an answer node.
- The algorithm uses two functions
  - Least-cost()
  - Add_node()
- Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.
- Add_node() to delete and add a live node from or to the list of live nodes.
- Add_node(x)adds the new live node x to the list of live nodes.

Branch and Bound

```
Algorithm LCSearch(t)
{
  if *t is an answer node then output *t and return;
  E = t;
  initialize the list of live nodes to be empty;
  repeat
  {
     for each child x of E do
    {
       if x is an answer node then output the path from x to t and return;
       Add(x);
       (x→parent) = E;
    }
    if there are no more live nodes then
    {
       write("No answer node"); return;
    }
    E= Least();
  }until(false);
}
```

Branch and Bound

Bounding

- A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.
- A good bounding helps to prune (reduce) efficiently the tree, leading to a faster exploration of the solution space. Each time a new answer node is found, the value of upper can be updated.
- Branch and bound algorithms are used for optimization problem where we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.
- A cost function $\hat{c}(x)$ is estimated to give the lower bound of the cost of a node x, $c(x)$, such that $\hat{c}(x) \leq c(x)$ and cost function upper is derived such that if $c(x) \geq \hat{c}(x) >$ upper then such nodes are killed or bounded.
- The initial value of upper is estimated by a heuristic search or set to ∞. Every time a live node is generated the value of upper is also updated.

0/1 Knapsack problem

- n objects are given and capacity of knapsack is m.
- Select some objects to fill the knapsack in such a way that it should not exceed the capacity of knapsack and maximum profit can be earned. The knapsack problem is maximization problem. It means we will always seek for maximum $p_i x_i$ (where p_i represents profit of object x_i).
- Since the branch bound deals only the minimization problems the objective function would be negated and changed to minimize $\Sigma p_i x_i$ subject to $\Sigma w_i x_i \leq m$.
- This modified knapsack problem is stated as,

$$\text{minimize} \; - \sum_{1 \leq i \leq n} p_i \, x_i$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, 1 \leq i \leq n$$

- The two functions $\hat{c}(x)$ and $U(x)$ are defined using two algorithms Bound and UBound .

0/1 Knapsack problem

- UBound computes the weights of the list of objects placed in the knapsack as a whole and their sum ≤ m, and the profit is correspondingly decremented from initial profit and returned.
- Bound is similar to UBound but it considers fractional objects to use the entire capacity of the sack $\Sigma w_i x_i = m$.

```
Algorithm Ubound(cp, cw, k, m)
{
 b = cp; c = cw;
 for i = k+1 to n do
 {
   if(c+w[i] ≤ m) then
  {
     c = c+w[i]; b = b – p[i];
  }
 }
 return b;
}
```

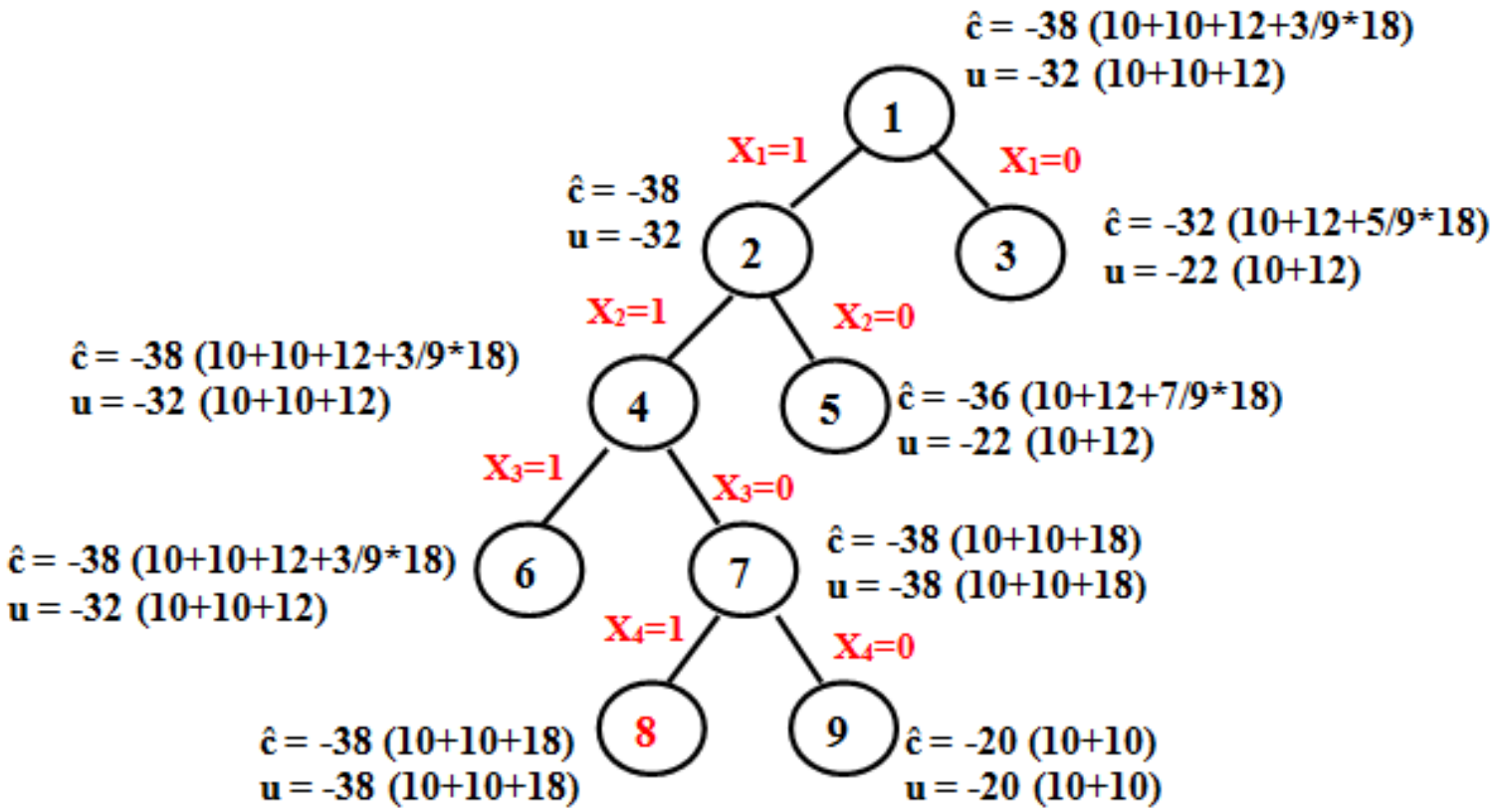
```
Algorithm Bound(cp, cw, k)
{
 b = cp; c = cw;
 for i = k+1 to n do
 {
   c = c+w[i];
   if(c < m) then b = p[i];
   else return b - (1 - (c - m) / w[i])*p[i];
 }
 return b;
}
```

0/1 Knapsack problem

n = 4; m = 15;

$(p_1, p_2, p_3, p_4) = \{10, 10, 12, 18\}$; $(w_1, w_2, w_3, w_4) = \{2, 4, 6, 9\}$

$x_1 = 1$,
$x_2 = 1$,
$x_3 = 0$,
$x_4 = 1$



Introduction to NP-Hard and NP-Complete

- P, NP, NP-Hard and NP-Complete are classes that any problem would fall under or would be *class*ified as.

P (Polynomial) problems

- P problems refer to problems where an algorithm would take a polynomial amount of time to solve.
- If an algorithm is polynomial, we can formally define its time complexity as:
  $T(n) = O(C * n^k)$ where $C > 0$ and $k > 0$ where C and k are constants and is input size.
- In general, for polynomial-time algorithms k is expected to be less than n.
- Many algorithms complete in polynomial time:
  - Linear Search (n)
  - Binary Search (logn)
  - Insertion Sort (n^2)
  - Merge Sort (nlogn)
  - Matrix Multiplication (n^3)

Introduction to NP-Hard and NP-Complete

NP (Non-deterministic Polynomial) Problems

- NP class problems don't have a polynomial run-time to *solve*, but have a polynomial run-time to *verify* solutions.
- These algorithms have an exponential complexity, which we'll define as:

  $T(n) = O(C_1 * k^{nC2})$ where $C_1 > 0$, $C_2 > 0$ and $k > 0$ where C_1, C_2, k are constants and n is the input size.
- There are several algorithms that fit this description.
  - 0/1 knapsack problem (2^n)
  - Traveling salesperson problem
  - Sum of Subsets problem
  - Graph coloring problem
  - Hamiltonian cycles problem

Introduction to NP-Hard and NP-Complete

Non deterministic Algorithms

When the result of every operation is uniquely defined then it is called deterministic algorithm.

When the outcome is not uniquely defined but is limited to a specific set of possibilities, we call it non deterministic algorithm.

New statements to specify such algorithms.

- **choice(S): arbitrarily choose one of the elements of set S**
- **failure: signals an unsuccessful completion**
- **success: signals a successful completion**

The assignment X:= choice(1:n) could result in X being assigned any value from the integer range[1..n]. There is no rule specifying how this value is chosen.

The nondeterministic algorithms terminates unsuccessfully if and only if there is no set of choices which leads to the successful signal.

Introduction to NP-Hard and NP-Complete

Example:

Searching an element x in a given set of elements A(1:n). We are required to determine an index j such that a(j) = x or j = 0 if x is not present.

Algorithm NSearch(A, n, key)

```
{
   j = choice();
   if(key = a[j]) then
  {
       write(j); Success();
  }
  write(0);
  Failure();
}
```

Introduction to NP-Hard and NP-Complete

- Any problem for which the answer is either zero or one (yes or no) is called a **decision problem**. An algorithm for a decision problem is termed a decision algorithm.
- Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an **optimization problem**. An optimization algorithm is used to solve an optimization problem
- Many problems will have decision and optimization versions.
  Eg. Traveling salesperson problem.
  - optimization: find Hamiltonian cycle of minimum weight
  - decision: is there a Hamiltonian cycle of weight ≤ k

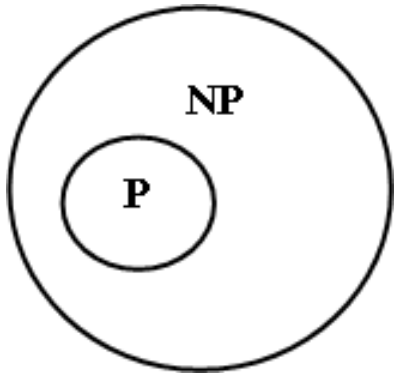
Introduction to NP-Hard and NP-Complete

Definition:

- P is a set of all decision problems solvable by a deterministic algorithm in polynomial time.
- NP is the set of all decision problems solvable by a nondeterministic algorithm in polynomial time.

$\Rightarrow \mathbf{P \subseteq NP}$

Definition. Let L_1 and L_2 be problems. L_1 reduces to L_2($L_1 \propto L_2$) if and only if there is a way to solve L_1 by deterministic polynomial time algorithm that solve L_2 in polynomial time.



Introduction to NP-Hard and NP-Complete

- A problem is **NP-hard** if all problems in NP are polynomial time reducible to it.
- A problem is **NP-complete** if the problem is both
  - NP-hard, and
  - NP
- All NP-Complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete.
- NP-Complete problems are subclass of NP-Hard.