UNIT - III

GRAPHS (Algorithm and Analysis): Breadth first search and traversal, Depth first search and traversal, Spanning trees, connected components and bi-connected components, Articulation points.

DYNAMIC PROGRAMMING: General method, applications - optimal binary search trees, 0/1 knapsack problem, All pairs shortest path problem, Travelling sales person problem, Reliability design.

GRAPHS

Techniques for graphs:

Given a graph G = (V, E) and a vertex V in V (G) traversing can be done in two ways.

1. Depth first search

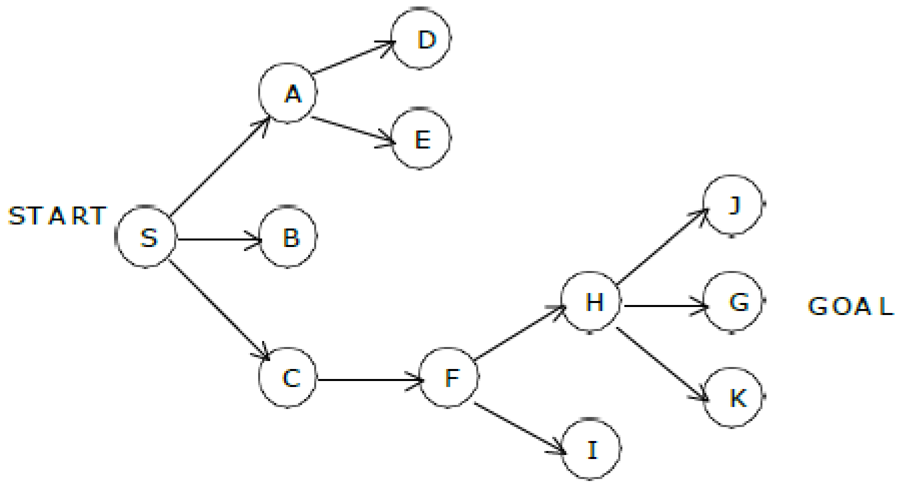
2. Breadth first search

DEPTH FIRST SEARCH:

With depth first search, the start state is chosen to begin, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it “backs up” that means it goes to the immediately previous state or predecessor formally, the state whose successor was ‘S’ originally.

For example consider the figure. The circled letters are state and arrows are branches.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state. So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get struck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. One more problem is that, the state space tree may be of infinite depth, to prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors.

3. We cannot come up with shortest solution to the problem.

Time Complexity:

Let n = |V| and e = |E|. Observe that the initialization portion requires □ (n) time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n assuming each vertex is reachable from the source). As, each vertex is visited at most once. At each vertex visited, we scan its adjacency list once. Thus, each edge is examined at most twice (once at each endpoint). So the total running time is O (n + e).

Alternatively,

If the average branching factor is assumed as 'b' and the depth of the solution as 'd', and maximum depth $m \geq d$.

The worst case time complexity is O(bm) as we explore bm nodes. If many solutions exists DFS will be likely to find faster than the BFS.

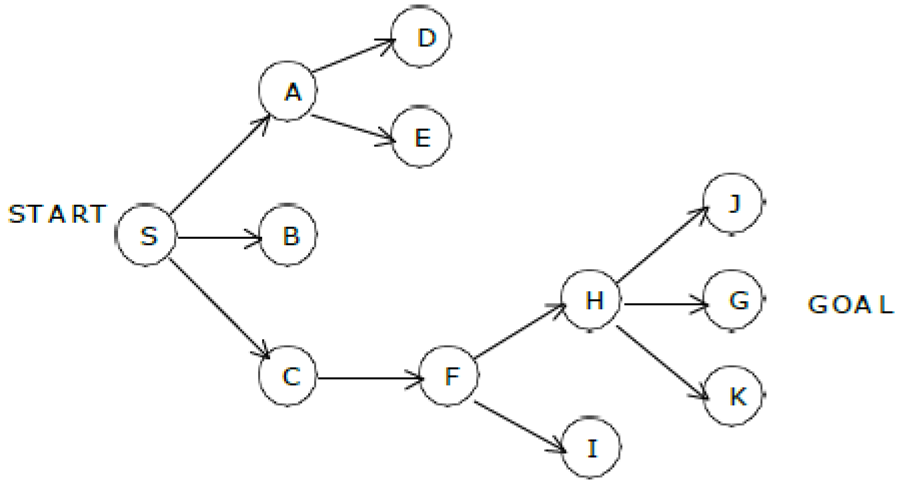
Space Complexity:

We have to store the nodes from root to current leaf and all the unexpanded siblings of each node on path. So, We need to store bm nodes.

BREADTH FIRST SEARCH:

Given an graph G = (V, E), breadth-first search starts at some source vertex S and "discovers" which vertices are reachable from S. Define the distance between a vertex V and S to be the minimum number of edges on a path from S to V. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths

(where the length of a path = number of edges on the path). Breadth-first search is named because it visits vertices across the entire breadth.
To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, e and F at level 2; H and I at level 3; and J, G and K at level 4. So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.
One simple way to implement breadth first search is to use a queue data structure consisting of just a start state. Any time we need a new state, we pick it from the front of the queue and any time we find successors, we put them at the end of the queue. That way we are guaranteed to not try (find successors of) any states at level 'N' until all states at level 'N – 1' have been tried.

Time Complexity:
The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Let n = |V| and e = |E|. Observe that the initialization portion requires (n) time. Since we never visit a vertex twice, the number of times we go through the loop is at most n (exactly n, assuming each vertex is reachable from the source). So, Running time is O (n + e) as in DFS. For a directed graph the analysis is essentially the same.
Alternatively,
If the average branching factor is assumed as 'b' and the depth of the solution as 'd'.
In the worst case we will examine 1 + b + b2 + b3 + . . . + bd = (bd + 1 - 1) / (b –1) = O(bd).
In the average case the last term of the series would be bd / 2. So, the complexity is still O(bd)

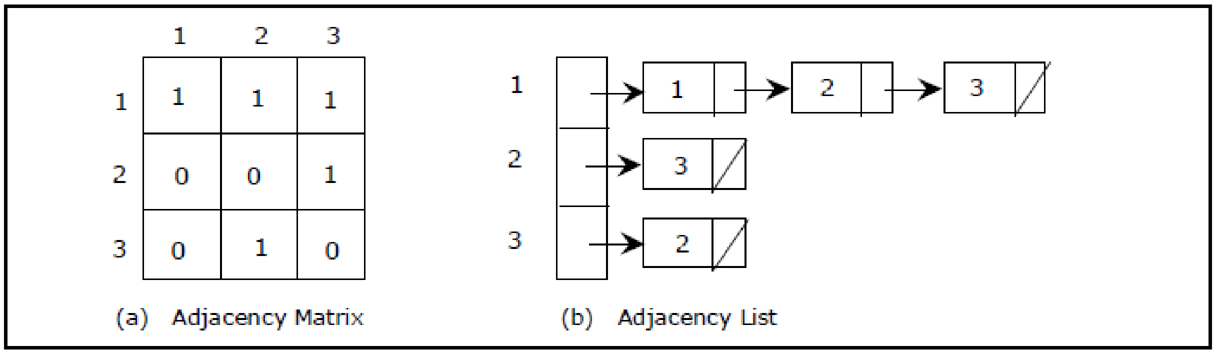
Space Complexity:
Before examining any node at depth d, all of its siblings must be expanded and stored. So, space requirement is also O(bd).

REPRESENTATION OF GRAPHS AND DIGRAPHS BY ADJACENCY LIST:

We will describe two ways of representing digraphs. We can represent undirected graphs using exactly the same representation, but we will double each edge, representing the undirected edge {v, w} by the two oppositely directed edges (v, w) and (w, v). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.
Let G = (V, E) be a digraph with n = |V| and let e = |E|. We will assume that the vertices of G are indexed {1, 2, ... , n}.

$$A[v, w] = \begin{cases} 1 & \text{if } (v,w) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency List: An array Adj [1 . . . . n] of pointers where for 1 < v < n, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.



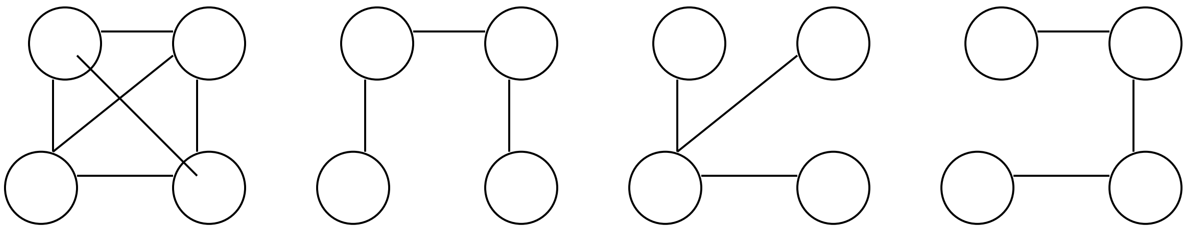
Adjacency matrix and adjacency list

An adjacency matrix requires **Θ** (n2) storage and an adjacency list requires **Θ** (n + e) storage.
Adjacency matrices allow faster access to edge queries (for example, is (u, v) E) and adjacency lists allow faster access to enumeration tasks (for example, find all the vertices adjacent to v).

SPANNING TREES

Definition: Let G = (V, E) be an undirected connected graph. A sub graph t = (V, E’) of G is a spanning tree of G iff t is a tree.

Example:

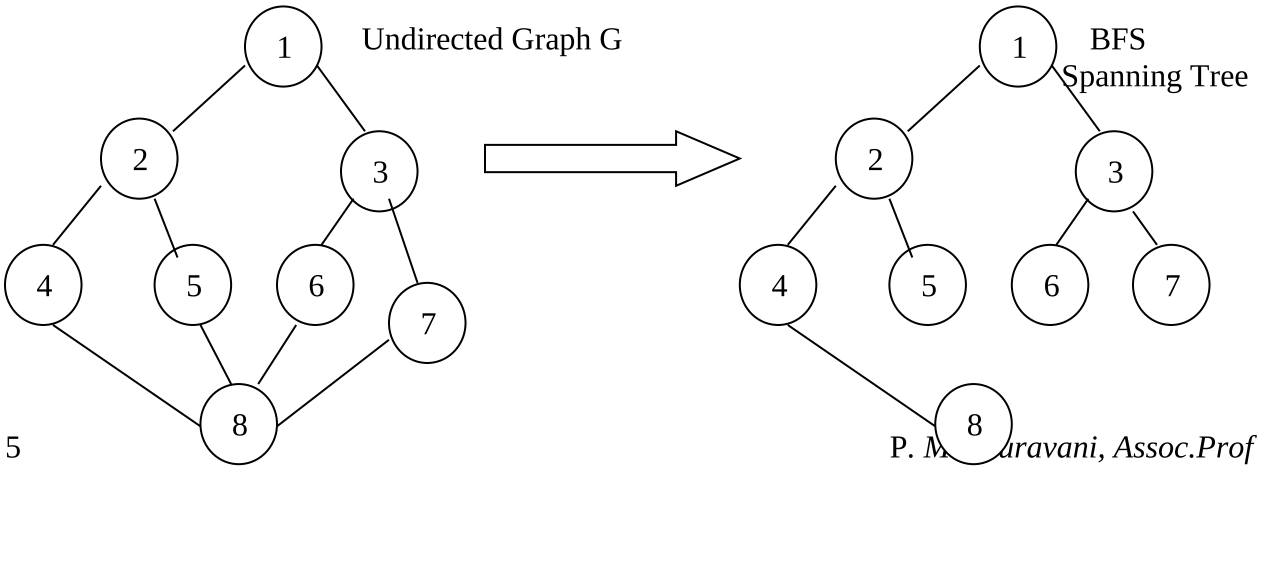


An undirected graph and three of its spanning trees.

Spanning trees have many applications. An application of spanning trees arises from the property that a spanning tree is a minimal sub graph G’ of G such that V(G’) = V(G) and G’ is connected. (A minimal sub graph is one with the fewest number of edges). Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is n-1. The spanning trees of G represent all feasible choices.

CONNECTED COMPONENTS

If G is a connected undirected graph, then all vertices of G will get visited on the first call to BFS. If G is not connected, then at least two calls to BFS will be needed. Hence BFS can be used to determine whether G is connected.



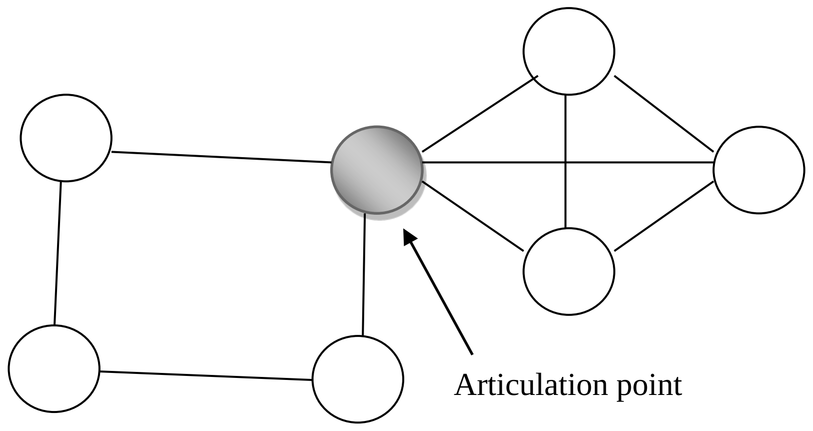
All newly visited vertices on a call to BFS from BST represent the vertices in a connected component of G. Hence the connected components of a graph can be obtained using BFT. For this, BFS can be modified so that all newly visited vertices are put onto a list. Then the sub graph formed by the vertices on this list makes up a connected component. Hence, if adjacency lists are used, a breadth first traversal will obtain the connected components in Θ(n + e) time.

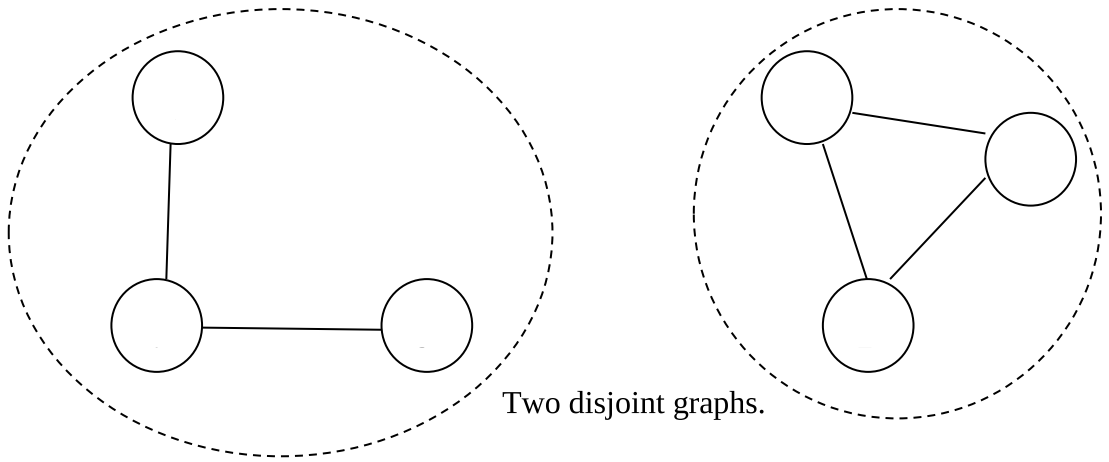
BICONNECTED COMPONENTS & ARTICULATION POINT

A "graph" means always an undirected graph. 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.

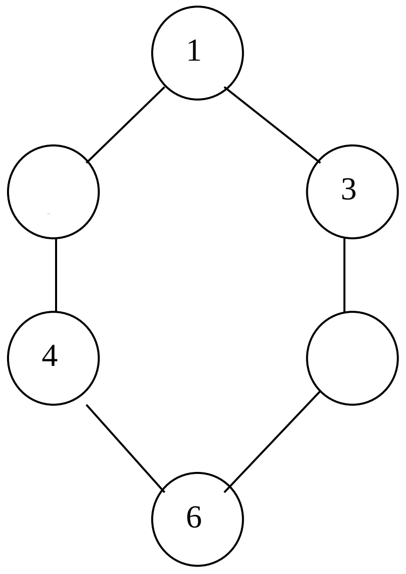
A graph G is biconnected if and only if it contains no articulation points.

Definition of Articulation point: Let G = (V, E) be a connected undirected graph, then an articulation point of graph G is a vertex whose removal disconnectes graph G. This articulation point is a kind of cut-vertex.



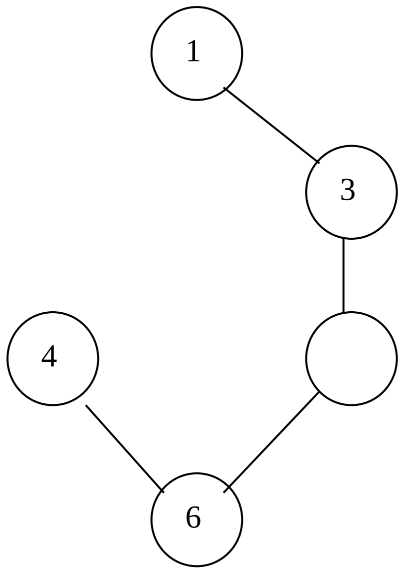


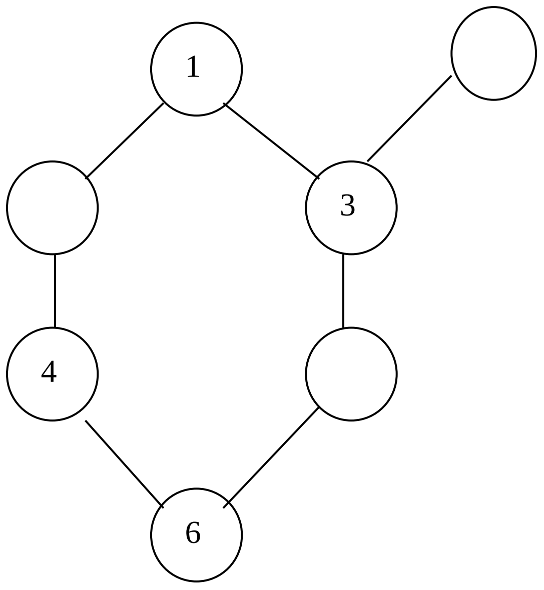
A graph G is said to be bi-connected if it contains no articulation points.
Eg:



Even though we remove any single vertex we do not get disjoint graphs. Let us remove vertex 2 and we will get

but





Biconnectivity Not a biconnective graph

Identification of Articulation Point

- The easiest method is to remove a vertex and its corresponding edges one by one from graph G and test whether the resulting graph is still disconnected or not. The time complexity of this activity will be O(V(V+E)).
- Another method is to use depth first search in order to find the articulation point. After performing depth first search on the given graph we get a 'DFS tree'.
- While building the DFS tree we number outside each vertex. These numbers indicate the order in which a depth first search visits the vertices. These number are called as depth first search numbers (dfn) of the corresponding vertex.
- While building the DFS tree we can classify the graph into four categories:
  i) Tree edge: It is an edge in depth first search tree.
  ii) Back edge: It is an edge (u, v) which is not in DFS tree and v is an ancestor of u. It basically indicates a loop.
  iii) Forward edge: An edge (u, v) which is not in search tree and u is an ancestor of v.
  iv) Cross edge: An edge (u, v) not in search tree and v is neither an ancestor nor a descendant of u.
- To identify articulation points following observations can be made:
  i) The root of the DFS tree is an articulation if it has two or more children.
  ii) A leaf node of DFS tree is not an articulation point.
  iii) If u is any internal node then it is not an articulation point if and only if from every child w of u it is possible to reach an ancestor of u using only a path made up of descendents of w and back edge.

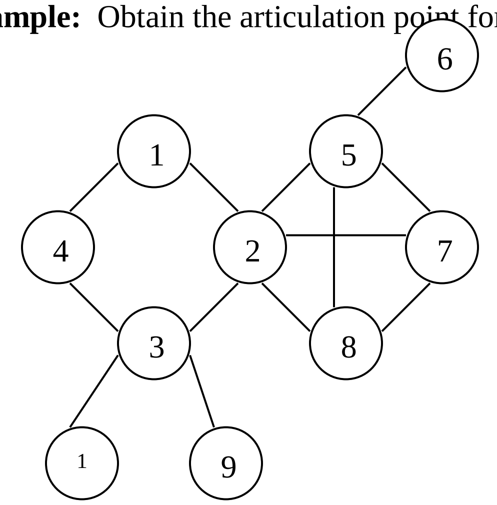
This observation leads to a simple rule as,

Low[u] = min { dfn[u], min{Low[w]/w is a child of u}, min{dfn[w]/(u, w) is a back edge}}

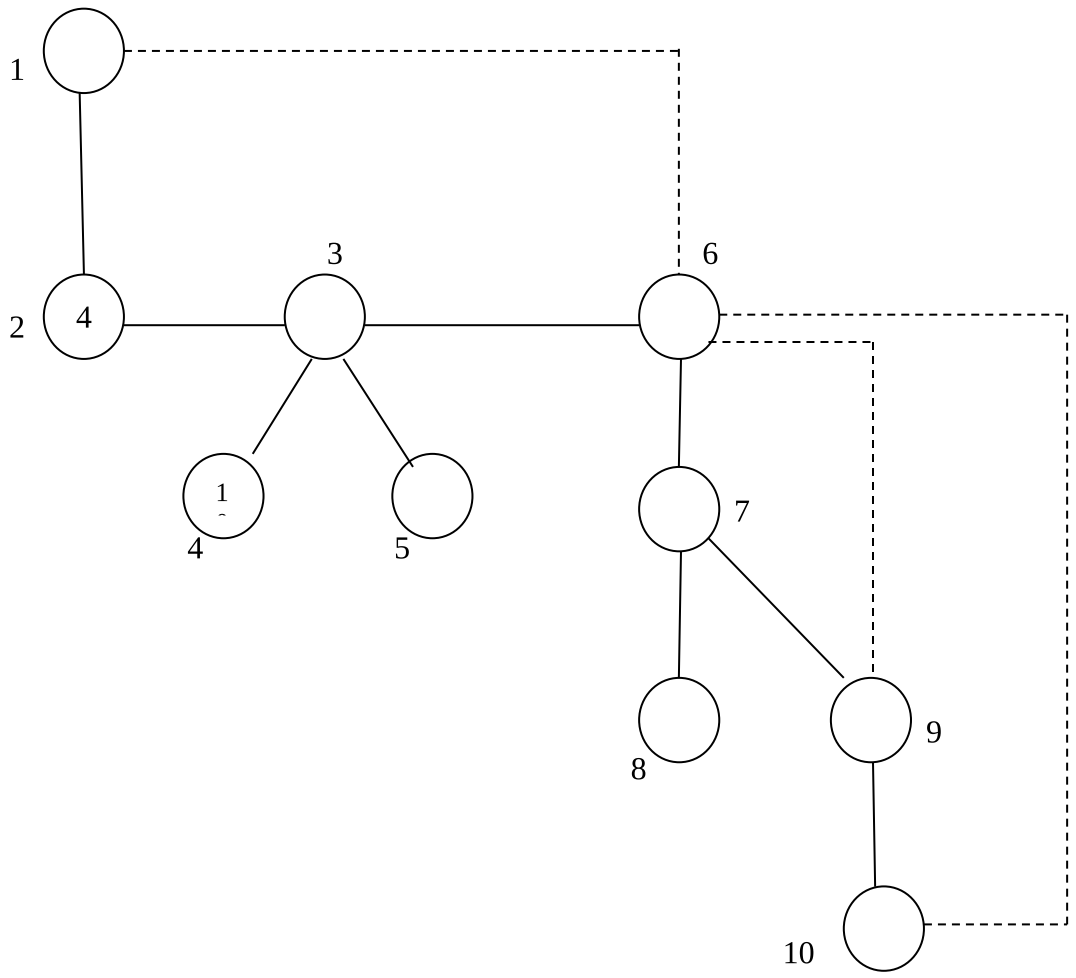
Where Low[u] is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge. The vertex u is an articulation point if u is child of w such that

L[w] $\geq$ dfn[u]

Example: Obtain the articulation point for following graph.



Solution: The DFS tree can be drawn as follows:



Let us compute Low[u] using formula

Low[u] = min { dfn[u], min{Low[w]/w is a child of u}, min{dfn[w]/(u, w) is a back edge}}

Low[1] = min { dfn[1] min{Low[4]}, dfn[2]}
= min{1, Low[4], 6}

Low[1] = 1 { nothing is less than 1}

Low[2] = min{dfn[2], min{Low[5]}, dfn[1]}
` = min{6, … Low[5], 1}

Low[2] = 1

Low[3] = min {dfn[3], min { Low[10], low[9], low[2]}, no back edge}
=min {3, min{Low[10], Low[9], 1} -}
Min{3, 1, -}

Low[3] = 1

Low[4] = min {dfn[4], min{Low[3]} -}
Min{2,1, -}

Low[4] = 1

Low[5] = min{dfn[5], min {Low[6], Low[7], -}}
= min{7, min{Low[6], Low[7], -}}

Low[5] = Keep it as it is after getting value of Low[6] and Low[7] we will decide Low[5]

Low[6] = min {dfn[6], -, -} leaf node

Low[6] = 8

Low[7] = min{dfn[8], -, dfn[2]}
= min{10, -, 6}

Low[7] = 6

As we have got Low[6] =8 and Low[7] = 6 we will compute our incomplete computation Low[5]

Low[5] = min{7, min{8, 6}, -}
= min{7, 6, -}

Low[5] = 6

Low[8] = min{dfn[8], -, dfn[2]}
=min{10, 6}

Low[8] = 6

Low[9] = min{dfn[9], -, -}
Low[9] = 5

Low[10] = min{dfn[10], -, -}

=min{4, -, -}
Low[10] = 4

Hence Low values are Low[1:10] = {1, 1, 1, 1, 6, 8, 6, 6, 5, 4}.
Here vertex 3 is articulation point because child of 3 is 10 and Low[10] = 4. Dfn[3] = 3. That is Low[w] ≥dfn[u].

Similarly vertex 2 is an articulation point. Because child of 2 is 5 and
Low[5] =6, dfn[2] = 6

i.e. Low[w] ≥dfn[u]

Vertex 5 is articulation point because child of 5 is 6.

Low[6] = 8, dfn[5] = 7

i.e. Low[w] ≥dfn[u]

Hence in the above graph vertex 2, 3, and 5 are articulation points.

Algorithm for articulation points:

Algorithm Art(u, v)
//The vertex u is a starting vertex for depth first search. V is its parent if any in the depth first //spanning tree. It is assumed that the global array dfn is initialized to 0 and that the global //variable num is initialized to 1. N is the number of vertices in G.

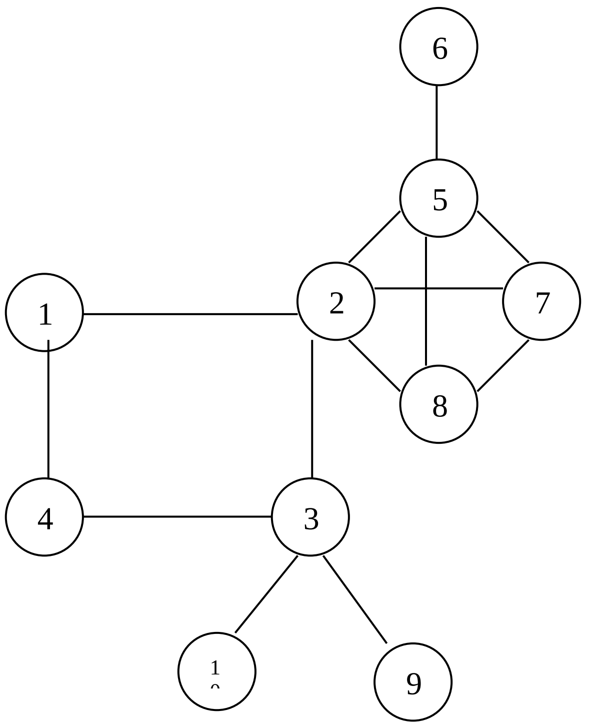
```
    {
      dfn[u] := num; L[u] := num; num := num+1;

     for each vertex w adjacent from u do
     {

       if(dfn[w] = 0) then
       {

         Art(w, u);   // w is unvisited
         L[u] := min(L[u], L[w]);
       }
       else if(w≠v) then L[u] := min(L[u], dfn[w]);
    }
  }
```

Analysis:

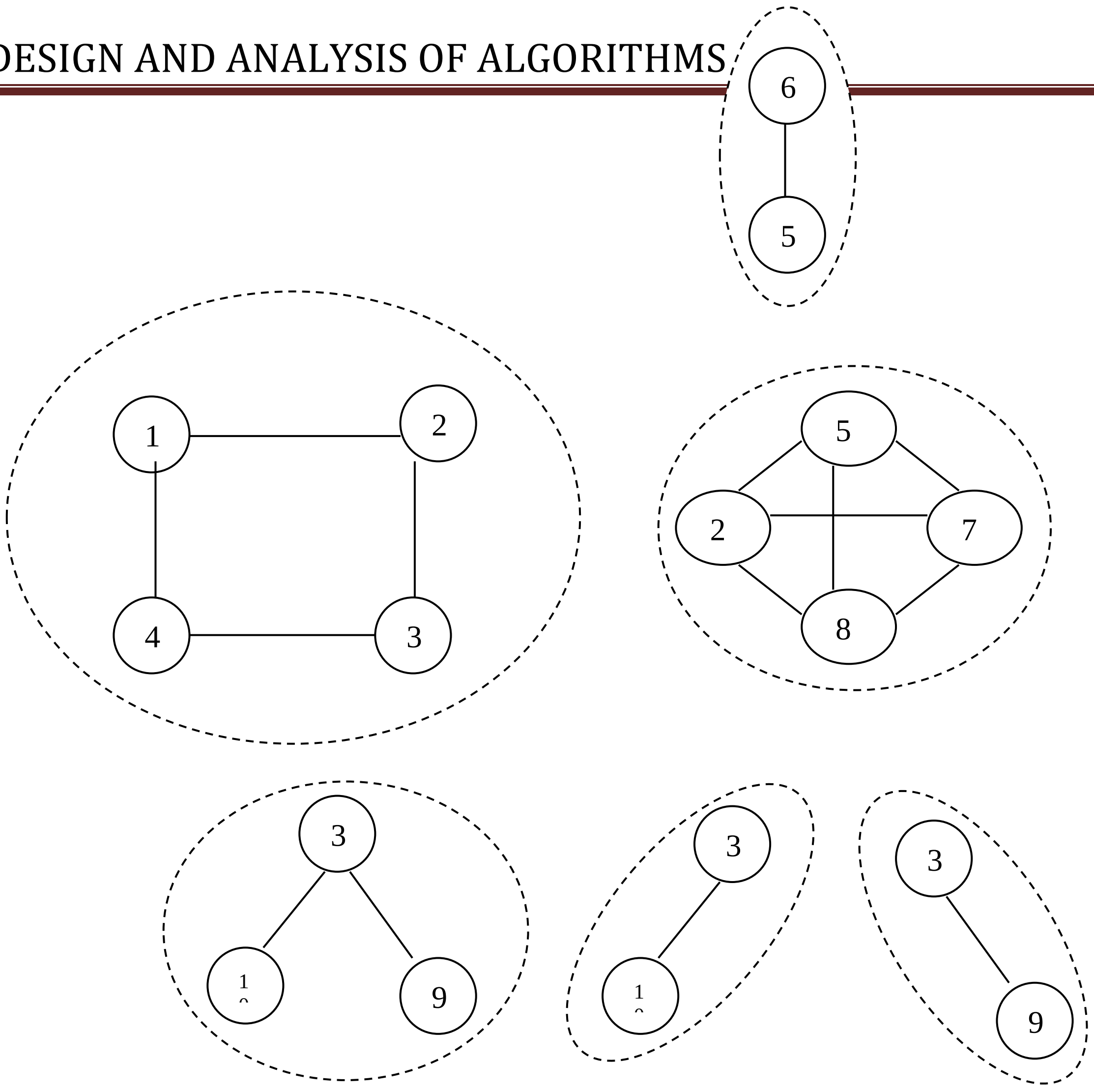
The Art has a complexity O(n+e) where e is the number of edges in G, the articulation points of G can be determined in O(n+e) time.

Identification of Bi-connected components

- A bi-connected graph G = (V, E) is a connected graph which has no articulation point.
- A bi-connected component of a graph G is maximal biconnected sub graph. That means it is not contained in any larger bi-connected sub graph of G.
- Some key observations can be made in regard to bi-connected components of graph.
  - i) Two different bi-connected components should not have any common edges.
  - ii) Two different bi-connected components can have common vertex.
  - iii) The common vertex which is attaching two (or more) bi-connected components must be an articulation point.



The articulation points are 2, 3, 5. Hence bi-connected components are



Algorithm for Bi-connected components

```
Algorithm BiComp(u, v)
// u is a start vertex for depth first search. v is its parent if any in the depth first spanning tree. It
//is assumed that the global array dfn is initialized to zero and that the global variable num is
//initialized to 1. n is the number of vertices in G.

{
  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
        {
                **if** (L[w] $\geq$ 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)));
                }
                BiComp(w, u);  // w is unvisited
                L[u] := min(L[u], L[w]);
        }
        **else if** (w$\neq$v) **then** L[u] := min(L[u], dfn[w]);
       }
    }
```

DYNAMIC PROGRAMMING

GENERAL METHOD

Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the ***principal of optimality***.

The ***principal of optimality*** states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Dynamic Programming Vs Greedy Method

1. The greedy method and dynamic programming algorithm both are the methods for obtaining optimum solution.
2. Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated.
3. The greedy method is a straight forward method for choosing the optimum choice or solution. It simply picks up the optimum solution without revising the previous solutions. Where as dynamic programming considers all possible sequences in order to obtain the optimum solution. Dynamic programming uses bottom up approach to obtain the final solution.
4. It is guaranteed that the dynamic programming will generate optimal solution using principle of optimality but in case of greedy algorithm there is no such guarantee.

Dynamic Programming Vs Divide and Conquer

1. In divide and conquer algorithm we take the problem and divide it into small sub problems. These sub problems are then solved independently. And finally all the solutions of sub problems are collected together to get the solution to the given problem. These sub problems are solved independently without considering the common sub instances of sub solutions. On the other hand in dynamic programming many decision sequences are generated and all the overlapping sub instances are considered. In divide and conquer the duplications in the sub solutions are neglected. And in dynamic computing the duplication in computing is avoided totally.
2. Hence dynamic programming is efficient than divide and conquer.
3. The divide and conquer uses top down approach while solving any problem where as dynamic programming uses the bottom up approach.
4. Divide and conquer splits its input at specific deterministic points usually in the middle. Dynamic programming splits its input at every possible split points rather than at a particular point. After trying all split points, it determines which split point is optimal.

It solves the problem using sub problem approach, but the sub problems are not independent. i.e. when sub problems share sub problems. A dynamic programming algorithm solves each sub problem just once and then saves its answer in a table, there by avoiding a work of re computing the answer every time the sub problem is encountered.
The development of this algorithm can be broken into a sequence of 4 steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution

Since recursion is involved it results in a recurrence relation. This relation can be solved by using two approaches: forward or backward.
In the forward approach, the formulation for decision x_i is made in terms of optimal decision sequences for $x_{i+1},\ldots,x_n$.
In the backward approach, the formulation for decision x_i is in terms of optimal decision sequences for $x_1,\ldots\ldots,x_{i-1}$
Thus in the forward approach -> "look" ahead on the decision sequence $x_1, x_2,\ldots,x_n$.
In the backward approach -> "look" backwards on the decision sequence $x_1, x_2,\ldots,x_n$

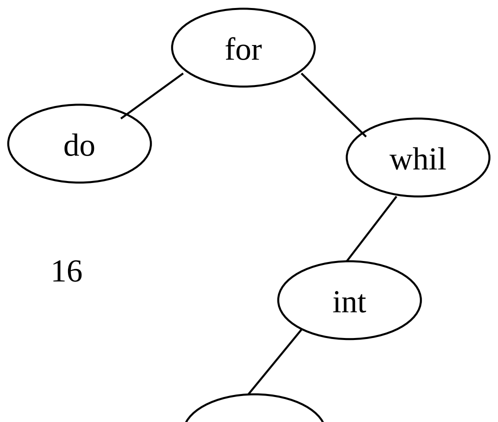
OPTIMAL BINARY SEARCH TREES

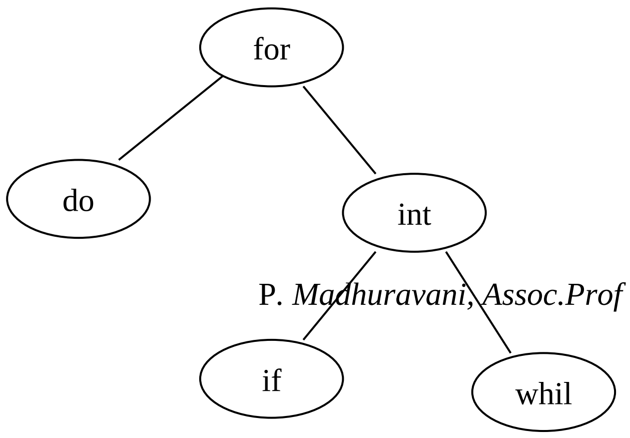
Definition: A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier and

i. all identifiers in the left sub tree of T are less than the identifiers in the root node T.
ii. all identifiersin the right subtree are greater than the identifier in the root node T.
iii. the left and right sub trees of T are also binary search trees.

Algorithm: Searching a binary search tree

```
procedure SEARCH(T, x, i)
i←T
while i≠0 do
case
: x<IDENT(i): i←LCHILD(i)
: x=IDENT(i): return
: x>IDENT(i): i←RCHILD(i)
endcase
repeat
end SEARCH
```





worst case = 4 comparisons worst case = 3 comparisons

(a) (b)

Given a fixed set of identifiers, wish to create a binary search tree organization. Assume that the given set of identifiers is $\{a_1, a_2, \ldots\ldots, a_n\}$ such that $a_1 < a_2 \ldots\ldots < a_n$

Let P(i) be the probability with which we search for an element a_i . Let Q(i) be the probability that the identifier x being searched for is such that $a_i < X < a_{i+1}$, $0 \le i \le n$ where $a_0 = -\infty$ and $a_{n+1} = +\infty$. Then $\sum_{0\le i\le n} Q(i)$ is the probability of an un successful search.

Hence

$$\sum_{1\le i\le n} P(i) + \sum_{0\le i\le n} Q(i) = 1$$

If a binary search tree represents n identifiers then there will be exactly n internal nodes and n+1 external nodes. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate. If a successful search terminates at an internal node at level l then the cost contribution from the internal node for a_i is

$a_i = p(i) * level(a_i)$

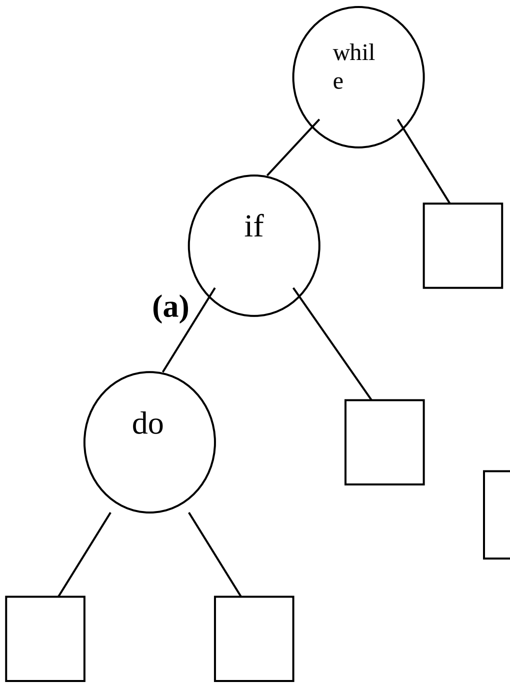
If it is an un successful search at level 'l' then the cost contribution is $Q(i)*(level(E_i)-1)$

Hence the expected cost of a binary search tree is

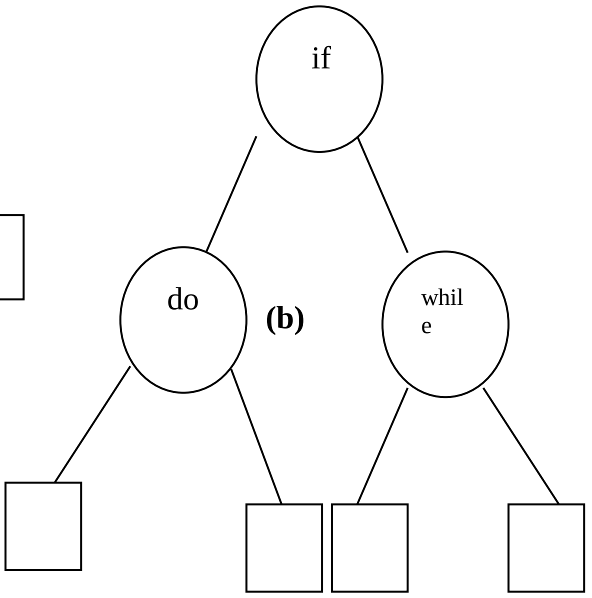
$$\sum_{1\le i\le n} P(i)*level(a_i) + \sum_{0\le i\le n} Q(i)*(level(E_i)-1)$$

Example:

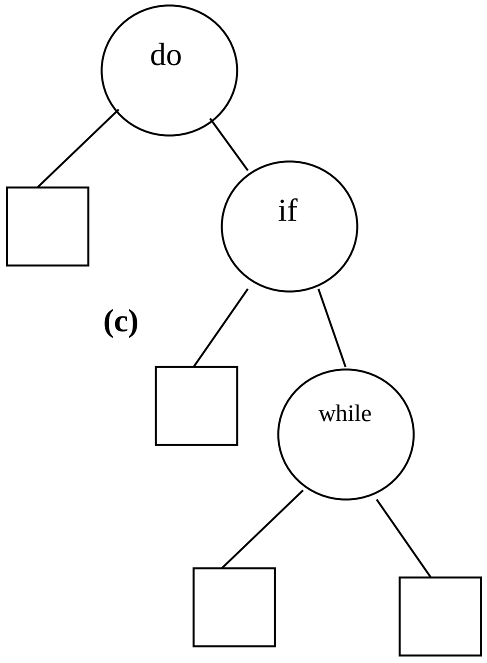
The possible binary search trees for the identifier set (a_1, a_2, a_3) = (do, if, stop) are:



(a)



(b)

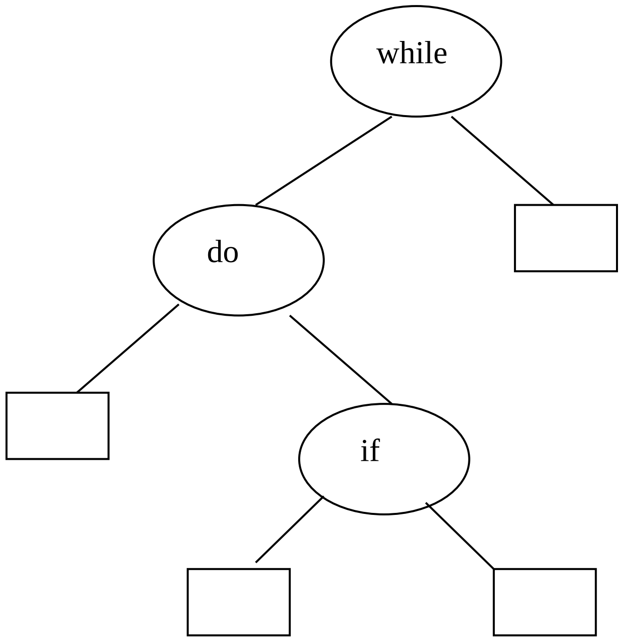


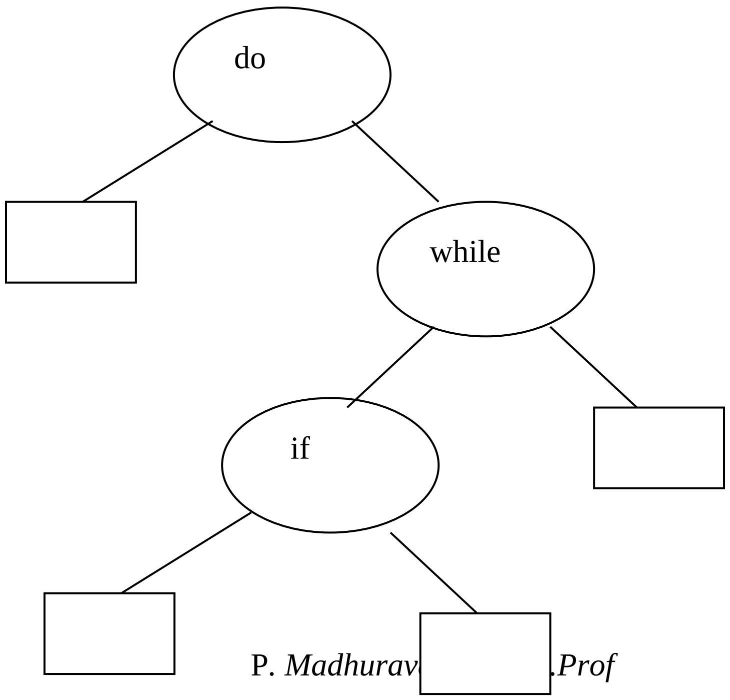
(c)

3

(d)

(e)





with equal probabilities P(i) = Q(j) = 1/7 for all i and j,

$$\text{cost(tree a)} = \underbrace{1*1/7 + 2*1/7 + 3*1/7}_{\text{Internal}} + \underbrace{1*1/7 + 2*1/7 + 3*1/7+3*1/7}_{\text{External}}$$

= 15/7

cost(tree b) = 1/7+2/7+2/7+2/7+2/7+2/7+2/7 = 13/7

cost(tree c) = 1/7+2/7+3/7+1/7+2/7+3/7+3/7 = 15/7

cost(tree d) = 1/7+2/7+3/7+1/7+2/7+3/7+3/7 = 15/7

cost(tree e) = 1/7+2/7+3/7+1/7+2/7+3/7+3/7 = 15/7

tree b is optimal.

with P(1) = 0.5 P(2) = 0.1 P(3) = 0.05

Q(0) = 0.15 Q(1) = 0.1 Q(2) = 0.05 Q(3) = 0.05

cost(tree a) = 0.05*1 + 0.1*2 + 0.5*3 + 0.15*3 + 0.1*3 + 0.05*2 + 0.05*1
= 2.65

cost(tree b) = 0.1*1 + 0.5*2 + 0.05*2 + 0.15*2 + 0.1*2 + 0.05*2 + 0.05*2
= 1.9

cost(tree c) = 0.5*1 + 0.1*2 + 0.05*3 + 0.15*1+ 0.1*2 + 0.05*3 + 0.05*3
= 1.5

cost(tree d) = 0.05*1 + 0.5*2 + 0.1*3 + 0.15*2 + 0.1*3 + 0.05*3 + 0.05*1
= 2.15

cost(tree e) = 0.5*1 + 0.05*2 + 0.1*3 + 0.15*1 + 0.1*3 + 0.05*3 + 0.05*2
= 1.6

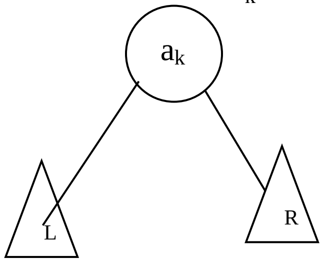
In order to apply dynamic programming to the problem of obtaining an optimal binary search tree we need to view the construction of such a tree as a sequence of decisions. A possible approach would be to make a decision as to which of the a_i's should be assigned to the root node of T. If we choose a_k then $a_{1,,}a_2$……,a_{k-1} and the external nodes for the classes $E_{0,<}E_1$……,E_{k-1} lie in the left sub tree of a_k. and a_{k+1}…….a_n and E_k…E_n lie in the right sub tree of $a_{k.}$

$$\text{cost}(L) = \sum_{1\le i\le k} P(i)*\text{level}(a_i) + \sum_{0\le i<k} Q(i)*(\text{level}(E_i)-1)$$

and

$$\text{cost}(R) = \sum_{k<i\le n} P(i)*\text{level}(a_i) + \sum_{k\le i\le n} Q(i)*(\text{level}(E_i)-1)$$

An optimal binary search tree with root a_k



Expected cost for entire tree is

$$\text{COST}(T) = P(k) + \underbrace{\text{cost}(L) + \text{cost}(R)}_{\text{cost for searching}} + \underbrace{W(0, k-1) + W(k, n)}_{\text{cost associated for constructing the tree}}$$

cost (L) = C(0, k-1)

cost(R) = c(k, n)

W(i, j) = P(j) + q(j) + W(i, j-1)

W(i, i) = q(i)

C(i, j) = 0

$$C(0, n) = \min_{1\le k\le n}\{ C(0, k-1) + C(k, n) + p(k) + W(0, k-1) + W(k, n)\}$$

General equation

$$C(i, j) = \min_{i<k\le j}\{ C(i, k-1) + C(k, j) + p(k) + W(1, k-1) + W(k, j)\}$$

$$= \min_{i<k\le j}\{ C(i, k-1) + C(k, j) + W(i, j)\}$$

W(i, j) = P(j) + q(j) + W(i, j-1)
r(i, j) = k
Record the root R(i, j) of each tree T_{ij} then an optimal binary search tree may be constructed from these R(i, j). R(i, j) is the value of k that minimizes the above equation.

Example:
Let n = 4 and (a_1, a_2, a_3, a_4) = (do, if, int, while). Let p (1:4) = (3, 3, 1, 1) q(0:4) = (2, 3, 1, 1, 1) construct the optimal binary search tree.

Initially, W(i, i) = q(i)
c(i, i) = 0
r(i, i) = 0
W(i, i+1) = q(i) + q(i+1) + p(i+1)
r(i, i+1) = i+1
c(i, i+1) = q(i) + q(i+1) + p(i+1)
w(i, j) = p(j) + q(j) + w(i, j-1)
r(i, j) = k

$$c(i,j) = \min_{i<k\leq j}\{ C(i, k-1) + C(k, j) + W(i, j)\}$$

Let i = 0
w(0,0) = q(0) = 2
when i=1 w(1,1) =q(1)= 3
when i=2 w(2,2) =q(2)= 1
when i=3 w(3,3) =q(3)= 1
when i=4 w(4,4) = q(4)=1

when i=0 and j-i = 1 then
w(0,1) = q(0)+q(1)+p(1)=2+3+3=8

when i=1 and j-i = 2 then
w(1,2) = q(1)+q(2)+p(2)=3+1+3=7

when i=2 and j-i = 3 then
w(2,3) = q(2)+q(3)+p(3)=1+1+1=3

when i=3 and j-i = 4 then
w(3,4) = q(3)+q(4)+p(4)= 1+1+1=3

now, when i=0 and j-i = 2 then
w(i, j) = p(j) + q(j) + w(i, j-1)
w(0,2)=w(0,1)+p(2)+q(2)=8+3+1=12

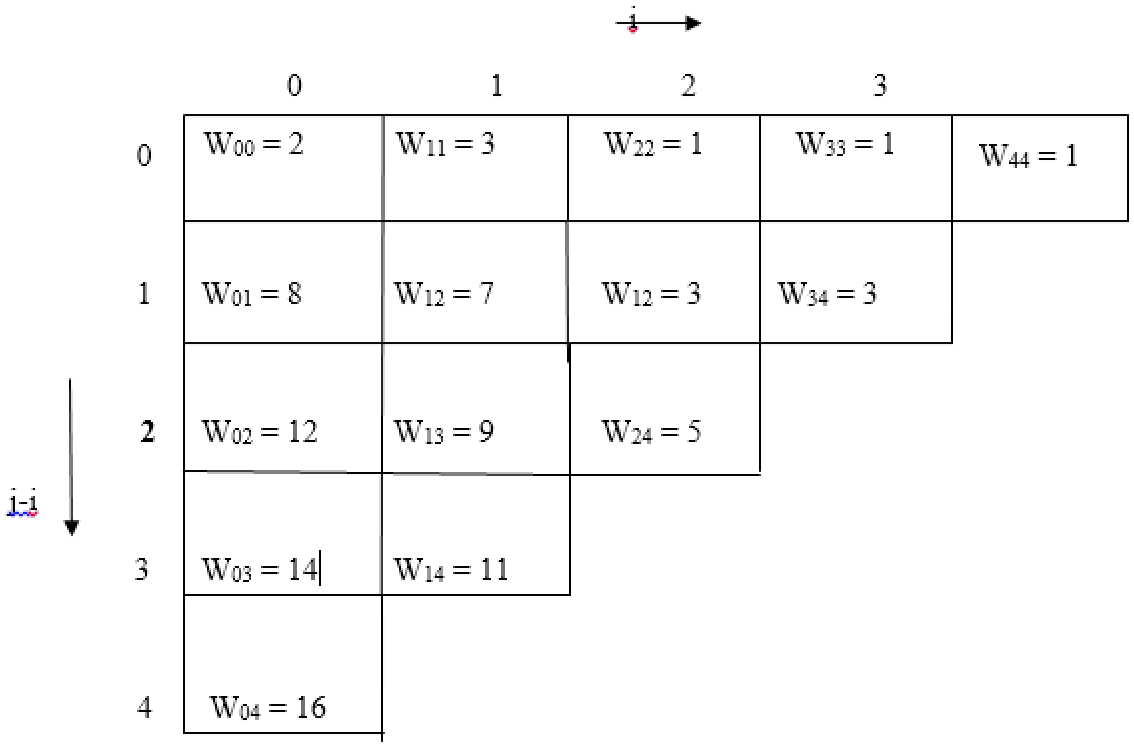
when i=1 and j-i = 2 then
w(1,3)=w(1,2)+p(3)+q(3)=7+1+1=9
when i=2 and j-i = 2 then
w(2,4)=w(2,3)+p(4)+q(4)=3+1+1=5

when i=0 and j-i = 3 then
w(0,3)=w(0,2)+p(3)+q(3)=12+1+1=14

when i=1 and j-i = 3 then
w(1,4)=w(1,3)+p(4)+q(4)=9+1+1=11

when i=0 and j-i = 4 then

w(0,4)=w(0,3)+p(4)+q(4)=14+1+1=16
The table for W can be represented as



	0	1	2	3	
0	$W_{00} = 2$	$W_{11} = 3$	$W_{22} = 1$	$W_{33} = 1$	$W_{44} = 1$
1	$W_{01} = 8$	$W_{12} = 7$	$W_{12} = 3$	$W_{34} = 3$	
2	$W_{02} = 12$	$W_{13} = 9$	$W_{24} = 5$		
3	$W_{03} = 14$	$W_{14} = 11$			
4	$W_{04} = 16$				

Now compute C and r
C(i,i) = 0 and r(i,i) = 0

C(0,0) = 0	C(1, 1) = 0	C(2, 2) = 0	C(3, 3) = 0	C(4, 4) = 0
r(0,0) = 0	r(1, 1) = 0	r(2, 2) = 0	r(3, 3) = 0	r(4, 4) = 0

Similarly, C(i, i+1) = q(i) + q(i+1) + p(i+1) and r(i, i+1) = i+1
when i=0
C(0, 1) = q(0) + q(1) + p(1) = 2+3+3=8
r(0, 1) =1

when i=1
C(1, 2) = q(1) + q(2) + p(2) = 3+1+3=7
r(1, 2) =2

when i=2
C(2, 3) = q(2) + q(3) + p(3) = 1+1+1=3
r(2, 3) =3

when i=3
C(3, 4) = q(3) + q(4) + p(4) = 1+1+1=3

r(3, 4) =4

Now we will compute C(i, j) and r(i, j) for $j-i \geq 2$

as $C(i, j) = \min_{i<k\leq j}\{ C(i, k-1) + C(k, j) + W(i, j)\}$, hence find k

Compute C(i, j) = C(i, k-1) + C(k, j)

For C(0,2) we have i=0, and j=2. for r(i, j-1) to r(i+1, j) i.e for r(0,1) to r(1,2) compute minimum value of C(i, j)

Let r(0, 1) = 1 and r(1, 2) = 2then for k=1 compute C(i, j) and for k=2 compute C(i, j) and will pick up minimum value of C(i, j) only.

for k=1, i=0 j=2

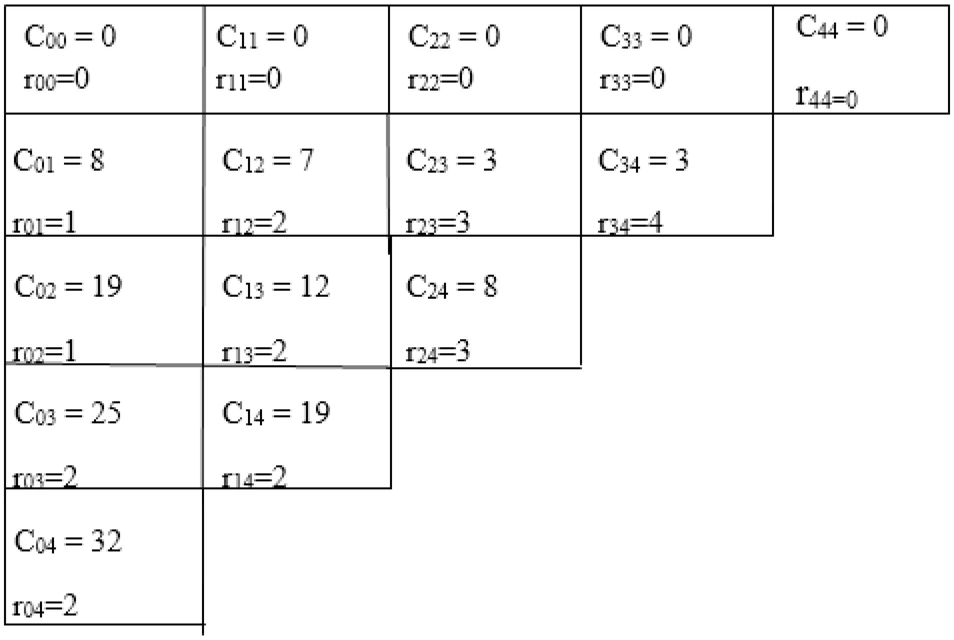
C(0,2) = C(0,0)+C(1,2)=0+7=7

for k=2, i=0 j=2

C(0,2) = C(0,1)+C(2,2)=8+0=8 i.e. for k=1 C(i,j) is minimum

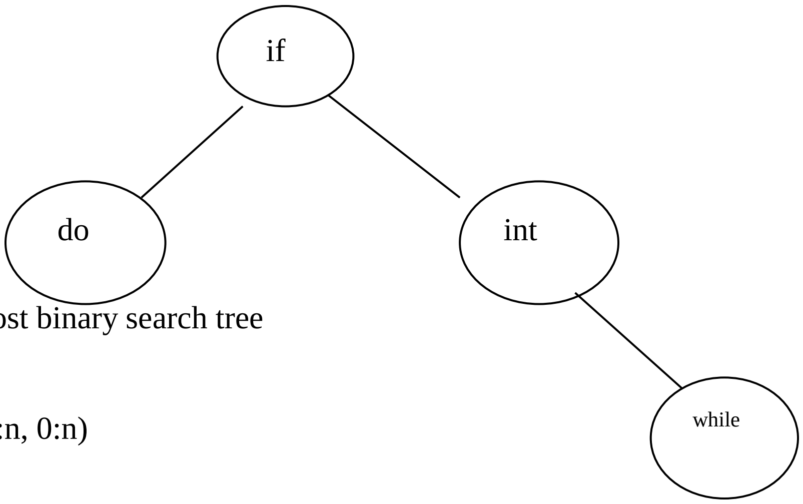
Hence r(i, j) = r(0,2)=k=1

Now C(0,2) = min(C(i,k-1)+C(k,j))+W(i,j)=7+W(0,2)=7+12=19



$C_{00} = 0$ $r_{00}=0$	$C_{11} = 0$ $r_{11}=0$	$C_{22} = 0$ $r_{22}=0$	$C_{33} = 0$ $r_{33}=0$	$C_{44} = 0$ $r_{44}=0$
$C_{01} = 8$ $r_{01}=1$	$C_{12} = 7$ $r_{12}=2$	$C_{23} = 3$ $r_{23}=3$	$C_{34} = 3$ $r_{34}=4$	
$C_{02} = 19$ $r_{02}=1$	$C_{13} = 12$ $r_{13}=2$	$C_{24} = 8$ $r_{24}=3$		
$C_{03} = 25$ $r_{03}=2$	$C_{14} = 19$ $r_{14}=2$			
$C_{04} = 32$ $r_{04}=2$				

The tree T_{04} has a root r_{04}. The value of r_{04} is 2. From (a_1, a_2, a_3,a_4) = {do, if, int, while) a_2 becomes the root node. r(i, j) = k, r_{04} =2
then $r_{i,k-1}$ becomes left child and r_{kj} becomes right child. i.e r_{01} left child and r_{24} right child of r_{04}. Here r_{01} =1 so a_1 becomes left child of a_2 and r_{24} =3 so a_3 becomes right child of a_2.
For node r_{24} i=2 and j=4, k=3, hence left child of it is r_{ik-1} = r_{22}=0 i.e left child of r_{24} = a_3 is empty. The right child of r_{24} is r_{34}=4. Hence a_4 becomes right child of a_3. Thus all n=4 nodes are used to build a tree. The optimal binary search tree is



Algorithm: Finding a minimum cost binary search tree

```
procedure OBST(P, Q, n)
real P(n), Q(0:n), C(0:n, 0:n), W(0:n, 0:n)
integer R(0:n, 0:n)
for i← 0 to n-1 do
(W(i, i), R(i, i), C(i, i)) ← (Q(i), 0, 0)
(W(i, i+1), R(i, i+1), C(i, i+1)) ← (Q(i) + Q(i+1) + P(i+1), i+1, Q(i) + Q(i+1) +P(i+1))

repeat

(W(n, n), R(n, n), C(n, n)) ← (Q(n), 0, 0)

for m← 2 to n do
for i← 0 to n-m do
j← i+m
W(i, j) ← W(i, j-1) +P(j) + Q(j)

k← a value of l in the range R(i, j-1)≤l≤R(i+1)
   that minimizes {C(i, l-1) + C(l, j)}

C(i, j) ← W(i, j) + C(i, k-1) + C(k, j)

R(i, j) ← k
repeat

repeat

end OBST
```

Analysis:

The total time for all C(i, j) with j-i = m is therefore $O(nm-m^2)$
The total time to evaluate all the C(i, j) and R(i, j) is therefore
$$\sum_{1\le m\le n} (nm-m^2) = O(n^3)$$

0/1 KNAPSACK

The Knapsack problem can be stated as follows-

- Given n objects and a knapsack or bag.
- Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0\le x_i\le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, the total weight of all chosen objects to be at most m.

The problem can be stated as

$$\text{maximize} \sum_{n}^{1} p_i x_i \quad \text{———} \quad (1)$$

$$\text{subject to} \quad \sum_{n}^{1} w_i x_i \le m \quad \text{———} \quad (2)$$

$$\text{and } 0\le x_i\le 1,\ 1\le i\le n. \quad \text{———} \quad (3)$$

The profits and weights are positive numbers.
A feasible solution (filling) is any set $(x_1,\ldots x_n)$ satisfying eqn (2) and (3). An optimum solution is a feasible solution for which eqn (1) is maximized.

To solve this problem using Dynamic Programming method we will perform following steps:
A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1,\ldots x_n$. A decision on variable xi involves determining which of the values 0 or 1 is to assigned to it. Let the decisions x_i are made in the order $x_n, x_{n-1}\ldots x_1$
Following a decision on x_n, there are two possible states:

1. The capacity remaining in the knapsack is m and no profit has accrued.
2. The capacity remaining is $m-w_n$ and a profit of p_n has accrued.

The remaining decisions $x_{n-1}\ldots x_1$ must be optimal w.r.t the problem state resulting from the decision on x_n. Otherwise $x_n\ldots x_1$ will not be optimal. Hence the principle of optimality holds.
Let $f_j(y)$ be the value of an optimal solution. Since the principle of optimality holds, we obtain
$f_n(m) = \max\{f_{n-1}(m), f_{n-1}(m-w_n)+p_n\}$

for arbitrary $f_i(y)$, $i > 0$ the above eqn generalizes to, $f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y-w_n)+p_i\}$ this eqn can be solved by,

$f_0(y) = 0 \quad \forall y$

$f_i(y) = -\alpha$, $y < 0$

⇨ $f_i(y)$ is an ascending step function.

$f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$

$f_i(y) = -\alpha$, $y<y_1$

$f_i(y) = f(y_k)$, $y \geq y_k;$ and

$f_i(y) = f_i(y_j)$, $y_j \leq y < y_{j+1}$

So compute only $f_i(y_j)$, $1 \leq j \leq k$

Let the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$.

Each member of S^i is a pair (P, W) where $P = f_i(y_j)$ and $W = y_j$

Initially,
$S^0 = \{(0, 0)\}$
$S^i_1 = \{(P, W) \mid (P-p_i, W-w_i) € S^i\}$
S^{i+1}_1 can be computed by merging S^i and S^i_1

⇨ S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded. Discarding or purging rules as this are also know as **dominance rules.** In purging rule basically the dominated tuples gets purged. In short, remove the pair with less profit and more weight.

Example:

Consider the knapsack instance M = 8, and n = 4

$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$

$(p_1, p_2, p_3, p_4) = (1, 2, 5, 6)$

Initially, $S^0 = \{(0, 0)\}$

$S^0_1 = \{(1, 2)\}$

$S^1 = \{$merge S^0 and $S^0_1\}$
$= \{(0, 0)\ (1, 2)\}$

$S^1_1 = \{$select next (P, W) pair and add it with $S^1\}$
$= \{(2, 3)\ (2+0, 3+0)\ (2+1, 3+2)\} = \{(2, 3)\ (3, 5)\}$ repetition of (2, 3) is avoided.

$S^2 =$ Merge S^1 and S^1_1
$= \{(0, 0)\ (1, 2)\ (2, 3)\ (3, 5)\}$

$S^2_1 = \{$select next (P, W) pair and add it with $S^2\}$
$= \{(0+5, 0+4)\ (1+5, 2+4)\ (2+5, 3+4)\ (3+5, 5+4)\}$
$= \{(5, 4)\ (6, 6)\ (7, 7)\ (8, 9)\}$

S^3 = Merge S^2 and S^2_1
= {(0, 0) (1, 2) (2, 3) (5, 4) (6,6) (7, 7) (8, 9)}
Note that the pair (3, 5) is purged from S^3. This is because, let us assume (P_j, W_j) = (3, 5) and (P_k, W_k) = (5, 4) here $P_j \le P_k$ and $W_j > W_k$ is true hence we will eliminate pair (P_j, W_j) i.e (3, 5) from S^3.
S^3_1 = {select next (P, W) pair and add it with S^3}
= {(0+6, 0+5) (1+6, 2+5) (2+6, 3+5) (5+6, 4+5) (6+6,6+5) (7+6, 7+5) (8+6, 9+5)}
S^4 = {(0, 0) (1, 2) (2, 3) (5, 4) (6, 6) (7, 7) (8, 9) (6, 5) (8, 8) (11, 9) (12, 11) (13, 12) (14, 14)}
Now M = 8, we get pair (8, 8) in S^4, hence $x_4 = 1$
Now select next object $(P-P_4)$ and $(W-W_4)$
i.e (8-6) and (8-5)
i.e (2, 3)
pair (2, 3) € S^2 Hence set $x_2 = 1$ So we get the final solution as (0, 1, 0, 1).

Algorithm:
PW = record {float p; float w}
Algorithm DKnap (p, w, x, n, m)

```
{
        //pair[] is an array of PW's
        b[0] := 1;
        pair[1].p := pair[1].w:=0; //S^0
        t:=1; h:=1; //start and end of S^0
        b[1] : = next :=2; //next free spot in pair[]
        for i:=1 to n-1 do
        {
                //Generate S^i
                k:=t;
                u:=Largest(pair, w, t, h, i, m);
                for j:= t to u do
                {
                // Generate S^{i-1}_1 and merge
                pp := pair[j].p+p[i];
                ww := pair[j].w+w[i];
                //(pp, ww) is the next element in S_1^{i-1}
                while ((k≤h) and (pair[k].w≤ww)) do
                {
                        pair[next].p := pair[k].p;
                        pair[next].w := pair[k].w;
                        next := next+1; k:=k+1;
                }
                if ((k≤h) and (pair[k].w = ww)) then
                {
                if pp < pair[k].p then pp:=pair[k].p;
                k:=k+1;
                }
                if pp > pair[next-1].p then
                {
                pair[next].p :=pp; pair[next].w := ww;
                next := next+1;
                }
                while ((k≤h) and (pair[k].p ≤ pair[next-1].p)) do
                k : = k+1;
        }
        //Merge in remaining terms from S^{i-1}
        while (k≤h) do
        {
                pair[next].p := pair[k].p;
                pair[next].w := pair[k].w;
                next := next+1;
                k := k+1;
        }
        //Initialize for s^{i+1}
        t:=h+1;
        h:=next-1;
        b[i+1} := next;
        }
        TraceBack(p, w, pair, x, m, n);
  }
```

Analysis:

The worst case time is $O(2^{n/2})$

ALL PAIRS SHORTEST PATH PROBLEM

Let G = (V, E) be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that *cost*(i, i) = 0, $1 \le i \le n$. Then *cost*(i, j) is the length (or cost) of edge (i, j) if (i, j)€E(G) and *cost*(i, j) = α if $i \neq j$ and (i, j) $\not\in$ E(G). The all pairs shortest path problem is to determine a matrix A such that A(i, j) is the length of a shortest path from i to j.

We will apply dynamic programming to solve the all pairs shortest path

Step 1: We will decompose the given problem into sub problems. Let, A^k (i, j) be the length of shortest path from node i to j such that the label for evry intermediate node will be ≤k. We will compute A^k for k=1 … n for n nodes.

Step 2: For solving all pair shortest path, the principle of optimality is used. That means any sub path of shortest path is a shortest path between the end nodes. Divide the paths from node i to j for every intermediate node k. Then there arises two cases:

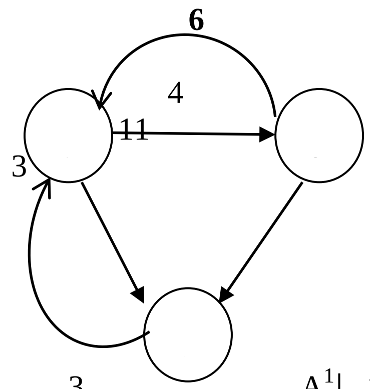
i) Path going from i to j via k.
ii) Path which is not going via k. Select only shortest path from two cases.

Step 3: The shortest path can be computed using bottom up computation method. Following is recursion method.

Initially A^0(i, j) = cost(i, j)

A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$
If it does not, then no intermediate vertex has index greater than k-1 Hence , $A^k(i, j) = A^{k-1}(i, j)$
combining, , $A^k(i, j) = \min \{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}$, $k \geq 1$

Example:



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	α	0

a) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

b) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

c)A^2

A^2	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

d) A^3

Algorithm:

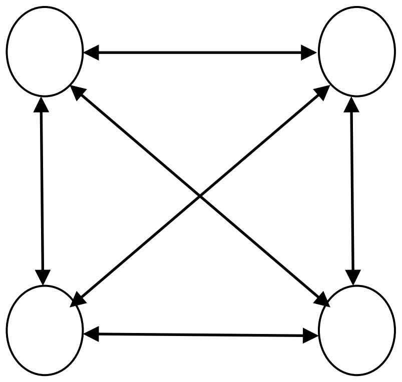
```
Algorithm AllPaths(cost, A, n)
//cost [1:n, 1:n] is the cost adjacency matrix of a graph with n vertices; A[i, j] is the cost of a
//shortest path from vertex i to vertex j. cost[i, i] = (0, 0) for 1≤i≤n.
{
        for i := 1 to n do
                for j := 1 to n do
                A[i, j] := cost[i, j];
        for k := 1 to n do
                for i := 1 to n do
                        for j := 1 to n do
                        A[i, j] := min (A[i, j], A[i, k] + A[k, j]);
}
```

Analysis:
Time complexity : $O(n^3)$

TRAVELLING SALES PERSON PROBLEM

Let G be directed graph denoted by (V, E) where V denotes set of vertices and E denotes set of edges. The edges are given along with their cost C_{ij}. The cost $C_{ij} > 0$ for all i and j. If there is no edge between i and j then $C_{ij} = \alpha$.
A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour. The travelling sales person problem is to find the tour of minimum cost.
A tour is a simple path that starts and ends at vertex 1. Every tour consists of an edge (1, k) for some k€V-{1} and a path from vertex k to vertex 1. This path goes through each vertex in V-{1, k} exactly once. If the tour is optimal then the path from k to 1 must be the shortest path going through all the vertices in S and terminating at vertex 1.
The function g(1, V-{1}) is the length of an optimal salesperson tour. From the principle of optimality it follows that
$g(1, V-\{1\}) = \min_{2\leq k\leq n}\{c_{1k}+g(k, V-\{1, k\})\}$
Generalizing we obtain, (for i€S)
$g(i, S) = \min_{j€s}\{c_{ij} + g(j, S-\{j\})\}$

Example:
Consider the directed graph and edge length matrix c as



$$\begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

Step 1:
Let S = Φ then
$g(2, \Phi) = c_{21} = 5$
$g(3, \Phi) = c_{31} = 6$
$g(4, \Phi) = c_{41} = 8$

Step 2: Now compute g(i, S) with S=1
$g(2, \{3\}) = c_{23} + g(3, \Phi) = 9 + 6 = 15$
$g(2, \{4\}) = c_{24} + g(4, \Phi) = 10 + 8 = 18$
$g(3, \{2\}) = c_{32} + g(2, \Phi) = 13 + 5 = 18$
$g(3, \{4\}) = c_{34} + g(4, \Phi) = 12 + 8 = 20$
$g(4, \{2\}) = c_{42} + g(2, \Phi) = 8 + 5 = 13$
$g(4, \{3\}) = c_{43} + g(3, \Phi) = 9 + 6 = 15$

Step 3: Now compute g(i, S) with S=2
$g(2, \{3,4\}) = \min \{c_{23} + g(3,\{4\}), c_{24}+g(4, \{3\})\} = \min \{9+20, 10+15\} = 25$
$g(3, \{2,4\}) = \min \{c_{32} + g(2,\{4\}), c_{34}+g(4, \{2\})\} = \min \{13+18, 12+13\} = 25$
$g(4, \{2,3\}) = \min \{c_{42} + g(2,\{3\}), c_{43}+g(3, \{2\})\} = \min \{8+15, 9+18\} = 23$

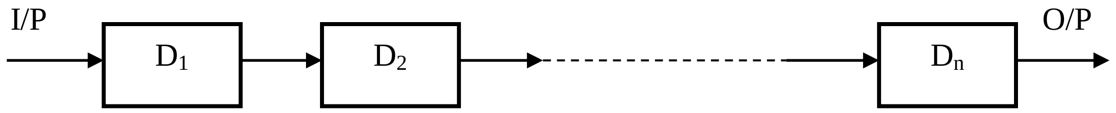
Step 4: Finally
$g(1, \{2,3,4\}) = \min \{c_{12} + g(2,\{3,4\}), c_{13}+g(3, \{2,4\}), c_{14}+g(4, \{2,3\})\} = \min \{35, 40, 43\} = 35$
The optimal tour of the graph has length 35. Consider step 4, from vertex 1 we obtain the optimum path as c_{12}. Hence select vertex 2. Now consider step 3, in which vertex 2 we obtain optimum cost from c_{24}. Hence select vertex 4. Now in step 2 we get remaining vertex 3 as c_{43} is optimum. Hence the optimal tour is 1-2-4-3-1

Analysis:
To find an optimal tour it requires $\Theta(n^2 2^n)$ time, and the drawback of this dynamic programming solution is the space needed, $O(n2^n)$.

RELIABILITY DESIGN

In this section we will discuss the problem based on multiplicative optimization function. We will consider such a system in which many devices are connected in series, as

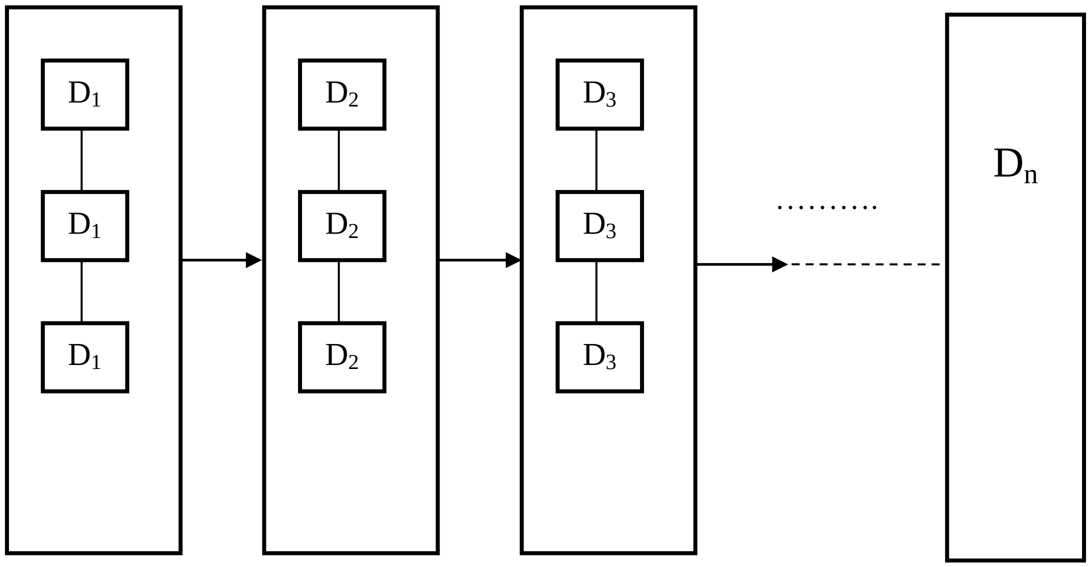
Devices Connected in series



When devices are connected together then it is a necessity that each device should work properly. The probability that device ‘i’ will work properly is called reliability of that device.

Let r_i be the reliability of device D_i then reliability of entire system is $\prod r_i$. It may happen that even if reliability of individual device is very good but reliability of entire system may not be good. Hence to obtain the good performance from entire system we can duplicate individual devices and can connect them in a series. To do so, we will attach switching circuits. The job of switching circuit is to determine which device in a group is working properly.

Multiple devices connected together



If stage i contains m_i be the copies of device D_i then $(1-r_i)^{mi}$ be the probability that all m_i have a malfunction. Hence the stage reliability is $1-(1-r_i)^{mi}$. We will denote reliability of stage i by $\Phi_i(m_i)$
Hence, $\Phi_i(m_i) = 1-(1-r_i)^{mi}$
i – stage no. r_i = reliability of D_i m_i = no of devices in stage i.
In reliability design problem we expect to get maximize reliability using device duplication. That means,
maximize $\prod_{1\le i\le n} \Phi_i(m_i)$
subject to $\sum_{1\le i\le n} C_i m_i \le C$ where C_i is the cost of each device and C is the maximum allowable cost of the system.
$1<=mi<=u_i$ and integer $1\le i\le n$
The upper bound u_i on the cost can be determined as:

$$u_i = \left(\left\lfloor C+C_i - \sum^{n}_{1} C_j \right\rfloor \Big/ C_i \right)$$

The dynamic programming solution S^i consist of tuples of the form (f, x) where $f=f_i(x)$. The $f_i(x)$ can be computed as

$$\left. \begin{array}{l} f_i(x) = \max_{1\le mi\le ui} \{ \Phi_i(m_i)F_{i-1}(x-C_i m_i) \\ \text{Initially } f_0(x) = 1 \end{array} \right\}$$

The dominance rule: (r_1x_1) dominates (r_2x_2) iff $r_1>r_2$ and $x_1<x_2$. The dominated tuples can be discarded from S^i.

Example:
Design a three stage system with device types D_1, D_2, D_3. The costs are Rs. 30, Rs.15, Rs. 20 respectively. The cost of the system is to be no more than Rs. 105. The reliability of each device type is 0.9, 0.8 and 0.5 respectively.
Solution: Let us compute u_1, u_2, and u_3

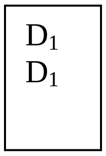
	D_1	D_2	D_3
cost	30	15	20
reliability	0.9	0.8	0.5

$u_1 = (105+30-(30+15+20))/30 = 2.33\approx 2$
$u_2 = (105+15-(30+15+20))/20 = 3$
$u_3 = (105+20-(30+15+20))/20 = 3$

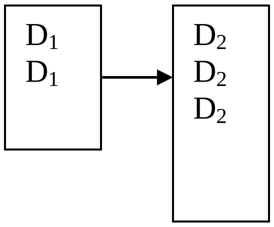
Each pair in this is represented by (r, x) where as r is reliability, x is cost.
Initially $S^0 = (1, 0)$ $1\le m_i\le u_i$ i.e.. $1\le m_1\le 2$
Calculate S^1_1 and S^1_2
$\mathbf{S^1_1}$
$\Phi_1(m_1) = 1 – (1 - r_1)^1 = 1 – (1-0.9) = 0.9$
First take one device D_1, for calculating multiply the reliability with 1 and cost is added to 0.

S^1_1 = (0.9x1, 30+0) = (0.9, 30)

Next calculate $\Phi_1(m_1)$ for $S^1_{2.}$ $m_1 = 2$
$\Phi_1(m_1) = 1-(1-0.9)^2 = 0.99$
$c_i m_i$ = 30X2 = 60
S^1_2 = (0.99, 60)
Combine two devices in parallel and merge two sets.
S^1 = { (0.9, 30) (0.99, 60)}



For device 2 calculate $S^2_{1,} S^2_{2,} S^2_3$
S^2_1 :
j = 1 since $1 \le m_2 \le 3$ i = 2
$\Phi_2(m_2) = 1-(1-r_2)^m{}_2$ = 1- (1-0.8) = 0.8
S^2_1 = { (0.9 X 0.8, 30+15), (0.99X0.8, 60+15)} = { (0.72, 45) (0.792, 75)}
j = 2 i = 2 m2=2
$\Phi_2(m_2) = 1-(1-r_2)^m{}_2 = 1- (1-0.8)^2$ = 0.96 cost = 2X15 = 30
S^2_2 = { (0.9X0.96, 30+30) (0.99X0.96, 60+30)} = {(0.864, 60) (0.9504, 90)}
j = 3 i = 2 m2=3
$\Phi_2(m_2) = 1-(1-r_2)^m{}_2 = 1- (1-0.8)^3$ = 0.992 cost = 3X15 = 45
S^2_3 = {(0.9 X 0.992, 30+45) (0.99 X 0.992, 60+45)}
= {(0.8928, 75), (0.98208, 105)}
Merge three devices in parallel



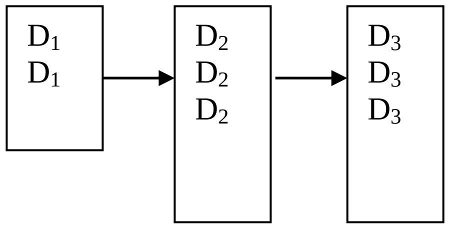
S^2 = { S^2_1 S^2_2 S^2_3} = {(0.72, 45) (0.864, 60) (0.8928, 75) (0.98208, 105)} Here (0.792, 75) is dominated by (0.864, 60) hence we will eliminate (0.792, 75) and (0.9504, 90)

Similarly $S^3_{1,} S^3_{2,} S^3_3$ can be calculated from S^2 as follows:
j = 1 since $1 \le m_3 \le 3$ i = 3
$\Phi_3(m_3) = 1-(1-r_2)^m{}_3$ = 1- (1-0.5) = 0.5 cost = 20

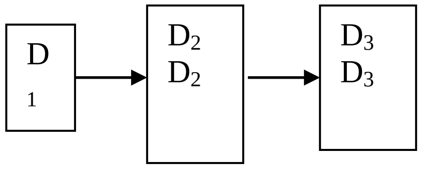
S^3_1 = {(0.72X0.5, 45+20) (0.864X0.5, 60+20), (0.8928X0.5, 75+20) ($\underbrace{\text{0.98208X0.5, 105+20}}_{\text{Cost is more than C so discard this tuple.}}$)}
= {(0.36, 65) (0.432, 80) (0.4464, 95)}

S^3_2
j = 2 m_3=2 i = 3
$\Phi_3(m_3) = 1-(1-r_2)^{m_3} = 1- (1-0.5)^2 = 0.75$ cost = 2X20 = 40
S^3_2 = {(0.72X0.75, 40+45) (0.864X0.75, 40+60)}={(0.54, 85) (0.648, 100)}

S^3_3
j = 3 m_3=3 i = 3
$\Phi_3(m_3) = 1-(1-r_2)^{m_3} = 1- (1-0.5)^3 = 0.875$ cost = 3X20 = 60
S^3_3 = {(0.72X0.875, 45+60)} = {(0.63, 105)}
We will combine above sets to obtain S^3



S^3 = {(0.36, 65) (0.432, 80) (0.4464, 95) (0.54, 85) (0.648, 100) (0.63, 105)}
Apply dominance rule"
S^3 = {(0.36, 65) (0.432, 80) (0.4464, 95) (0.54, 85) (0.648, 100) }
Hence the best reliability design will be with reliability 0.648 and cost 100. It comes from S^3_2
So m_3 = 2. m_2 = 2 because S^3_2 comes from S^2_2. next m_1 = 1


D_1
D_2
D_2
D_3
D_3