DESIGN AND ANALYSIS OF ALGORITHMS

Remember this: “No problem is too tough if u spend enuf time on it”

-A great man

Question:

$a^n + b^n = c^n$

find a,b,c

Answer: Etho ingay irukku!!

$a^n + b^n = c^n$

PRESENTED BY ARVIND KRISHNAA J

UNIT –V SYLLABUS

<u>CONTENTS OF THE UNIT</u>

- GRAPH TRAVERSALS
- CONNECTED COMPONENTS
- SPANNING TREES
- BI-CONNECTED COMPONENTS
- BRANCH AND BOUND:GENERAL METHODS(FIFO & LC)
- 0/1 KNAPSACK PROBLEM
- INTRODUCTION TO NP-HARD AND NPCOMPLETENESS

PRESENTED BY ARVIND KRISHNAA J

GRAPH TRAVERSALS

- AIM:

The main aim of any graph traversal algorithm is for a given graph G=(V,E) is to determine whether there exists a path starting from a vertex *v to a vertex* u

GRAPH TRAVERSALS

THERE ARE TWO DIFFERENT GRAPH TRAVERSALS WHICH WE WILL BE DEALING WITH

- BREADTH FIRST TRAVERSAL(BFS)
- DEPTH FIRST TRAVERSAL(DFS)

BREADTH FIRST SEARCH

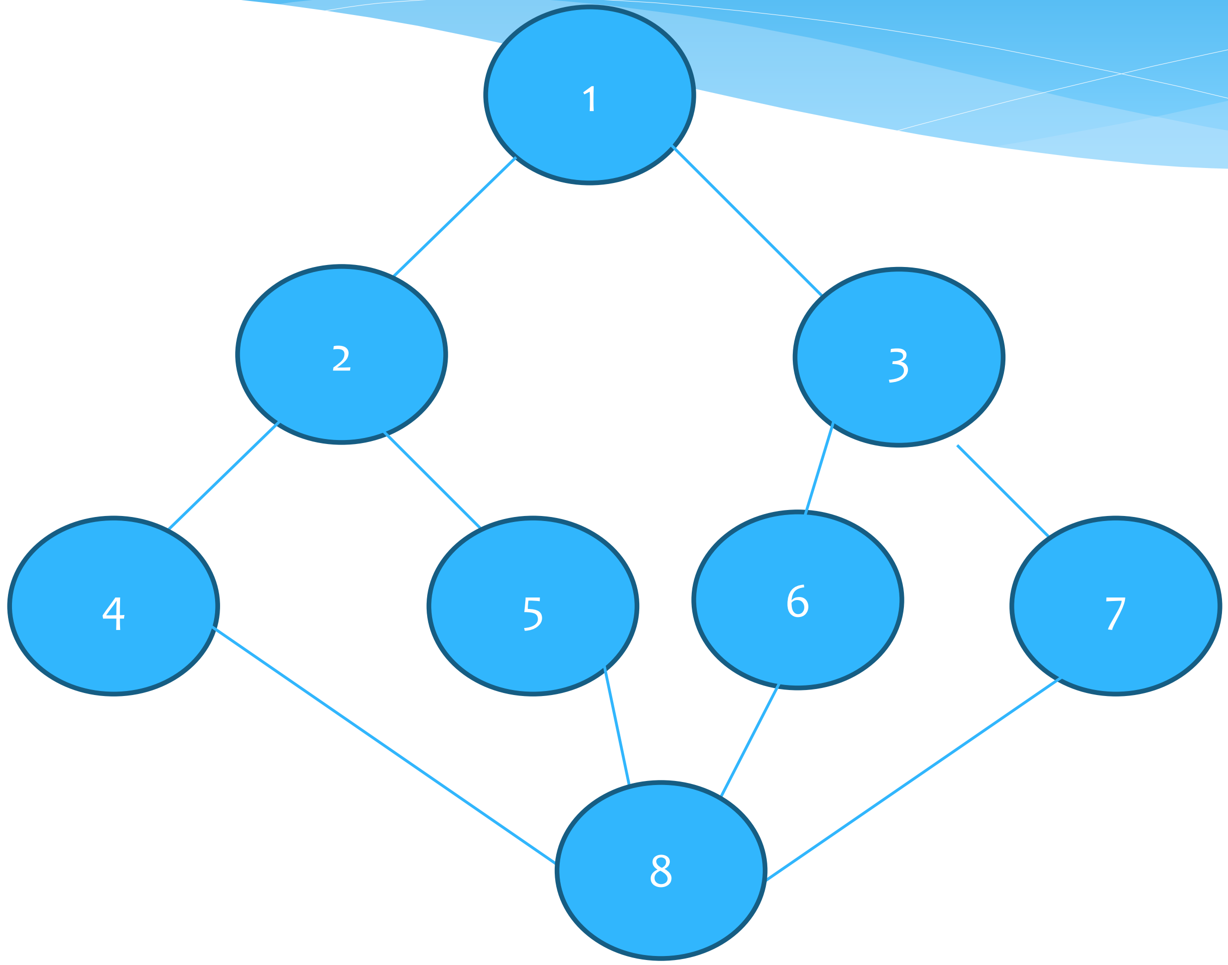
<u>BASIC IDEA:</u>

* **Visits graph vertices by moving across to all the neighbors of last visited vertex** b
* **BFS uses a queue**
* **a vertex is inserted into queue when it is reached for the first time, and is marked as visited.**
* **a vertex is removed from the queue, when it identifies all unvisited vertices that are adjacent to the vertex** b

ALGORITHM FOR BFS

```
void BFS(int v)
{
//v being a starting vertex
int u=v;
Queue q[SIZE];
visited[v]=1;
do
{
for all vertices w adjacent to to u
{
        if(visited[w]==0)//w is unvisited
        {
                q.AddQ(w);
                visited[w]=1;
        }
}
if(Q.Empty())  return;
q.Delete(u);
}while(1);
}
```

AN EXAMPLE GRAPH



PERFORMING THE BREADTH FIRST TRAVERSAL

```
void BFT(graph G,int n)
{
int i;
boolean visited[SIZE];

for(i=1;i<=n;i++)
        visited[i]=0;
for(i=1;i<=n;i++)
        if(!visited[i])
                BFS[i};
}
```

DEPTH FIRST SEARCH

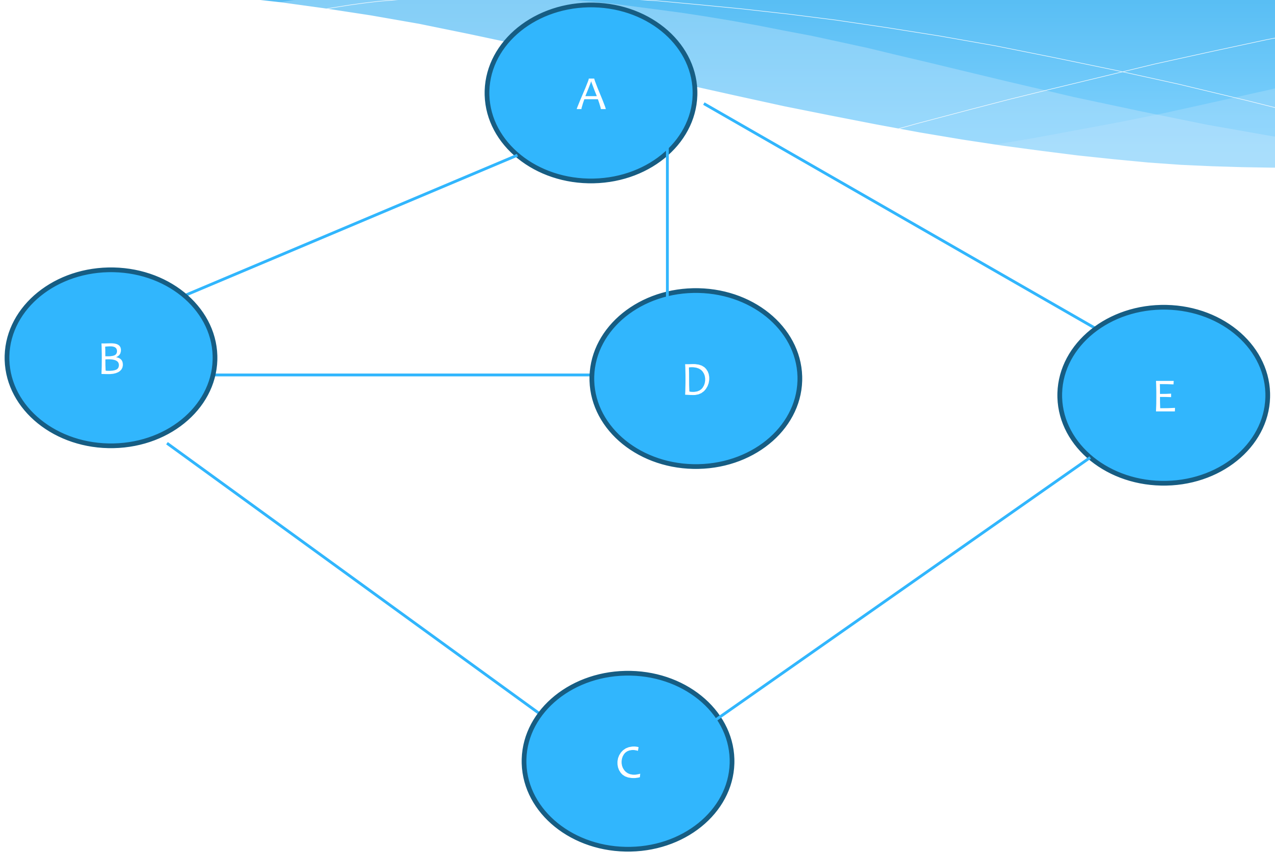
<u>**BASIC IDEA**</u>

- **Starting vertex is arbitrarily chosen or determined by the problem.**
- **Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.**
- **Uses a stack**
- **a vertex is pushed onto the stack when it's reached for the first time**
- **a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex**

ALGORITHM FOR DFS

```
void DFS(int v)
{
     visited[v]=1;
     for each vertex w adjacent to v
     {
          if(!visited[w])
               DFS[w];
     }
}
```

for the graph.....



APPLICATIONS

BREADTH FIRST SEARCH:

- Greedy graph algorithms
  1. finding the minimum spanning tree using PRIM'S ALGORITHM
  2. single source (or) all pair shortest path using DIJKSTRA'S ALGORITHM
  3. NETWORK FLOW PROBLEM
- Testing for connected components

DEPTH FIRST SEARCH:

- Testing for biconnected components(bi-connectivity)
- for eg., checking for the connectivity of a network

CONNECTED COMPONENTS

<u>DEFINITION:</u>

Two vertices in a graph are in the same connected component if and only if there is a path from one vertex to the other

<u>NOTE:</u>

BFS algorithm can be used to test whether a graph is connected or not

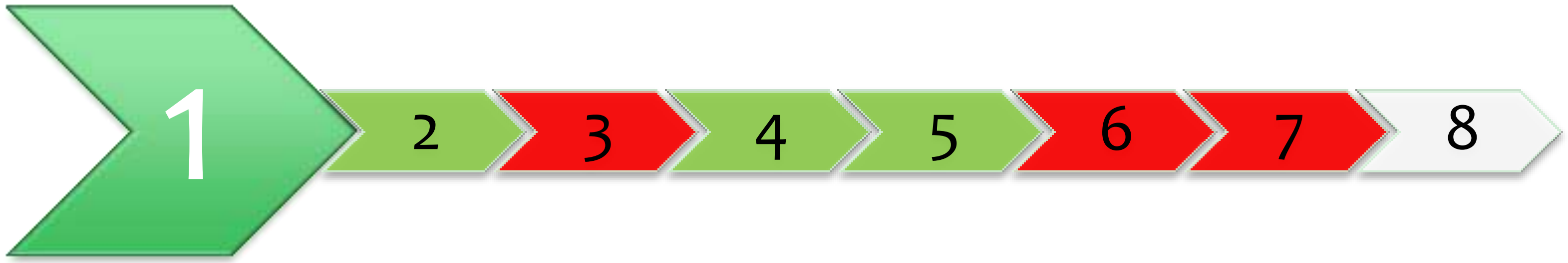
- ***If a graph G is connected then only 1 call to the function BFT(G,n) is made***
- ***The number of calls made to the BFT function can be used to roughly determine the number of "disconnected" components***

DETERMINING A CONNECTED COMPONENT

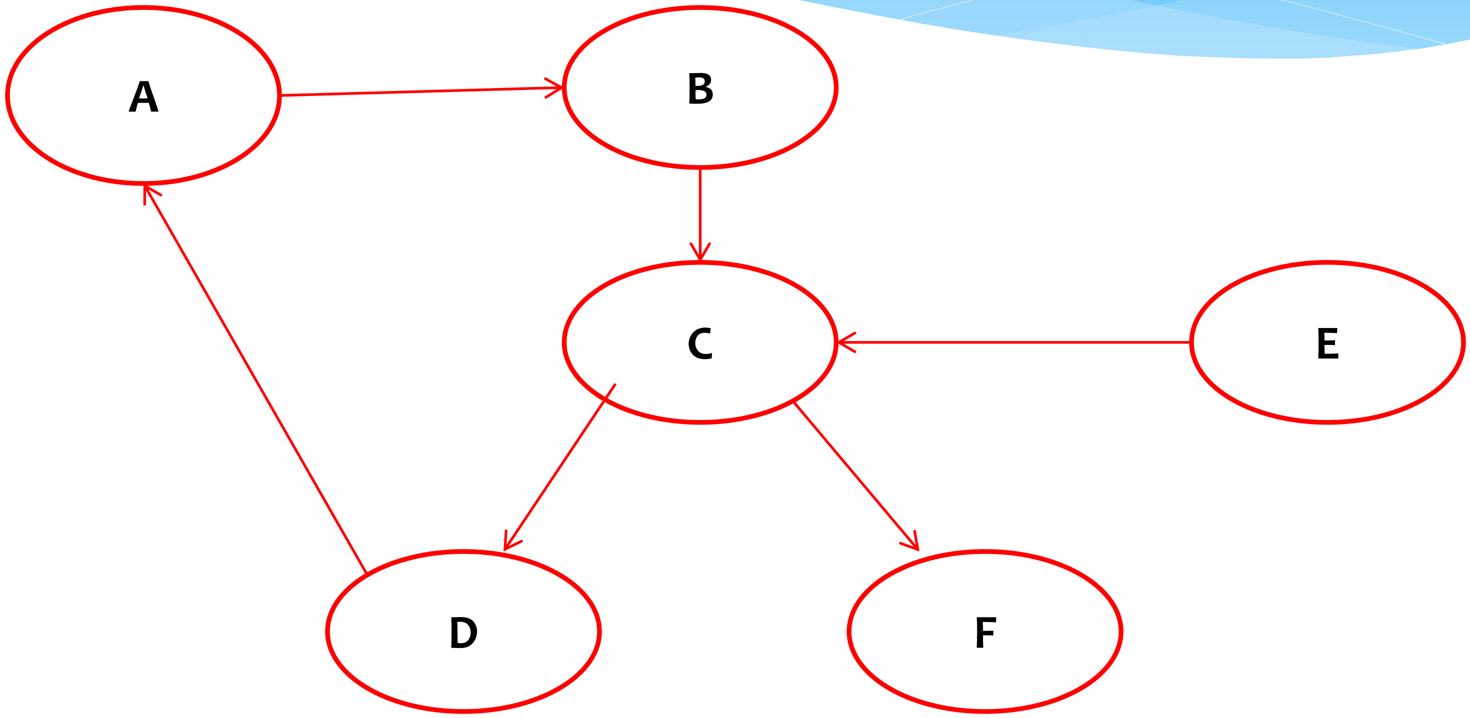
STRATEGY

- **All newly visited vertices on a call to BFS represent vertices in a connected component of G**
- **To display the connected components modify BFS to put newly visited vertices into a list**

Consider for the graph discussed previously.....

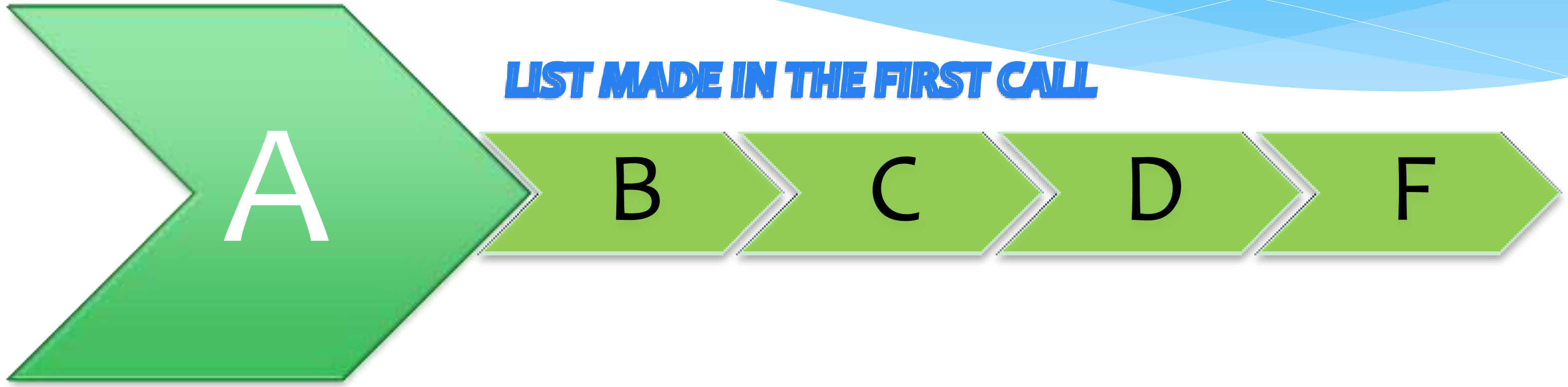


FOR THE DIRECTED GRAPH

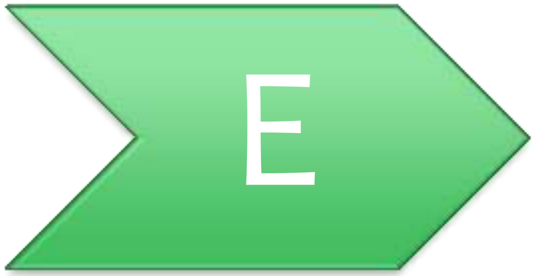

$$\begin{tikzcd}
A \arrow[r] & B \arrow[d] & \\
& C \arrow[dl] \arrow[dr] & E \arrow[l] \\
D \arrow[uu] & & F
\end{tikzcd}$$

THE DFS LIST IS

LIST MADE IN THE FIRST CALL



LIST MADE DURING THE SECOND CALL

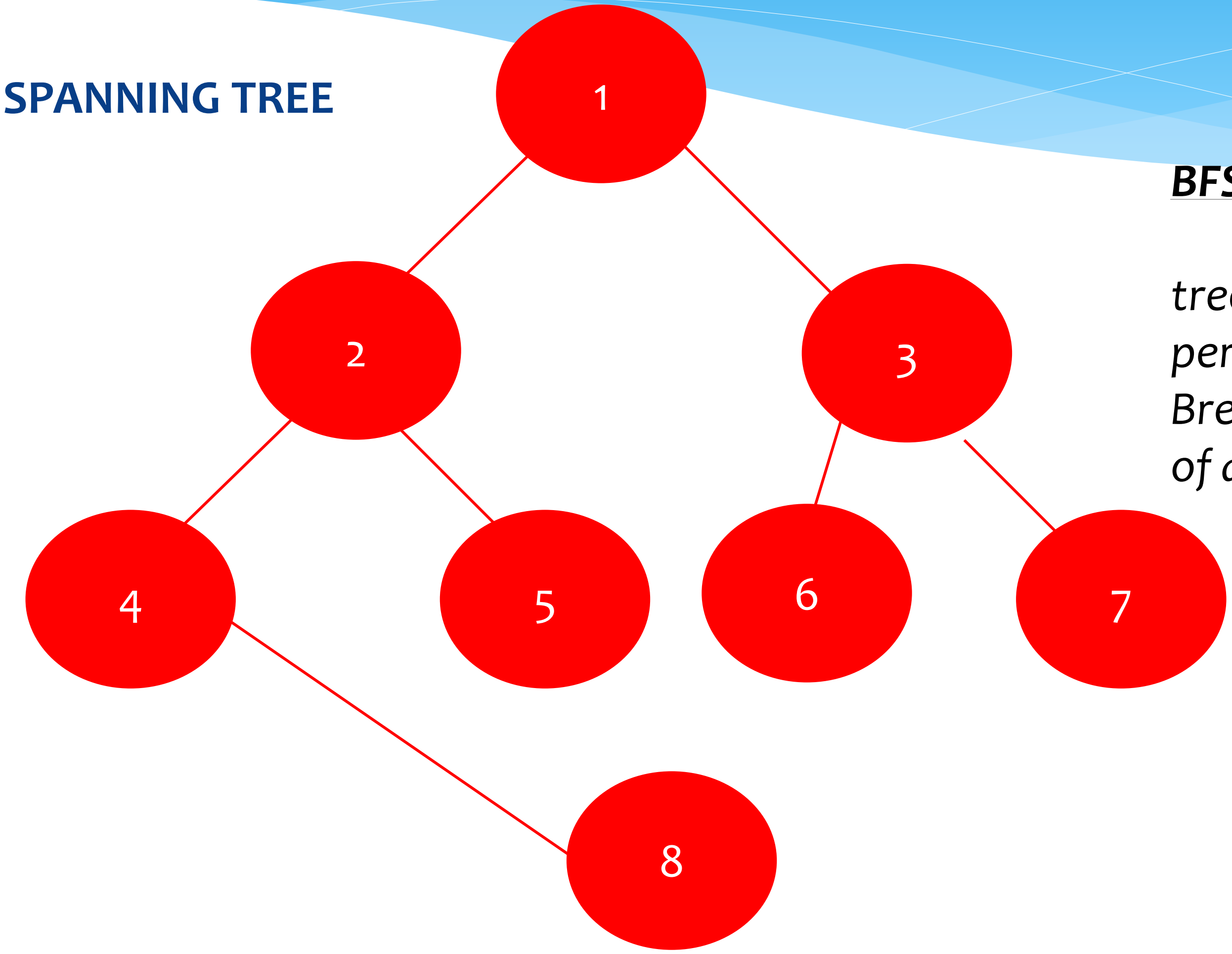


As the number of calls made to the function is 2 it means that the graph is not connected

There are two sets of connected components represented above

SPANNING TREE

BFS SPANNING TREE

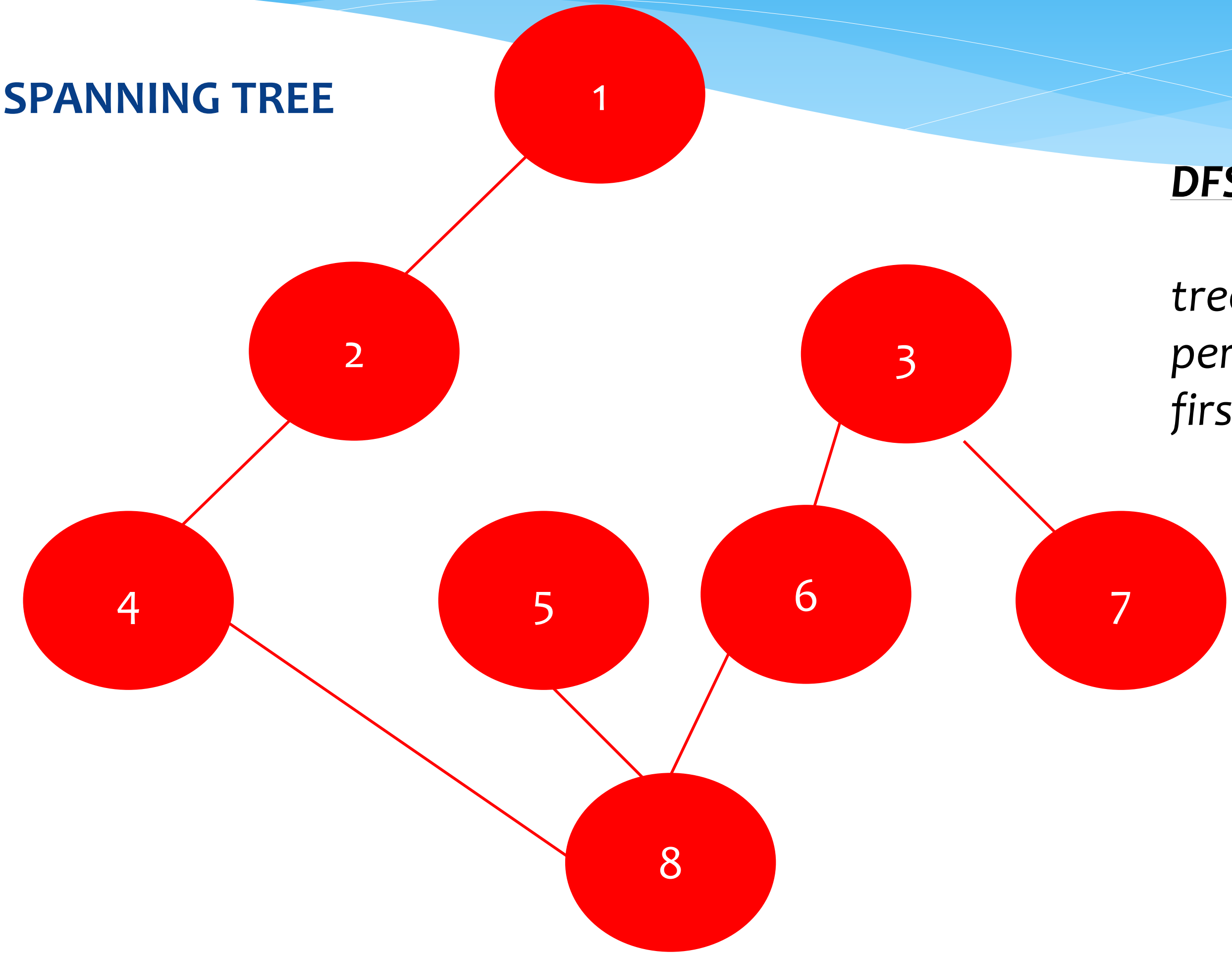


BFS SPANNING TREE:

A spanning tree constructed by performing the Breadth first search of a graph

SPANNING TREE

DFS SPANNING TREE



DFS SPANNING TREE: *A spanning tree constructed by performing the Depth first search of a graph*

BICONNECTED COMPONENTS

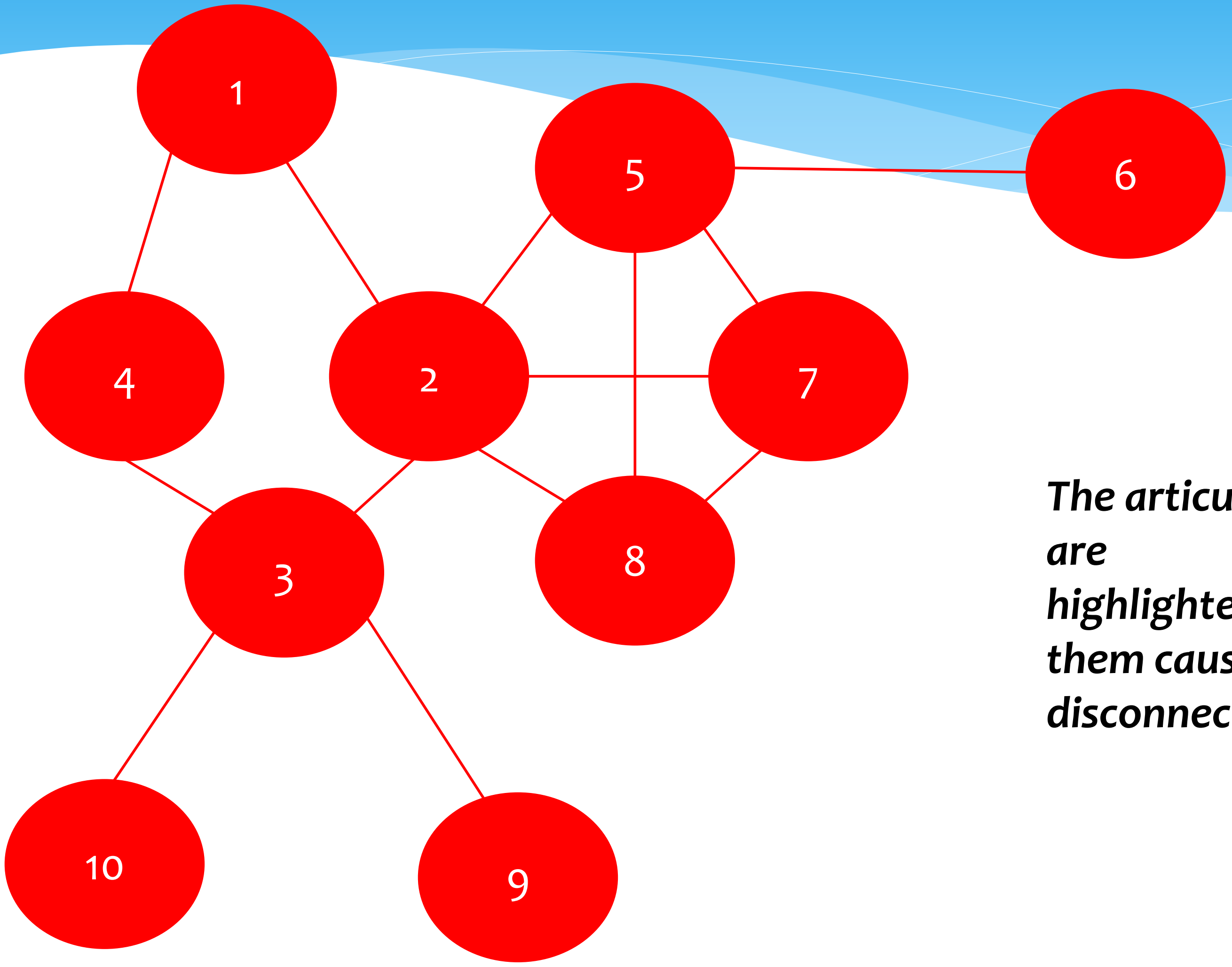
<u>NOTE:</u> Henceforth the word "Graph" would be used instead of the term "undirected Graph"

<u>ARTICULATION POINT:</u> A vertex v in a connected graph is said to be an articulation point if and only if the the deletion of the vertex v and all its edges incident to it "disconnects" the graph into two or more non-empty components.

<u>Biconnected graph:</u> A graph G is said to be biconnected if and only if it contains no articulation points

Let us see an example....

A connected graph



The articulation points are highlighted…removing them causes disconnection

POINTS TO NOTE

LEMMA: ***Two biconnected components can have at most one vertex in common and this vertex is an articulation point.***

TRANSLATION: ***The common vertex of two biconnected components is an articulation point.***

NOTE: ***No edge can be in two different biconnected components as this would require two common vertices(violation of Lemma!!)***

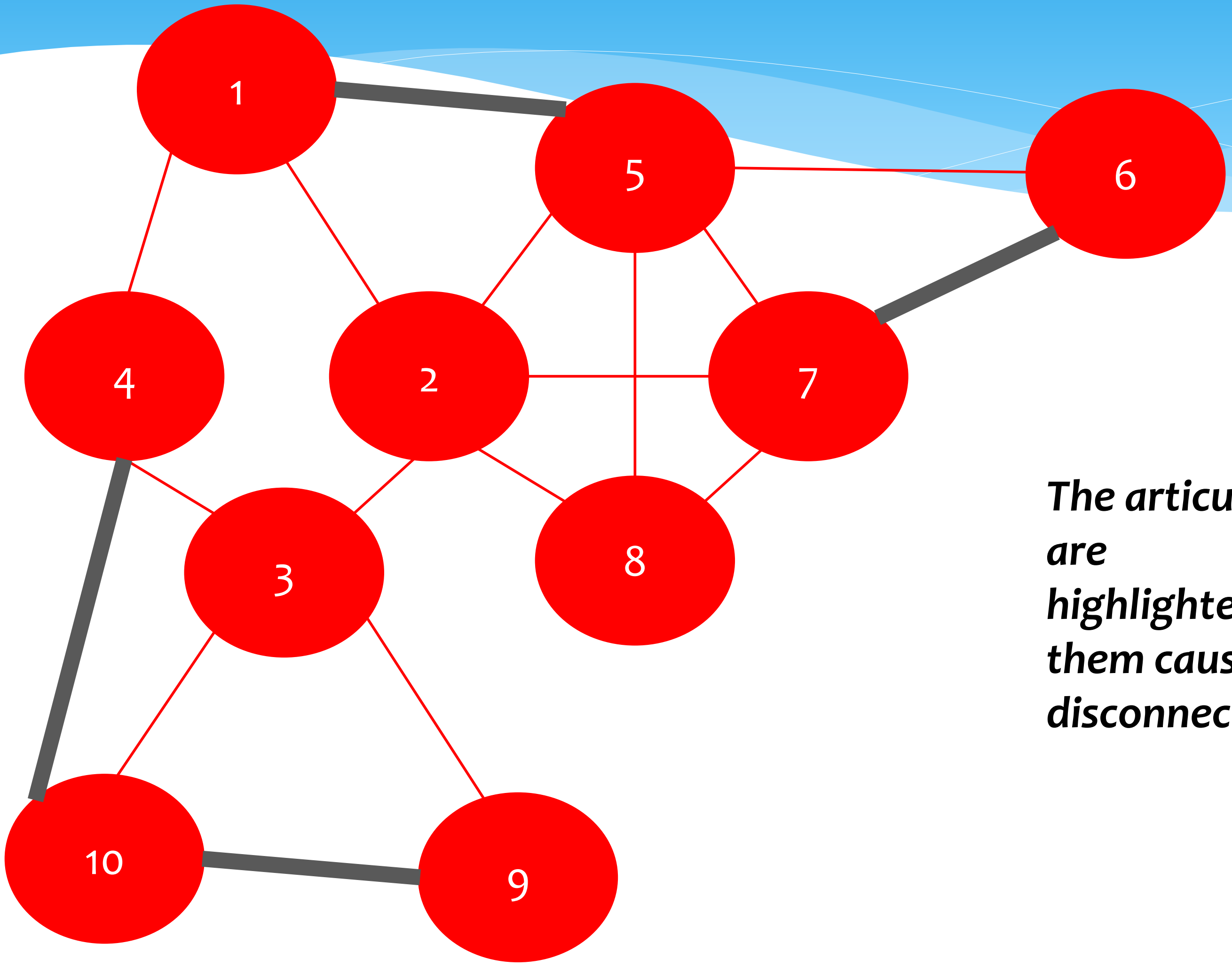
MAKING A CONNECTED GRAPH BICONNECTED

TECHNIQUE:

Simply add “redundant” edges between the connected components that have the articulation point (say a) in common...

for example the previously discussed connected graph can be made biconnected by adding the throbbing links(see next slide...if u r asleep pls continue...)

Connected to biconnected…



The articulation points are highlighted…removing them causes disconnection

SCHEME TO CONSTRUCT A BICONNECTED COMPONENT

```
for each articulation point a
{
      let $B_1, B_2, \ldots B_k$ be the biconnected
      components containing vertex a;

      let $v_i, v_i \neq$ a, be a vertex in $B_i$ ,
                                 1≤i≤k;
      add to G the edges ($v_i, v_{i+1}$),
                                 1≤i≤k;
}
```

IDENTIFYING ARTICULATION POINTS

SIMPLE ALGORITHM:

- ***Construct a DFS spanning tree for the given graph***
- ***Have parameters dfn[u],L[u]***
- ***Do a preorder traversal of the spanning tree and compute dfn[u] for each node as the ith node that is visited***
- ***Compute the value of L[u] as***

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

IDENTIFYING ARTICULATION POINTS contd…

SIMPLE ALGORITHM:

- ***Nodes which satisfy***

  L[w]≥dfn[u], w being the children of u are identified as articulation points

- ***SPECIAL CASE OF ROOT NODE***

  Note: The root node is always listed as an articulation point

```
if root node has exactly one child
then exclude the root node from AP list
else
root node is also an articulation point
```

COMPUTATIONS FOR THE GIVEN GRAPH

NODE	DFN[U]	L[U]
1	1	MIN{1,1,-}=1
2	6	MIN{6,MIN{6,8,6,6},1}=1
3	3	MIN{3,MIN{4,5,1}}=3
4	4	MIN{2,1}=1
5	7	MIN{7,min{8,6,6}}=6
6	8	MIN{8,-,-}=8
7	9	MIN{9,6,6}=6
8	10	MIN{10,-,6}=6
9	5	MIN{5,-,-}=5
10	4	MIN{4,-,-}=4

Branch & Bound

Review of Backtracking

1. Construct the *state-space tree*

- nodes: partial solutions
- edges: choices in extending partial solutions

2. Explore the state space tree using depth-first search

3. "Prune" *nonpromising nodes*
- dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems
- Breadth first search(FIFO B&B) or D-search(LIFO B&B) is performed on the state space tree
- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
- Uses the bound for:
  - ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far
  - guiding the search through state-space

Example: Assignment Problem

Select one element in each row of the cost matrix C so that:

- **no two selected elements are in the same column**
- **the sum is minimized**

Example

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

N people n jobs cost should be small.

Example: First two levels of the state-space tree

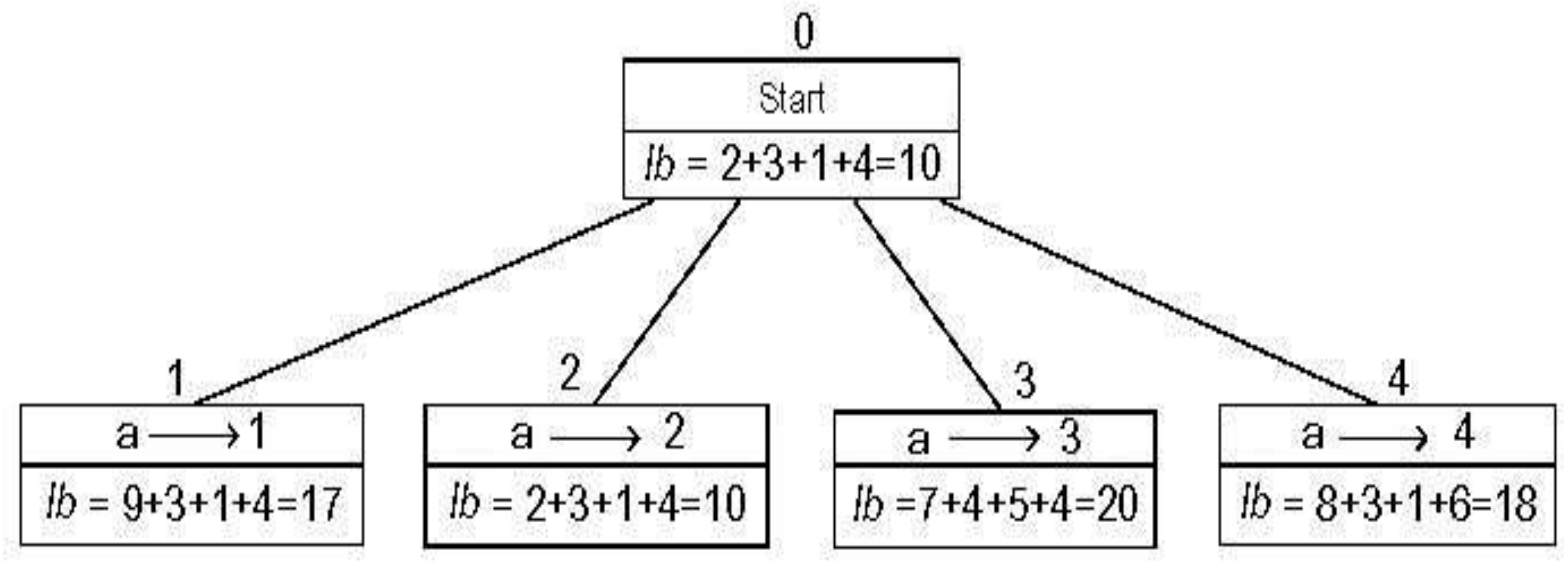


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person *a* and the lower bound value, *lb*, for this node.

Explanation

Select 1st element from first row i.e 9

Select minimum from remaining rows 3, 1, 4 not from 1st column
So 9+3+1+4 = 17.

Select 2nd element from first row i.e 2

Select minimum from remaining rows 3, 1, 4 not from 2nd column
So 2+3+1+4 = 10.

Select 3rd element from first row i.e 7

Select minimum from remaining rows 3, 1, 4 not from 3rd column

So 7+4+5+4 = 20.
Last one it is 8+3+1+6 = 18

Example (cont.)

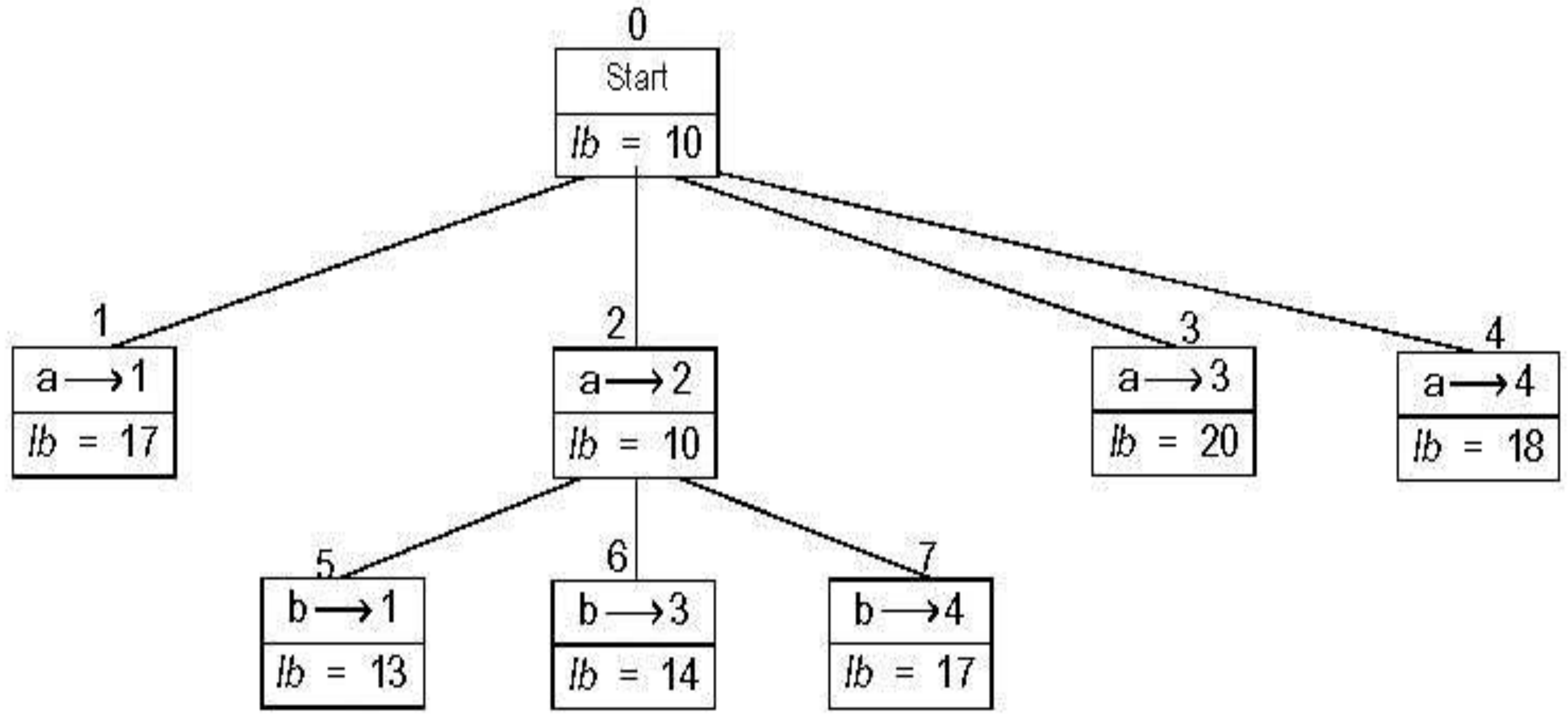


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Explanation(contd…)

1. Select 2nd element from first row i.e 2

Select 1st element from second row i.e 6

Select minimum from remaining rows 1, 4 not from 1st column

So 2+6+1+4 = 13.

2. Select 2nd element from first row i.e 2

Cannot Select 2nd element from second row

3. Select 2nd element from first row i.e 2

Select 3rd element from third row i.e 3

Select minimum from remaining rows 5, 4 not from 3rd column

So 2+3+5+4 = 14.

4. 2 + 7 + 1+7 =17

Note: Promising node is live node a->2 is live node.

Example: Complete state-space tree

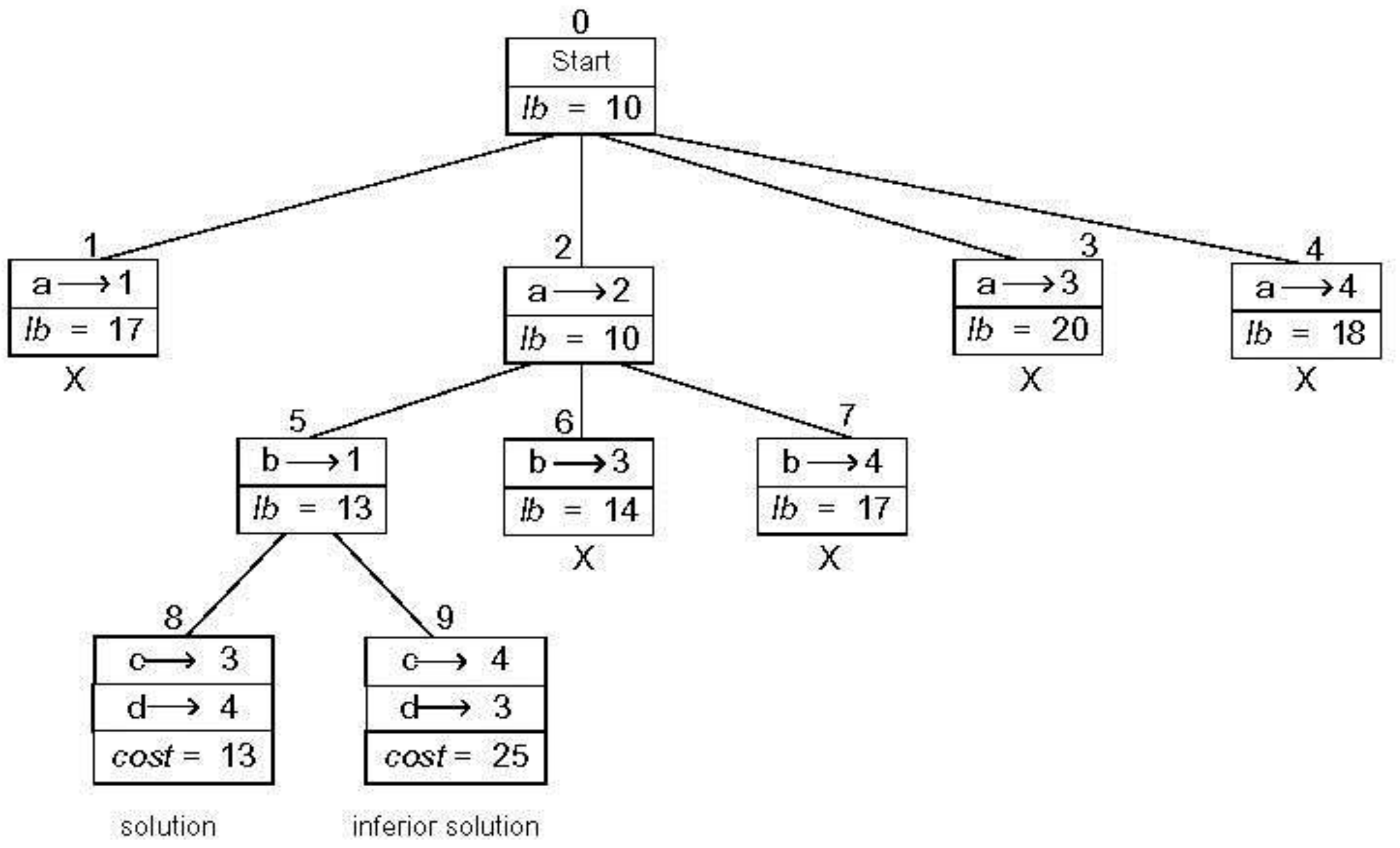


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

0/1 KNAPSACK PROBLEM

COMPARISON OF B&B AND BACKTRACKING

Branch and bound

- **Applicable to optimization problem**
- **The state space tree is generated using best first rule.**

Backtracking

- **Not constrained but mostly applied to Non-optimization problem**
- **The state space tree is generated using DFS**

Greedy Algorithm for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:

$v_1/w_1 \geq \ldots \geq v_n/w_n$

Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 15

i	P_i	W_i	P_i/W_i
1	\$45	3	\$15
2	\$30	5	\$ 6
3	\$45	9	\$ 5
4	\$10	5	\$ 2

Branch and Bound Scheme for Knapsack Problem

Step 1: Order the items in decreasing order of relative values: $v_1/w_1 \geq \ldots \geq v_n/w_n$

Step 2: For a given integer parameter k, $0 \leq k \leq n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios

Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm's output

SOME THINGS TO NOTE

totalSize = currentSize + size of remaining objects that can be fully placed

bound (maximum potential value) = currentValue + value of remaining objects fully placed + (K - totalSize) * (value density of item that is partially placed)

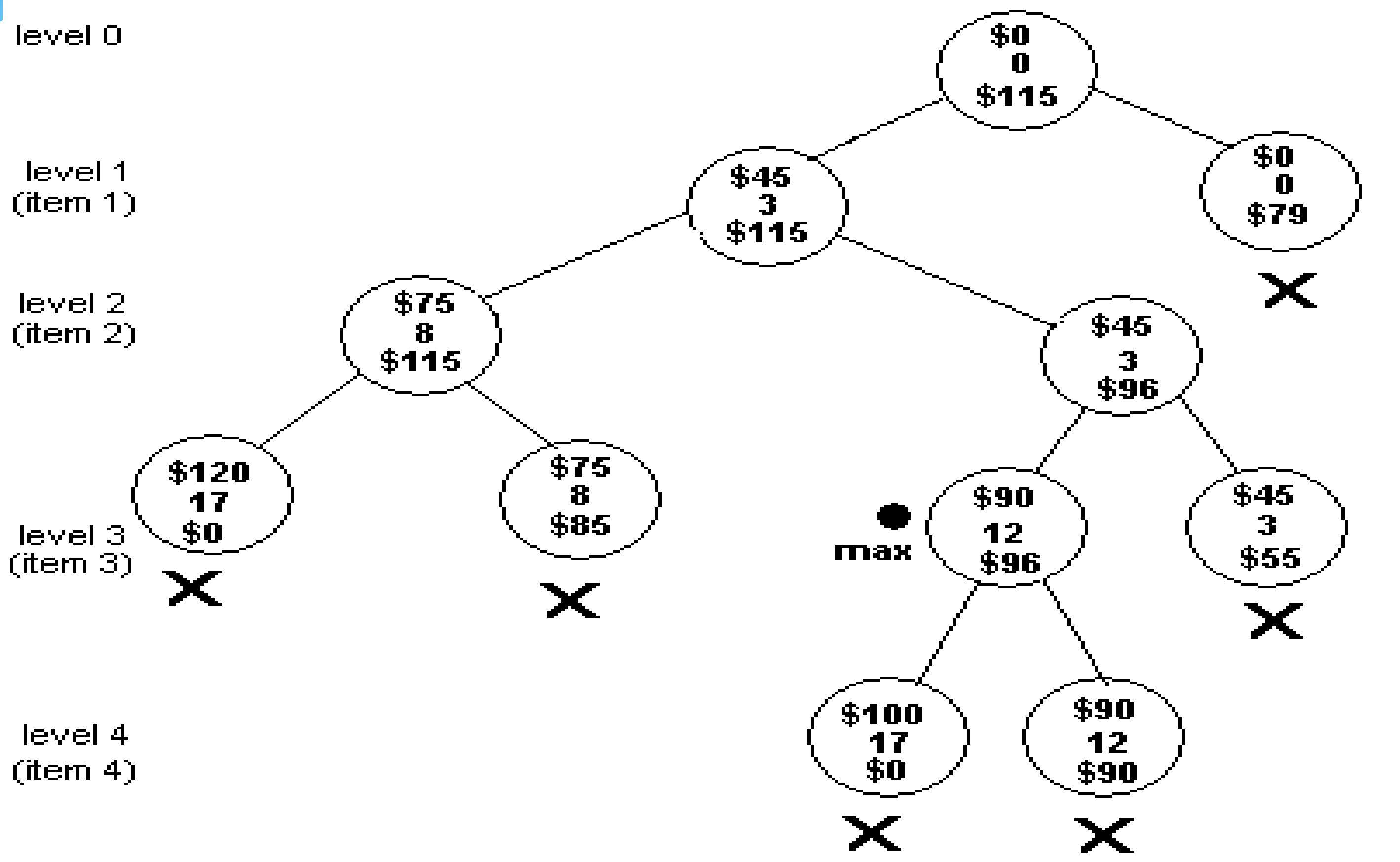
IN FORMULAE

$$\text{totalSize} = \text{currentSize} + \sum_{j=i+1}^{k-1} s_j$$

$$\text{bound} = \text{currentValue} + \sum_{j=i+1}^{k-1} v_j + (K - \text{totalSize}) * (v_k/s_k)$$

For the root node, currentSize = 0, currentValue = 0

THE STATE-SPACE TREE(WITH BOUND)[PERFORMING A BFS TRAVERSAL]



HENCE THE FINAL ANWER SET WOULD BE

X={1,0,1,0}

i.e., objects 1 and 3 are chosen...2 and 4 are discarded

PRESENTED BY ARVIND KRISHNAA J

WHAT THE PREVIOUS PROBLEM DID

Step 1: *We will construct the state space where each node contains the total current value in the knapsack, the total current size of the contents of the knapsack, and maximum potential value that the knapsack can hold. In the algorithm, we will also keep a record of the maximum value of any node (partially or completely filled knapsack) found so far.*

Step 2: Perform the breadth first traversal of the state space tree computing the bound and totalsize

Step 3: Discard(prune)those "non-promising nodes" which either have

(a) a lower bound than the other nodes at same level

(b) whose size exceeds the total knapsack capacity

Step 4: The above method involving the BFS of the state space tree is called

FIFO BRANCH & BOUND ALGORITHM

LC ALGORITHM(LEAST COST)

- This is a generic form/generalization of both the BFS and DFS algorithms
- *In LC algorithm a traversal is performed on the state space tree*
- ***Each node is given a rank based on a minimization rule(also known as best first rule) f(x)***

(in case of the Knapsack problem the minimization rule can be stated as f(x)=--g(x),where g(x) is the maximization rule for the Knapsack problem)

- ***Once all nodes have been ranked eliminate/prune the nodes with the poorest ranks.***
- ***Repeat till a feasible solution is obtained***

(sorry i cant go into too much further detail....its getting too mathematical...)

$\mathcal{NP}$-Hard AND $\mathcal{NP}$-Completeness

Introduction

TWO KINDS OF ALGORITHMS

Polynomial Time: ***Algorithms whose solution is found in polynomial time***

eg., ***Sorting, searching etc.,***

Non-polynomial Time: ***Algorithms which DO NOT take polynomial time to find the solution***

eg., ***Travelling salesperson problem (O($n^2 2^n$))***

TWO KINDS OF ALGORITHMS

- ***Problems for which there is no polynomial time complexity, are computationally related***
- ***There are two classes of such problems***
- **1. NP HARD AND**

  2. NP COMPLETE

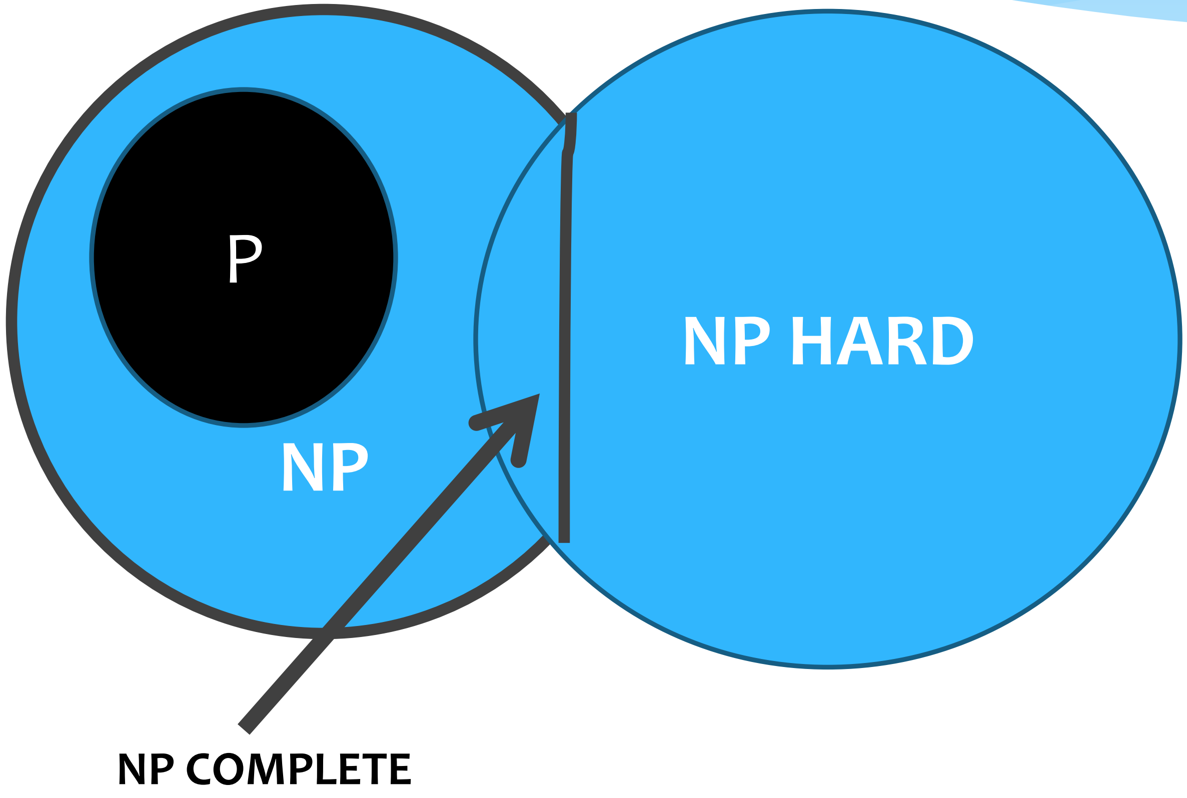
NP COMPLETE

The problem that is NP complete has the property that it can be solved in polynomial time if and only if all the other NP complete problems can be solved in polynomial time

$\mathcal{NP}$ HARD

If an $\mathcal{NP}$-hard problem can be solved in ploynomial time, then all $\mathcal{NP}$ complete problems can also be solved in polynomial time.

SET REPRESENTATION OF THE ABOVE STATEMENTS



NON DETERMINISTIC ALGORITHMS

Deterministic Algorithm: *Has the property that result of every operation is uniquely defined*

Non-Deterministic Algorithm: *Outcome of an operation is not uniquely defined*

THREE NEW FUNCTIONS

Every non-deterministic algorithm makes use of three functions

- *choice(S): arbitrarily choosing an element in a s et S*

  for eg., x=choice(1,n) | means x€[1,n]
- *failure(): unsuccessful completion*
- *success(): successful completion*

SOME EXAMPLES OF NON-DETERMINISTIC ALGORITHMS

- NON DETERMINISTIC SEARCHING
- ND SORTING
- MAXIMUM CLIQUE PROBLEM
- 0/1 KANPSACK PROBLEM
- SATISFIABLITY

and many, many more

NON DETERMINISTIC SEARCHING

```
nondeterministic_search(x)
{
      int j=choice(1,n);
      if(A[j]==x)
      {
            cout<<j;
            success();
      }
      cout<<‘0’;
      failure();
}
```

NON DETERMINISTIC SORTING

```
voidnondeterministic_sort(int A[],int n)
{
        int B[SIZE],i,j;
        for(i=1;i<=n;i++) B[i]=0;
        for(i=1;i<=n;i++)
        {
                j=choice(1,n);
                if(B[j])
                        failure();
                B[j]=A[i];
        }
        for(i=1;i<=n;i++)//verify order
                if(B[i] > B[i+1])
                        failure();
        for(i=1;i<=n;i++)
                cout<<B[i]<<‘ ‘;
        success();
}
```

NON DETERMINISTIC Clique

```
void DCK(int G[]{SIZE],int n,int k)
{
        S=null;//initially empty
        for(int i=1;i<=k;i++)
        {
                int t=choice(1,n);
                if(t is in S)
                        failure();
                S=S U {t};
        }
        //now S contains k distinct vertices
        for(all pairs (i,j) such that i is in S,j is in S
and i!=j)
                if((i,j) is not an edge of G)
                        failure();
        success();
}
```

PRESENTED BY ARVIND KRISHNAA J

NON DETERMINISTIC KNAPSACK

```
void DKP(int p[],int w[],int n,int m,int r,int x[])
{
        int W=0,P=0;
        for(int i=1;i<=n;i++)
        {
                x[i]=choice(0,1);
                W+=x[i]*w[i];
                P+=x[i]*p[i];
        }
        if((W>m) || (P < r))
                failure();
        else
                success();

}
```

NON DETERMINISTIC SATISFIABLITY

```
void eval(cnf E,int n)
//determine whether the prop. formula is
satisfiable
//variables are x[1],x[2],x[3],...x[n]
{
int x[SIZE];
//choose a truth value assignment
for(int i=1;i<=n;i++)
      x[i]=choice(0,1);
if(E(x,n))
      success();
else
      failure();
}
```

SOME EXAMPLES OF *NP* HARD PROBLEMS

- GRAPH PROBLEMS LIKE
  - NODE COVERING PROBLEM
  - GRAPH COMPLEMENT PROBLEM
- SCHEDULING PROBLEMS
- CODE GENERATION PROBLEM

and many more

SOME EXAMPLES OF *NP* complete PROBLEMS

CHECK OUT THE TEXTBOOK

PAGE NO571

BASIC TEHNIQUES INVOLVED IN SOLVING *NP* COMPLETE PROBLEMS

- APPROXIMATION
- RANDOMIZATION
- RESTRICTION
- PARAMETRIZATION
- HEURISTICS

OR IN SHORT PERFORMING A NON-DETERMINISTIC ANALYSIS ON THE PROBLEM

ONE FINAL QUESTION

Remember this: “No problem is too tough if u spend enuf time on it”

-A great man

WHO IS THIS GREAT MAN???

THANK YOU!!!!!

PRESENTED BY ARVIND KRISHNAA J