Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

UNIT III

BACKTRACKING
Branch-And-Bound

(Reference Book - Horowithz Sahni)

Basic Concepts

- Live Node
  - A node which has been generated and all of whose children have not yet been generated is called a live node.
- E-node
  - The live node whose children are currently being generated is called the E-node.
- Dead Node
  - A generated node which is not to be expanded further or all of whose children have been generated is called a dead node.
- Bounding function
  - It is used to kill live nodes without generating all their children.

Backtracking
Outline

- General method
- Recursive backtracking algorithm
- Iterative backtracking method.
- 8-Queen problem
- Hamiltonian Cycle
- 0/1 Knapsack Problem

Backtracking Algorithm

- Based on depth-first recursive search
- Approach
  1. Tests whether solution has been found
  2. If found solution, return it
  3. Else for each choice that can be made
     a) Make that choice
     b) Recur
     c) If recursion returns a solution, return it
  4. If no choices remain, return failure
- Some times called “search tree”

Recursive backtracking algorithm

- Initially invoked by Backtrack(1)
- $T(x_1,x_2,...,x_i)$: set of all possible values for x_{i+1} such that $(x_1,x_2,...,x_i,x_{i+1})$ is also a path to a problem state
- $B_{i+1}(x_1,x_2,...,x_i,x_{i+1})$: bounding function
  - If $B_{i+1}(x_1,x_2,...,x_i,x_{i+1})$ is false, then the path cannot be extended to reach an answer node

```
void Backtrack( int k )
// This is a schema that describes the backtracking process using
//recursion. On entering, the first k-1 values x[1], x[2], ...., x[k-1] of
the
//solution vector x[1:n] have been assigned. x[] and n are global.
{
     for (each x[k] such that x[k] ∈ T(x[1], ..., x[k-1]) {
          if (Bk (x[1], x[2], ..., x[k])) {
             if (x[1], x[2], ..., x[k] is a path to an answer node)
                   output x[1:k];
             if (k < n) Backtrack(k+1);
          }
     }
}
```

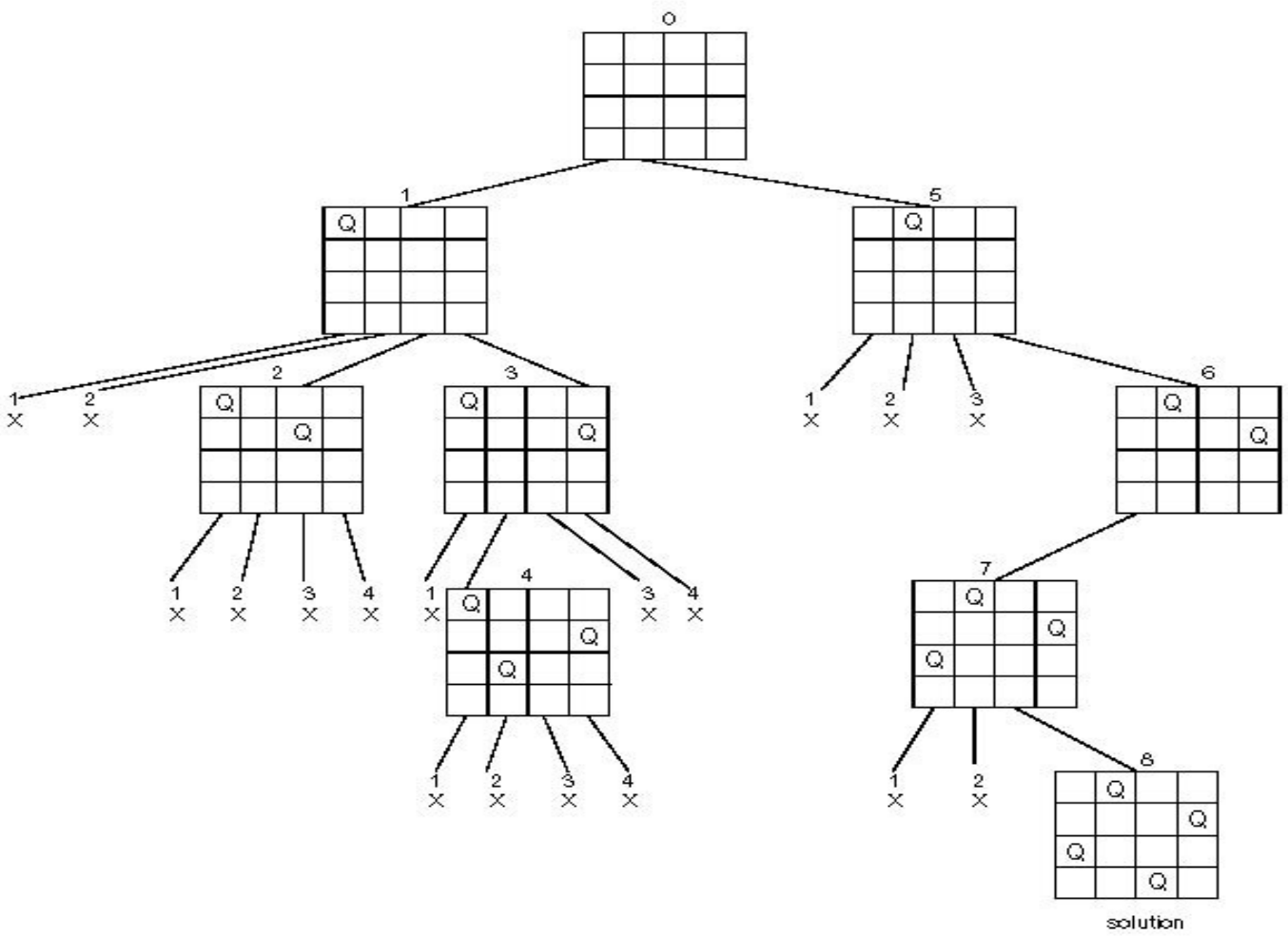
Iterative backtracking method

```
void IBacktrack( int n )
// This is a schema that describes the backtracking
//process. All solutions are generated in x[1:n] and printed
// as soon as they are determined.
{
      int k=1;
      while (k) {
          if (there remains an untried x[k] such that x[k] is in
                    T( x[1], x[2], ..., x[k-1] ) and
               Bk(x[1], ...,x[k]) is true) {

                        if ( x[1], ..., x[k] is a path to an answer node )
                                  output x[1:k];
                    k++; // Consider the next set.
          }
          else k--; // Backtrack to the previous set.
      }
}
```

Example: The n-Queen problem

- Place *n* queens on an *n* by *n* chess board so that no two of them are on the same row, column, or diagonal

State Space Tree of the Four-queens Problem

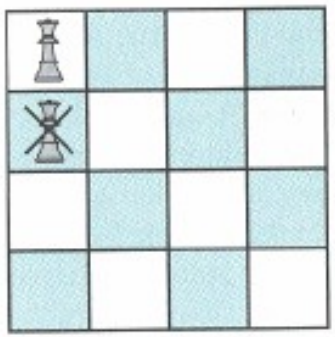


Algorithm – The n-queen problem

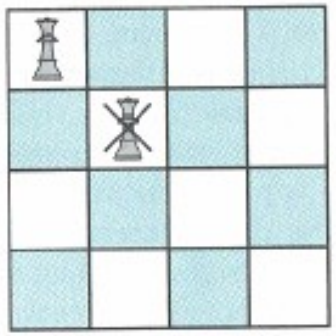
```
Plac(k, i)
{
    for j = 1 to k - 1 do
      if((x[j] = i) or (Abs(x[j] – i) = Abs(j - k )))
            return false;
    return true;
}
```

```
NQueen(k, n)
{
   for i = 1 to n do{
          if Place(k, i) then{
                 x[k] = i;
                 if (k = n) then write (x[1:n]);
                 else Nqueen(k+1, n);
          }
  }
}
```

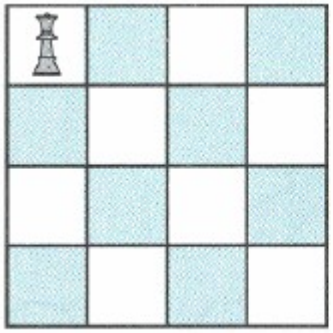
Example



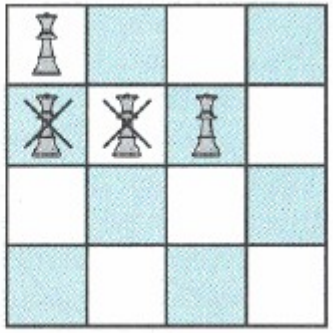
(a)



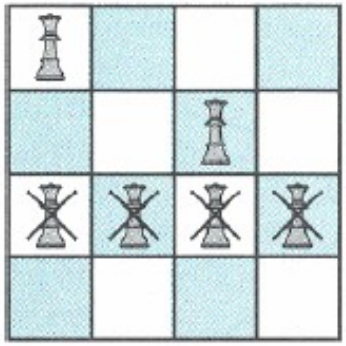
(b)



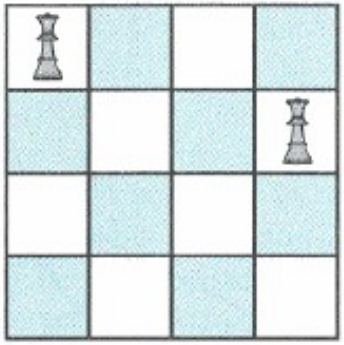
(a)



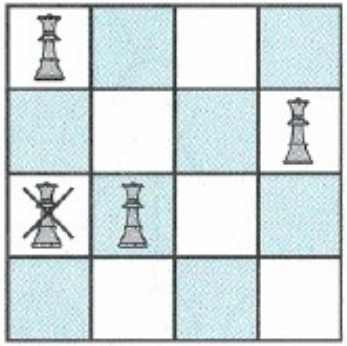
(b)



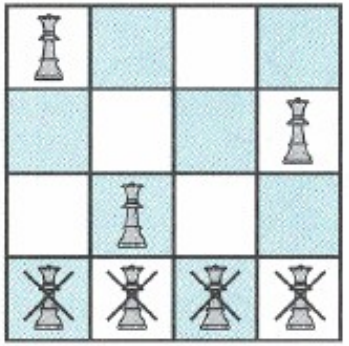
(c)



(d)

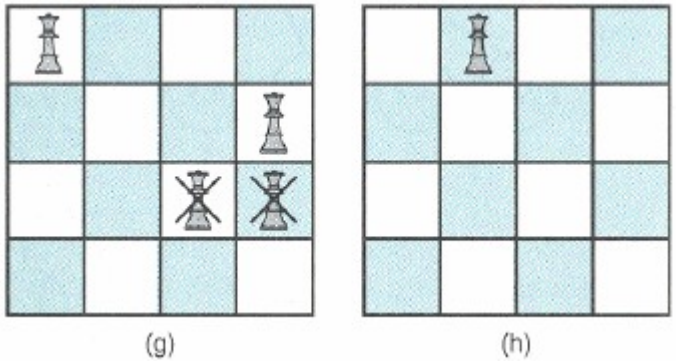


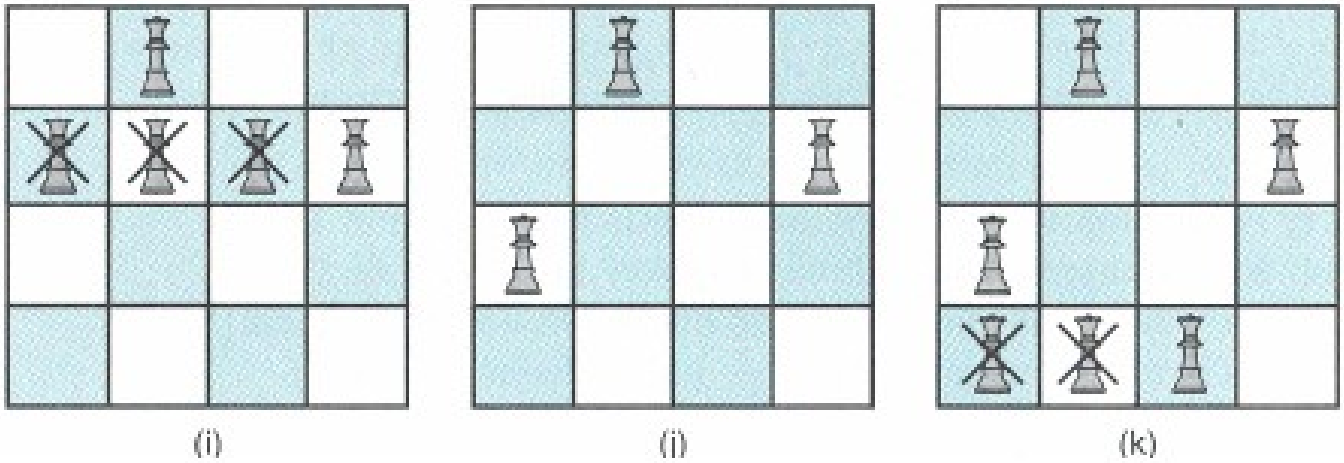
(e)



(f)

Continued…



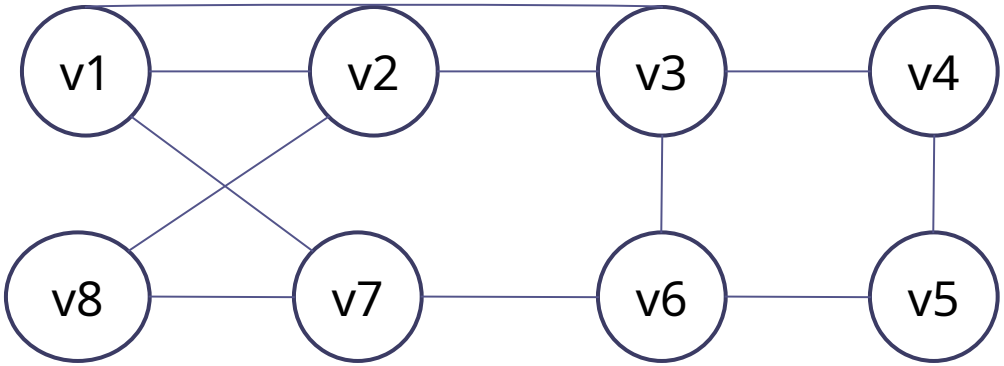


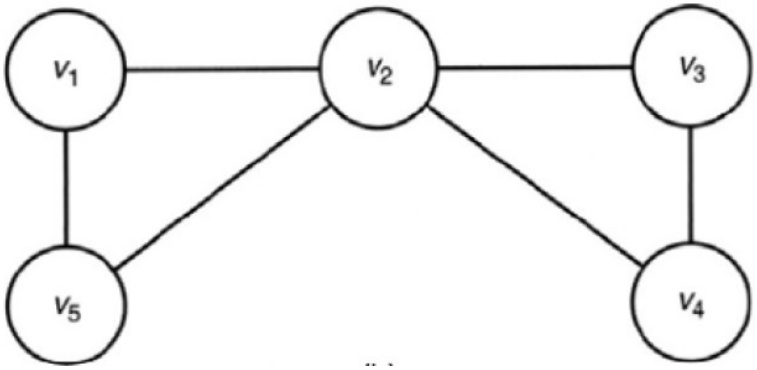
Hamiltonian Cycles

- Definitions
  - Hamiltonian cycle (HC): is a cycle which passes once and exactly once through every vertex of G and returns to starting position
  - Hamiltonian path: is a path which passes once and exactly once through every vertex of G (G can be digraph).
- A graph is Hamiltonian iff a Hamiltonian cycle (HC) exists.

The Hamiltonian Circuits Problem

- Hamiltonian Circuit
- [v1, v2, v8, v7, v6, v5, v4, v3, v1]

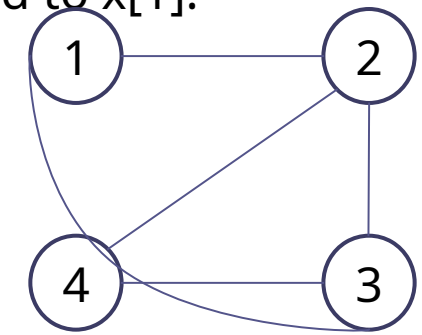




No Hamiltonian Circuit

Algorithm

X[1: k-1] is a path of k-1 distinct vertices if x[k]=0, then no vertex has as yet been assigned to x[k]. After execution x[k] is assigned to the next highest number vertex which does not already appear in x[1, k-1] & is connected by an edge to x[k-1]. otherwise x[k] =0, if k=n then in addition x[k] is connected to x[1].




```
Algorithm Nextvalue(k) {
    repeat {
            x[k] := (x [k] + 1) mod (n + 1) //next vertex
            If ( x[k] = 0) then return
            If G[x[k – 1], x[k]] ≠ 0 ) then { //Is there an edge?
                    for j = 1 to k – 1 do if (x[j] = x[k]) then break;
                    //check distinctness
            If ( j = k) then //if true then vertex is distinct
                    If (( k < n ) or (( k = n) and G [x[n] , x[1] ≠ 0))
                            then return
            }
    } until ( false) ;
}
```

Next

Continued…

This algorithm uses the recursive formulation of backtracking to find all the hamiltonion cycles of a graph. The graph is stored as an adjacency matrix G[1:n, 1:n]. All cycles begins at node 1.

```
Algorithm Hamiltonian(k)
{                       //Set x[2:n] to zero and x[1] to 1. Execute
   repeat {             Hamiltonian(2).
   //generate values for x[k]
            Nextvalue (k); //assign a legal next value to x[k]
            if ( x[k] = 0 ) then return
            if (k = n) then write ( x[1 : n]);
            else Hamiltonian(k + 1);
   } until(false);
}
```

Application

- Hamiltonian cycles in fault random geometric network
- In a network, if Hamiltonian cycles exist, the fault tolerance is better.

Knapsack backtracking

- We are given n objects and a knapsack or bag. The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- Given
  - n = number of weights
  - w = weights
  - P =profits m= knapsack capacity
- Using greedy approach Choosing a subset of the weights such that
  - Max $\Sigma_{1 \le i \le n} p_i x_i$ subject to $\Sigma_{1 \le i \le n} w_i x_i \le m$

$$0 \le x_i \le 1 \qquad 1 \le i \le n$$

The backtracking method

- A given problem has a set of constraints and possibly an objective function.
- The solution must be feasible and it may optimize an objective function.
- We can represent the solution space for the problem using a state space tree
  - The *root of the tree represents 0 choice,*
  - Nodes at depth 1 represent first choice
  - Nodes at depth 2 represent the second choice, etc.
  - In this tree a *path from a root to a leaf represents a* candidate solution

Continued…

- Definition: We call a node *nonpromising if it* cannot lead to a feasible (or optimal) solution, otherwise it is *promising.*

- Main idea: Backtracking consists of doing a DFS of the state space tree, checking whether each node is *promising* and if the node is *nonpromising* backtracking to the node's parent.

Backtracking solution to the 0/1 knapsack problem

```
Algorithm Bknap( k, cp, cw) {
    If ( cw + w[k] ≤ m) then // generate left child then {
        y[k] := 1
        If ( k < n) then Bknap(k + 1, cp + p[k], cw + w[k]);
        If (( cp + p[k] > fp ) and (k = n) then {
            fp := cp + p[k]; fw := cw + w[k];
            for j := 1 to k do x[ j ] := y[ j ];
        }
    }
    If(bound (cp, cw ,k) ≥ fp ) then //generate right child  {
        y[k] := 0; if ( k < n) then Bknap( k + 1, cp, cw);
        If (( cp > fp ) and ( k = n )) then {
            fp := cp; fw := cw;
            for j := 1 to k do x[ j ] := y[ j ]
        }
    }
}
```

Continued…

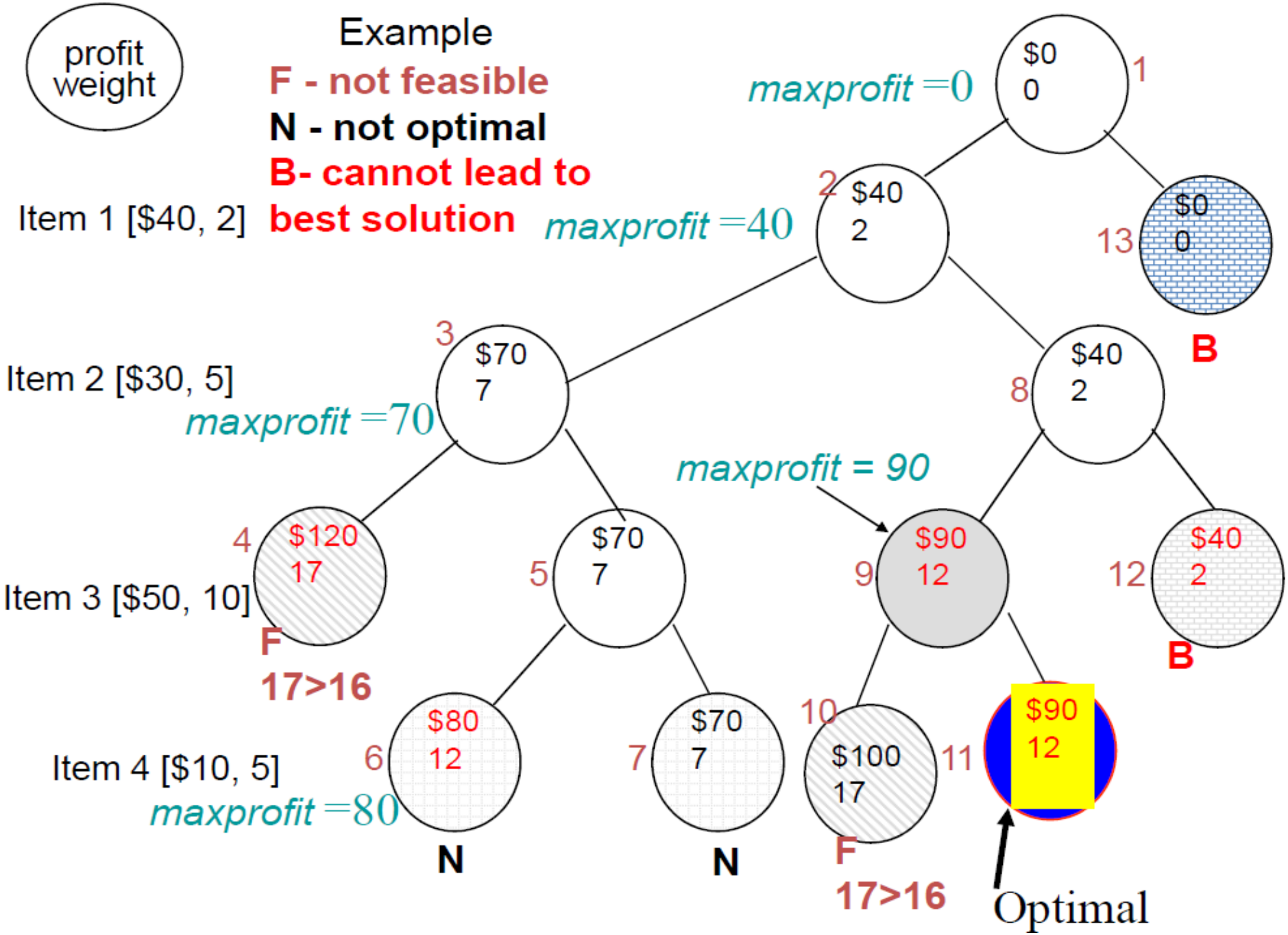
```
Algorithm bound( cp, cw, k)
//cp is the current profit, cw is the current wt total, k is
   the index of last removed item and m is the knapsack
   size
{
   b := cp; c := cw;
   for i := k + 1 to n do {
            c := c + w[i];
            if ( c < m) then b := b + p[i] ;
            else return b + (1 - (c - m)/ w[i]) * p[i];
   }
   return b;
}
```

Back

Example

- Suppose *n = 4, W = 16, and we have the following:*

i	*pi*	*wi*	*pi / wi*
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2


profit
weight
Example
F - not feasible
N - not optimal
B- cannot lead to
best solution
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
maxprofit =0
maxprofit =40
maxprofit =70
maxprofit = 90
maxprofit =80
$0
0
1
$40
2
2
$0
0
13
B
$70
7
3
$40
2
8
$120
17
4
$70
7
5
$90
12
9
$40
2
12
B
F
17>16
$80
12
6
$70
7
7
$100
17
10
$90
12
11
N
N
F
17>16
Optimal

Worst-case time complexity

- Check number of nodes:
  - $1 + 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 1$
- Time complexity:
  - $O(2^n)$

Branch – bound

Outline

- The method
- Control abstractions
  - Least Cost Search (LC Search)
  - Bounding
- FIFO branch and bound
- LC branch and bound
- 0/1 Knapsack problem
  - LC branch and bound
  - FIFO branch and bound solution
- Traveling sales person problem

Branch and Bound

- In backtracking, we used depth-first search with pruning to traverse the (virtual) state space. We can achieve better performance for many problems using a breadth-first search with pruning. This approach is known as branch-and-bound
- Branch and Bound is a general search method.
- Starting by considering the root problem (the original problem with the complete feasible region), the lower bounding and upper-bounding procedures are applied to the root problem.
- If the bounds match, then an optimal solution has been found and the procedure terminates

Continued…

- Otherwise, the feasible region is divided into two or more regions, these sub-problems partition the feasible region.
- The algorithm is applied recursively to the sub-problems. If an optimal solution is found to a sub problem, it is a feasible solution to the full problem, but not necessarily globally optimal.

Continued…

- If the lower bound for a node exceeds the best known feasible solution, no globally optimal solution can exist in the subspace of the feasible region represented by the node. Therefore, the node can be removed from consideration.
- The search proceeds until all nodes have been solved or pruned, or until some specified threshold is met between the best solution found and the lower bounds on all unsolved sub-problems.

Least cost (LC) – search

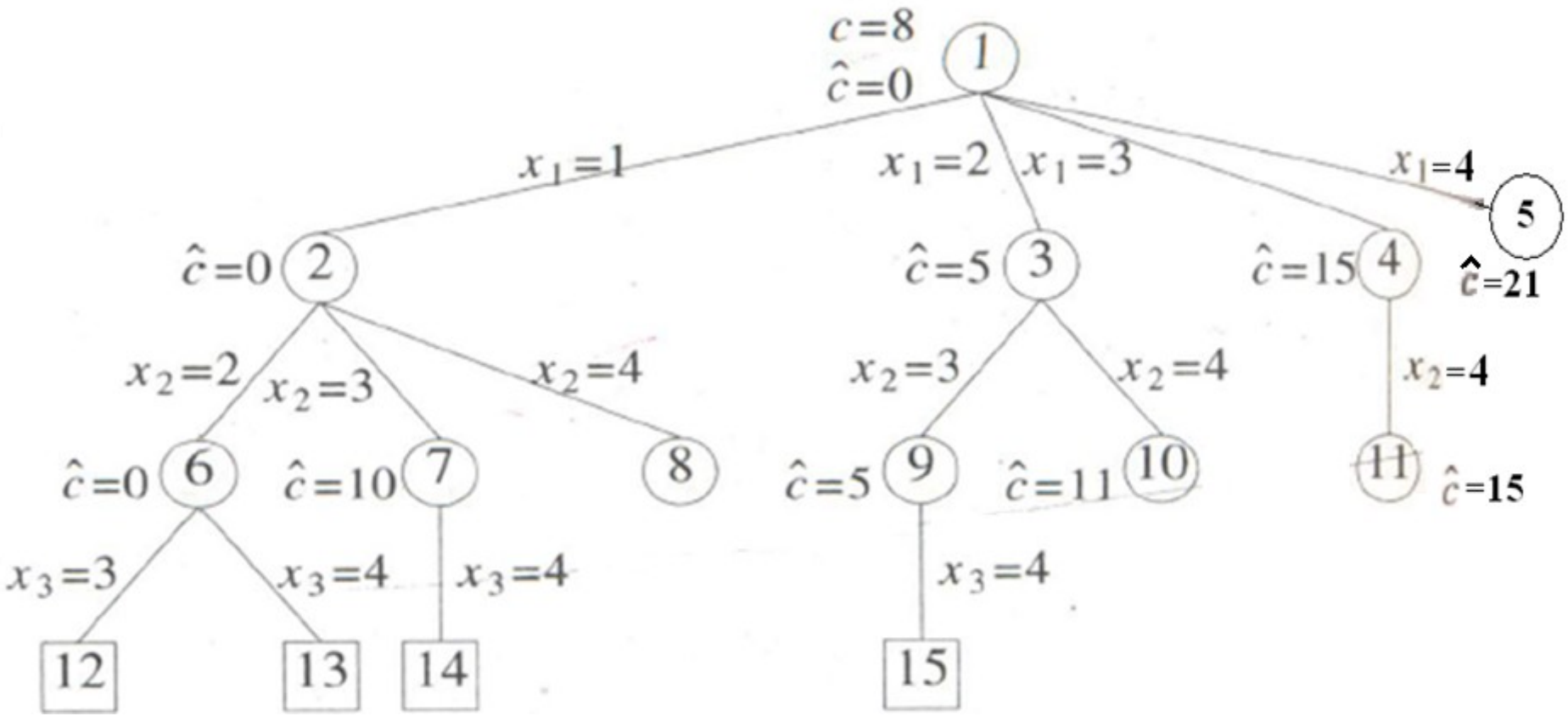
```
listnode = record {
    listnode *next, *parent; float cost;
}

1  Algorithm LCSearch(t)
2  // Search t for an answer node.
3  {
4      if *t is an answer node then output *t and return;
5      E := t; // E-node.
6      Initialize the list of live nodes to be empty;
7      repeat
8      {
9          for each child x of E do
10         {
11             if x is an answer node then output the path
12                 from x to t and return;
13             Add(x); // x is a new live node.
14             (x → parent) := E; // Pointer for path to root.
15         }
16         if there are no more live nodes then
17         {
18             write ("No answer node"); return;
19         }
20         E := Least();
21     } until (false);
22 }
```

bounding

- Consider that there are n jobs and one processor. Each job i has associated with it a three tuple (p_i, d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i, then a penalty is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines.
- The penalty can be incurred only on those jobs not in J.

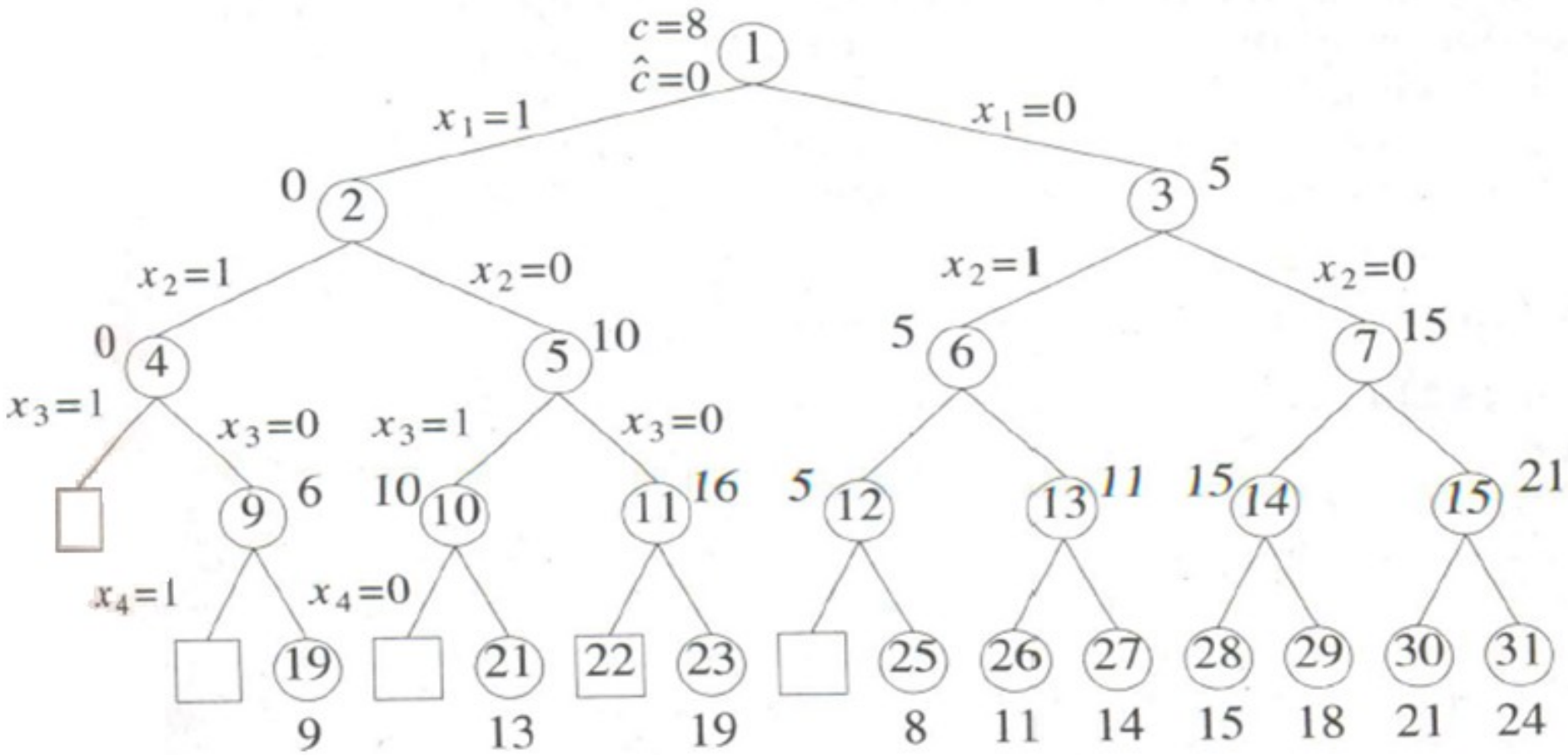
example

- n = 4 $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6, 2, 1)$, $(p_4, d_4, t_4) = (3, 1, 1)$

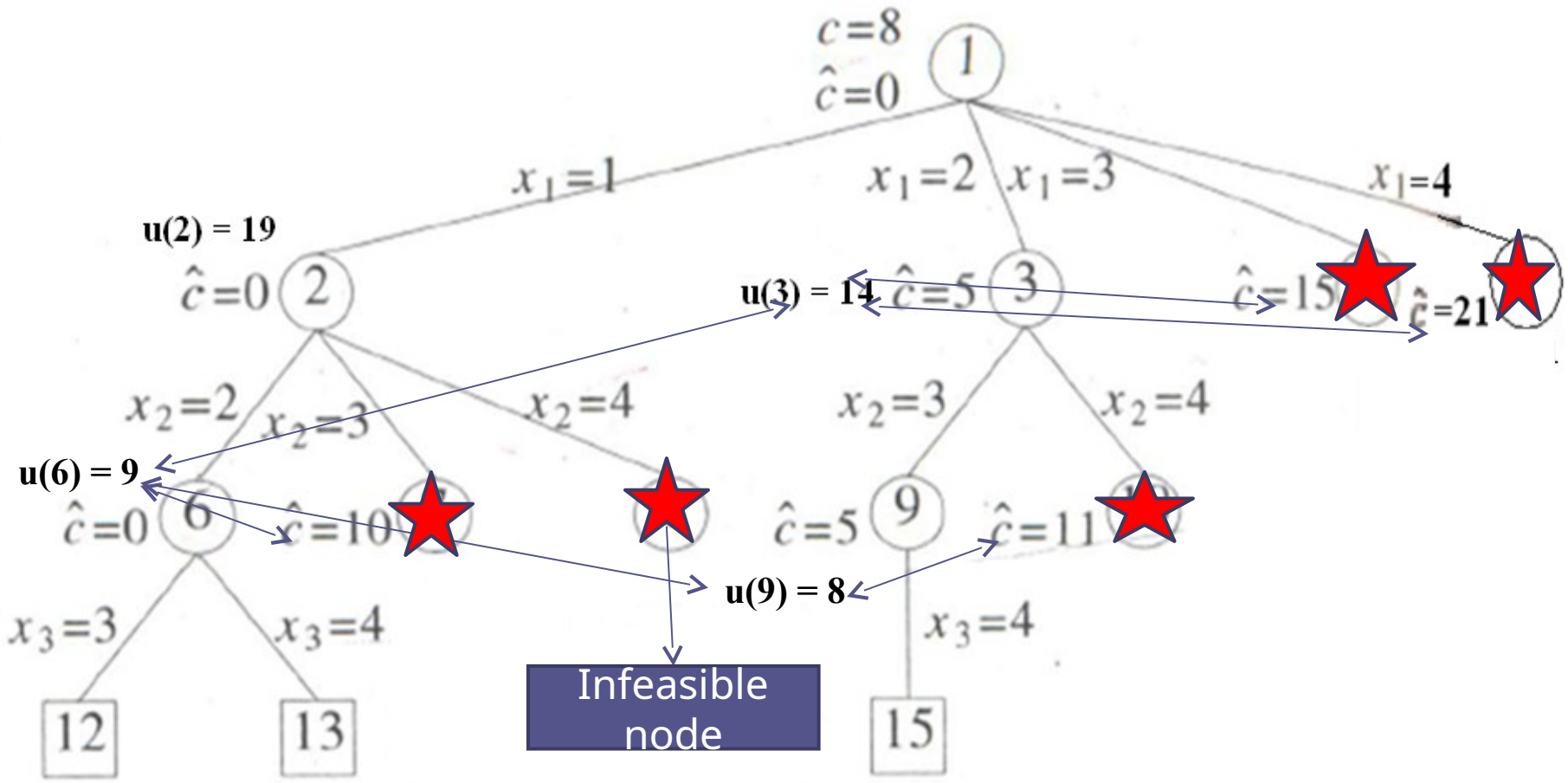


State space tree correspond to variable tuple size formulation

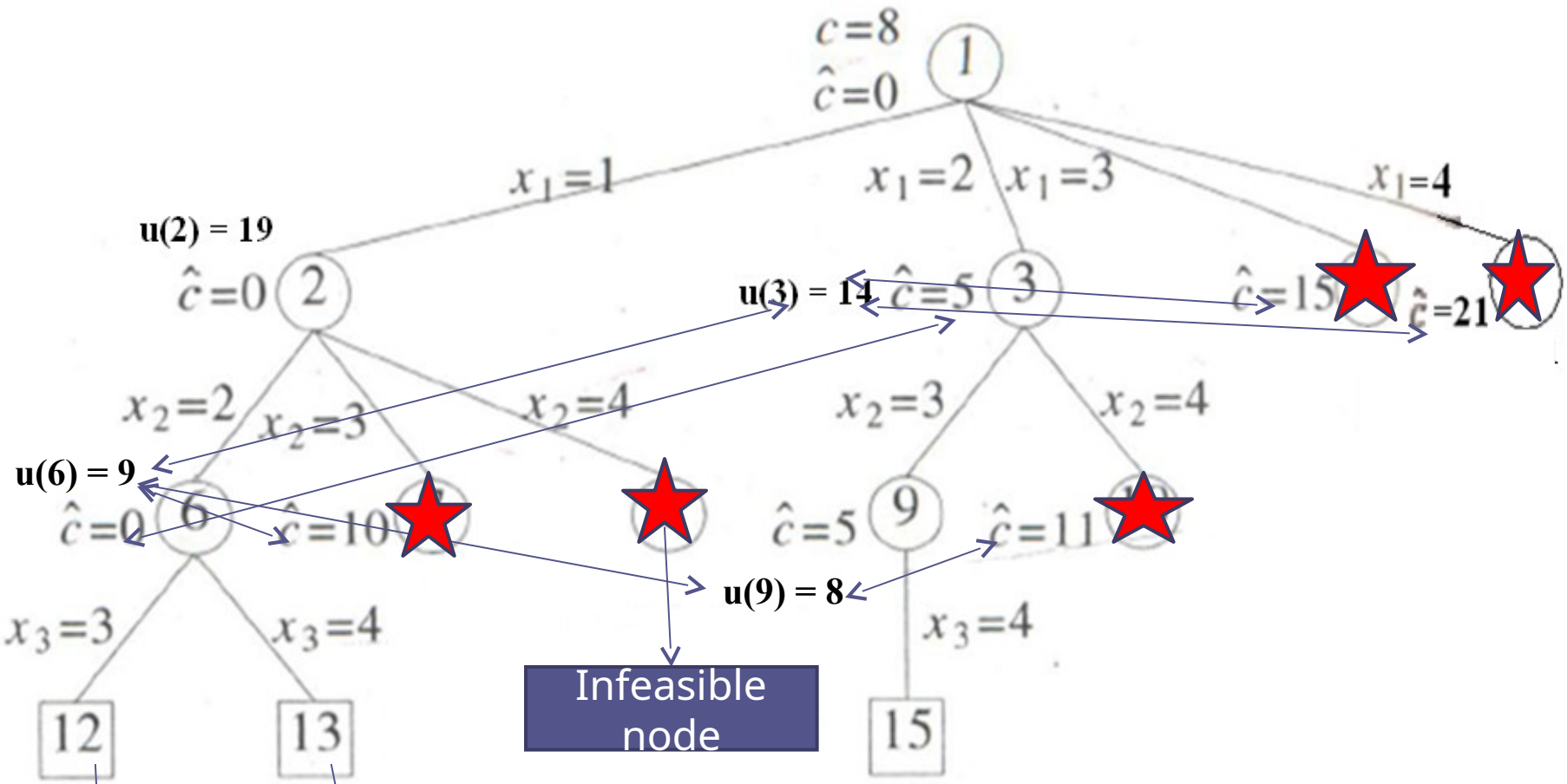
- n = 4 $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6, 2, 1)$, $(p_4, d_4, t_4) = (3, 1, 1)$



State space tree correspond to fixed tuple size formulation



- If Ĉ(x) > upper, kill node x $u(x) = \sum_{1 \le i \le n} p_i$



- If $\hat{C}(x) > upper$, kill node x

$$u(x) = \sum_{1 \le i \le n} p_i$$

Infeasible node

Infeasible node

Traveling sales person problem

- For a given instance G of TSP, the row (column) is said to be reduced if and only if it contains at least one zero and all remaining entries are non-negative.
- The matrix G is reduced if and if only every row and column is reduced.

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

a) Cost Matrix

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

b) Reduced cost matrix L = 25 u = ∞

Continued…

- The total amount subtracted from all the rows and columns is a lower bound on the length of minimum cost tour and may be used as C` value for the root of the state space tree.
- Let A be the reduced cost matrix for node R, Let S be a child of R and an edge <i, j> is added to S.

Continued…

- If S in not a leaf node, then the reduced cost matrix for S may be obtained as
  - Change all entries in row i and column j of A to ∞
  - Set A(j,1) to ∞
  - Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞
- If r is the total amount subtracted in step 3 then c`(S) = C`(R)+A(i,j)+r
- For upper bound function *u*, *u*(R) = ∞ for all nodes R.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

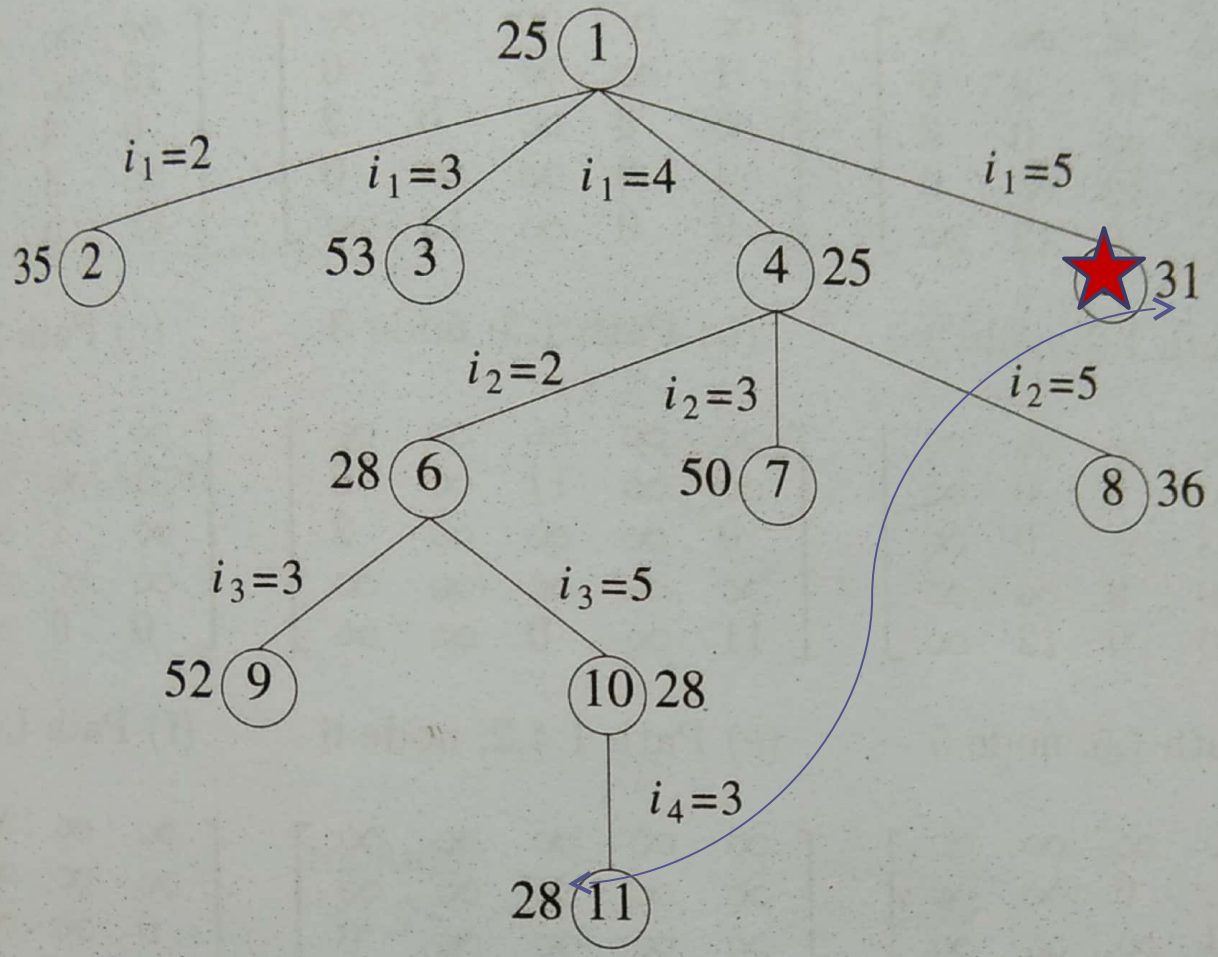
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

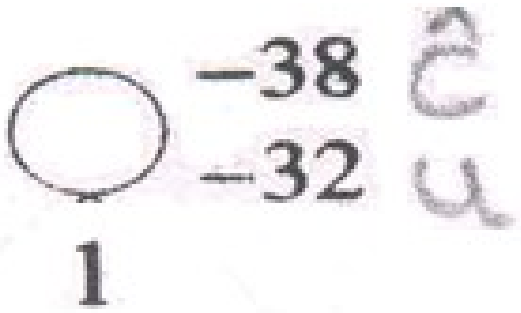


Numbers outside the node are $\hat{c}$ values

Figure 8.12 State space tree generated by procedure LCBB

0/1 Knapsack problem

- Example: n = 4, (p_1, p_2, p_3, p_4) = (10, 10, 12, 18), (w_1, w_2, w_3, w_4) = (2, 4, 6, 9) and m = 15
- $\hat{C}(1) = -[10 + 10 + 12 + 18 * (3/9)] = -38$
- *u(1) = -10 –10 –12 = -32*



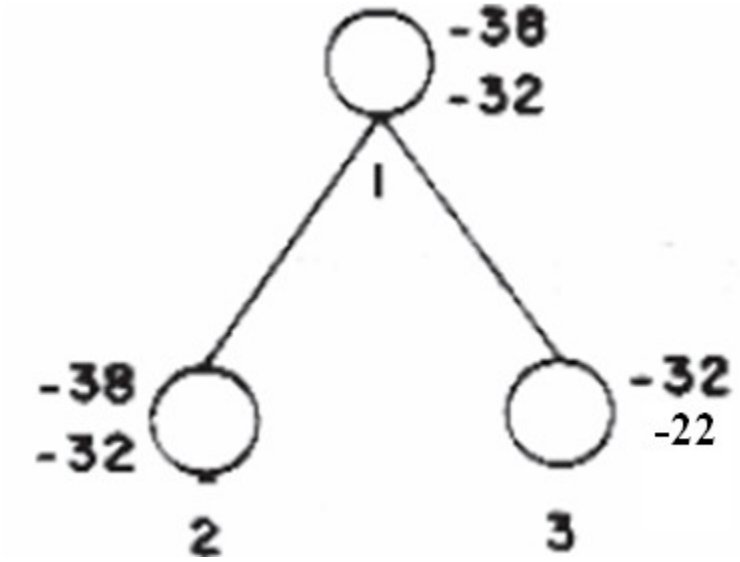
Ubound function for knapsack problem

```
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;
}
```

Continued…

- $\hat{C}(2) = [-10] + [- 10 – 12 - 18 *(3/9)] = -38$
- $u(2) = [-10] + [-10 -12] = -32$
- $\hat{C}(3) = -0 –[10 + 12 + 18 *(5/9)] = -32$
- $u(3) = 0 + [-10 –12] = -22$

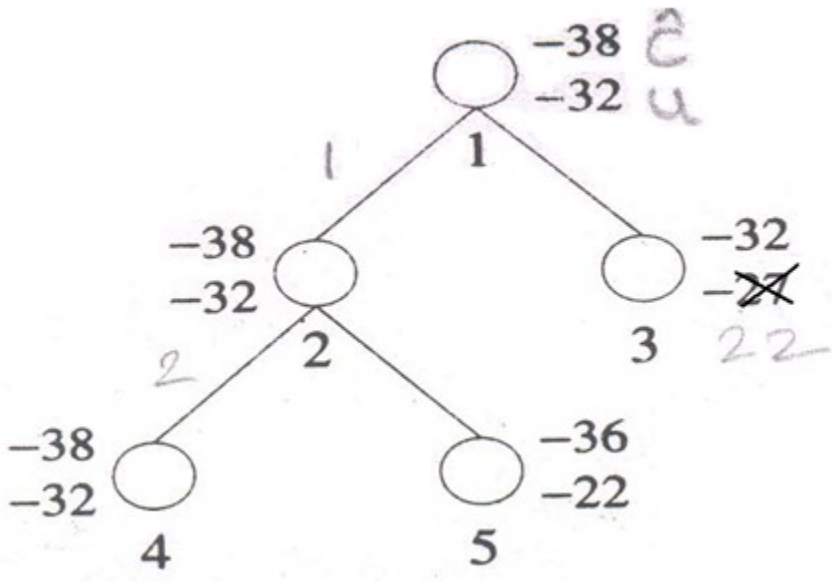
(p1, p2, p3, p4) = (10, 10, 12, 18)
(w1, w2, w3, w4) = (2, 4, 6, 9)
m = 15



Continued…

- $\hat{C}(4) = [-10 - 10] + [-12 - 18 * (3/9)] = -38$
- *u(4) = [-10 - 10] + (-12) = -32*
- $\hat{C}(5) = [-10 - 0] + [-12 - 18 * (7/9)] = -36$
- *u(5) = -10 - 12 = -22*

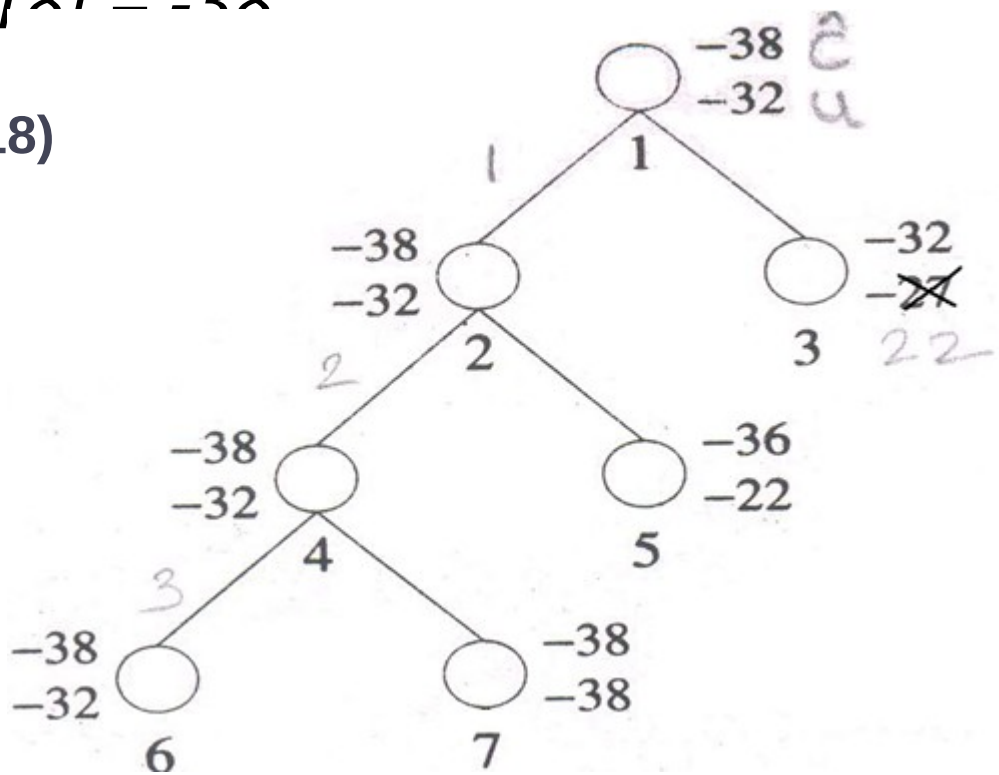
(p1, p2, p3, p4) = (10, 10, 12, 18)
(w1, w2, w3, w4) = (2, 4, 6, 9)
m = 15



Continued…

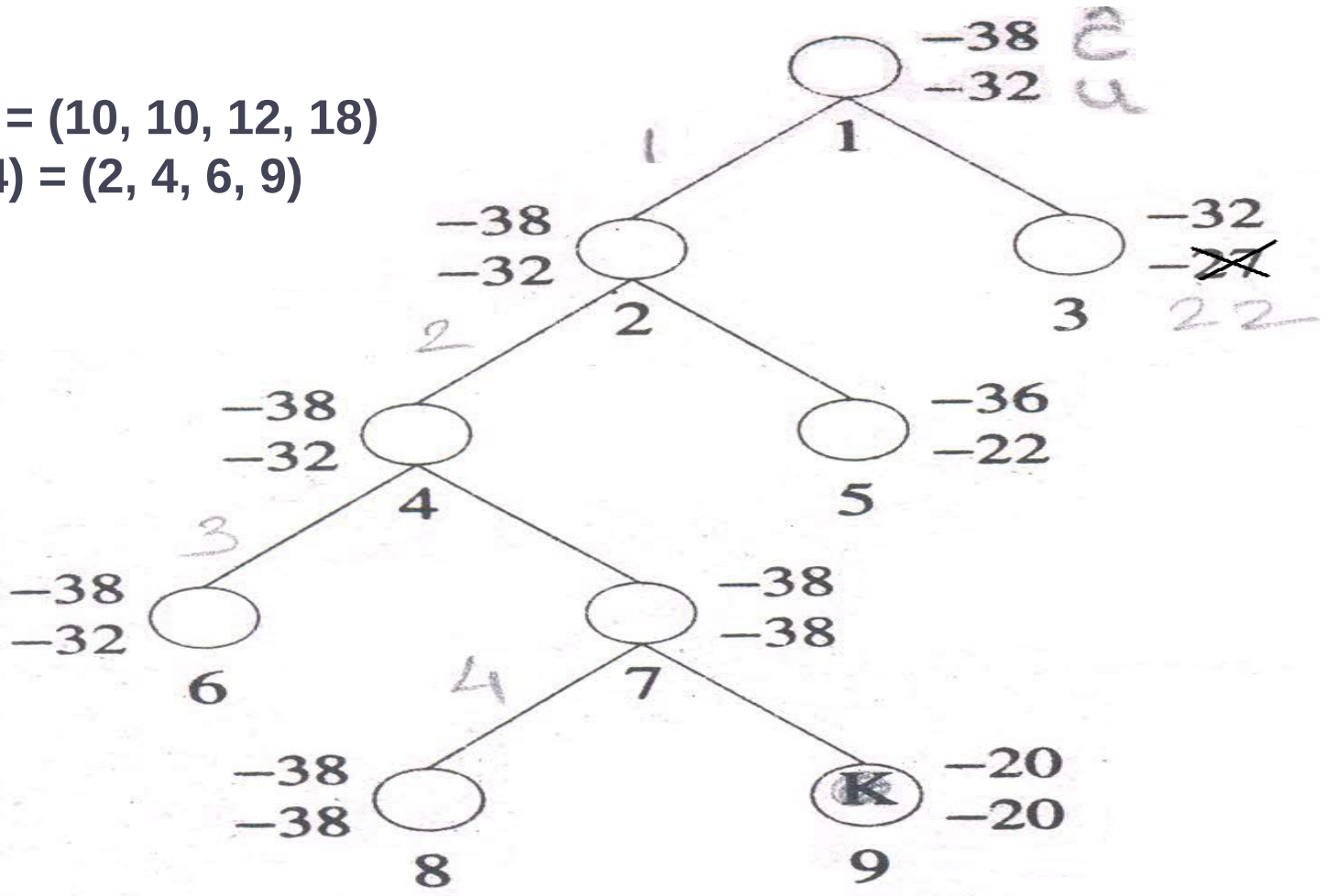
- Ĉ(6) =[-10 –10 - 12] + [- 18 *(3/9)] = -38
- *u(6) = [-10 - 10 - 12] + 0 = -32*
- Ĉ(7) =[-10 –10] + [-18] = -38
- *u(7) = [-10 - 10] + [-18] = -38*

(p1, p2, p3, p4) = (10, 10, 12, 18)
(w1, w2, w3, w4) = (2, 4, 6, 9)
m = 15



LC branch and bound solution

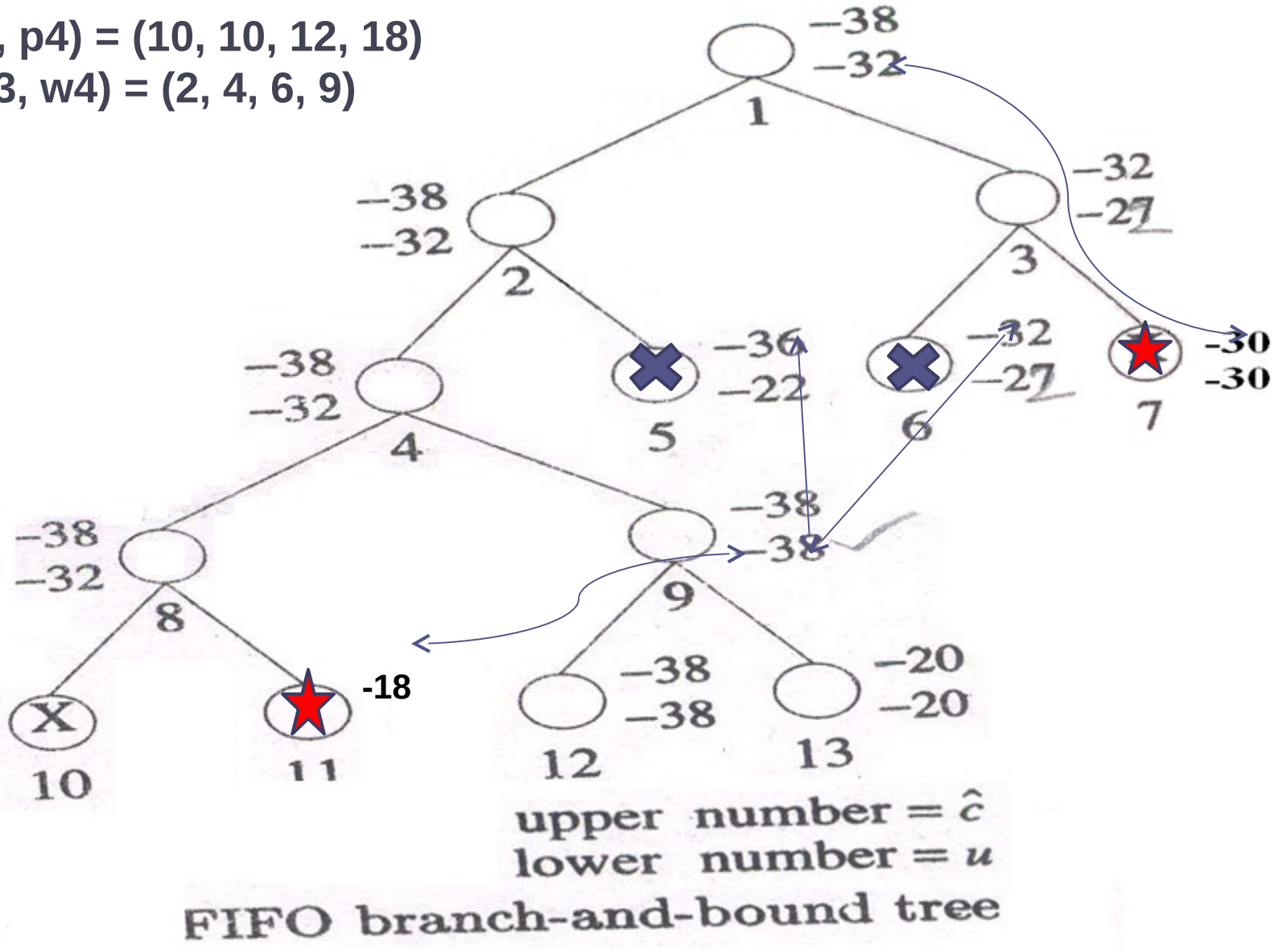
(p1, p2, p3, p4) = (10, 10, 12, 18)
(w1, w2, w3, w4) = (2, 4, 6, 9)
m = 15



LC branch-and-bound tree

FIFO branch and bound solution

(p1, p2, p3, p4) = (10, 10, 12, 18)
(w1, w2, w3, w4) = (2, 4, 6, 9)
m = 15



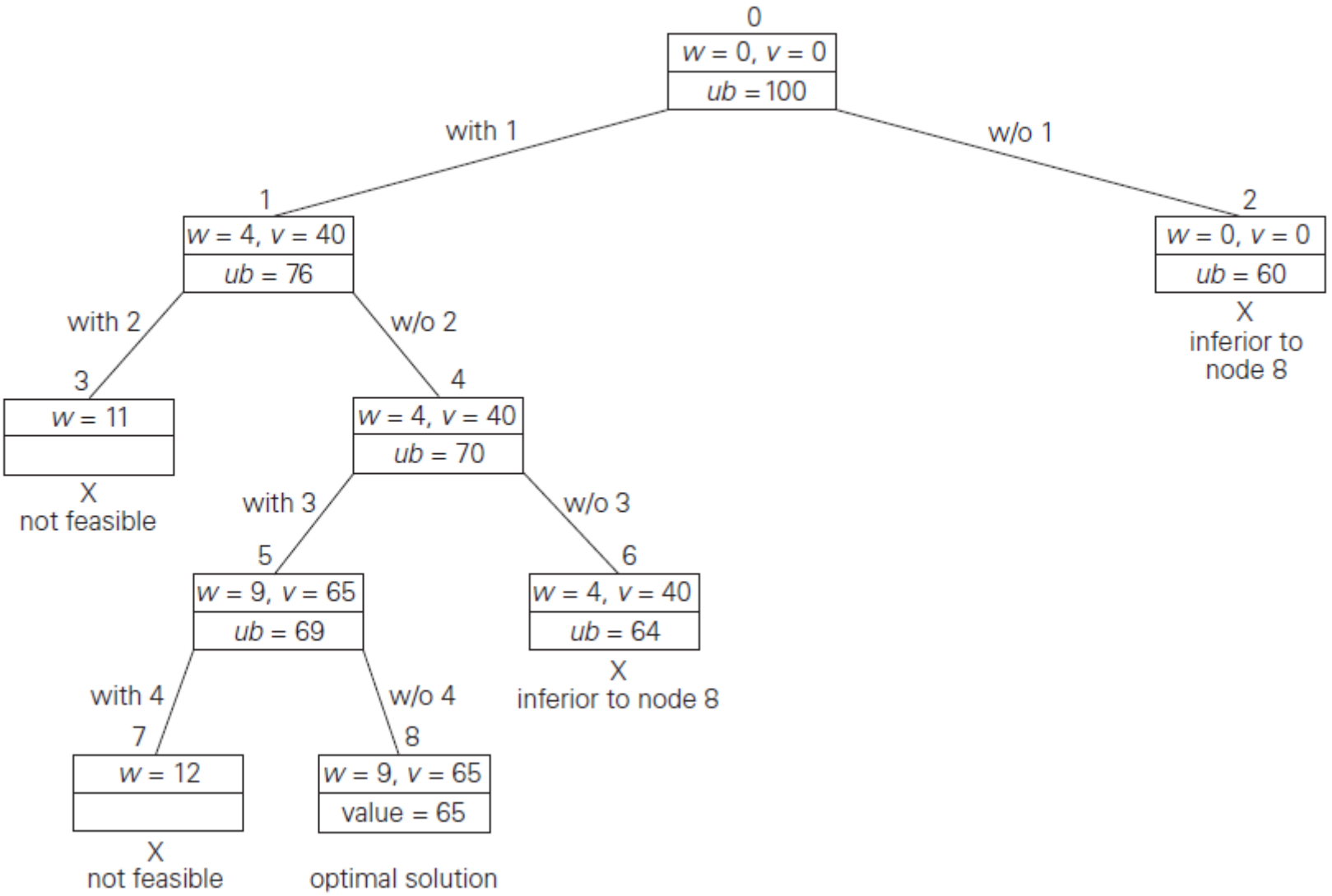
FIFO branch-and-bound tree

Example : 0/1 Knapsack with Branch and Bound

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.


0
w = 0, v = 0
ub = 100
with 1
w/o 1
1
w = 4, v = 40
ub = 76
2
w = 0, v = 0
ub = 60
X
inferior to node 8
with 2
w/o 2
3
w = 11
X
not feasible
4
w = 4, v = 40
ub = 70
with 3
w/o 3
5
w = 9, v = 65
ub = 69
6
w = 4, v = 40
ub = 64
X
inferior to node 8
with 4
w/o 4
7
w = 12
X
not feasible
8
w = 9, v = 65
value = 65
optimal solution