UNIT IV

Backtracking
Branch-And-Bound

(Reference Book - Horowithz Sahni)

Exercise: Dice rolls

- Write a method `diceRoll` that accepts an integer parameter representing a number of 6-sided dice to roll, and output all possible combinations of values that could appear on the dice.

`diceRoll(2);`

```
[1, 1] | [3, 1] | [5, 1]
[1, 2] | [3, 2] | [5, 2]
[1, 3] | [3, 3] | [5, 3]
[1, 4] | [3, 4] | [5, 4]
[1, 5] | [3, 5] | [5, 5]
[1, 6] | [3, 6] | [5, 6]
[2, 1] | [4, 1] | [6, 1]
[2, 2] | [4, 2] | [6, 2]
[2, 3] | [4, 3] | [6, 3]
[2, 4] | [4, 4] | [6, 4]
[2, 5] | [4, 5] | [6, 5]
[2, 6] | [4, 6] | [6, 6]
```

`diceRoll(3);`

```
[1, 1, 1]
[1, 1, 2]
[1, 1, 3]
[1, 1, 4]
[1, 1, 5]
[1, 1, 6]
[1, 2, 1]
[1, 2, 2]
   ...
[6, 6, 4]
[6, 6, 5]
[6, 6, 6]
```

Examining the problem

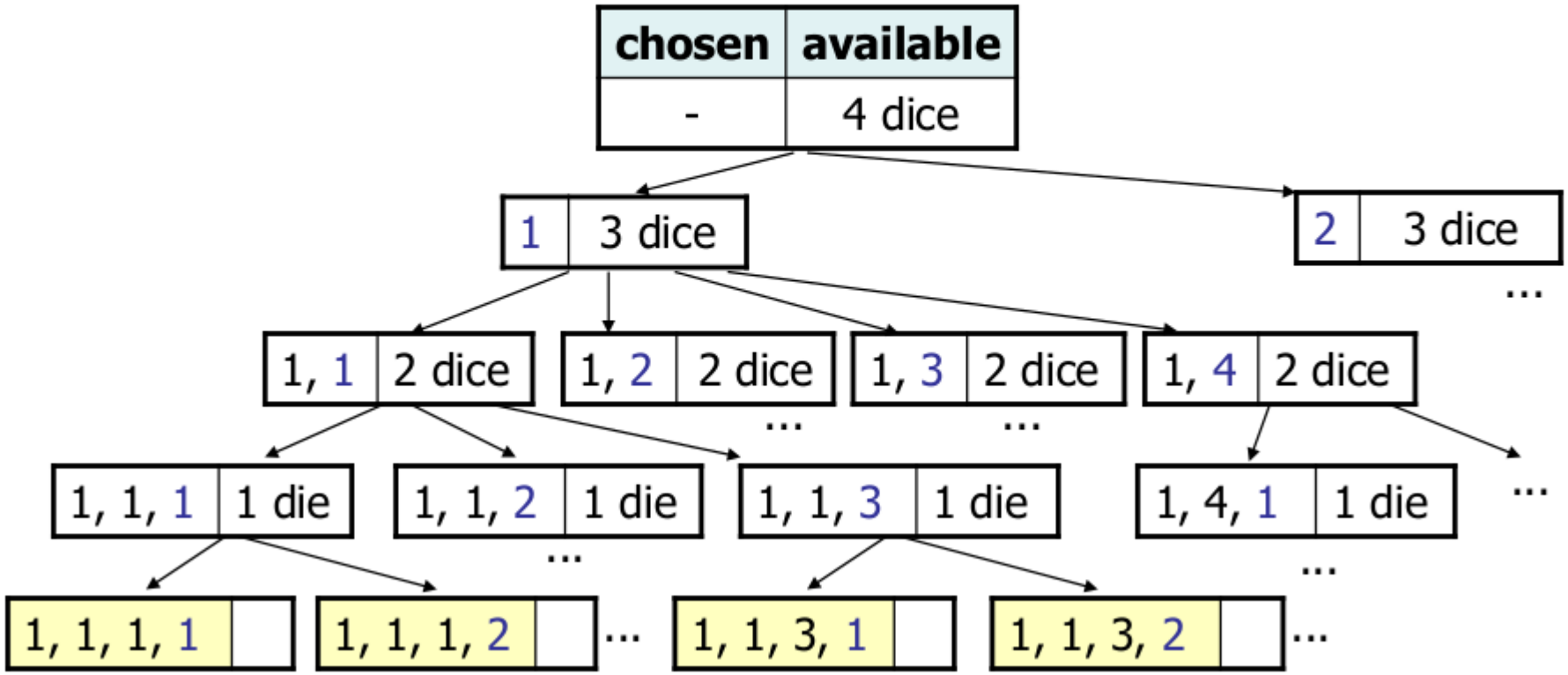
- We want to generate all possible sequences of values.

```
for (each possible first die value):
    for (each possible second die value):
        for (each possible third die value):
            ...
                print!
```

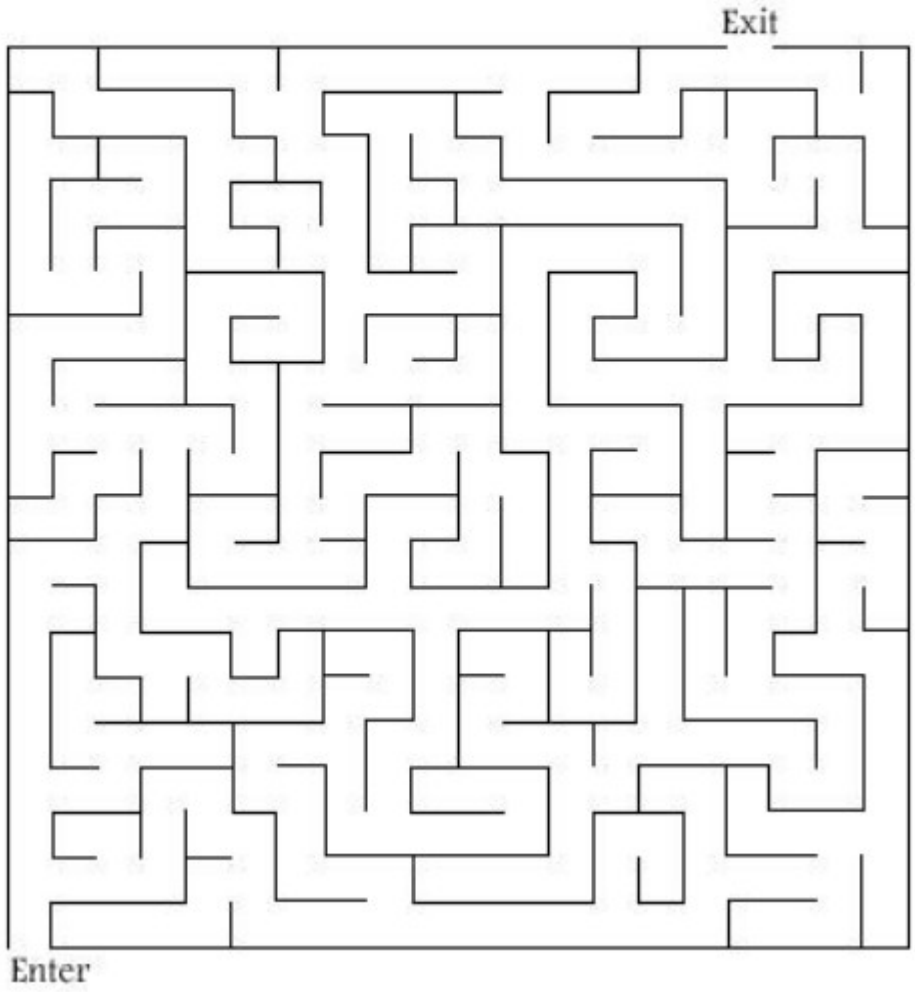
  - This is called a **depth-first search**

  - How can we completely explore such a large search space?

A decision tree



Backtracking



Backtracking

- **backtracking**: Finding solution(s) by trying partial solutions and then abandoning them if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths)
  - often implemented recursively

  Applications:
  - producing all permutations of a set of values
  - parsing languages
  - games: anagrams, crosswords, word jumbles, 8 queens
  - combinatorics and logic programming

Backtracking algorithms

A general pseudo-code algorithm for backtracking problems:

Explore(**choices**):

- if there are no more **choices** to make: stop.
- else:
  - Make a single choice **C**.
  - Explore the remaining **choices**.
  - Un-make choice **C**, if necessary. (backtrack!)

Exercise: Dice roll sum

- Write a method `diceSum` similar to `diceRoll`, but it also accepts a desired sum and prints only combinations that add up to exactly that sum.

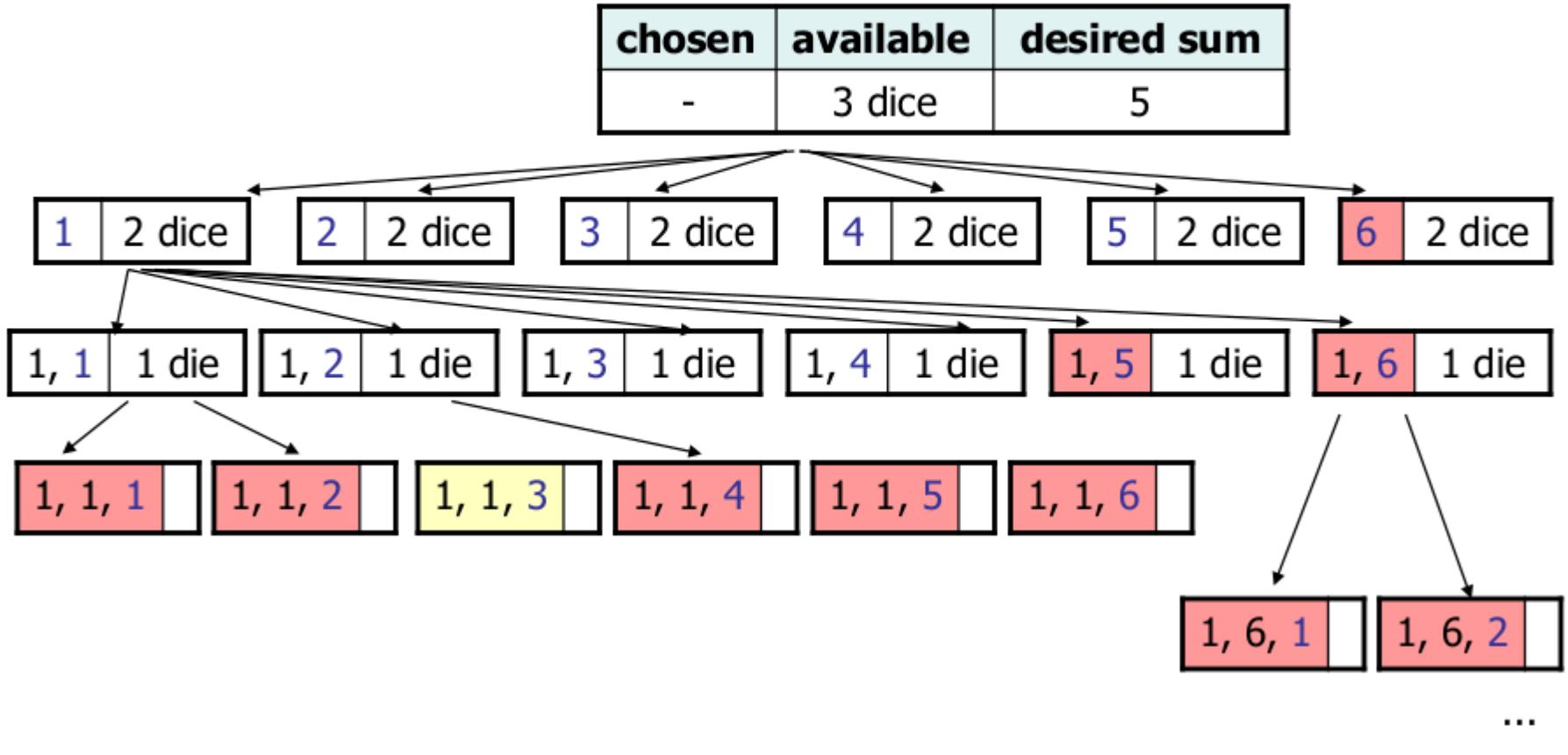
```
diceSum(2, 7);
```

```
[1, 6]
[2, 5]
[3, 4]
[4, 3]
[5, 2]
[6, 1]
```

```
diceSum(3, 7);
```

```
[1, 1, 5]
[1, 2, 4]
[1, 3, 3]
[1, 4, 2]
[1, 5, 1]
[2, 1, 4]
[2, 2, 3]
[2, 3, 2]
[2, 4, 1]
[3, 1, 3]
[3, 2, 2]
[3, 3, 1]
[4, 1, 2]
[4, 2, 1]
[5, 1, 1]
```

New decision tree



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

Recursive backtracking algorithm

```
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] in T(x[1], …, x[k-1]) {
            if (B_k (x[1], x[2], …, x[k])) == True{
               if (x[1], x[2], …, x[k] is a path to an answer node)
                     print 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 )
                    print 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

The N Queens Problem

- Place N Queens on an N by N chessboard so that none of them can attack each other
- Number of possible placements?
- In 8 x 8

  64 * 63 * 62 * 61 * 60 * 59 * 58 * 57
  = 178,462, 987, 637, 760 / 8!
  = 4,426,165,368

  $$\binom{n}{k} = \frac{n \cdot (n-1) \cdots (n-k+1)}{k \cdot (k-1) \cdots 1} = \frac{n!}{k!(n-k)!} \quad \text{if } 0 \leq k \leq n \qquad (1)$$

  n choose k

  – How many ways can you choose k things from a set of n items?

  – In this case there are 64 squares and we want to choose 8 of them to put queens on

- For a safe solution, how many queens can be placed in a given column?

A. 0

B. 1

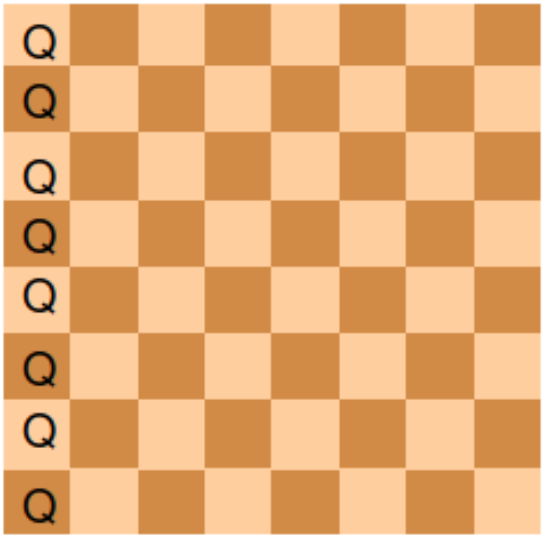
C. 2

D. 3

E. Any number

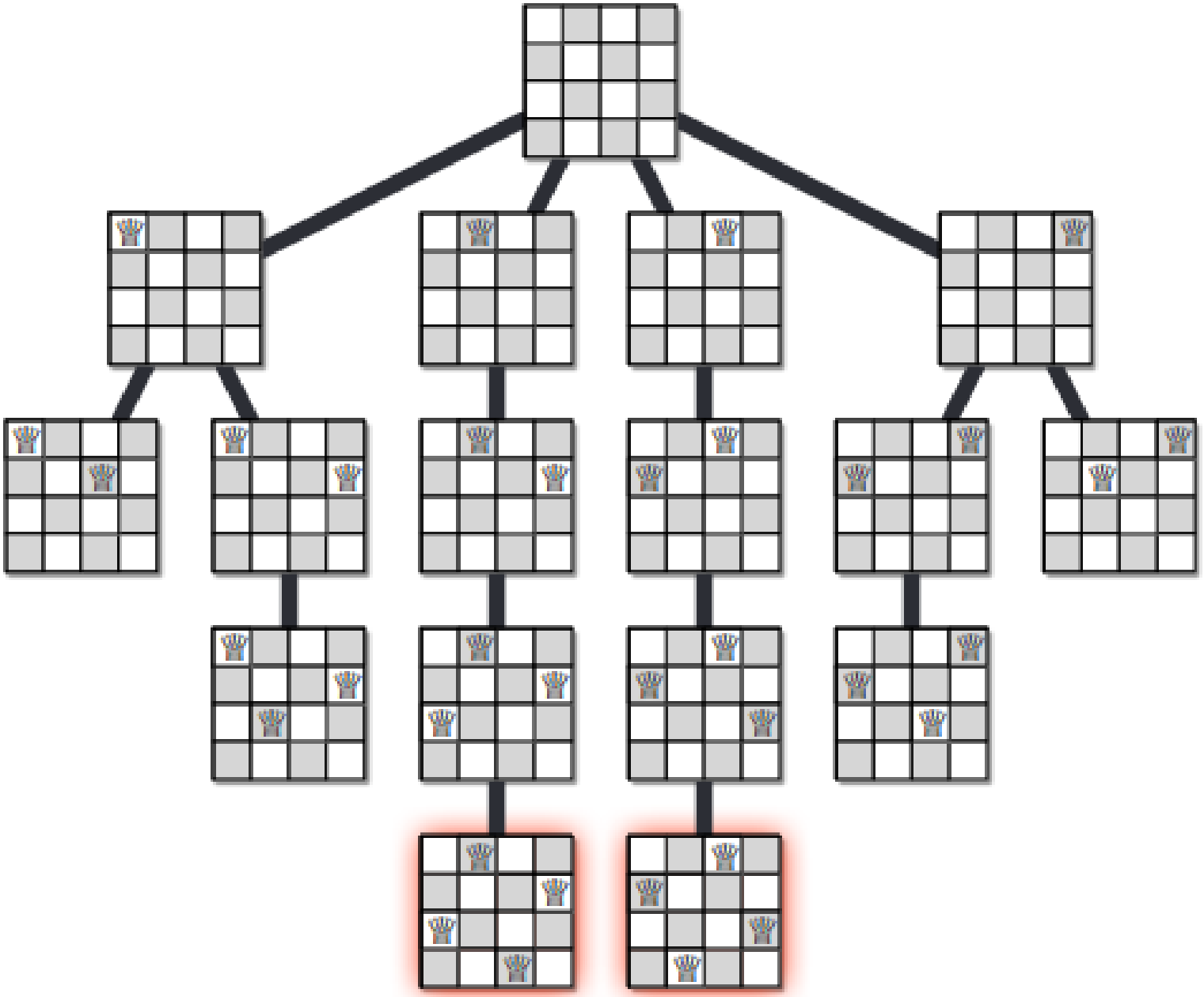
Reducing the Search Space

- The previous calculation includes set ups like this one

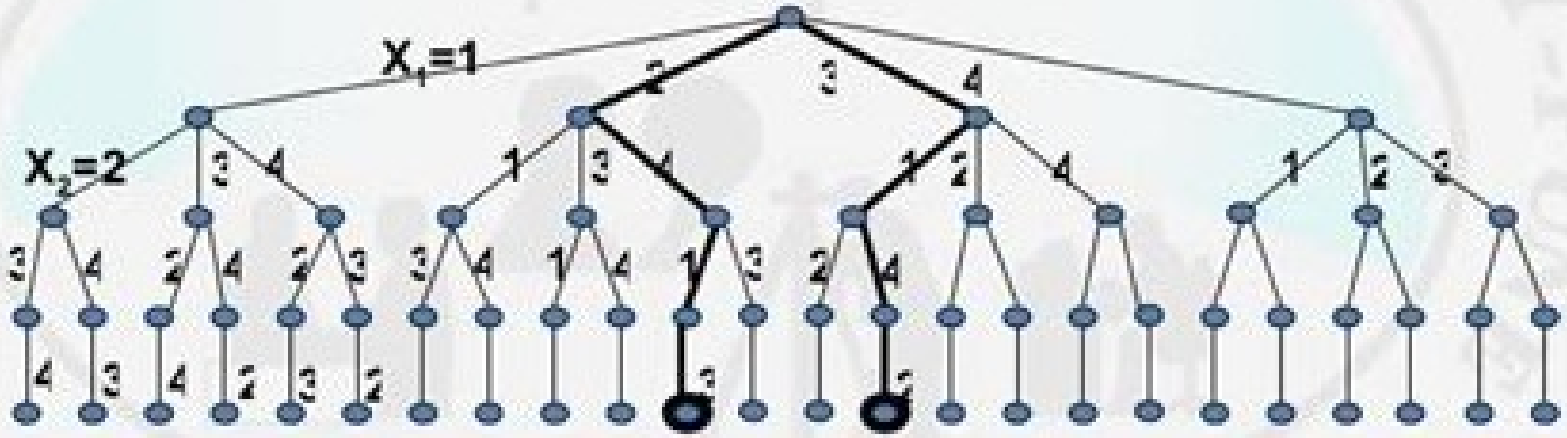


- Includes lots of set ups with multiple queens in the same column
- How many queens can there be in one column?
- Number of set ups
  8 * 8 * 8 * 8 * 8 * 8 * 8 * 8 = 16,777,216
- We have reduced search space by two orders of magnitude by applying some logic

- We place queens on the board one row at a time, starting with the top row.
- To place the r^{th} queen, we methodically try all n squares in row r from left to right in a simple for loop.
- If a particular square is attacked by an earlier queen, we ignore that square; otherwise, we tentatively place a queen on that square and recursively grope for consistent placements of the queens in later rows.



- State Space Tree for 4-Queens Problem



$(x_1, x_2, x_3, x_4)=(2, 4, 1, 3)$

- #leaf nodes : n!=4!
- DFS : "Backtrack" at dead ends – 4! leaf nodes (n!)

Algorithm – The n-queen problem

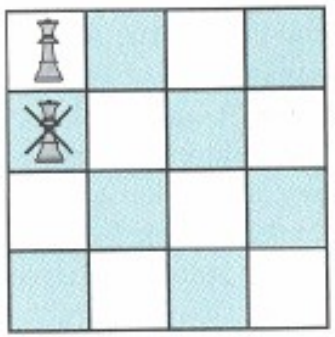
```
NQueen(k, n) // Prints all Solution to the n-queens problem
{
    for i = 1 to n do{
        if Place(k, i) then{
            x[k] = i;
            if (k = n) then print(x[1:n]);
            else Nqueen(k+1, n);
        }
    }
}
```

```
Place(k, i)// Try placing k^th queen at i^th position
{
    for j = 1 to k - 1 do
// in the same column or in the same diagonal of
previous queens
        if((x[j] = i) or (abs(x[j] – i) = abs(j - k )))
            return false;
    return true;
}
```

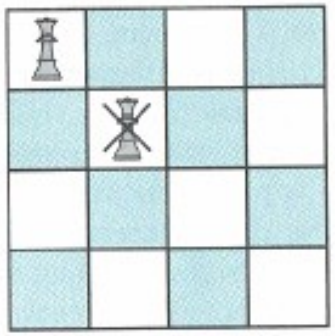
try to place 1st queen
SUCCESS
    try to place 2nd queen
    SUCCESS
        try to place 3rd queen
        FAIL
    try to place 2nd queen in another position
    SUCCESS
        try to place 3rd queen
        SUCCESS
            try to place 4th queen

1st queen
i=1
Can place? Yes. Cool, recurse.
    2nd queen
    i=1
    Can place? No
    i=2
    Can place? No
    i=3
    Can place? Yes. Cool, recurse.
        3rd queen
        i=1
        Can place? No
        i=2
        Can place? No
        ... (can be placed at no position)
        FAIL
        back to 2nd queen
    i=4
    Can place? Yes. Cool, recurse.
        3rd queen
        i=1
        Can place? No
        ....

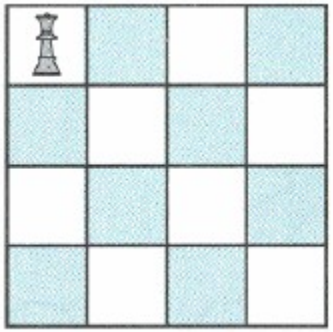
Example



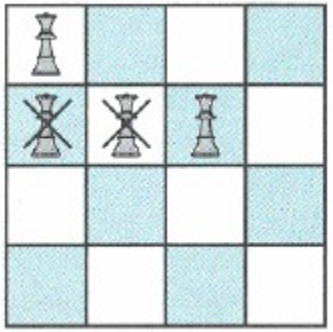
(a)



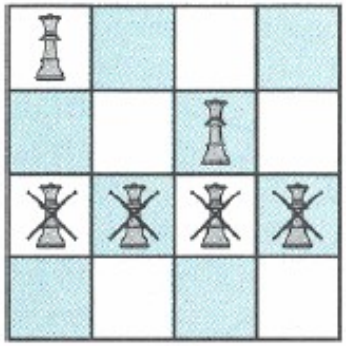
(b)



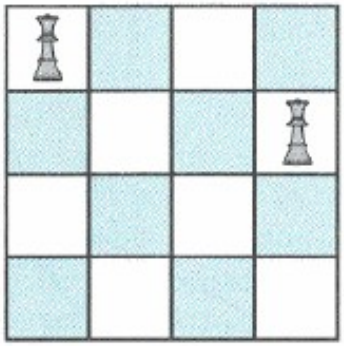
(a)



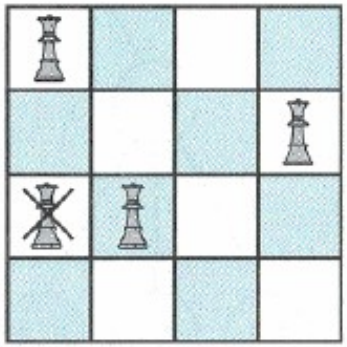
(b)



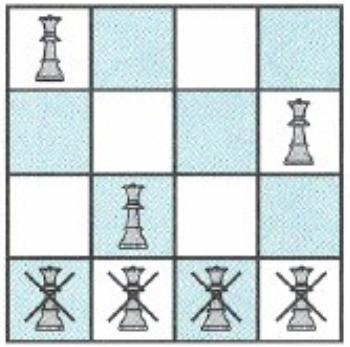
(c)



(d)

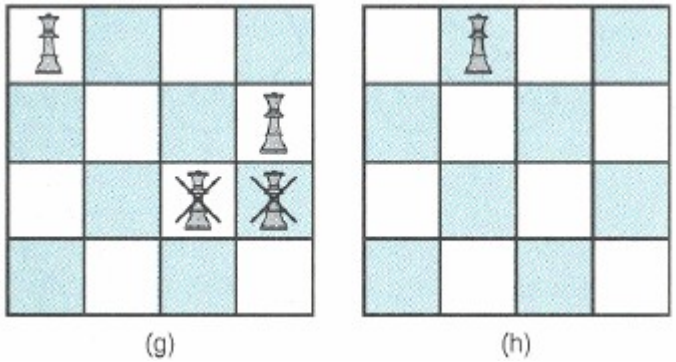


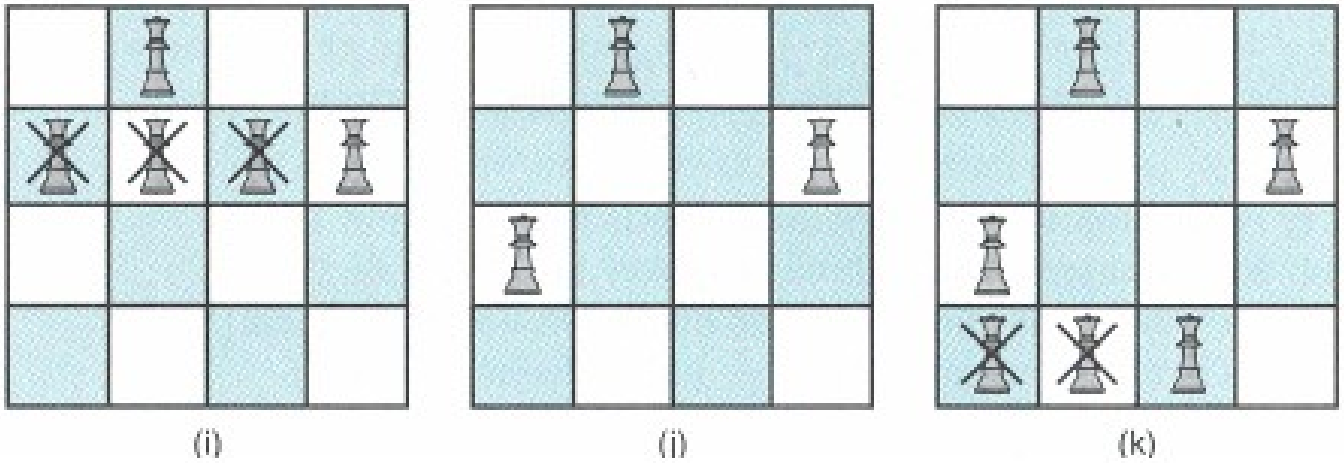
(e)



(f)

Continued...



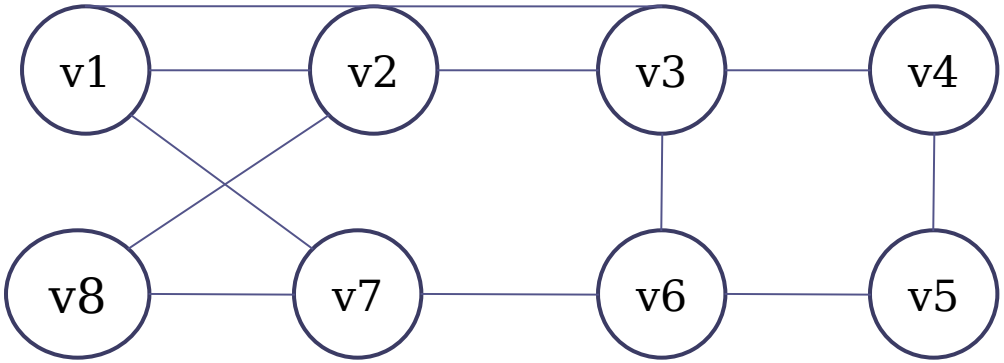


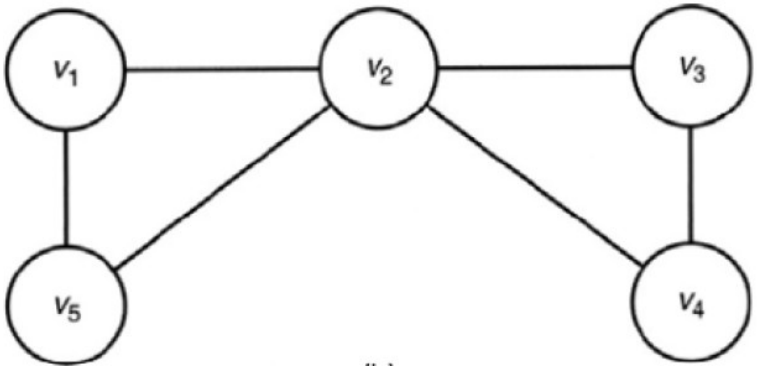
Hamiltonian Cycles

- 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

The Hamiltonian Circuits Problem

- **Conditions to check**

1)The node should not be repeated

2)The previous and current node should be connected, refer adjacency matrix

3)If all nodes are visited, there should be a path to the starting node, refer adjacency matrix

X[0,0,0,0] // initially

X[**1**,0,0,0] // first call

X[1,1,0,0] // violates requirement same node

X[1,**2**,0,0] // fulfils all requirement

X[1,2,1,0] // violates requirement same node

X[1,2,2,0] // violates requirement same node

X[1,2,**3**,0] // fulfils all requirement

X[1,2,3,1] // violates requirement same node

X[1,2,3,2] // violates requirement same node

X[1,2,3,3] // violates requirement same node

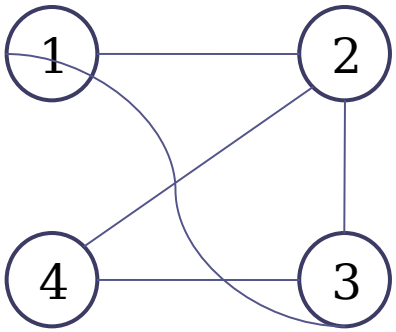
X[1,2,3,4] // violates requirement that last node is not connected to first

X[1,2,**4**,0] // try next node in previous place, fulfils all requirement

X[1,2,4,1] // violates requirement same node

X[1,2,4,2] // violates requirement same node

X[1,2,4,**3**] // fulfils all requirements and prints the solution



Continued...

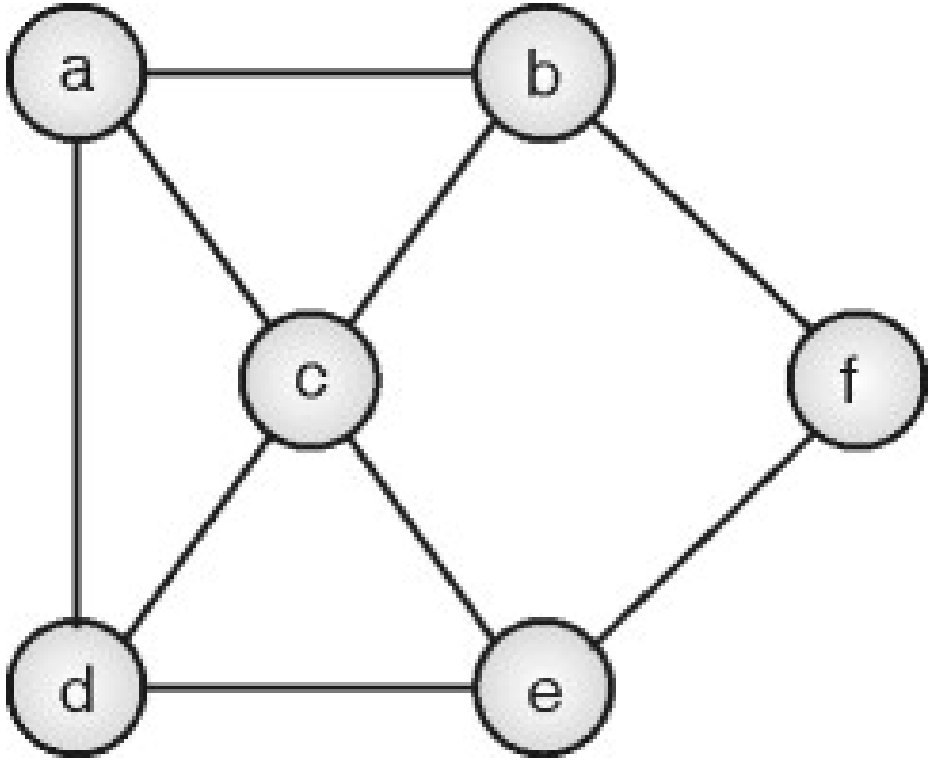
// This algorithm uses the recursive formulation of backtracking to find all the hamiltonion cycles of a graph. The graph is stored as an *adjacency matrix G[1:n, 1:n]*. All cycles begins at node 1. if x[k]=0, then no vertex has yet been assigned to x[k].

Algorithm Hamiltonian(k)

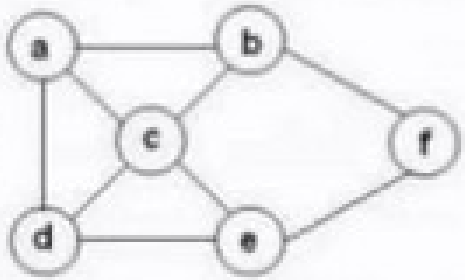
```
{                       //Set x[2:n] to zero and x[1] to 1. Execute
  repeat {              Hamiltonian(2).
        //generate values for x[k]
        NextValue (k); // assign a legal next value to x[k]
        // if not assigned, backtrack to previous node
        if ( x[k] = 0 ) then return;
        if (k = n) then print ( x[1:n]);
        else Hamiltonian(k+1);
  }until(false);
}
```

x[1: k-1] is a path of k-1 distinct vertices, if x[k]=0, then no vertex has 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]. *adjacency matrix G[1:n, 1:n]*

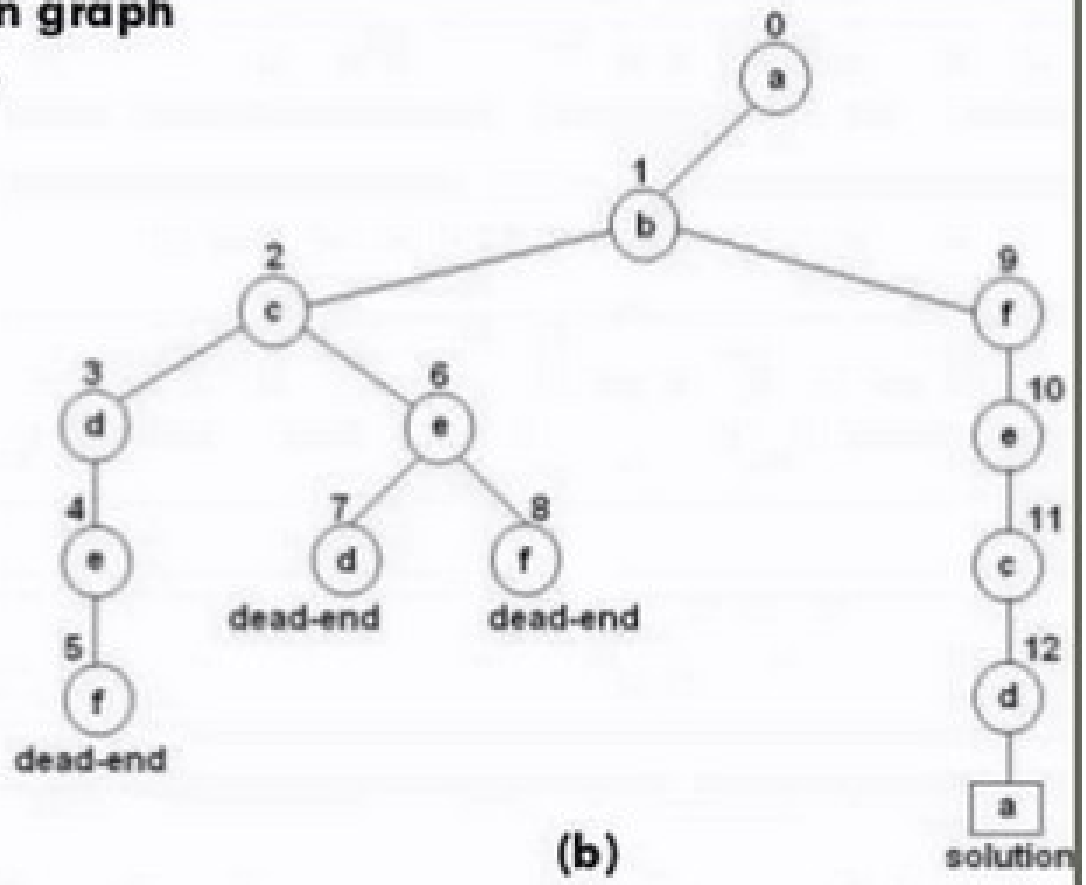
```
Algorithm NextValue(k) {
   repeat {
            x[k] = (x[k] + 1) mod (n+1) // next vertex
            if ( x[k] = 0) then return
            if (G(x[k–1], x[k]) ≠ 0 ) then { // is there an edge?
                     for j = 1 to k–1 do if (x[j] = x[k]) then break;
                     // check distinctness, i.e. if already visited
            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) ;
}
```


a
b
c
f
d
e

For example consider the given graph and evaluate the mechanism:-

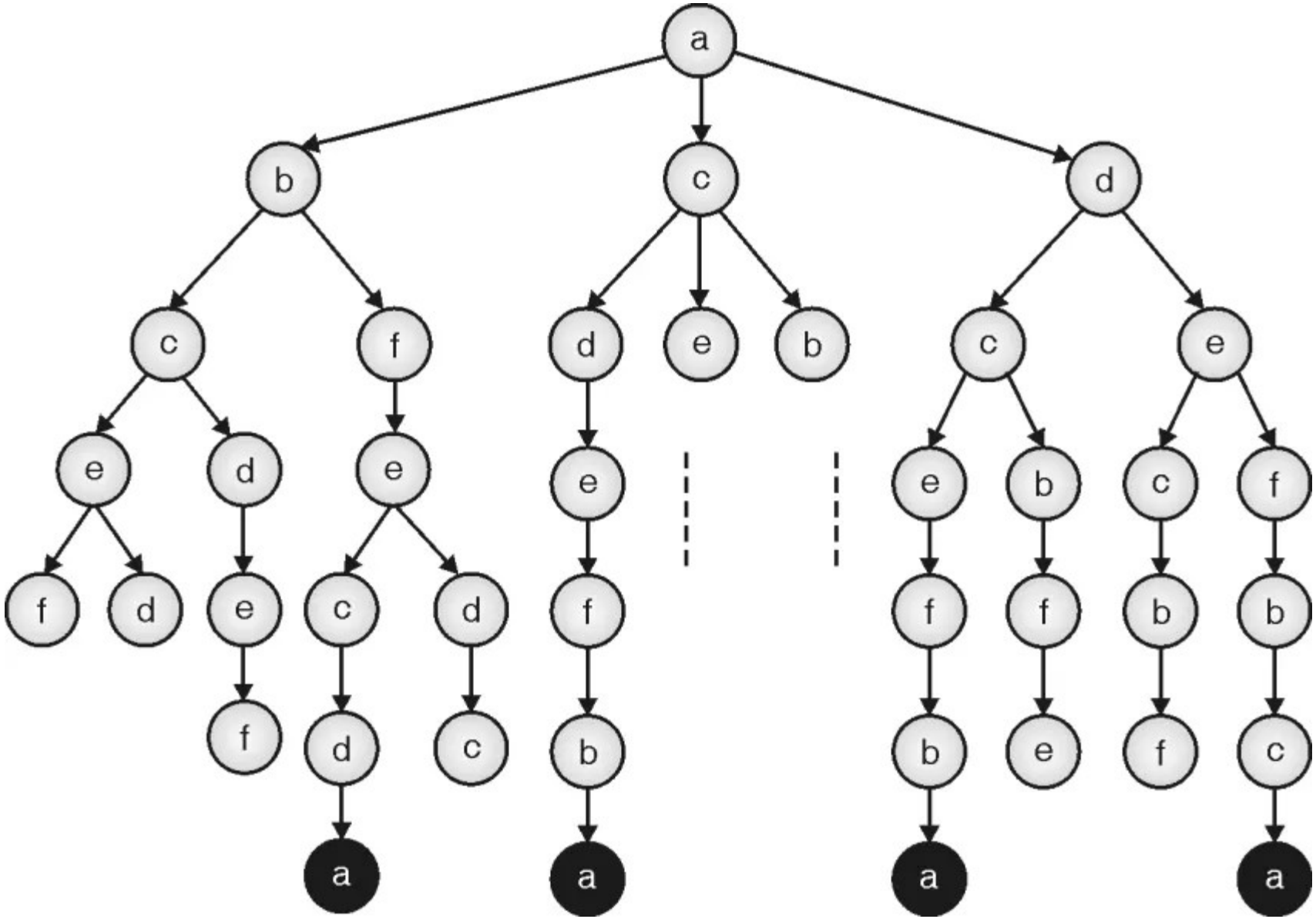


(a)



(b)

Figure:
- (a) Graph.
- (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order the order in which nodes are generated.


a
b
c
d
c
f
d
e
b
c
e
e
d
e
e
b
c
f
f
d
e
c
d
f
f
f
b
b
f
e
f
c
d
c
b
b
e
f
c
a
a
a
a

Application

A **fault-tolerant network** is one that can continue to function if an intermediate device or path fails.

In a network, if Hamiltonian cycles exist, the fault tolerance is better.

Knapsack backtracking

- We are given n items 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 items

  w = weight of each item

  p = profit of each item, 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 $\Sigma_{1 \le i \le n} \mathbf{w_i x_i \le m}$

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

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 *non-promising if it* cannot lead to a feasible (or optimal) solution, otherwise it is *promising*.

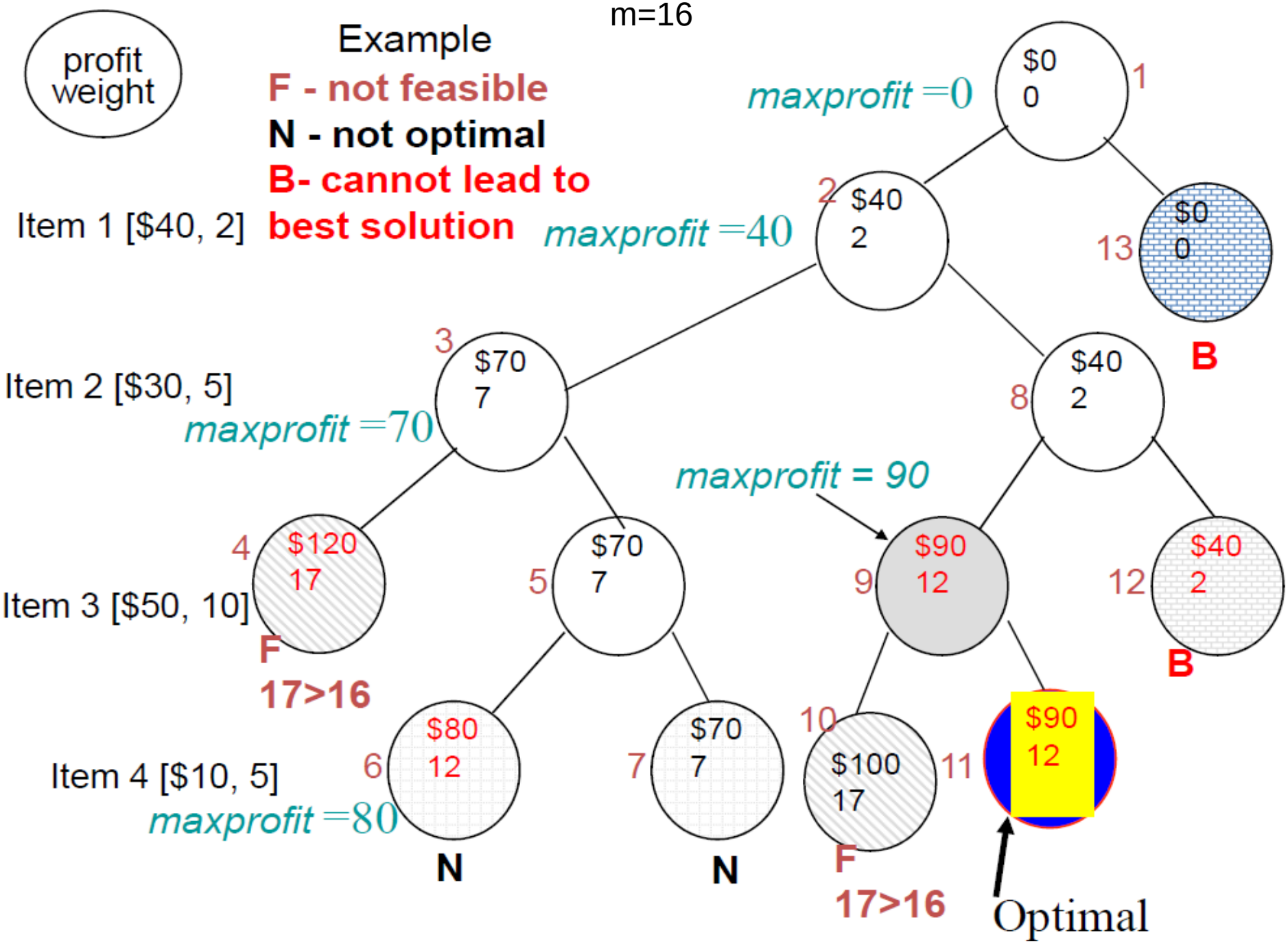
- **Main idea**: Backtracking consists of doing a **DFS** of the state space tree, checking whether each node is *promising* and if the node is *non-promising* backtracking to the node's parent.

- **Algorithm variables**
- **cp** : the current profit, **cw** : the current wt total, **k** : the index being processed, and **m** : the knapsack size, **fw** : Final knapsack weight, **fp** : Final earned profit
- The x's constitute a zero-one valued vector.

Example

- Suppose *n = 4, m = 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


m=16
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
1 $0 0
2 $40 2
13 $0 0
B
3 $70 7
8 $40 2
4 $120 17
F
17>16
5 $70 7
9 $90 12
12 $40 2
B
6 $80 12
N
7 $70 7
N
10 $100 17
F
17>16
11 $90 12
Optimal

Backtracking solution to the fractional knapsack problem

```
Algorithm Bknap( k, cp, cw) {
   if ( cw + w[k] ≤ m)  // generate left child {
            y[k] = 1 // k is added
            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 )  //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; // current profit
   c = cw; // current wt total
   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]; //fraction
   }
   return b;
}
```

Back

Worst-case time complexity

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

Backtracking

- Construct the state space tree:
  - *Root represents an initial state*
  - *Nodes reflect specific choices made for a solution's components.*
    - Promising and nonpromising nodes
    - leaves
- Explore the state space tree using depth-first search
- "Prune" non-promising nodes
  - *dfs stops exploring subtree rooted at nodes leading to no solutions and...*
  - *"backtracks" to its parent node*

Branch – bound

Branch and Bound

- In backtracking, we used **depth-first search with pruning** to traverse the state space tree.

- 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 Technique

- The Branch and Bound technique is a problem solving strategy, which is most commonly used in optimization problems, where the **goal is to minimize** a certain value.
- The optimized solution is obtained by means of a state space tree. (A state space tree is a tree where the solution is constructed by adding elements one by one, starting from the root. Note that the root contains no element)
- This method is best when used for combinatorial problems with exponential time complexity, since it provides a more efficient solution to such problems.

The Algorithm: Branch and Bound Technique

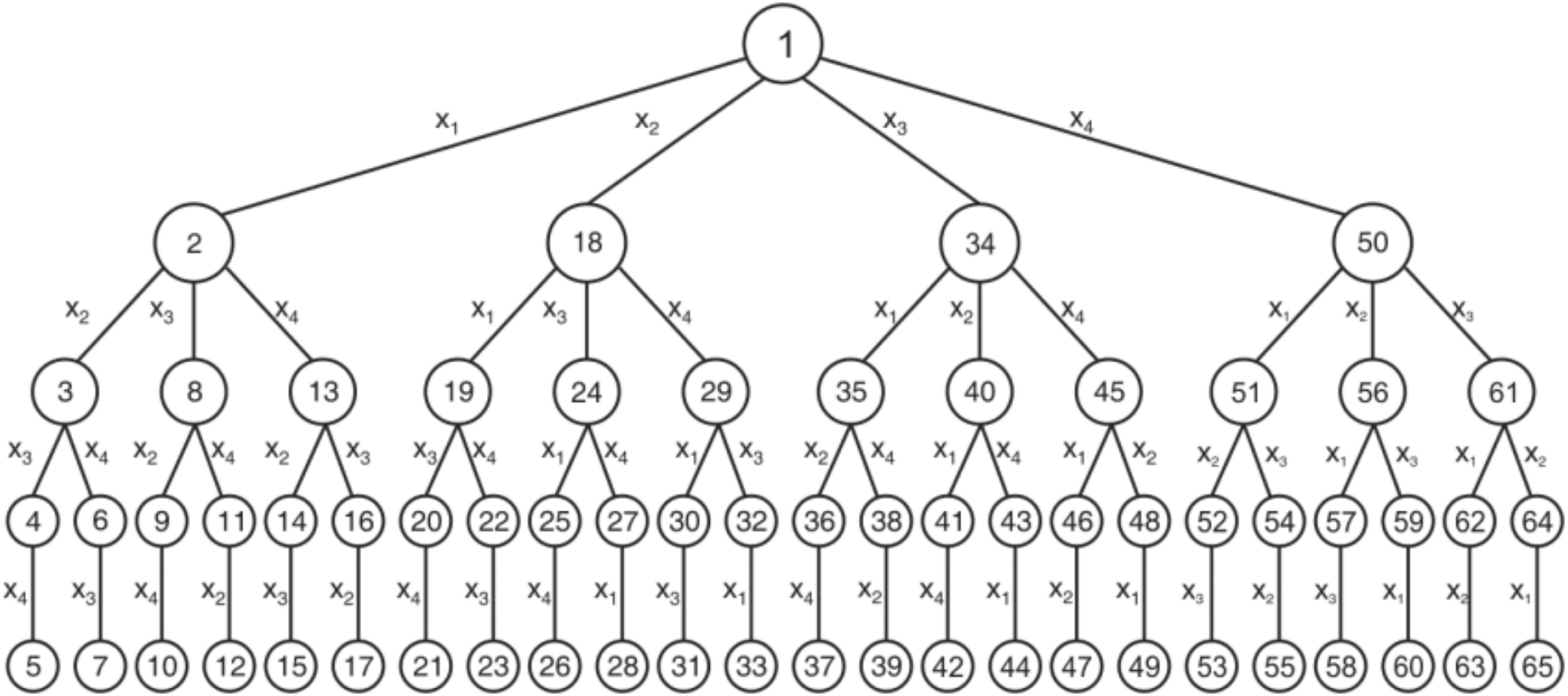
- In this technique, the first step is to create a function **U** (which represents an upper bound to the value that node and its children shall achieve), that we intend to minimize.
- We call this function the **objective function**. Note that the branch and bound technique can also be used for maximization problems, since multiplying the objective function by -1 coverts the problem to a minimization problem.
- Let this function have an initial maximum value, according to the conditions of the given problem. Also, let U_0 be the initial value of U. We also calculate a cost function C which will give us the exact value that the particular node shall achieve.

Types of Solutions

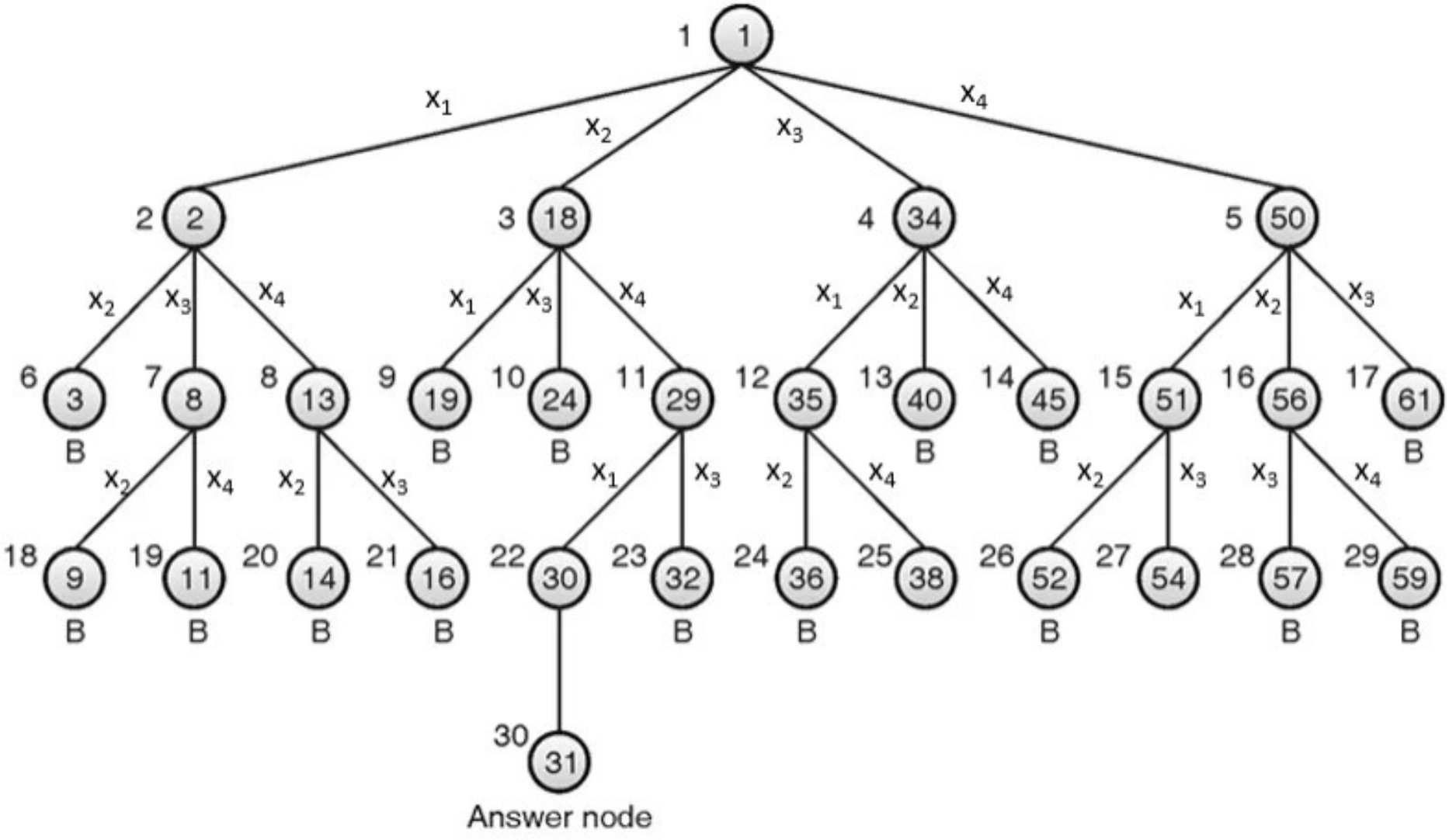
For a branch and bound problem, there are two ways in which the solution can be represented:

- **Variable size solution:** This solution provides the subset of the given set that gives the optimized solution for the given problem. For example, if the we are to select a combination of elements from {A, B, C, D, E} that optimizes the given problem, and it is found that A, B, and E together give the best solution, then the solution will be {A, B, E}.
- **Fixed size solution:** This solution is a sequence of 0s and 1s, with the digit at the i^{th} position denoting whether the i^{th} element should be included or not. Hence, for the earlier example, the solution will be given by {1, 1, 0, 0, 1}.

State space tree for backtracking (4 column options, x_i, exist for each queen, each row in the tree represents each queen)



State space tree for branch and bound (Each row in the tree represents each queen, 4 column options, x_i, exist for each queen)



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 pi** 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.

Job sequencing with deadline and penalty

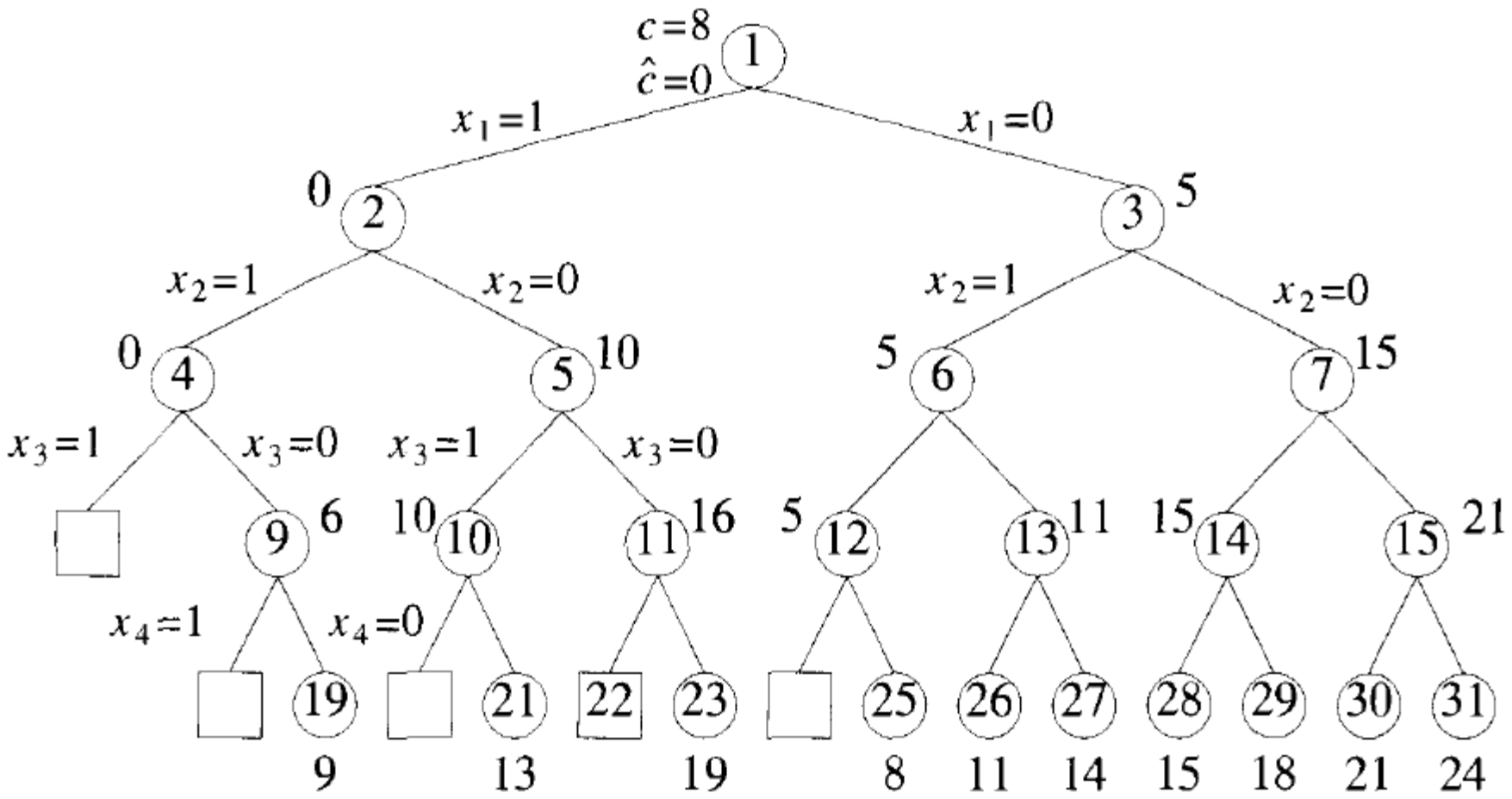
job_index	1	2	3	4
Penalty (cost)	5	10	6	3
Deadline (d)	1	3	2	1
Processing time (t)	1	2	1	1

Terminology

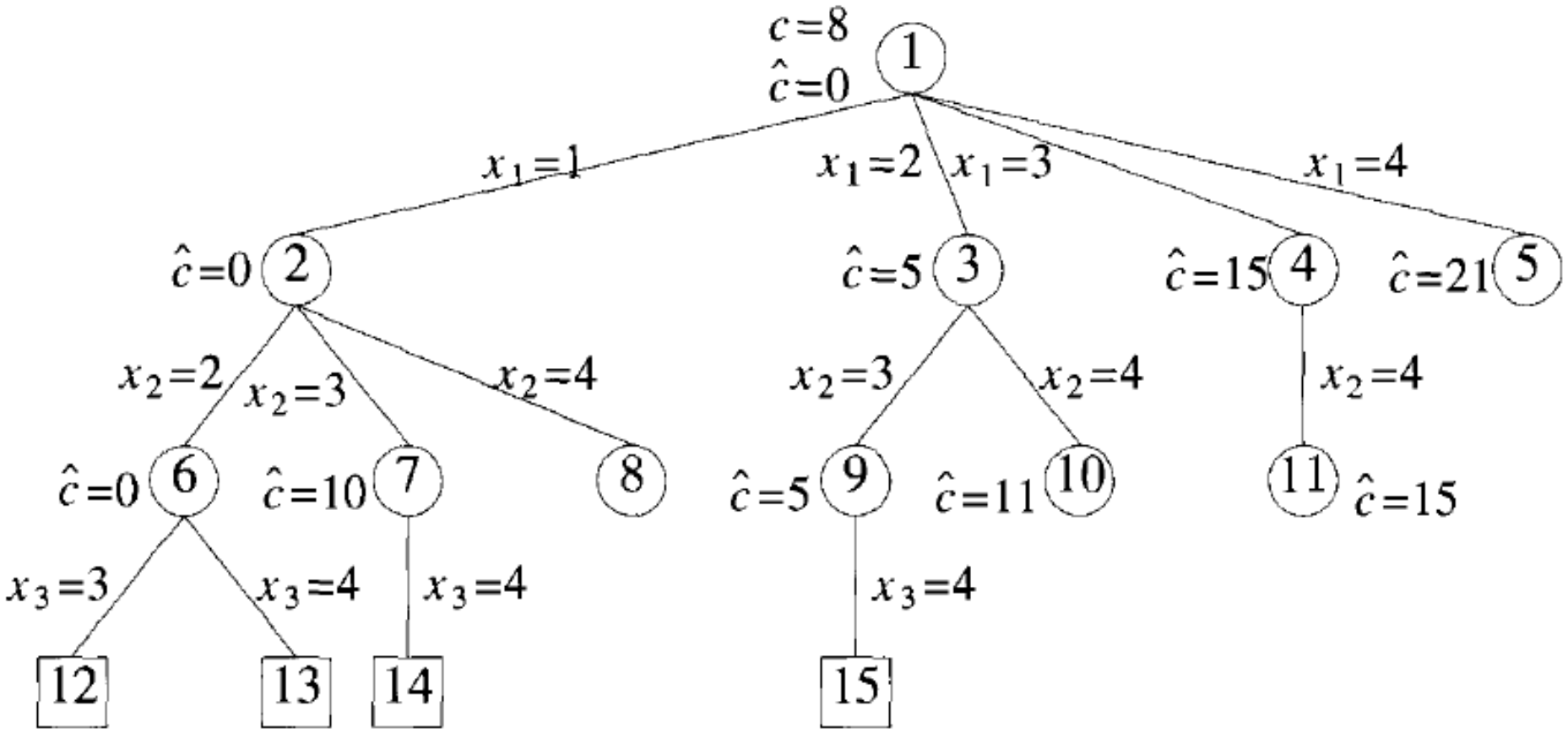
- Square nodes represent infeasible subsets.
- All non-square nodes are answer nodes.
- For any circular node x, **c(x)** is the minimum penalty corresponding to any node in the sub-tree with root x.
- The value of c(x)=infinity for a square node.
- It is obvious that, c(1) is the penalty corresponding to an optimal selection J.
- **$\hat{C}(x)$** is the bound function
- $\hat{C}(x)$ corresponds to the summation of penalties of the missed jobs in sequence.

State space tree corresponding to fixed tuple size formation

$c(1) = 8$, $c(2) = 9$, $c(5) = 13$, and $c(6) = 8$.



State space tree corresponding to variable tuple size formation



Here Node 8 = Node 22 in last slide i.e. infeasible

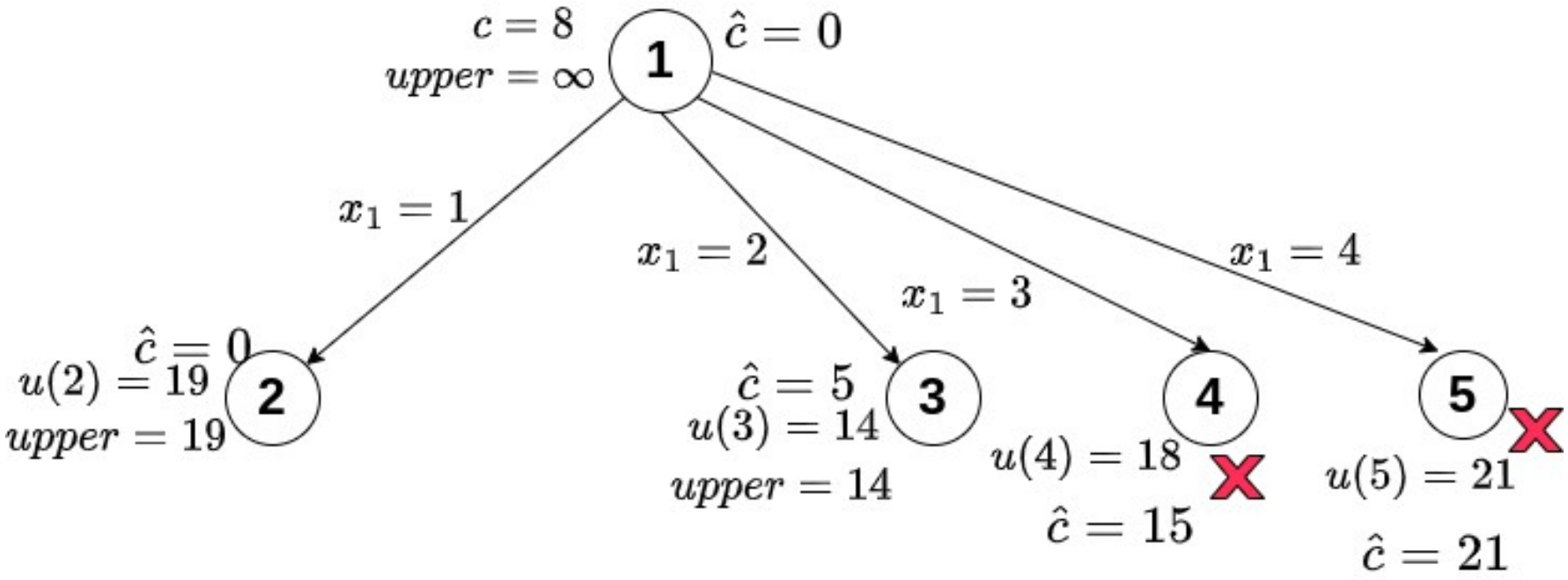
$c(3) = 8,\ c(2) = 9,$ and $c(1) = 8$

FIFO branch and bound

Terminology and variable definitions

- The variable, **u(x)**, is updated by summing the penalties of all other jobs.
- **upper** is updated to the value of lowest u(x).
- Initially, upper is set to infinity.
- **$\hat{C}(x)$** gives the sum of penalties not considered till now in the sequence.
- For any circular node x, **c(x)** is the minimum penalty corresponding to any node in the sub-tree with root x.
- The value of c(x)=infinity for a square node.

FIFO Branch and Bound

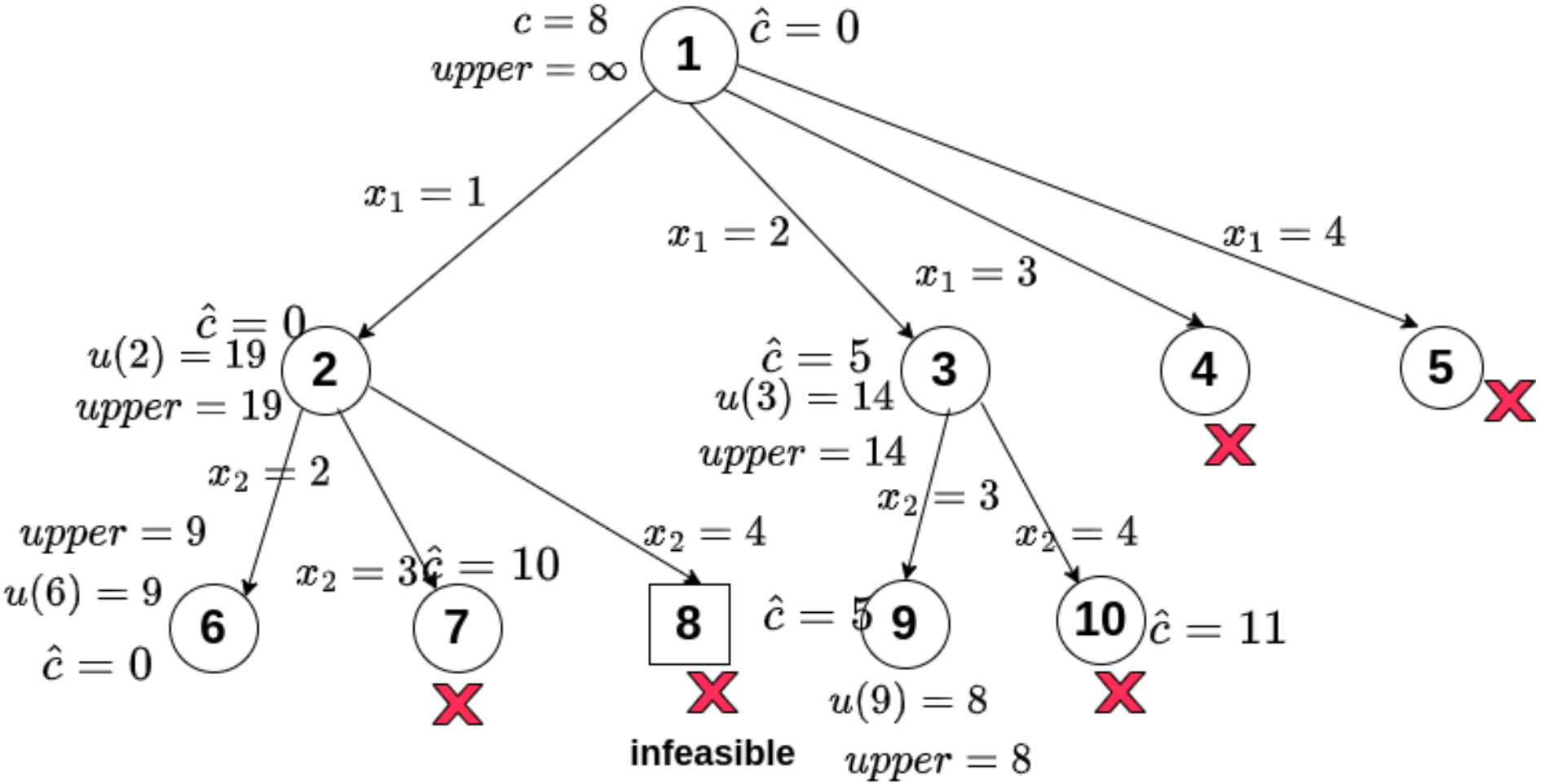


$upper = sum\ of\ penalties\ of\ other\ jobs$

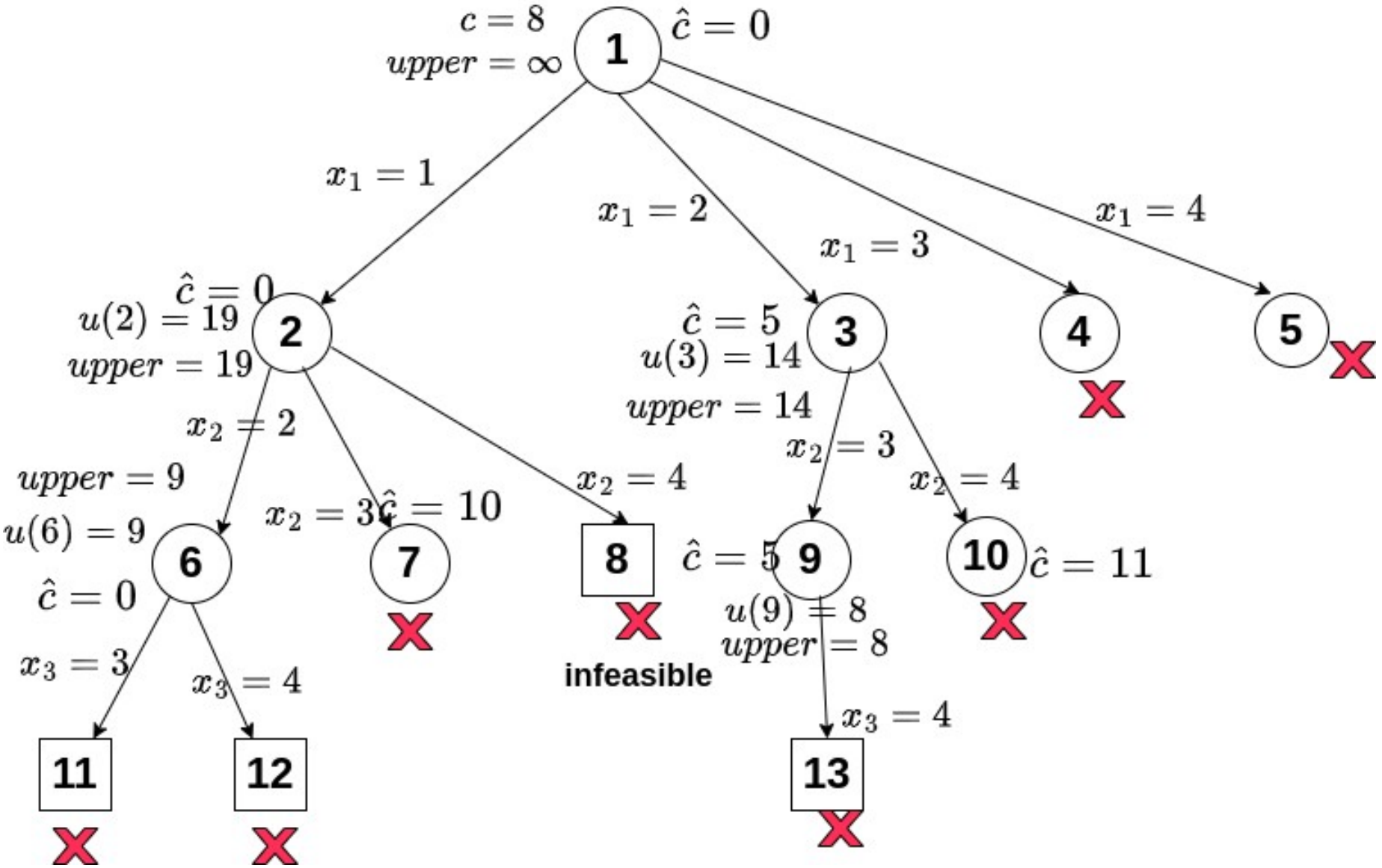
$\hat{c}(4) > upper$ so KILL the node

$\hat{c}(5) > upper$ so KILL the node

FIFO Branch and Bound



FIFO Branch and Bound



LC (least-cost) Branch and Bound

- In this method, after the children of a particular node have been explored, the next node to be explored would be that node out of the unexplored nodes which has the least cost.
- LC branch and bound processes a live node with lowest cost, i.e., lowest $\hat{C}(x)$.
- So, in the FIFO example on previous slide, node 6 will be processed instead of node 3 as $\hat{C}(6) < \hat{C}(3)$.
- Rest of the tree structure remains same.

Algorithm for Least Cost Branch and Bound

```
typedef struct
{
list_node_t * next;
list_node_t  *  parent;  // Helps  in  tracing  path  when answer is found
float cost;
} list_node_t;

list_node_t list_node;
```

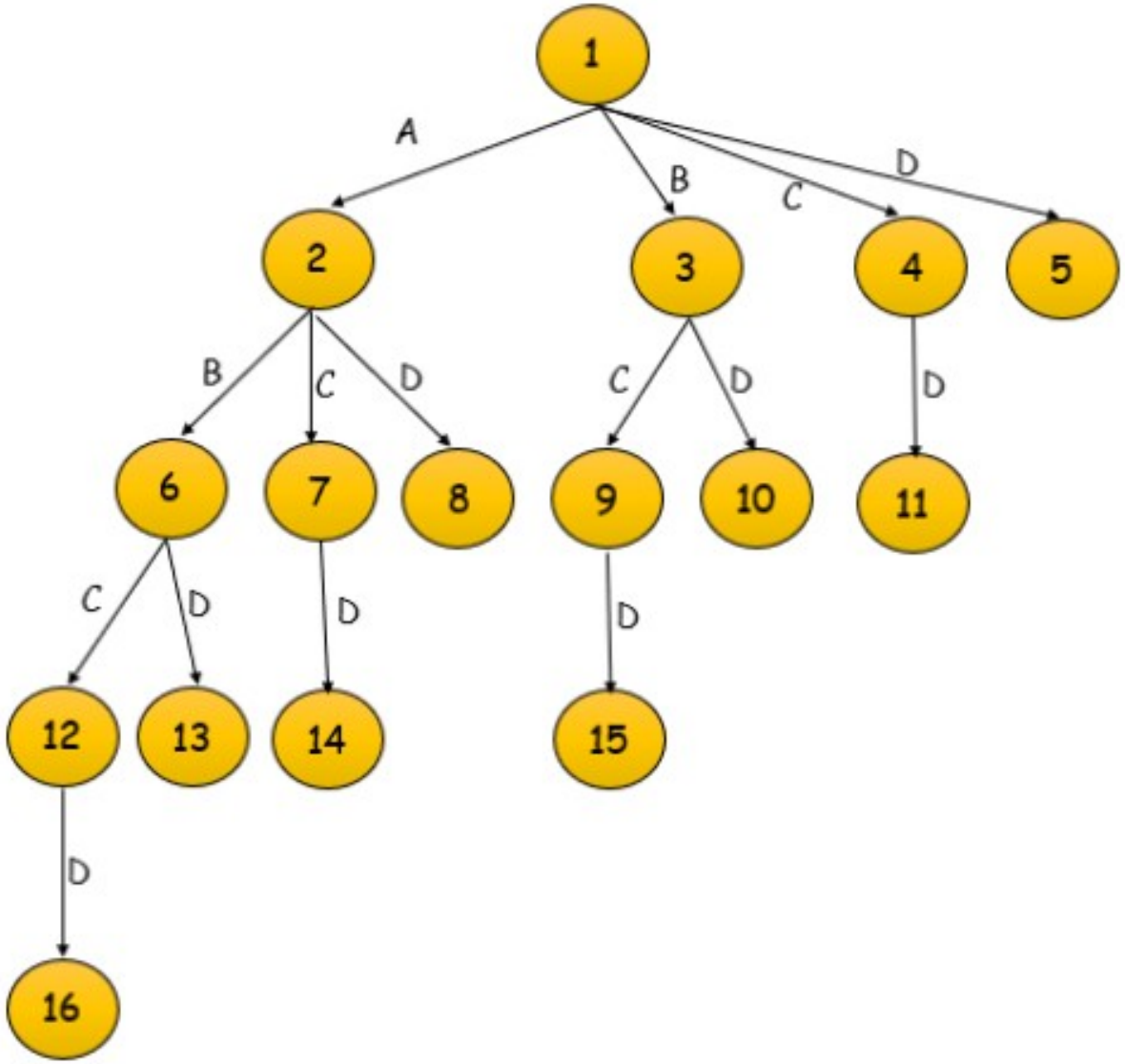
```
algorithm LCSearch ( t )
{ // Search t for an answer node, Input: Root node of tree t,
  // Output: Path from answer node to root
        if ( *t is an answer node)
            print ( *t ); return;
        E = t; // E-node, live node whose children are currently
               // being generated
        Initialize the list of live nodes to be empty;
        while ( true )
        {
            for each child x of E
            {
                if x is an answer node
                      print the path from x to t; return;
                Add ( x ); // Add x to list of live nodes;
                x->parent = E; // Pointer for path to root
            }
            if there are no more live nodes
                  print ( "No answer node" ); return;
            E = Least(); // Find a live node with least estimated cost
             // The found node is deleted from the list of live nodes
         }
}
```

LIFO Branch and Bound

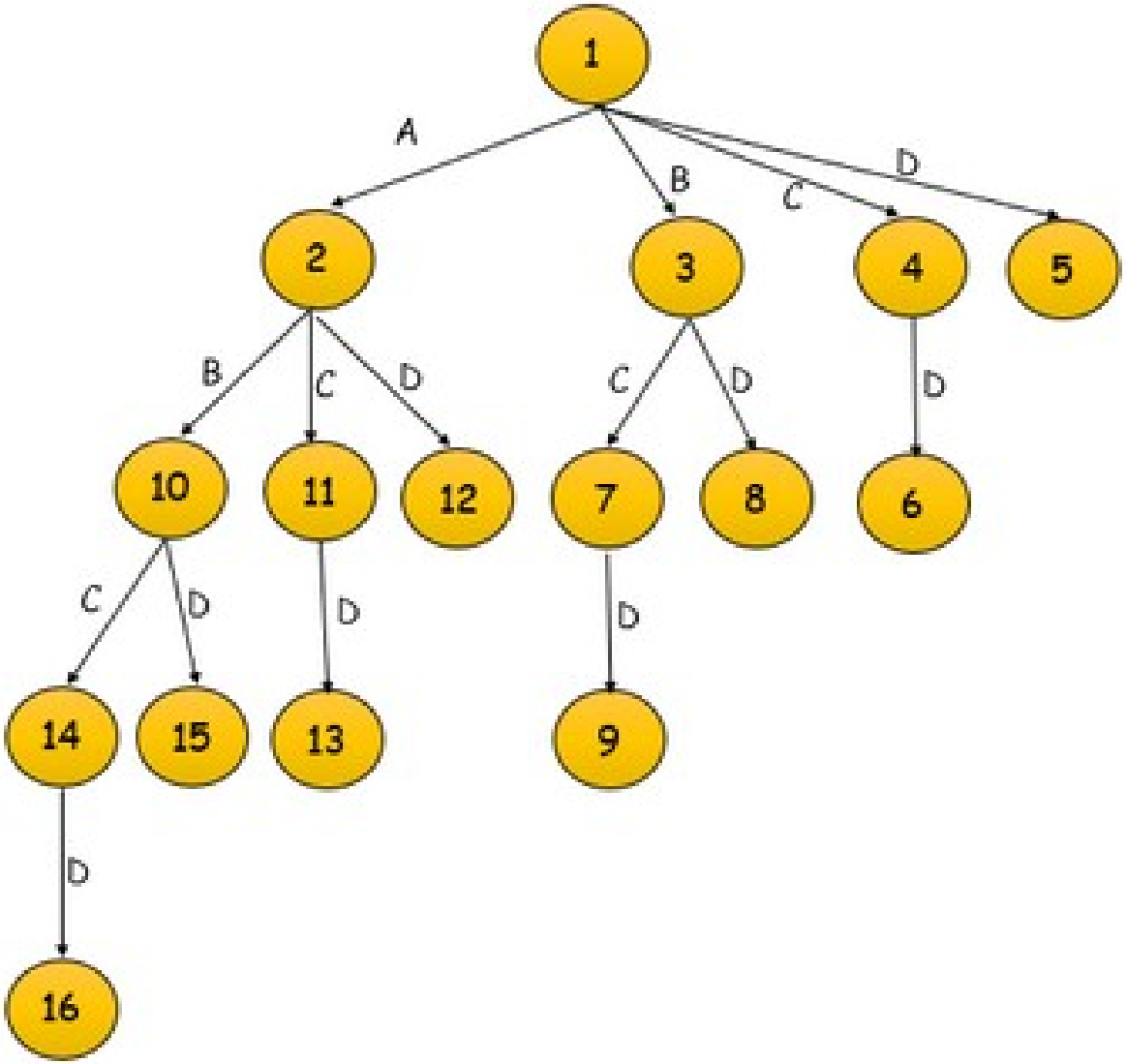
- The Last-In-First-Out (LIFO) approach follows the stack approach in creating the state space tree.
- Here, when nodes get added to the state space tree, think of them as getting added to a stack.
- When all nodes of a level are added, we pop the topmost element from the stack and then explore it.

- **TRY :** For a given set {A, B, C, D}, create the state space tree for FIFO and LIFO branch and bound approach.

FIFO Branch and Bound (node processing)



LIFO Branch and Bound (node processing)



0/1 Knapsack Problem using LC-BB

	1	2	3	4
Profit	10	10	12	18
Weight	2	4	6	9
	m = 15			

We convert the problem into minimization by considering negative profits for each item.

$$initially\ upper = \infty$$

$$u(1) = -32$$
$$\hat{c}(1) = -38$$

(1)

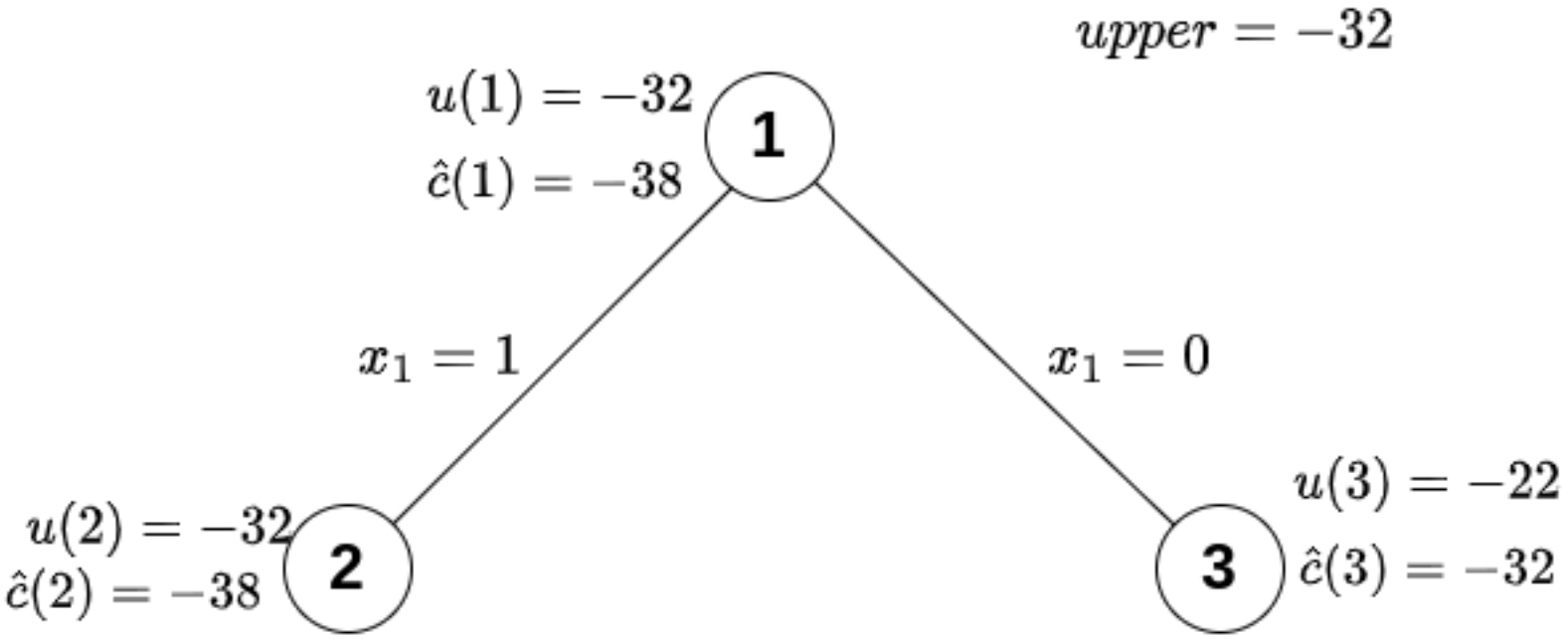
$$cost,\ \hat{c}()\ = sum\ of\ profits\ of\ all\ items\ with\ fraction$$
$$upper\ bound,\ u()\ = sum\ of\ profits\ of\ all\ items\ without\ fraction$$

$$\hat{c}(1) = -10 - 10 - 12 - (18/9) \times 3 = -38$$
$$u(1) = -10 - 10 - 12 = -32$$

As we got u(1) less than infinity, the upper will be -32

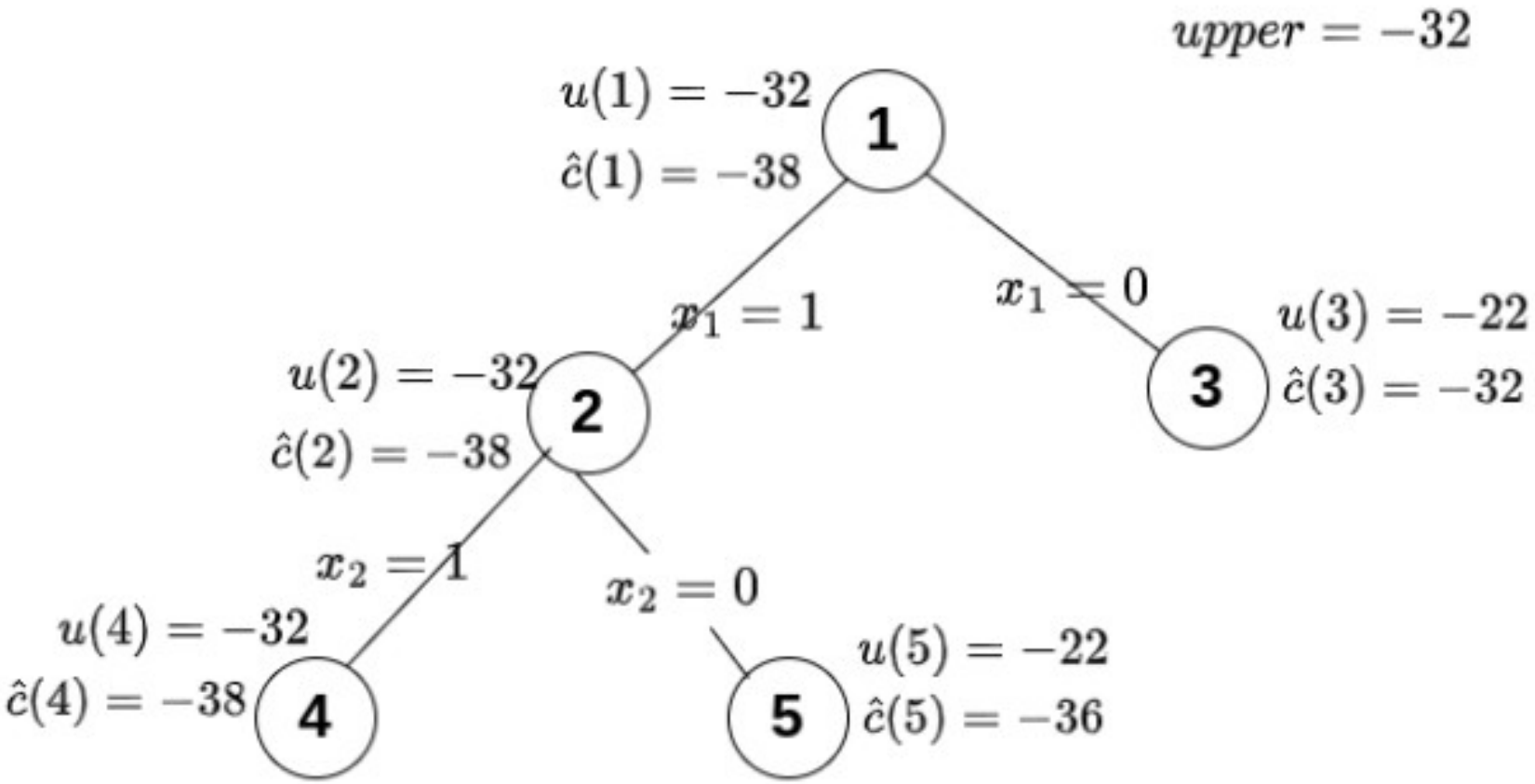
$$upper = -32$$

At each new node created, if its cost is greater than upper **kill it**.



$$\hat{c}(3) = -10 - 12 - (18/9) \times 5 = -32$$
$$u(3) = -10 - 12 = -22$$

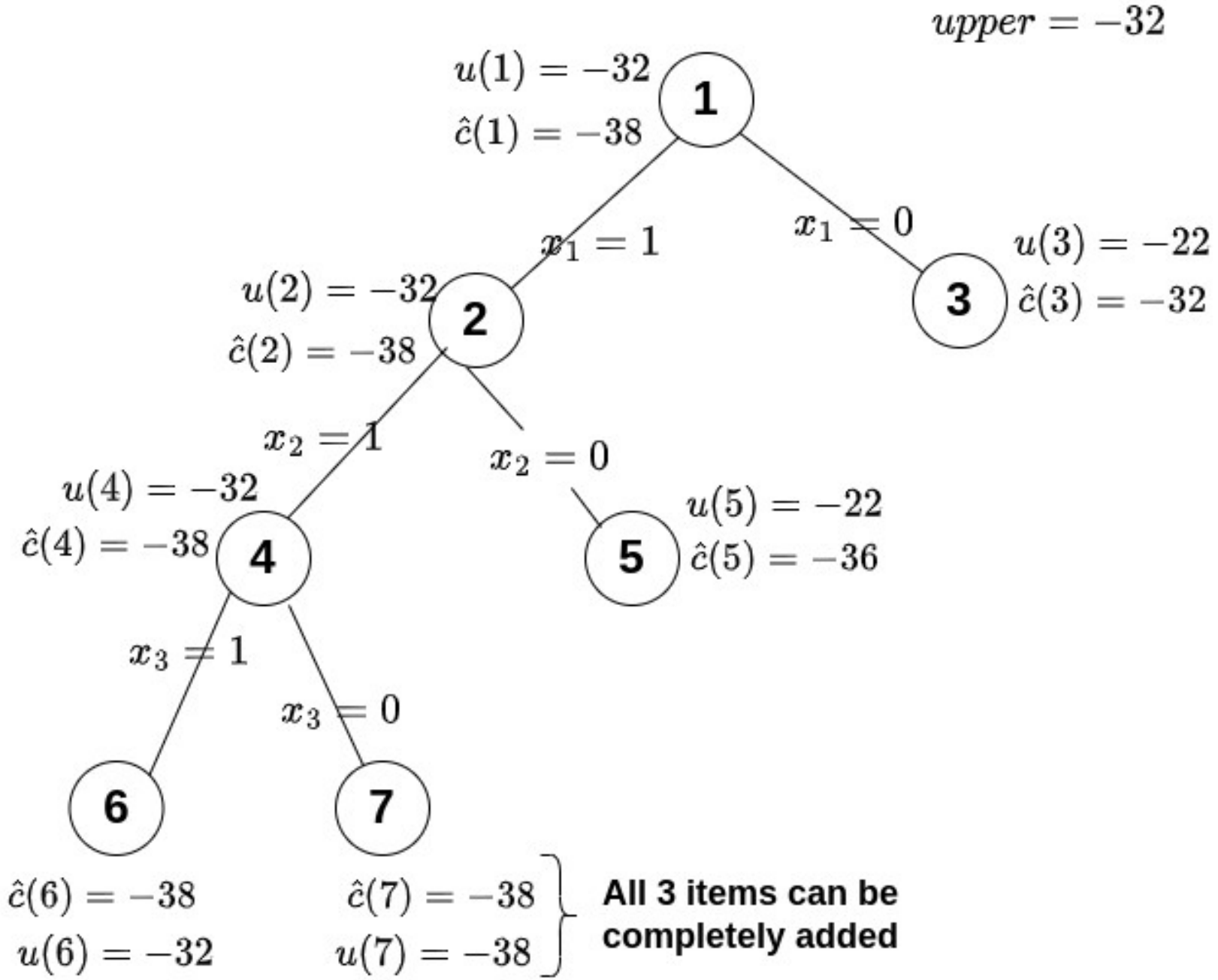
As item 1 is added, there is no change in cost(2) and u(2), as its already in the cost(1) and u(1). If cost(3)>upper, kill it. No need to kill here. If u(3)< upper, update upper=u(3). Not required here. The node with least cost will be processed next.

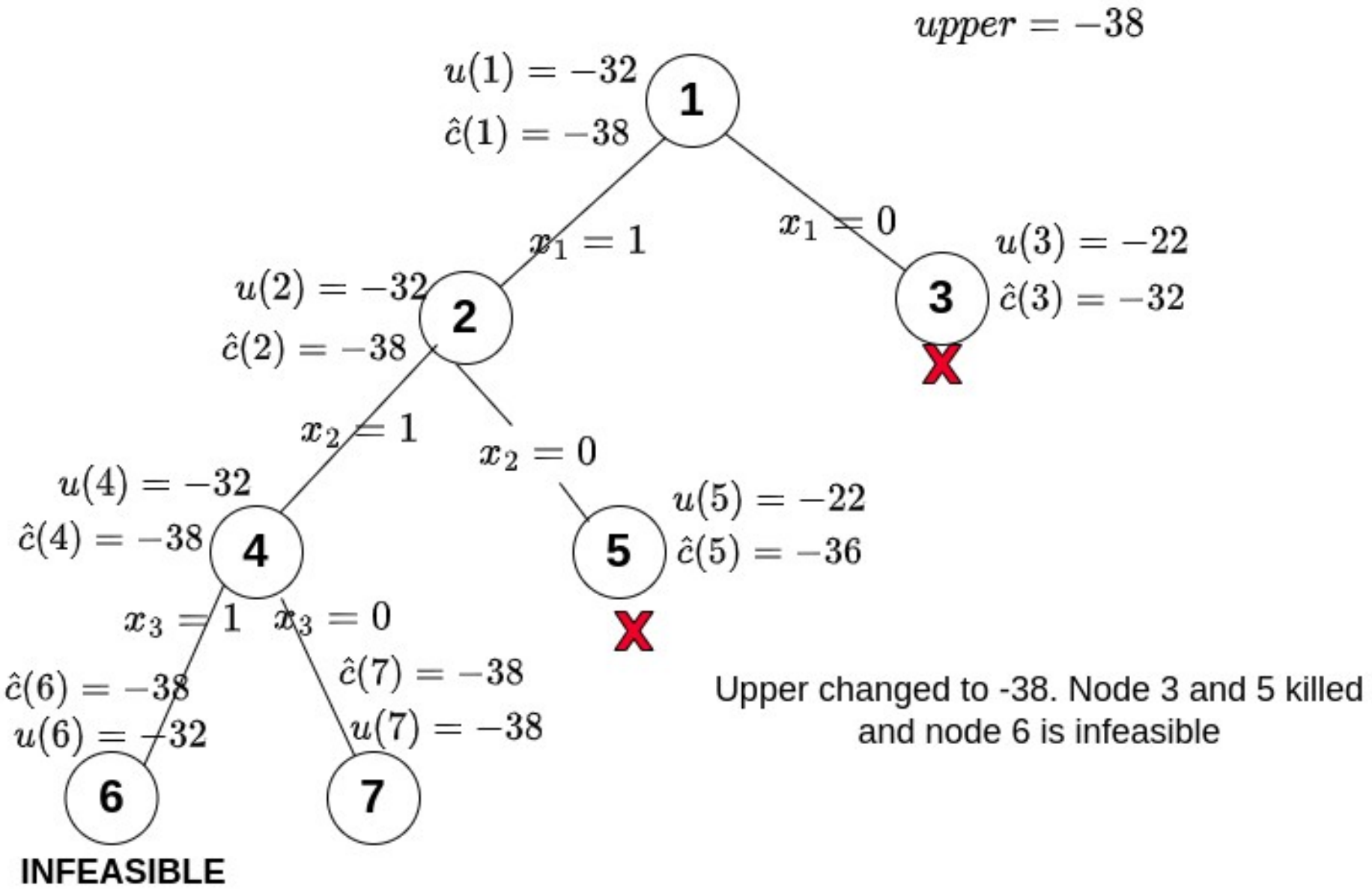


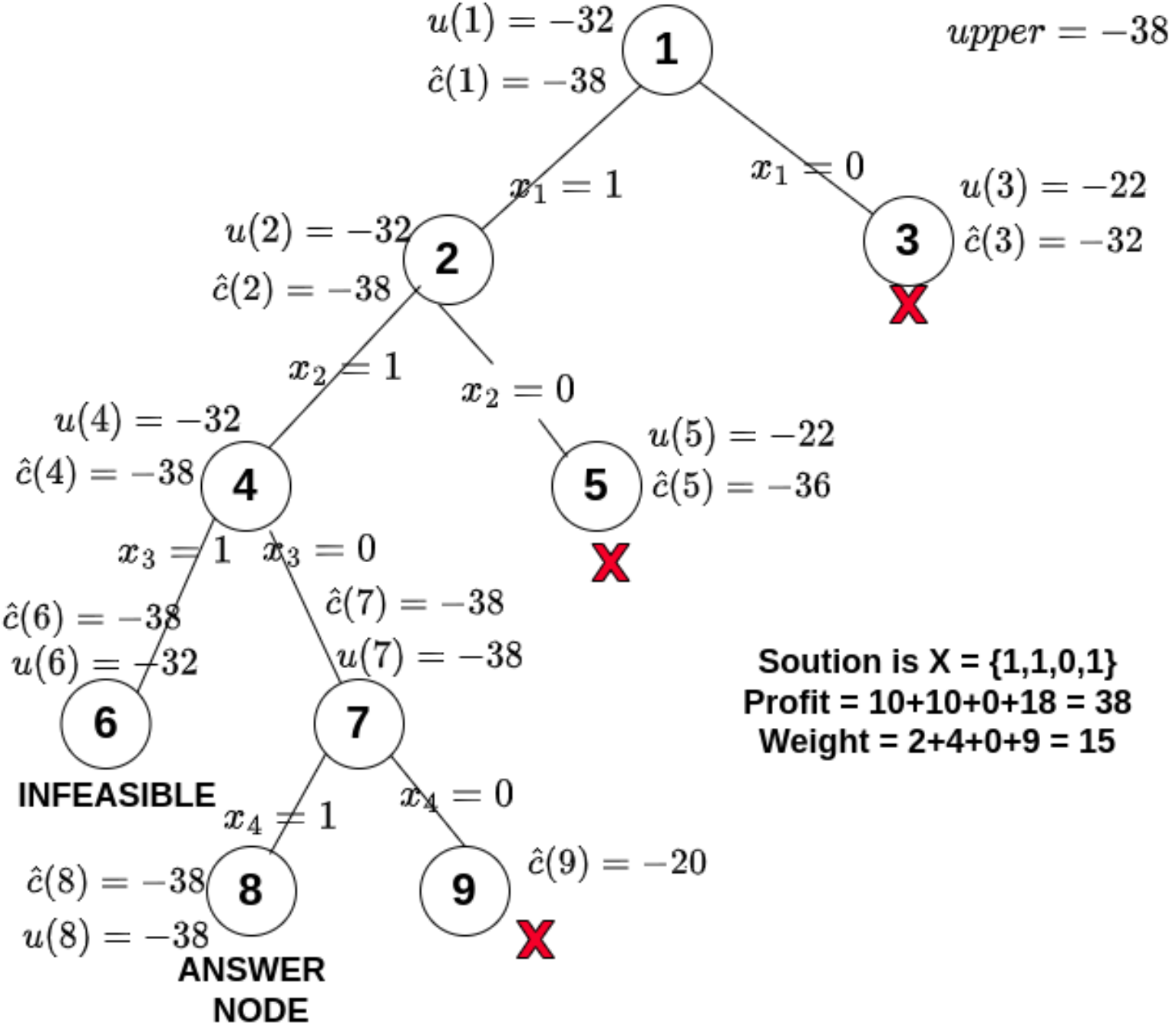
$$u(5) = -10 - 12 = -22$$
$$\hat{c}(5) = -10 - 12 - (18/9) \times 7 = -36$$

No need to kill the node or update the upper.

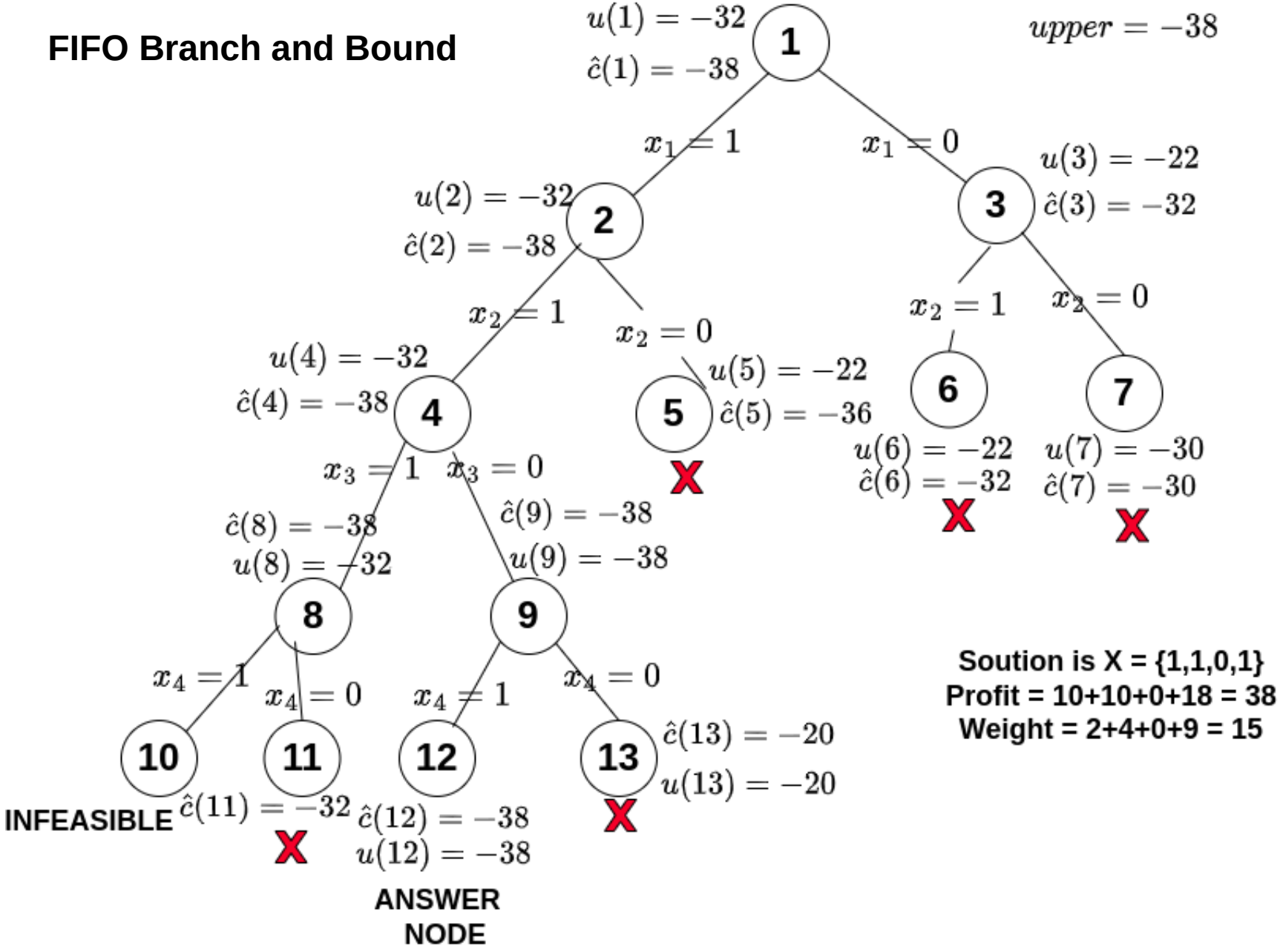
The next node processed is 4


upper = −32
u(1) = −32
ĉ(1) = −38
1
x1 = 1
x1 = 0
u(2) = −32
ĉ(2) = −38
2
3
u(3) = −22
ĉ(3) = −32
x2 = 1
x2 = 0
u(4) = −32
ĉ(4) = −38
4
5
u(5) = −22
ĉ(5) = −36
x3 = 1
x3 = 0
6
7
ĉ(6) = −38
u(6) = −32
ĉ(7) = −38
u(7) = −38
All 3 items can be completely added


upper = −38
u(1) = −32
ĉ(1) = −38
1
x1 = 1
x1 = 0
u(2) = −32
ĉ(2) = −38
2
3
u(3) = −22
ĉ(3) = −32
X
x2 = 1
x2 = 0
u(4) = −32
ĉ(4) = −38
4
5
u(5) = −22
ĉ(5) = −36
X
x3 = 1
x3 = 0
ĉ(6) = −38
u(6) = −32
ĉ(7) = −38
u(7) = −38
6
7
INFEASIBLE
Upper changed to -38. Node 3 and 5 killed and node 6 is infeasible


$u(1) = -32$
$\hat{c}(1) = -38$
1
$upper = -38$
$x_1 = 1$
$x_1 = 0$
$u(2) = -32$
$\hat{c}(2) = -38$
2
3
$u(3) = -22$
$\hat{c}(3) = -32$
X
$x_2 = 1$
$x_2 = 0$
$u(4) = -32$
$\hat{c}(4) = -38$
4
5
$u(5) = -22$
$\hat{c}(5) = -36$
X
$x_3 = 1$
$x_3 = 0$
$\hat{c}(6) = -38$
$u(6) = -32$
$\hat{c}(7) = -38$
$u(7) = -38$
6
7
INFEASIBLE
$x_4 = 1$
$x_4 = 0$
$\hat{c}(8) = -38$
$u(8) = -38$
8
9
$\hat{c}(9) = -20$
X
ANSWER NODE
Soution is X = {1,1,0,1}
Profit = 10+10+0+18 = 38
Weight = 2+4+0+9 = 15

FIFO Branch and Bound



Soution is X = {1,1,0,1}
Profit = 10+10+0+18 = 38
Weight = 2+4+0+9 = 15

Traveling salesperson 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.
- We select the **smallest number in each row (column)** and subtract it from each element in respective row (column) to reduce a given matrix.
- The sum of such integers constitutes the **reduction cost.**

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

a) Cost Adjacency Matrix

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
9	0	3	12	∞

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

b) Reduced cost matric, RCM(1)

- As the given CAM does not contain 0 in each row and column, we select the smallest number in each row and column and subtract it from each element of respective row and column to get the reduced matrix.
- $\hat{C}(1) = (10+2+2+3+4) + (1+0+3+0+0) = 25$
- Initially, upper is set to infinity and changed only when an answer node is found.

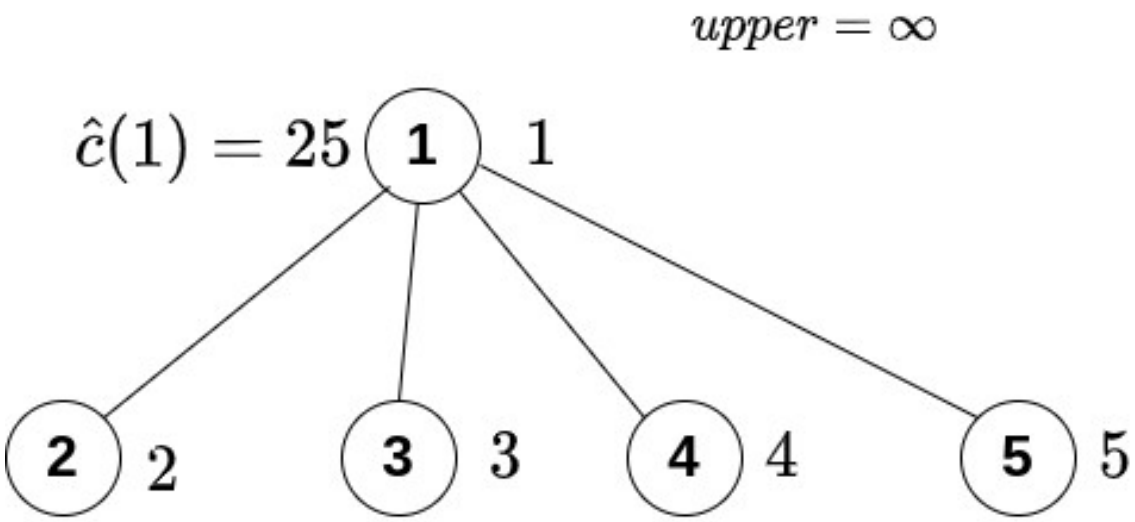
$$upper = \infty$$

$$\hat{c}(1) = 25 \quad \textbf{1} \quad 1$$

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

RCM(1)

The number in circle is the node number in state space tree. The number near the circle is the city number from the given graph. As we have five cities the numbers near the circle will range from 1-5.



Now, we have four cities to visit. So, four new nodes are created. To find the cost of node 2, we will consider a path from city 1 to city 2. So, make 1st row and 2nd column of node 1 matrix, RCM(1), as infinity. And also the change the path from city 2 to city 1 is infinity.

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

RCM(1)

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

RCM(2)

Since, each row and column contains zero and no negative number the cost of reduction is 0 , i.e. rc(2)=0.

$\hat{C}(2) = \hat{C}(1) + \text{cost}(1,2)$ [refer RCM(1), row 1, col 2] + rc(2)

$= 25 + 10 + 0 = 35$

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

RCM(1)

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2
15	3	∞	∞	0
11	0	∞	12	∞

To find the cost of node 3, we will consider a path from city 1 to city 3. So, make 1st row and 3rd column of node 1 matrix as infinity. And also the change the path from city 3 to city 1 is infinity. Since, the first column does not contain zero, we subtract 11 from every element of column 1.

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

RCM(3)

Since, we subtracted 11 from 1st row, the cost of reduction is 11 , i.e. rc(3)=11.

$\hat{C}(3) = \hat{C}(1) + \text{cost}(1,3)$ [refer RCM(1), row 1, col 3] + rc(3)

$= 25 + 17 + 11 = 53$

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

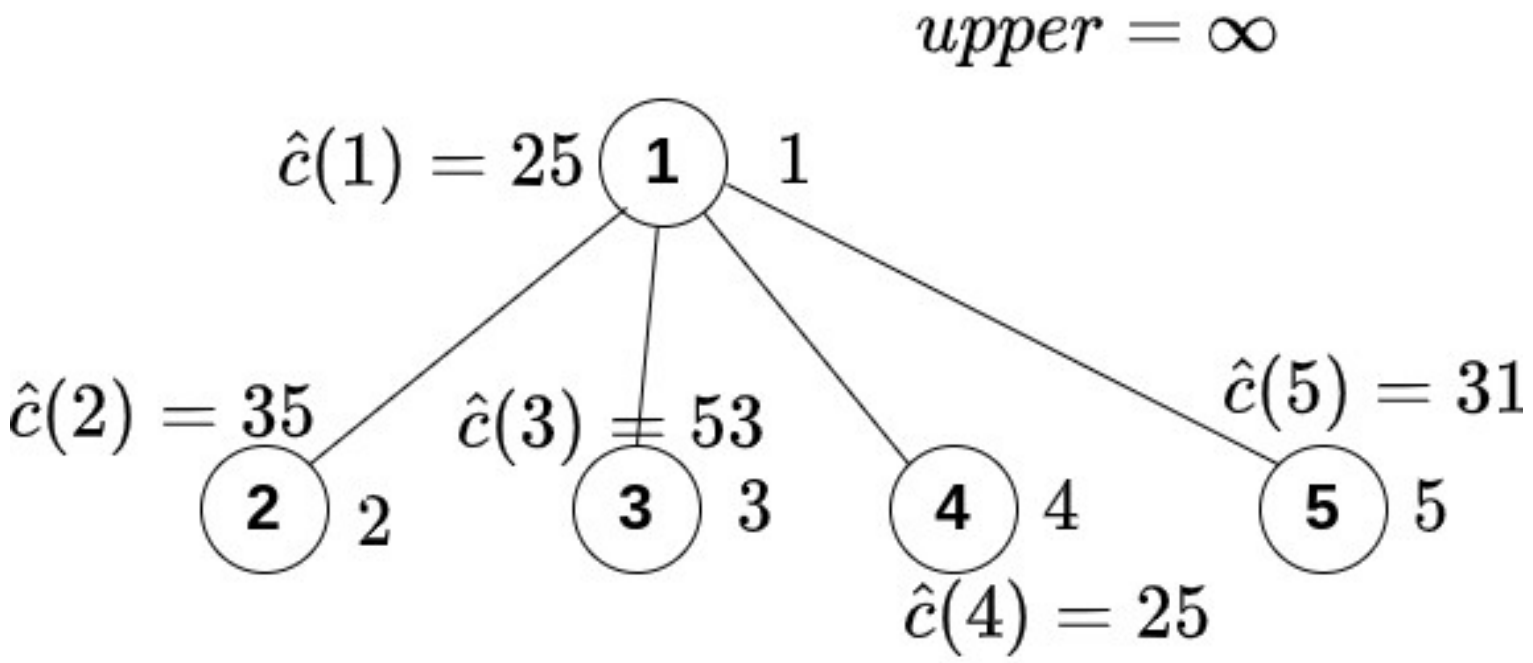
RCM(4)

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

RCM(5)

$\hat{C}(4) = 25$

$\hat{C}(5) = 31$

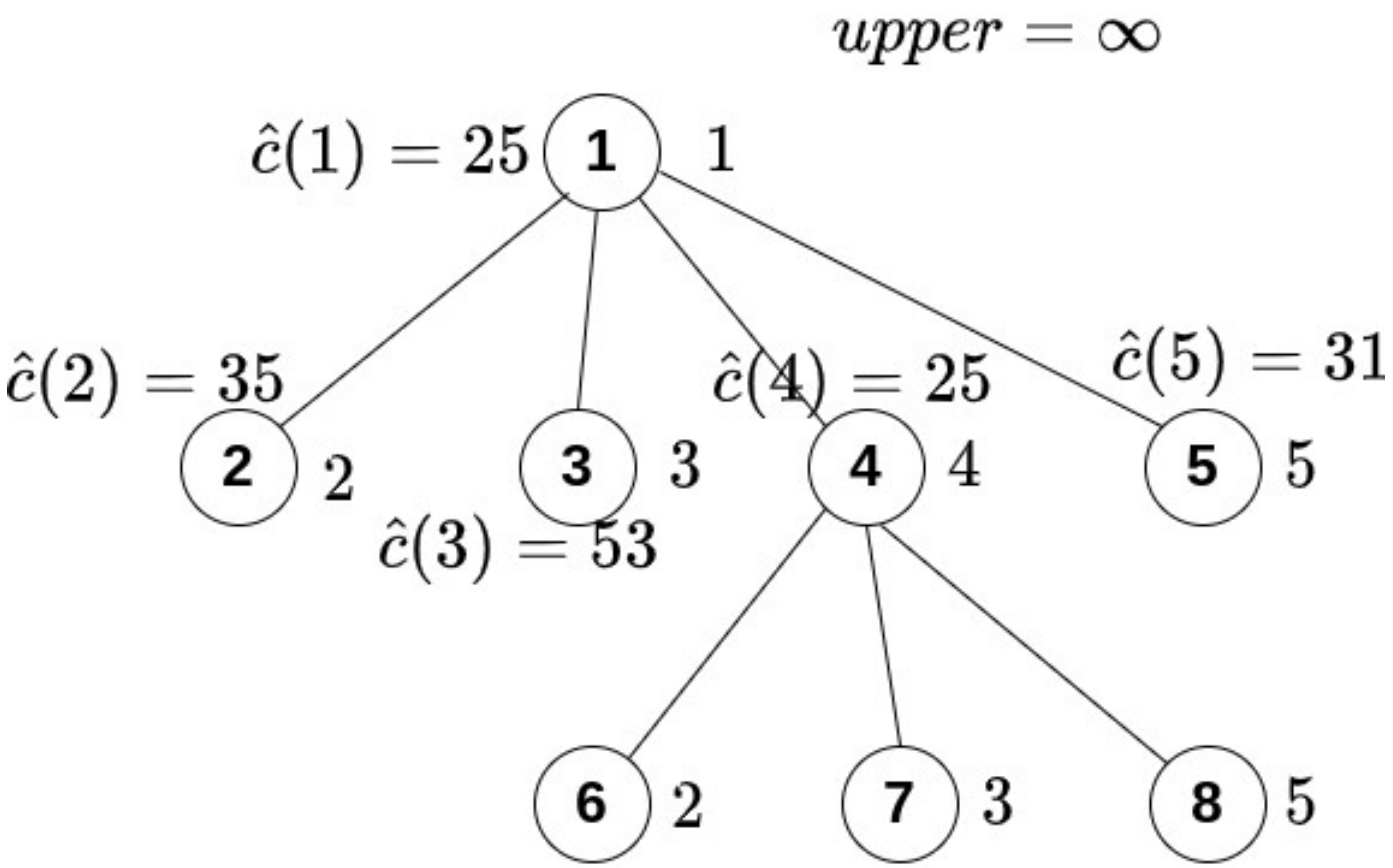

$upper = \infty$
$\hat{c}(1) = 25$
1
1
$\hat{c}(2) = 35$
2
2
$\hat{c}(3) = 53$
3
3
4
4
$\hat{c}(4) = 25$
$\hat{c}(5) = 31$
5
5

- Now as we are using LC-BB, use the unexplored node with Least Cost, i.e. node 4 with cost 25.

- Now the matrix of node 4 RCM(4) will be used for further processing.

Which node we have to explore?

- We came from city 1 to city 4. Now, from city 4 we can visit city 2, city 3, and city 5.


$upper = \infty$
$\hat{c}(1) = 25$
1
1
$\hat{c}(2) = 35$
2
2
3
3
$\hat{c}(3) = 53$
$\hat{c}(4) = 25$
4
4
$\hat{c}(5) = 31$
5
5
6
2
7
3
8
5

To find cost of node 6, make 4th row and 2nd column of node 4, i.e. RCM(4) matrix to infinity. Also the path in the same matrix from from city 2 to city 1 (root) is changed to infinity.

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

RCM(4)

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

RCM(6)

Since each row and column of RCM(6) has zero and no negative numbers, the cost of reduction is 0, rc(6)=0.

$\hat{C}(6) = \hat{C}(4) + \text{cost}(4,2)$ [refer RCM(4), row 4, col 2] + rc(6)

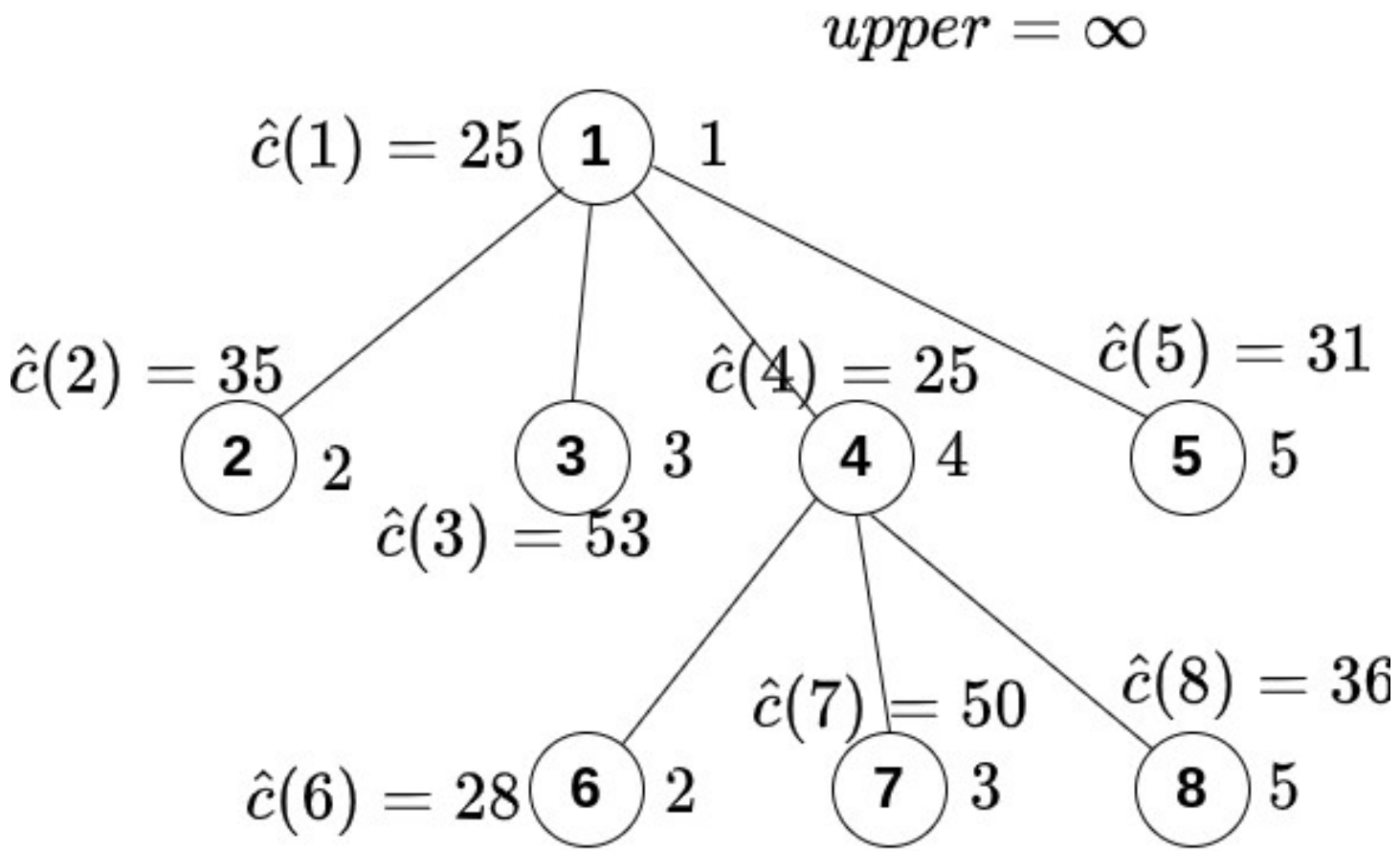
$= 25 + 3 + 0 = 28$

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

RCM(7)

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

RCM(8)

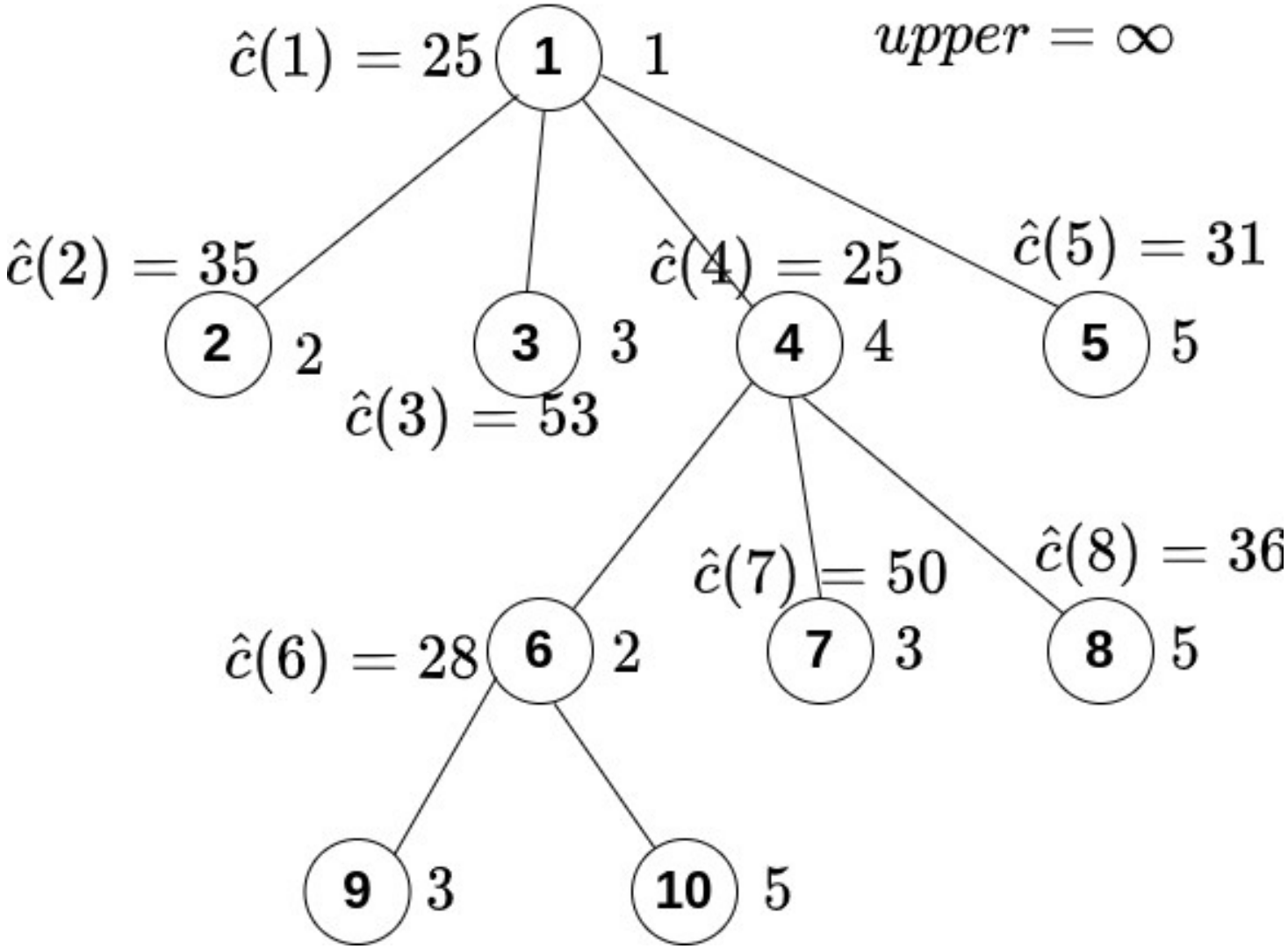

$upper = \infty$
$\hat{c}(1) = 25$
1
1
$\hat{c}(2) = 35$
2
2
$\hat{c}(3) = 53$
3
3
$\hat{c}(4) = 25$
4
4
$\hat{c}(5) = 31$
5
5
$\hat{c}(6) = 28$
6
2
$\hat{c}(7) = 50$
7
3
$\hat{c}(8) = 36$
8
5

- Now as we are using LC-BB, use the unexplored node with Least Cost, i.e. node 6 with cost 28.

- Now the matrix of node 6 RCM(6) will be used for further processing.

Which node we have to explore?

- We came from city 1 to city 4, and then city 2. Now, from city 2 we can visit city 3, and city 5.


$\hat{c}(1) = 25$
1
1
$upper = \infty$
$\hat{c}(2) = 35$
2
2
$\hat{c}(3) = 53$
3
3
$\hat{c}(4) = 25$
4
4
$\hat{c}(5) = 31$
5
5
$\hat{c}(6) = 28$
6
2
$\hat{c}(7) = 50$
7
3
$\hat{c}(8) = 36$
8
5
9
3
10
5

To get the cost for node 9, make all values in 2nd row and 3rd column of node 6, RCM(6), matrix as infinity. Also, mark entry for city 3 to city 1 as infinity.

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

RCM(6)

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

As 3rd row and 1st colum do not contain 0, we subtract 11 from 1st column and 2 from 2nd row. So, the reduction cost, rc(9)=11+2=13.

$\hat{C}(9) = \hat{C}(6) + \text{cost}(2,3)$ [refer RCM(6), row 2, col 3] + rc(9)

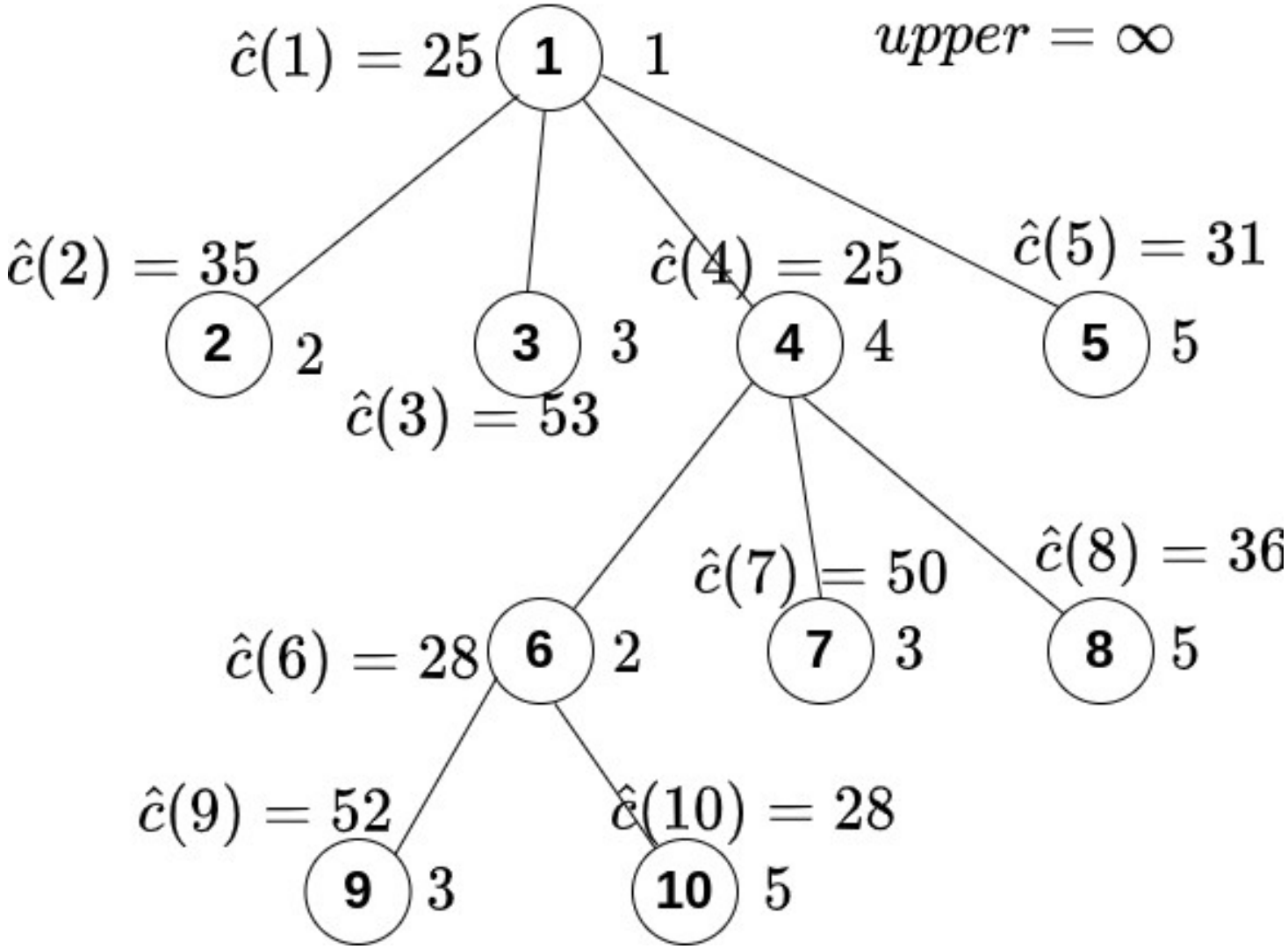
$= 28 + 11 + 13 = 52$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	0	∞	∞

RCM(9)

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

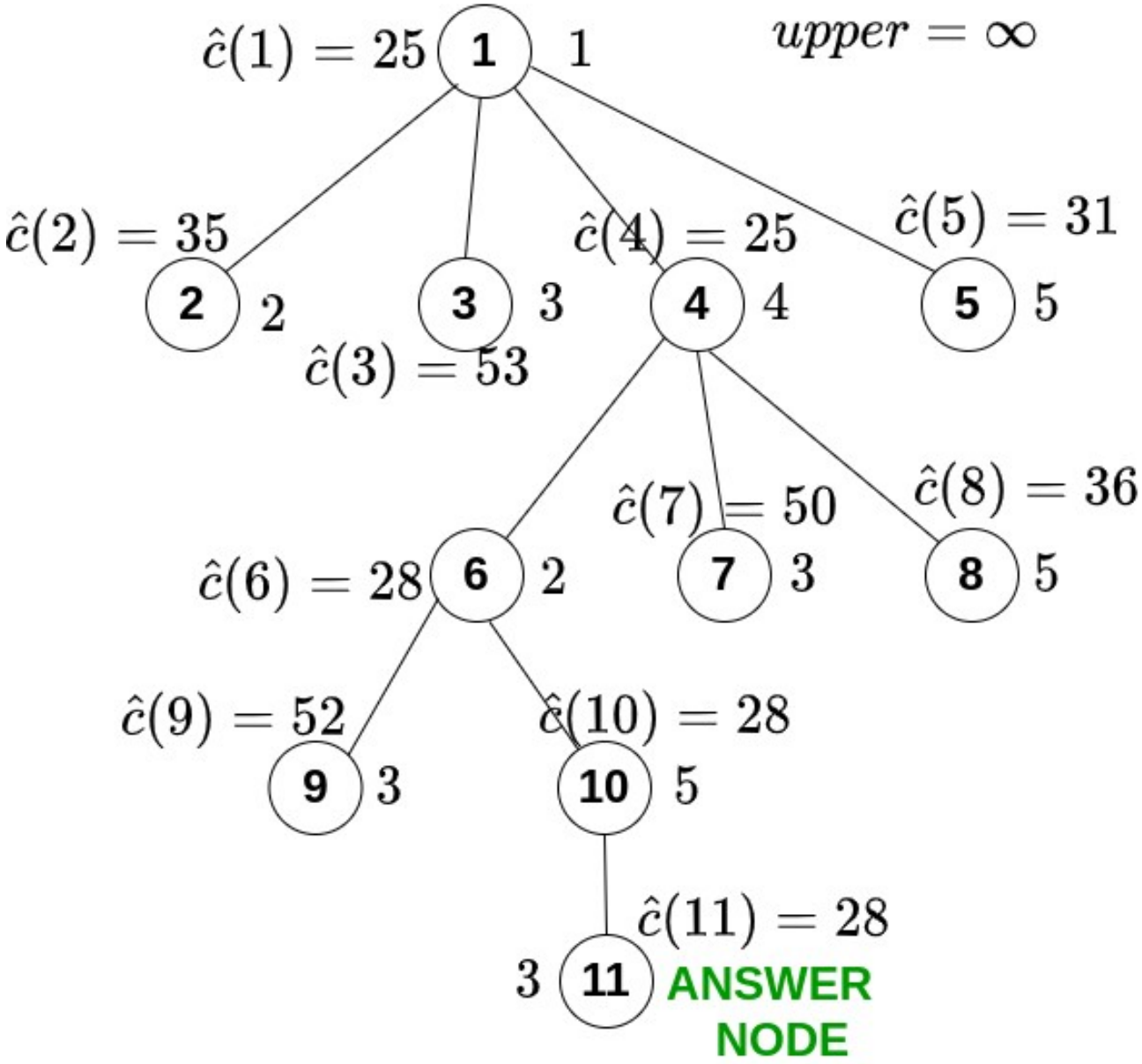
RCM(10)


$\hat{c}(1) = 25$
1
1
$upper = \infty$
$\hat{c}(2) = 35$
2
2
$\hat{c}(3) = 53$
3
3
$\hat{c}(4) = 25$
4
4
$\hat{c}(5) = 31$
5
5
$\hat{c}(6) = 28$
6
2
$\hat{c}(7) = 50$
7
3
$\hat{c}(8) = 36$
8
5
$\hat{c}(9) = 52$
9
3
$\hat{c}(10) = 28$
10
5

Which node to process next?
The one with least cost, i.e. node 10, cost 28

$\hat{C}(11) = \hat{C}(10) + \text{cost}(5,3)$ [refer RCM(10), row 5, col 3] + rc(11)

$= 28 + 0 + 0 = 28$

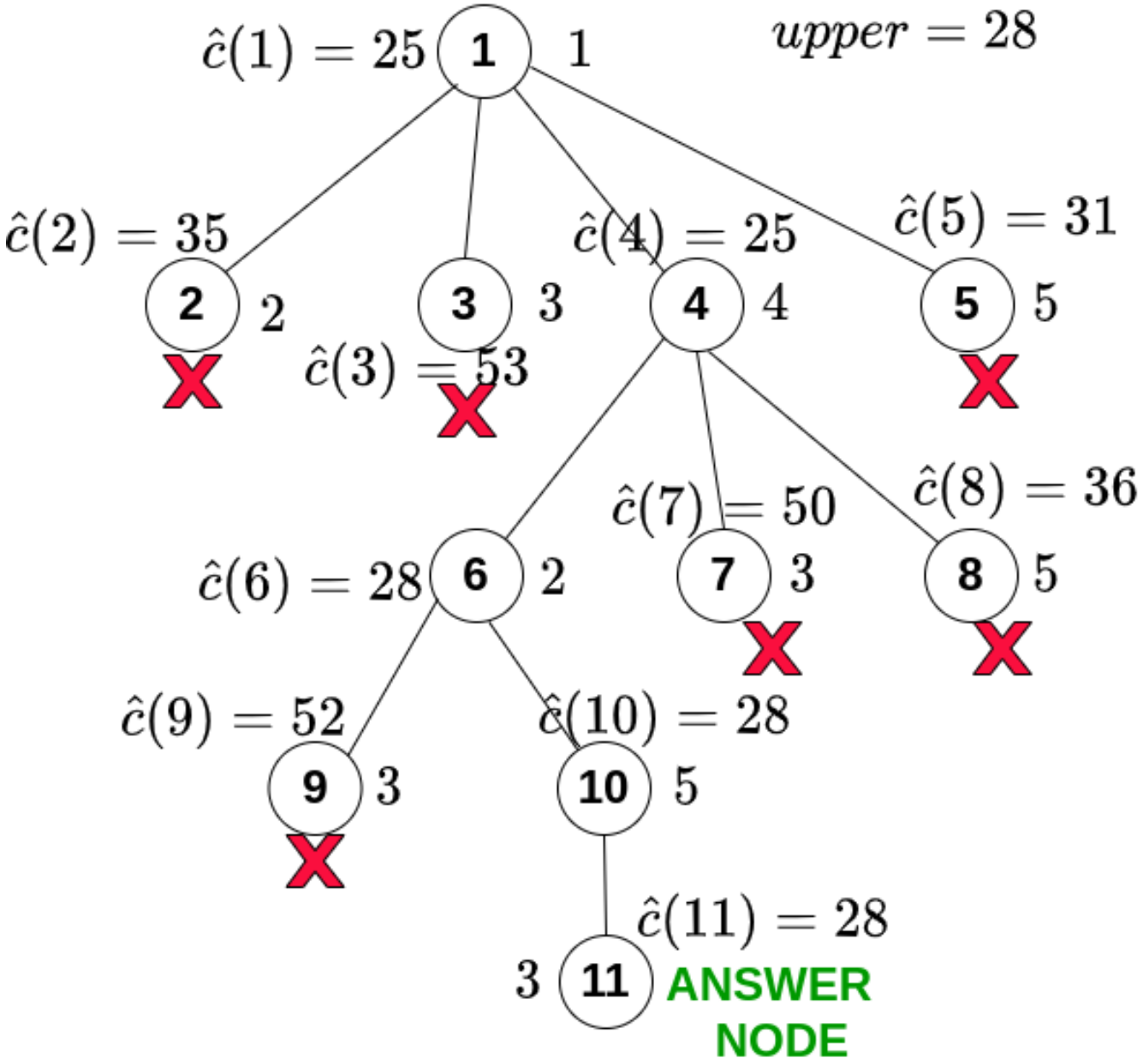

$\hat{c}(1) = 25$
1
1
$upper = \infty$
$\hat{c}(2) = 35$
2
2
$\hat{c}(3) = 53$
3
3
$\hat{c}(4) = 25$
4
4
$\hat{c}(5) = 31$
5
5
$\hat{c}(6) = 28$
6
2
$\hat{c}(7) = 50$
7
3
$\hat{c}(8) = 36$
8
5
$\hat{c}(9) = 52$
9
3
$\hat{c}(10) = 28$
10
5
$\hat{c}(11) = 28$
3
11
ANSWER
NODE

Change upper to 28

Kill the unexplored nodes with $\hat{C}$ > upper

This kills all remaining nodes.

Therefore, the shortest path is 1-4-2-5-3-1 with cost 28.


$\hat{c}(1) = 25$
1
1
$upper = 28$
$\hat{c}(2) = 35$
2
2
$\hat{c}(3) = 53$
3
3
$\hat{c}(4) = 25$
4
4
$\hat{c}(5) = 31$
5
5
$\hat{c}(6) = 28$
6
2
$\hat{c}(7) = 50$
7
3
$\hat{c}(8) = 36$
8
5
$\hat{c}(9) = 52$
9
3
$\hat{c}(10) = 28$
10
5
$\hat{c}(11) = 28$
3
11
ANSWER
NODE