Presentation Topic

Design and Analysis of Algorithms

Department of Computer Engineering

BRACT'S, Vishwakarma Institute of Information Technology, Pune-48

(An Autonomous Institute affiliated to Savitribai Phule Pune University)

Objective/s of this session

- To study the **analysis** of algorithms
- To study the **greedy and dynamic programming** algorithmic strategies
- <u>To study the **backtracking and branch and bound algorithmic** strategies</u>
- To study the concept of **hard problems** through understanding of **intractability and NP-Completeness**
- To study some **advance techniques to solve intractable problems**
- To study **multithreaded** and **distributed algorithms**

Learning Outcome/Course Outcome

- Analyze algorithms for their time and space complexities in terms of asymptotic performance.
- Apply greedy and dynamic programming algorithmic strategies to solve a given problem
- **<u>Apply backtracking and branch and bound algorithmic strategies to solve a given problem</u>**
- Identify intractable problems using concept of NP-Completeness
- Use advance algorithms to solve intractable problems
- Solve problems in parallel and distributed scenarios

Unit III	**Backtracking, Branch and Bound**

Backtracking: The General Method
8 Queen's problem,
Graph Coloring

Branch and Bound: 0/1 Knapsack,
Traveling Salesperson Problem.

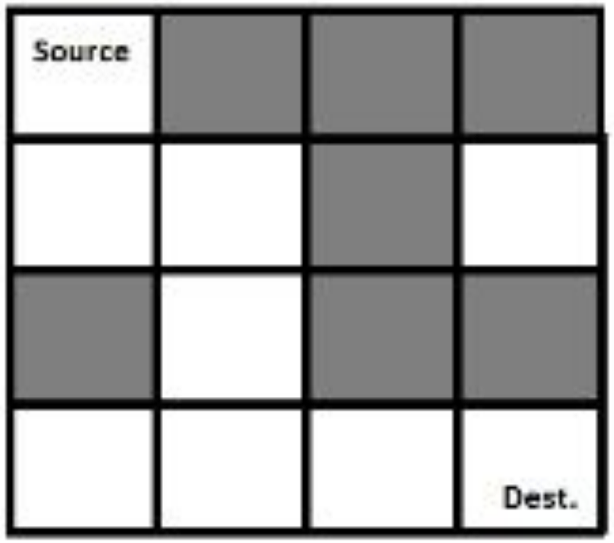
Backtracking

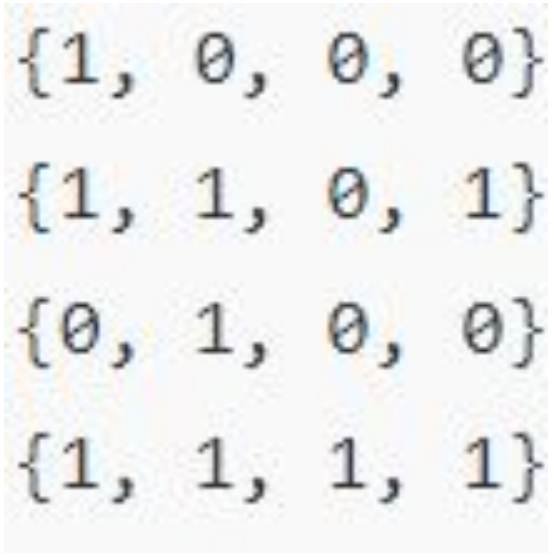
Backtracking

- Suppose you have to make a **series of *decisions***, among various *choices*, where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem

Backtracking

- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”
- Sample problem : MAZE PROBLEM and its binary matrix representation





```
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}
```

Backtrack method

- Mostly, desired solution is expressible as an n-tuple (x1,x2,..,xn), xi is chosen from a finite set S_i
- Often, problem calls for finding one vector that **maximizes** (**minimizes/satisfies**) a *criterion function* P(x1,…xn)
- Sometimes, all vectors satisfying P are required

Example :**Sorting a[1:n]:** *for purpose of understanding only*

- **Backtrack solution** (this sorting, never used practically)
  - Solution is n-tuple
    - Where, xi is index in *a* of the ith smallest element
  - Criterion function P is
    - $a[x_i] <= a[x_{i+1}]$ for $1 <= i < n$
  - Set S_i is a finite set containing integers 1 to n

Comparison with brute force method

- Say set Si has mi elements
- There are *m=m1m2…mn* n-tuples candidate for satisfying the function P
- Brute force – form all these n-tuples
  - Evaluate each
  - Save the best (optimal)
- Backtracking – solution in far fewer than above *m* solutions

Basic Idea of backtracking

- Build up solution vector, one component at a time
- Use *modified criterion functions* Pi(x1, .. ,xn), sometimes called *bounding functions* to test whether the vector being formed has any chance of success
- If partial vector (x_1,..x_i) cannot lead to success, then m_{i+1}..m_n possible test vectors can be ignored entirely

Contd..

- Backtracking is more advanced and optimized than Brute Force.
- It usually uses recursion
- It is applied to problems having low input constraints (for example $N<=20$).

Constraints

- Many problems using backtracking require that all solutions **satisfy** a **complex set of constraints**
- **2 categories of constraints**
  - Implicit
  - Explicit

Explicit constraint

- It restricts x_is to take values only from a given set
  - $x_i > 0$
  - $x_i = 0$ or 1
  - $j < x_i < k$ (i.e. between j and k)
- It depends on the particular instance I of the problem to be solved
- All tuples that **satisfy** the explicit constraints define a possible solution space for I

Implicit constraints

- Rules that determine which of the tuples in the solution space of I satisfy the **criterion function**

- They describe the way in which the **xi must relate to each other**

Backtracking algorithms

- Find a solution by trying one of several choices.
- If the choice proves incorrect, computation *backtracks* or restarts at the point of choice and tries another choice.
- It is often convenient to maintain choice points and alternate choices using *recursion*.
- *Conceptually, a backtracking algorithm does a depth-first search of a tree of possible (partial) solutions. Each choice is a node in the tree.*

Revision of backtracking algorithm

- **Backtracking algorithms** try each possibility until they find the right one.

- It is a depth-first search of the set of possible solutions.

- During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative.

Contd..

- When the alternatives are exhausted, the search returns to the previous choice point and tries the next alternative there.

- If there are no more choice points, the search fails.

8 Queens

N Queens problem

- Place N queens on a N by N chessboard so that no 2 queens CAN attack i.e no two of them

  are on the same row, col or diagonal
- Number the rows, cols as :   1,2,3,4,5,6,7,8…n
- Sol is n tuple, representing n rows (_,_,_,_,_,_,_,_)
- Values in the tuple represent column at that row
- Si={1,2,3,4,5,6,7,8…..n}

8 Queens problem

- Place 8 queens on a 8 by 8 chessboard so that no 2 queens attack
  - No 2 queens are on the same row, col or diagonal
- Number the rows, cols as :   1,2,3,4,5,6,7,8
- Sol is 8 tuple, representing 8 rows (_,_,_,_,_,_,_,_)
- Values in the tuple represent column at that row
- Si={1,2,3,4,5,6,7,8}

8 Queens : Explicit constraint

- It restricts x_is to take values only from a given set
  - x_i =1,2,3,4 for 4 Queens
  - x_i =1,2,3,4,5,6,7 for 7 Queens
  - x_i =1,2,3,4,5,6,7,8 for 8 Queens
- It depends on the particular instance I of the problem to be solved
- All tuples that **satisfy** the explicit constraints define a possible solution space for I

8 Queens : Implicit constraints

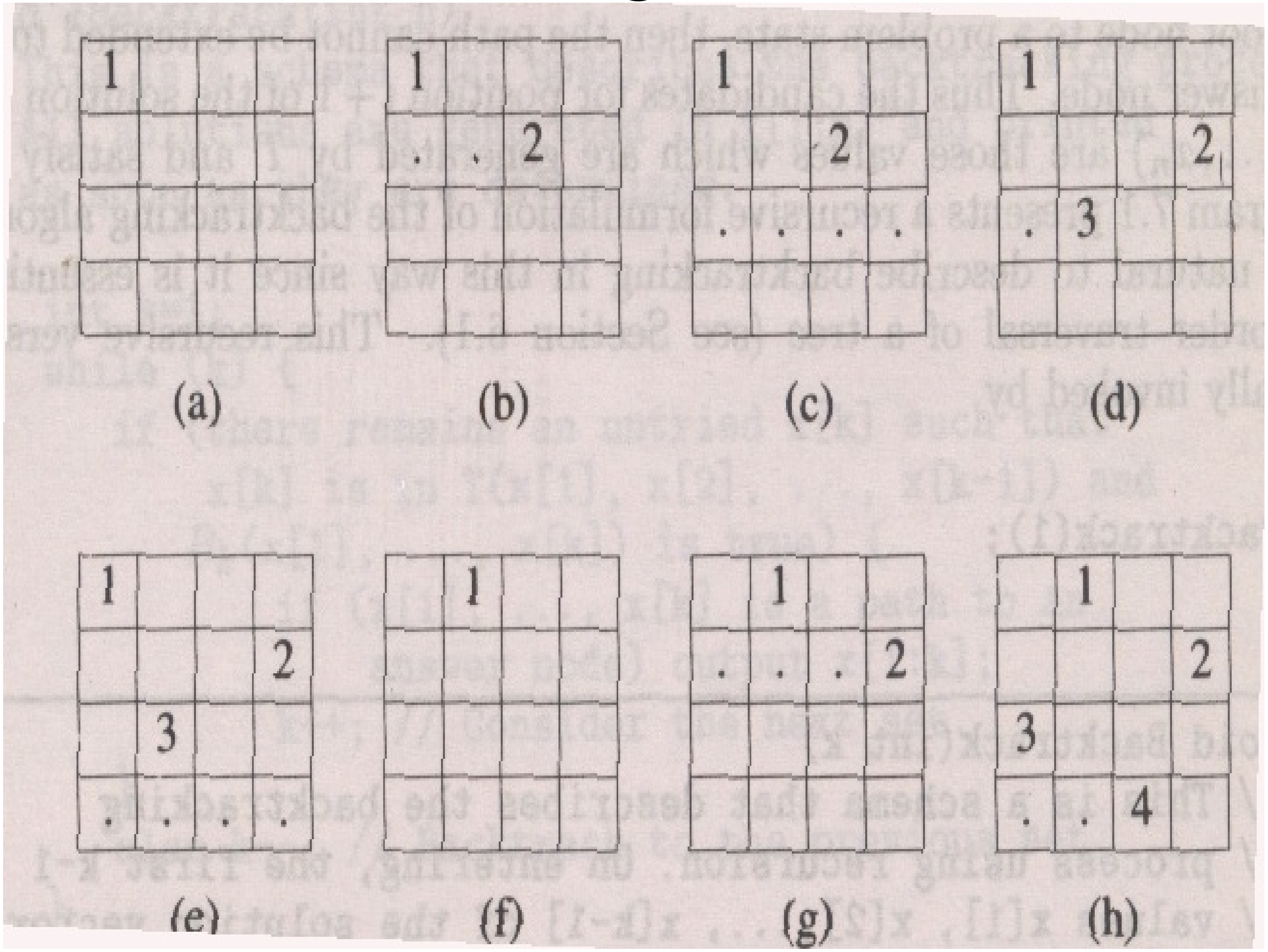
Rules that describe how **xi must relate to each other** –i.e. satisfy **criterion function**

- no 2 xis must be same i.e, no 2 queens in same column
- And, no 2 in the same diagonal

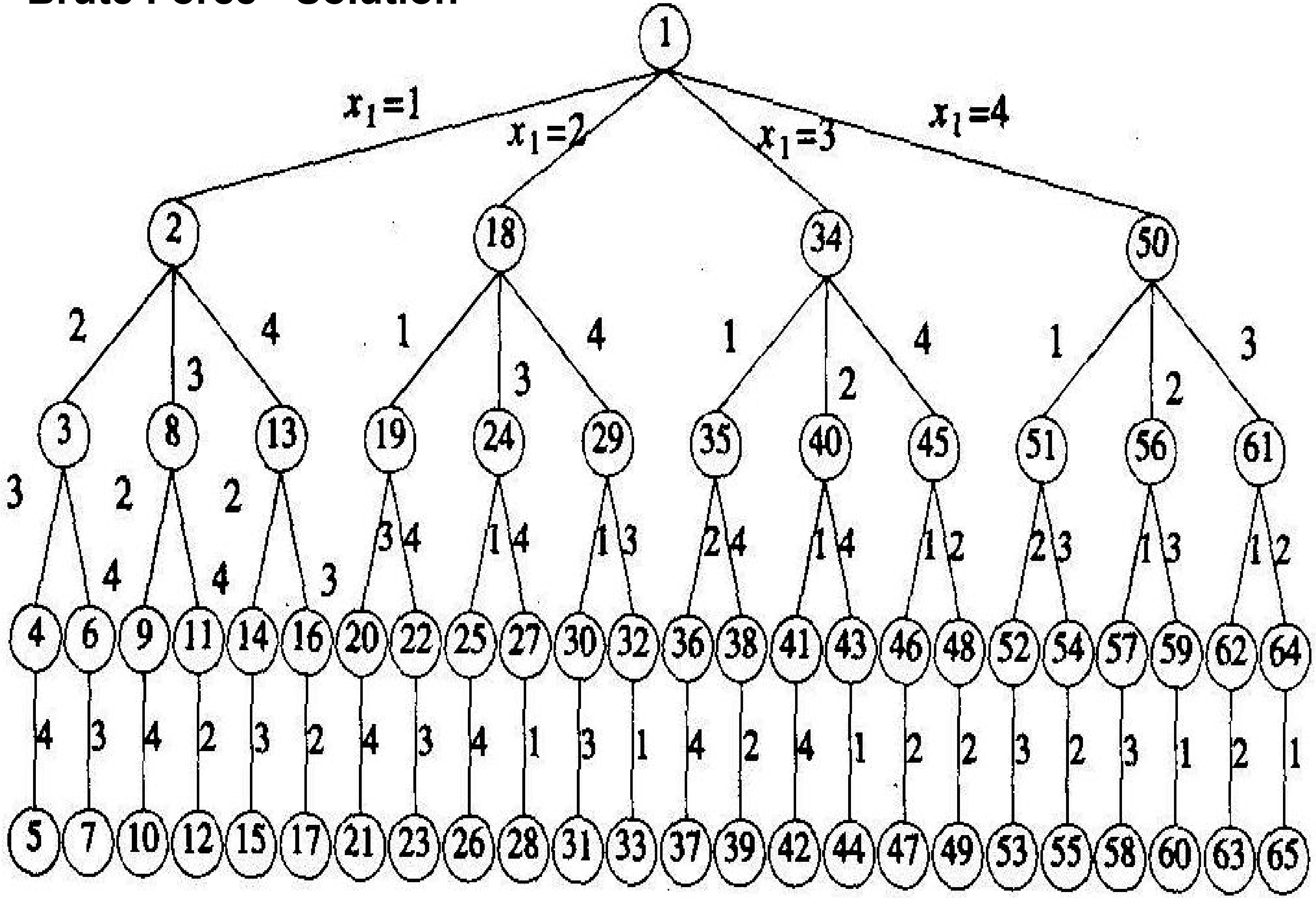
(Rules that determine which of the tuples in the solution space of I satisfy the **criterion function**

They describe the way in which the **xi must relate to each other)**

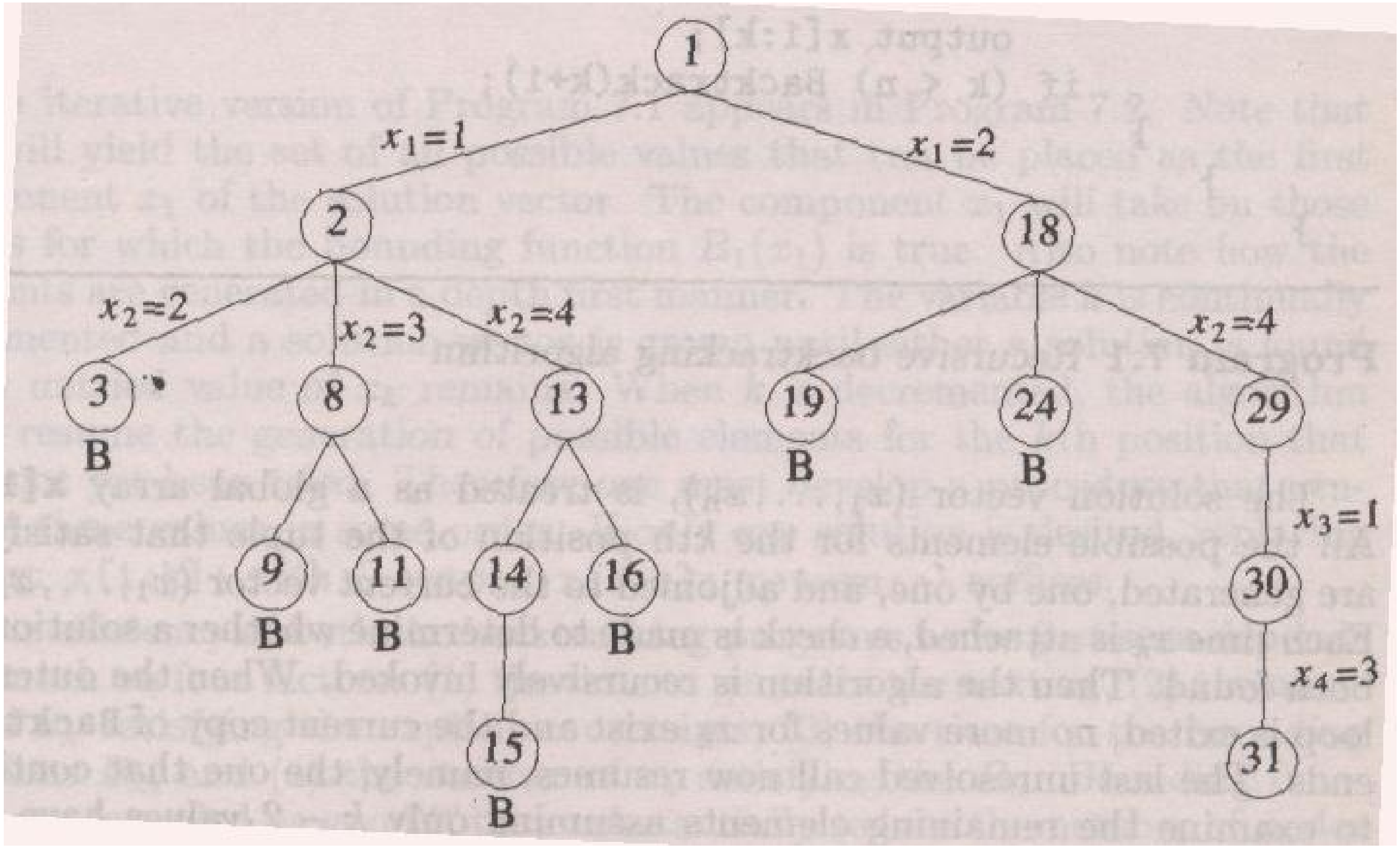
Backtracking: 4Queens



Brute Force - Solution



Backtrack - Solution



Tree organization of **solution space** (terminologies)

- Each NODE= ***problem state***
- All Paths from root to node = ***state space***
- ***Path*** from root to a state S if defines tuple in solution space , implies S is ***Solution state***

Some terminologies

Answer states are those solution states *s* for which the path from **root** to ***s*** defines a tuple that is a member of the set of solutions of the problem i.e. those that satisfy *implicit costraints.*

State space tree : tree organization of solution space

Terminologies Continued…

- **Live node** : A Node which has been generated and all of whose children have not yet been generated
- **E-node** : The live node whose children are currently being generated.
- **Dead node** : It is a generated node which is not to be further expanded.

- **Depth first node generation** with **bounding function** is called **backtracking**

- In **branch and bound** **E-Node remains E-Node till it is dead**

Terminologies Continued…

- **Static state space tree** organizations:

Tree organizations that are INDEPENDENT

of the problem instance being solved, as shown on the right

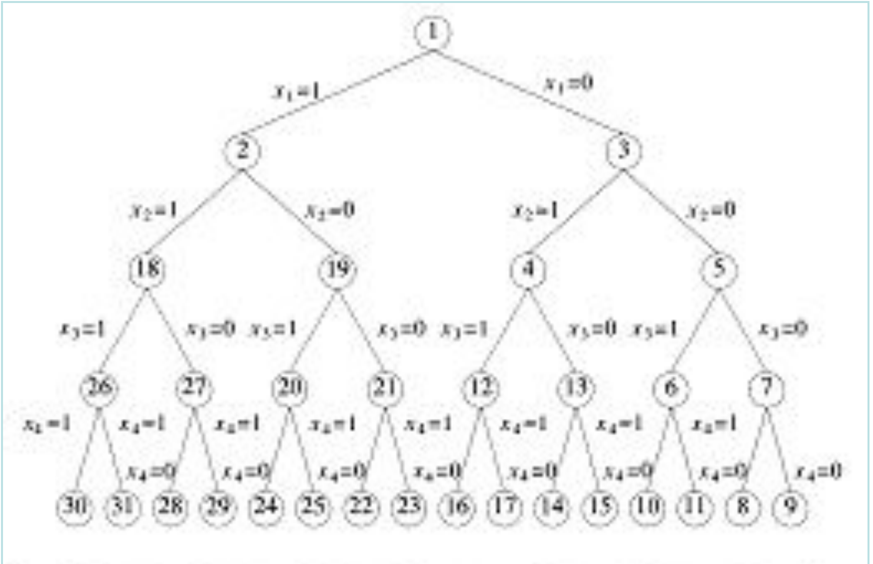
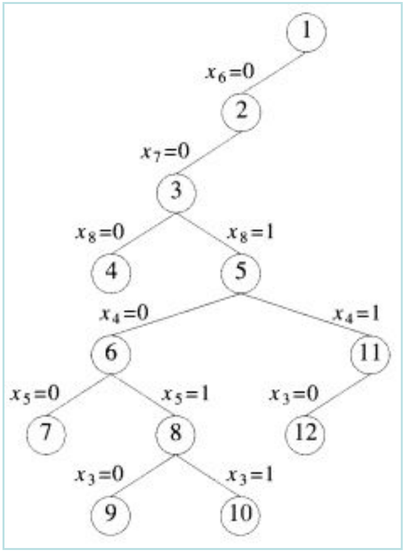


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in *D*-search.

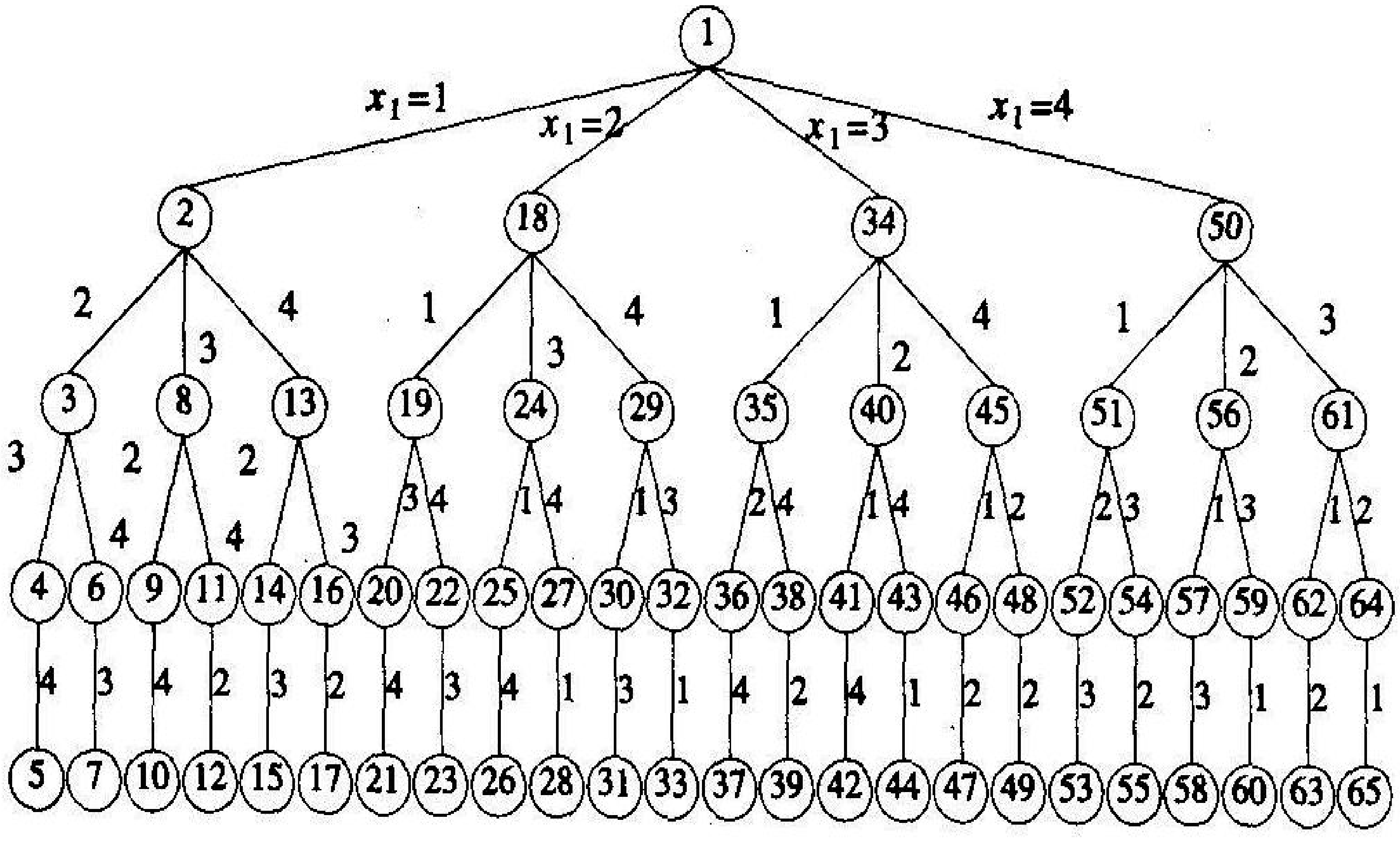
- **Dynamic state space tree** organizations:

Tree organizations that are DEPENDENT on

the problem instance being solved i.e 0/1 knapsack

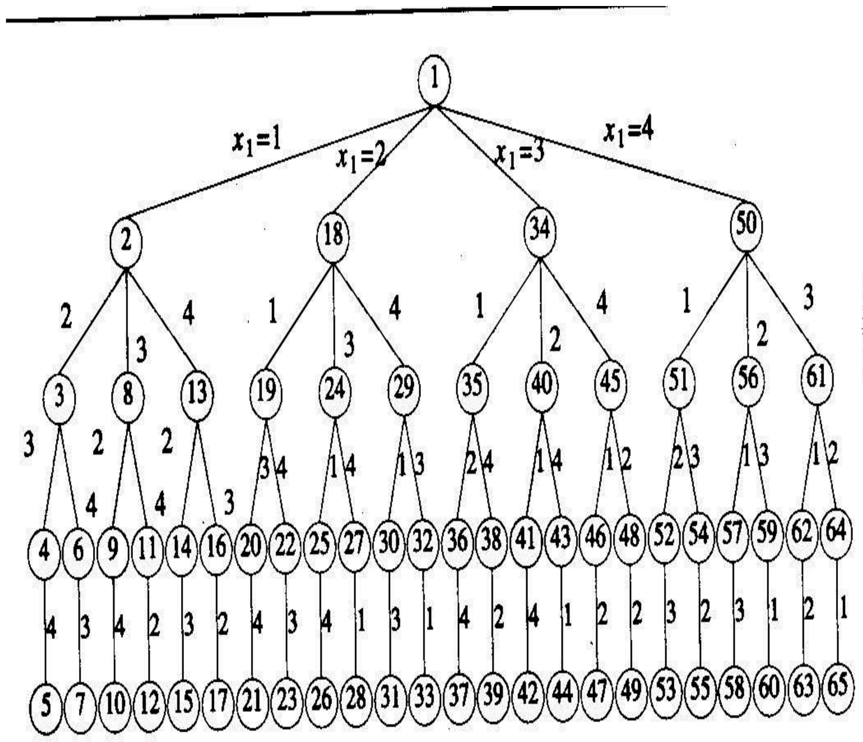


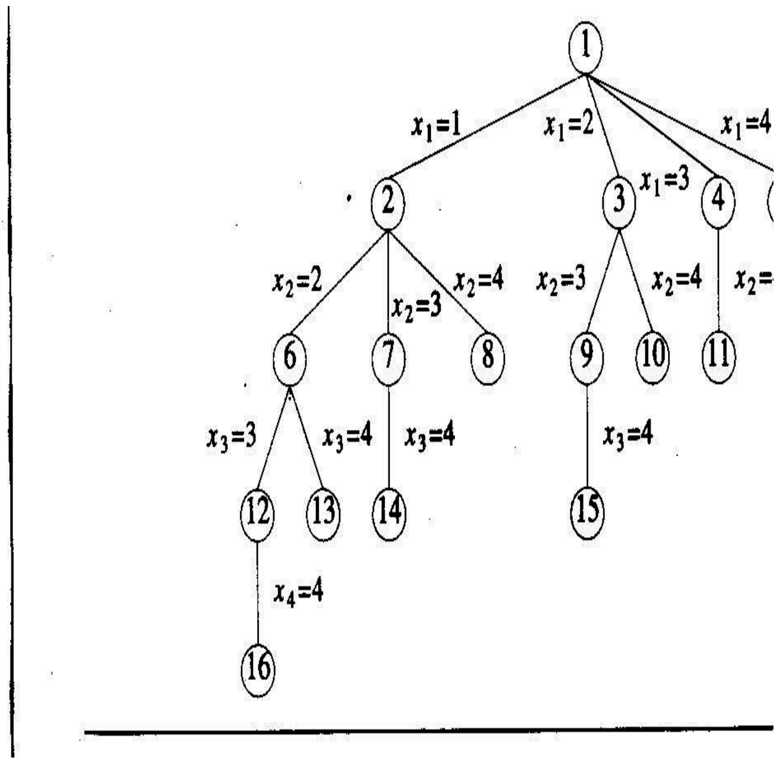
Static State Space tree for 4 queens problem



State Space tree for 4 queens problem

Example Solution: Using Backtracking





8 Queens Problem

- Generalization of problem to nxn chessboard.

- If Chessboard squares are numbered as indices of two dimensional array [1…n,1…n]then we observe that every element on same diagonal that runs from upper left to lower right has same row – column value. Example :a[3,1],a[5,3],a[6,4],a[7,5],a[8,6]. All these have row – column value =2

- Every element on same diagonal that runs from upper right to lower left has same row + column value. Example : a[1,5], a[2,4],a[3,3],a[4,2],a[5,1]

- Suppose two queen are placed at position (i,j) and (k,l) then from above rules we have :

  $(i-j) = (k- l)$ or $(i+j) =(k+l)$

  First Equation implies $j - l = i - k$
  &
  Second Equation Implies $j - l = k - i$

- There fore two queens lie on same diagonal if and only if $| j – l | = | k – i|$

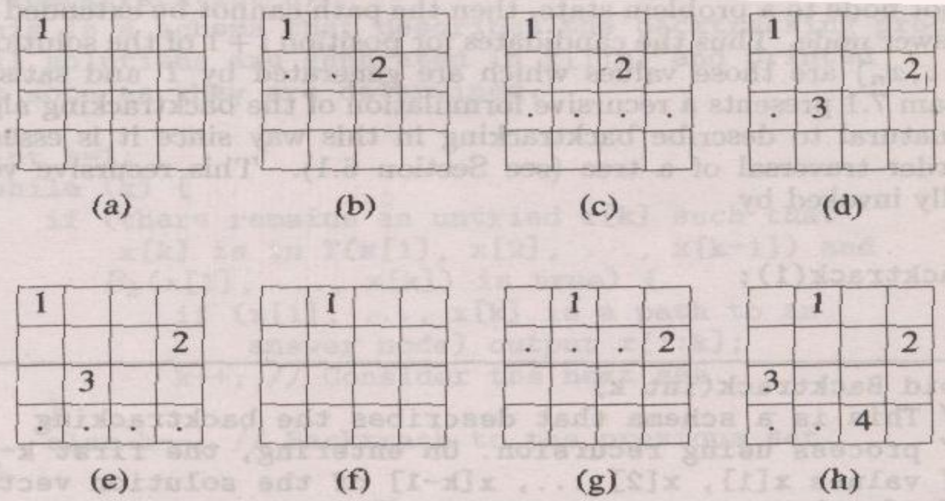
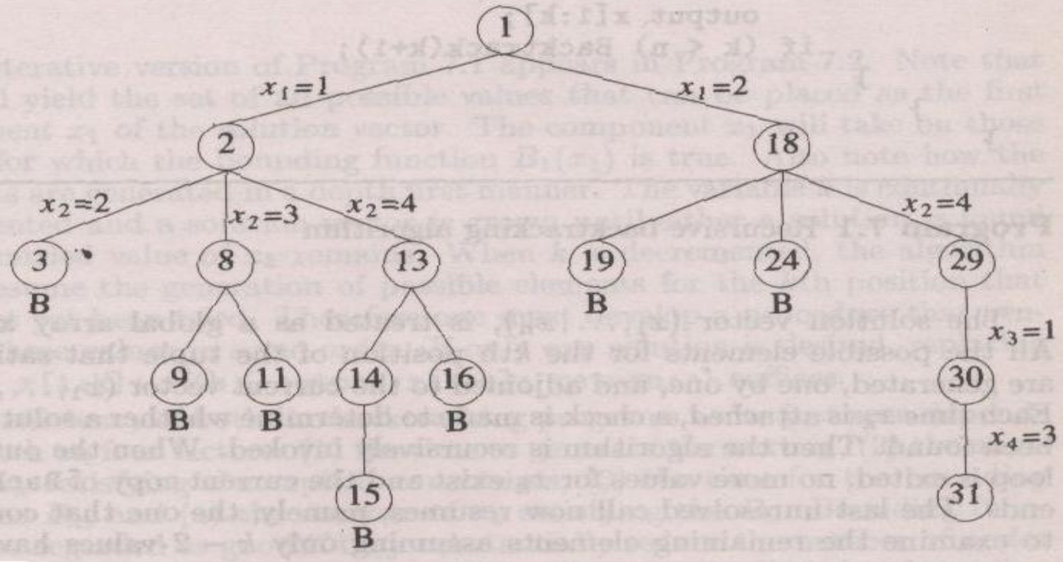


Figure 7.5 Example of a backtrack solution to the 4-queens problem



```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

ithm 7.4 Can a new queen be placed?

```
Algorithm NQueens($k, n$)
// Using backtracking, this procedure prints all
// possible placements of $n$ queens on an $n \times n$
// chessboard so that they are nonattacking.
{
    for $i := 1$ to $n$ do
    {
        if Place($k, i$) then
        {
            $x[k] := i$;
            if ($k = n$) then write ($x[1 : n]$);
            else NQueens($k + 1, n$);
        }
    }
}
```

rithm 7.5 All solutions to the n-queens problem

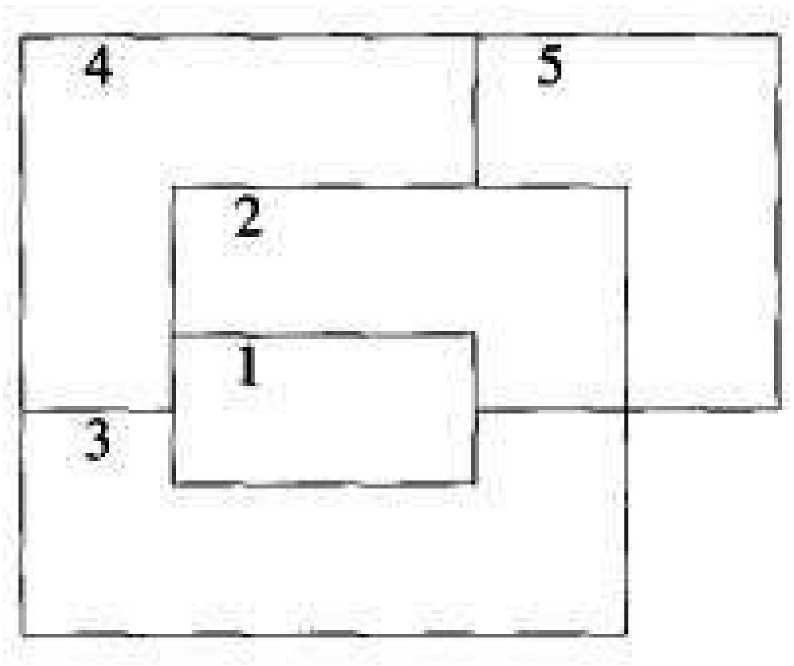
Analysis of N Queens problem

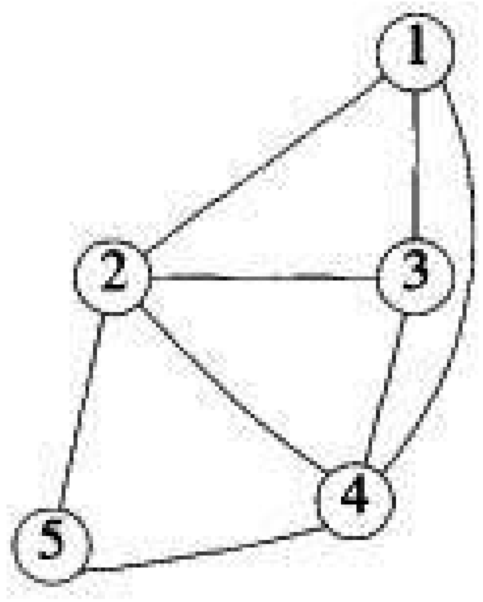
- Using brute force approach for an 8x8 chess board there are (64 C 8) possible ways to place 8 queens or approximately 4.4 billion 8 tuples to examine.
  - (Using brute force approach for an nxn chess board there are (n^2 C n) possible ways to place n queens or n tuples to examine)
- However using NQueens function we allow placement of 8 queens on distinct rows and columns, hence we require to examine of at most 8! Or only 40,320 tuples
- So time complexity N queens problem is O(N!)

Graph Coloring

Backtracking

Planar Graph





Graph Coloring problem

- Map coloring: In cartography, the problem of coloring a map with different colors such that no two adjacent regions have the same color is an instance of the graph coloring problem. This problem arises when creating political maps or geographical maps.

Graph Coloring problem

- **Assign colors to vertices** in a graph such that **no two adjacent vertices have the same color.**
- i.e. given a graph, we want to color each vertex of the graph with one of the available colors (say, k colors), such that no two adjacent vertices have the same color.
  - The **minimum number of colors** required to color a graph such that no two adjacent vertices have the same color is called the chromatic number of the graph.
- Real life applications:

  Scheduling problems

  Register allocation in compilers

  Frequency assignment in wireless communication networks

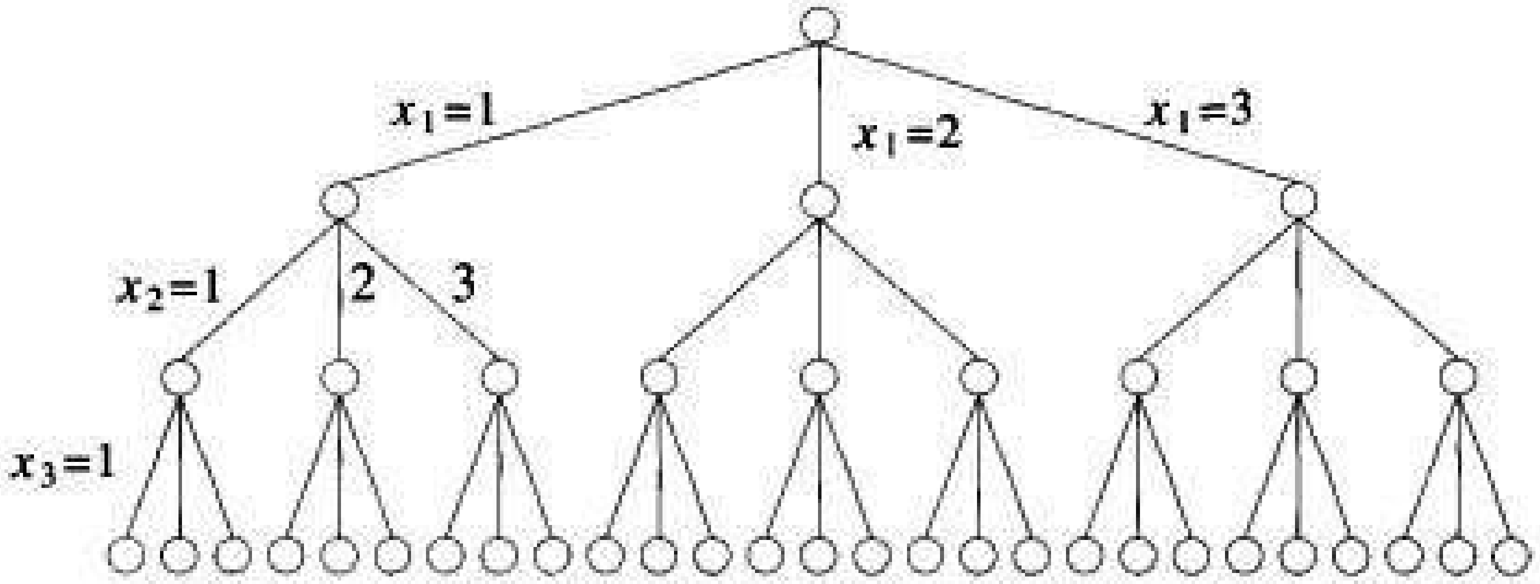
  etc

Graph Coloring Real life Problem :

- Scheduling: Institute situation - Limited number of classrooms, teachers, and time slots available. The scheduling problem can be modeled as a graph, where vertices represent courses and edges represent conflicting time slots. By assigning colors to vertices (courses), we can ensure that no two conflicting courses are scheduled at the same time.
- Timetabling: In universities or schools, there are multiple courses that need to be scheduled, and the problem is to schedule them in such a way that there are no conflicts. This problem can be formulated as a graph coloring problem, where vertices represent courses and edges represent conflicts between courses. By assigning colors to vertices, we can ensure that no two conflicting courses are scheduled at the same time.
- Wireless channel allocation: In wireless communication networks, channels need to be assigned to different users to avoid interference. This problem can be formulated as a graph coloring problem, where vertices represent users and edges represent interference between users. By assigning colors (channels) to vertices, we can ensure that no two adjacent users are assigned the same channel.
- Register allocation in compilers: When compiling a program, the compiler needs to allocate registers to different variables to optimize the use of memory. This problem can be modeled as a graph coloring problem, where vertices represent variables and edges represent conflicts between variables.

Graph Coloring problem

- The problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve the problem exactly for all possible inputs.
- However, there are many efficient algorithms and heuristics that can solve the problem approximately, with varying degrees of accuracy and efficiency.



State space tree for mColoring when n =3 and m = 3

Time Complexity of Graph Coloring

- In general, the time complexity of backtracking algorithms for the graph coloring problem can be expressed **as O(k^n),** where k is the maximum degree of the graph and **n is the number of vertices**.
  - This is because each vertex can be colored with **at most k colors**, and there are n vertices in the **graph**, so there are at most k^n possible color assignments

- However, **the actual running time** of backtracking algorithms can be much better than O(k^n) if the algorithm can prune large portions of the search space using heuristics or other techniques.

M – Coloring Problem

- It is a **DECISION** problem : -
  - To find if it is **possible** to **assign nodes** with **M** different colors, such that no two adjacent vertices of the graph are of the same colors
  - Ans required is – Y/N   or True / false
  - This can be used for the **Optimization Problem of Graph coloring** – where we need to find the MINIMUM number of colors required to color the nodes of a given graph, G, such that no 2 adjascent nodes have same color

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then      // At most m colors have been
                             // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}
```

```
Algorithm NextValue($k$)
// $x[1], \ldots, x[k-1]$ have been assigned integer values in
// the range $[1, m]$ such that adjacent vertices have distinct
// integers. A value for $x[k]$ is determined in the range
// $[0, m]$. $x[k]$ is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex $k$. If no such color exists, then $x[k]$ is 0.
{
    repeat
    {
        $x[k] := (x[k] + 1)$ mod $(m + 1)$; // Next highest color.
        if $(x[k] = 0)$ then return; // All colors have been used.
        for $j := 1$ to $n$ do
        {   // Check if this color is
            // distinct from adjacent colors.
            if $((G[k, j] \neq 0)$ and $(x[k] = x[j]))$
            // If $(k, j)$ is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if $(j = n + 1)$ then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

Unit III	**Backtracking, Branch and Bound**
Backtracking: The General Method 8 Queen's problem, Graph Coloring **Branch and Bound: 0/1 Knapsack,** **Traveling Salesperson Problem.**	

General Strategy: Branch and Bound

The General Technique

- Like Backtracking, this also searches for the solution in state space tree, however, the state space tree is searched in a manner that, all children of the E-node are generated, before any other live node can become E-node.
  - Or we can say that – E-node (node being expanded / whose children are being generated) remains E-node till it is dead (i.e all its children are generated)
  - BFS, and D-Search both can be generalized to Branch and Bound
- Branch and bound can solve optimization problems, such as finding the minimum or maximum of a function or finding the shortest path in a graph.

- In branch and bound, the smaller subproblems are solved recursively. The solutions to the subproblems are used to determine if certain branches of the **solution space** can be pruned, resulting in a more efficient search.

- The implementation can use different strategies. The choice of strategy depends on the nature of the problem and the resources available.

Branch and Bound

- First we need to conceive a state-space tree for the given problem
- Then generate nodes such that E-node remains E-node till all it’s children are generated
- Each answer node x has cost c(x) associated with it, and a minimum

3 common strategies for implementing the branch and bound algorithm

- These strategies use queue or stack to store the subproblems to be solved.
  - FIFO (First In, First Out)
  - LIFO (Last In, First Out)
  - LC (Least Cost)
    - FIFO is useful when the solution space is relatively flat and uniform,
    - LIFO is useful when the solution space is deep and narrow,
    - and LC is useful when there is a good heuristic estimate of the optimal solution.

3 common strategies for implementing the branch and bound algorithm

- **FIFO (First In, First Out):** In this strategy, the subproblems are stored in a queue, and the first subproblem to be added to the queue is the first one to be removed for processing. This strategy is also known as breadth-first search because it explores all the subproblems at a given level of the search tree before moving on to the next level.
- **LIFO (Last In, First Out):** In this strategy, the subproblems are stored in a stack, and the last subproblem to be added to the stack is the first one to be removed for processing. This strategy is also known as depth-first search because it explores the deepest branches of the search tree first before backtracking to shallower levels.
- **LC (Least Cost):** In this strategy, the subproblems are sorted by their estimated cost, and the subproblem with the lowest estimated cost is selected for processing first. This strategy is also known as best-first search because it explores the most promising branches of the search tree first. The estimated cost can be based on factors such as the distance to the goal, the remaining time, or the number of unsatisfied constraints.

FIFO – 4 Queens

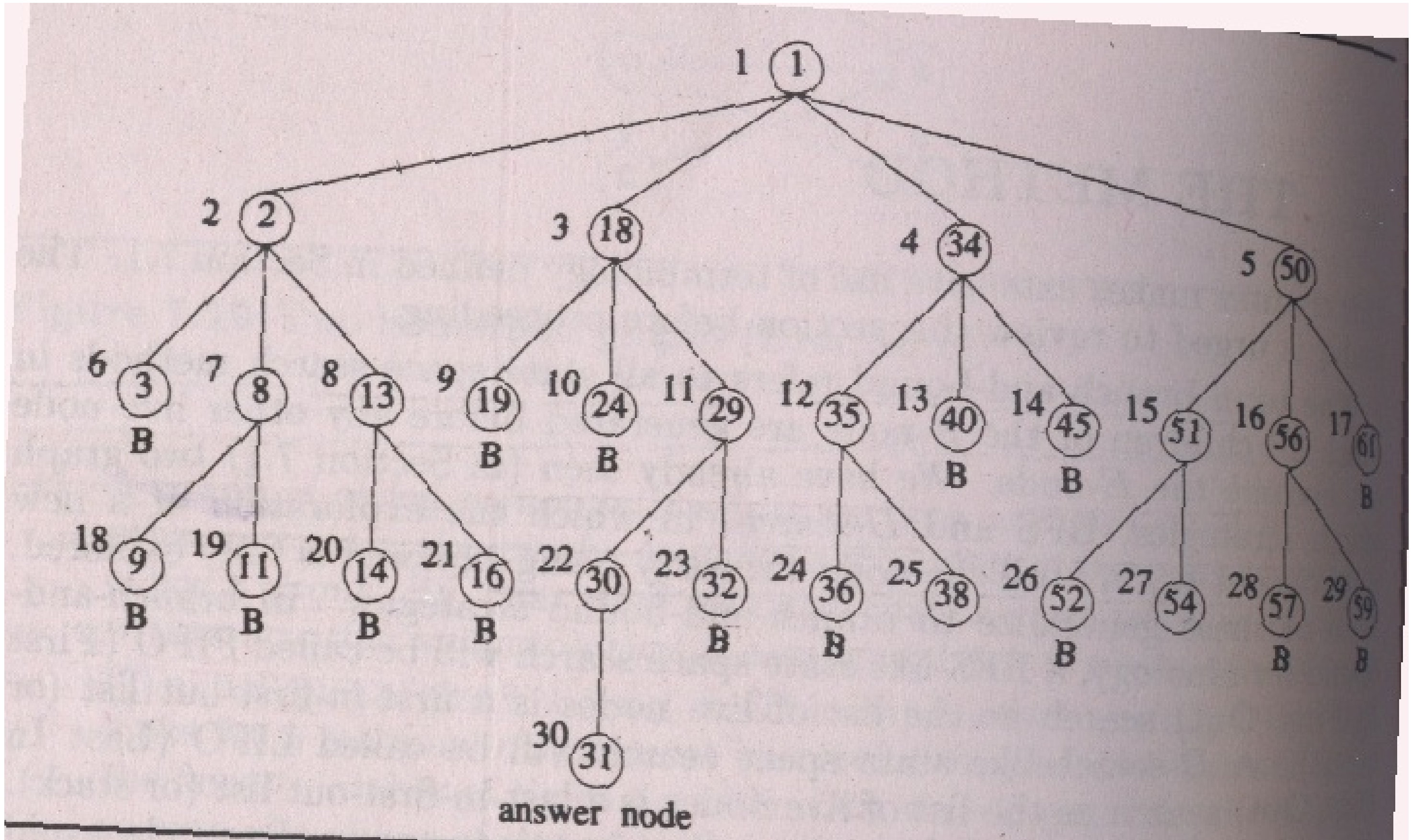


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

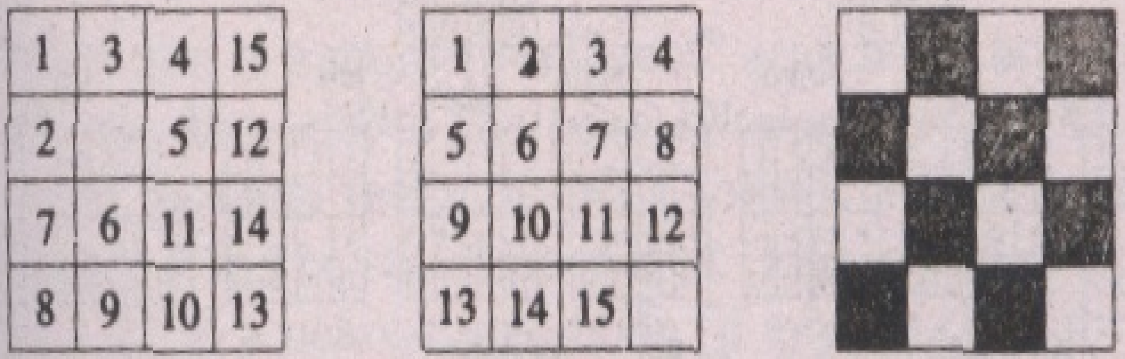
LC Branch and Bound

```
Algorithm LCSearch($t$)
// Search $t$ for an answer node.
{
    if $*t$ is an answer node then output $*t$ and return;
    $E := t$; // $E$-node.
    Initialize the list of live nodes to be empty;
    repeat
    {
        for each child $x$ of $E$ do
        {
            if $x$ is an answer node then output the path
                from $x$ to $t$ and return;
            Add($x$); // $x$ is a new live node.
            $(x \rightarrow parent) := E$; // Pointer for path to root.
        }
        if there are no more live nodes then
        {
            write ("No answer node"); return;
        }
        $E :=$ Least();
    } until (false);
}
```

15-Puzzle problem

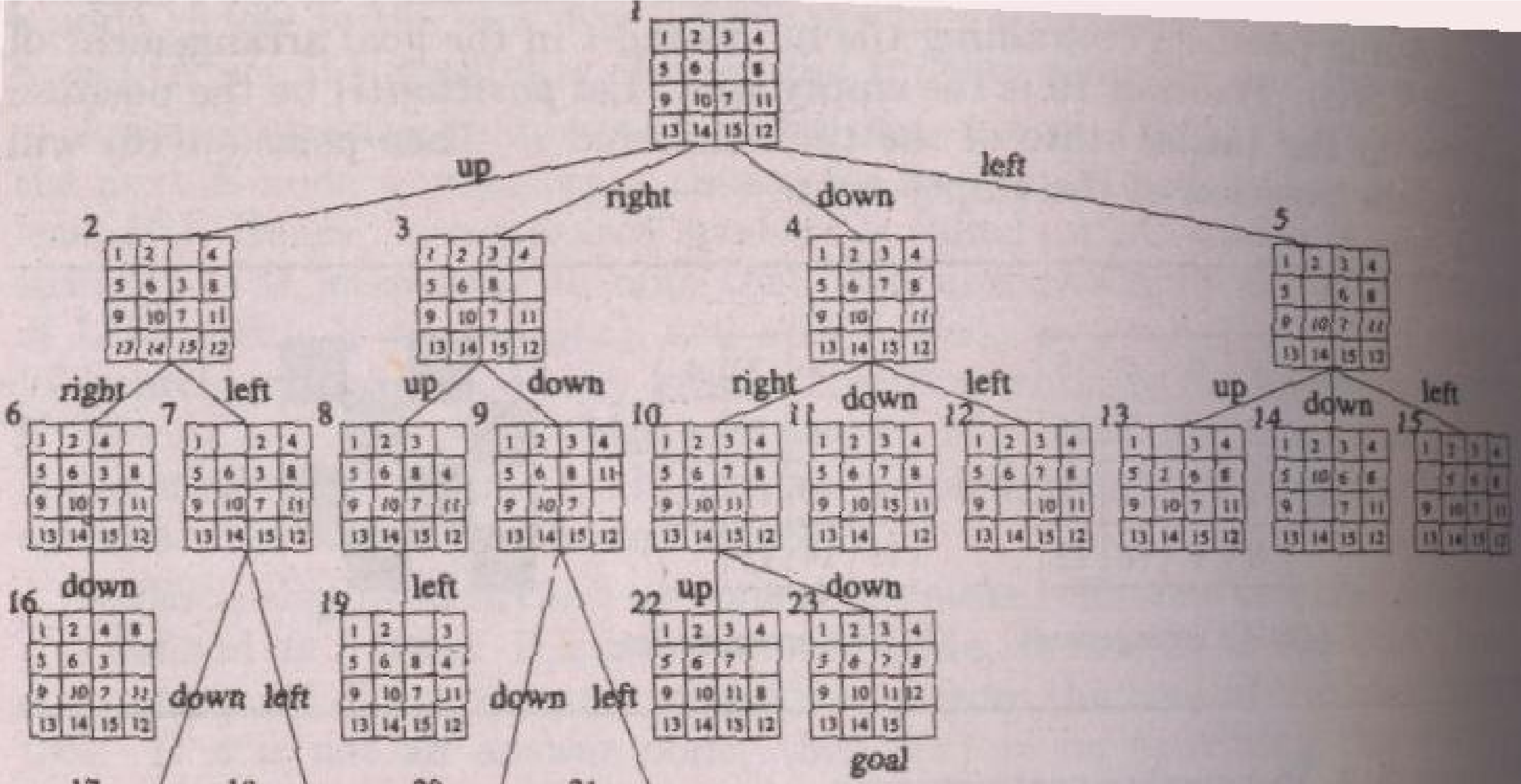
(State space tree)

(a) A Random arrangement of 15 Puzzle (b) A Goal arrangement of 15 Puzzle (c) 4 x 4 locations for 15 Puzzle problem



(a) An arrangement (b) Goal arrangement (c)

15-puzzle arrangements



Edges are labeled according to the direction
in which the empty space moves

Figure 8.3 Part of the state space tree for the 15-puzzle

Depth First (Not Suitable)

- Each step gets us further from GOAL

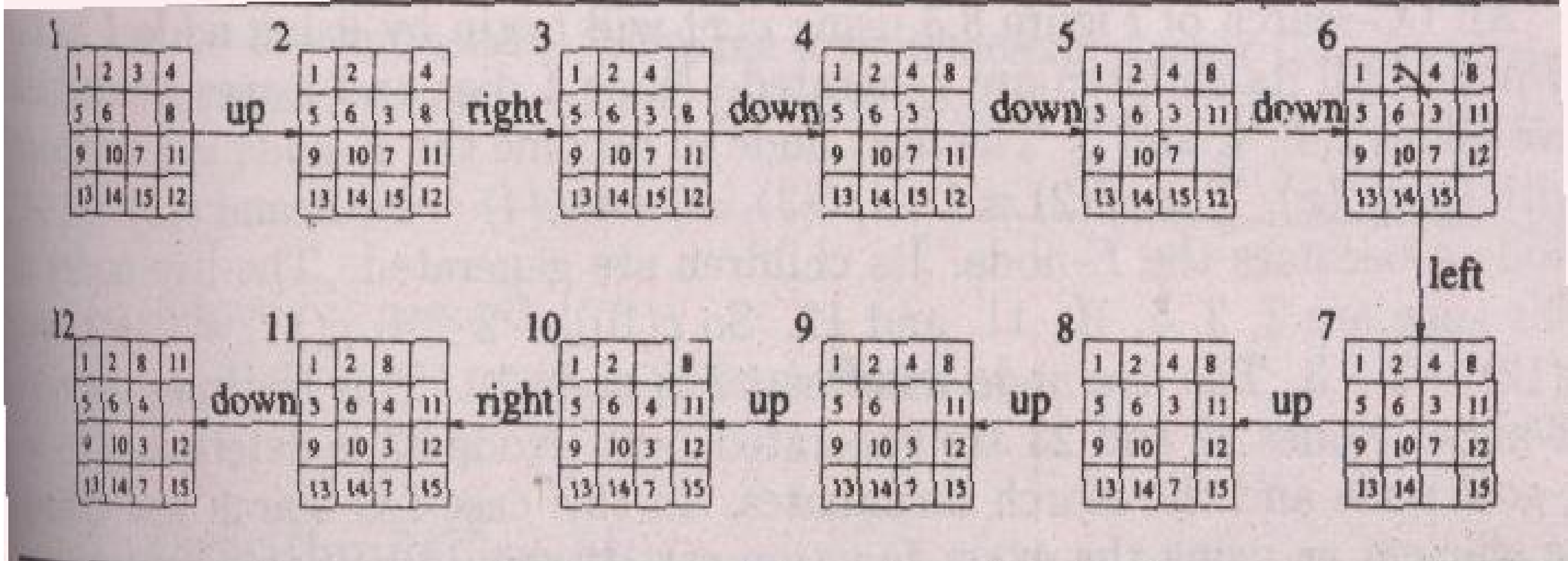
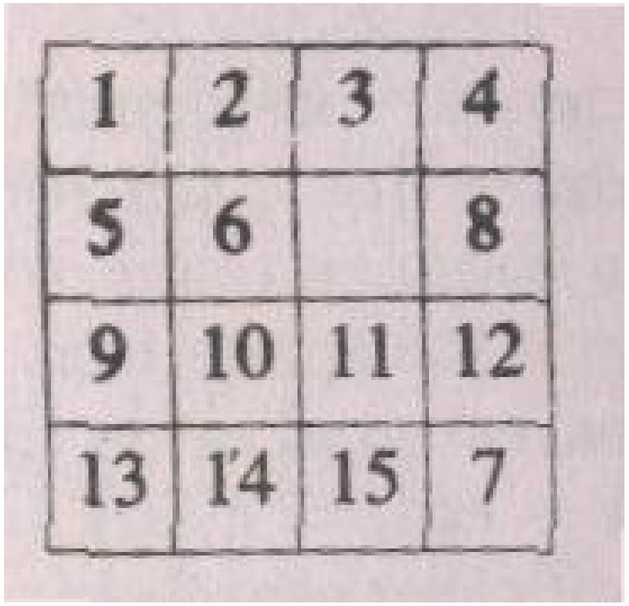


Figure 8.4 First ten steps in a depth first search

15 puzzle sample state

- Only 1 tile out of place but, more than 1 moves will be required for putting that tile in place



1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

0/1 Knapsack

Branch and Bound

Problem: 0/1 Knapsack

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i$$

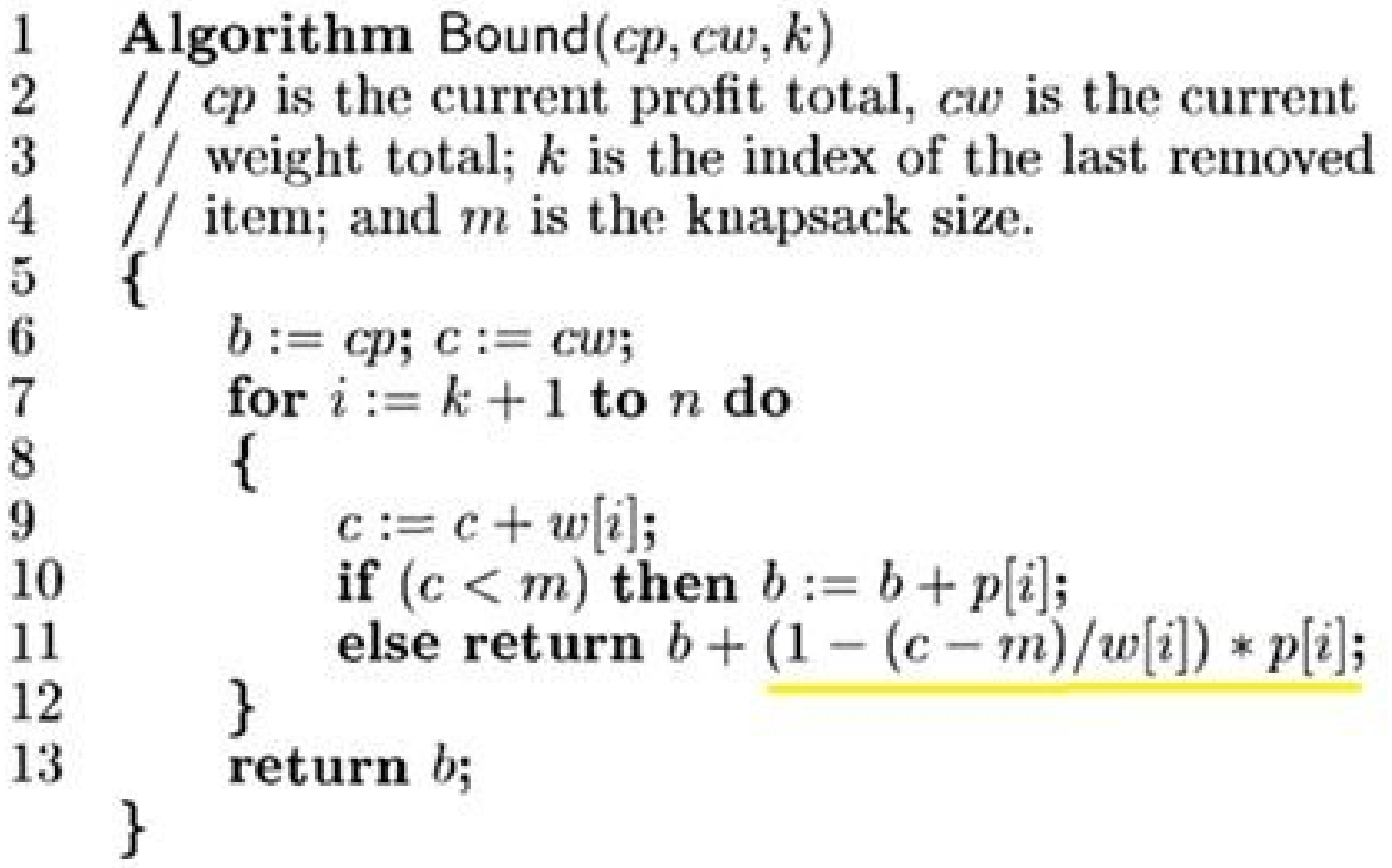
$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq m$$

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

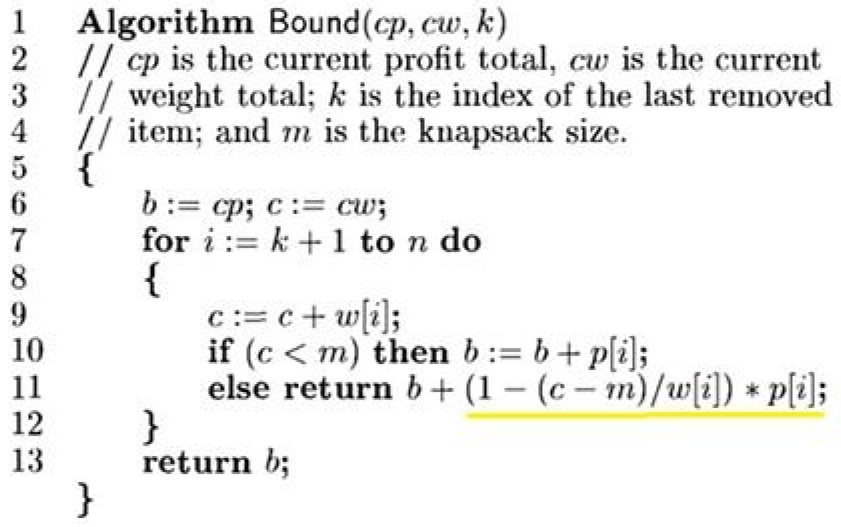
Sample Problem

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

Bound



```
Algorithm Bound($cp, cw, k$)
// $cp$ is the current profit total, $cw$ is the current
// weight total; $k$ is the index of the 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 $b$;
}
```



```
1   Algorithm Bound(cp, cw, k)
2   // cp is the current profit total, cw is the current
3   // weight total; k is the index of the last removed
4   // item; and m is the knapsack size.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           c := c + w[i];
10          if (c < m) then b := b + p[i];
11          else return b + (1 - (c - m)/w[i]) * p[i];
12      }
13      return b;
    }
```

<u>Explanation of Line 11</u>: we add to “b” the profit obtained by adding fraction of object “i”

This can be understood by observing that, in line 9: weight w[i] is added to c (even if it will exceed m).
So in line 11: when c>m , we subtract that amount (c-m) and divide it by weight w[i], and then take the fraction 1- ((c-m)/w[i]) to get the actual fraction to multiply with p[i], to get the correct fractional profit

For example before line 9: :
If m=15, w[i]=9, and c=12,
We actually need to add only 3 of the w[i] (so as not to exceed 15), and so **<u>(3/w[i]) * p[i]</u>** needs to be added to total profit, which is (3/9)*p[i], but
after line 9, c=12+9=21 , which is 6 greater than 15.
So, in line 11: we do (c-m) =6, then divide it by w[i] to get 6/w[i] and then subtract this fraction from 1, to get **3/w[i]** and then multiply it by **p[i],** to get **<u>(3/w[i]) * p[i]</u>**

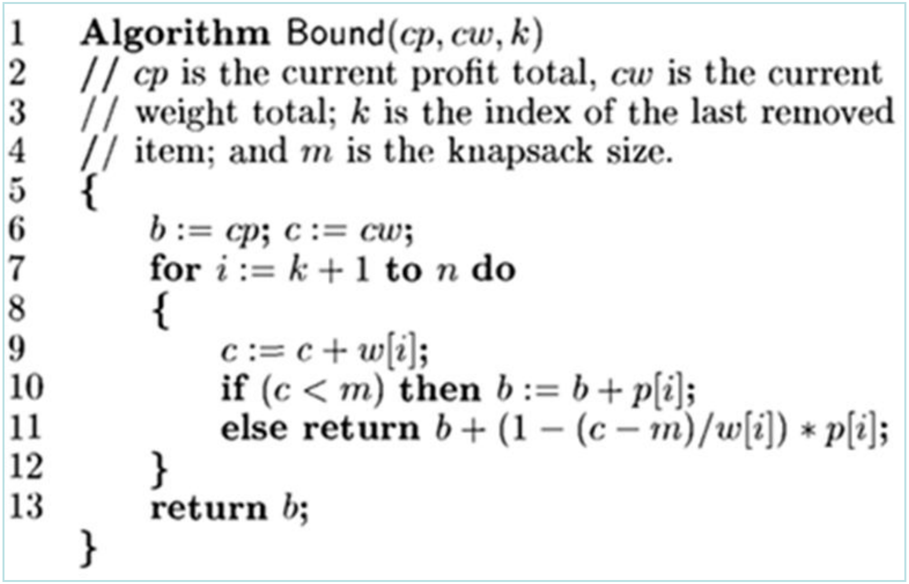
Finding upper bound for 0/1 knapsack

```
Algorithm UBound($cp, cw, k, m$)
// $cp, cw, k$, and $m$ have the same meanings as in
// Algorithm 7.11. $w[i]$ and $p[i]$ are respectively
// the weight and profit of the $i$th object.
{
    $b := cp$; $c := cw$;
    for $i := k + 1$ to $n$ do
    {
        if ($c + w[i] \leq m$) then
        {
            $c := c + w[i]$; $b := b - p[i]$;
        }
    }
    return $b$;
}
```

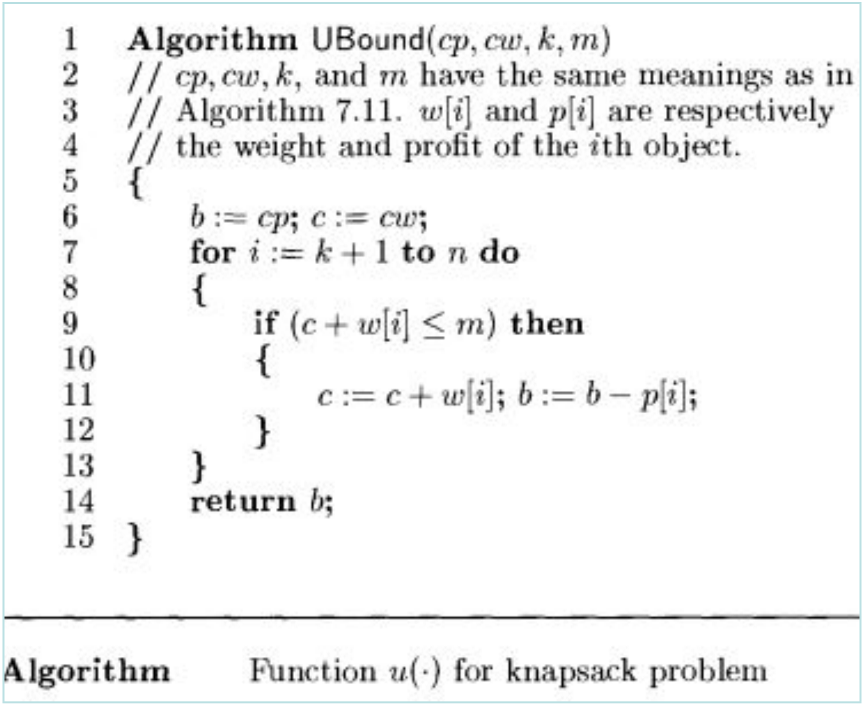
Sample Problem

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

Call initially with – Bound(0,0,0) UBound(0,0,0,15)



```
Algorithm Bound($cp, cw, k$)
// $cp$ is the current profit total, $cw$ is the current
// weight total; $k$ is the index of the 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 $b$;
}
```



```
Algorithm UBound($cp, cw, k, m$)
// $cp, cw, k$, and $m$ have the same meanings as in
// Algorithm 7.11. $w[i]$ and $p[i]$ are respectively
// the weight and profit of the $i$th object.
{
    $b := cp$; $c := cw$;
    for $i := k + 1$ to $n$ do
    {
        if ($c + w[i] \leq m$) then
        {
            $c := c + w[i]$; $b := b - p[i]$;
        }
    }
    return $b$;
}
```

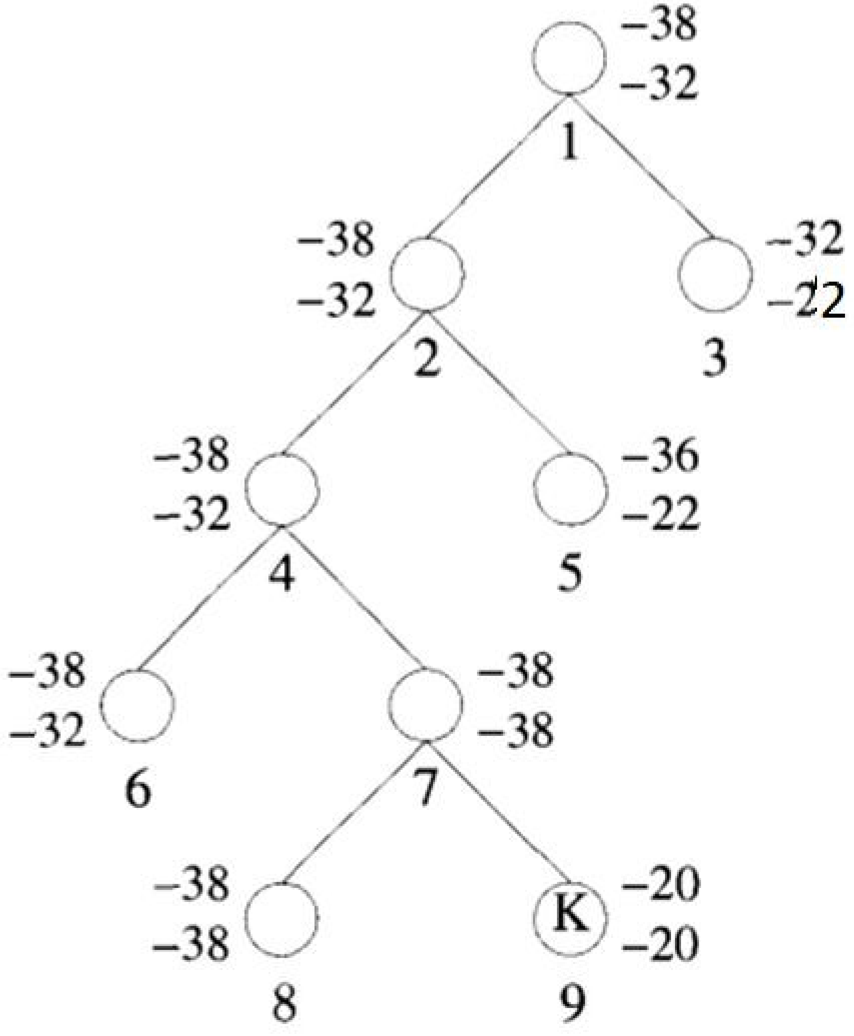
Algorithm Function $u(\cdot)$ for knapsack problem

Sample Problem

n=4, m=15 ;

(p1,p2,p3,p4)=(10,10,12,18)

; (w1,w2,w3,w4=(2,4,6,9)

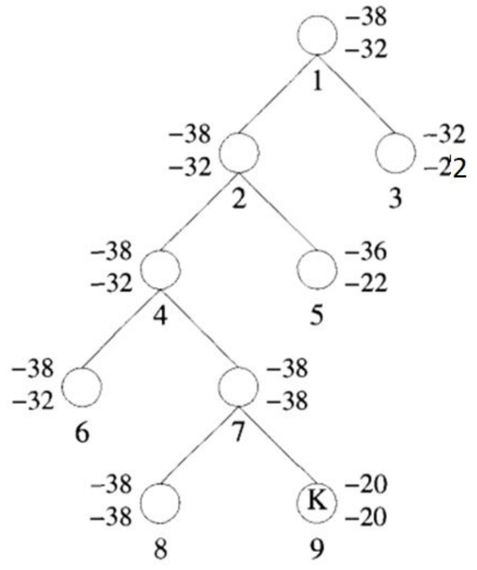


Upper number = $\hat{c}$

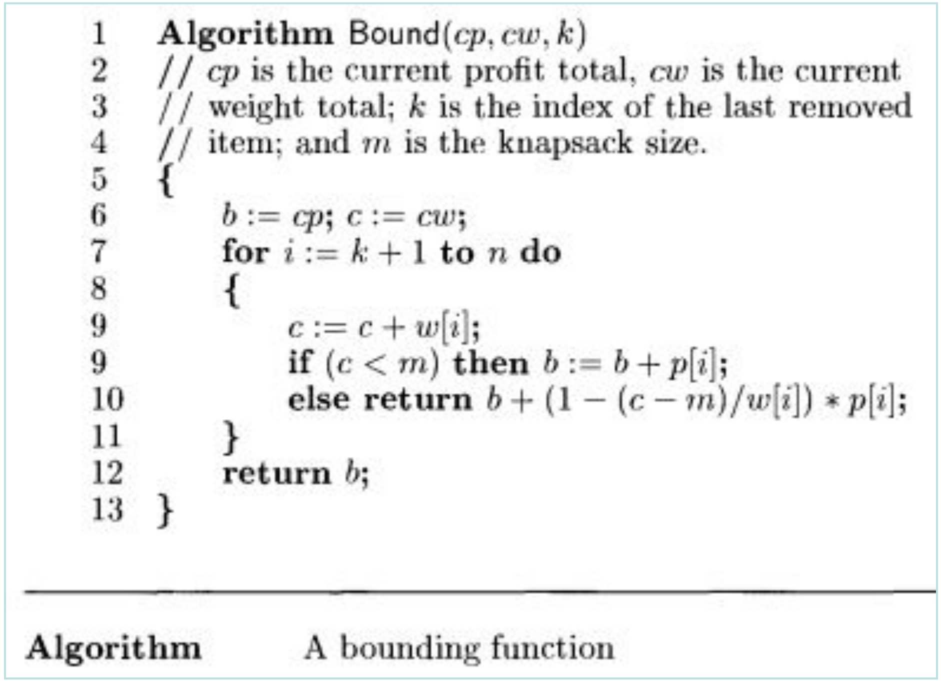
Lower number = u

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

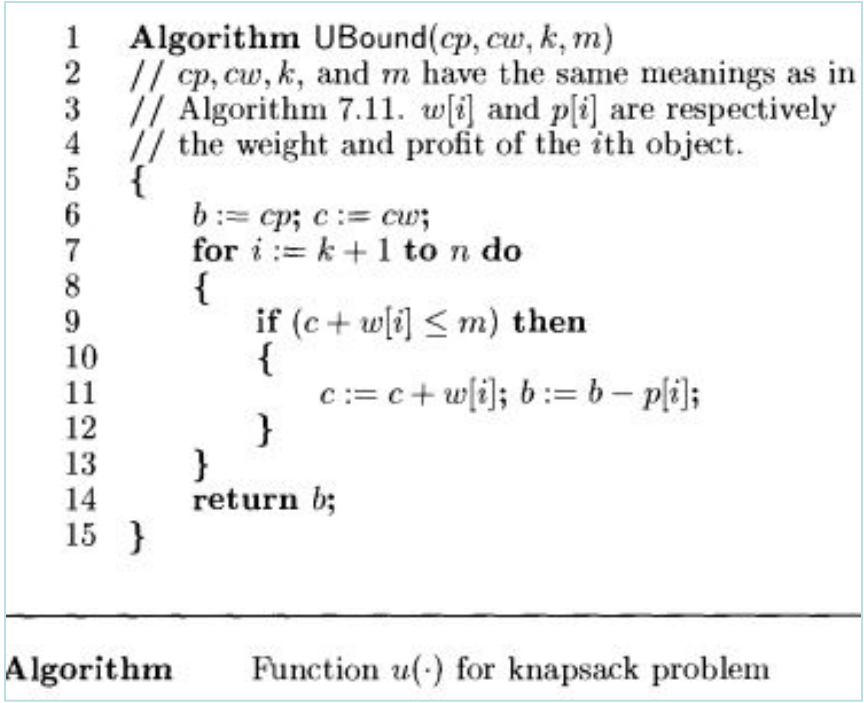
Call bound initially with –
Bound(0,0,1)



Upper number = $\hat{c}$
Lower number = u


```
Algorithm Bound(cp, cw, k)
// cp is the current profit total, cw is the current
// weight total; k is the index of the 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;
}
```

Algorithm A bounding function


```
Algorithm UBound(cp, cw, k, m)
// cp, cw, k, and m have the same meanings as in
// Algorithm 7.11. w[i] and p[i] are respectively
// the weight and profit of the ith object.
{
    b := cp; c := cw;
    for i := k + 1 to n do
    {
        if (c + w[i] ≤ m) then
        {
            c := c + w[i]; b := b − p[i];
        }
    }
    return b;
}
```

Algorithm Function $u(\cdot)$ for knapsack problem

Travelling Salesperson Problem (TSP)

Branch and Bound

Goal of TSP

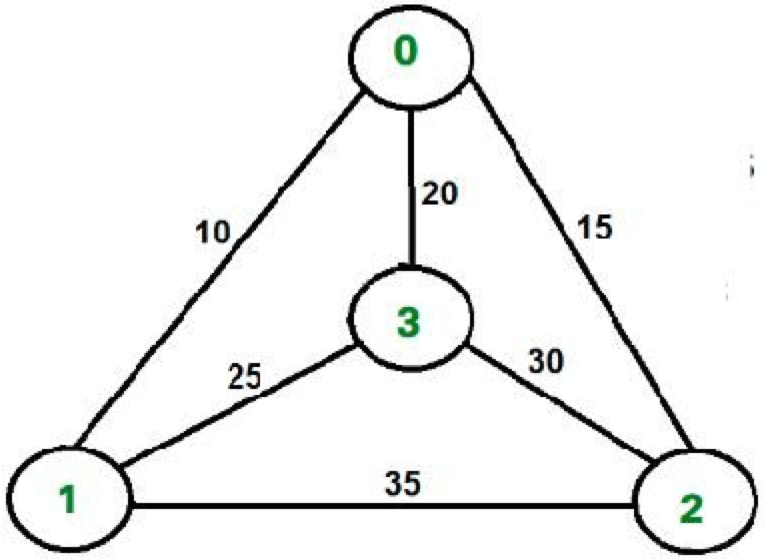
- The goal of the TSP is - to find the shortest possible route that **visits a set of cities exactly once** and **returns to the starting city.**

TSP-BB : Points to be covered

- Problem statement
- LCBB technique for TSP
- Reduced Cost Matrix
- State Space Tree

The least cost branch and bound technique (LCBB) algorithm for solving the traveling salesperson problem (TSP).

TSP Branch & Bound



TSP with LCBB

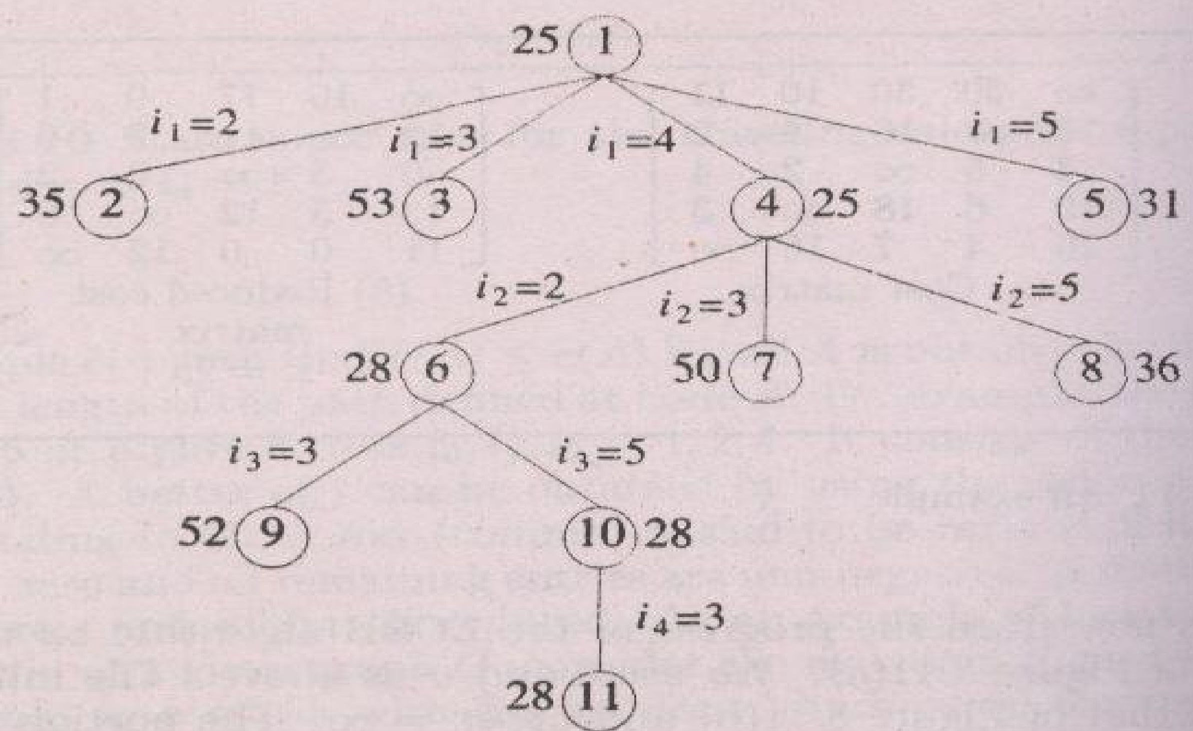
$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

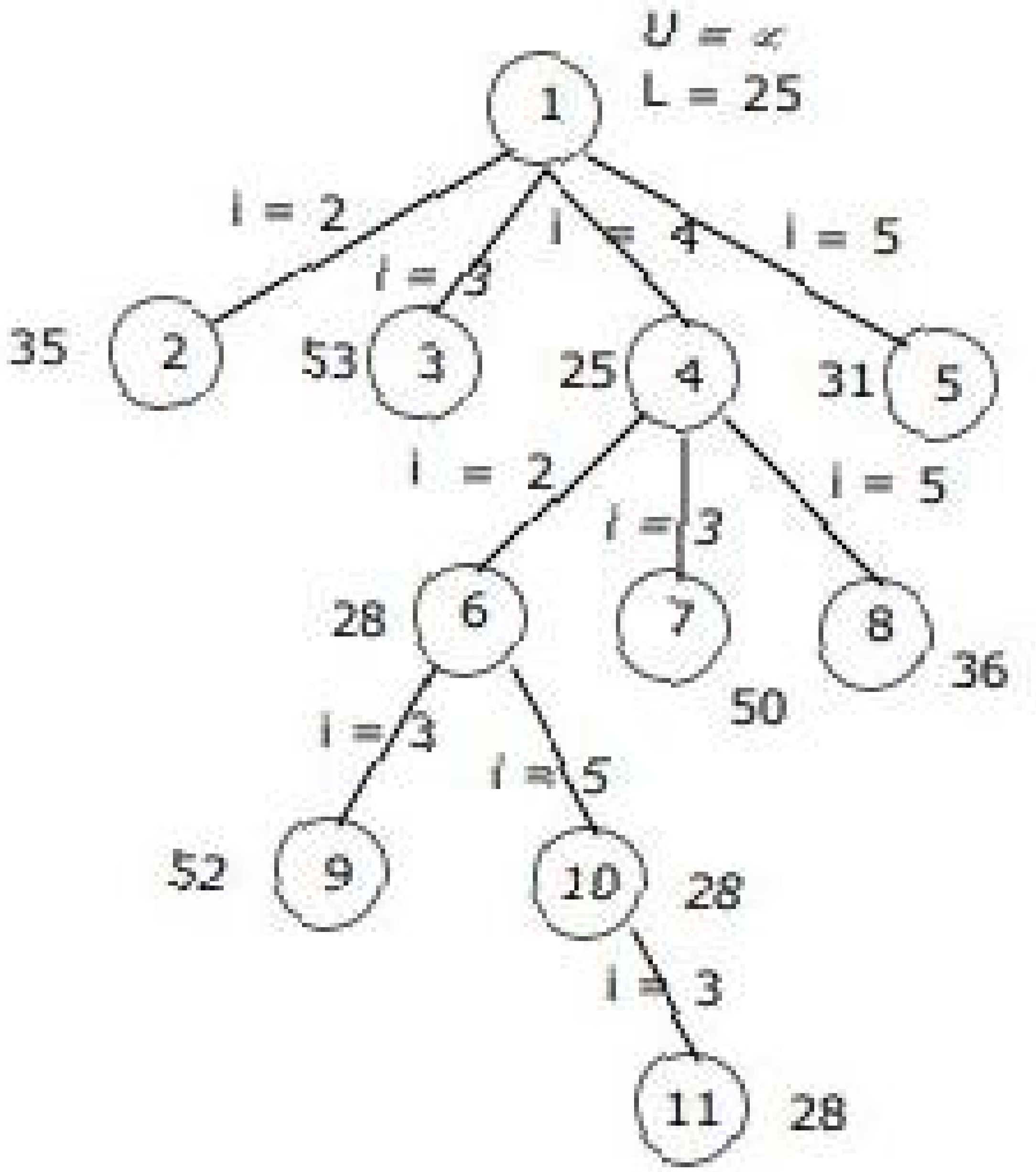
$$\hat{c}(S) = \hat{c}(R) + A(i,j) + r$$



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

2 State space tree generated by procedure LCBB

State Space Tree for the given problem



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

(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

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 11 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 12 & \infty & 0 \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(a) Node 2

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 1 & \infty & 11 & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & 12 & \infty & 0 \\ 0 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Node 3

$$\begin{bmatrix} \infty & 7 & \infty & 0 & \infty \\ \infty & \infty & \infty & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(c) Node 4

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 0 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 1 & \infty & 0 \\ \infty & 0 & \infty & 12 & \infty \end{bmatrix}$$

(d) Node 5

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

(e) Node 6

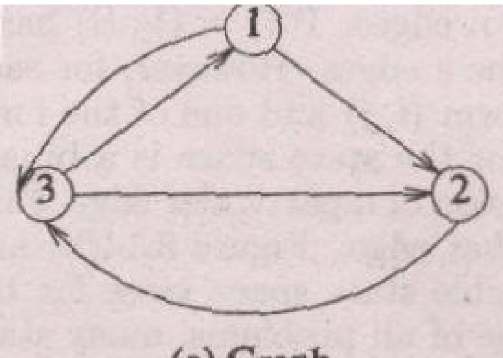
$$\begin{bmatrix} \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(f) Node 7

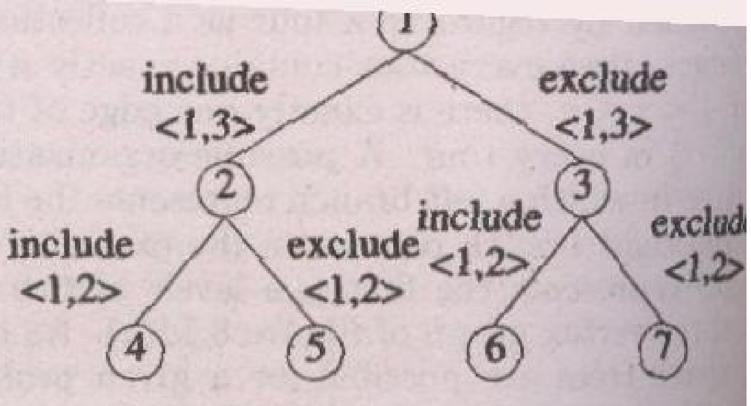
Procedure for solving traveling sale person problem

```
Algorithm LCSearch($t$)
// Search $t$ for an answer node.
{
    if $*t$ is an answer node then output $*t$ and return;
    $E := t$; // $E$-node.
    Initialize the list of live nodes to be empty;
    repeat
    {
        for each child $x$ of $E$ do
        {
            if $x$ is an answer node then output the path
                from $x$ to $t$ and return;
            Add($x$); // $x$ is a new live node.
            $(x \rightarrow parent) := E$; // Pointer for path to root.
        }
        if there are no more live nodes then
        {
            write ("No answer node"); return;
        }
        $E :=$ Least();
    } until (false);
}
```

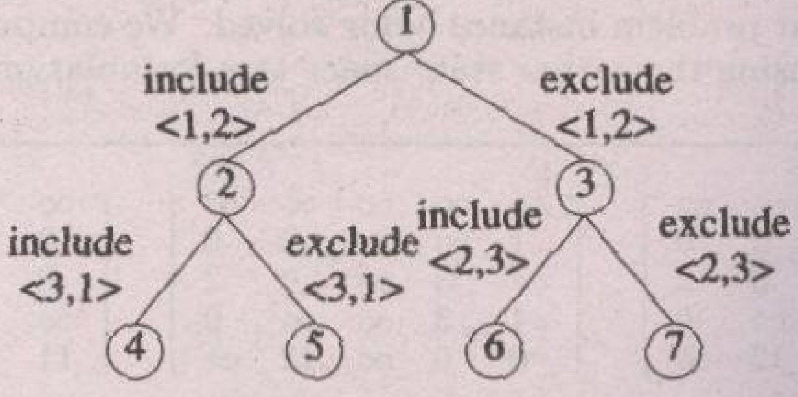
LCBB with **Dynamic binary tree formulation**



(a) Graph

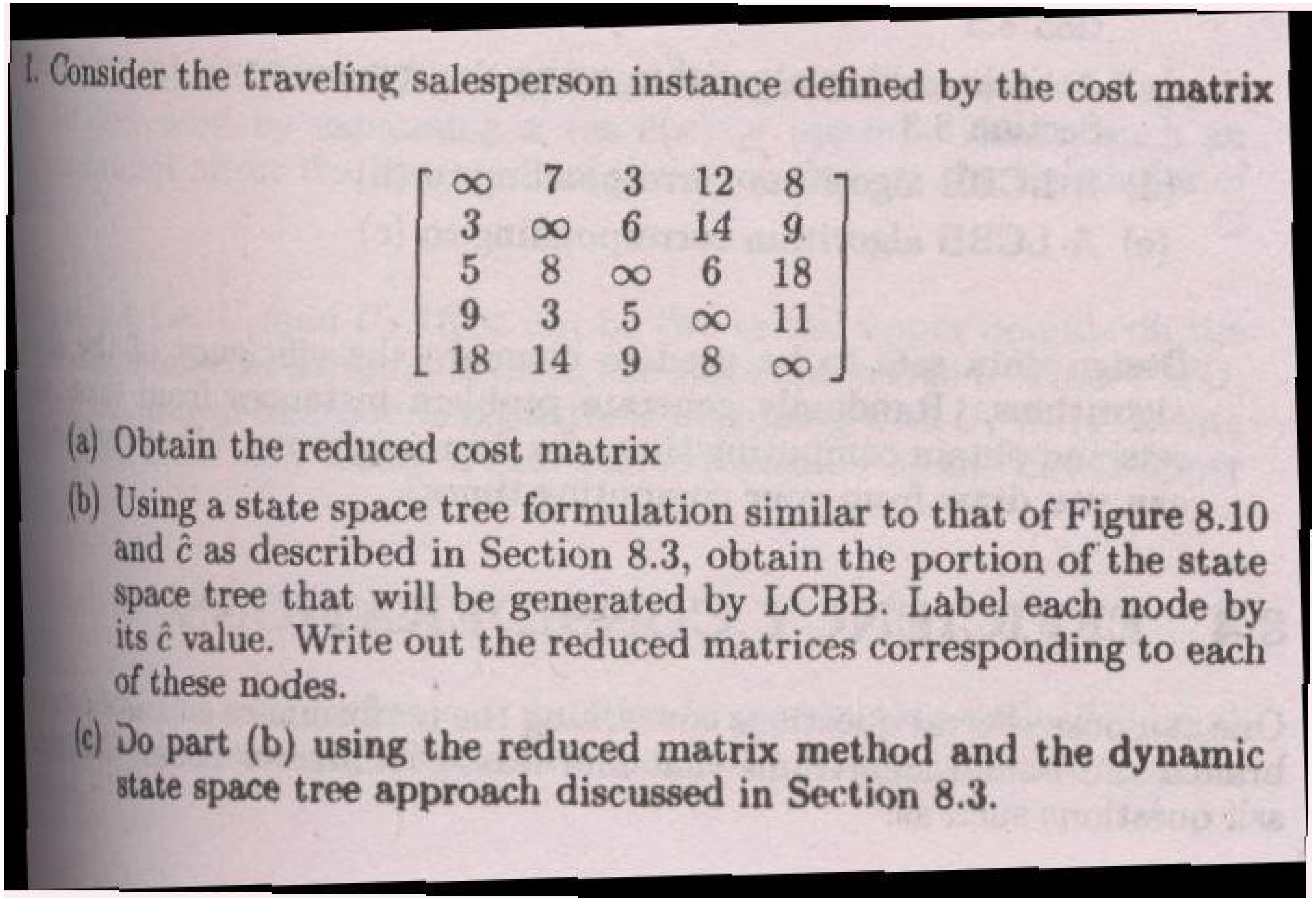


(b) Part of a state space tree



(c) Part of a state space tree

(Refer to Book: DAA by Horowitz, Sahani)

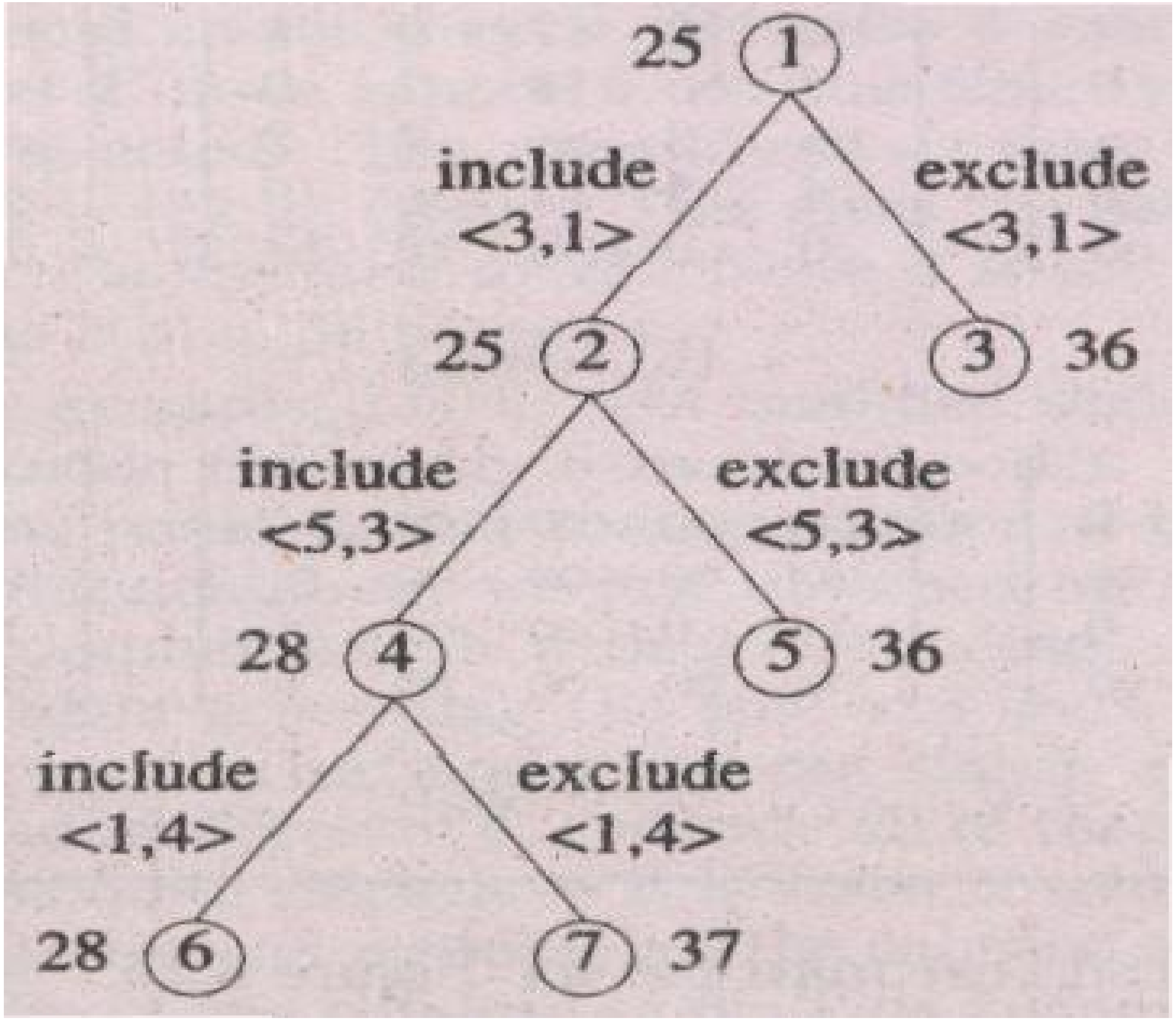

1. Consider the traveling salesperson instance defined by the cost matrix

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

(a) Obtain the reduced cost matrix

(b) Using a state space tree formulation similar to that of Figure 8.10 and $\hat{c}$ as described in Section 8.3, obtain the portion of the state space tree that will be generated by LCBB. Label each node by its $\hat{c}$ value. Write out the reduced matrices corresponding to each of these nodes.

(c) Do part (b) using the reduced matrix method and the dynamic state space tree approach discussed in Section 8.3.


25
1
include <3,1>
exclude <3,1>
25
2
3
36
include <5,3>
exclude <5,3>
28
4
5
36
include <1,4>
exclude <1,4>
28
6
7
37

```
private static boolean isSafePlace(int column, int Qi, int[] board) {

//check for all previously placed queens

for (int i = 0; i < Qi; i++) {

if (board[i] == column) { // the ith Queen(previous) is in same column

return false;

}

//the ith Queen is in diagonal

//(r1, c1) - (r2, c1). if |r1-r2| == |c1-c2| then they are in diagonal

if (Math.abs(board[i] - column) == Math.abs(i - Qi)) {

return false;
}
//the ith Queen is in diagonal
//(r1, c1) - (r2, c1). if |r1-r2| == |c1-c2| then they are in diagonal
if (Math.abs(board[i] - column) == Math.abs(i - Qi)) {
return false;
}
}
return true;
}
```

Thank You !!