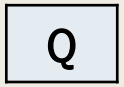
Design and Analysis of Algorithms

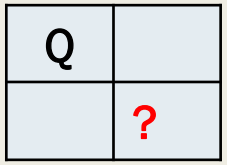
Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Backtracking – N Queens Problem

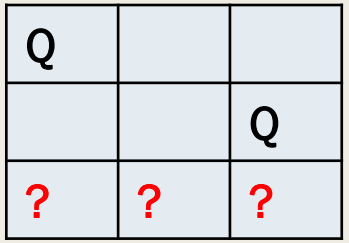
- The problem is to place *n* queens on an *n* × *n* chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- When n= 1?

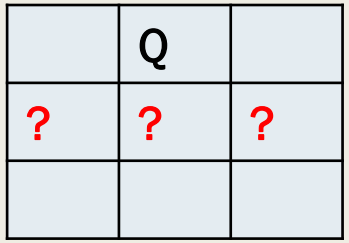


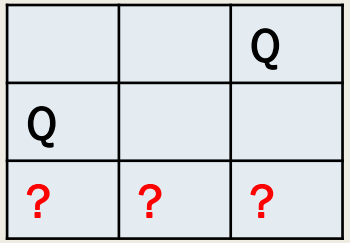
- When n =2 ?



- When n= 3?



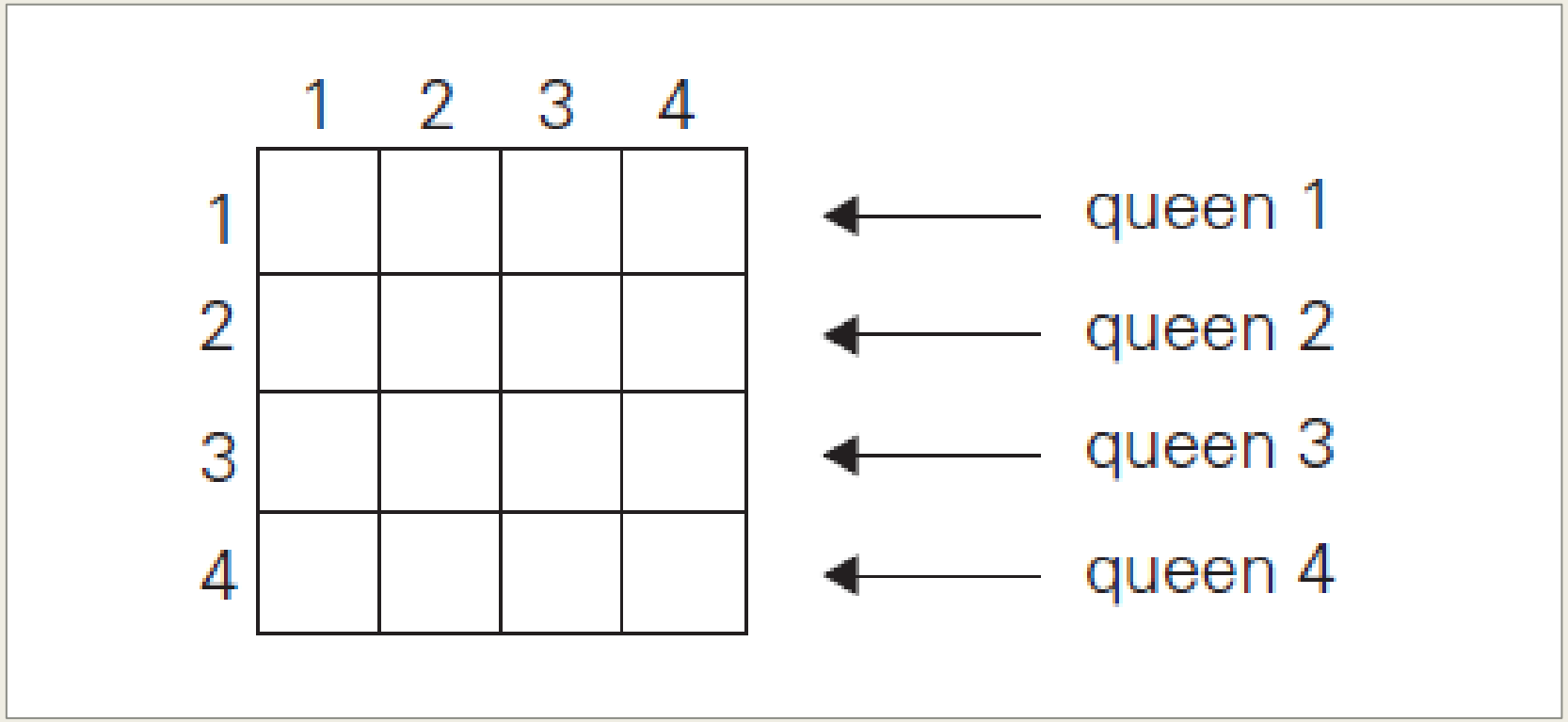




- When n=4?

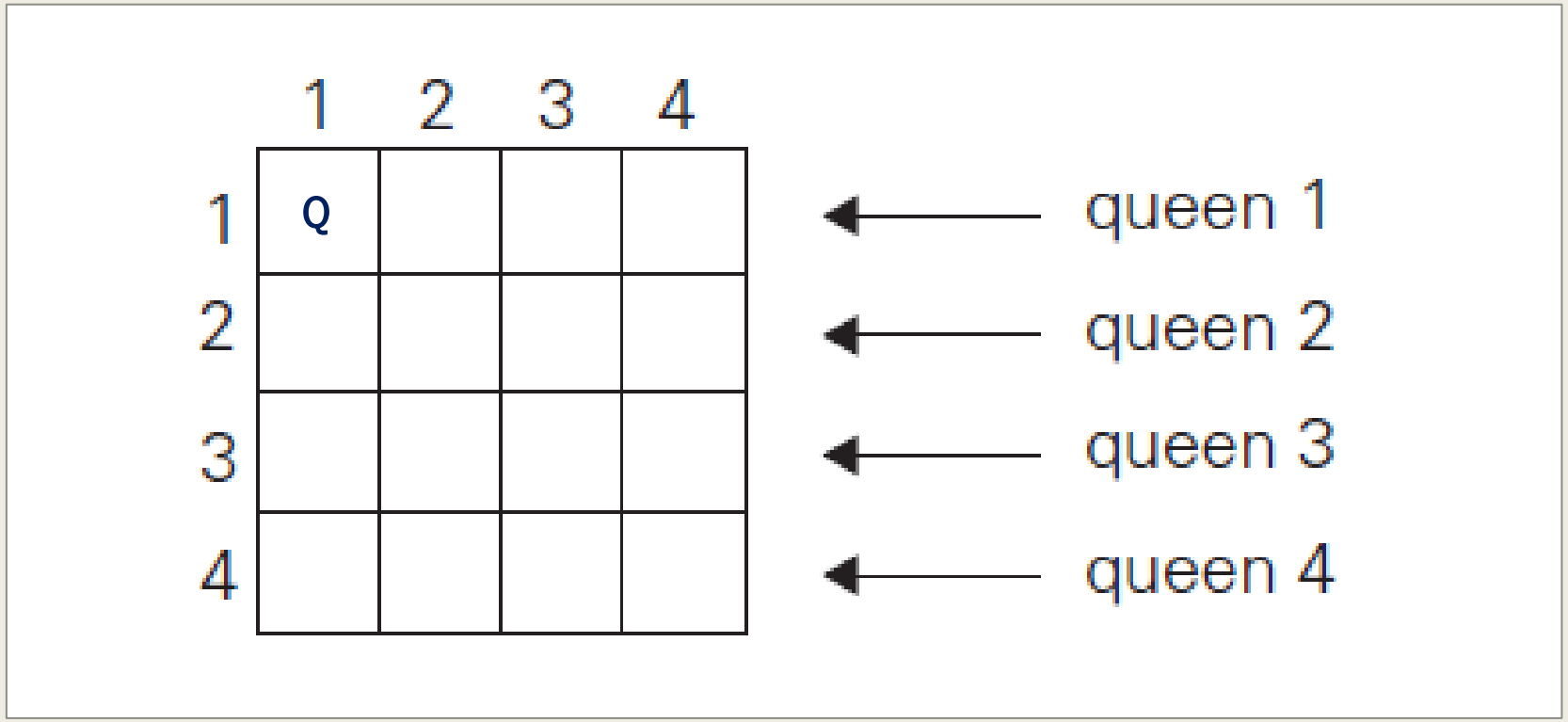
Backtracking – N Queens Problem

- When n=4?



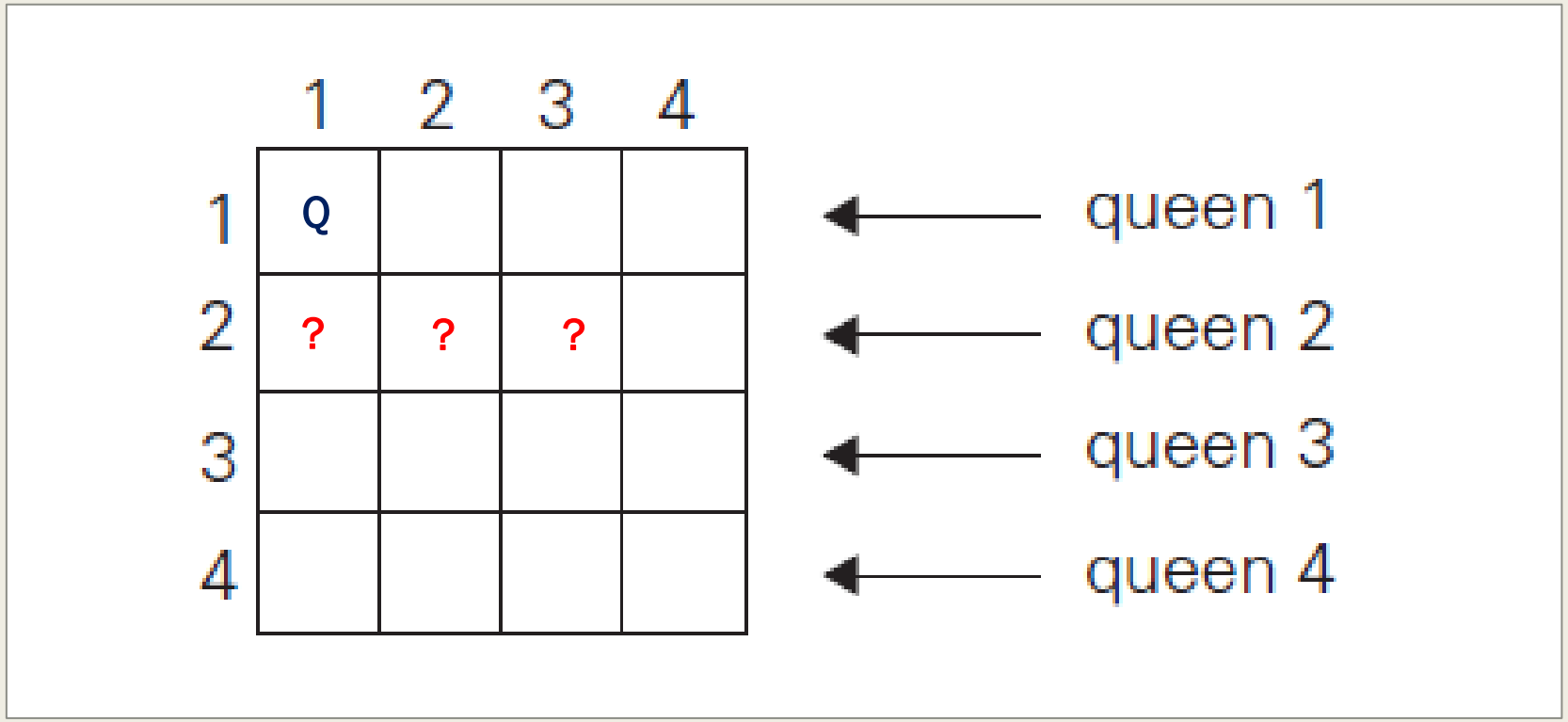
Backtracking – N Queens Problem

- When n=4?



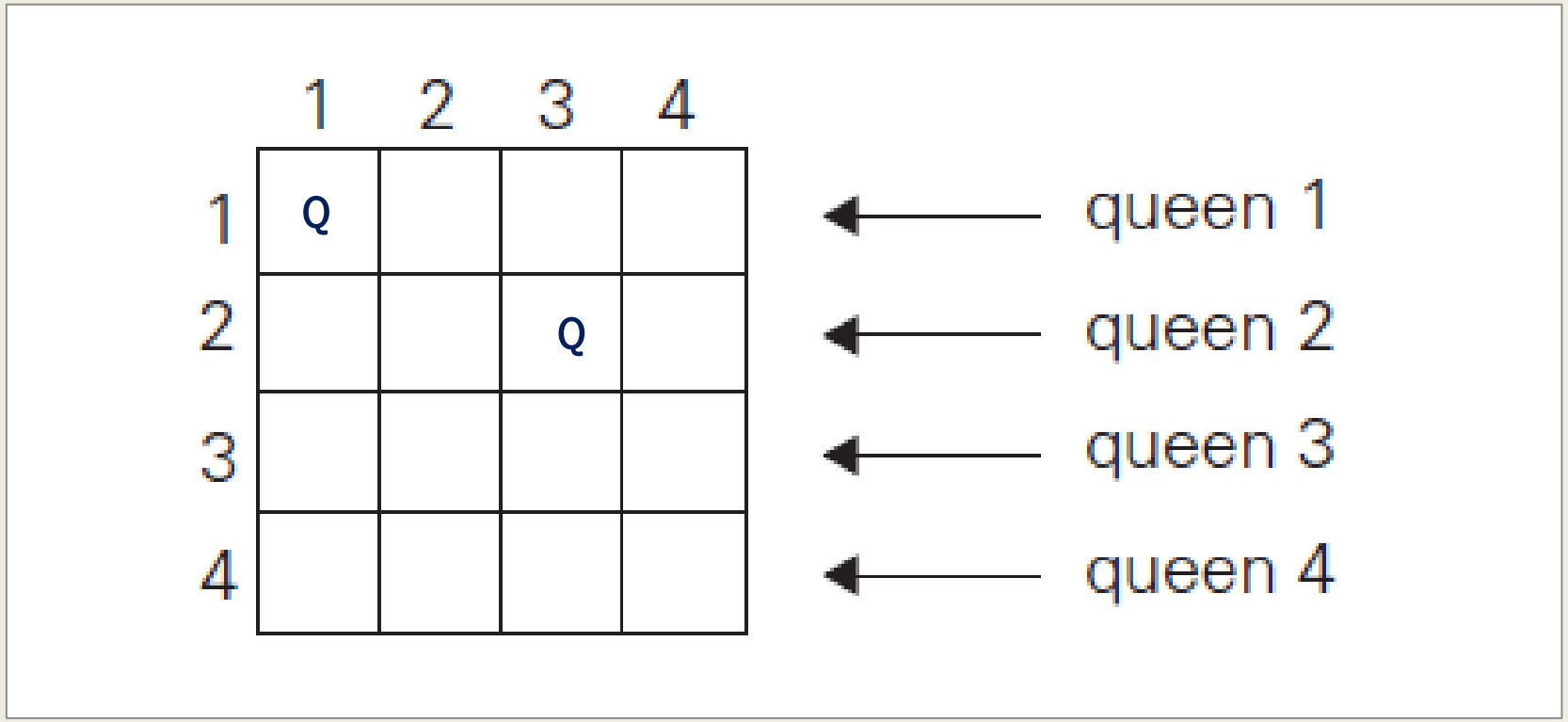
Backtracking – N Queens Problem

- When n=4?



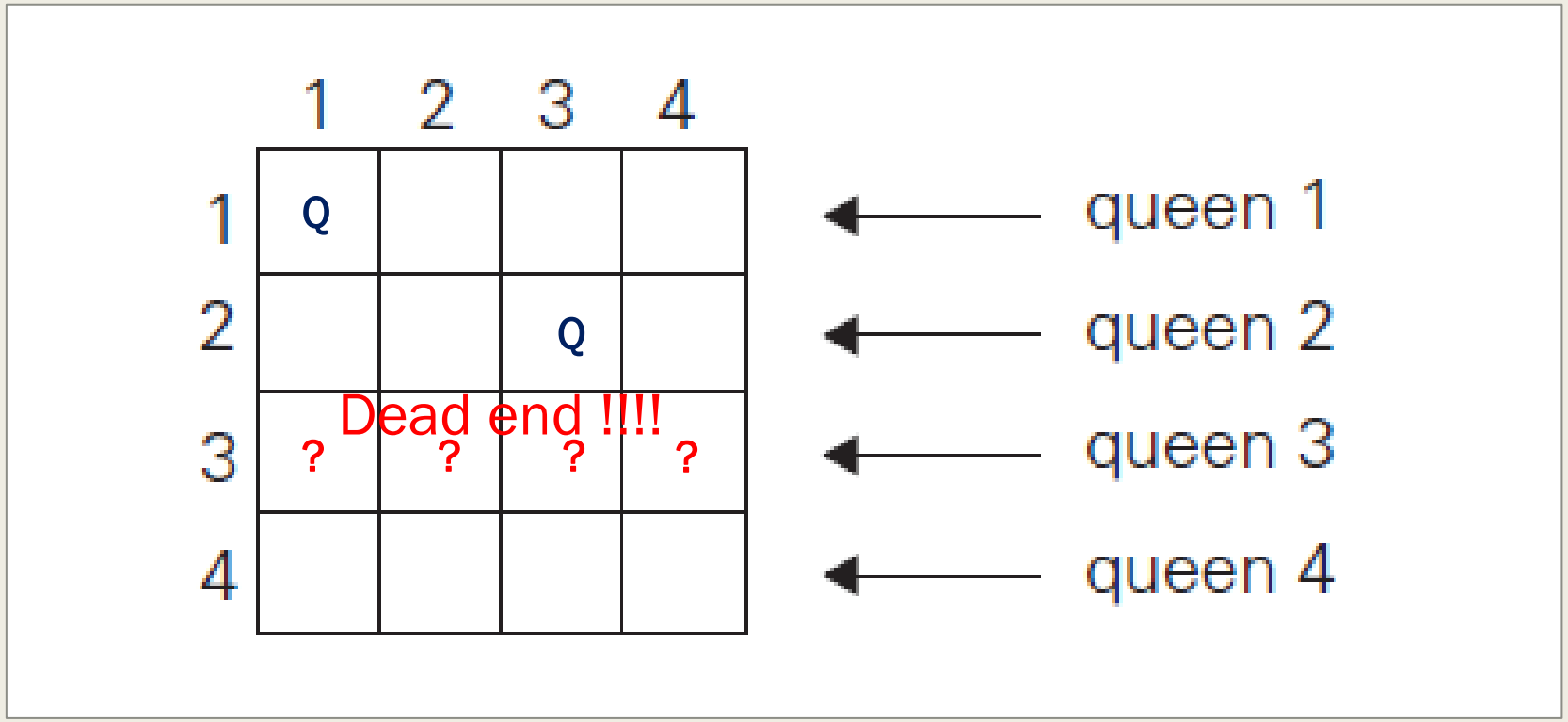
Backtracking – N Queens Problem

- When n=4?



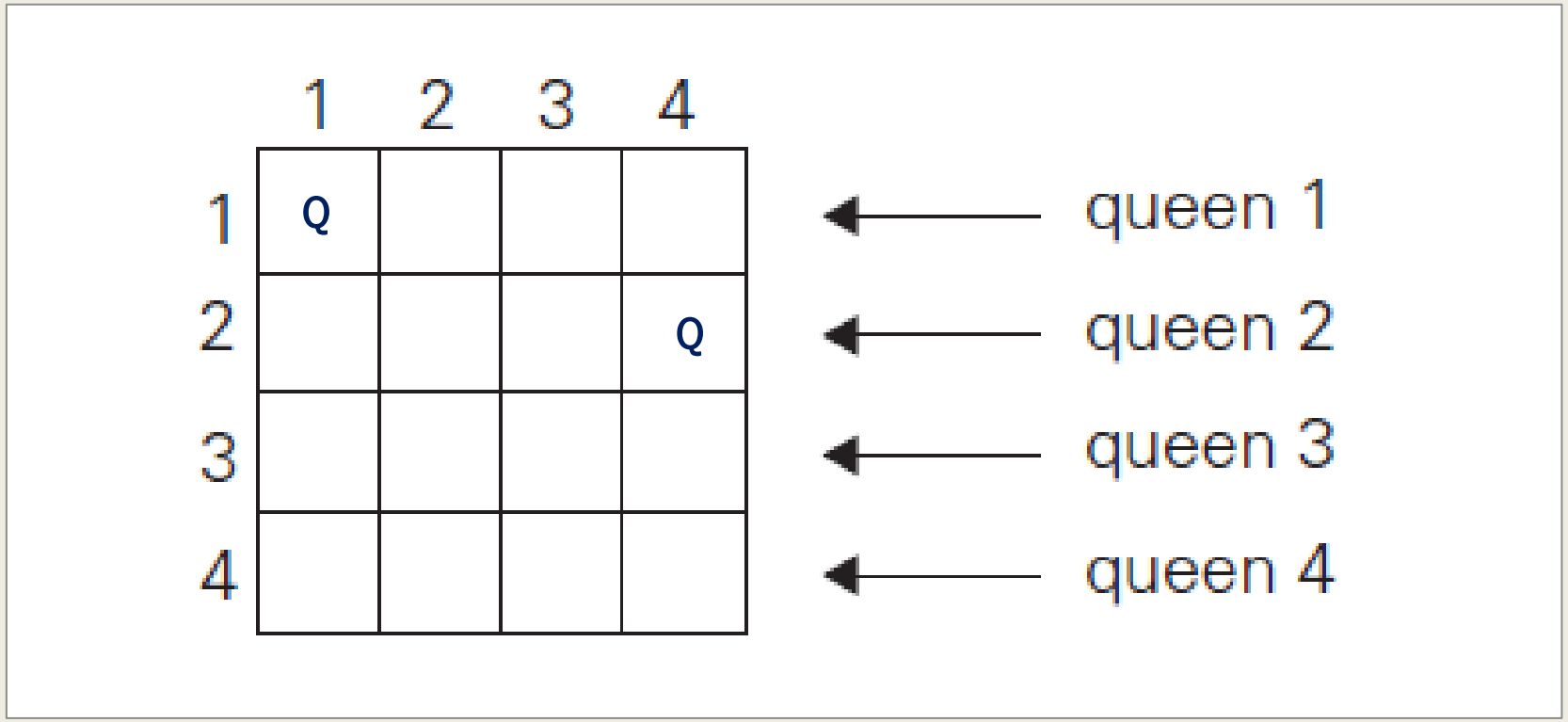
Backtracking – N Queens Problem

- When n=4?



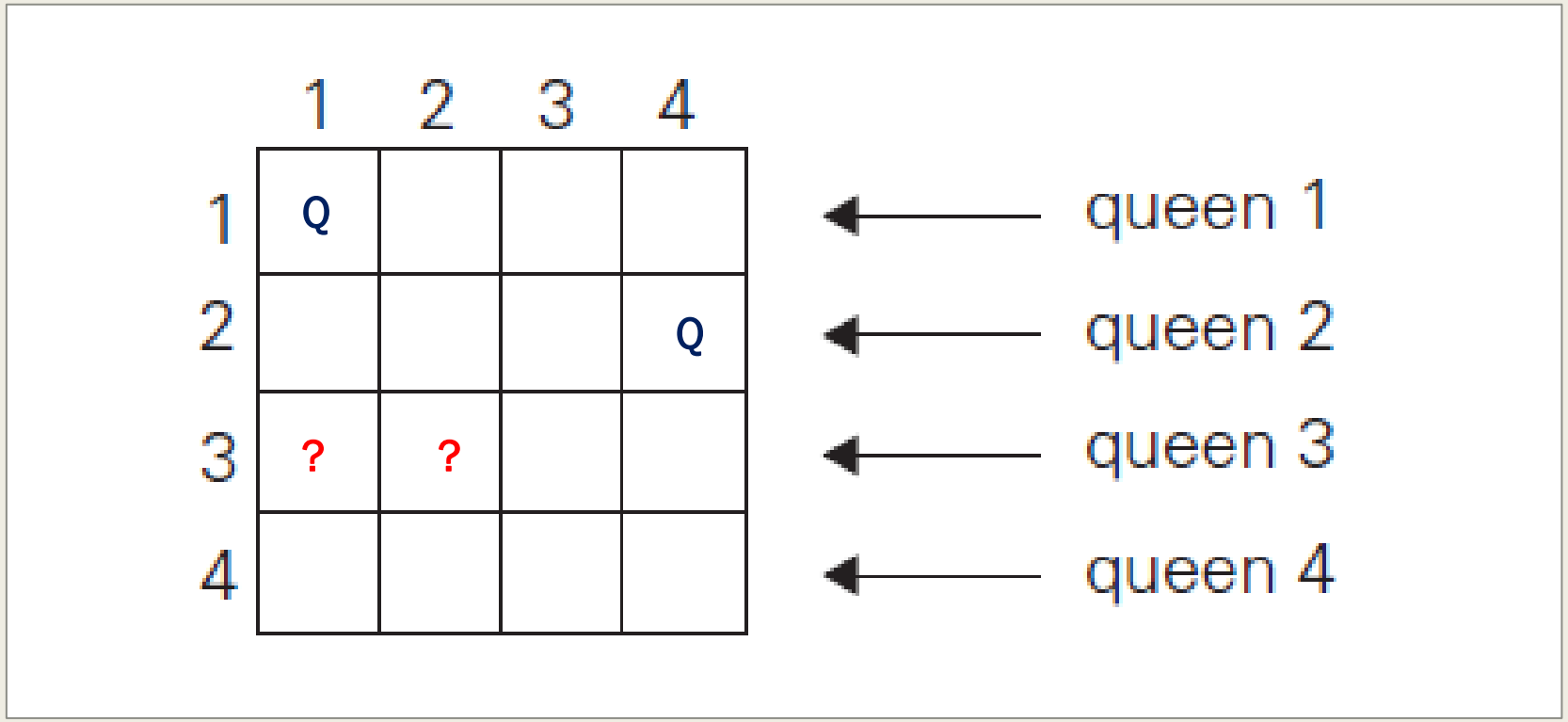
Backtracking – N Queens Problem

- When n=4?



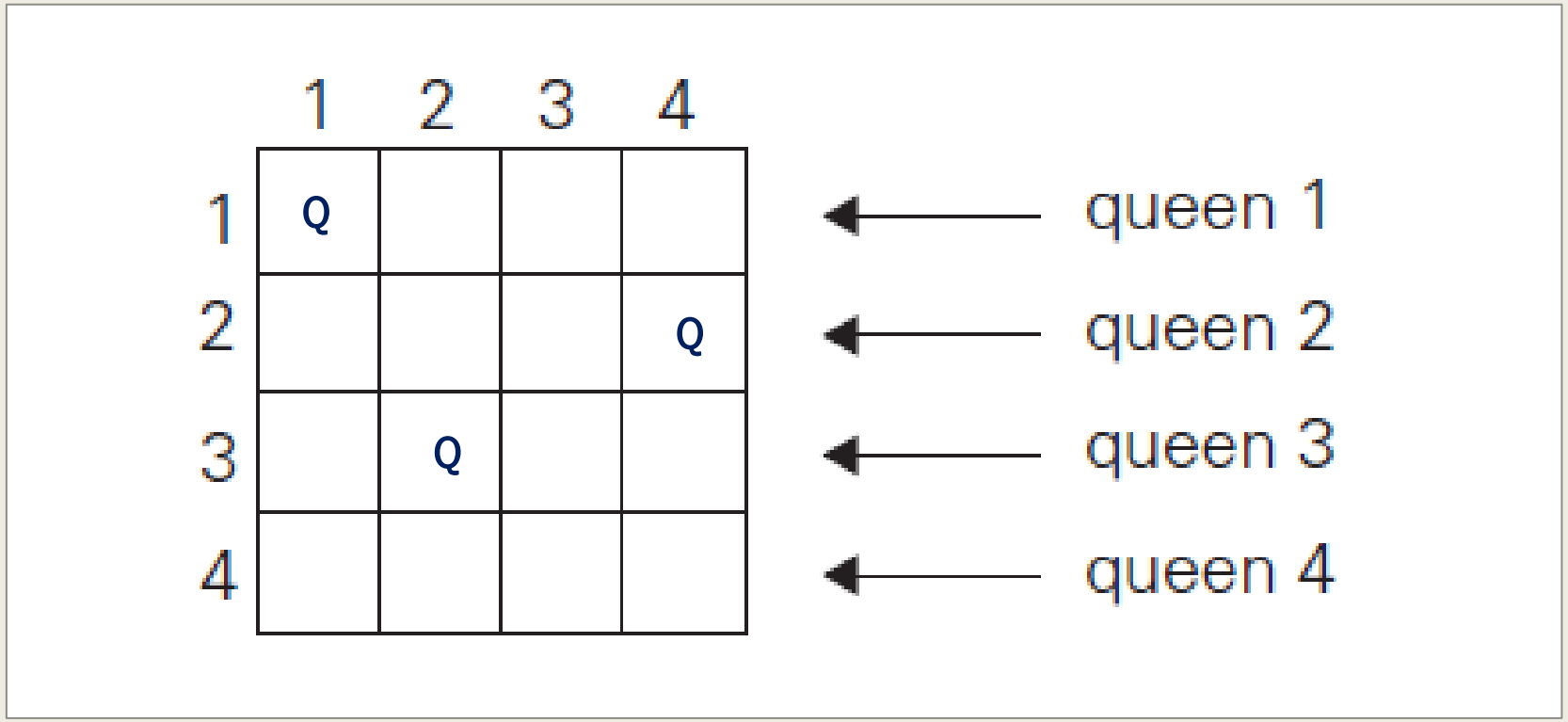
Backtracking – N Queens Problem

- When n=4?



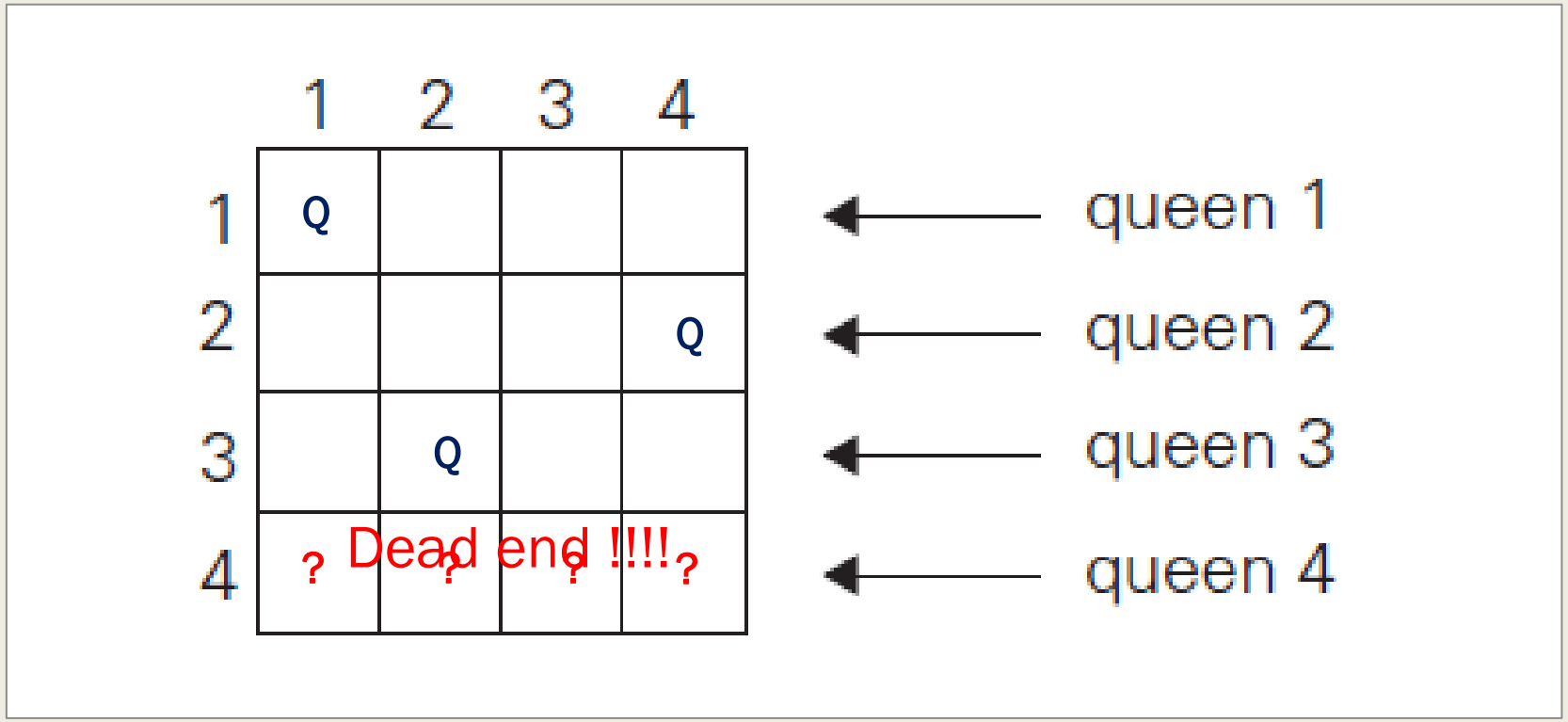
Backtracking – N Queens Problem

- When n=4?



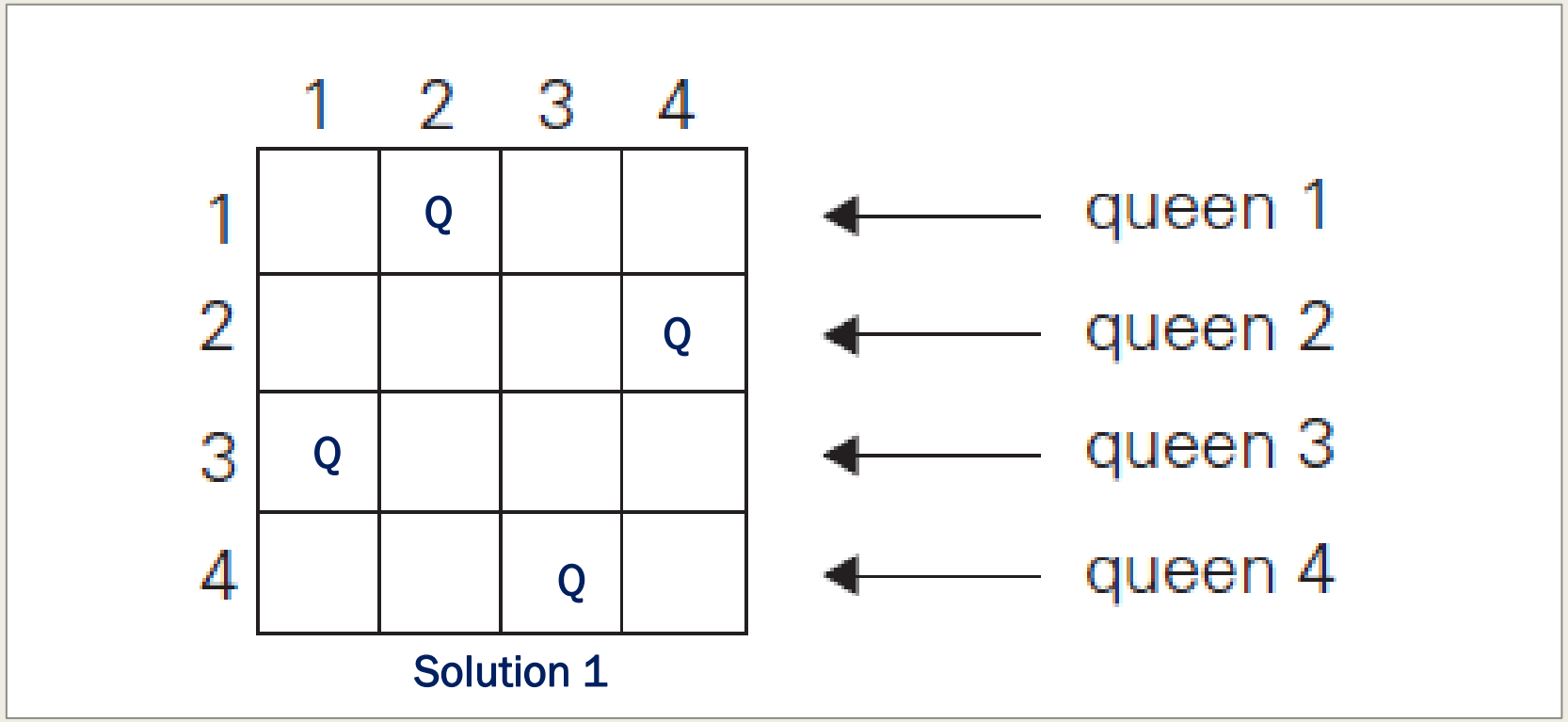
Backtracking – N Queens Problem

- When n=4?



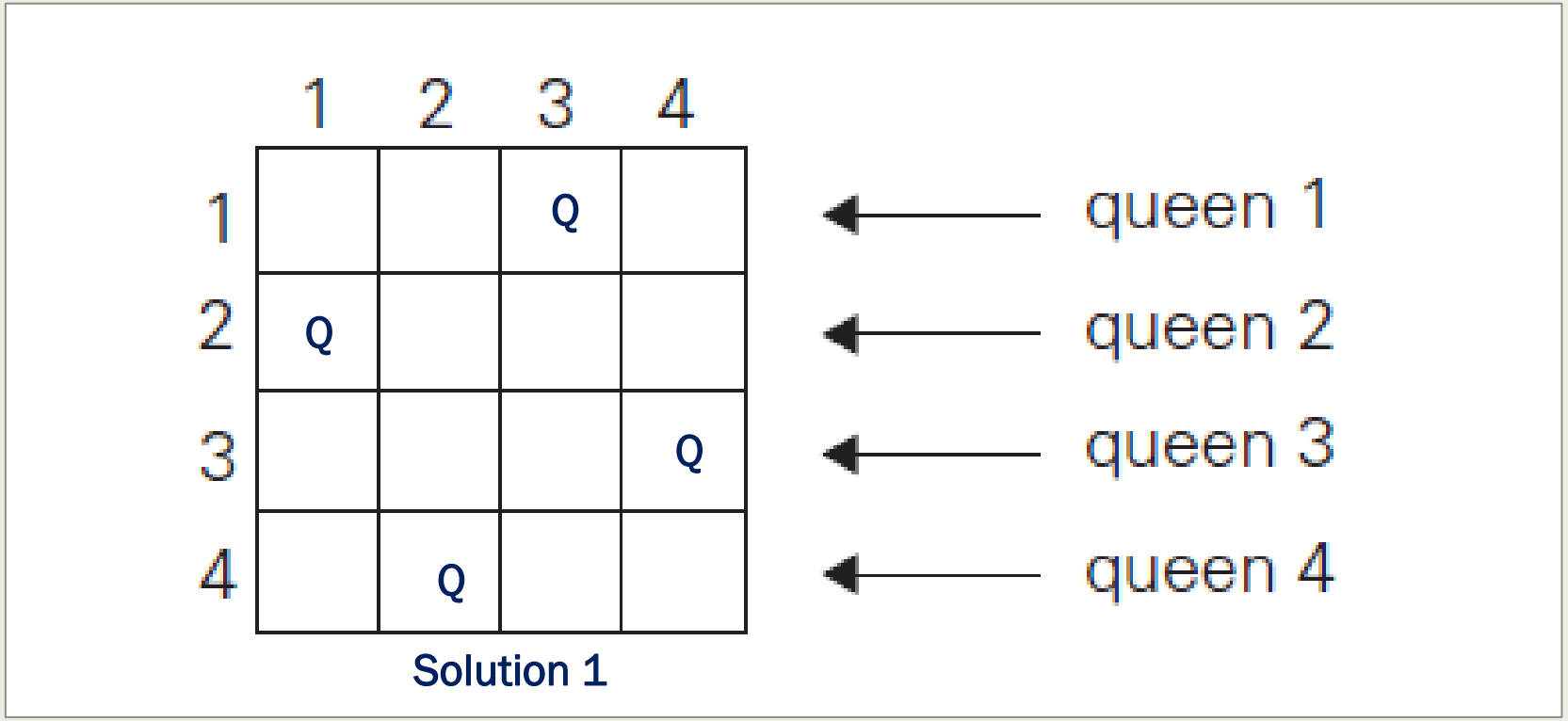
Backtracking – N Queens Problem

- When n=4?

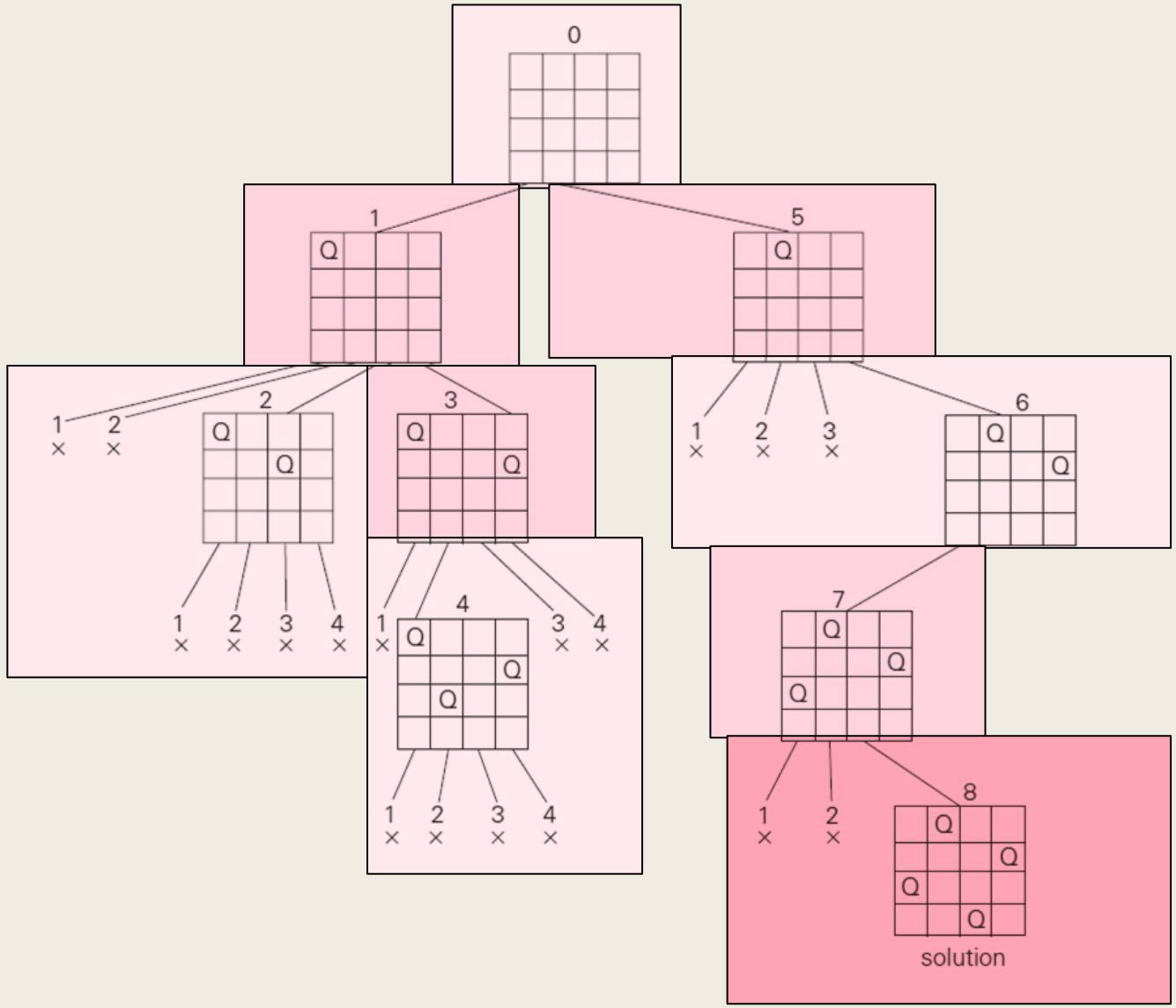


Backtracking – N Queens Problem

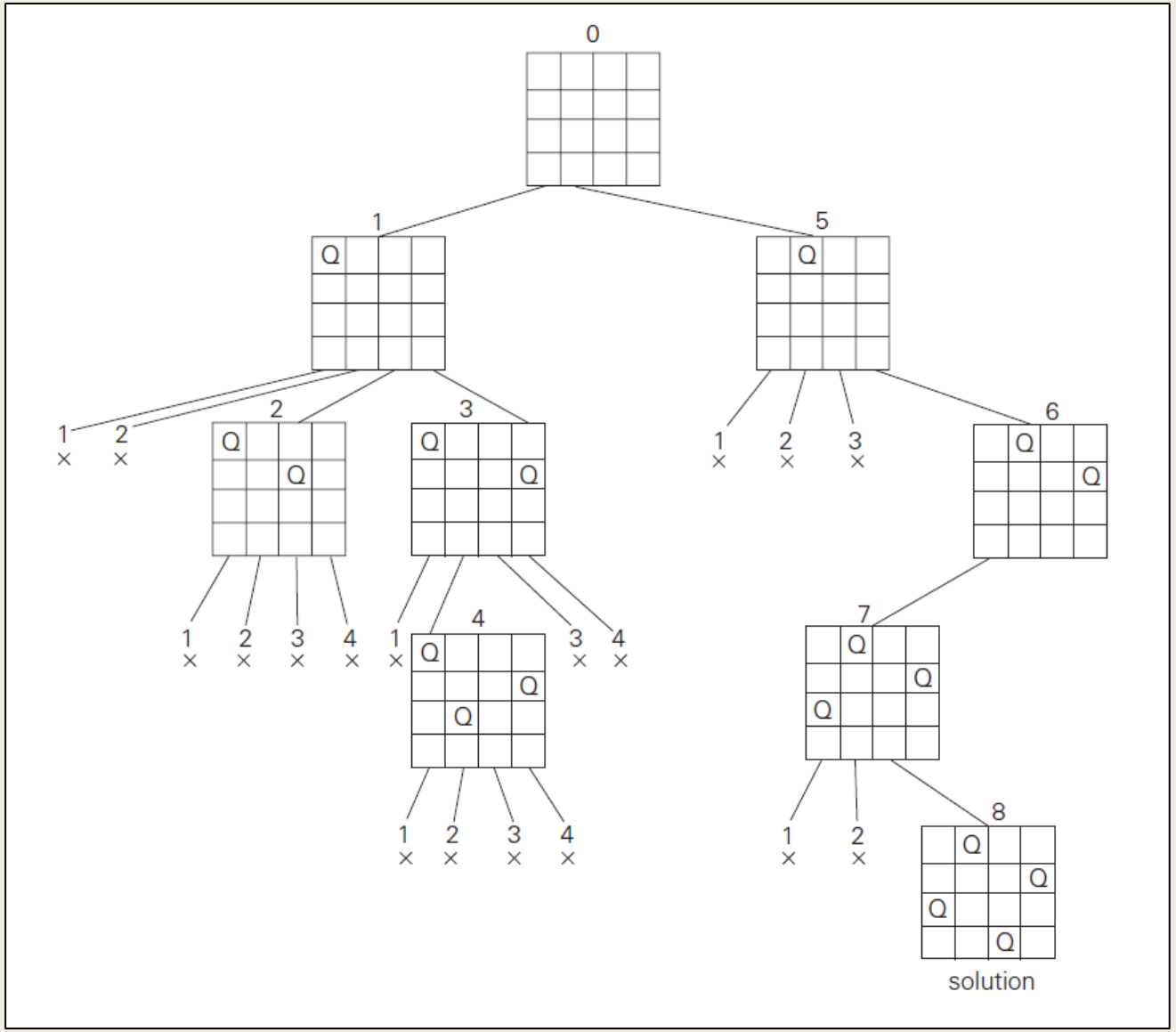
- When n=4? Alternate solution ??



Backtracking – N Queens Problem



Backtracking – N Queens Problem



Design and Analysis of Algorithms

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Backtracking – Sum of subsets

- Given set **A = {a_1, . . . , a_n}** of **n** positive integers, find **subset/s** whose sum is equal to a given positive integer **d**.
- **Example**:
  - **A = {1, 2, 5, 6, 8}** and **d = 9**,
  - there are two solutions:
    - {1, 2, 6}
    - {1, 8}.
  - **B={11, 13, 24, 7} and d= 31**
  - **There are two solutions**
    - {11,13,7}
    - {24, 7}

Backtracking – Sum of subsets

Solution approach

- Sort the set's elements in increasing order.

$$a_1 < a_2 < \ldots < a_n$$

- The **state space tree** can be constructed as **binary tree**
- The **root** of the tree represents the **starting point**, with no decisions about the given elements made as yet
- Its **left** and **right** children represent, respectively, **inclusion** and **exclusion** of **a1** in a set
- Similarly, going to the **left** from a node of the first level corresponds to **inclusion** of **a2** while going to the **right** corresponds to its **exclusion**, and so on

Backtracking – Sum of subsets

- Thus, a path from the root to a node on the ***i*th** level of the tree indicates which of the first **i** numbers have been included in the subsets represented by that node.
- We record the value of **s**, the sum of these numbers, in the node.
  - If **s** is equal to **d**, we have a **solution** to the problem.
- We can either **report** this result and **stop or**, if **all** the **solutions** need to be found, then, continue by **backtracking** to the node's parent.
- If ***s*** is not equal to ***d***, we can terminate the node as **nonpromising** if either of the following two inequalities holds:

$$s + a_{i+1} > d \qquad \text{(sum } \mathbf{s} \text{ is too } \mathbf{large}\text{)}$$

$$s + \sum_{j=1+1}^{n} a_j < d \qquad \text{(sum } \mathbf{s} \text{ is too } \mathbf{small}\text{)}$$

Backtracking – Sum of subsets

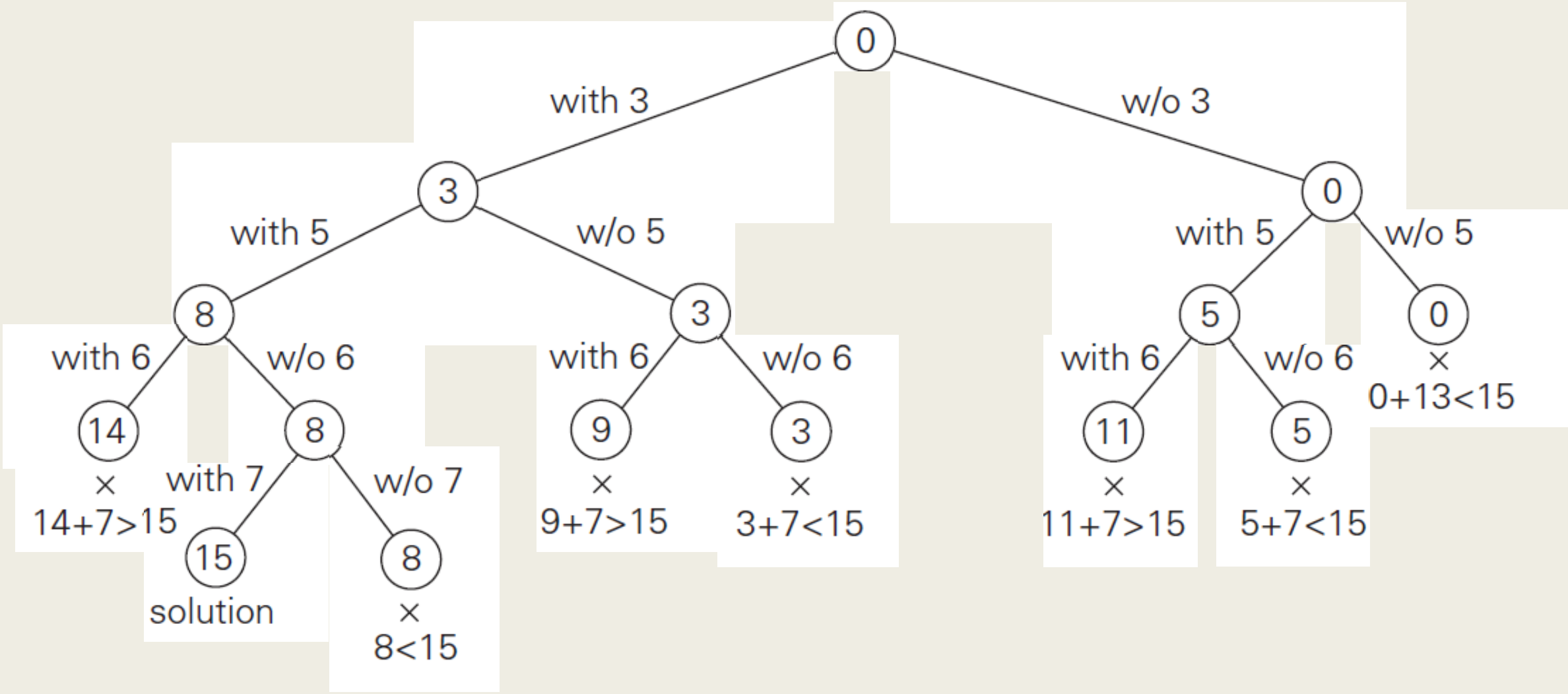
Apply backtracking to solve the following instance of the subset sum problem: **A = {3, 5, 6, 7}** and **d = 15.**

- *Draw the complete state space tree*

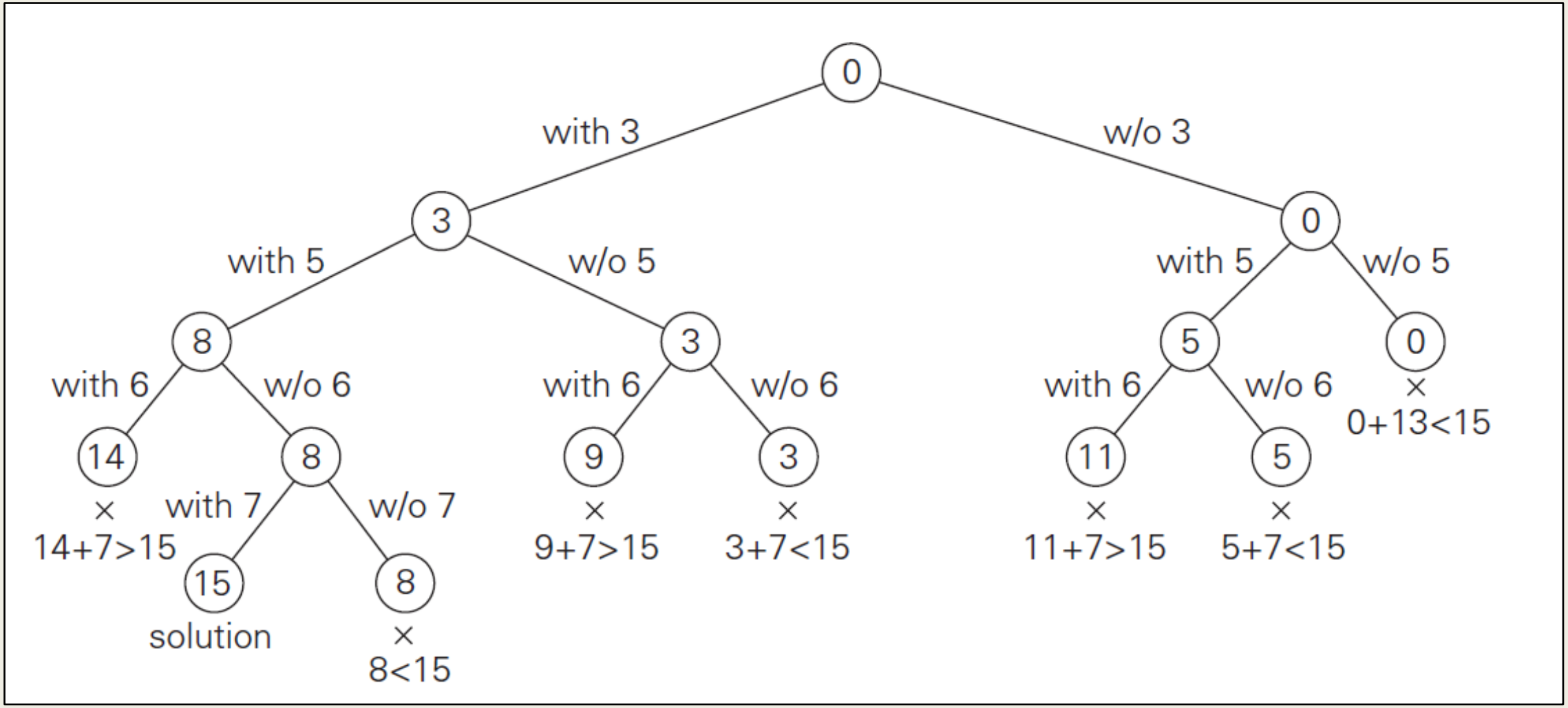
Solution

- Arrange the elements in **ascending** order
- Compute the sum **Ts** of all the elements
- If **Ts** is less than **d**
  - *then no solution*
- Start drawing the state space tree

Backtracking – Sum of subsets



Backtracking – Sum of subsets



Backtracking – Sum of subsets

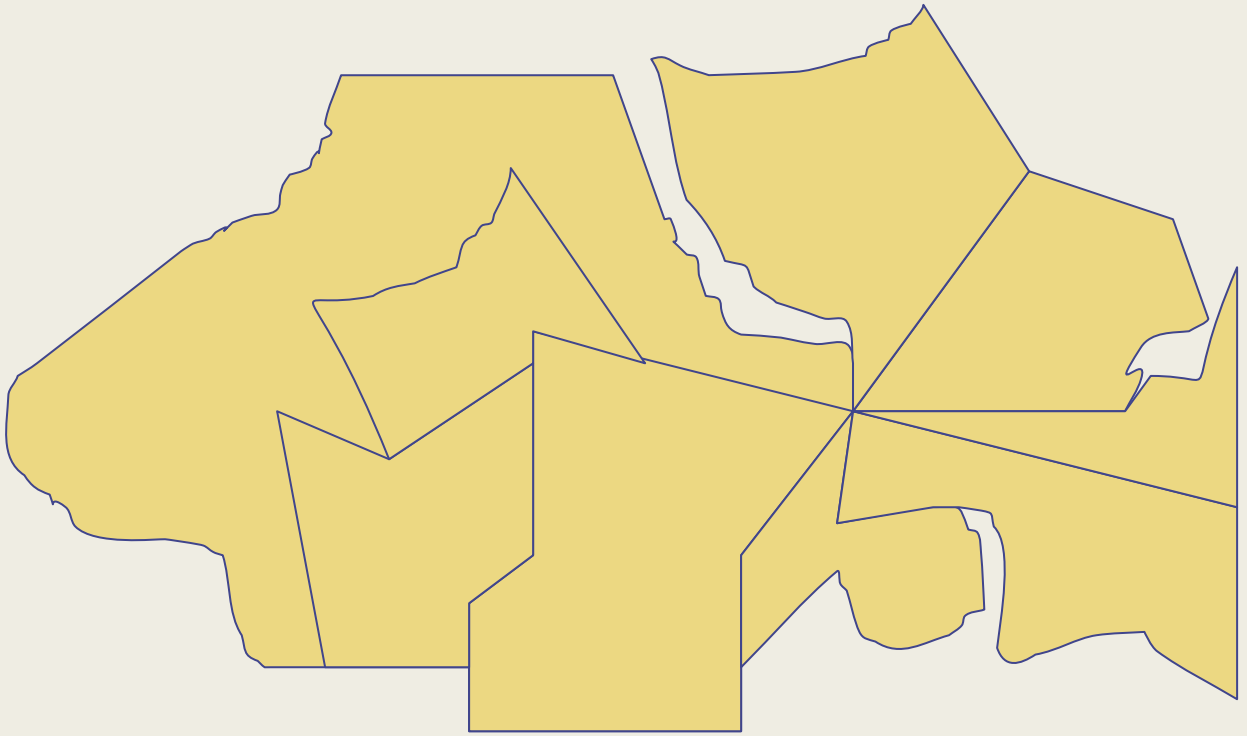
- Apply backtracking to solve the following instance of the subset sum problem: **A = {1, 3, 4, 5}** and **d = 11**.
- Draw the complete state space tree

Design and Analysis of Algorithms

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Backtracking – Graph Coloring

- Consider this fictional continent

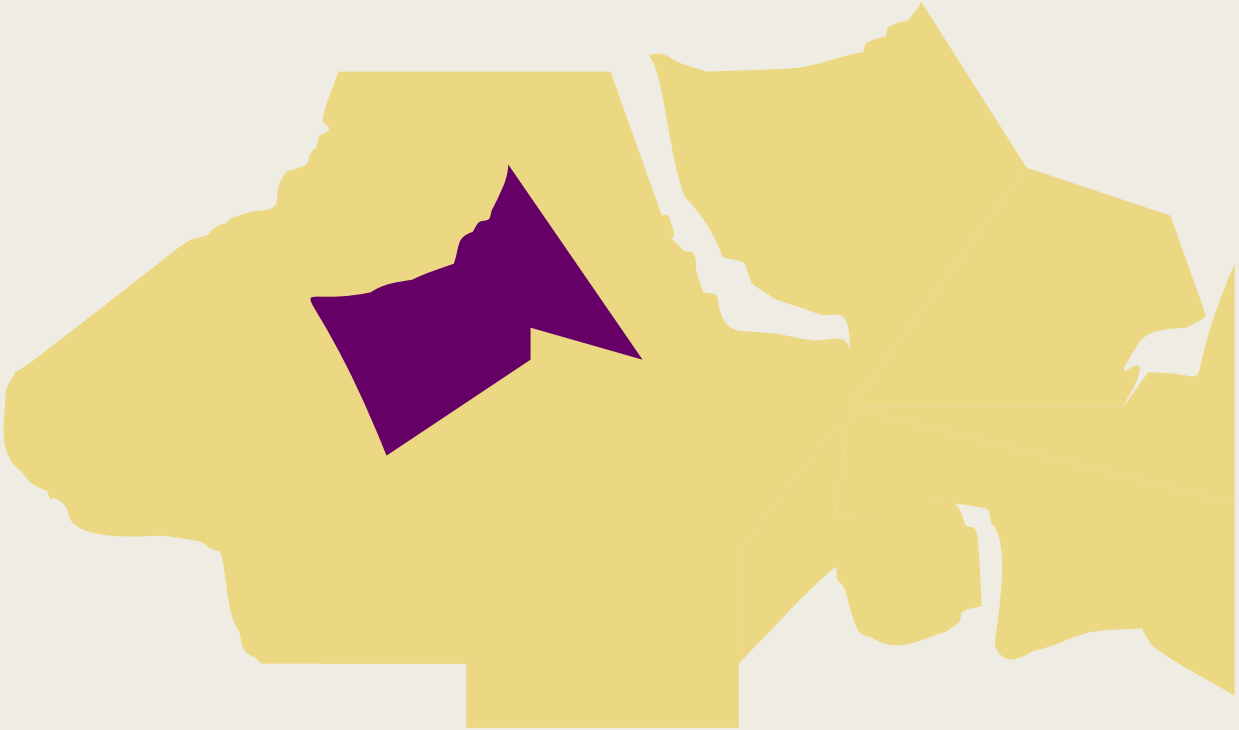


Backtracking – Graph Coloring

- Lets remove all borders but you may still want to see all the countries.
- 1 color is sufficient? No !!!! Its insufficient.

Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.



Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.

Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.

Backtracking – Graph Coloring

- So add another color.
- Now try to fill in every country with one of the two colors.

Backtracking – Graph Coloring

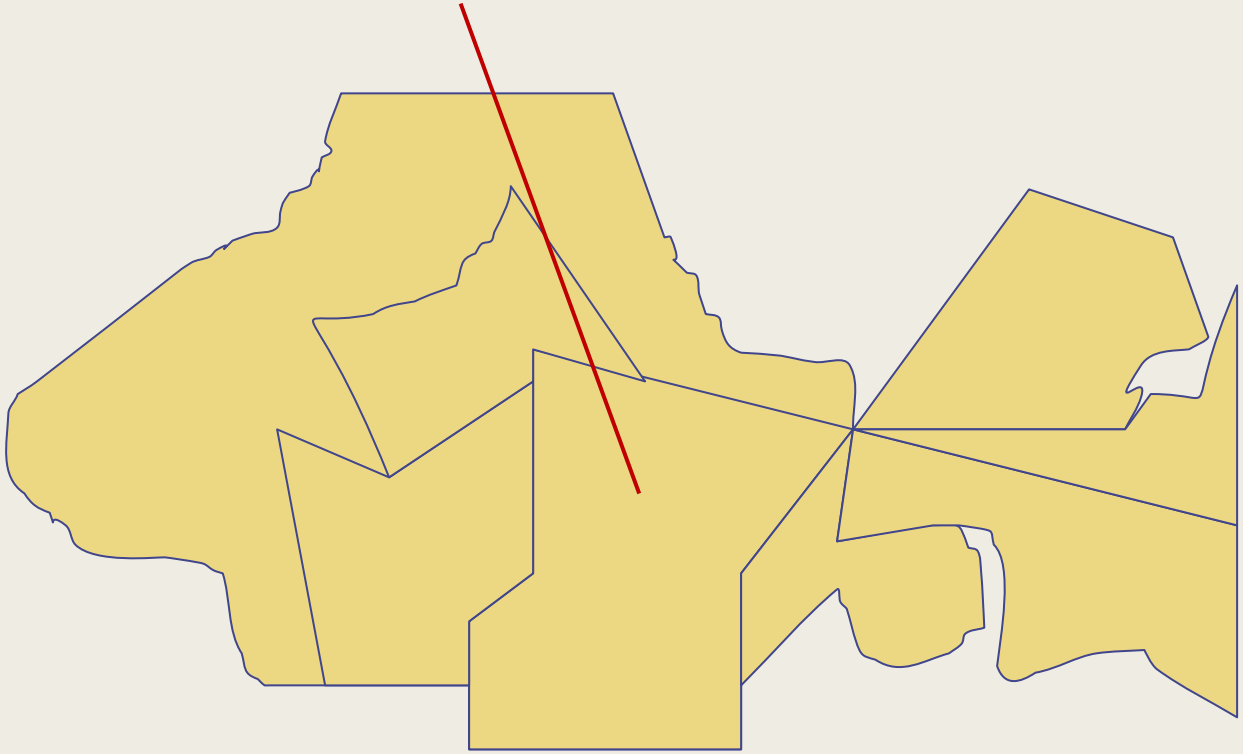
- **PROBLEM !!!!!**
- Two adjacent countries forced to have same color. Border unseen.

Backtracking – Graph Coloring

- So add another color

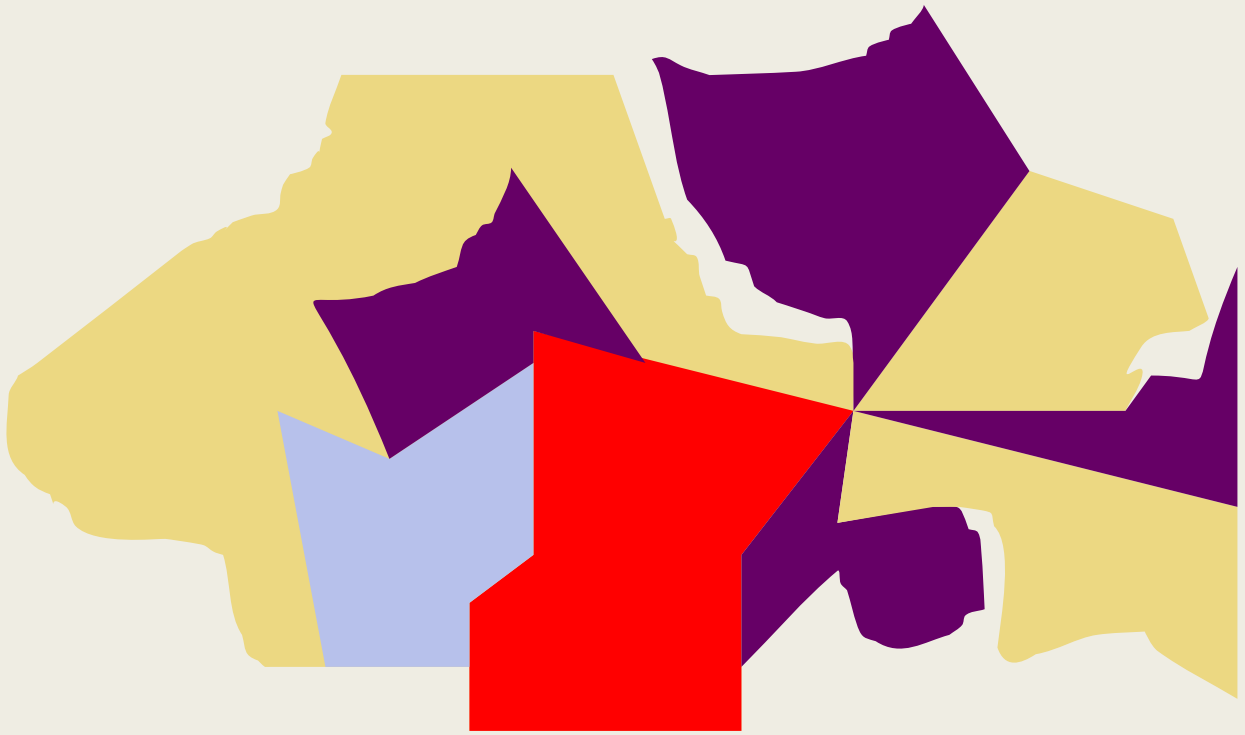
Backtracking – Graph Coloring

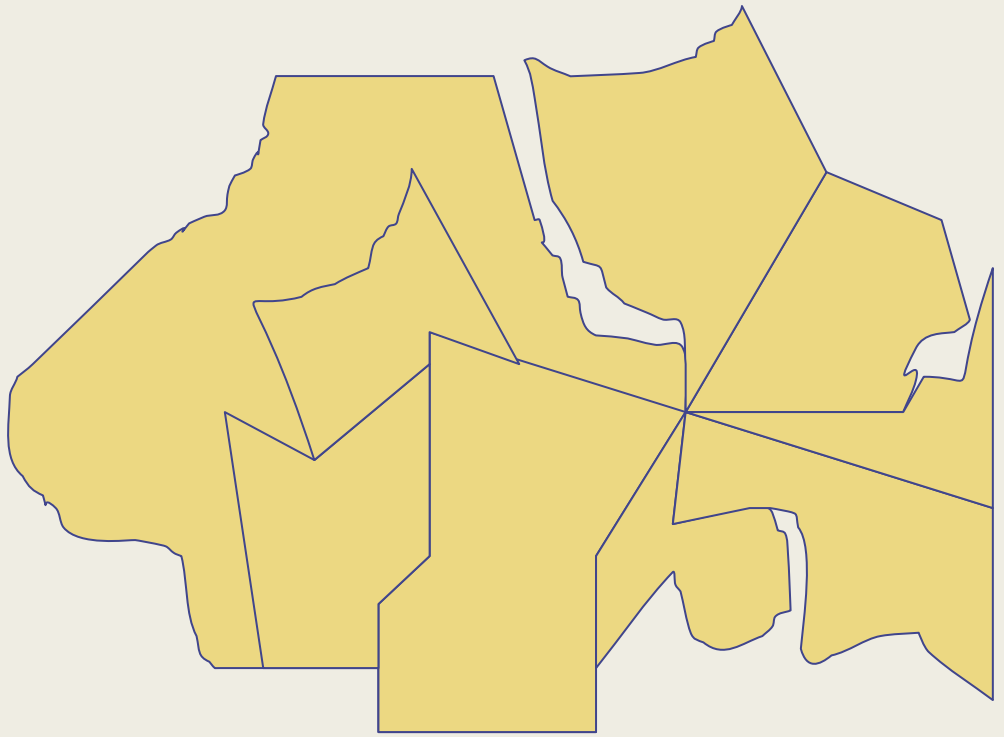
- **Insufficient.**
- **Need 4 colors because of this country.**



Backtracking – Graph Coloring

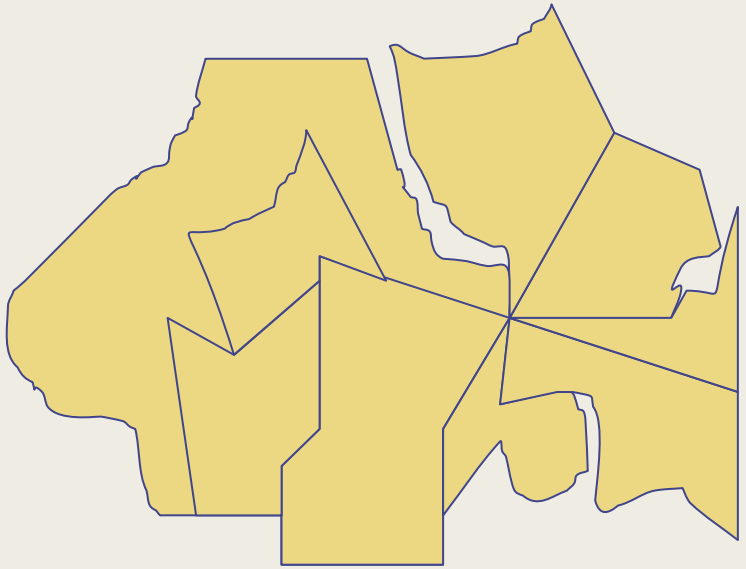
- Using 4 colors, the entire graph is colored and the borders are clearly seen



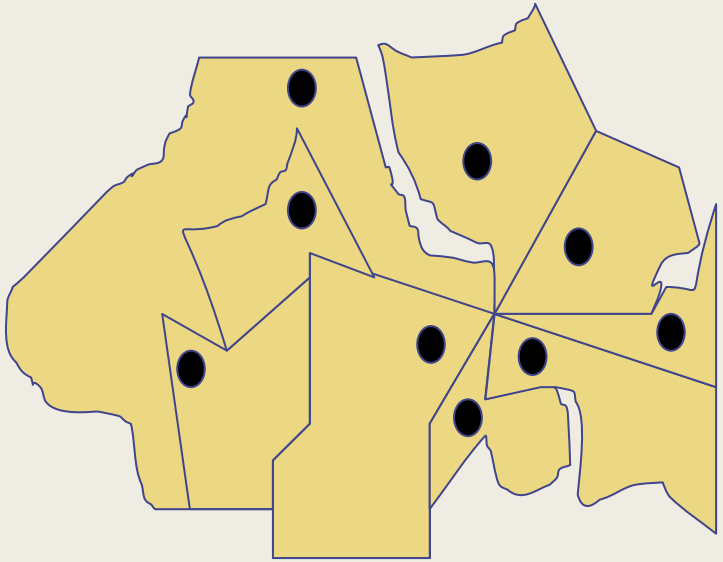


Backtracking – Graph Coloring

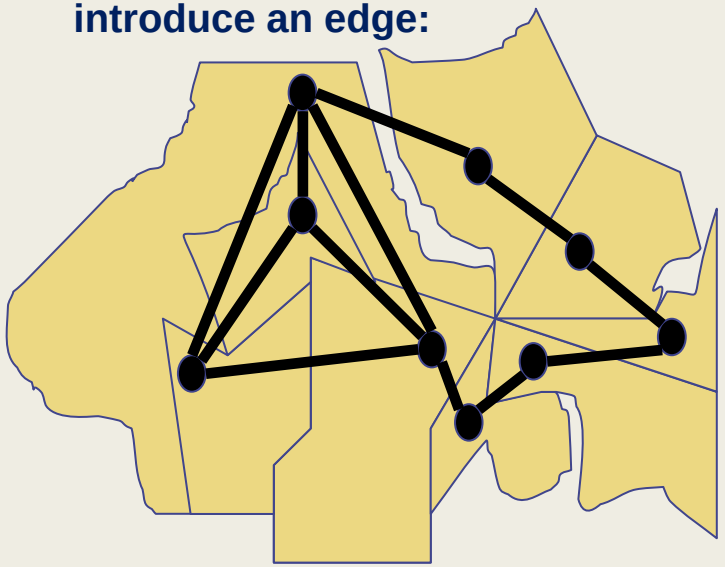
Region to be coloured



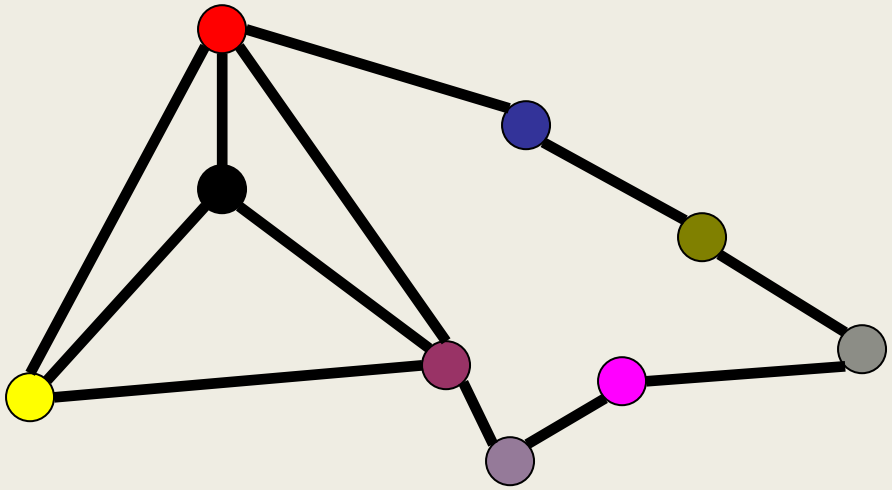
For each region introduce a vertex



For each pair of regions with a positive-length common border introduce an edge:

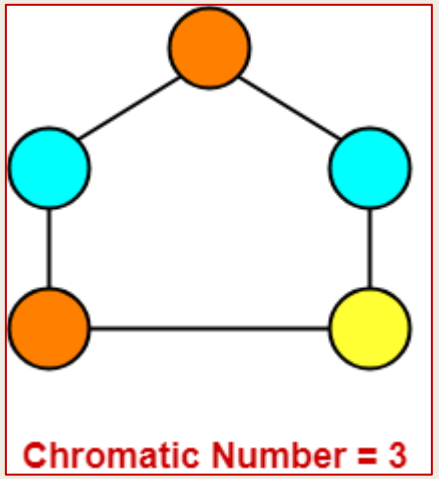


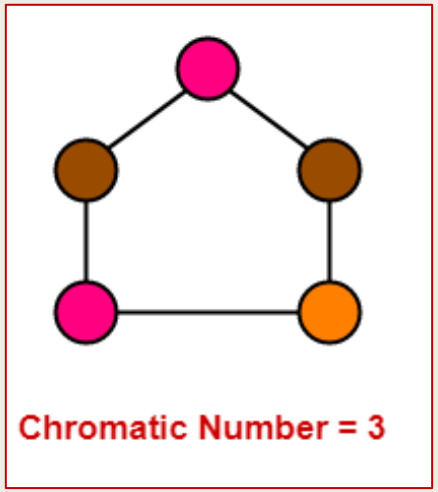
Coloring regions is equivalent to coloring vertices of dual graph.



Backtracking – Graph Coloring

- Given an **undirected graph** and m colours, the task is to find whether is it possible to **colour the vertices** of the graph using m colours with a constraint that **no two adjacent** vertices can have **same colour.**
- Minimum number of colours required to colour the graph with the said constraint is called **chromatic number**

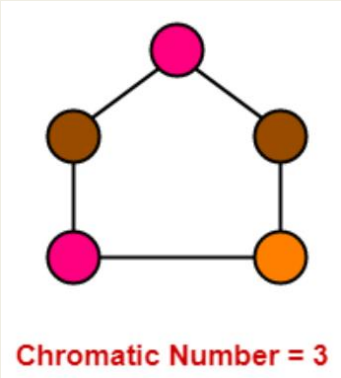


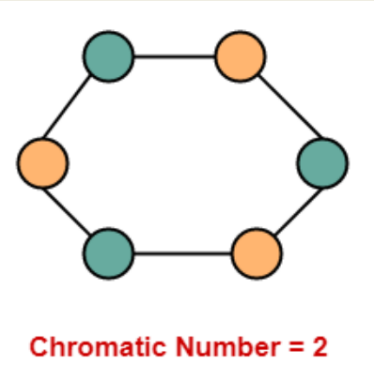


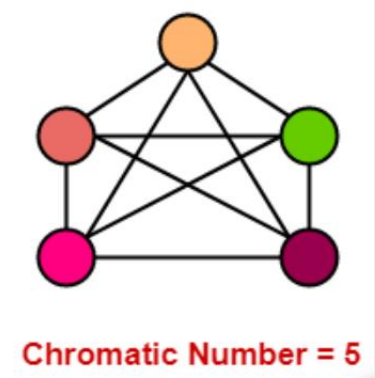
Backtracking – Graph Coloring

Some interesting facts about chromatic number

- In a cyclic graph
  - If number of vertices in cycle graph is even, then its chromatic number = 2.
  - If number of vertices in cycle graph is odd, then its chromatic number = 3.
- In a planar graph (a graph that can be drawn in a plane such that none of its edges cross each other.)
  - Chromatic number is ≤ 4
- In a complete graph (each vertex is connected with every other vertex)
  - Chromatic number = number of vertices in that complete graph



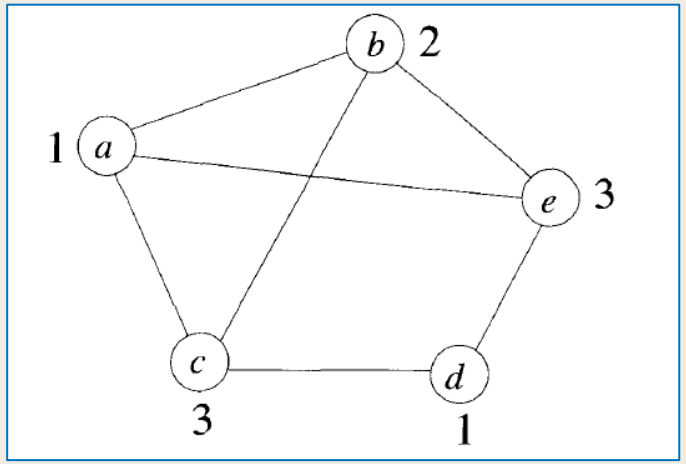


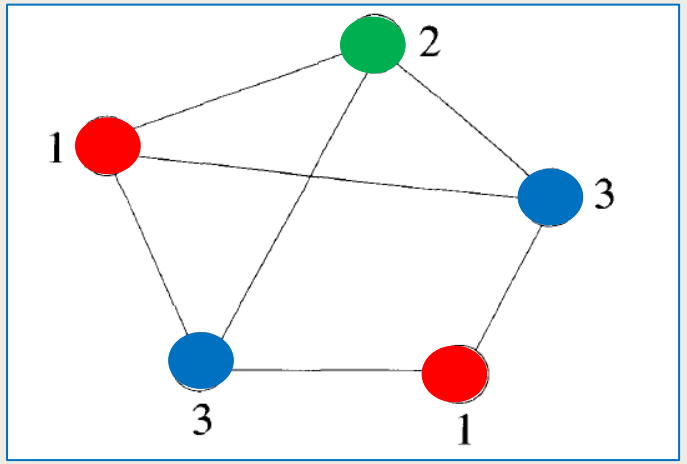


Backtracking – Graph Coloring

Problem statement

- Let **G** be a graph and **m** be a given positive integer.
  - Discover whether the nodes of **G** can be colored in such a way that no two adjacent nodes have the same color yet only **m** colors are used.
  - This is termed the **m-colorability** decision problem
  - The **m-colorability** optimization problem asks for the smallest integer **m** for which the graph **G** can be colored
  - This integer **m** is referred to as the **chromatic number** of the graph





Chromatic no. =3

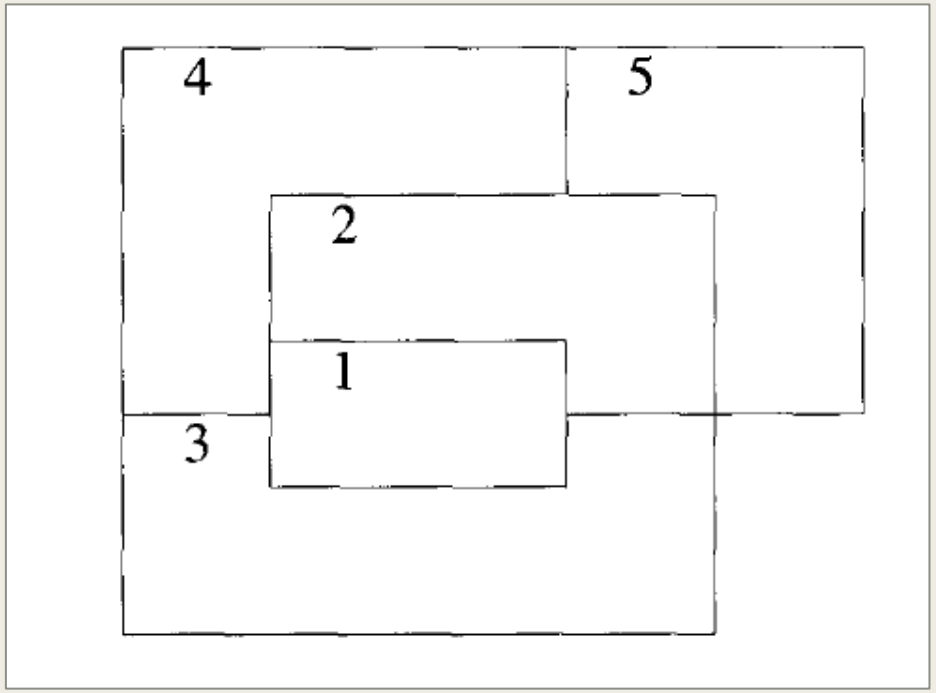
Backtracking – Graph Coloring

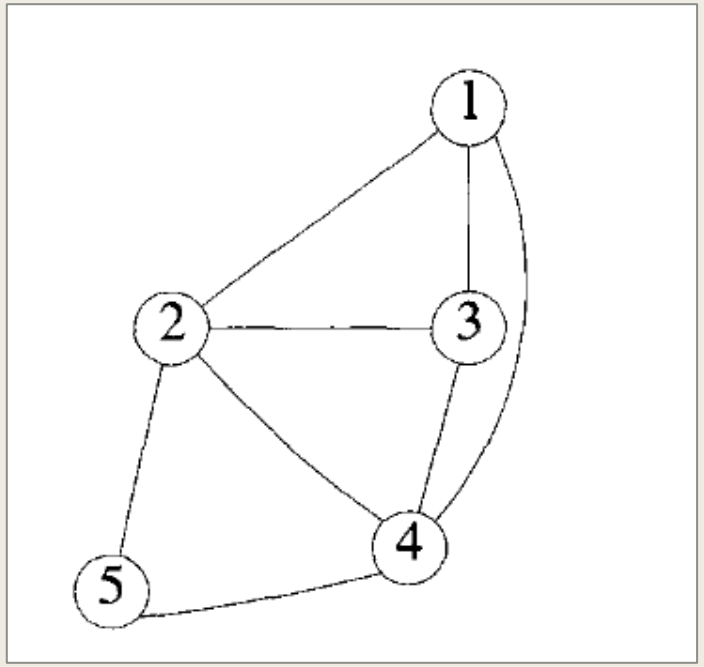
4 colour problem for planar graphs

- A graph is said to be planar iff it can be drawn in a plane in such away that no two edges cross each other
- A famous special case of the m colourability decision problem is the 4-color problem for planar graphs
- This problem asks the following question:
  - given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed
- This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph.
- Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge

Backtracking – Graph Coloring

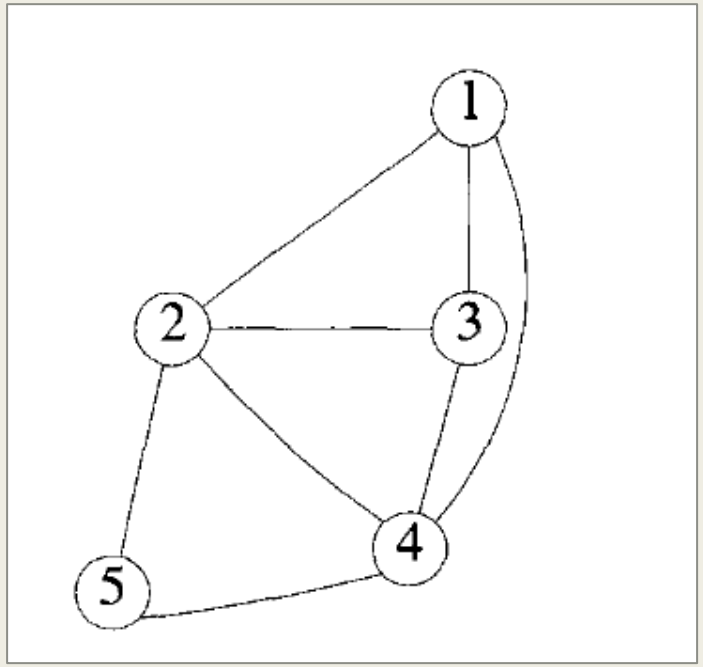
- Map and its corresponding graph representation



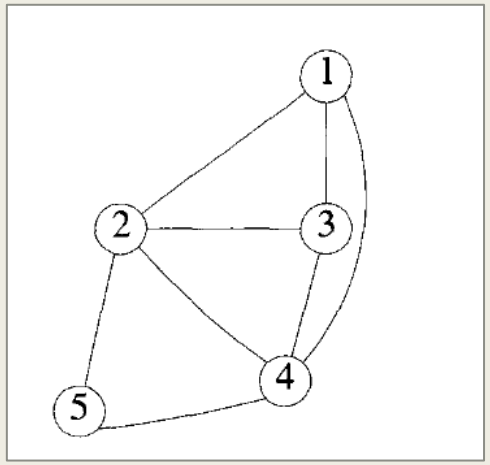


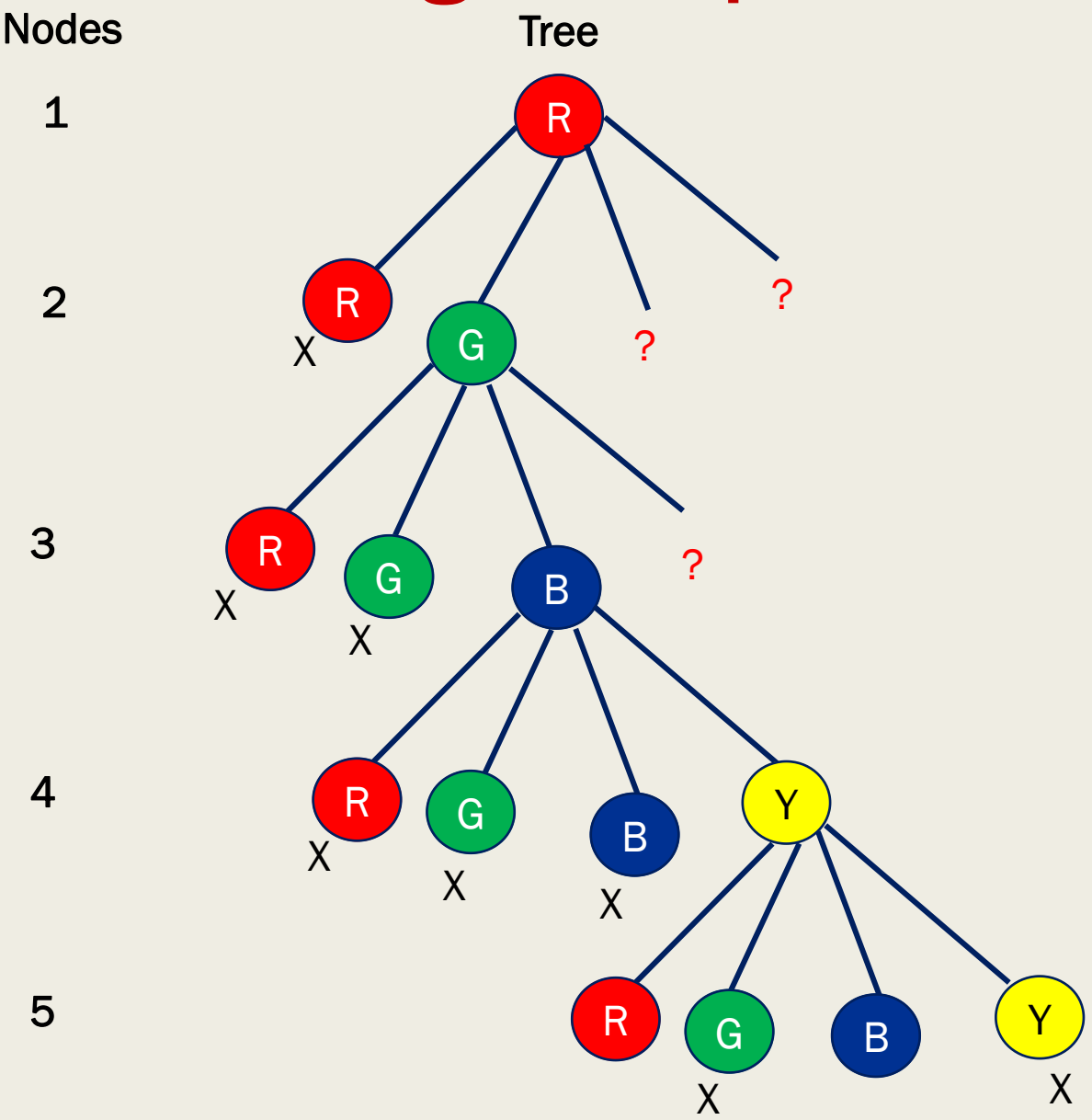
Backtracking – Graph Coloring

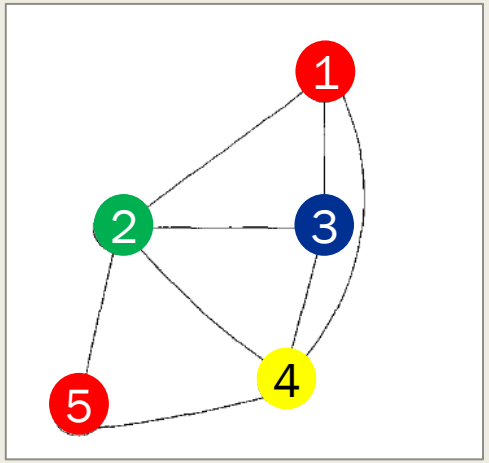
- Solve the following instance of graph colouring. Draw the state space tree
-

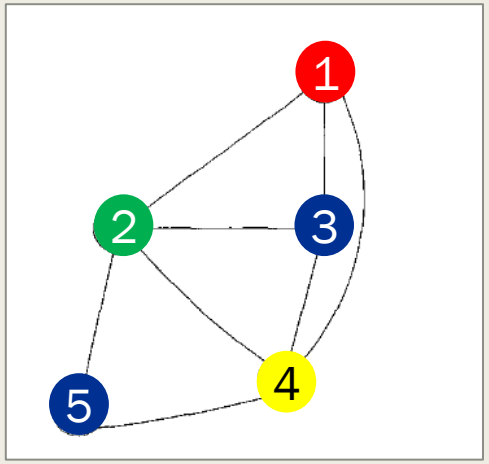


Backtracking – Graph Coloring







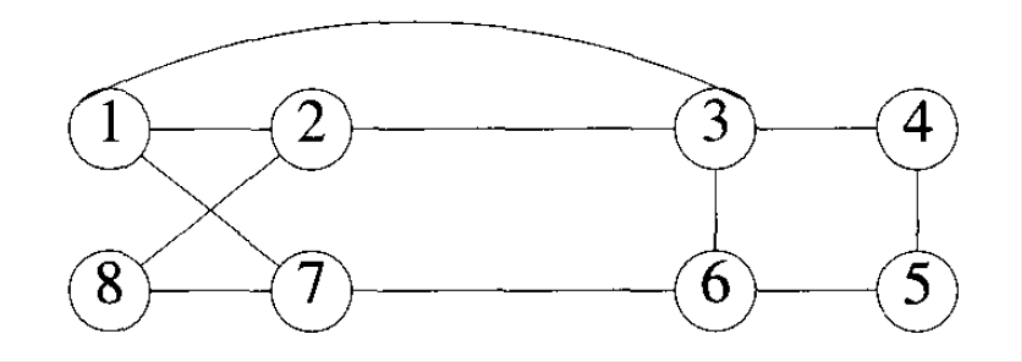


Design and Analysis of Algorithms

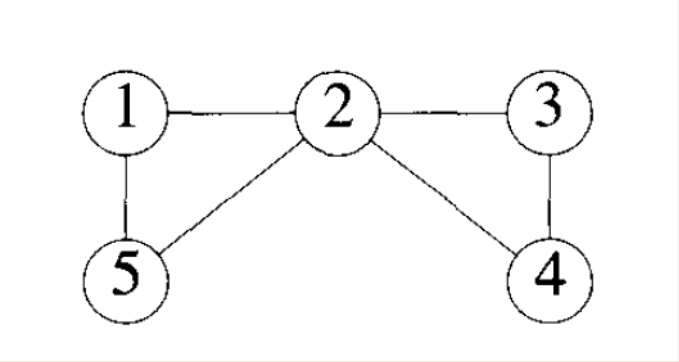
Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Backtracking – Hamiltonian Cycle

- Let **G = (V, E)** be a connected graph with n vertices
- A Hamiltonian cycle (Sir William Hamilton)is a round-trip path along **n** edges of **G** that visits every vertex once and returns to its starting position
- In. other words if a Hamiltonian cycle begins at some vertex $v_i \in G$ and the vertices of **G** are visited in the order $v_1, v_2\ v_3 \ldots,.\ v_{n+1}$, then the edges (v_i, v_{i+1}) are in **E**, $1 \leq i \leq n$ and v_i are distinct except for $v_1$and v_{n+1} which are equal



1, 2, 8, 7, 6, 5, 4, 3, 1



No Hamiltonian cycle !!!!

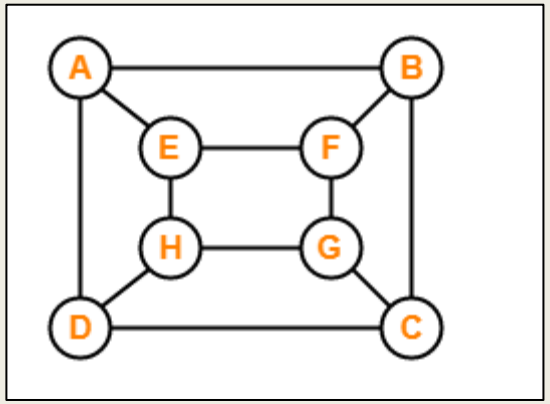
Backtracking – Hamiltonian Cycle

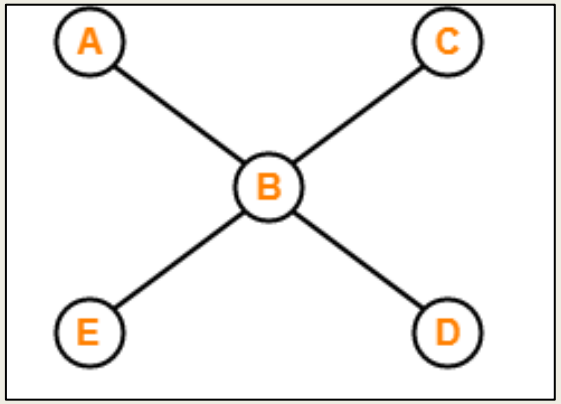
Solution Approach

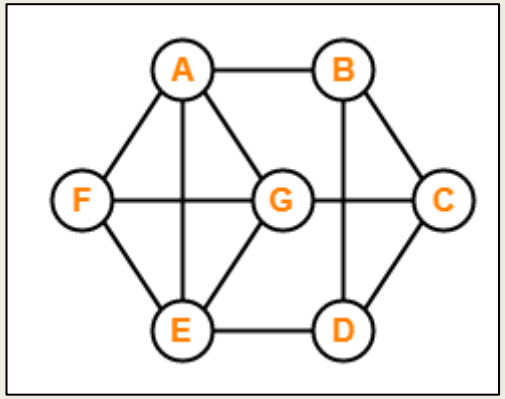
- Let $(x_1, x_2 \ldots \ldots x_n)$ be the solution vector so that x_i represents the **ith** visited vertex in the proposed cycle
- We need to determine how to compute the set of possible vertices for x_k , **if** $x_1, x_2 \ldots \ldots x_{k-1}$ have already been chosen
- If **k=1** then x_1 can be any of the **n** vertices
- To avoid printing the same cycles n times, we require that $x_1 = 1$
- If $1 < k < n$, then x_k can be any vertex **v** that is distinct from $x_1, x_2 \ldots \ldots x_{k-1}$ and v is connected by an edge to x_{k-1}
- The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1

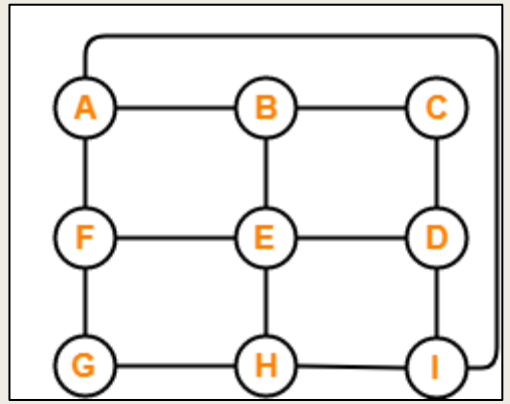
Backtracking – Hamiltonian Cycle

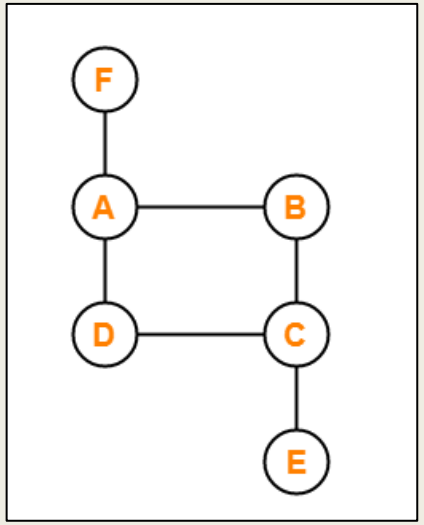
Which of the following are Hamiltonian graphs ?







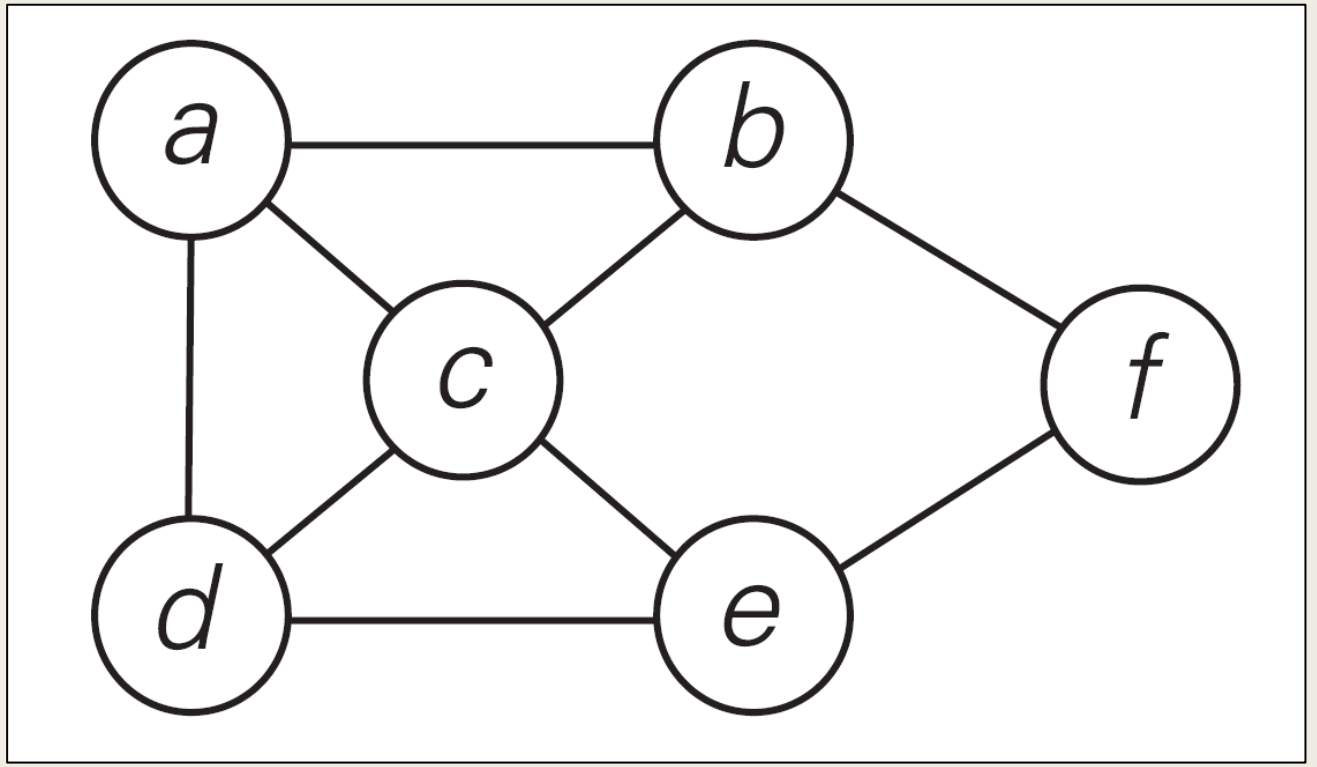




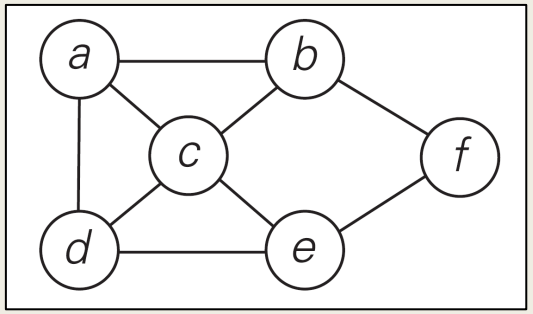
Backtracking – Hamiltonian Cycle

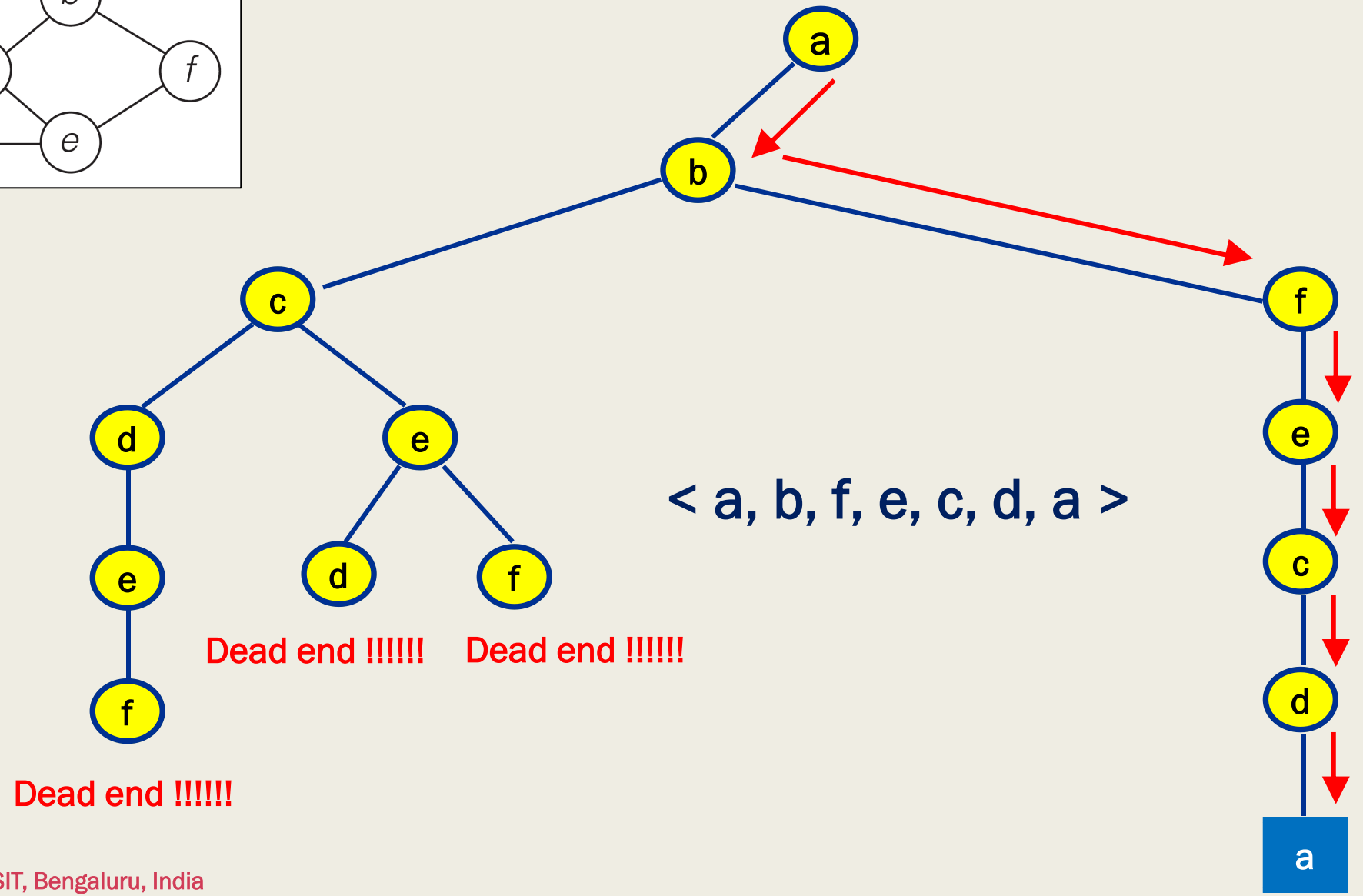
State Space Tree Construction

- Solve the following instance of Hamiltonian cycle and draw the state space tree



Backtracking – Hamiltonian Cycle





BACKTRACKING

Branch and Bound

- The idea of backtracking is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution
- To handle optimization problems, the same concept can be strengthened
- An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment etc.) subjected to constraints
- Recall
  - Feasible solution
  - Optimal Solution

Branch and Bound

- Compared to backtracking, branch-and-bound requires two additional items:
  - a way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
  - the value of the best solution seen so far
- If this information is available, we can compare a node's bound value with the value of the best solution seen so far.
  - If the bound value is not better than the value of the best solution seen so far
  - i.e., not smaller for a minimization problem and not larger for a maximization problem
  - the node is nonpromising and can be terminated (some people say the branch is "pruned").

Branch and Bound

- A search path is terminated at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
- The value of the node's bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—
  - in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Branch and Bound – Assignment Problem

Problem Statement

- Assign n people to n jobs so that the total cost of the assignment is as small as possible
- Assignment problem is specified by an $n \times n$ cost matrix C
  - Select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

Branch and Bound – Assignment Problem

- How to find a lower bound on the cost of an optimal selection without actually solving the problem
- It is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows
- For the instance here, this sum is 2 + 3+ 1+ 4 = 10
- It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix);
- it is just a lower bound on the cost of any legitimate selection

Branch and Bound – Assignment Problem

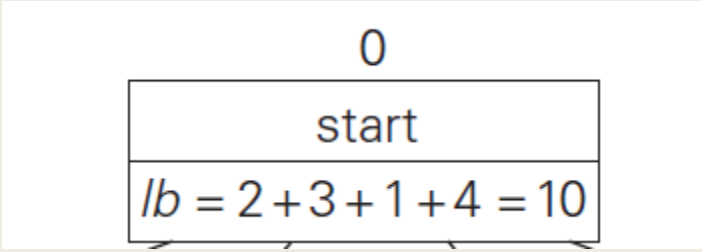
- Solve the following instance of assignment problem an obtain the state space tree

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

- First compute the lower bound by finding the sum of the smallest elements in each row

LB= 2+3+1+4=10

Branch and Bound – Assignment Problem

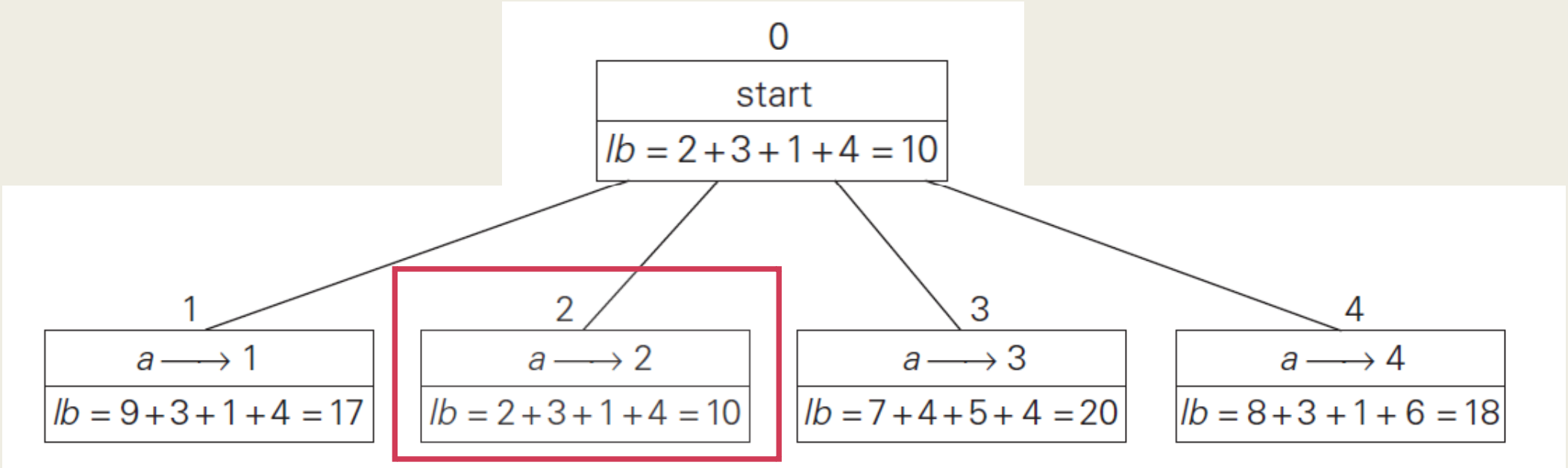


- Consider a job for person a
  - a->1 lb=9+3+1+4=17
  - a->2 lb=2+3+1+4=10
  - a->3 lb=7+4+5+4=20
  - a->4 lb=8+3+1+6=18

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

- Since a->2, lb=10 is more promising node, we branch from there !!!!

Branch and Bound – Assignment Problem



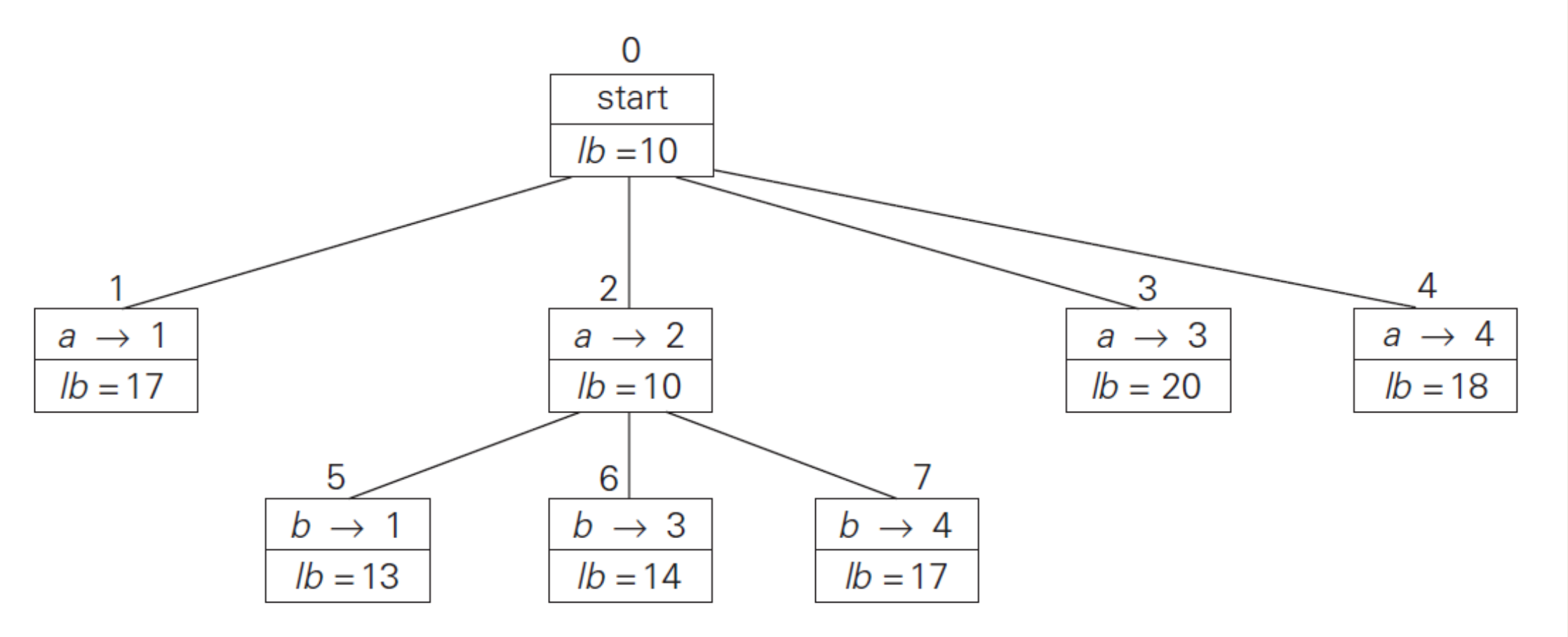
Branch and Bound – Assignment Problem

- Consider a job for person b
  - b->1 lb=2+ (6+1+4)=13
  - b->3 lb=2+ (3+5+4)=14
  - b->4 lb=2+(7+1+7)=17

- Since b->1, lb=13 is more promising node, we branch from there !!!!

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

Branch and Bound – Assignment Problem



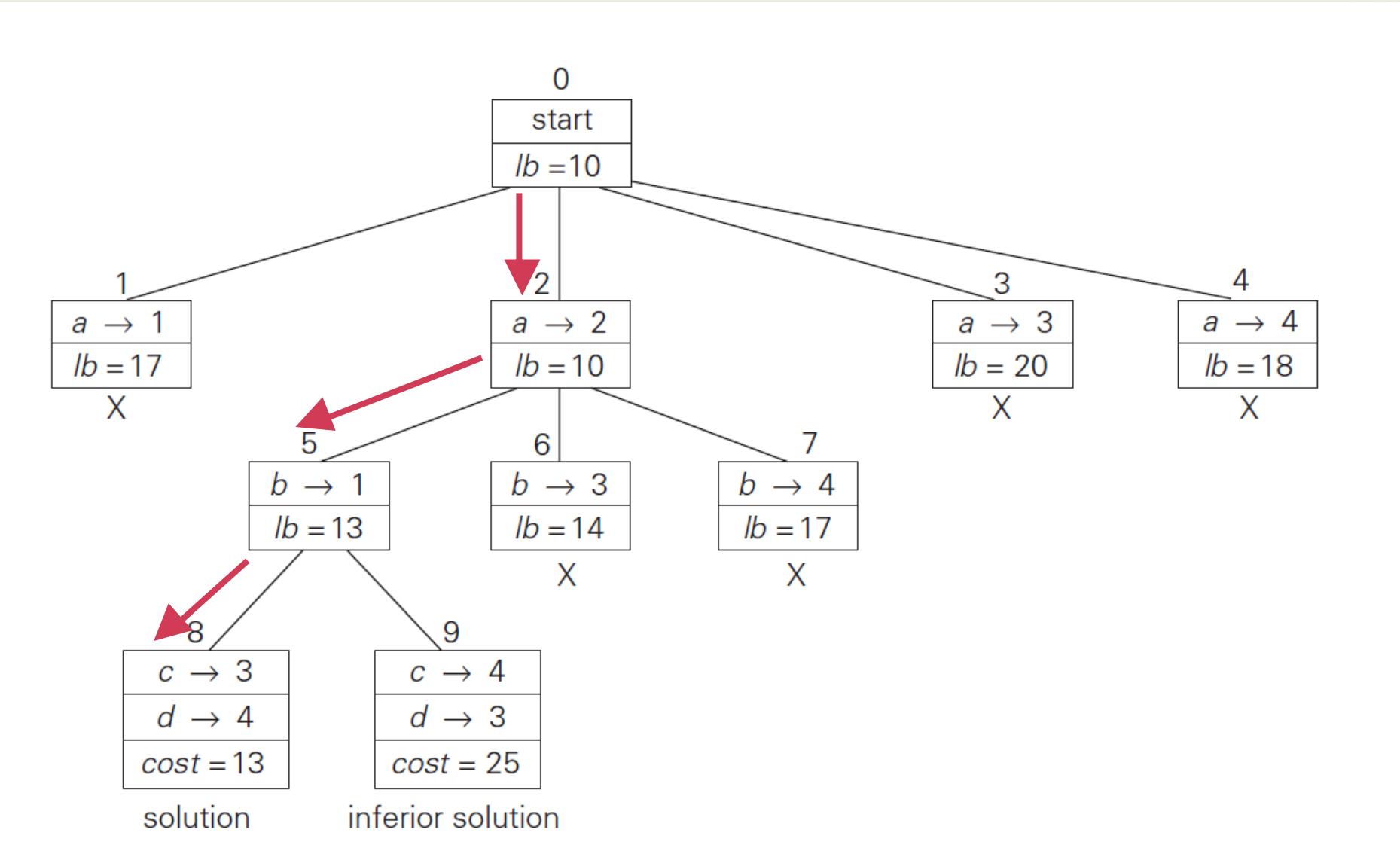
Branch and Bound – Assignment Problem

- Consider job for persons **c** and **d**
  - c->3, d->4 lb=2+6+(1+4)=13
  - c->4, d->3 lb=2+6+(8+9)=25
  - c->3, d->4 is more promising
- The final assignment of jobs is as follows

a->2 b->1 c->3 d->4

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

Branch and Bound – Assignment Problem



Design and Analysis of Algorithms

Dr. Bhavanishankar K
Asst. Prof. Dept. of CSE
RNSIT, Bengaluru, India

Branch and Bound – Knapsack Problem

- **Problem statement**
- Given n items of known weights w_i and values v_i , i = 1, 2, . . . , n, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

Solution approach

- It is convenient to order the items of a given instance in descending order by their value-to-weight ratios
- Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

Branch and Bound – Knapsack Problem

Construction of State Space Tree

- Each node on the i^{th} level of this tree, $0 \leq i \leq n$, represents all the subsets of **n** items that include a particular selection made from the first **i** ordered items.
- This particular selection is uniquely determined by the path from the root to the node:
  - a branch going to the left indicates the inclusion of the next item, and
  - a branch going to the right indicates its exclusion
- Record the total weight **w** and the total value **v** of this selection in the node, along with some upper bound **ub** on the value of any subset that can be obtained by adding zero or more items to this selection.

Branch and Bound – Knapsack Problem

Computation of upper bound

- A simple way to compute the upper bound **ub** is to add to **v**, the total value of the items already selected, the product of the remaining capacity of the knapsack **W – w** and the best per unit payoff among the remaining items.

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

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.

Branch and Bound – Knapsack Problem

- Solve the following instance of knapsack problem using branch and bound method

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.

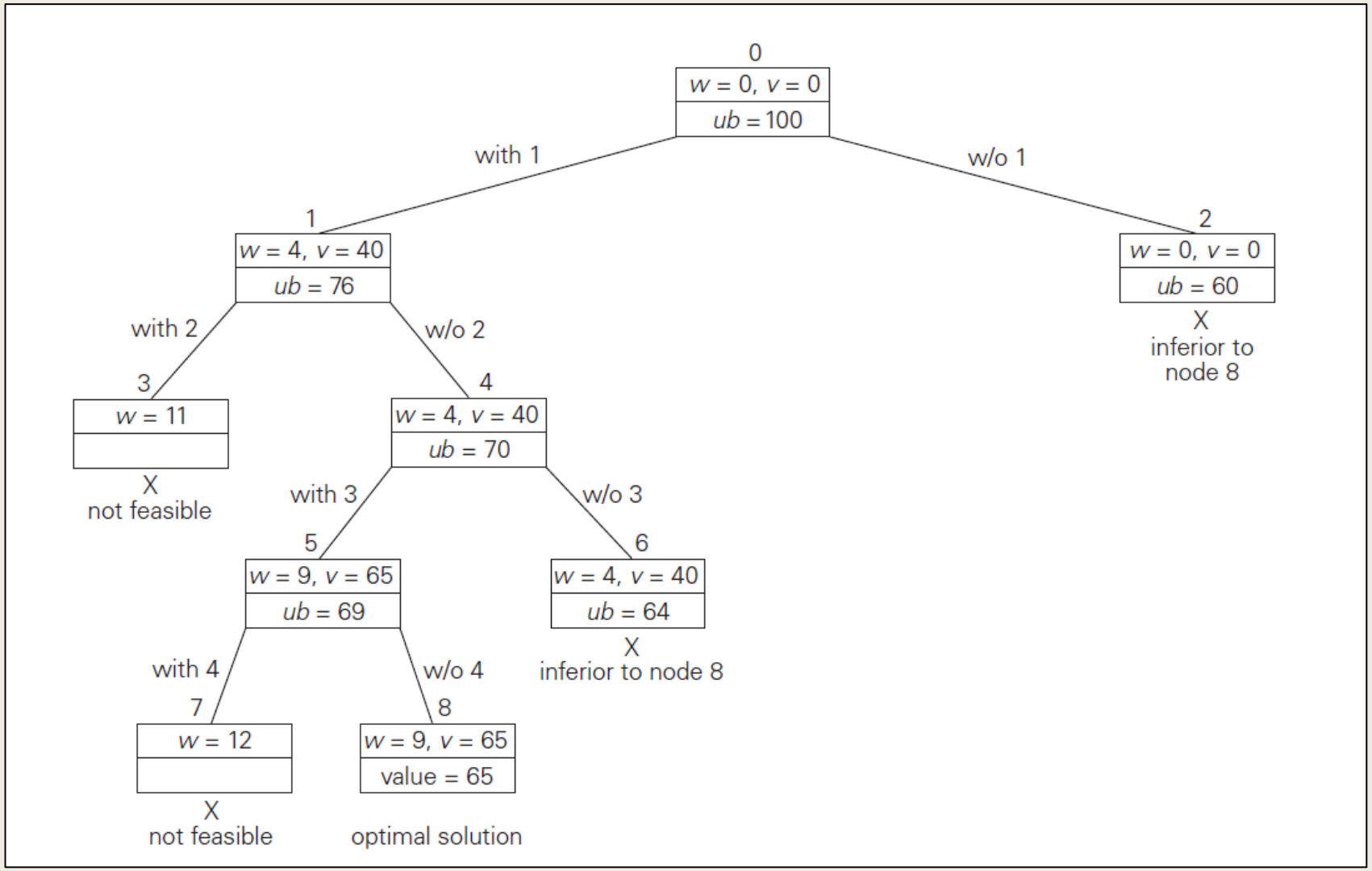
Branch and Bound – Knapsack Problem

Solution

Click for the solution

Branch and Bound – Knapsack Problem

- **State space tree**



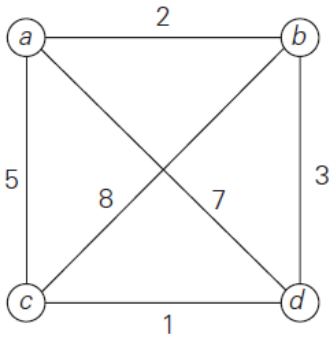
Branch and Bound – TSP

- The problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started
- The tour is a sequence of vertices $v_{i0}, v_{i1}, \ldots, v_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct.
- Al the tours starts and ends at one particular vertex (Hamiltonian circuit)
- Thus, we can get all the tours by generating all the permutations of $n - 1$ intermediate cities, compute the tour lengths, and find the shortest among them.

Branch and Bound – TSP

Observations

- Three pairs of tours that differ only by their direction.
- Hence, we could cut the number of vertex permutations by half.
- for example, choose any two intermediate vertices, say, ***b*** and ***c***, and then consider only permutations in which ***b*** precedes ***c***.
- The total number of permutations needed is still $\mathbf{1/2}\ (\boldsymbol{n}\ -\ \mathbf{1})!$
- Exhaustive search approach impractical!



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Branch and Bound – TSP

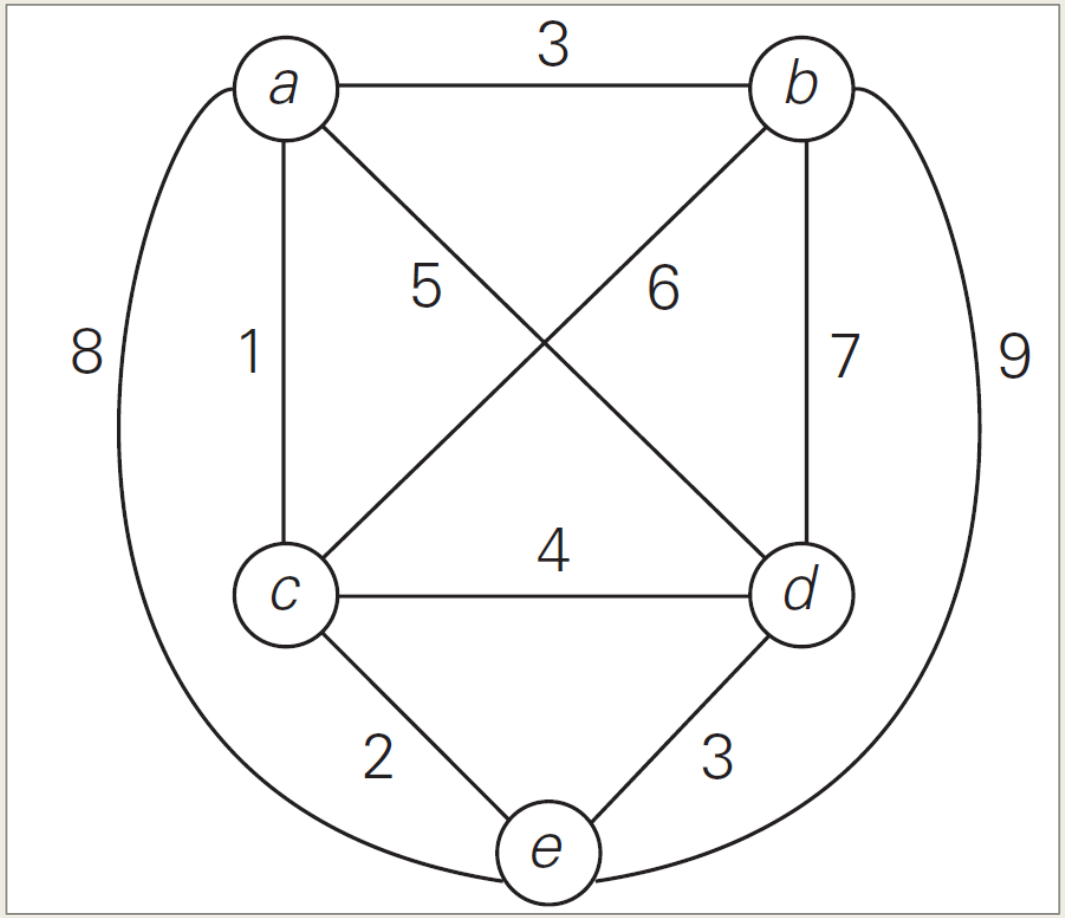
- **Computation of lower bound**
- For each city ***i*, 1≤ *i* ≤ *n***, find the sum s_i of the distances from city ***i*** to the two nearest cities;
- compute the sum ***s*** of these ***n*** numbers, divide the result by ***2,*** and,
- if all the distances are integers, round up the result to the nearest integer:

$$\mathbf{lb} = \lceil s/2 \rceil$$

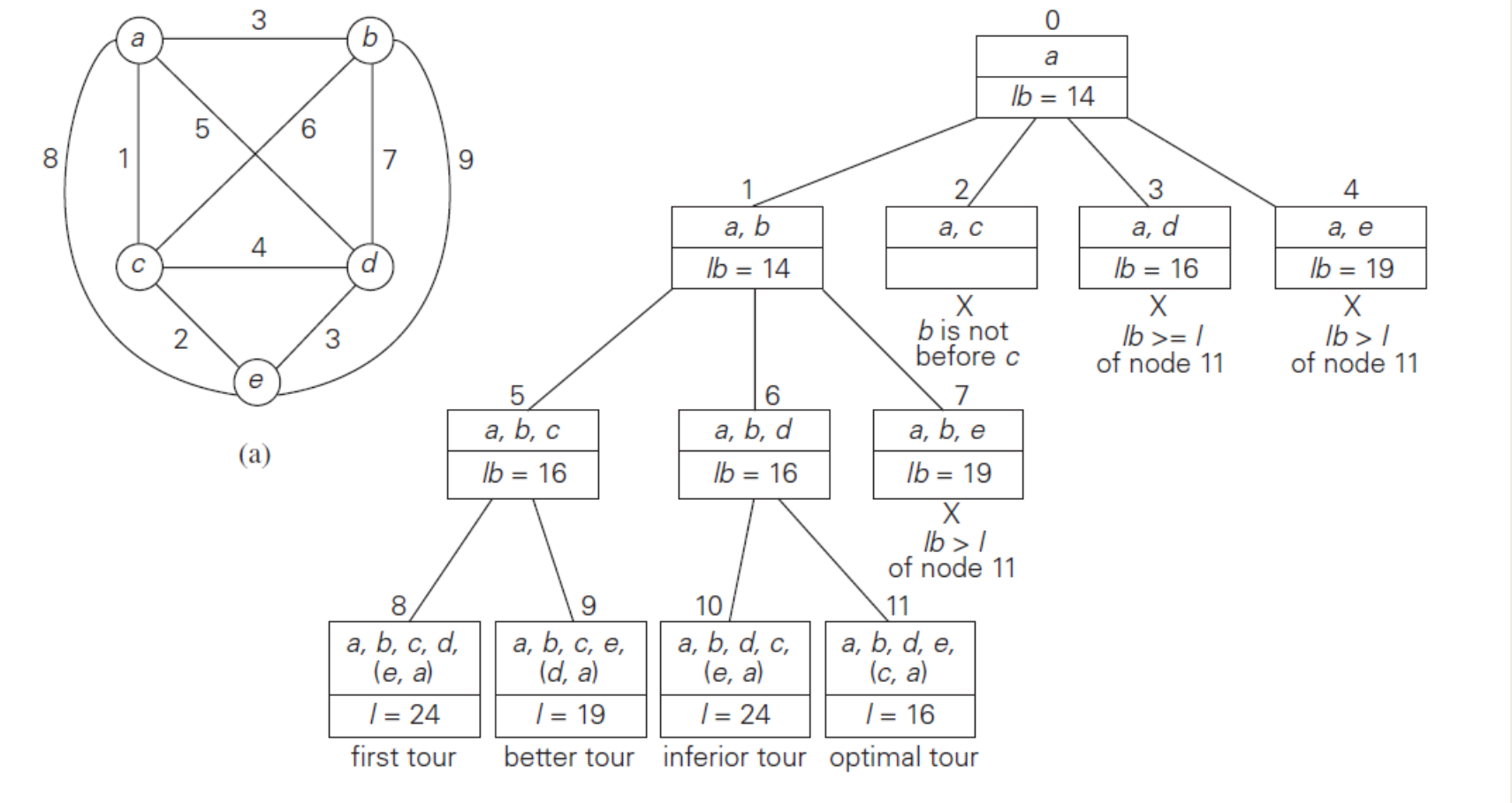
Branch and Bound – TSP

- **Solve the following instance of TSP using branch and bound method**

Solution



Branch and Bound – TSP



THANK YOU