Backtracking

What is backtracking?

- Useful technique for optimizing search under some constraints
- Express the desired solution as an n-tuple ($x_1, \ldots, x_n$) where each $x_i \in S_i$, S_i being a finite set
- The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function $P(x_1, \ldots, x_n)$
- Sorting an array a[n]
  - Find an n-tuple where the element x_i is the index of i^{th} smallest element in a
  - Criterion function is given by $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i < n$

What is backtracking?

- Brute force approach
  - Let the size of set S_i be m_i
  - There are $m = m_1 m_2 \cdots m_n$ n-tuples that satisfy the criterion function P
  - In brute force algorithm, you have to form all the m n-tuples to determine the optimal solutions
- Backtrack approach
  - Form a solution (partial vector) and check at every step if this has any chance of success
  - If the solution at any point seems not-promising, ignore it
  - If the partial vector $(x_1, x_2, \ldots, x_i)$ does not yield an optimal solution, ignore $m_{i+1} \cdots m_n$ possible test vectors even without looking at them
  - Determine problem solution by systematically searching the solution space for the given problem instance using a tree organization for solution space

Explicit & Implicit Constraints

- All the solutions are required to satisfy a set of constraints divided into two categories: explicit and implicit constraints

Explicit Constraints

- Explicit constraints are rules that restrict each x_i to take on values only from a given set.
- Explicit constraints depend on the particular instance I of problem being solved
- All tuples that satisfy the explicit constraints define a possible solution space for I
- Examples of explicit constraints
  - $x_i \geq 0$, or all nonnegative real numbers
  - $x_i = \{0, 1\}$
  - $l_i \leq x_i \leq u_i$

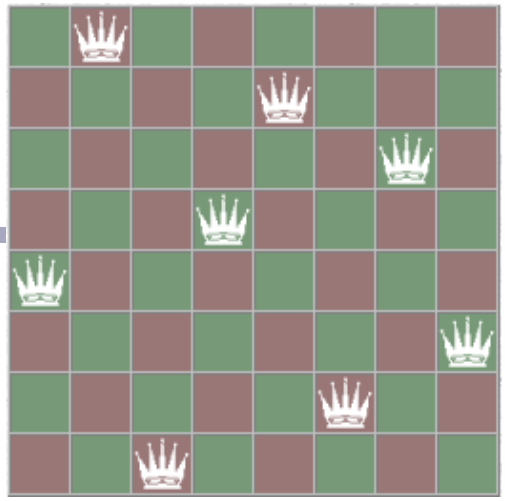
Implicit Constraints

- Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.
- Implicit constraints describe the way in which the x_is must relate to each other.

Backtracking

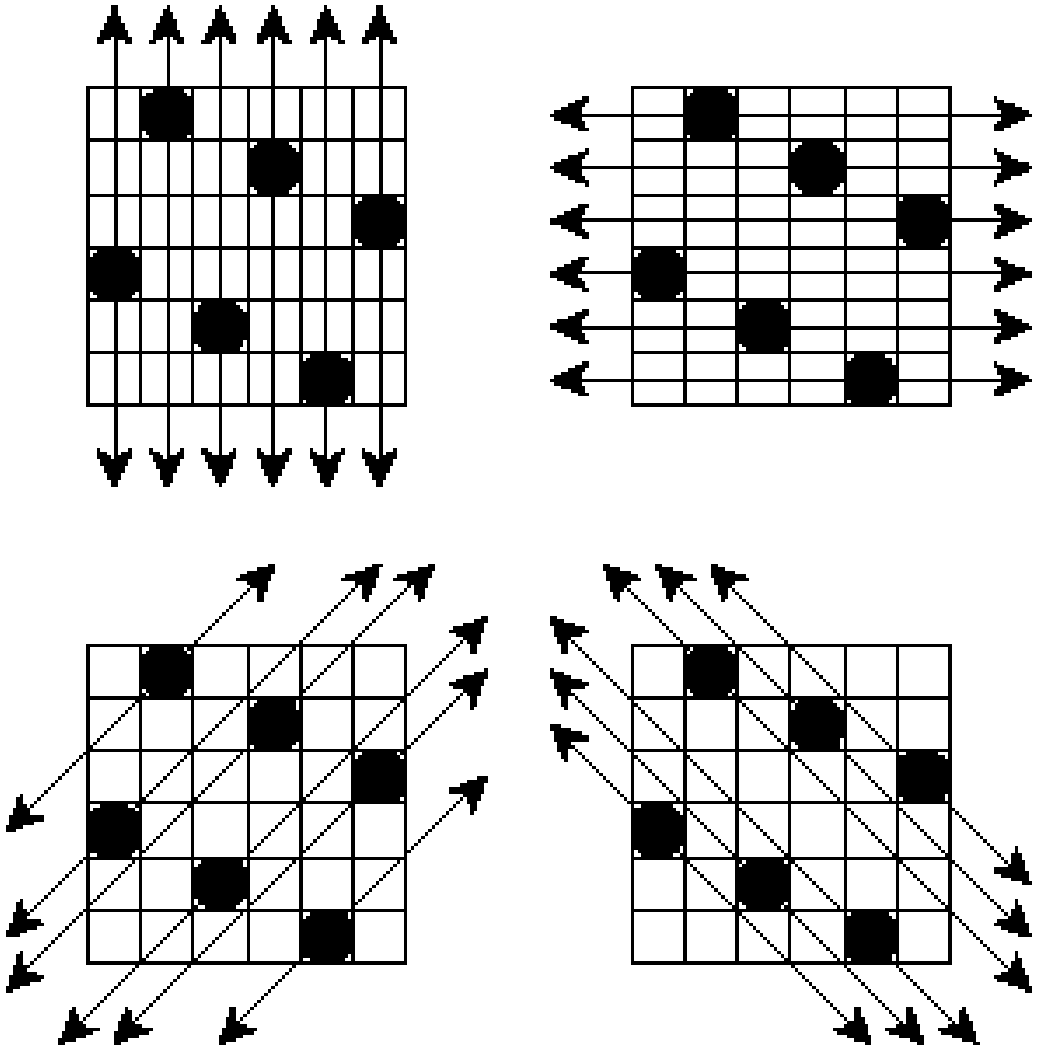
- Two versions of backtracking algorithms
  - Solution needs only to be feasible (satisfy problem's constraints)
    - sum of subsets
    - N- queens problem
  - Solution needs also to be optimal
    - knapsack

N-Queens Problem



- *N-Queens* dates back to the 19th century (studied by Gauss)
- Classical combinatorial problem, widely used as a benchmark because of its simple and regular structure
- Problem involves placing N queens on an $N \times N$ chessboard such that no queen can attack any other
- Queen attacks other at the same row, column or diagonal line
- Benchmark code versions include finding the first solution and finding all solutions

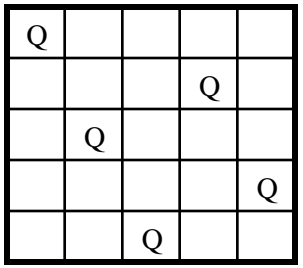
A Solution for 6-Queen



8-queens problem

- Identify data structures to solve the problem
  - We may define the solution to be an 8×8 array
  - However, since each queen is in a different row, define the chessboard solution to be an 8-tuple $(x_1, \ldots, x_8)$, where x_i is the column for i^{th} queen

i =	1	2	3	4	5
x [ ]	1	3	5	2	4



8-queens problem

- Identify explicit constraints
  - Explicit constraints using 8-tuple formulation are S = {1, 2, 3, 4, 5, 6, 7, 8}
  - Solution space of 8^8 8-tuples
- Identify implicit constraints
  - No two x_i can be the same, or all the queens must be in different columns
  - All solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8)
  - Reduces the size of solution space from 8^8 to 8! tuples
  - No two queens can be on the same diagonal

The Backtracking Method

- A given **problem** has a set of constraints and possibly an objective function
- The **solution** optimizes an objective function, and/or is feasible.
- We can represent the **solution space** for the problem using a **state space tree**
  - The *root* of the tree represents 0 choices,
  - 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

Terms Related to Solution Space Trees

- **Problem state** is each node in the depth-first search tree
- **State space** is the set of paths from root node to other nodes
- **Solution states** are the problem states s for which the path from the root node to s defines a tuple in the solution space
  - Partitioned into disjoint sub-solution spaces at each internal node
- **Answer states** are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions
  - These states satisfy implicit constraints
- **State space tree** is the tree organization of the solution space
- **Static trees** are ones for which tree organizations are independent of the problem instance being solved
- **Dynamic trees** are ones for which organization is dependent on problem instance

BACKTRACKING (Contd..)

- The problem may be solved by systematically generating the problem states determining which are solution states, and determining the answer states.
- Let us see the following terminology
- LIVE NODE A node which has been generated and all of whose children are not yet been generated .
- E-NODE (Node being expanded) - The live node whose children are currently being generated .
- DEAD NODE - A node that is either not to be expanded further, or for which all of its children have been generated.

BACKTRACKING (Contd..)

- DEPTH FIRST NODE GENERATION- In this, as soon as a new child C of the current E-node R is generated, C will become the new E-node.

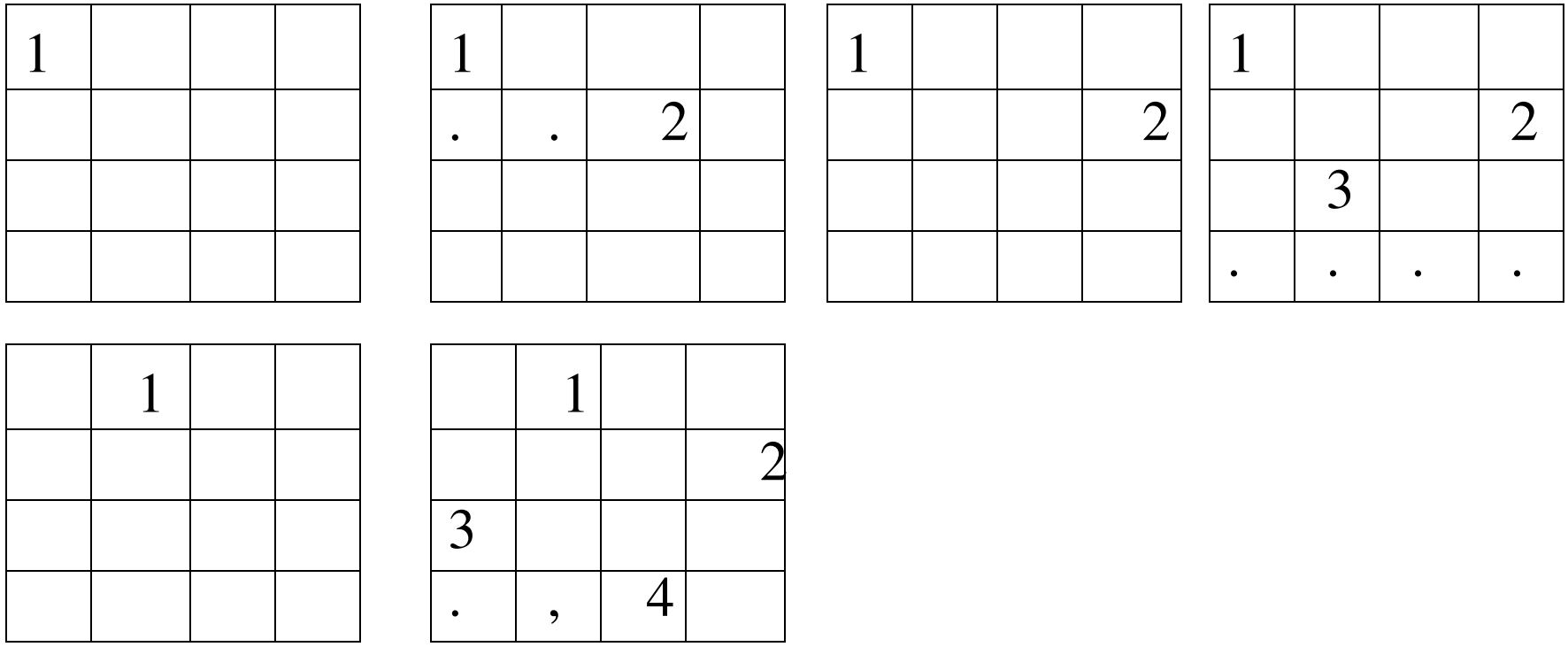
  R will become E-node again when C has been fully explored.
- BOUNDING FUNCTION - will be used to kill live nodes without generating all their children.

BACKTRACKING (Contd..)

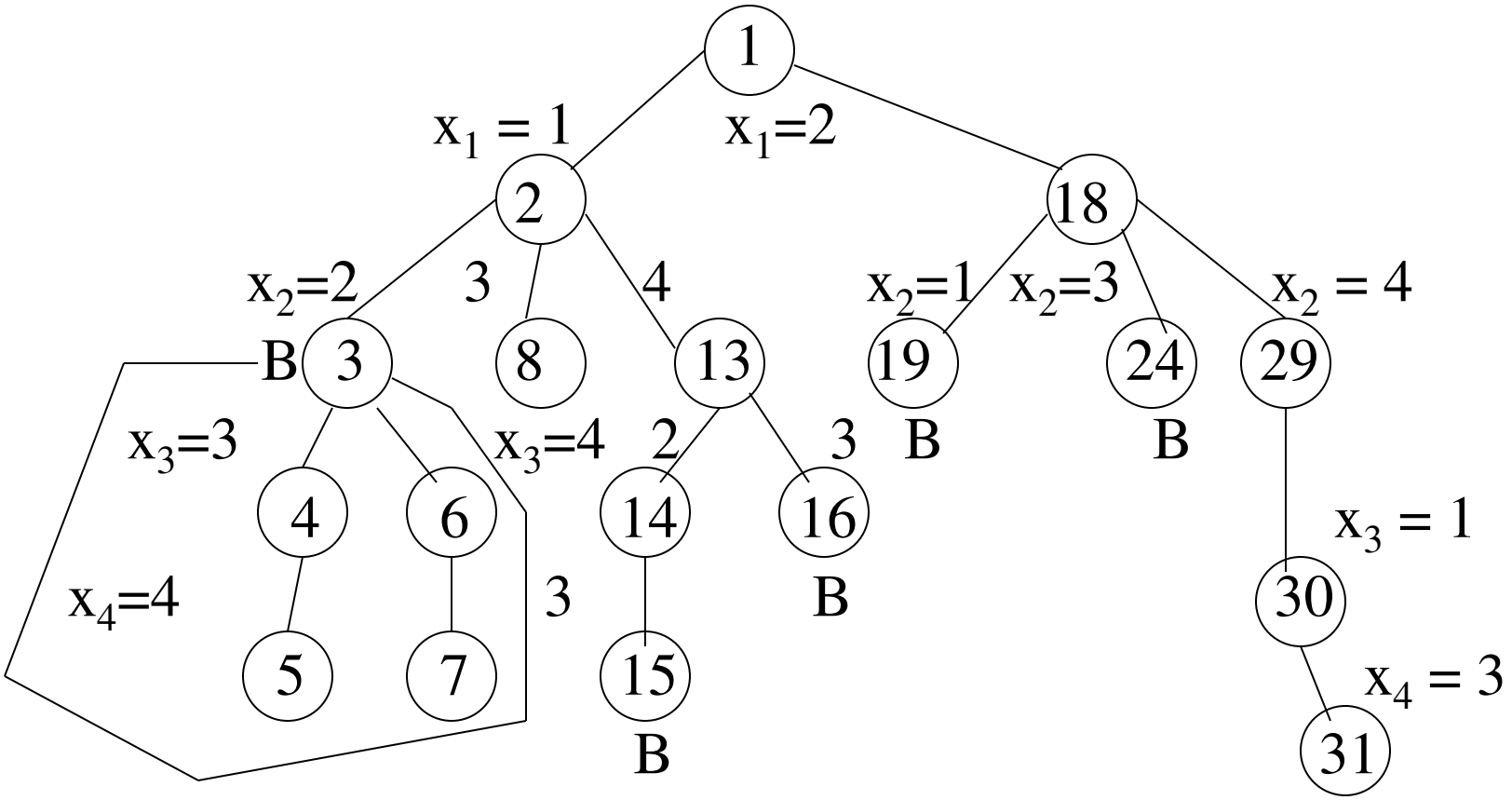
- BACTRACKING
  - Depth – first node generation with bounding functions.
  - New nodes are placed in to a stack. The last node added is the first to be explored.
- BRANCH-and-BOUND
  - Breadth – first node generation method in which E-node remains E-node until it is dead.
  - Each new node placed in a queue. The front of the queue becomes the new E-node.

BACKTRACKING (Contd..)

Example : 4 Queens problem



BACKTRACKING (Contd..)



No two queens on the same diagonal

If two queens are placed at (i,j) & (k,l), then they are on the same diagonal only if

$i - j = k - l \Rightarrow j - l = i - k$

Or

$i + j = k + l \Rightarrow j - l = k - i$

This means that two queens are at the same diagonal iff

$Abs(j - l) = Abs(k - i)$

Condition for testing if queen k placed on column i is on the same diagonal as queen j

abs(x[j]-i) == abs(j-k)

N-queens Backtracking Algorithm

```
Nqueens(k, n)
{
for i = 1 to n
    { if place(k, i) then
          {
          x[k] = i
          if (k==n) then
                    write (x[1..n])
          else Nqueens(k+1, n);
          }
    }
}
```

```
place(k, i)
{
for j= 1 to k-1
    {
    if (x[j] == i) or
          (abs(x[j]-i) == abs(j-k))
    then return false
    }
return true
}
```

Backtracking
Sum of Subsets
Hamiltonian Cycle
Graph Coloring

Sum of subsets

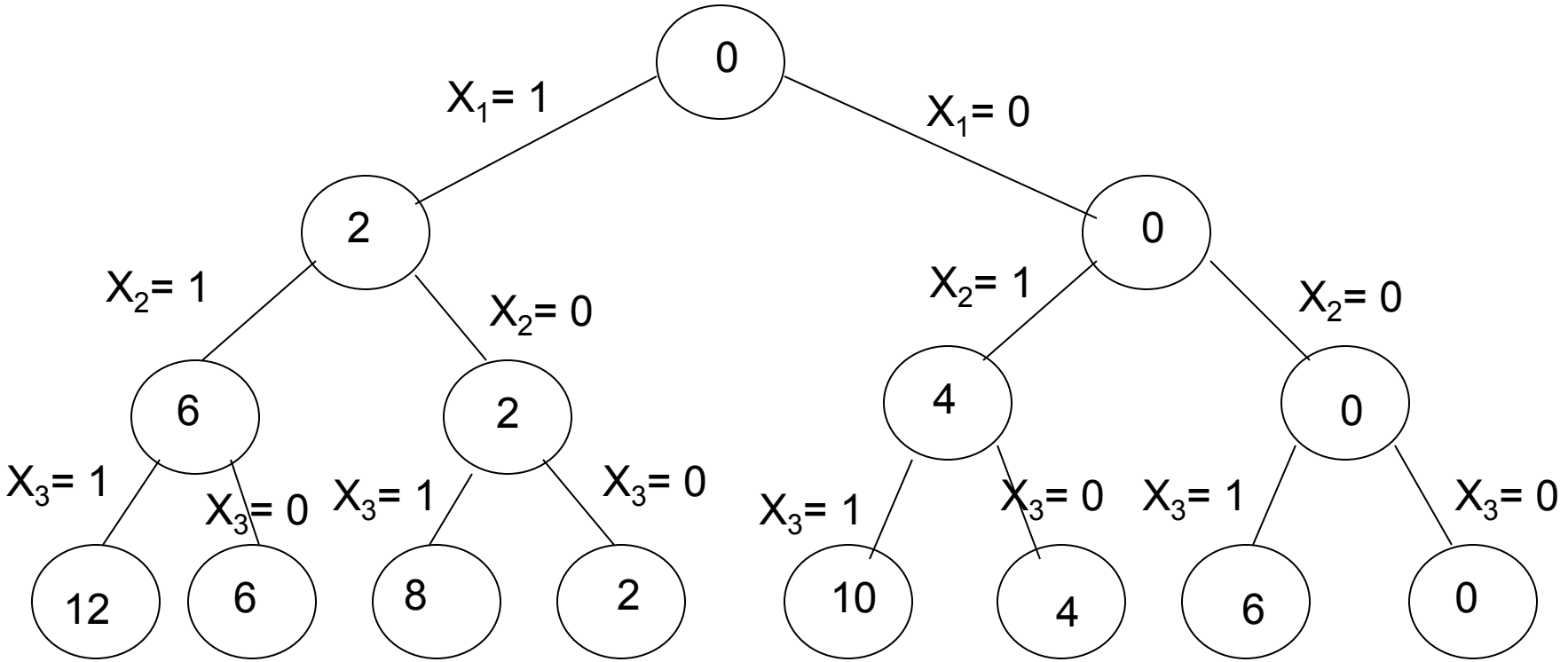
- **Problem**: Given n positive integers $w_1, \ldots w_n$ and a positive integer m. Find all subsets of $w_1, \ldots w_n$ that sum to m.
- **Example**:
  n = 4, w = (11, 13, 24, 7), and m = 31
- **Solutions**:
  (11, 13, 7) and (24, 7)

Solution Vectors - Fixed size tuples

- A simple formulation of the problem is where the solution subset is represented by an n-tuple ($x_1, \ldots, x_n$) such that $x_i = 0$ if w_i is not chosen and $x_i = 1$ if w_i is chosen.
- **Explicit constraint -** $x_i \in \{0, 1\}$
- The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1)
- **Implicit constraint** – For all $x_j = 1$, $\Sigma w_j = m$
- The solution space is 2^n distinct tuples

State SpaceTree for 3 items using fixed tuple formulation

*w*1 = 2, *w*2 = 4, *w*3 = 6 and *m* = 6



The sum of the included integers is shown at the node.

Draw State SpaceTree for 4 items using fixed tuple formulation

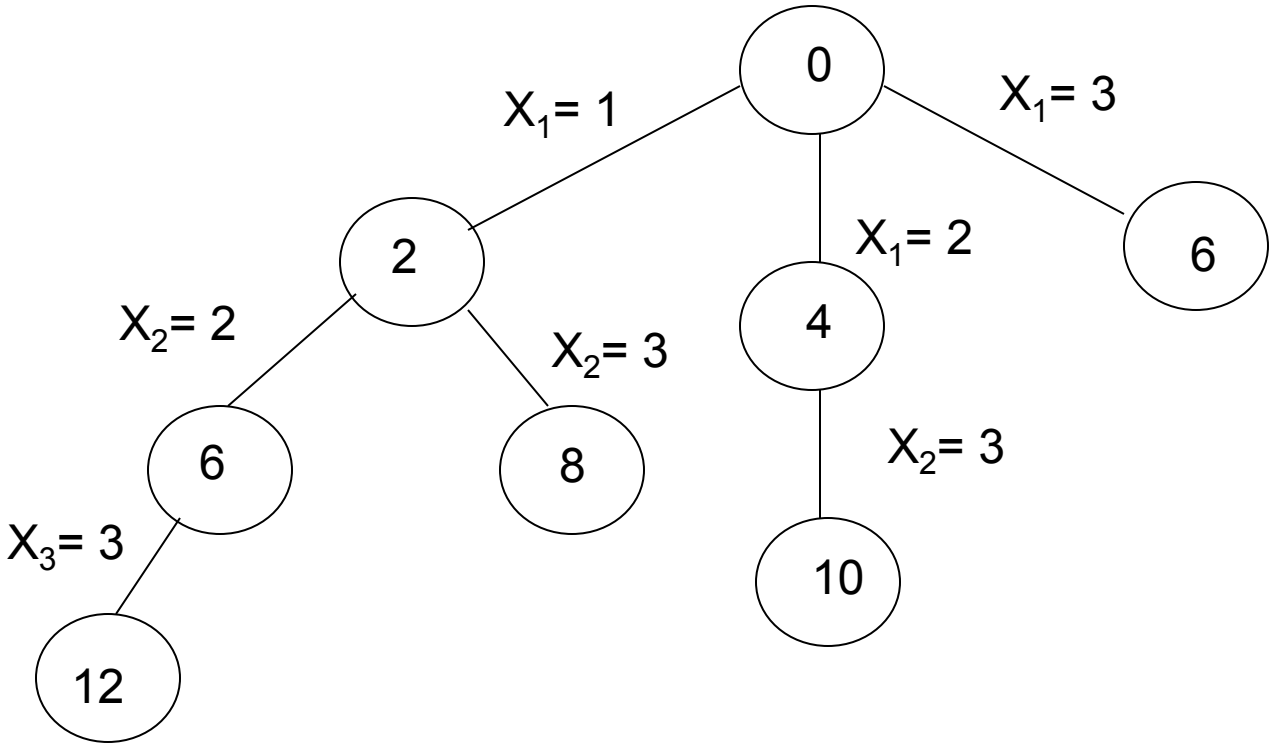
$w1 = 2$, $w2 = 4$, $w3 = 6$, $w4 = 8$ and $m = 12$

Solution Vectors - Variables size tuples

- The solution vectors can also be represented by the indices of the numbers as (1, 2, 4) and (3, 4)
  - All solutions are k-tuples, $1 \leq k \leq n$
- **Explicit constraints**
  - $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$
- **Implicit constraints**
  - No two x_i can be the same
  - $\Sigma x_i = m$
  - $x_i < x_{i+1}$, $1 \leq i < k$ (total order in indices) – this avoids generating multiple instances of the same set e.g. (1,2,4) and (2,4,1)
- The solution space is 2^n distinct tuples

State SpaceTree for 3 items using variable tuple formulation

*w*1 = 2, *w*2 = 4, *w*3 = 6 and *m* = 6



The sum of the included integers is shown at the node.

Draw State SpaceTree for 4 items using variable tuple formulation

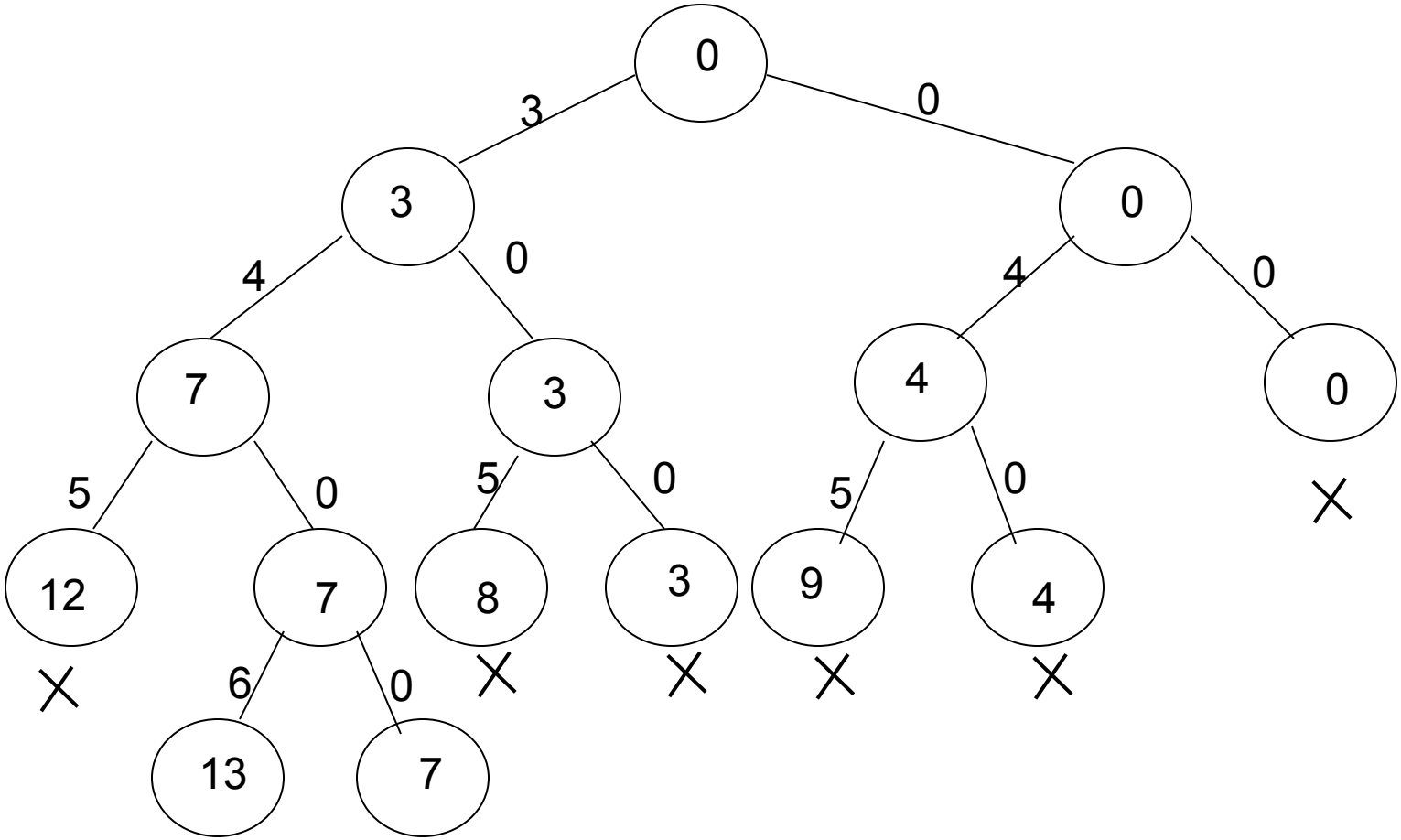
$w1 = 2$, $w2 = 4$, $w3 = 6$, $w4 = 8$ and $m = 12$

When is a node “promising”?

- Sort the weights in non-decreasing order
- Let *weight* be the subtotal from root to node i at level i.
- A node is non-promising if
  - ***weight +w[i+1] > W*** - any descendant of node i will be nonpromising because w[i+1] is the lightest weight remaining.
  - ***weight + weight of all remaining items < W*** - any descendant of node i will be nonpromising

A Pruned State Space Tree (find all solutions)
$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6;$ m = 13

There are only 15 nodes in the pruned state space tree
The full state space tree has 31 nodes



Sum of Subsets – Algorithm

```
sumOfSubsets (weight, i, totalLeft )
{
if ((weight +  totalLeft ≥ m) and (weight +  w[i + 1] ≤ m or weight
    = m ))
    if ( weight = m )
        then print x[ 1 ] to x[ i ]      //found solution
    else
    {
    x [ i+1 ] = 1                        //try including
    sumOfSubsets (weight + w[i+1], i + 1, totalLeft - w[i+1] )
    x [ i+1 ] = 0                        //try excluding i+1
    sumOfSubsets (weight, i + 1,  totalLeft - w[i+1] )
    }
}
```

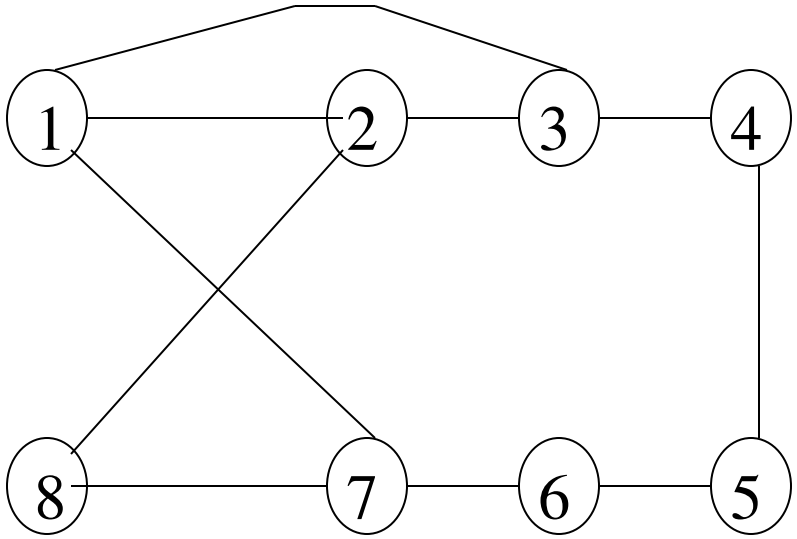
Initial call sumOfSubsets(0, 0, $\sum_{i=1}^{n} w_i$)

n=6, w[1:6]={5,10,12,13,15,18}, m=30

HAMILTONIAN CYCLES

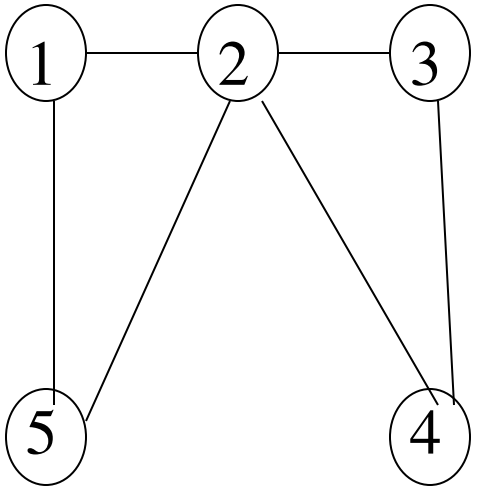
- Let G=(V,E) be a connected graph with n vertices .
- A Hamiltonian cycle is a round path along n edges of G which visits every vertex once and returns to its starting position.
- The tour of a traveling salesperson problem is a Hamiltonian cycle.
- A tour may exist or not.

HAMILTONIAN CYCLES (Contd..)



(a)

Hamiltonian Cycle is 1,2,8,7,6,5,4,3,1



(b)

No Hamiltonian Cycle

HAMILTONIAN CYCLES (Contd..)

- The backtracking solution is a vector $(x_1,\ldots\ldots,x_n)$ where x_i represents the i^{th} visited vertex of the cycle.
- To avoid printing of the same cycle n times we require $X_1 = 1$ (as 12876543, 28765431, 8765431 are same Ham Cycle)
- We compute X_k given $(x_1\ldots..x_{k-1})$ have already been chosen.
  - If $1<k<n$ then x_k can be any vertex v that is distinct from $x_1\ldots..x_{k-1}$ and v is connected by an edge to x_{k-1}.
  - The vertex x_n must be the only one remaining vertex and it must be connected to both x_{n-1} and x_1
- Two algorithms NEXTVALUE(k) and HAMILTONIAN are used,to find the tour.
- We initialize Graph (1:n,1:n) and X(2:n)$\leftarrow$0, X(1)$\leftarrow$1 and start with HAMILTONIAN (2).

HAMILTONIAN CYCLES - Algorithm

```
HAMILTONIAN(k)
    {
    repeat // generate values for X(k) //
            {
            NEXTVALUE(k) // assign a legal vertex to X(k)  //
            if X(k)=0 then return
            if k = n then
                        print X   // a cycle is printed //
            else
                        HAMILTONIAN (k+1)
            } until(false)

    }
```

HAMILTONIAN CYCLES -Next Value Algorithm

```
NEXTVALUE(k)
{
repeat
    {
    X(k)← (X(k)+1) mod(n+1) // next vertex //
    if (X(k) = 0) then return
    if GRAPH (X(k-1),X(k)) ≠ 0) then // if there is an edge //
            {
            for j←1 to k-1 do // check whether the same  vertex is generated //
                    if X(j) = X(k) then break  // for loop exit //
            if (j = k) then // the for loop is exited with  last value //
                    if ((k < n) or (k = n and GRAPH[X(n),X[1]] ≠ 0)) then return

            }
    } until (false)
}
```

HAMILTONIAN CYCLES - Algorithm Example

Example:

Let n = 8

X(1) = 1, HAMILTONIAN(2) i.e.H(2)

is called, so NEXTVALUE(2) i.e.N(2) is called.

Initially X(2)=0

X(2)=0+1 mod 9 = 1 but X(1) = X(2)

so loop is repeated and X(2) = 2 mod 9 = 2

X(1)≠X(2) and j=k=2, k < 8 so return 2

NV(3)=8 as Graph(2,3), Graph(2,5)

HAMILTONIAN CYCLES - Algorithm Example contd..

Graph(2,6),Graph(2,7),Graph(2,4) are false.

Thus NV(4) = 7,NV(5) = 6,NV(6) = 5 NV(7) = 4, NV(8) = 3.

At NV(8), k = 8 and GRAPH(X(8),1)

is satisfied. Thus

The cycle is printed.

GRAPH COLOURING PROBLEM

- Let G be a graph and m be a positive integer .
- The problem is to colour the vertices of G using only m colours in such a way that no two adjacent nodes / vertices have the same color.
- It is necessary to find the smallest integer m.

  m is referred to as the chromatic number of G.
- A special case of graph colouring problem is the four colour problem for planar graphs .

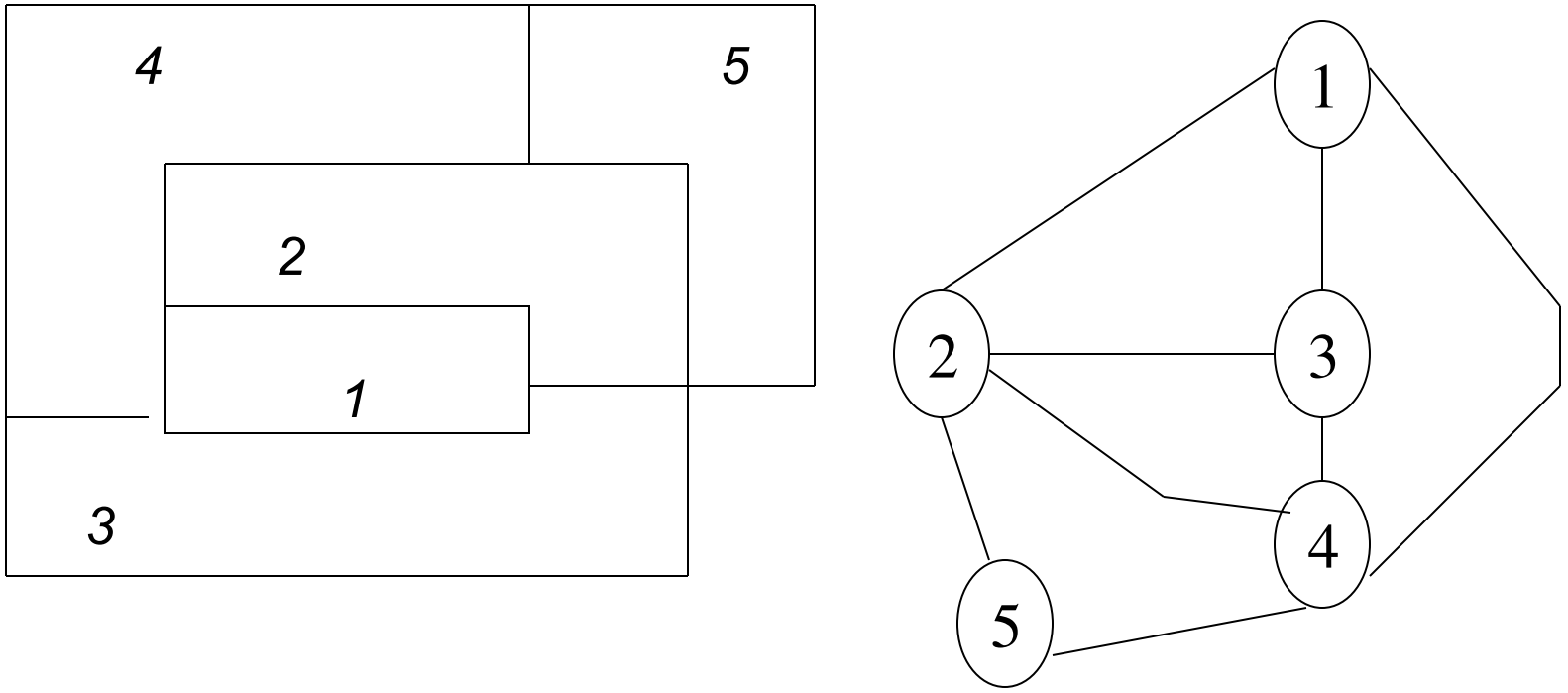
GRAPH COLOURING PROBLEM (Contd..)

- A graph is planar iff it can be drawn in a plane in such a way that no two edges cross each other.
- 4- colour problem for planar graphs Given any map, can the regions be coloured in such a way that no two adjacent regions have the same colour with only four colours?

GRAPH COLOURING PROBLEM (Contd..)

- A map can be transformed into a graph by representing each region of map into a node and if two regions are adjacent, then the corresponding nodes are joined by an edge.
- For many years it was known that 5 colours are required to colour any map.
- After a several hundred years, mathematicians with the help of a computer showed that 4 colours are sufficient.

GRAPH COLOURING PROBLEM (Contd..)



A map and its planar graph representation

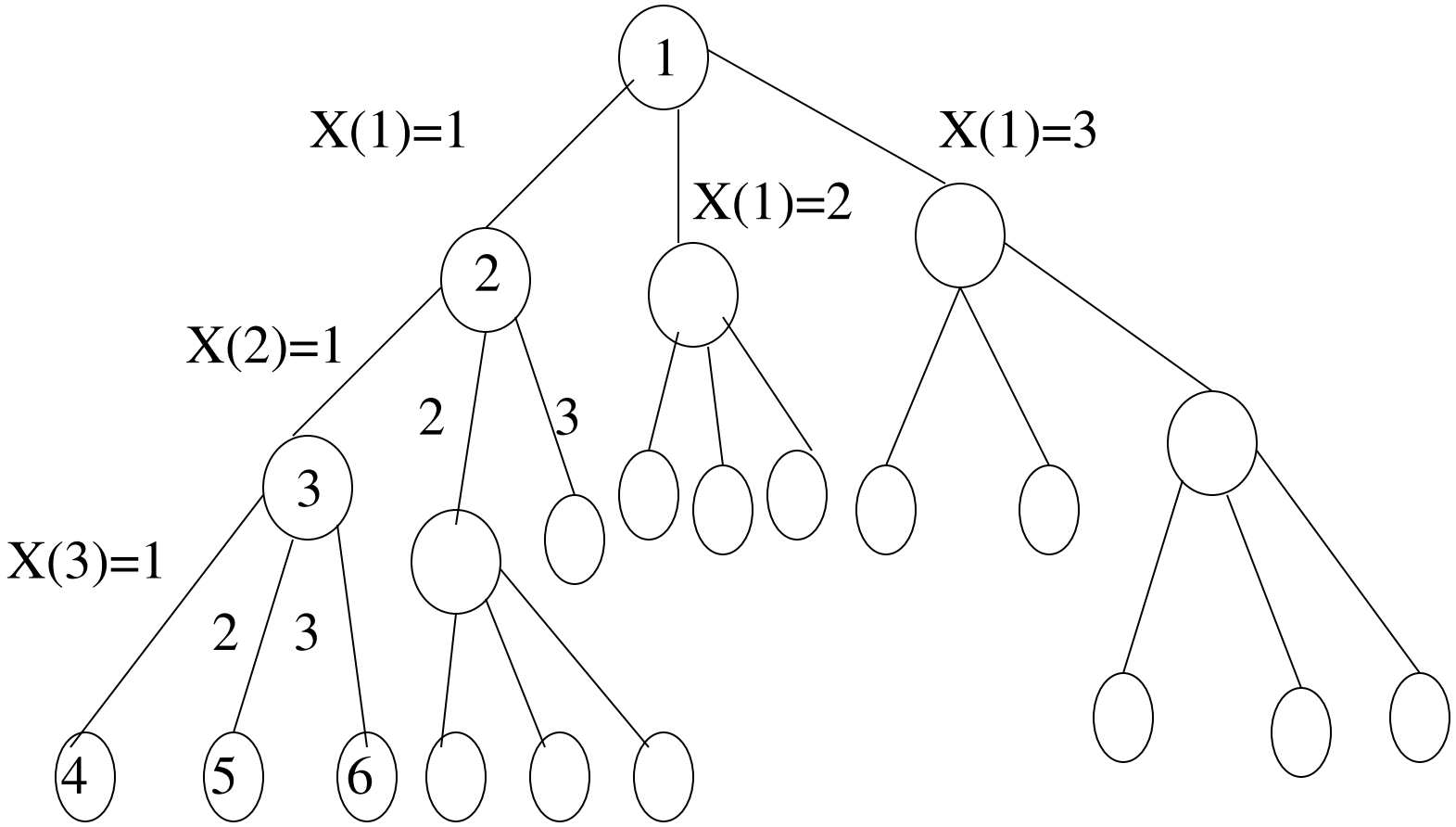
Solving the Graph Colouring Problems

- The graph is represented by its adjacency matrix Graph (1:n,1:n) where GRAPH (i,j) = 1 if <i,j> is an edge and Graph (i,j) = 0 otherwise.
- The colours will be represented by the integers 1,2….m and the solution with n–tuple (X(1),….X(n)), where X(i) is the colour of node i.

Solving the Graph Colouring Problems (Contd..)

- The solution can be represented as a state space tree.
- Each node at level i has m children corresponding to m possible assignments to X(i) $1 \leq i \leq m$.
- Nodes at level n+1, are leaf nodes. The tree has degree m with height n+1.

State space tree for m colouring problem with n = 3 and m = 3



Backtracking Algorithm for Graph Coloring Problem

- Two algorithms NEXTVALUE(k) and MCOLORING are used.
- First the array X is initialized to 0.
- Initial call is MCOLORING(0)
- When NEXTVALUE(k) is invoked
  - X(1)….X(k-1) have been assigned to vertices 1,…k-1
  - X(k) is the next highest numbered colour different from those adjacent to k

Graph Colouring Problem-Algorithm

```
MCOLORING (k)
{
repeat
    {
    NEXTVALUE(K) // assign to X(k) a legal colour //
    if X(k) = 0 then return // no new colour is possible //
    if k = n then Print X // m colours are assigned to n vertices //
    else MCOLOURING(k+1)
    } until (false)
}
```

Solving the Graph Colouring Problems (Contd..)

```
NEXTVALUE(k)
{
repeat
    {
    X(k)←( X(k) +1 ) mod(m+1)  //next highest colour//
    if X(k)=0 then return //All colours have been   exhausted//
    for j← 1 to n do //Check if this colour is distinct from adjacent colours//
            {
            if (GRAPH(k,j)≠0 and X(k)=X(j))  then break
            }
    if j = n+1 then return
            // the loop is exited with the last value //
            // i.e. j = n+1 and hence new colour is found //
    } until (false) // try to find another colour //
}
```

Solving the Graph Colouring Problems (Contd..)

Example for NEXTVALUE(k)

Let m = 3 n = 4

Nextvalue(1)

X(1)=(0+1) mod 4 =1

for j←1 to 4

1≠0 so j=5 so return

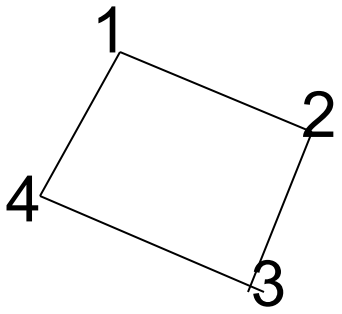
Nextvalue(2) returns (0+1)mod4=1

but X(2)=X(1) for j=1 and j≠5 so

loop is repeated and Nextvalue(2) is 2

Similarly Next value(3) is 3 and

Next value (4) is 2



initially

X(1)=X(2)=X(3)

=X(4)=0

Branch and Bound Algorithms

Terminology

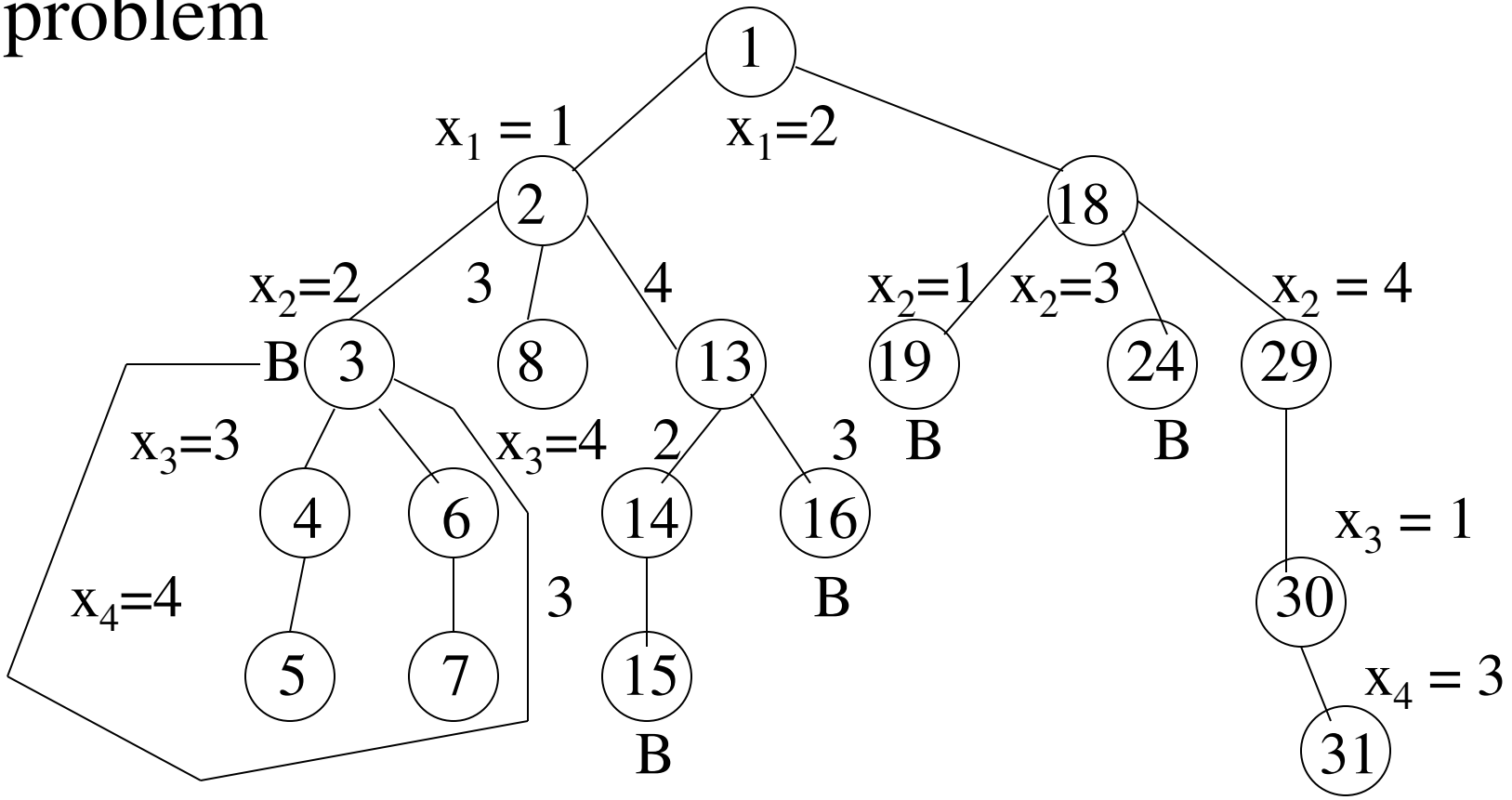
- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead noden have already been expanded.
- **Branch-and-bound** refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.
  - Used for state space search
  - In BFS, exploration of a new node cannot begin until the node currently being explored is fully explored

General method

- Both BFS and DFS generalize to branch-and-bound strategies
  - BFS is an FIFO search in terms of live nodes
  - List of live nodes is a queue
- DFS is an LIFO search in terms of live nodes
  - List of live nodes is a stack
- Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node

Example: 4-queens

- Backtracking is superior method for this search problem



Least Cost (LC) search

- Selection rule does not give preference to nodes that will lead to answer quickly but just queues those behind the current live nodes
  - In 4-queen problem, if three queens have been placed on the board, it is obvious that the answer may be reached in one more move
  - The rigid selection rule requires that other live nodes be expanded and then, the current node be tested
- Rank the live nodes by using a heuristic $\hat{c}(\cdot)$

LC Search

- $\hat{c}(x) = f(h(x)) + \hat{g}(x)$
  - $\hat{g}(x)$ is an estimate of the additional effort needed to reach an answer from node x
  - h(x) is the cost of reaching x from root and $f(\cdot)$ is any nondecreasing function
- Choose next E-node a live node with least $\hat{c}$

15-puzzle problem

- 15 numbered tiles on a square frame with a capacity for 16 tiles. Given an initial arrangement, transform it to the goal arrangement through a series of legal moves
- The state space of an initial state is all the states that can be reached from initial state

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

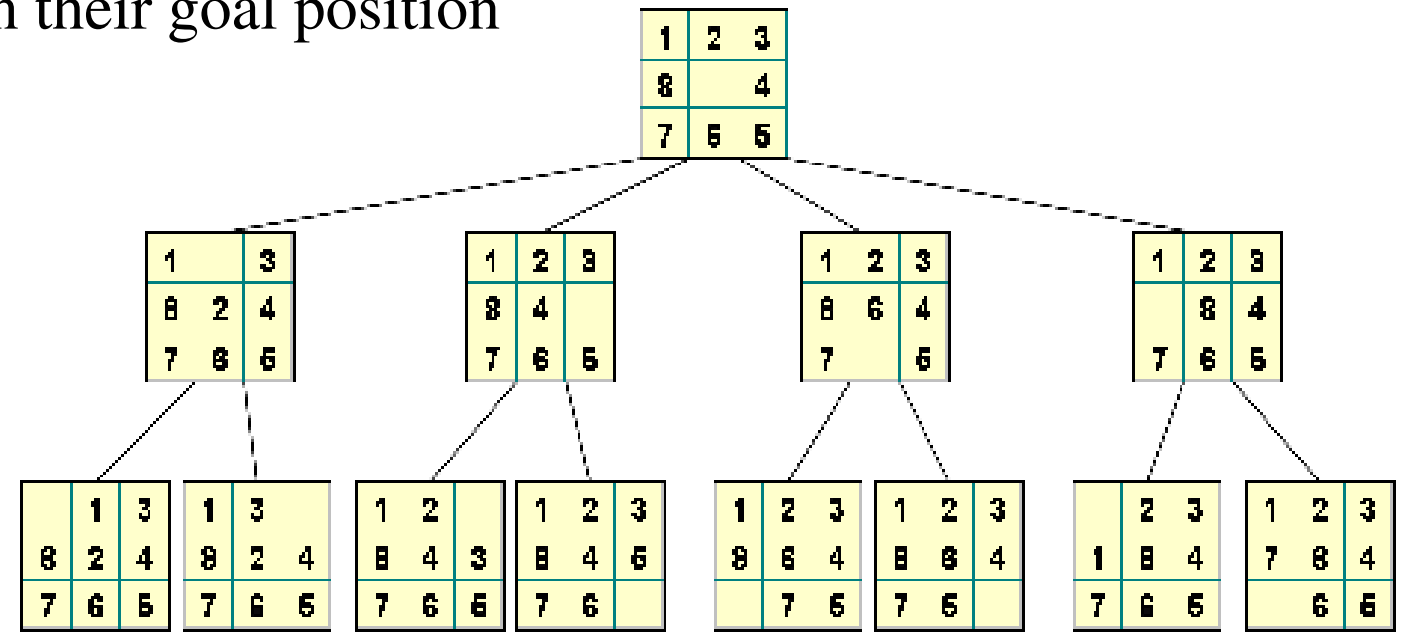
Initial State

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

Goal State

LC Search on 15-puzzle problem

- Children of each node x represent the states reachable from x in one legal move
- Consider the move as move of empty space rather than tile
- Empty space can have four legal moves: up, down, left, right
- One possible choice for $\hat{g}(x)$ is the number of nonblank tiles not in their goal position



Bounding

- Each answer node x has a cost c(x) and we have to find a minimum-cost answer node
- Use a cost function $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ provides lower bound on the solution obtainable from any node x
- If U is the upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > U$ may be killed
  - All answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > U$
  - Starting value for U can be obtained by some heuristic or set to ∞
  - Each time a new answer node is found, the value of U can be updated
- Optimization/minimization problem
  - Maximization converted to minimization by changing sign of objective function
  - Formulate the search for an optimal solution as a search for a least-cost answer node in a state space search tree

TSP using LCBB

- Dynamic Programming Solution takes $O(n^2 2^n)$ time
- Worst case of BB $O(n^2 2^n)$
- Significant time improvement through use of good bounding functions with Branch and Bound
- G = (V, E) represented through cost matrix C
- Assume every tour starts and ends at vertex 1
- Solution space is given by $S = \{1, \pi, 1 \mid \pi$ is a permutation of $(2,3,\ldots,n)\}$
- $|S| = (n-1)!$
- Size can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $(i_j, i_{j+1}) \in E$

TSP using LCBB

- $c(A) = \begin{cases} \text{length of tour defined by the path from} \\ \text{the root to node A, if A is a leaf} \\ \text{cost of min-cost in the subtree A, if A is not} \\ \text{a leaf} \end{cases}$
- $\hat{c}(A)$ = length of path defined at node A
- $\hat{c}(r) \leq c(r) \leq u(r)$ for all nodes r
- Better $\hat{c}(A)$ is obtained by reduced cost matrix corresponding to G
  - A row(column) is said to be reduced iff it contains at least one zero and all reaining entries are non-negative

Example

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

Cost Matrix

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

Reduced Cost Matrix

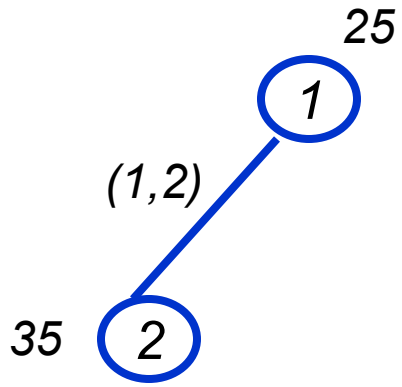
- A minimum cost tour remains a minimum cost tour after this operation.
- The total cost subtracted is the lower bound on the min cost tour
- This value is used as $\hat{c}$ (=25)

TSP using LCBB

- Associate a RCM with every node
- Let A be the RCM for node R & let S be a child of R such that tree edge (R,S) corresponds to including edge (i,j) in the tour.
- If S is not a leaf then RCM for S can be obtained as follows
  1. Change all entries in row i and column j to ∞. (Any edges leaving i or entering j are not used)
  2. A(j,1) = ∞ (edge (j,1) cannot be used again)
  3. Reduce all rows and columns except those containing only ∞. B is the resultant matrix
- If r is the total amount subtracted in step 3, then $\hat{c}(S) = \hat{c}(R) + A(i,j) + r$
- Initialize *upper* to ∞. Update upper after getting the first answer node. After that discard any nodes with $\hat{c} >$ *upper*

RCM for node 1

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



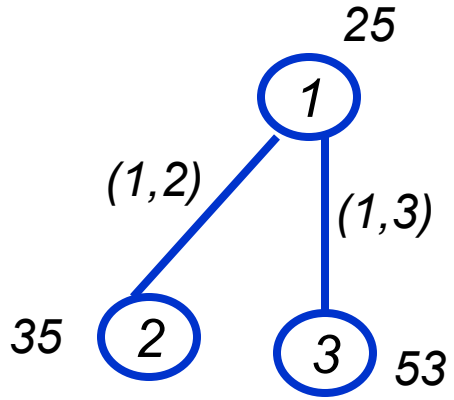
RCM for node 2

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

$\hat{C}(2) = \hat{C}(1) + A(1,2) + r$
$= 25 + 10 + 0 = 35$

RCM for node 1

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



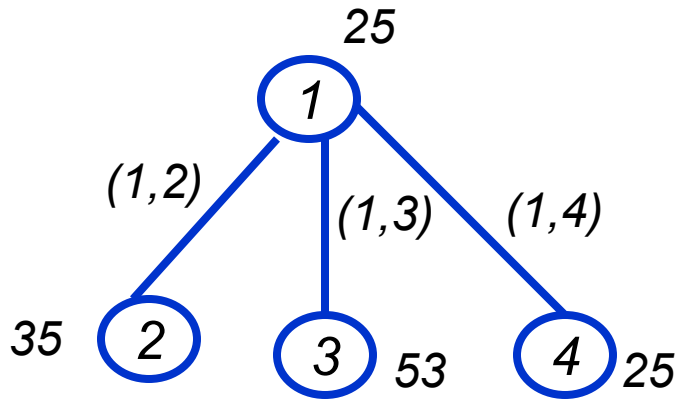
RCM for node 3

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

$\hat{C}(3) = \hat{C}(1) + A(1,3) + r$
$= 25 + 17 + 11 = 53$

RCM for node 1

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



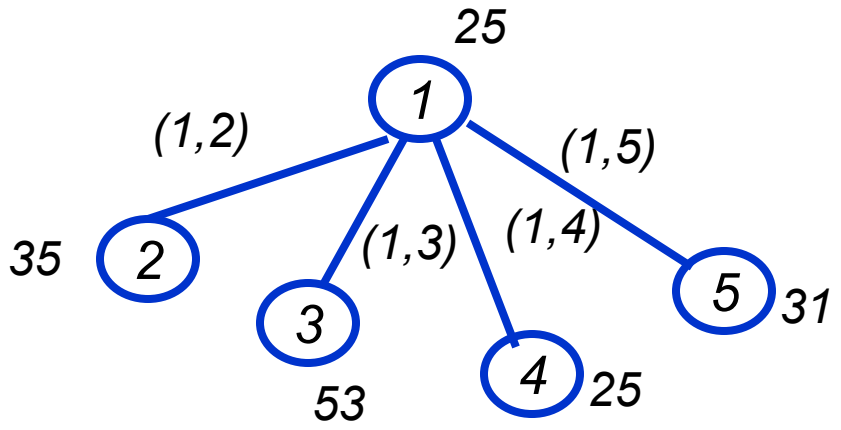
RCM for node 4

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

$\hat{C}(4) = \hat{C}(1) + A(1,4) + r$
$= 25 + 0 + 0 = 25$

RCM for node 1

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



RCM for node 5

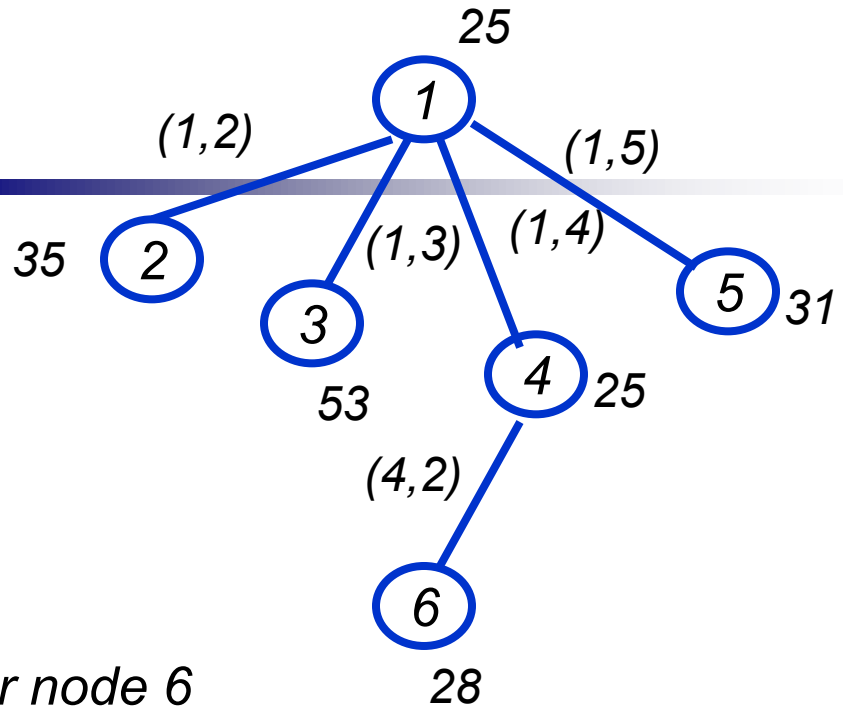
∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

$\hat{C}(5) = \hat{C}(1) + A(1,5) + r$
$= 25 + 1 + 5 = 31$

Node 4 becomes the next E-node

RCM for node 4

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



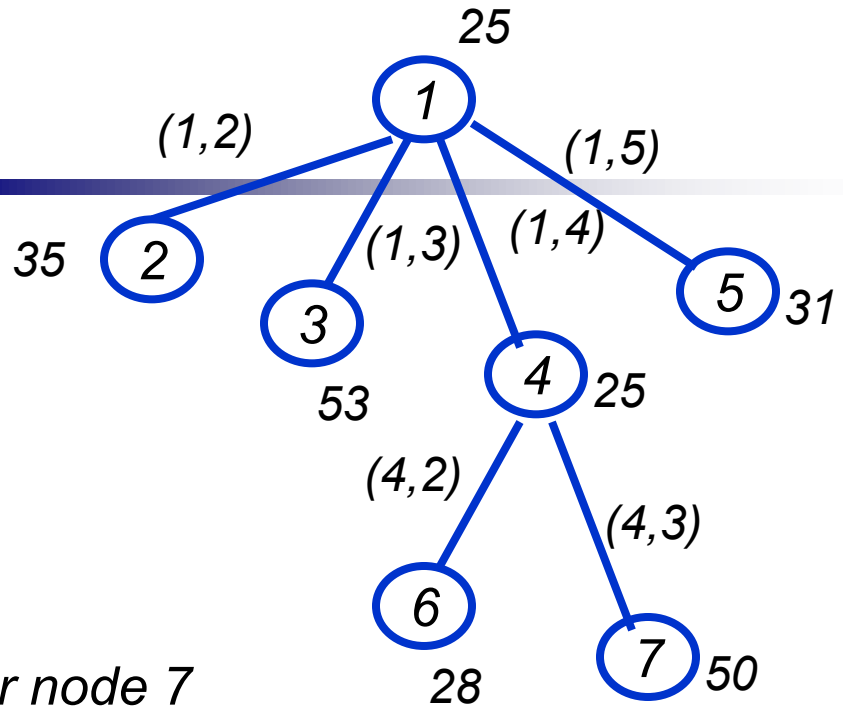
RCM for node 6

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

$\hat{C}(6) = \hat{C}(4) + A(4,2) + r$
$= 25 + 3 + 0 = 28$

RCM for node 4

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



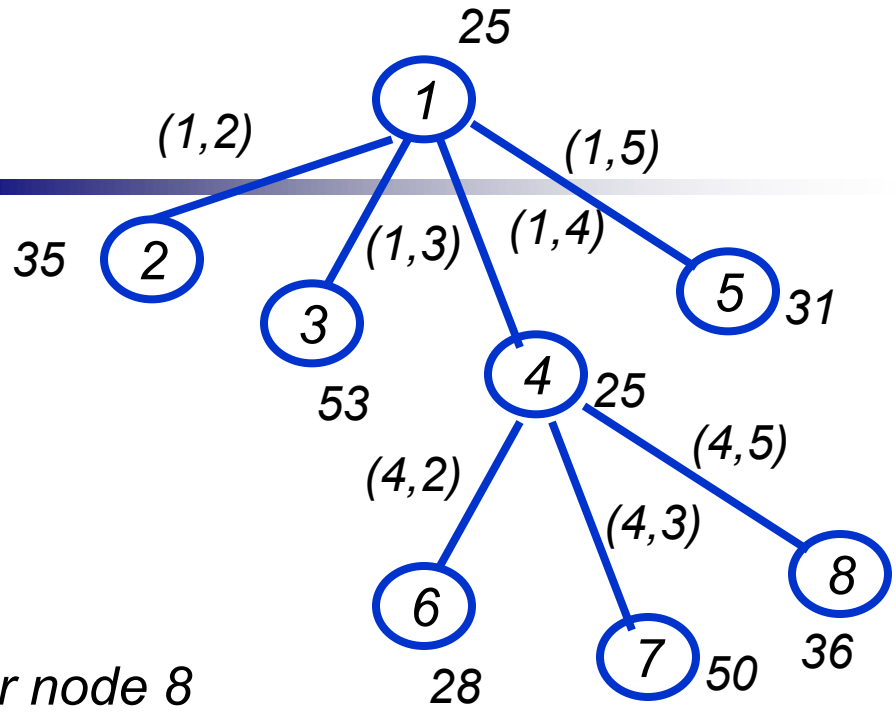
RCM for node 7

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

$\hat{C}(7) = \hat{C}(4) + A(4,3) + r$
$= 25 + 12 + 13 = 50$

RCM for node 4

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



RCM for node 8

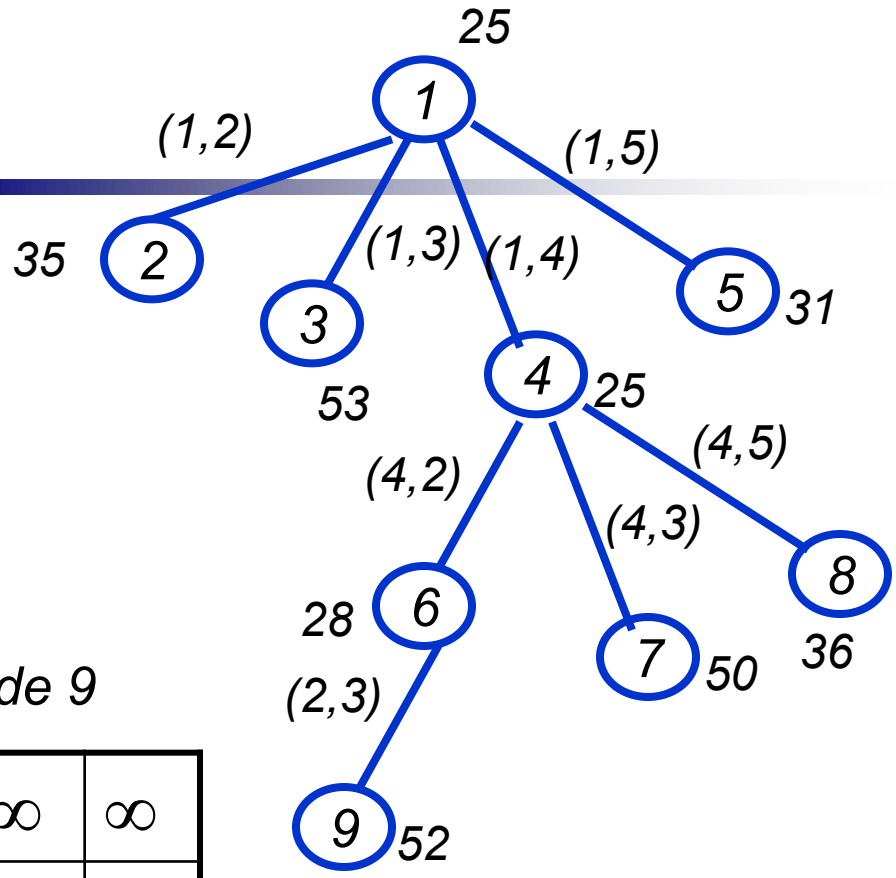
∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

$\hat{C}(8) = \hat{C}(4) + A(4,5) + r$
$= 25 + 0 + 11 = 36$

Node 6 becomes the next E-node

RCM for node 6

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



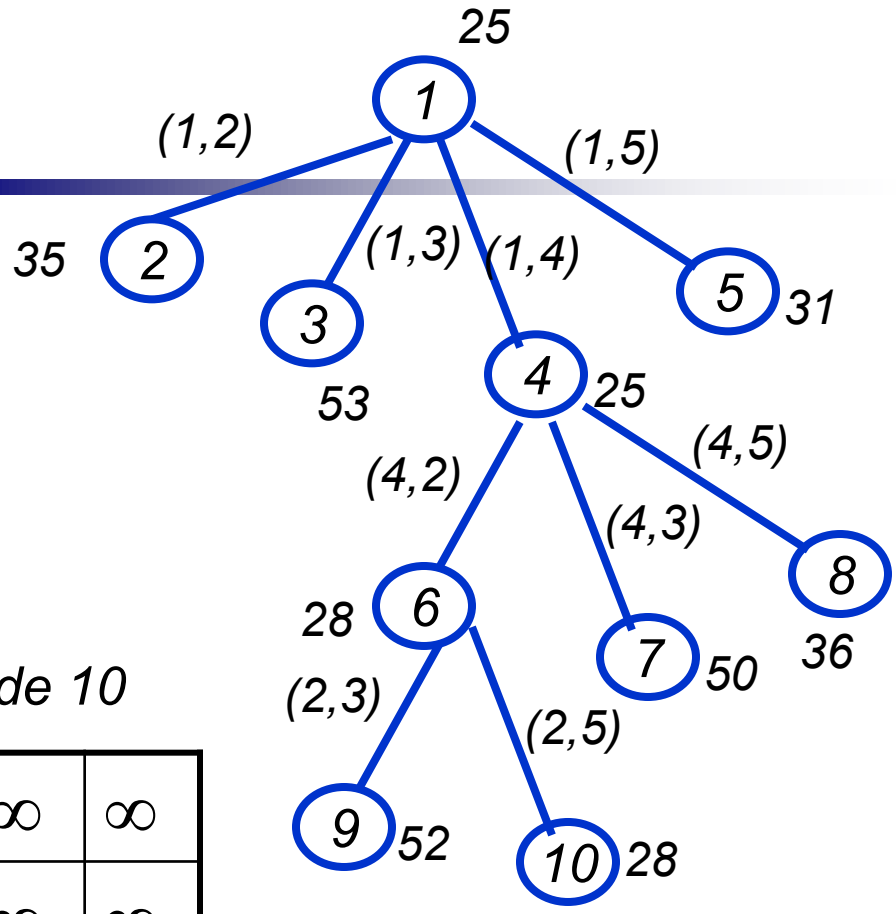
RCM for node 9

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

$\hat{C}(9) = \hat{C}(6) + A(2,3) + r$
$= 28 + 11 + 13 = 52$

RCM for node 6

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



RCM for node 10

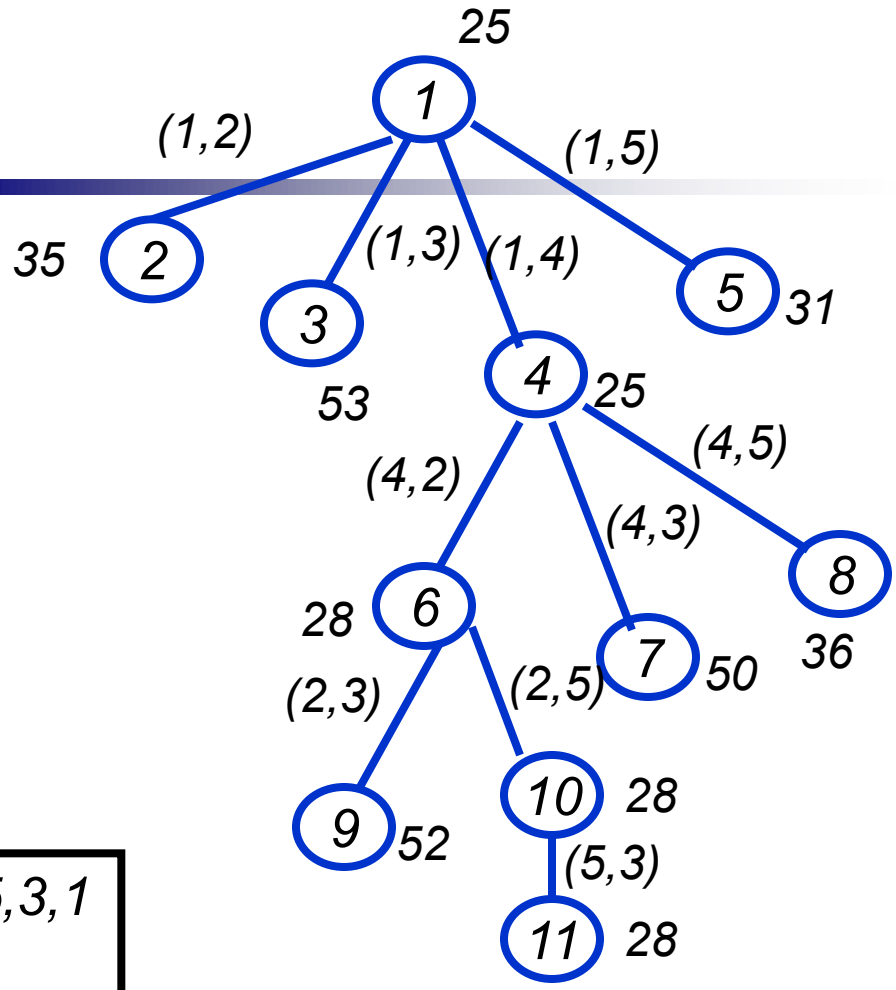
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

$\hat{C}(10) = \hat{C}(6) + A(2,5) + r$
$= 28 + 0 + 0 = 28$

Node 10 becomes the next E-node

RCM for node 11

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



11 is the solution node - path 1,4,2,5,3,1
upper = 28
For the next E-node 5 Ĉ(5) = 31 > upper
Hence, LCBB terminates

$\hat{C}(11) = \hat{C}(10) + A(5,3) + r$
$= 28 + 0 + 0 = 28$