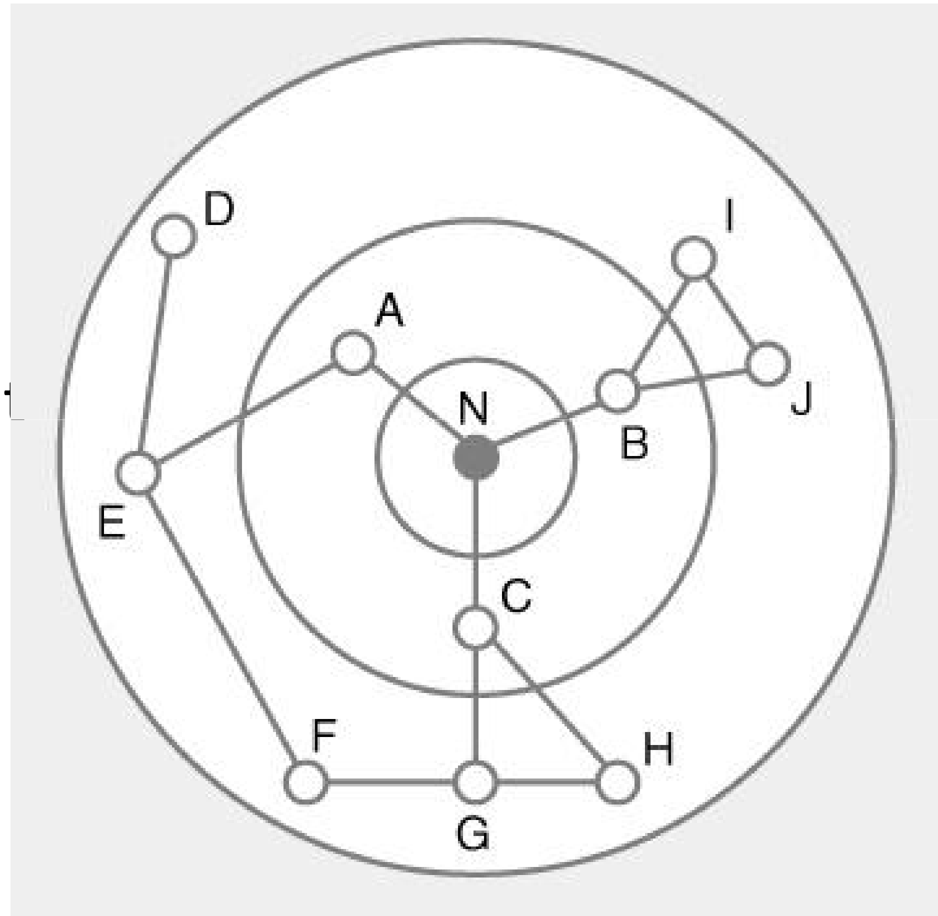
Backtracking

Graph Operations

- Traversal (search)
  - Visit each node in graph exactly once
  - Usually perform computation at each node
  - Two approaches
    - Breadth first search (BFS)
    - Depth first search (DFS)

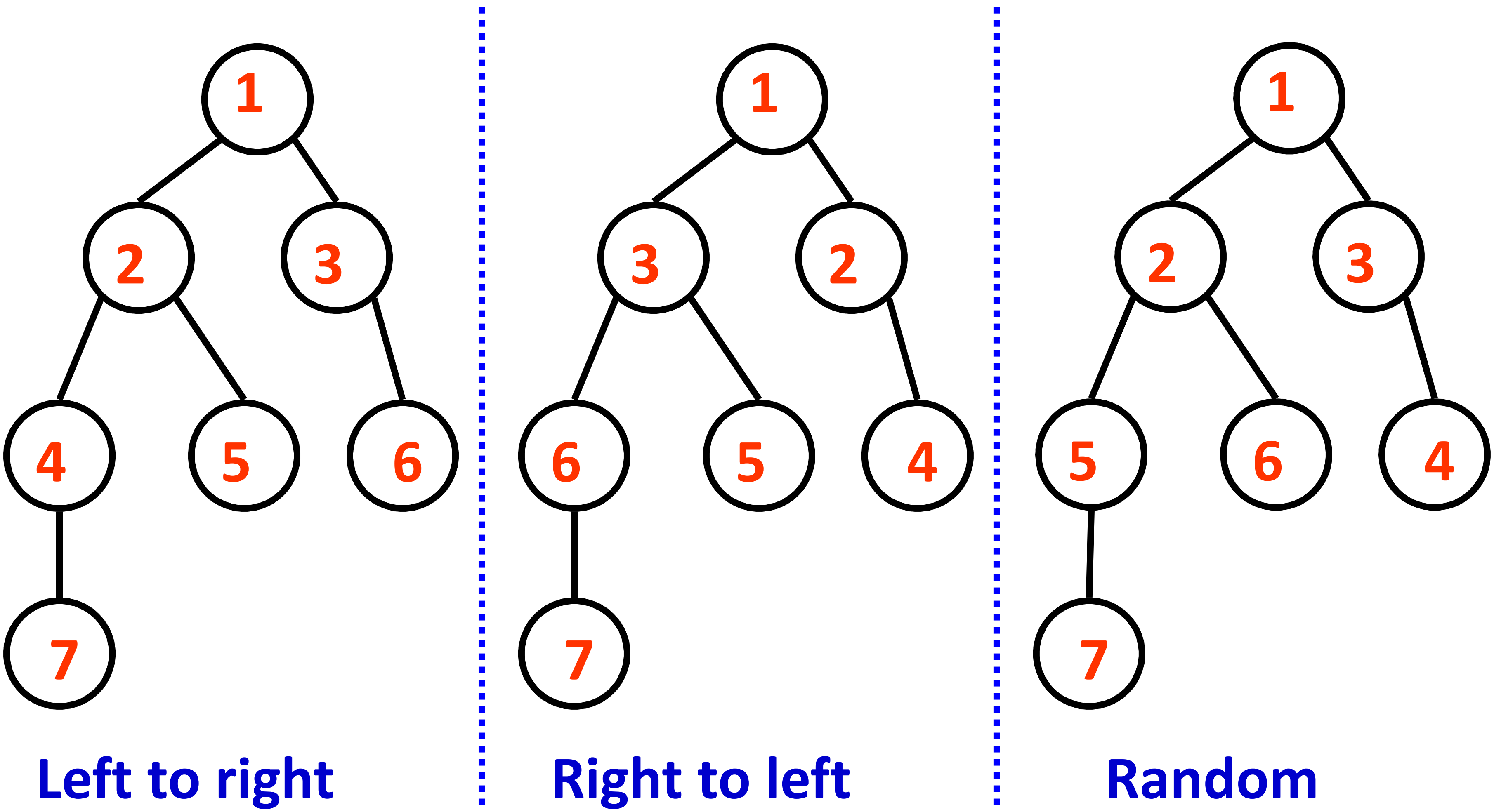
Breadth-first Search (BFS)

- Approach
  - Visit all neighbors of node first
  - View as series of expanding circles
  - Keep list of nodes to visit in queue
- Example traversal
  1) n
  2) a, c, b
  3) e, g, h, i, j
  4) d, f



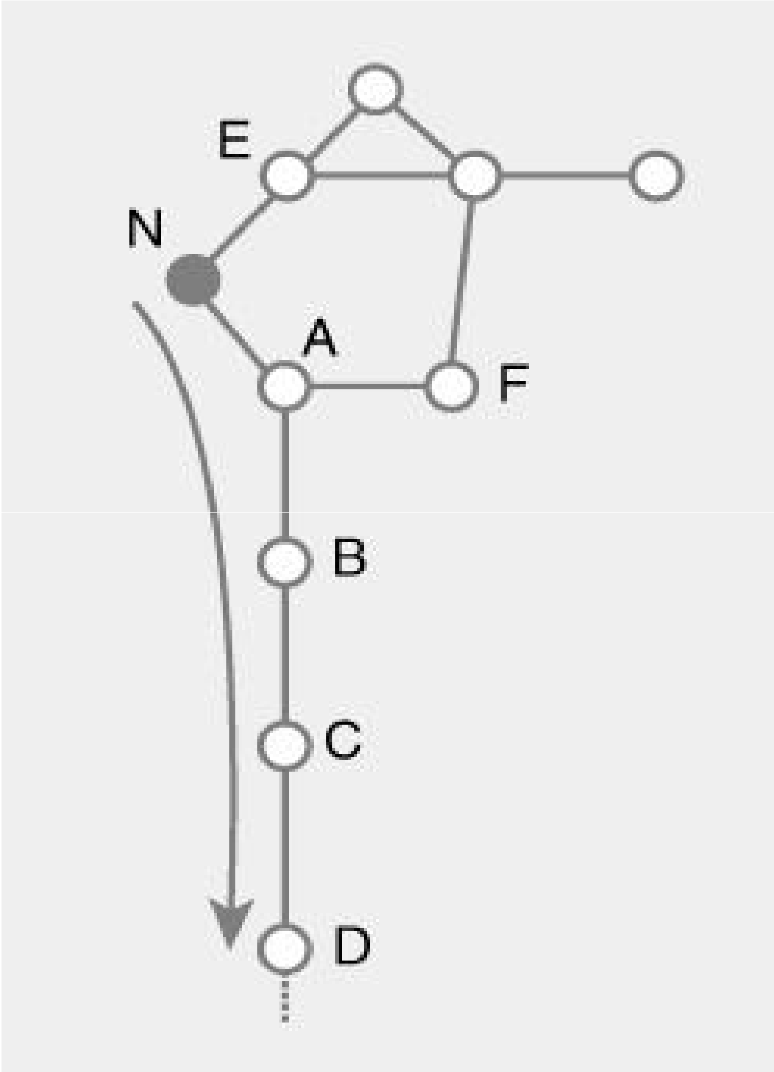
Breadth-first Search (BFS)

- Example traversals



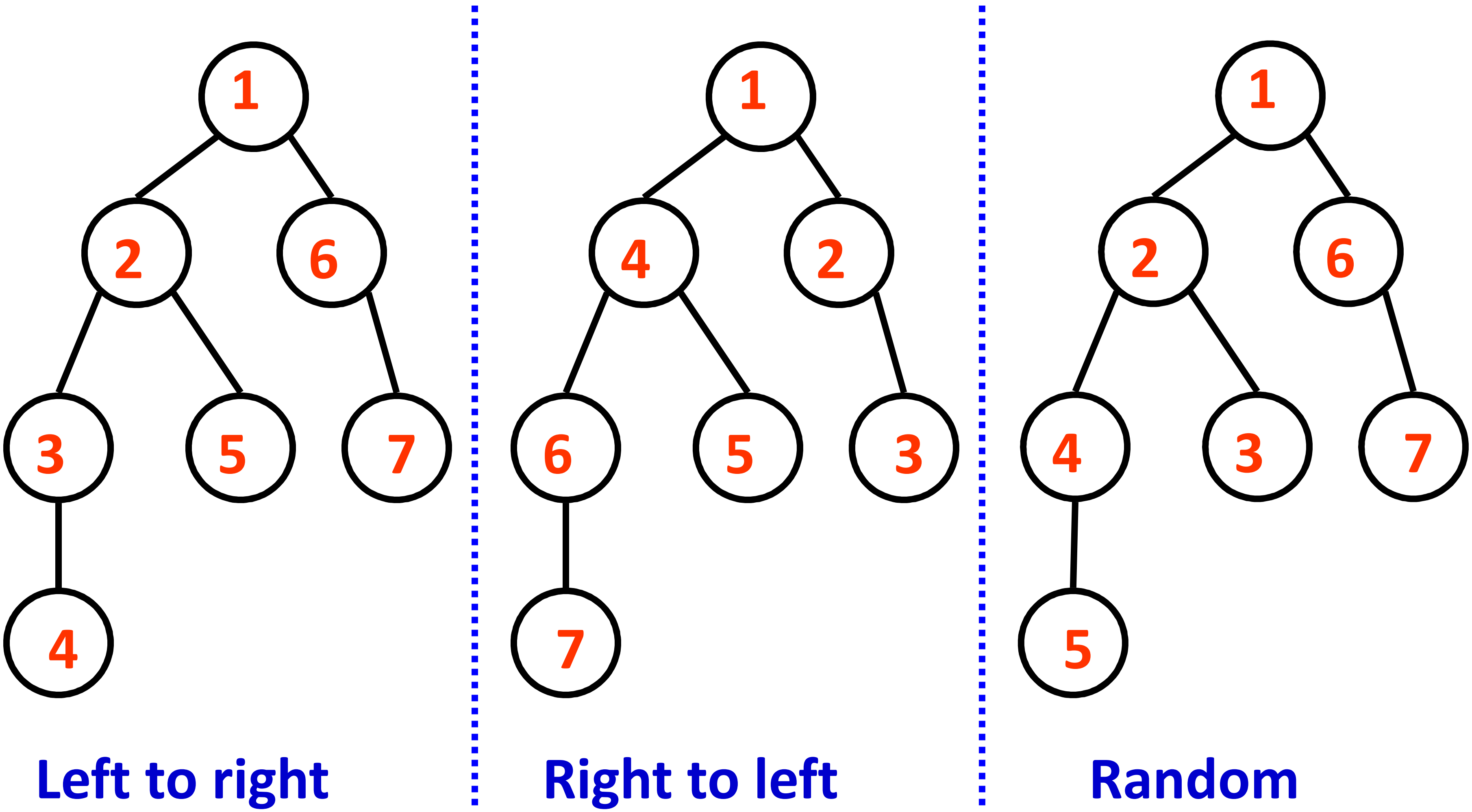
Depth-first Search (DFS)

- Approach
  - Visit all nodes on path first
  - Backtrack when path ends
  - Keep list of nodes to visit in a stack
- Example traversal
  1) n, a, b, c, d, …
  2) f …

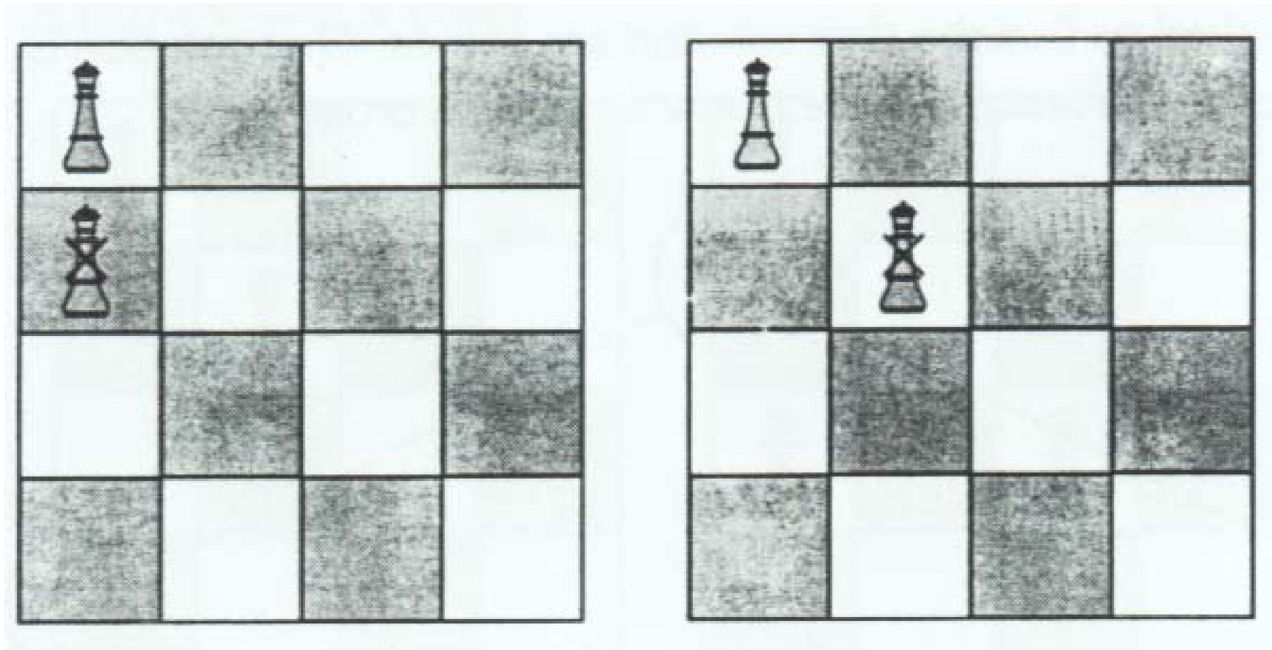


Depth-first Search (DFS)

- Example traversals



The 4 Queens Problem



The goal of this problem is to position n queens on nxn chessboard so that no two quens threathen each other. That is no two queens may be in the same row, column, or diagonal.

What is backtracking?

- It is a systematic search strategy of the state-space of combinatorial problems

- It is mainly used to solve problems which ask for finding elements of a set which satisfy some restrictions. Many problems which can be solved by backtracking have the following general form:

  " Find S subset of A_1 x A_2 x … x A_n (A_k – finite sets) such that each element s=(s_1,s_2,…,s_n) satisfy some restrictions"

What is backtracking?

Basic ideas:

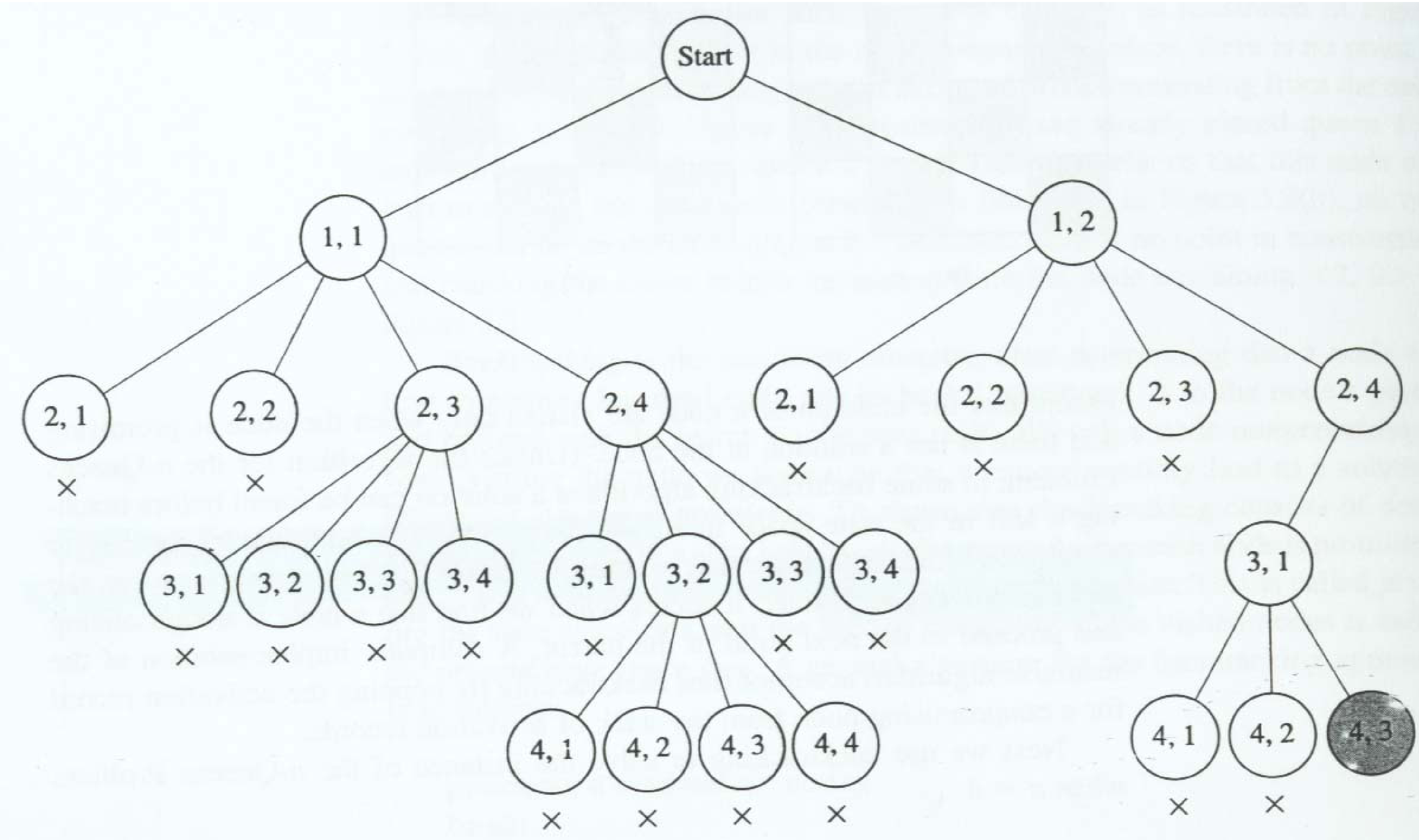
- each partial solution is evaluated in order to establish if it is **promising** (a promising solution could lead to a final solution while a **non-promising** one does not satisfy the partial restrictions induced by the problem restriction)
- if all possible values for a component do not lead to a promising partial solution then one come back to the previously component and try another value for it

- backtracking implicitly constructs a state space tree:
  - The root corresponds to an initial state (before the search for a solution begins)
  - An internal node corresponds to a promising partial solution
  - An external node (leaf) corresponds to either to a non-promising partial solution or to a final solution

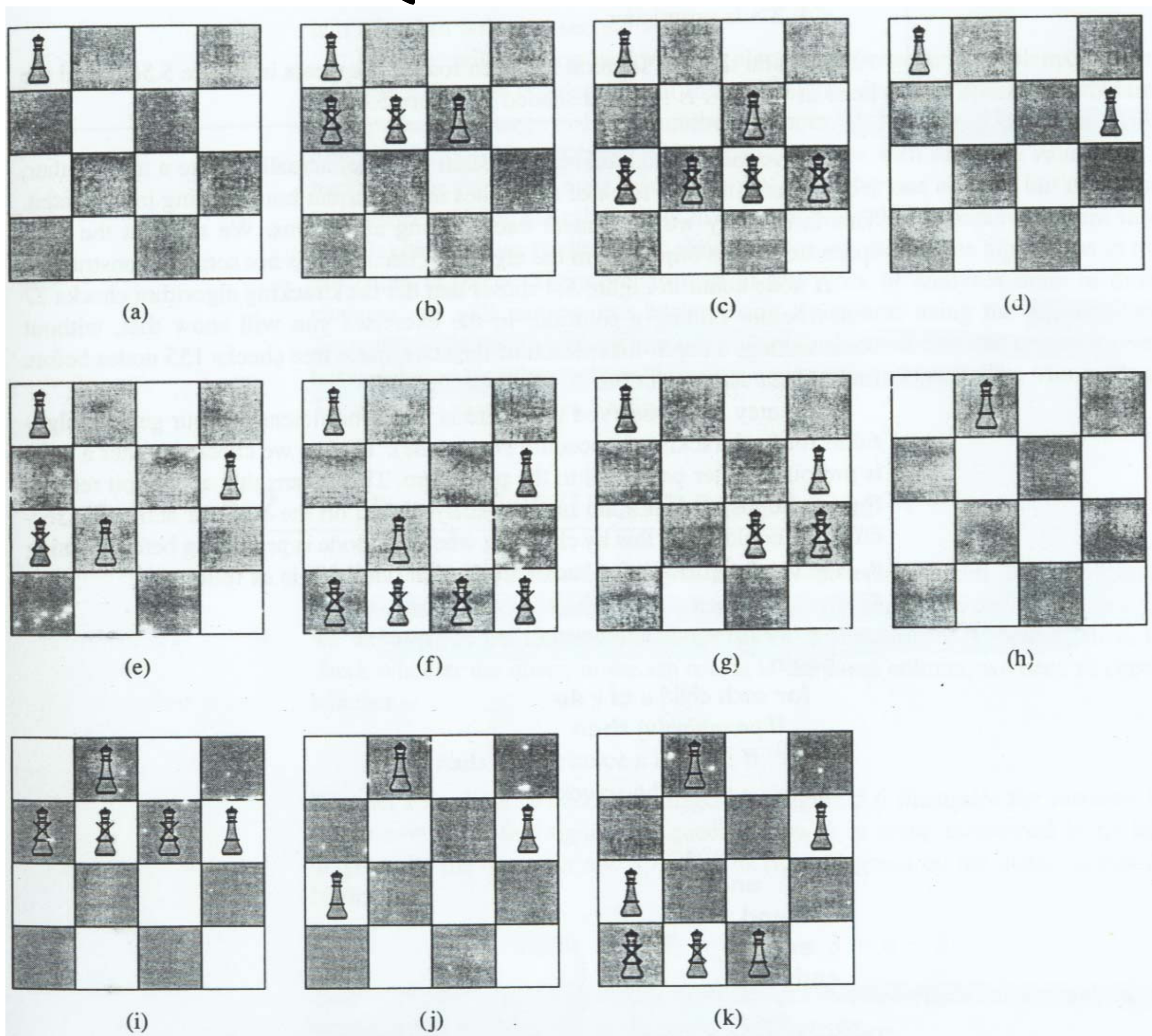
General algorithm for backtrack

```
Procedure checknode(v:node)
Begin
   if promising(v) then
        if there is a solution then
              write the solution
        else
               for each child u of v do
                    checknode(u)
               end
          end
    end
end
```

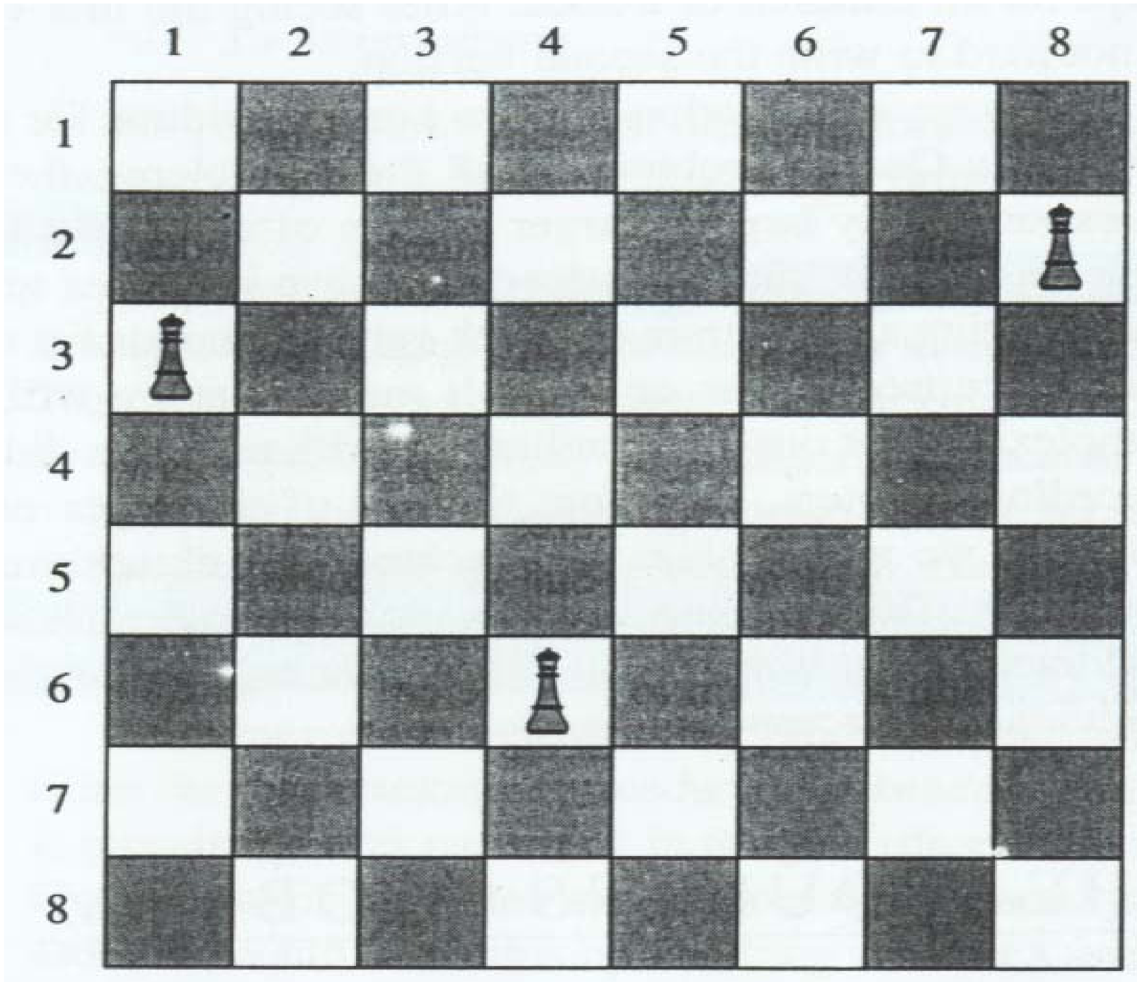
The 4 Queens Problem



The 4 Queens Problem



How to check the diagonal/column ?



How to check the diagonal/column ?

Let *col(i)* be the column where the queen in the *i*th row is located.

- Check column → *col(i) = col(k)*
- Check diagonal → *col(i) – col(k) = i-k* or *col(i) – col(k) = k – i*

Examples. In the figure, the queen in row 6 is being threatened in its left diagonal by the queen in row 3, and in its right diagonal by the queen in row 2.

col(6) – col(3)= 4 – 1= 3 = 6 – 3

col(6) – col(2)= 4 – 8= -4 =2 – 6

Backtracking algorithm for the *n* queens

```
Procedure queens(i:index);
Var j:index;
Begin
  if promising(i) then
      if i=n then
            write(col[1] through col[n])
      else
            for j:=1 to n do
                  col[i+1]:=j;
                  queens(i+1)
            end
      end
  end
End;
```

Backtracking algorithm for the *n* queens

```
function promising(i:index):boolean;
Var k:index;
Begin
   k:=1;
   promising:=true;
   while k<i and promising do
        if col[i]=col[k] or abs(col[i]-col[k])=i-k then
              promising:=false
        end
        k:=k+1
   end
End;
```

tingkat	i	promising	ket	j	aksi	col			
						1	**2**	**3**	**4**
0	0	TRUE		1	col[i+1]=j --> col[1]=1	1			
					queens(i+1)=queens(1)				
1	1	TRUE		1	col[i+1]=j --> col[2]=1	1	1		
					queens(i+1)=queens(2)				
2	2	FALSE	col[2]=col[1]		finish 2 back to 1				
1	1			2	col[i+1]=j --> col[2]=2	1	2		
					queens(i+1)=queens(2)				
2	2	FALSE	abs(col[2]-col[1])=2-1		finish 2 back to 1				
1	1			3	col[i+1]=j --> col[2]=3	1	3		
					queens(i+1)=queens(2)				
2	2	TRUE		1	col[i+1]=j --> col[3]=1	1	3	1	
					queens(i+1)=queens(3)				
3	3	FALSE	col[3]=col[1]		finish 3 back to 2				
2	2			2	col[i+1]=j --> col[3]=2	1	3	2	
					queens(i+1)=queens(3)				
3	3	FALSE	abs(col[3]-col[2])=3-2		finish 3 back to 2				
2	2			3	col[i+1]=j --> col[3]=3	1	3	3	
					queens(i+1)=queens(3)				
3	3	FALSE	col[3]=col[2]		finish 3 back to 2				
2	2			4	col[i+1]=j --> col[3]=4	1	3	4	
					queens(i+1)=queens(3)				
3	3	FALSE	abs(col[3]-col[2])=3-2		finish 3 back to 2				
dst									

Backtracking algorithm for the *n* queens

- Top level call to *queens* is

```
queens(0);
```

- Total number of nodes (lower bound):

$$1 + n + n^2 + n^3 + \ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

- Upper bound ?

The sum-of-subset Problem

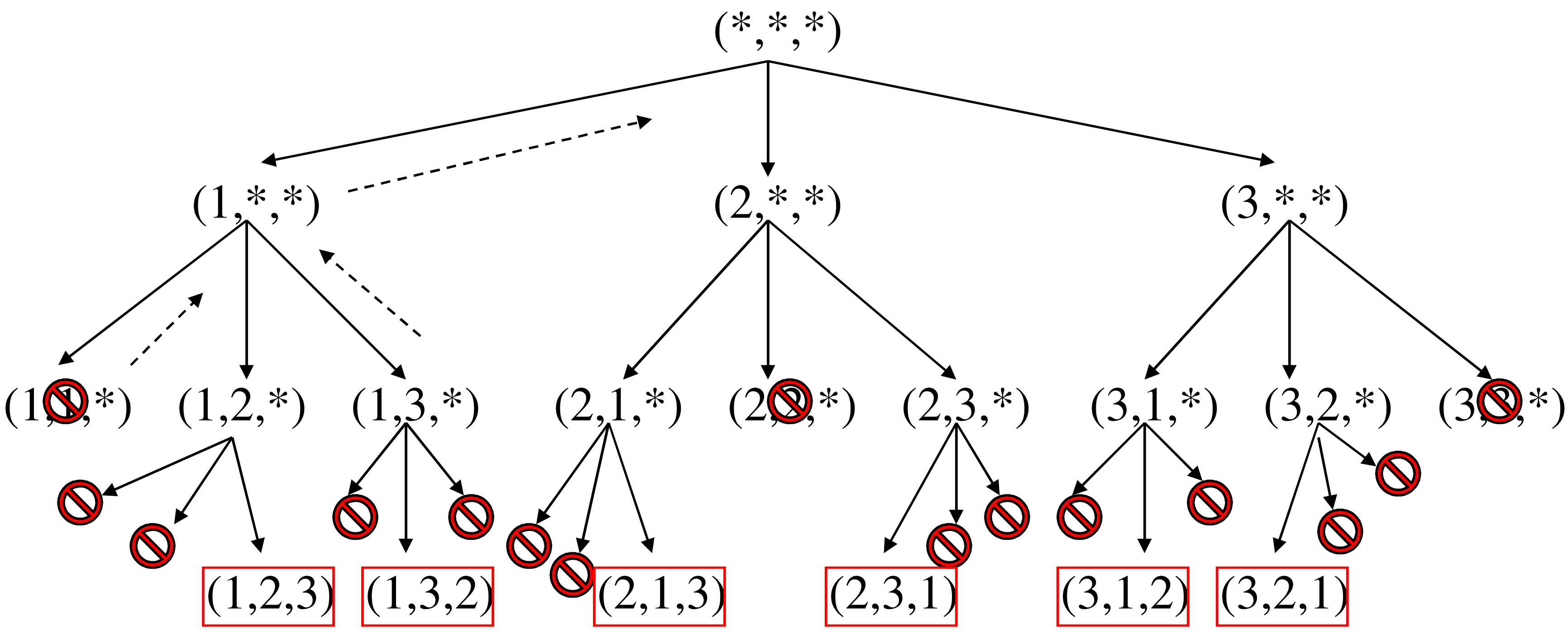
- Recall the knapsack problem
- Goal : to find all the subset of integers that sum to W
- Example :

  w1 = 3, w2 = 4, w3 = 5, w4 = 6
- A node at the i-th level is non-promising if :

  weight + w_{i+1} > w or

  weight + total < W

Permutation Generation



Another example

- Graph coloring
- Hamiltonian problem
- Knapsack problem