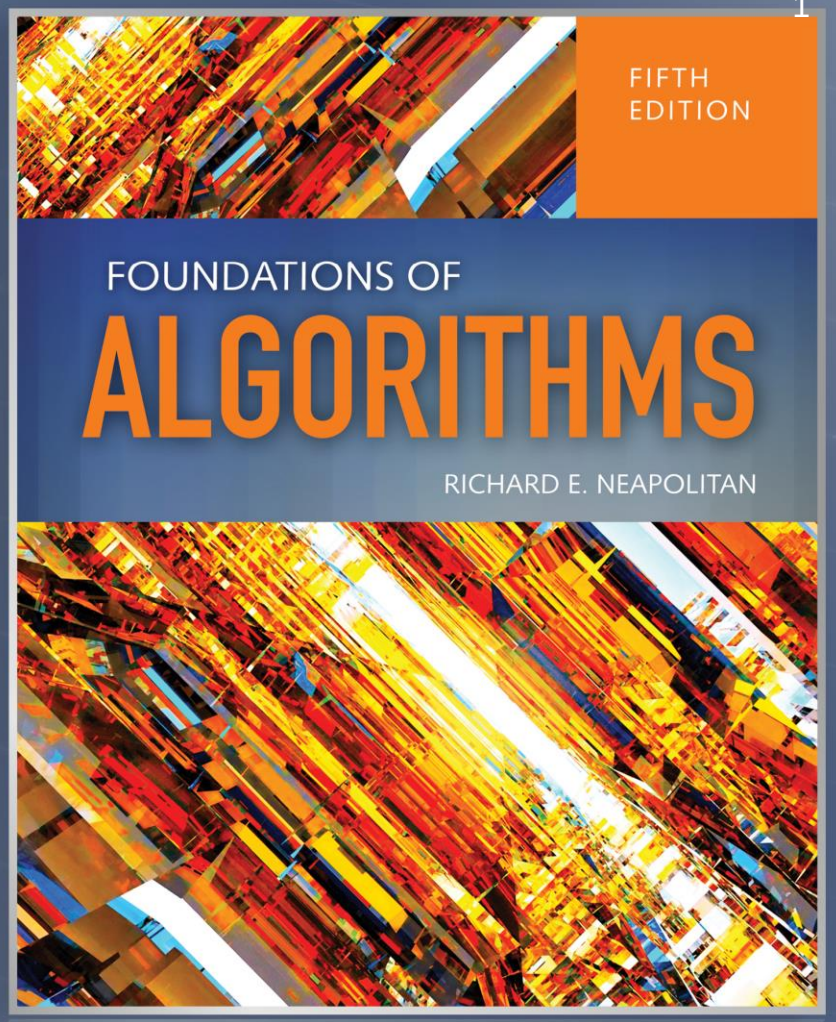
Backtracking

Chapter 5



Objectives

- Describe the backtrack programming technique
- Determine when the backtracking technique is an appropriate approach to solving a problem
- Define a **state space tree** for a given problem
- Define when a node in a state space tree for a given problem is **promising/non-promising**
- Create an algorithm to **prune** a state space tree
- Create an algorithm to apply the backtracking technique to solve a given problem

Finding Your Way Thru a Maze

- Follow a path until a dead end is reached
- Go back until reaching a fork
- Pursue another path
- Suppose there were signs indicating path leads to dead end?
- Sign positioned near beginning of path – time savings enormous
- Sign positioned near end of path – very little time saved

Sudoku

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

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Solving Sudoku

- Solving Sudoku puzzles involves a form of an exhaustive search of possible configurations.
- However, exploiting constraints to rule out certain possibilities for certain positions enables us to prune the search to the point that people can solve Sudoku by hand.
- Backtracking is the key to implementing exhaustive search programs correctly and efficiently.

Backtracking

- **Backtracking** is a systematic method to iterate through all possible configurations of a search space. It is a general algorithm which must be customized for each application.
- We model our solution as a vector a = (a1; a2; …; an), where each element ai is selected from a finite ordered set Si.
- Such a vector might represent an arrangement where ai contains the ith element of the permutation. Or the vector might represent a given subset S, where ai is true if and only if the ith element of the universe is in S.
- Can be used to solve NP-Complete problems such as 0-1 Knapsack **more efficiently**

Backtracking vs Dynamic Programming

- Dynamic Programming – **subsets of a solution are generated**
- Backtracking – Technique **for deciding that some subsets need not be generated**
- Backtracking is used to solve problems in which a **sequence of objects is chosen from a specified set** so that the sequence satisfies some **criterion.**
- Efficient for many large instances of a problem (but not all)

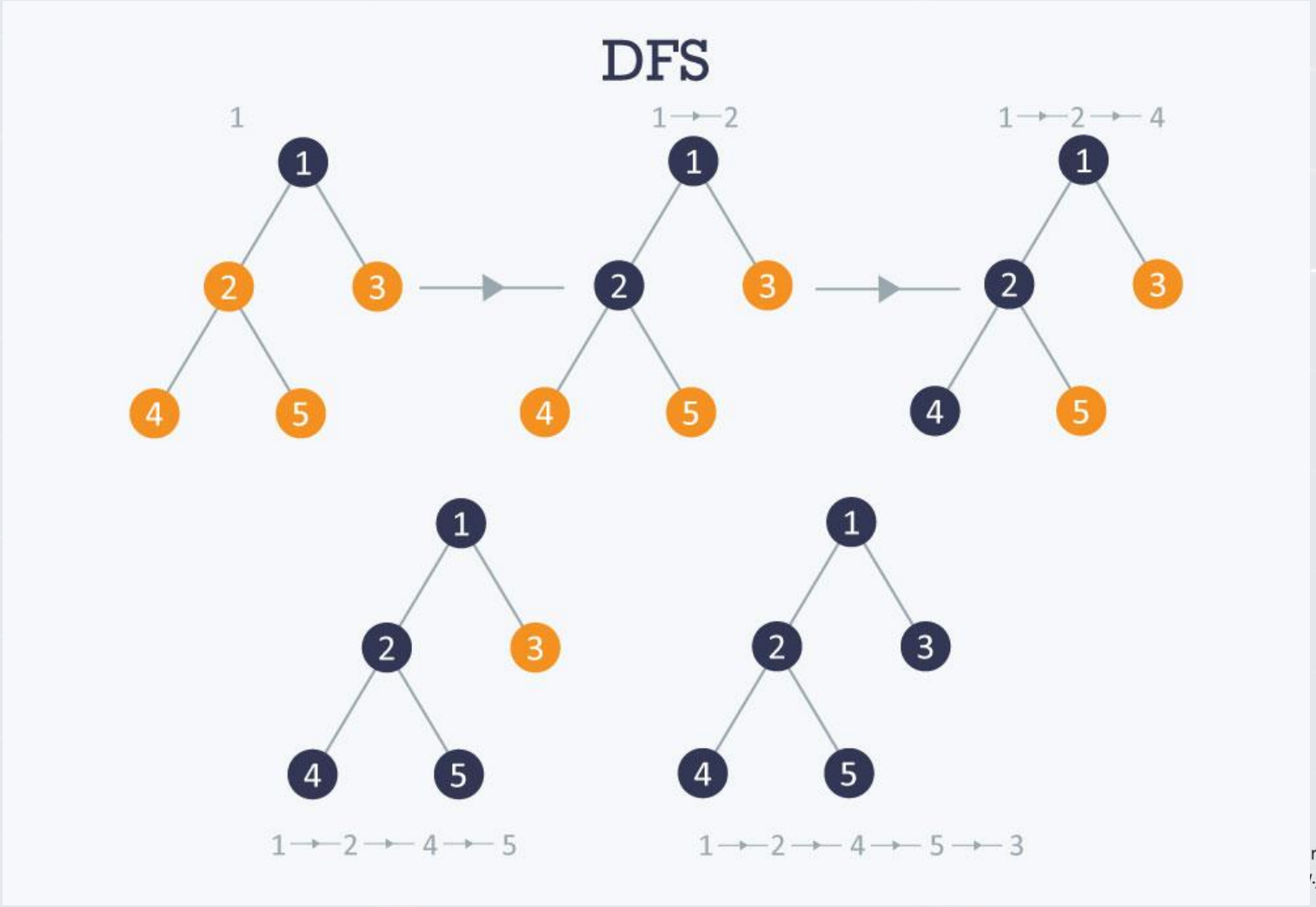
Procedure called by passing root at the top level

Backtracking is a modified **depth-first search** of a rooted tree

```
void depth_first_tree_search(node v)
{
        node u;
        visit v

        for( each child u of v)
                depth_first_tree_search(u);
}
```

DFS Example

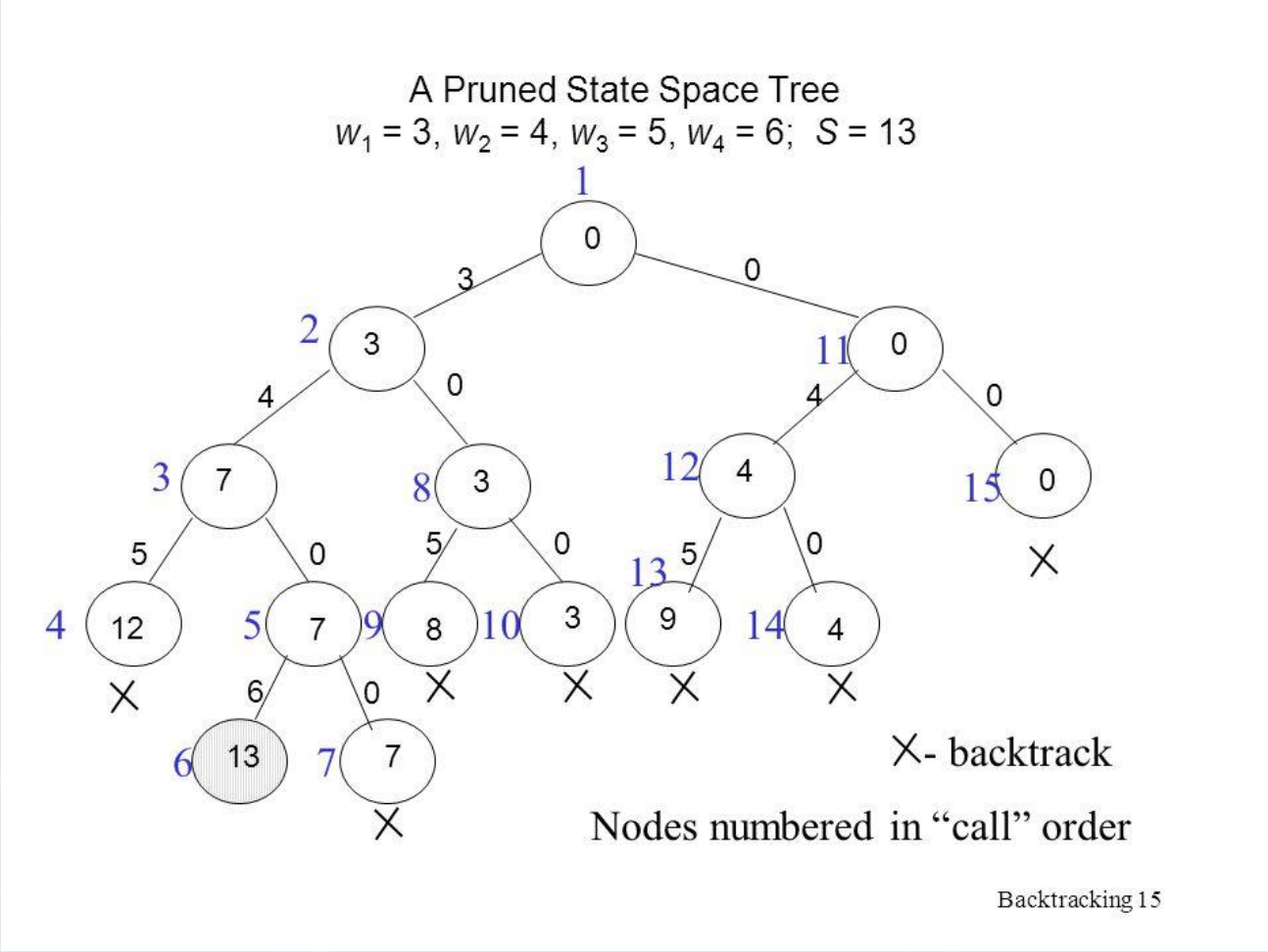


Backtracking Procedure

- After determining a node cannot lead to a solution, backtrack to the node's parent and proceed with the search on the next child
- **Non-promising node**: when the node is visited, it is determined the node cannot lead to a solution
- **Promising node**: may lead to a solution
- **Backtracking**
  - DFS of state space tree
  - **Pruning state space tree**: if a node is determined to be non-promising, back track to its parent

Backtracking Procedure

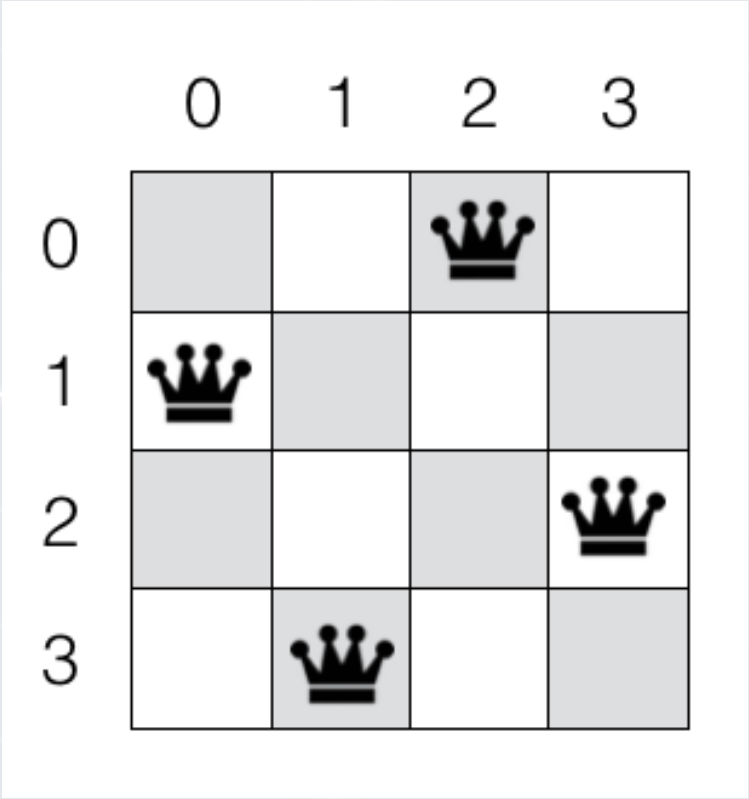
- Pruned State Space Tree: sub-tree consisting of visited nodes



N-Queen Problem

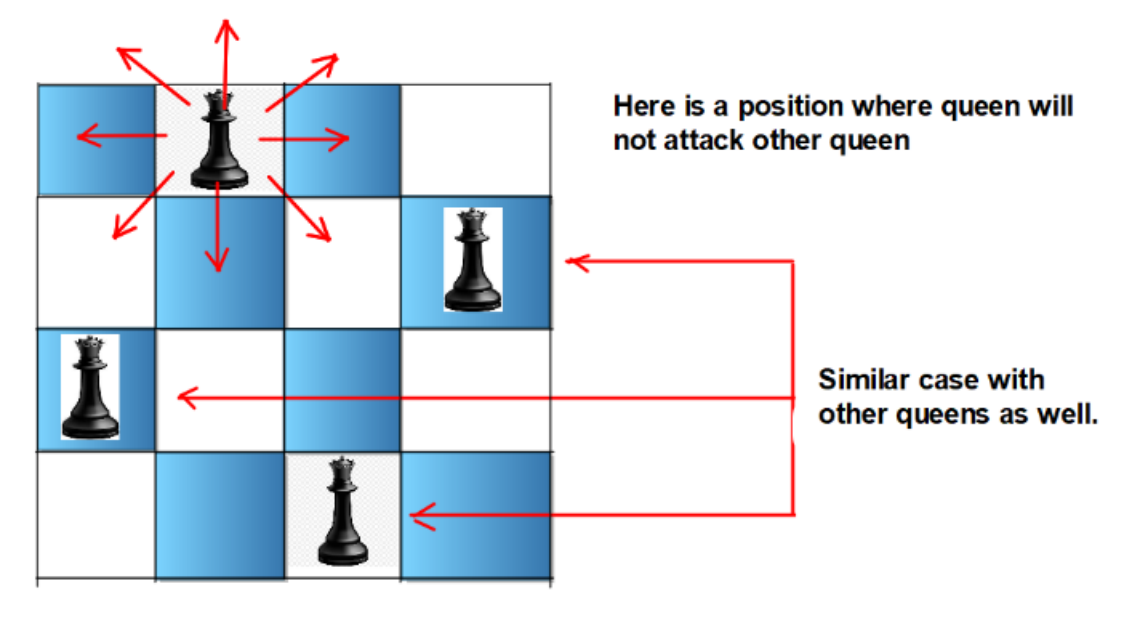
- Goal: **position n queens on a n x n board such that no two queens threaten each other**
  - No two queens may be in the same row, column, or diagonal
- **Sequence**: n positions where queens are placed
- **Set**: n^2 positions on the board
- **Criterion**: no two queens threaten each other

e.g. 4-Queen Problem Solution

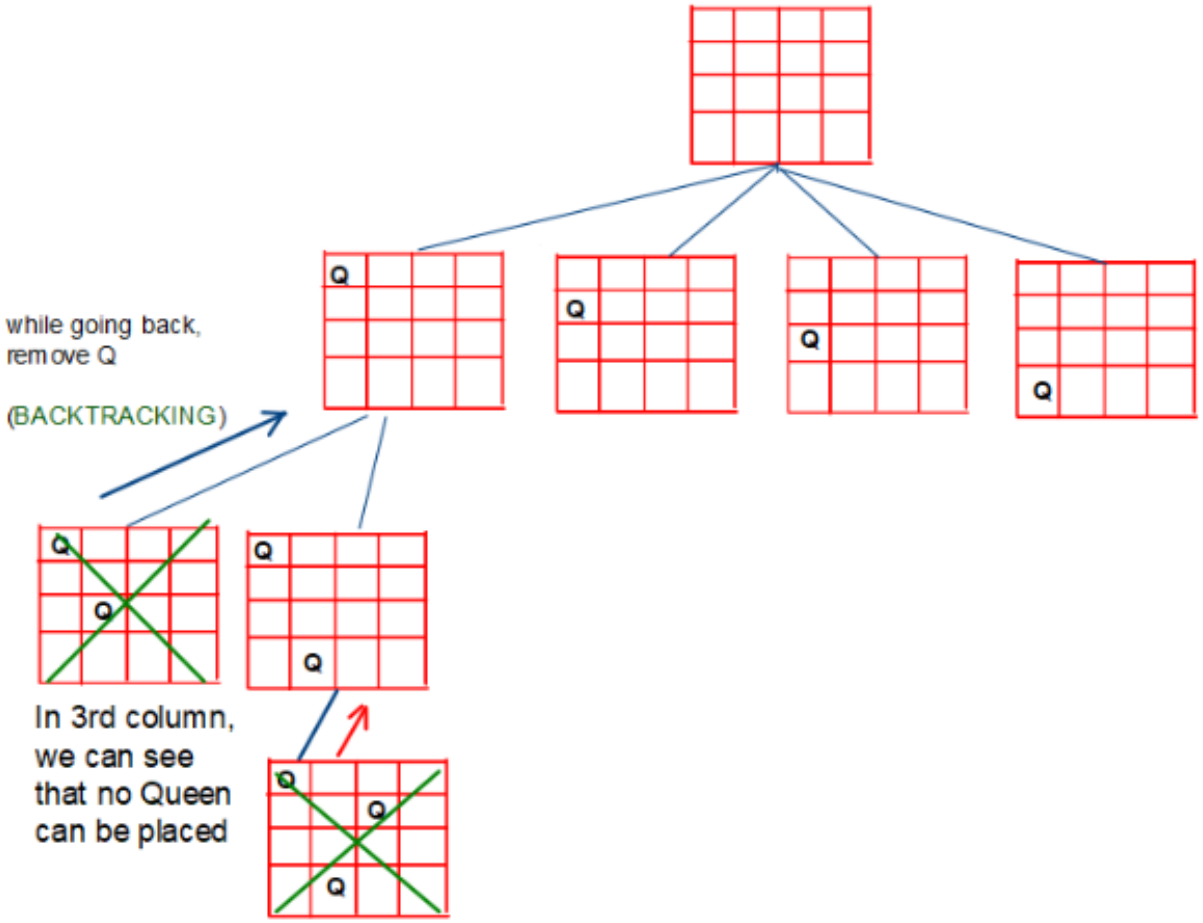


4-Queen Problem

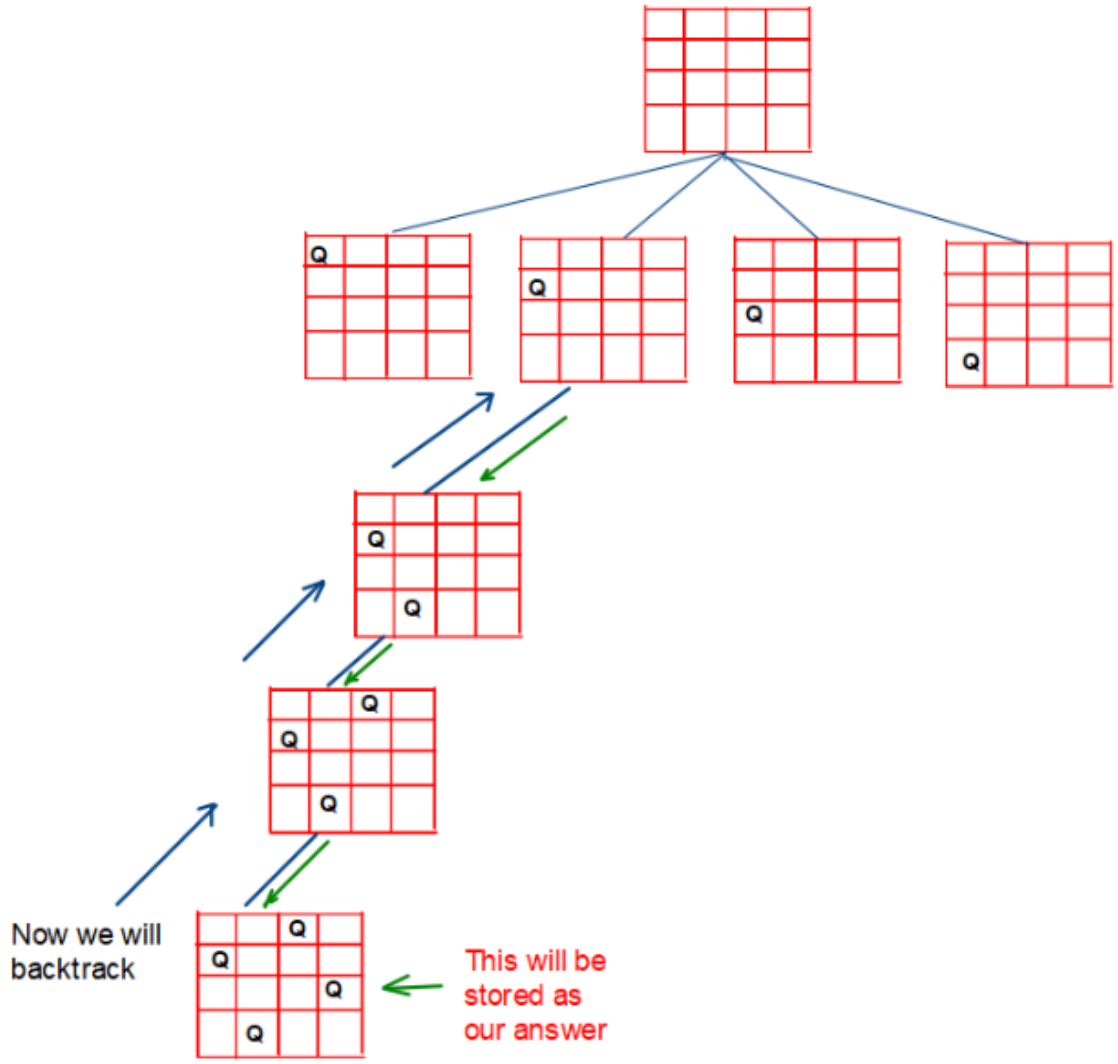
- Assign each queen a different row
- Check which column combinations yield solutions
- Each queen can be in any one of four columns: 4 x 4 x 4 x 4 = 256



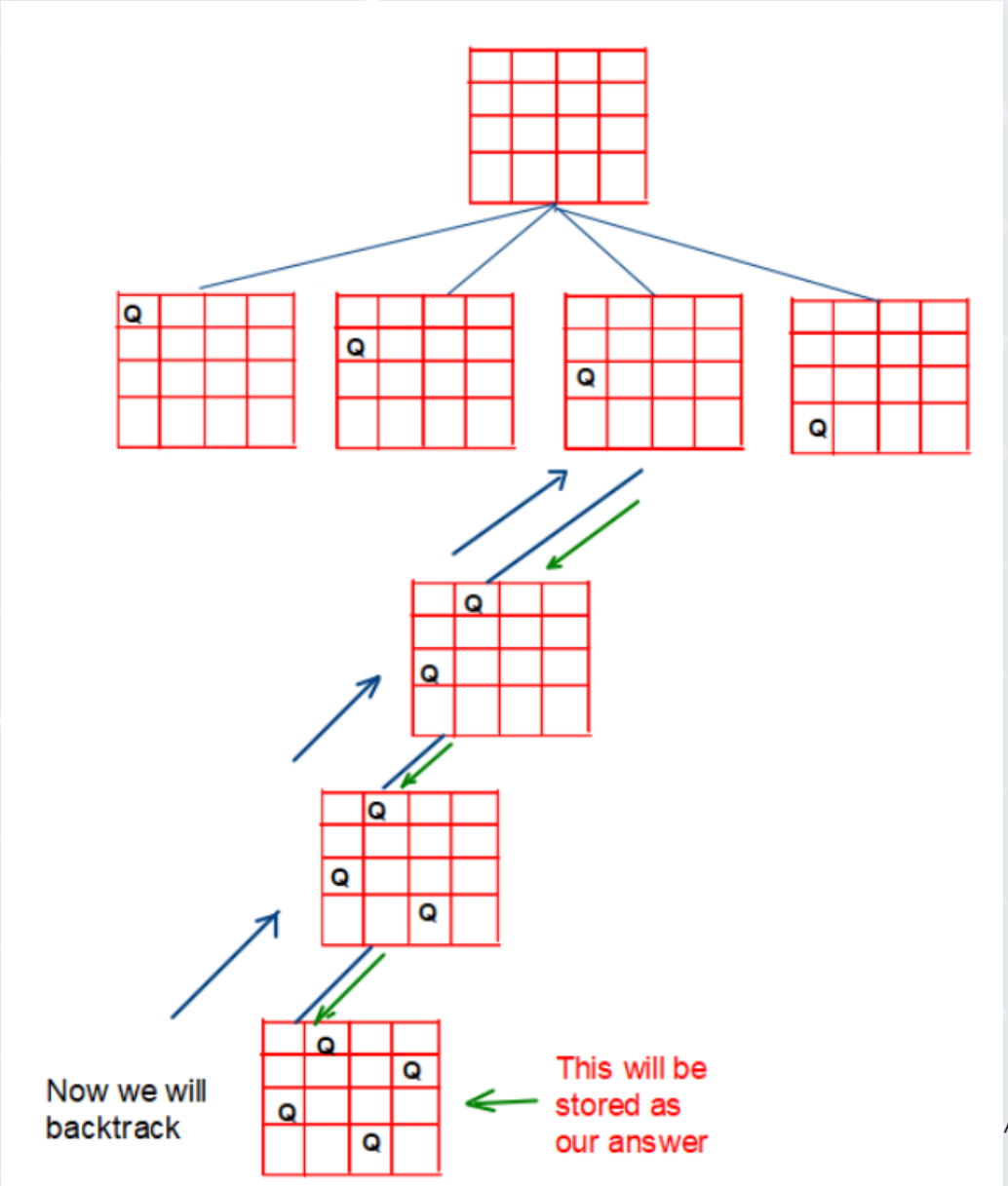
Ist position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 3rd column, the Queen will be killed at all possible positions of row.



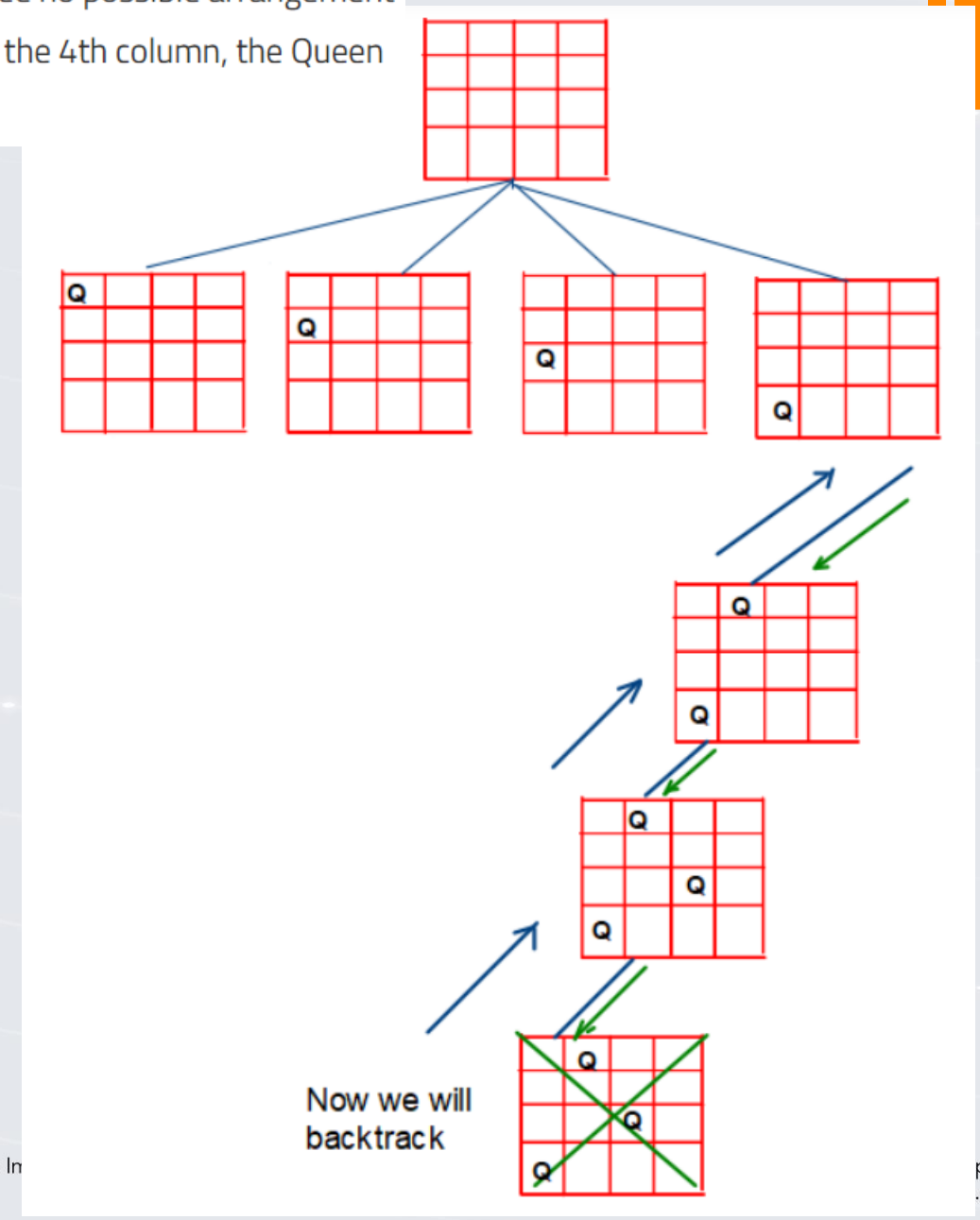
2nd position: One of the correct possible arrangements is found. So we will store it as our answer.



3rd position: One of the correct possible arrangements is found. So we will store it as our answer.



4th position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 4th column, the Queen will be killed at all possible positions of row.



Construct State-Space Tree – Candidate Solutions

- Root – start node
- Column choices for first queen stored at level-1 nodes
- Column choices for second queen stored at level-2 nodes
- Etc.
- Path from root to a leaf is candidate solution
- Check each candidate solution in sequence starting with left-most path

Pruning

- DFS of state space tree
- Check to see if each node is **promising**
- **If a node is non-promising, backtrack to node's parent**
- **Pruned state space tree** – subtree consisting of visited nodes
- Promising function – application dependent
- Promising function n-queen problem: returns false if a node and any of the node's ancestors place queens in the same column or diagonal

A general algorithm for Backtracking


```
void checknode (node v)
{
  node u;

  if (promising(v))
     if (there is a solution at v)
        write the solution;
     else
        for (each child u of v)
           checknode(u);
}
```

- The root of SST is passed to ***checknode*** at the top level
- Check node – promising ? Yes → solution
- No solution → visit children of the node

- Funtion ***promising()*** is different in each application of backtracking
- **Promising function for the algorithm**

checknode()

```
void checknode (node v)
{
  node u;

  if (promising(v))
     if (there is a solution at v)
        write the solution;
     else
        for (each child u of v)
           checknode(u);
}
```

- **Backtracking algorithm for the n-Queens Problem**
- State space tree implicit – **tree not actually created**
  - Values in the current branch under investigation kept track of
- Algorithm **inefficient**
- Checks node promising after passing it to the procedure
- Activation records checked and pushed onto the stack for non-promising nodes

expand()

- **Improves efficiency**
- Check to see if node is promising before passing the node

```
void expand(node v)
{
  node u;

  for (each child u of v)
     if (promising(u))
        if (there is a solution at u)
           write the solution;
        else
           expand(u);
}
```

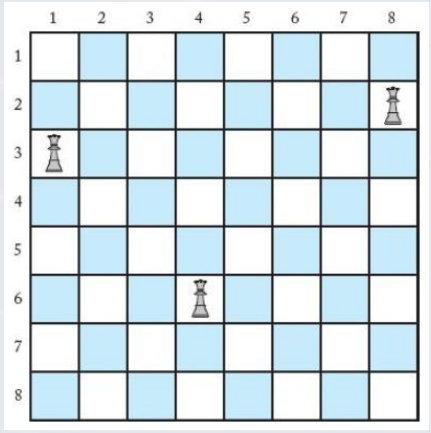
```
void checknode (node v)
{
  node u;

  if (promising(v))
     if (there is a solution at v)
        write the solution;
     else
        for (each child u of v)
           checknode(u);
}
```

Algorithm 5.1 Backtracking Algorithm for N-Queens Problem

- All solutions to N-Queens problem
- The promising function must check whether two queens are in the same column or diagonal.
- Let **col(i)** be the column where the queen in the ith row is located

- Promising function:
  - **2 queens on the same row?**
    col(i) == col(k)
  - **2 queens on the same diagonal?**
    col(i) – col(k) == i – k
    || col(i) – col(k) == k - i



col(6) – col(3) = 4 – 1 = 3 = 6 – 3

col(6) – col(2) = 4 – 8 = -4 = 2 - 6

Algorithm 5.1 Backtracking Algorithm for N-Queens Problem


```
void queens (index i)
{
  index j;

  if (promising(i))
     if (i == n)
        cout << col[1] through col [n];
     else
        for (j = 1; j <= n; j++){      // See if queen in
           col[i + 1] = j;             // (i + 1)st row can be
           queens(i + 1);              // positioned in each of
                                       // the n columns.
        }
}
bool promising (index i)
{
   index k;
   bool switch;

   k = 1;
   switch = true;                  // Check if any queen threatens
   while (k < i &&  switch){       // queen in the ith row.
      if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
         switch = false;
      k++;
   }
   return switch;
}
```

If **n** and **col** are defined as global variables, then the top-level call to **queens()** would be

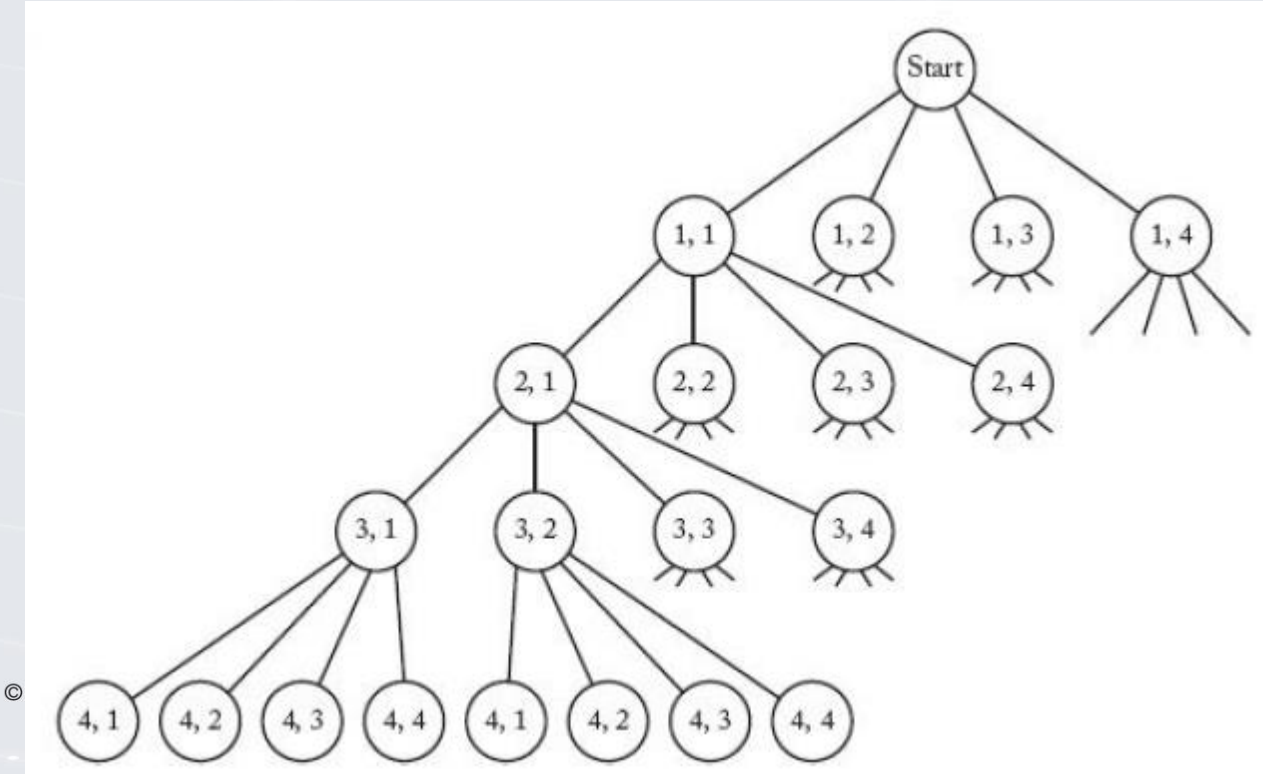
queens (0);

How many solutions does this algorithm produce?

Analysis of queens theoretically difficult

- Upper bound on the number of nodes checked in the pruned state space tree by counting the number of nodes in the entire state space tree:
  - 1 node level 0
  - n nodes level 1
  - n^2 nodes level 2

    . . .
  - N^n nodes level n



Total Number of nodes

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

This equality is obtained in Example A.4 in Appendix A. For the instance in which $n = 8$, the state space tree contains

$$\frac{8^{8+1} - 1}{8 - 1} = 19,173,961 \text{ nodes.}$$

This analysis is of limited value because the whole purpose of backtracking is to avoid checking many of these nodes.

Backtracking savings by executing code and counting nodes checked

- Algorithm1 – DFS of state space tree without backtracking
- Algorithm2 – checks no two queens in the same row or same column

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

*Entries indicate numbers of checks required to find all solutions.

†Algorithm 1 does a depth-first search of the state space tree without backtracking.

‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.

Sum-of-Subsets Problem

- Let S = $\{s_1 , s_2 , \ldots , s_n\}$
- Let W be a positive integer
- Find every S' $\subseteq$ S such that

$$\sum_{s \in S'} s = W$$

Example

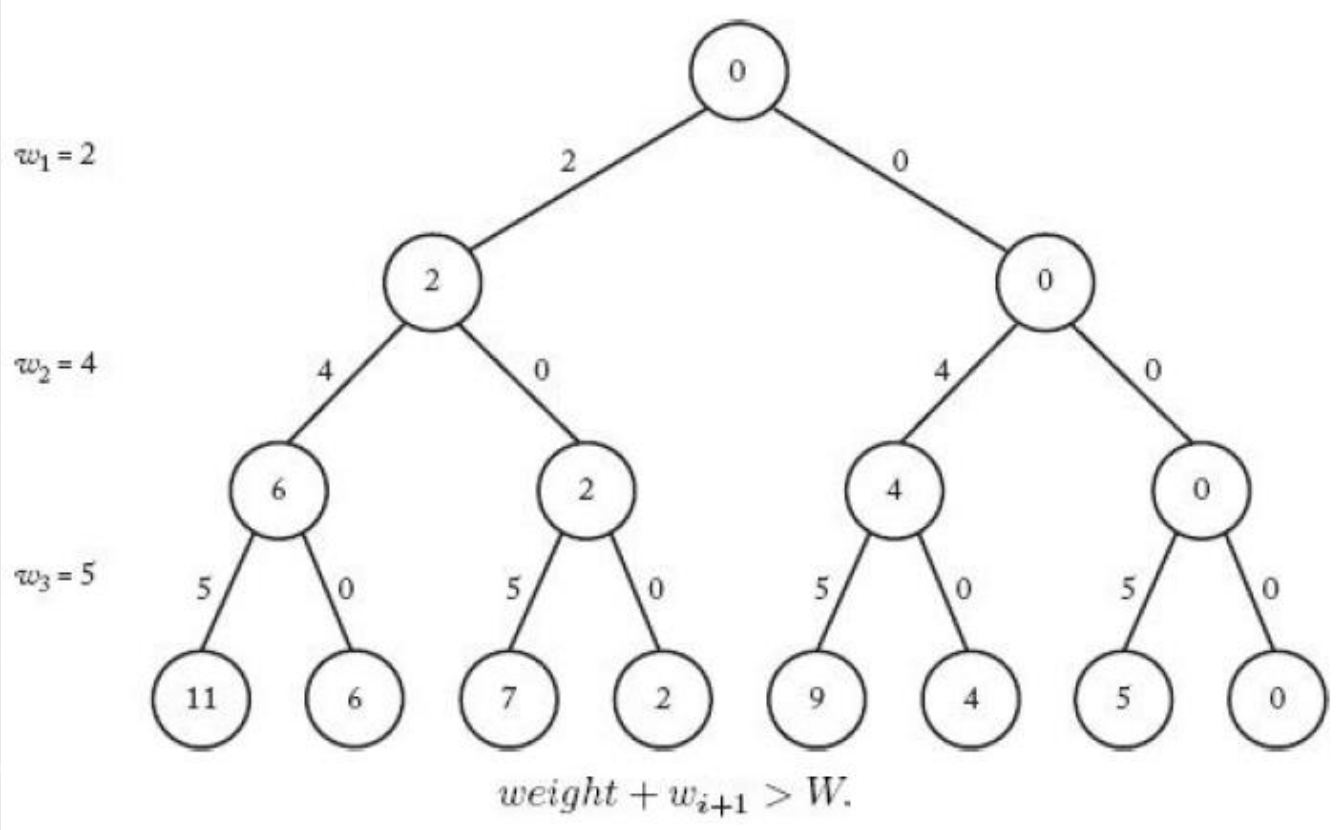
- S = {$w_1 = 5$, $w_2 = 6$, $w_3 = 10$, $w_4 = 11$, $w_5 = 16$} and W = **21**

- Solutions:
  - {w_1 , w_2 , w_3 } : 5 + 6 + 10 = **21**
  - {w_1 , w_5 } : 5 + 16 = **21**
  - {w_3 , w_4 }: 10 + 11 = **21**

the solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.

Example 5.3

- n = 3
- **W = 6**
- $w_1 = 2$
- $w_2 = 4$
- $w_3 = 5$



Prune the Tree

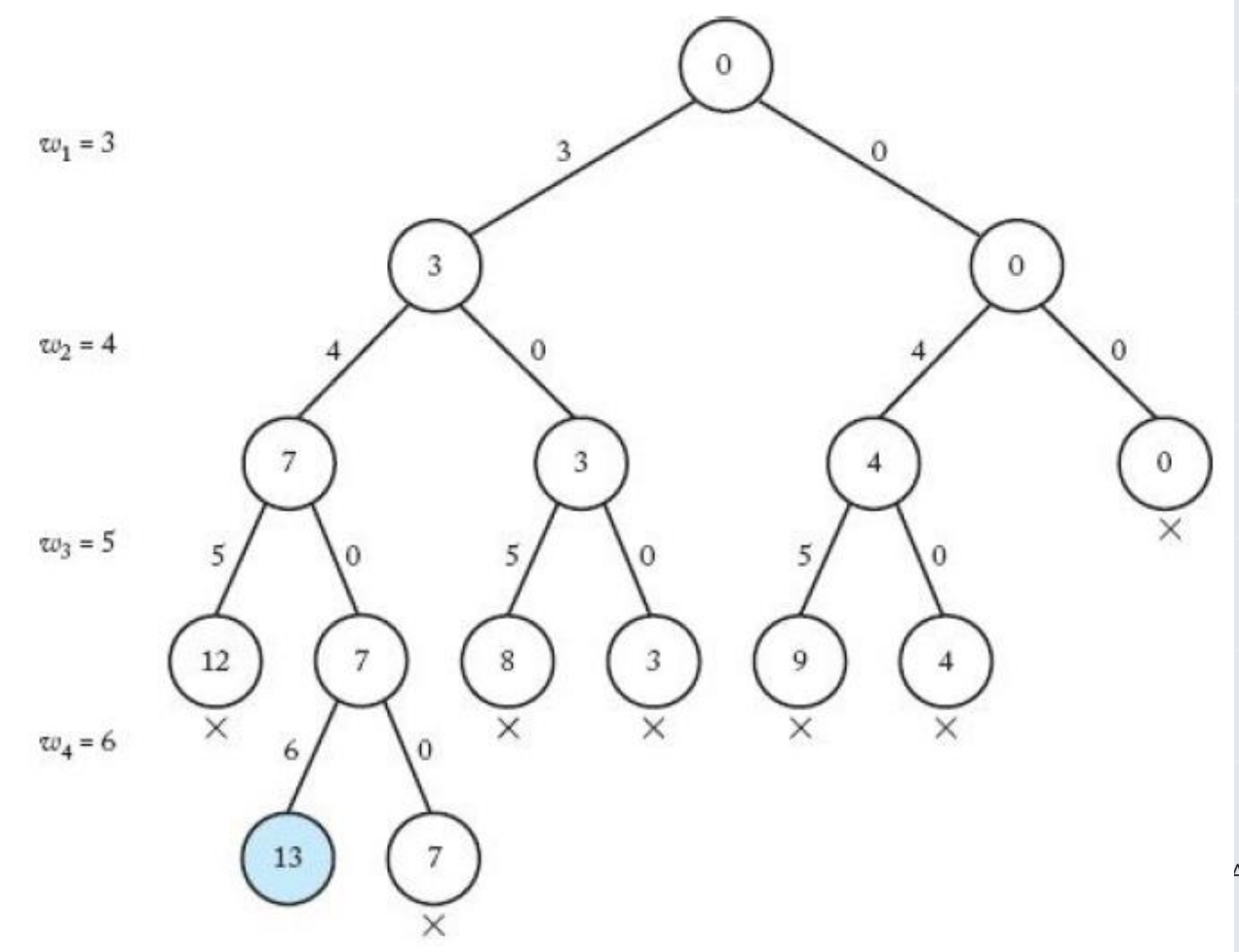
- Sort weights in non-decreasing order before search
- Let ***weight*** be the **sum of weights** that have been included up to a node at level i: node at level I is non-promising if $\boldsymbol{weight + w_{i+1} > W}$
- Let *total* be the total weight of the remaining weights at a given node.
  - A node is non-promising if *weight* + *total* < W

Example 5.4

Figure 5.9 shows the pruned state space tree when backtracking is used with $n = 4$, $W = 13$, and

$$w_1 = 3 \qquad w_2 = 4 \qquad w_3 = 5 \qquad w_4 = 6.$$

The only solution is found at the node shaded in color. The solution is $\{w_1, w_2, w_4\}$. The nonpromising nodes are marked with crosses. The nodes containing 12,



The Backtracking Algorithm for the Sum-of-Subsets Problem

Problem: Given n positive integers (weights) and a positive integer W, determine all combinations of the integers that sum to W.

Inputs: positive integer n, sorted (nondecreasing order) array of positive integers w indexed from 1 to n, and a positive integer W.

Outputs: all combinations of the integers that sum to W.

```
void sum_of_subsets (index i,
                     int weight, int total)
{
  if (promising(i))
     if (weight == W)
        cout << include[1] through include[i];
     else{
        include[i + 1] = "yes";                  // Include w[i + 1].
        sum_of_subsets(i + 1, weight + w[i +1], total - w[i + 1]);
        include[i + 1] = ''no";                  // Do not include w[i + 1].
        sum_of_subsets(i + 1, weight, total - w[i + 1]);
     }
}

bool promising (index i);
{
  return (weight + total >= W) && (weight == W || weight + w[i + 1] <= W);
}
```

If ***n, w, W*** and ***include*** variables were defined globally, the top-level call to ***sum_of_subsets()*** would be as follows:

```
sum_of_subsets(0, 0, total);
```

where initially

$$total = \sum_{j=1}^{n} w[j].$$

Algorithm 5.4 Backtracking Algorithm for Sum-of-Subsets

- Total number of nodes in the state space searched by Algorithm 5.4

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

- Worst case could be better – i.e. for every instance, only a small portion of the state space tree is search
- **Not the case**
- For each n, it is possible to construct an instance for which algorithm visits exponentially large number of nodes – even if only seeking one solution
- **The algorithm can be efficient for many large instances.**

Optimization problem

- General algorithm of backtracking in the case of optimization problem

```
void checknode (node v)
{
  node u;
  if (value(v) is better than best)
     best = value(v);
  if (promising(v))
     for (each child u of v)
        checknode(u);
}
```

best = value of the best solution so far

value(v) = value of the solution at the node v

node is promising only **if we should expand to its children** (different from the previous algorithms)

profit is the sum of the profits of the items included up to the node
weight is the sum of the weights of those items

1. initialise variables ***bound*** and **totweight** to **profit** and **weight**, respectively
2. until we get to an item that, if grabbed, would bring **totweight** above **W**
   greedily grab items, adding their **profits** to **bound** and their **weights** to **totweight**

Suppose the node is at level i, and the node at level k is the one that would bring the sum of the weights above W. Then

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j, \quad \text{and}$$

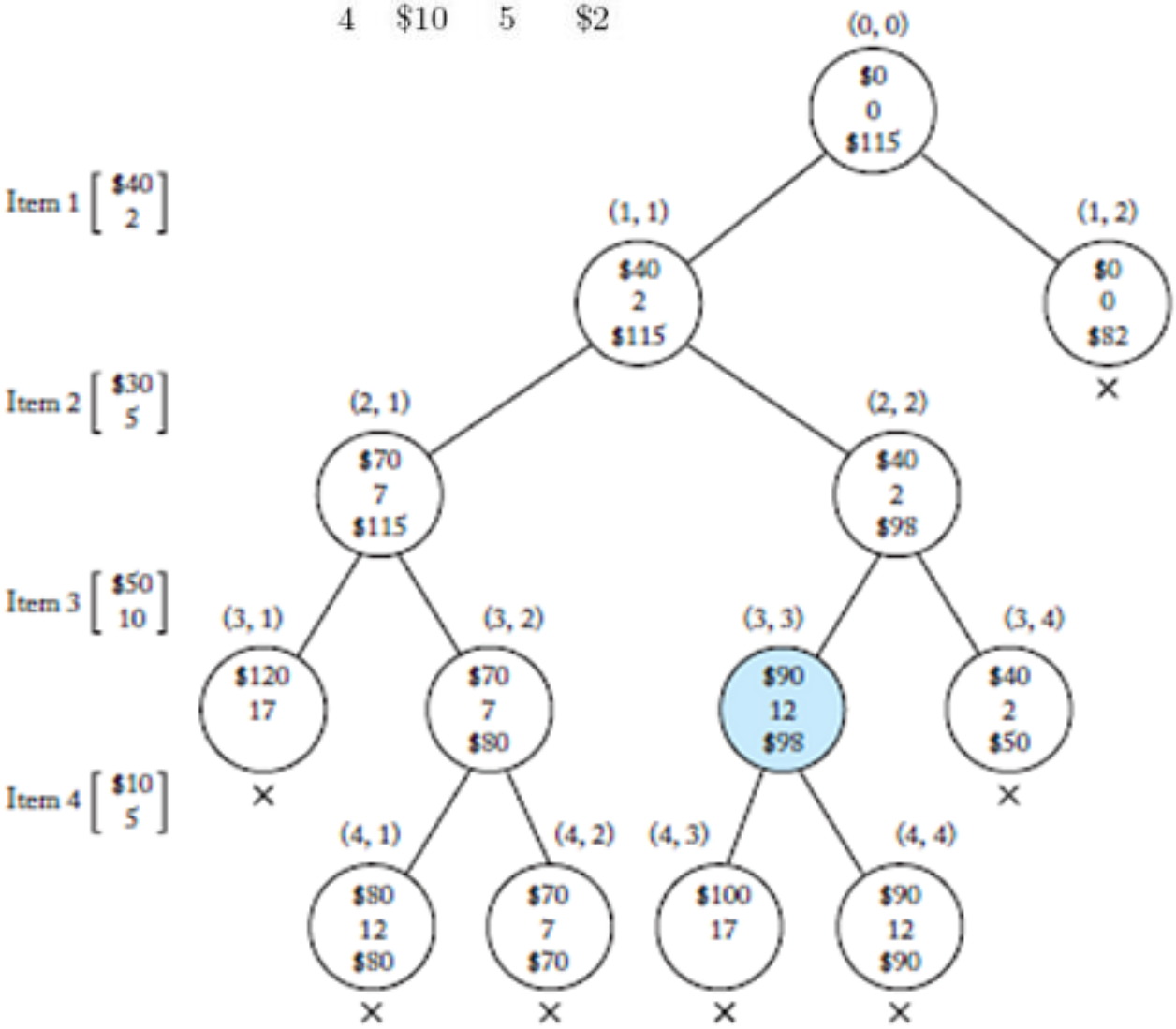
$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

If *maxprofit* is the value of the profit in the best solution found so far, then a node at level *i* is nonpromising if

$$bound \leq maxprofit.$$

Suppose that $n = 4$, $W = 16$, and we have the following:

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2



0-1 Knapsack Problem

- Optimization problem
- Best solution so far

The state space tree in the 0-1 Knapsack problem is the same as that in the Sum-of-Subsets problem. As shown in Section 5.4, the number of nodes in that tree is

$$2^{n+1} - 1.$$

Algorithm 5.7 checks all nodes in the state space tree for the following instances. For a given n, let $W = n$, and

$$p_i = 1 \qquad w_i = 1 \qquad \text{for } 1 \leq i \leq n-1$$
$$p_n = n \qquad w_n = n.$$

The optimal solution is to take only the nth item, and this solution will not be found until we go all the way to the right to a depth of $n - 1$ and then go left.

Comparing the **Dynamic Programming Algorithm** and the **Backtracking Algorithm** for the 0-1 Knapsack Problem

- dynamic programming algorithm for the 0-1 Knapsack problem is in O(minimum (2^n, nW))
- In the worst case, the backtracking algorithm checks Θ (2^n) nodes

- **it is difficult to analyse theoretically the relative efficiencies of the two algorithms**

- Horowitz and Sahni (1978) found that the backtracking algorithm is **usually more efficient** than the dynamic programming algorithm.