Chapter 5

Backtracking

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

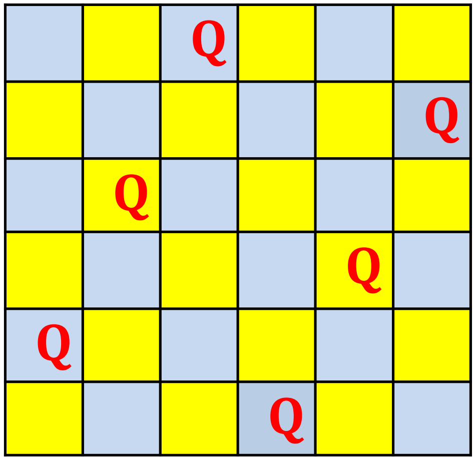
Backtracking vs Dynamic Programming

- Dynamic Programming – subsets of a solution are generated
- Backtracking – Technique for deciding that some subsets need not be generated

5.1 Backtracking Technique

- Problems: a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.
- The classic example : is in the *n-Queens problem.*

- *Goal: is to position n queens on an* $n \times n$ *chessboard so that*, no two queens may be in the same row, column, or diagonal.
- The sequence in n-queen problem is the ***n*** positions in which the queens are placed, the set for each choice is the n^2 possible positions on the chessboard.
- The n-Queens problem is a generalization of its instance when *n = 8, which is the* instance using a standard chessboard.
- We will illustrate backtracking using the instance when *n = 4.*



N =6

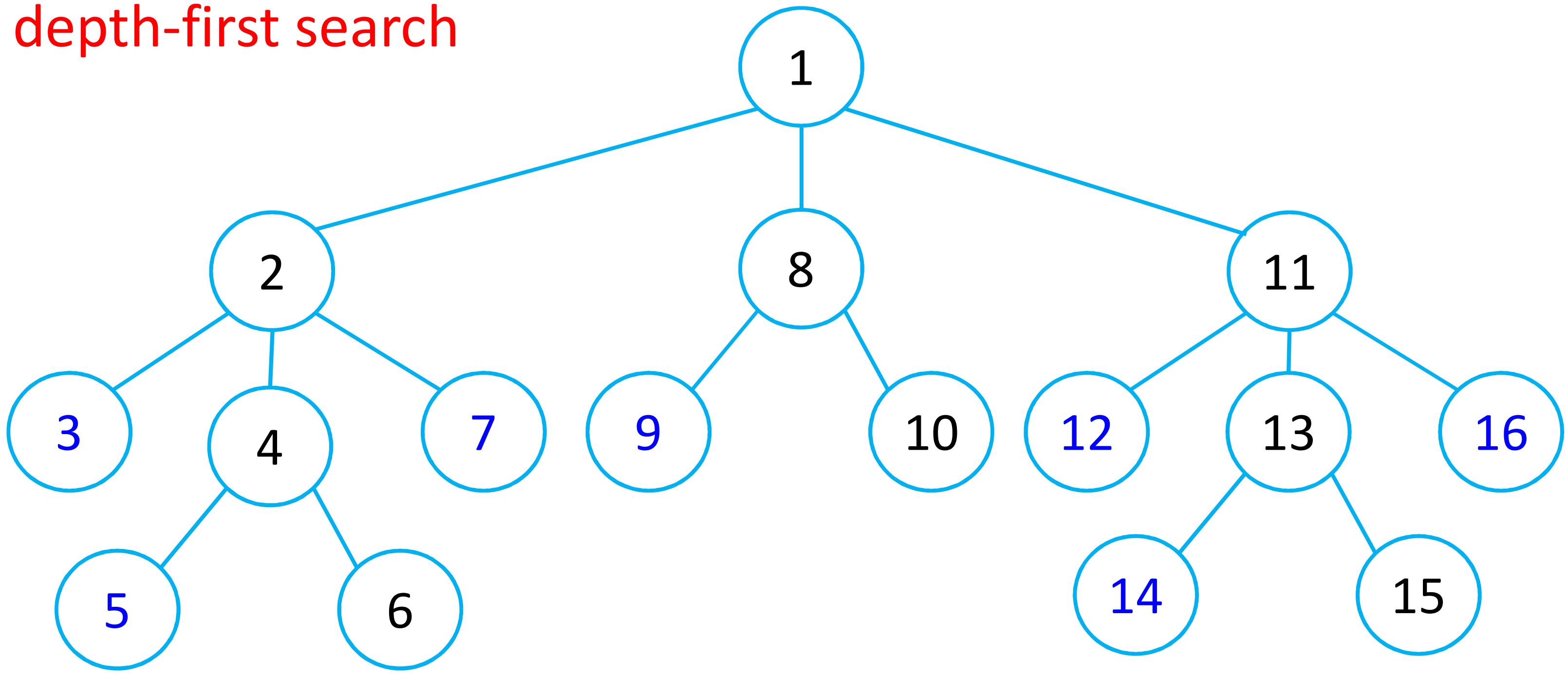


Figure 5.1 depth-first search

- The nodes are numbered in the order in which they are visited.
- path is followed as deeply as possible until a dead end is reached.
- At a dead end we back up until we reach a node with an unvisited child, and then we again proceed to go as deep as possible.

- Backtracking is a modified depth-first search of a tree
- A ***preorder tree traversal*** is a depth-first search of the tree.
- This means that
  - the root is visited first, and
  - a visit to a node is followed immediately by visits to all descendants of the node.
- Although a depth-first search does not require that the children be visited in any particular order, we will visit the children of a node from left to right in the applications in this chapter.

Function called by passing root at the top level

```
void depth_first_tree_search(node v)
{
        node u;
        visit v;
        for( each child u of v)
                depth_first_tree_searach(u);
}
```

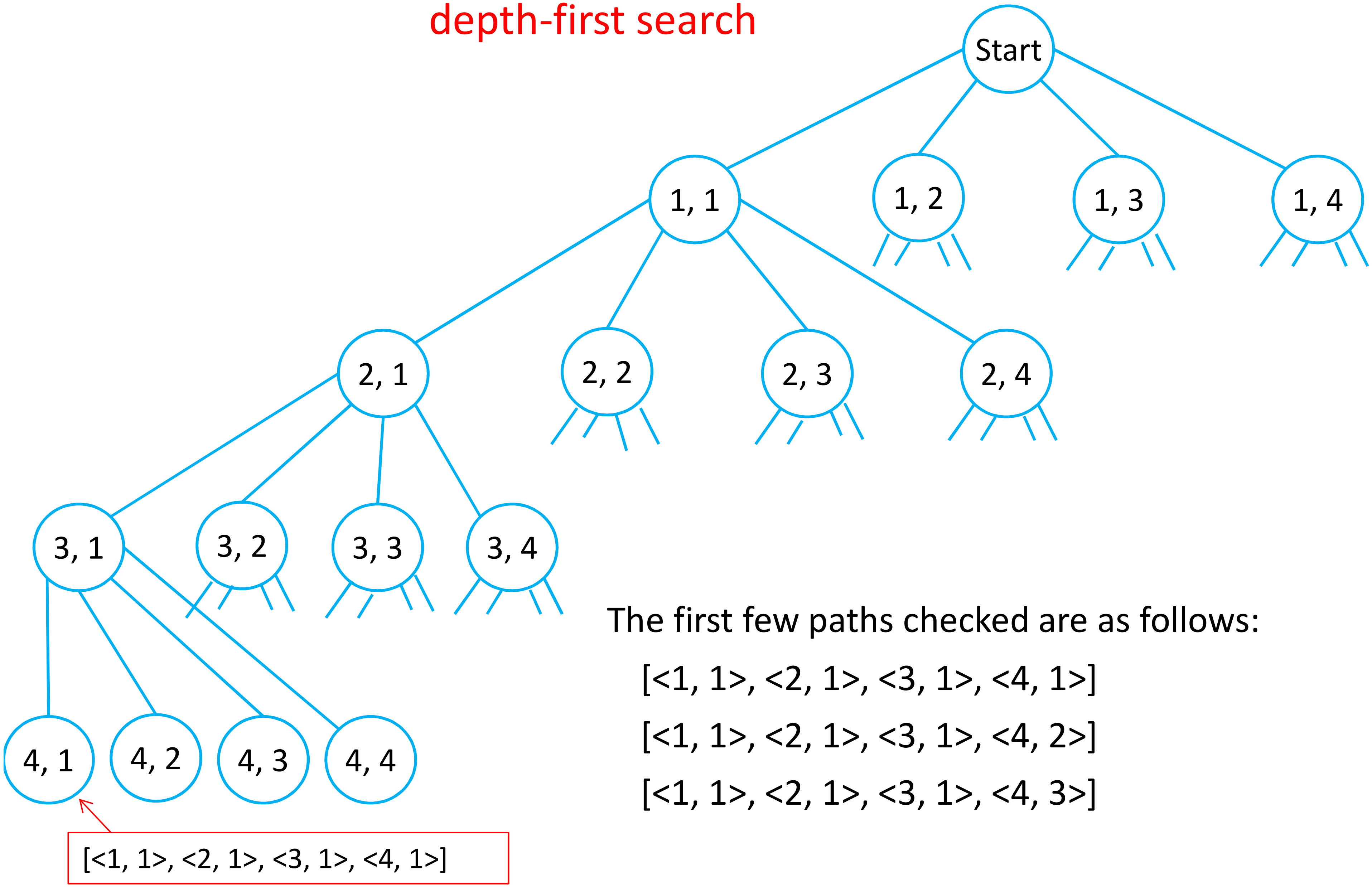
- Consider n queens, $n = 4$.
- To simplify : **no** two queens can be in the same row.
- Assign to each queen a different row and checking which column combinations yield solutions.
- Because each queen can be placed in one of four columns, there are:

$$4 \times 4 \times 4 \times 4 = 256 \text{ candidate solutions}$$

- **Nodes**:
  - Non-promising node: the node cannot lead to a solution
  - Promising node: may lead to a solution
- If a node is non-promising, backtrack to the node's parent and proceed with the search on the next child

- We can create the candidate solutions by constructing a tree in which the column choices for the first queen (the queen in row 1) are stored in level-1 nodes in the tree , the column choices for the second queen (the queen in row 2) are stored in level-2 nodes, and so on. This tree is called a ***state space tree***
- A path from the root to a leaf is a candidate solution
- ***The*** entire tree has 256 leaves, one for each candidate solution.

depth-first search



The first few paths checked are as follows:

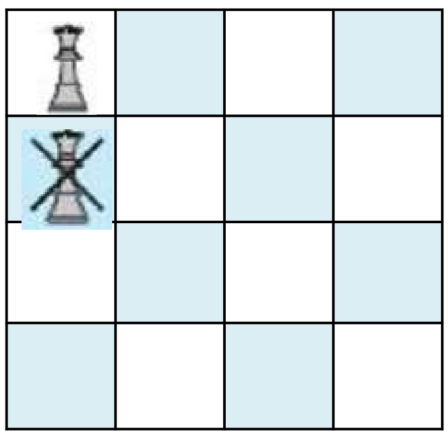
[<1, 1>, <2, 1>, <3, 1>, <4, 1>]

[<1, 1>, <2, 1>, <3, 1>, <4, 2>]

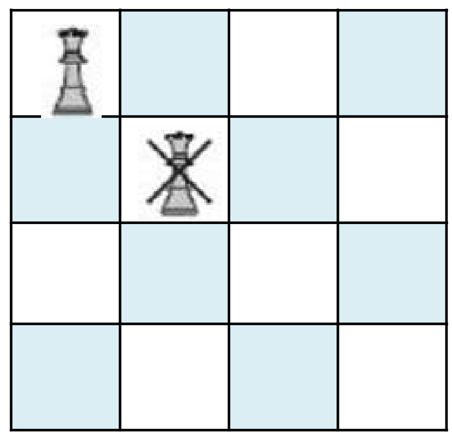
[<1, 1>, <2, 1>, <3, 1>, <4, 3>]

state space tree

- We can make the search more efficient as follows: no two queens can be in the same column.
- This sign tells us that this node can lead to nothing but dead ends.
- Similarly, as illustrated in Fig. 5.3(b) .



(a)



(b)

- Backtracking : a depth-first search of a state space tree, checking whether each node is promising
- If it is nonpromising, backtracking to the node's parent. This is called ***pruning the state space tree,***
- ***The subtree*** consisting of the visited nodes is called the ***pruned state space tree.***
- A general algorithm for the 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 );
}
```

Example 5.1

- *n-Queens* problem: the function *promising* must return false if a node and any of the node's ancestors place queens in the same column or diagonal.

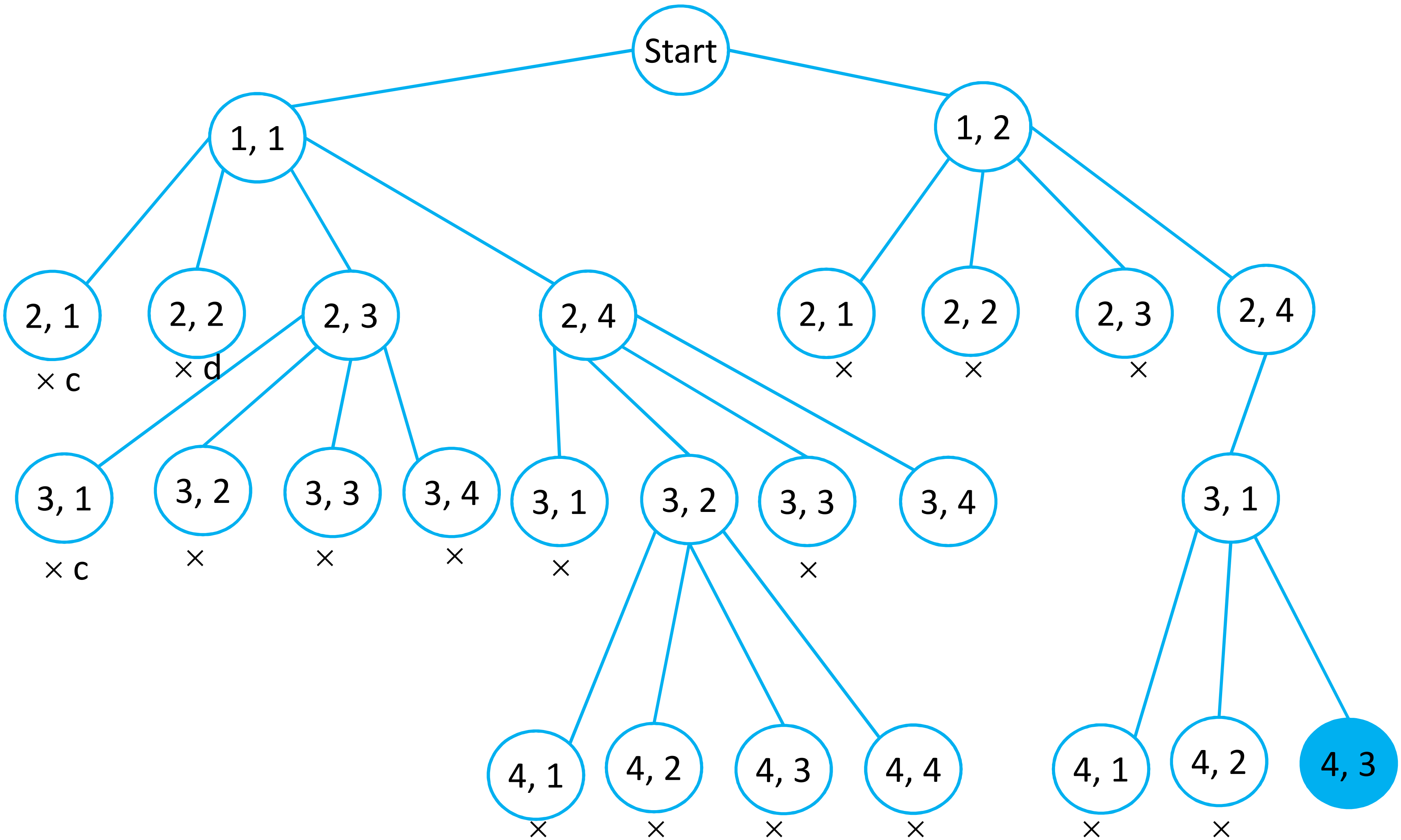


Figure 5.4 shows a portion of the pruned state space tree produced when backtracking is used to solve the instance in which *n = 4.*

(a) < 1, 1 > is promising. { because queen 1 is the first queen positioned }

(b) < 2, 1> is nonpromising. {because queen 1 is in column 1}

< 2, 2> is nonpromising. {because queen 1 is on left diagonal }

< 2, 3> is promising

(c) < 3, 1 > is nonpromising. { because queen 1 is in column 1 }

< 3, 2> is nonpromising. {because queen 2 is on right diagonal }

< 3, 3> is nonpromising. {because queen 1 is in column 1 }

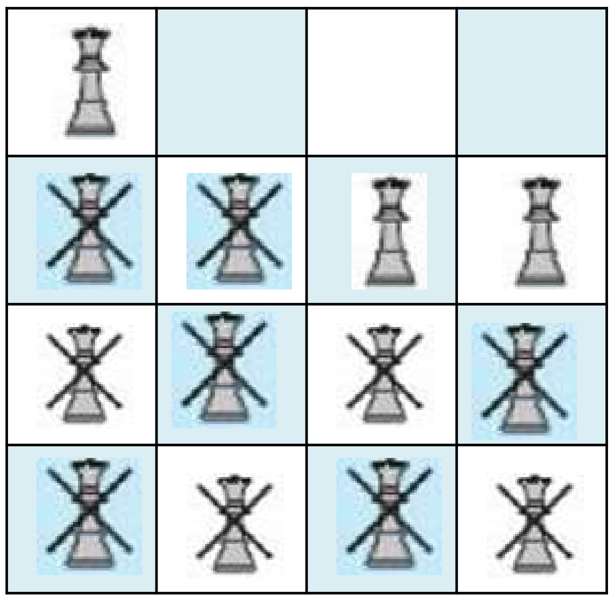
< 3, 4> is nonpromising. {because queen 2 is on left diagonal }

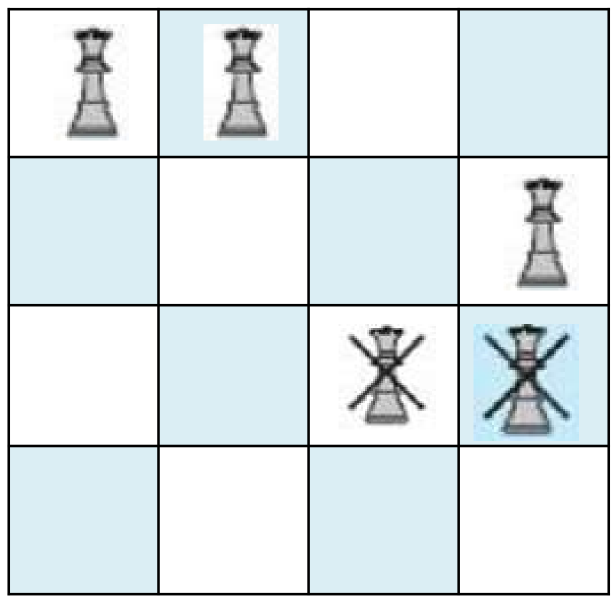
(d) Backtrack to < 1, 1 >

< 2, 4> is promising

(e) < 3, 1> is nonpromising. {because queen 1 is in column 1 }

< 3, 2> is promising. {This is the second time we've tried < 3, 2> }





- There is some inefficiency in *checknode,* we check whether a node is promising after passing it to the procedure. This means that activation records for nonpromising nodes are unnecessarily placed on the stack of activation records.
- We could avoid this by checking whether a node is promising before passing it. A general algorithm for backtracking that does this is as follows:

```
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 );
}
```

5.2 The *n-Queens* Problem

- All solutions to N-Queens problem
- ***col* (*i*) :** the column where the queen in the ith row is located Promising function:
  - 2 queens same column?

    col(i) == col(k)
  - 2 queens same diagonal?

    col(i) – col(k) == i – k or col(i) – col(k) == k - i

col(2)=8, col(3)=1, col(6)=4,

col(2)- col(6)=4=6-2

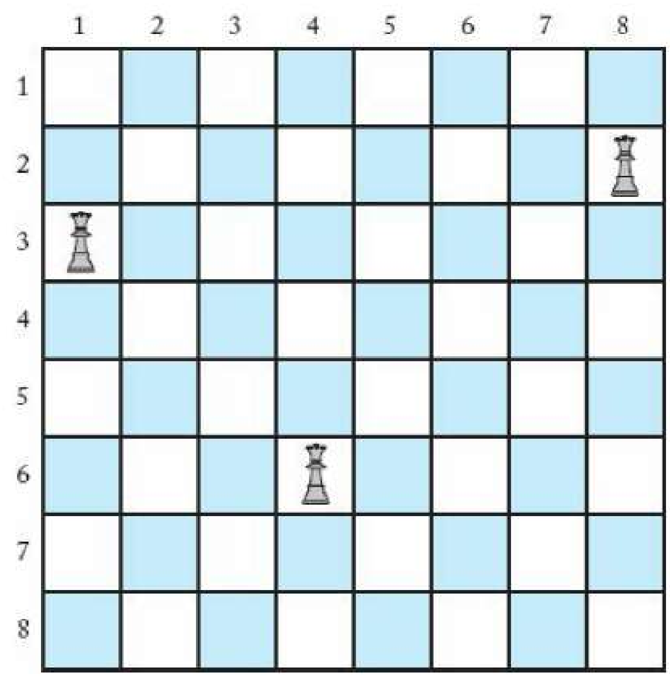
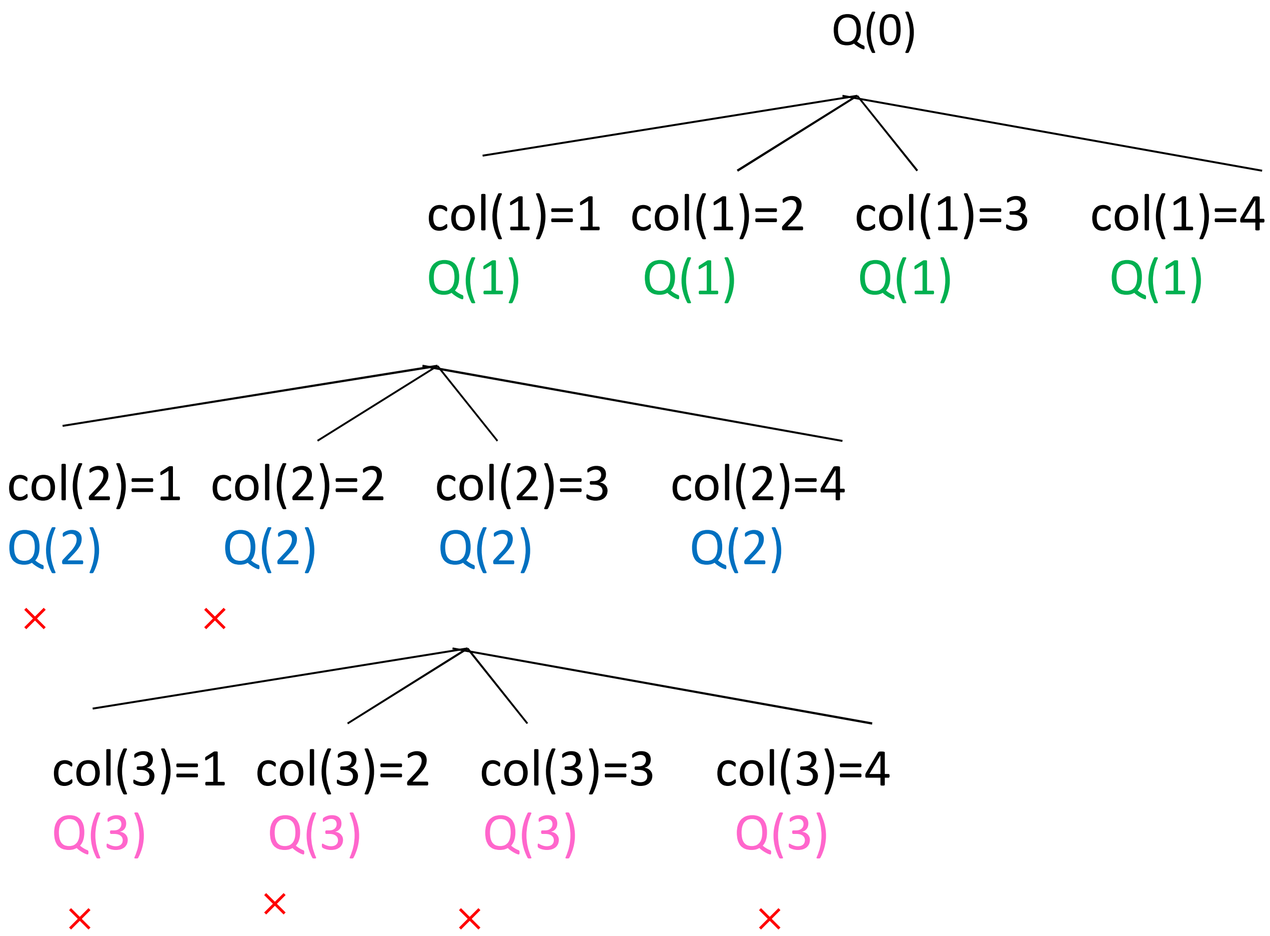


Figure 5.6 The queen in row 6 is being threatened in its left diagonal by the queen in row 3 and in its right diagonal by the queen in row 2.


Q(0)
col(1)=1 Q(1)
col(1)=2 Q(1)
col(1)=3 Q(1)
col(1)=4 Q(1)
col(2)=1 Q(2) ×
col(2)=2 Q(2) ×
col(2)=3 Q(2)
col(2)=4 Q(2)
col(3)=1 Q(3) ×
col(3)=2 Q(3) ×
col(3)=3 Q(3) ×
col(3)=4 Q(3) ×

Algorithm 5.1

The Backtracking Algorithm for the *n-Queens Problem*

- Problem: Position *n* queens on a chessboard so that no two are in the same row, column, or diagonal.
- Inputs: positive integer *n.*
- Outputs: all possible ways *n* queens can be placed on an *n* × *n* chessboard so that no two queens threaten each other.
- Each output consists of an array of integers ***col*** *indexed from* ***1*** *to* **n***, (col[i]*] is the position of the queen in the *ith row .*

```
void queens ( index i  )
{
 if (promising ( i ))
     if ( i == n )    cout col [ 1 ]  through  col [ n ]
     else
        for (int j = 1; j <= n; j++){            // see if queen in (i+1) st row
           col [ i + 1 ] = j;                     // can be positioned in each of
           queens ( i + 1 );                      // the nth column
        }
}
bool promising ( index i )
{
   index k;   bool switch;
   k = 1;     switch = true;
   while ( k < i && switch){                                //check if any queen threatens
                                                 // queen in th ith row
     if (col [ i ] == col [ k ] || abs ( col [ i ] - col[ k ]) == ( i - k )   switch = false;
     k++;
   }
   return switch;
}
```

- In Algorithm 5.1, the main routine is *queens.*
- n and *col* are not inputs to the recursive routine queens.
- *The top-level call to queens : queens(0)*
- In general, the problems in this chapter can be stated to require one, several, or all solutions.
- It is a simple modification to make the algorithms stop after finding one solution.

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

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

For the instance in which $n = 8$, the state space tree contains

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

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

- To obtain an upper bound on the number of promising nodes: no two queens can ever be placed in the same column.
- For example, consider the instance in which *n = 8.*
- The first queen can be positioned in any of the eight columns, the second can be positioned in at most seven columns; once the second is positioned, the third can be positioned in at most six columns; and so on.
- Therefore, there are at most

  $1 + 8 + 8 \times 7 + 8 \times 7 \times 6 + \ldots + 8! = 109\,601$ promising nodes
- Generalizing this result to an arbitrary *n,*

  $1 + n + n \times (n-1) + \ldots + n! =$ promising nodes
- This analysis does not give us a very good idea as to the efficiency of the algorithm for the following reasons: First, it does not take into account the diagonal check in function *promising.*
- Therefore, there could be far less promising nodes than this upper bound. Second, the total number of nodes checked includes both promising and nonpromising nodes.

- Algorithm 1 : depth-first search without backtracking.
- The number of nodes it checks is the number in the state space tree.
- Algorithm 2 :no two queens can be in the same row or in the same column. Algorithm 2 generates *n! candidate solution*
- Nqueen, n=8:
  - promo 2057
  - nonpromo 13664

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}

Algorithm 5.4 Backtracking Algorithm for Sum-of-Subsets

- Sum-of-Subsets problem: there are n positive integers (weights) w_i and a positive integer W.
- The goal is to find all subsets of the integers that sum to W.

Example 5.2

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

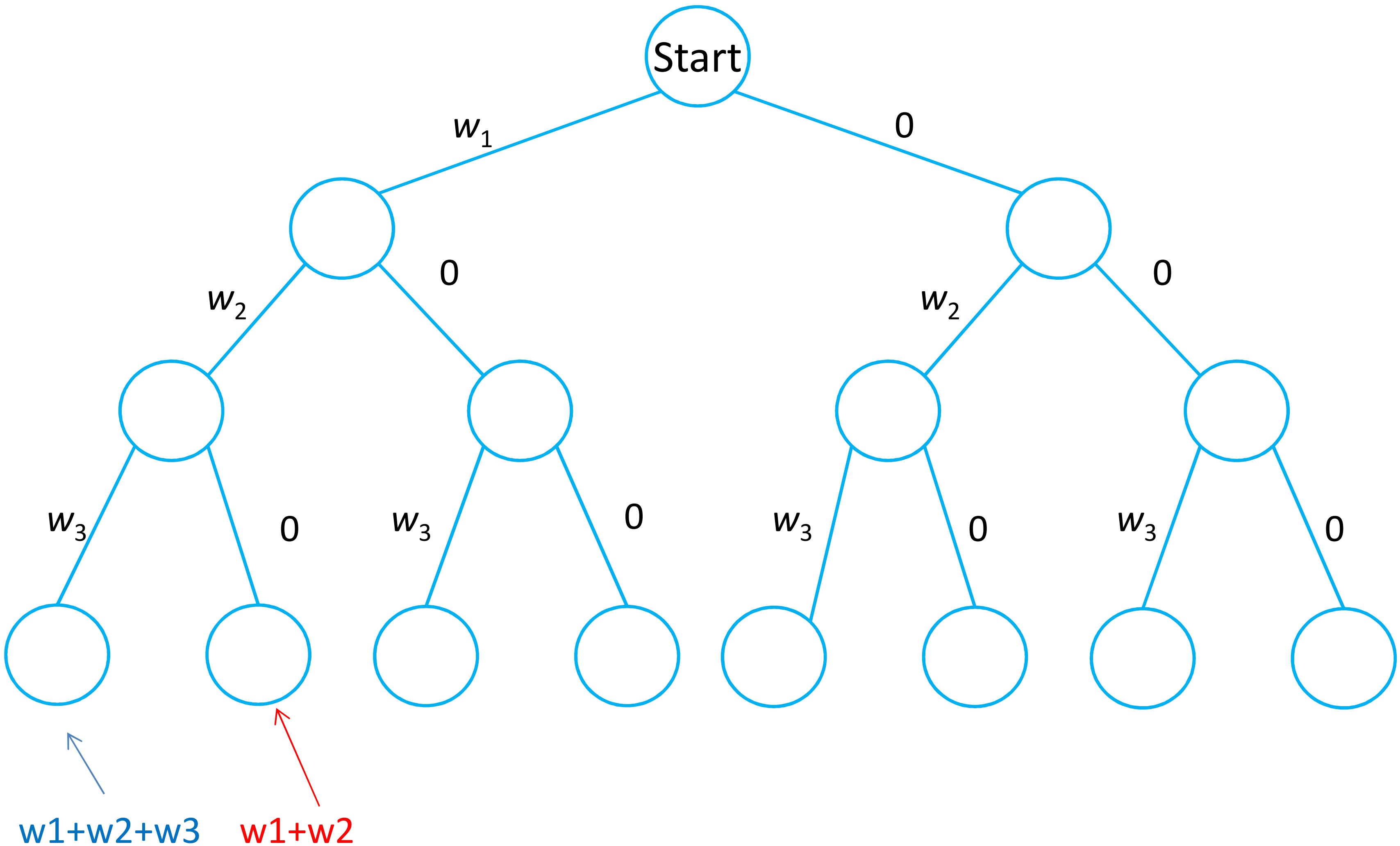
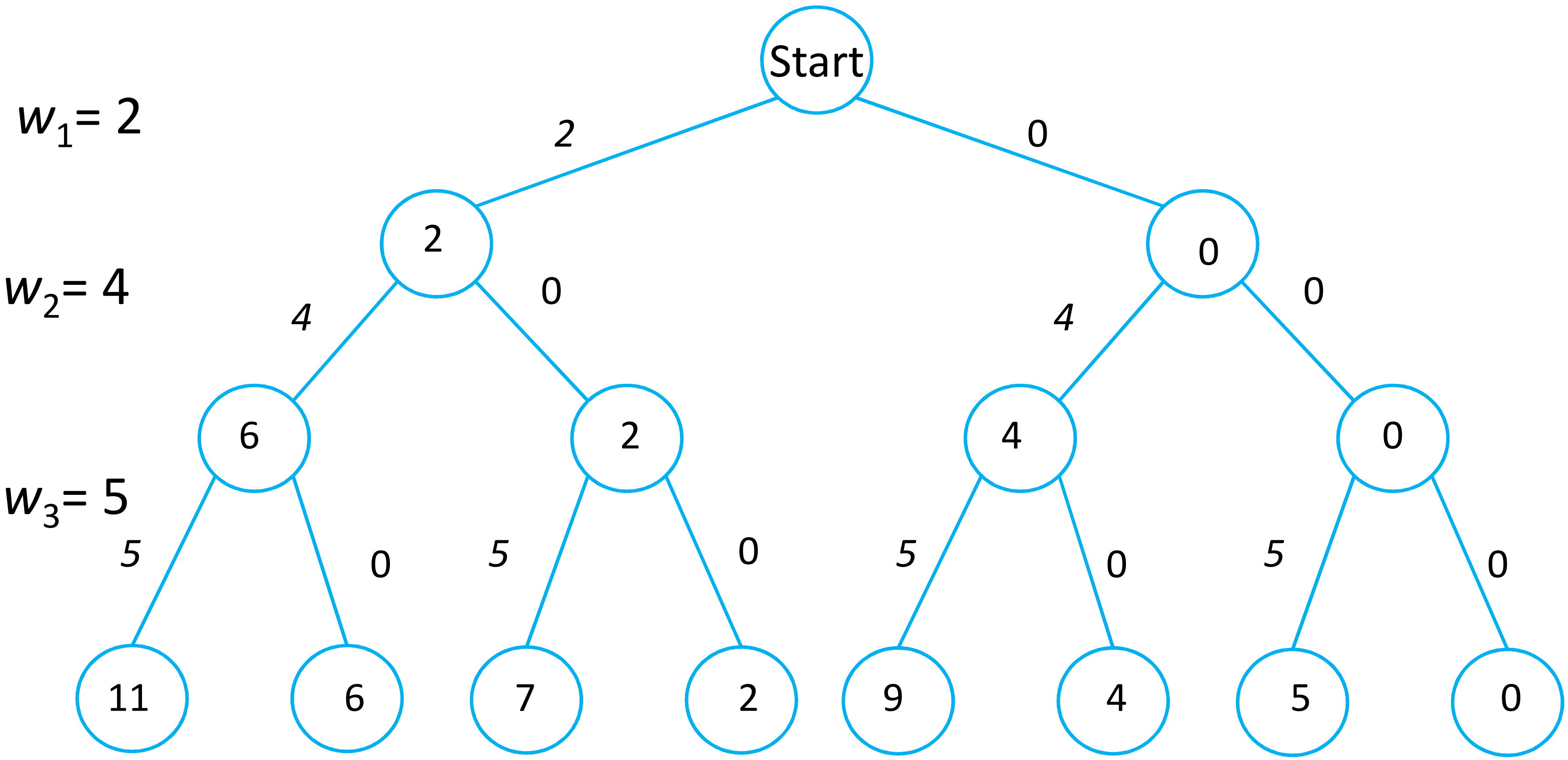


Figure 5.7 A state space tree for instances of the Sum-of-Subsets problem in which *n = 3.*

Example 5.3

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

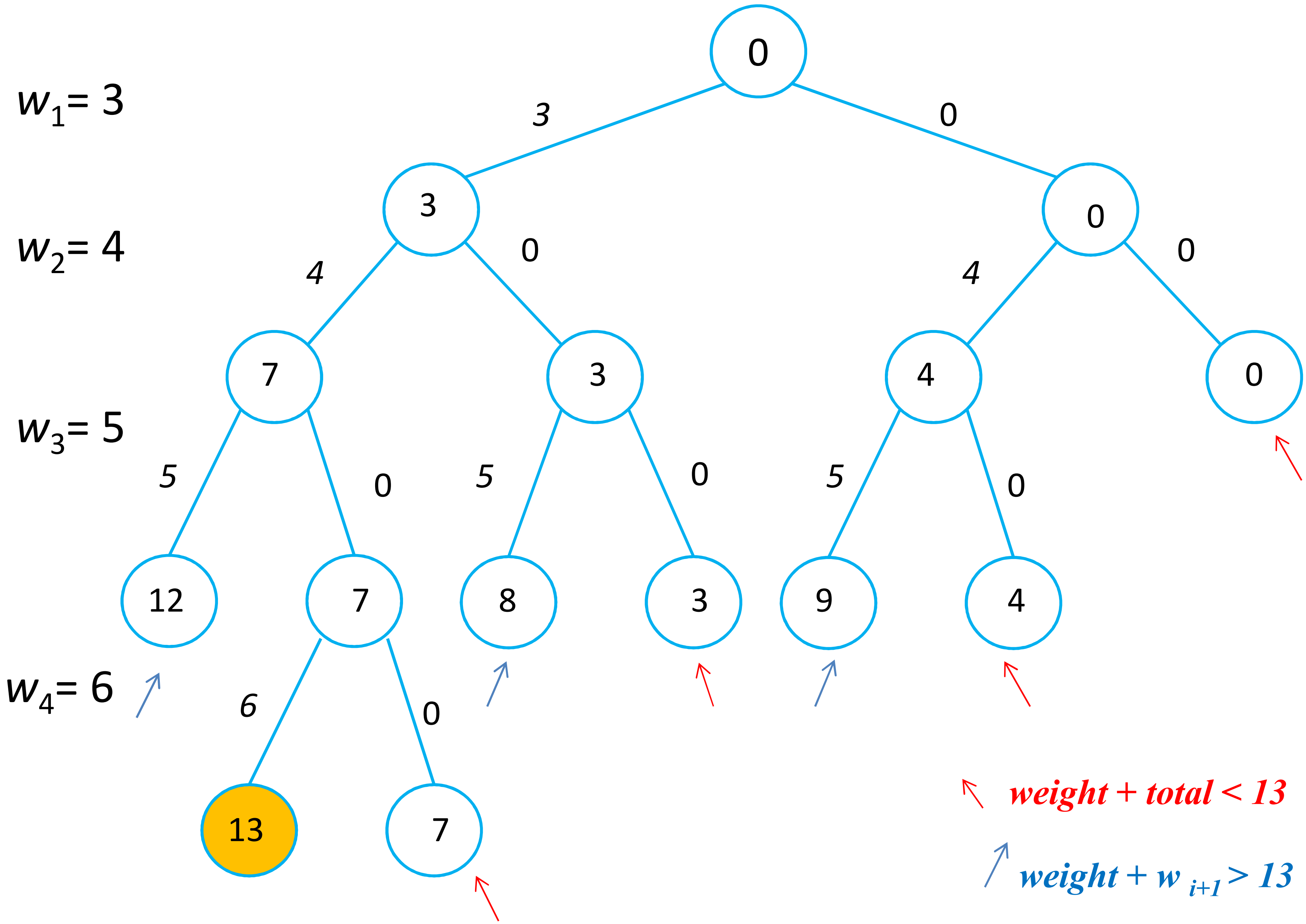


$$Weight + w_{i+1} > W$$

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

Example 5.4

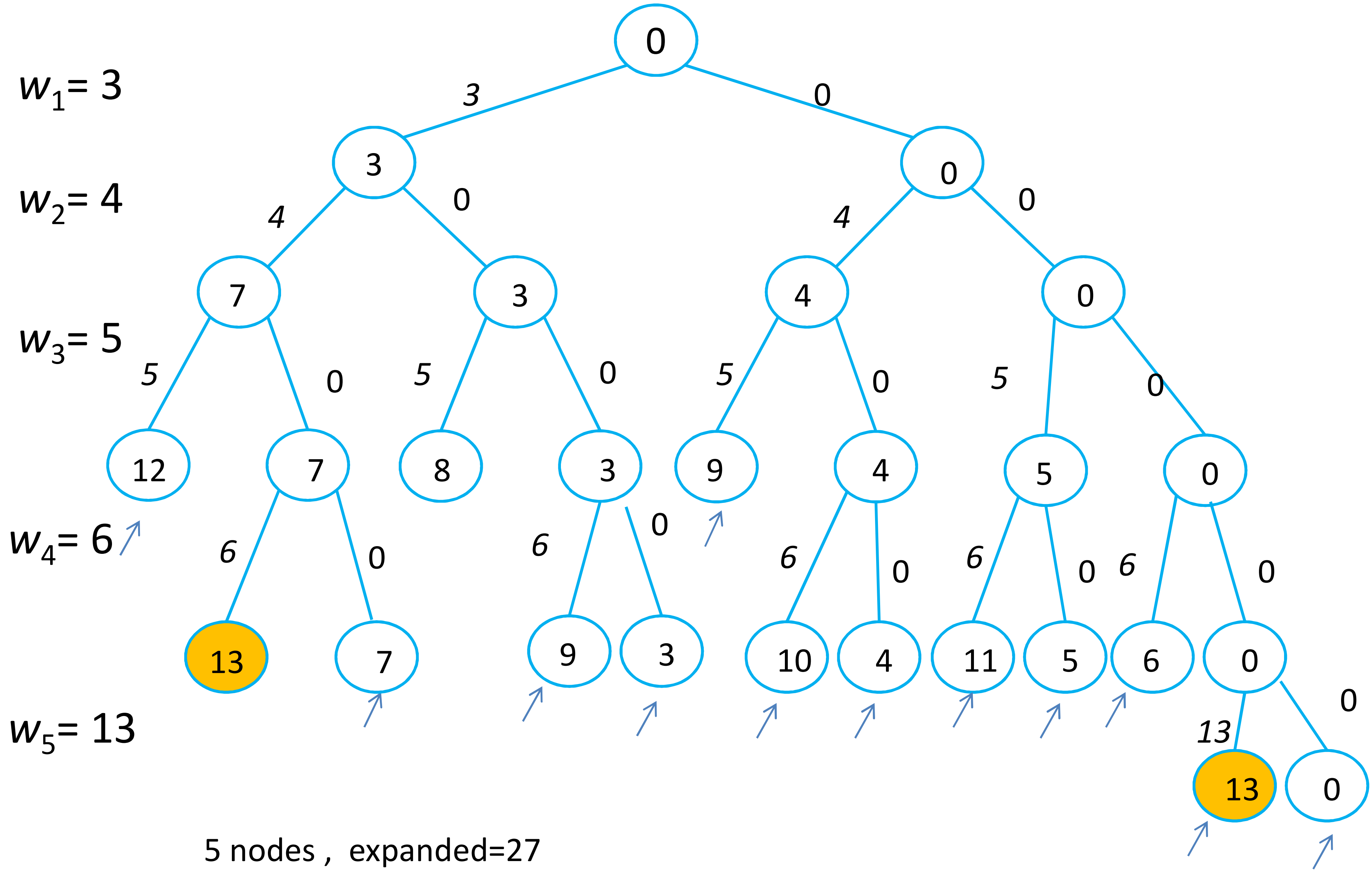
- Figure 5.9 shows the pruned state space tree when backtracking is used with *$n = 4$, $W = 13$, and* , $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$



5 nodes , expanded=15

$2^{n+1} - 1 = 2^5 - 1 = 31$

$n = 13, W = 13, and\ , w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6, w_5 = 13$



5 nodes , expanded=27

$2^{n+1} – 1= 2^6 – 1= 63$

weight + total < 13

weight + w_{i+1} > 13

Algorithm 5.4

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.*

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

void sum_of_subsets (int i, int weight, int total )
{
  if (promosing (i, weight, total))
     if (weight==W)
          cout  include[1] through cout  include[k]
     else {
          include [ i + 1 ] = "yes";
          sum_of_subsets (i+1, weight + w [ i + 1 ],  total – w [ i + 1 ] );
          include[i+1]="no";
          sum_of_subsets ( i + 1, weight, total – w [ i + 1 ] );
     }
}
```

First call: Sum-of-Subsets(0, 0, total), $total = \sum_{j=1}^{n} w[j]$

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

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

- It could be that for every instance only a small portion of the state space tree is searched.
- This is not the case. For each *n, it is possible to* construct an instance for which the algorithm visits an exponentially large number of nodes.