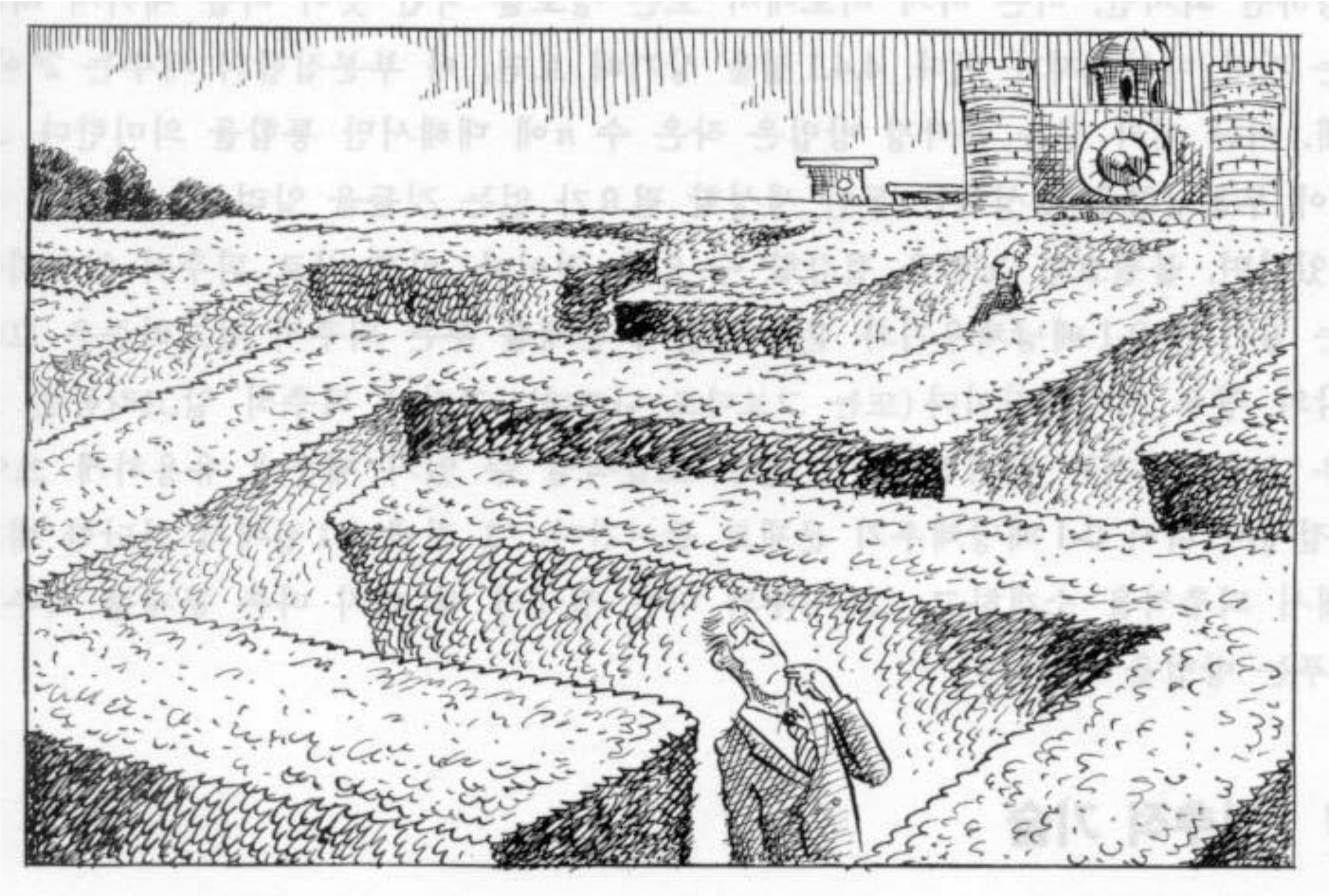
Lect05. Backtracking



Backtracking

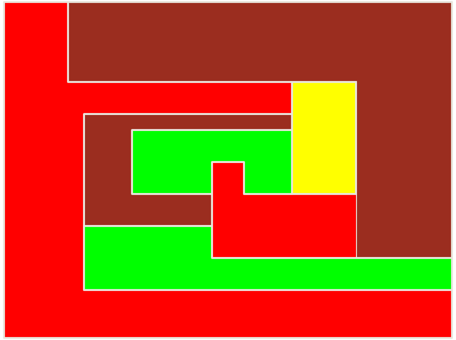
- Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"
- Backtracking is kind of like trying **to find your way out of a maze**.
- You try heading in one direction, and if you hit a dead end, you go back to the last intersection and try a different direction.
- In the worst case, you end up trying every possible passage in the maze.

Solving a maze

- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don’t have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

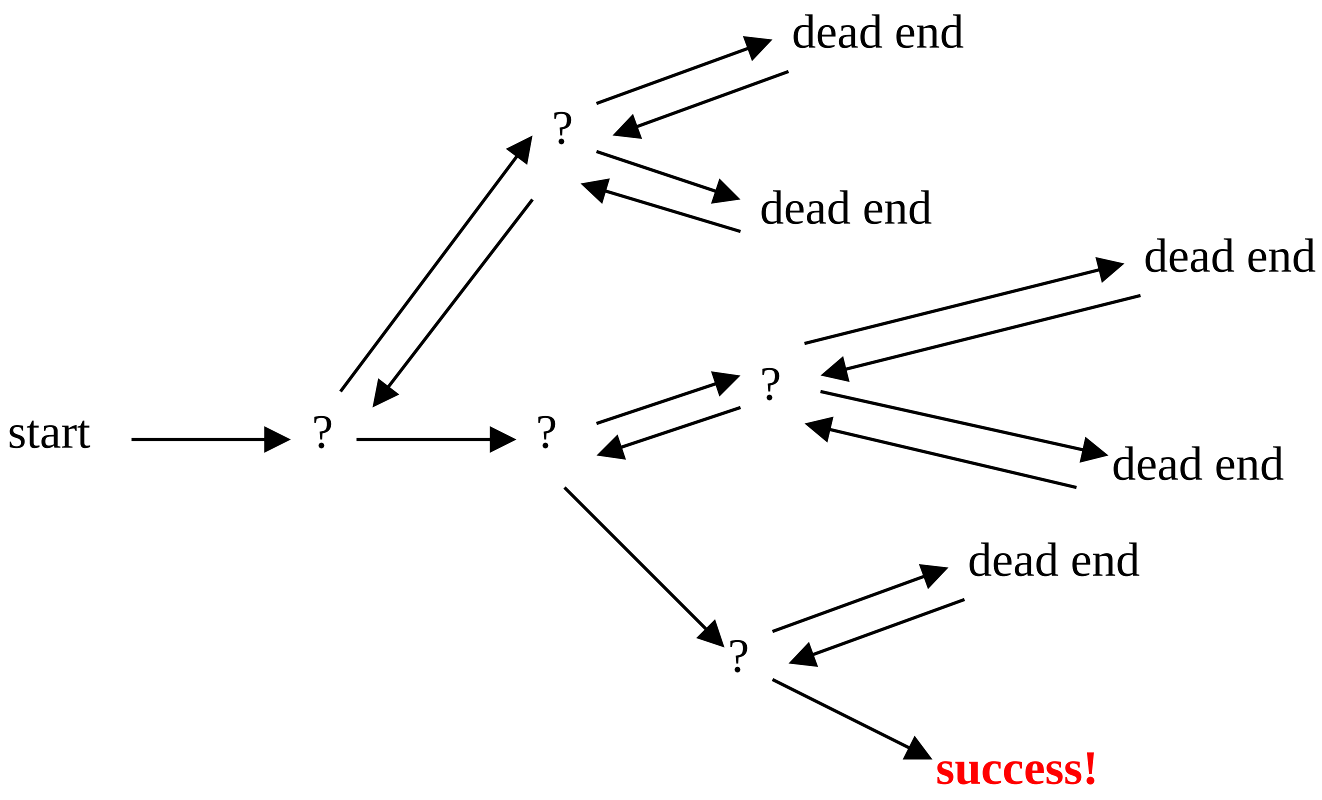
Example : Coloring a map

- You wish to color a map with not more than four colors
  - red, yellow, green, blue
- Adjacent countries must be in different colors



- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

Backtracking (animation)



The Backtracking Technique

- Backtracking is used to solve problems in which a **sequence** of objects is selected from a specified **set** so that the sequence satisfies some **criterion**.
- **n-Queens Problem** : to position ***n*** queens on an n x n chessboard so that no two queens threaten each other.
  - **No** two queens can be in the **same row, column, or diagonal**.
  - The sequence for the problem is the n positions in which the queens are placed.
  - The set for each choice is n^2 possible positions on the chessboard, and the criterion is that no two queens can threaten each other.
  - The n-Queens problem is a generalization of its instance when n=8, which is the instance using a standard chessboard.
- Depth-First Search : find a spanning tree from a graph. A **preorder** tree traversal is a depth-first search of a 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. Backtracking is a modified **depth-first search** of a tree.

Depth-First Search

- Visit a root node, then visit all descendants of the node from left to right(= preorder tree traversal).

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

DFS

- A depth-first search does not require that the children be visited in any particular order, but we will visit the **children** of a node from **left to right**.

Example of DFS

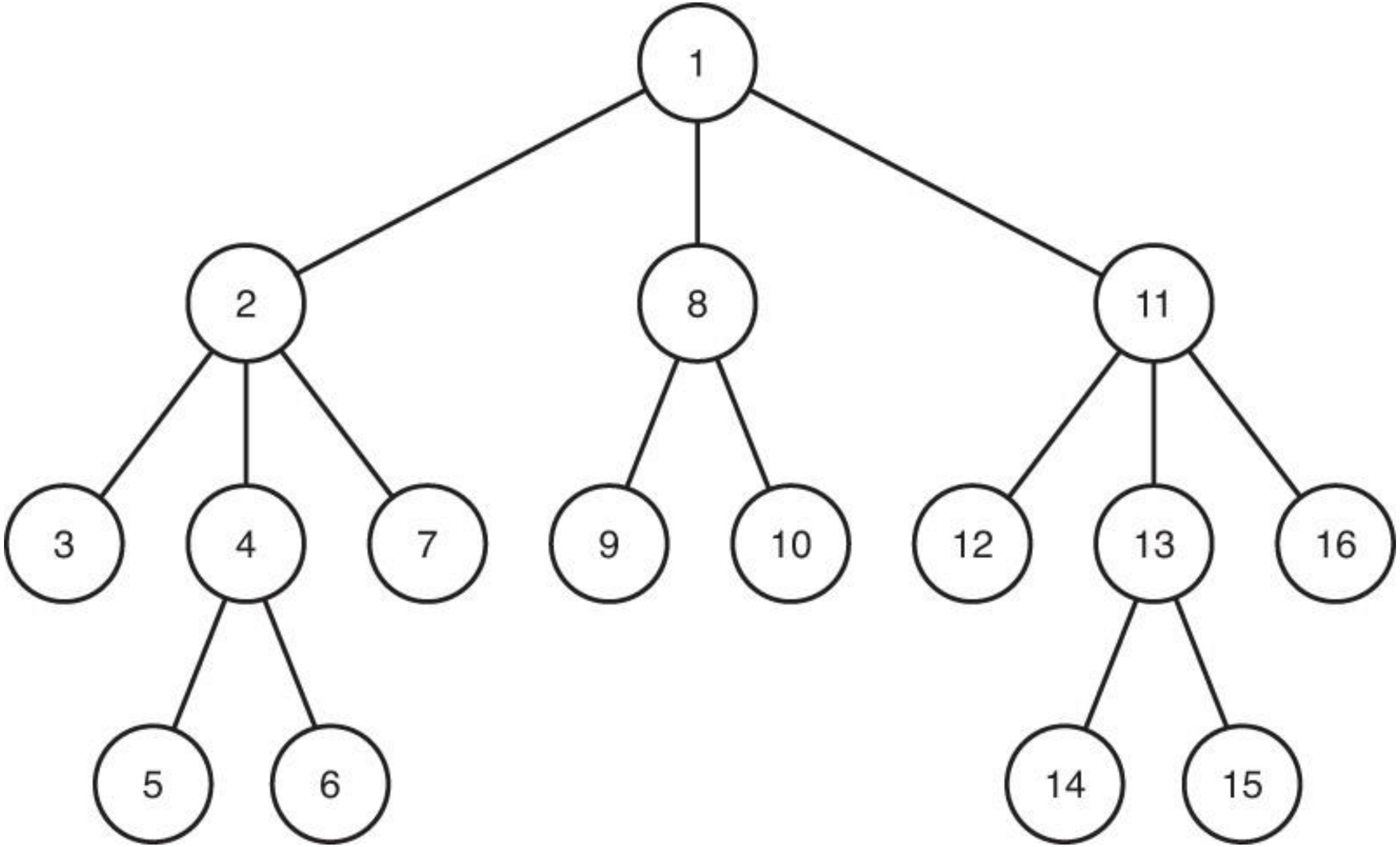


Figure 5.1: A tree with notes numbered according to a deph-first search

4-Queen Problem

- To return to the ***n*-Queens problem** when ***n* = 4**, our task is **to position four queens on a 4 x 4 chessboard** so that no two queens threaten each other.
- We already know that no two queens can be in the same row.
- The instance can be solved by assigning each queen a different row and checking which column combinations yield solutions. In this case there are **4 x 4 x 4 x 4=256** candidate solutions.

State space tree for 4-Queens Problem

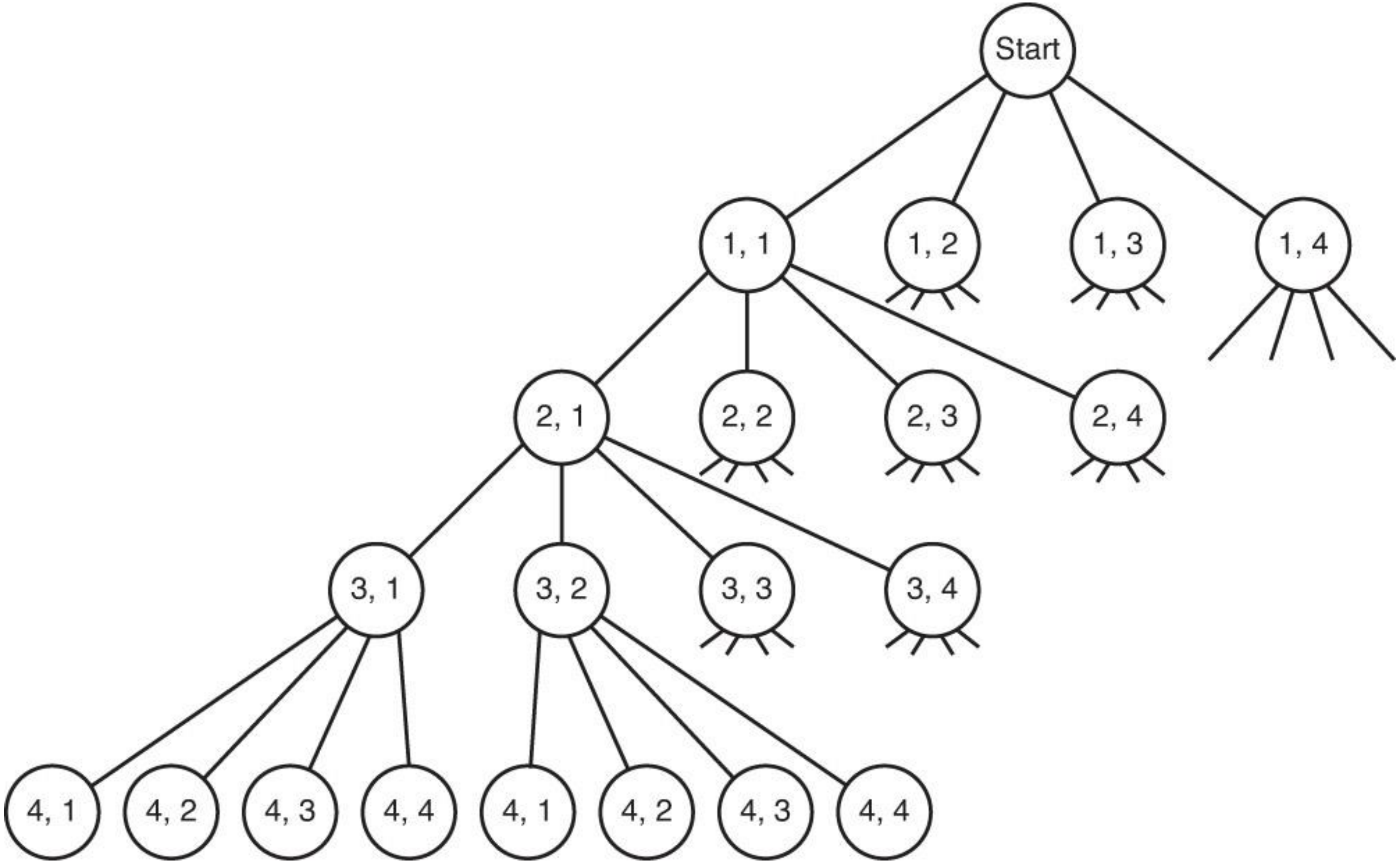


Figure 5.2: A portion of the state space tree for the instance of the n-Queens problem in which $n=4$.

State Space Tree

- We can create the **candidate solutions by constructing a tree** in which the column choices for the first queen (in row 1) are stored in level-1 nodes in the tree, the column choices for the second queen are in level-2 nodes, and so on. **A path from the root to the leaf is a candidate solution**. This tree is called a **state space tree**.
- The procedure whereby, after determining that a node lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child.
- Promising (node)：the node can lead to a solution, otherwise, it is called as nonpromising.
- Pruning：check each node whether it is promising, if not, backtracking to the node's parent.

Backtracking is the procedure to pruned state space tree

Algorithm and Pruned State Space Tree

- **Backtracking** consists of doing a depth-first search of a **state space tree**,
  - **checking** whether **each node is promising**, and
  - if it is **nonpromising**, **backtracking**(going back) to the node's parent. This process is called **pruning**.
- A subtree consisting of the visited nodes is called the **pruned state space tree.**
- Backtracking Algorithm
  1. DFS of state space tree
  2. Checking each node is promising
  3. If it is non promising, backtracking to the node's parent and doing the search

Backtracking Algorithm

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

Example

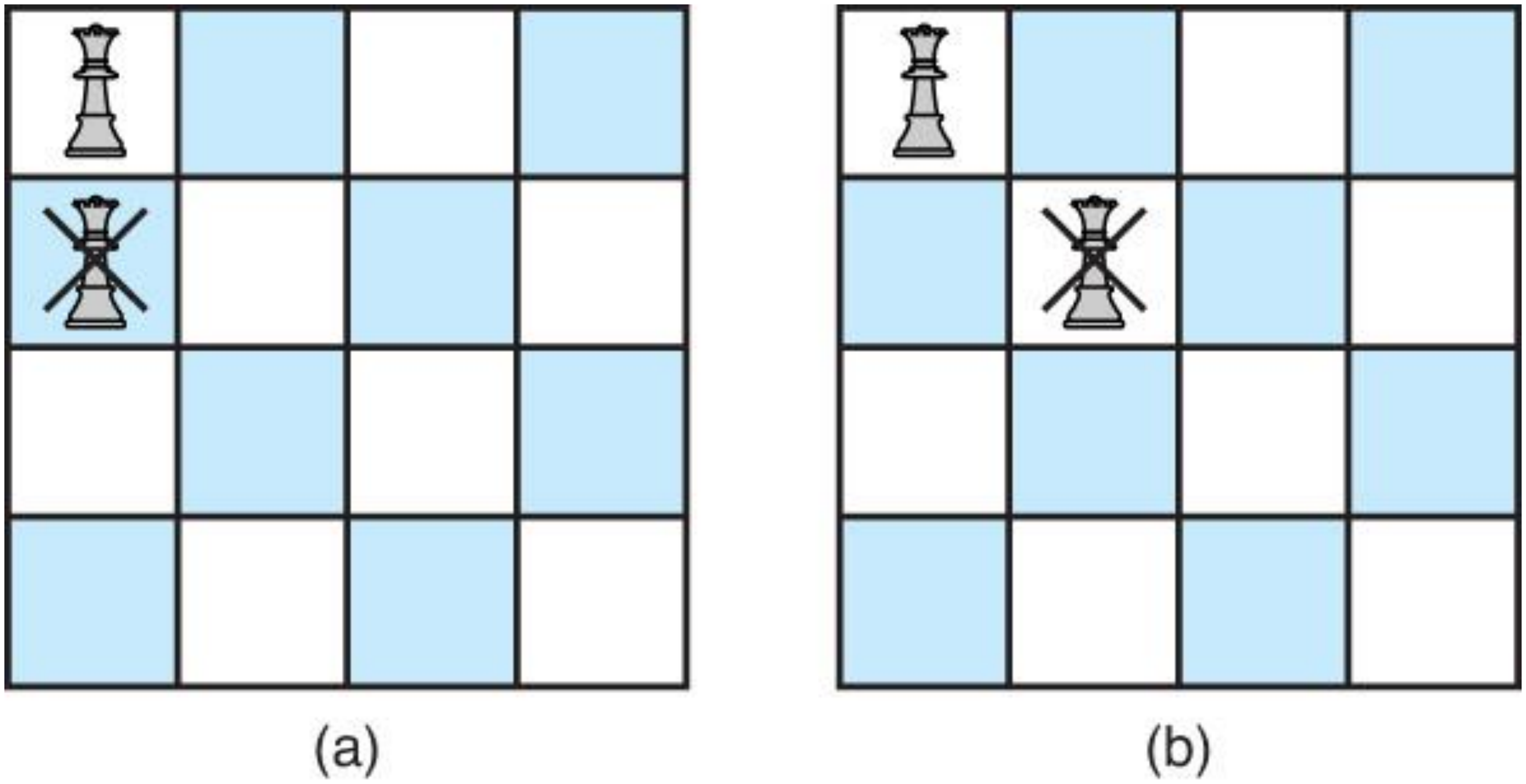


Figure 5.3: Diagram showing that if the first queen is placed in column 1, the second queen cannot be placed in column 1 (a) or column 2 (b).

Pruned State Space Tree of 4-Queens Problem

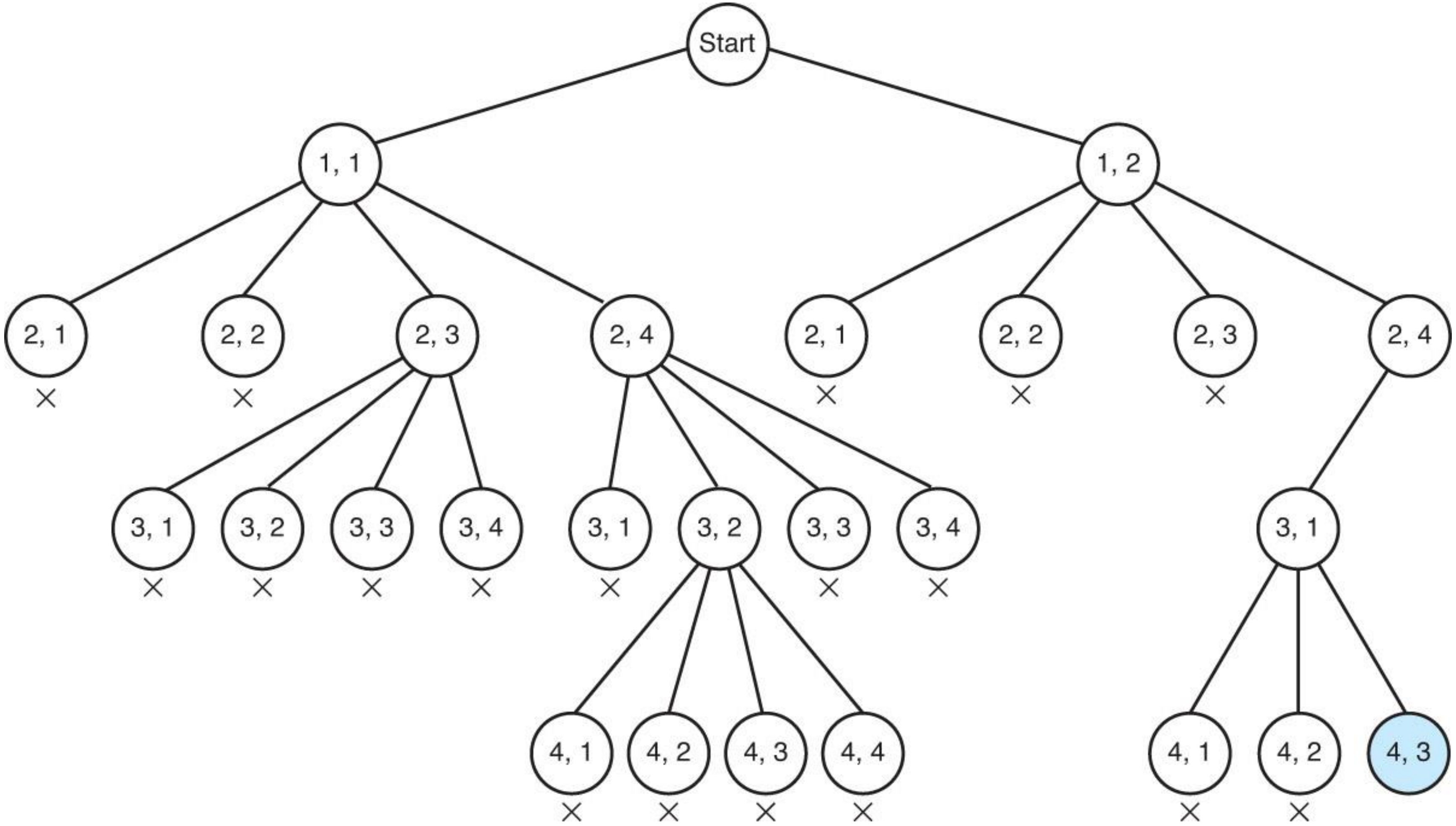


Figure 5.4: A protion of the pruned state space tree produced when backtracking is used to solve the instance of the *n*-Queens problems in which *n*=4.

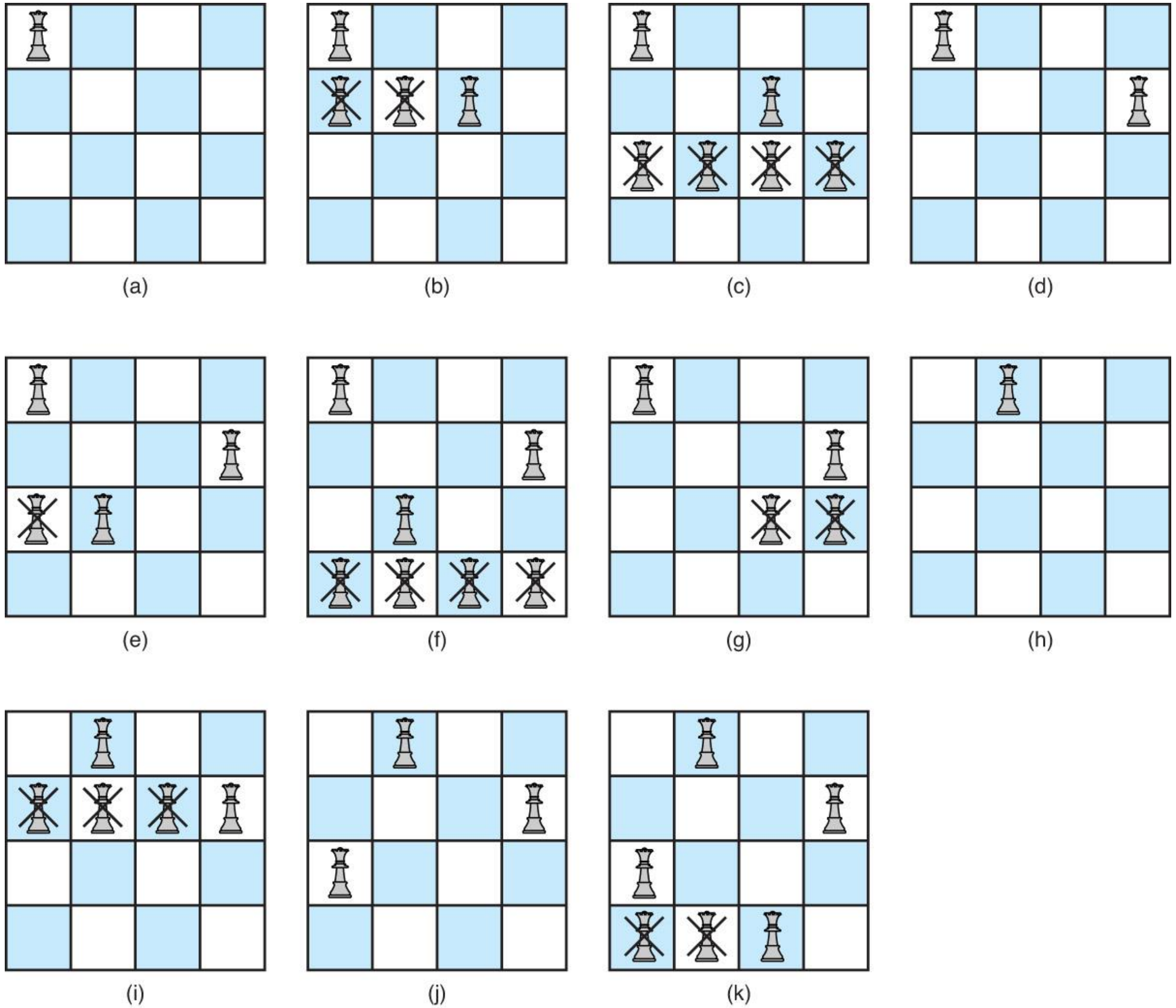


Figure 5.5: The actual chessboard positions that are tried when backtracking is used to solve the instance of the *n*-Queens problem in which *n*=4. Each nonpromising position is marked with a cross.

DFS vs. Backtracking

- Comparison of # of visited nodes
  - DFS = 155 nodes
  - Backtracking = 27 nodes

n-Queen Problem

n-Queens Problem

- In the *n*-Queens problem,
  - one way of eliminating a branch is if it would put two queens in the same column.
  - Another way of eliminating branches is to recognize that no two queens can be on the same diagonal.
- When we identify **a node as nonpromising**, we **go back to the parent** and try another branch.
- Eventually we will get to one or more leaf nodes and have several candidate sequences to evaluate.
- **Get to any leaf node** : **solution**.

n-Queens Problem

- Let *col (i)* be the column where the queen in the *i*th row is located, then to check whether the queen in the *k*th row is in the same column, we need to check whether

$$col\ (i) = col\ (k)$$

- Now to check the diagonals, consider this example:

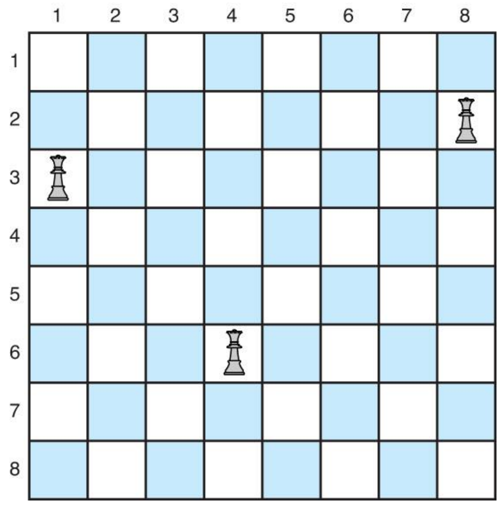


Figure 5.6: The queen in row 6 is being threatened in its left diagonal by the queen in row 3 and in its right diagonal by the queen in row 2.

n-Queens Problem

- In this example, 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 row2. Notice that

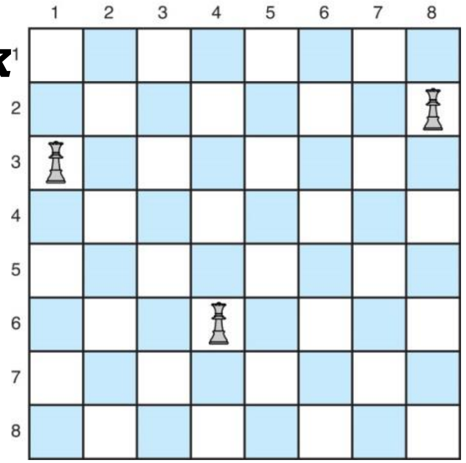
  col (6) - col (3) = 4 -1 = 3 = 6- 3.

- That is, for the queen threatening from the left, the difference in the columns is the same as the difference in the rows. Furthermore,

  col (6) - col (2) = 4 -8 = -4 = 2 - 6.

- That is, for the queen threatening from the right, the difference in the columns is the negative of the difference in the rows.
- Therefore, if the queen in the kth row threatens the queen in the ith row along one of its diagonals, then

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



n-Queens Problem

- **Criterion:**
- In the chessboard, two queens can not be in the same row, column or diagonal.

- *col*(*i*) : the column where the queen is in the *i*th row.

- In the same column:
- $col(i) = col(k)$
-
- In the diagonal:
- $col(i) - col(k) = i - k$ or $col(i) - col(k) = k - i$

Algorithm 5.1 : *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 *nxn* chessboard so that no two queens threaten each other. Each output consists of an array of integers col indexed from 1 to *n*, where *col*[*i*] is the column where the queen in the ith row is placed.

```
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 (i+1)st row can be
            col[i+1]=j;        // positioned in each of the n columns.
            queens(i+1);
         }
}
```

```
bool promising(index i){
   index k;
   bool switch;

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

Analysis of *n*-Queens Problem

- To determine the number of nodes checked as a function of n, the number of queens. Upper bound on the # of nodes in the pruned state space tree by **counting the # of nodes n the entire state space tree**. Root (level 0) 1 node, n nodes at level 1, n^2 nodes at level 2, … and n^n nodes at level n. The total # of nodes

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

n = 8, the state space tree contains

$$\frac{8^9-1}{8-1} = 19{,}173{,}961$$ nodes.

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

Analysis of *n*-Queens Problem

- To obtain an **upper bound on the # of promising nodes** using no two queens can ever be placed in the same column..
- Ex. $n = 8$. The first queen can be positioned in any of the eight columns. Once the first queen is positioned, the second can be positioned in at most seven columns; once the second is positioned, the third can be positioned in a t most six columns; and so on. There are at most

$$1 + 8 + 8 \times 7 + 8 \times 7 \times 6 + \ldots + 8!$$

$$= 109{,}601 \text{ promising nodes.}$$

- Generalizing this result to an arbitrary n, there are at most

$$1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n!$$

promising nodes.

Discussion

- This analysis does **not give us a very good idea** as to the efficiency of the algorithm for the following reasons:
  - Does not take into account the diagonal check in function promising. Therefore, there could be **far less promising nodes** than this upper bound.
  - Total # of nodes checked includes both promising and nonpromising nodes. The # of nonpromising nodes can be substantially greater than the # of promising nodes.

Discussion

- A straightforward way to determine the efficiency of the algorithm is to **actually run** the algorithm on a computer and **count how many nodes are checked**.
- Actually running an algorithm to determine its efficiency is **not really an analysis**. The purpose of an analysis is to determine ahead of time whether an algorithm is efficient.

Run Time

Table 5.1: An illustration of how much checking is saved by backtracking in the *n*-Queens problem[*]

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.

Usage Tips for using Backtracking

- The time spent in the promising function is a consideration in any backtracking algorithm.
- That is, our goal is not strictly to cut down on the number of nodes checked; rather, it is to improve overall efficiency.
- A very **time-consuming promising function** could offset the advantage of checking fewer nodes.

Using a Monte Carlo Algorithm to Estimate the Efficiency of a Backtracking Algorithm

Using a Monte Carlo Algorithm to Estimate the Efficiency of a Backtracking Algorithm

- The state space tree for a problem may contain an exponentially large or larger number of nodes.
- Given two instances with the same value of n, one of them may require that very few nodes be checked, whereas the other requires that the entire state space tree be checked.
- If we had an estimate of how efficiently a given backtracking algorithm would process a particular instance, we could decide whether using the algorithm on that instance was reasonable.
- We can obtain such an estimate using a Monte Carlo algorithm.

Probabilistic Algorithm

- Monte Carlo algorithms are probabilistic algorithms.
  - By a probabilistic algorithm, we mean one in which the **next instruction executed** is sometimes **determined at random** according to some probability distribution.
    - Unless otherwise stated, we assume that probability distribution is the uniform distribution.
  - By a deterministic algorithm, we mean one in which this cannot happen.
- A Monte Carlo algorithm estimates the expected value of a random variable, defined on a sample space.
- No guarantee the estimate is closed to the true expected value, but the probability that it is close increases as the time available to the algorithm increases
- Generate a "**typical path**" in the tree consisting of the nodes the would be checked, then **estimate the number of nodes** in this tree from the path.

Monte Carlo Algorithm

- Monte Carlo Algorithm requires the analyzed algorithm to satisfy the following two conditions:
  1. The **same promising function** must be used on all nodes at the **same level** in the state space tree.
  2. Nodes at the **same level** in the state space tree must **have the same number of children**.

Monte Carlo Algorithm

1. Let m_0 be the number of promising children of the root.
2. Randomly generate a promising node at level 1. Let m_1 be the number of promising children of this node.
3. Randomly generate a promising child of the node obtained in the previous step. Let m_2 be the number of promising children of this node. :
4. Randomly generate a promising child of the node obtained in the previous step. Let m_i be the number of promising children of this node.

5. This process continues until no promising children are found.

Monte Carlo Algorithm

- Because we assume that nodes at the same level in the state space tree all have the same number of children, m_i is an estimate of the average number of promising children of nodes at level i. Let

  t_i = total # of children of a node at level i.

- Because all t_i children of a node are checked and only the m_i promising children have children that are checked, an estimate of the **total # of nodes checked by the backtracking algorithm** to find all solutions is given by

$$1 + t_0 + m_0 t_1 + m_0 m_1 t_2 + \cdots + m_0 m_1 \cdots m_{i-1} t_i + \cdots$$

Algorithm 5.2 : Monte Carlo Estimate

- Problem : Estimate the efficiency of a backtracking algorithm using a Monte Carlo algorithm.

  Inputs : an instance of the problem that the backtracking algorithm solves.

  Output : an estimate of the number of nodes in the pruned state space tree produced by the algorithm.

```
int estimate ( )
{   node  v;
    int m, mprod, t, numnodes;
    v = root of state space tree;
    numnodes = 1;
    m = 1;
    while ( m!= 0)
        {   t = number of children of v;
            mprod = mprod*m;
            numnodes = numnodes + mprod*t;
            m = number of promising
                   children of v;
            if ( m!= 0)
                v = randomly selected
                      promising child of v;
        }
    return numnodes;
}
```

Algorithm 5.3 : Estimate for Algorithm 5.1

- Problem : Estimate the efficiency of Algorithm 5.1.

  Inputs : positive integer n.

  Output : an estimate of the number of nodes in the pruned state space tree produced by the algorithm.

```
int estimate_n_queens (int n)
{   index  i, j, col[1..n];
    int m, mprod, numnodes;
    set_of_index prom_children;
    i = 0;
    numnodes = 1;
    m = 1;
    while ( m != 0 && i != n )
        {   mprod = mprod*m;
            numnodes += mprod*t;
            i ++;
            m = 0;
            prom_children = ∅;
            for (j = 1; j <= n; j++)
              {  col[i] = j;
                 if ( promising(i))
                   {  m ++;
                      prom_children =
                          prom_children ∪{j};} }
            if ( m!= 0)
              { j = randomly selection from
                              prom_children;
                col[i] = j; }
        }
    return numnodes;
}
```

The Sum-of-Subsets Problem

The Sum-of-Subsets Problem

In the knapsack problem, if the profit of each item is the same, then the goal is to maximize the total weight while not exceed W.

So the thief might first try to determine whether there was a set whose total weight equaled W.

The problem of determining such sets is called the **Sum-of-Subsets** Problem.

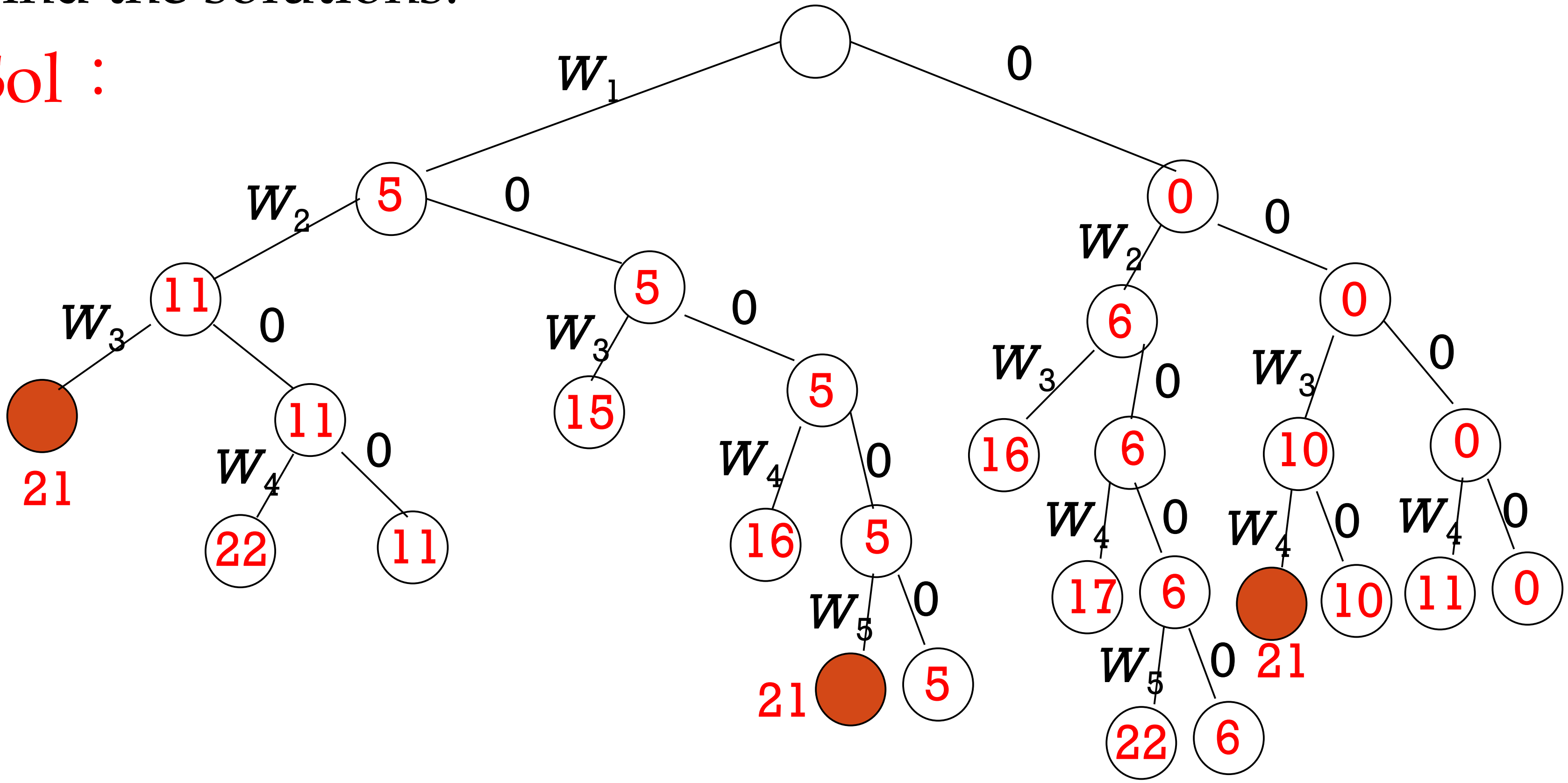
Example 5.2

Suppose that $n = 5$, $W = 21$, and

$w_1 = 5$, $w_2 = 6$, $w_3 = 10$, $w_4 = 11$, and $w_5 = 16$.

Find the solutions.

Sol：



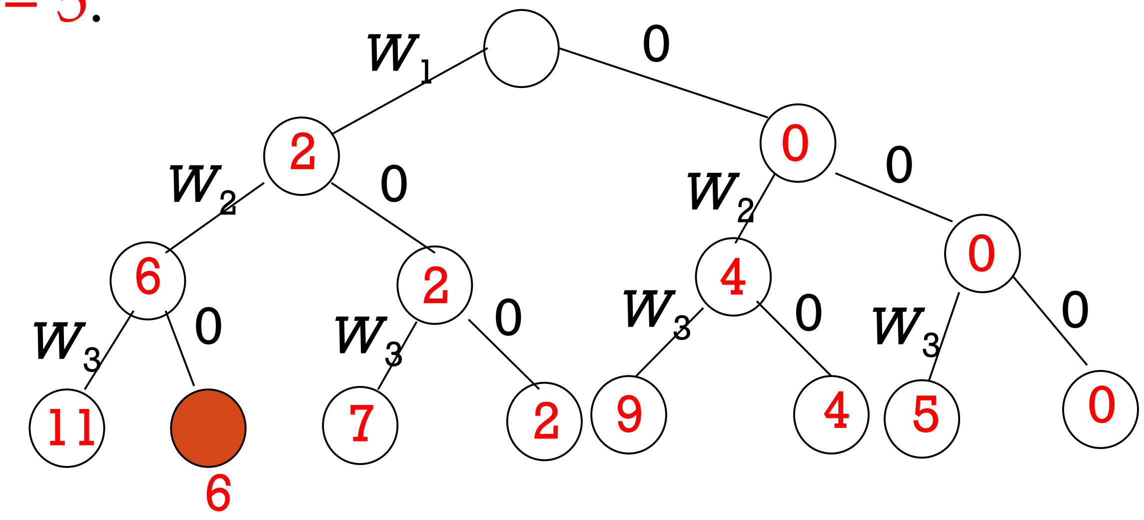
Example 5.3

Suppose that $n = 3$, $W = 6$, and

$w_1 = 2$, $w_2 = 4$, $w_3 = 5$.

Find the solutions.

Sol :



Example 5.4

Suppose that $n = 4$, $W = 13$, and

$w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$.

Find the solutions.

Sol ： For the weights sorting in nondecreasing order,

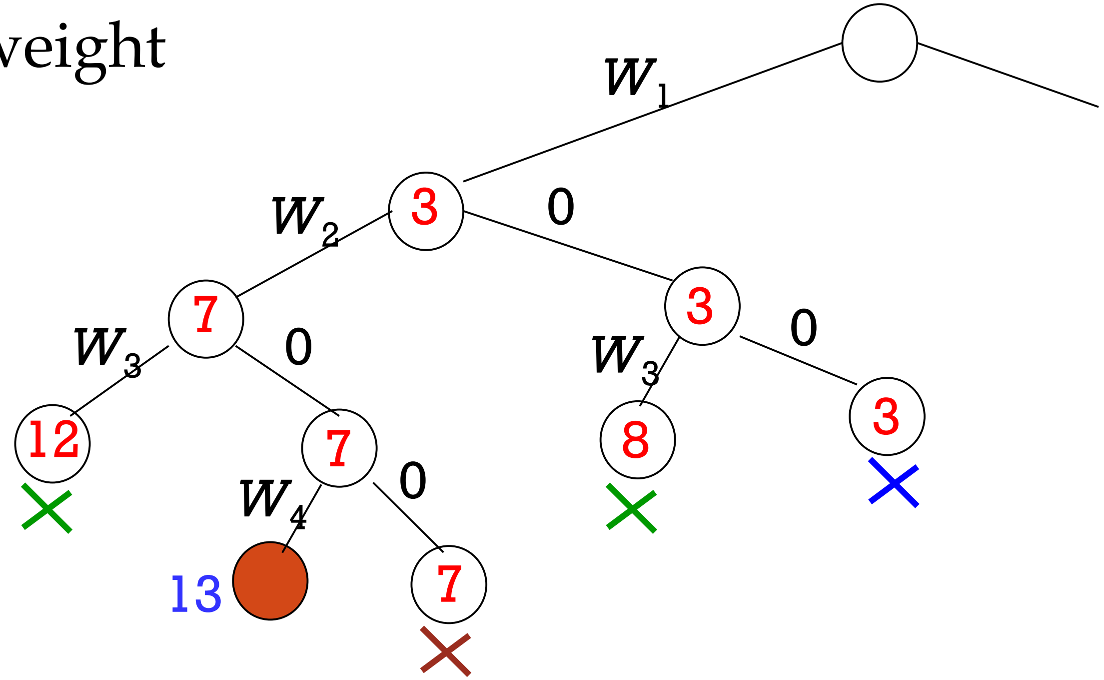
a node is nonpromising if $weight + w_{i+1} > W$

where *weight* is the total weight

up to a node at level *i*.

$weight + total_r < W$

is also nonpromising.



Algorithm 5.4

- **Problem**：Given n positive weights and a positive integer W, find all combinations of the weights that sum to W.

  Inputs：positive integer n, sorted array w index from 1 to n, and a positive integer W.
  Output： all combinations of the weights 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";
                 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]); }
}
 bool promising (index  i)
 {   return  (weight+total >= W) && (weight =W || weight+w[i+1]<=W); }
```

Analysis of Algorithm 5.4

Worst-Case Time Complexity：

Number of nodes in the state space tree searched

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

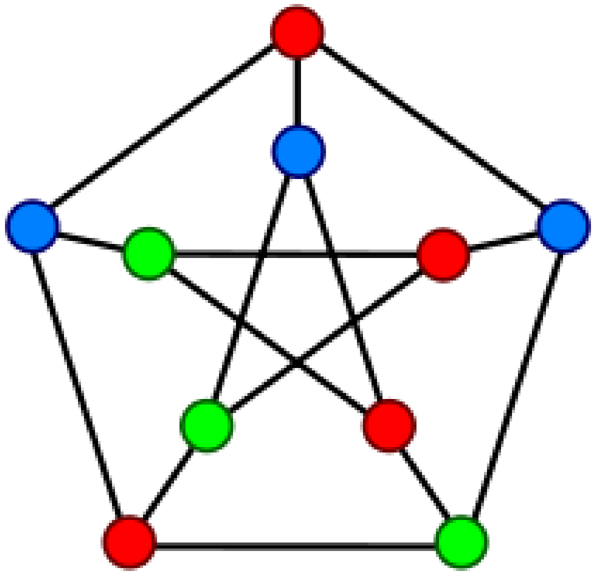
If $\sum_{i=1}^{n-1} w_i < W$ and $w_n = W$,

it needs an exponentially large number of nodes to be visited.

Graph Coloring

Graph Coloring

- m-Coloring Problem:
  - Find all ways to color an undirected graph using at most *m* color, so that no two adjacent vertices are the same color.



Planar Graph

- It can be drawn in a plane in such a way that no two edges cross each other.
- Each region in the map is represented by a vertex. If one region is adjacent to another region, we join their corresponding vertices by an edge.

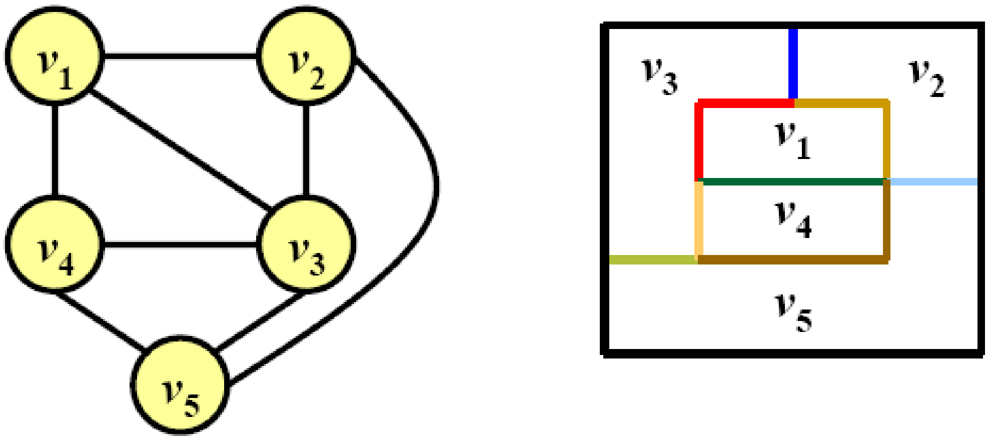
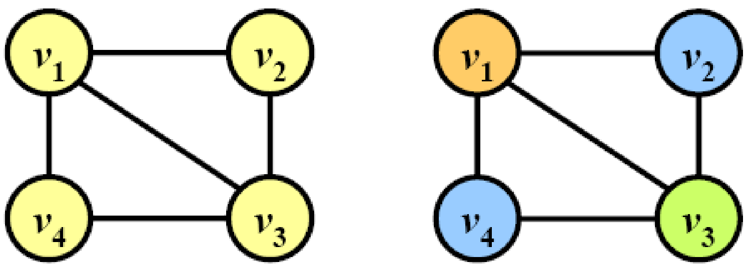


Figure 5.11: Map (top) and its planar graph representation (bottom)

Example 5.5



Find all ways to color the 4 vertices graph.

Sol : If m = 2, no solution.

If m = 3, (v_1: color 1), (v_2, v_4: color 2), (v_3: color 3);

(v_1: color 1), (v_2, v_4: color 3), (v_3: color 2);

(v_1: color 2), (v_2, v_4: color 1), (v_3: color 3);

(v_1: color 2), (v_2, v_4: color 3), (v_3: color 1);

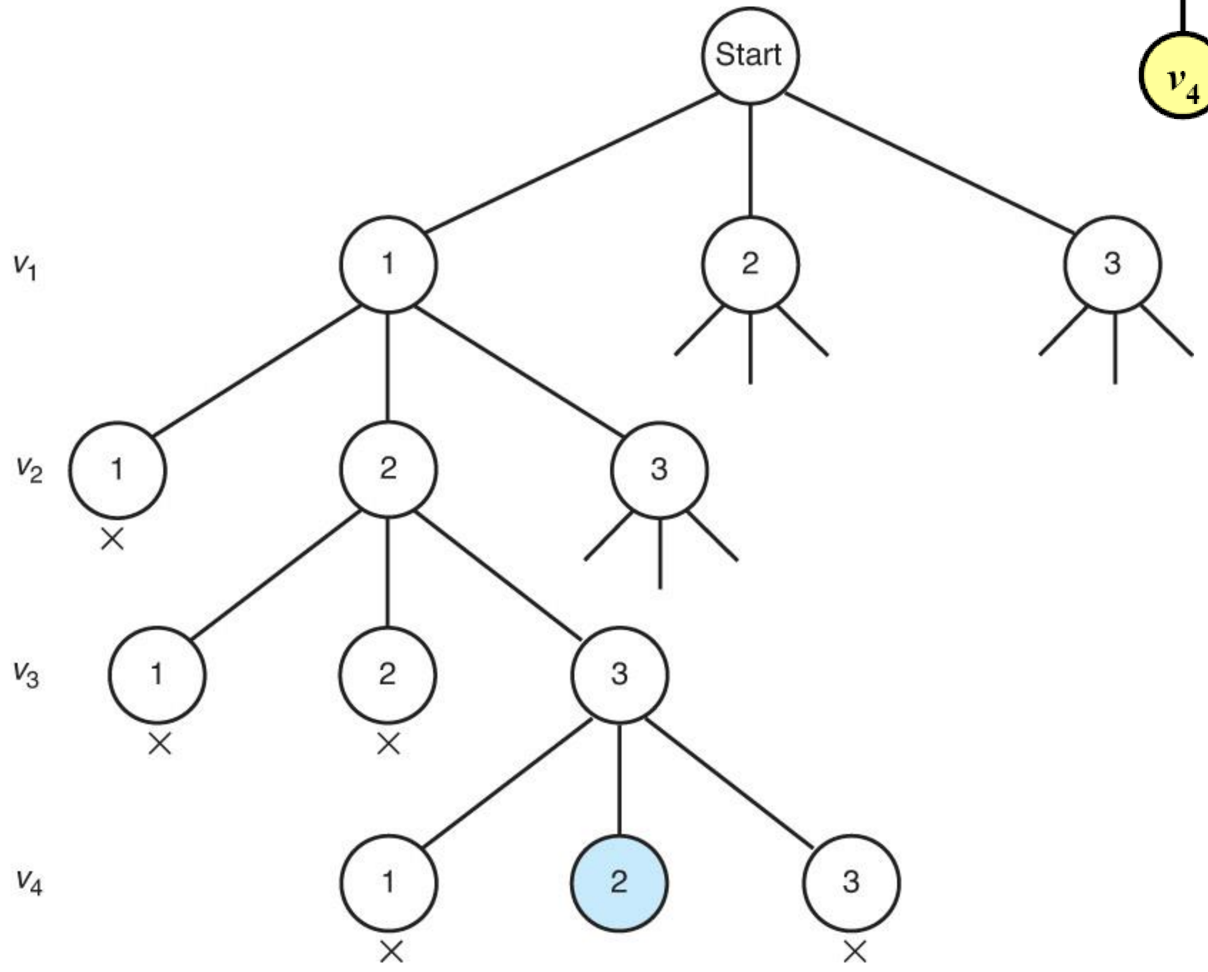
(v_1: color 3), (v_2, v_4: color 1), (v_3: color 2);

(v_1: color 3), (v_2, v_4: color 2), (v_3: color 1).

If m = 4, (v_i: color i), for i = 1 to 4, total 4! = 24.

So what we concerned is the case $m < N$, the number of vertices.

Graph Coloring : State Space Tree



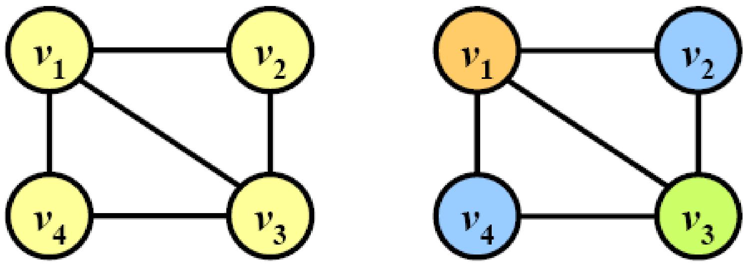


Figure 5.12: A protion of the pruned state space tree produced using backtracking to do a 3-coloring of the graph in figure 5.10.

Algorithm 5.5

- Problem : Find all ways for the *m*-Coloring Problem.
  Inputs : positive integer *n*, *m*, and an adjacent matrix *W*.
  Output : an array *vcolor*, where *vcolor*[*i*] is the color of vertex *i*.

```
void m_coloring (index i)
{ int  color;
  if (promising(i));
   if  (i=n);
     cout << vcolor[1] through vcolor[n];
   else
     for (color=1; color<=m; color++)
         { vcolor[i+1]=color;
             m_coloring(i+1);  }
}
```

```
bool promising(index i)
{  index  j;
   bool switch;
   switch = true;
   j = 1;
    while (j < i && switch)
       {  if  (W[i][j] &&
                      vcolor[i]=vcolor[j])
                 switch = false;
            j++;  }
    return switch;
}
```

Analysis of Algorithm 5.5

Worst-Case Time Complexity：

Number of nodes in the state space tree searched

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

54

The Hamiltonian Circuits Problem

The Hamiltonian Circuits Problem

Hamiltonian Circuit (Tour)

A path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex.

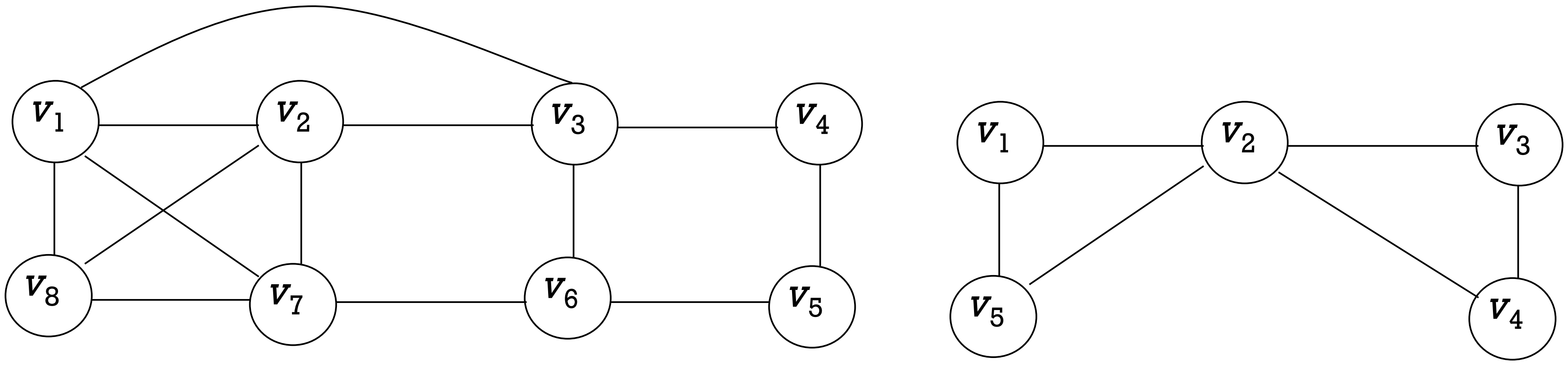


Figure 5.13: The graph in (a) contains the Hamiltonian Circuit; the graph in (b) contains no Hamiltonian Circuit.

State Space Tree

- Put the starting vertex at level 0 in the tree; call it the zeroth vertex on the path.
- At level 1, consider each vertex other than the starting vertex as the first vertex after the starting one.
- At level 2, consider each of these same vertices as the second vertex, and so on.
- Finally, at level n-1, consider each of these same vertices as the (n-1)st vertex.

Backtrack in the State Space Tree

1. The ith vertex on the path must be adjacent to the (i–1)st vertex on the path.
2. The (n–1)st vertex must be adjacent to the 0th vertex (the starting one).
3. The ith vertex cannot be one of the first (i–1) vertices.

Algorithm 5.6

- Problem : Find all Hamiltonian Circuits for the graph.

  Inputs : an undirected graph with n vertices, and an adjacency matrix W.

  Output : an array *vindex*, where *vindex*[*i*] is the index of the *i*-th vertex on the path.

```
void hamiltonian (index i)
{ index  j;
   if (promising(i))
     if  (I = n–1);
        cout << vindex[0] through
                            vindex[n–1];
     else
        for (j = 2; j <= n; j++)
           {  vindex[i+1]= j;
              hamiltonian(i+1);  }
}
```

```
bool promising(index i)
{  index  j;        bool switch;
   if  (i=n–1 &&! W[vindex[n–1]][vindex[0]]);
        switch = false;
   else if (i>0 &&! W[vindex[i–1]][vindex[i]]);
        switch = false;
   else  { switch = true;
           j = 1;
           while (j < i && switch)
              {  if  (vindex[i] = vindex[j])
                     switch = false;
                 j++;  } }
   return switch;
}
```

Analysis of Algorithm 5.6

Worst-Case Time Complexity：

Number of nodes in the state space tree searched

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

The 0-1 Knapsack Problem

The 0-1 Knapsack Problem

Backtracking for the 0-1 Knapsack 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);
}
```

Nonpromising Nodes

weight : the sum of the weights of the items that have been included up to some node. $\Rightarrow$ $weight \geq W$

profit : the sum of the profits of the items included up to some node.

Initialization: *bound* = *profit*, *totweight* = *weight*

If $bound \leq maxprofit$

$\Rightarrow$ *Nonpromising*

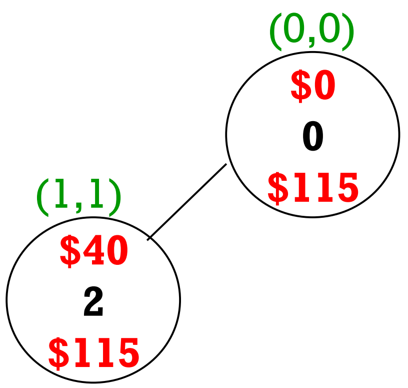
$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

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

Example 5.6

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

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



Find the solutions.

Sol :

1. Set $maxprofit = 0$
2. Visit node(0,0)

   $profit = 0$

   $weight = 0$

   $bound = 0+40+30+(16-7)50/10=115$

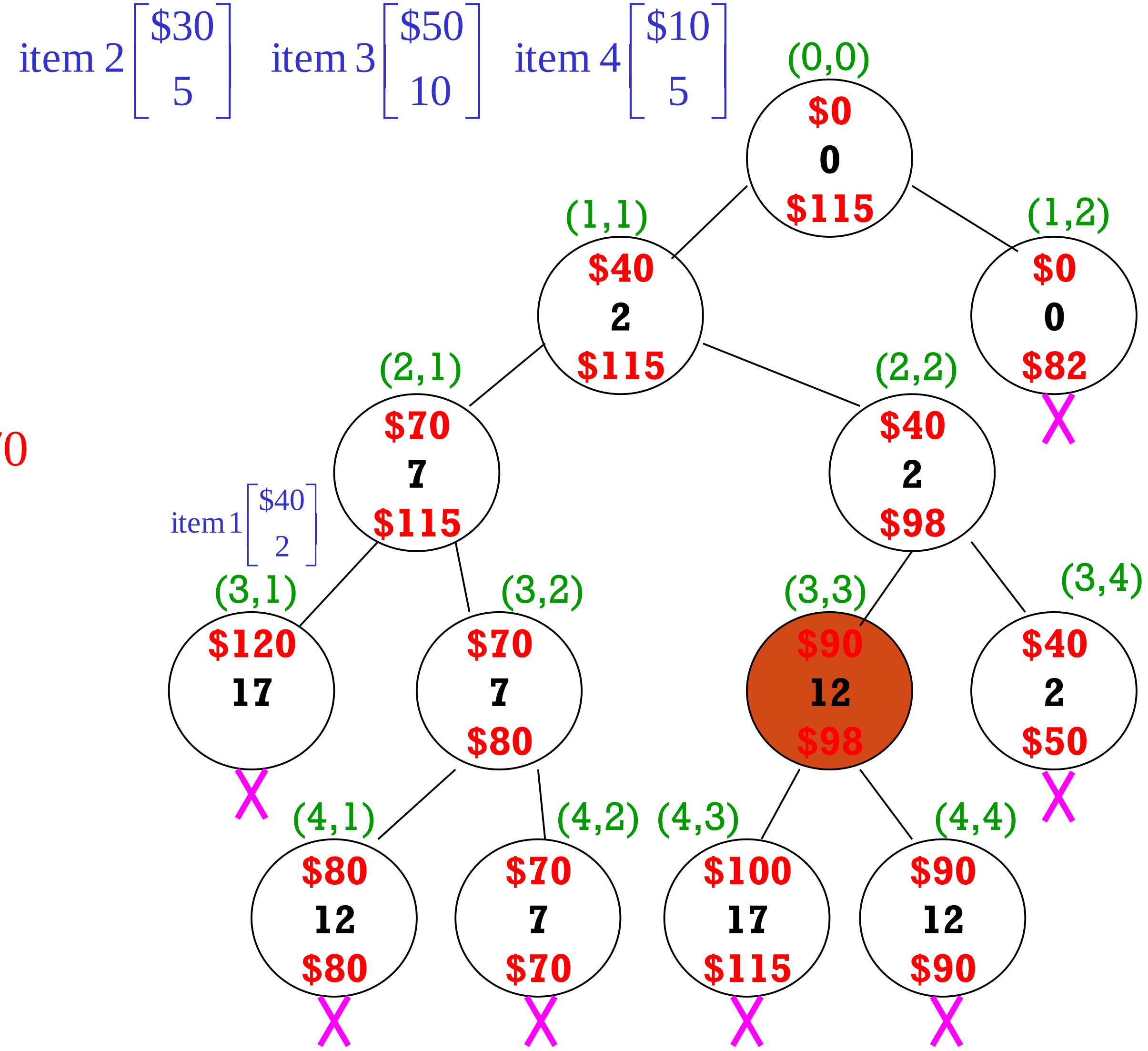
3. Visit node (1,1) : $profit = 0+40=40$ $weight = 0+2=2$ maxprofit=40

   bound=40+30+(16-7)50/10=115

Example 5.6 (Cont'd)

n = 4, W=16,

4. Visit node(2,1)
   - *profit* = 40+30=70
   - *weight* = 2+5=7
   - *maxprofit :*
     - *profix(70)>* 40(maxprofit)=>70
   - *bound* = 70+(16-7)50/10=115
5. Visit node (3,1)
   - *profit* = 70+50=120
   - *weight* = 7+10=17 >W : false
   - *maxprofit : no change*
   - *bound* : nonpromising-no computing



Algorithm 5.7

- Problem：Given n positive weights and a positive integer W, find all combinations of the weights that sum to W.

  Inputs：positive integer *n*, sorted array w index from 1 to n, and a positive integer W.

  Output： all combinations of the weights that sum to W.

```
void knapsack (index  i, int  profit, int  weight)
{  if  (weight <= W  && profit > maxprofit)
      {  maxprofit = profit;
         numbest = i ;
         bestset = include;}
   if  (promising(i))
      {   include[i+1] = "yes";
         knapsack(i+1, profit+p[i+1], weight+w[i+1]);
         include[i+1] = "no";
         knapsack(i+1, profit, weight); }     }
```

Algorithm 5.7

```
bool promising (index  i)
{   index  j, k;
    int  totweight;
    float bound;
    if  (weight >= W )
         return false;
   else {  j = i + 1;
           bound = profit ;
           totweight = weight;
           while (j <= n && totweight+w[j] <= W)
               {   totweight = totweight + w[j];
                   bound = bound + p[j];
                   j ++; }
           k = j;
           if  (k <= n)
                bound = bound + (W – totweight)*p[k]/w[k];
           return bound > maxprofit;  }       }
```