Backtracking

V. Balasubramanian

Introduction

- introduce two algorithm design techniques—***backtracking*** and ***branch-and-bound***—that often make it possible to solve at least
- some large instances of difficult combinatorial problems. Both strategies can be
- considered an improvement over exhaustive search,
- The name first coined by D.H. Lehmer

Tackling Difficult Combinatorial Problems

There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):

- Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time

- Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

Exact Solution Strategies

- *exhaustive search* (brute force)
  - useful only for small instances

- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)

- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential

- *branch-and-bound*
  - further refines the backtracking idea for optimization problems

Backtracking

- Construct the ***state-space tree***
  - nodes: partial solutions
  - edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- "Prune" ***nonpromising nodes***
  - dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

- Basic idea of backtracking
- • Desired solution expressed as an *n-tuple* ($x_1,x_2,\ldots,x_n$) where x_i are chosen from
- some set S_i
- – If $|S_i|=m_i$, $m=m_1m_2..m_n$ candidates are possible
- • *Brute force* approach
- – Forming all candidates, evaluate each one, and saving the optimum one

- Backtracking
- •Yielding the same answer with far fewer than m trials
- “Its basic idea is to build up the solution vector one component at a time and to use modified criterion function Pi(x1,..,xi) (sometimes called
- *bounding function*) to test whether the vector being formed has any chance of success. The major advantage is: if it is realized that the partial vector (x1,..,xi) can in no way lead to an optimum solution, then
- mi+1…mn possible test vectors can be ignored entirely.”
- Based on depth-first recursive search

– Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set.

- example

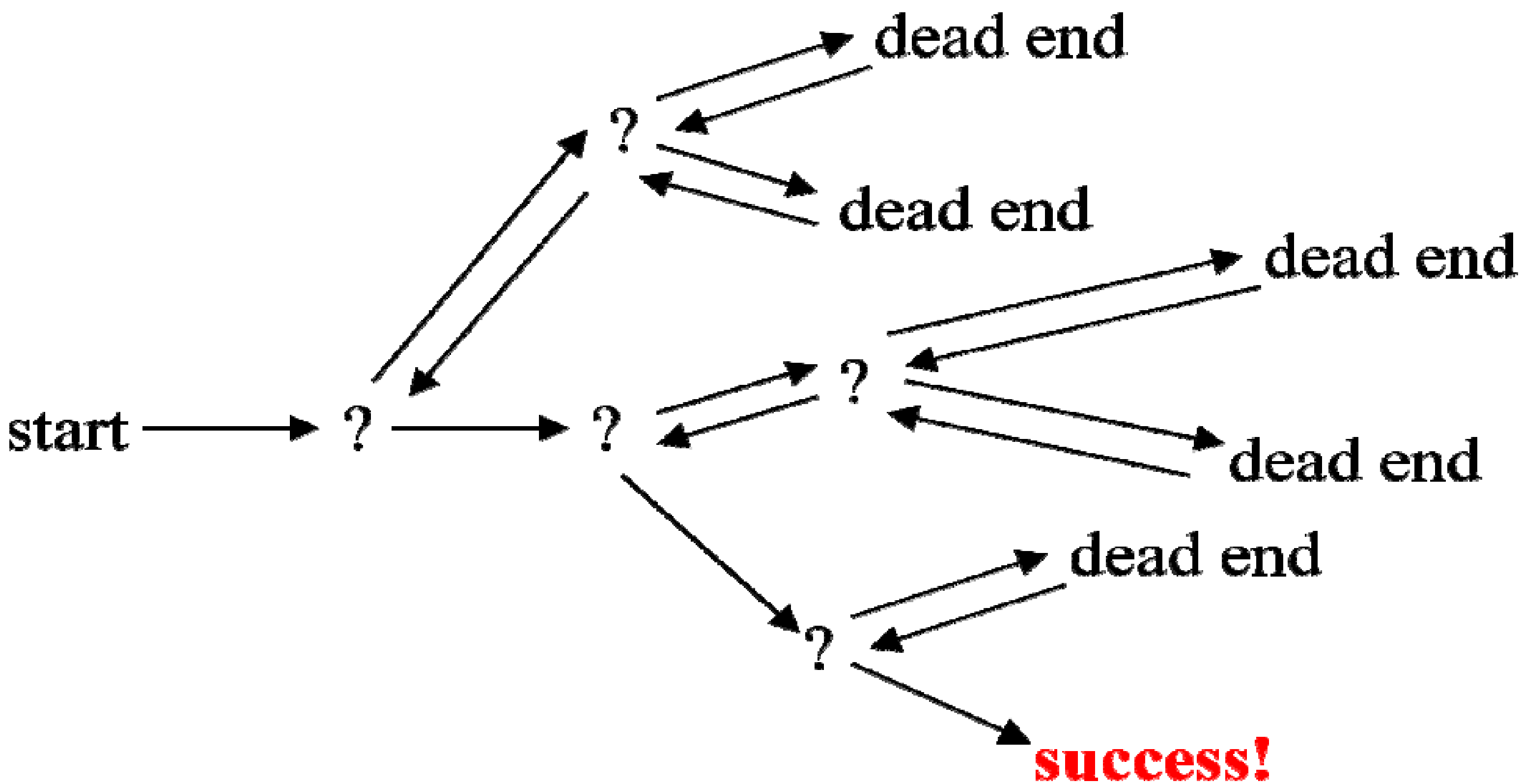
  $x_i \geq 0$ or S_i = { all nonnegative real numbers }

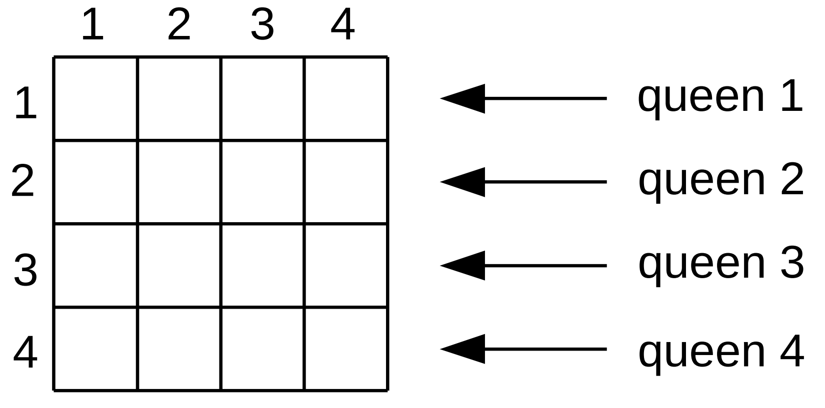
  $x_i = 0$ or 1 or $S_i = \{ 0, 1 \}$

  $l_i \leq x_i \leq u_i$ or $S_i = \{a : l_i \leq a \leq u_i \}$

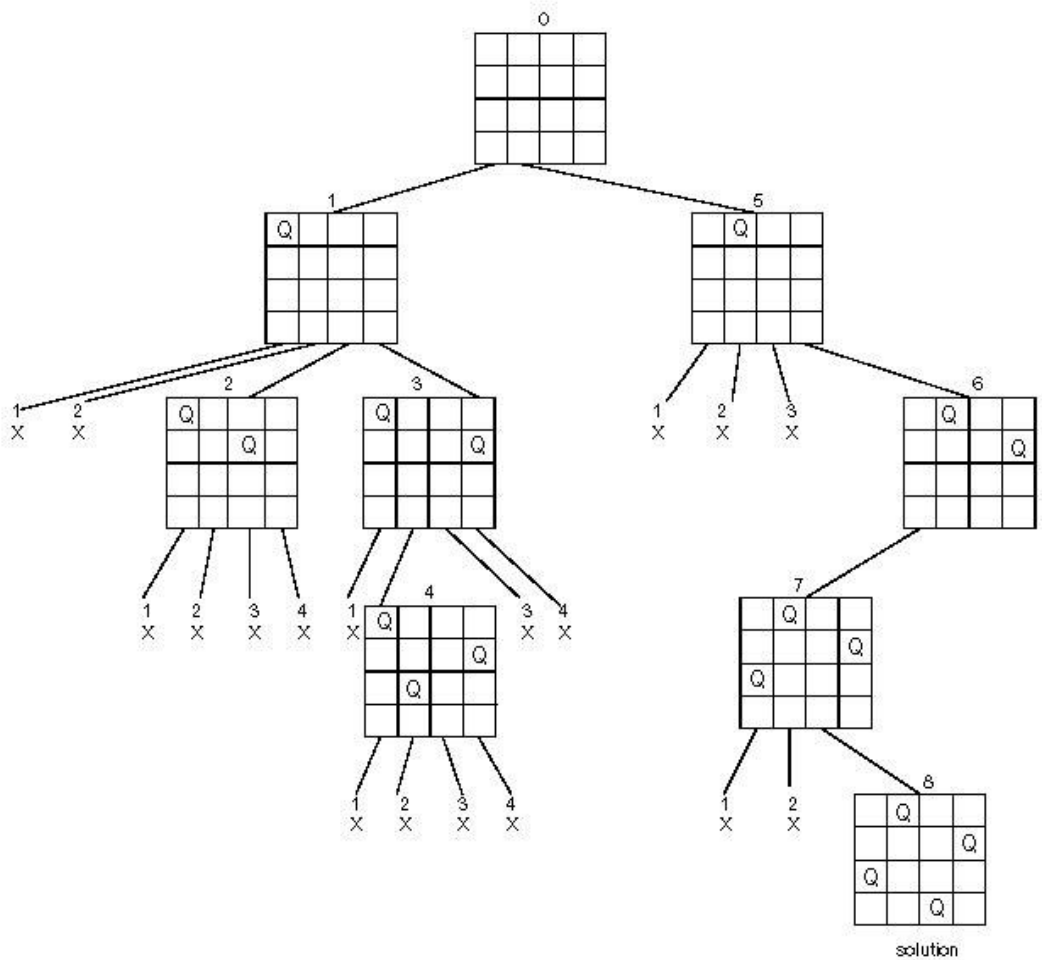
- All tuples satisfying the explicit constraints define a possible *solution space* for I (I=problem instance)

– Definition 2: The implicit constraints are rules that determine which of the tuples in the solution space of *I* satisfy the criterion function. Thus implicit constrains describe the way in which the x_i must relate to each other.


dead end
?
dead end
dead end
start
?
?
?
dead end
dead end
?
success!


1
2
3
4
1
2
3
4
queen 1
queen 2
queen 3
queen 4

State Space tree



ALGORITHM *Backtrack*$(X[1..i])$

//Gives a template of a generic backtracking algorithm
//Input: $X[1..i]$ specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if $X[1..i]$ is a solution **write** $X[1..i]$
else //see Problem 8 in the exercises
  for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints **do**
    $X[i+1] \leftarrow x$
    Backtrack$(X[1..i+1])$

- N-queens:

```
void NQueens(int k, int n)
  // Using backtracking, this procedure prints all
  // possible placements of n queens on an nXn
  // chessboard so that they are nonattacking.
  {
     for (int i=1; i<=n; i++) {
        if (Place(k, i)) {
           x[k] = i;
           if (k==n) { for (int j=1;j<=n;j++)
                          cout << x[j] << ' '; cout << endl;}
           else NQueens(k+1, n);
        }
     }
  }
```

```
bool Place(int k, int i)
    // Returns true if a queen can be placed in kth row and
    // ith column. Otherwise it returns false. x[] is a
    // global array whose first (k-1) values have been set.
    // abs(r) returns the absolute value of r.
    {
        for (int j=1; j < k; j++)
            if ((x[j] == i)  // Two in the same column
                || (abs(x[j]-i) == abs(j-k)))
                                    // or in the same diagonal
                return(false);
        return(true);
    }
```

The idea

- Maze of hedges by Hampton Court Palace
- A sequence of objects is chosen from a specified set so that the sequence satisfies some criterion
- Example: *n*-Queens problem
  - Sequence: *n* positions on the chessboard
  - Set: n^2 possible positions
  - Criterion: no two queens can threaten each other
- Depth-first search of a tree (preorder tree traversal)

Depth first search

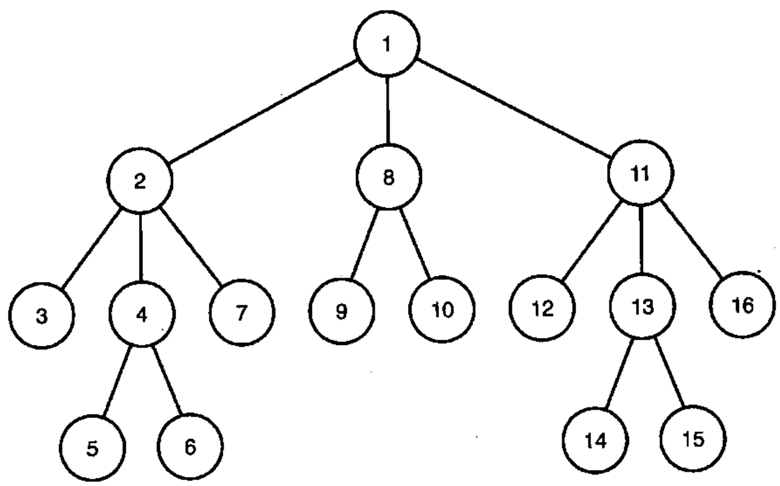


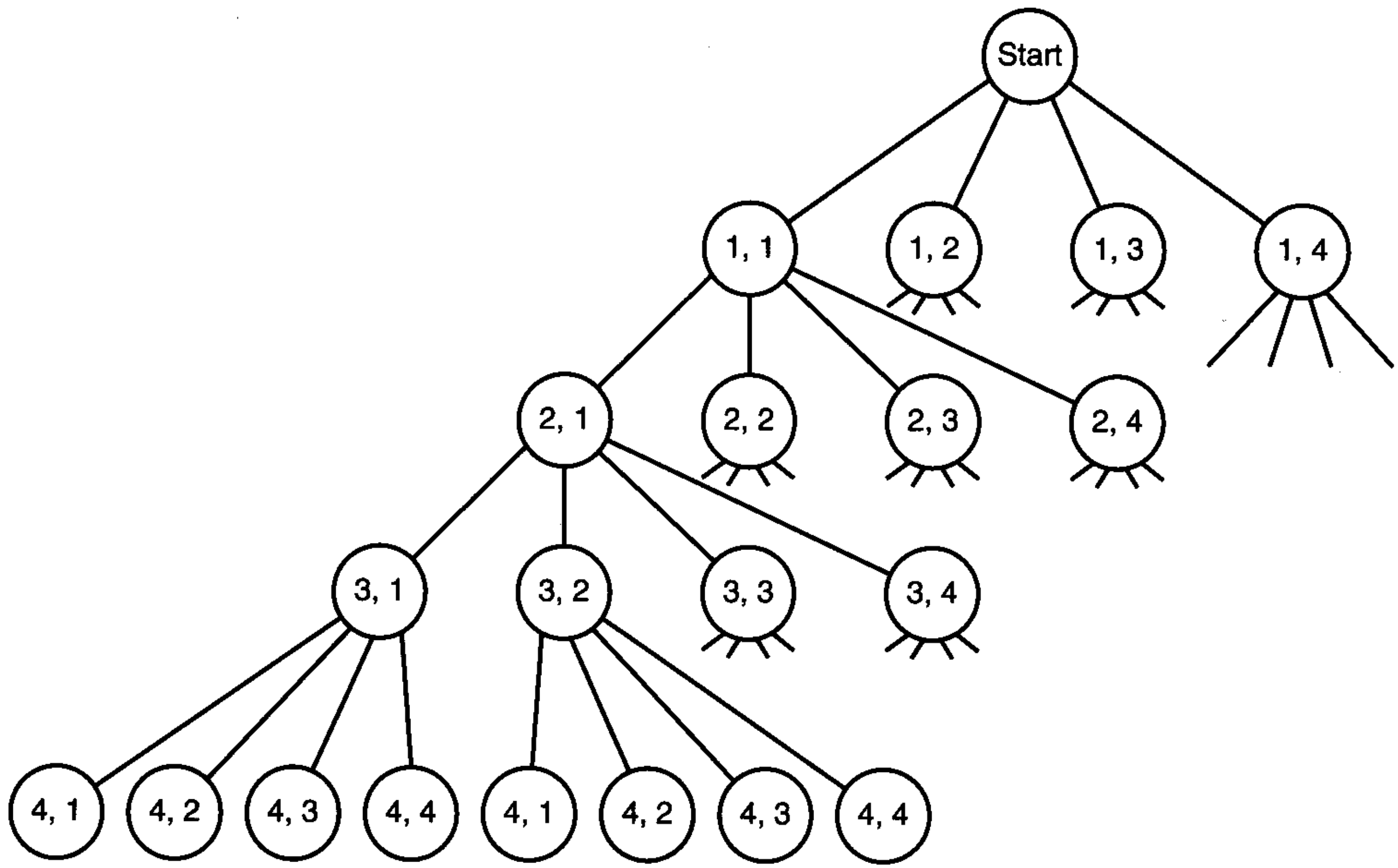
Figure 5.1 ● A tree with nodes numbered according to a depth-first search.

The algorithm

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

4-Queens problem

- State space tree



If checking each candidate solution …

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

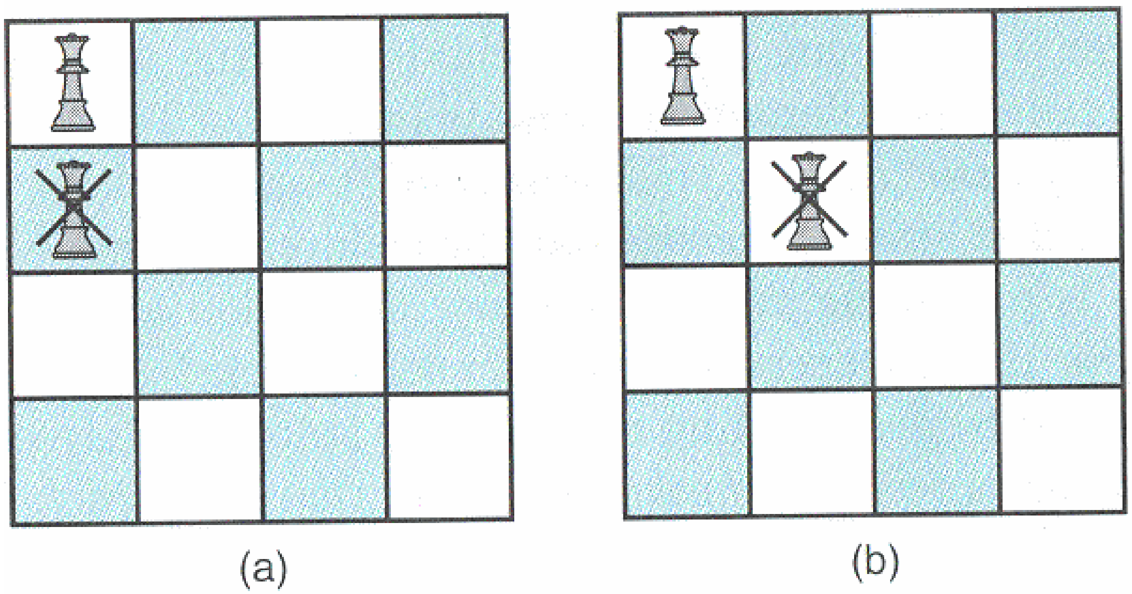
$$[<1,1>,<2,1>,<3,1>,<4,2>]$$

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

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

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

Looking for signs for dead ends



Backtracking

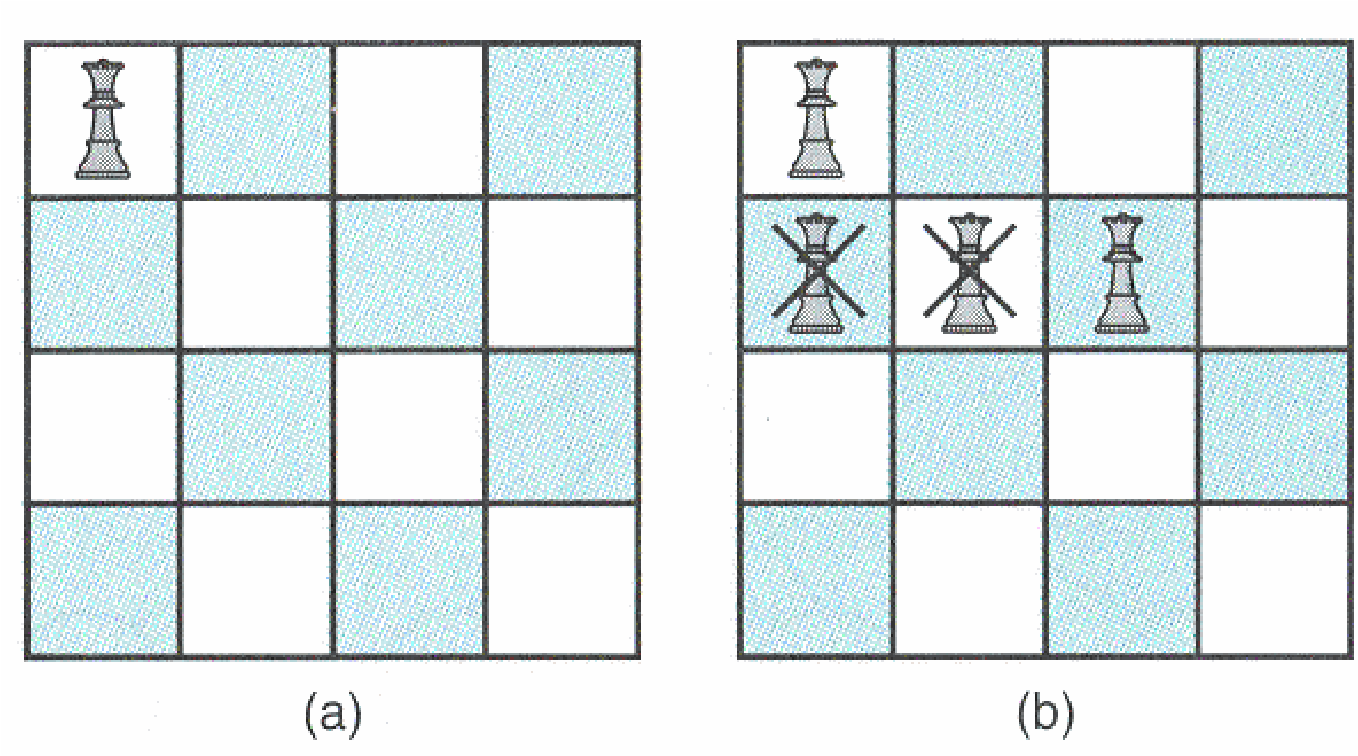
- Nonpromising node
- Promising node
- Promising function
- Pruning the state space tree
- Pruned state space tree

The generic algorithm

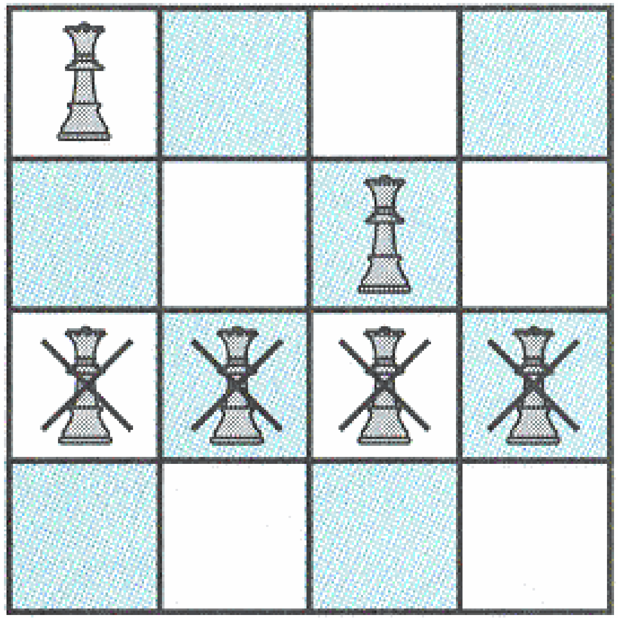
```
void checknode (node v)
{
  node u;

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

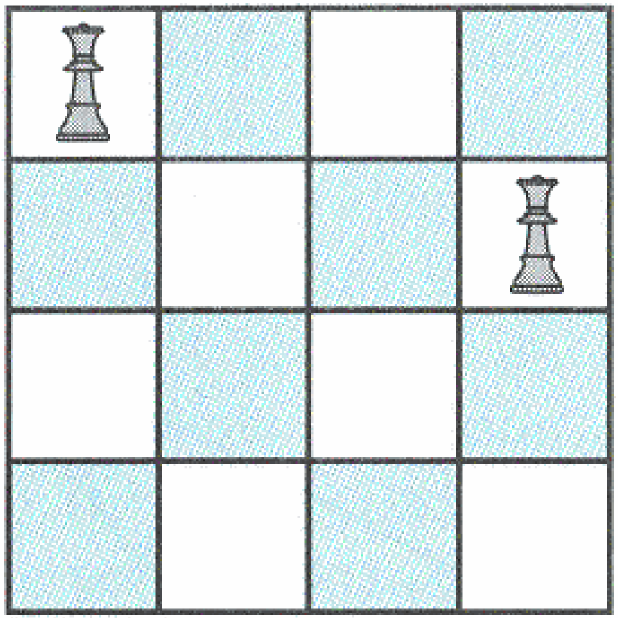
4-Queens problem (1)



4-Queens problem (2)

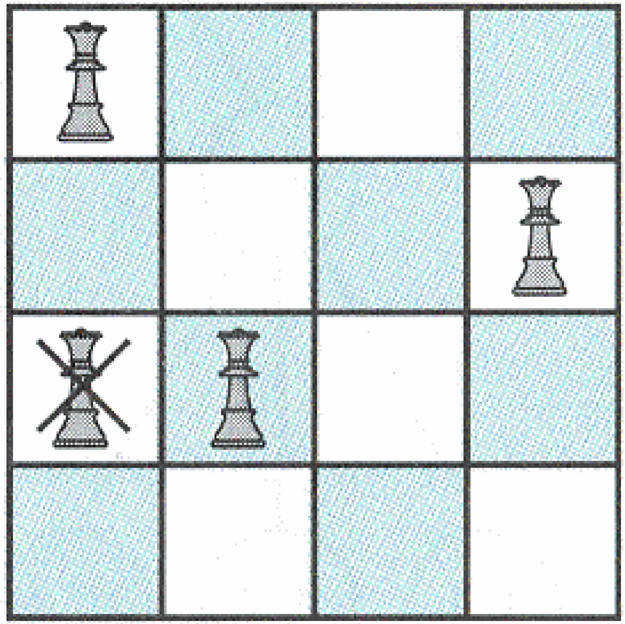


(c)

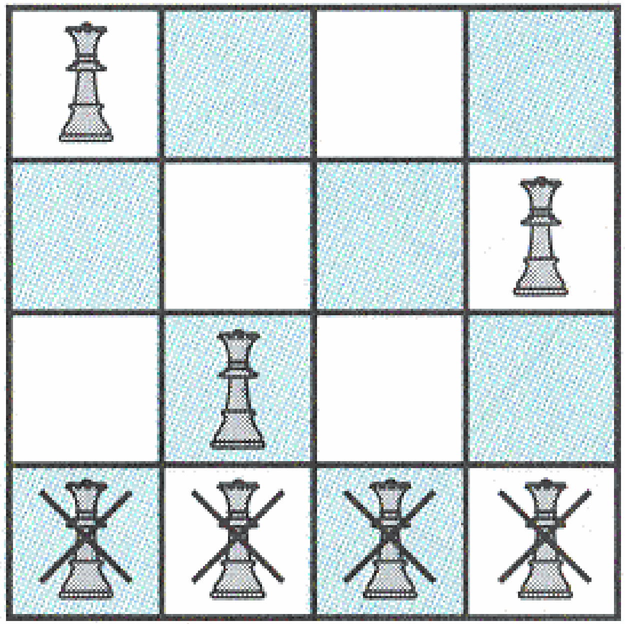


(d)

4-Queens problem (3)

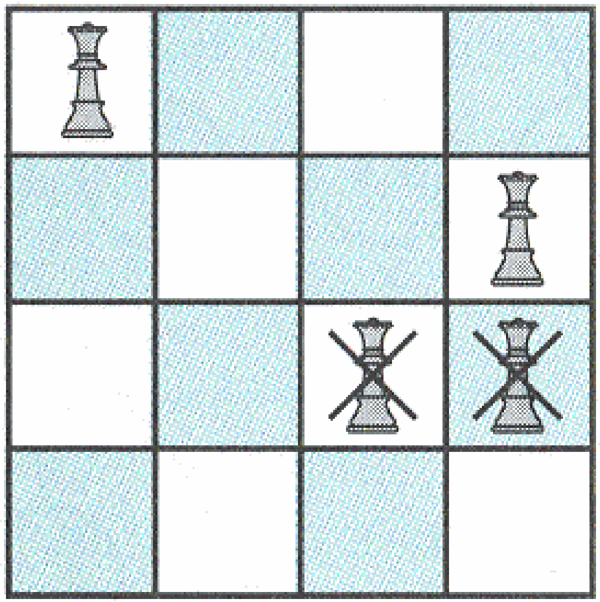


(e)

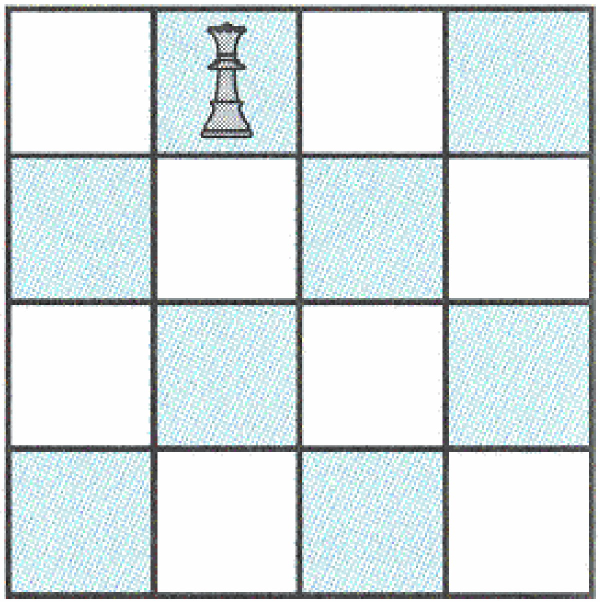


(f)

4-Queens problem (4)

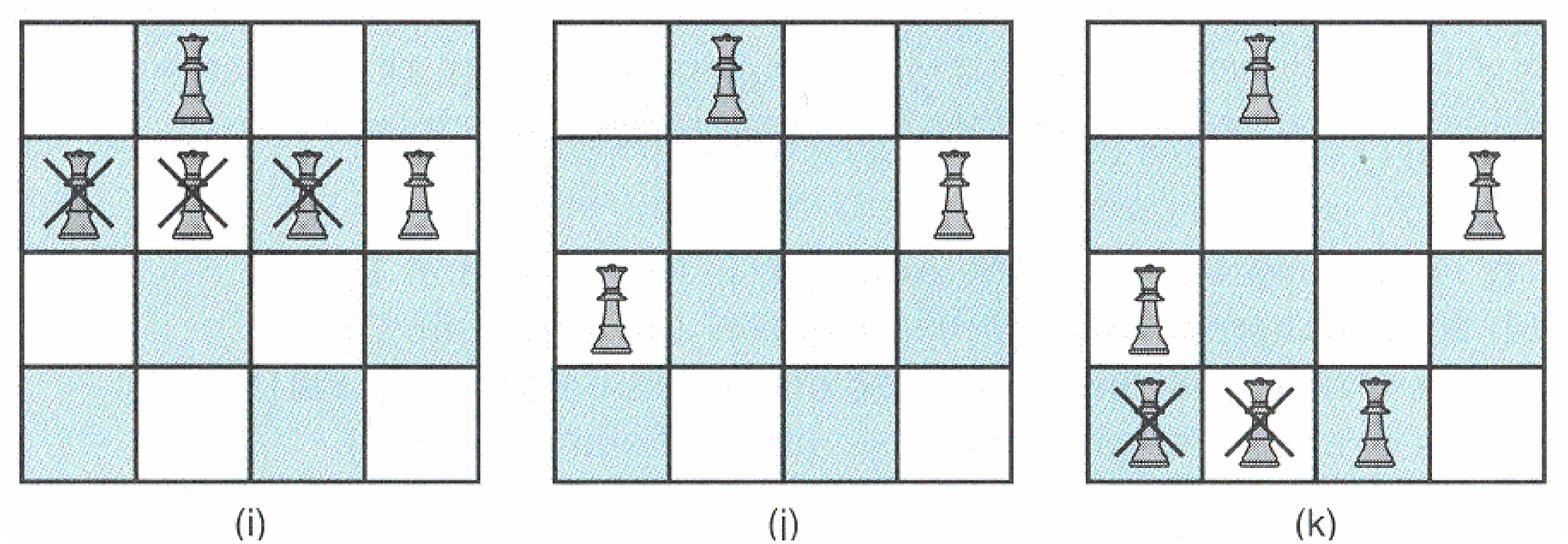


(g)

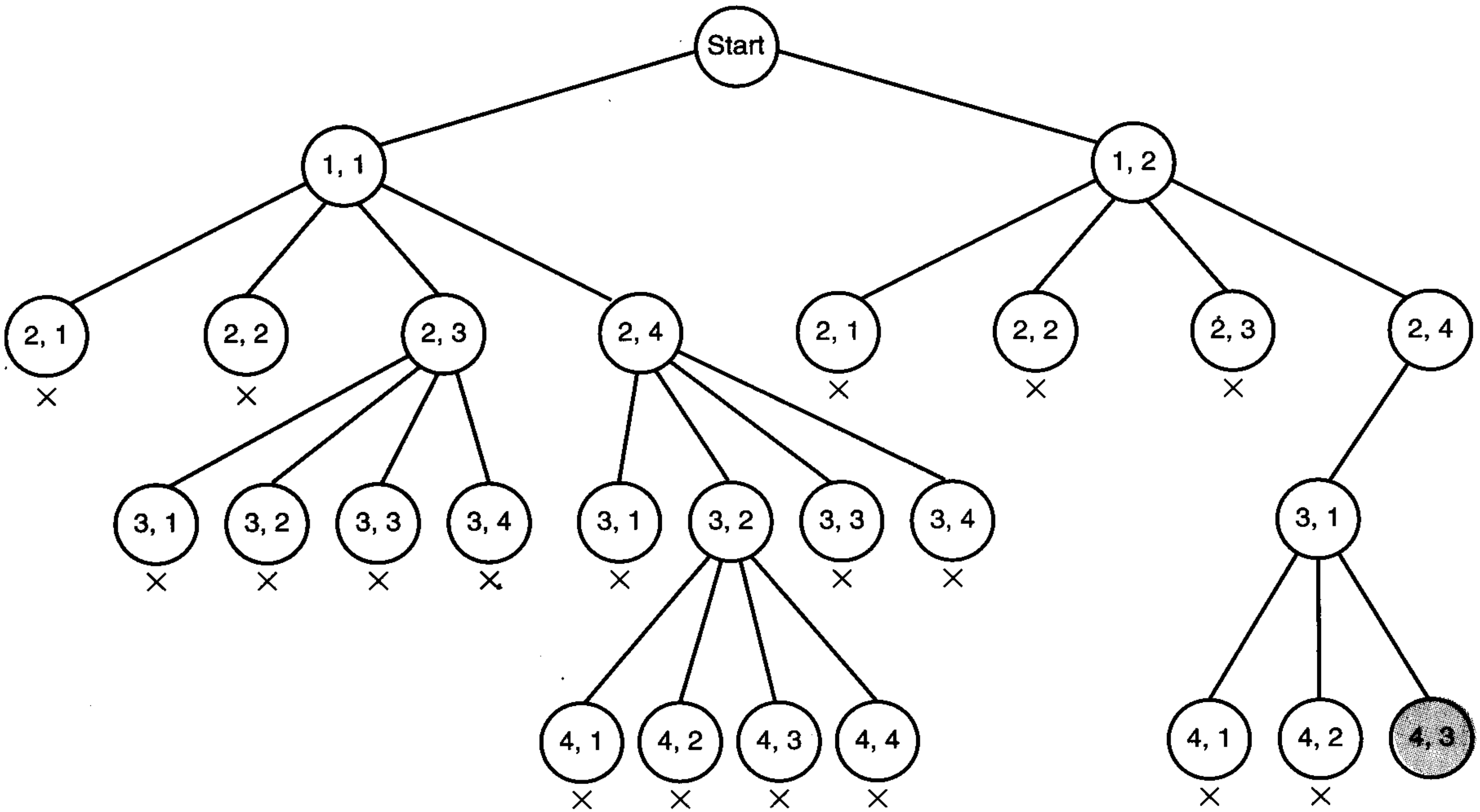


(h)

4-Queens problem (5)



Pruned state space tree



Avoid creating nonpromising nodes

```
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);
}
```

The *n*-Queens Problem

- Check whether two queens threaten each other:
  - $col(i) - col(k) = i - k$
  - $col(i) - col(k) = k - i$

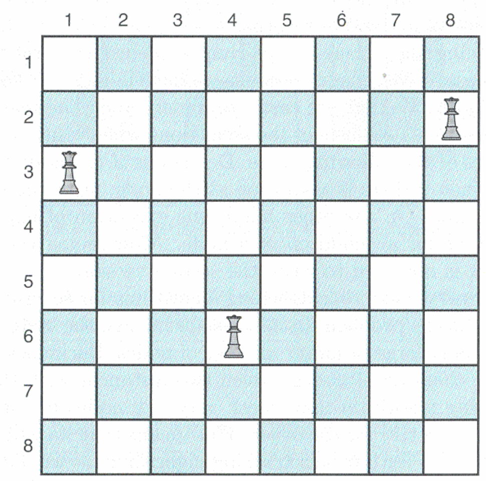


Figure 5.6 ● The queen in row 6 is being threatened in its left diagonal by the queen in row 3 and in its right diagonal by the queen in row 2.

Efficiency

- Checking the entire state space tree (number of nodes checked)

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

- Taking the advantage that no two queens can be placed in the same row or in the same column

  1 + n + $n(n-1)$ + $n(n-1)(n-2)$ + … + $n!$ promising nodes

Comparison

● **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.

The Sum-of-Subsets Problem

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

$$w_1 = 5 \qquad w_2 = 6 \qquad w_3 = 10 \qquad w_4 = 11 \qquad w_5 = 16.$$

Because

$$\begin{aligned} w_1 + w_2 + w_3 &= 5 + 6 + 10 = 21, \\ w_1 + w_5 &= 5 + 16 = 21, \text{and} \\ w_3 + w_4 &= 10 + 11 = 21, \end{aligned}$$

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

State Space Tree

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

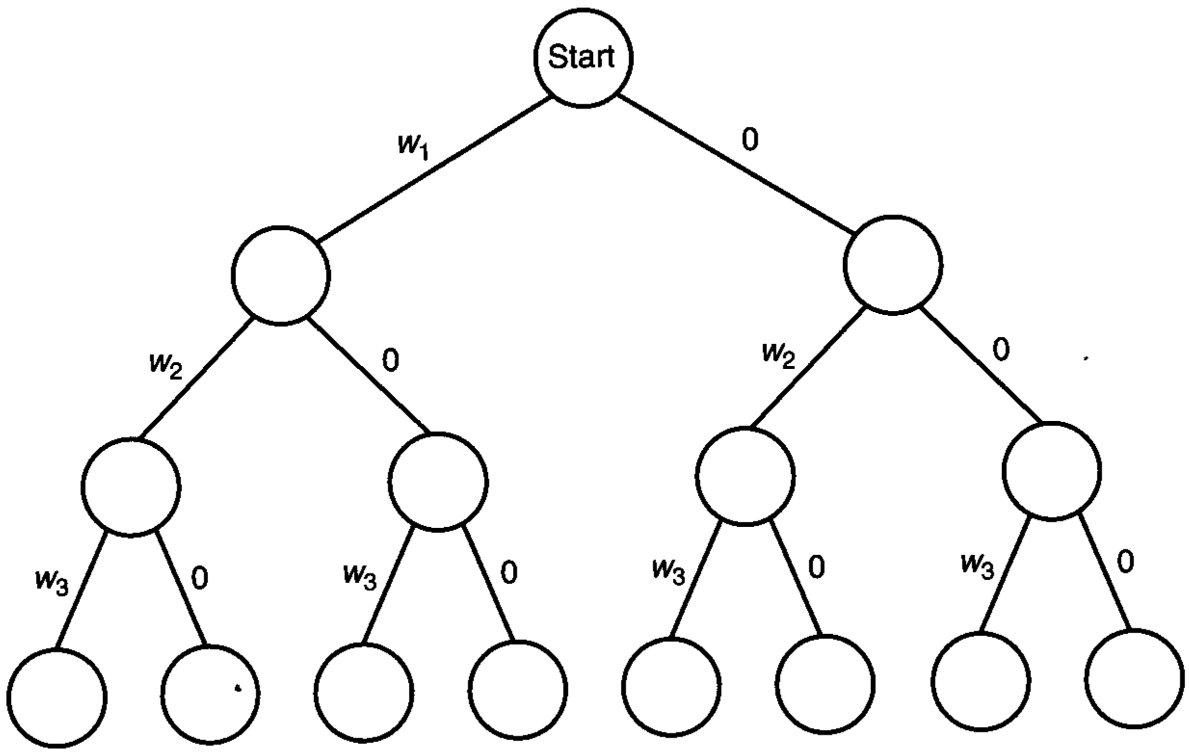
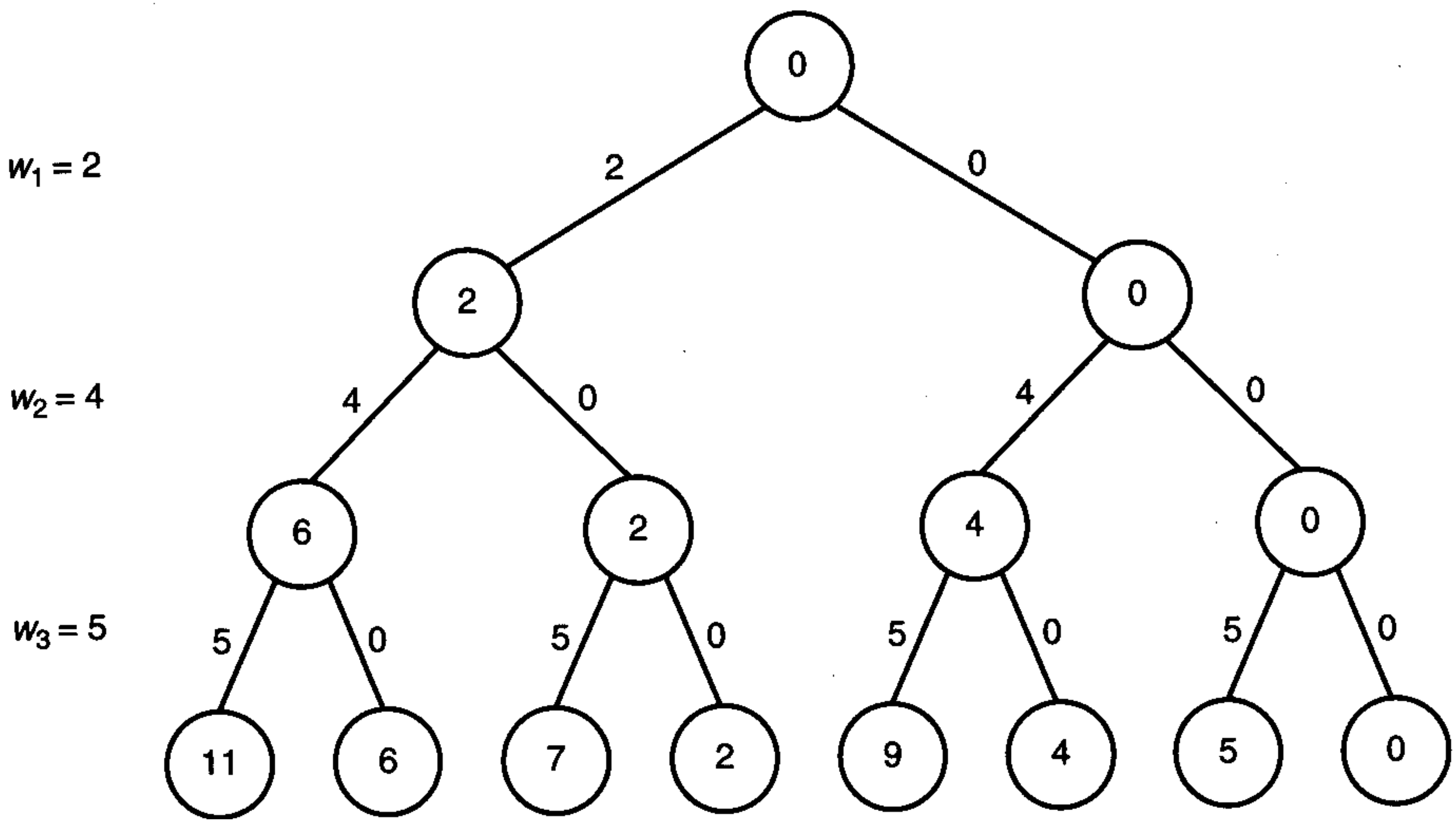


Figure 5.7 ● A state space tree for instances of the Sum-of-Subsets problem in which $n = 3$.

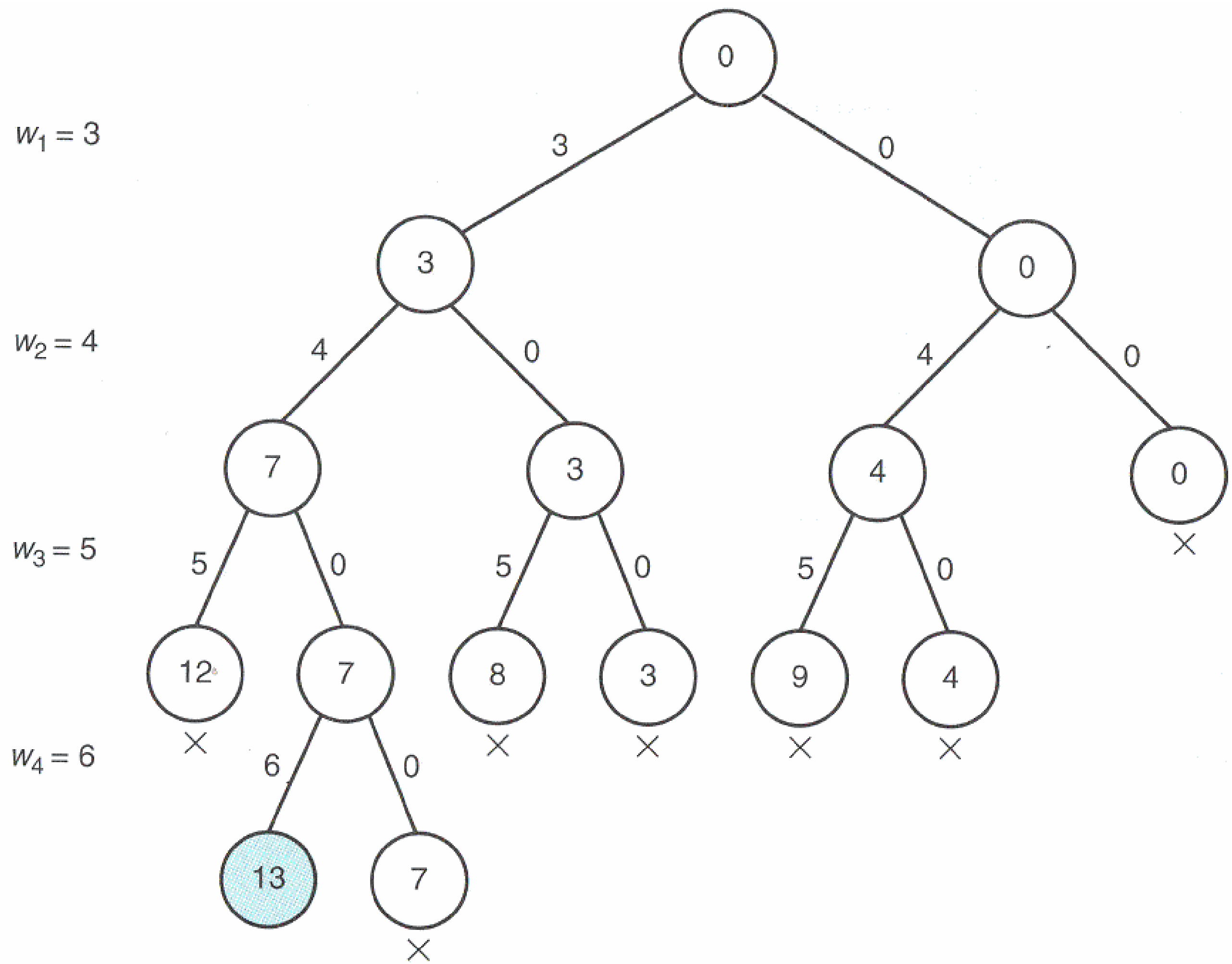
When $W = 6$ and $w_1 = 2$, $w_2 = 4$, $w_3 = 5$



To check whether a node is promising

- Sort the weights in nondecreasing order
- To check the node at level i
  - $weight + w_{i+1} > W$
  - $weight + total < W$

When $W = 13$ and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$



The algorithm 5.4

```
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);
}
```

Time complexity

- The first call to the function

  sum_of_subsets(0, 0, *total*) where

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

- The number of nodes checked

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

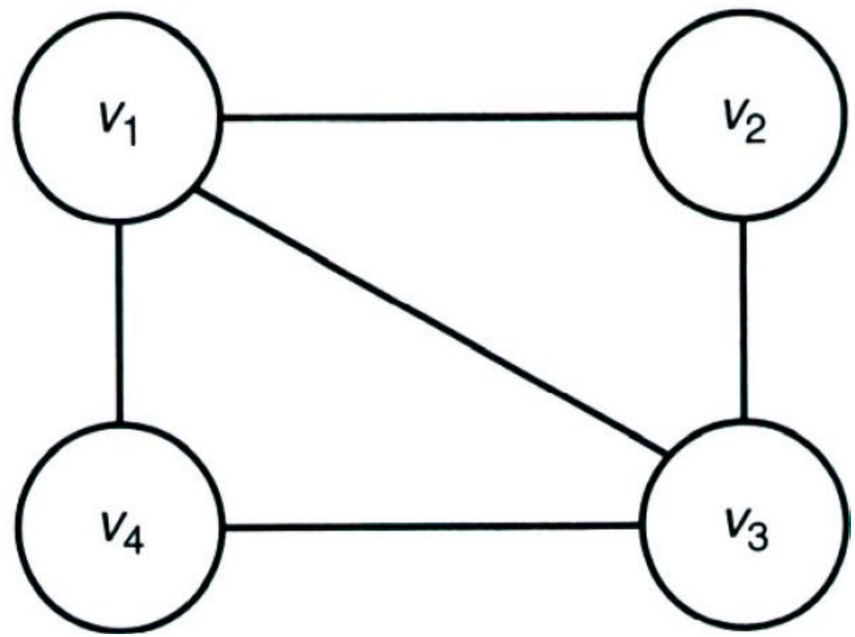
Graph coloring

- The *m*-Coloring problem
  - Finding all ways to color an undirected graph using at most *m* different colors, so that no two adjacent vertices are the same color.
  - Usually the *m*-Coloring problem consider as a unique problem for each value of *m*.

Example

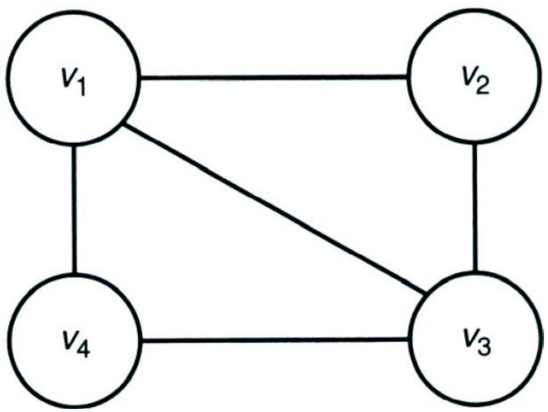
- 2-coloring problem
  - No solution!
- 3-coloring problem

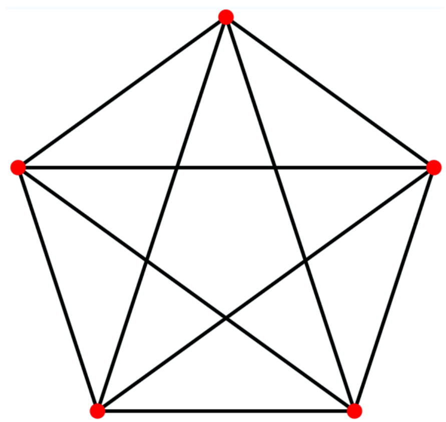
Vertex	***Color***
*v*1	color1
*v*2	color2
*v*3	color3
*v*4	color2



Application: Coloring of maps

- **Planar** graph
  - It can be drawn in a plane in such a way that no two edges cross each other.

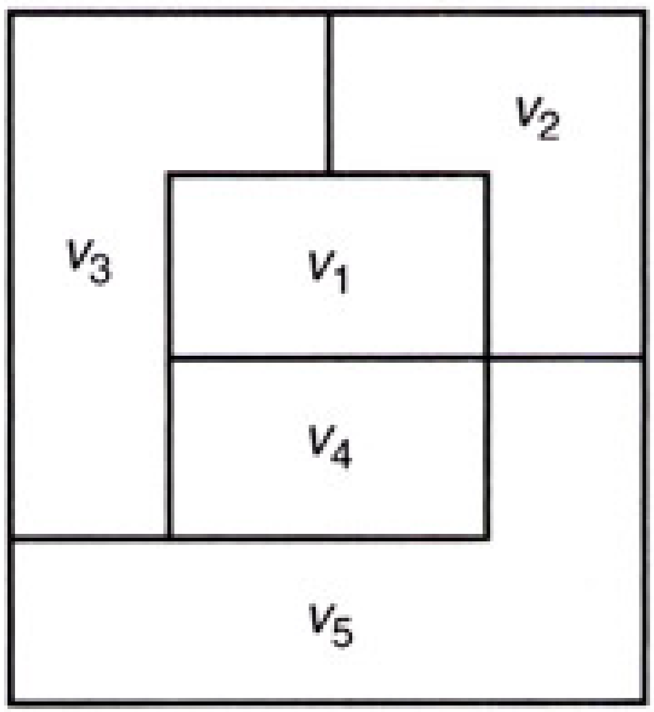




- To every map there corresponds a planar graph

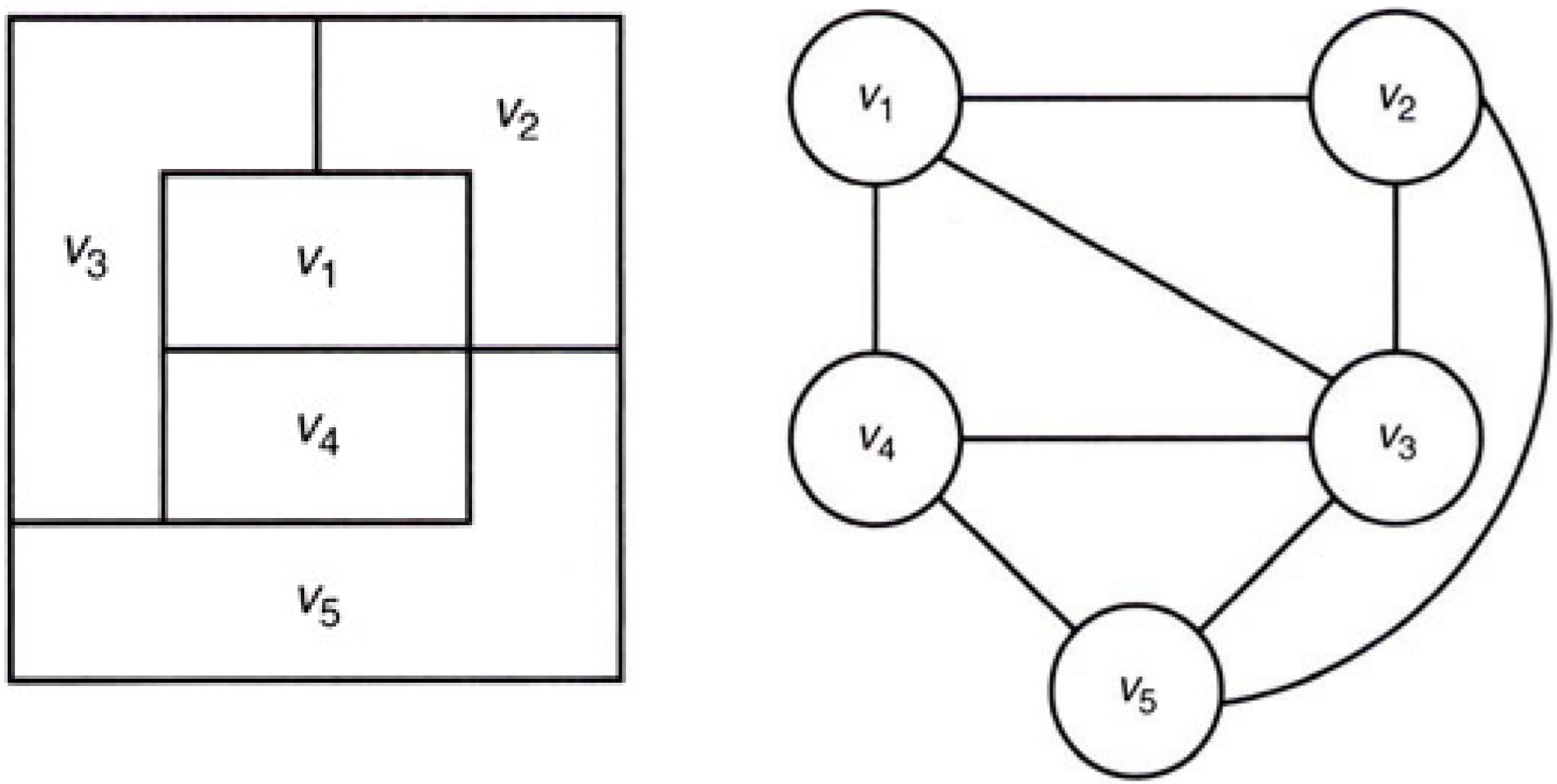
Example (1)

- Map

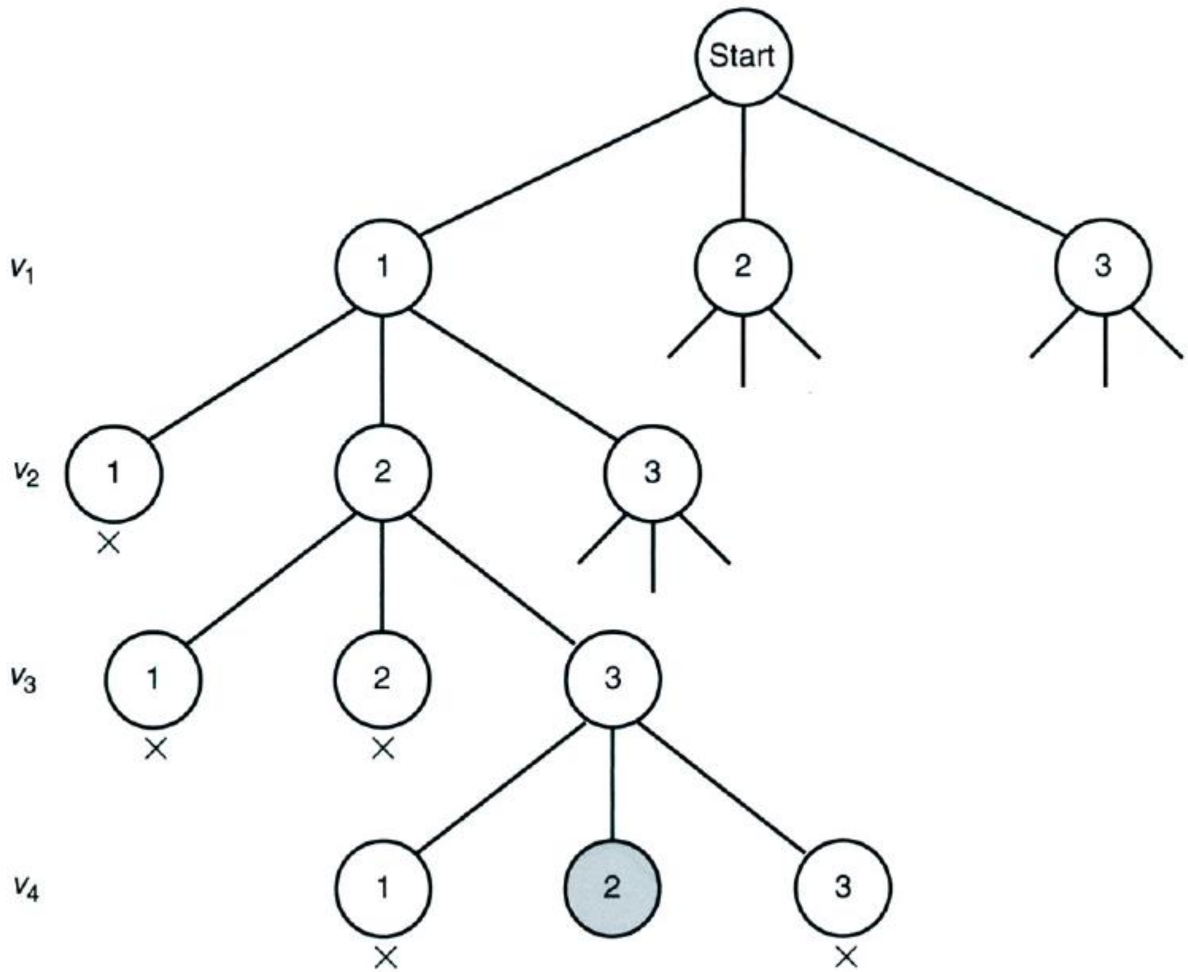


Example (2)

- corresponded planar graph



The pruned state space tree



Algorithm 5.5 (1)

```
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);
               }
}
```

Algorithm 5.5 (2)

```
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;
}
```

Algorithm 5.5 (3)

- The top level call to *m*_coloring
  - m_coloring(0)
- The number of nodes in the state space tree for this algorithm

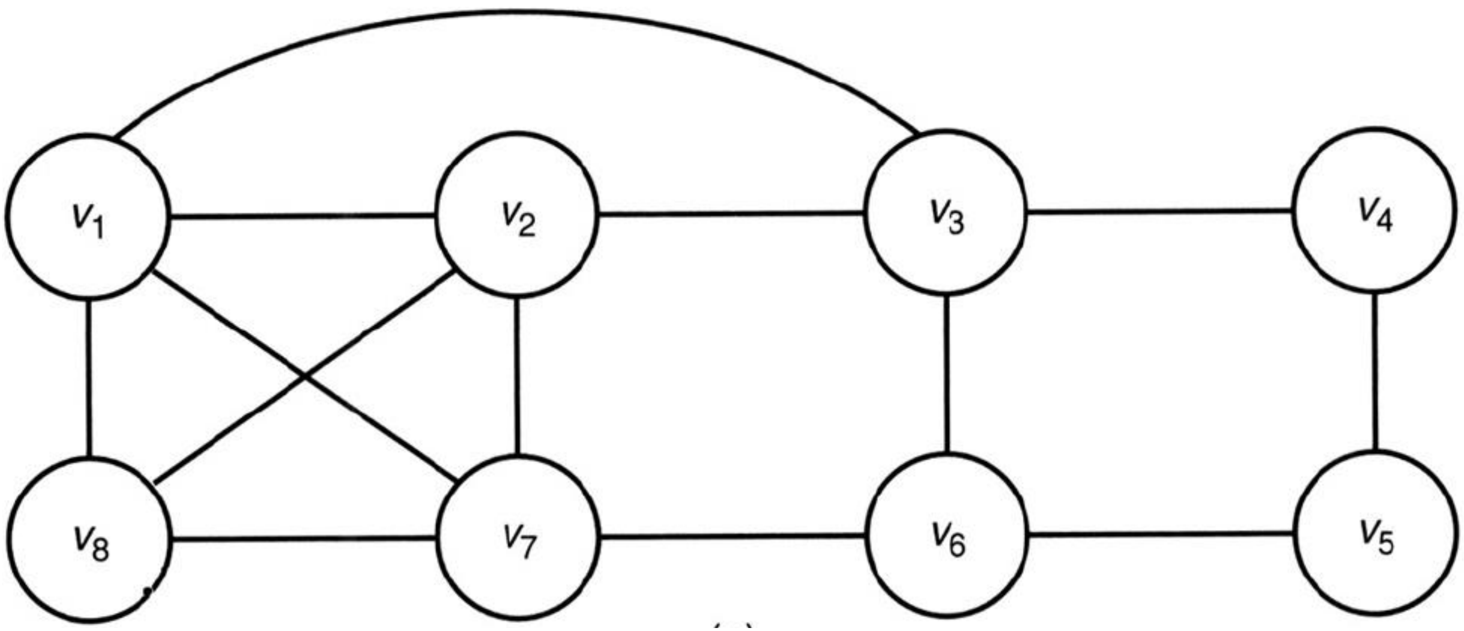
$$1 + m + m^2 + \cdots + m^n = \frac{m^{n+1} - 1}{m - 1}$$

The Hamiltonian Circuits Problem

- The traveling sales person problem
  - Chapter 3: Dynamic programming
  - $T(n) = (n-1)(n-2)2^{n-3}$
- ***Hamiltonian Circuit*** (also called a tour)
  - Given a connected, undirected graph
  - A path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex

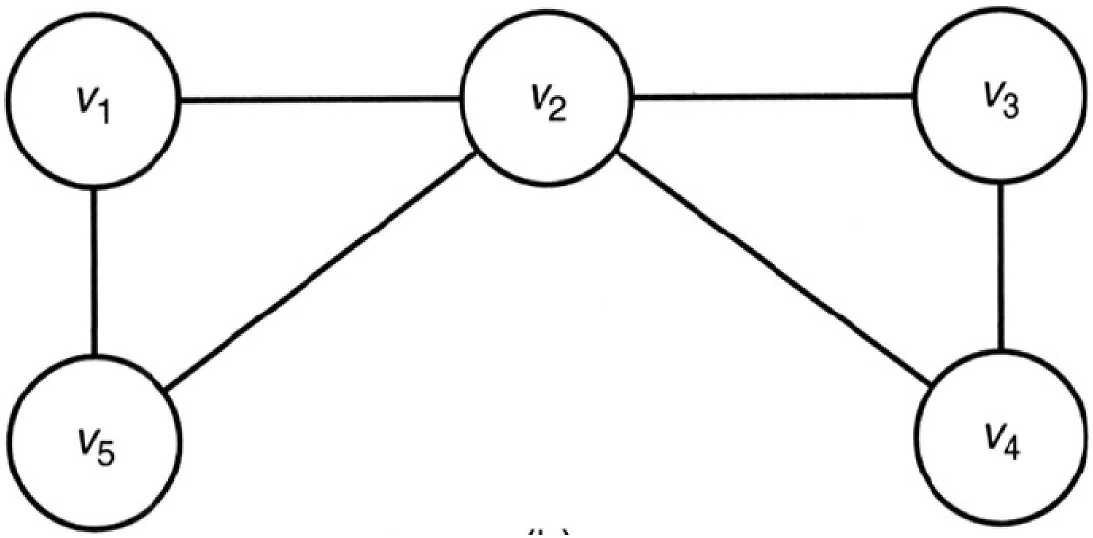
Example (1)

- Hamiltonian Circuit
  - [v1, v2, v8, v7, v6, v5, v4, v3, v2]



Example (2)

- No Hamiltonian Circuit!



Algorithm 5.6 (1)

```
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);
               }
   }
```

Algorithm 5.6 (2)

```
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;
}
```

Example

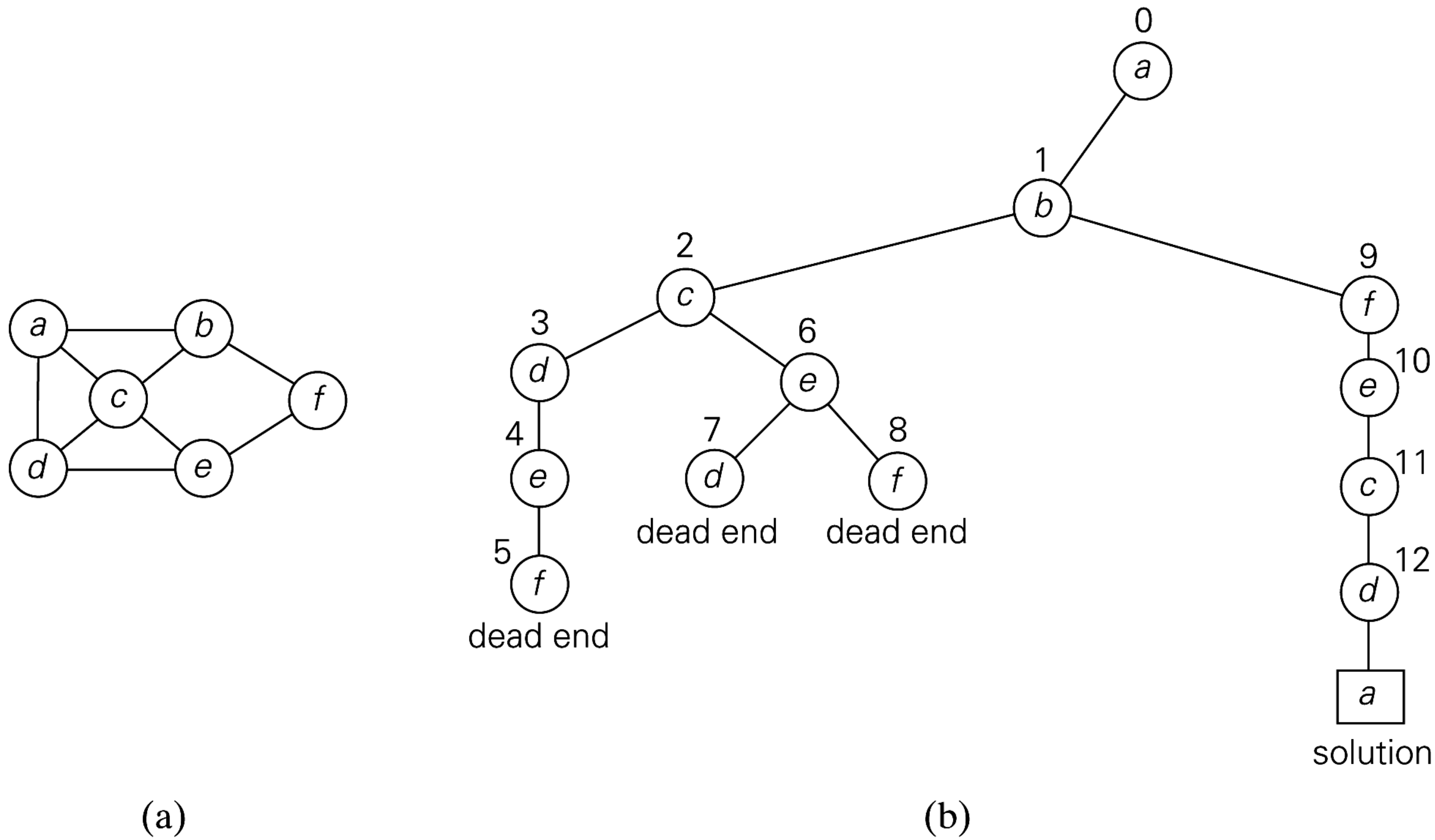


FIGURE 12.3 (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

- bool Place(int k, int i) // Returns true if a queen can be placed in kth row and // ith column. Otherwise it returns false. x[] is a // global array whose first (k-1) values have been set. // abs(r) returns the absolute value of r. { for (int j=1; j < k; j++) if ((x[j] == i) // Two in the same column || (abs(x[j]-i) == abs(j-k))) // or in the same diagonal return(false); return(true); }

N-Queens algorithm place(int k,int i)

- bool Place(int k, int i)
- // Returns true if a queen can be placed in kth row and // ith column. Otherwise it returns false. x[] is a
- // global array whose first (k-1) values have been set.
- // abs(r) returns the absolute value of r.
- {
  - for (int j=1; j < k; j++)
  - if ((x[j] == i) // Two in the same column
    - || (abs(x[j]-i) == abs(j-k)))
    - // or in the same diagonal

```
return(false);
return(true);
}
```

N-Queens algorithm

- void NQueens(int k, int n)
- // Using backtracking, this procedure prints all
- // possible placements of n queens on an n X n
- // chessboard so that they are nonattacking.
- {
- for (int i=1; i<=n; i++)
- {
  - if (Place(k, i))
  - { x[k] = i;
  - if (k==n) { for (int j=1;j<=n;j++)
  - cout << x[j] << ' '; cout << endl;}
  - else NQueens(k+1, n); } } }

Sum of subsets

- void SumOfSub(float s, int k, float r)
- // Find all subsets of w[1:n] that sum to m. The values
- // of x[j], 1<=j<k, have already been determined.
- // s=sigma{j=1}^{k-1} w[j]*x[j] and r=sigma{j=k}^n w[j].
- // The w[j]'s are in nondecreasing order.
- // It is assumed that w[1]<=m and sigma{i=1}^n w[i]>=m.
  {

```
// Generate left child. Note that s+w[k] <= m
 // because B_{k-1} is true.
 x[k] = 1;
 if (s+w[k] == m) { // Subset found
 for (int j=1; j<=k; j++) cout << x[j] << ' ';
 cout << endl;      }
   // There is no recursive call here
   // as w[j] > 0, 1 <= j <= n.
 else if (s+w[k]+w[k+1] <= m)
                SumOfSub(s+w[k], k+1, r-w[k]);
```

Contd…

- // Generate right child and evaluate B_k.
- if ((s+r-w[k] >= m) && (s+w[k+1] <= m)) {
- x[k] = 0;
  - SumOfSub(s, k+1, r-w[k]); } }

Graph Coloring algorithm

- void mColoring(int k)
- // This program was formed using the recursive backtracking
- // schema. The graph is represented by its boolean adjacency
- // matrix G[1:n][1:n]. All assignments of 1,2,...,m to the // vertices of the graph such that adjacent vertices are
- // assigned distinct integers are printed. k is the index of // the next vertex to color.

Contd…

- { do { // Generate all legal assignments for x[k]. NextValue(k); // Assign to x[k] a legal color.
- if (!x[k]) break; // No new color possible
- if (k == n) { // At most m colors have been used // to color the n vertices.
- for (int i=1; i<=n; i++) cout << x[i] << ' ';
- cout << endl; } //if
- else mColoring(k+1);
- } while(1);
- }

Contd… algorithm for next color

- void NextValue(int k)
- // x[1],..., x[k-1] have been assigned integer values in
- // the range [1,m] such that adjacent vertices have distinct
- // integers. A value for x[k] is determined in the range
- // [0,m]. x[k] is assigned the next highest numbered color
- // while maintaining distinctness from the adjacent vertices
- // of vertex k. If no such color exists, then x[k] is zero.
- {

Contd…

- do {
- x[k] = (x[k]+1) % (m+1); // Next highest color
- if (!x[k]) return; // All colors have been used.
- for (int j=1; j<=n; j++) { // Check if this color is // distinct from adjacent colors.
- if (G[k][j] // If (k, j) is an edge
- && (x[k] == x[j])) // and if adj. vertices have the same color break; }
- if (j == n+1) return; // New color found
- } while (1); // Otherwise try to find another color.
- }

Hamiltonian circuit – Next Vertex

- void NextValue(int k)
- // x[1],...,x[k-1] is a path of k-1 distinct vertices. If
- // x[k]==0, then no vertex has as yet been assigned to x[k].
- // After execution x[k] is assigned to the next highest
- // numbered vertex which i) does not already appear in
- // x[1],x[2],...,x[k-1]; and ii) is connected by an edge
- // to x[k-1]. Otherwise x[k]==0. If k==n, then
- // in addition x[k] is connected to x[1].
- {

Contd…

- do {
- x[k] = (x[k]+1) % (n+1); // Next vertex
- if (!x[k]) return;
- if (G[x[k-1]][x[k]]) { // Is there an edge?
- for (int j=1; j<=k-1; j++) if (x[j]==x[k]) break; // Check for distinctness.
- if (j==k) // If true, then the vertex is distinct.
- if ((k<n) || ((k==n) && G[x[n]][x[1]]))
- return;
- }
- } while(1);
- }

Contd…

```
void Hamiltonian(int k)
 // This program uses the recursive formulation of
 // backtracking to find all the Hamiltonian cycles
 // of a graph. The graph is stored as an adjacency
 // matrix G[1:n][1:n]. All cycles begin at node 1.
 {
  do { // Generate values for x[k].
 NextValue(k);  // Assign a legal next value to x[k].
       if (!x[k]) return;
       if (k == n) {
             for (int i=1; i<=n; i++) cout << x[i] << ' ';
             cout << "1\n";
          }
       else Hamiltonian(k+1);
     } while (1);
}
```