Chapter 5

Backtracking

پسگرد، عقبگرد

The idea

- دنباله‌ای از اشیاء از یک مجموعه مشخص انتخاب می شود به صورتی‌که دنباله به دست آمده معیارهایی را برآورده کند.

- Example: *n*-Queens problem
  - Sequence: *n* positions on the chessboard
  - Set: n^2 possible positions
  - Criterion: no two queens can threaten each other
- روش پسگرد یک جستجوی عمقی تغییریافته یک درخت است.

Depth first search (Example)

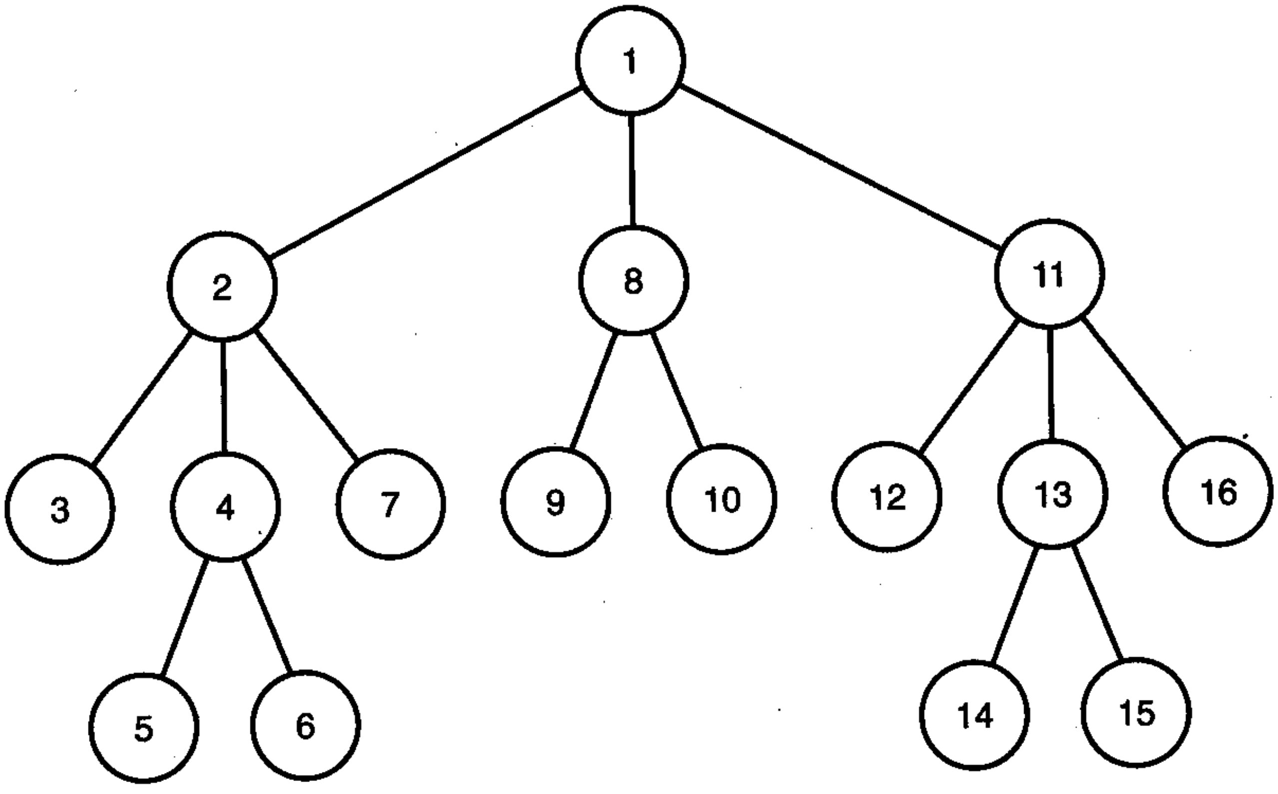
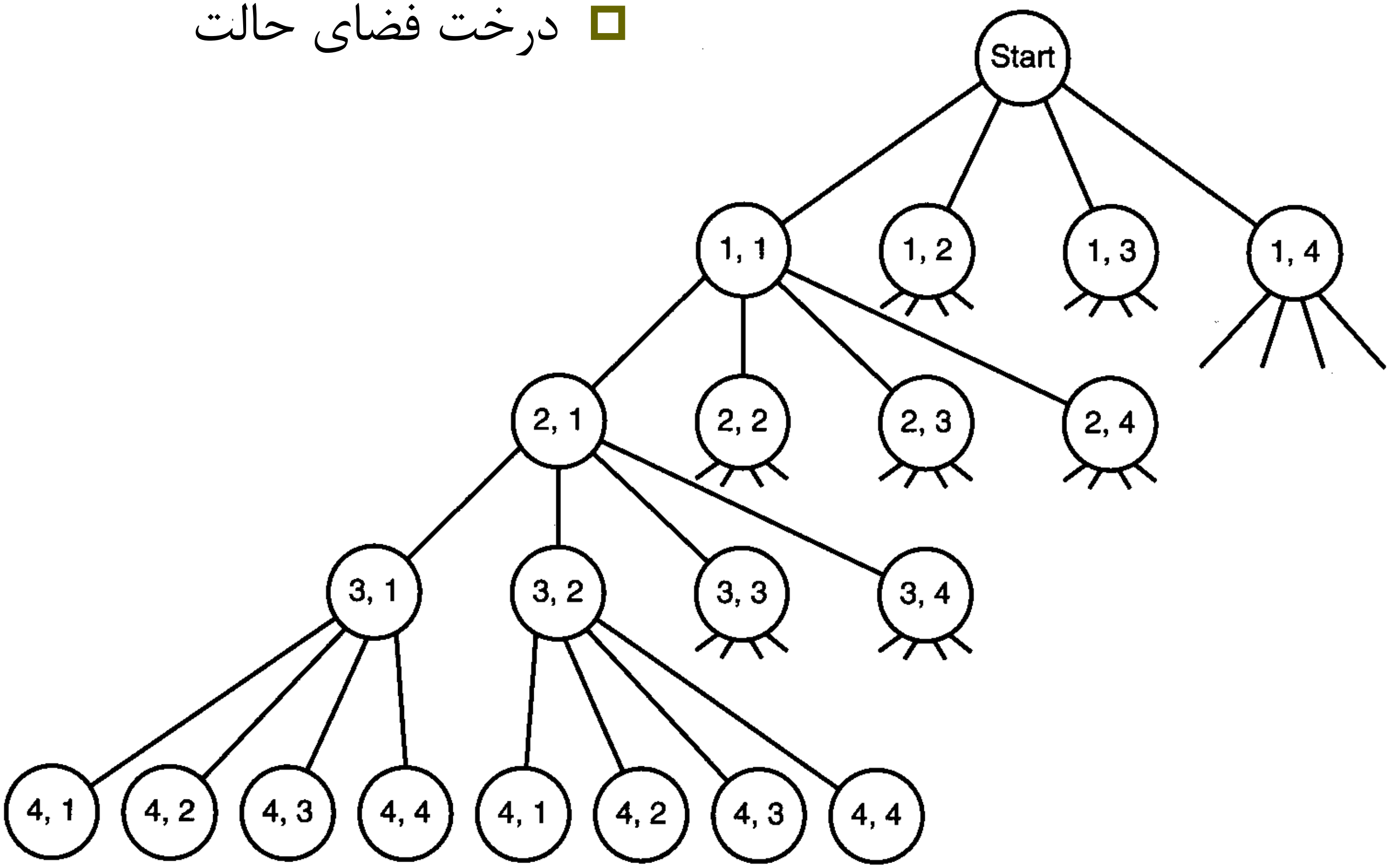


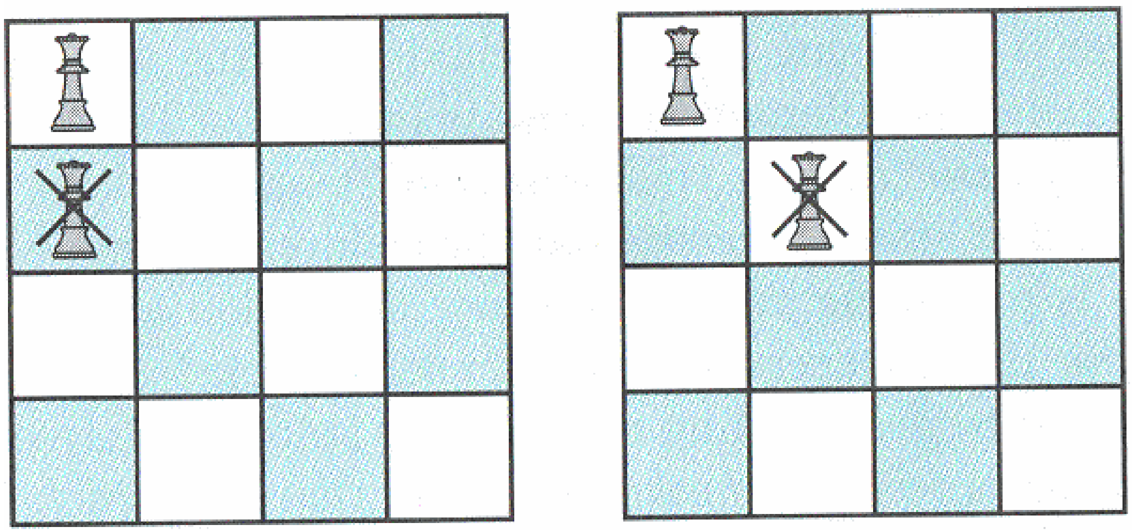
Figure 5.1 ● A tree with nodes numbered according to a depth-first search.

4-Queens problem
مساله چهار وزیر

- State space tree
  - درخت فضای حالت



Looking for signs for dead ends

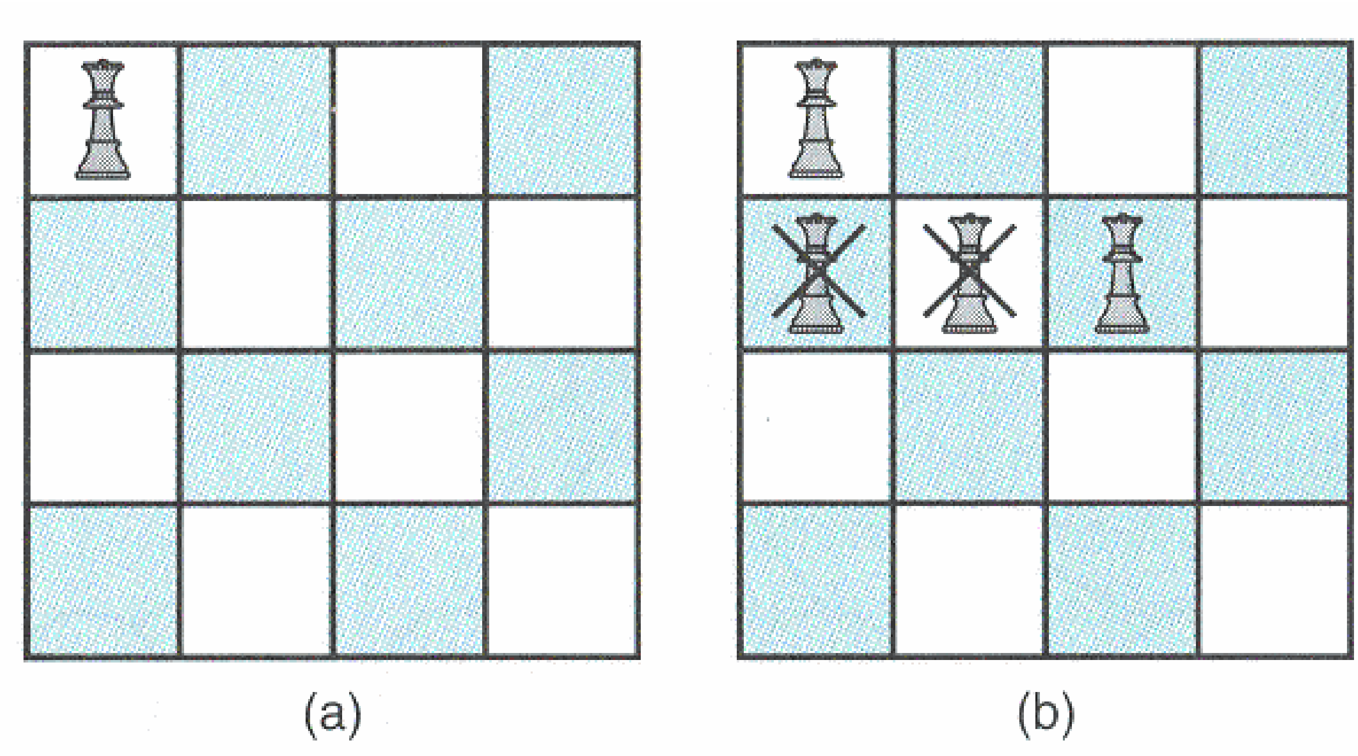


(a) (b)

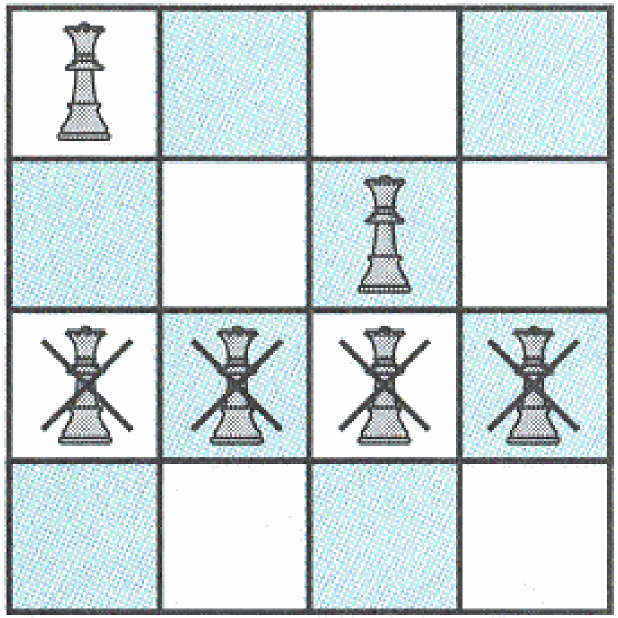
پسگرد

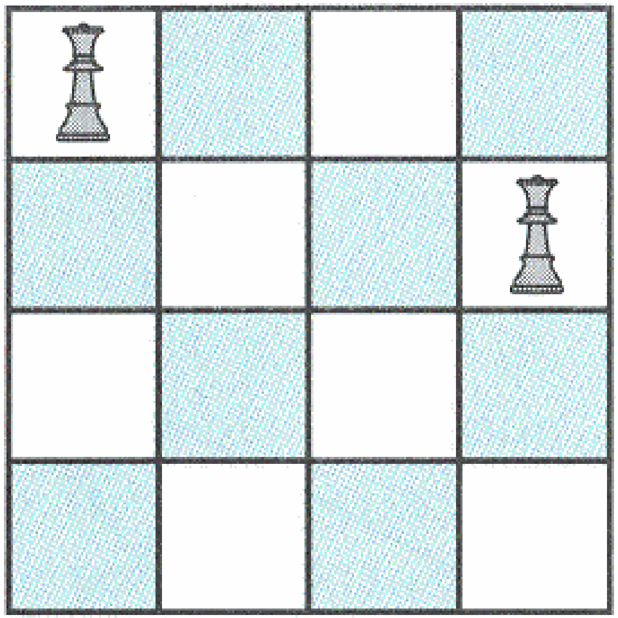
- در پسگرد وقتی مشخص شد که یک گره به بن‌بست منجر می‌شود به گره والد آن باز می‌گردیم و به همزاد آن وارد می‌شویم.
- اگر گرهی به راه‌حل منجر نشود؛ غیر امیدبخش (nonpromising) نامیده می‌شود، در غیر این صورت امید بخش (promising) است.
- پسگرد درخت جستجو را به صورت عمقی پیمایش می‌نماید، اگر گرهی غیر امیدبخش باشد؛ به والد آن بر می‌گردد. این کار به اصطلاح هرس (pruning) نام دارد.

4-Queens problem (1)

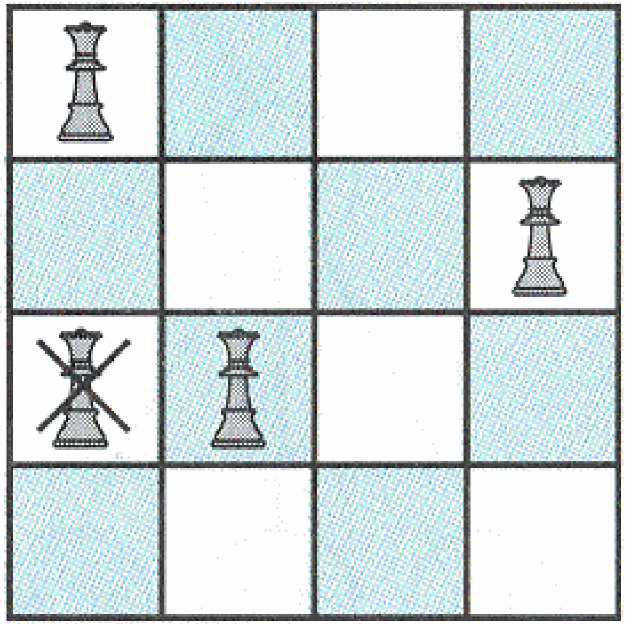


4-Queens problem (2)

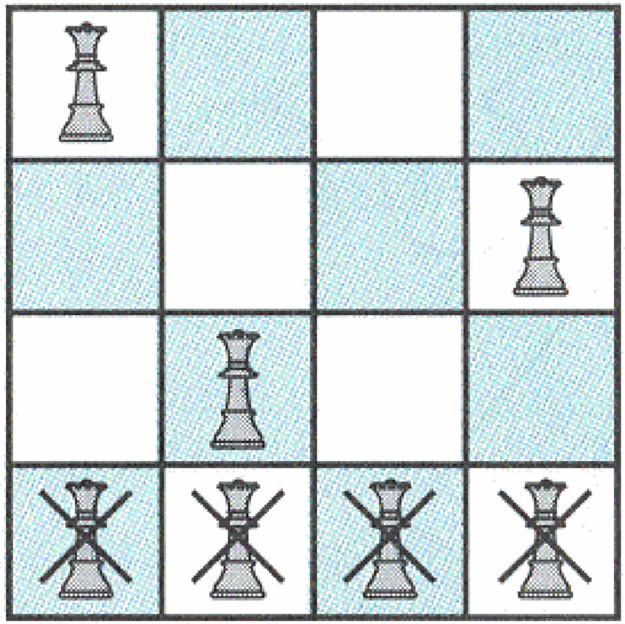

(c)


(d)

4-Queens problem (3)

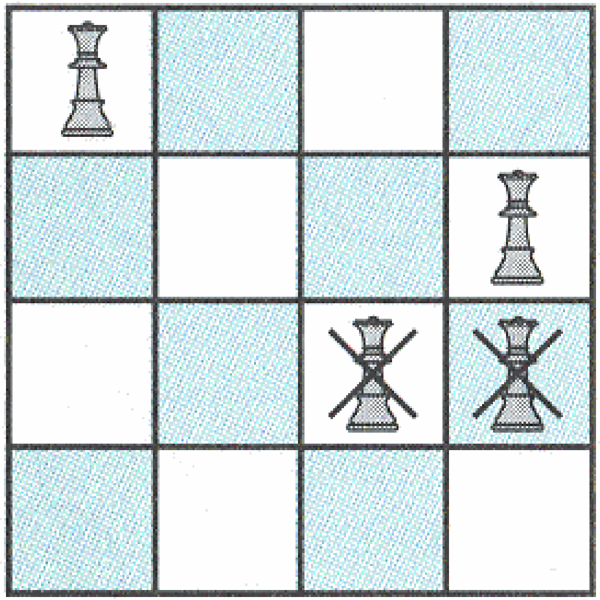


(e)

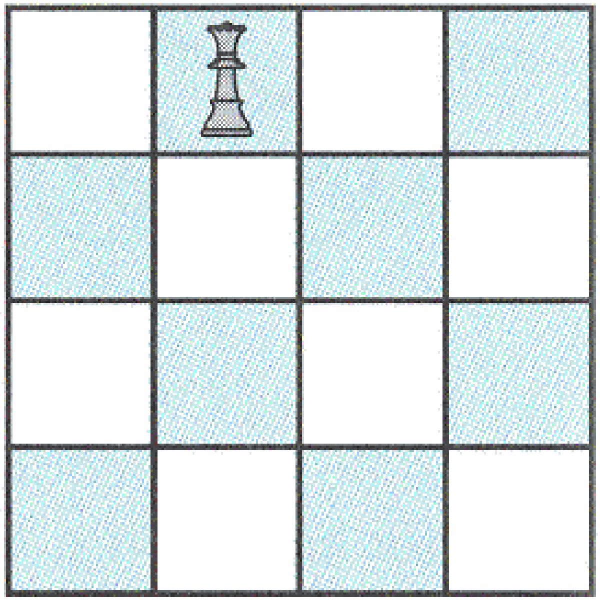


(f)

4-Queens problem (4)

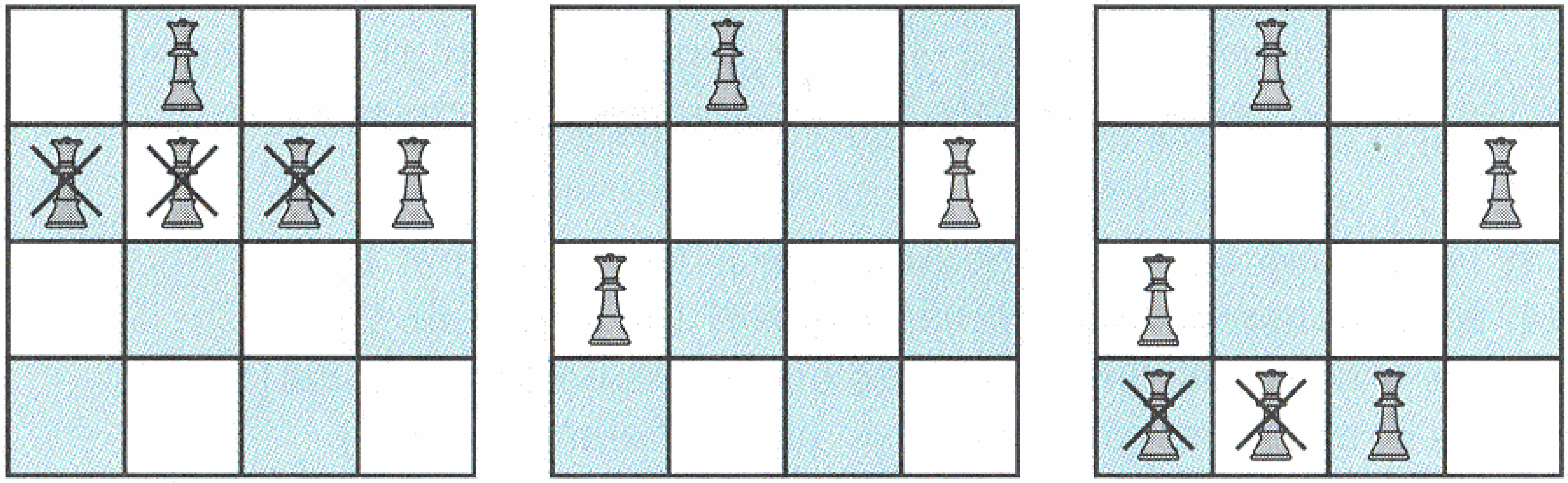


(g)



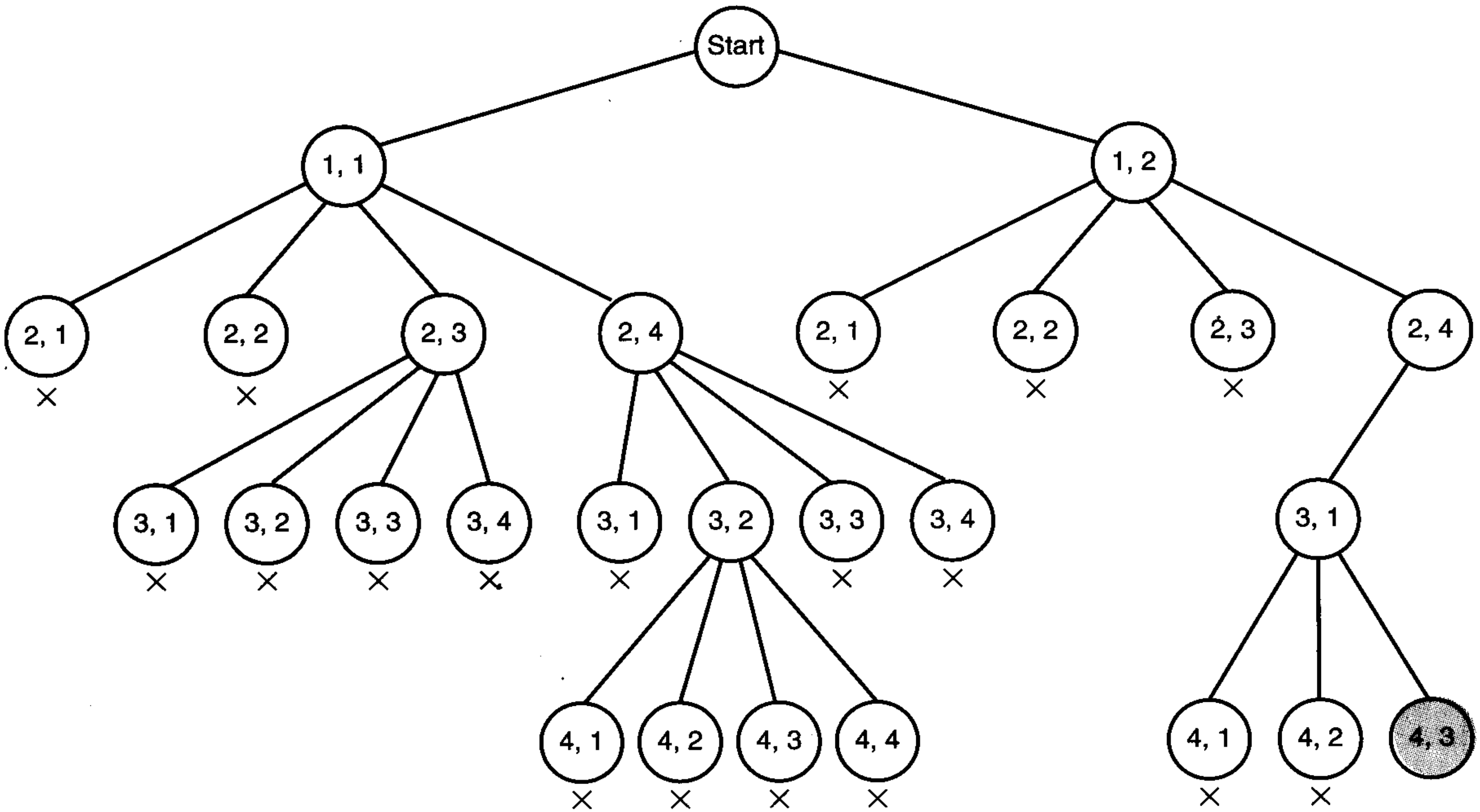
(h)

4-Queens problem (5)



(i) (j) (k)

Pruned state space tree



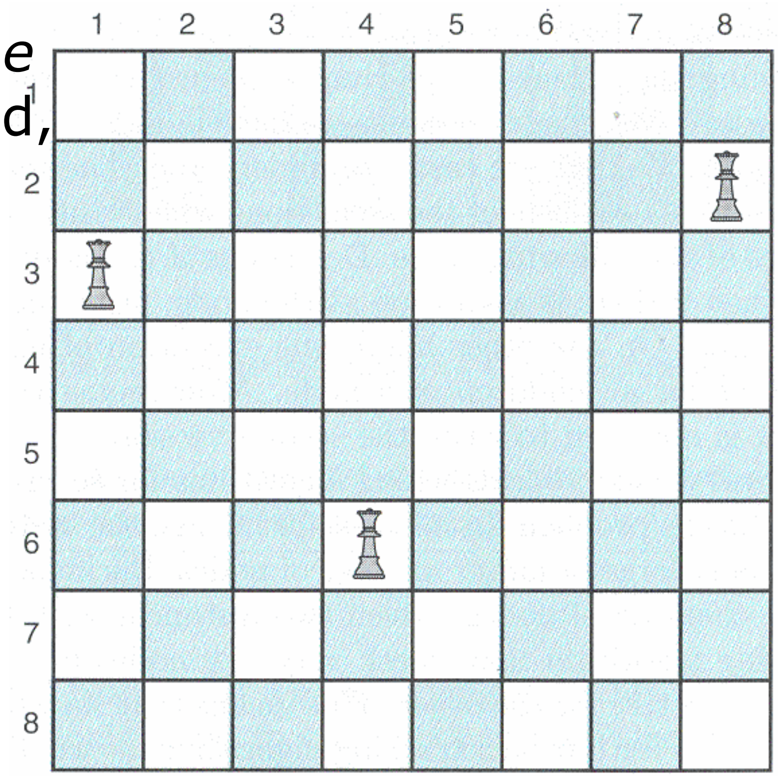
Avoid creating nonpromising nodes

```
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) is the column where the queen in the ith* row is located,
- *Check diagonal*
  - $col(i) - col(k) = i - k$
  - $col(i) - col(k) = k - i$



The algorithm

```
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.}
}
```

The algorithm (2)

```
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;}
```

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.

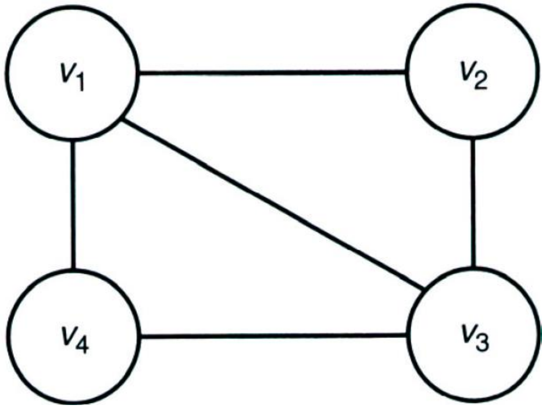
†Algorithm 1 does a depth-first search of the state space tree without backtracking.

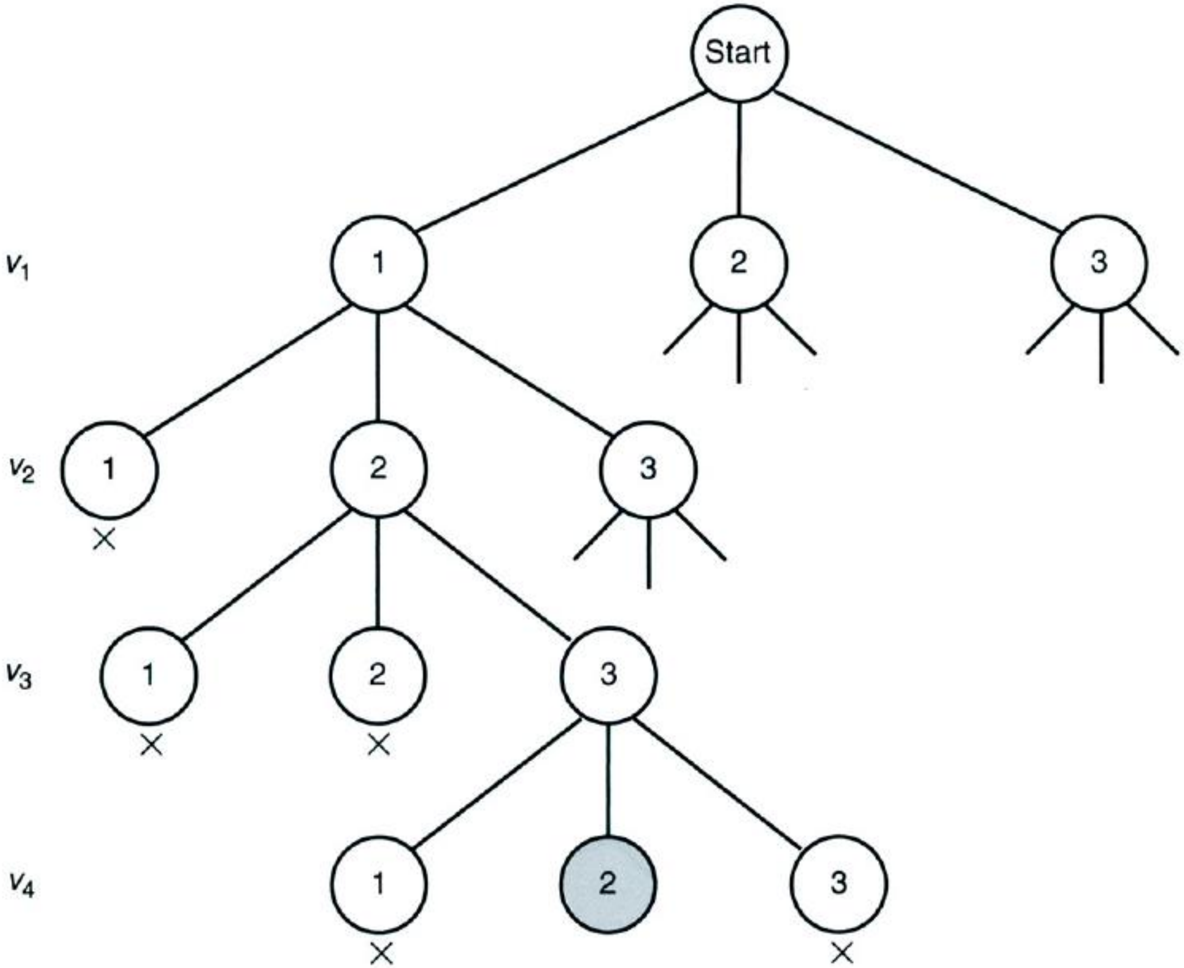
‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.

Graph coloring
رنگ‌آمیزی نقشه

- یافتن تمام حالت‌هایی که می‌توان کشورهای مختلف را با m رنگ، رنگ‌آمیزی نمود به صورتی که دو کشور همسایه همرنگ نباشند

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 Sum-of-Subsets Problem
مساله جمع زیر مجموعه‌ها

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

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

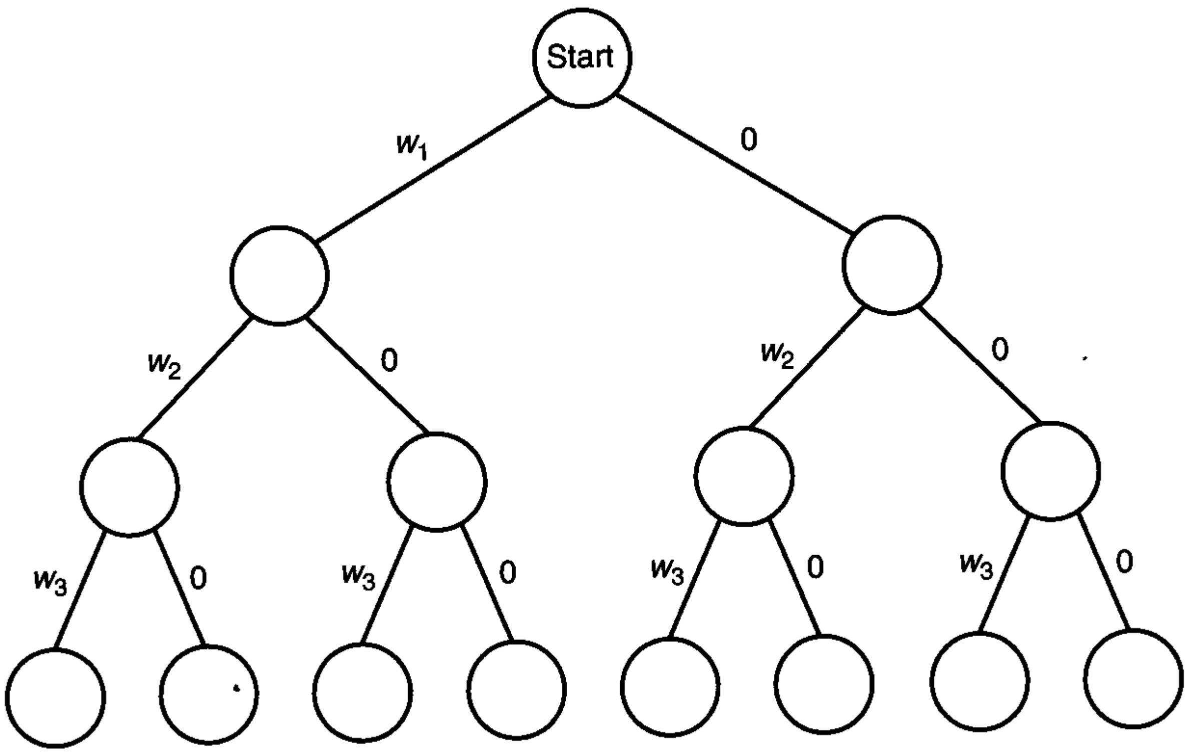
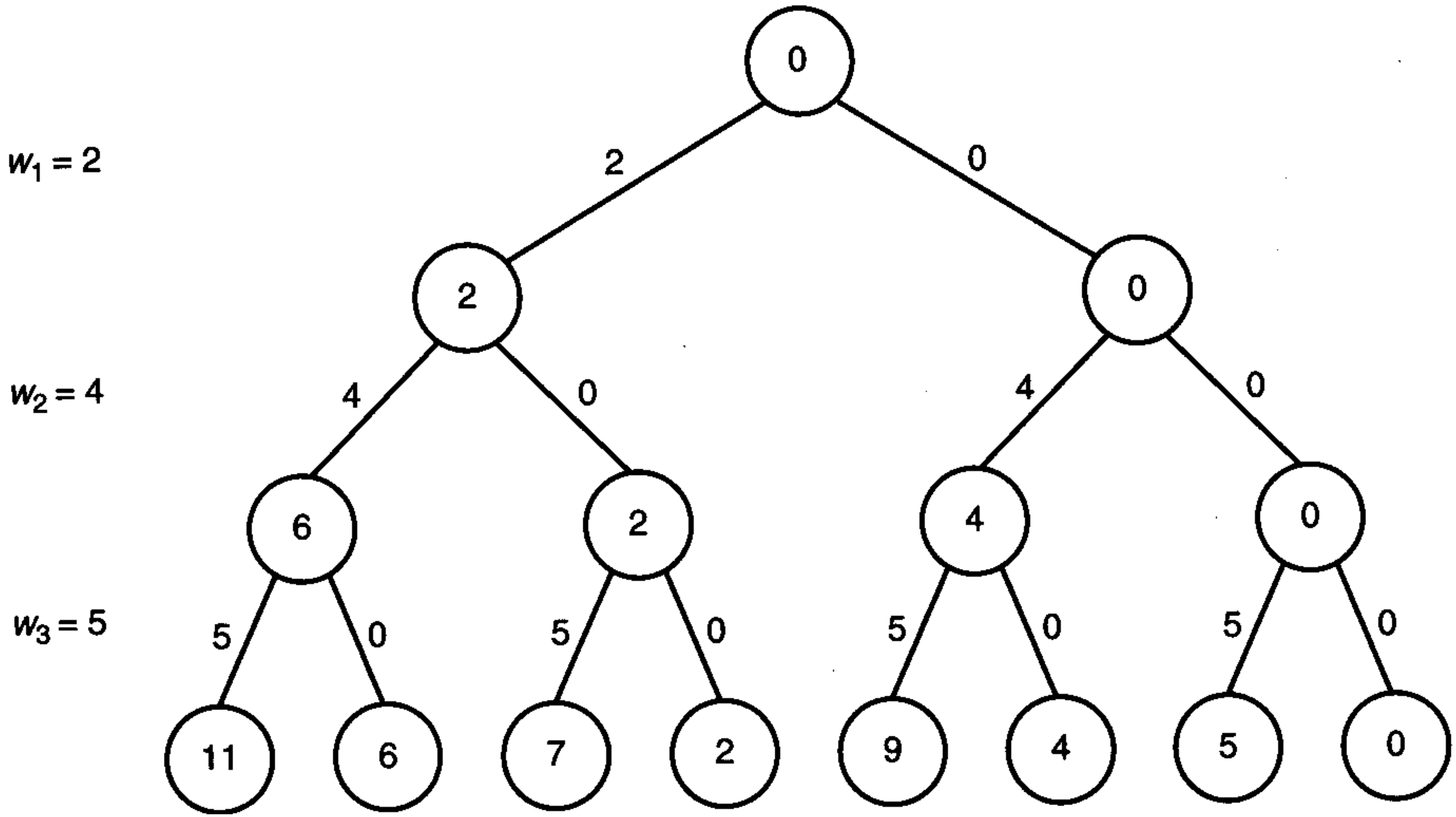


Figure 5.7 ● A state space tree for instances of the Sum-of-Subsets problem in which $n = 3$.

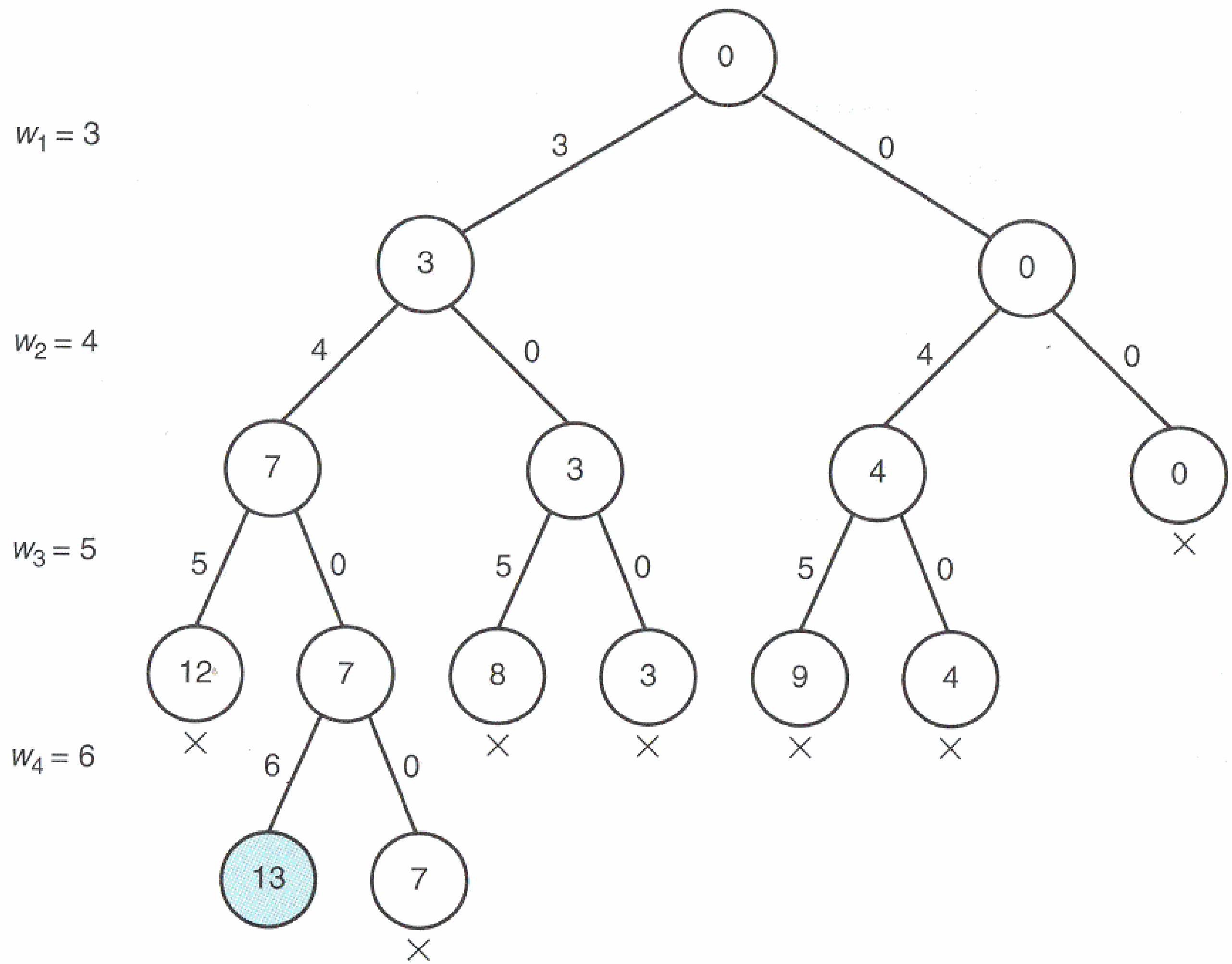
When $W = 6$ and $w_1 = 2$, $w_2 = 4$, $w_3 = 5$



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 in the state space tree are

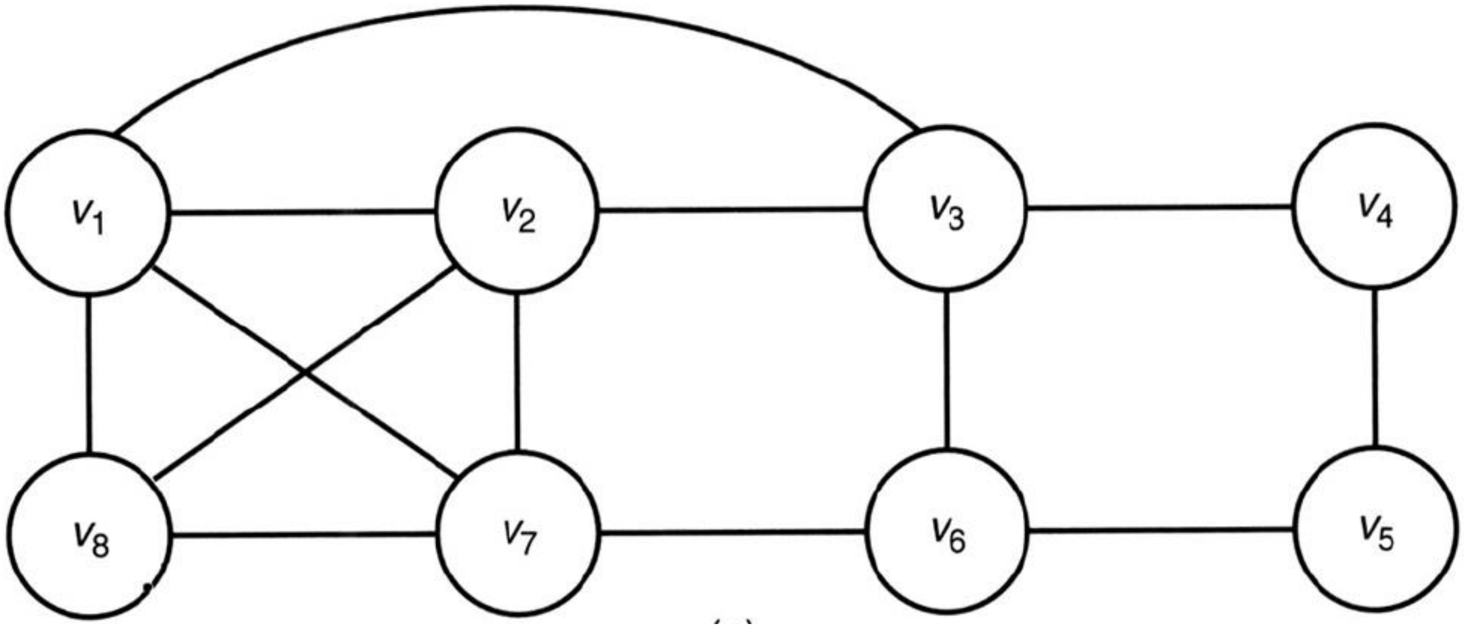
  $1 + 2 + 2^2 + \ldots + 2^n = -1+2^{n+1}$

The Hamiltonian Circuits Problem

- مساله دور همیلتونی
- مسیری که از همه راس‌ها دقیقا یک بار عبور کند و به راس اول بازگردد.
- فروشنده دوره‌گرد برای گراف بدون وزن

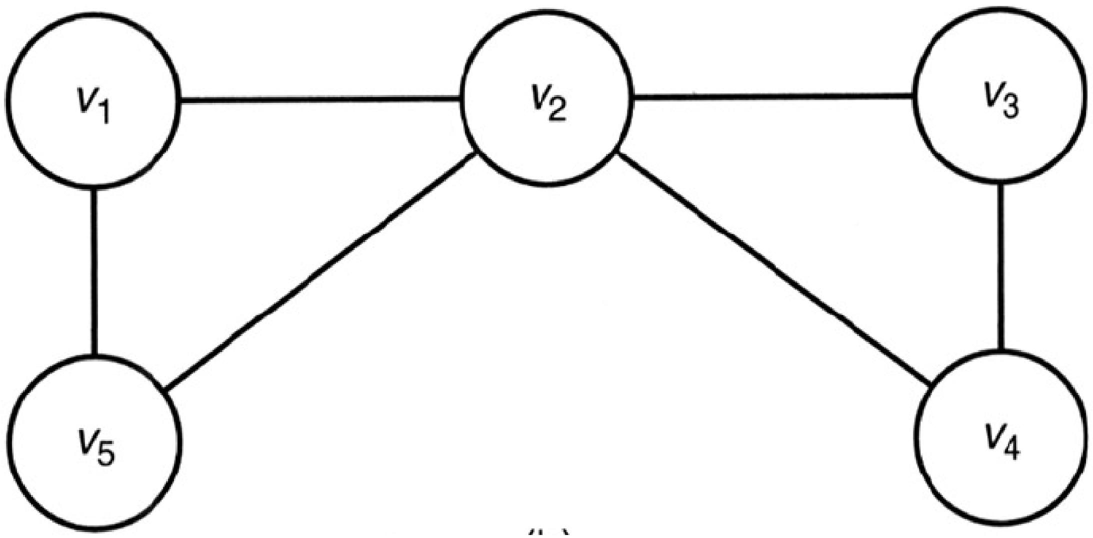
Example (1)

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



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;
}
```

Algorithm 5.6 (3)

- The top level call to hamiltonian:
  - vindex [0] = 1; *//Make* v_1 *the starting vertex*.
  - hamiltonian (0);
- The number of nodes in the state space tree is

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

The 0-1 Knapsack Problem
مساله کوله‌پشتی صفر و یک

- درخت فضای حالت این مساله دقیقاً مانند مساله جمع زیر مجموعه‌ها است.
- این یک مساله بهینه‌سازی است.
- ما نمی‌دانیم که یک گره حاوی یک راه حل است تا زمانی که جستجو به پایان برسد.

A general algorithm for backtracking in the case of optimization problems.

```
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);
 }
```

- In the case of optimization problems, "promising" means that we should expand to the children.

Promising check

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

The node at level k is the one that would bring the sum of the weights above W

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

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

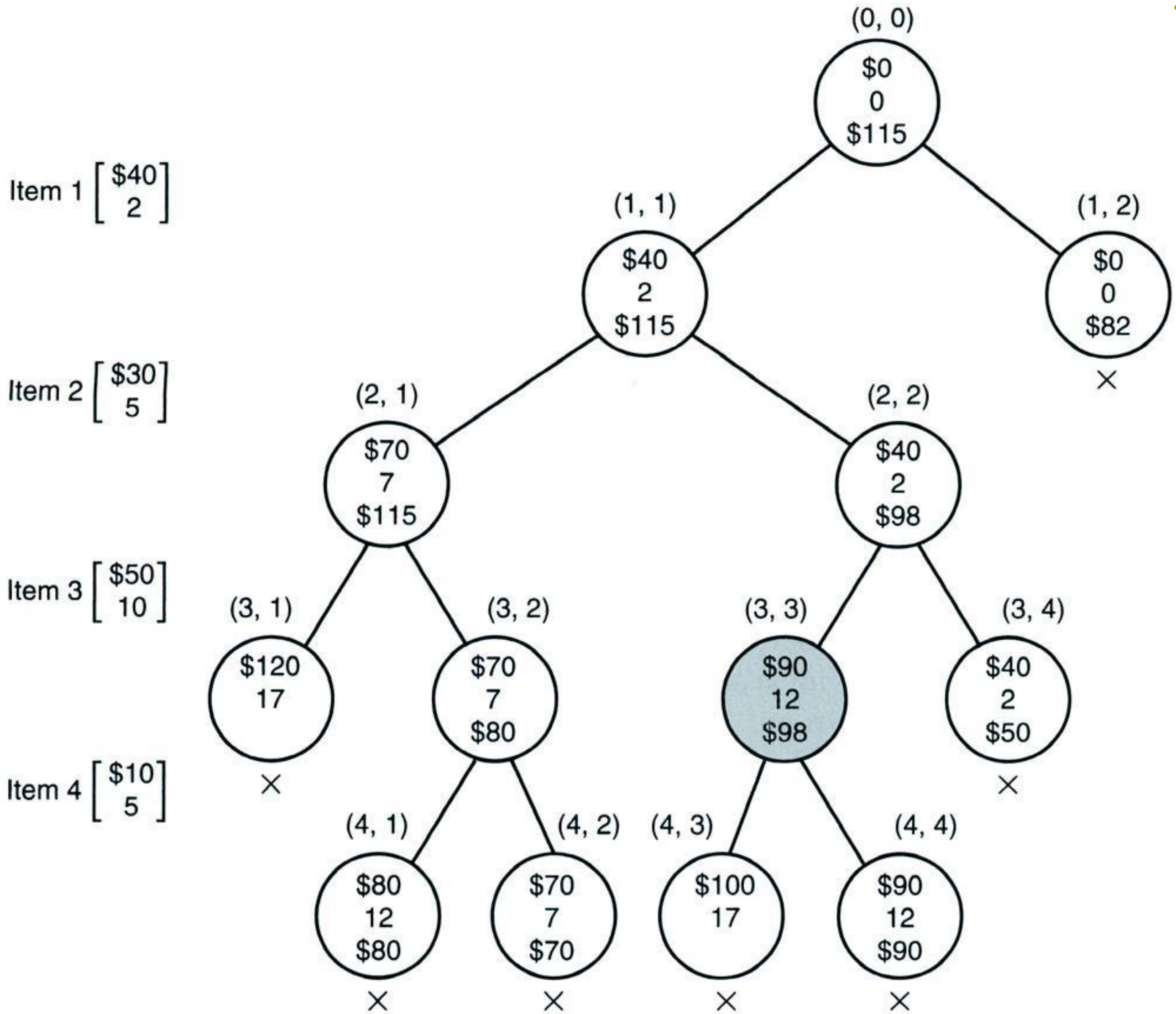
$$bound \le \max profit$$

Example

- n=4
- W = 16
- The items is ordered according to *pi/wi.*

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

The pruned state space tree produced using backtracking



Algorithm 5.7: The Backtracking Algorithm for the 0–1 Knapsack (1)

```
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"; // Include w[i + 1].
    knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
    include [i + 1] = "no"; // Do not include w[i+1]
    knapsack (i + 1, profit, weight);
}
}
```

Algorithm 5.7: The Backtracking Algorithm for the 0–1 Knapsack (2)

```
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;
     }
}
```