بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

طراحی الگوریتم‌ها

حسین فلسفین

ذکر اجمالی یک نکته درمورد مسئلهٔ *set-covering*

MINIMUM SET COVER

INSTANCE: Collection C of subsets of a finite set S.

SOLUTION: A set cover for S, i.e., a subset $C' \subseteq C$ such that every element in S belongs to at least one member of C'.

MEASURE: Cardinality of the set cover, i.e., $|C'|$.

Good News: Approximable within $1 + \ln|S|$ [Johnson, 1974a].

Bad News: Not in APX [Lund and Yannakakis, 1994] and [Bellare, Goldwasser, Lund, and Russell, 1993].

Chapter 5: Backtracking

If you were trying to find your way through the well-known maze of hedges by Hampton Court Palace in England, you would have no choice but to follow a hopeless path until you reached a dead end. When that happened, you'd go back to a fork and pursue another path. Anyone who has ever tried solving a maze puzzle has experienced the frustration of hitting dead ends. Think how much easier it would be if there were a sign, positioned a short way down a path, that told you that the path led to nothing but dead ends. If the sign were positioned near the beginning of the path, the time savings could be enormous, because all forks after that sign would be eliminated from consideration. This means that not one but many dead ends would be avoided. There are no such signs in the famous maze of hedges or in most maze puzzles. However, as we shall see, they do exist in backtracking algorithms.

Backtracking is very useful for problems such a the 0-1 Knapsack problem. Although in in previous sessions we found a dynamic programming algorithm for this problem that is efficient if the capacity W of the knapsack is not large, the algorithm is still exponential-time in the worst case.

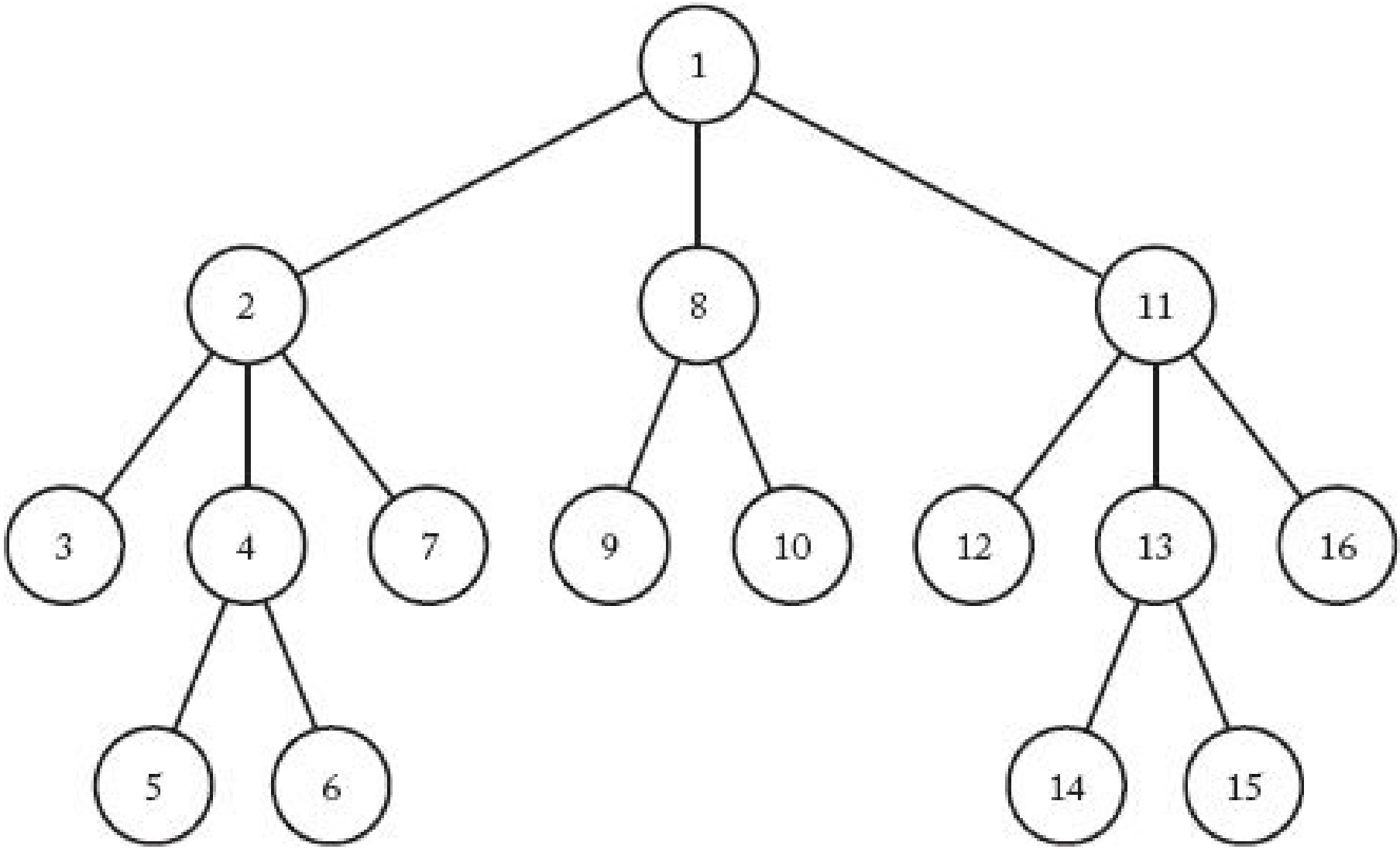
The 0-1 Knapsack problem is NP-hard. No one has ever found an algorithm for such a problem whose worst-case time complexity is better than exponential, but no one has ever proved that such algorithm does not exist.

One way to try to handle the 0-1 Knapsack problem would be to actually generate all the subsets, but this would be like following every path in a maze until a dead end is reached. Recall that there are 2^n subsets, which means that this brute-force method is feasible only for small values of n. However, if while generating the subsets we can find signs that tell us that many of them need not be generated, we can often avoid much unnecessary labor. This is exactly what a backtracking algorithm does. Backtracking algorithms for problems such as the 0-1 Knapsack problem are still exponential-time (or even worse) in the worst case. They are useful because they are efficient for many large instances, not because they are efficient for all large instances.

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. The classic example of the use of backtracking is in the n-Queens problem. The goal in this problem is to position n queens on an $n \times n$ chessboard so that no two queens threaten each other; that is, no two queens may be in the same row, column, or diagonal. The sequence in this 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, 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. For the sake of brevity, we will illustrate backtracking using the instance when $n = 4$.

Backtracking is a modified depth-first search of a tree (here "tree" means a rooted tree).

Although the depth-first search is defined for graphs in general, we will discuss only searching of trees, because backtracking involves only a tree search. 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. In a depth-first search, a 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.

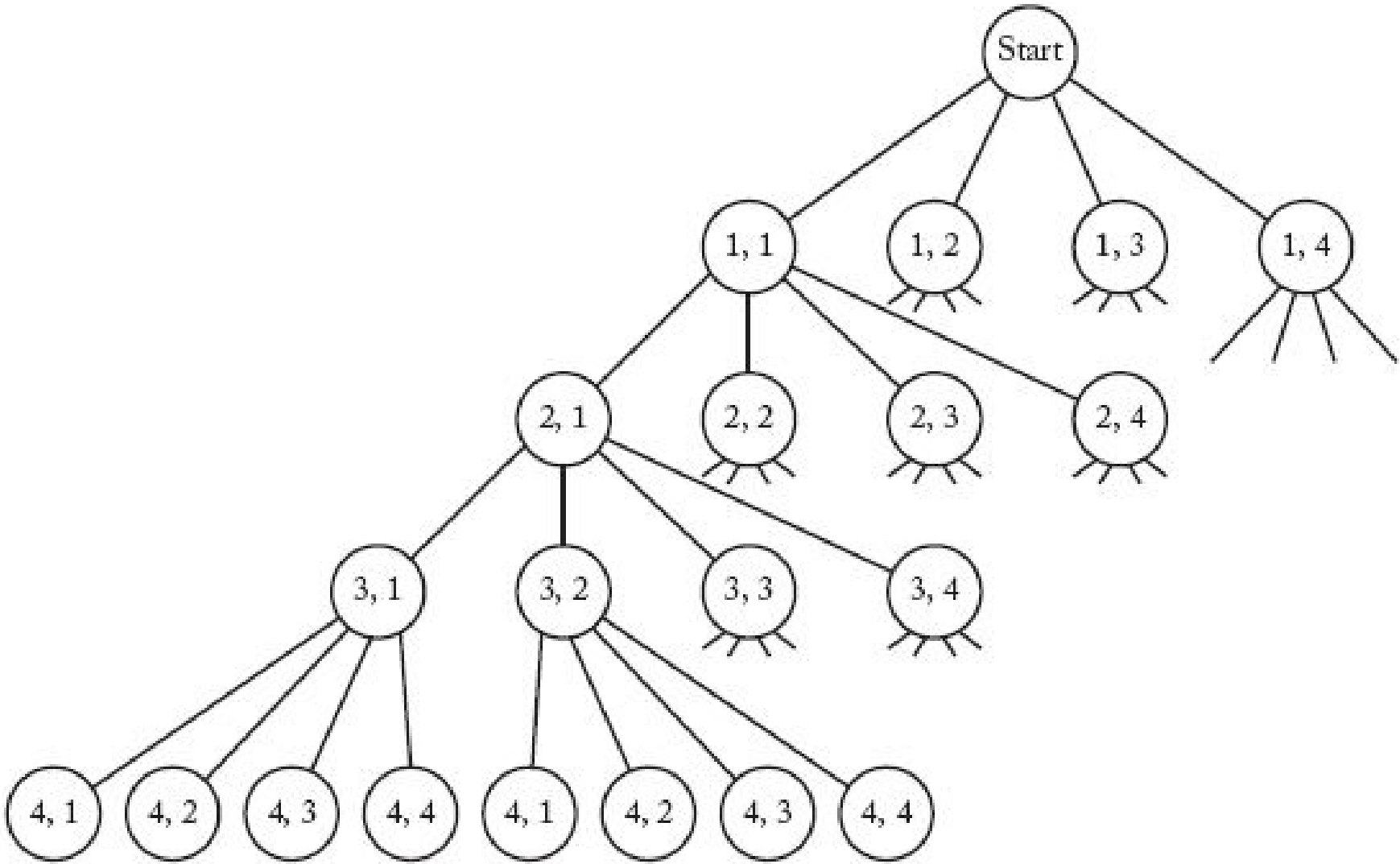

1
2
8
11
3
4
7
9
10
12
13
16
5
6
14
15

There is a simple recursive algorithm for doing a depth-first search:

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

The procedure is called by passing the root at the top level. This general-purpose algorithm does not state that the children must be visited in any particular order. However, as mentioned previously, we visit them from left to right.

Now let's illustrate the backtracking technique with the instance of the n-Queens problem when $n = 4$. Our task is to position four queens on a 4×4 chessboard so that no two queens threaten each other. We can immediately simplify matters by realizing that no two queens can be in the same row. The instance can then be solved by assigning 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 $4 \times 4 \times 4 \times 4 = 256$ candidate solutions. 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 (recall that the root is at level 0), the column choices for the second queen (the queen in row 2) are stored in level-2 nodes, and so on. A path from the root to a leaf is a candidate solution (recall that a leaf in a tree is a node with no children). This tree is called a state space tree.


Start
1, 1
1, 2
1, 3
1, 4
2, 1
2, 2
2, 3
2, 4
3, 1
3, 2
3, 3
3, 4
4, 1
4, 2
4, 3
4, 4
4, 1
4, 2
4, 3
4, 4

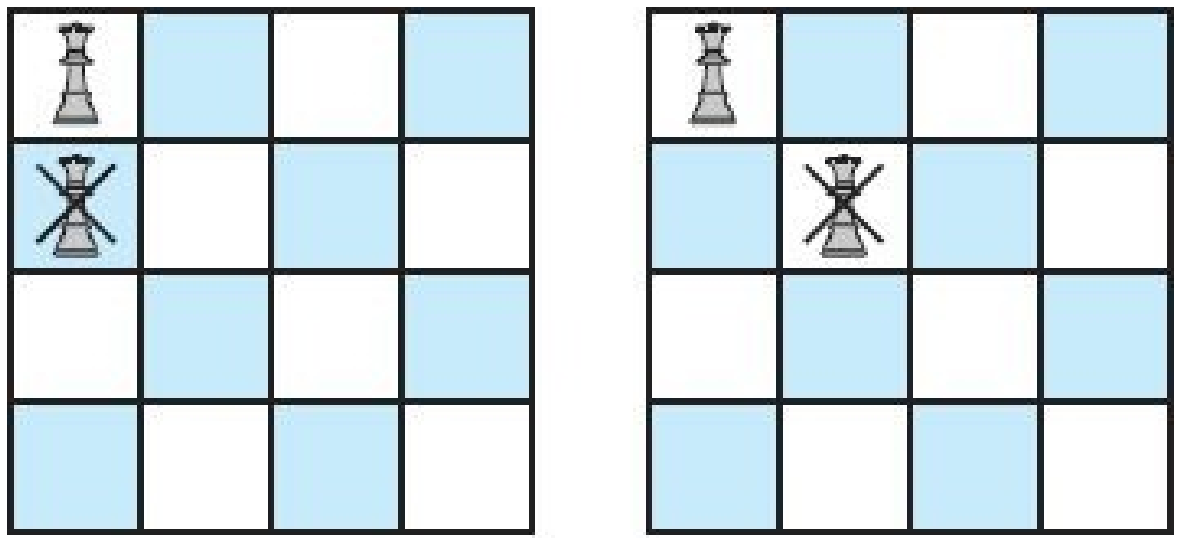
The entire tree has 256 leaves, one for each candidate solution. Notice that an ordered pair $\langle i, j \rangle$ is stored at each node. This ordered pair means that the queen in the ith row is placed in the jth column. To determine the solutions, we check each candidate solution (each path from the root to a leaf) in sequence, starting with the leftmost path.

The first few paths checked are as follows:

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

Notice that the nodes are visited according to a depth-first search in which the children of a node are visited from left to right.

A simple depth-first search of a state space tree is like following every path in a maze until you reach a dead end. It does not take advantage of any signs along the way. We can make the search more efficient by looking for such signs.



(a) (b)

For example, as illustrated in the above figure, no two queens can be in the same column. Therefore, there is no point in constructing and checking any paths in the entire branch emanating from the node containing $\langle 2, 1 \rangle$ in the figure. (Because we have already placed queen 1 in column 1, we cannot place queen 2 there.) This sign tells us that this node can lead to nothing but dead ends. Similarly, as illustrated in the figure, no two queens can be on the same diagonal. Therefore, there is no point in constructing and checking the entire branch emanating from the node containing $\langle 2, 2 \rangle$ in our figure.

جمع بندی (مهم)

Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("back-track") to the node's parent and proceed with the search on the next child. We call a node nonpromising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it promising. To summarize, 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 to the node's parent. This is called pruning the state space tree, and the subtree consisting of the visited nodes is called the pruned state space tree.

A general algorithm for the backtracking approach

```
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);
}
```

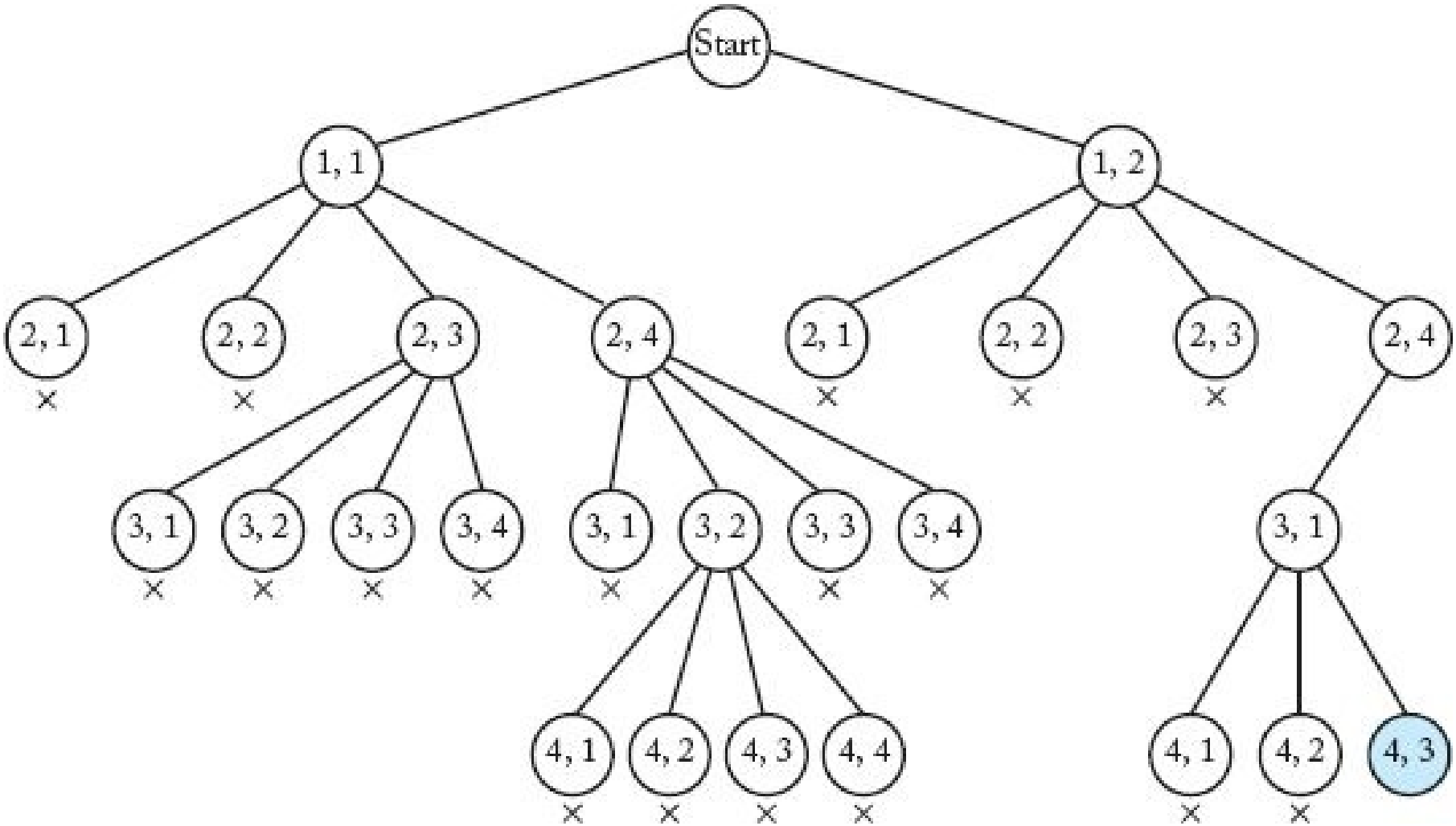
The root of the state space tree is passed to checknode at the top level.

* *A visit to a node consists of first checking whether it is promising. If it is promising and there is a solution at the node, the solution is printed. If there is not a solution at a promising node, the children of the node are visited.*

* *The function $promising$ is different in each application of backtracking. A backtracking algorithm is identical to a depth-first search, except that the children of a node are visited only when the node is promising and there is not a solution at the node.*

* *Unlike the algorithm for the n-Queens problem, in some backtracking algorithms a solution can be found before reaching a leaf in the state space tree.*

The instance of the n-Queens problem when $n = 4$



A node count in the above figure shows that the backtracking algorithm checks 27 nodes before finding a solution. It can be shown that, without backtracking, a depth-first search of the state space tree checks 155 nodes before finding that same solution:

$$\underbrace{(1+4+16+64)}_{rooted\ at\ <1,1>} + \underbrace{1}_{<1,2>} +$$

$$\underbrace{3 \times (1+4+16)}_{rootet\ at\ <2,1>,\ rootet\ at\ <2,2>,\ rootet\ at\ <2,3>} +$$

$$\underbrace{5}_{rootet\ at\ <2,4>} + \underbrace{1}_{root} = 155$$

The instance of the n-Queens problem when $n = 4$

(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 2 is in column 3}
<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>.}

The instance of the n-Queens problem when $n = 4$

(f) <4, 1> is nonpromising. {because queen 1 is in column 1}
<4, 2> is nonpromising. {because queen 3 is in column 2}
<4, 3> is nonpromising. {because queen 3 is on left diagonal}
<4, 4> is nonpromising. {because queen 2 is in column 4}

(g) Backtrack to <2, 4>.
<3, 3> is nonpromising. {because queen 2 is on right diagonal}
<3, 4> is nonpromising. {because queen 2 is in column 4}

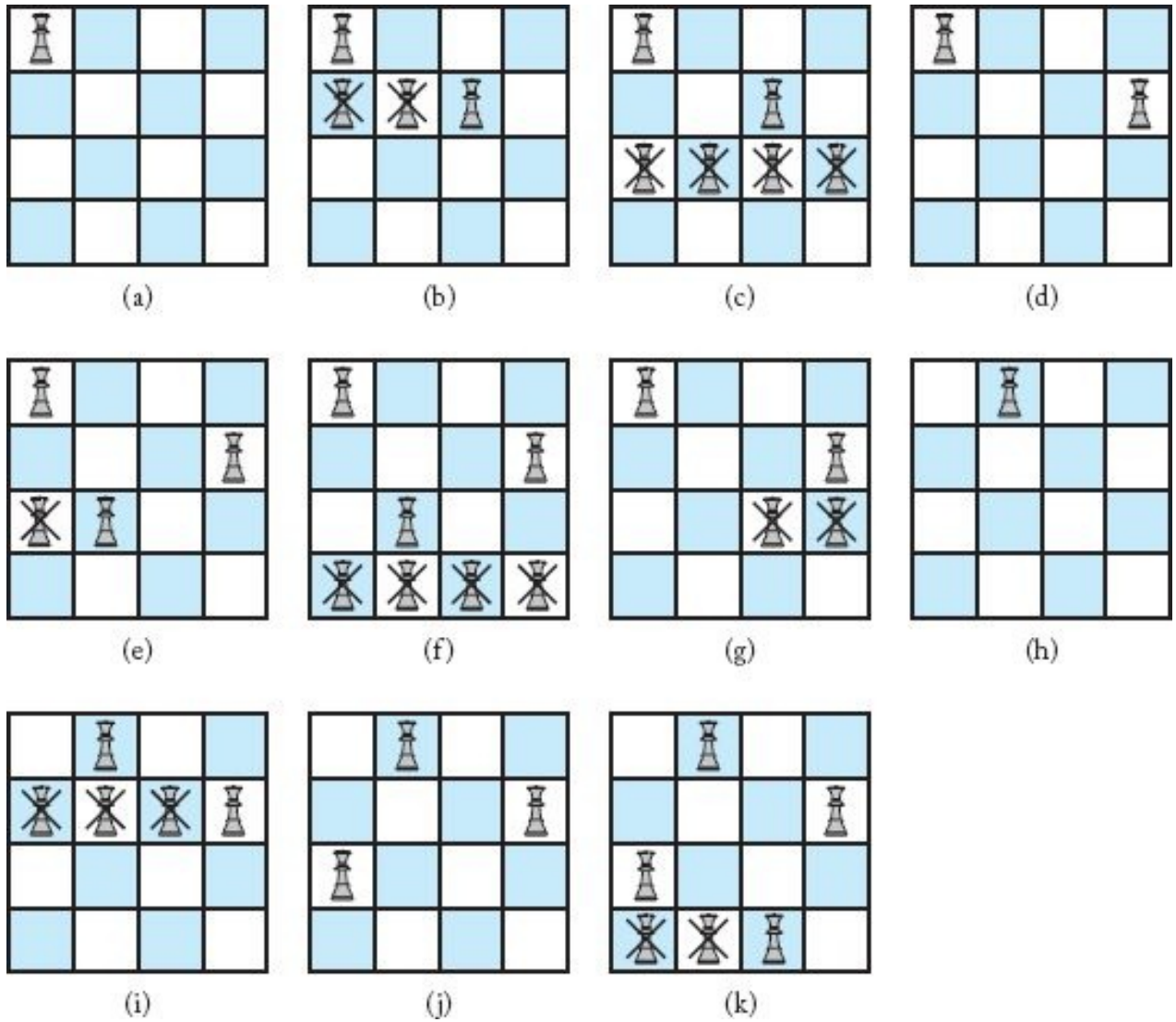
(h) Backtrack to root.
<1, 2> is promising.

(i) <2, 1> is nonpromising. {because queen 1 is on right diagonal}
<2, 2> is nonpromising. {because queen 1 is in column 2)
<2, 3> is nonpromising. {because queen 1 is on left diagonal}
<2, 4> is promising.

(j) <3, 1> is promising. {This is the third time we've tried <3, 1>.}

(k) <4, 1> is nonpromising. {because queen 3 is in column 1}
<4, 2> is nonpromising. {because queen 1 is in column 2}
<4, 3> is promising.

The instance of the n-Queens problem when $n = 4$



در عمل، درختی در کار نیست!!!

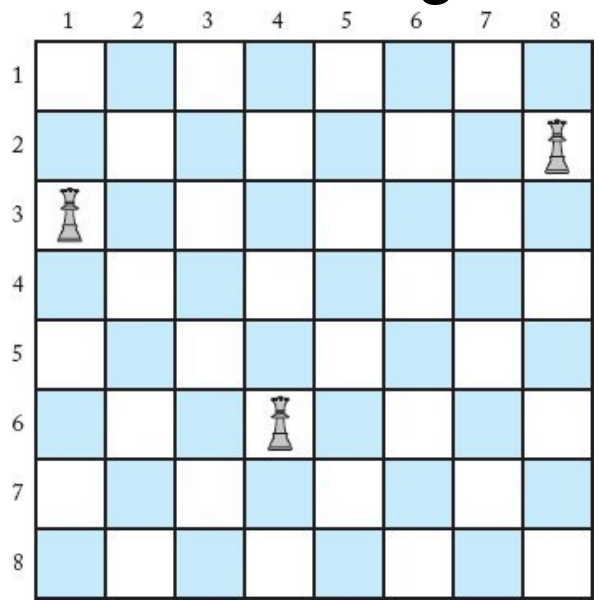
Notice that a backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithms. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

☞ *The promising function must check whether two queens are in the same column or diagonal.*

☞ *If we let $col(i)$ be the column where the queen in the ith row is located, then to check whether the queen in the kth row is in the same column, we need to check whether*

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

☞ *Next let's see how to check the diagonals:*



The above figure illustrates the instance in which $n = 8$. In that figure, 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. 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. These are examples of the general result that if the queen in the kth row threatens the queen in the ith row along one of its diagonals, then

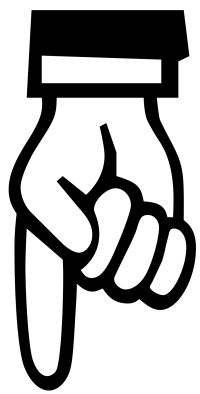
$$col(i) - col(k) = i - k \quad \textbf{or} \quad col(i) - col(k) = k - i.$$

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 $n \times n$ 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
           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;
}
```

☞ n *and* col *are not inputs to the recursive routine* $queens$*. If the algorithm were implemented by defining* n *and* col ***globally****, the top-level call to* $queens$ *would be* $queens(0);$

☞ *The above produces* ***all*** *solutions to the* n*-Queens problem because that is how we stated the problem. We stated the problem this way to eliminate the need to exit when a solution is found.*

☞ *Most of our algorithms are written to produce all solutions. It is a simple modification to make the algorithms stop after finding one solution.*

دقیقاً چه تعداد نود ملاقات می‌شوند ؟ نمی‌دانیم! اما:

We can get an upper bound on the number of nodes in the pruned state space tree by counting the number of nodes in the entire state space tree:

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

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

Another analysis we could try is to obtain an upper bound on the number of promising nodes. To compute such a bound, we can use the fact that 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. Once the first queen is positioned, the second can be positioned in at most seven columns; once the second is positioned, the third can he positioned in at most six columns; and so on. Therefore, there are at most

$$1 + 8 + 8 \times 7 + 8 \times 7 \times 6 + 8 \times 7 \times 6 \times 5 + \cdots + 8!$$

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

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

$$1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n! \text{ promising nodes.}$$

(اینهم زیاد خوب نیست!)

DFS vs Backtracking

n	# of Solutions	# of Visited Nodes
4	2	15 / 341
5	10	44 / 3906
6	4	149 / 55987
7	40	512 / 960800
8	92	1965 / 19173961
9	352	8042 / 435848050
10	724	34815 / 11111111111

The Sum-of-Subsets Problem

In the 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.

هدف ما یافتن همهٔ جواب‌های ممکن، یا لااقل یک جواب ممکن است (در صورت وجود). مسئلهٔ ***sum-of-subsets*** یک مسئلهٔ تصمیم‌گیری است و نه بهینه‌سازی (بر خلاف مسئلهٔ ***0-1 knapsack***).

دقت کنید که مسئلهٔ *sum-of-subsets* با مسئلهٔ *0-1 knapsack* تفاوت دارد. (تفاوت در چیست؟)

Example

$$n = 5,\ W = 21,\ \{w_1 = 5, w_2 = 6, w_3 = 10, w_4 = 11, w_5 = 16\}$$

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

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

for $\{1, 2, 5, 6, 8\}$ ***and*** $W = 9$, ***there are two solutions:*** $\{1, 2, 6\}$ ***and*** $\{1, 8\}$. ***Of course, some instances of this problem may have no solutions.***

این مسئله یک مسئلهٔ تصمیم‌گیری ***NP-complete*** است.