Chapter 5.

Backtracking

Foundations of Algorithms, 5th Ed.

Richard E. Neapolitan

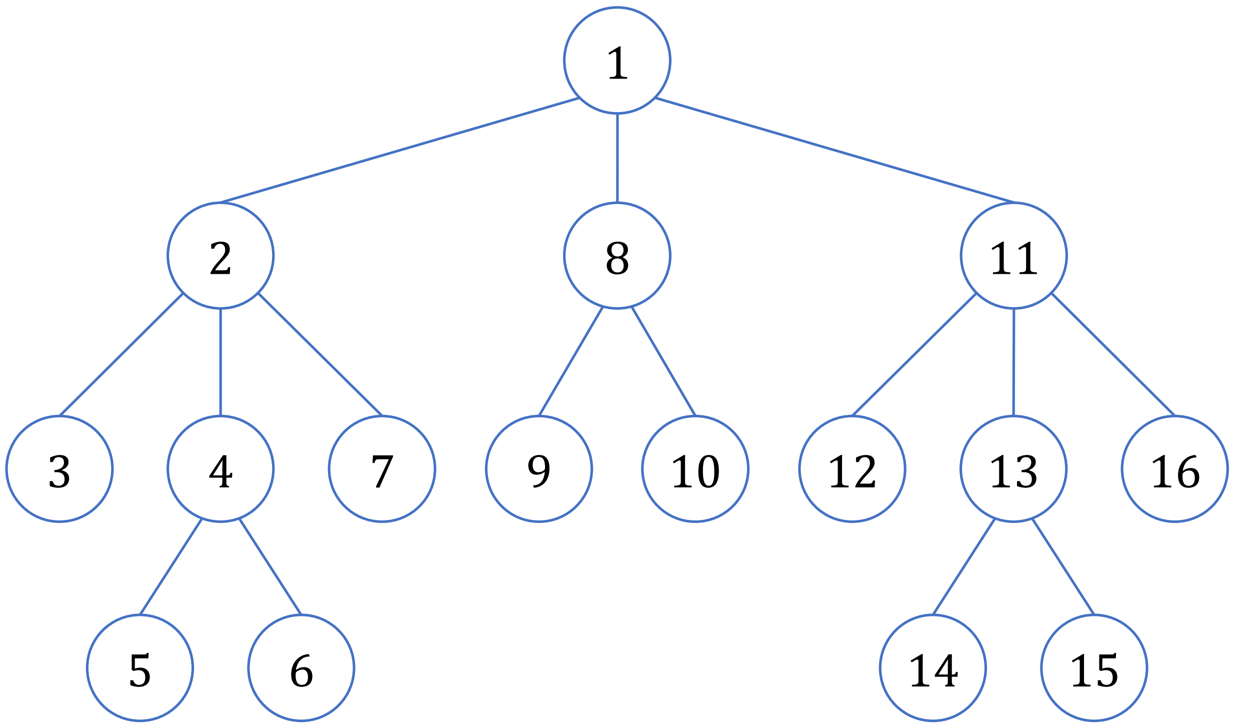
Contents

5.1 The Backtracking Technique

- ***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*.
  - For example,
    - n-Queens problem
    - Sum-of-Subsets problem
    - Graph Coloring problem
    - Hamiltonian Circuits problem
    - 0-1 Knapsack problem

5.1 The Backtracking Technique

- Backtracking
  - is a *modified depth-first-search* (DFS) of a tree.
  - Note that a *preorder tree traversal* is a depth-first-search in the tree.



5.1 The Backtracking Technique

- A simple algorithm for doing a depth-first-search:

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

- The *n-Queens* Problem:
  - The goal 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.
  - The *criterion* is that
    - *no two queens* can threaten each other.

5.1 The Backtracking Technique

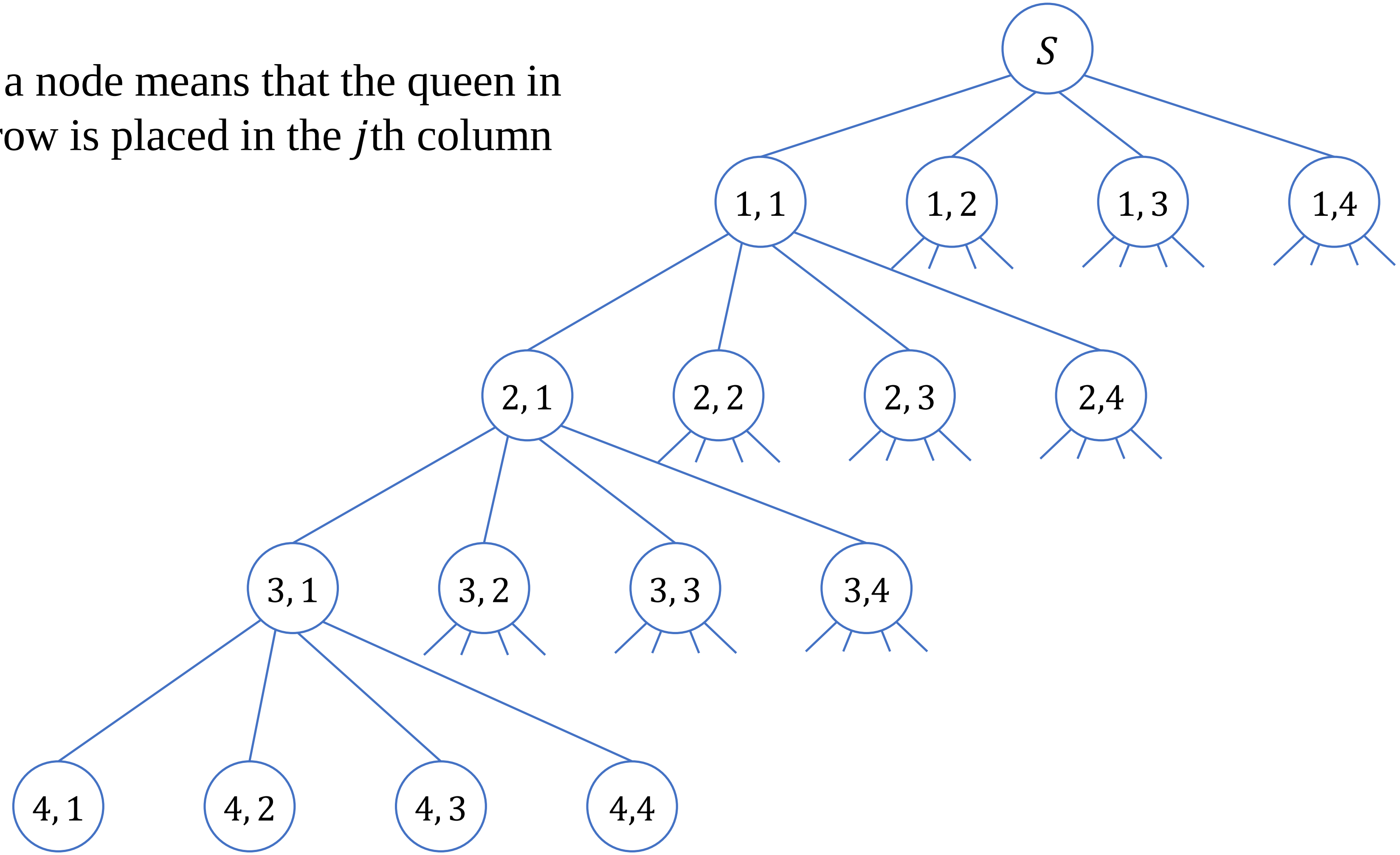
- Backtracking for the n-Queens Problem:
  - When $n = 4$, our task is
    - to position 4 queens on a 4×4 chessboard.
  - We can immediately *simplify* matters
    - by realizing that *no two queens* can be placed in the *same row*.
  - Then, the instance can be solved
    - by *assigning* each queen *a different row*,
    - and *checking* which *column combinations* yield solutions.
  - Because each queen can be place in one of four columns,
    - there are $4 \times 4 \times 4 \times 4 = 256$ candidate solutions.

5.1 The Backtracking Technique

- The ***State Space Tree***:
  - A *state space tree* is a tree of *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 (the root is at level 0).
    - The column choices for the first queen (the queen in row 2)
      - are stored in level-2 nodes in the tree, and so on.
  - A *candidate solution* is a *path* from the *root* to a *leaf node*.

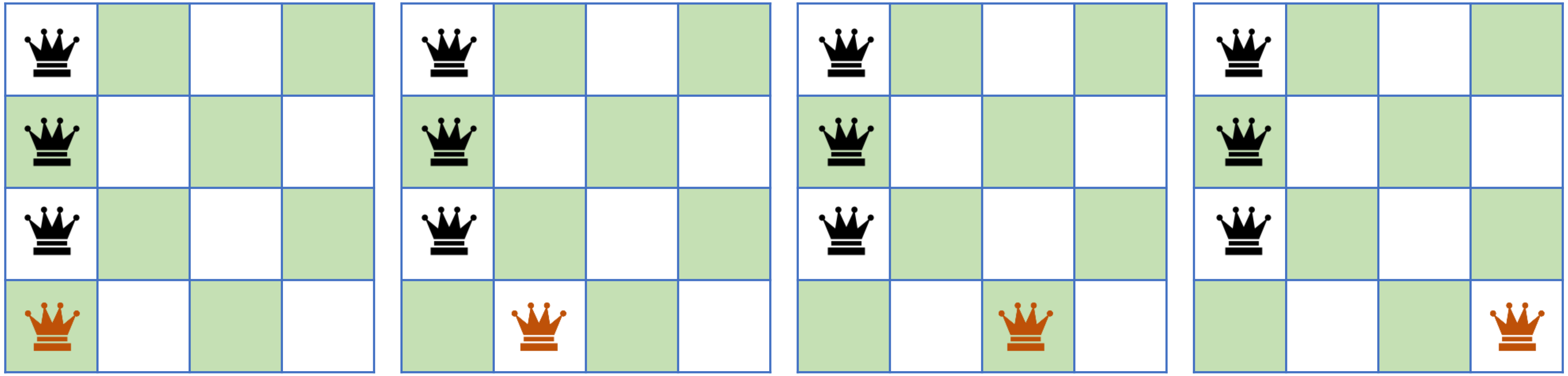
5.1 The Backtracking Technique

- (i, j) in a node means that the queen in the ith row is placed in the jth column



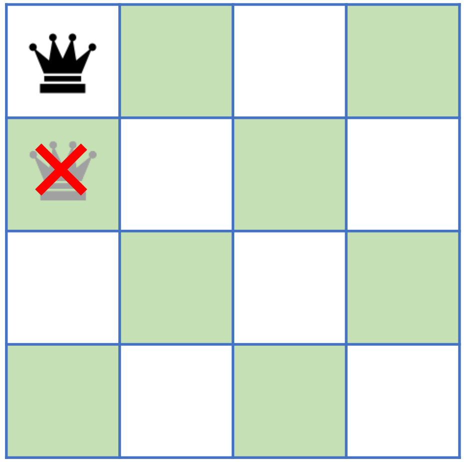
5.1 The Backtracking Technique

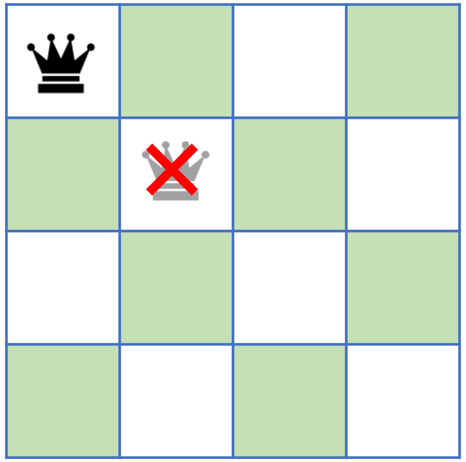
- *Searching* the State Space Tree:
  - To determine the solutions, check each candidate solution in sequence,
    - for each path from the root to a leaf, starting with the leftmost path.
  - Note that a simple *depth-first-search* of a tree
    - follows *every path* in the *state space tree*.



5.1 The Backtracking Technique

- *More Efficient Search* in the State Space Tree:
  - We can make the search more efficient
    - by taking advantage of any *sign* (*criterion*) along the *search path*.
  - There are two signs in the problem:
    - *No two queens* can be in the *same column* or *diagonal*.

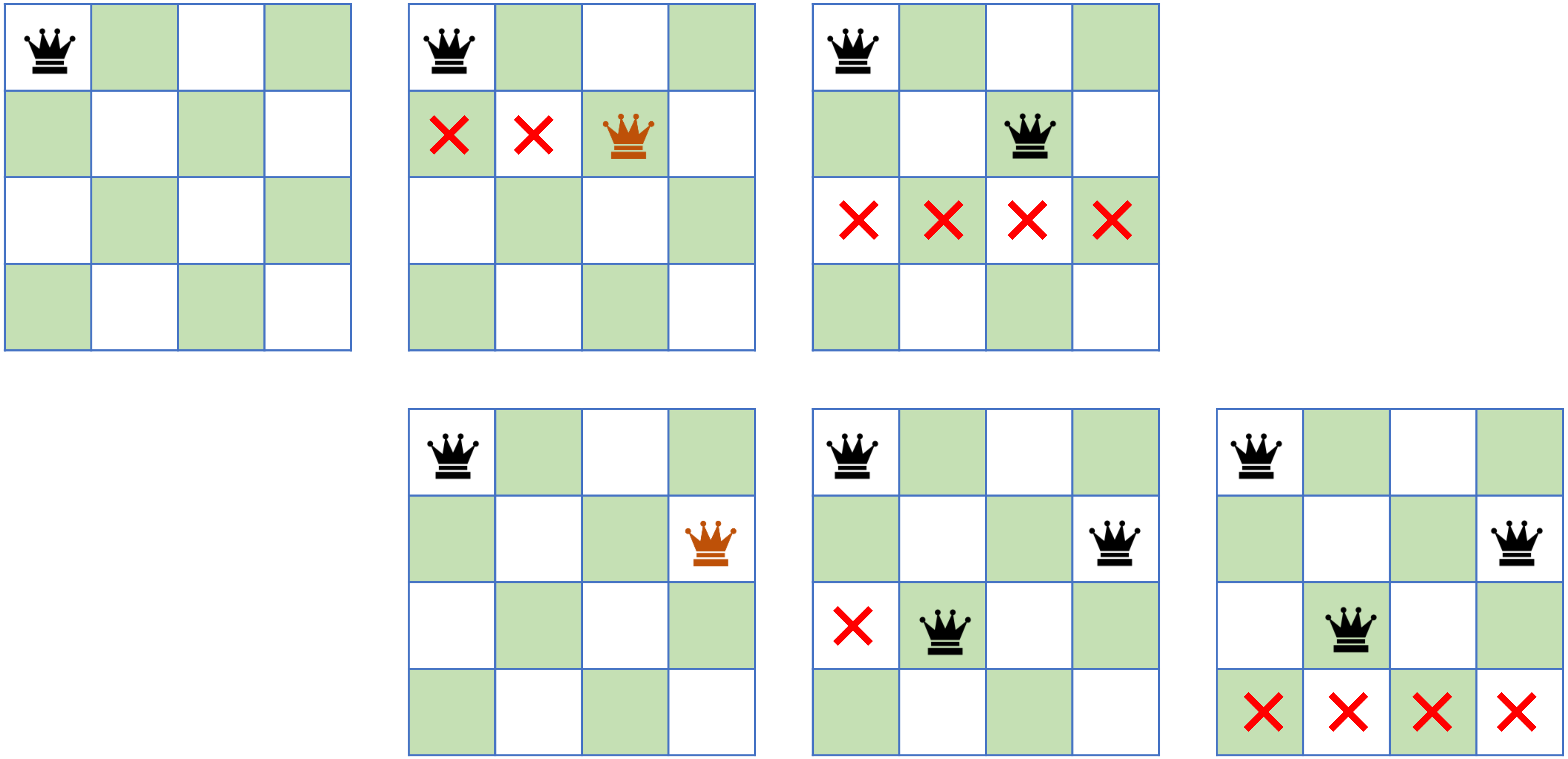




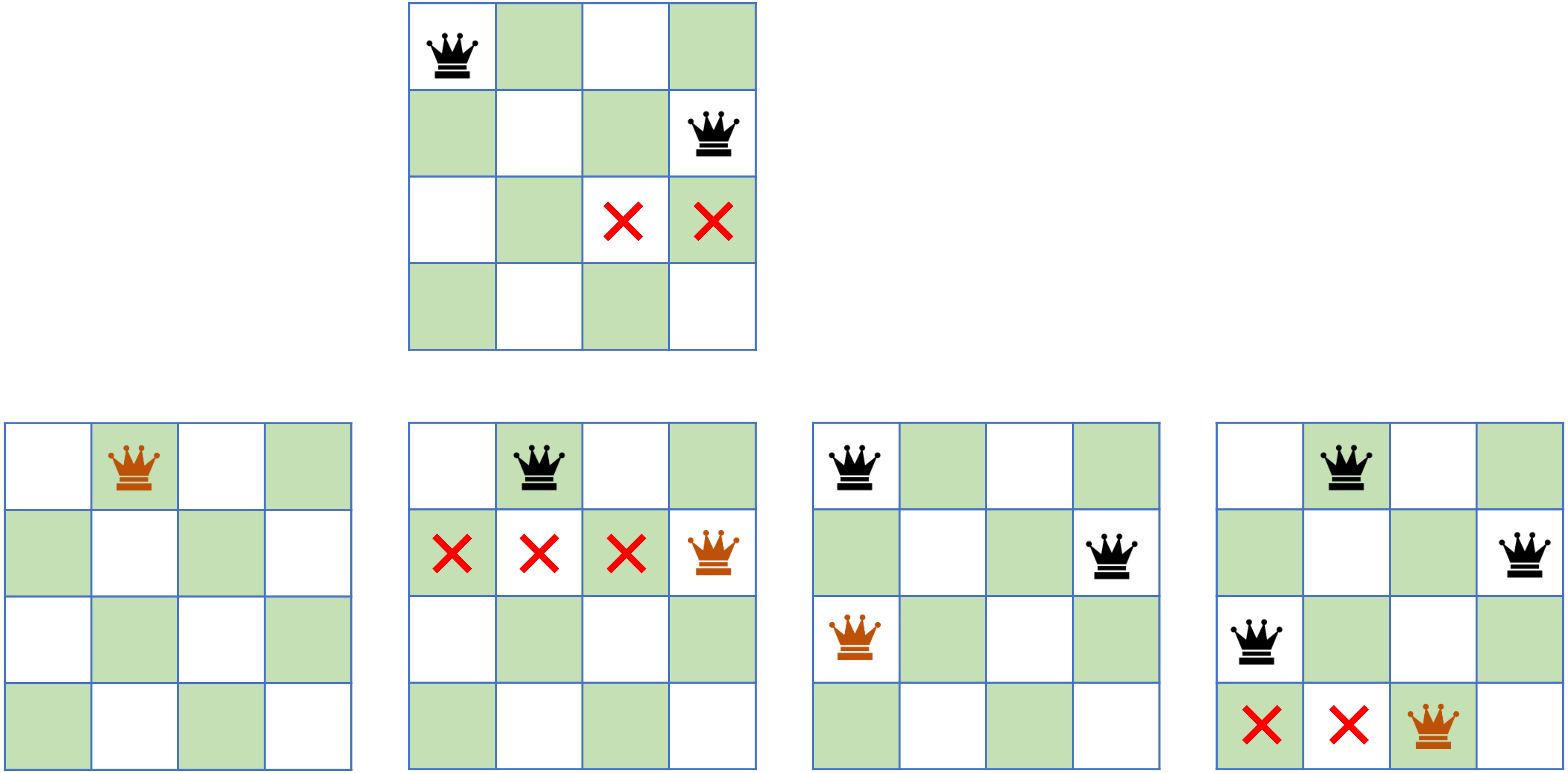
5.1 The Backtracking Technique

- The concepts of *Promising* and *Pruning*:
  - Backtracking is the *procedure* whereby,
    - after determining that a node can lead to nothing but dead ends,
    - we *go back* (*backtrack*) to the parent and *proceed* on the next child.
  - A node is ***nonpromising***
    - if it cannot possibly lead to a solution when visiting the node.
    - Otherwise, a node is ***promising***.
  - ***Pruning*** the state space tree is
    - *backtracking* to the parent node if the node is *nonpromising*.

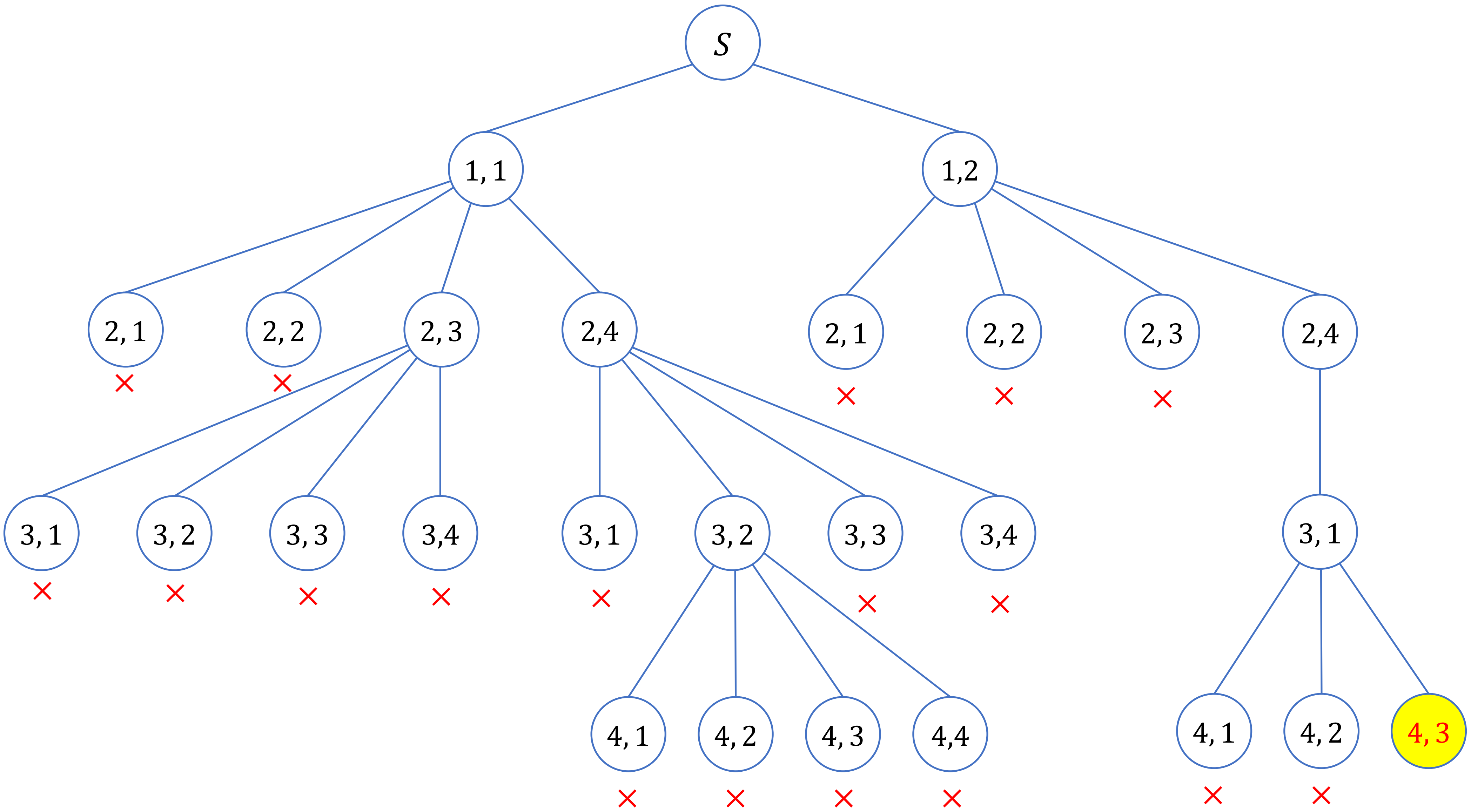
5.1 The Backtracking Technique



5.1 The Backtracking Technique



5.1 The Backtracking Technique



5.1 The Backtracking Technique

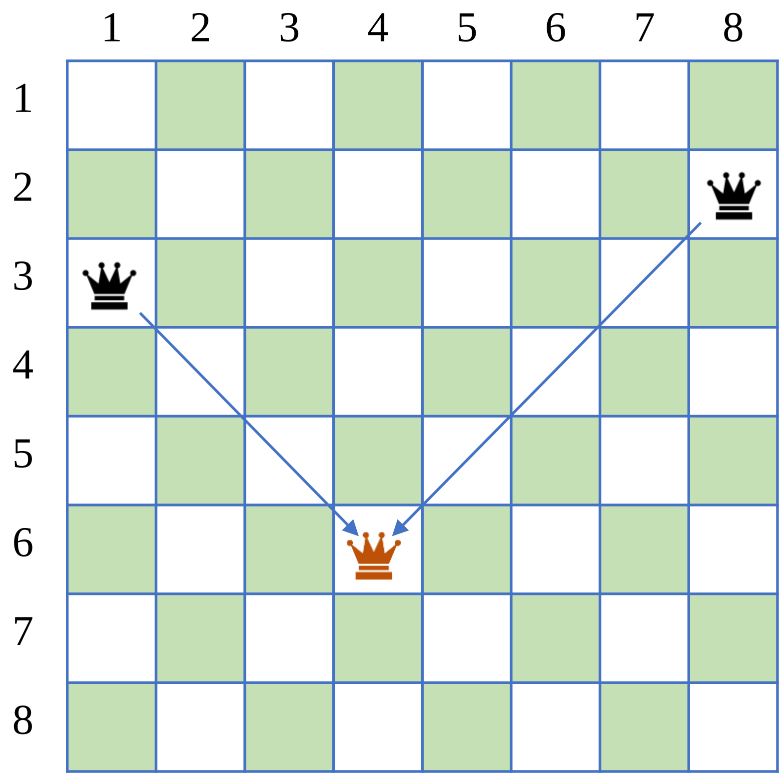
- A general algorithm for the backtracking approach:

```
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);
    }
}
```

- Solving the n-Queens Problem:
  - The *promising function* must check
    - whether *two queens* are in the *same column* or *diagonal*.
  - Let $col(i)$ be the *column*
    - where the queen in the *ith row* is located.
  - We need to check $col(i) = col(k)$,
    - to check whether two queens are in the *same column*.

5.2 The n-Queens Problem

- Checking the diagonal:
  - The queen in *row 6* is threatened by
    - the queen in *row 3*: $col(6) - col(3) = 4 - 1 = 3 = 6 - 3$.
    - the queen in *row 2*: $col(6) - col(2) = 4 - 8 = -4 = 2 - 6$.
  - Check $|col(i) - col(k)| = |i - k|$
    - to check whether two queens are in the *same diagonal*.



5.2 The *n*-Queens Problem

ALGORITHM 5.1: The Backtracking Algorithm for the n-Queens Problem

```
void queens(int i) {
    int j;

    if (promising(i)) {
        if (i == n)
            cout << col[1] through col[n];
        else
            for (j = 1; j <= n; j++) {
                col[i + 1] = j;
                queens(i + 1);
            }
    }
}
```

5.2 The *n*-Queens Problem

ALGORITHM 5.1: The Backtracking Algorithm for the n-Queens Problem

```
bool promising(int i) {
    int k = 1;
    bool flag = true;

    while (k < i && flag) {
        if ((col[i] == col[k]) || (abs(col[i] - col[k]) == i - k))
            flag = false;
        k++;
    }
    return flag;
}
```

5.2 The n-Queens Problem

- Complexity Analysis of the Algorithm 5.1
  - An *upper bound* can be the total number of nodes in the *entire tree*.
    - $1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1}-1}{n-1}$.
    - When $n = 8$, the *state space tree* contains $\frac{8^9-1}{8-1} = 19{,}173{,}961$ nodes.
  - Another *upper bound* can be the *number of promising nodes*,
    - using the fact that no two queens can be placed in the same column.
    - $1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n!\mathrm{n}$
    - When $n = 8$, $1 + 8 + 8 \times 7 \cdots + 8! = 109{,}601$ *promising* nodes.
  - In general, it is *difficult*
    - to analyze the complexity of backtracking algorithm *theoretically*.

5.2 The *n*-Queens Problem

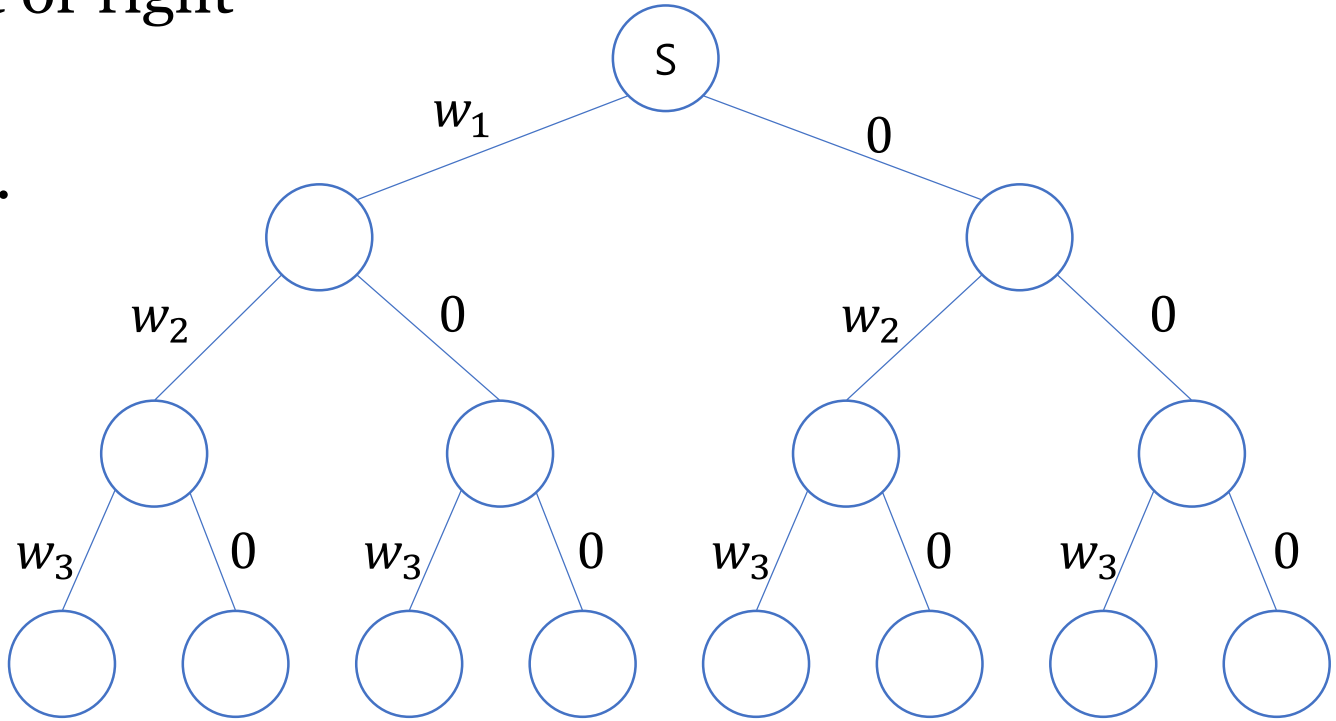
- Using a *Monte-Carlo Algorithm*:
  - A straightforward way to determine the efficiency of the algorithm is
    - to *actually run* the algorithm on a computer
    - and *count* how many *nodes* are *checked*.
  - *Deterministic* .vs. *Probabilistic* algorithm.
    - In a probabilistic algorithm, the next instruction executed
      - is sometimes determined at random with a probabilistic distribution.
  - Monte-Carlo algorithms are *probabilistic* algorithms.
  - A Monte-Carlo algorithm estimates
    - the *expected value* of a *random variable*,
    - from its average value on a random sample of the *sample space*.

5.4 The Sum-of-Subsets Problem

- The Sum-of-Subsets Problem:
  - There are n positive integers w_i and a positive integer W.
  - The goal of the sum-of-subsets problem is
    - to find *all subsets* of the integers that *sum to W*.
  - For example,
    - $n = 5$, $W = 21$, and $w_i = [5, 6, 10, 11, 16]$.
  - The solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.
    - $w_1 + w_2 + w_3 = 5 + 6 + 10 = 21$,
    - $w_1 + w_5 = 5 + 16 = 21$,
    - $w_3 + w_4 = 10 + 11 = 21$.

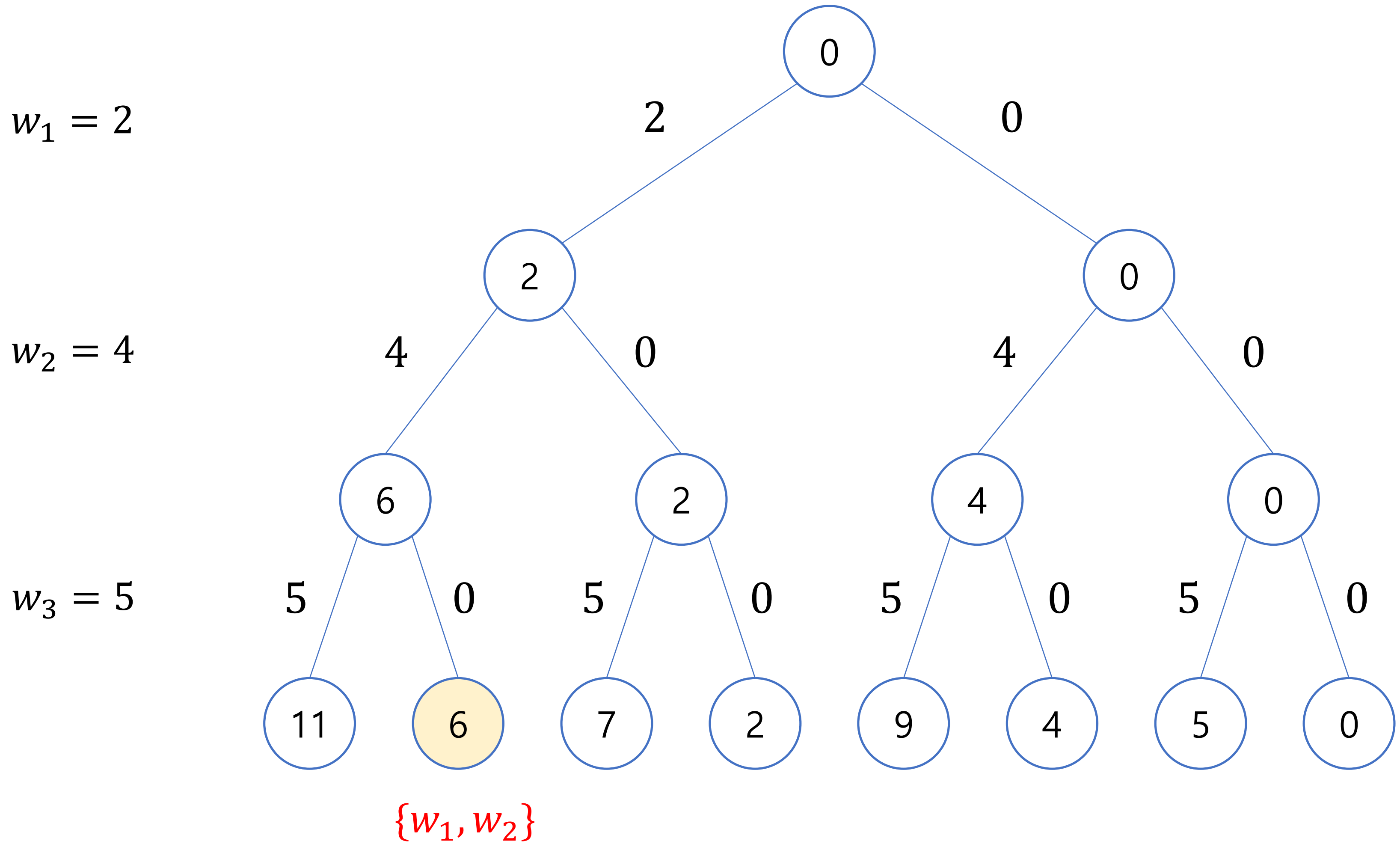
5.4 The Sum-of-Subsets Problem

- Creating the *State Space Tree*:
  - Go to the left from the root to include w_1.
    - and go to the right from the root to exclude w_1.
  - Similarly, we go to the left or right
    - from a node at level i
    - to include or exclude w_i.



5.4 The Sum-of-Subsets Problem

- $n = 3, W = 6, w_i = \{\, 2, 4, 5 \,\}$



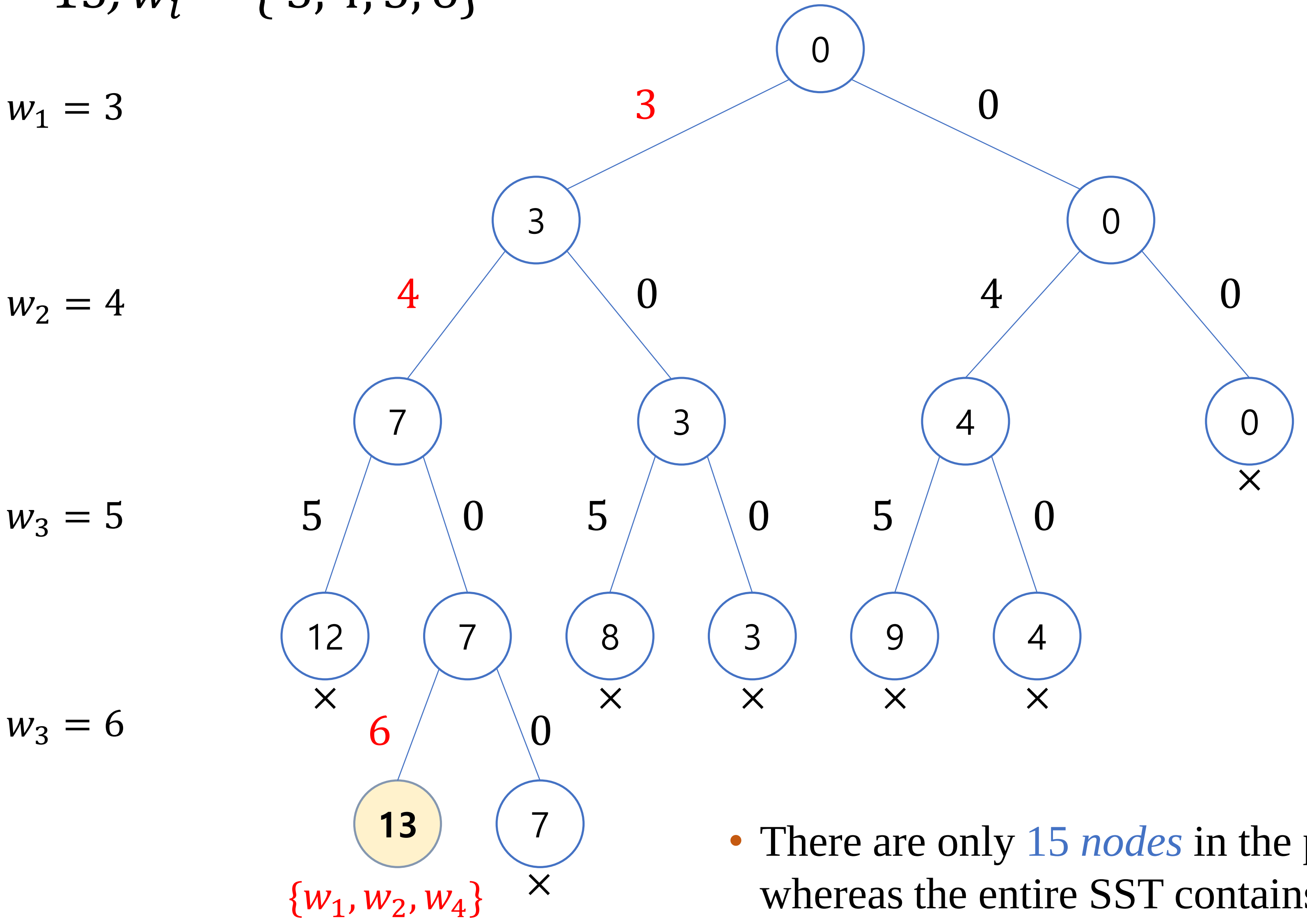
5.4 The Sum-of-Subsets Problem

- *Pruning* Strategies:
  - An *obvious sign* telling us that a node is *promising*.
  - If we sort the weights in nondecreasing order before doing the search,
    - then w_{i+1} is the lightest weight remaining at the ith level.
  - Let *weight* be the sum of the weights included up to a node at level i.
  - If w_{i+1} would bring the value of *weight* *above* W,
    - then so would any other weight following it.
  - Therefore, a node at the ith level is *nonpromising* if
    - $weight + w_{i+1} > W$.

- *Pruning* Strategies:
  - Another *less obvious sign* telling us that a node is *promising*.
  - If adding all the weights of the remaining items to *weight*
    - *does not* make *weight* at least equal to W,
    - then *weight* could *never become* equal to W.
  - This means that if *total* is the total weight of the remaining weights,
    - a node is nonpromising if
    - $weight + total < W$.

5.4 The Sum-of-Subsets Problem

- $n = 4, W = 13, w_i = \{3, 4, 5, 6\}$



- There are only 15 *nodes* in the pruned SST, whereas the entire SST contains 31 *nodes*.

ALGORITHM 5.4: The Backtracking Algorithm for the Sum-of-Subsets Problem

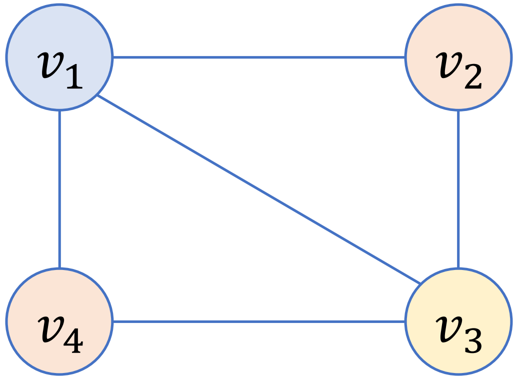
```
void sum_of_subsets(int i, int weight, int total) {
    if (promising(i, weight, total)) {
        if (weight == W)
            cout << include[1] through include[i];
        else {
            include[i + 1] = true;
            sum_of_subsets(i + 1, weight + w[i + 1], total - w[i + 1]);
            include[i + 1] = false;
            sum_of_subsets(i + 1, weight, total - w[i + 1]);
        }
    }
}

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

5.4 The Sum-of-Subsets Problem

- Algorithm 5.4 Explained:
  - As usual, n, W, w, and *include* are defined as *global variables*.
  - The top-level call to the algorithm would be
    - $sum\_of\_subsets(0, 0, total);$
    - where initially $total = \sum_{j=1}^{n} w[j]$.
  - The algorithm *needs not to check* for the terminal condition $i = n$,
    - because a leaf that does not contain a solution is nonpromising.
  - The number of nodes in the state space tree is equal to
    - $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$.
  - The Sum-of-Subsets problem is
    - in the class of the *NP-Complete* problems.

5.5 Graph Coloring

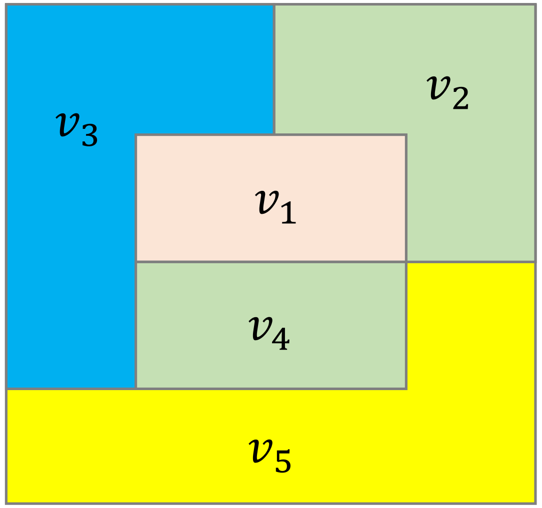
- The ***m-Coloring*** Problem:
  - concerns finding all ways to color an undirected graph
    - using at most m different colors,
    - so that *no two adjacent vertices* are the *same color*.

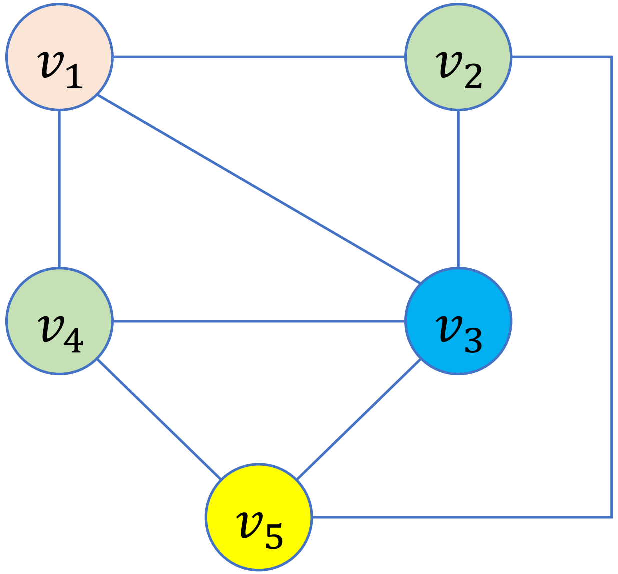


- There is no solution to the 2-Coloring problem.
- Total 6 solutions to the 3-Coloring problem:
  - One solution is colored in the left graph.
  - Note that all the six solutions are
    - only different in the way the colors are permuted.

5.5 Graph Coloring

- The *Coloring* of *Maps*:
  - A graph is called ***planar*** if it can be drawn in a plane
    - in such a way that *no two edges cross* each other.
  - To *every map*, there exist a corresponding *planar graph*.
    - Each *region* in the map is represented by a *vertex*.
    - An *edge* represents that one region is *adjacent* to another region.



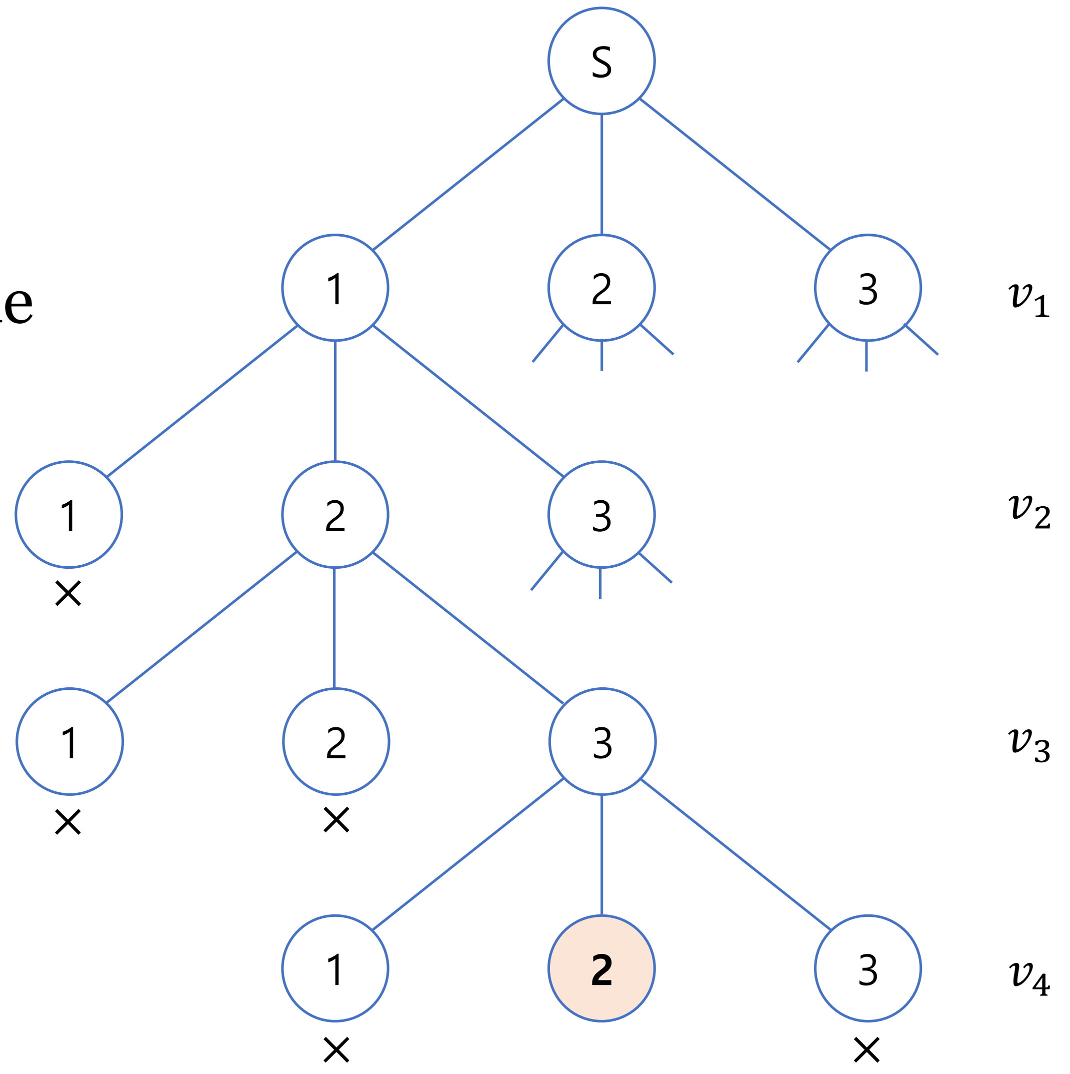


5.5 Graph Coloring

- The m-Coloring Problem for Planar Graph:
  - is to determine how many ways the map can be colored,
    - using at most m colors,
    - so that no two adjacent regions are the same color.
  - A straightforward *state space tree* for the problem is one
    - in which each possible color is tried for vertex v_1 at level 1,
    - each possible color is tried for vertex v_2 at level 2, and so on,
    - until each possible color has been tried for vertex v_n at level n.
  - Then, each path from the root to a leaf is a candidate solution.

5.5 Graph Coloring

- *Pruning* Strategies:
  - A node is *nonpromising*
    - if a vertex that is adjacent to
      - the vertex being colored at the node
    - has already been colored the color
      - that is being used at the node.



ALGORITHM 5.5: The Backtracking Algorithm for the m-Coloring Problem

```
void m_coloring(int 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: The Backtracking Algorithm for the m-Coloring Problem (continued)

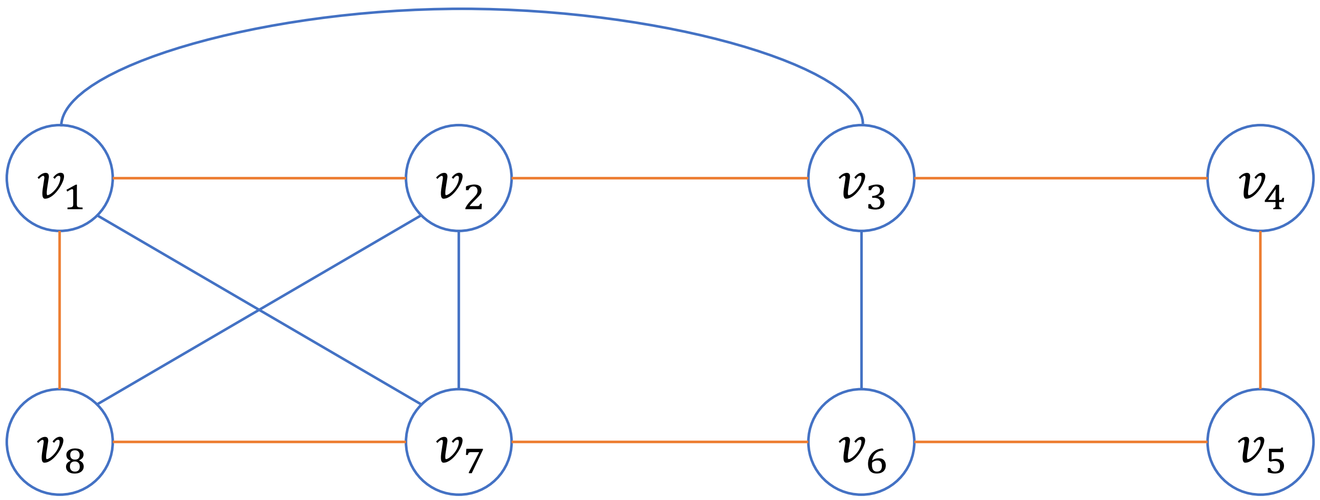
```
bool promising(int i) {
    int j = 1;
    bool flag = true;

    while (j < i && flag) {
        if (W[i][j] && vcolor[i] == vcolor[j])
            flag = false;
        j++;
    }
    return flag;
}
```

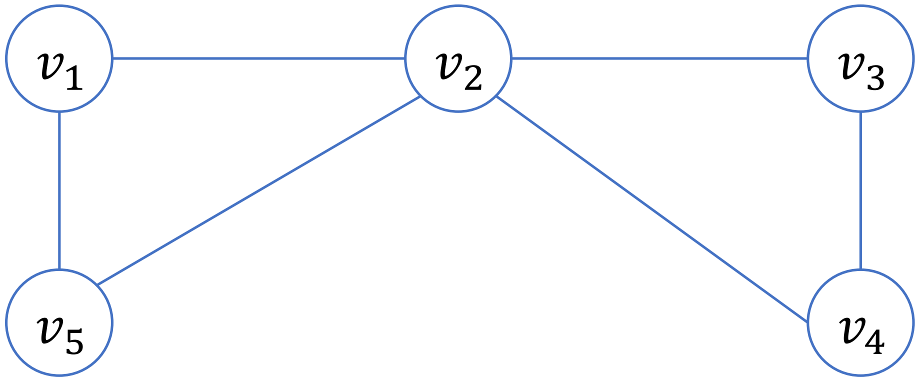
- Algorithm 5.5 Explained:
  - As usual, n, m, W, and $vcolor$ are defined globally.
  - The top level call to $m\_coloring$ would be
    - $m\_coloring(0);$
  - The number of nodes in the state space tree is equal to
    - $1 + m + m^2 + \cdots + m^n = \frac{m^n - 1}{m - 1}.$
  - The *m-Coloring problem* for $m \geq 3$ is
    - in the class of the *NP-Complete* problems.

5.6 The Hamiltonian Circuits Problem

- The *Hamiltonian Circuits Problem*:
  - Given a connected, undirected graph, a ***Hamiltonian Circuit*** is
    - a path that *starts at* a given vertex,
    - *visits each vertex* in the graph *exactly once*,
    - and *ends at* the starting vertex.
  - The *Hamiltonian Circuits problem* is
    - to determine the Hamiltonian Circuits in a given graph.

5.6 The Hamiltonian Circuits Problem





5.6 The Hamiltonian Circuits Problem

- *Pruning* Strategy:
  - A *state space tree* for this problem is as follows.
    - Put the starting vertex at level 0 in the tree: the *zero*th vertex.
    - 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.

- *Pruning* Strategy:
  - *Backtracking considerations* in the state space tree:
    - The ith vertex on the path
      - must be *adjacent* to the $(i-1)$st vertex on the path.
    - The $(n-1)$st vertex
      - must be *adjacent* to the 0th vertex (the starting one).
    - The ith vertex *cannot be* one of the first $i-1$ vertices.

5.6 The Hamiltonian Circuits Problem

ALGORITHM 5.6: The Backtracking Algorithm for the Hamiltonian Circuits Problem

```
void hamiltonian(int i) {
    int j;

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

5.6 The Hamiltonian Circuits Problem

ALGORITHM 5.6: The Backtracking Algorithm for the Hamiltonian Circuits Problem

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

- **Algorithm 5.6 Explained:**
  - As usual, n, W, and $vindex$ are defined globally.
  - The top-level called to $hamiltonian$ would be
    - $vindex[0] \ = \ 1;$
    - $hamiltonian(0);$
  - The number of nodes in the state space tree is
    - $1 + (n-1) + (n-1)^2 + \cdots + (n-1)^n = \frac{(n-1)^n - 1}{n-2}.$
  - The Hamiltonian Circuits problem is
    - in the class of the *NP-Complete* problems.

5.7 The 0-1 Knapsack Problem

- The 0-1 Knapsack Problem:
  - We can solve this problem *using* *backtracking*.
  - The state space tree of this problem is
    - exactly like the one in the Sum-of-Subsets problem.
  - That is, we go to the *left or right* to *include or exclude* an item.
    - Each path from the root to a leaf is a candidate solution.

- The 0-1 Knapsack as an *Optimization* Problem:
  - This problem is different from the others
    - in that it is an optimization problem.
  - Therefore, we *backtrack* a little *differently*.
  - If the items included have a greater total profit than the best solution,
    - we change the value of the best solution so far.
  - However, we may still find a better solution afterwards.
  - Therefore, for optimization problems,
    - we always visit a promising node's children.

5.7 The 0-1 Knapsack Problem

- A general backtracking algorithm for the 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);
}
```

- *Pruning* Strategy:
  - An *obvious sign* that a node is *nonpromising*.
    - There is no capacity left in the knapsack for more items.
    - If *weight* is the sum of weights of the items included up to a node,
      - the node is *nonpromising* if $weight \geq W$.
  - Note that it is nonpromising even if weight equals to W,
    - in the case of optimization problems,
    - "*promising*" means that we should *expand to the children*.

- *Pruning* Strategy:
  - There is a *less obvious sign* that a node is *nonpromising*,
    - using greedy considerations to limit our search.
  - First, order the items in nonincreasing order
    - according to the values of p_i/w_i of the ith item.
  - Then, we can obtain an *upper bound* on the profit
    - that could be obtained by *expanding beyond that node*.
  - To that end,
    - Let *profit* be the sum of the profits of the items included.
    - Recall that *weight* is the sum of weights of those items.
  - Then, initialize *bound* and *totweight* to *profit* and *weight*, respectively.

5.7 The 0-1 Knapsack Problem

- *Pruning* Strategy:
  - Next, we greedily grab items,
    - adding their profits to *bound* and their weights to *totweight*,
    - until we get to an item that, if grabbed ,would bring *totweight* above W.
  - We grab the fraction of that item allowed by the remaining weight,
    - and we add the value of that fraction to *bound*.
  - If we are able to get only a fraction of this last weight,
    - this node cannot lead to a profit equal to *bound*,
    - but *bound* is still and upper bound of the profit we could achieve.

5.7 The 0-1 Knapsack Problem

- *Pruning* Strategy:
  - Suppose the *node* is at level i, and the *node* at level k is
    - the one that would bring the sum of weights above W.

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

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

  - If *maxprofit* is the value of the profit in the *best solution* found so far,
    - then a node at level i is *nonpromising* if $bound \leq maxprofit$.

5.7 The 0-1 Knapsack Problem

- An illustrative example:
  - $n = 4, W = 16$
  - $p_i = [40, 30, 50, 10]$
  - $w_i = [\ 2,\ \ 5, 10,\ \ 5]$
  - $\frac{p_i}{w_i} = [20,\ \ 6,\ \ 5,\ \ 2]$
  - Note that we have already ordered the items according to p_i/w_i.

5.7 The 0-1 Knapsack Problem

(0, 0) *(level, position)*

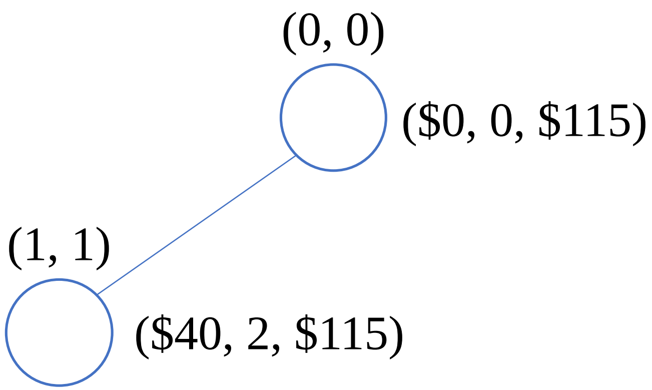
$maxprofit = \$0$ ($0, 0, $115)

(profit, weight, bound)

1. Set $maxprofit = \$0$.
2. Visit node (0, 0) (the root).
   - Compute its profit and weight.
     - $profit = \$0, weight = 0$
   - Compute its bound.
     - $totweight = 0 + 2 + 5 = 7, bound = \$0 + \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115$
   - Promising? yes
     - $weight(0) \; < \; W(16)$: *true*
     - $bound(\$115) \; > \; maxprofit(\$0)$: *true*

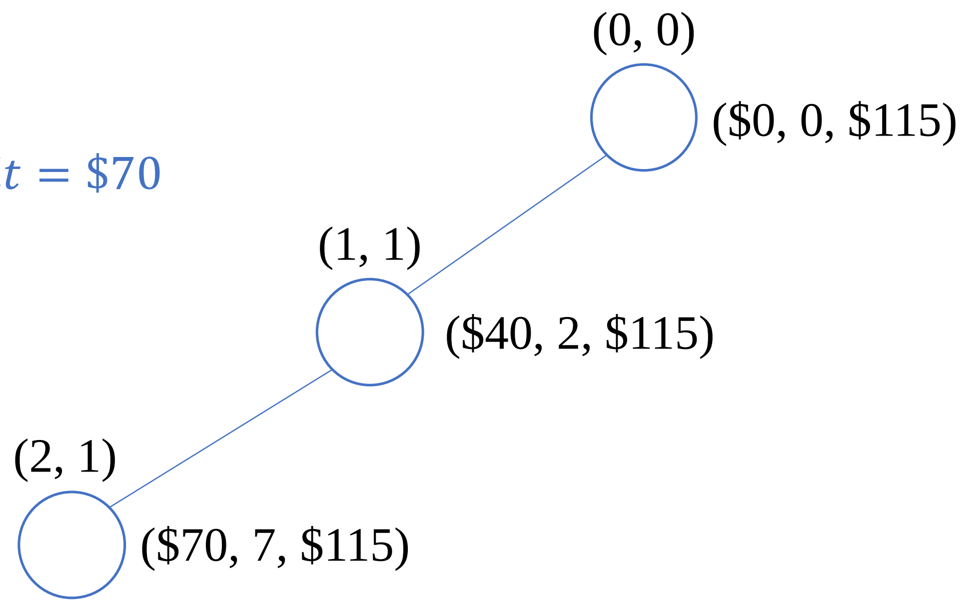
5.7 The 0-1 Knapsack Problem

$maxprofit = \$40$



3. Visit node (1, 1).
   - Compute its profit and weight.
     - $profit = \$0 + \$40 = \$40$, $weight = 0 + 2 = 2$
   - Set $maxprofit = \$40$.
     - $weight(2) \leq W(16)$ and $profit(\$40) > maxprofit(\$0)$
   - Compute its bound.
     - $totweight = 2 + 5 = 7$, $bound = \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115$
   - Promising? yes
     - $weight(2) < W(16)$: *true*
     - $bound(\$115) > maxprofit(\$40)$: *true*

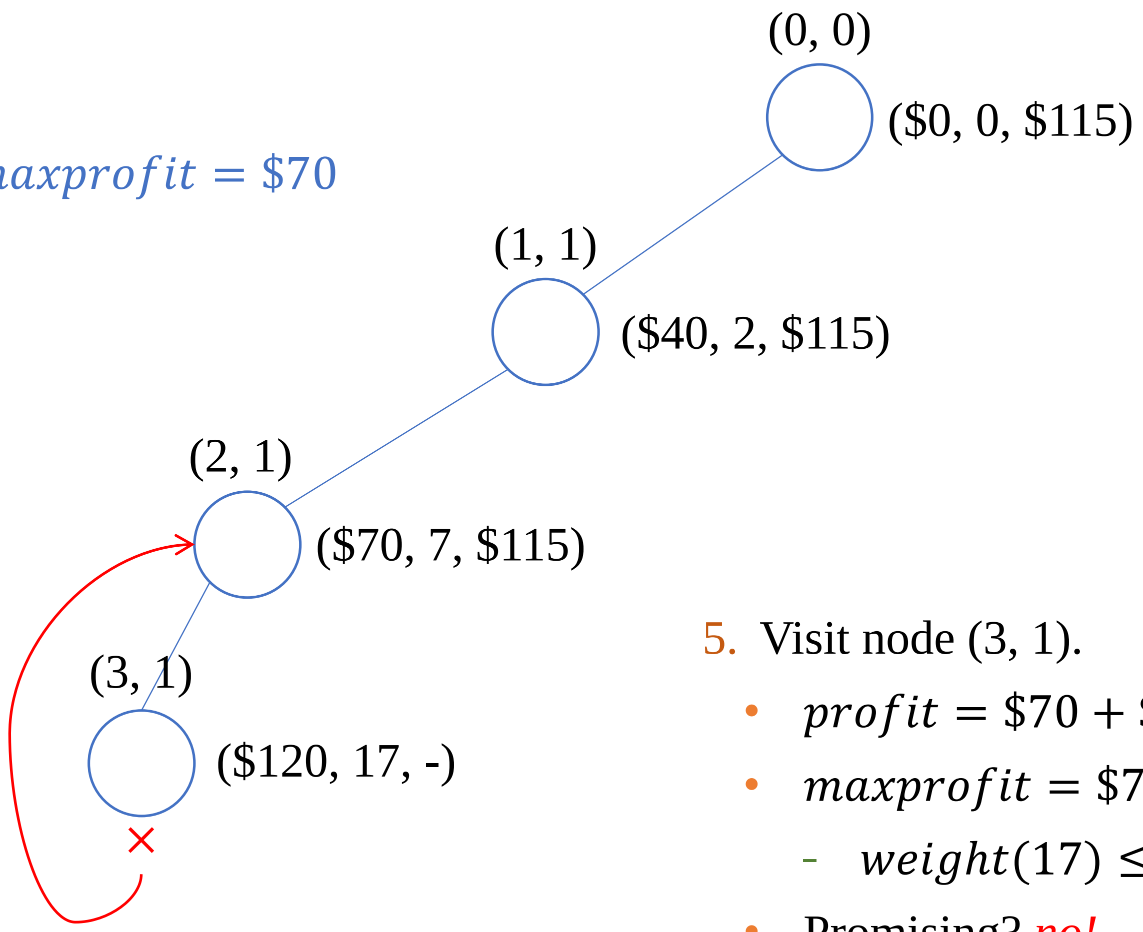
$maxprofit = \$70$



4. Visit node (2, 1).
   - $profit = \$40 + \$30 = \$70,\ weight = 2 + 5 = 7$
   - $maxprofit = \$70$
     - $weight(7) \leq W(16), profit(\$70) > maxprofit(\$40)$
   - $totweight = 7, bound = \$70 + (16 - 7) \times \frac{\$50}{10} = \$115$
   - Promising? yes!
     - $weight(7) < W(16),\ bound(\$115) > maxprofit(\$70)$

5.7 The 0-1 Knapsack Problem

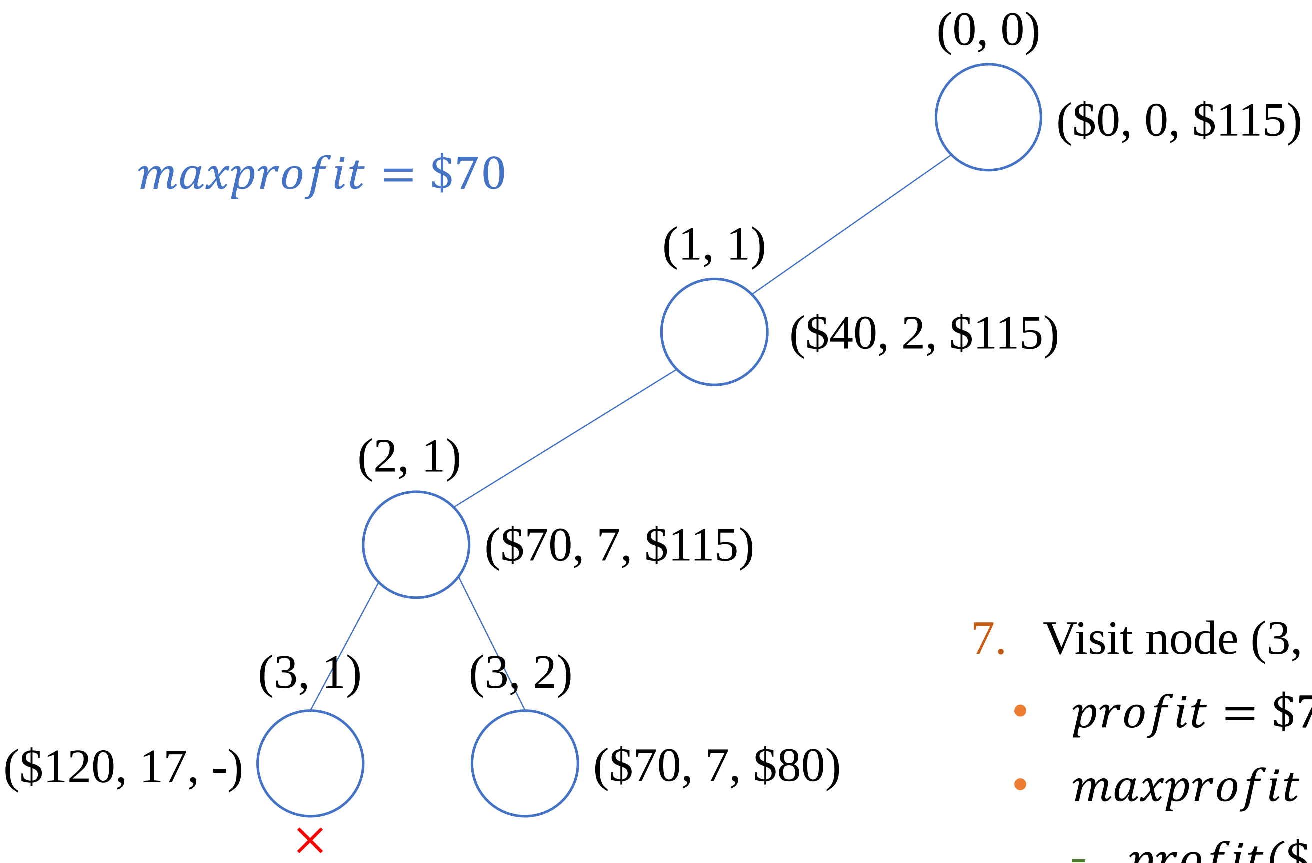
$maxprofit = \$70$



5. Visit node (3, 1).
   - $profit = \$70 + \$50 = \$120,\ weight = 7 + 10 = 17$
   - $maxprofit = \$70$ does not change.
     - $weight(17) \leq W(16)$: *false*
   - Promising? *no!*
   - The bound is *not computed*,
     - because this node is nonpromising.

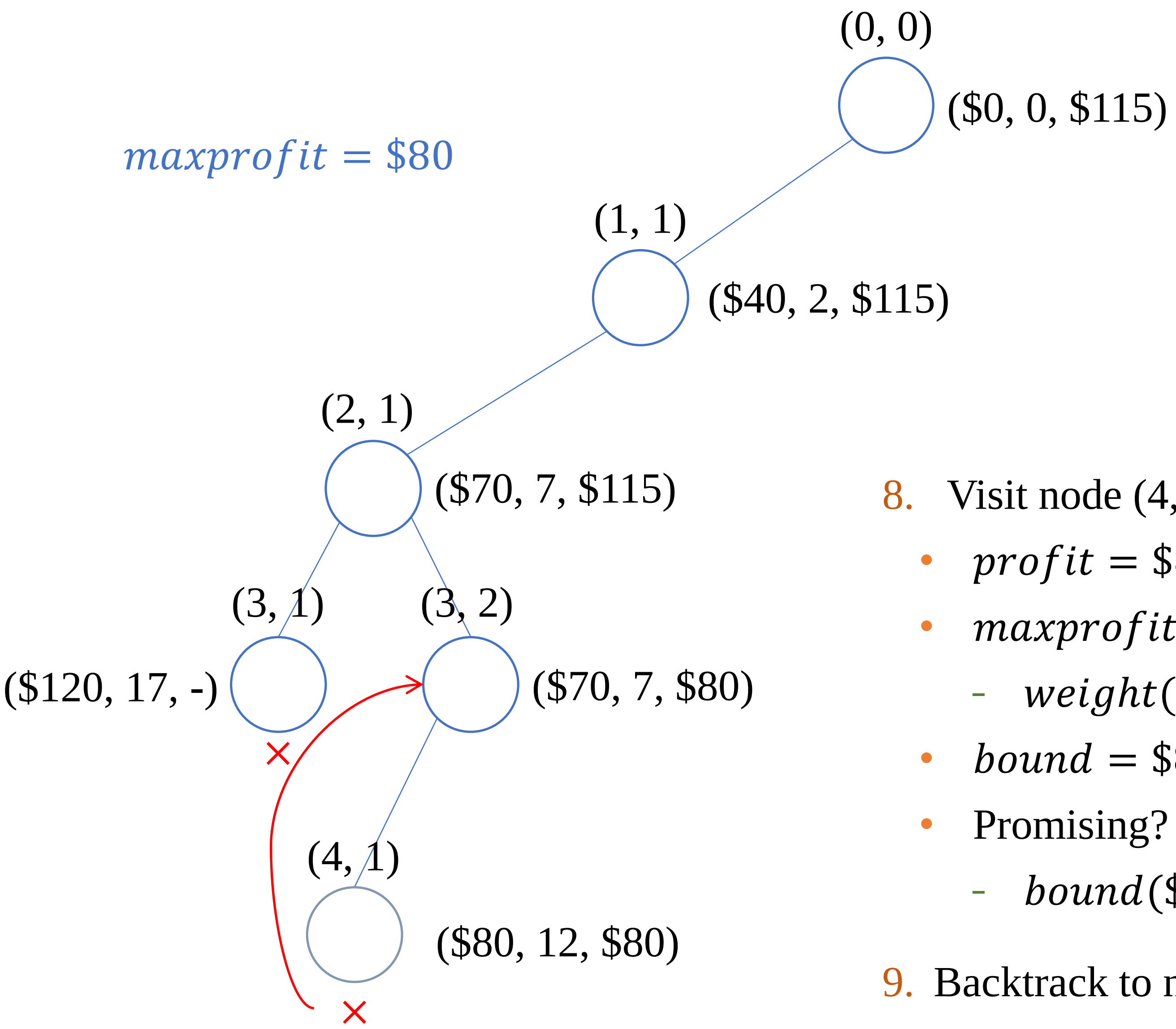
6. Backtrack to node (2, 1).

5.7 The 0-1 Knapsack Problem



7. Visit node (3, 2).
   - $profit = \$70,\ weight = 7$
   - $maxprofit = \$70$ does not change.
     - $profit(\$70) > maxprofit(\$70)$: *false*
   - $bound = \$70 + \$10 = \$80$
   - Promising? yes!

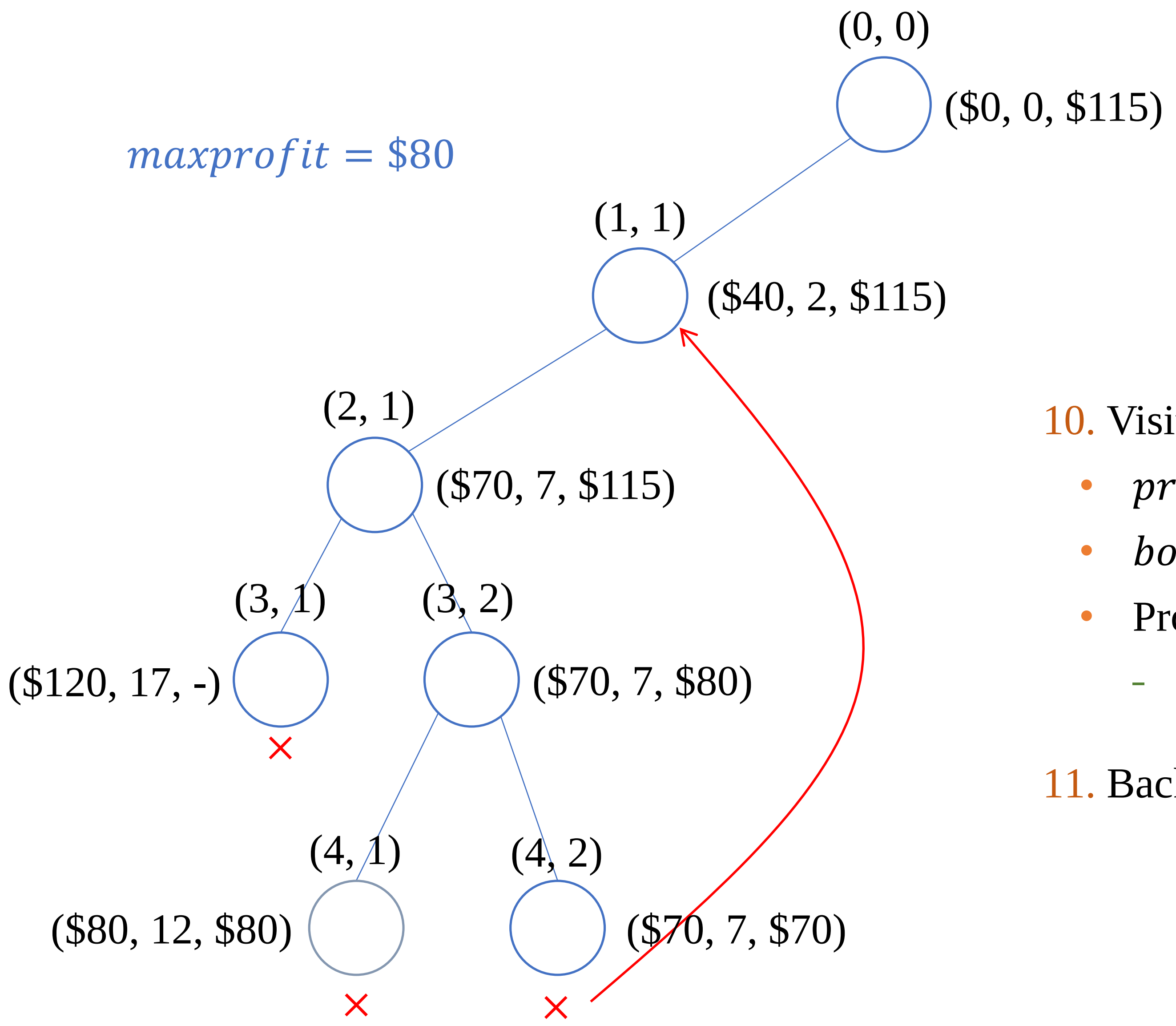
5.7 The 0-1 Knapsack Problem



8. Visit node (4, 1).
   - $profit = \$80,\ weight = 12$
   - $maxprofit = \$80$
     - $weight(12) \leq W(16), profit(\$80) > maxprofit(\$70)$
   - $bound = \$80$
   - Promising? *no!*
     - $bound(\$80) > maxprofit(\$80)$: *false*

9. Backtrack to node (3, 2).

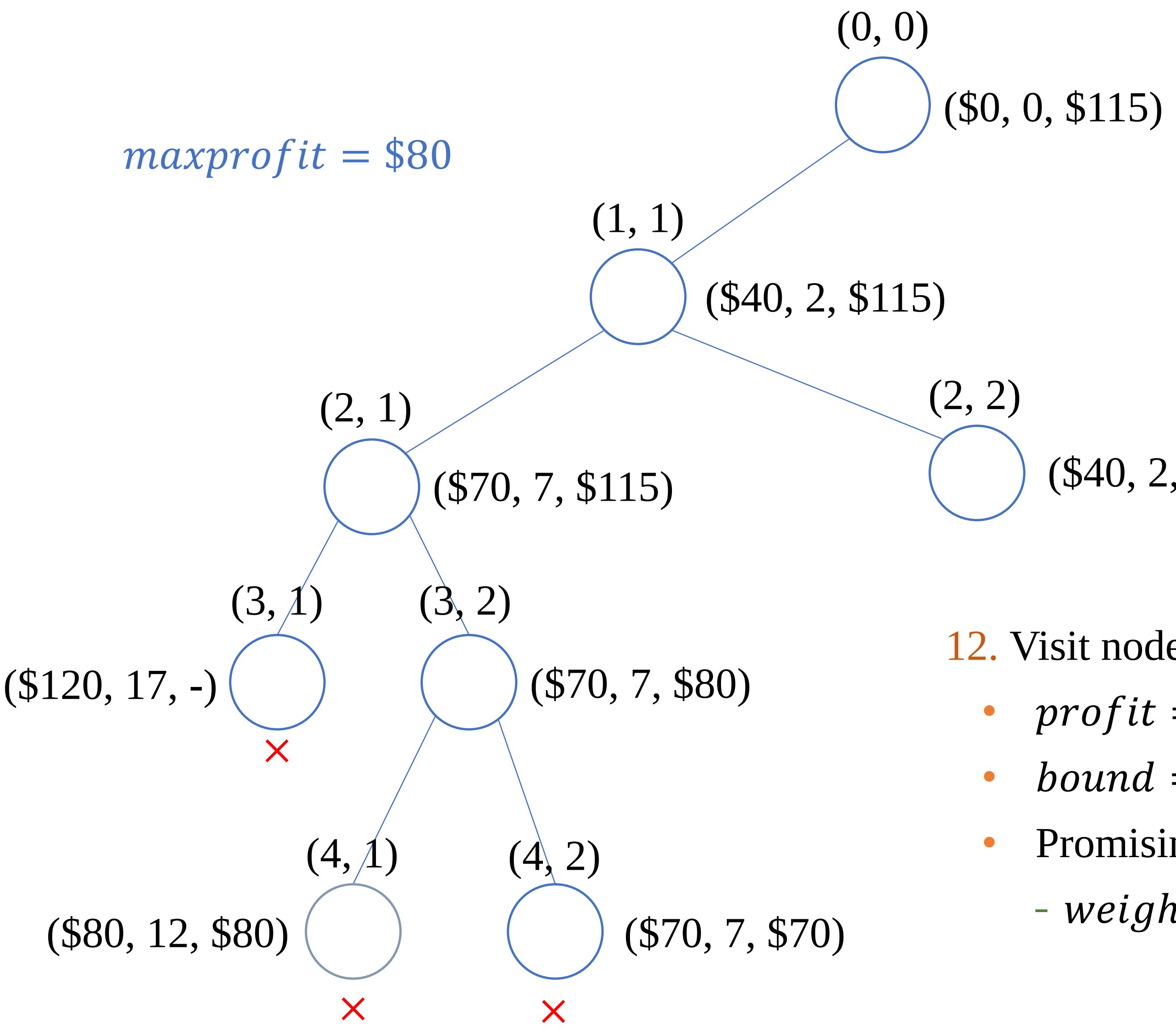
5.7 The 0-1 Knapsack Problem



10. Visit node (4, 2).
    - $profit = \$70,\ weight = 7$
    - $bound = \$70$
    - Promising? *no!*
        - $bound(\$70) > maxprofit(\$80)$: *false*

11. Backtrack to node (1, 1).

5.7 The 0-1 Knapsack Problem

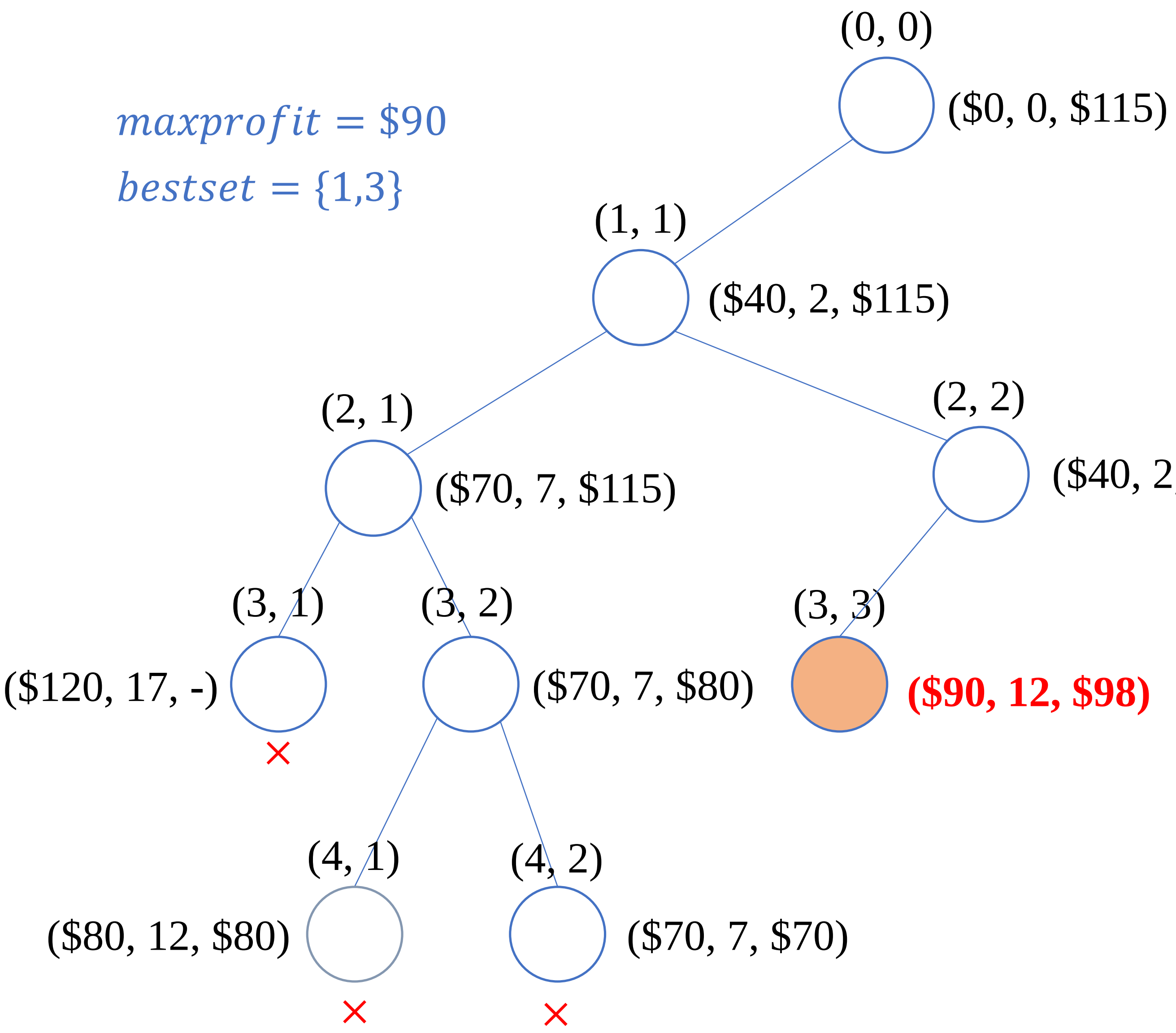


12. Visit node (2, 2).
    - $profit = \$40,\ weight = 2$
    - $bound = \$98$
    - Promising? yes!
      - $weight(2) < W(16), bound(\$98) > maxprofit(\$80)$

5.7 The 0-1 Knapsack Problem

$maxprofit = \$90$

$bestset = \{1,3\}$

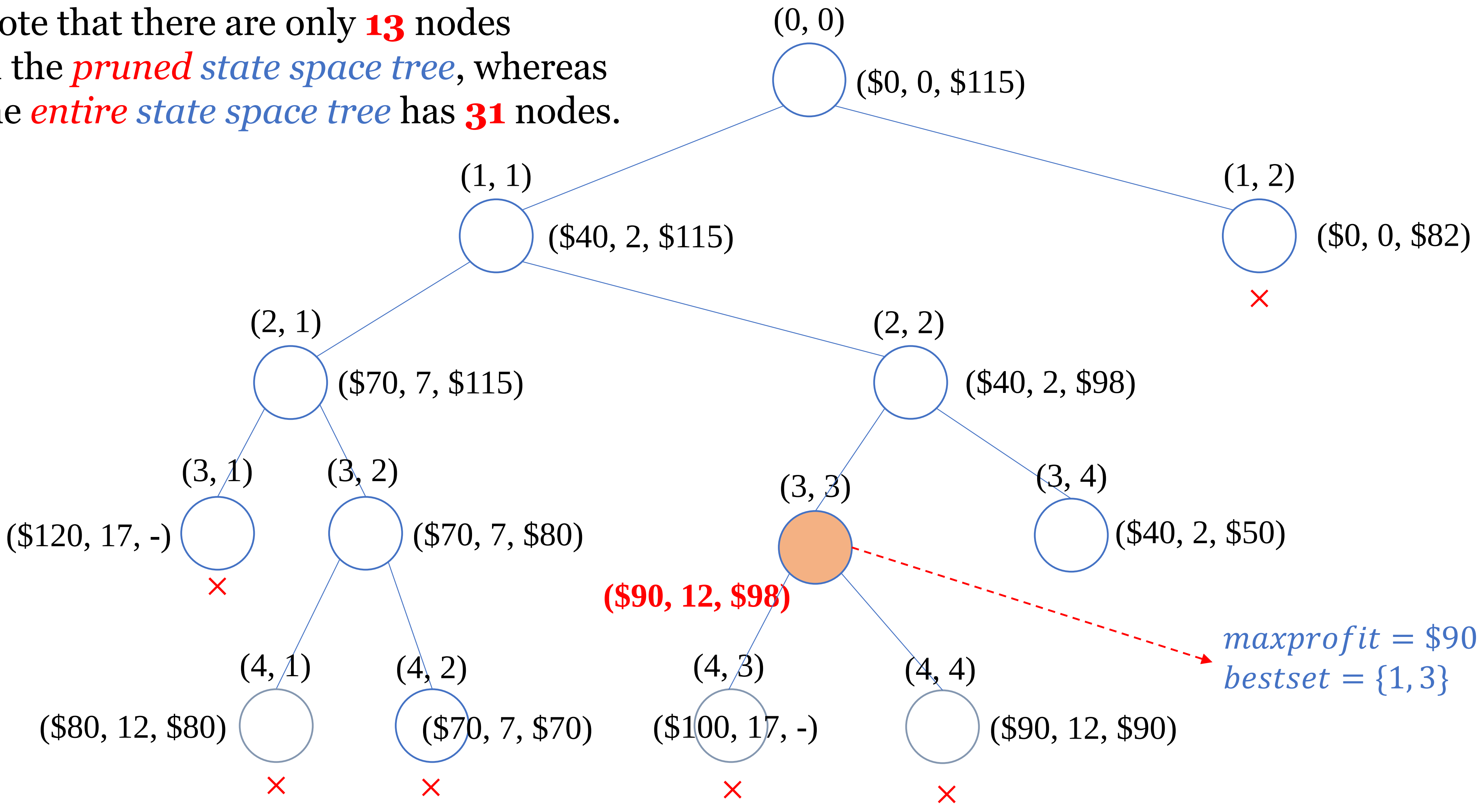


12. Visit node (3, 3).

- $profit = \$90,\ weight = 12$
- $maxprofit = \$90$
- $bound = \$98$
- Promising? yes!
  - $weight(12) < W(16)$,
  - $bound(\$98) > maxprofit(\$90)$

5.7 The 0-1 Knapsack Problem

Note that there are only **13** nodes in the *pruned state space tree*, whereas the *entire state space tree* has **31** nodes.



ALGORITHM 5.7: The Backtracking Algorithm for the 0-1 Knapsack Problem

```
void knapsack4(int i, int profit, int weight) {
    if (weight <= W && profit > maxprofit) {
        maxprofit = profit;
        array_copy(include, bestset); // copy from include to bestset.
    }

    if (promising(i, profit, weight)) {
        include[i + 1] = true;
        knapsack4(i + 1, profit + p[i + 1], weight + w[i + 1]);
        include[i + 1] = false;
        knapsack4(i + 1, profit, weight);
    }
}
```

ALGORITHM 5.7: The Backtracking Algorithm for the 0-1 Knapsack Problem (continued)

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

5.7 The 0-1 Knapsack Problem

- Algorithm 5.7 Explained:
  - As usual, n, W, w, p, $maxprofit$, $include$, $bestset$ are defined globally.
  - Then, the following code would produce the solution:

```
maxprofit = 0;
knapsack4(0, 0, 0);
cout << maxprofit << endl;
for (int i = 1; i <= n; i++)
    if (bestset[i]) cout << i << ":" << p[i] << " ";
```

  - The state space tree in the 0-1 Knapsack problem
    - is the same as that in the Sum-of-Subsets problem, $\Theta(2^n)$.
  - Comparing the Dynamic Programming with the Backtracking Algorithm.
    - Time Complexity: $O(minimum(2^n, nW))\ .vs.\ \Theta(2^n)$.
    - However, it is difficult to analyze theoretically.

Any Questions?