CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 8: Backtracking

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

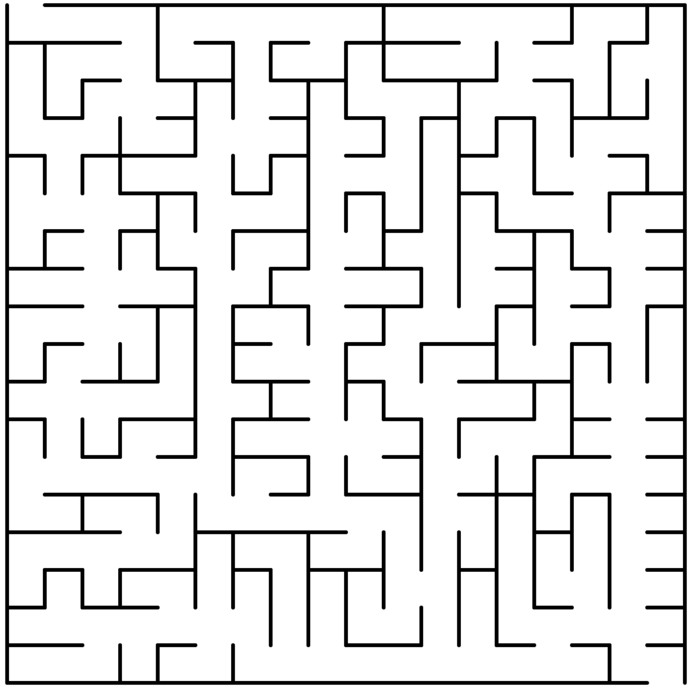
Office hour: 2pm-4pm Mon & Thur

Outlines

- n-Queens Problem
- The Sum-of-Subsets Problem
- Graph Coloring
- The Hamiltonian Circuits Problem
- The 0-1 Knapsack Problem

Backtracking

- A simple and straightforward strategy to escape from a maze is:
  - Go as deep as possible until reach a dead end.
  - Go back to the last fork and choose another path.
- If we have a sign at the fork to show dead ends, we can avoid that path.
  - This is backtracking.
- 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*.



A maze

Image source: https://upload.wikimedia.org/wikipedia/commons/thumb/2/28/Prim_Maze.svg/1200px-Prim_Maze.svg.png

Depth-First Search

- A *preorder* tree traversal is a *depth-first search (DFS)* of the tree.
  - 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.

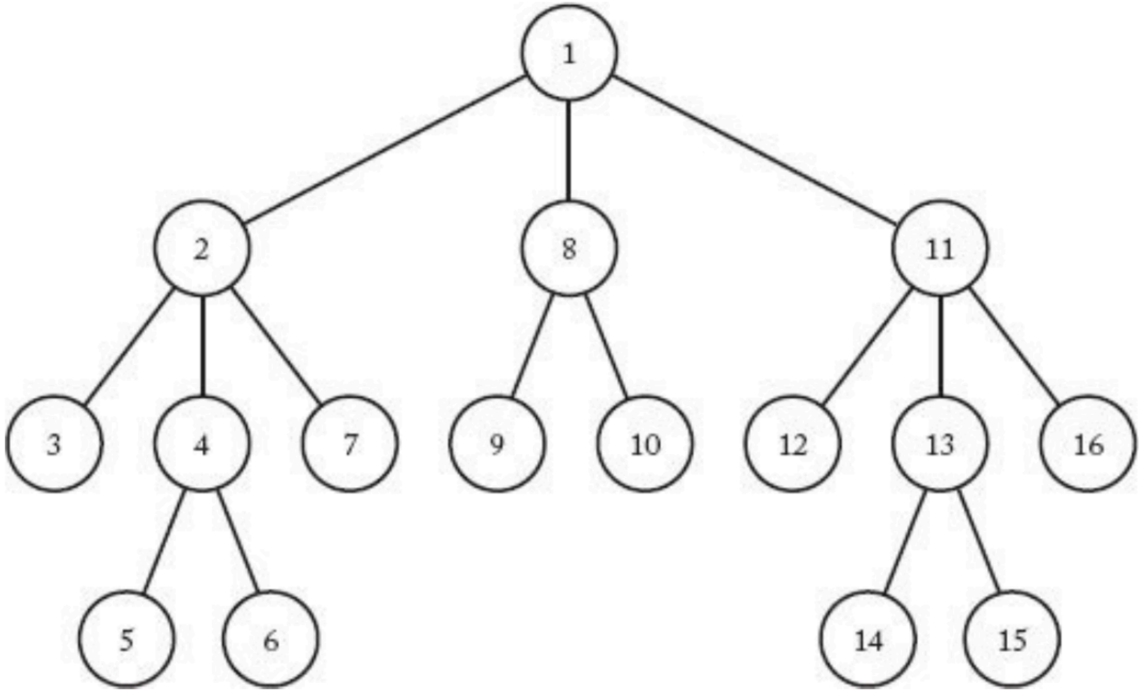


Image source: Figure 5.1, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

n-QUEENS PROBLEM

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.
  - 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.
- 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 $n = 4$.

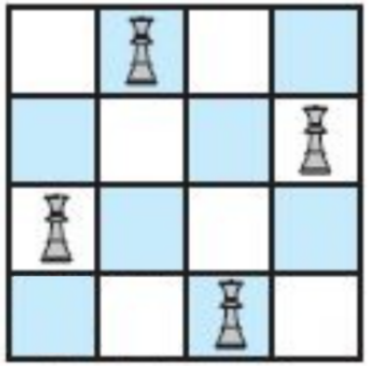


Image source: Figure 5.5, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

n-Queens Problem

- 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.
  - There are $4 \times 4 \times 4 \times 4 = 256$ candidate solutions.

n-Queens Problem

- We can create the candidate solutions by constructing a *state space tree*.
- A path from the root to a leaf is a candidate solution.
- Actually, we don't need to check every leaf.
    - We may early stop if we find out that this path definitely leads to a dead end.

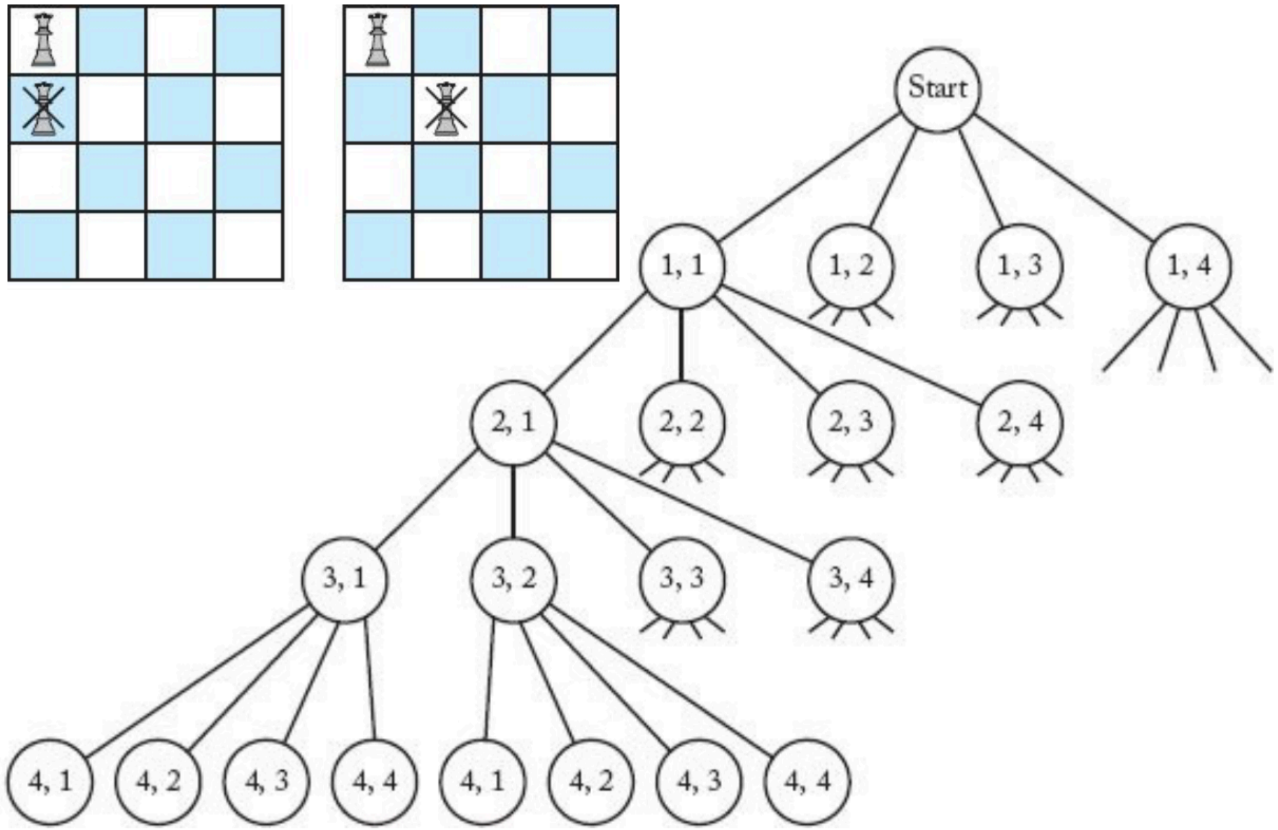


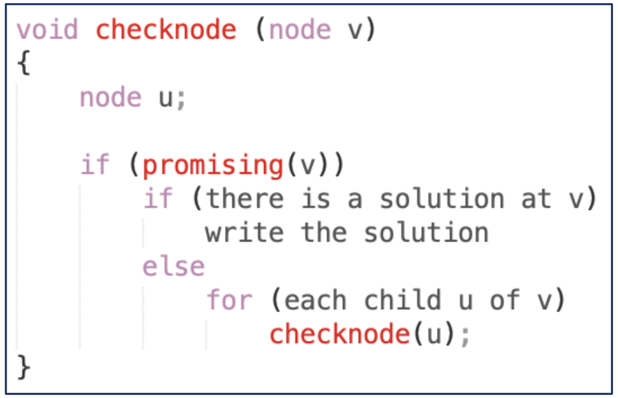
Image source: Figure 5.2-5.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Promising Function

- *Backtracking* is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("backtrack") 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*.
- The promising checking is done with DFS.
- This process called *pruning* the state space tree, and the subtree consisting of the visited nodes is called the *pruned state space tree*.

Promising Function

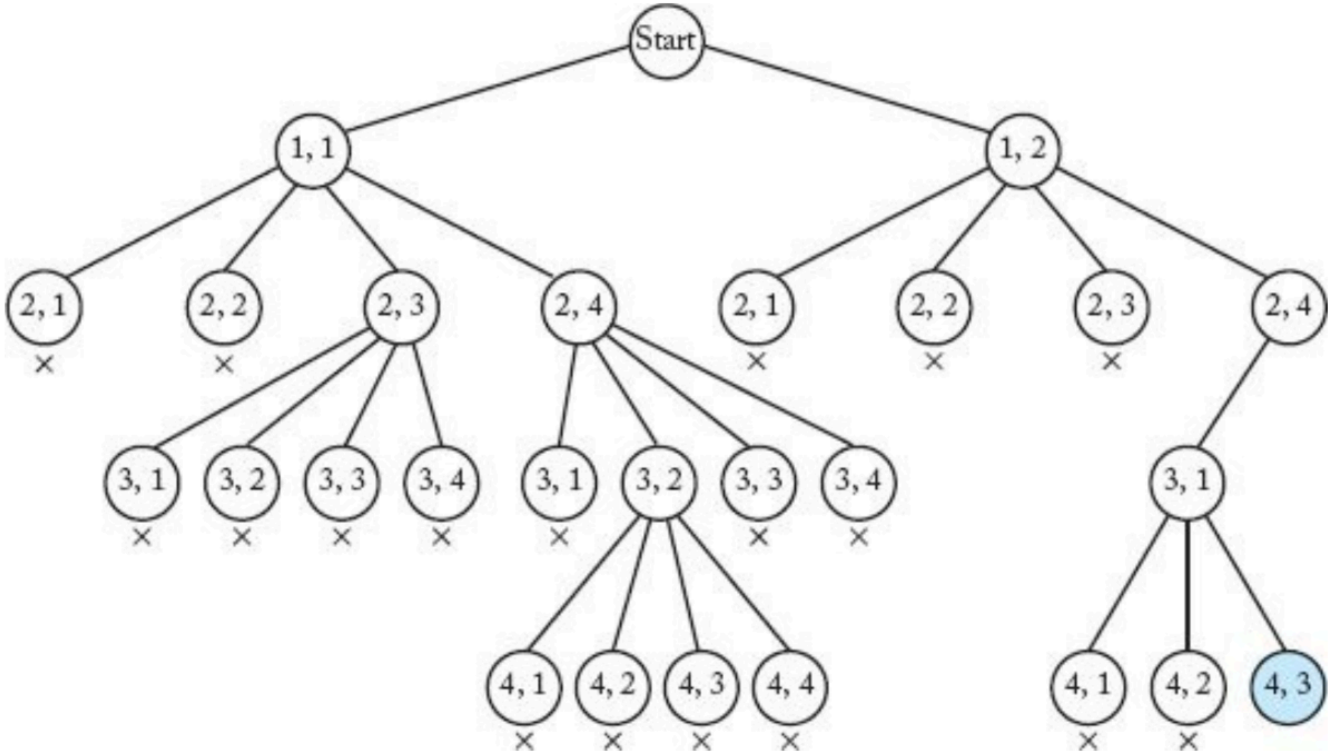
- The root of the state space tree is passed to `checknode` at the top level.
- A visit to a node consists of first checking whether it is promising.
    - If it is promising and there is a solution at the node, the solution is printed.
    - If there is not a solution at a promising node, the children of the node are visited.
- We call it the *promising function* for the algorithm, which is different in each application of backtracking.
- A backtracking algorithm is same as DFS, except that the children of a node are visited only when the node is promising and there is not a solution at the node.

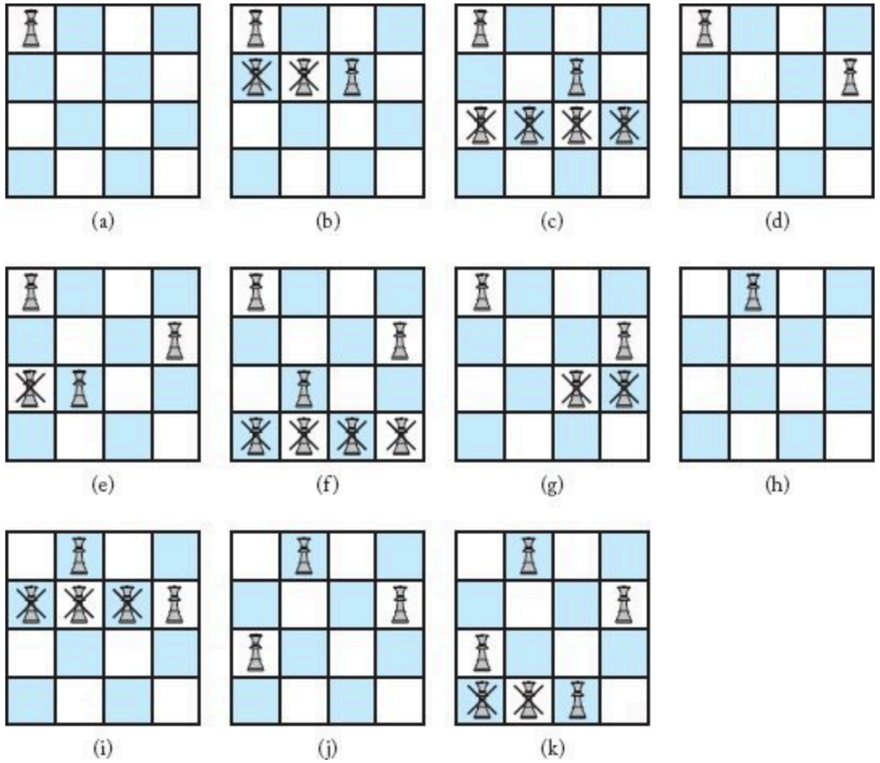


```
void checknode (node v)
{
    node u;

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

Backtracking of n-Queens Problem





The backtracking algorithm only checks 27 nodes, while DFS checks 155 nodes before finding that same solution.

Image source: Figure 5.4-5.5, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Backtracking

- Notice that a backtracking algorithm **does not need to actually create a tree**.
    - Usually, they are implemented by recursion (thus a stack).
- Rather, it only needs to keep track of the values in the current branch being investigated.
- The state space tree exists **implicitly** in the algorithm because it is not actually constructed.

n-Queens Problem

- For each row, we put one queen. Thus, the promising function only needs to check if two queens are in the same column or diagonal.
- Let $col(i)$ be the column where the queen in the ith row is located.
- Condition that two queens are in the same column:

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

- Condition that two queens are in the same diagonal :

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

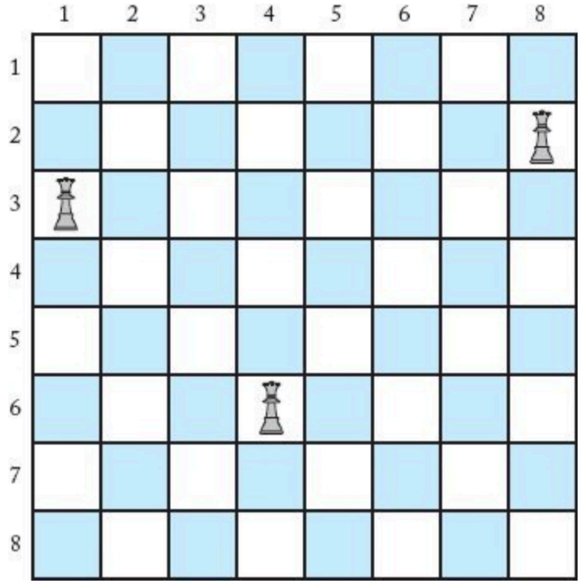
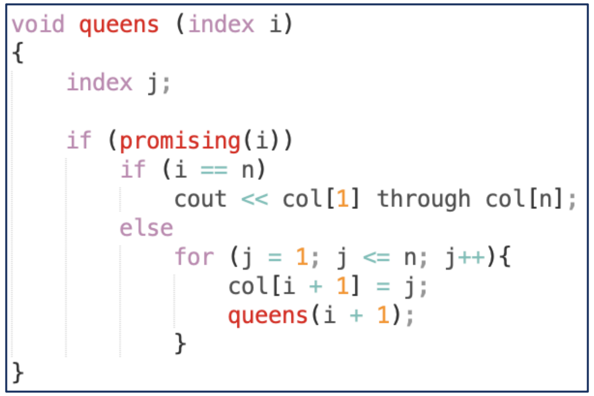


Image source: Figure 5.6, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

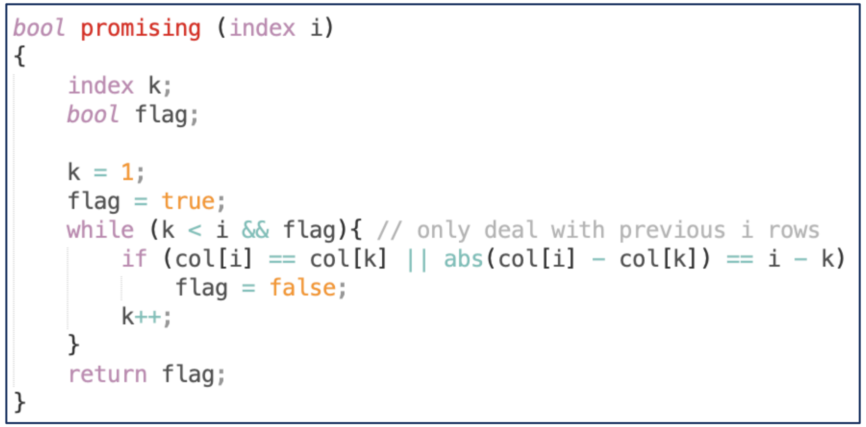
Pseudocode of n-Queens Problem

- As usual, non-changing variables `n` and `col` are not inputs to the recursive function. They are defined globally.
- The top level call is `queens(0)`.
- For the terminate condition `i == n`, the program doesn't stop, until all solutions are found.



```
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++){
                col[i + 1] = j;
                queens(i + 1);
            }
}
```



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

    k = 1;
    flag = true;
    while (k < i && flag){ // only deal with previous i rows
        if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
            flag = false;
        k++;
    }
    return flag;
}
```

Analysis of n-Queens Problem

- For DFS, the tree contains 1 node at level 0, n nodes at level 1, n^2 nodes at level 2, … , and n^n nodes at level n. The total number of nodes is

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

- For backtracking, if we only check the column, the upper bound of promising nodes are

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

- For $n = 8$, DFS has 19,173,961 nodes while backtracking only has at most 109,601 promising nodes.
- Thus, the purpose of backtracking is to use promising function to improve DFS as much as possible.
    - Save time by stop earlier.

THE SUM-OF-SUBSETS PROBLEM

The Sum-of-Subsets Problem

- In the Sum-of-Subsets problem, there are n positive integers (weights) w_i and a positive integer W.
  - Similar to 0-1 Knapsack problem but without value.
- The goal is to find all subsets of the integers that sum to W.
- Example:
  - Suppose that $n = 5, W = 21$, and

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

  - The solutions is $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$ and $\{w_3, w_4\}$ because

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

The Sum-of-Subsets Problem

- One approach is to create a state space tree.
- Each subset is represented by a path from the root to a leaf.
  - We go to the left from the root to include w_1, and we go to the right to exclude w_1.
  - We go to the left from a node at level 1 to include w_2, and we go to the right to exclude w_2.
  - …
- When we include w_i, we write w_i on the edge where we include it. When we do not include w_i, we write 0.

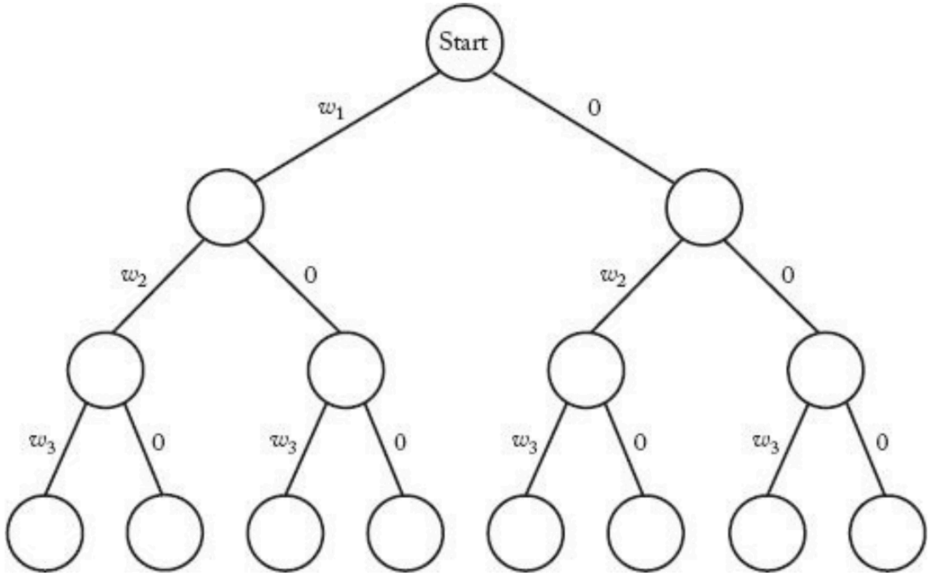


Image source: Figure 5.7, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Promising Function

- If we sort the weights in nondecreasing order before doing the search, there is an obvious sign telling us that a node is nonpromising.
- Let $weight$ to be the sum of the weights that have been included up, and $remain$ is the sum of the weight that is remained to be checked.
- There are two cases that a node at the ith level is nonpromising:
  - Case 1: Including w_{i+1} exceeds W:

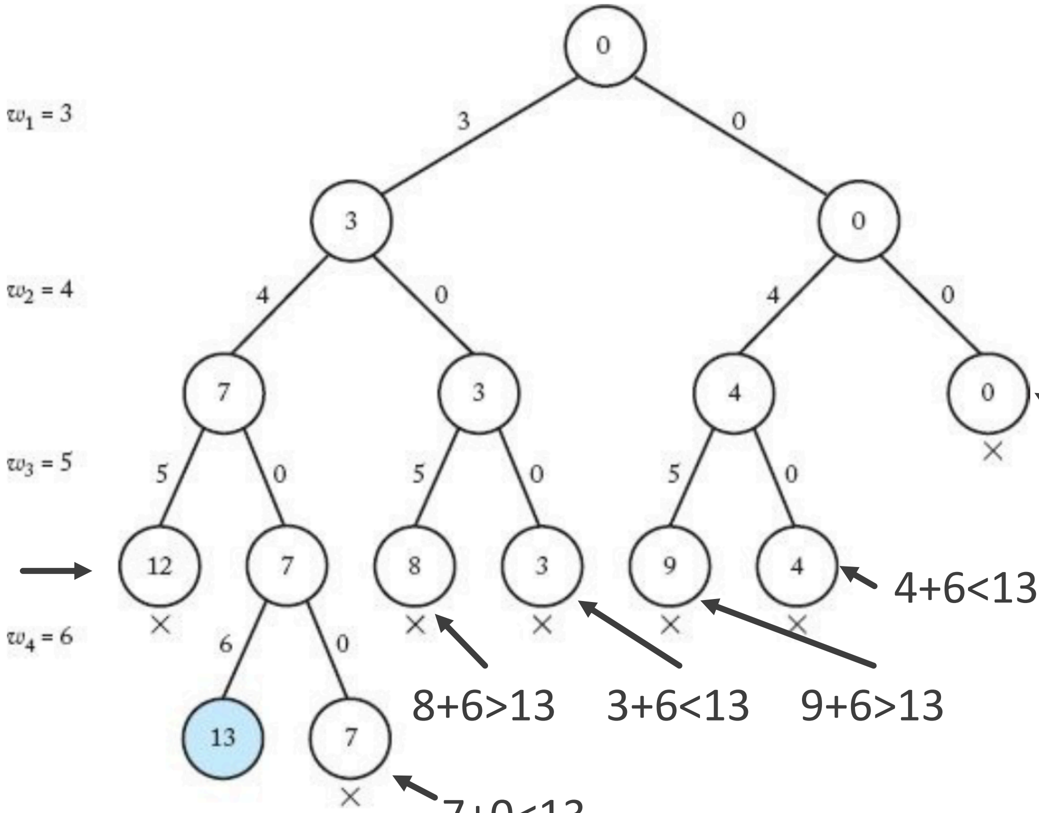
$$weight + w_{i+1} > W.$$

  - Case 2: Including all the remaning can’t reach W:

$$weight + remain < W.$$

Example

$n = 4, W = 13$



0+11<13

12+6>13

4+6<13

8+6>13

3+6<13

9+6>13

7+0<13

Image source: Figure 5.9, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode

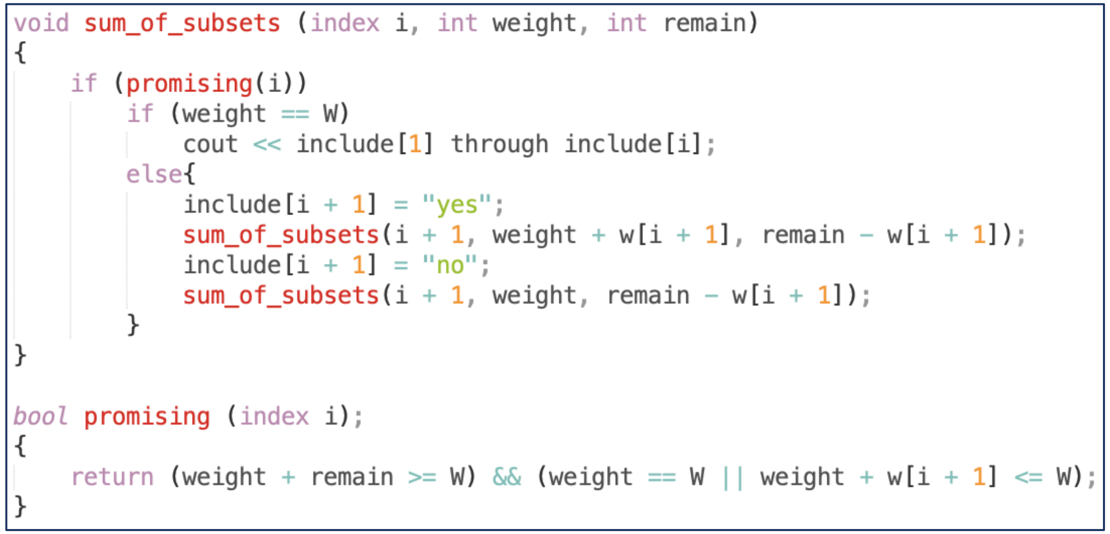
- `n`, `w`, `W` and `include` are defined globally.
- The top-level call is

  `sum_of_subsets(0, 0, remain)`

  where `remain` is initialized as:

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

- Actually, we don't need to test if `i==n`, because it has been tested by `weight+remain>=W` in function `promising`.
  - When `i==n`, `remain` must be 0.



```
void sum_of_subsets (index i, int weight, int remain)
{
    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], remain - w[i + 1]);
            include[i + 1] = "no";
            sum_of_subsets(i + 1, weight, remain - w[i + 1]);
        }
}

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

GRAPH COLORING

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 for this example graph.
- One solution to the 3-Coloring problem for this graph is as follows:

v_1	color 1
v_2	color 2
v_3	color 3
v_4	color 2

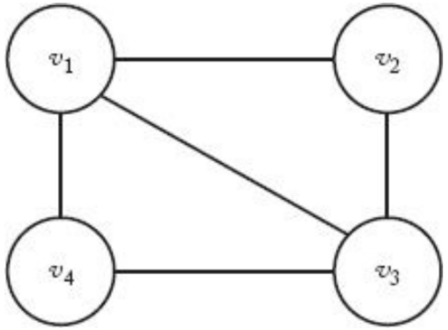


Image source: Figure 5.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Graph Coloring

- An important application of graph coloring is the coloring of maps.
- In mathematics, a very famous problem is called the **four color theorem**.
    - It has been proved with a computer software in 1976.
- Given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color.

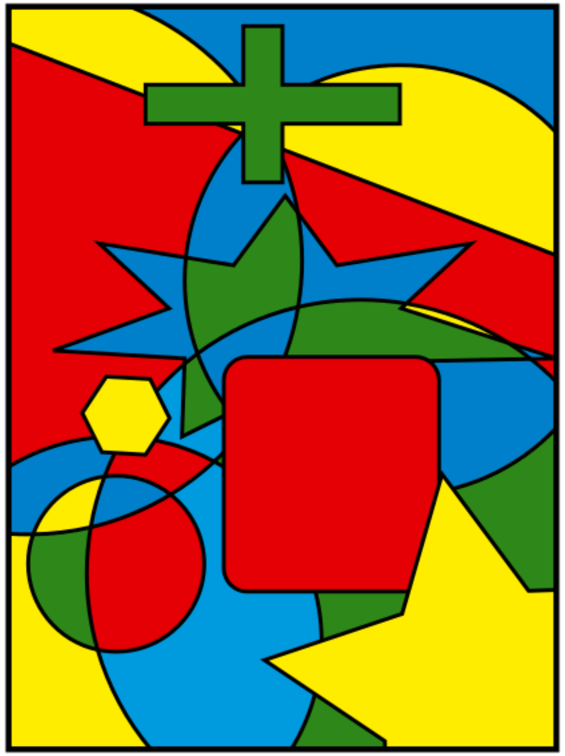
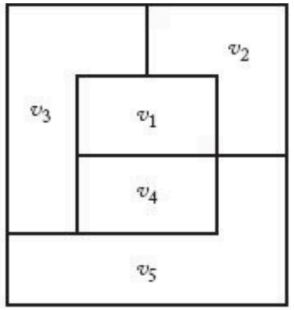
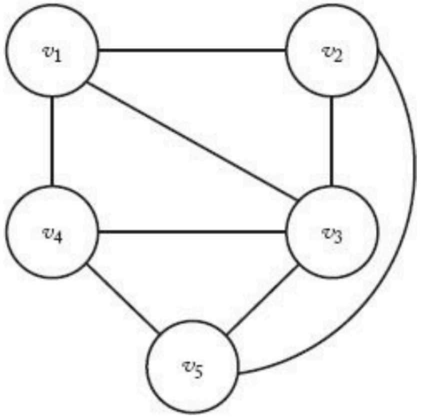


Image source: https://en.wikipedia.org/wiki/Four_color_theorem

Graph Coloring

- A graph is called *planar* if it can be drawn in a plane in such a way that no two edges cross each other.
    - However, if we were to add the edges (v_1, v_5) and (v_2, v_4) it would no longer be planar.
- To every map there corresponds a planar graph.
- The m-Coloring problem for planar graphs 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.

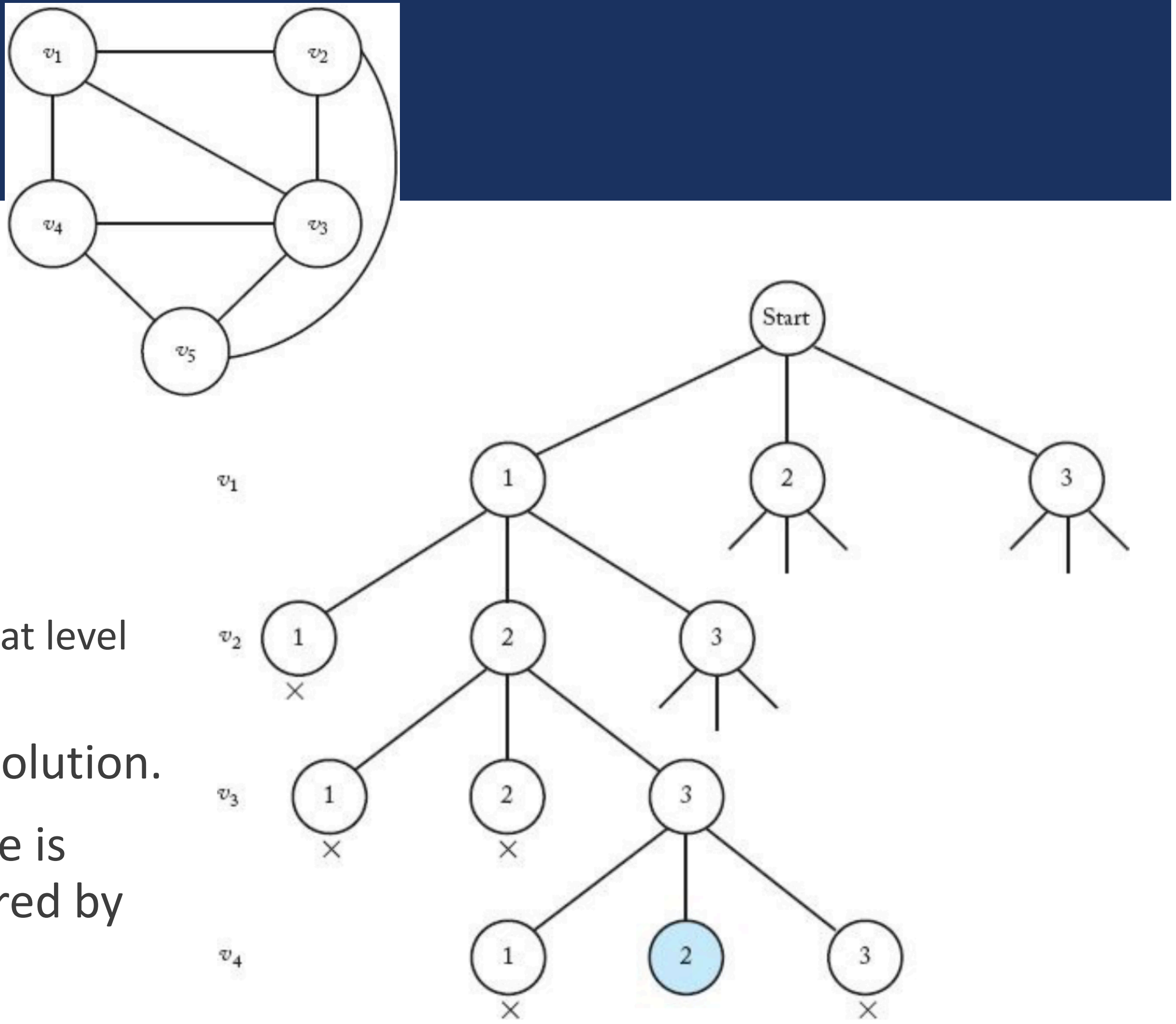




A planar graph

Image source: Figure 5.11, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Graph Coloring



- A straightforward state space tree is:
  - Each possible color is tried for vertex v_1 at level 1;
  - Each possible color is tried for vertex v_2 at level 2;
  - …
  - Until each possible color has been tried for vertex v_n at level n.
- Each path from the root to a leaf is a candidate solution.
- We can backtrack in this problem because a node is nonpromising if a two adjacent vertices are colored by the same color.

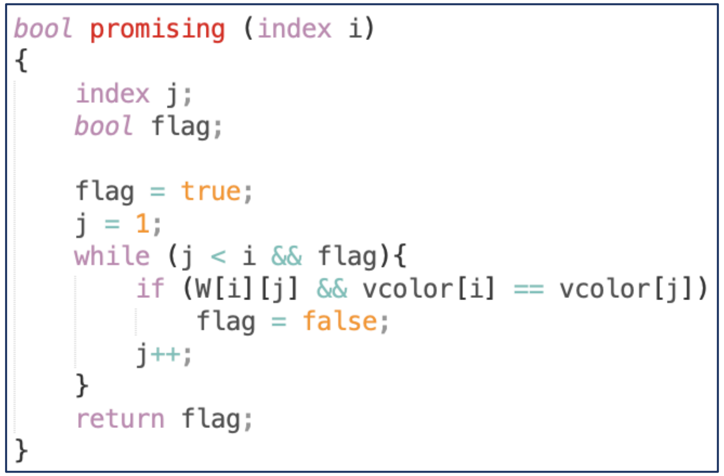
Image source: Figure 5.12, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of Graph Coloring

- The top level call is `m_coloring(0).`
- The pseudocode is exactly same as the n-Queens problem, except the if-condition in promising function.

```
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);
            }
}
```



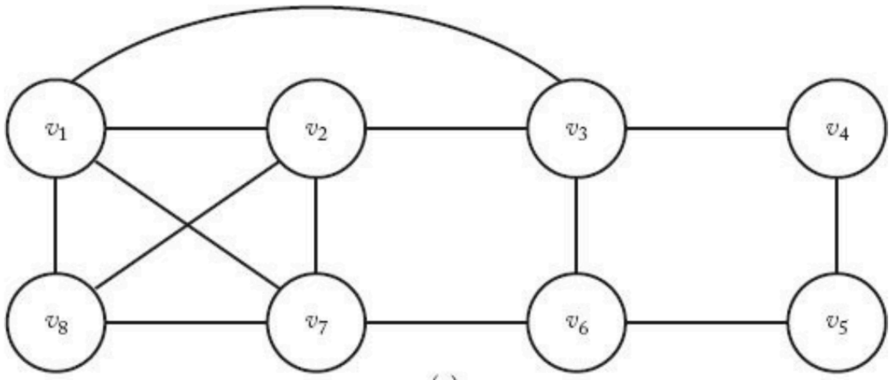
```
bool promising (index i)
{
    index j;
    bool flag;

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

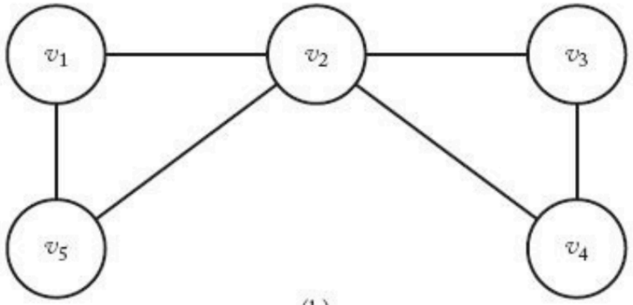
THE HAMILTONIAN CIRCUITS PROBLEM

The Hamiltonian Circuits Problem

- Given a connected, undirected graph, a *Hamiltonian Circuit* (also called a tour) is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex.
- The graph in Figure (a) contains the Hamiltonian Circuit $[v_1, v_2, v_8, v_7, v_6, v_5, v_4, v_3, v_1]$, but the one in Figure (b) does not contain a Hamiltonian Circuit.



(a)



(b)

Image source: Figure 5.13, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

The Hamiltonian Circuits Problem

- A state space tree for this problem is as follows.
  - 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.
  - 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.
- Consider backtrack in this 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.

The Hamiltonian Circuits Problem

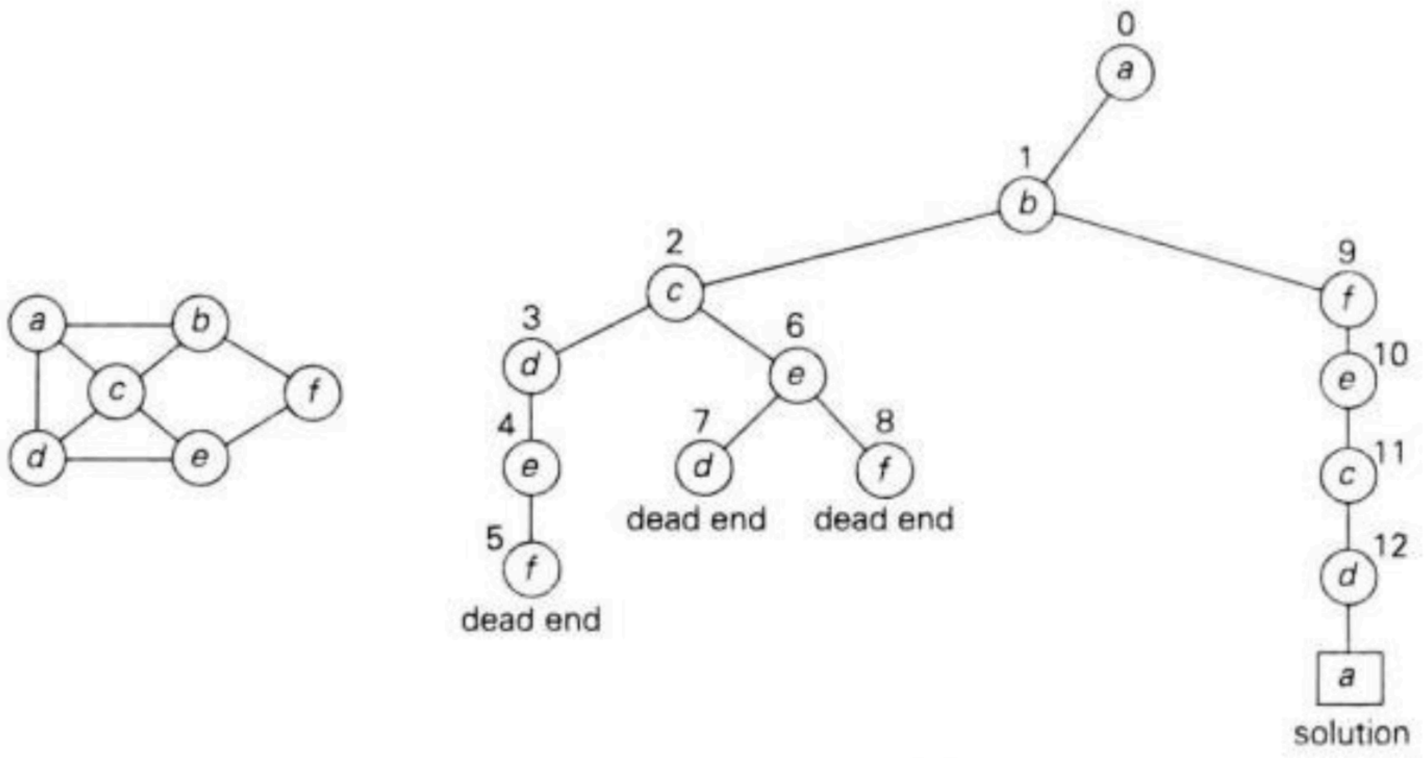
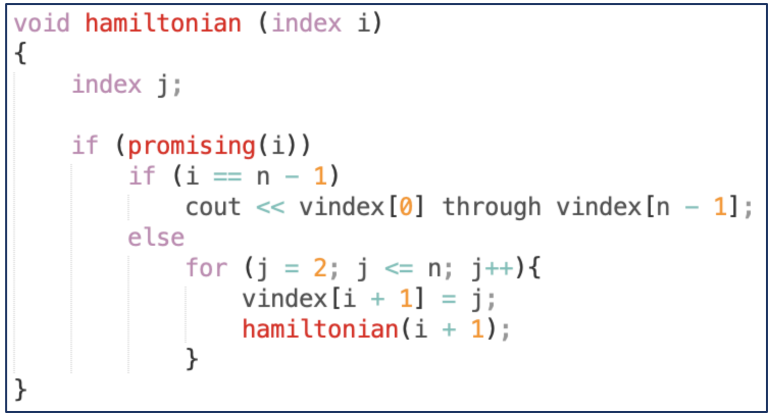


Image source: http://www.brainkart.com/article/Hamiltonian-Circuit-Problem_7981/

Pseudocode of the Hamiltonian Circuits Problem

- The top-level call is:

```
vindex[0]=1;
hamiltonian(0);
```



```
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 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;
}
```

THE 0-1 KNAPSACK PROBLEM

Knapsack Problem Recall

- Problem description:
  - Given n items and a "knapsack.”
  - Item i has weight $w_i > 0$ and has value $v_i > 0$.
  - Knapsack has capacity of W.
  - Goal: Fill knapsack so as to maximize total value.
- Mathematical description:
  - Given two n-tuples of positive numbers $< v_1, v_2, \dots, v_n >$ and $< w_1, w_2, \dots, w_n >$, and $W > 0$, we wish to determine the subset $T \subseteq \{1,2, \dots, n\}$ that

$$\text{maximize} \sum_{i \in T} v_i \qquad \text{subject to} \sum_{i \in T} w_i \leq W$$

- Can backtracking solve this problem?

The 0-1 Knapsack Problem

- We can solve this problem using a state space tree exactly like the one in the Sum-of-Subsets problem.
  - We go to the left from the root to include the first item, and we go to the right to exclude it.
  - We go to the left from a node at level 1 to include the second item, and we go to the right to exclude it.
  - …
  - Each path from the root to a leaf is a candidate solution.

The 0-1 Knapsack Problem

- This problem is different from the others discussed in this chapter in that it is *an optimization problem*.
  - It finds the maximum value, rather than a solution satisfying some conditions.
- We do not know if a node contains a solution until the search is over.
- If the items included up to a node have a greater total profit than the best solution so far, we change the value of the best solution so far.
  - However, we may still find a better solution at one of the node's descendants (by including more items).
  - Therefore, for optimization problems we always visit a promising node's children.

Promising Function

- Similar to the sum-of-subsets problem, there are two cases that a node is nonpromising:
  - Case 1: Weights of included items exceeds W: $weight \geq W$.
    - $weight = W$ is also nonpromising because it may not be a solution and it cannot expand to its children.
  - Case 2: Even including all the remaining possible items can’t exceed the existing best profit.

Promising Function

- For the second case, we should calculate the profit bound of including all remaining possible items.
  - We use the idea of fractional knapsack with greedy approach, because it can bring us the upper bound.
  - We first sort the items in nonincreasing order according to the values of v_i/w_i.
  - The profit bound is calculated by fill the knapsack with fractional items in this order.
- For example, $n = 4, W = 16$:
  - If we don't include any item yet, the profit bound is

$$40 + 30 + (16 - 2 - 5) \times 5 = 115.$$

  - If now we include item 1 and don't include item 2, the profit bound is

$$40 + 50 + (16 - 2 - 10) \times 2 = 98.$$

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

Promising Function

- Suppose the node is at level i, we first calculate k such that the level k is the one that would bring the sum of the weights *exceeds* W.
- Then we have:

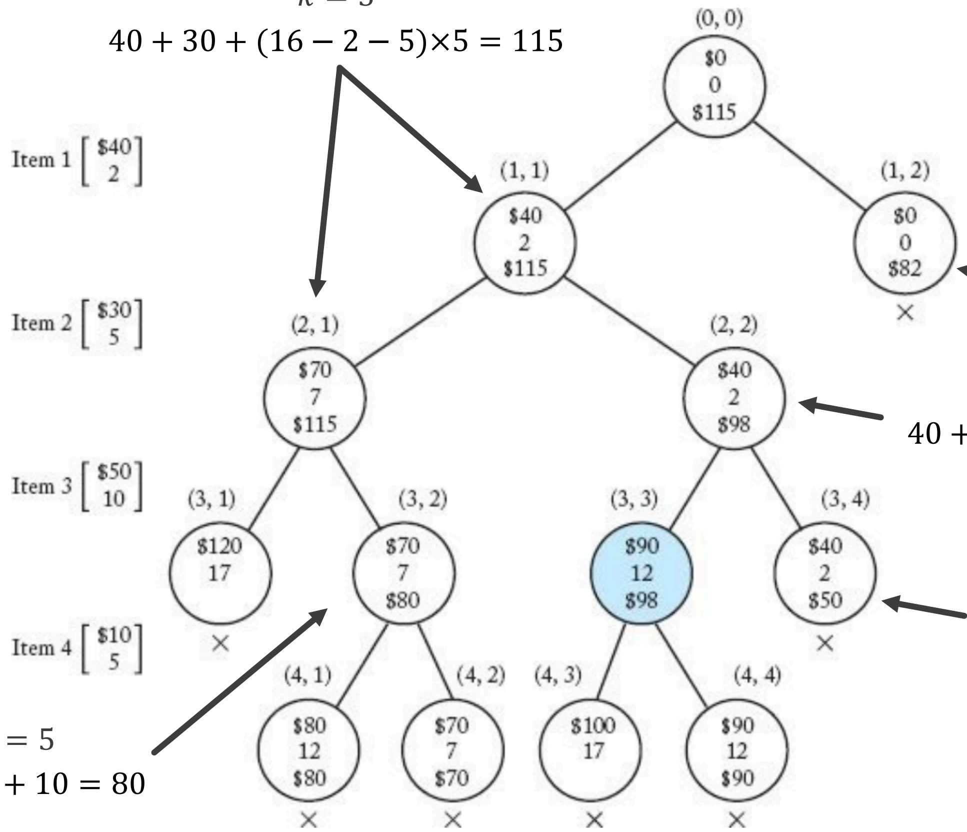
$$totweight = weight + \sum_{j=i+1}^{k-1} w_j \,,$$

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

Promising Function

$W = 16$

i	v_i	w_i	v_i/w_i
1	$40	2kg	20$/kg
2	$30	5kg	6$/kg
3	$50	10kg	5$/kg
4	$10	5kg	2$/kg



$k = 3$

$40 + 30 + (16 - 2 - 5) \times 5 = 115$

$k = 4$

$30 + 50 + (16 - 5 - 10) \times 2 = 82$

$k = 4$

$40 + 50 + (16 - 2 - 10) \times 2 = 98$

$k = 5$

$40 + 10 = 50$

$k = 5$

$40 + 30 + 10 = 80$

XIAMEN UNIVERSITY MALAYSIA 厦門大學馬來西亞分校

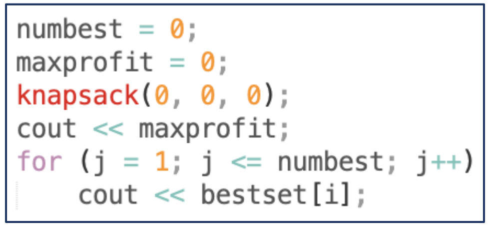
厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系 Computer Science Department of Xiamen University

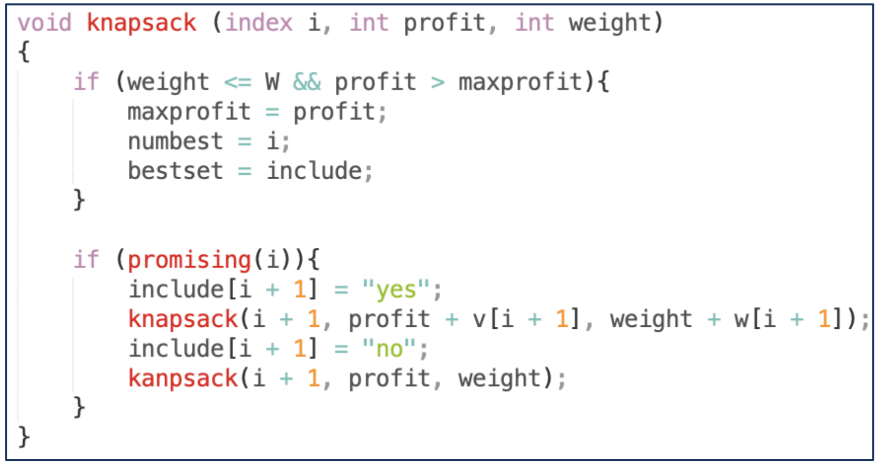
Image source: Figure 5.14, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of the 0-1 Knapsack Problem

Top level call

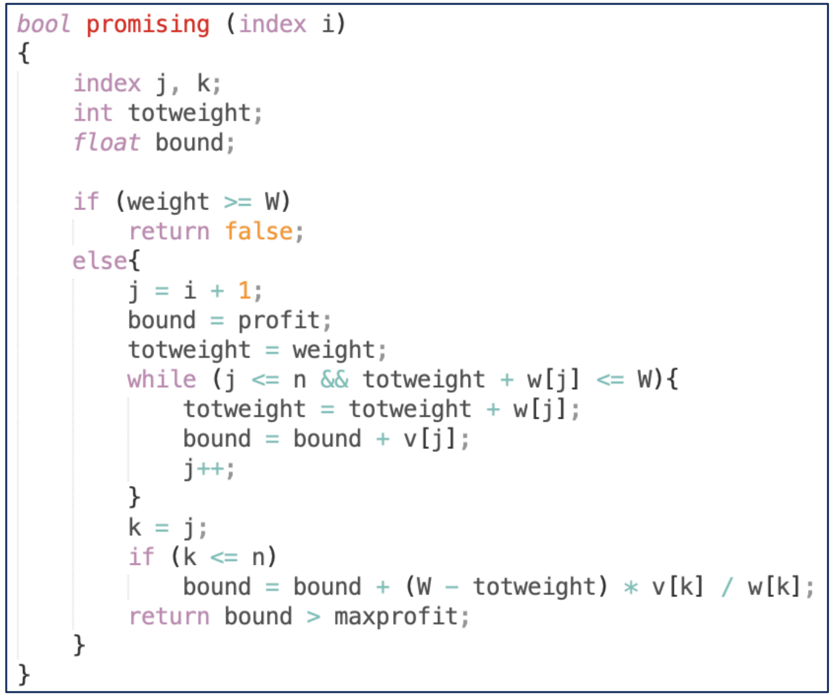


```
numbest = 0;
maxprofit = 0;
knapsack(0, 0, 0);
cout << maxprofit;
for (j = 1; j <= numbest; j++)
    cout << bestset[i];
```



```
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 + v[i + 1], weight + w[i + 1]);
        include[i + 1] = "no";
        kanpsack(i + 1, profit, weight);
    }
}
```



```
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 + v[j];
            j++;
        }
        k = j;
        if (k <= n)
            bound = bound + (W − totweight) ∗ v[k] / w[k];
        return bound > maxprofit;
    }
}
```

Conclusion

General process of developing a backtracking algorithm:

- Construct a state space tree.
- Design a promising function to stop at some nonpromising nodes and thus avoid full DFS over this state space tree.

Conclusion

After this lecture, you should know:

- What is the difference between DFS and backtracking.
- What is a state space tree.
- What is a promising function.
- What kind of problems can be solved by backtracking.

Thank you!

- Any question?
- Don’t hesitate to send email to me for asking questions and discussion. ☺