

# *Analysis and Design of Algorithms*

## Chapter 6: Decrease and Conquer



*School of Software Engineering © Yanling Xu*



# Decrease and Conquer

## ■ Three variations of Decrease and Conquer tech.

*exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance*

### ✦ Decrease by a constant

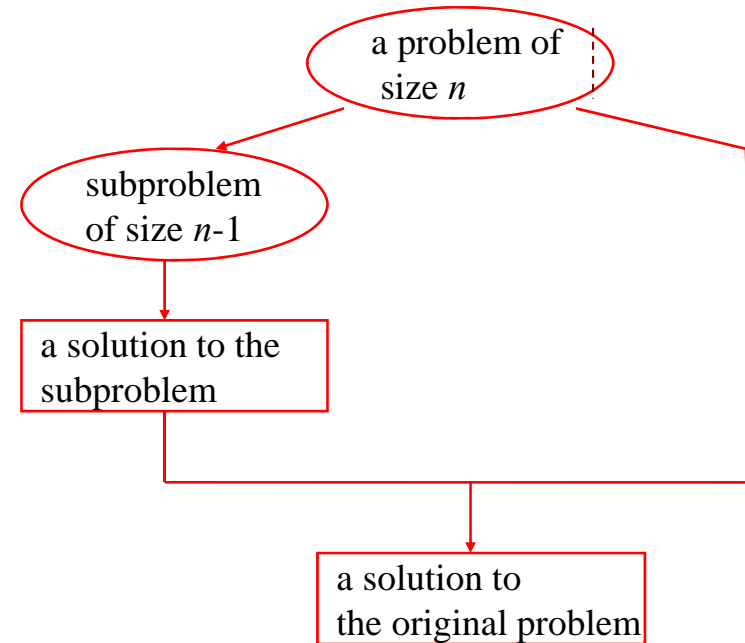
*the size of the problem is reduced by the same constant on each iteration/  
recursion of the algorithm.*

*Eg.*

- $n !$

- $a^n = a * a^{n-1}$

$$a^n = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



# Decrease and Conquer

## ■ Three variations of Decrease and Conquer tech.

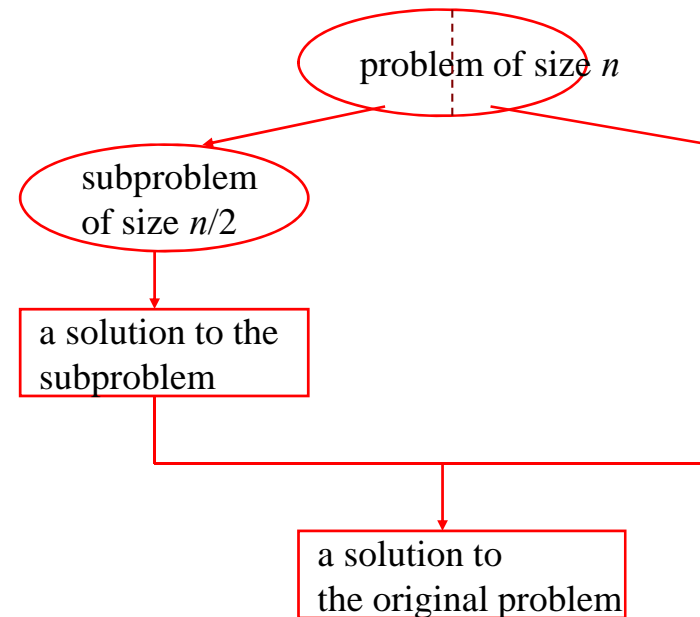
### ✦ Decrease by a constant factor (by half)

*the size of the problem is reduced by the same constant on each iteration/  
recursion of the algorithm.*

*Eg.*

■  $a^n = (a^{n/2})^2$

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd and } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



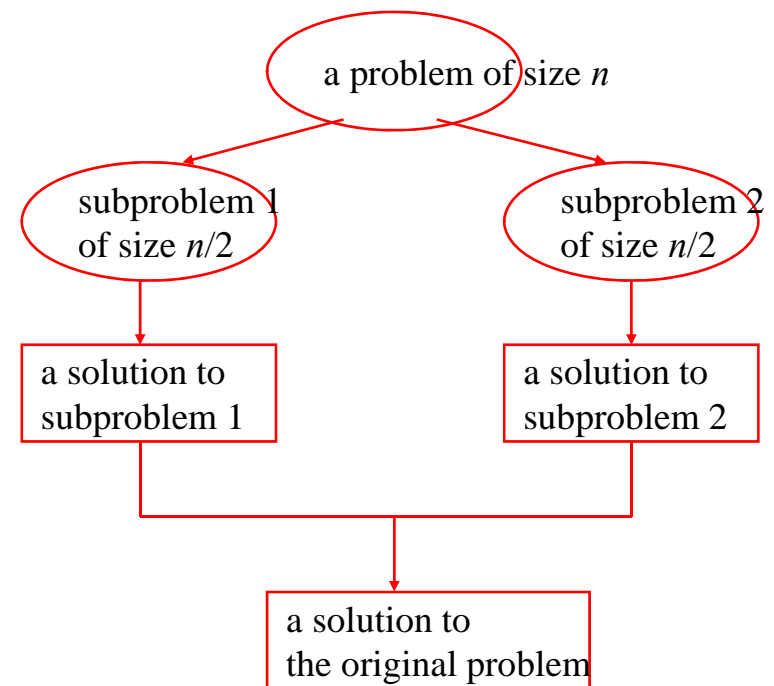
# Decrease and Conquer

---

## ■ Three variations of Decrease and Conquer tech.

for comparison: Divide and Conquer

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil} & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



# Decrease and Conquer

---

## ■ **Three variations of Decrease and Conquer tech.**

### ✦ *Variable-size decrease*

*the size reduction pattern varies from one iteration of an algorithm to another.*

*Eg.*

#### ■ *Euclid's algorithm*

$\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$  iteratively while  $n \neq 0$

$\text{gcd}(m, 0) = m$

# Insertion Sort

---

## ■ algorithm --- Decrease by one

### ✦ idea

- assume that the smaller problem of sorting array  $A[0 \dots n-2]$  has been solved to give a sorted array of size  $n-1$ :  $A[0 \dots n-2]$
- we can take advantage of this solution to the smaller problem to get a solution to the original problem, ----- to find an appropriate position for  $A[n-1]$  among the sorted elements  $A[0 \dots n-2]$  and insert it there.

### ✦ Three ways to achieve it:

- scan the sorted subarray from left to right,  
----- the first element  $\geq A[n-1]$   
----- insert  $A[n-1]$  before the element *Insertion Sort*
- scan the sorted subarray from right to left,  
----- the first element  $\leq A[n-1]$   
----- insert  $A[n-1]$  after the element *Insertion Sort*
- use binary search to find appropriate position for  $A[n-1]$  in the sorted subarray  
*binary Insertion Sort*

# Insertion Sort

---

## ■ Insertion Sort: an Iterative Solution


### ALGORITHM InsertionSortIter(A[0..n-1])

//An iterative implementation of insertion sort

//Input: An array A[0..n-1] of n orderable elements

//Output: Array A[0..n-1] sorted in nondecreasing order

```
for  $i \leftarrow 1$  to  $n - 1$  do           // i: the index of the first element of the unsorted part.
     $v = A[i]$ 
     $j = i - 1$                        // j: the index of the sorted part of the array
    while  $j \geq 0$  and  $A[j] > v$  do //compare the key with each element in the sorted part
         $A[j+1] \leftarrow A[j]$ 
         $j \leftarrow j-1$ 
     $A[j+1] \leftarrow v$               //insert the key to the sorted part of the array
```



$A[0] < \dots < A[j] < A[j+1] < \dots < A[i-1] \mid \textcolor{red}{A[i]} \dots A[n-1]$   
Smaller than or equal to  $A[i]$                       greater than  $A[i]$

# Insertion Sort

---

- *Example*

1:	89	<b>45</b>	68	90	29	34	17
2:	45	89	<b>68</b>	90	29	34	17
3:	45	68	89	<b>90</b>	29	34	17
4:	45	68	89	90	<b>29</b>	34	17
5:	29	45	68	89	90	<b>34</b>	17
6:	29	34	45	68	89	90	<b>17</b>
7:	17	29	34	45	68	89	90

*The vertical bar separates the sorted part of the array from the remaining elements  
The element to be inserted is in bold.*



# Insertion Sort

---

## ■ Analysis of Insertion Sort

### ✦ Worst-case:

- $A[j] > v$  is executed for every  $j = i-1, \dots, 0$
- If and only if  $A[j] > A[i]$  for  $j = i-1, \dots, 0$
- i.e., the original input is an array of strictly decreasing values

$$C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \theta(n^2)$$

# Insertion Sort

---

## ■ Analysis of Insertion Sort

### ✦ Best-case:

- $A[j] > v$  is executed only once on every iteration
- If and only if  $A[i-1] \leq A[i]$  for every  $i = 1, \dots, n-1$
- the original input is already sorted in ascending order

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \theta(n)$$

- Application: *almost sorted files*

*while sorting by quicksort, after subarrays become smaller than some predefined size, we can switch to using insertion sort*

*-- typically decreases the total running time of quicksort by about 10%*

# Insertion Sort

---

## ■ Analysis of Insertion Sort

✦ Average-case:

$$C_{avg}(n) \approx \frac{n^2}{4} \in \theta(n^2)$$

☆ Selection Sort  $\Theta(n^2)$  ; Bubble Sort  $\Theta(n^2)$  ;

☆ Shell Sort see Exer. 5.1 -10

# Graph Traversal

---

## ■ **Graph traversal algorithms** --- **Decrease by one**

- Many problems require processing all graph vertices or edges in a systematic fashion

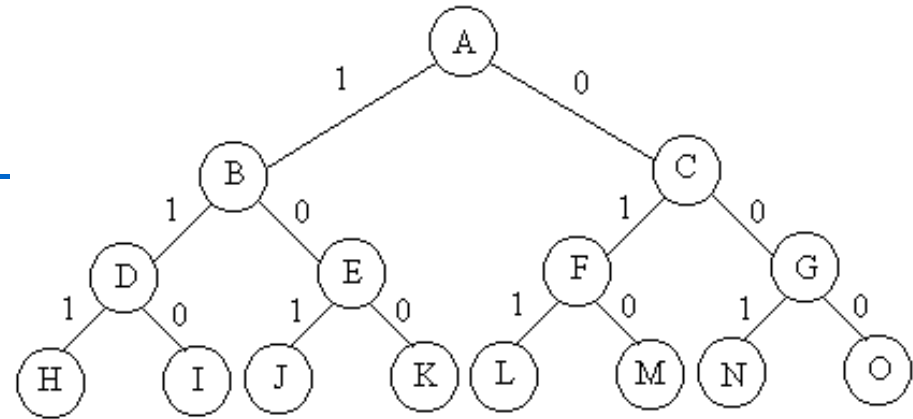
✦ *Depth-first search*

✦ *Breadth-first search*

# Depth-First Search

## ■ Depth-First Search

### ✦ Idea



- *traverse “deeper” whenever possible.*
- *Starts visiting vertices of a graph at an arbitrary vertex, making it as having been visited*
- *On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. If there are more than one neighbors, break the tie by the alphabetic order of the vertices.*
- *When reaching a dead end ( a vertex with no adjacent unvisited vertices), the algorithm backs up one edge to the parent and tries to continue visiting unvisited vertices from there.*
- *The algorithm halts after backing up to the starting vertex, with the latter being a dead end.*

# Depth-First Search

---

## ■ Depth-First Search

### ✦ Idea

- *use a stack to trace the operation of depth-first search.*
  - *Push a vertex onto the stack when the vertex is reached for the first time.*
  - *Pop a vertex off the stack when it becomes a dead end.*
- *Constructing the depth-first search forest*
  - *The traversal's starting vertex serves as the root of the first tree in the forest*
  - *A new unvisited vertex is reached , it is attached to the tree as a child to the vertex from which it is reached*
  - *back edge: a tree edge leading to a previously visited vertex (its ancestor)*

# Depth-First Search

---

**DFS(G)** // Use depth-first to visit  $G=(V,E)$ , which might contain multiple connected components

count  $\leftarrow$  0 //visiting sequence number

mark each vertex with 0 // (unvisited)

for each vertex  $v \in V$  do

    if  $v$  is marked with 0 //  $v$  has not been visited yet.

        dfs( $v$ )

**dfs(v)** //Use depth-first to visit a connected component starting from vertex  $v$ .

count  $\leftarrow$  count + 1

mark  $v$  with count //visit vertex  $v$

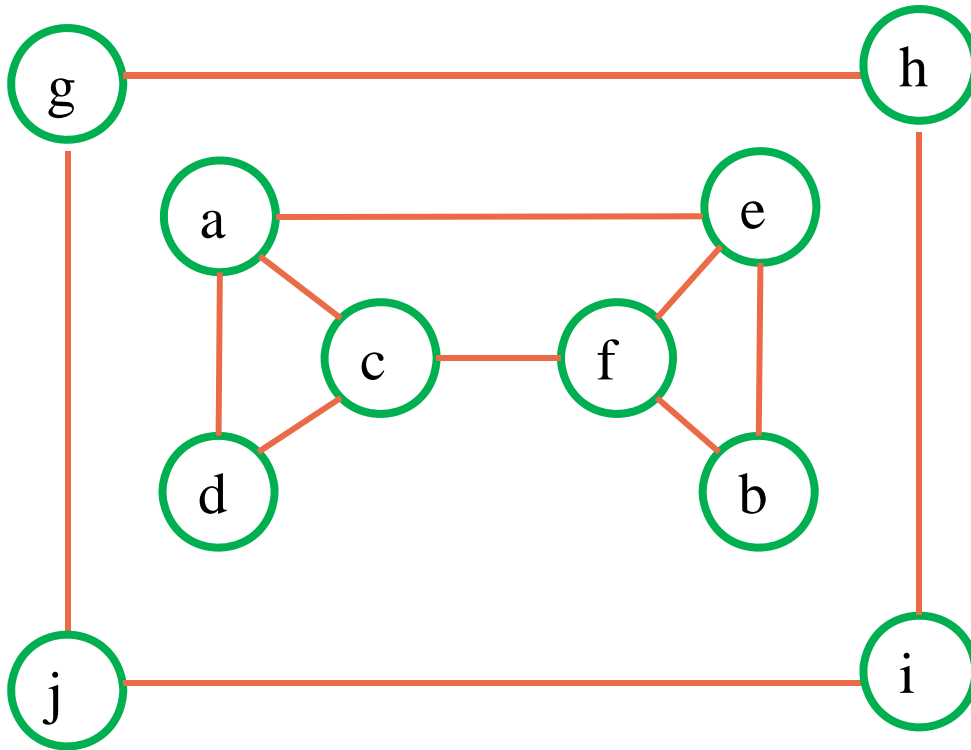
for each vertex  $w$  adjacent to  $v$  do

    if  $w$  is marked with 0 //  $w$  has not been visited yet.

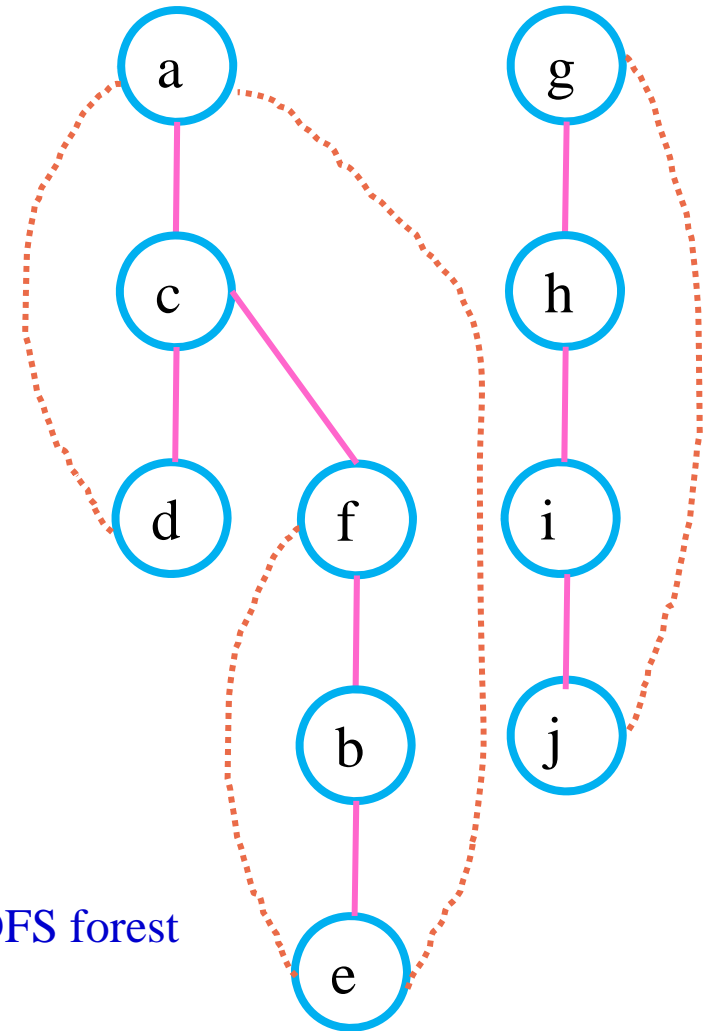
        dfs( $w$ )

# Depth-First Search

- Example: DFS



graph



DFS forest



# Depth-First Search

---

## ■ applications of DFS

### ✦ Applications

- *Checking connectivity of a graph*
- *Checking acyclicity of a graph*
- *Find the strongly connected components of a directed graph.*
- *Find the articulation points of an undirected graph*
- *Topological sort of a directed graph*
- *Classification of edges.*
- *Verify if an undirected graph is connected or not.*

# Backtracking 回溯法

---

## ■ 问题的解空间:

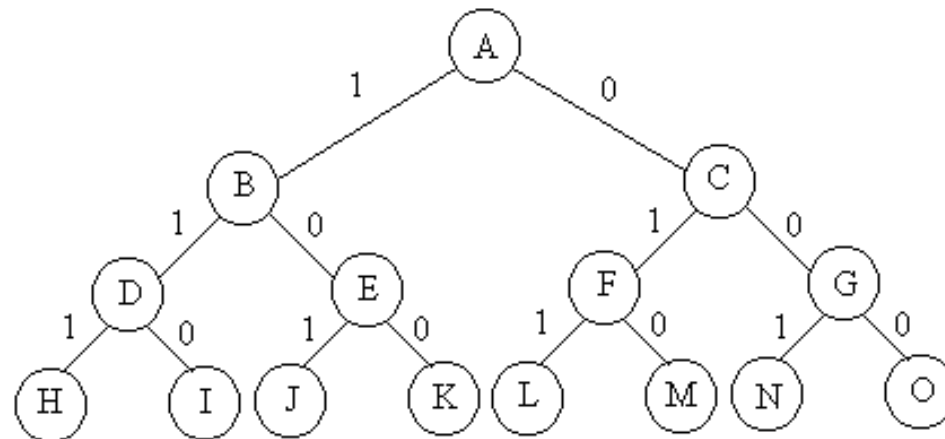
### ✦ 问题的解空间

- 问题的解向量：回溯法希望一个问题的解能够表示成一个n元式  $(x_1, x_2, \dots, x_n)$  的形式。
- 显约束：对分量 $x_i$ 的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。
  - ✦ 注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）

# Backtracking

## ✦ 解空间的组织

- 将解空间组织成树或图的形式，方便回溯法进行搜索。
- **Example:**  $n=3$ 时的0-1背包问题的解空间用完全二叉树表示  
第 $i$ 层到第 $i+1$ 层边上的标号给出了分量的值。  
从树根到叶的任一路径表示解空间中的一个元素。



# Backtracking

---

## ■ 回溯法的基本思想

### ✦ 深度优先的问题状态生成法

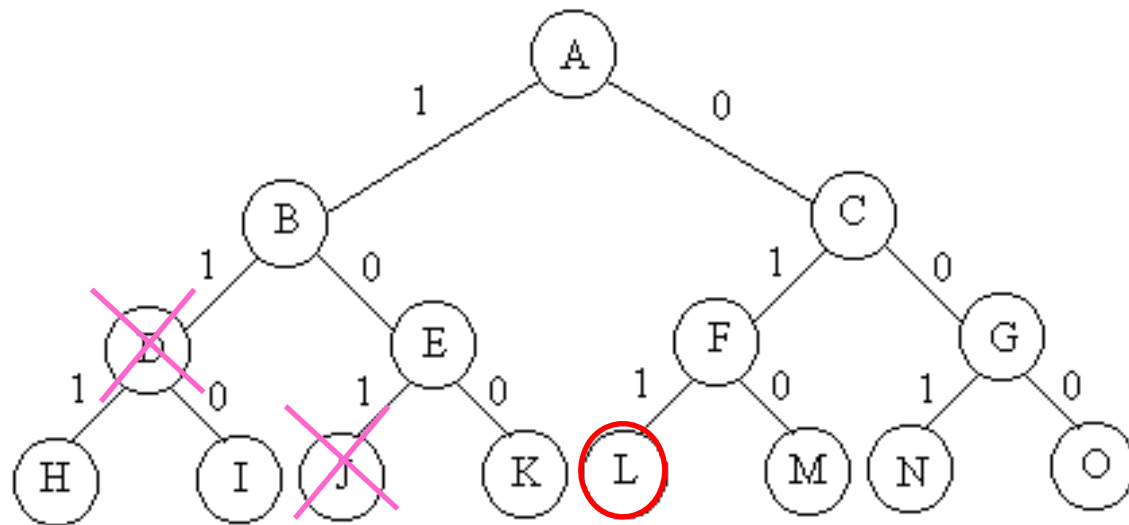
- 从开始结点（根结点）出发，开始结点成为当前的扩展结点
- 在当前的扩展结点处搜索向纵深方向发展，移至一个新结点，这个新结点成为新的活结点，也成为当前扩展结点
- 如果在当前的扩展结点处不能向纵深方向移动，则当前结点成为死结点，应回溯（往回移动）到最近的一个活结点，并使这个活结点成为当前的扩展结点
- 直到找到所要求的解，或解空间已无活结点时为止。

# Backtracking

---

✦ *Example:*

$n=3$ 时的0-1背包问题,  $w = [16, 15, 15], p = [45, 25, 25], c = 30$



# Backtracking

## ✦ Example: 旅行售货员问题

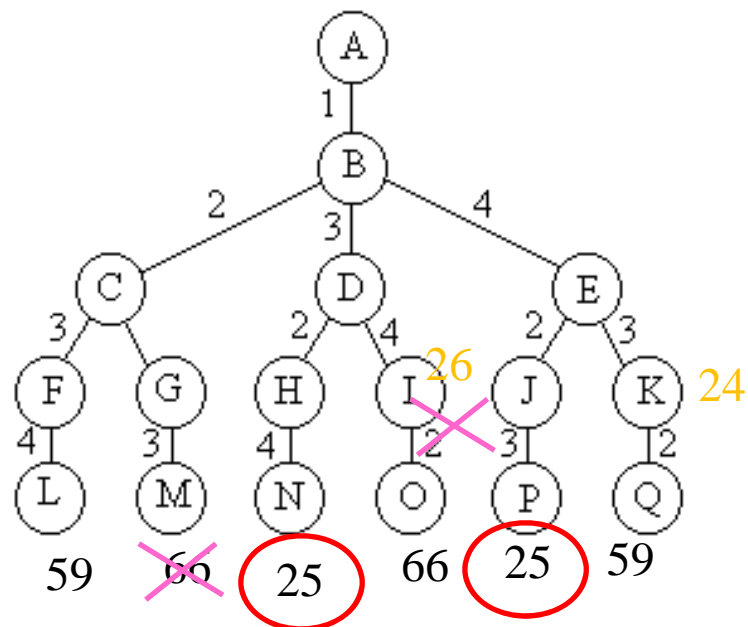
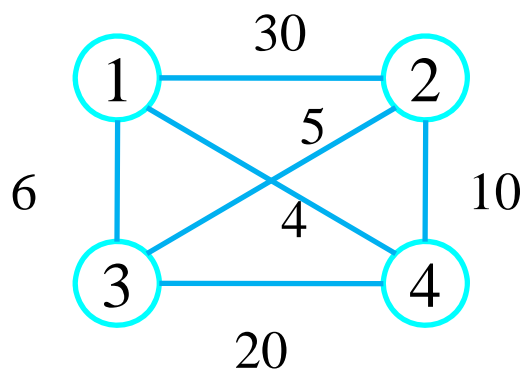
某售货员要到若干城市去推销商品，已知各城市间的路程(或费用)，要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程(总旅费)最小。

用图论的形式描述：

无向带权图 $G=(V, E)$ ，图中一条周游路线是包括 $V$ 中各顶点在内的一条回路。

周游路线的费用是这条路线上所有边的权(费用, 正数)之和。

对应于解空间树上一条从根结点到叶结点的路径。解空间树的叶结点的个数 $(n-1)!$



# Backtracking

---

## ■ 剪枝函数

- 避免无效搜索，提高回溯法的搜索效率
- 用约束函数在扩展结点处剪去不满足约束的子树；
- 用限界函数剪去得不到最优解的子树。

**具有限界函数的深度优先生成法称为回溯法。**

*Example:*

- 01背包问题的回溯法用剪枝函数剪去导致不可行解的子树
- 在旅行售货员问题的回溯法中，如果从根结点到当前扩展结点处的部分周游路线的费用已经超过当前找到的最好的周游路线的费用，则可以断定以该结点为根的子树中不含最优解，可将该子树剪去

# Backtracking

---

## 回溯法 in 0-1背包问题

- 0-1背包问题是子集选取问题，解空间可用子集树表示
- 上界函数：  
计算右子树中解的上界：将剩余物品按单位重量价值排序，依次装入物品，直至装不下时，再装入该物品的一部分而装满背包，得到的价值是右子树中解的上界。

### *Example:*

$n = 4, c = 7, p = [9, 10, 7, 4], w = [3, 5, 2, 1]$

物品的单位重量价值分别为  $[3, 2, 3.5, 4]$ ,

按递单位重量价值的减序装入：4—3—1—0.2的物品2

相应的价值为22

故最优值的上界不超过22



# Backtracking

---

## ✦ 回溯法 in 旅行售货员问题

- 旅行售货员问题的解空间是一颗排列树
- 当 $i=n$ 时，当前扩展结点是排列树的叶结点的父结点，此时检测图 $G$ 是否存在一条从顶点 $x[n-1]$ 到顶点 $x[n]$ 的边和一条从顶点 $x[n]$ 到顶点1的边。
  - 如果这两条边都存在，则找到一条旅行售货员回路
    - 此时还需判断这条回路的费用是否优于已找到的当前最优回路的费用 $bestc$ 。如果是，则必须更新当前的最优值 $bestc$ 和当前最优解 $bestx$
- 当 $i < n$ 时，当前扩展结点位于排列树的第 $i-1$ 层，图 $G$ 中存在从顶点 $x[i-1]$ 到顶点 $x[i]$ 的边时， $x[1:i]$ 构成图 $G$ 的一条路径，且当 $x[1:i]$ 的费用小于当前的最优值时算法进入排列树的第 $i$ 层，否则剪去相应的子树。
- 复杂度分析

算法backtrack在最坏情况下可能需要更新当前最优解 $O((n-1)!)$ 次，每次更新 $bestx$ 需计算时间 $O(n)$ ，从而整个算法的计算时间复杂性为 $O(n!)$ 。

# Backtracking

---

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
    if (i == n) {
        if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
            (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge)) {
            for (int j = 1; j <= n; j++) bestx[j] = x[j];
            bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];
        }
    }
    else {
        for (int j = i; j <= n; j++)
            // 是否可进入x[j]子树?
            if (a[x[i-1]][x[j]] != NoEdge &&
                (cc + a[x[i-1]][x[i]] < bestc || bestc == NoEdge)) {
                // 搜索子树
                Swap(x[i], x[j]);
                cc += a[x[i-1]][x[i]];
                Backtrack(i+1);
                cc -= a[x[i-1]][x[i]];
                Swap(x[i], x[j]);
            }
    }
}
```

# Backtracking

---

## ✦ 回溯法的解题步骤

- 针对所给问题，定义问题的解空间；
- 确定易于搜索的解空间结构；
- 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

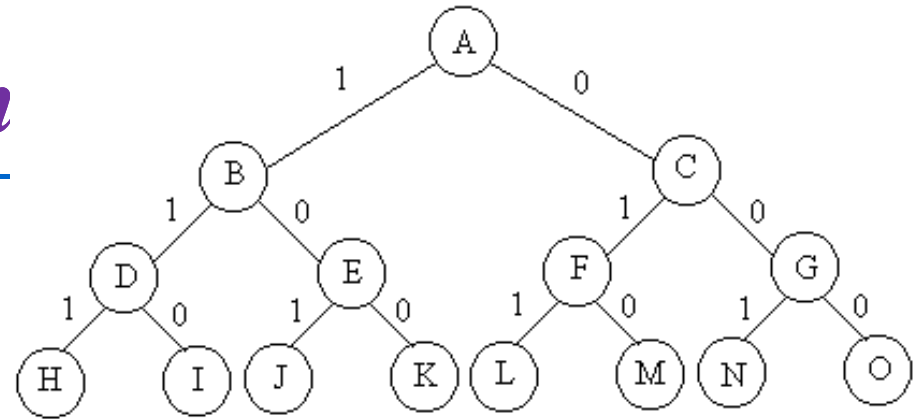
## ✦ 回溯法的效率分析

- ◆ 用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。
- ◆ 如果解空间树中从根结点到叶结点的最长路径的长度为 $h(n)$ ，则回溯法所需的计算时间通常为 $O(h(n))$ 。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 内存空间。

# Breadth-First Search

## ■ Breadth-First Search

### ✦ Idea

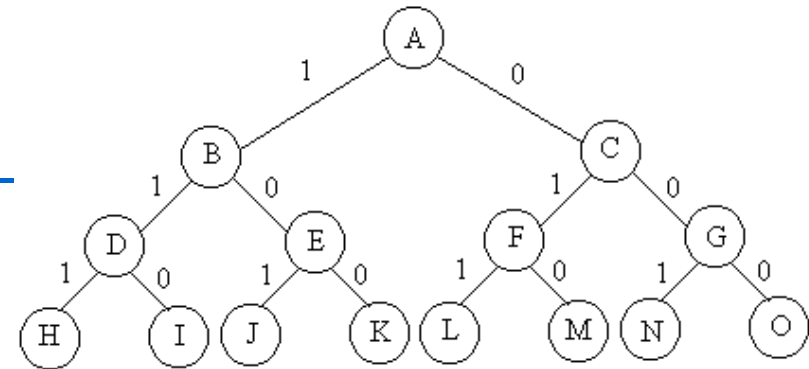


- *Traverse “wider” whenever possible*
- *Discover all vertices at distance  $k$  from  $s$  (on level  $k$ ) before discovering any vertices at distance  $k + 1$  (at level  $k + 1$ ).*
- *Starts visiting vertices of a graph at an arbitrary vertex, making it as having been visited*
- *Visit all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on...*
- *Similar to level-by-level tree traversals*
- *Until all vertices in the same connected component as the starting vertex are visited, if there still remain unvisited vertices, it will restart at an arbitrary vertex of another connected component of the graph*

# Breadth-First Search

## ■ Breadth-First Search

👉 queue



- Instead of a stack, breadth-first uses a **queue** to trace the operation.
  - The queue is initialized with the starting vertex, and is marked as visited
  - On each iteration, the process identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, adds them to the queue
  - The front vertex is removed from the queue
- The queue is **FIFO**, first-in-first-out,
  - The order in which vertices are added to the queue is the same order in which they are removed from it.

# Breadth-First Search

---

## ■ Breadth-First Search

- Constructing the *Breadth-First search forest*
  - The traversal's starting vertex serves as the root of the first tree in the forest
  - A new unvisited vertex is reached , it is attached to the tree as a child to the vertex from which it is reached
  - *Cross edge*: a tree edge leading to a previously visited vertex ( other than its immediate predecessor)
  - Cross edges connect *vertices either on the same or adjacent levels of a BFS tree*

# Breadth-First Search

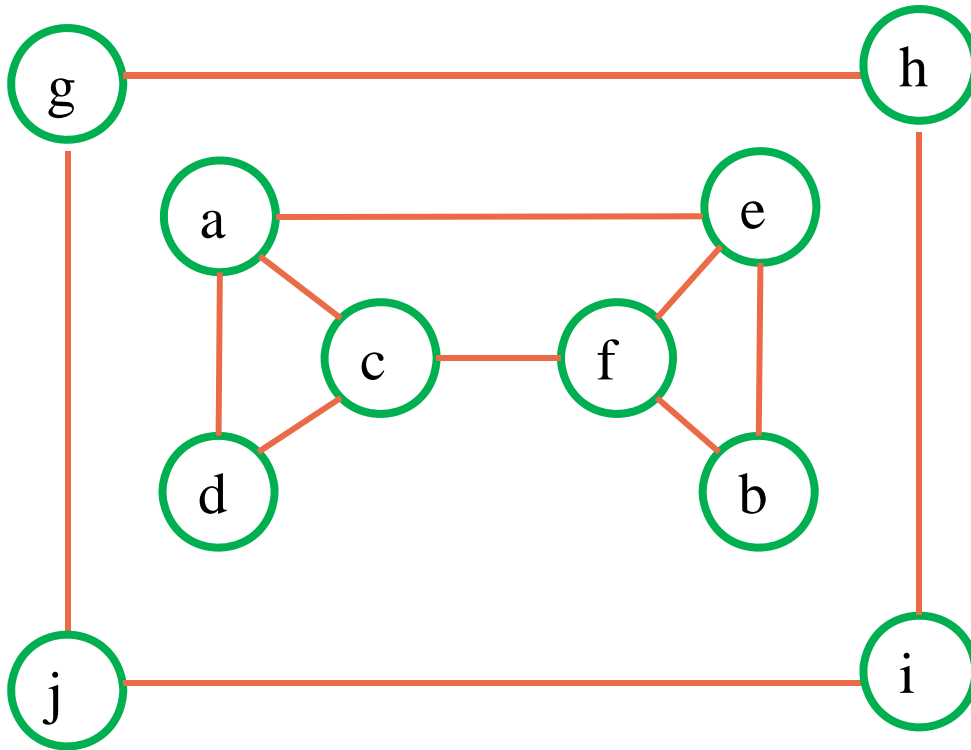
---

```
BFS(G)
count  $\leftarrow$  0
mark each vertex with 0
for each vertex  $v \in V$  do
    if  $v$  is marked with 0
        bfs( $v$ )
```

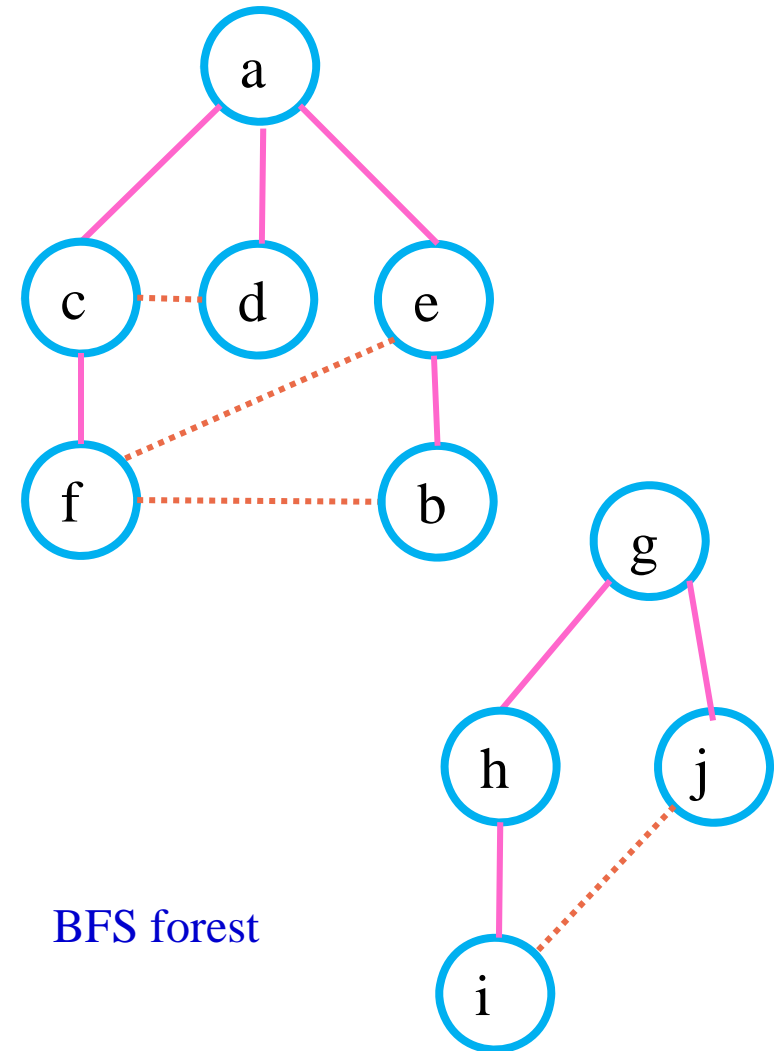
```
bfs( $v$ )
count  $\leftarrow$  count + 1
mark  $v$  with count           //visit  $v$ 
initialize queue with  $v$     //enqueue
while queue is not empty do
     $a \leftarrow$  front of queue //dequeue
    for each vertex  $w$  adjacent to  $a$  do
        if  $w$  is marked with 0 //w hasn't been visited.
            count  $\leftarrow$  count + 1
            mark  $w$  with count //visit  $w$ 
            add  $w$  to the end of the queue //enqueue
    remove  $a$  from the front of the queue
```

# Breadth-First Search

- Example: BFS



graph



BFS forest

BFS queue: a c d e f b g h j i



# Breadth-First Search

---

## ■ Breadth-First Search

### ✦ Arrays for BFS.

- $d[v]$ : The shortest distance from  $s$  to  $v$
- $\pi[u]$ : The predecessor or parent  $\pi[v]$ , which is used to derive a shortest path from  $s$  to vertex  $v$
- $color[v]$ 
  - WHITE means undiscovered
  - GRAY means discovered but not “processed”
  - BLACK means finished processing.

# Breadth-First Search

---

## ■ Efficiency of BFS

*BFS has same efficiency as DFS*

- *Adjacency matrices:  $\Theta(|V|^2)$*
- *Adjacency linked lists:  $\Theta(|V| + |E|)$*

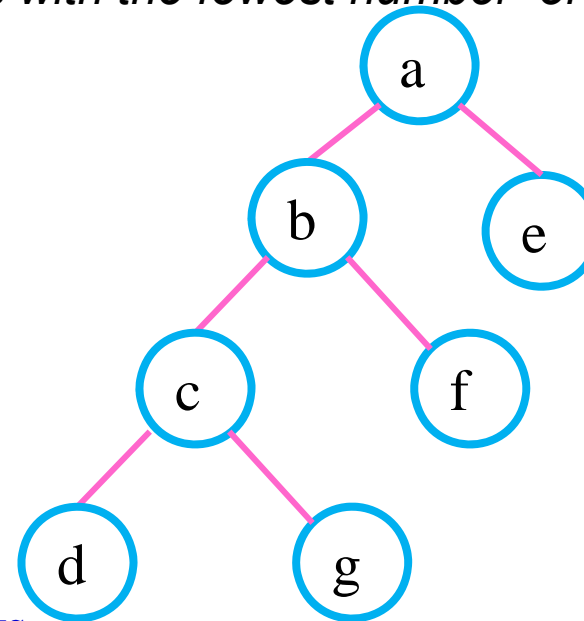
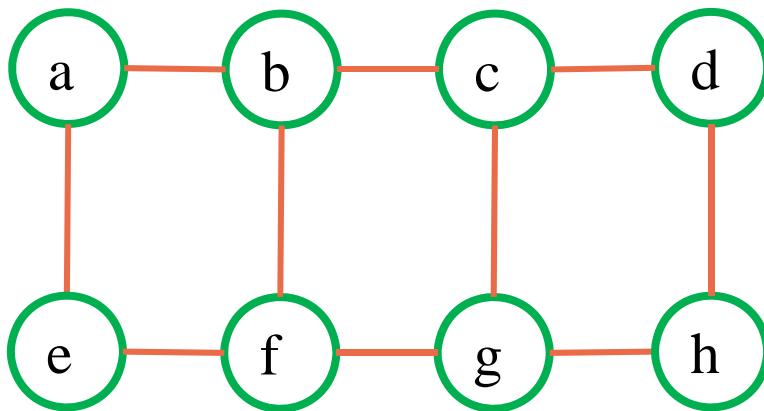
# Breadth-First Search

---

## ■ applications of BFS

### ★ Applications

- Checking connectivity of a graph
- Checking acyclicity of a graph
- Find a path between two given vertices with the fewest number of edges



Part of BFS tree  
Identifies the minimum-edge path from *a* to *g*

# *Branch-and-Bound* 分支限界法

---

## ▣ 分支限界法基本思想

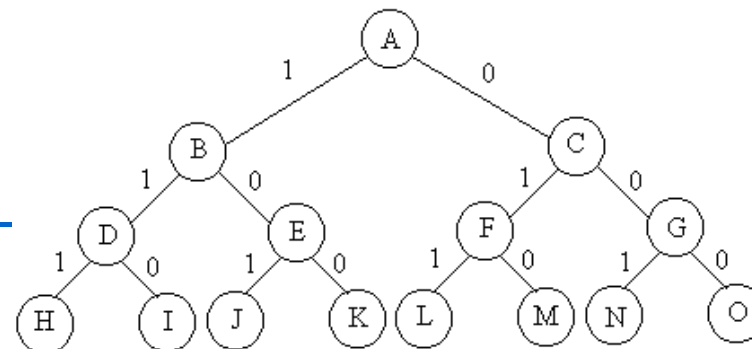
✦ 解空间树 - 有序树

- 子集树
- 排列树

✦ 以广度优先或以最小耗费优先的方式搜索解空间树

# Branch-and-Bound

## ✦ 对当前结点的扩展方式：



- 在分支限界法中，每一个活结点只有一次机会成为扩展结点。
- 活结点一旦成为扩展结点，就一次性产生其所有儿子结点(分支)
- 在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中。
- 此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。
- 为有效选择下一扩展结点，加速搜索进程，在每一活结点处，计算一个函数值(限界)，并根据限界函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间上有最优解的分支推进

# Branch-and-Bound

---

✦ 从活结点表中选择下一个扩展结点的两种方式——两种分支限界法

- **队列式(FIFO)分支限界法**

将活结点表组织成队列，按照队列先进先出（FIFO）原则选取下一个结点为扩展结点

- **优先队列式分支限界法**

- 将活结点表组织成优先队列，按照优先队列中规定的优先级选取优先级最高的结点成为当前扩展结点。

- 结点优先级用一个与该结点相关的数值 $p$ 表示

- **最大优先队列**： $p$ 值较大的结点优先级高

用最大堆来实现，用最大堆的 $\text{deletemax}$ 运算抽取堆中下一个结点成为当前扩展结点

- **最小优先队列**： $p$ 值较小的结点优先级高

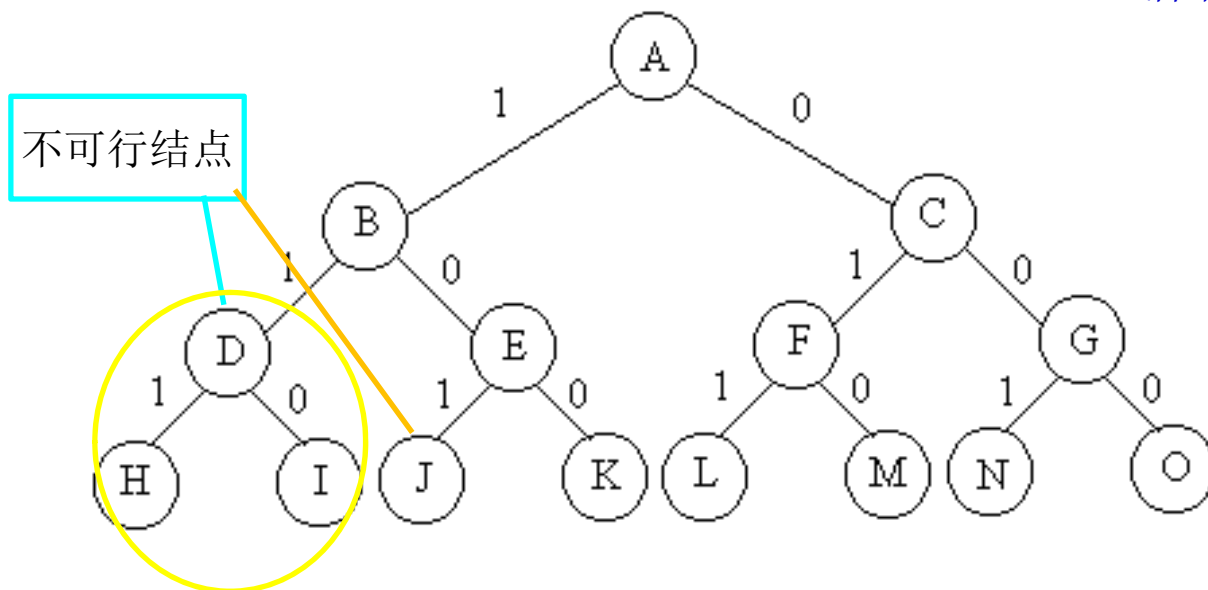
用最小堆来实现，用最小堆的 $\text{deletemin}$ 运算抽取堆中下一个结点成为当前扩展结点

# Branch-and-Bound

✦ **Example:**  $n=3$ 时的0-1背包问题,  $w = [16, 15, 15]$ ,  $p = [45, 25, 25]$ ,  $c = 30$

➤ 队列式分支限界法

活结点队列 → 先进先出原则



*BC*

*BC E FG*

*BC E FG K*

*BC E FG K LM*

*BC E FG K LM N O*

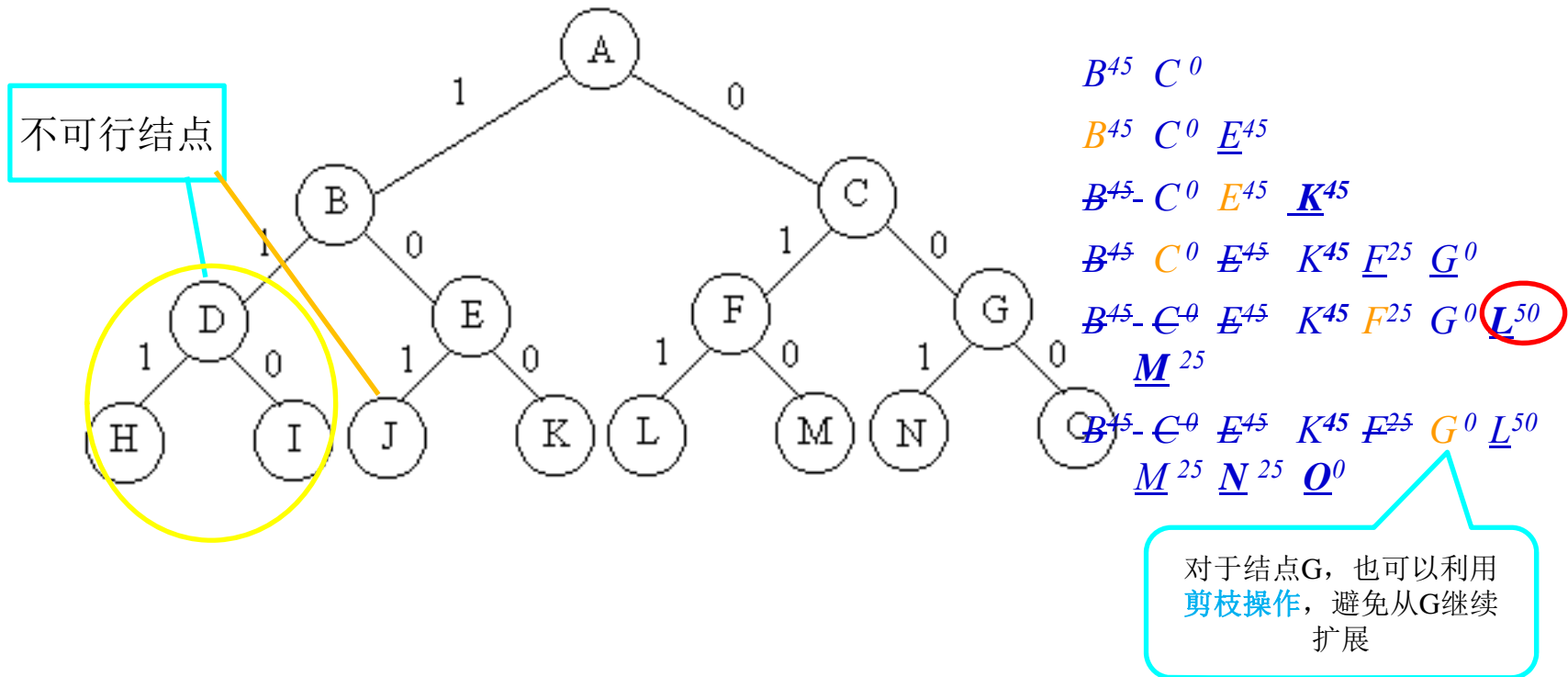
✦ 队列式分支限界法类似于广度优先的遍历算法, 但不搜索以不可行结点为根的子树

# Branch-and-Bound

✦ **Example:**  $n=3$ 时的0-1背包问题,  $w = [16, 15, 15]$ ,  $p = [45, 25, 25]$ ,  $c = 30$

➤ 优先队列式分支限界法

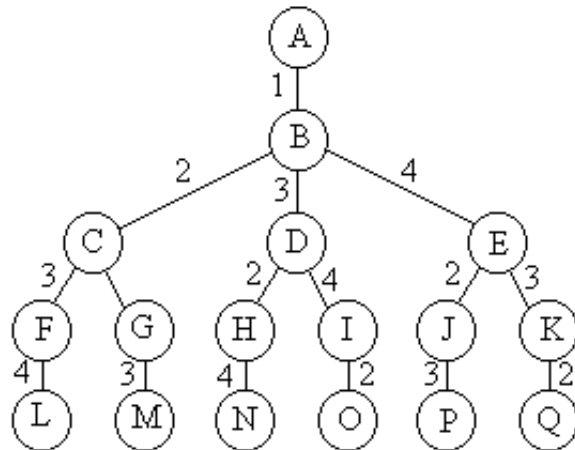
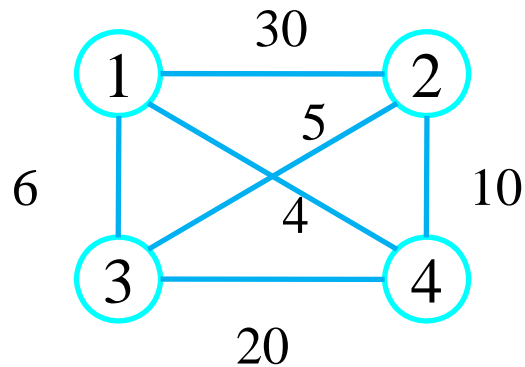
活结点队列  $\rightarrow$  极大堆





# Branch-and-Bound

✦ Example: 四城市旅行售货员问题



➤ 队列式分支限界法

$B \quad \underline{C \quad D \quad E}$

$B \quad \underline{C} \quad D \quad E \quad F \quad G$

$B \in \underline{D} \quad E \quad F \quad G \quad H \quad I$

$B \in \underline{D} \quad \underline{E} \quad F \quad G \quad H \quad I \quad J \quad K$

$B \in \underline{D} \quad \underline{E} \quad \underline{F} \quad G \quad H \quad I \quad J \quad K \quad L^{59}$

$B \in \underline{D} \quad \underline{E} \quad \underline{F} \quad \underline{G} \quad H \quad I \quad J \quad K \quad L^{59} \quad M^{66}$

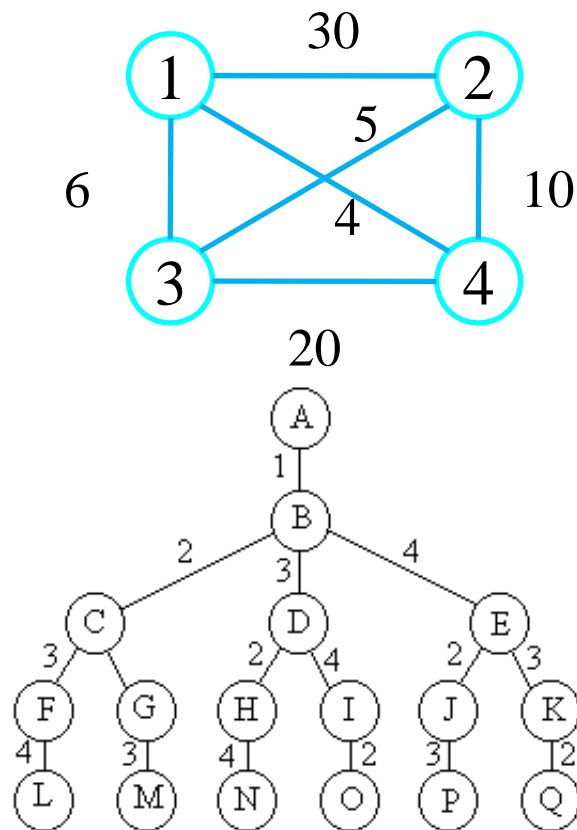
$B \in \underline{D} \quad \underline{E} \quad \underline{F} \quad \underline{G} \quad \underline{H} \quad I \quad J \quad K \quad L^{59} \quad M^{66} \quad \underline{N^{25}}$

$B \in \underline{D} \quad \underline{E} \quad \underline{F} \quad \underline{G} \quad \underline{H} \quad \underline{I} \quad \underline{J} \quad K \quad L^{59} \quad M^{66} \quad N^{25} \quad P^{25}$

$B \in \underline{D} \quad \underline{E} \quad \underline{F} \quad \underline{G} \quad \underline{H} \quad \underline{I} \quad \underline{J} \quad \underline{K} \quad L^{59} \quad M^{66} \quad N^{25} \quad P^{25} \quad Q^{59}$

# Branch-and-Bound

✦ Example: 四城市旅行售货员问题



➤ 优先队列式分支限界法 → 极小堆

$B \ C \ D \ E^4$

$B \ C^{30} \ D^6 \ E^4 \ J^{14} \ K^{24}$

$B \ C^{30} \ D^6 \ E^4 \ J^{14} \ K^{24} \ H^{11} \ I^{26}$

$B \ C^{30} \ D^6 \ E^4 \ J^{14} \ K^{24} \ H^{11} \ I^{26} \ N^{25}$

$B \ C^{30} \ D^6 \ E^4 \ J^{14} \ K^{24} \ H^{11} \ I^{26} \ N^{25} \ P^{25}$

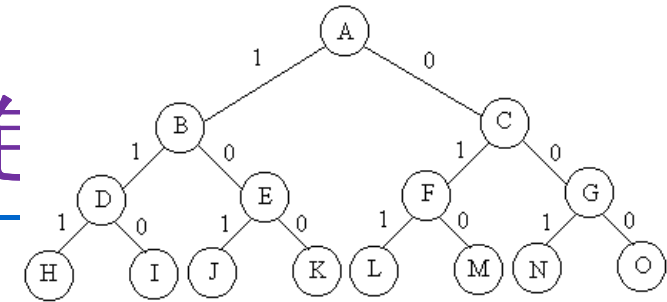
$B \ C^{30} \ D^6 \ E \ J^{14} \ K^{24} \ H^{11} \ I^{26} \ N^{25} \ P^{25} \ Q$

# *Branch-and-Bound* 0-1背包问题

---

- Knapsack, 要对输入数据进行预处理, 将各物品依其单位重量价值从大到小进行排列。
- 结点的优先级由结点的上界函数Bound计算出uprofit。
- 上界函数: 已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和。
- 算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点可行, 则将它加入到子集树和活结点优先队列中。
- 当前扩展结点的右儿子结点一定是可行结点, 仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶结点时, 优先队列中所有活结点的价值上界均不超过该叶结点的价值, 则该叶结点的值为问题的最优值。

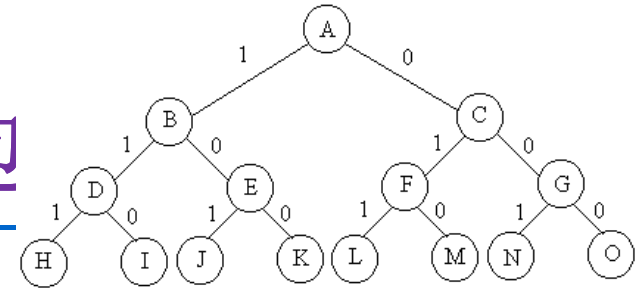
# Branch-and-Bound 0-1背包



## ■ 上界函数Bound

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bound(int i)
{ // 计算上界
    Typew cleft = c - cw; // 剩余容量
    Typep b = cp;
    // 以物品单位重量价值递减序装入物品
    while (i <= n && w[i] <= cleft) {
        cleft -= w[i];
        b += p[i];
        i++;
    }
    if (i <= n) b += p[i]/w[i] * cleft; // 装满背包
    return b;
}
```

# Branch-and-Bound 0-1背包

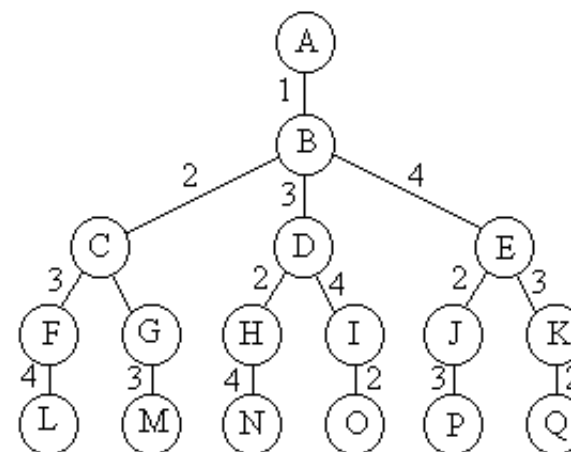
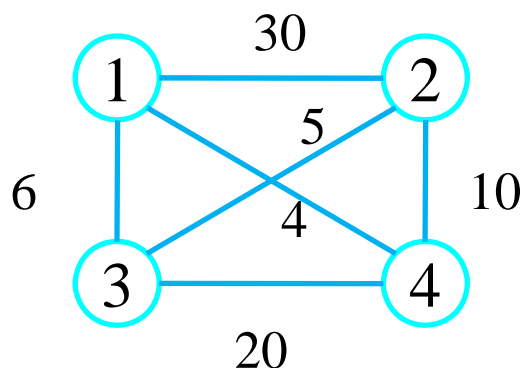


## ■ MaxKnapsack中的分支限界搜索过程

```
while (i != n+1) { // 非叶结点
    // 检查当前扩展结点的左儿子结点
    Typew wt = cw + w[i];
    if (wt <= c) { // 左儿子结点为可行结点
        if (cp+p[i] > bestp) bestp = cp+p[i];
        AddLiveNode(up, cp+p[i], cw+w[i], true, i+1); } //加入到优先队列
    up = Bound(i+1);
    // 检查当前扩展结点的右儿子结点
    if (up >= bestp) // 右子树可能含最优解
        AddLiveNode(up, cp, cw, false, i+1);
    // 取下一个扩展节点（略）
    HeapNode < Typep, Typew > N;
    H -> DeleteMax(N);
    (略)
}
```

# Branch-and-Bound 旅行售货员问题

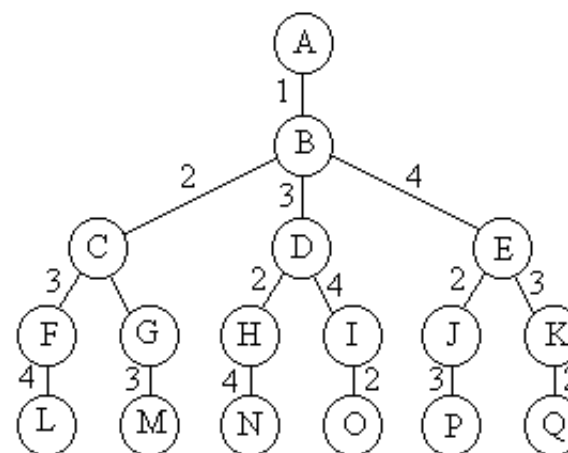
- 某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。
- 路线是一个带权图 $G=(V,E)$ 。图中各边的费用(权)为正数。图的一条周游路线是包括 $V$ 中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。
- 旅行售货员问题的解空间可以组织成一棵排列树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。旅行售货员问题要在图 $G$ 中找出费用最小的周游路线。



# Branch-and-Bound 旅行售货员问题

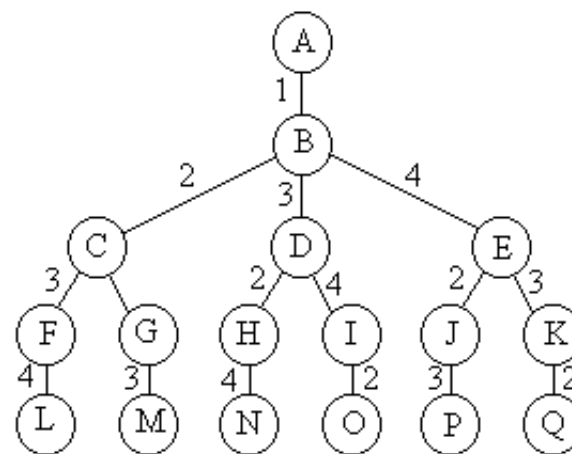
## ✦ 算法描述

- 用剪枝函数加速搜索，剪枝函数是该结点的最小费用的下界
- 如果在当前结点处，这个下界不比当前最优值更小，则可以剪去以该结点为根的子树
- 可以将每个结点的下界作为优先级，按该优先级的非减序从活结点优先队列中抽取下一个扩展结点



# Branch-and-Bound 旅行

## ✦ 算法描述



- 算法开始时创建一个最小堆，用于表示活结点优先队列。堆中每个结点的子树费用的下界 $lcost$ 值是优先队列的优先级。
- 接着计算出图中每个顶点的最小费用出边并用 $minout$ 记录。如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。如果每个顶点都有出边，则根据计算出的 $minout$ 作算法初始化。
- 算法的while循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分2种情况进行处理：
  - 1、首先考虑 $s = n-2$ 的情形，此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。



# Branch-and-Bound 旅行售货员问题

---

## ✦ 算法描述

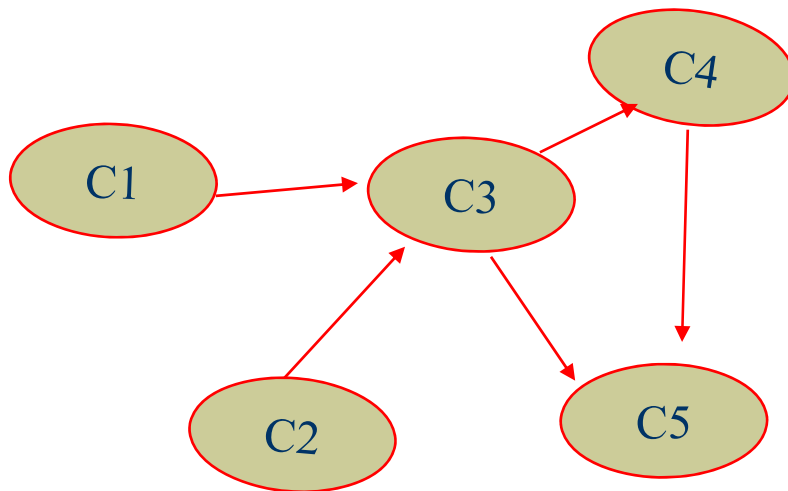
- 2、当 $s < n-2$ 时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所对应的路径是 $x[0:s]$ ，其可行儿子结点是从剩余顶点 $x[s+1:n-1]$ 中选取的顶点 $x[i]$ ，且 $(x[s], x[i])$ 是所给有向图 $G$ 中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀 $(x[0:s], x[i])$ 的费用 $cc$ 和相应的下界 $lcost$ 。当 $lcost < bestc$ 时，将这个可行儿子结点插入到活结点优先队列中。
- 算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点。当 $s=n-1$ 时，已找到的回路前缀是 $x[0:n-1]$ ，它已包含图 $G$ 的所有 $n$ 个顶点。因此，当 $s=n-1$ 时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路费用等于 $cc$ 和 $lcost$ 的值。剩余的活结点的 $lcost$ 值不小于已找到的回路费用。它们都不可能导致费用更小的回路。因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。
- 算法结束时返回找到的最小费用，相应的最优解由数组 $v$ 给出。

# Topological Sorting

---

## ■ **problem**

- *Problem: Given a **Directed Acyclic Graph(DAG)**  $G = (V, E)$ , find a linear ordering of all its vertices such that if  $G$  contains an edge  $(u, v)$ , then  $u$  appears before  $v$  in the ordering.*
- *Example: Give an order of the courses so that the prerequisites are met*



Is the problem solvable if the digraph contains a cycle?

Directed graph (**digraph**):  
All edges with directions;  
Adjacency matrix not have to be symmetric

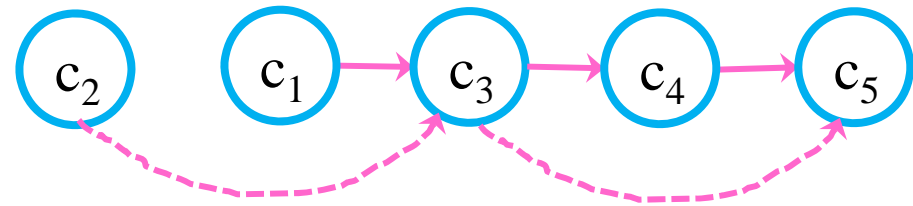
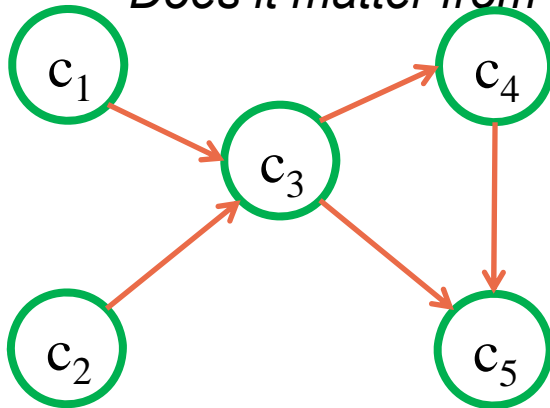
- *Being a dag is not only necessary but also sufficient for Topological Sorting to be possible*

# Topological Sorting

---

## ■ DFS-Based method

- DFS traversal noting the order in which vertices are popped off stack (the order in which the dead end vertices appear)
- Reverse the above order
- Questions
  - Can we use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?
  - Does it matter from which node we start?



# Topological Sorting

## ■ Source Removal method

- Based on a direct implementation of the decrease-by-one techniques.
- Repeatedly identify and remove a **source vertex**, ie, a vertex that has no incoming edges, and delete it along with all the edges outgoing from it.

