Analysis and Design of Algorithms

Chapter 7: Backtracking and Branch & Bound Algorithm

School of Software Engineering © Ye Luo

搜索算法介绍

(1) 穷举搜索

(2) 盲目搜索

— 深度优先(DFS)或回溯搜索(Backtracking);

— 广度优先搜索(BFS);

— 分支限界法 (Branch & Bound);

— 博弈树搜索 (α - β Search)

(3) 启发式搜索

— A* 算法和最佳优先(Best-First Search)

— 迭代加深的A*算法

— B*, AO*, SSS*等算法

— Local Search, GA等算法

Content

- *Exhaustive Search by Enumerating*
- *DFS and BFS*
- *Backtracking*
- *Branch & Bound*

Exhaustive Search by Enumerating

- ***Problem***

  searching for an element with a special property, in a domain that grows exponentially (or faster) with an instance size,

  usually involve combinatorial objects such as permutations, combinations, or subsets of a set.

  Many such problems are optimization problems, to find an element that maximizes or minimizes some desired characteristic

  such as a path's length or an assignment's cost

  E.g. Traveling Salesman Problem, 0-1 Knapsack, Assignment, etc.

Exhaustive Search by Enumerating

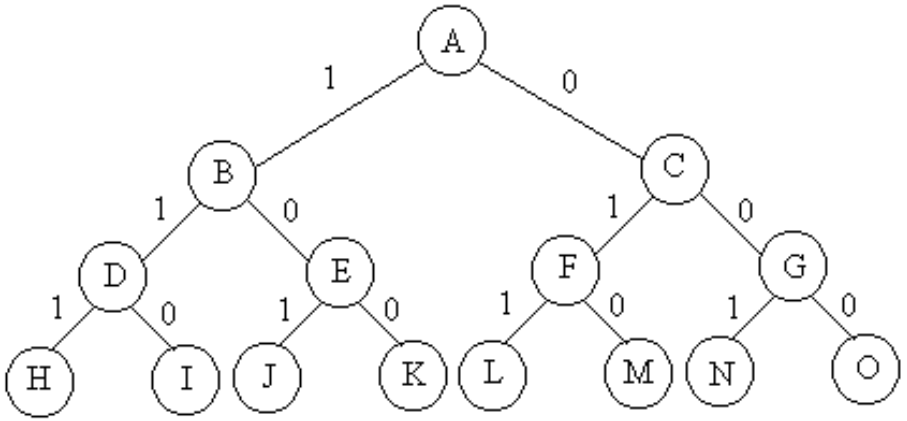
- ***Exhaustive Search— Brute-Force for combinatorial***
  - generate a list of **all potential solutions** to the problem in a systematic manner
  - selecting those of them that satisfy all the constraints
  - evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
  - then search ends, announce the desired solution(s) found (e.g. the one that optimizes some objective function)
  - *typically requires for generating certain combinatorial objects*

Graph Traversal by DFS and BFS

Graph traversal algorithms

- 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)*

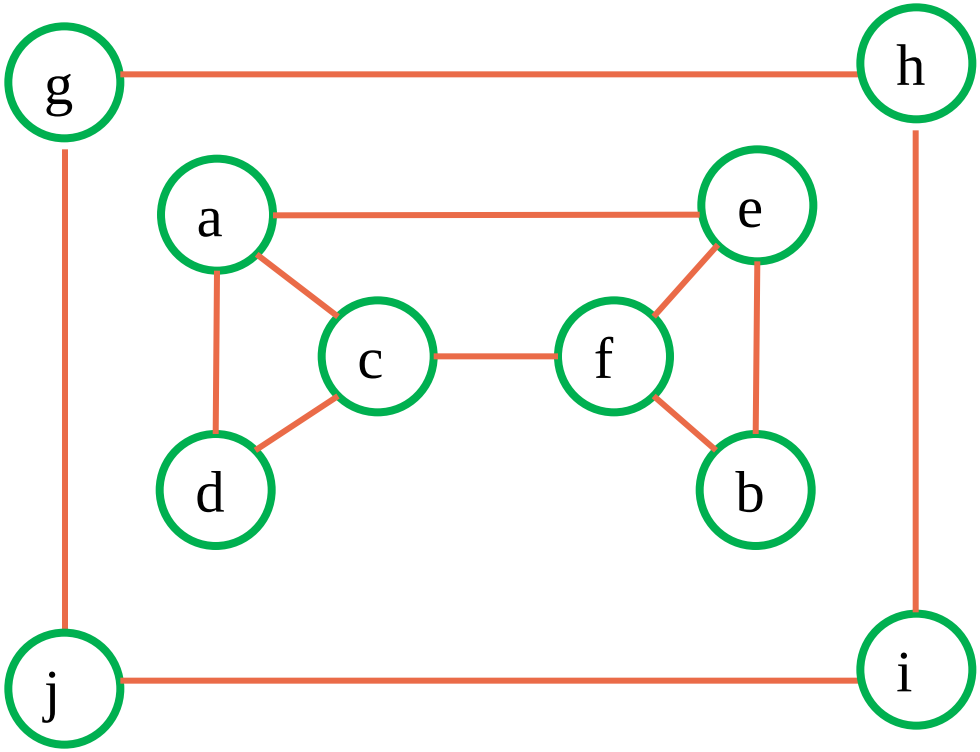
Depth-First Search

```
DFS(G) // Use depth-first to visit G=(V,E), which might contain multiple connected components
    count ← 0                          //visiting sequence number
    mark each vertex with 0            // (unvisited)
    for each vertex v ∈ V do
        if v is marked with 0          //v has not been visited yet.
            dfs(v)
```

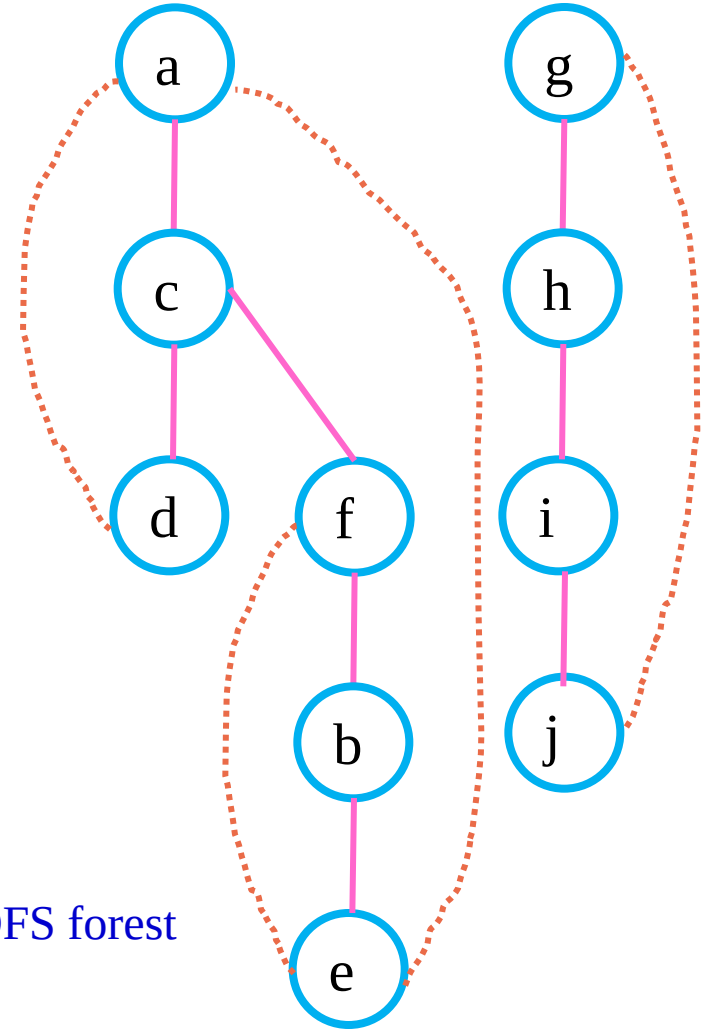
```
dfs(v)          //Use depth-first to visit a connected component starting from vertex v.
    count ← count + 1
    mark v with count                  //visit vertex v
    for each vertex w adjacent to v do
        if w is marked with 0          //w has not been visited yet.
            dfs(w)
```

Depth-First Search

- ***Example: DFS***



graph



DFS forest

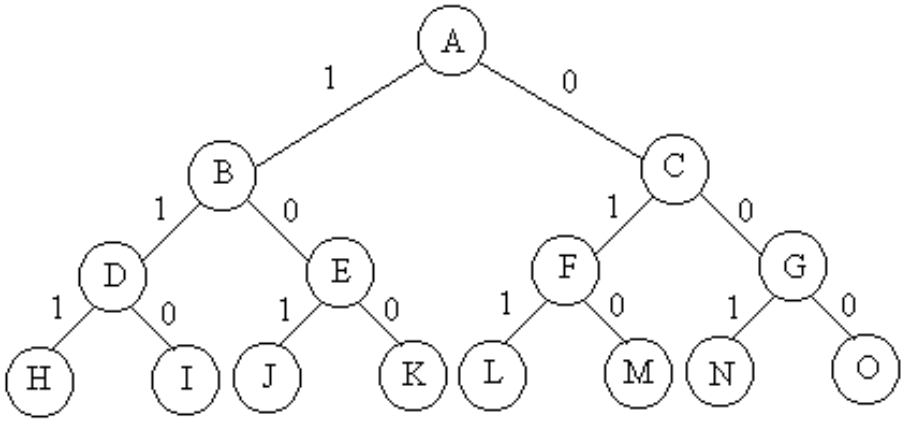
Depth-First Search

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.*

Breadth-First Search

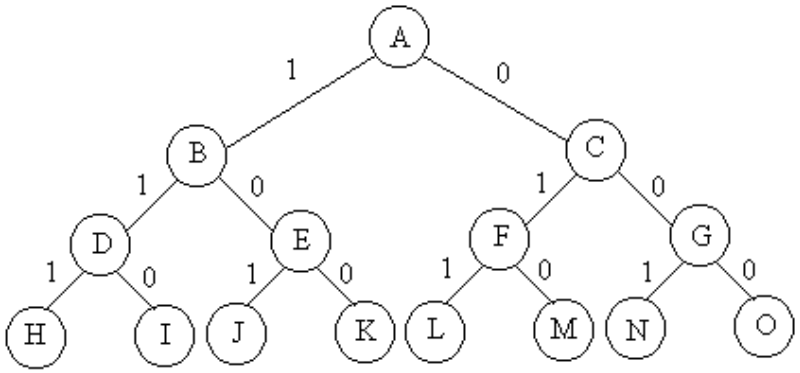


Breadth-First Search

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, fist-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

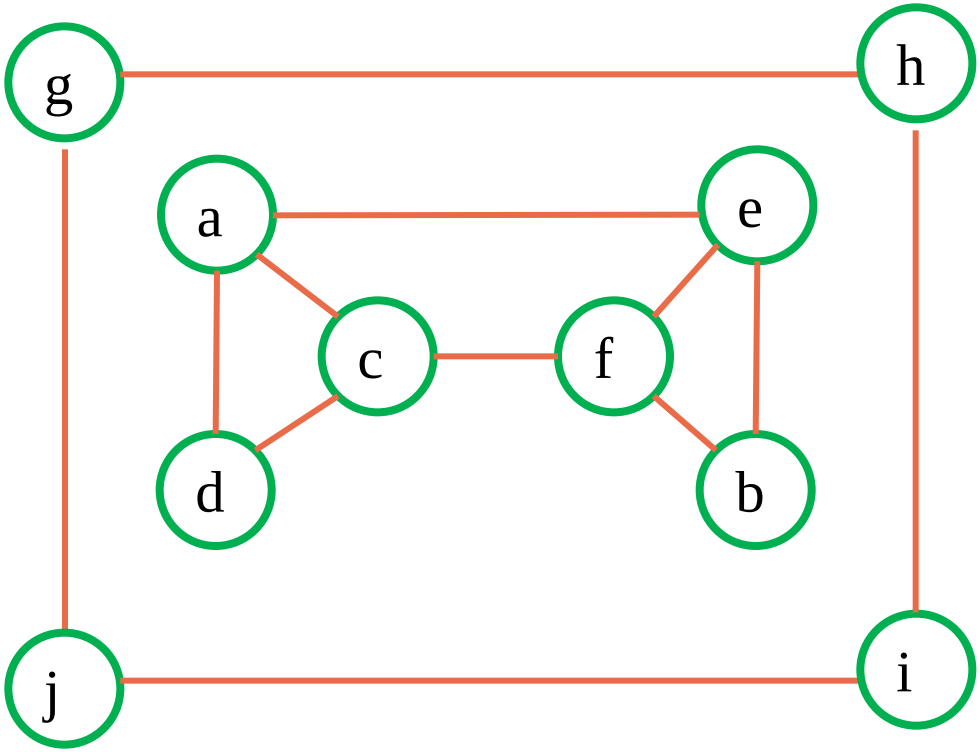
Breadth-First Search

```
BFS(G)
count ← 0
mark each vertex with 0
for each vertex v ∈ V do
      if v is marked with 0
            bfs(v)
```

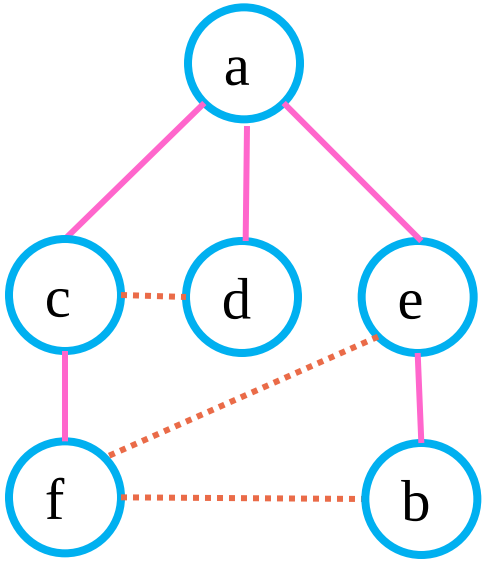
```
bfs(v)
count ← count + 1
mark v with count                          //visit v
initialize queue with v                    //enqueue
while queue is not empty do
  a ← front of queue                       //dequeue
  for each vertex w adjacent to a do
        if w is marked with 0              //w hasn't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

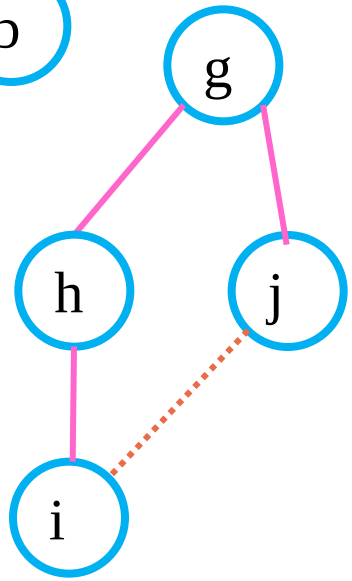
Breadth-First Search

- ***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*
  - *π[u]: The predecessor or parent π[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

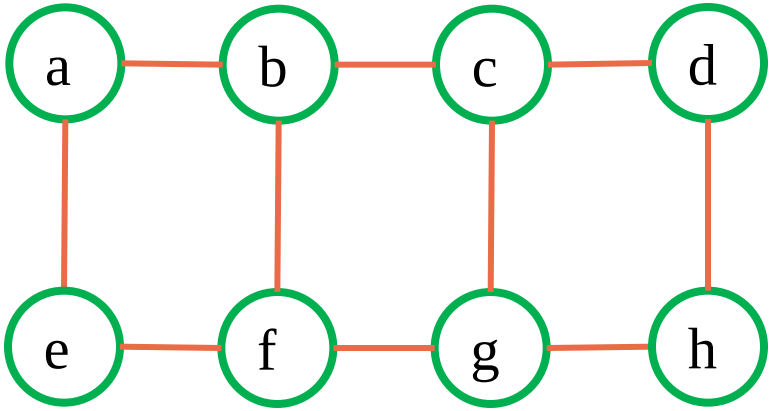
  - *Adjacency matrices:* $\Theta(|V|^2)$
  - *Adjacency linked lists:* $\Theta(|V|+|E|)$

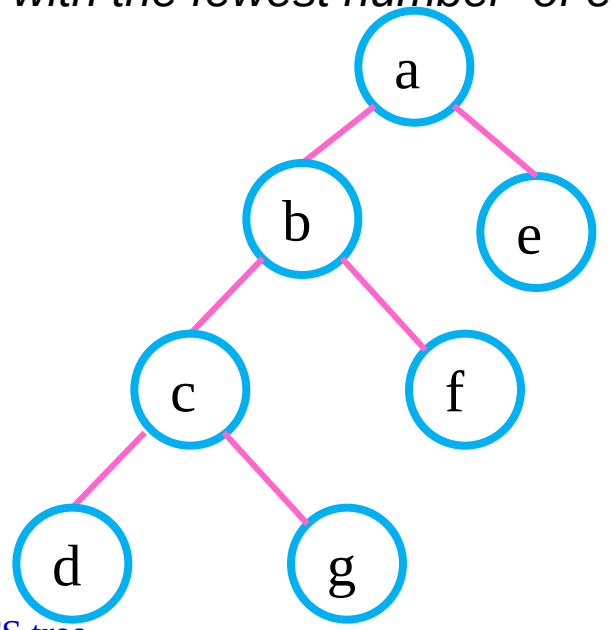
Breadth-First Search

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*

Backtracking Algorithm

- 理解回溯法的深度优先搜索策略。
- 掌握用回溯法解题的算法框架

  （1）递归回溯

  （2）迭代回溯

  （3）子集树算法框架

  （4）排列树算法框架
- 通过应用范例学习回溯法的设计策略。

1 方法概述

- **搜索空间的三种表示：**

— 表序表示：搜索对象用线性表数据结构表示；

— 显示图表示：搜索对象在搜索前就用图（树）的数据结构表示；

— 隐式图表示：除了初始结点，其他结点在搜索过程中动态生成。缘于搜索空间大，难以全部存储。

- **搜索效率的思考：随机搜索**

— 上世纪70年代中期开始，国外一些学者致力于研究随机搜索求解困难的组合问题，将随机过程引入搜索；

— 选择规则是随机地从可选结点中取一个，从而可以从统计角度分析搜索的平均性能；

— 随机搜索的一个成功例子是：判定一个很大的数是不是素数，获得了第一个多项式时间的算法。

1 方法概述

- **回溯法：**

— 回溯法是一个既带有系统性又带有跳跃性的搜索算法；

— 它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。——系统性

— 算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以该结点为根的子树的搜索，逐层向其祖先结点回溯。否则，进入该子树，继续深度优先的策略进行搜索。——跳跃性

— 这种以深度优先的方式系统地搜索问题的解得算法称为回溯法，它适用于解一些组合数较大的问题。

1 方法概述

- **问题的解空间**

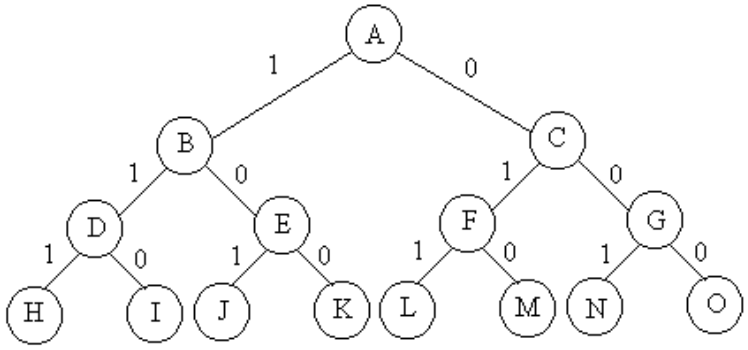
— 问题的解向量：回溯法希望一个问题的解能够表示成一个n元式 $(x_1,x_2,\cdots,x_n)$ 的形式。

— 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。

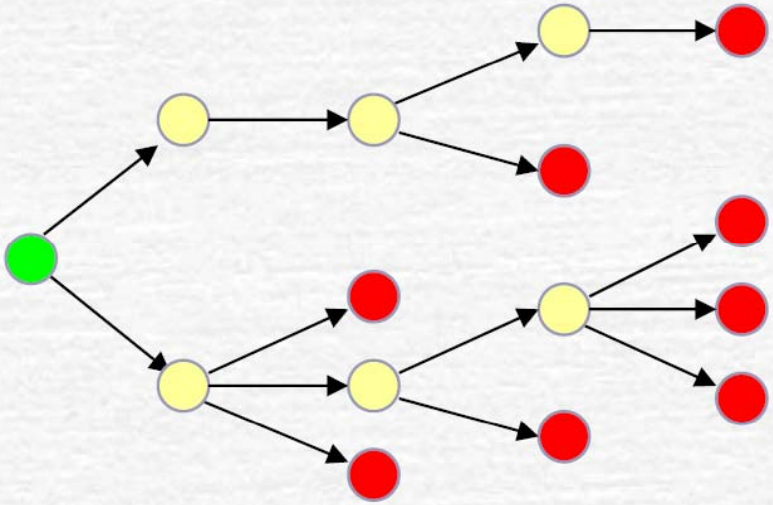
注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。



n=3时的0-1背包问题用完全二叉树表示的解空间

1 方法概述

解空间树：



有三种结点：

- 根结点（搜索的起点）
- 中间结点（非终端结点）
- 叶结点（终端结点）
  ，叶结点为解向量

搜索过程就是找一个或一些特别的叶结点。

1 方法概述

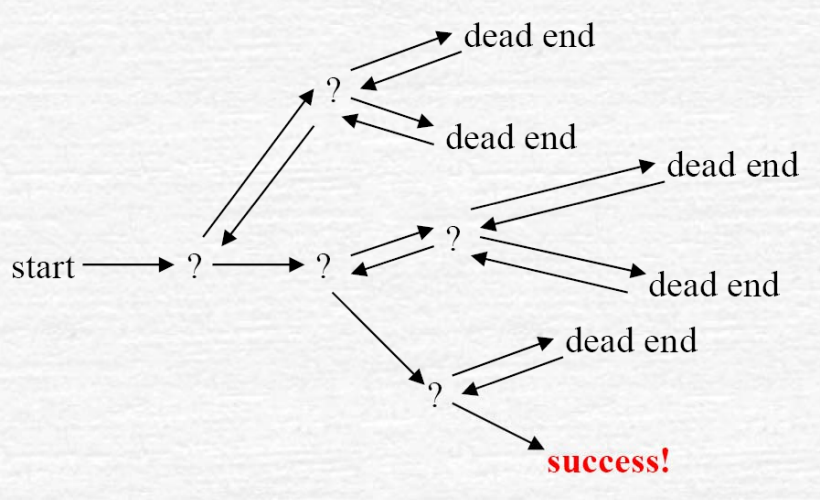
- **基本思想：**

— 搜索从开始结点（根结点）出发，以深度优先搜索整个解空间。

— 这个开始结点成为**活结点**，同时也成为当前的**扩展结点**。在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。

— 如果在当前的扩展结点处不能再向纵深方向扩展，则当前扩展结点就成为**死结点**。

— 此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点；直到找到一个解或全部解。



1 方法概述

- **基本步骤：**

① 针对所给问题，定义问题的解空间；

② 确定易于搜索的解空间结构；

③ 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。

常用剪枝函数：

① 用约束函数在扩展结点处剪去不满足约束的子树；

② 用限界函数剪去得不到最优解的子树。

1 方法概述

- **二类常见的解空间树：**

① **子集树**：当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树称为子集树。子集树通常有2^n个叶子结点，其总结点个数为$2^{n+1}-1$，遍历子集树时间为$\Omega(2^n)$。如0-1背包问题，叶结点数为2^n，总结点数2^{n+1}；

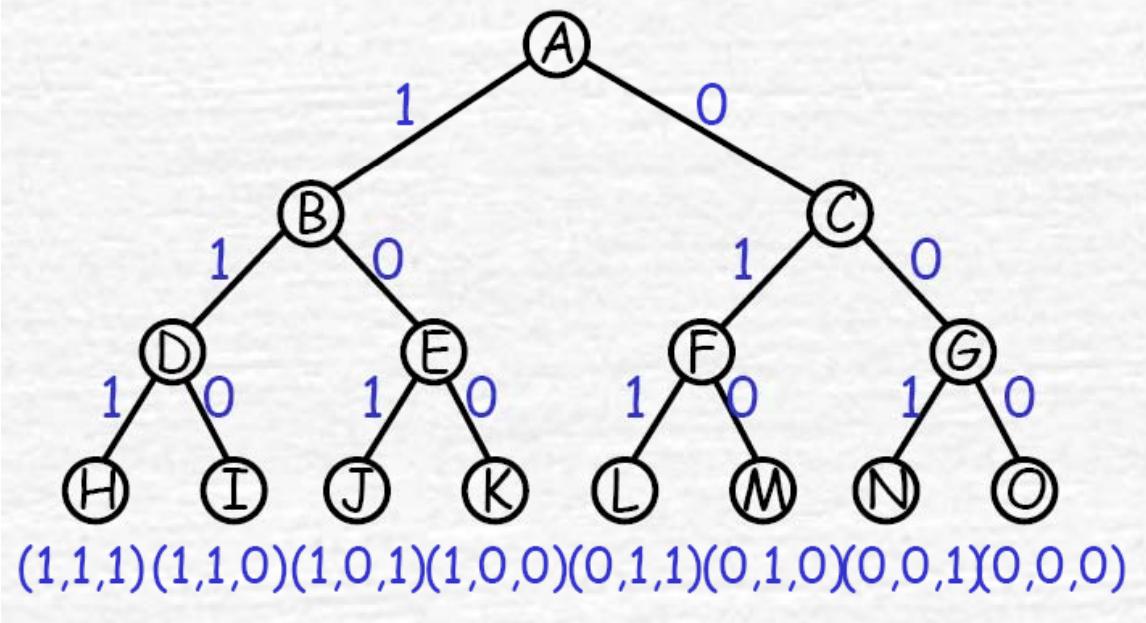
② **排列树**：当所给问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有n!个叶子结点，因此，遍历排列树需要$\Omega(n!)$的计算时间。如TSP问题，叶结点数为n!，遍历时间为$\Omega(n!)$。

1 方法概述

例1 [0-1背包]：n=3, w=(16, 15, 15), v=(45, 25, 25), c=30

(1) 定义解空间：X={(0,0,0), (0,0,1), (0,1,0),…, (1,1,0), (1, 1, 1)}

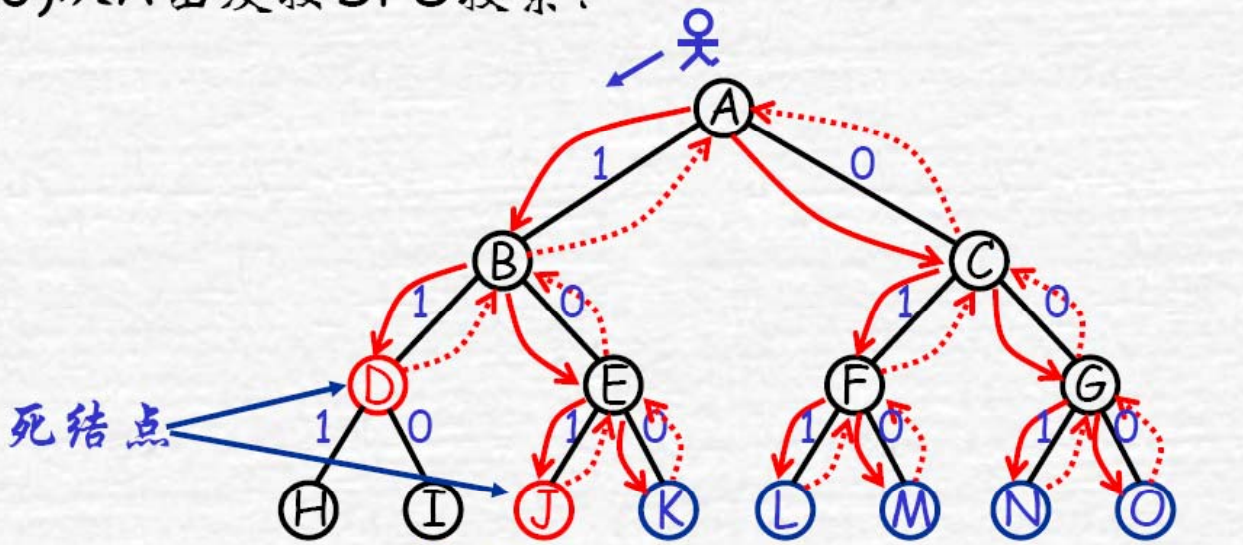
(2)构造解空间树：



1 方法概述

- 例1(cont.)：n=3, w=(16,15,15), v=(45,25,25), c=30

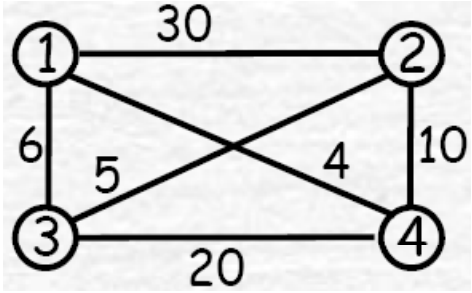
(3)从A出发按DFS搜索：



最优解：L=(0,1,1)，最优值为 50

1 方法概述

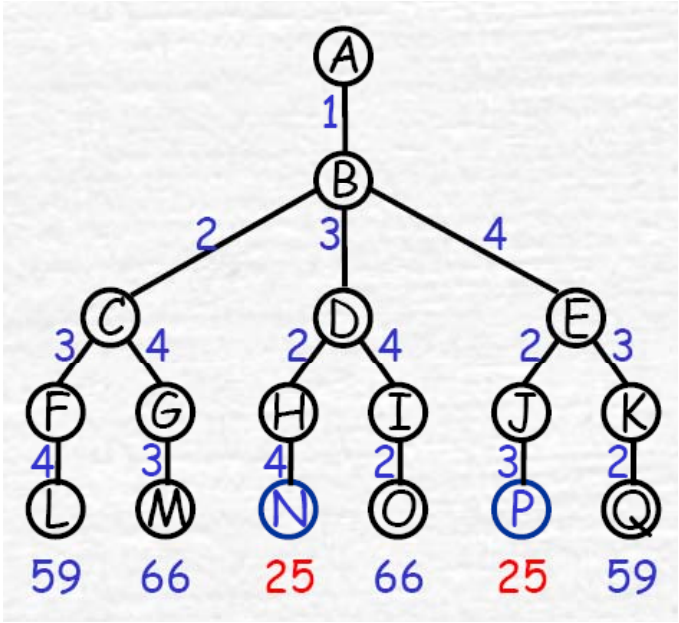
- 例2 [TSP问题]:



(1) 定义解空间：X={12341, 12431, 13241, 13421, 14231, 14321}

(2) 构造解空间树：

(3) 从A出发按DFS搜索整棵树：



最优解：13241，14231

成本：25

1 方法概述

- 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空间通常为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。

2 算法框架

- 回溯法对解空间作深度优先搜索，因此在一般情况下可用递归函数来实现回溯法：

- 子集树回溯算法

```
Backtrack( int t)   //搜索到树的第t层
{ //由第t层向第t＋1层扩展，确定x[t]的值
    if  t>n then output(x);  //叶子结点是可行解
    else
        while( all Xt) do          //Xt为当前扩展结点的所有可能取值集合
        {
            x[t] = Xt中的第i个值；
            if( Constraint(t) and Bound(t) )
                Backtrack( t+1 );
        }
}
```

执行时，从Backtrack(1)开始。

2 算法框架

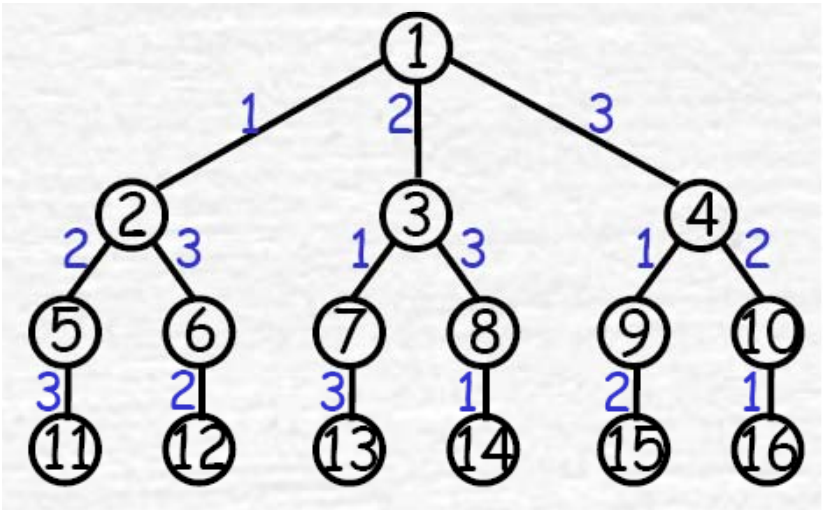
- **排列树回溯法**

```
Backtrack( int t)    //搜索到树的第t层
{   //由第t层向第t+1层扩展，确定x[t]的值
    if  t>n then output(x);  //叶子结点是可行解
    else
        for i=t to n    do
        {
            swap( x[t], x[i]);
            if( Constraint(t) and Bound(t) )
                Backtrack( t+1);
            swap(x[t], x[i]);
        }
}
```

3 排列生成问题

- 问题定义：给定正整数n，生成1, 2, …, n所有排列。
- 解空间树（排列树）：

  当n=3时，



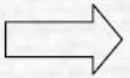
3 排列生成问题

- 回溯算法

```
Backtrack(int t)
{
   if t>n then output(x);
   else
      for i=t to n do
      {   swap(x[t], x[i]);
          Backtrack(t+1);
          swap(x[t], x[i]);
      }
}
main(int n)
{
    for i=1 to n do  x[i]=i;
    Backtrack(1);
}
```

- 对n=3的执行情况验证：

```
– Backtrack(1)
  Backtrack(2)
  x[1]↔x[2]
  Backtrack(2)
  x[1]↔x[2]
  x[1]↔x[3]
  Backtrack(2)
  x[1]↔x[3]
– Backtrack(2)
  Backtrack(3)
  x[2]↔x[3]
  Backtrack(3)
  x[2]↔x[3]
– Backtrack(3)
  Backtrack(4)
– Backtrack(4)
  output(x)
```



```
123
132
213
231
321
312
```

4 TSP问题

- 问题描述：略
- 基本思想：利用排列生成问题的回溯算法Backtrack(2)，对x[]={1, 2, …, n}的x[2..n]进行全排列，则(x[1], x[2])，(x[2], x[3])，…，(x[n], x[1])构成一个回路。在全排列算法的基础上，进行路径计算保存以及进行限界剪枝。
-
```
main( int n )
{
    a[n][n]; x[n] = {1,2,…,n}; bestx[]; cc=0.0;
    bestv = ∞;    //bestx保存当前最佳路径，bestv保存当前最优值
    input( a );    //输入邻接矩阵
    TSPBacktrack(2);
     output( bestv, bestx[] );
}
```

4 TSP问题

```
TSPBacktrack( int i)
{   // cc记录(x[1],x[2]),…, (x[i-1],x[i])的距离和
    if (i>n){  //搜索到叶结点，输出可行解与当前最优解比较
        if ( cc + a[x[n]][1] < bestv or bestv = ∞ ) {
            bestv = cc + a[x[n]][1];
            for( j=1 ; j <= n; j++)   bestx[j] = x[j];
        }
    }
    else{
        for( j = i ; j<=n; j++ )
            if ( cc + a[x[i-1]][x[j]] < bestv or bestv = ∞ ){  //限界裁剪子树
                swap( x[i], x[j]);
                cc += a[x[i-1]][x[i]];
                TSPBacktrack( i+1);
                cc -= a[x[i-1]][x[i]];
                swap(x[i],x[j]);
            }
    }
}
```

5 n皇后问题

- **问题描述：**

  在4×4棋盘上放上4个皇后，使皇后彼此不受攻击。不受攻击的条件是彼此不在同行（列）、斜线上。求出全部的放法。

- **解表示：**

– 解编码：(x_1, x_2, x_3, x_4)4元组，x_i表示皇后i放在i行上的列号，如(3,1,2,4)

– 解空间：$\{(x_1, x_2, x_3, x_4) \mid x_i \in S, i=1\sim4\}$ $S=\{1,2,3,4\}$

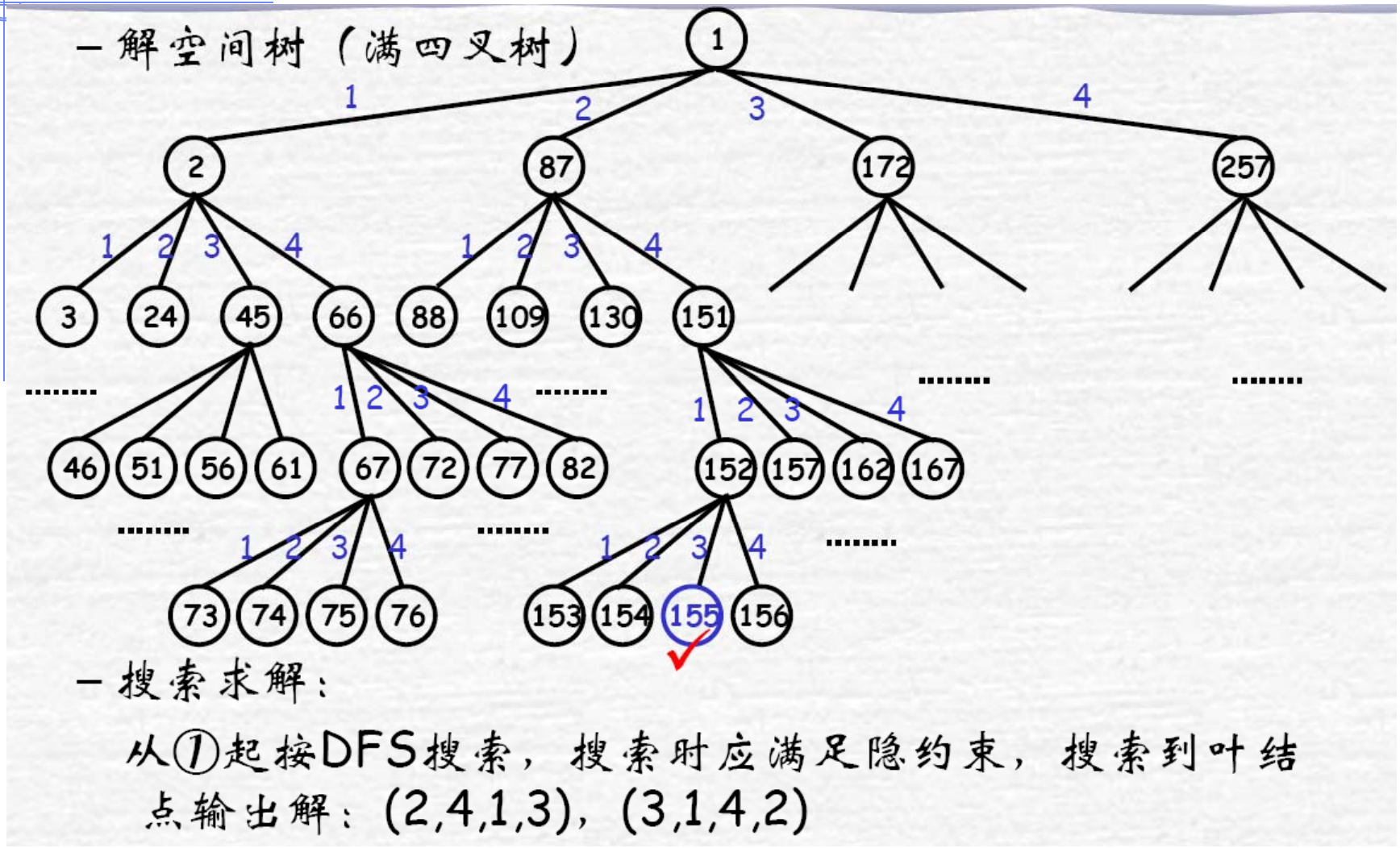
可行解满足：

显约束：$x_i \in S$, $i=1\sim4$

隐约束$(i \neq j)$：

$$\begin{cases} x_i \neq x_j & \text{（不在同一列）} \\ |x_i - x_j| \neq |i-j| & \text{（不在同一斜线）} \end{cases}$$

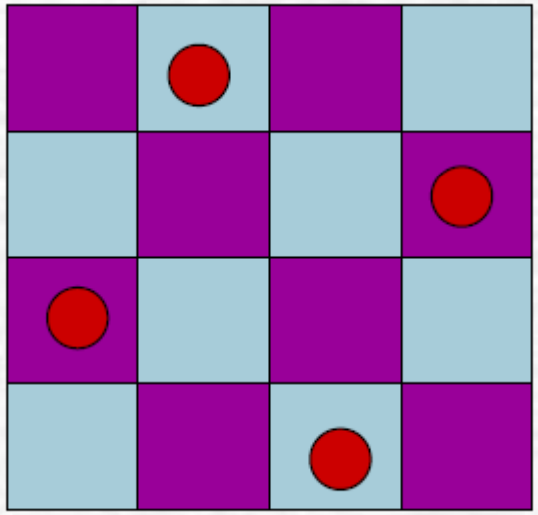
5 n皇后问题


一解空间树（满四叉树）
1
2
87
172
257
3
24
45
66
88
109
130
151
46
51
56
61
67
72
77
82
152
157
162
167
73
74
75
76
153
154
155
156
一搜索求解：
从①起按DFS搜索，搜索时应满足隐约束，搜索到叶结点输出解：(2,4,1,3)，(3,1,4,2)

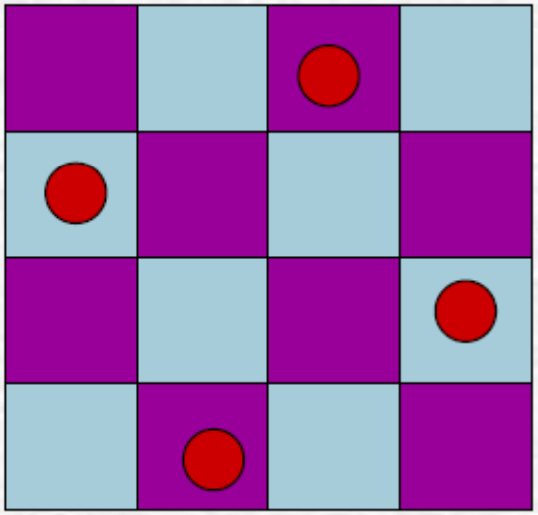
5 n皇后问题

输出解：

(2,4,1,3)



(3,1,4,2)



5 n皇后问题

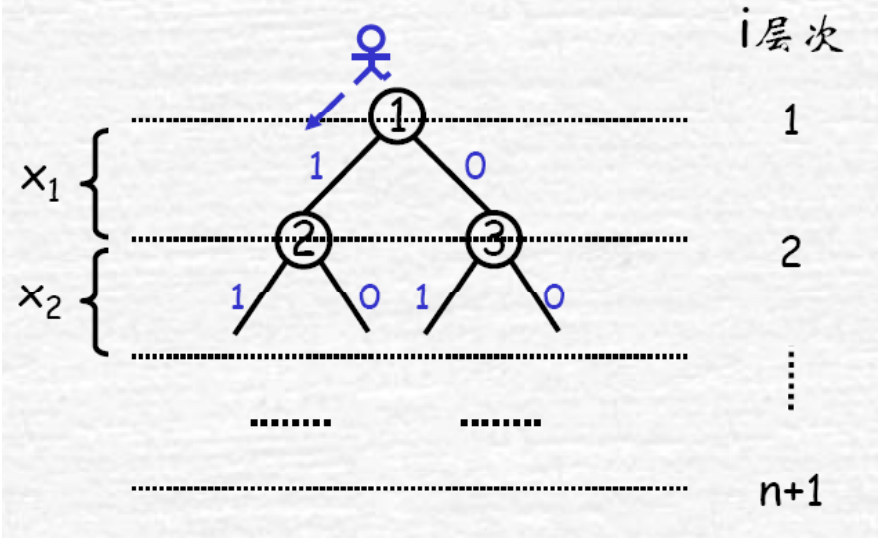
- 递归算法

```
NQueen(int k)
{//由第K层向第K+1层扩展，确定x[k]的值
    if k>n then printf(x[1], … ,x[n]);    //搜索到叶结点输出解
    else
       for i=1 to n do
       {   x[k]=i;
           if placetest(k) then NQueen(k+1);
        }
 }
Placetest(int k)
{//检查x[k]位置是否合法
    for i=1 to k-1 do
        if ( x[i]=x[k] or abs(x[i]-x[k])=abs(i-k) ) then return false;
    return true;
 }
```

注：求解时执行NQueen(1)

6 0-1背包问题

- 问题描述：略
- 解表示和解空间：$\{(x_1,x_2,\cdots,x_n) \mid x_i \in \{0,1\},\ i=1\sim n\}$
- 解空间树：



6 0-1背包问题

- 无限界函数的算法

```
KnapBacktrack( int i )
{  //cw当前背包重量，cv当前背包价值，bestv当前最优价值
    if( i > n) {  //搜索到可行解
        bestv = ( bestv < cv )? cv : bestv;
        output( x );
    }
    else{
        if ( cw + w[i] <=c ) { //走左子树
            x[i]= 1; cw+=w[i]; cv += v[i];
            KnapBacktrack( i+1);
            cw -= w[i];  cv -= v[i];
        }
        //以下走右子树
        x[i] = 0;
        KnapBacktrack( i+1);
    }
}
```

```
main(float c, int n, float w[], float v[], int x[])
{  //主程序
    float cw=0.0, cv = 0.0, bestv = 0.0;
    KnapBacktrack(1);
}
```

6 0-1 背包问题

- **有限界函数的算法**

—基本思想：

- 设r是当前扩展结点z的右子树（或左子树）的价值上界，如果cv+r <= bestv时，则可以裁剪掉右子树（或左子树）。
- 一种简单的确定z的左、右子树最优值上界的方法（设z为第k层结点）：

$$\text{左子树上界} = \sum_{i=k}^{n} v_i,\quad \text{右子树上界} = \sum_{i=k+1}^{n} v_i$$

— 求经扩展结点z的可行解价值上界的方法：

- 计算至扩展结点的当前背包价值

  已知x_i，i=1~k-1，当前背包价值$cv = \sum_{i=1}^{k-1} v_i x_i$
- 最后，

  经z左子树的可行解价值上界=cv + 左子树上界

  经z右子树的可行解价值上界= cv + 右子树上界

— 算法（略）

回溯法效率分析

通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

(1)产生x[t]的时间；

(2)满足显约束的x[t]值的个数；

(3)计算约束函数constraint的时间；

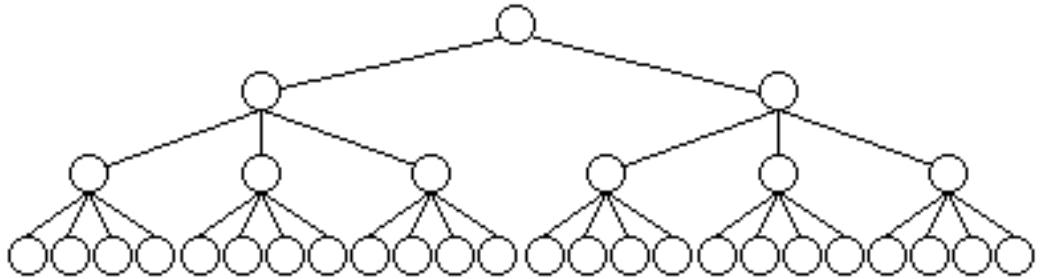
(4)计算上界函数bound的时间；

(5)满足约束函数和上界函数约束的所有x[k]的个数。

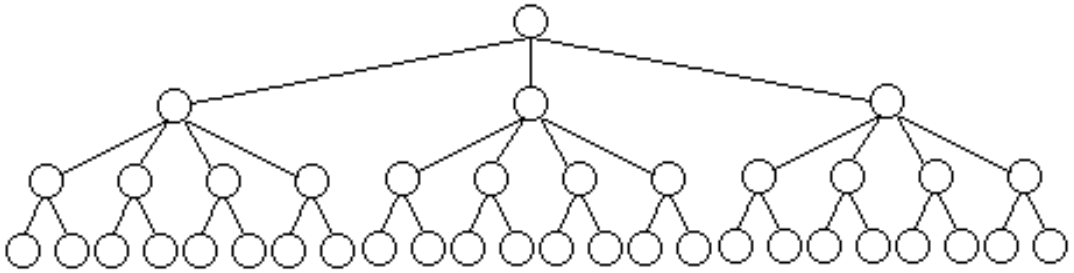
好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。

回溯法效率分析

- 对于许多问题而言，在搜索试探时选取x[i]的值顺序是任意的。**在其它条件相当的前提下，让可取值最少的x[i]优先**。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



(a)



(b)

图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

Branch & Bound Algorithm

- 理解分支限界法的剪枝搜索策略
- 掌握分支限界法的算法框架

  （1）队列式(FIFO)分支限界法

  （2）优先队列式分支限界法
- 通过应用范例学习分支限界法的设计策略

1 方法概述

基本思想：

- 分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树，裁剪那些不能得到最优解的子树以提高搜索效率。
- 搜索策略是：在扩展结点处，先生成其所有的儿子结点（分支），然后再从当前的活结点表中选择下一个扩展结点。为了有效地选择下一个扩展结点，以加速搜索的进程，在每一活结点处，计算一个函数值（优先值），并根据这些已计算出的函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。

1 方法概述

求解步骤：

- 定义解空间（对解编码）；
- 确定解空间的树结构；
- 按BFS等方式搜索：

  a. 每个活结点仅有一次机会变成扩展结点；

  b. 由扩展结点生成一步可达的新结点；

  c. 在新结点中，删除不可能导出最优解的结点；//限界策略

  d. 将剩余的新结点加入活动表（队列）中；

  e. 从活动表中选择结点再扩展； //分支策略

  f. 直至活动表为空；

1 方法概述

常见的两种分支限界法

- 队列式（FIFO）分支限界法：从活结点表中取出结点的顺序与加入结点的顺序相同，因此活结点表的性质与队列相同；
- 优先队列（代价最小或效益最大）分支限界法：每个结点都有一个对应的耗费或收益，以此决定结点的优先级：

— 如果查找一个具有最小耗费的解，则活结点可用小根堆来建立，下一个扩展结点就是具有最小耗费的活结点；

— 如果希望搜索一个具有最大收益的解，则可用大根堆来构造活结点表，下一个扩展结点是具有最大收益的活结点。

1 方法概述

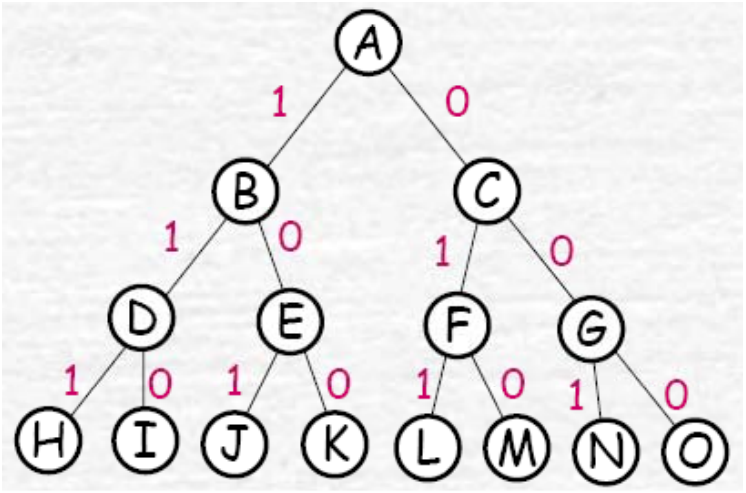
- 例子：【0-1背包问题】

  物品数量n=3，重量w=(20,15,15)，价值v=(40,25,25)，背包容量c=30，试装入价值和最大的物品？

- FIFO队列分支限界法求解：

① 解空间：{(0,0,0),(0,0,1),…,(1,1,1)}

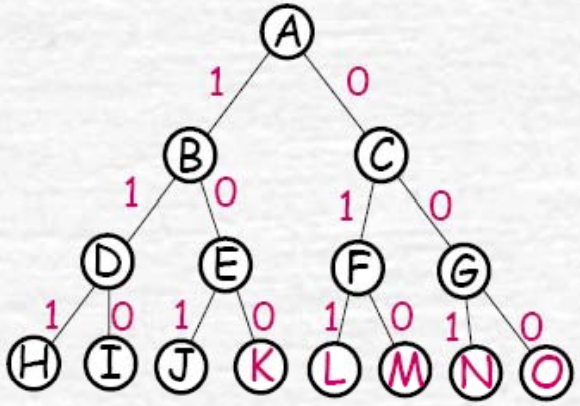
② 解空间树：



1 方法概述

③ BFS搜索（FIFO队列）

扩展结点	活结点	队列(可行结点)	可行解(叶结点)	解值
A	B,C	BC		
B	D,E(D死结点)	CE		
C	F,G	EFG		
E	J,K(J死结点)	FG	K	40
F	L,M	G	L,M	50,25
G	N,O	ϕ	N,O	25,0



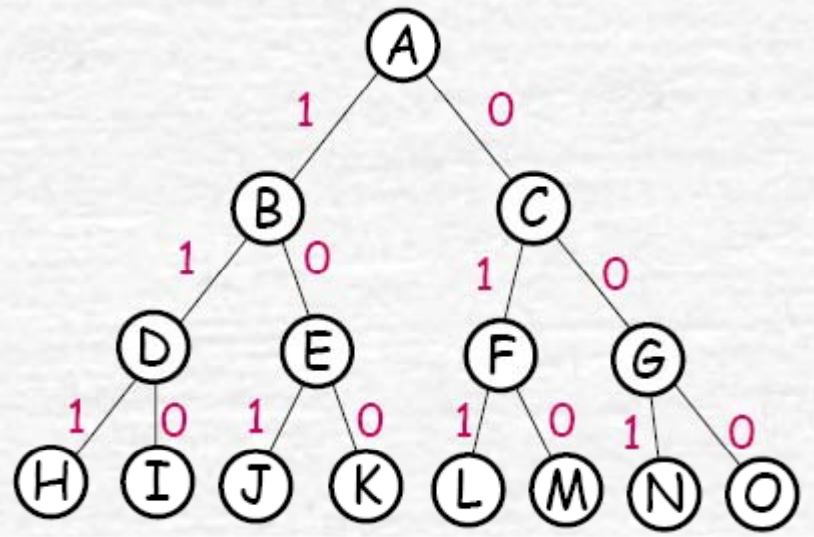
∴ 最优解为L，即(0,1,1)；解值为50

1 方法概述

- 优先队列分支限界法求解：

① 解空间：{(0,0,0),(0,0,1),…,(1,1,1)}

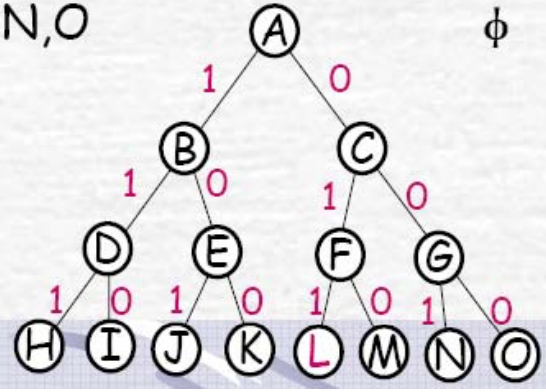
② 解空间树：



1 方法概述

- BFS搜索（优先队列：按照价值率优先）

扩展结点	活结点	队列(可行结点)	可行解(叶结点)	解值
A	B,C	B C		
B	D,E(D死结点)	E C		
E	J,K(J死结点)	C	K	40
C	F,G	F G		
F	L,M	G	L	50(最优)
G	N,O	ϕ		



Code 0-1背包问题

- **算法思想：**

首先，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。

在下面描述的优先队列分支限界法中，节点的优先级由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。如果该左儿子结点是可行结点，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。当扩展到叶节点时为问题的最优值。

Code 0-1背包问题

```
MaxKnapsack( n, c, w[ ], p[ ] )
{   //优先队列式分支限界法，返回最大价值，n为物品数目，c为背包容量，w为物品重量，p为物品价值
    //算法开始之前，已经按照物品单位价值率按照降序顺序排列好了
    cw = 0, cp = 0;   //cw为当前装包重量，cp为当前装包价值
    bestp = 0;  //当前最优值
    i=1,  up = Bound(1) ;  //函数Bound(i)计算当前结点相应的价值上界
    while( i != n+1 ) { //非叶子结点
        //首先检查当前扩展结点的左儿子结点为可行结点
        if( cw + w[i] <= c ) { //左孩子结点为可行结点
            if( cp + p[i] > bestp )  bestp = cp + p[i];
            AddLiveNode( up, cp + p[i] + cw + w[i], true, i + 1);  //将左孩子结点插入到优先队列中
        }
        up = Bound( i+1);
        //检查当前扩展结点的右儿子结点
        if( up >= bestp )  //右子树可能包含最优解
             AddLiveNode( up, cp, cw, false , i + 1);   //将右孩子结点插入到优先队列中
        //从优先级队列（堆数据结构）中取下一个扩展结点N
        H->DeleteMax( N) ;
         i = N.level;
    }
}
```

分支限界搜索过程

Code 0-1背包问题

```
Bound( i )
{  //计算结点所对应的价值的上界
    cleft = c  -  cw;       //剩余背包容量
    b = cp;                 //价值上界
    //以物品单位重量价值递减顺序装填剩余容量
    while( i <= n  &&  w[i] <= cleft ){
          cleft -= w[i];            //w[i]表示i物品的重量
          b += p[i];                //p[i]表示i物品的价值
          i++;
    }
    //装填剩余容量装满背包
    if( i<=n) b += p[i]/w[i] * cleft;
    return b;
}
```

0-1 Knapsack Example

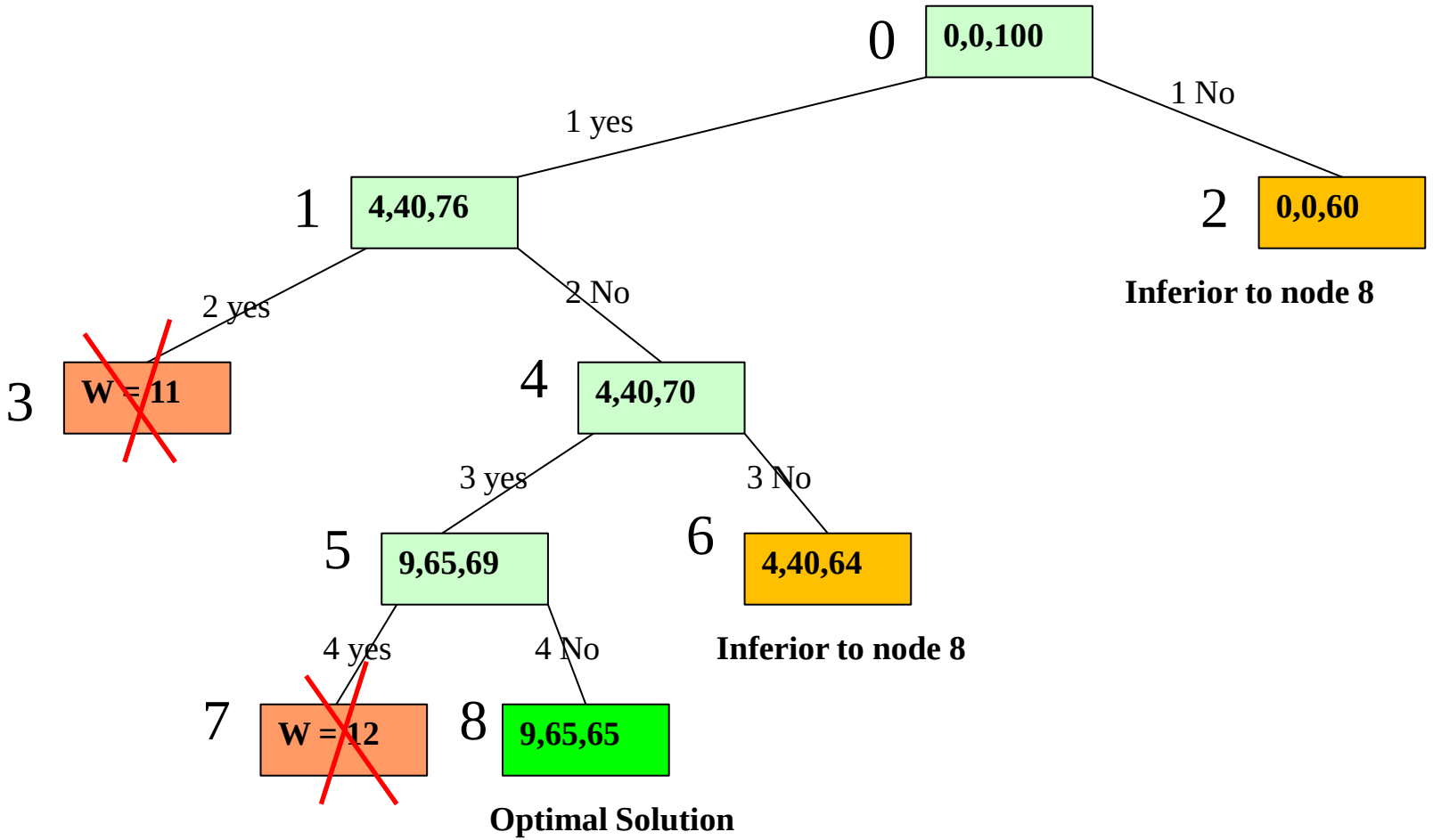
W = 10, n = 4

Item	Weight	Value	Value Density
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

1. Item (i) has a weight w_i and a value v_i, the value densit y is $d_i = v_i / w_i$
2. Arrange the items according to the value density, $d_1 \geq d_2 \geq \ldots \geq d_n$

0-1 Knapsack Example

B & B State Space: Numbers in each node are for **W, V, UB**



0-1 Knapsack Example

1. Level (0):

   Node (0) w = 0 v = 0 d_{best} = 10

   ub = 0 + (**W** - 0)*10 = 100

2. Level (1):
   - Node (1) left: with item (1) w = 4 v = 40 ub = 40 + (**W** - 4)*6 = 76
   - Node (2) right: without (1) w = 0 v = 0 ub = 0 + (**W** – 0)*6 = 60
   - Node (1) is more promising, Branch from that node.
3. Level (2):
   - Node (3) left: with item (2) w = 4 + 7 = 11 Not feasible
   - Node (4) right: without (2) w = 4 v = 40 ub = 40 + (**W** – 4)*5 = 70

0-1 Knapsack Example

4. Level (3): Branching from node (4)
   - Node (5) left: with item (3) w = 9 v = 65 ub = 65 + (**W** – 9)* 4 = 69
   - Node (6) right: without (3) w = 4 v = 40 ub = 40 + (**W** – 4) *4 = 64
   - Node (5) is more promising, Branch from that node
5. Level (4): Branching from node (5)
   - Node (7) left: with item (4) w = 9 + 3 = 12 Not feasible
   - Node (8) right: without (4) w = 9 v = 65 ub = 65
   - Node (8) is the optimal solution
6. The final solution is:

   (item 1 + item 3): Total weight is 9

   Total value is 65

0-1 Knapsack Example Hints

Procedure:

1. Consider w = total weight of items selected so far,
   v = total value obtained so far.
2. Consider dbest = best value density among remaining items.
3. The upper bound on the value so far is computed as:
   $ub = v + (\mathbf{W} - w)\, d_{best}$
4. Branch to the next level from the more promising node (higher ub).

5. Terminate at the current node for one of the following reasons:
   ① The ub of node is not more promising than what is obtained so far.
   ② The node violates the constraint $w \leq \mathbf{W}$
   ③ No further choices.

2 装载问题

- **问题描述：**

有一批共个集装箱要装上2艘载重量分别为C1和C2的轮船，其中集装箱i的重量为Wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

容易证明：如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1)首先将第一艘轮船尽可能装满；

(2)将剩余的集装箱装上第二艘轮船。

2 装载问题

- 队列式分支限界法

在算法的while循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则将其加入到活结点队列中。然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。2个儿子结点都产生后，当前扩展结点被舍弃。

活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部标记-1，故在取队首元素时，活结点队列一定不空。当取出的元素是-1时，再判断当前队列是否为空。如果队列非空，则将尾部标记-1加入活结点队列，算法开始处理下一层的活结点。

2 装载问题

```
while (true)
{
    if (ew + w[i] <= c)     enQueue( ew + w[i], i );      // 检查左儿子结点
    enQueue(ew, i);                                       //右儿子结点总是可行的
    ew = ((Integer) queue.remove()).intValue();           // 取下一扩展结点
    if (ew == -1)   //如果是本层的结尾结点
    {
        if (queue.isEmpty())    return bestw;
        queue.put(new Integer(-1));                       // 同层结点尾部标志
        ew = ((Integer) queue.remove()).intValue();       // 取下一扩展结点
        i ++;                                             // 进入下一层
     }
}
```

2 装载问题

- 算法改进

节点的左子树表示将此集装箱装上船，右子树表示不将此集装箱装上船。设bestw是当前最优解；ew是当前扩展结点所相应的重量；r是剩余集装箱的重量。则当ew+r≤bestw时，可将其右子树剪去，因为此时若要船装最多集装箱，就应该把此箱装上船。

另外，为了确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新bestw的值。

2 装载问题

```
// 检查左儿子结点
int wt = ew + w[i];
if (wt <= c)
{   // 可行结点
    if (wt > bestw) bestw = wt;
    // 加入活结点队列
    if (i < n)
        queue.put(new Integer(wt));
}
```

提前更新bestw

右儿子剪枝

```
// 检查右儿子结点
if (ew + r > bestw && i < n)
// 可能含最优解
    queue.put(new Integer(ew));
ew=((Integer)queue.remove()).intValue();
// 取下一扩展结点
```

2 装载问题

- **构造最优解**

为了在算法结束后能方便地构造出与最优值相应的最优解，算法必须存储相应子集树中从活结点到根结点的路径。为此目的，可在每个结点处设置指向其父结点的指针，并设置左、右儿子标志。找到最优值后，可以根据parent回溯到根节点，找到最优解。

```
private static class QNode
{   QNode parent;        //父结点
    boolean leftChild;   //左儿子标志
    int weight;      //结点所相应的载重量
```

```
 for (int j = n; j > 0; j--)
{
    bestx[j] = (e.leftChild) ? 1 : 0;
    e = e.parent;
}
```

2 装载问题

- **优先队列式分支限界法**

解装载问题的优先队列式分支限界法用最大优先队列存储活结点表。活结点x在优先队列中的优先级定义为从根结点到结点x的路径所相应的载重量再加上剩余集装箱的重量之和。

优先队列中优先级最大的活结点成为下一个扩展结点。以结点x为根的子树中所有结点相应的路径的载重量不超过它的优先级。子集树中叶结点所相应的载重量与其优先级相同。

在优先队列式分支限界法中，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法。

3 方法总结比较

B&B与Backtracking区别：

- **求解目标不同：**

  一般而言，回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标则是尽快地找出满足约束条件的一个解；

- **搜索方法不同：**

  回溯法使用深度优先方法搜索，而分支限界一般用宽度优先或最佳优先方法来搜索；

- **对扩展结点的扩展方式不同：**

  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；

- **存储空间的要求不同**

  分支限界法的存储空间比回溯法大得多，因此当内存容量有限时，回溯法成功的可能性更大。

回溯法与穷举法的区别联系

联系： 都是基于试探搜索方法的。

区别：

- 穷举法要将一个解的各个部分全部生成后，才检查是否满足条件，若不满足，则直接放弃该完整解，然后再尝试另一个可能的完整解，它并没有沿着一个可能的完整解的各个部分逐步回退生成解的过程。
- 回溯法，一个解的各个部分是逐步生成的，当发现当前生成的某部分不满足约束条件时，就放弃该步所做的工作，退到上一步进行新的尝试，而不是放弃整个解重来。

Exercise 1

Solve the following instance of the knapsack problem by the b ranch-and-bound algorithm: W = 16, n = 4

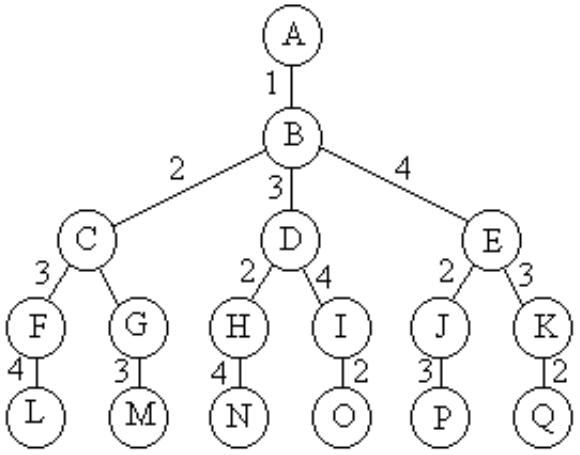
Item	Weight	Value	Value Density
1	10	100	10
2	7	63	9
3	8	56	7
4	4	12	3

Exercise 2

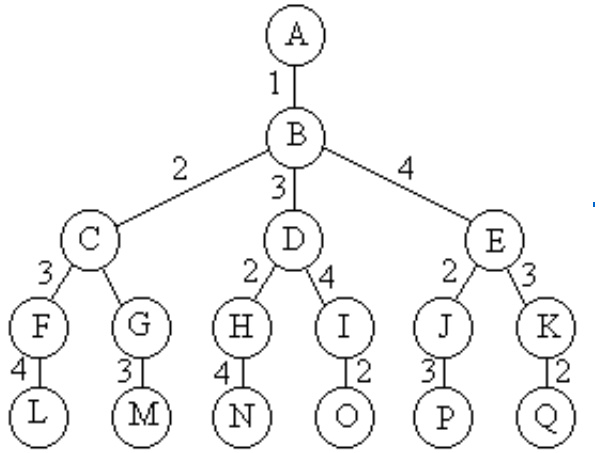
Writing down the code of the TSP problem by Branch & Bound Algorithm with either queue or priority queue. Some analysis about the implementation are provided here.

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 旅行售



- 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。

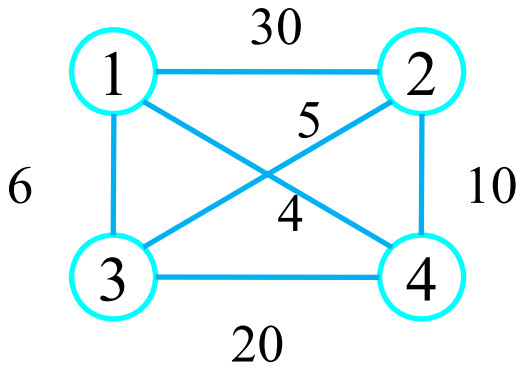
Branch-and-Bound 旅行售货员问题

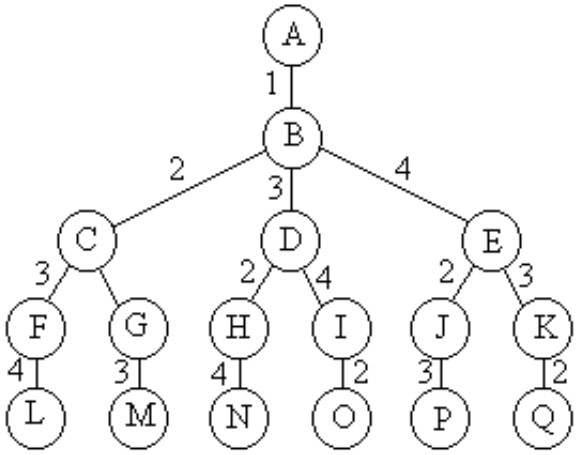
- 算法描述

  - 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的费用等于cc和lcost的值。剩余的活结点的lcost值不小于已找到的回路的费用。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。

Branch-and-Bound example TSP ..

- ***Example:*** 四城市旅行售货员问题





➢ 队列式分支限界法

B *C* *D* *E*

~~*B*~~ *C* *D* *E* *F* *G*

~~*B*~~ ~~*C*~~ *D* *E* *F* *G* *H* *I*

~~*B*~~ ~~*C*~~ ~~*D*~~ *E* *F* *G* *H* *I* *J* *K*

~~*B*~~ ~~*C*~~ ~~*D*~~ ~~*E*~~ *F* *G* *H* *I* *J* *K* *L*^{59}

~~*B*~~ ~~*C*~~ ~~*D*~~ ~~*E*~~ ~~*F*~~ *G* *H* *I* *J* *K* *L*^{59} *M*^{66}

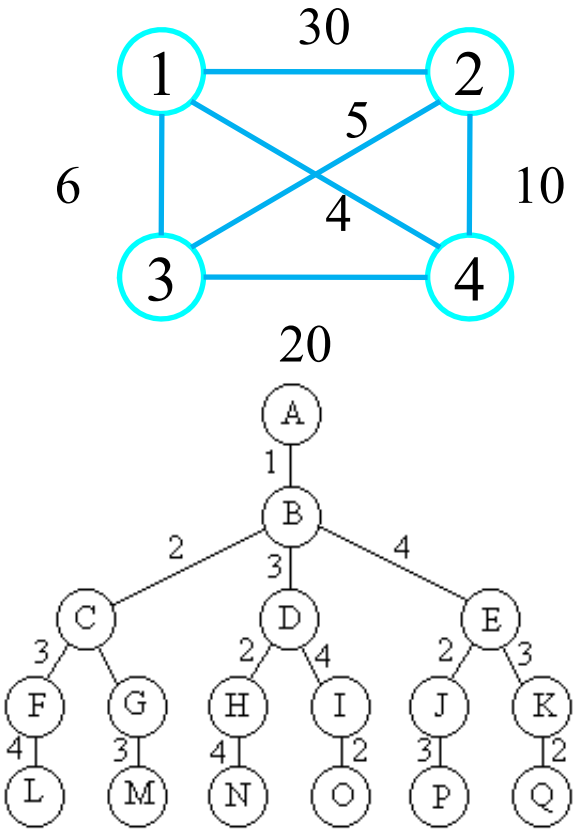
~~*B*~~ ~~*C*~~ ~~*D*~~ ~~*E*~~ ~~*F*~~ ~~*G*~~ *H* *I* *J* *K* *L*^{59} *M*^{66} *N*^{25}

~~*B*~~ ~~*C*~~ ~~*D*~~ ~~*E*~~ ~~*F*~~ ~~*G*~~ ~~*H*~~ ~~*I*^{26}~~ *J* *K* *L*^{59} *M*^{66} *N*^{25} *P*^{25}

~~*B*~~ ~~*C*~~ ~~*D*~~ ~~*E*~~ ~~*F*~~ ~~*G*~~ ~~*H*~~ ~~*I*^{26}~~ *J* *K* *L*^{59} *M*^{66} *N*^{25} *P*^{25} *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 $\underline{C\ D\ E^{4}}$

~~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} $\underline{N^{25}}$

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

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

Q & A

Thanks!