回溯与分支限界

Backtracking & Branch and Bound

刘 铎

liuduo@bjtu.edu.cn

回溯与分支限界

- 穷竭式搜索的改进
  - 穷竭式搜索的搜索空间可能很大（可参看例5.1~例5.3）
- 回溯法一般用以处理寻找有效解的问题
- 分支限界法一般用以处理最优化问题

回溯法

Backtracking

例 5.1

n 皇后问题

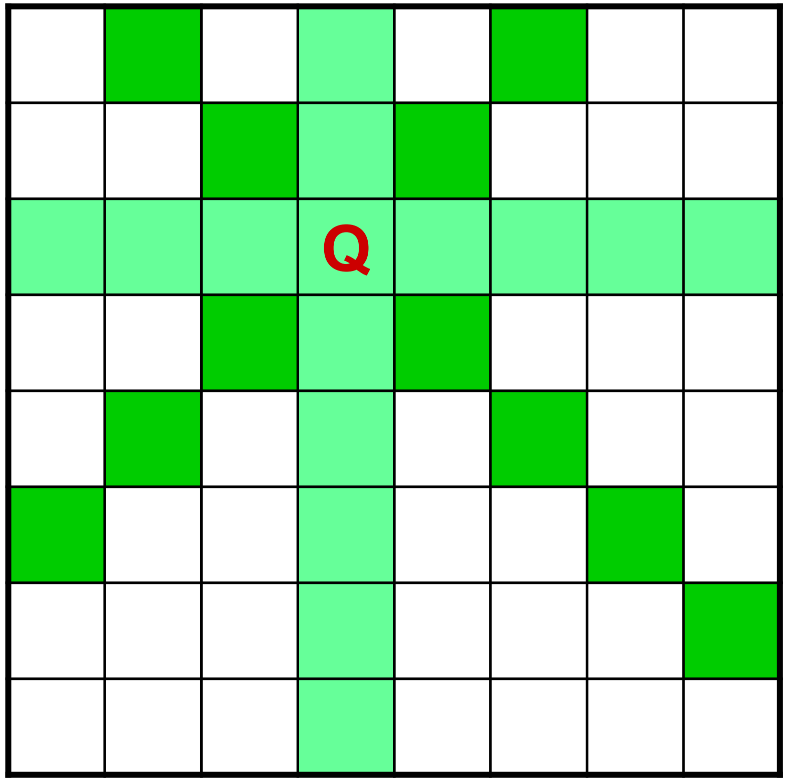
The n-Queen Problem

刘铎

liuduo@bjtu.edu.cn

n 皇后问题

- 将 n 个皇后放在一个 $n \times n$ 的棋盘上，使得没有两个皇后出现在同一行、同一列或同一与对角线平行的斜线上



n 皇后问题

- 将 n 个皇后放在一个 $n \times n$ 的棋盘上，使得没有两个皇后出现在同一行、同一列或同一与对角线平行的斜线上
- 仅当 $n = 1$ 或 $n \geq 4$ 时存在解

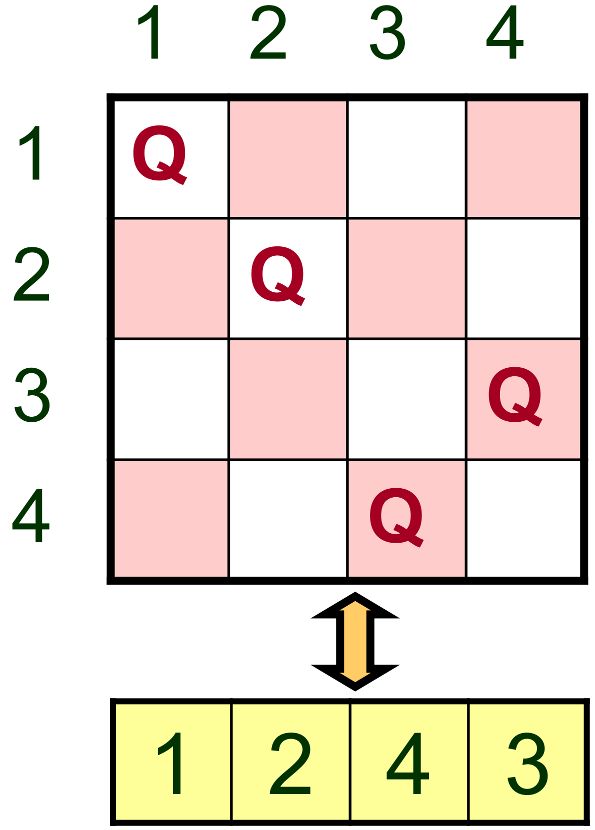
	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

n 皇后问题的历史

- 最初由国际象棋选手马克斯·贝泽尔（Max Bezzel）于1848年提出
- 弗兰兹·诺克（Franz Nauck）在1850年提出了第一个解
- 诺克还将其扩展到了 *n* 皇后问题
- 多年来，许多数学家——包括高斯和康托都致力于解决这一难题及其广义 *n* 皇后问题
- 1972年，Edsger Dijkstra用这个问题来表述他称之为结构化编程（structured programming）的威力。他发表了一篇关于深度优先回溯算法设计的非常详细的描述

第一个解法

- 不要将两个皇后放在同一行或同一列中
  - 生成 **1, 2, …,** ***n*** 的所有置换
  - 现在需要检查多少种位置配置？
  - ***n*!**
  - **8! = 40, 320**

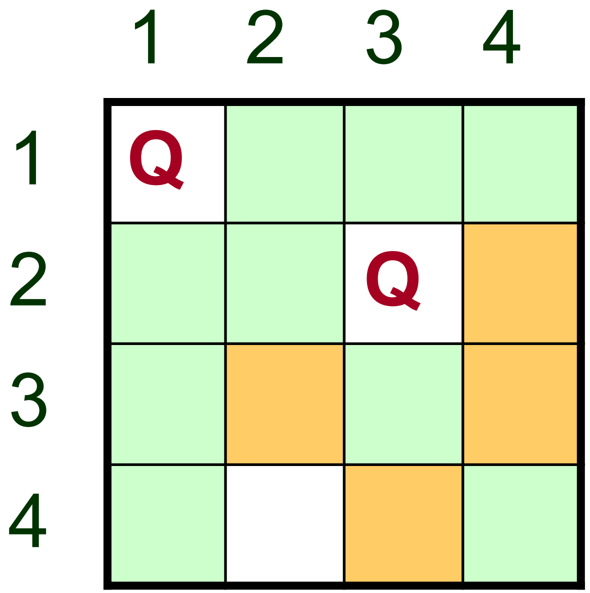


生成置换

- 枚举所有置换是很容易的
  - 给定一个置换，可以很容易确定待检查的“下一个”置换
- 只在放置了所有 n 个皇后后才检查解决方案
  - 需要使用一个判定函数（**criterion function** ）（或者解的检验测试方法）

4-皇后问题

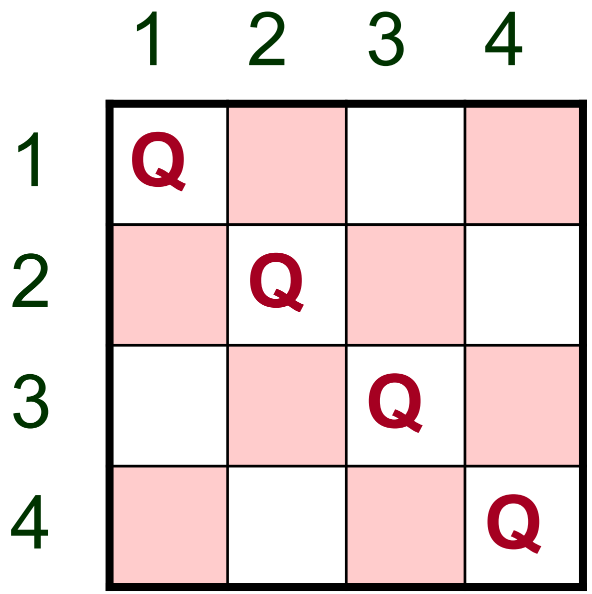
- 适合于教学过程



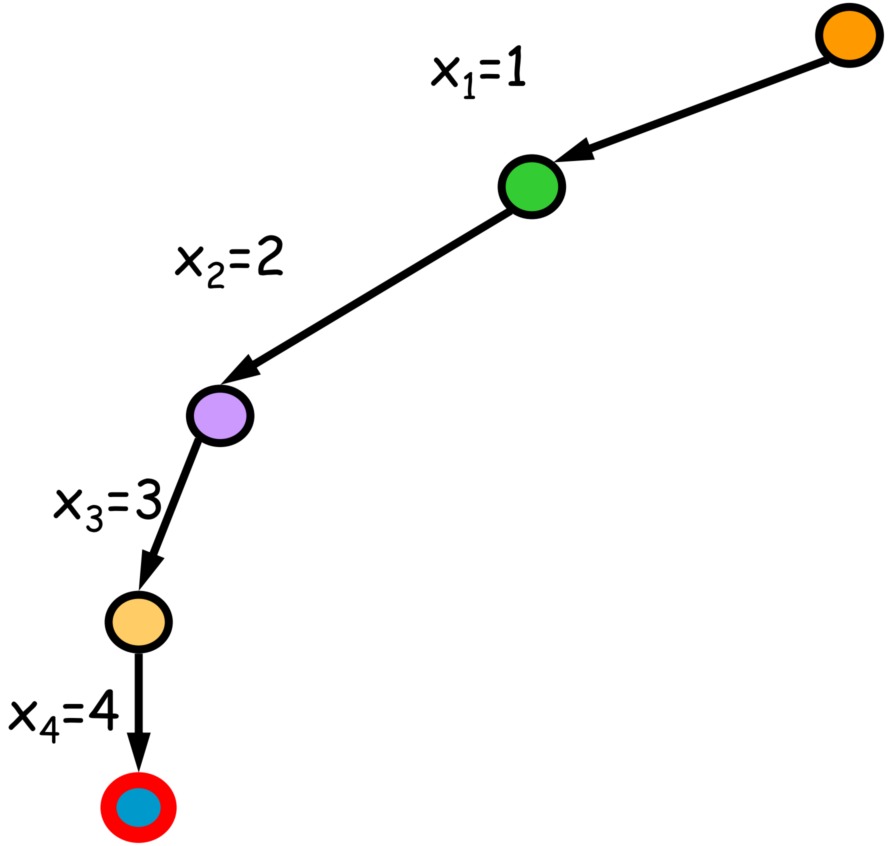
构造一个解

- 树的每一个分枝点都代表着一个放置皇后的决定
- 判定函数只能应用于叶子顶点

构造一个解



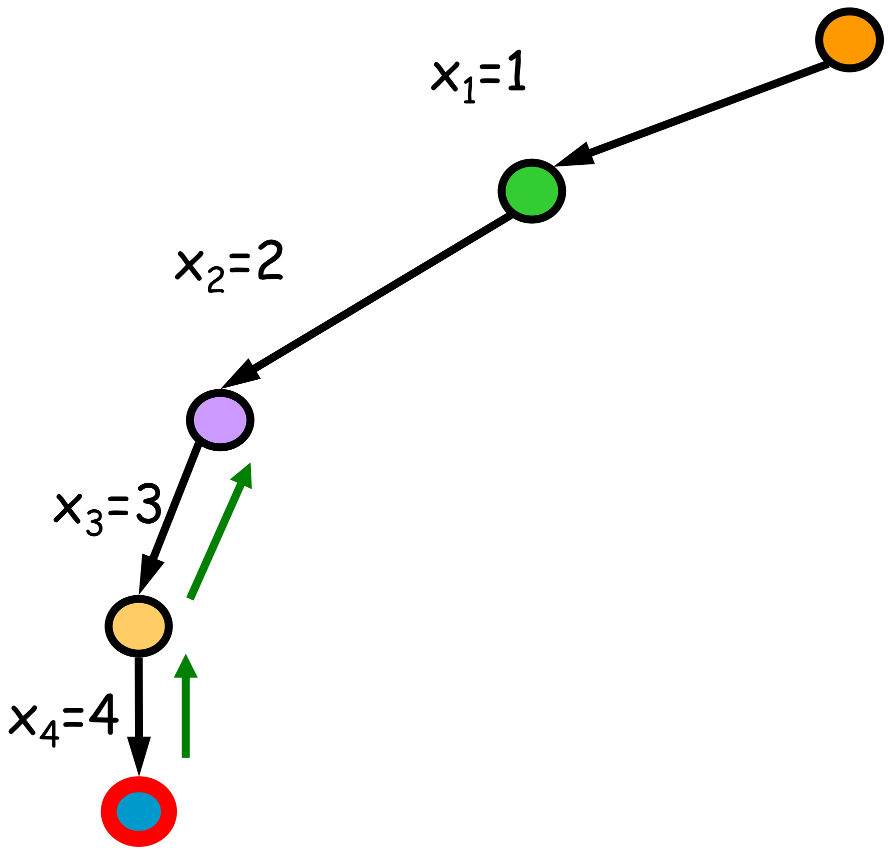
- 树的每一个分枝点都代表着一个放置皇后的决定
- 判定函数只能应用于叶子顶点



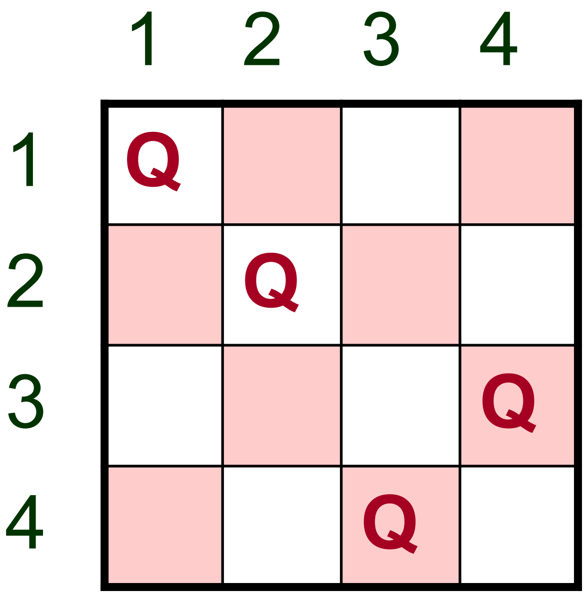
构造一个解

	1	2	3	4
1	Q			
2		Q		
3			Q	
4				Q

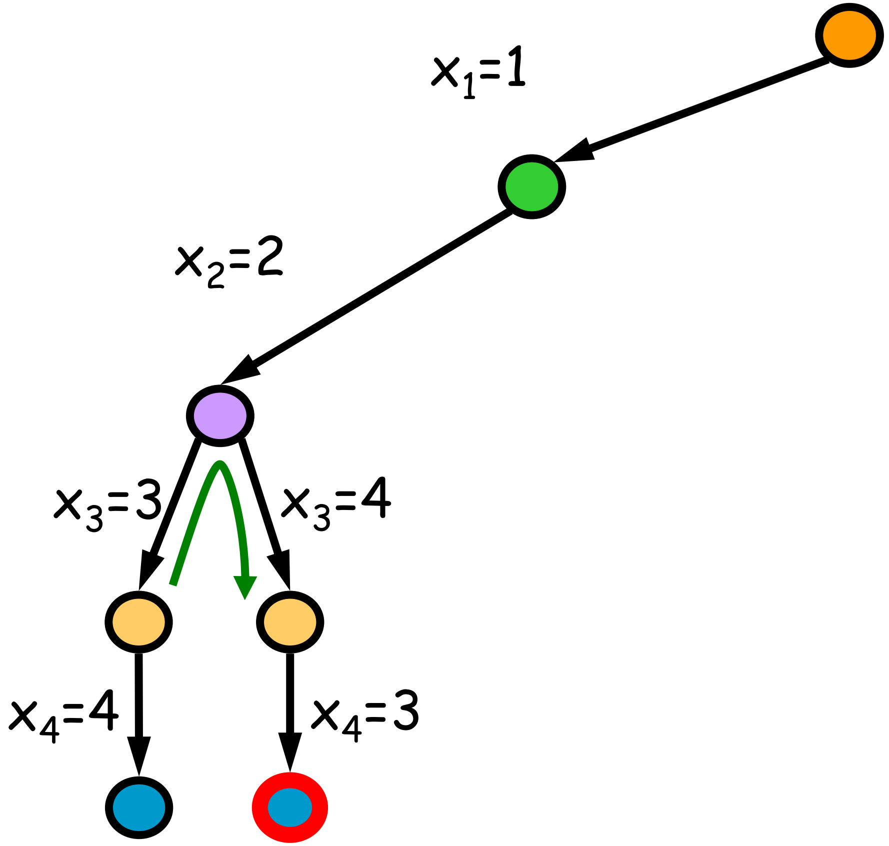
- 叶子顶点是否构成一个解？



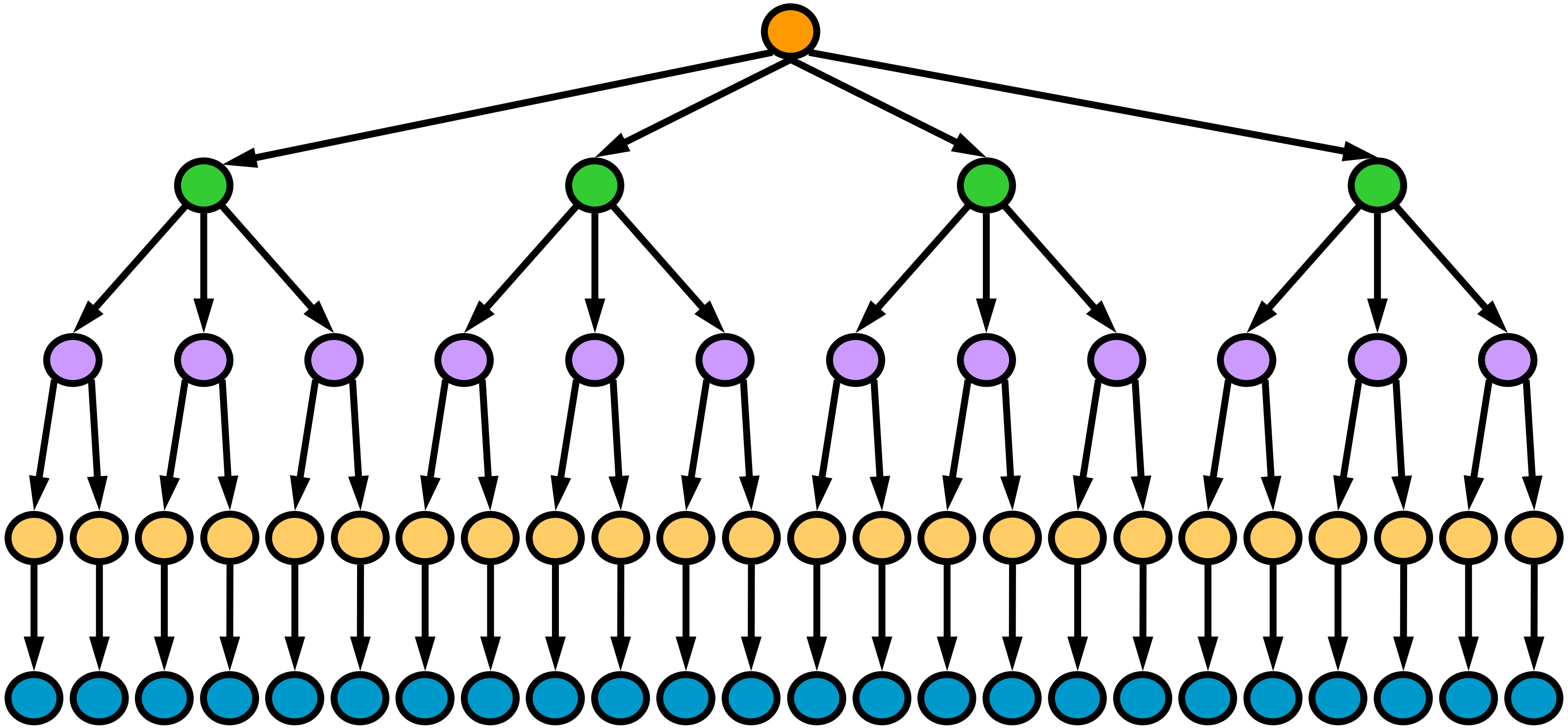
构造一个解



- 叶子顶点是否构成一个解？



4-皇后问题

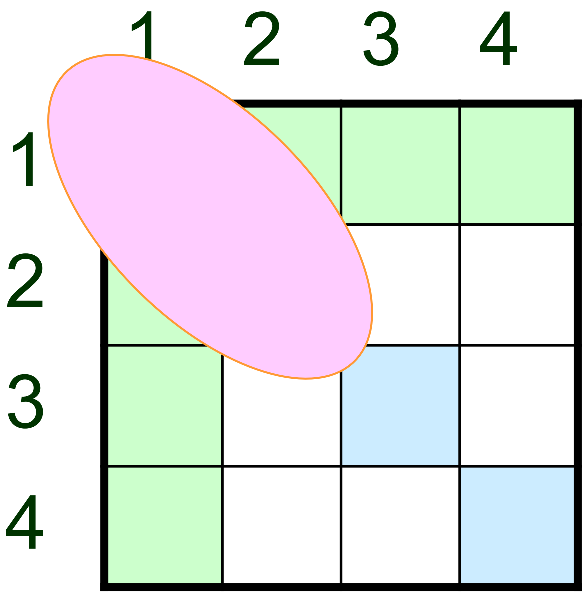


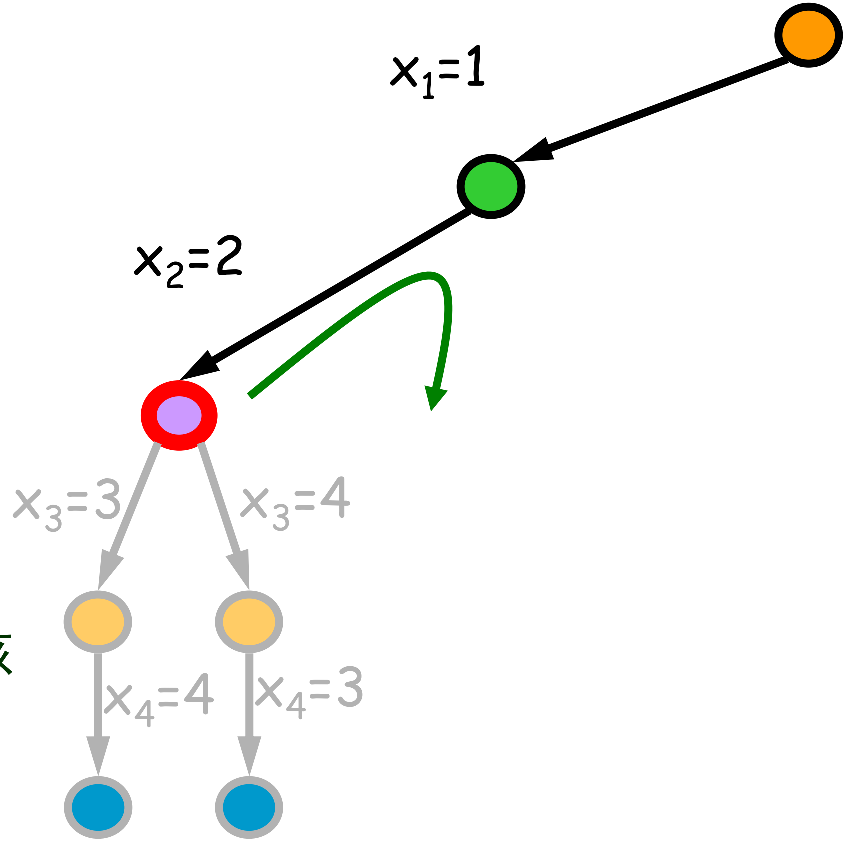
使用判定函数的搜索树（search tree）

n-皇后问题

- 一个更好的办法是在每次放置皇后后都进行检查
  - 部分判定函数（**partial-criterion function** ）或称可行性检查
  - 思想：在探索整条路之前就可以确定我们已经在一条死胡同上了
- 任何包含两个可以相互攻击到的皇后的部分解决方案都可以被放弃了，因为它不可能成为有效的解

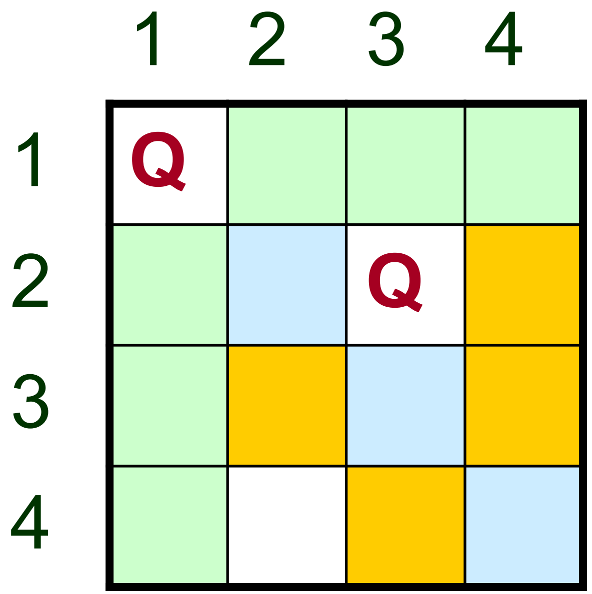
4-皇后问题

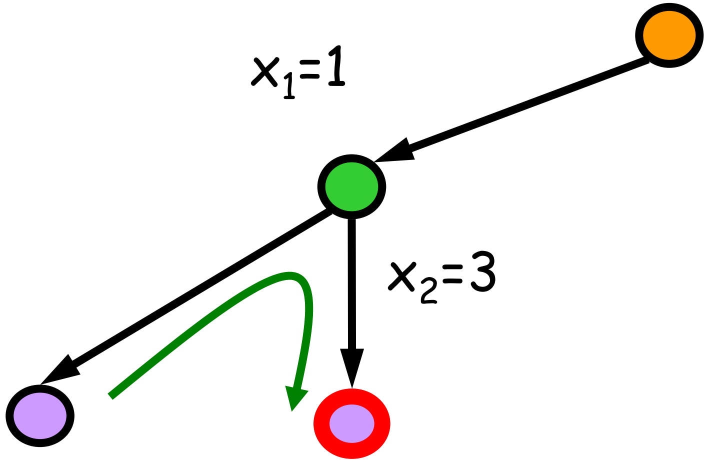




- 部分判定函数/可行性函数检查该顶点是否可能得到一个最终的解
- 这个顶点可能得到一个解么？无可能
- 退回，做其他尝试

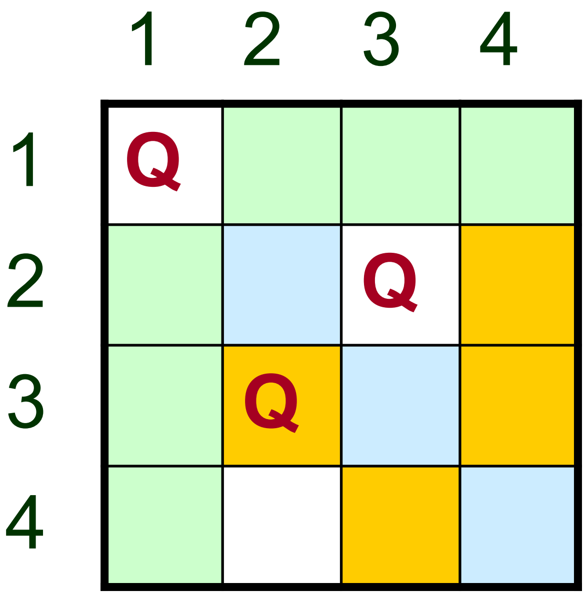
4-皇后问题

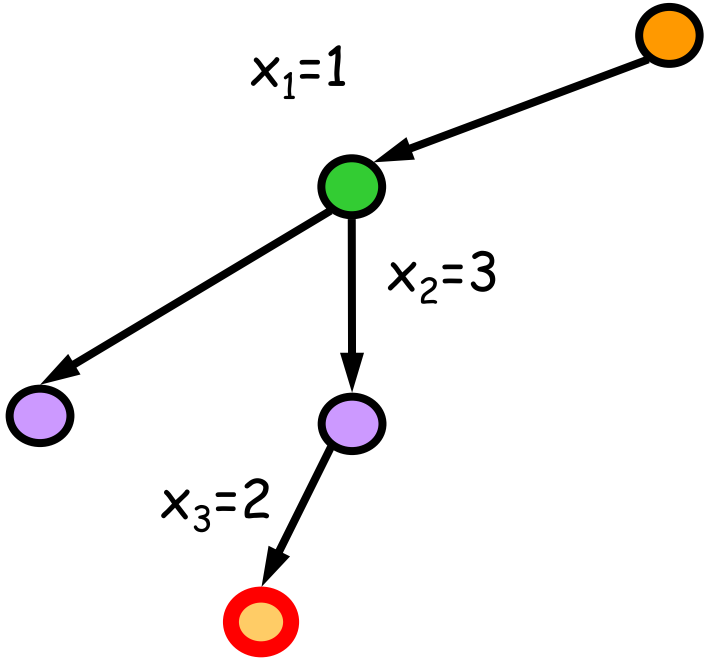




- 这个顶点可能得到一个解么？有可能

4-皇后问题

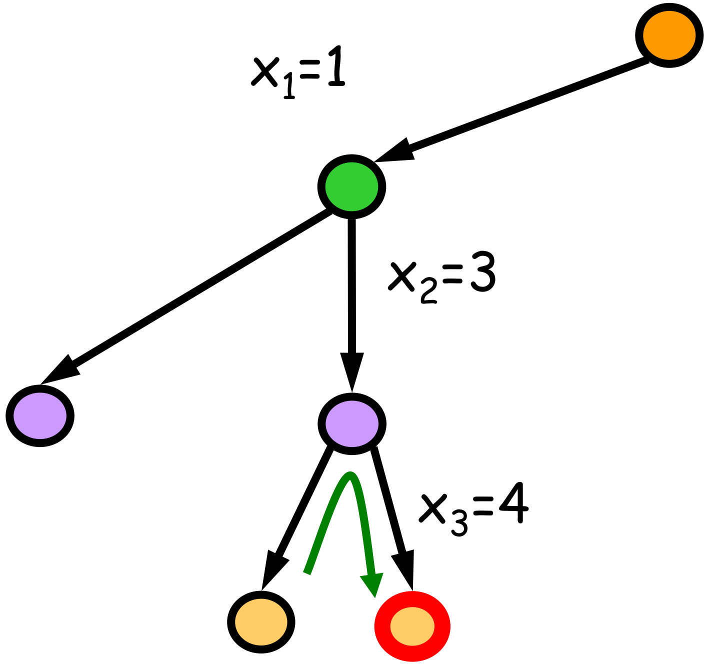




- 这个顶点可能得到一个解么？无可能
- 退回，做其他尝试

4-皇后问题

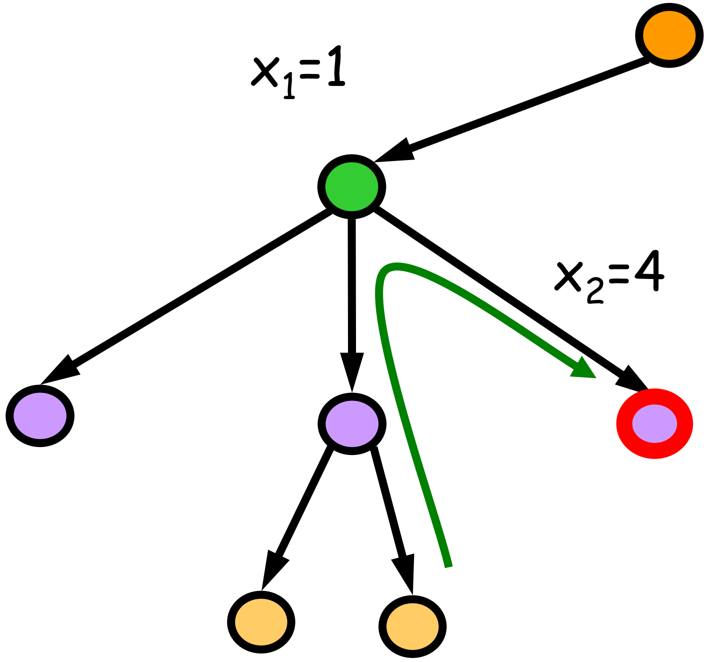
	1	2	3	4
1	Q			
2			Q	
3				Q
4				



- 这个顶点可能得到一个解么？无可能
- 退回，做其他尝试

4-皇后问题

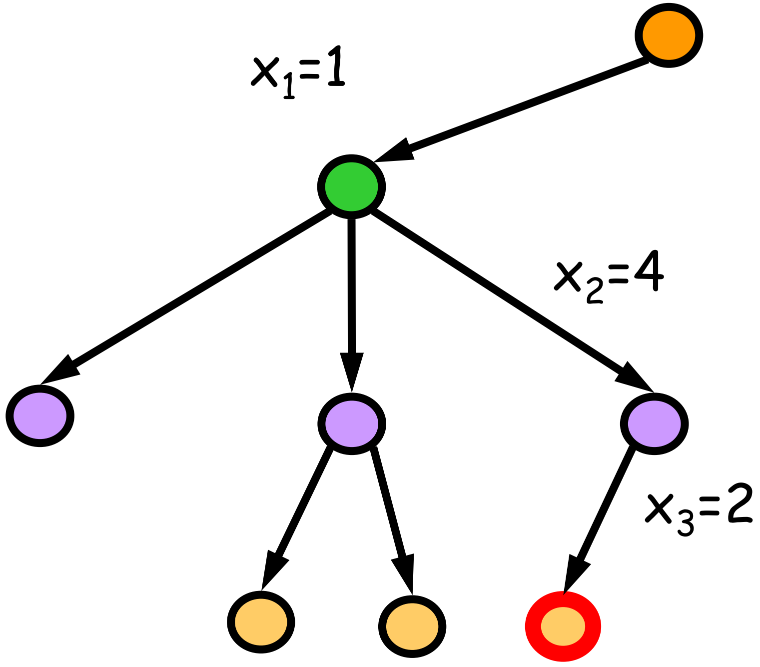
	1	2	3	4
1	Q			
2				Q
3				
4				



- 这个顶点可能得到一个解么？有可能

4-皇后问题

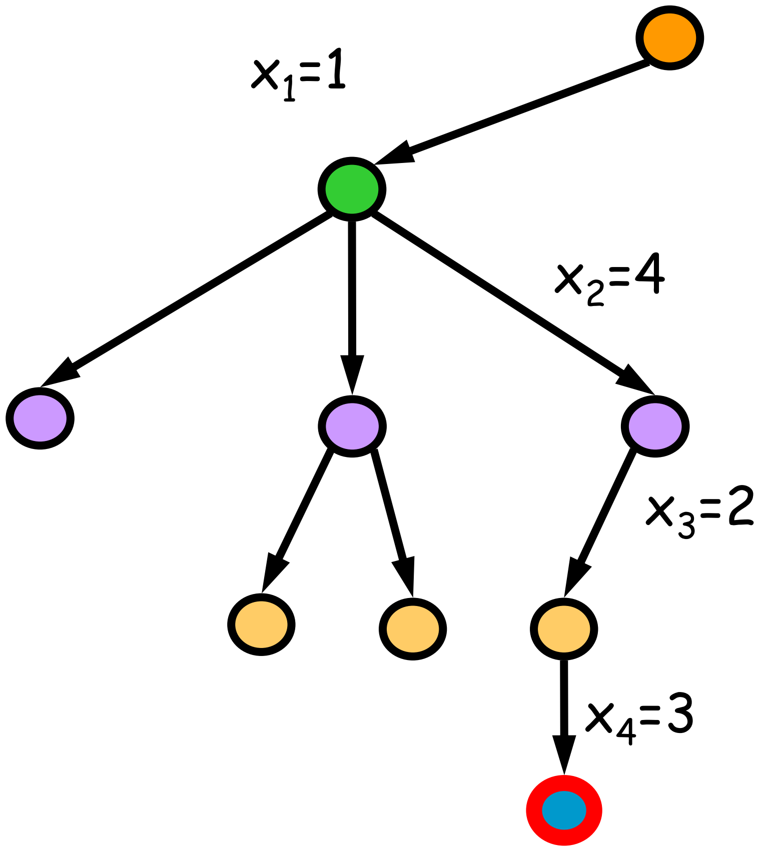
	1	2	3	4
1	Q			
2				Q
3		Q		
4				



- 这个顶点可能得到一个解么？有可能

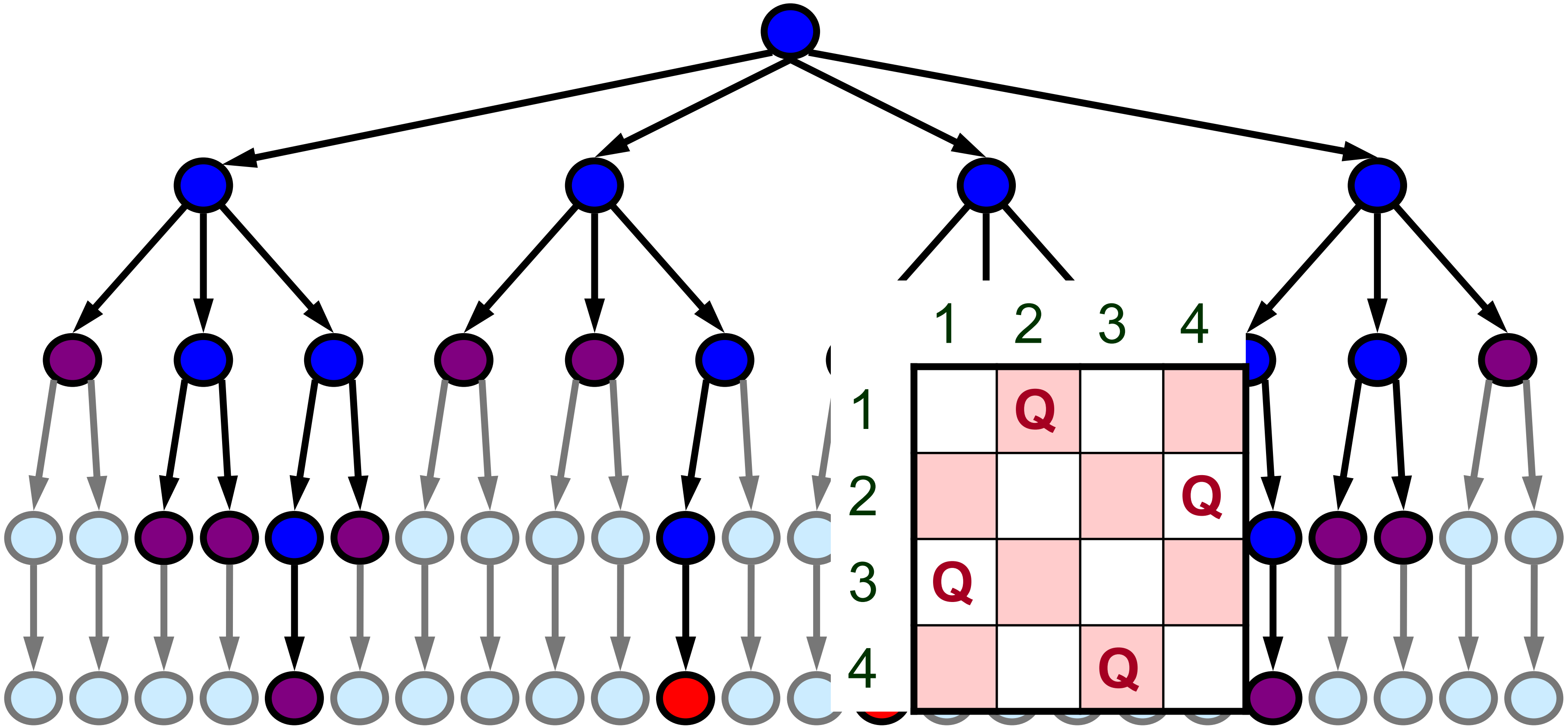
4-皇后问题

	1	2	3	4
1	Q			
2				Q
3		Q		
4			Q	



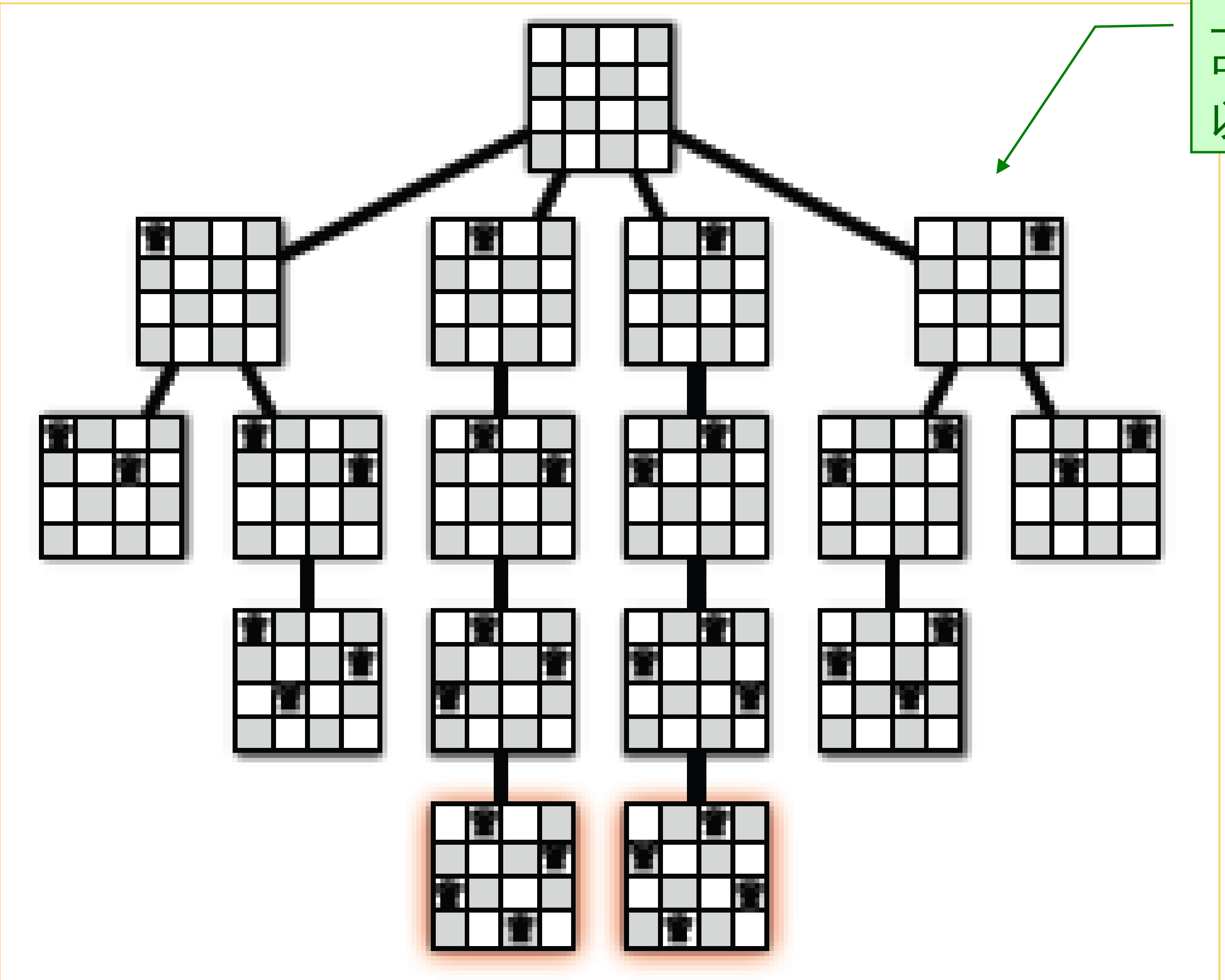
- 这个顶点可能得到一个解么？无可能
- 退回，做其他尝试

4-皇后问题



使用部分判定函数的搜索树（search tree）

4-皇后问题



上一页图中
中度蓝色顶点（有希望的状态），
以及红色顶点（合法解）

n-皇后问题

行号

保存结果

NQueens (k, n, x)

```
1. for i = 1 to n
2.     if ( Place (k, i, x) ) then
3.         x[k] ← i
4.         if ( k = n ) then
5.             for j = 1 to n
6.                 output x[j]
7.         else NQueens( k+1, n, x )
```

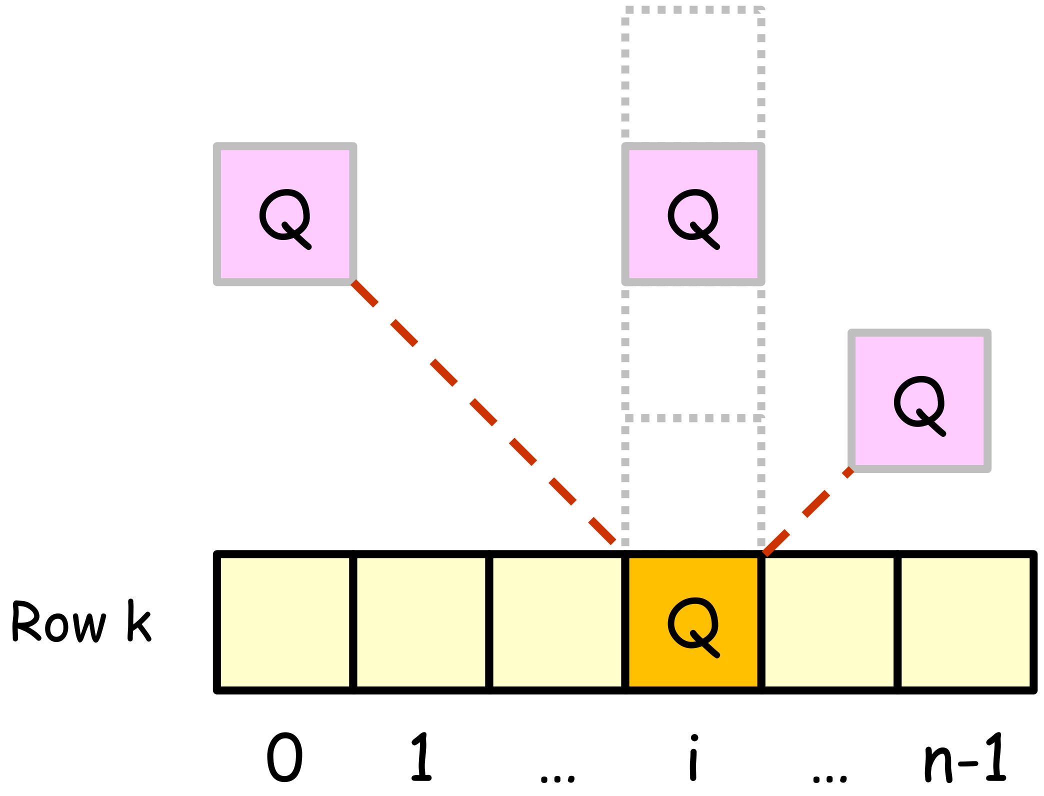
Place (k, i, x)

```
1. for j = 1 to k – 1
2.     if ( x[j] = i or abs(x[j]–i) = k–j ) then
3.         return false
4. return true
```

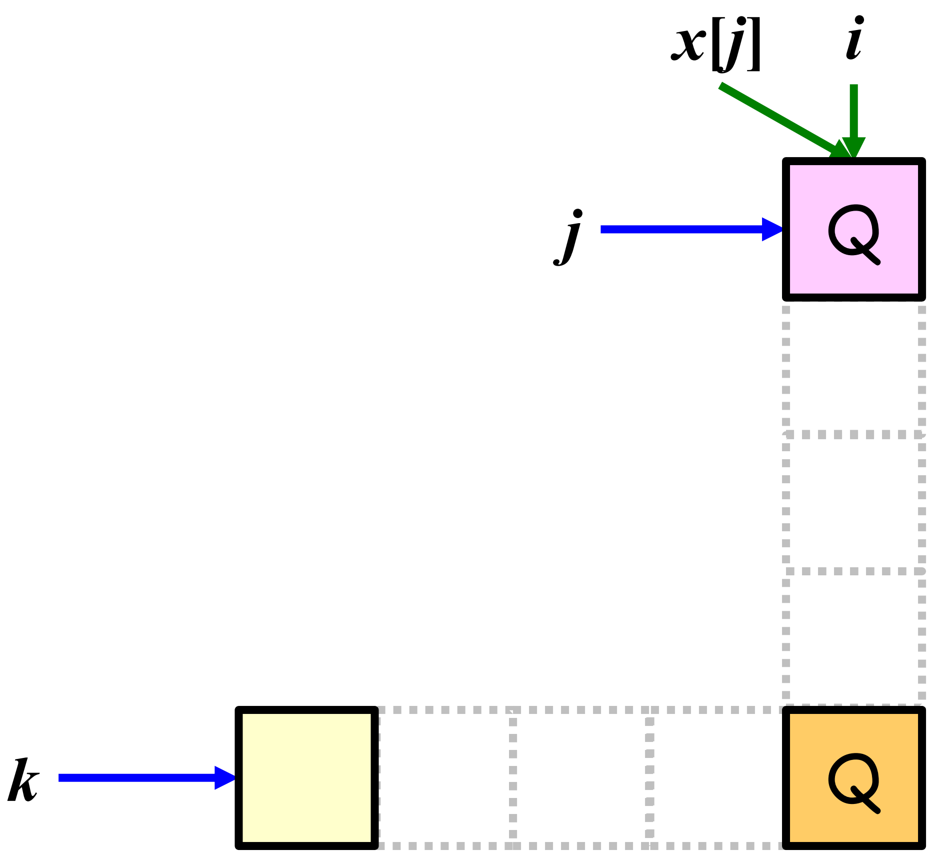
NQueens (n, x)

```
    NQueens ( 1, n, x )
```

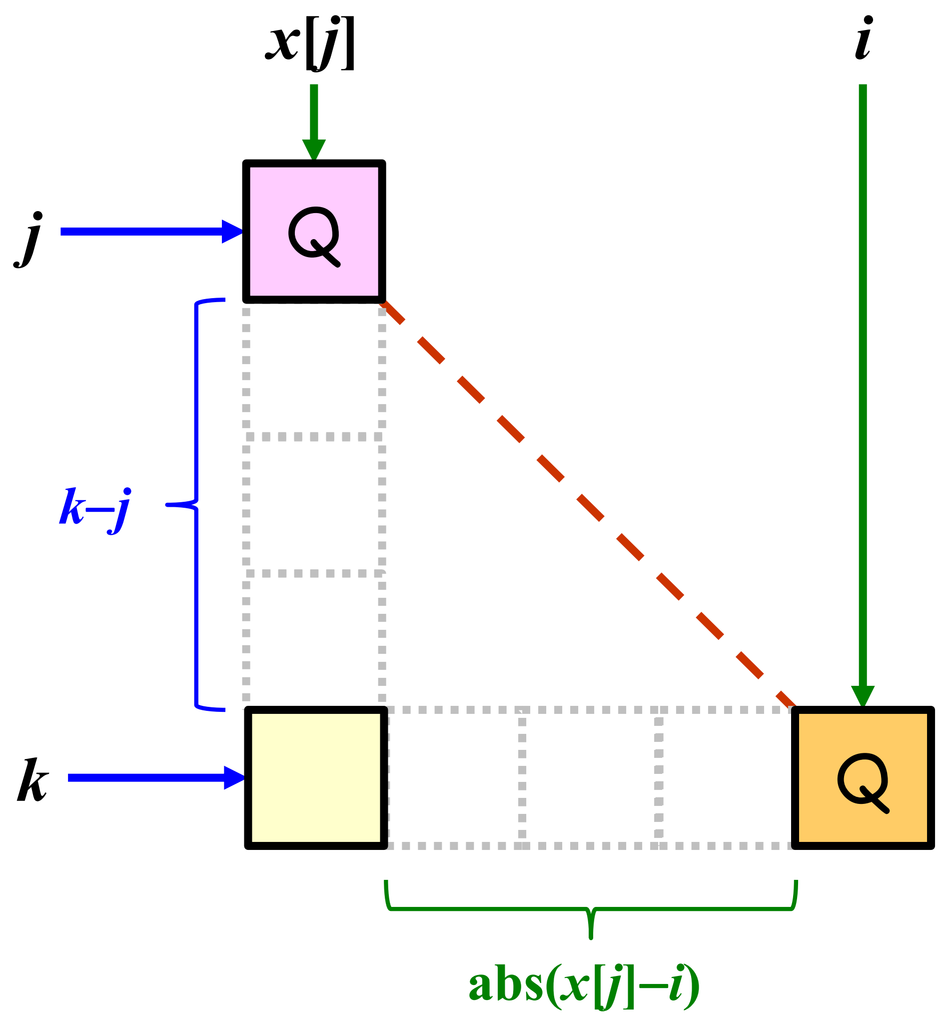
n-皇后问题



n-皇后问题



n-皇后问题



n-皇后问题

行号

保存结果

NQueens (k, n, x)

```
1. for i = 1 to n
2.    if ( Place (k, i, x) ) then
3.        x[k] ← i
4.        if ( k = n ) then
5.            for j = 1 to n
6.                output x[j]
7.        else NQueens( k+1, n, x )
```

Place (k, i, x)

```
1. for j = 1 to k – 1
2.    if ( x[j] = i or abs(x[j]–i) = k–j ) then
3.        return false
4. return true
```

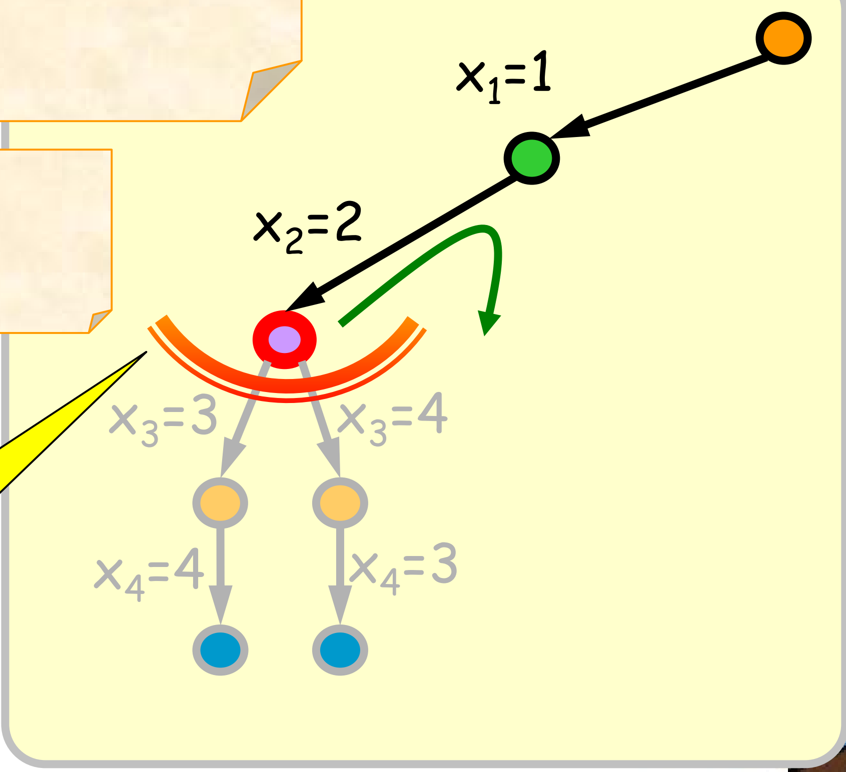
Initial call

NQueens (1, n, x)

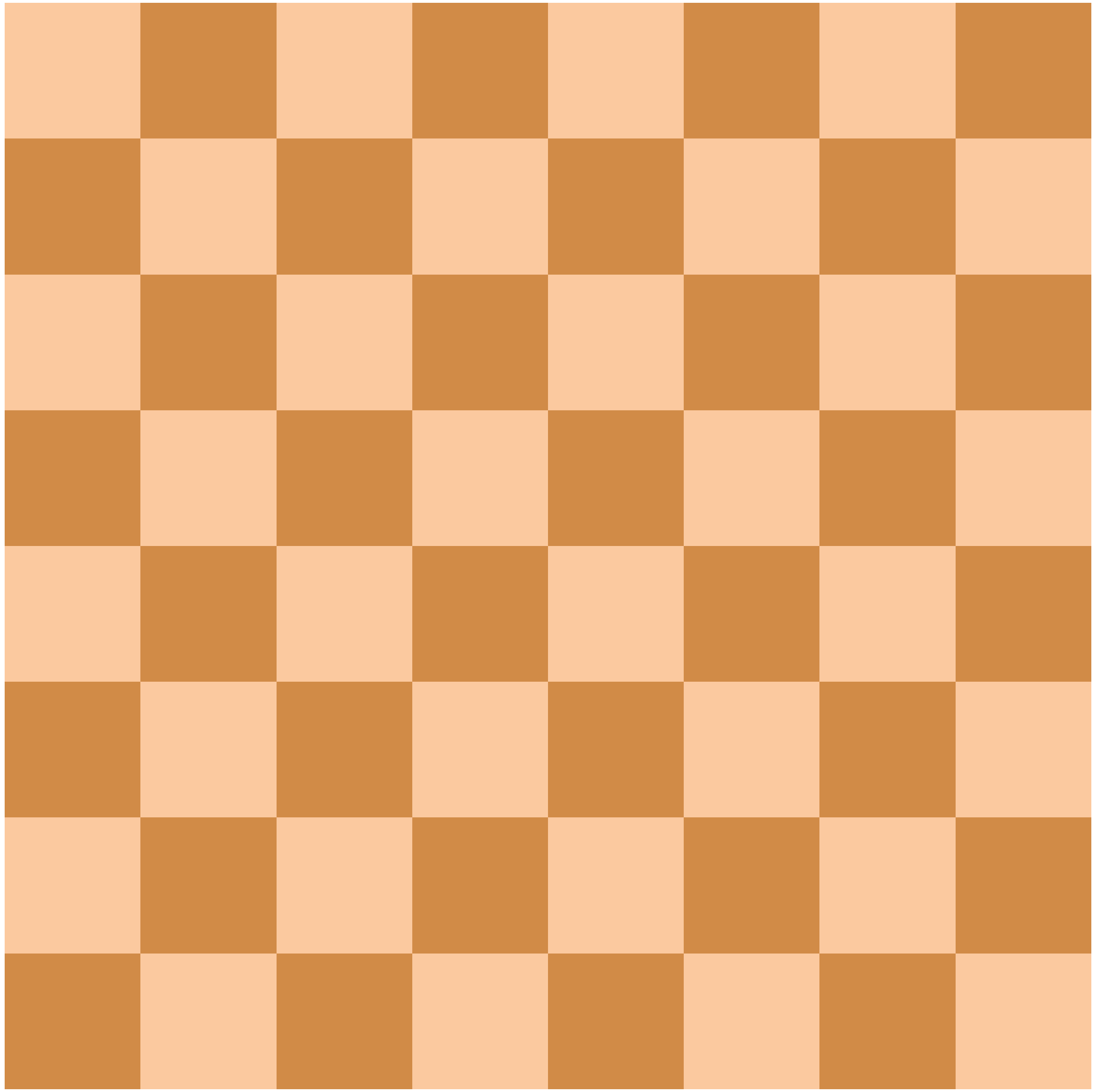
Place 执行 3~7 行

不执行 3~7 行

剪枝
（prune）



8-皇后问题



8-皇后问题的解

- 8-皇后问题共有92个不同的解
- 将对称的（旋转和翻转）的解视作相同，则8-皇后问题共有12个不同的（基本）解

回溯法

Backtracking

回溯法

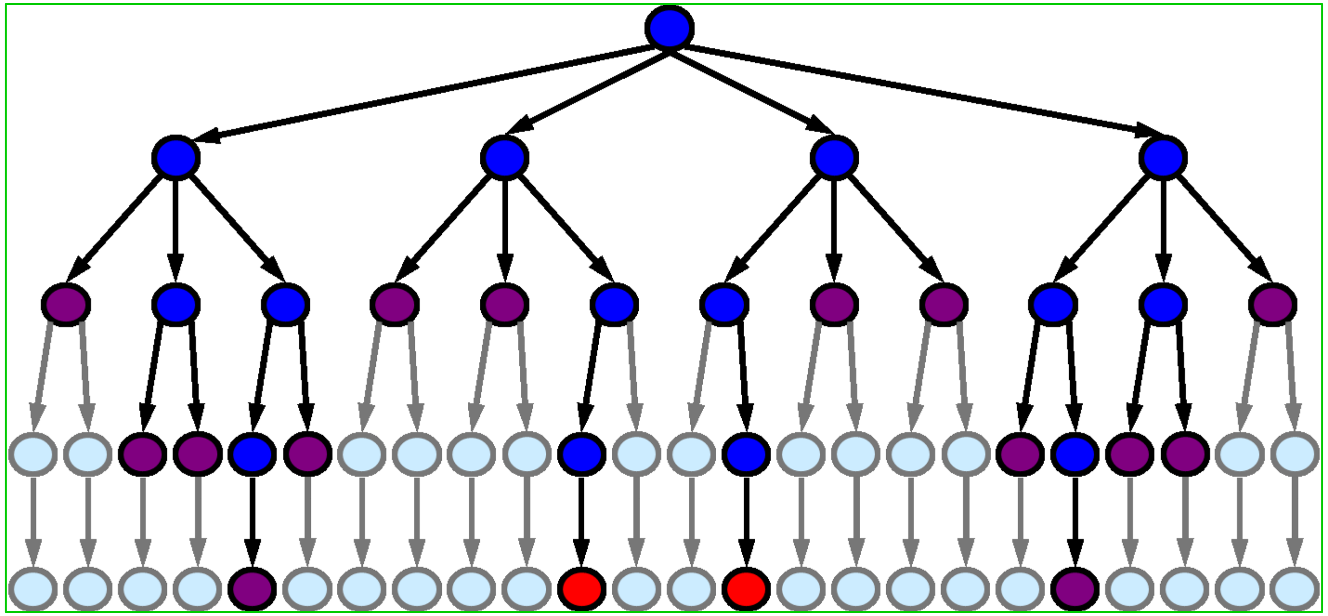
- “回溯”一词是美国数学家D.H.Lehmer在20世纪50年代创造的
- 回溯算法从根开始、以深度优先的顺序递归遍历整棵搜索树，枚举部分候选解（partial candidates）的集合部分
- 候选解是（潜在）搜索树的顶点
  - 每个部分候选项都通过单个扩展步骤得到其孩子顶点
  - 树的叶子是不能进一步扩展的部分候选项
  - 候选项可能给出给定问题的可行解

回溯法

- “剪掉”无前景/无希望（non-promising）顶点——即剪枝
  - DFS将不会探索以其为根的子树（因为不可能得到有效解），并“回溯”到其父亲顶点
- 具体而言，在每个顶点 *c* 处，检查 *c* 是否可能成为有效的解
  - 如果不可能，那么就跳过以 *c* 为根的整个子树（即，剪枝，prune）
  - 否则，算法
    - 检查 *c* 本身是否已经构成有效解，如果是的话则输出
    - 递归枚举 *c* 的所有子树
- 因此，算法遍历的实际搜索树只是潜在搜索树（potential tree）的一部分

回溯法

- 算法遍历的实际搜索树只是潜在搜索树（potential tree）的一部分



回溯法

纯
DFS

回溯法

hen

Solve(i)

```
If i is a leaf then
    test whether it is a solution
Else
    Solve (i+1, choice 1)
    Solve (i+1, choice 2)
    ……
    Solve (i+1, choice k)
```

大意如此，非严谨表述

例 5.6

子集和问题

The Sum-of-Subsets Problem

刘铎
liuduo@bjtu.edu.cn

子集和问题

- 给定 n 个正整数 $w_1, \ldots, w_n$ 构成的集合 S 及一个整数 W， 找出 S 的元素总和恰好为 W 的所有子集

子集和问题

- 示例:
  - $n = 5, W = 21$

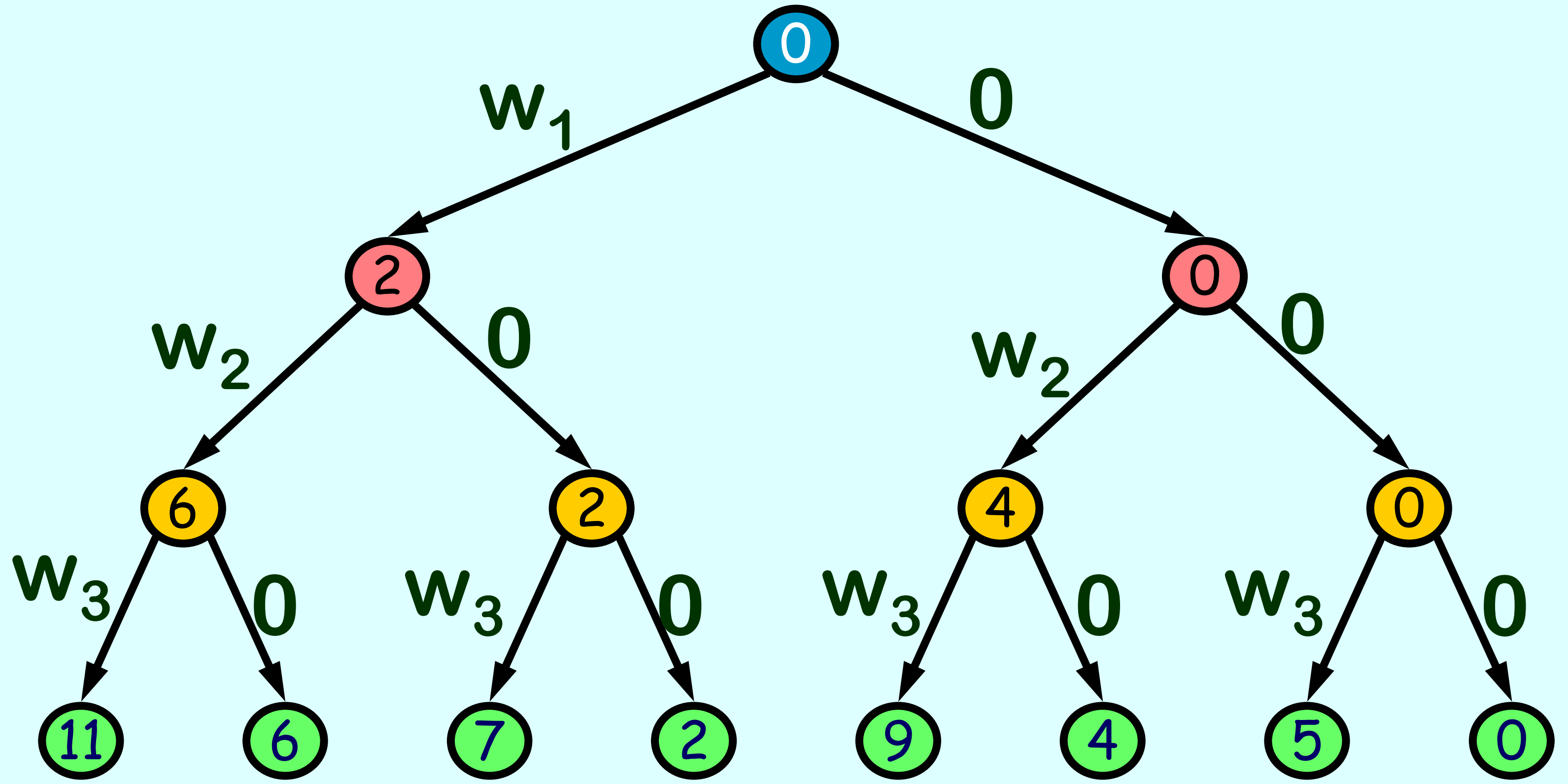
    $$w_1=5, w_2=6, w_3=10, w_4=11, w_5=16$$

  - 由于

    $$w_1+w_2+w_3 = w_1+w_5 = w_3+w_4 = 21$$

    因此全部有效解为 $\{w_1, w_2, w_3\}, \{w_1, w_5\}, \{w_3, w_4\}$

状态空间树



W = 6， $w_1=2$, $w_2=4$, $w_3=5$

子集和问题

- **solve(i, demand, bitmap)**
  - 找到 $\{w_i, \ldots, w_n\}$ 的元素总和恰好为 **demand** 的所有子集
  - **bitmap** 表示从根到该顶点的道路

考虑第i个值

算 法

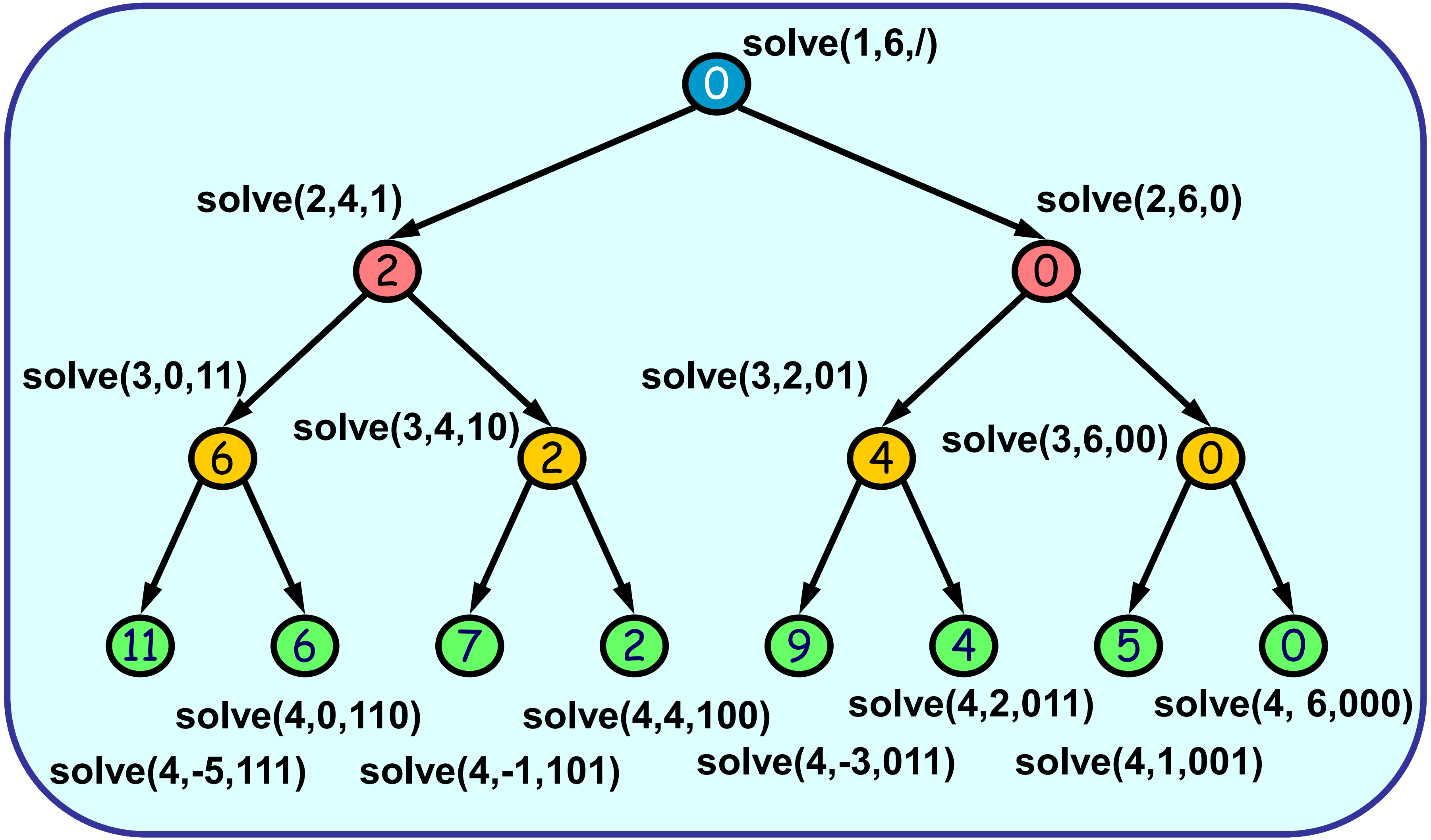
solve (*i, demand, bitmap*)

```
if ( i > n ) then
    if ( demand = 0 ) then
        output bitmap[1] ~ bitmap[n]
else
    bitmap[i] ← "1"
    solve ( i + 1, demand – w_i, bitmap )
    bitmap[i] ← "0"
    solve ( i + 1, demand, bitmap )
```

Initial

solve (1, *W*, *bitmap*)

状态空间树



$W = 6$，$w_1=2, w_2=4, w_3=5$

检查顶点是否是有希望的

- 穷竭式搜索/蛮力法：
  - 基于深度优先查找
  - 树的前序遍历
- 当 $\boldsymbol{demand = 0}$ 时
  - 输出结果并进行回溯（寻找下一个结果）
- 当 $\boldsymbol{demand < 0}$ 时
  - 此为一个无希望/无前景顶点

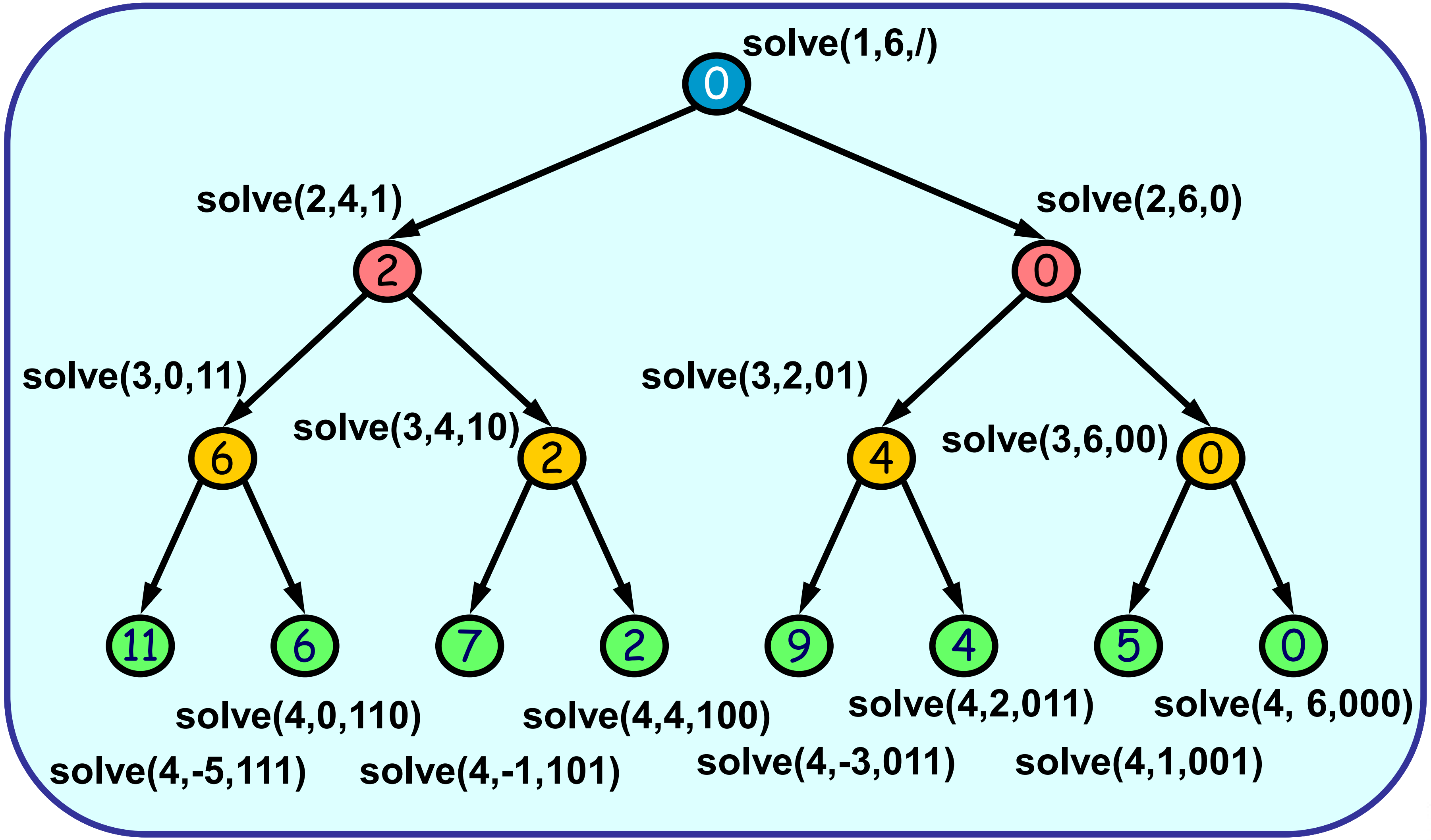
算 法

```
solve ( i, demand, bitmap )
if ( promising (i) )
    if (demand = 0) then
        output bitmap[1] ~ bitmap[i-1]
    else if i ≤ n
        bitmap[i] ← "1"
        solve ( i + 1, demand – w_i, bitmap )
        bitmap[i] ← "0"
        solve ( i + 1, demand, bitmap )
```

```
promising (i)
if demand < 0 then return false
else return true
```

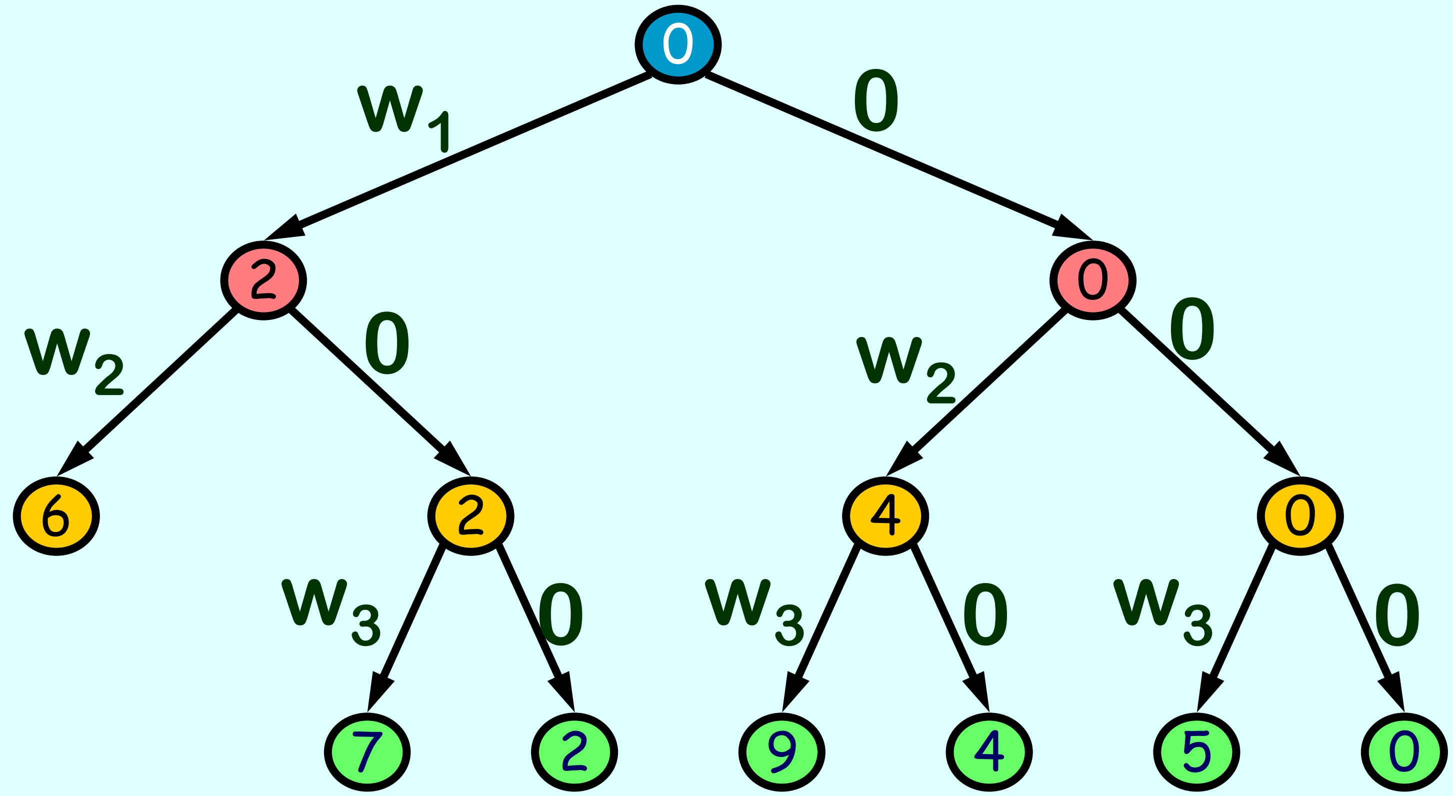
```
Initial call
  solve ( 1, W, bitmap )
```

状态空间树



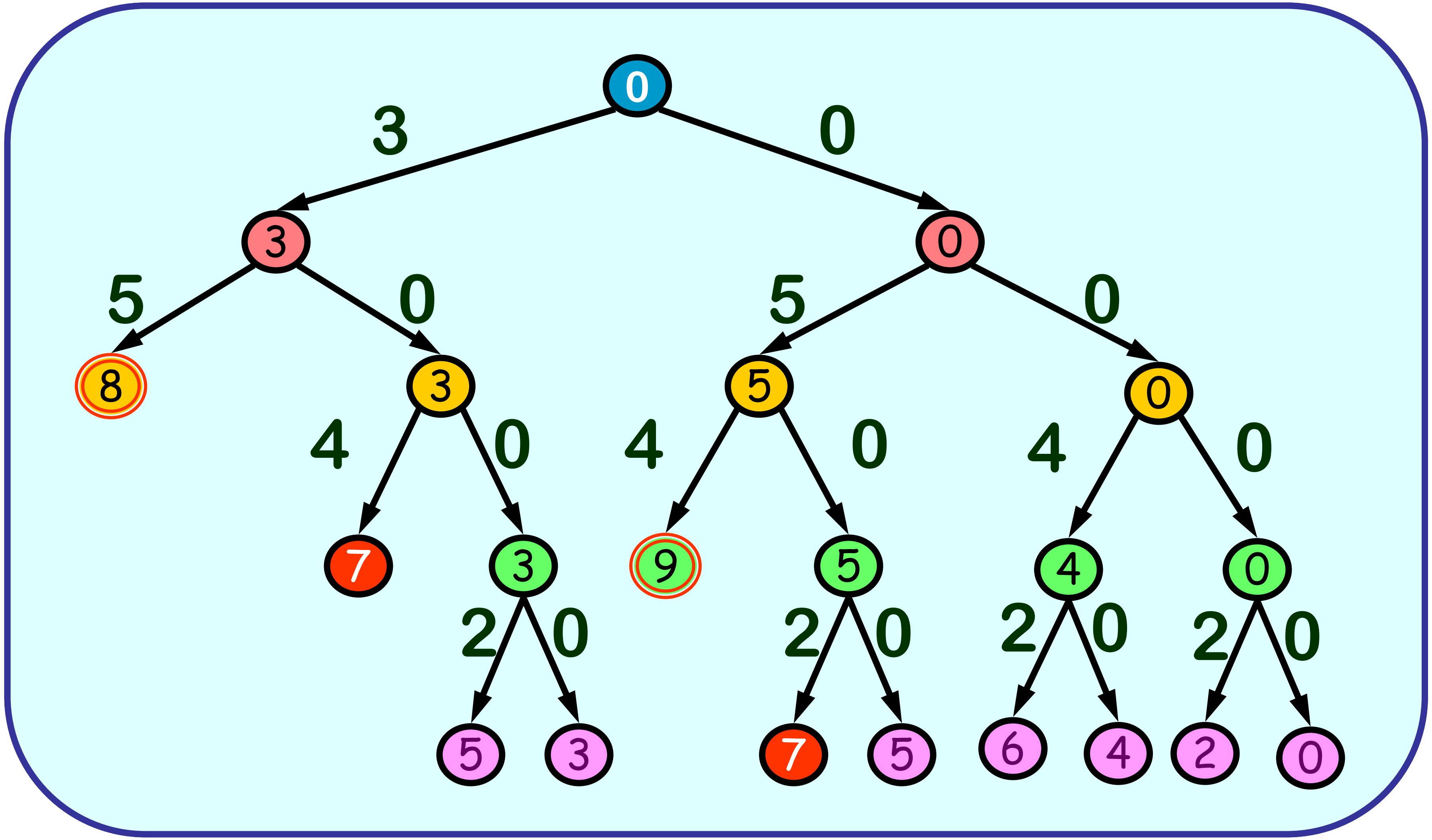
$W = 6$， $w_1=2, w_2=4, w_3=5$

状态空间树



$W = 6$，$w_1=2, w_2=4, w_3=5$

状态空间树

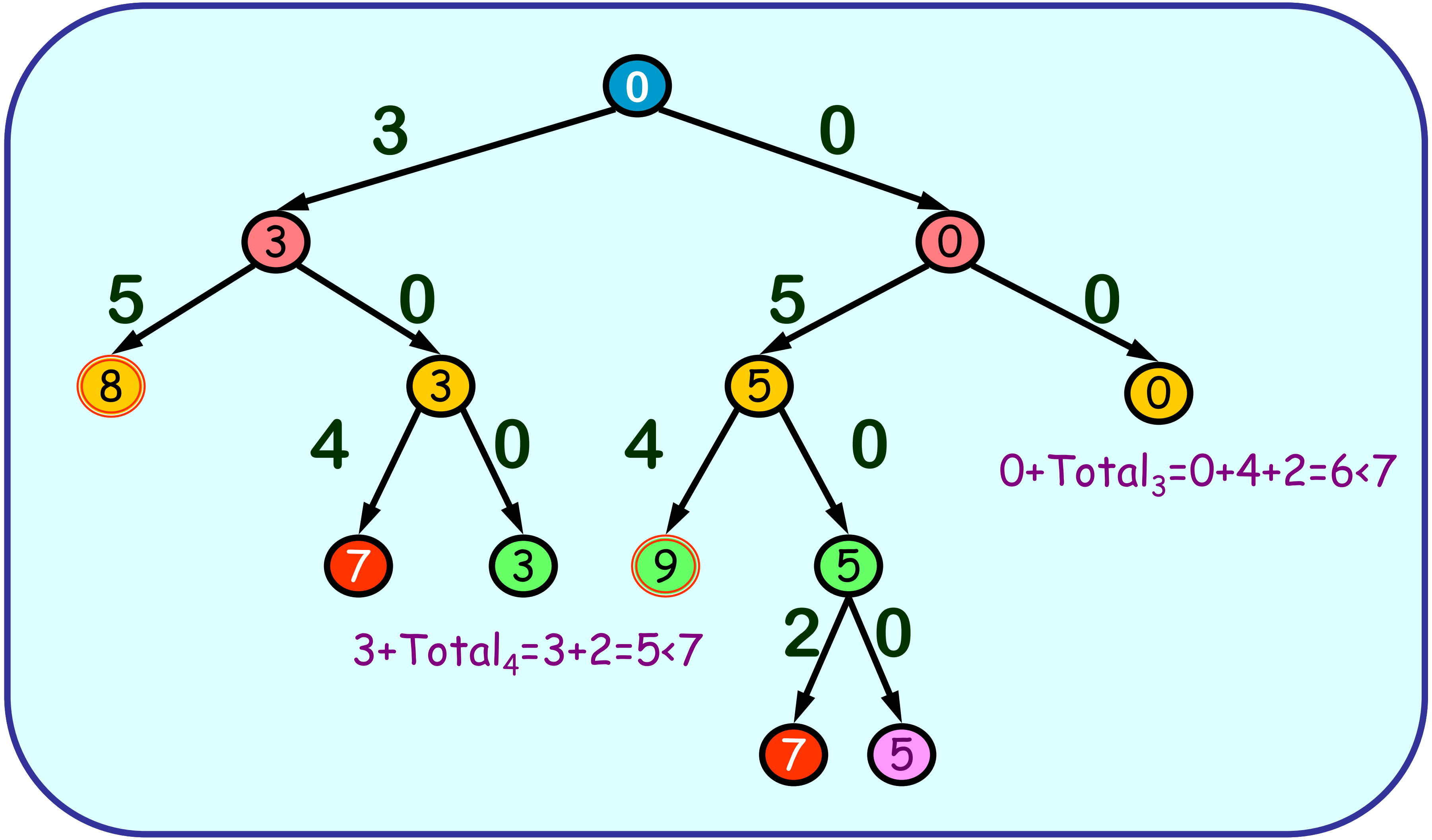


W = 7， w_1=3, w_2=5, w_3=4 , w_4=2

子集和问题

- 更进一步：
  - 如果和 $w_i+\ldots+w_n$ 严格小于 *demand* 则这是一个无希望/无前景顶点
  - 将和 $w_i+\ldots+w_n$ 记做 $total_i$

状态空间树



$W = 7$， $w_1=3, w_2=5, w_3=4 , w_4=2$

solve (*i, demand, bitmap, total*)

```
if ( promising (i) )
    if (demand = 0) then
        output bitmap[1] ~ bitmap[i-1]
    else if i ≤ n
        bitmap[i] ← "1"
        solve ( i + 1, demand – w_i, bitmap, total – w_i )
        bitmap[i] ← "0"
        solve ( i + 1, demand, bitmap, total – w_i )
```

promising (*i*)

```
if demand < 0 then return false
else if total < demand then return false
else return true
```

Initial call

solve (1, W, *bitmap* , $w_1 + w_2 + \ldots + w_n$)

例 5.6

顶点着色问题

Vertex Coloring Problem

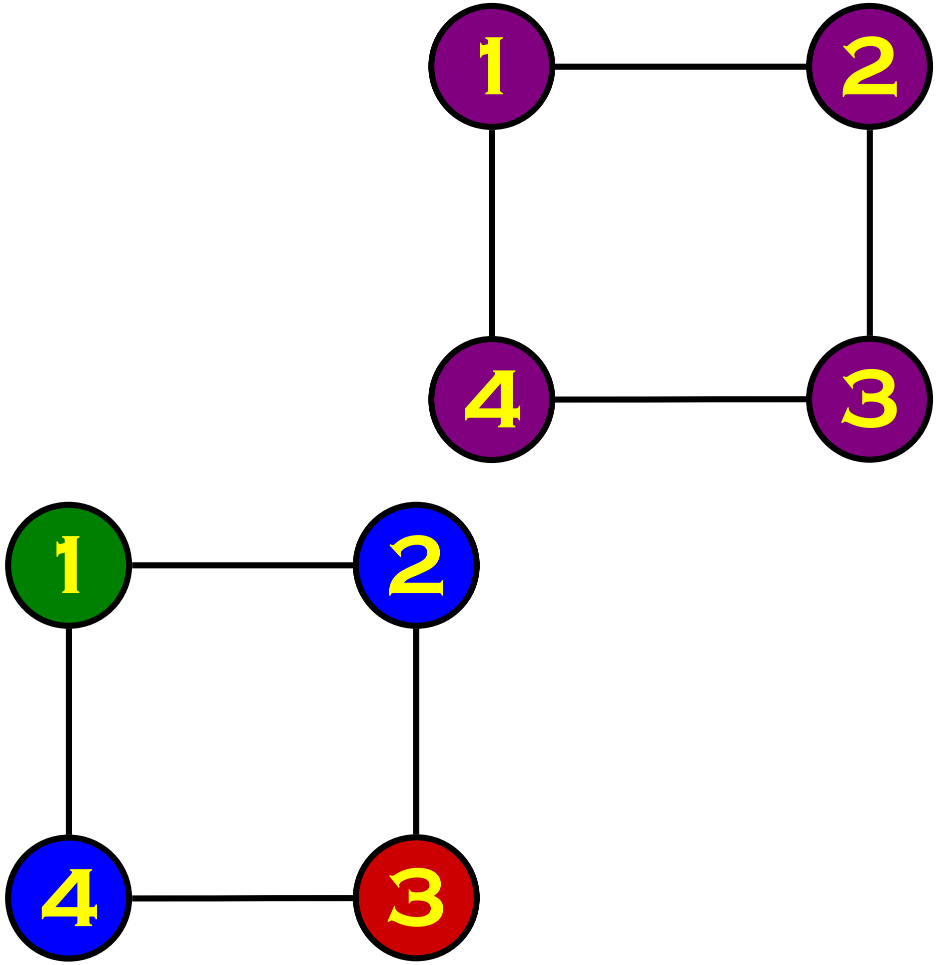
刘铎
liuduo@bjtu.edu.cn

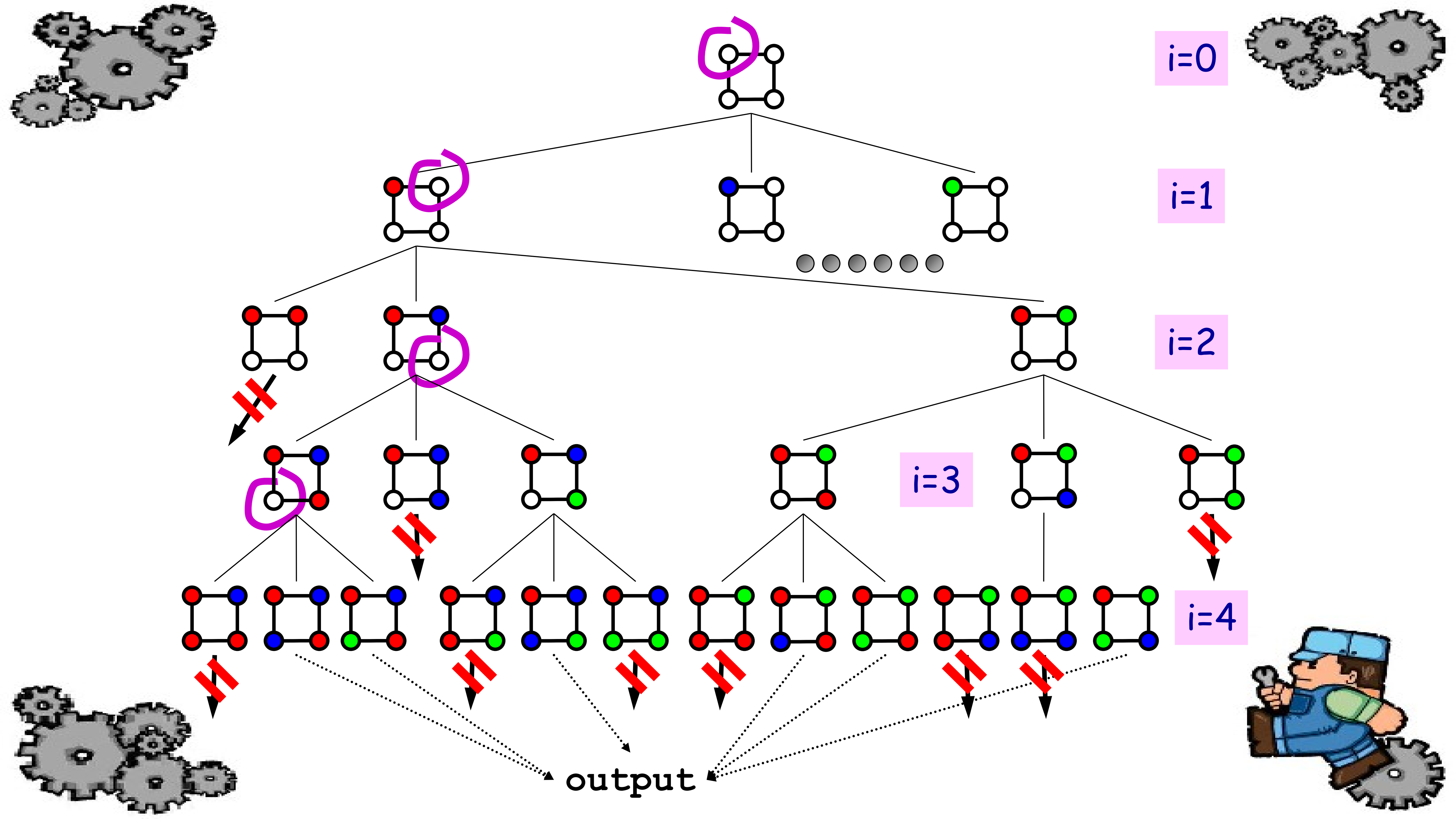
顶点着色问题

- 为图的顶点指定颜色，满足相邻顶点不共享相同的颜色
  - 如果从顶点 i 到顶点 j 有一条边，则顶点 i 和 j 是相邻的
- 求图的所有 k-着色
  - 为给定图找到所有至多使用 k 种颜色的点着色方案

顶点着色问题

- 示例
  - 3-着色问题
- 顶点着色
  - v_1 color1
  - v_2 color2
  - v_3 color3
  - v_4 color2





i=0
i=1
i=2
i=3
i=4
output

顶点着色问题

Algorithm mcoloring (index *i*)

```
1. if ( promising (i) ) then
2.     if ( i = n ) then
3.         output vcolor[1] ~ vcolor[n]
4.     else
5.         for color = 1 to k
6.             vcolor[i + 1] ← color
7.             mcoloring ( i + 1 )
```

Initial call

```
mcoloring ( 0 )
```

已经尝试染了i个顶点

Algorithm promising (index *i*)

```
1. switch ← true
2. j ← 1
3. while ( j < i and switch )
4.     if ( W[i][j] and vcolor[i] = vcolor[j] ) then
5.         switch ← false
6.     j ← j + 1
7. return switch
```

switch

执行 2~7 行

不执行 2~7 行

W 是图的邻接矩阵

回溯法

纯
DFS

回溯法

hen

Solve(i)

If i is a leaf then
    test whether it is a solution
Else
    Solve (i+1, choice 1)
    Solve (i+1, choice 2)
    ……
    Solve (i+1, choice k)

大意如此，非严谨表述

分支限界法

Branch and Bound

刘铎
liuduo@bjtu.edu.cn

A

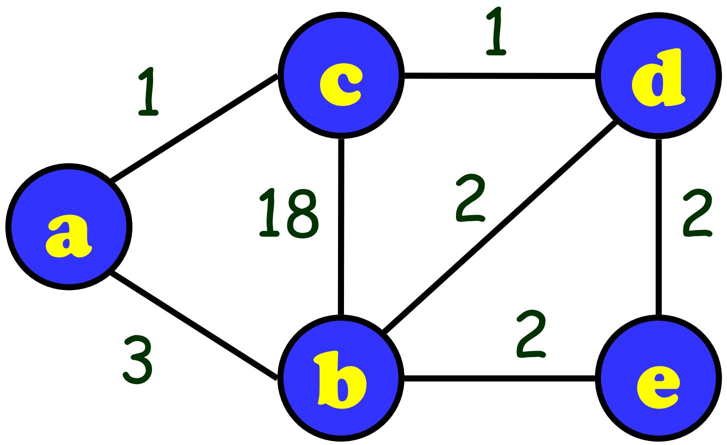
最短道路

Shortest Path

刘铎
liuduo@bjtu.edu.cn

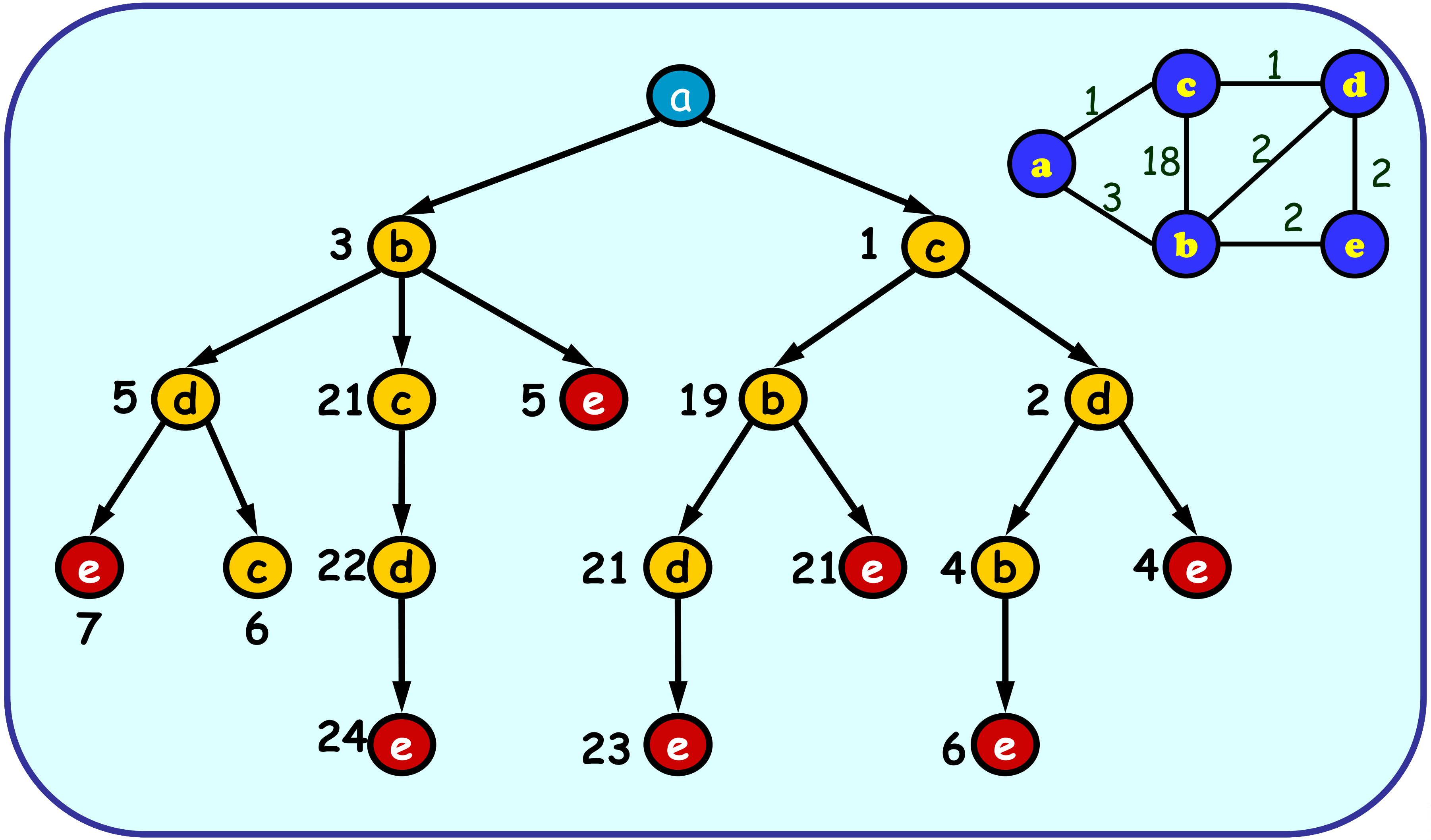
最短道路

- 找到从 a 到 e 的最短道路



- 蛮力法：给出从 a 到 e 的所有简单道路，并从中通过比较得到最优者
- 观察结果：不会重复通过同一个顶点

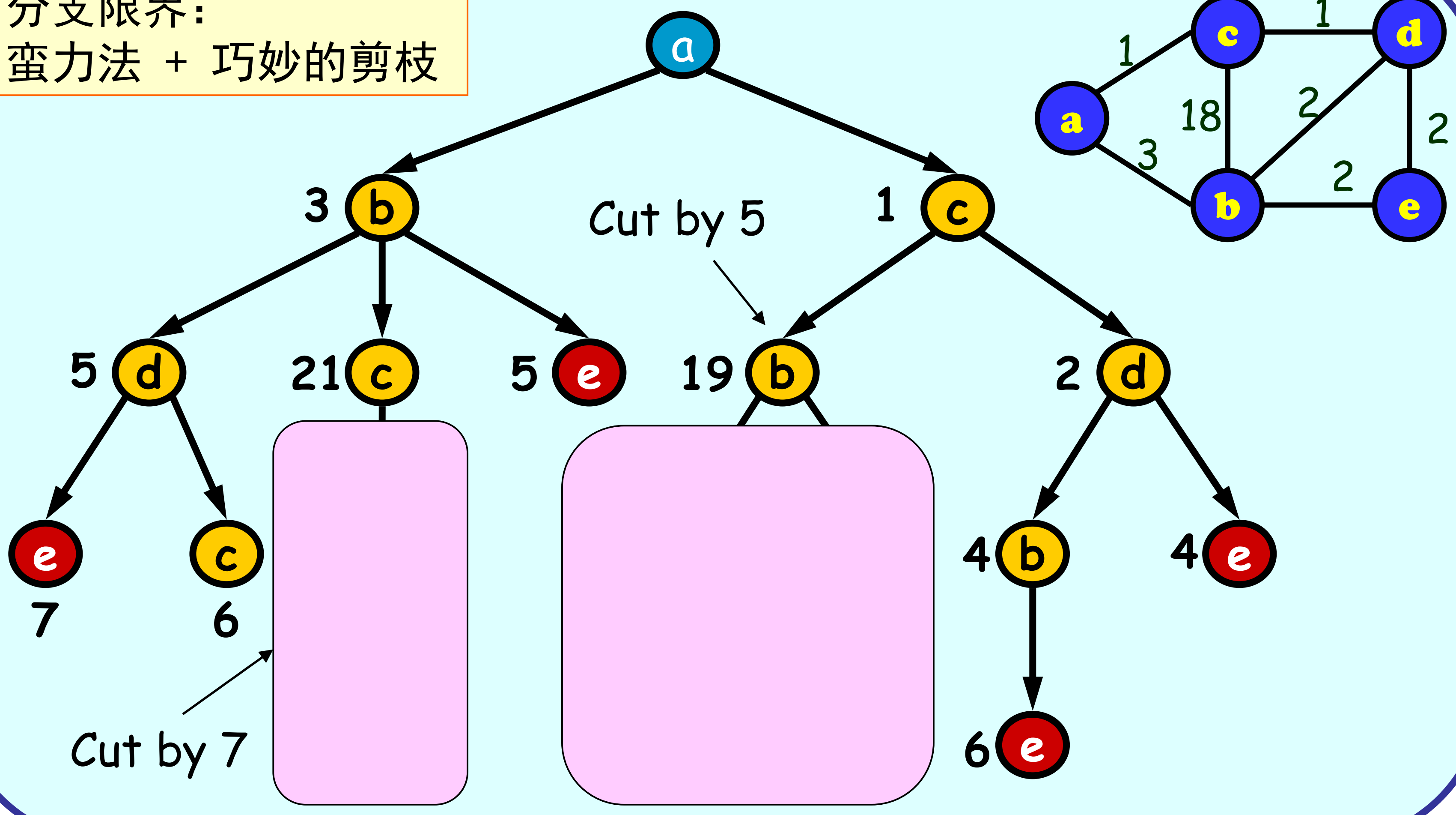
最短道路



a
3 b
1 c
5 d
21 c
5 e
19 b
2 d
e
7
c
6
22 d
21 d
21 e
4 b
4 e
24 e
23 e
6 e
a
c
d
b
e
1
1
18
2
2
3
2

最短道路

分支限界：
蛮力法 + 巧妙的剪枝



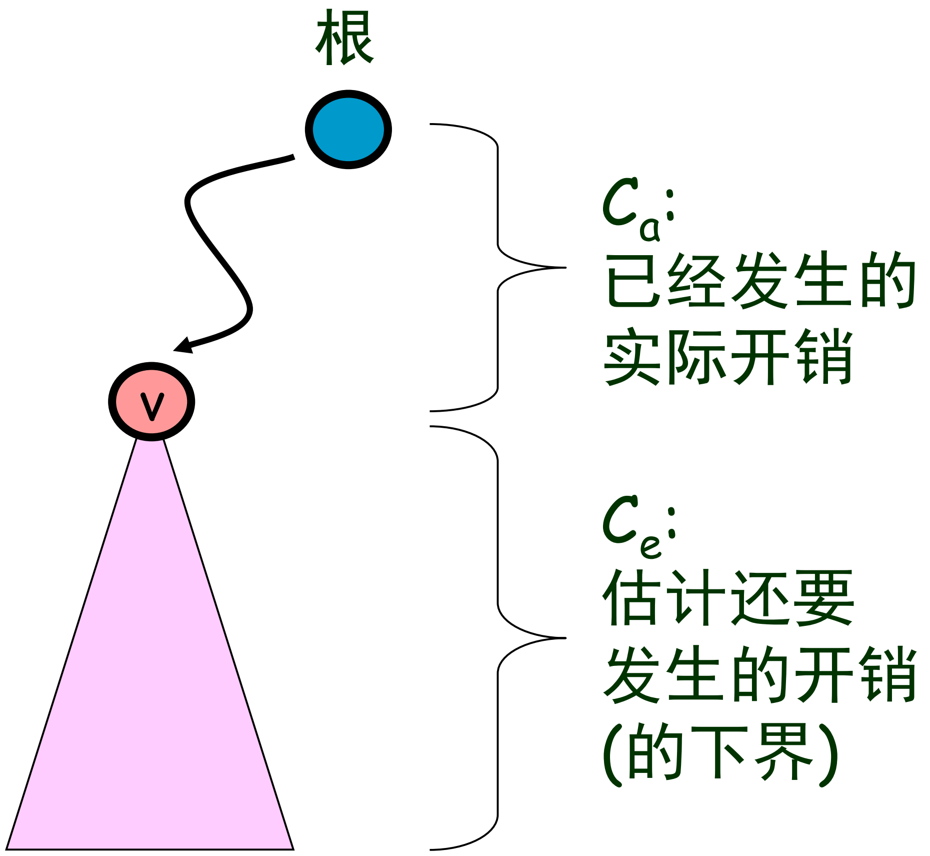
分支限界的基本思想

- 维护“到目前为止的最优值”
- 更新“到目前为止的最优值”
- 进行估值和剪枝

分支限界的基本思想

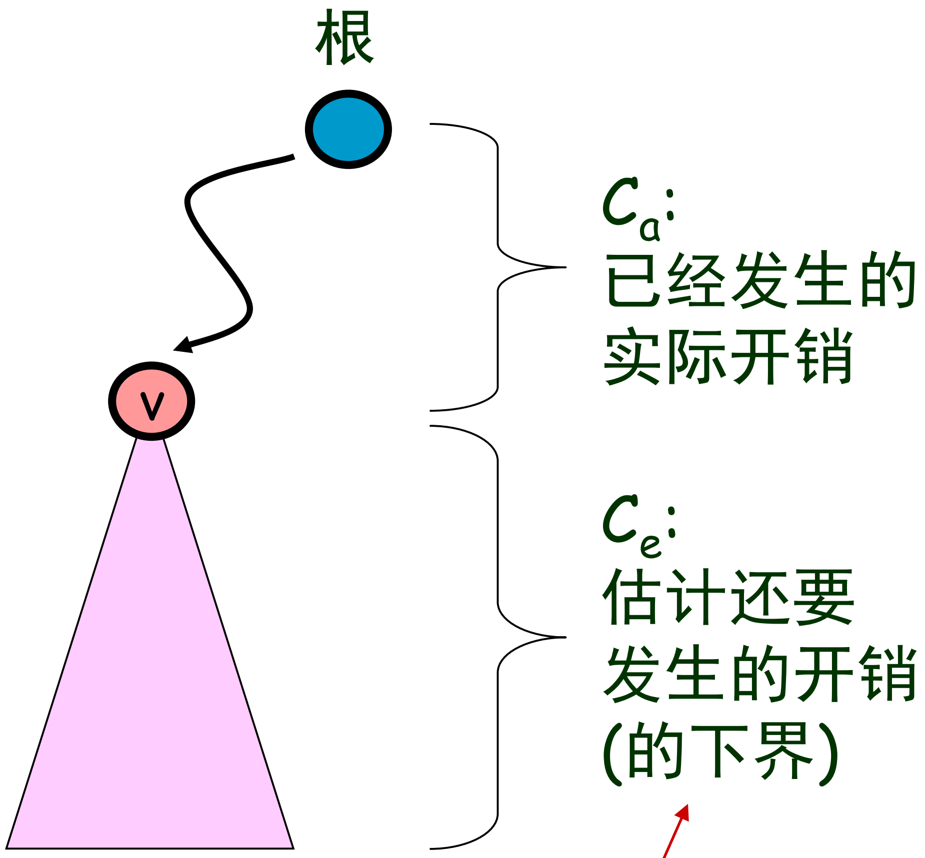
- (对于“最小”的目标)
- ***b***: 目前的最优值

  (初始化：$b = \infty$)
- 在顶点 v 处进行回溯，若
  - v 是叶子顶点，或者
  - $C_a + C_e \geq b$
- 如果得到了更优的解，那么就以其更新 ***b***



分支限界的基本思想

- (对于“最小”的目标)
- b: 目前的最优值
  (初始化：$b = \infty$)
- 在顶点 v 处进行回溯，若
  - v 是叶子顶点，或者
  - $C_a + C_e \geq b$
- 如果得到了更优的解，那么就以其更新 b

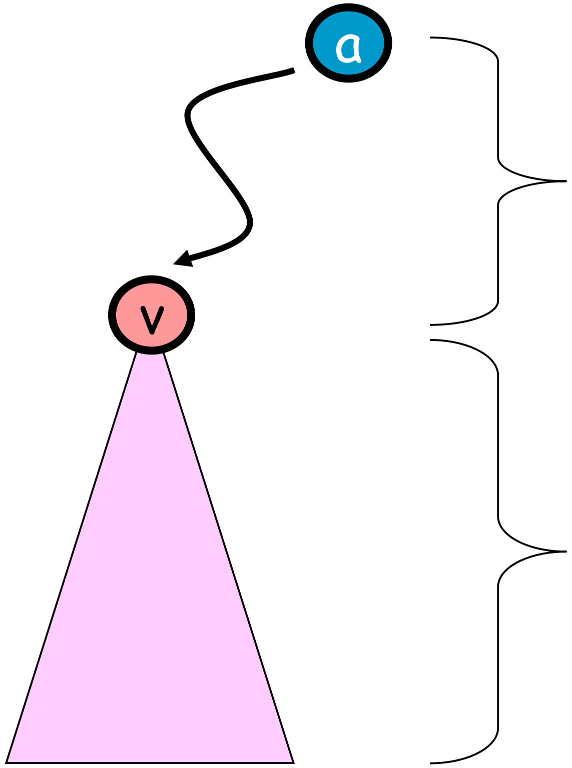


$C_a + C_e$ 称为在 v 处的估界函数/代价函数值

这是最困难和最具技巧性的部分

最短道路

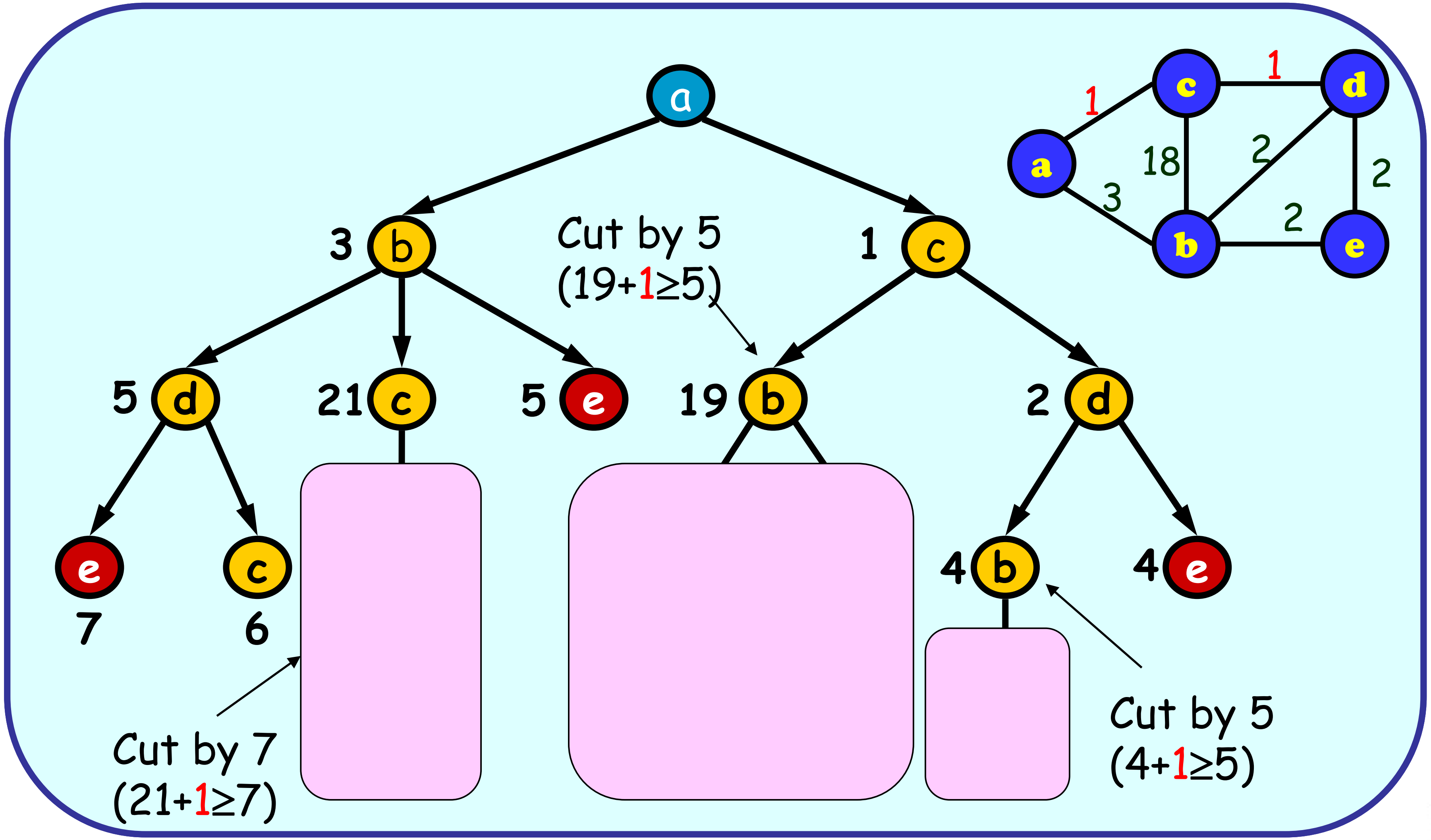
- 找到从 a 到 e 的最短道路



C_a: 由 a 到 v 的道路长度

C_e: 图中最短边的长度

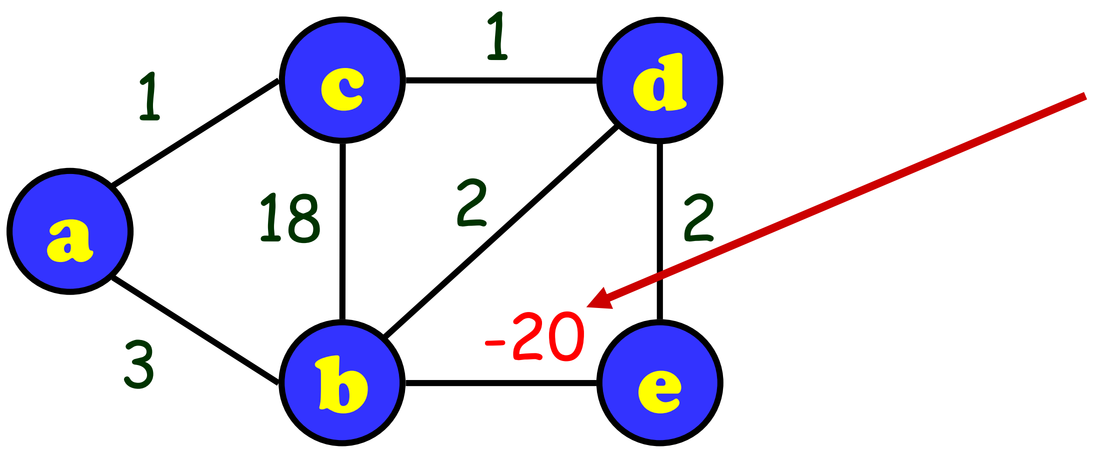
最短道路



a
3 b
1 c
Cut by 5
(19+1≥5)
5 d
21 c
5 e
19 b
2 d
e
7
c
6
4 b
4 e
Cut by 7
(21+1≥7)
Cut by 5
(4+1≥5)
1
c
1
d
a
18
2
2
3
2
b
e

最短道路

- 寻找从 a 到 e 的一条最短道路



- 分支限界法是否依然适用？

回溯与分支限界

- 穷竭式搜索的改进
  - 穷竭式搜索的搜索空间可能很大
- 回溯法一般用以处理寻找有效解的问题
- 分支限界法一般用以处理最优化问题

分支限界法

Branch and Bound

刘铎
liuduo@bjtu.edu.cn

分支限界法

- 在单调性假设下，分支限界法可确保能够找到最优解
- 它是求解各种最优化问题——特别是离散优化问题和组合优化问题——的通用算法

分支限界法

- 设置一个估界函数/代价函数，用于计算状态空间树上某个顶点的界（目标函数的值），并确定其是否有希望/有前景
  - 有希望/有前景（如果估界值优于当前最优值）：继续扩展此顶点
  - 无希望/无前景（如果估界值不优于当前最优值）：不扩展到节点之外（即对状态空间树进行剪枝）

分支限界法

Solve(i)

纯DFS

分支限界法

```
If i is a leaf then
    If current_value is better than current_best then
        current_best ← current_value
Else
    Solve (i+1, choice 1)
    Solve (i+1, choice 2)
    ……
    Solve (i+1, choice k)
```

大意如此，非严谨表述

分支限界法

- 如何计算界？
  - 得到的第一个可行解——有可能需要很长时间
  - 一个显而易见的解——比如贪婪策略得到的解

分支限界法

- 可以看做回溯法的一个“加强版”
  - 相似之处
    - 都使用状态空间树来解决问题
  - 不同之处
    - 分支限界法用于最优化问题
    - 回溯法用于非优化问题

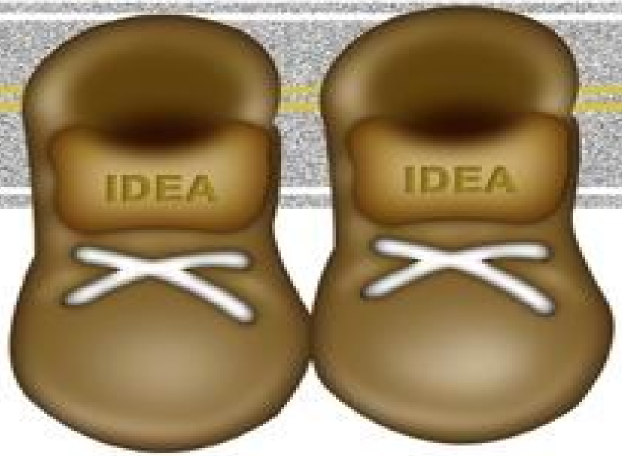
分支限界法

- 何时使用穷竭式搜索？
  - 问题规模非常小
  - 生成一个候选解和检验一个候选解是否可行/有效都非常容易
- 何时使用分支限界？
  - 最优化问题
  - 想不出更好的算法
  - 穷竭式搜索不现实

分支限界法

- 此处介绍的都只是可分解为多步骤的比较简单的问题
  - 学习算法框架使用
- 涉及到图搜索的问题可能会更加复杂些
- 除基于DFS的方法外，还有一些其他的扩展顶点方法和搜索算法
  - 例如A*算法等

A



装载问题

Packing Problem

刘铎

liuduo@bjtu.edu.cn

装载问题

- 给定 n 个物品的集合 U，每个物品有其重量 $w(i)\in\mathbb{Z}^+$，并给定总重量限制 W，满足

$$W \geq \max\{\, w(i) : i \in U \,\}$$

- 找到 U 的一个子集 $U' \subseteq U$ 使得

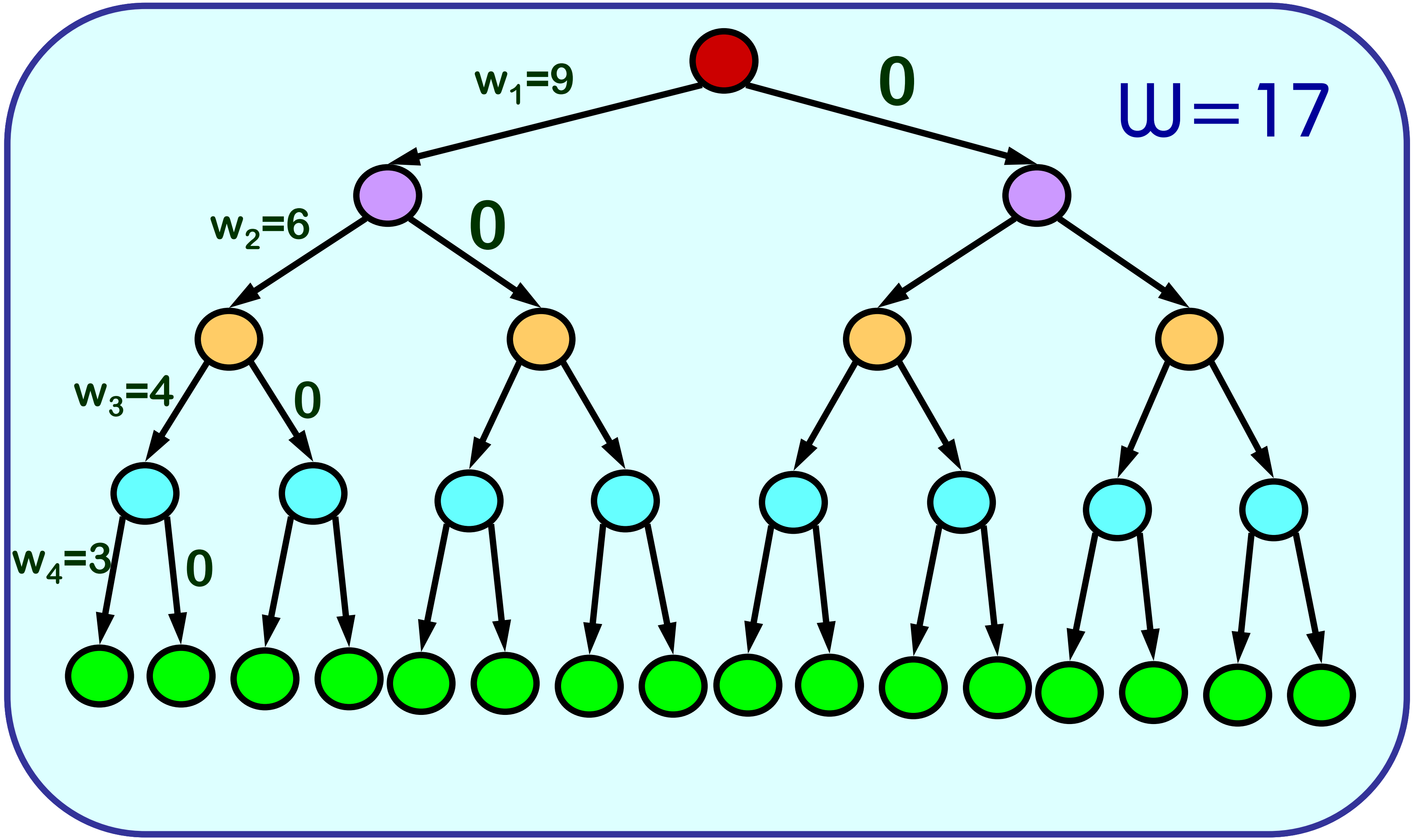
$$\Sigma_{i\in U'}\, w(i) \leq W$$

且上述和式达到可能的最大值

装载问题

- 示例
  - $W = 17$, $n = 4$, $w_1 = 9$, $w_2 = 6$, $w_3 = 4$, $w_4 = 3$

装载问题



装载问题

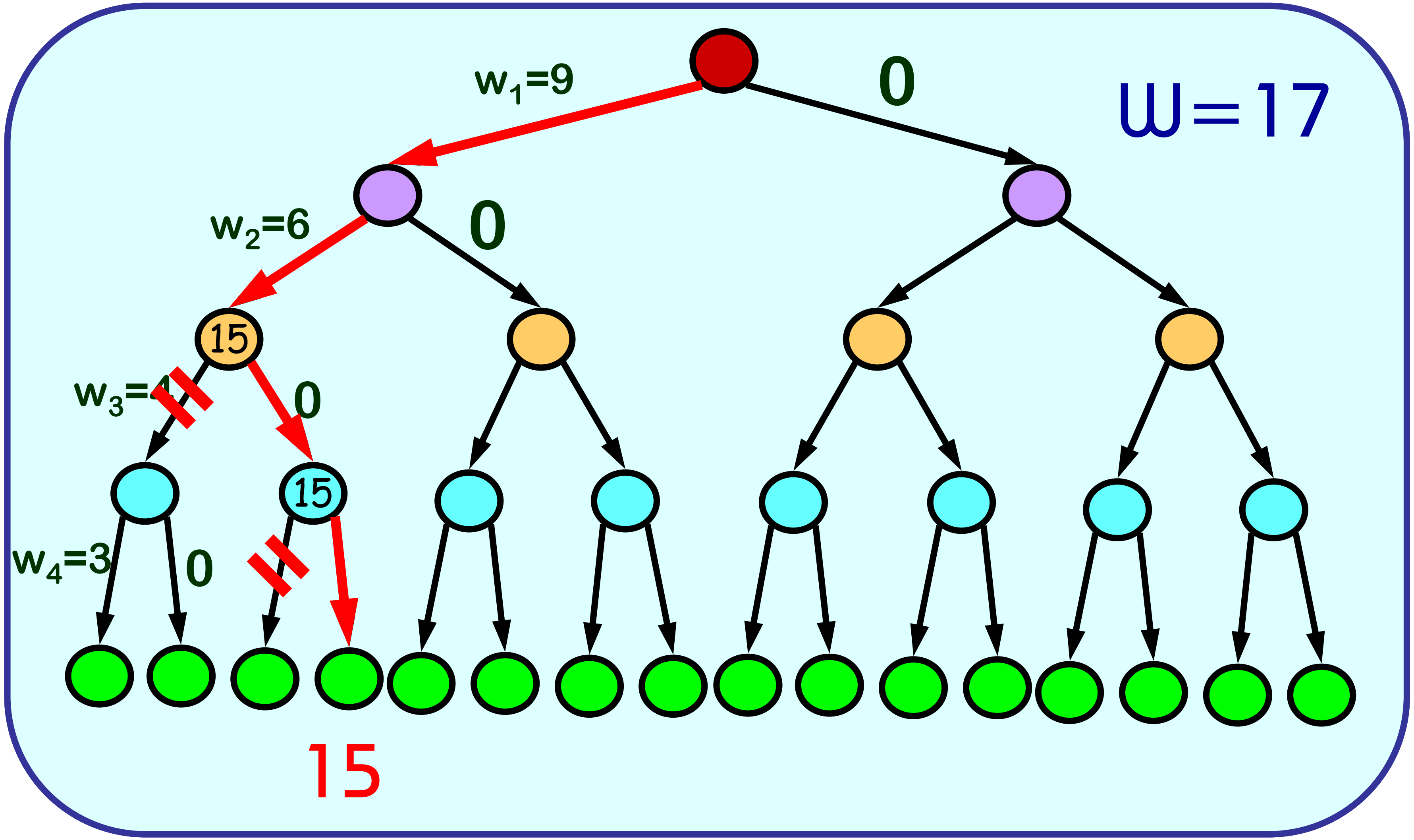
Algorithm Solve (*current_load, i*)

```
1.  if i = n then
2.      if current_load > current_best then
3.          current_best ← current_load
4.  else
5.      if current_load+weight[i+1] ≤ capacity then
6.          call Solve ( current_load+weight[i+1], i+1 )
7.      call Solve ( current_load, i+1 )
```

Initial Call ()

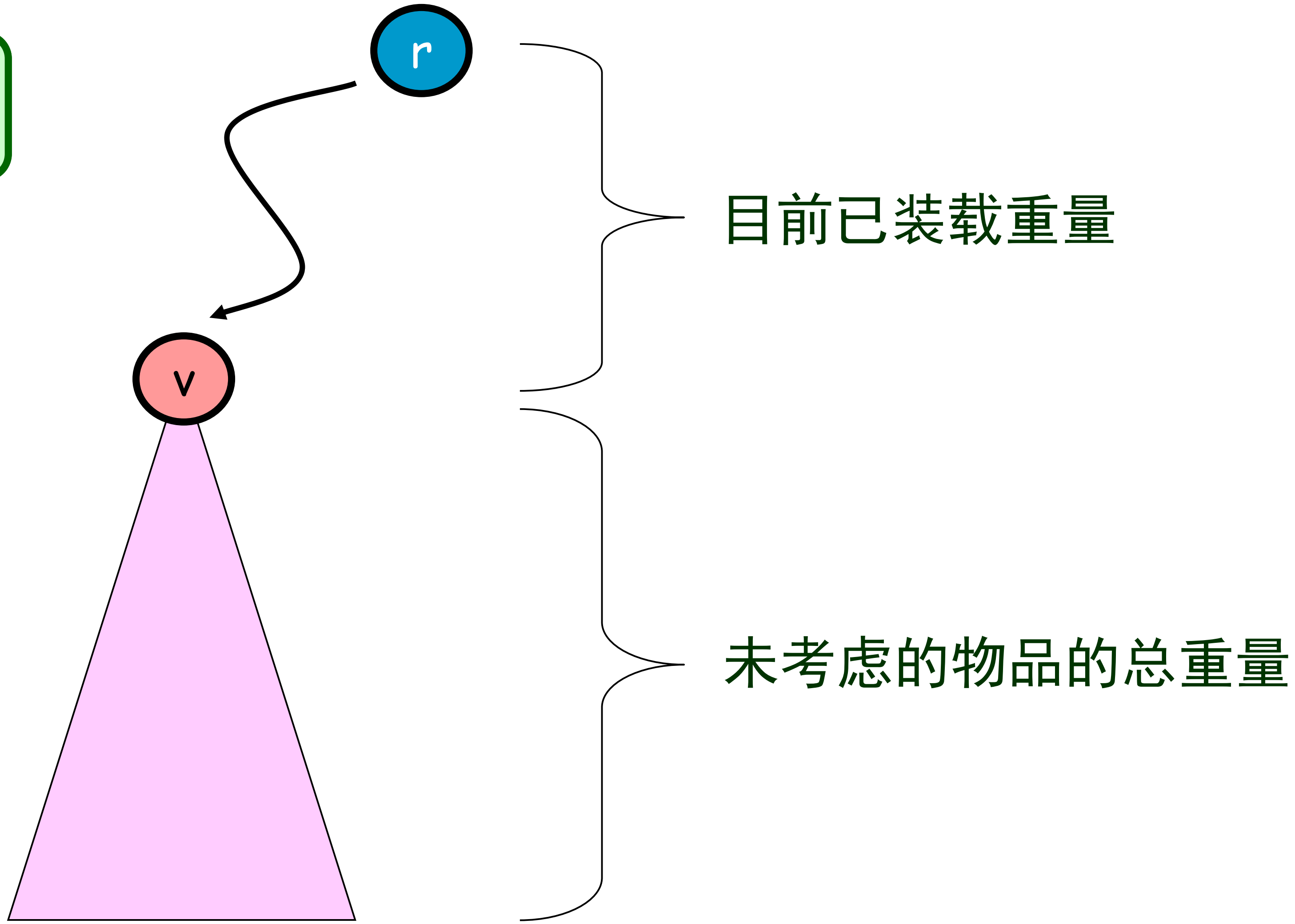
```
1.  current_best ← −∞        //也可以初始化为 0
2.  call Solve ( 0, 0 )
```

装载问题



装载问题

目标为求最大值
对上界进行估计



装载问题

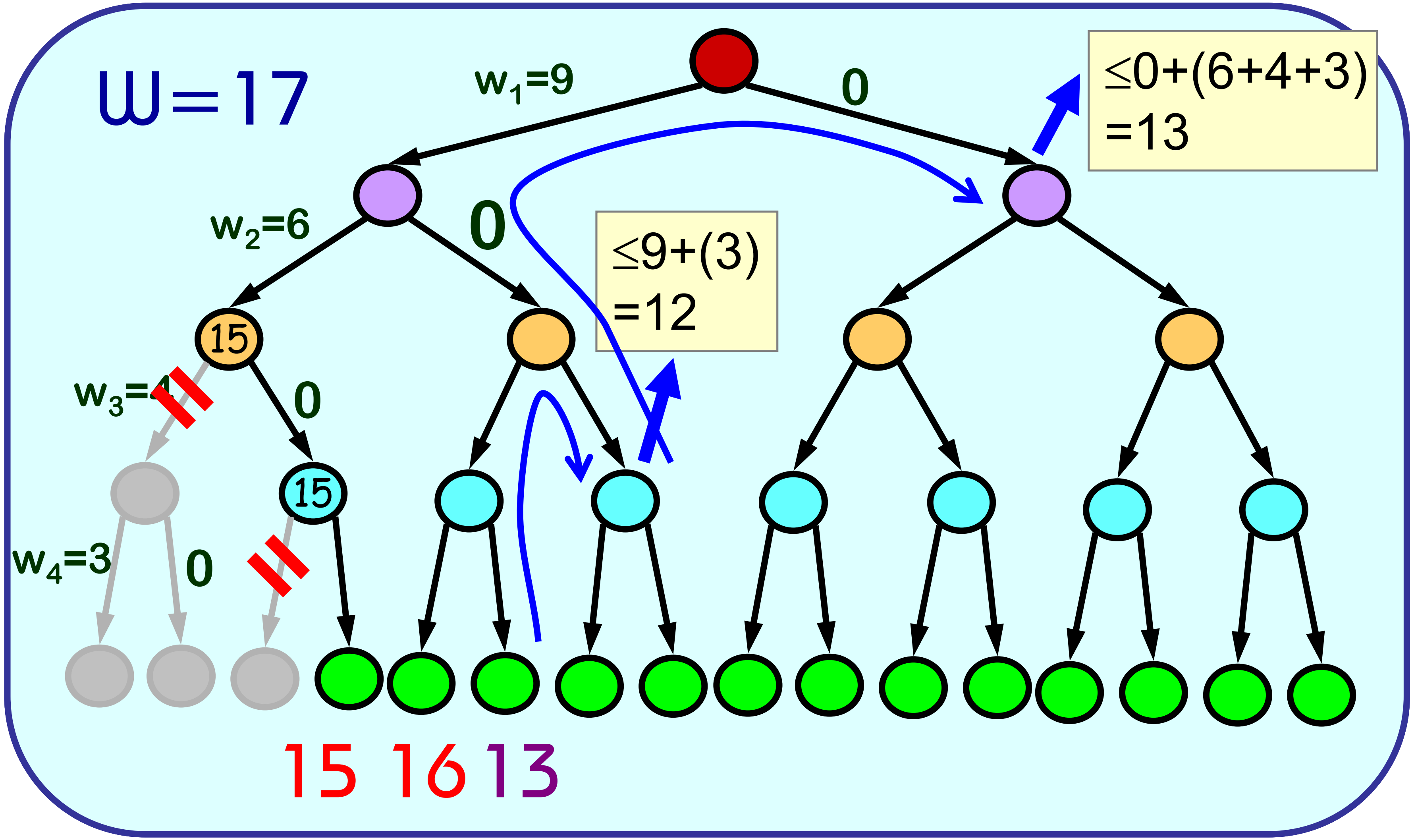
- 类似于子集和问题，可以增加一个判则：
  - 如果和 $current\_load + w_{i+1} + \ldots + w_n$ 严格小于 $current\_best$，那么这是一个无前景顶点

装载问题

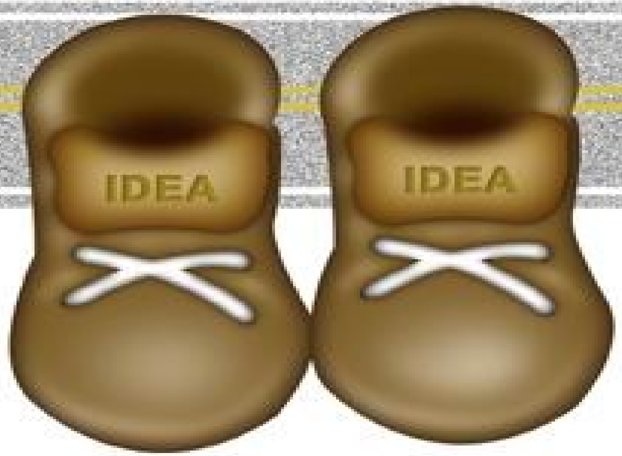
```
Algorithm Solve ( current_load, i )
if i = n then
        if current_load > current_best then
                current_best ← current_load
 else
        if current_load + total[i+1] > current_best then
                if current_load+weight[i+1] ≤ capacity then
                        call Solve ( current_load+weight[i+1], i+1 )
                call Solve ( current_load, i+1 )

Initial Call ()
current_best ← −∞    //也可以初始化为 0
call Solve ( 0, 0 )
```

装载问题



例 5.8



0-1背包问题

0-1 Knapsack Problem

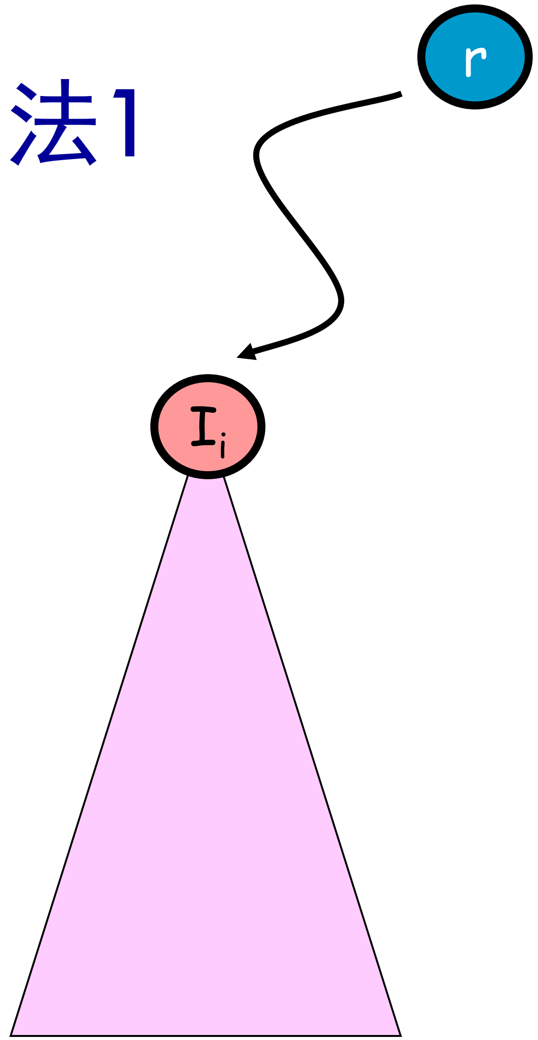
刘铎
liuduo@bjtu.edu.cn

0-1背包问题

- 示例
  - $W = 16, n = 4,$
  - $w = [2, 4, 6, 10],$
  - $v = [16, 10, 18, 22],$
  - $t = v/w = [8, 2.5, 3, 2.2]$

0-1背包问题

估计方法1



当前已取得的总价值

对物品以 *t*()
的不增顺序进行排序

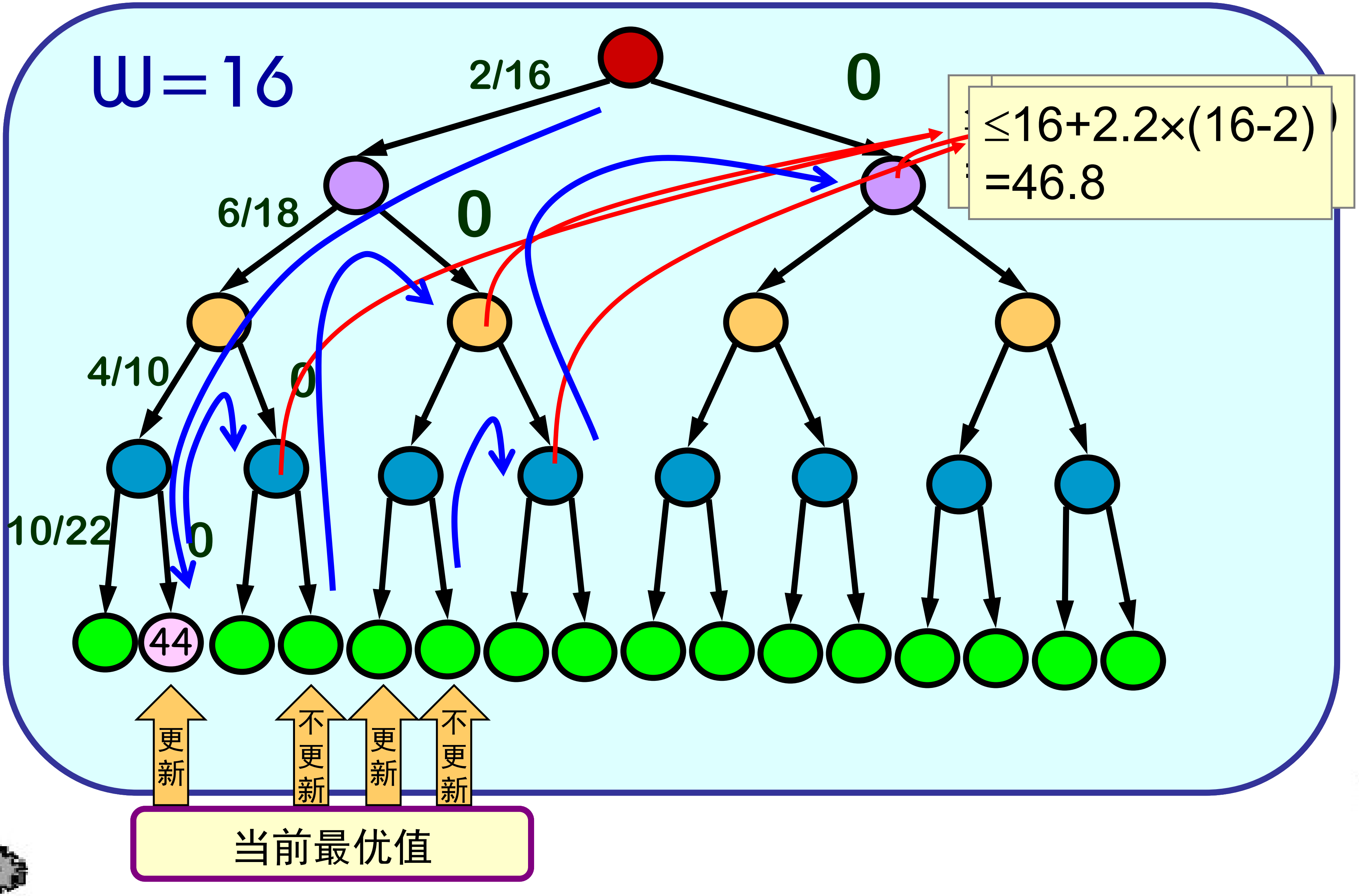
$(W-current\_weight) \times t(i+1)$

目标为求最大值
对上界进行估计

0-1背包问题

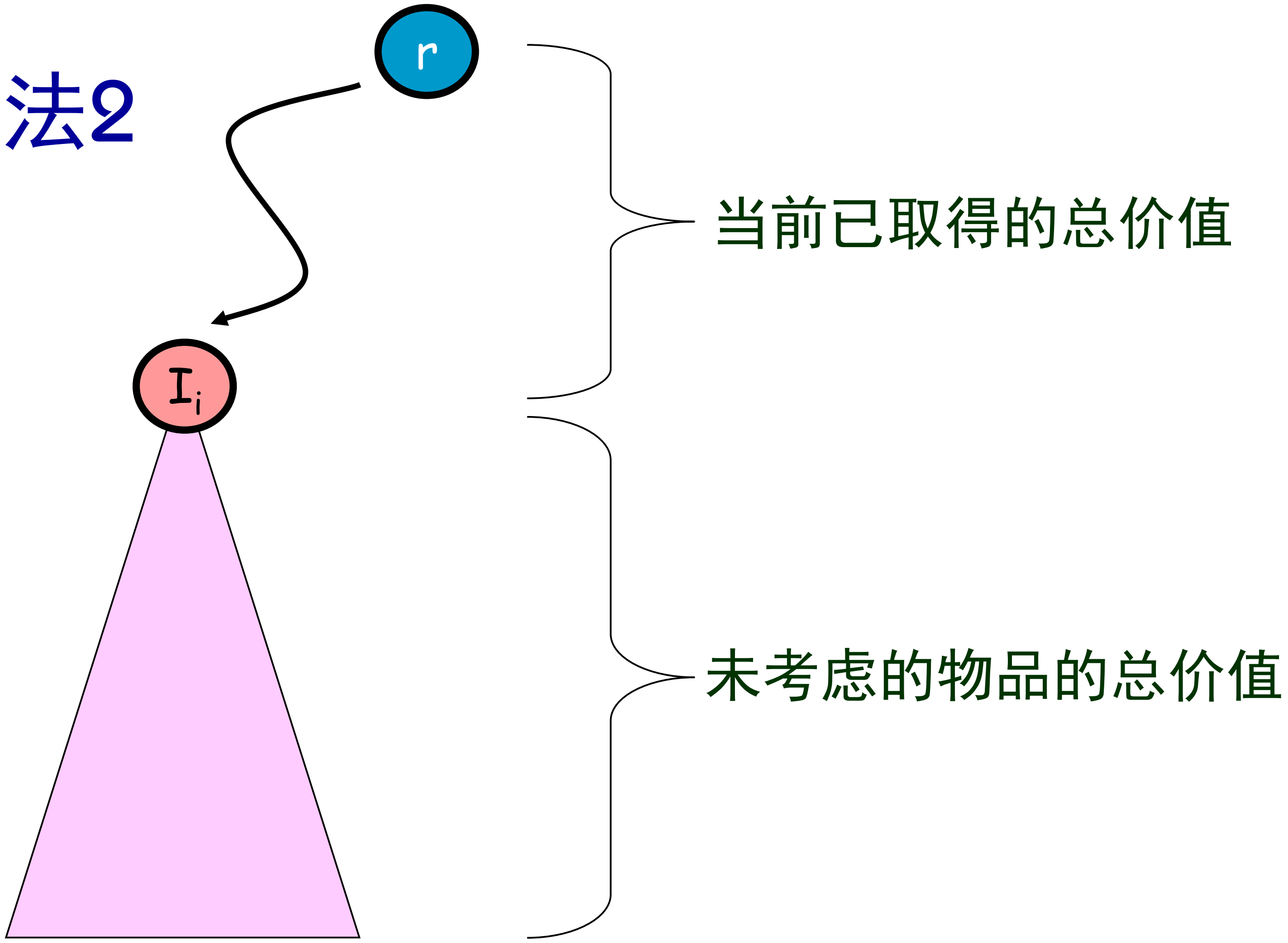
- 示例
  - $W = 16$, $n = 4$,
  - $w = [2, 6, 4, 10]$,
  - $v = [16, 18, 10, 22]$,
  - $t = v/w = [8, 3, 2.5, 2.2]$

0-1背包问题



0-1背包问题

估计方法2

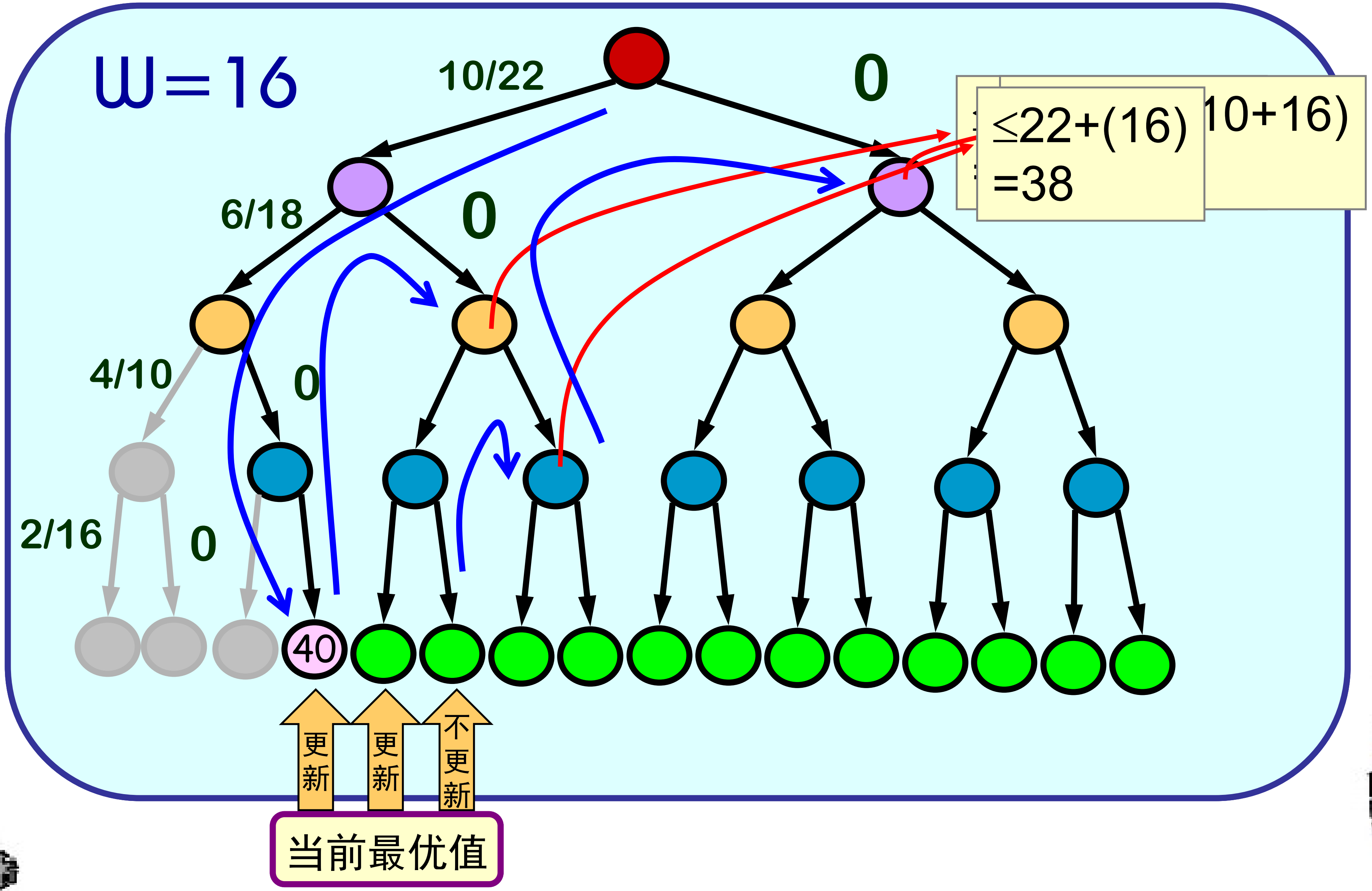


目标为求最大值
对上界进行估计

0-1背包问题

- 示例
  - $W = 16$, $n = 4$,
  - $w = [10, 6, 4, 2]$,
  - $v = [22, 18, 10, 16]$

0-1背包问题

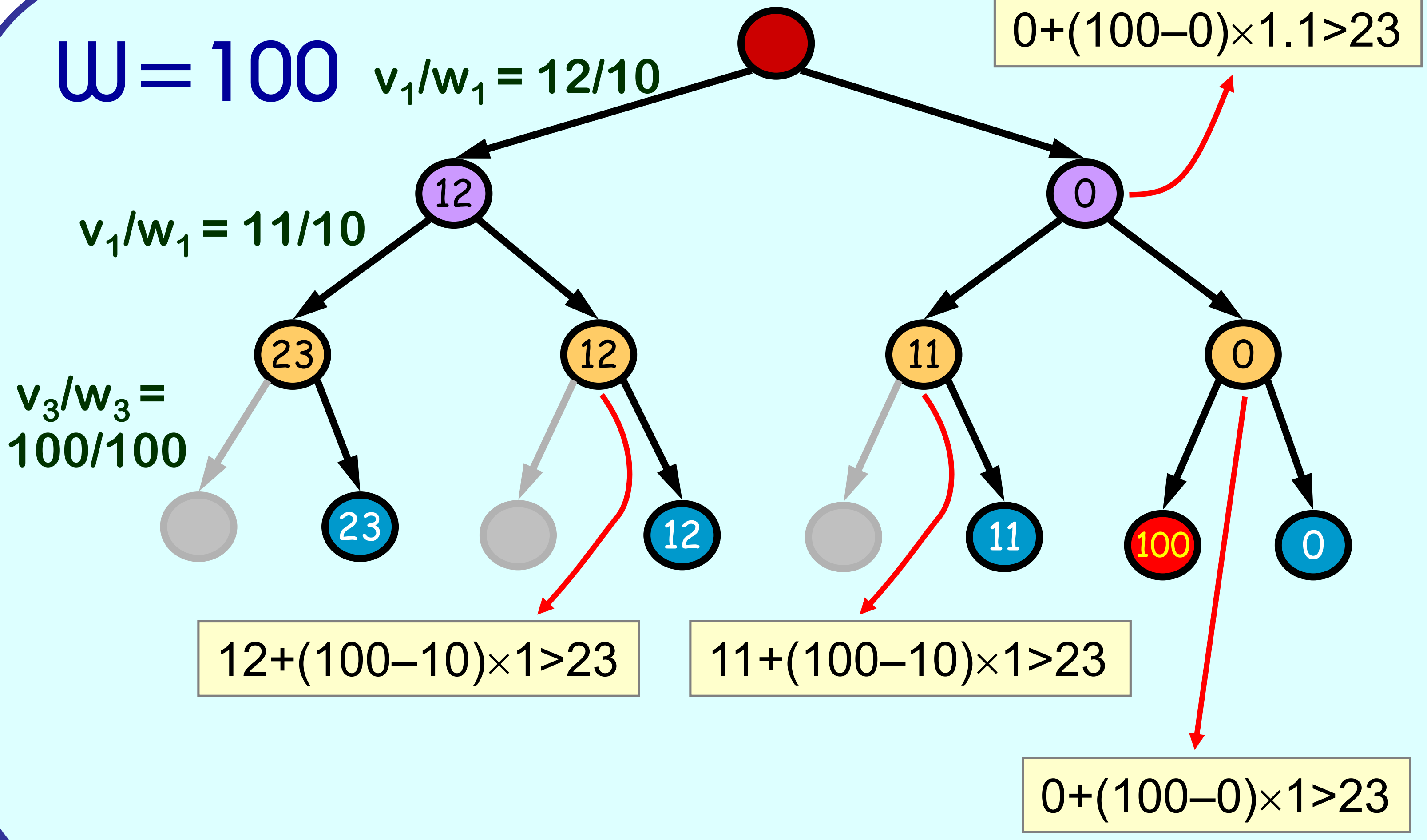


0-1背包问题

- 对同一问题可以设计不同的估界方法
- 不同估界方法在同一实例上的表现可能有所不同
- 同一个估界方法在不同实例上的表现也可能有所不同

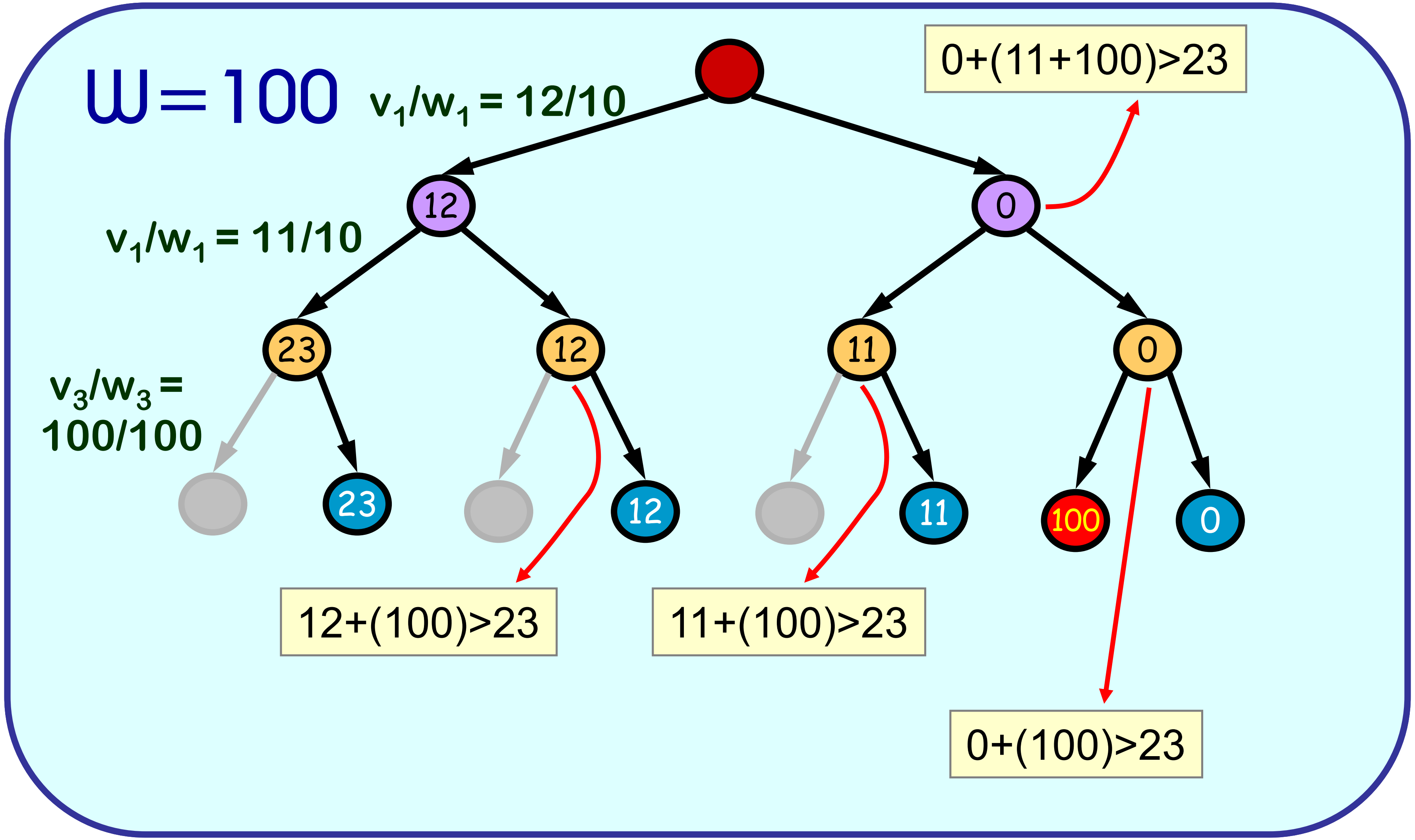
- 分支限界不能保证在所有实例上都有很好的剪枝效果

0-1背包问题



采用估计方法一

0-1背包问题



采用估计方法二

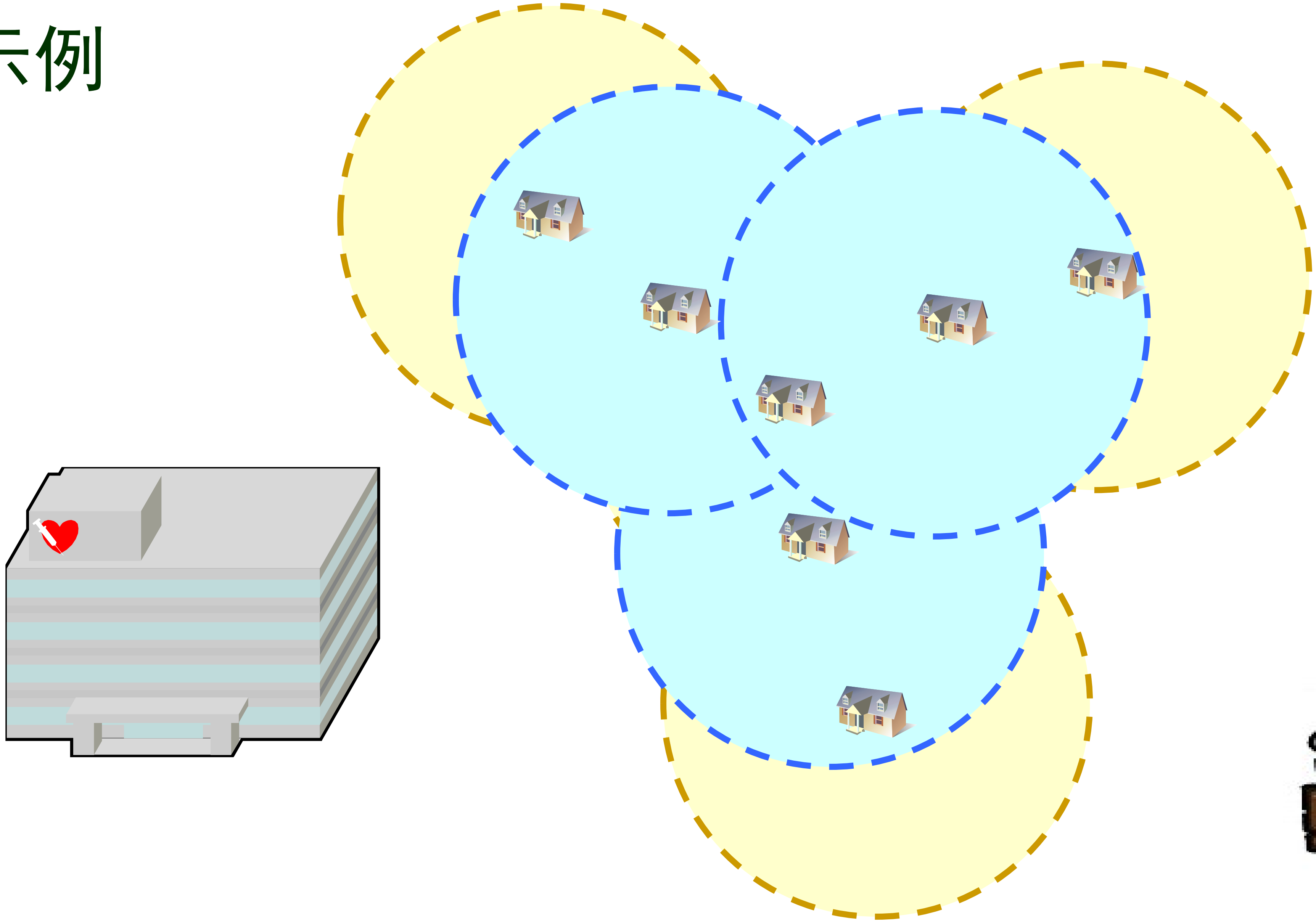
A

集合覆盖问题

Set Cover Problem

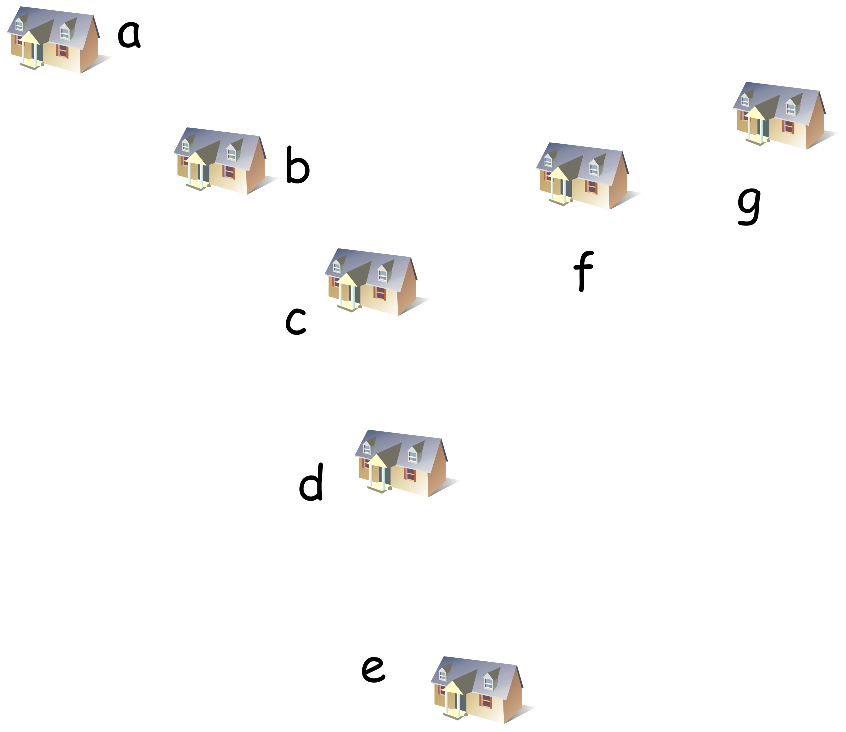
集合覆盖问题

- 应用示例



集合覆盖问题

- 示例
  - $S_a = \{ a, b \}$
  - $S_b = \{ a, b, c \}$
  - $S_c = \{ b, c, d, f \}$
  - $S_d = \{ c, d, e \}$
  - $S_e = \{ d, e \}$
  - $S_f = \{ c, f, g \}$
  - $S_g = \{ f, g \}$



集合覆盖问题

- 集合覆盖问题是计算机科学和复杂性理论中的一个经典问题。
  - 它（的判定性版本）是Karp在1972年提出的**21**个**NP**完全问题之一
- 给定一个基础集合 ***S*** 以及它的一些子集，从中选取若干个子集，使得它们的并集恰好是 ***S***，而且选取的子集数目要尽可能小

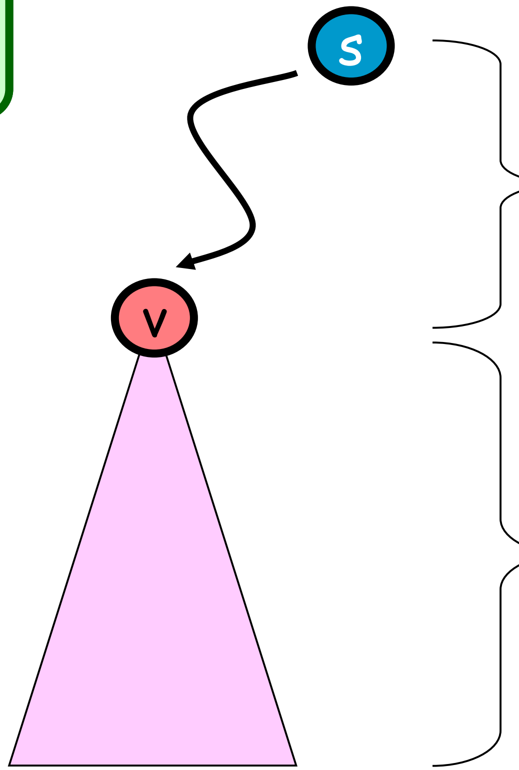
集合覆盖问题

- 示例
  - $S_a = \{ a, b \}$
  - $S_b = \{ a, b, c \}$
  - $S_c = \{ b, c, d, f \}$
  - $S_d = \{ c, d, e \}$
  - $S_e = \{ d, e \}$
  - $S_f = \{ c, f, g \}$
  - $S_g = \{ f, g \}$
- 初始的界（当前的最优值）

  ＝贪婪策略解

集合覆盖问题

目标为求最小值
对下界进行估计



C_a: 已选择的子集数

C_e: ⌈未被覆盖的元素数 / 最大子集的基数⌉

分支限界法

- 何时使用穷竭式搜索？
  - 问题规模非常小
  - 生成一个候选解和检验一个候选解是否可行/有效都非常容易
- 何时使用分支限界？
  - 最优化问题
  - 想不出更好的算法
  - 穷竭式搜索不现实

分支限界法

- 此处介绍的都只是可分解为多步骤的比较简单的问题
  - 学习算法框架使用
- 涉及到图搜索的问题可能会更加复杂些
- 除基于DFS的方法外，还有一些其他的扩展顶点方法和搜索算法
  - 例如A*算法等

End