状态空间搜索

Youtube频道： **hwdong**

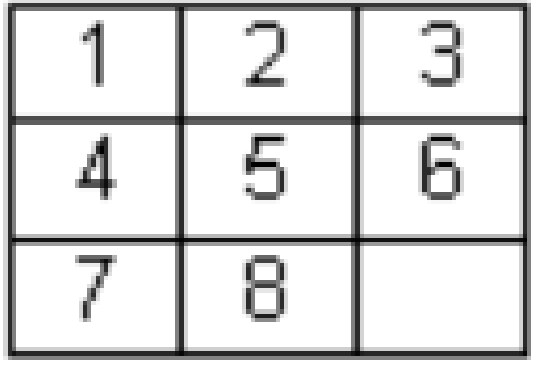
博客： **hwdong-net**.github.io

状态空间搜索

- 状态空间搜索是人工智能 (AI) 和计算机科学中使用的一种解决问题的技术，通过探索可以导致解决方案的各种状态或配置来找到问题的解决方案。
- 在状态空间搜索中，问题被表示为一组状态，搜索算法试图通过探索和评估每个状态的潜在移动来找到从初始状态到目标状态的路径。

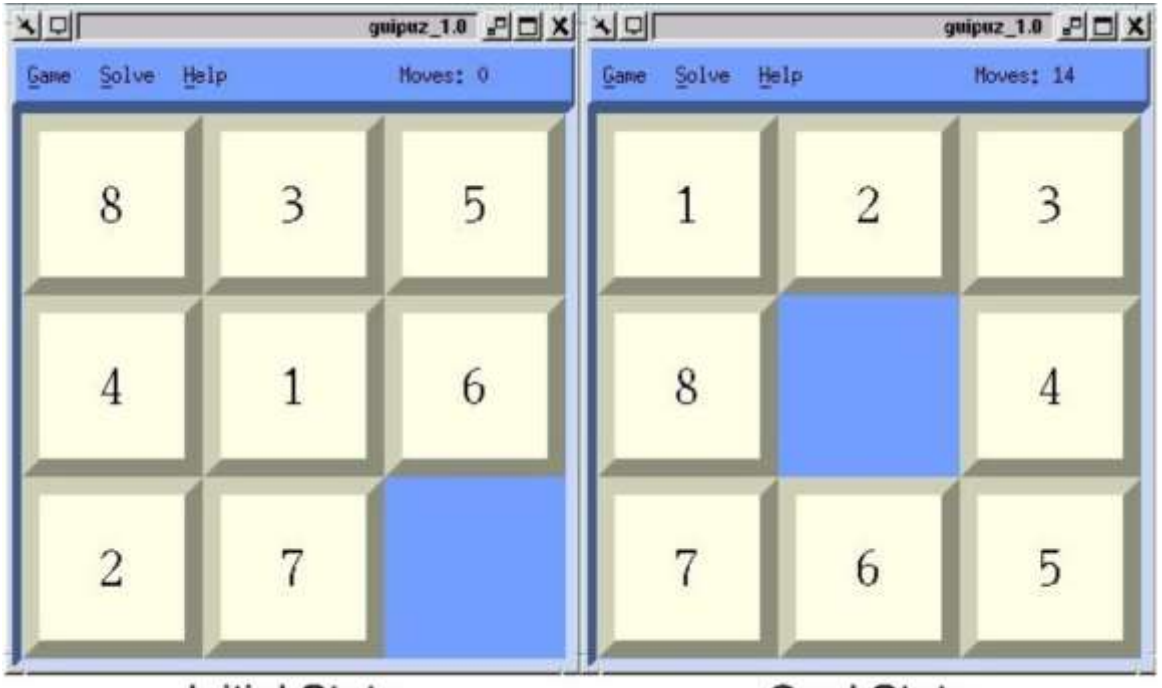
8-puzzle 问题

- 8-puzzle 是一种简单的游戏，由 3 x 3 的网格（包含 9 个方块）组成。其中一个方格是空的。



8-puzzle 问题

- 目的是通过移动方块，从初始配置(状态)到达目标配置(状态)。

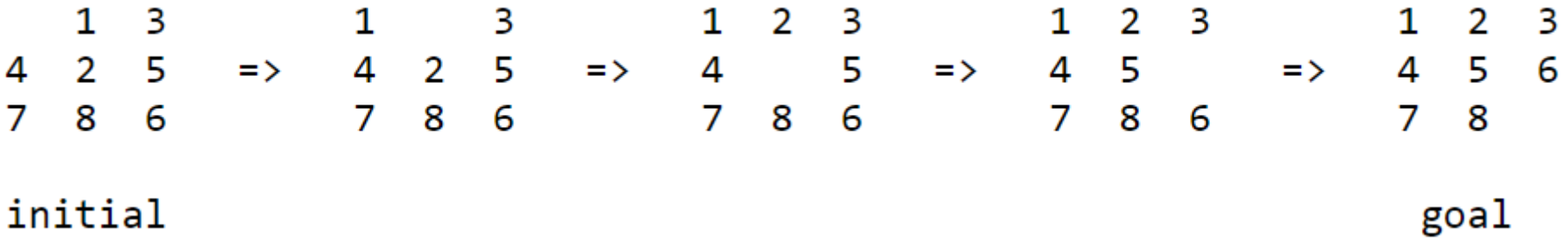


8-puzzle 问题

- 目的是通过移动方块，从初始状态到达目标状态。

8-puzzle 问题

- 目的是通过移动方块，从初始状态到达目标状态。

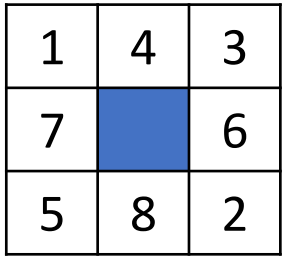


多步决策

- 状态空间搜索是一个多步决策过程，通过一系列步骤寻找问题的解。从一个初始状态开始，在每一步的状态下探索不同的可能性过渡到不同状态，直到达到目标状态。

8-puzzle （8-数码问题）

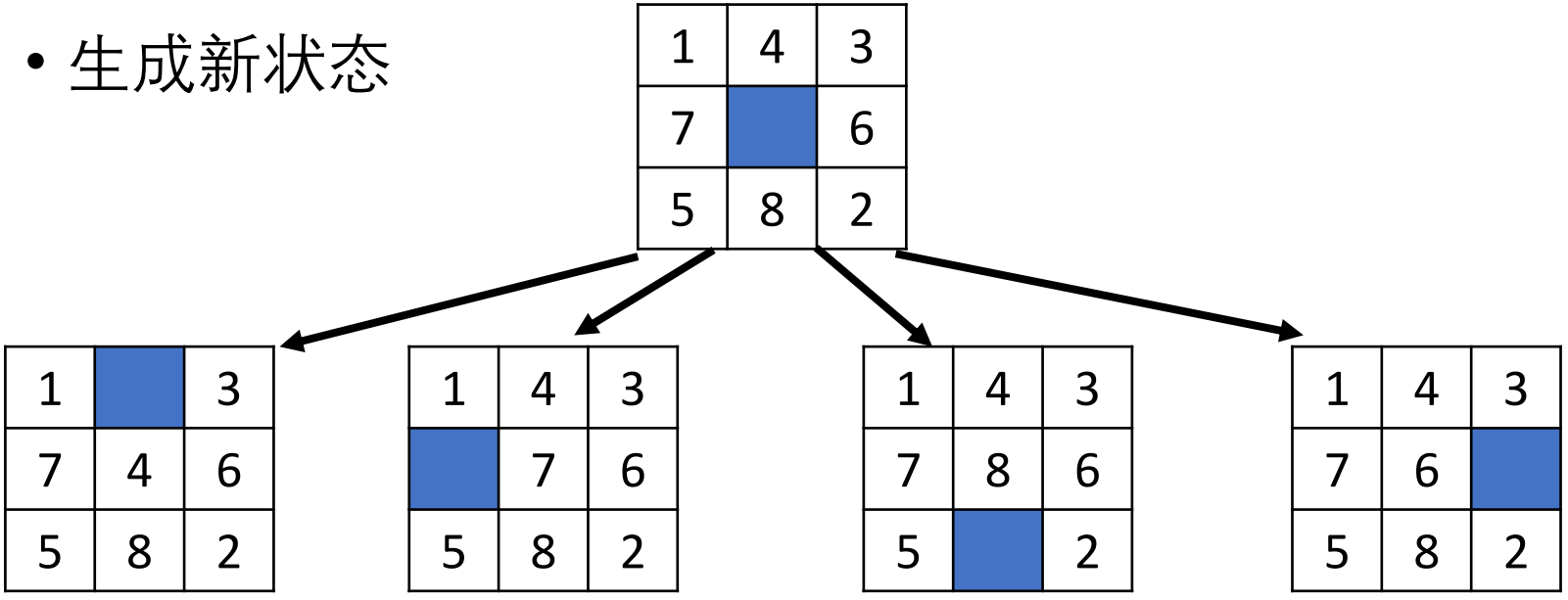
- 初始状态



状态就是 3*3的网格，每个网格是1到8数字或空格

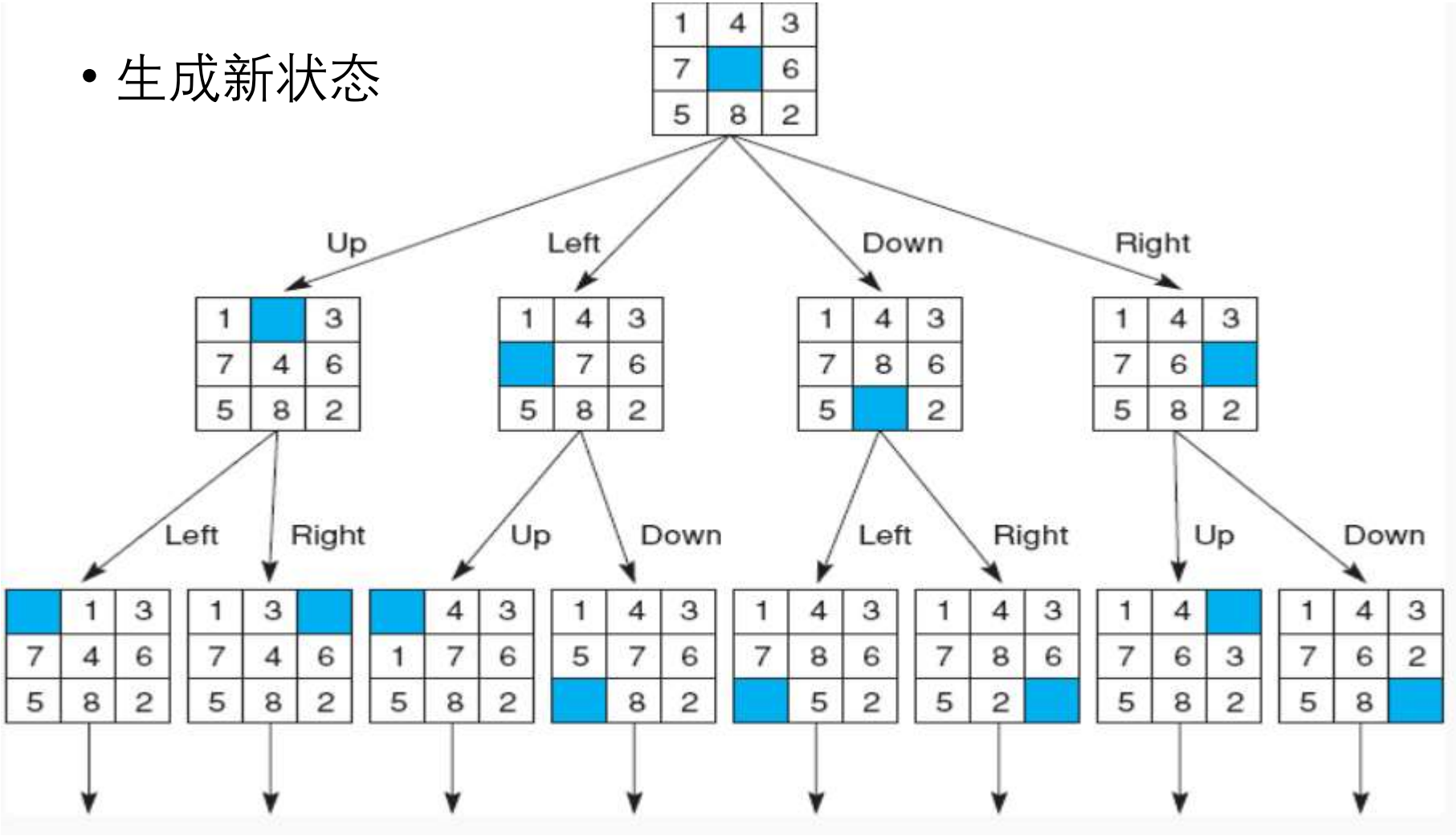
8-puzzle （8-数码问题）

- 生成新状态



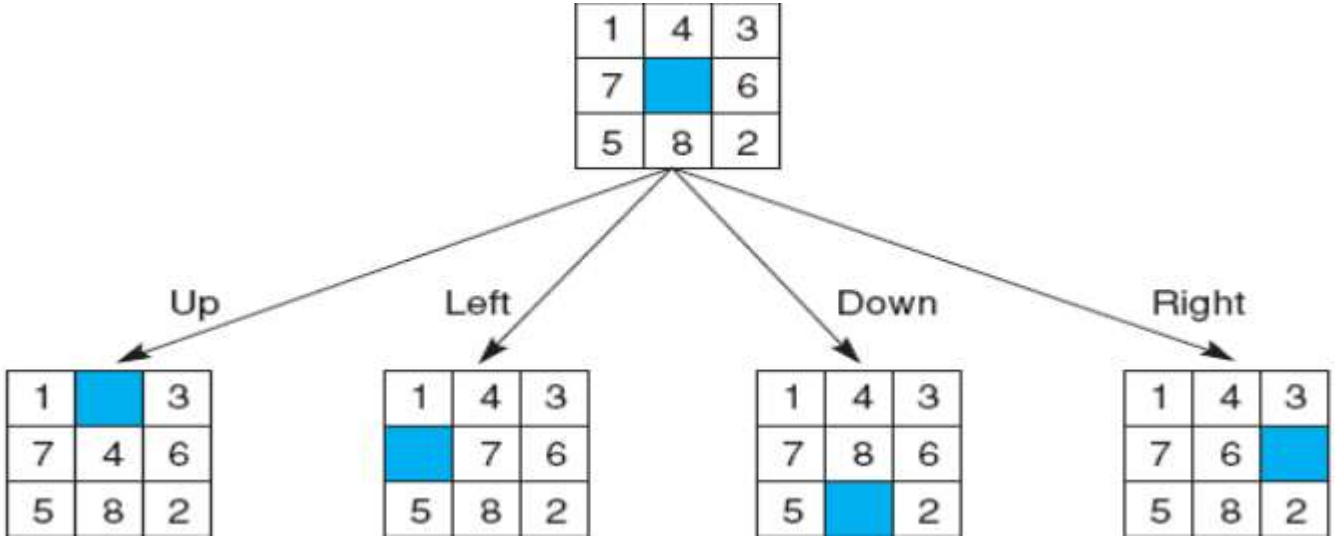
8-puzzle （8-数码问题）

- 生成新状态



状态空间是图形结构

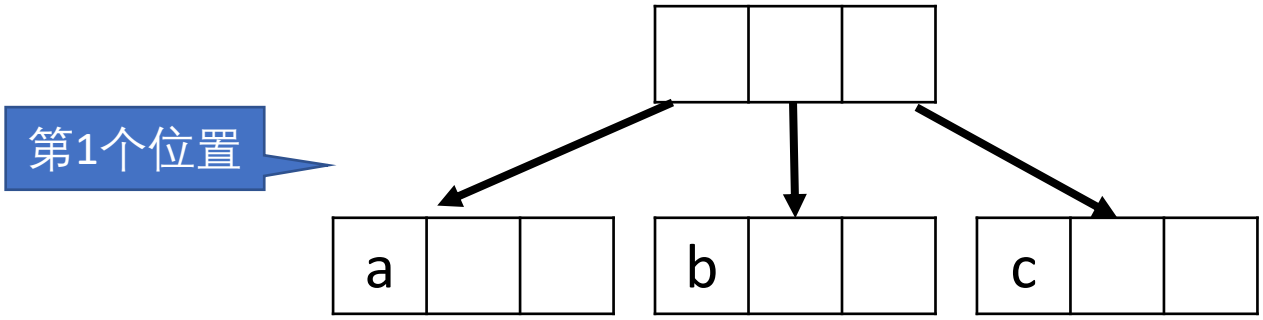
- 状态之间的关系可以表示为图（或树），其中每个节点表示一个状态，边表示从一种状态到另一种状态的可能移动或转换。

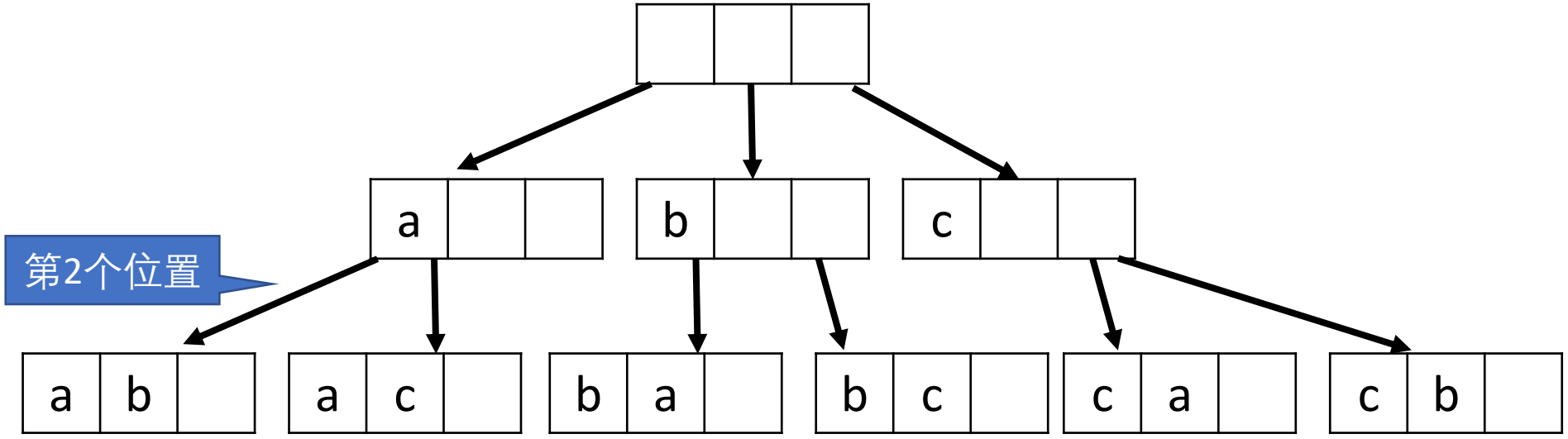


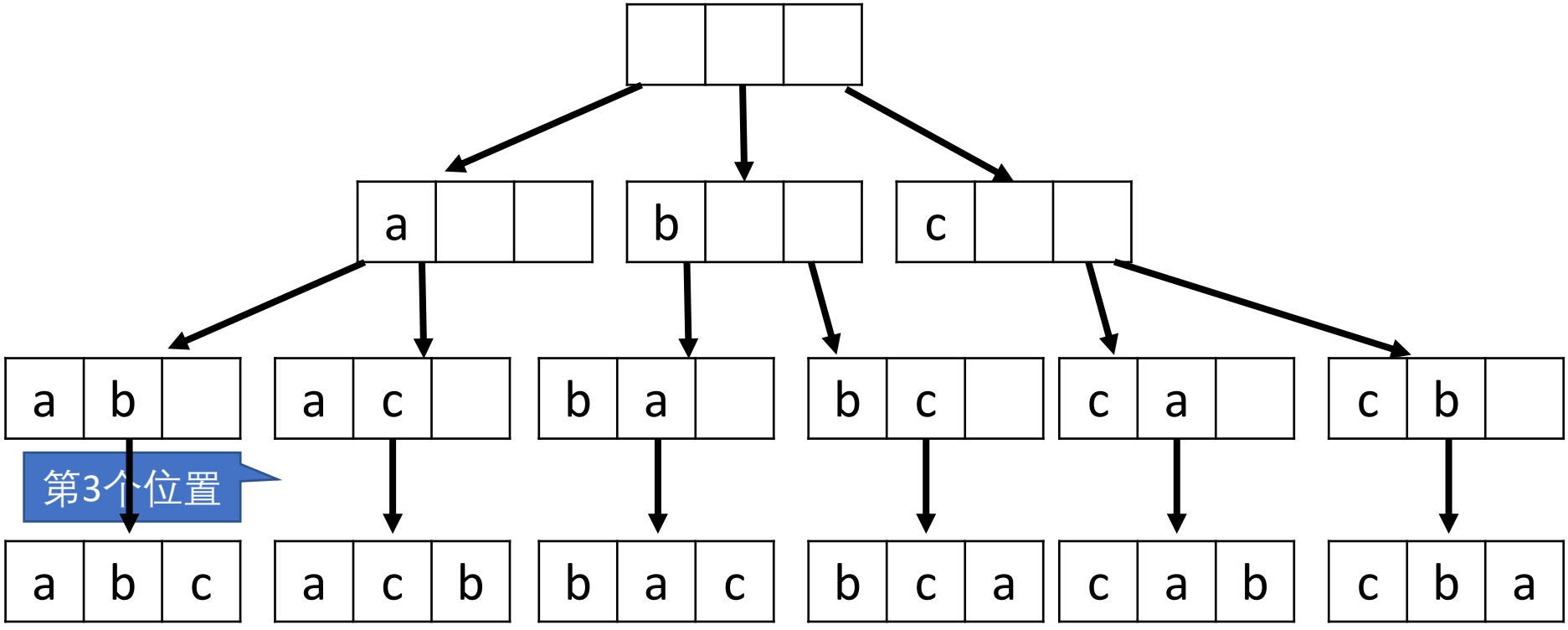
全排列

- 不含重复元素的一组元素的所有可能排列。
- 输入: a b c
- 输出:

a b c
a c b
b a c
b c a
c a b
c b a


第1个位置
a
b
c


a
b
c
第2个位置
a b
a c
b a
b c
c a
c b


a
b
c
a b
a c
b a
b c
c a
c b
第3个位置
a b c
a c b
b a c
b c a
c a b
c b a

0-1背包问题knapsack problem

- n个不同重量($w_1,w_2,\ldots,w_n$)和价值($v_1,v_2,\ldots,v_n$)的物品，要放进载重量不超过W的背包里，如何不超重情况使背包中物品价值最大？

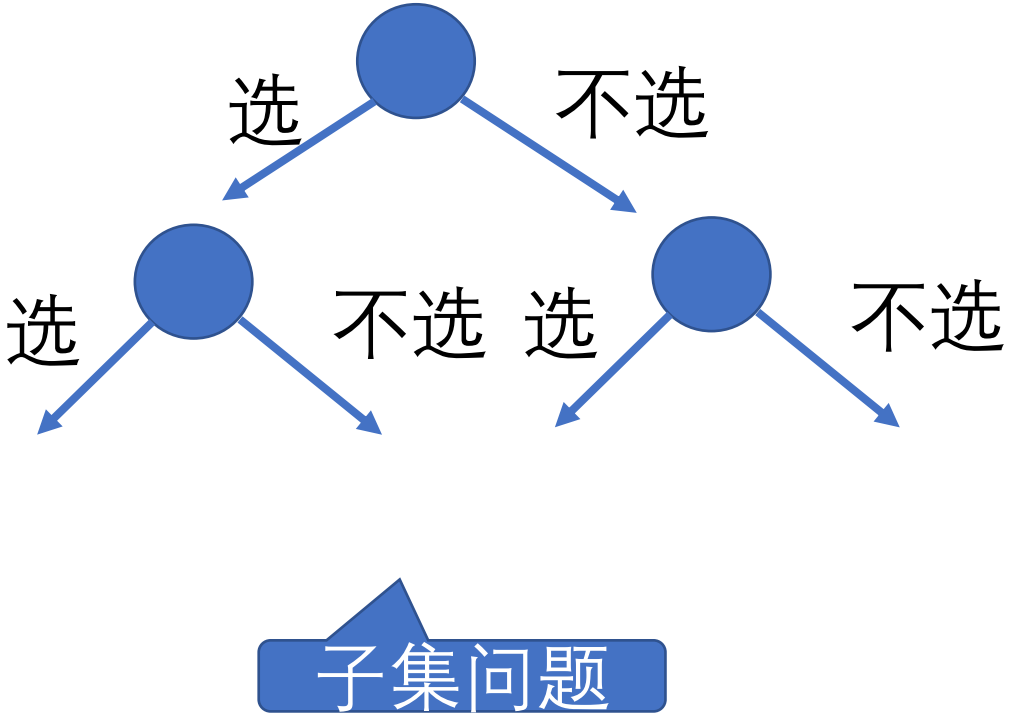
$$\max \sum_{i=1}^{n} v_i x_i$$

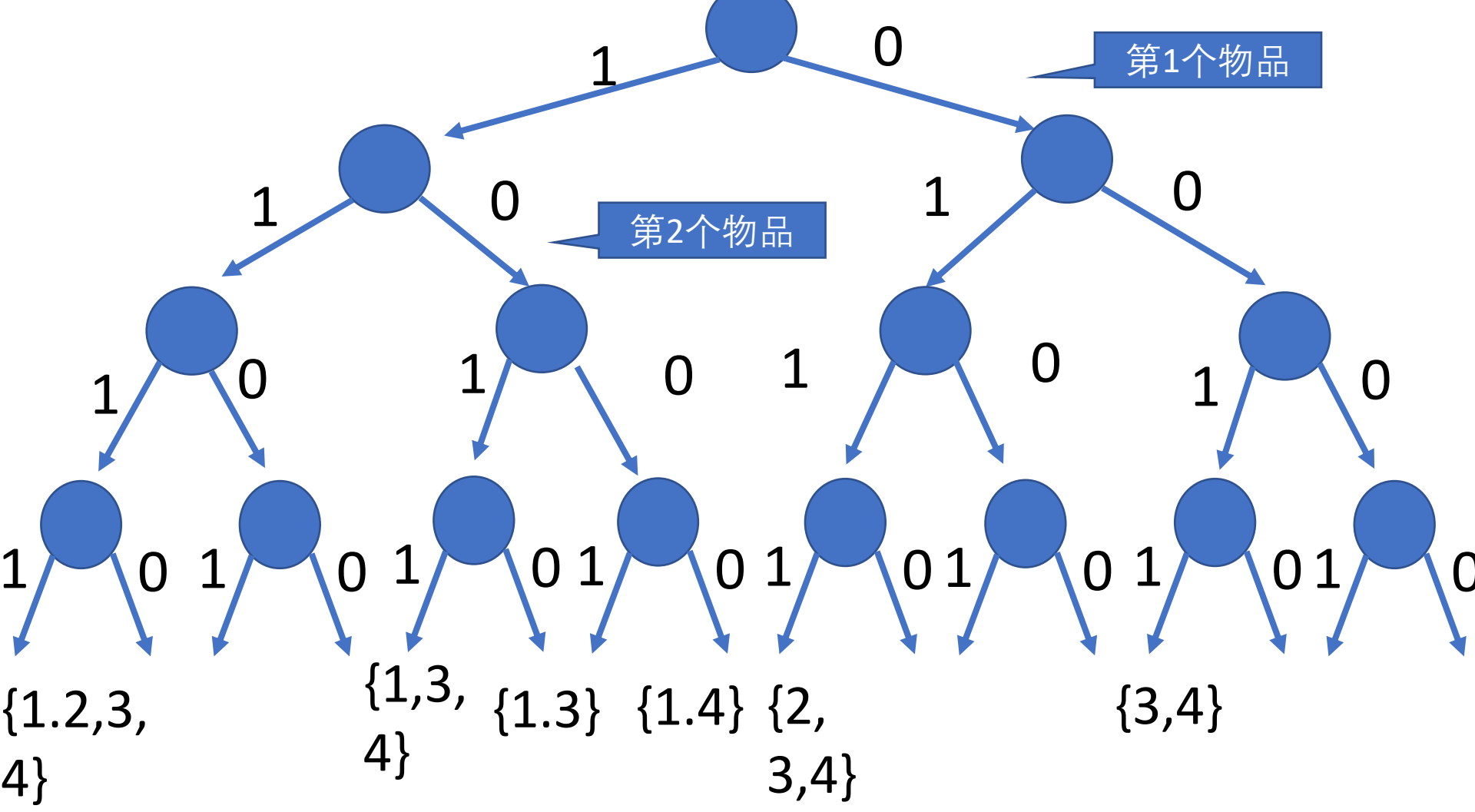
$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le W \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

- 每个物体都试试放或不放，n个物体一共有2^n种可能性

编号	重量	价值
1	2	$20
2	5	$30
3	10	$50
4	5	$10

背包最大容量：16




1
0
第1个物品
1
0
第2个物品
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
1
0
{1.2,3,
4}
{1,3,
4}
{1.3}
{1.4}
{2,
3,4}
{3,4}

子集树

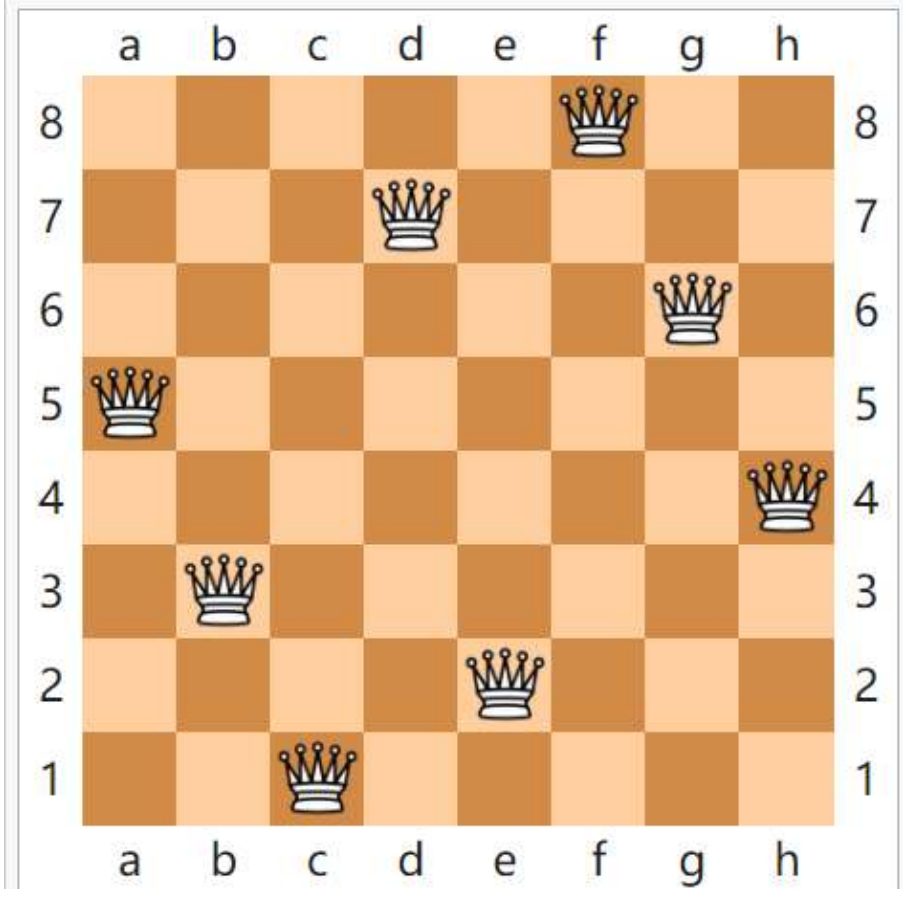
编号	重量	价值
1	2	$20
2	5	$30
3	10	$50
4	5	$10

背包最大容量： 16

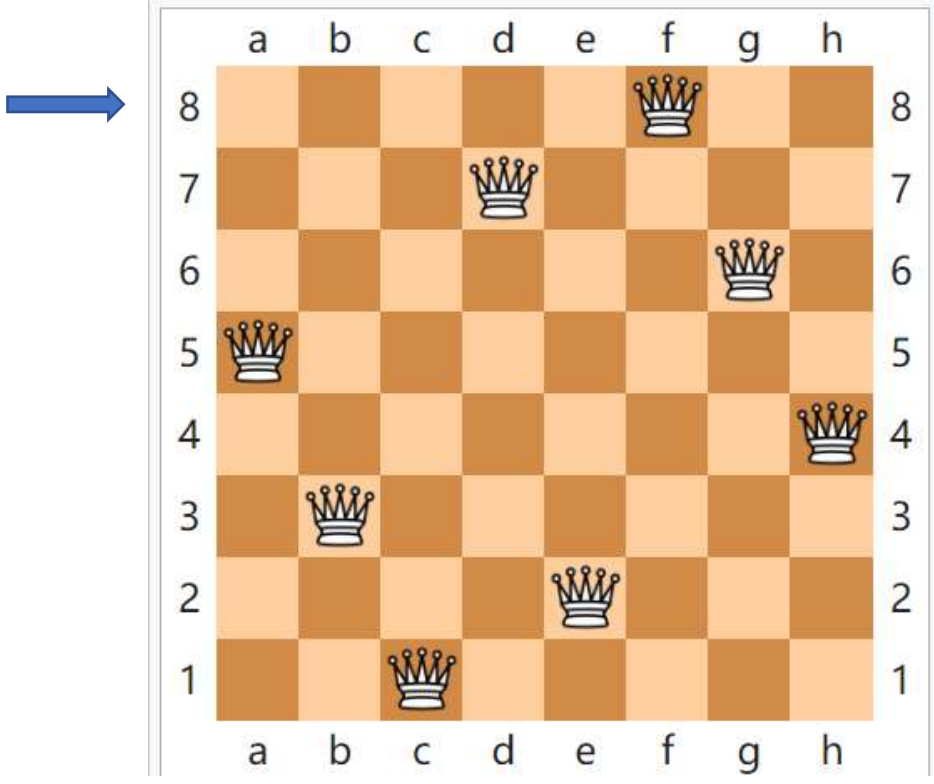
子集	总重量	总价值
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	不可行
{1,2,4}	12	$60
{1,3,4}	17	不可行
{2,3,4}	20	不可行
{1,2,3,4}	22	不可行

N皇后问题

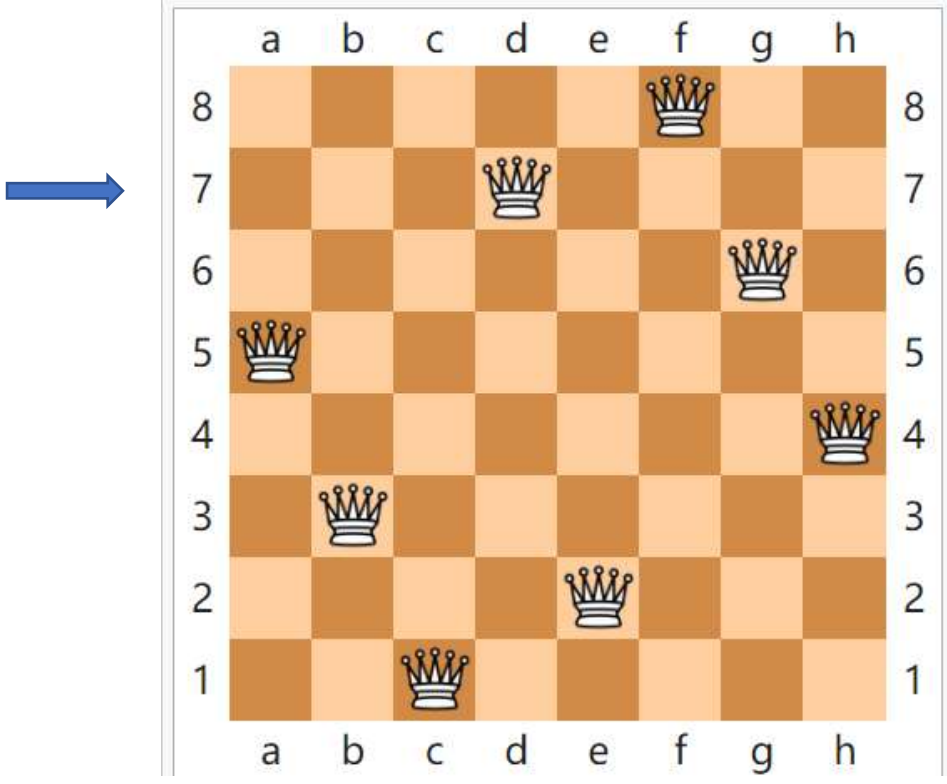
- 8*8棋盘，8个皇后，任两个皇后都不能处于同一条横行、纵行或斜线上。



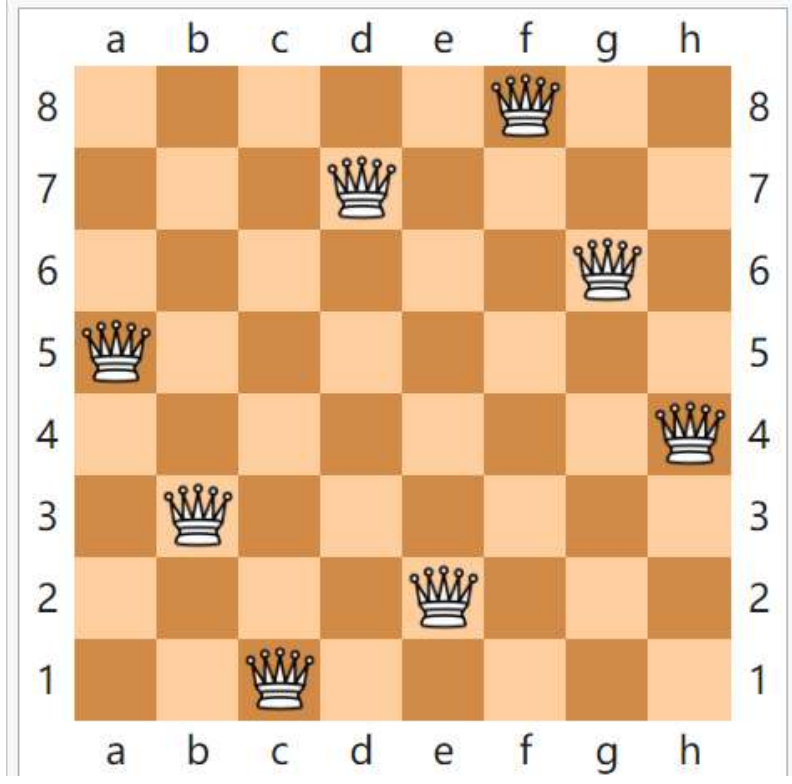
- 多步决策:
- 第 1 步：第一行皇后的位置决策



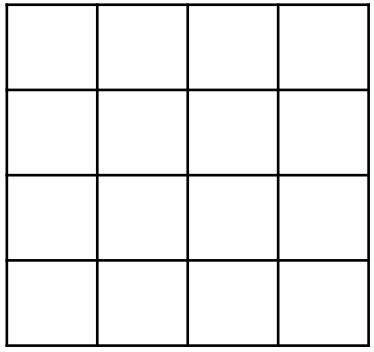
- 多步决策：
- 第 1 步：第1行皇后的位置决策
- 第 2 步：第2行皇后的位置决策



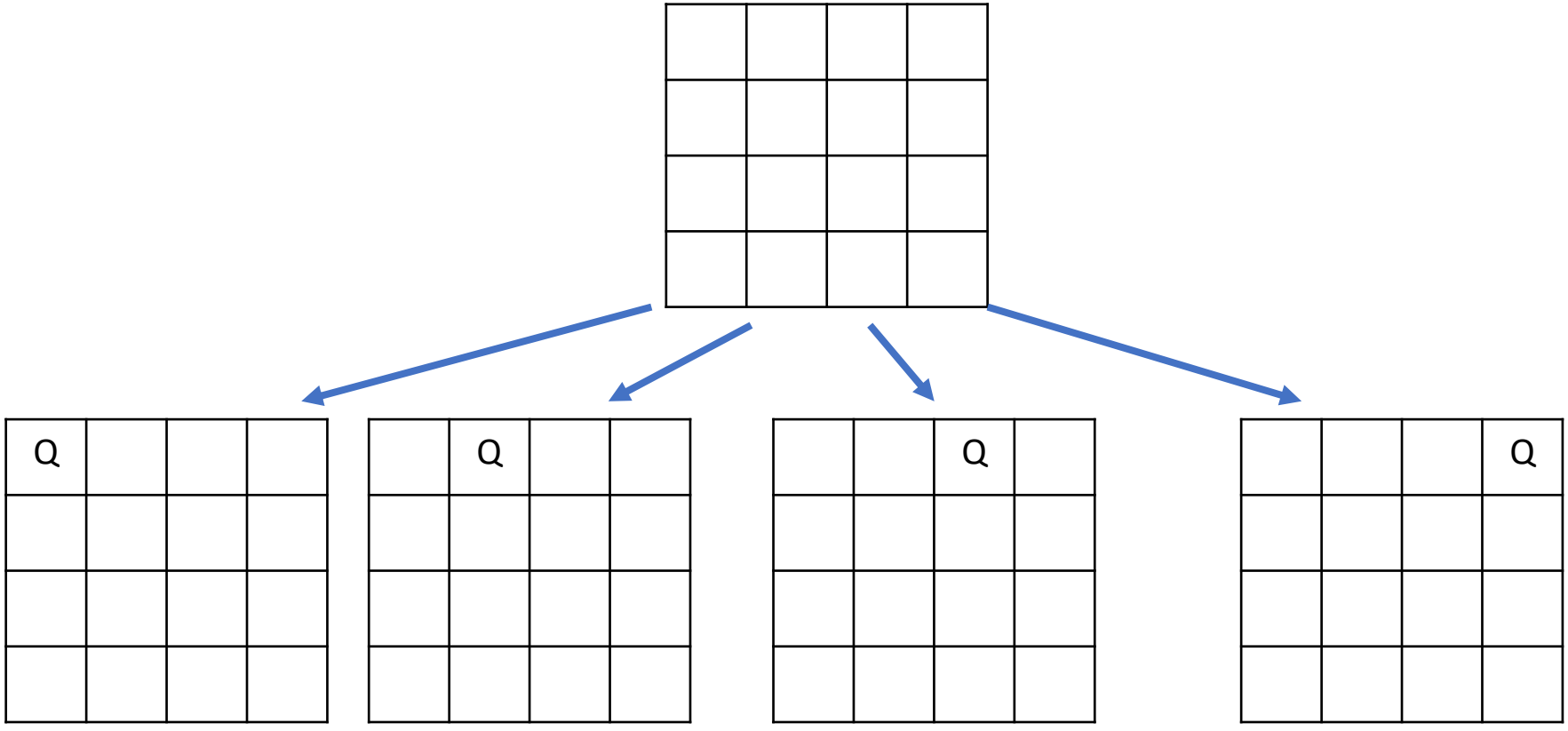
- 多步决策：
- 第 1 步：第1行皇后的位置决策
- 第 2 步：第2行皇后的位置决策
- 第 3 步：第3行皇后的位置决策

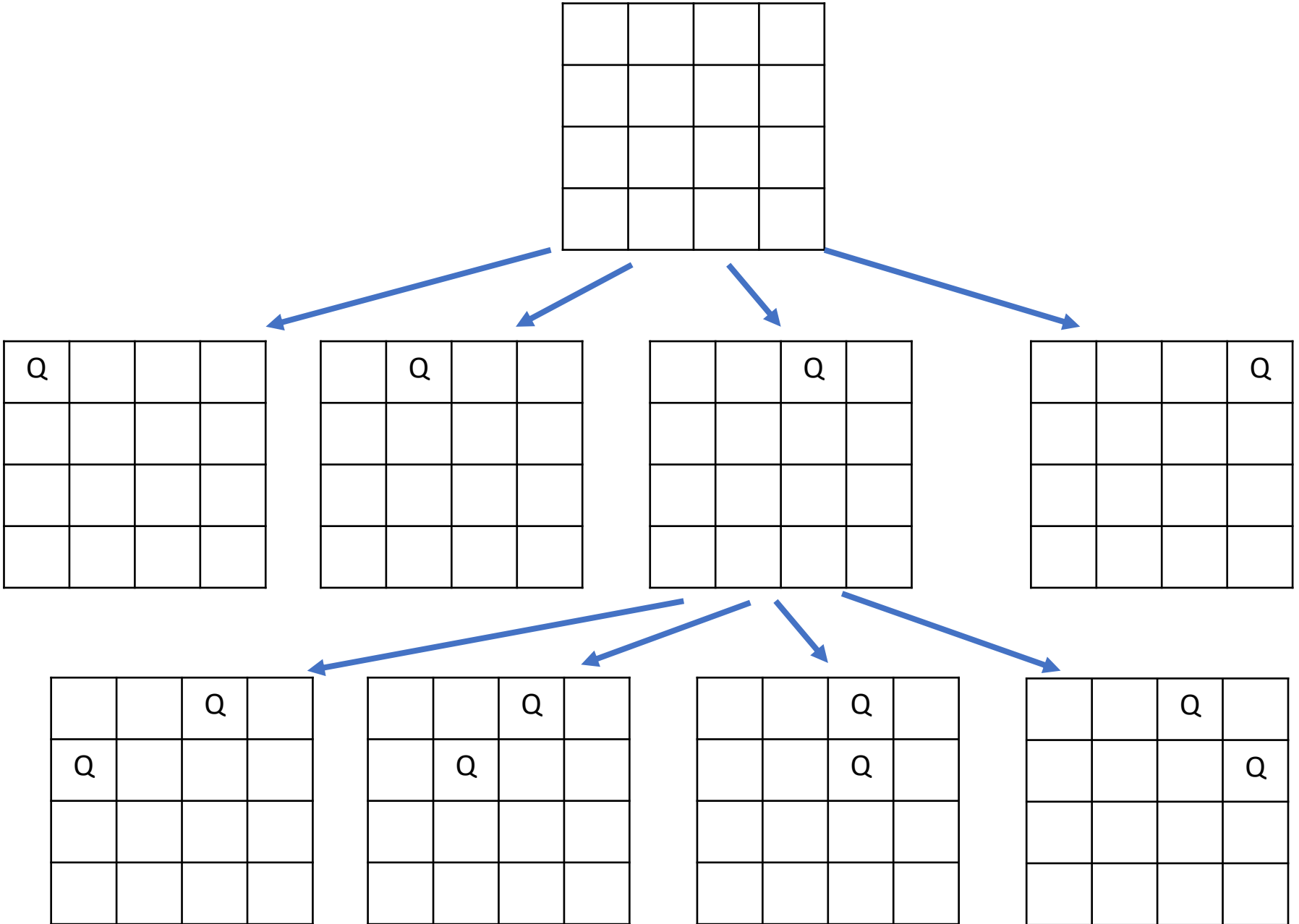


从初始**状态**出发



每一步决策：在某个**状态**，有**多个决策选项**




Q
Q
Q
Q
Q
Q
Q
Q
Q
Q
Q
Q

多步决策

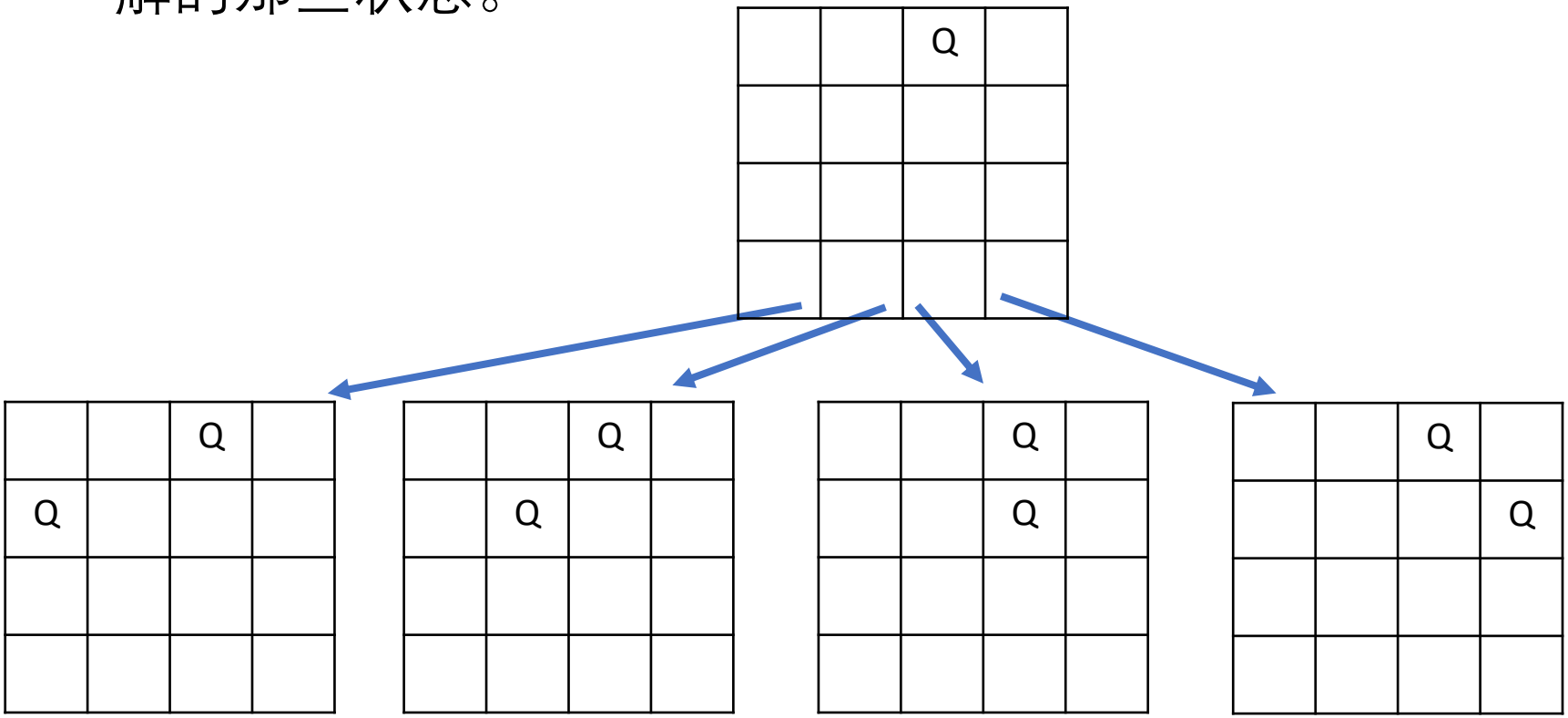
- 状态空间搜索是一个多步决策过程，通过一系列步骤寻找问题的解。从一个初始状态开始，在每一步的状态下探索不同的可能性过渡到不同状态，直到达到目标状态。

多步决策

- 全排列：第1个位置、第2个位置、...
- 0-1背包问题：对每个物品的选择：选或不选
- 8-puzzle：从初始位置，每个位置有不同的选择，经过很多次选择，到达目标位置
- 8皇后问题：第1行的皇后位置、第2行的皇后位置、...

剪枝

- 状态空间搜索可以通过**剪枝**技术避免探索不可能得到解的那些状态。



常见搜索策略

- Depth-first search (DFS)　深度优先搜索
- Breadth-first search (BFS)　广度优先搜索
- Uniform cost search (UCS)
- A* search　A*搜索
- Iterative deepening search (IDS)
- Best-first search　最佳优先搜索（贪婪法）
- Bidirectional search　双向搜索

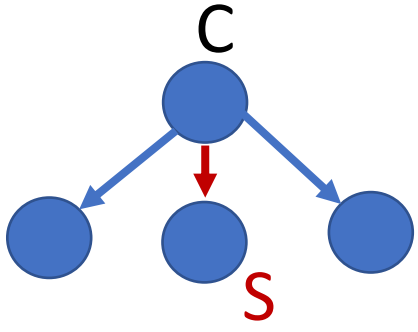
深度优先搜索(回溯法)

Depth-first search(**DFS**)

backtracking(**BK**)

DFS/BT

- 0：从一个初始状态出发
- 1：在每个当前状态，做一个决策，确定下一步怎么走?
- 2：检查所选步骤是否有效。如果无效，请撤消上一步并尝试其他步骤。
- 3：重复步骤 1 和 2，直到找到解决方案或用尽所有可能性。



深度优先搜索：是一个递归过程

DFS(C):

标记当前状态C已访问

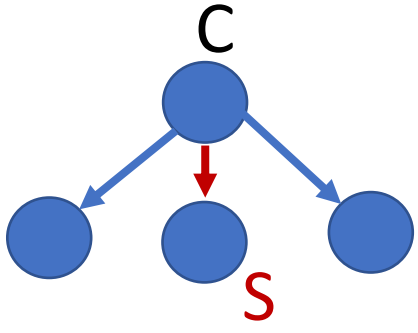
if C是目标状态：输出解，return

处理当前状态C

for(C的后续未访问状态S)

DFS(S)

回溯



解答路径：状态中包含了路径

DFS(C):

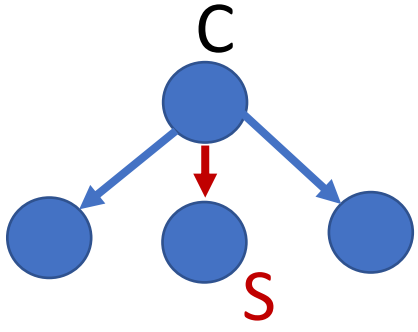
标记当前状态C已访问

if C是目标状态：输出解，return

处理当前状态C

for(C的后续未访问状态S)

DFS(S)



解答路径：子状态指向父状态

DFS(C):

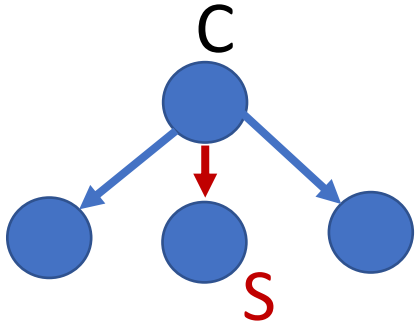
标记当前状态C已访问

if C是目标状态：输出解，return

处理当前状态C

for(C的后续未访问状态S)

DFS(S)



解答路径：用堆栈保存

DFS(C):

标记当前状态C已访问

if C是目标状态：输出解，return

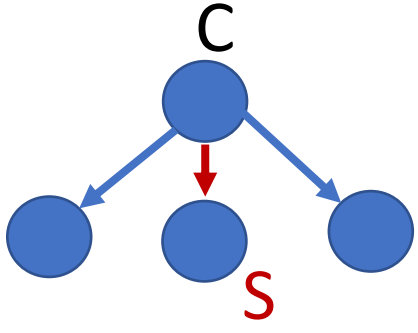
处理当前状态C

for(C的后续未访问状态S)

path.push(S) //进入

DFS(S, path)

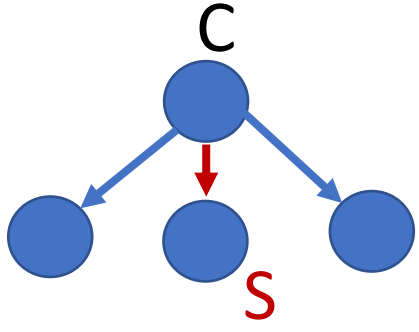
path.pop(); //回到C



path记录决策过程的每一步。

解答路径：子问题返回子问题解

```
DFS(C):
    标记当前状态C已访问
    if C是目标状态：  return solution
    处理当前状态C
    for(C的后续未访问状态S)
        path.push(S) //进入
        sub_solution =DFS(S, path)
        if sub_solution  is not null:
            combine a solution
            return solution
        path.pop();  //回到C
     return null
```



深度优先搜索 (回溯法) Depth-first search(DFS)

```
DFS(state):
    visited.add(state) // 标记当前状态已访问

    if state is the goal state:
        return solution // 返回 解答路径

    children = generate_children(state) // 生成当前状态的所有子状态

    for child in children:
        if child is not in visited:
            solution = DFS(child) // 对每个孩子递归调用DFS
            if solution is not null:
                return solution // 返回解答路径

    return null // 如果未发现目标状态，返回null
```

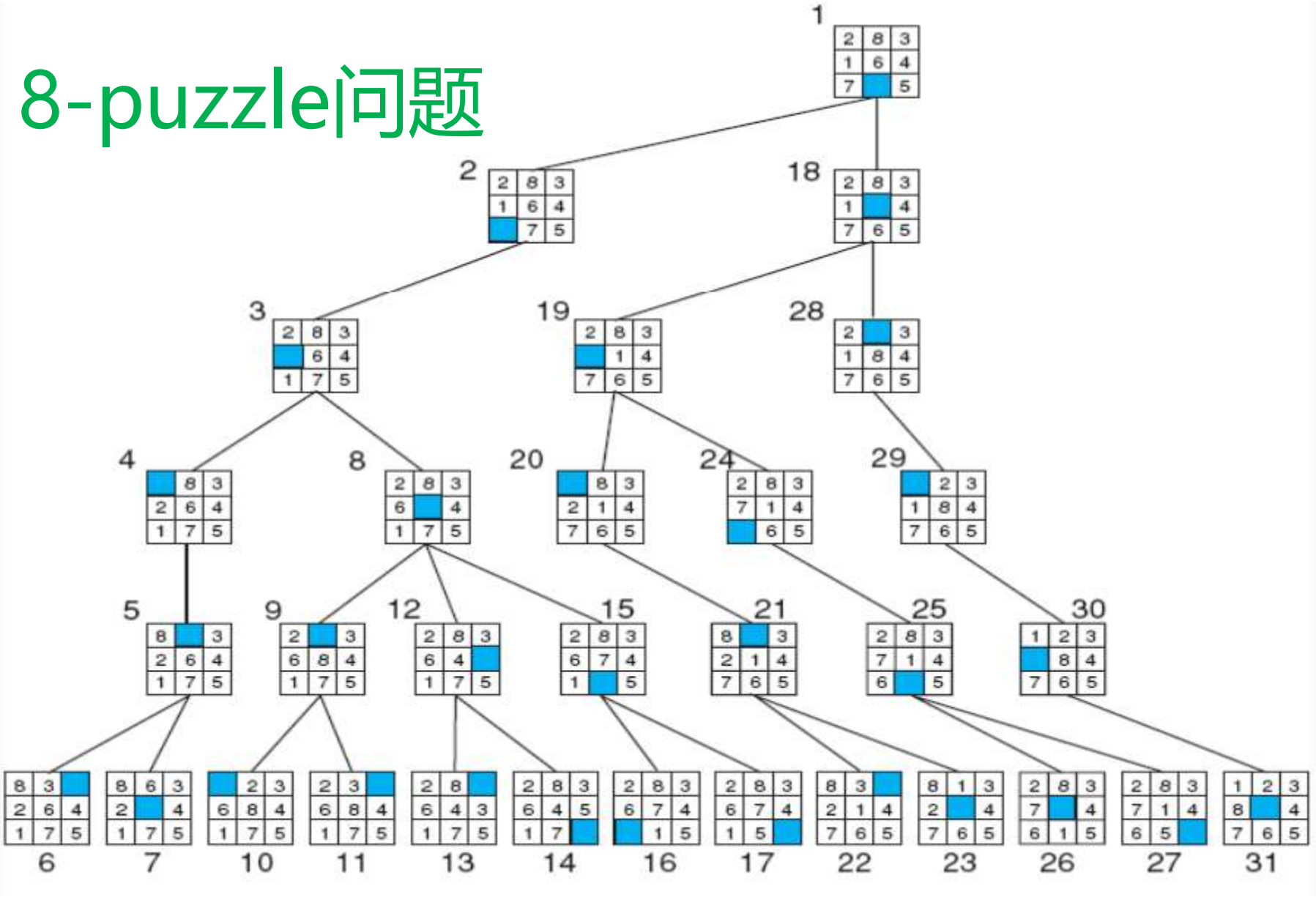
深度优先搜索 (回溯法) Depth-first search(DFS)

```
DFS(state):
    stack.push(state)  // 初始状态入栈
    visited.add(state) // 标记初始状态已访问

    while stack is not empty:
        node = stack.pop() // 栈顶状态出栈

        if node is the goal state:
            return solution // 返回解答路径

        children = generate_children(node) // 生成当前状态的所有子状态

        for child in children:
            if child is not in visited:   //对于未访问过的子状态
                visited.add(child) // 标记子状态已访问
                stack.push(child)  // 将子状态入栈

    return failure // 如果未找到目标状态，返回失败信息
```

深度优先搜索 vs 回溯法

- 深度优先搜索 (DFS)和回溯法(backtracking)**本质上是一样的**。
- 也有人认为backtracking是更通用的方法，DFS针对树形结构的具体的backtracking技术。 DFS针对显式的树，而backtracking处理的是隐含的树。
- 也有人认为backtracking需要记录走过路径，而DFS不需要记录走过的路径。这种观点实际上并不正确。
- 也有人认为DFS是可以用于图形结构的更通用技术，而backtracking则是DFS的一种特殊形式。

8-puzzle问题


1
2
18
3
19
28
4
8
20
24
29
5
9
12
15
21
25
30
6
7
10
11
13
14
16
17
22
23
26
27
31

8-puzzle问题的DFS递归实现

```
def dfs(node, visited, goal):
    if node.state == goal:
        return node

    visited.add(node.state)

    for child in node.expand():
        if child.state not in visited:
            result = dfs(child, visited, goal)
            if result is not None:
                return result

    return None
```

8-puzzle问题的基于栈的DFS实现

```
# depth-first search algorithm
def dfs(initial_state):
    stack = [(initial_state, [])]
    visited = set()
    while stack:
        state, path = stack.pop()
        if is_goal_state(state):
            return path
        if tuple(state) not in visited:
            visited.add(tuple(state))
            for next_state in get_next_states(state):
                stack.append((next_state, path + [next_state]))
```

DFS：全排列

Youtube频道： **hwdong**

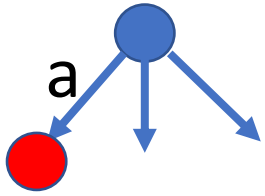
博客： **hwdong-net**.github.io

全排列

- 不含重复元素的一组元素的所有可能排列。
- 输入: a b c
- 输出 :

a b c
a c b
b a c
b c a
c a b
c b a

全排列:深度优先



第1个位置

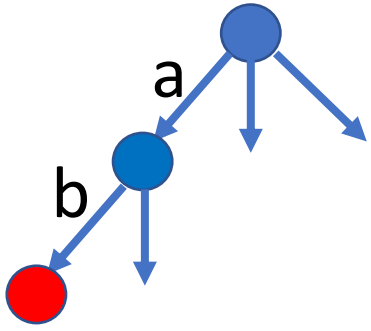
- {a,b,c}的全排列


```
Permute([a,b,c]):
    [a,Permute([b,c])]
```

状态：a

全排列

- {a,b,c}的全排列



第1个位置

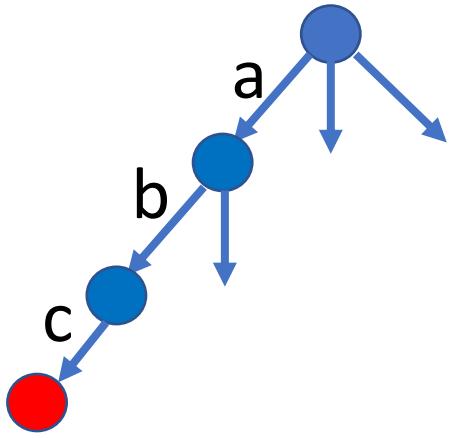
第2个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])]
      [a,b,Permute([c])]
```

状态：a,b

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

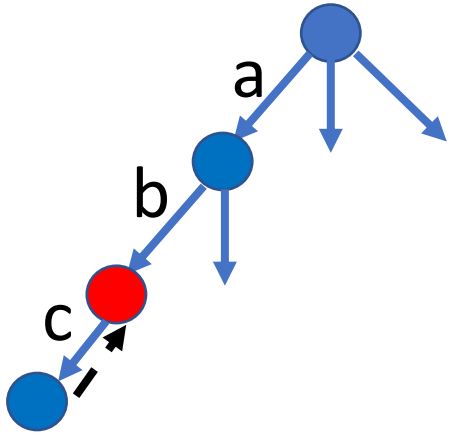
第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])]
      [a,b,Permute([c])]
        [a,b,c,Permute()]
```

状态：a,b,c

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

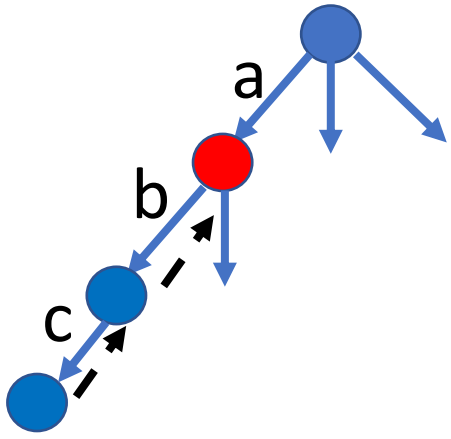
第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])
      [a,b]
```

状态：a,b

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

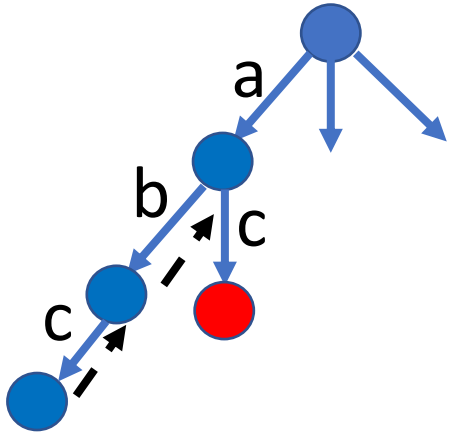
第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])
```

状态：a

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

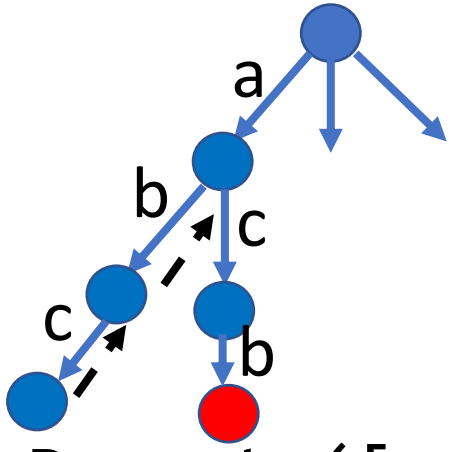
第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])]
      [a,c,Permute([b])]
```

状态：a,c

全排列

- {a,b,c}的全排列



第1个位置

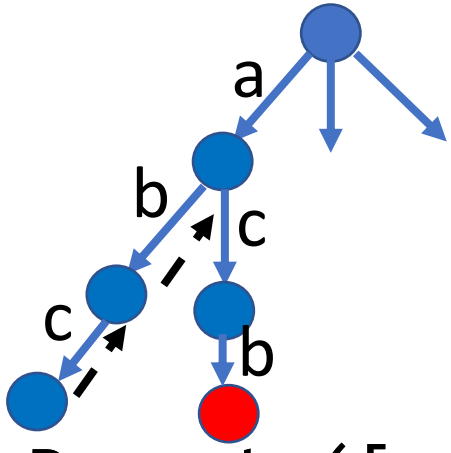
第2个位置

第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])]
      [a,c,Permute([b])]
```

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

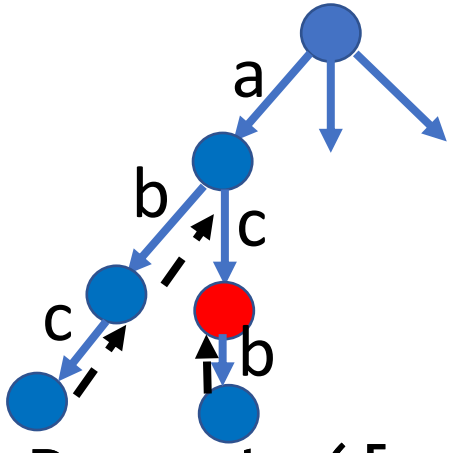
第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])]
      [a,c,b,Permute([])]
```

状态： a,c,b

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

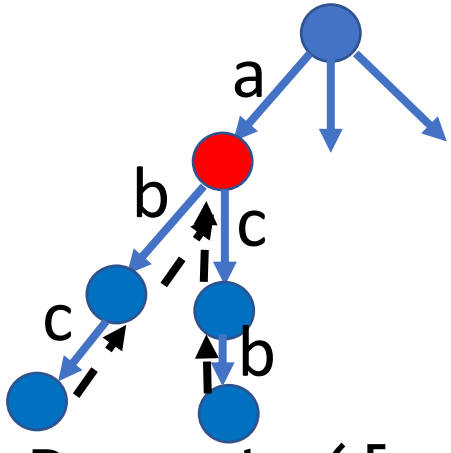
第3个位置

```
Permute([a,b,c]):
    [a,Permute([b,c])]
        [a,c]
```

状态：a,c

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

第3个位置

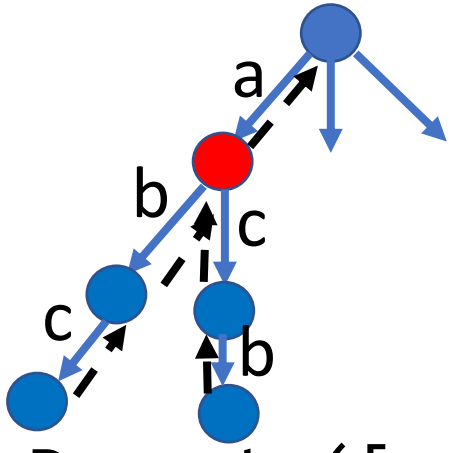
Permute([a,b,c]):

[a]

状态：a

全排列

- {a,b,c}的全排列



第1个位置

第2个位置

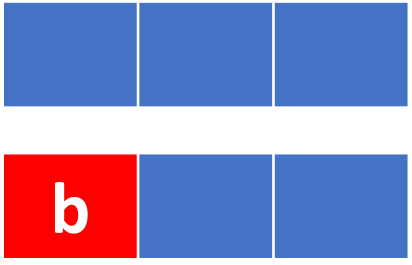
第3个位置

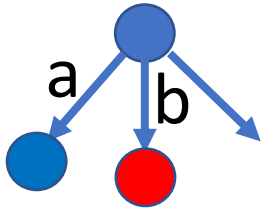
```
Permute([a,b,c]):
```

当前状态：初始状态

全排列

- {a,b,c}的全排列





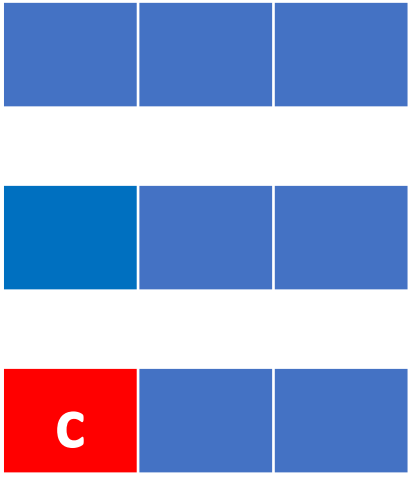
第一个位置

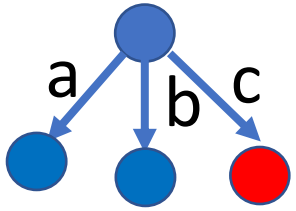
```
Permute([a,b,c]):
    [a,Permute([b,c])
    [b,Permute([a,c])
```

状态：b

全排列

- {a,b,c}的全排列



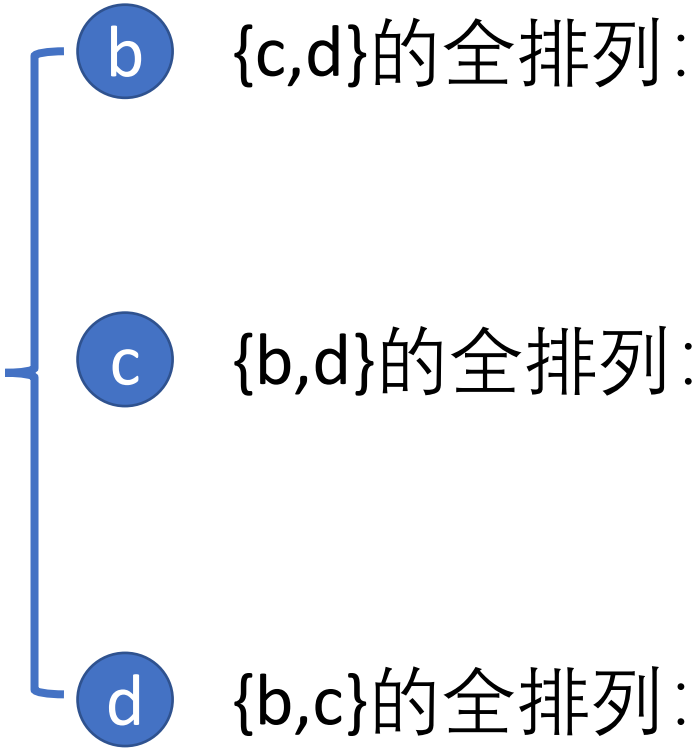


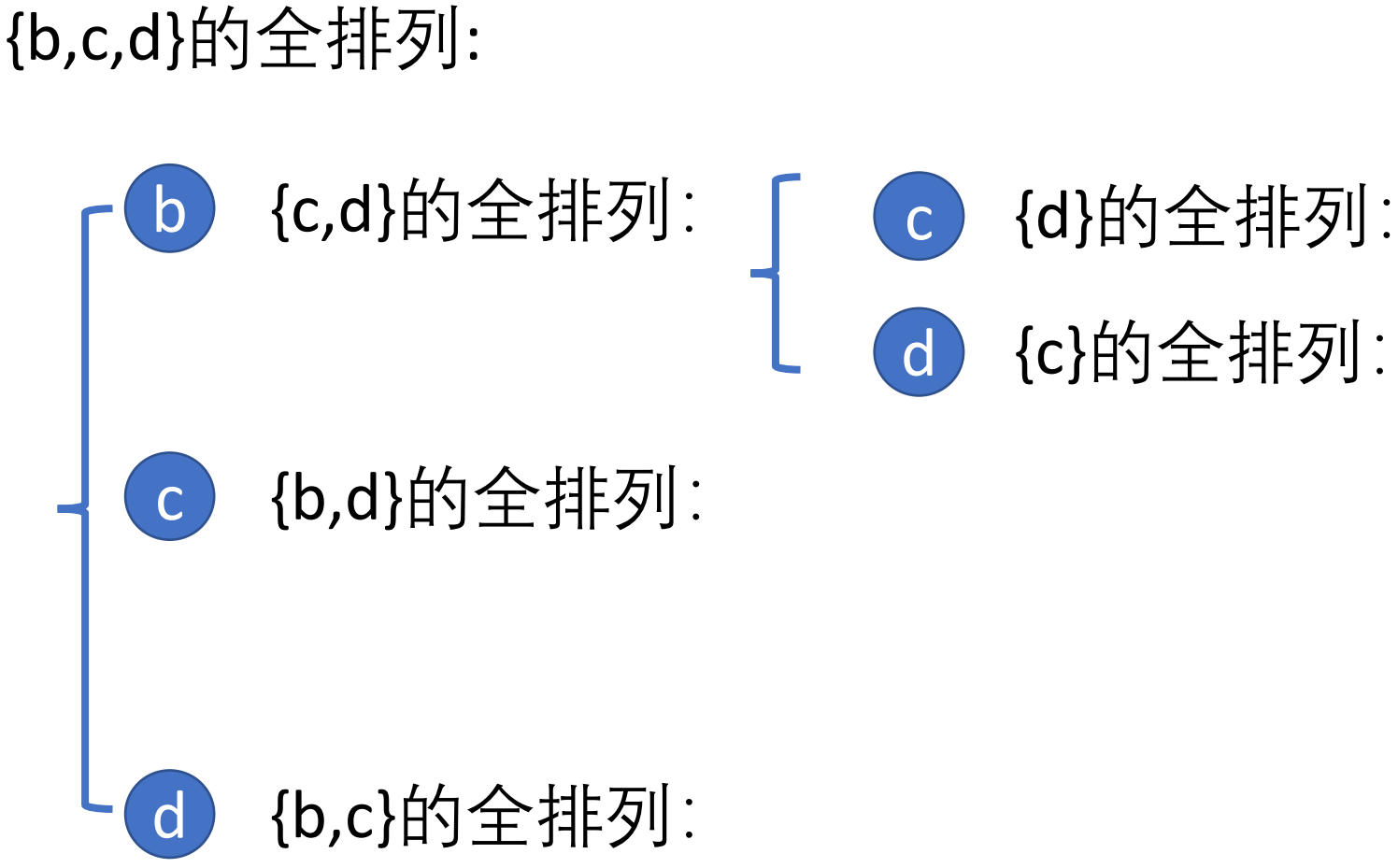
第一个位置

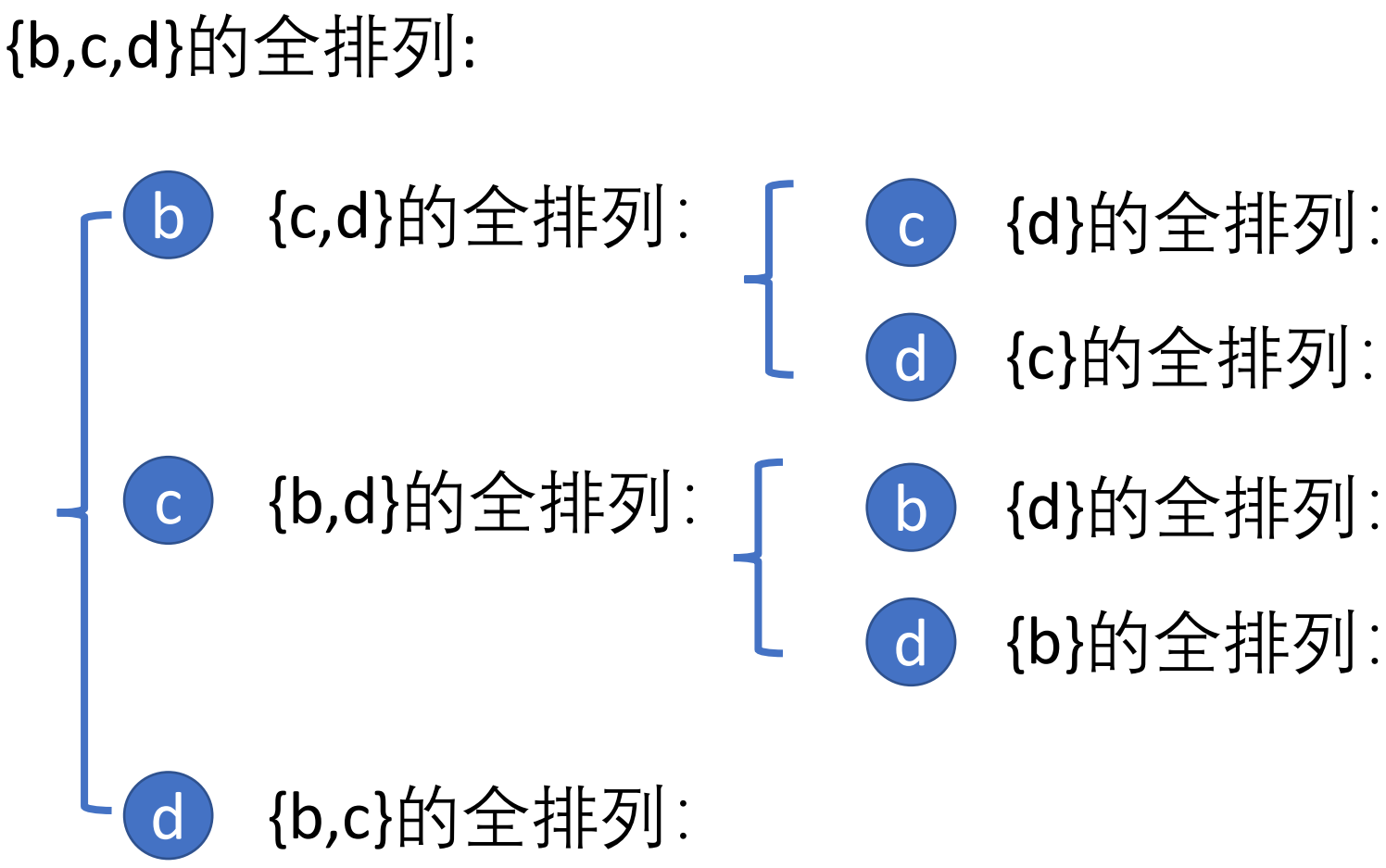
```
Permute([a,b,c]):
    [a,Permute([b,c])
    [b,Permute([a,c])
    [c,Permute([b,a])
```

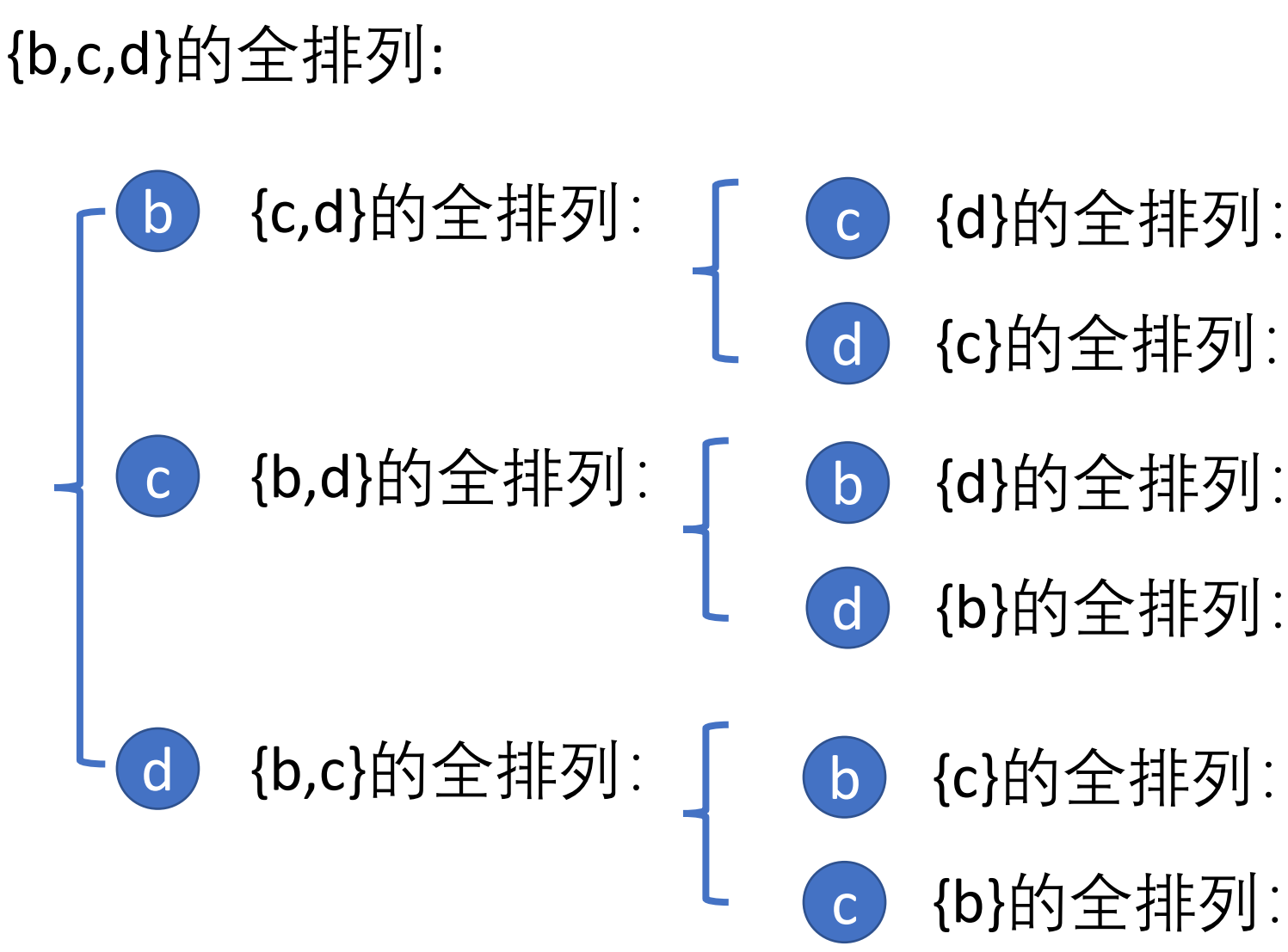
状态：c

{b,c,d}的全排列:




{b,c,d}的全排列:
b
{c,d}的全排列:
c
{d}的全排列:
d
{c}的全排列:
c
{b,d}的全排列:
d
{b,c}的全排列:

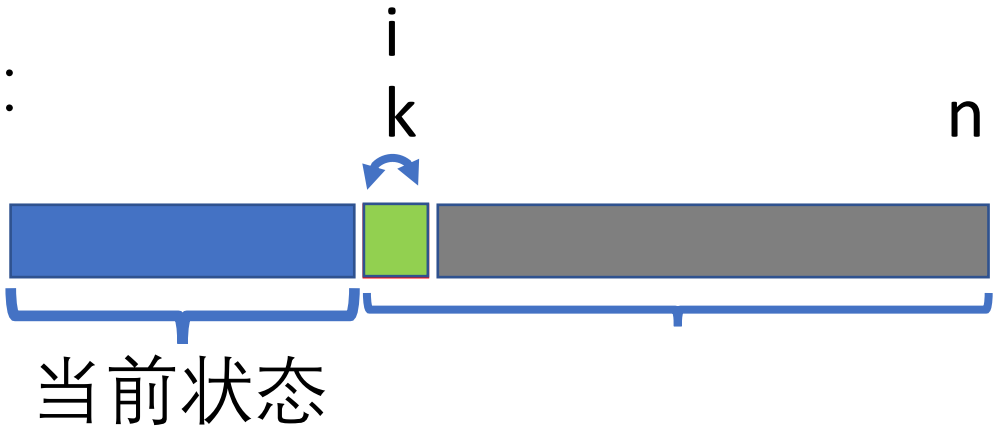

{b,c,d}的全排列:
b
{c,d}的全排列:
c
{d}的全排列:
d
{c}的全排列:
c
{b,d}的全排列:
b
{d}的全排列:
d
{b}的全排列:
d
{b,c}的全排列:

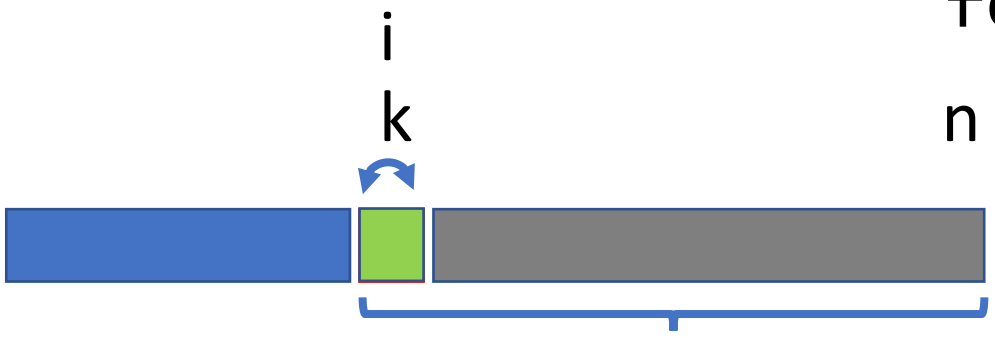

{b,c,d}的全排列:
b
{c,d}的全排列:
c
{d}的全排列:
d
{c}的全排列:
c
{b,d}的全排列:
b
{d}的全排列:
d
{b}的全排列:
d
{b,c}的全排列:
b
{c}的全排列:
c
{b}的全排列:

{b,c,d}的全排列:

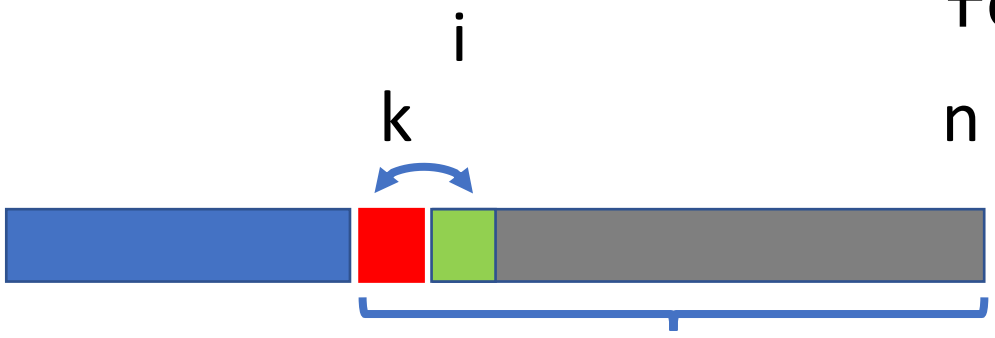
- b {c,d}的全排列:
- c {b,d}的全排列:
- d {b,c}的全排列:

```
Permute(S,k,n):
    if k==n:
        cout<<S
    for i=k to n:
        swap(S[i],S[k])
        Permute(S,k+1,n):
        swap(S[i],S[k])
```

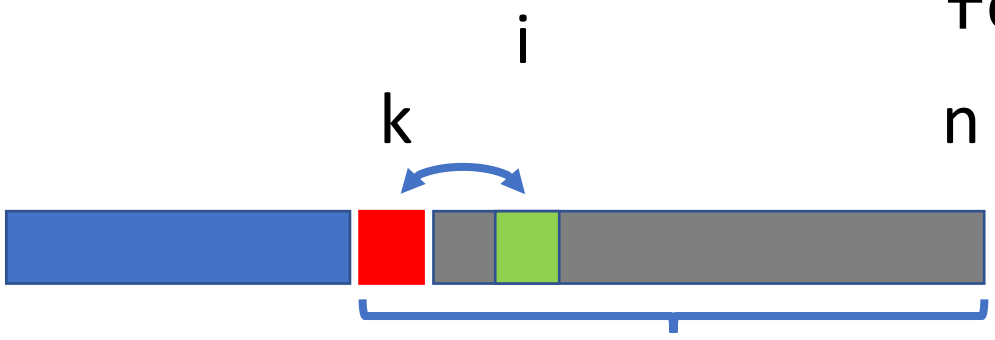




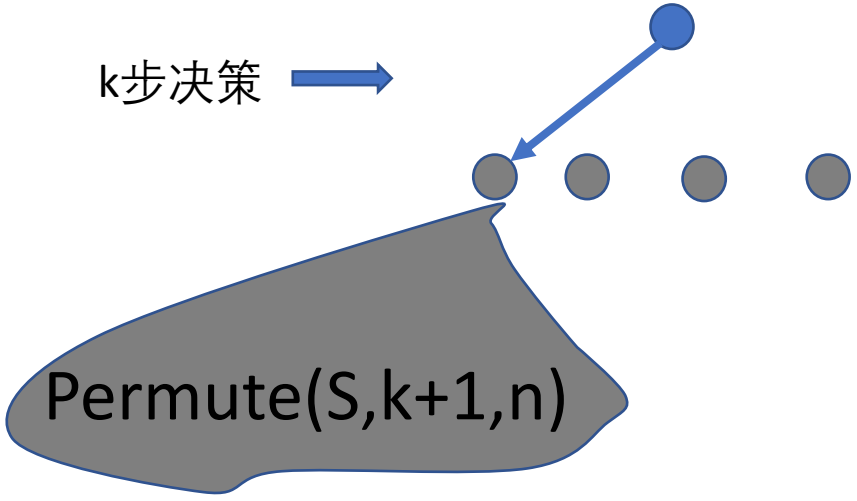

```
Permute(S,k,n):
    if k==n:
        cout<<S
    for i=k to n:
        swap(S[i],S[k])
        Permute(S,k+1,n):
        swap(S[i],S[k])
```



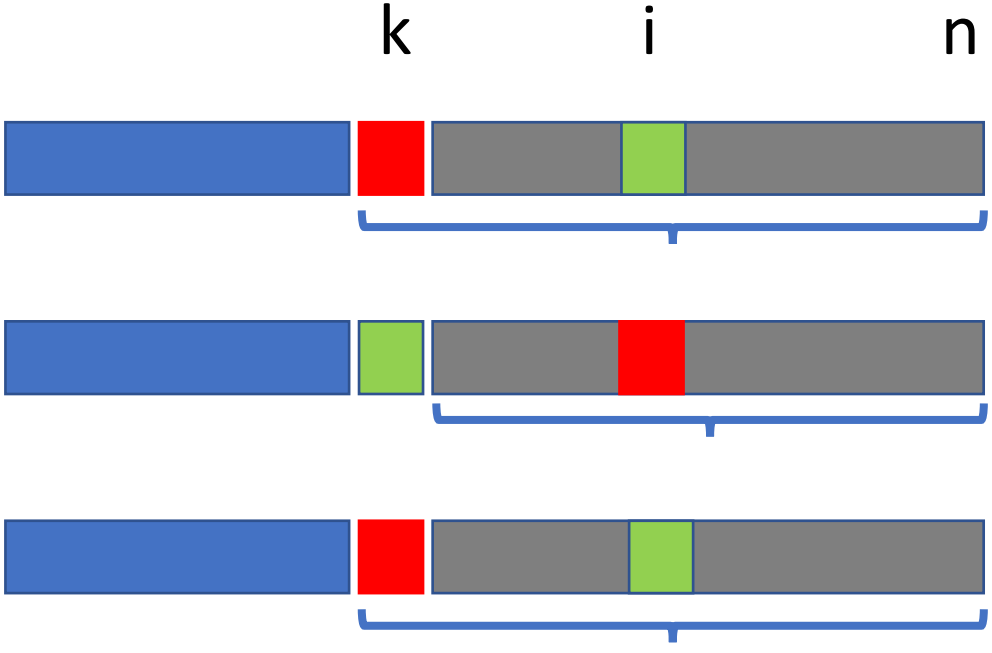

```
Permute(S,k,n):
    if k==n:
        cout<<S
    for i=k to n:
        swap(S[i],S[k])
        Permute(S,k+1,n):
        swap(S[i],S[k])
```




```
Permute(S,k,n):
    if k==n:
        cout<<S
    for i=k to n:
        swap(S[i],S[k])
        Permute(S,k+1,n):
        swap(S[i],S[k])
```



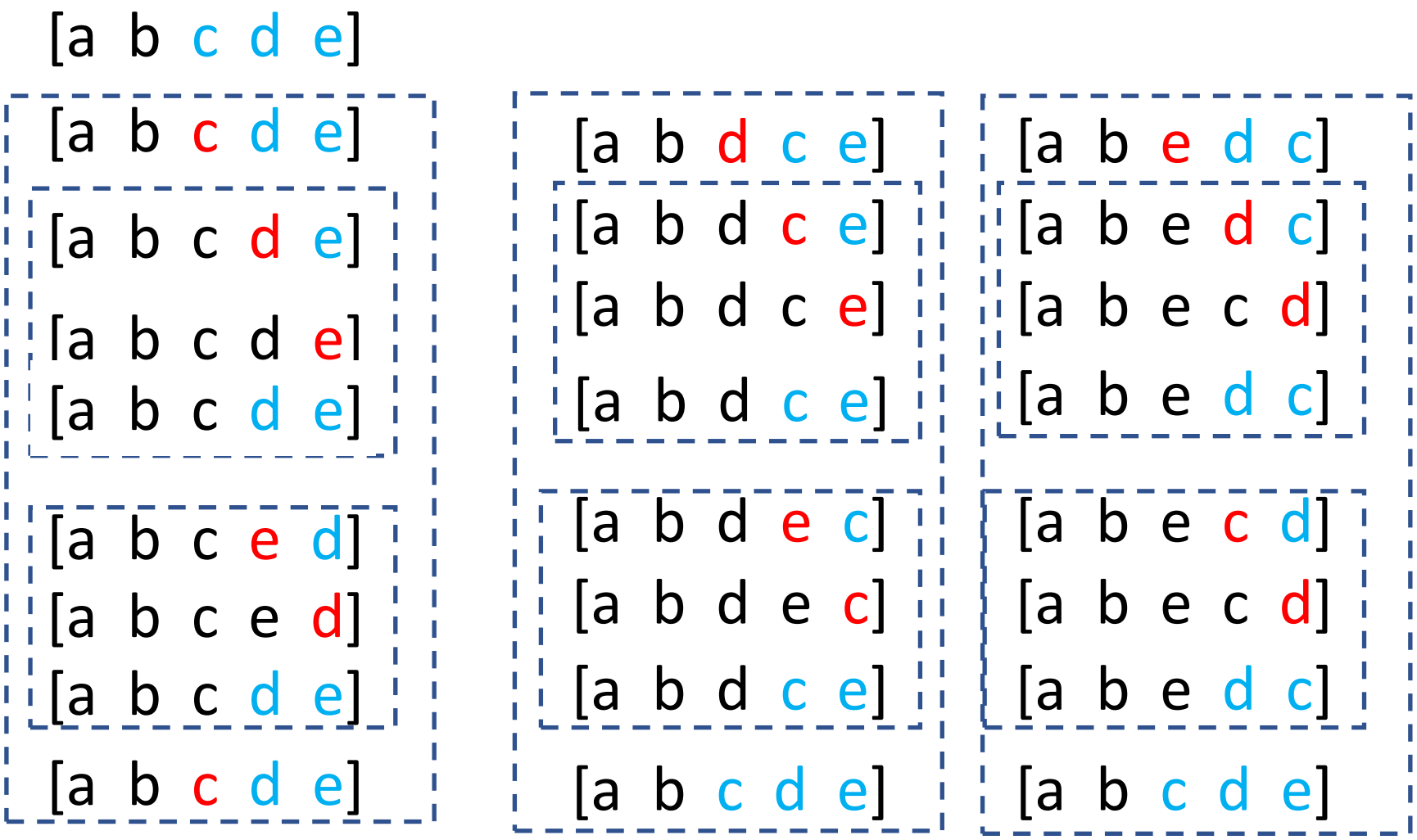
Permute(S,k,n)



Swap(S[k],S[i])

Permute(S,k+1,n)

Swap(S[k],S[i])


[a b c d e]
[a b c d e]
[a b c d e]
[a b c d e]
[a b c d e]
[a b c e d]
[a b c e d]
[a b c d e]
[a b c d e]
[a b d c e]
[a b d c e]
[a b d c e]
[a b d c e]
[a b d e c]
[a b d e c]
[a b d c e]
[a b c d e]
[a b e d c]
[a b e d c]
[a b e c d]
[a b e d c]
[a b e c d]
[a b e c d]
[a b e d c]
[a b c d e]

DFS：背包问题

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

DFS： 背包问题

- 给定一组物品，每种物品都有自己的重量和价格，在限定的总重量内，我们如何选择，才能使得物品的总价格最高。问题的名称来源于如何选择最合适的物品放置于给定背包中。
- 如$w_1,w_2,\ldots,w_n$表示n个物品的重量，而它们的价值是$v_1,v_2,\ldots,v_n$,背包的总重量为W。

$$\max \sum_{i=1}^{n} v_i x_i$$

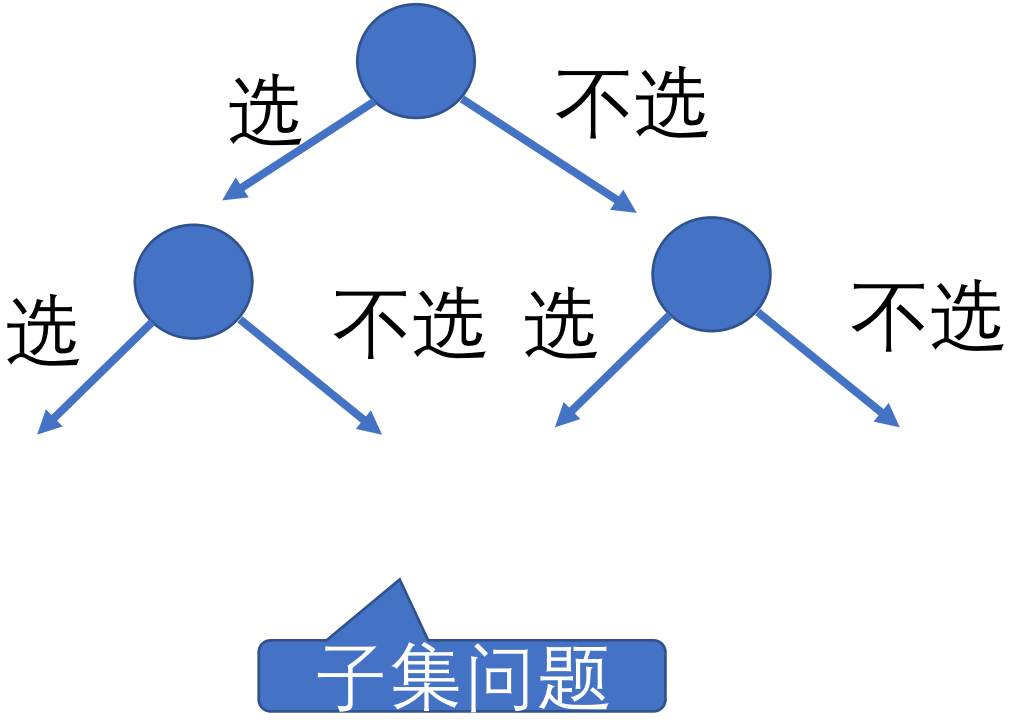
$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le W \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

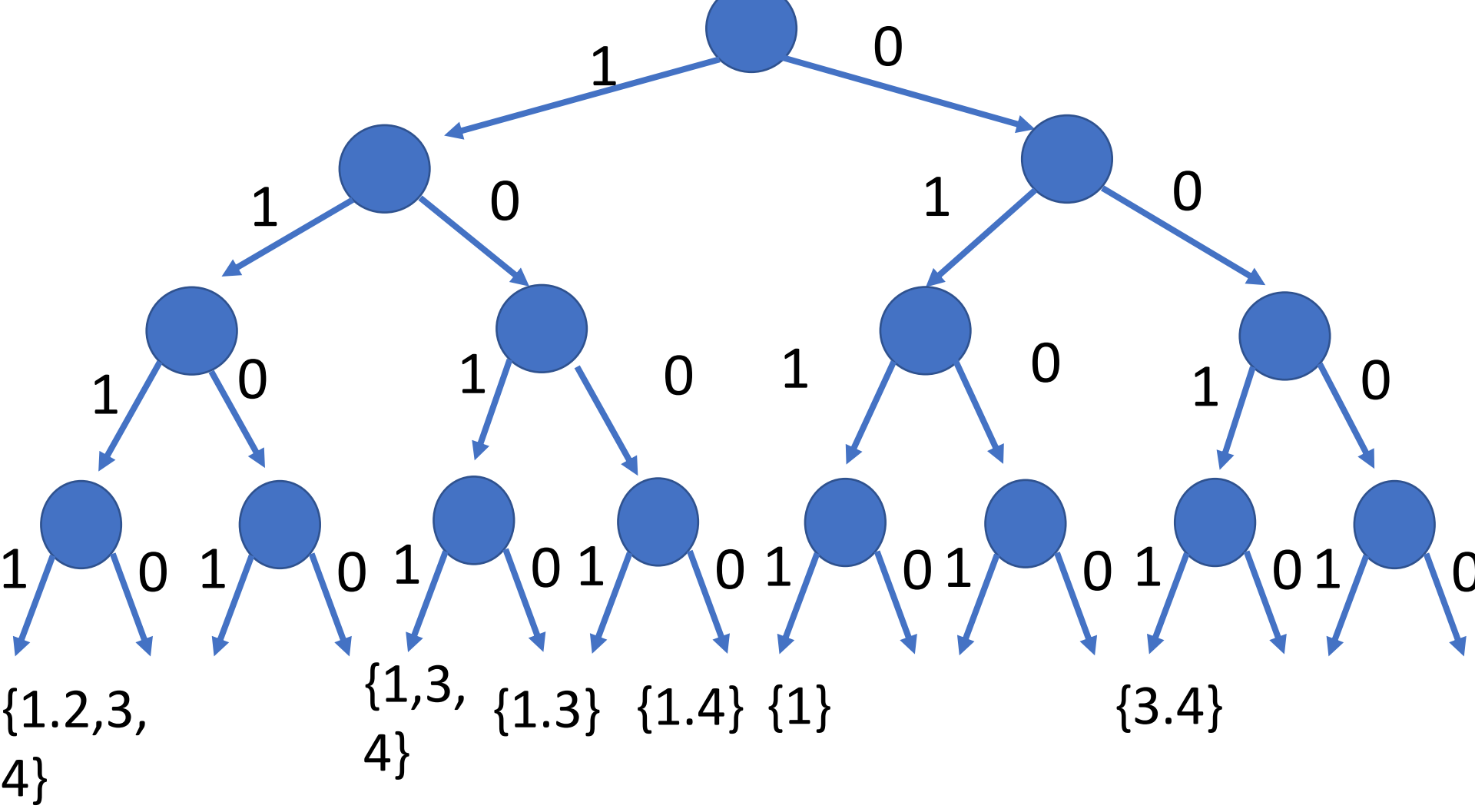
蛮力法

- 每个物体都试试放或不放，n个物体一共有2^n种可能性

编号	重量	价值
1	2	$20
2	5	$30
3	10	$50
4	5	$10

背包最大容量：16





子集树

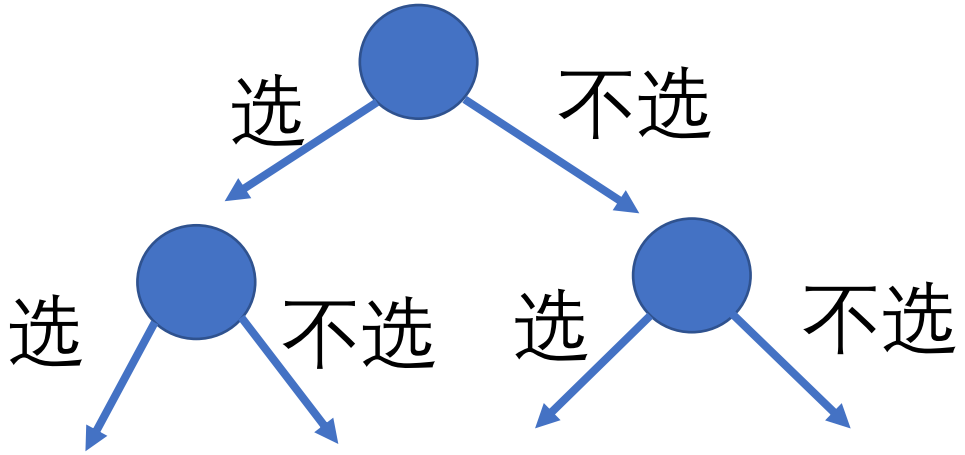
编号	重量	价值
1	2	$20
2	5	$30
3	10	$50
4	5	$10

背包最大容量： 16

子集	总重量	总价值
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	不可行
{1,2,4}	12	$60
{1,3,4}	17	不可行
{2,3,4}	20	不可行
{1,2,3,4}	22	不可行

```
knapsack(W, w[], v[], P[], value, k, &maxValue, &maxP):
    if k==n+1:
        if value>maxValue:
            maxValue = value;    maxP = P; return;
    if w_k>W :
        knapsack(W,w,v,P,value,k+1,maxValue,maxP)
    else:
        P.push_back(k)  //选
        knapsack(W- w_k,w,v,P,value + v_k,k+1,maxValue,maxP)
        P.pop_back(k)   //不选
        knapsack(W,w,v,P,value,k+1,maxValue,maxP)
```

```
int knapSack(W, wt[],val[], n)
    // Base Case
    if (n == 0 || W == 0)
        return 0;


    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1],
                           wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
```

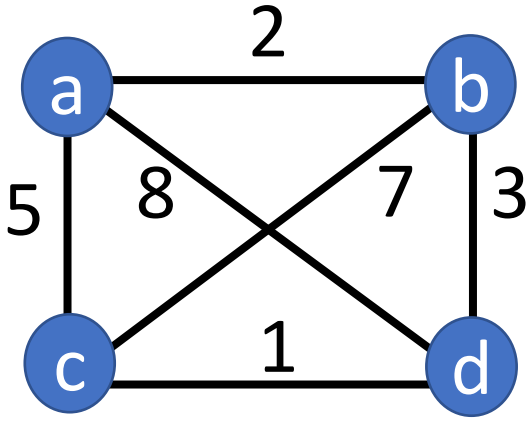


DFS: 旅行销售商TSP

YouTube频道： hwdong
博客： hwdong-net.github.io

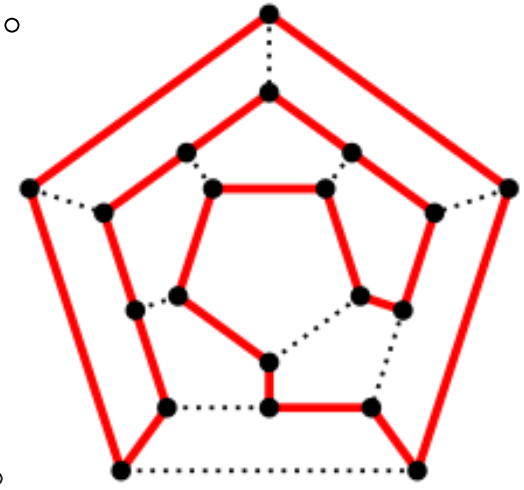
TSP (Traveling Salesman Problem)

- 旅行推销员问题、货郎担问题，是数学领域中著名问题之一。
- 一个售货员希望从一个城市出发访问 n 个城市,每个城市只访问一次，开始和结束于同一城市。问题：设计一个算法,找出售货员能采取的最短路径。



哈密尔顿(Hamilton)回路

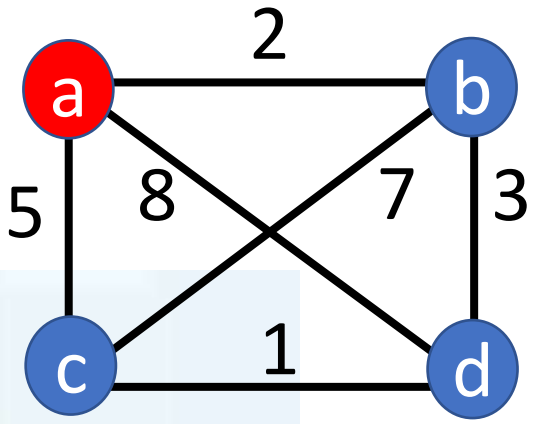
- 爱尔兰数学家哈密尔顿提出著名的周游世界问题，他用正十二面体的20个顶点代表20个城市，要求从一个城市出发，经过每个城市恰好一次，然后回到出发城市。



- 寻找哈密顿路径是一个典型的NP-完全问题。
- TSP问题和哈密顿回路的区别是加了路程最短的要求。

穷举法

起点a, 剩余顶点的全排列

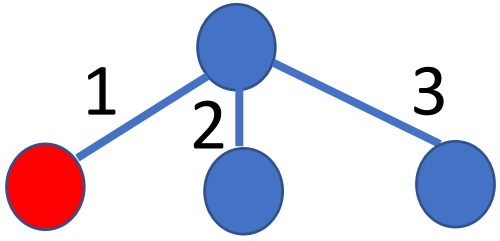


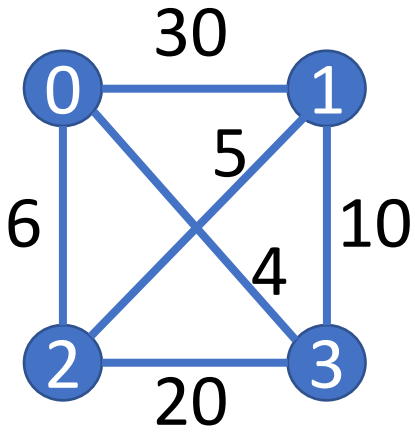
路线	旅程长度	
a→b→c →d →a	l=2+8+1+7=18	
a→b→d →c →a	l=2+3+1+5=11	最佳
a→c→b →d →a	l=5+8+3+7=23	
a→c→d →b →a	l=5+1+3+2=11	最佳
a→d→b →c →a	l=7+3+8+5=23	
a→d→c →b →a	l=7+1+8+2=18	

- $T(n) = \Theta((n-1)!)$

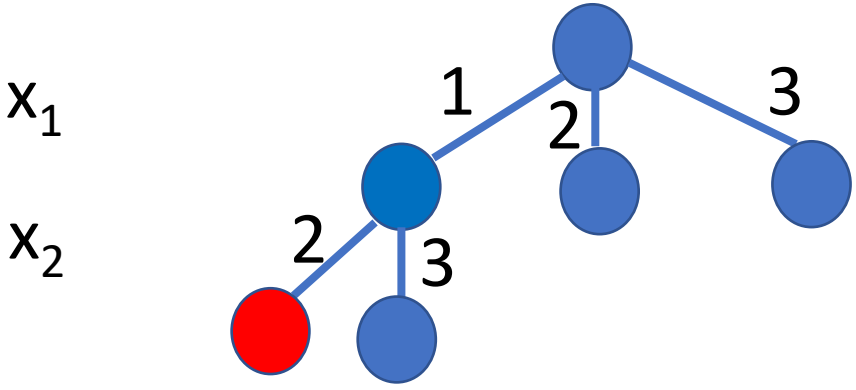
TSP旅行销售商问题

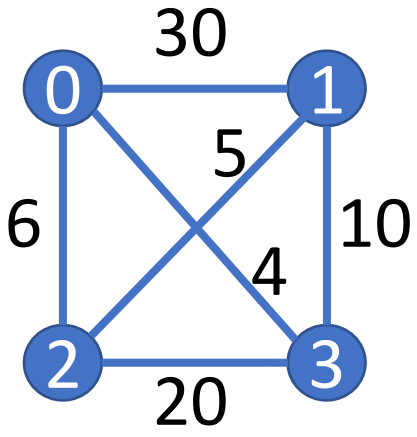
x_1



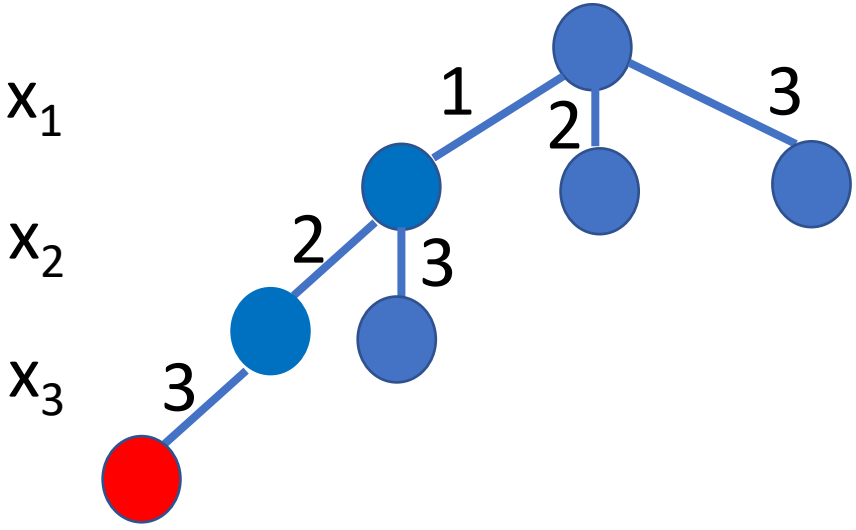


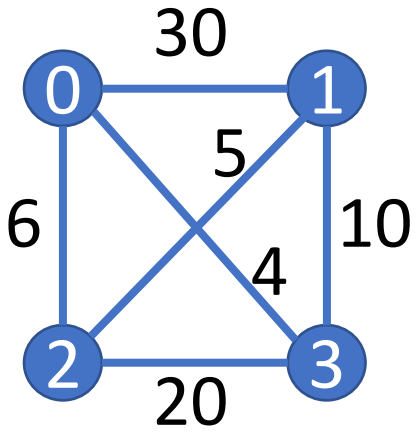
TSP旅行销售商问题



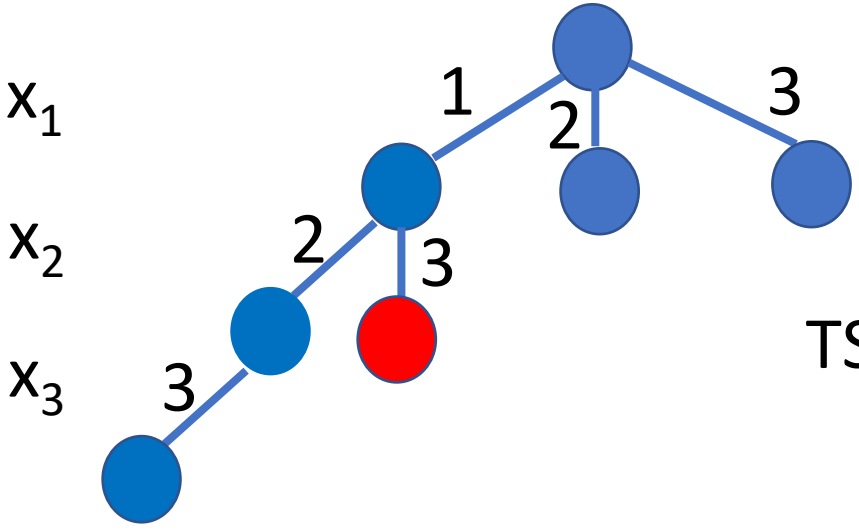


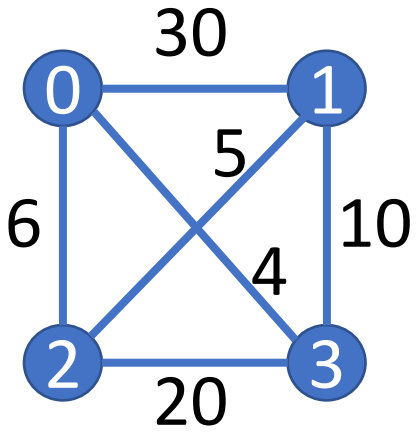
TSP旅行销售商问题





TSP旅行销售商问题






```
TSP(G,S,k)
    //S[...,k-1,k,...,n]
    if 没走完:
      for i=k to n:
        swap(S[k,S[i]])
        TSP(G,S,k+1)
        swap (S[k,S[i]])
```

```
TSP_BruteForce(G, S, k,minCost)  //S[…,k-1,k,…,n]
    if k==n+1:
        minCost = min(minCost, cost(S)); return
```

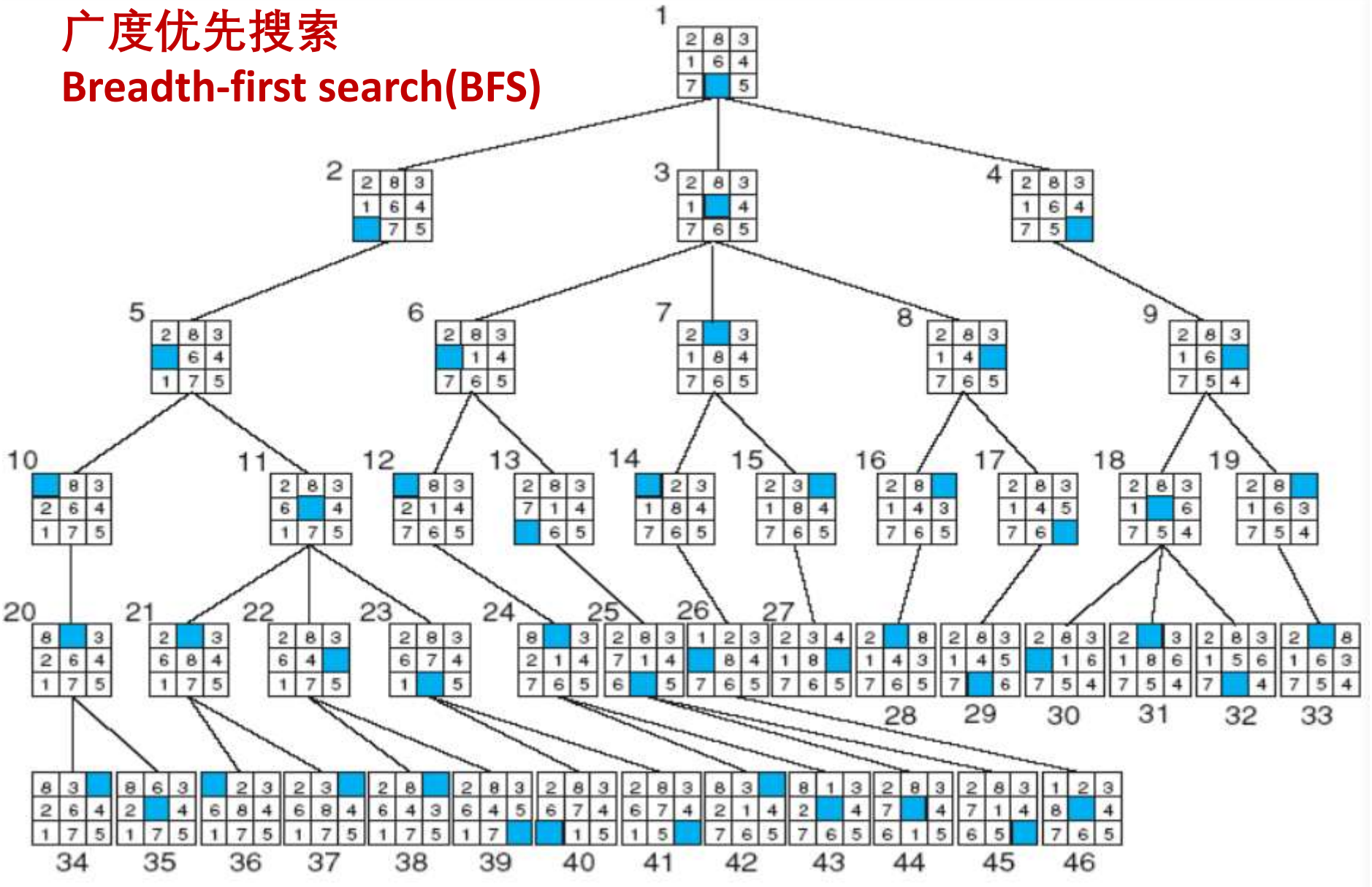
```
TSP_BruteForce(G, S, k,minCost)  //S[…,k-1,k,…,n]
    if k==n+1:
        minCost = min(minCost, cost(S)); return
    for i=k to n:
        swap(S[k, S[i]])
        TSP_BruteForce(G,S,k+1,minCost)
        swap (S[k, S[i]])
```

广度优先搜索

Youtube频道： **hwdong**

博客： **hwdong-net**.github.io

广度优先搜索
Breadth-first search(BFS)



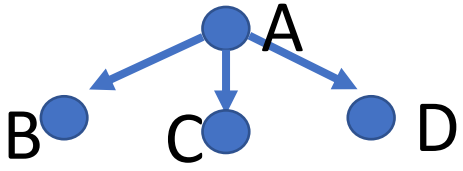
广度优先搜索

- 层次搜索

A

广度优先搜索

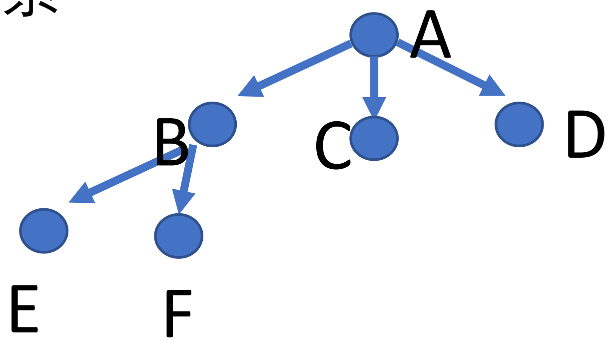
- 层次搜索

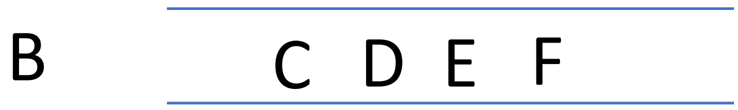


A　　B　C　D

广度优先搜索

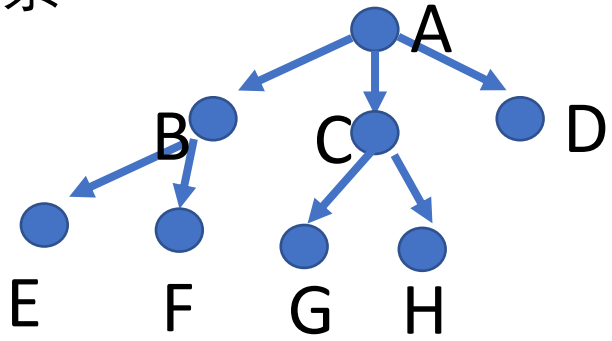
- 层次搜索





广度优先搜索

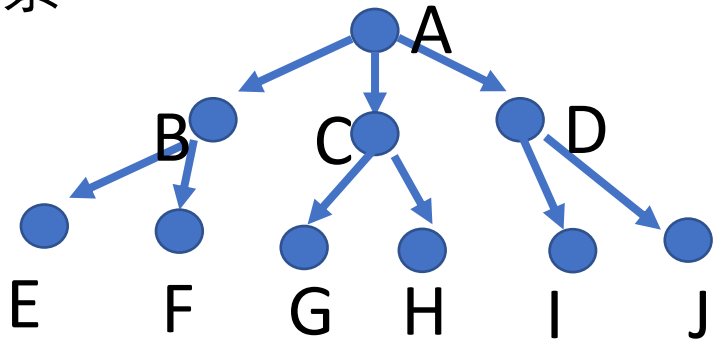
- 层次搜索



C　　D E F G H

广度优先搜索

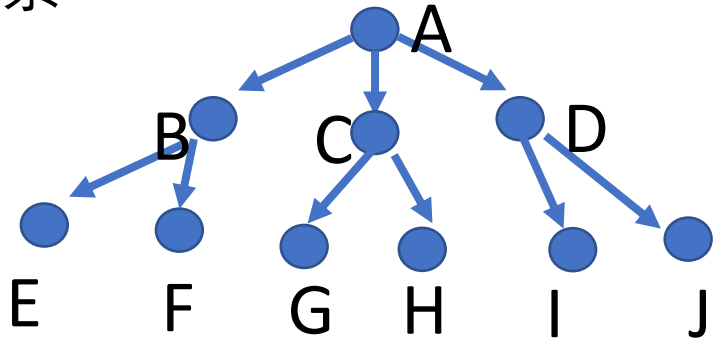
- 层次搜索



D　　E F G H I J

广度优先搜索

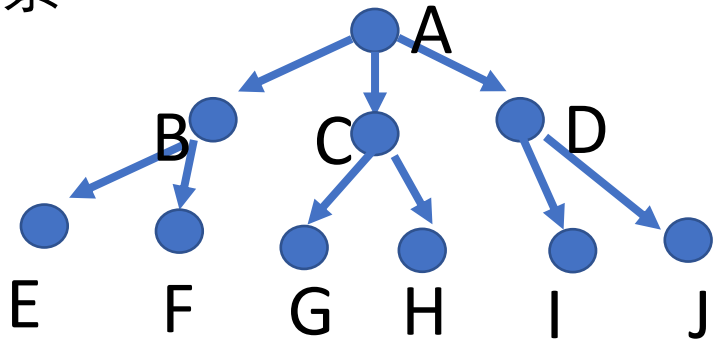
- 层次搜索



E　　F G H I J

广度优先搜索

- 层次搜索

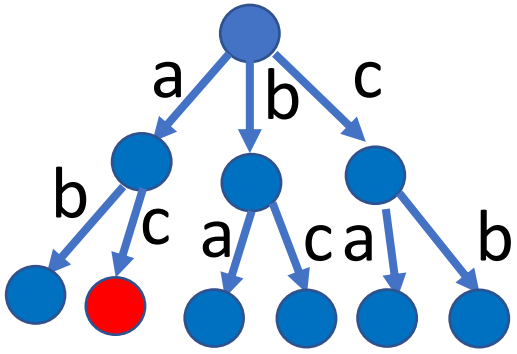


F　　G H I J

广度优先搜索的算法伪代码

```
BFS(start_state, goal_state):
  创建一个队列 Q
  初始状态入队 Q
  while Q 非空:
      从Q出队一个状态 n
      if n 是目标状态:
          return n
      for n的未访问过每个邻居 m:
          将m加入队列Q
          标记 m 访问过
  return "目标状态未找到"
```

全排列：广度优先



- 用一个数组P表示已有的决策，
  初始P=[]表示没有任何决策
- P=[‘a’,’c’]有2个元素，说明做了2次决策
- 在每个状态下，探索其他可行的状态
  如P=[‘a’,’c’]只剩余一个元素’b’可以作为决策选项

```
for 每个元素s:
    如果s不在P中，则
        P.add(s),将新状态加入队列
```

子集

subset

YouTube频道： hwdong
博客： hwdong-net.github.io

子集问题（leetcode 78 **Subsets**）

- 给定一个不同元素组成的集合，要求这个集合的所有子集（the power set）。

Example 1:

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

子集问题（leetcode 78 **Subsets**）

- 给定一个不同元素组成的集合，要求这个集合的所有子集（the power set）。

Example 2:

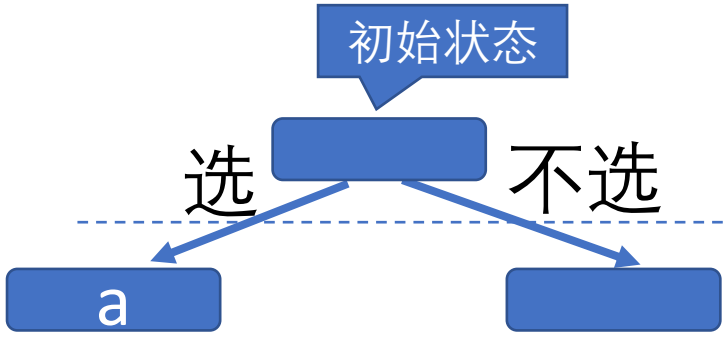
```
Input: nums = [0]
Output: [[],[0]]
```

多步决策

- 对每个物品，选和不选

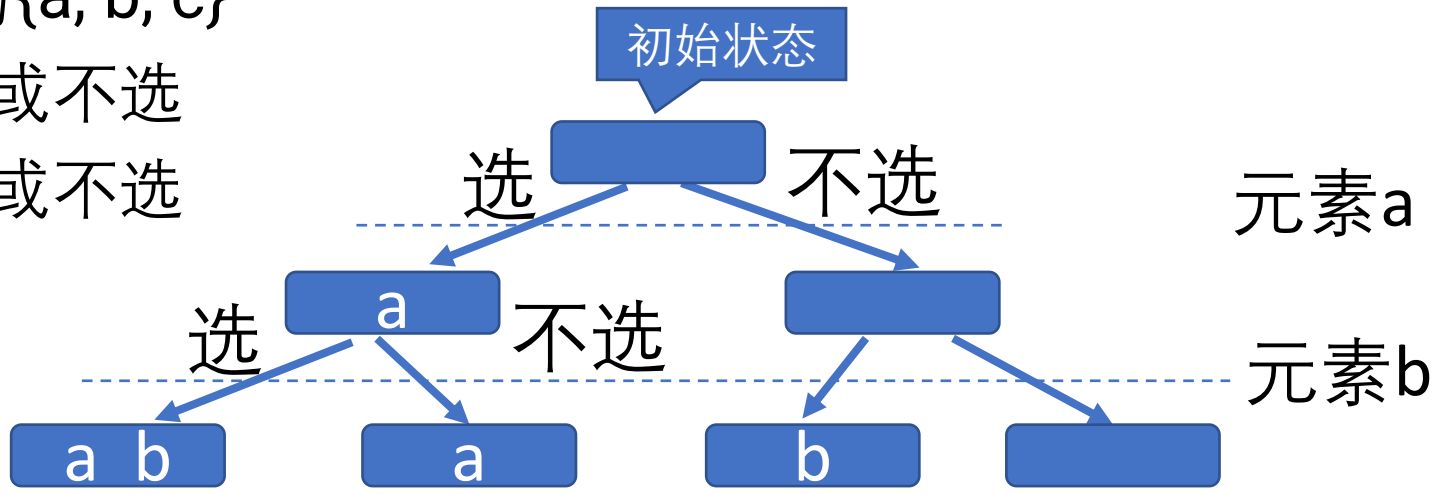
- 设集合为{a, b, c}
- 初始状态：没做任何决策

- 设集合为{a, b, c}
- 第1个选或不选

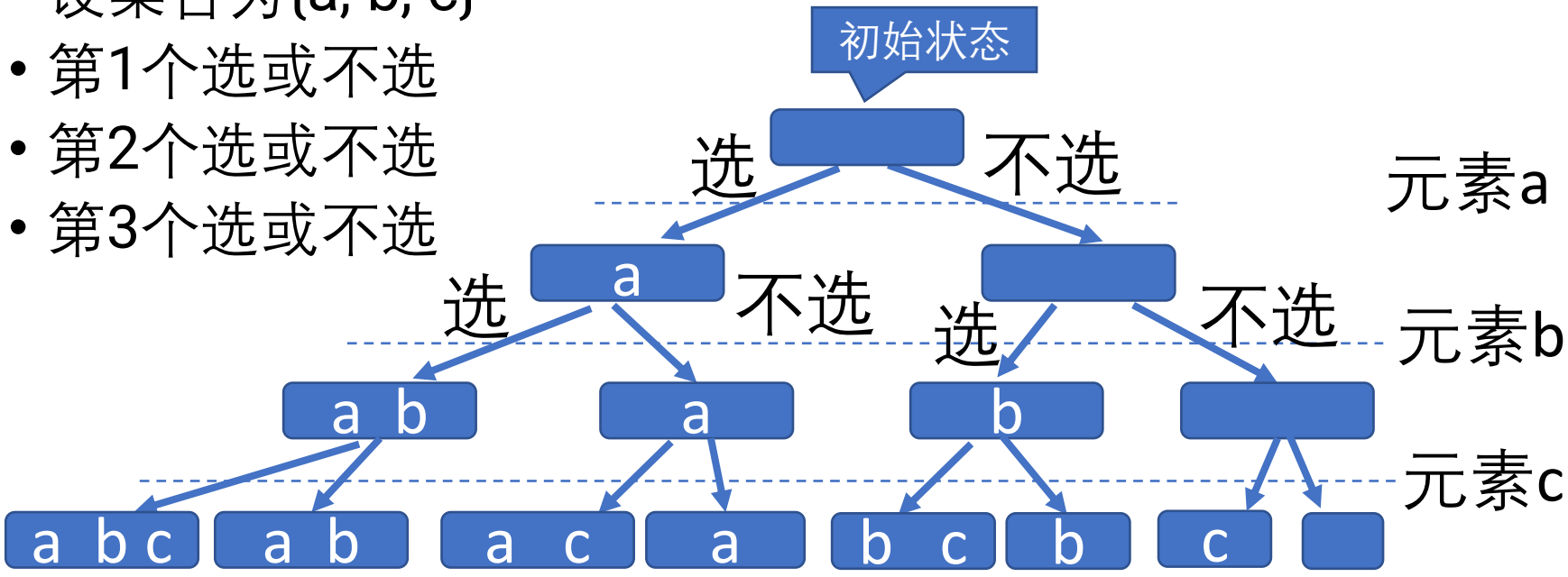


元素a

- 设集合为{a, b, c}
- 第1个选或不选
- 第2个选或不选



- 设集合为{a, b, c}
- 第1个选或不选
- 第2个选或不选
- 第3个选或不选



$T(n) = 2^n$

子集树

深度优先(回溯法)：递归实现

- 类似于全排列问题

```
subset(S={a,b,c}, P=[], k=1, subsets):
    if k==n+1:
        subsets.add(P); return;
    P.push_back(S[k])          //选：进入子状态
    subset(S,P,k+1,subsets)
    P.pop_back();              //回退到父状态
    subset(S,P,k+1,subsets)
```

广度优先搜索：练习

- 类似于全排列

剪枝:
约束、分支限界

YouTube频道： hwdong
博客： hwdong-net.github.io

优化状态空间搜索:剪枝

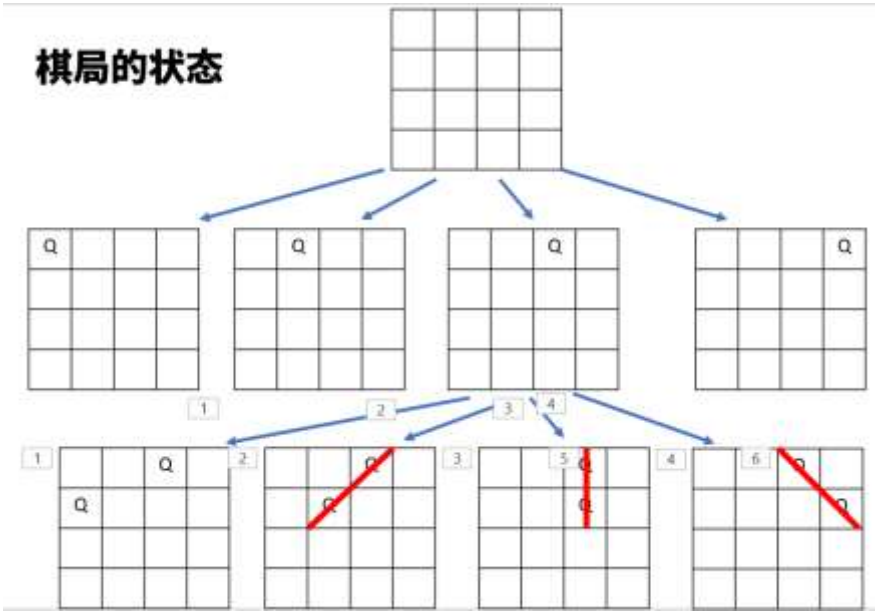
- 蛮力法树形搜索时间复杂度超过指数级!
- 必须采用各种**剪枝技术**减少搜索量!
- 剪枝：如果发现从某个状态探索**不可能得到解**或**不可能产生**比已知价值**更有价值的解**，则停止从该状态去探索其后续子状态（子树上的所有状态）。例如，下棋种某一步的走子会导致失败，就停止尝试这种走子探索!

剪枝

- 用限界函数/约束函数避免无效搜索。
- 约束函数：不满足约束的子树不再探索
- 限界函数：不能得到最优价值的子树不再探索

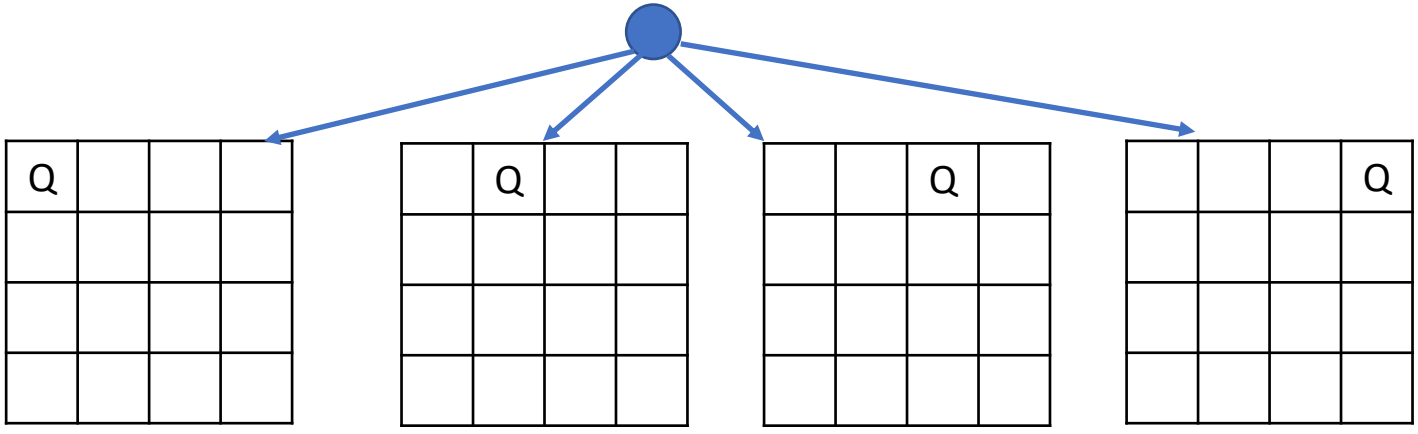
分支限界

针对最优问题



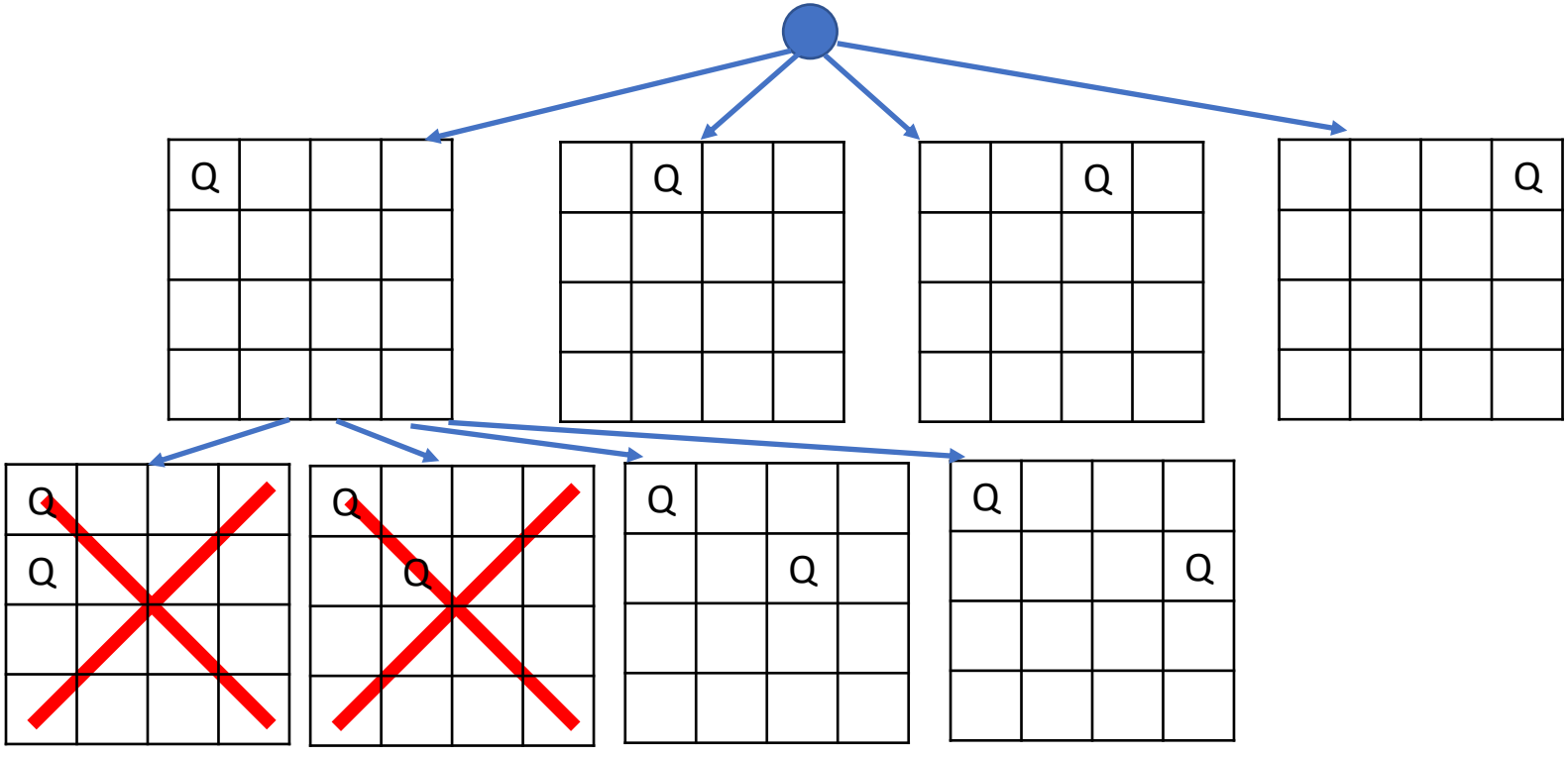
剪枝:约束

- 为了避免搜索整个状态空间，通常利用约束条件避免不必要的状态搜索。



剪枝:约束

- 例如，八皇后问题，当在某行放置皇后到某一列时，如果发现不符合约束条件，则放弃该状态的探索。

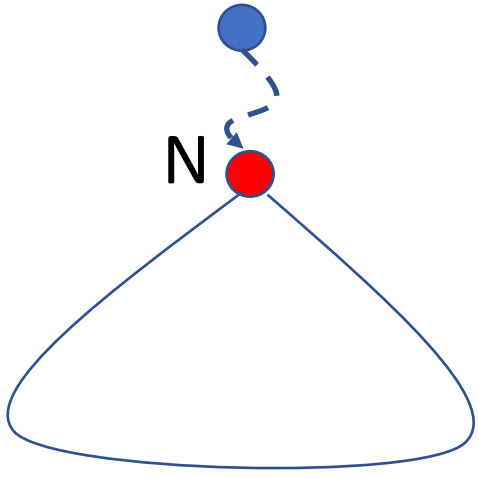


剪枝:分支限界

- 对于优化问题，如果从某个状态出发的所有解的最优值B不会优于当前已知的最优价值V，则停止从该状态的探索!
- 因此，为每个状态计算该状态下所有解的界，并在探索该状态之前将这个界与当前最佳解进行比较。

剪枝:分支限界

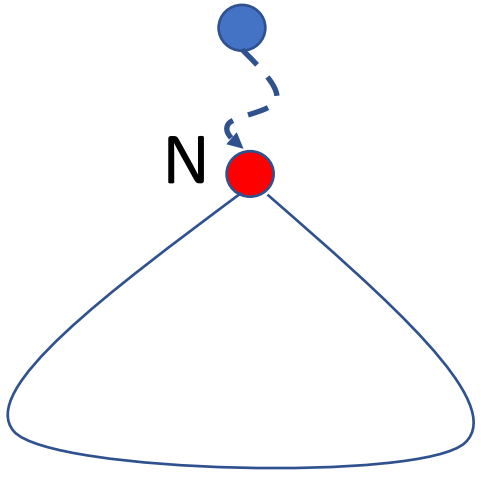
- 对于优化问题，如果从某个状态出发的所有解的最优值B不会优于当前已知的最优价值V，则停止从该状态的探索！



B(N)是状态N代表的状态树所有可能解价值的界

剪枝:分支限界

- 对于优化问题，如果从某个状态出发的所有解的最优值B不会优于当前已知的最优价值V，则停止从该状态的探索！



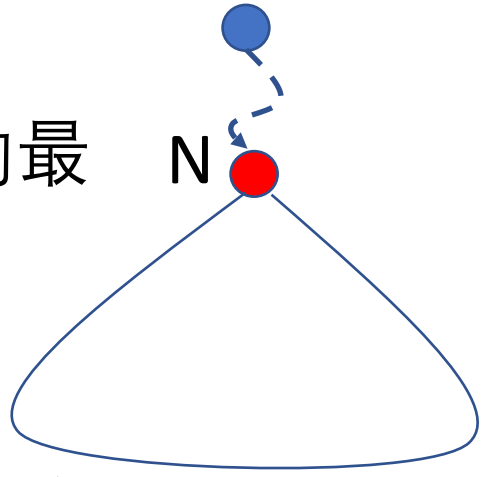
B(N)是状态N代表的状态树所有可能解价值的界

对于最小(大)值问题，这个界是所有可能价值的下(上)界。

剪枝:分支限界

- 对于优化问题，如果从某个状态出发的所有解的最优值B不会优于当前已知的最优价值V，则停止从该状态的探索!

V是当前已知解的最优值.



如B不优于V，则停止从N出发的探索

B(N)是状态N代表的状态树所有可能解价值的界

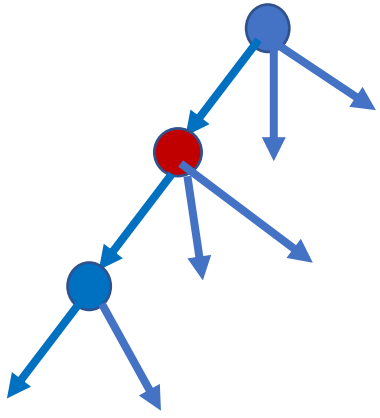
对于最小(大)值问题，这个界是所有可能价值的下(上)界。

背包问题：最大值

- 当前已有可行解的最大价值，记为V。
- 当前(状态)节点N子树的最大价值，记为B(N)。

if B(N) <V:

不需要再探索节点N了



- 限界函数：当前已有可能解的最优价值为V，状态N的估计价值为B(N)不优于V，则从该状态继续探索将不可能产生比V更优的解，因此，停止从状态N的探索！

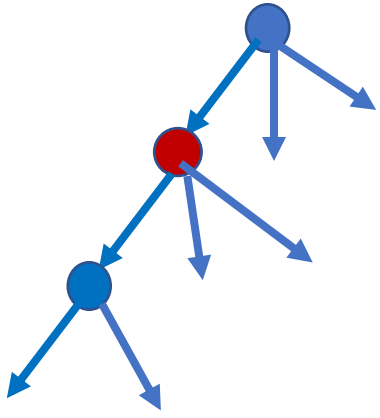
如背包问题：

已探索方案的最高价值

cv + remaining_value < bestV

已装物品价值

剩余物品价值



分支限界

根据**限界**停止节点**分支**的搜索:

- 分支的界B(N) **不优于** 当前可行解的最优值V
  - 对于极大值问题，　上界B(N) <V
  - 对于极小值问题，　下界B(N) >V

分支限界

基本问题:

- 如何估算节点的界?
- 如何估算当前最优解?
- 如何优化搜索过程（子节点选择策略）？

  启发式搜索：最可能产生最优解的先探索

剪枝：深度优先搜索

```
dfs(c) :  //c是当前状态
    if accept(c) :  output()

    for c的每个可达状态s
        if not reject(s) :
            dfs(s)
```

剪枝

剪枝：深度优先搜索

```
dfs(c) :  //c是当前状态
    if accept(c) :  output()

    for c的每个可达状态s
        push(s);
        if not reject(s) :
            dfs(s)
        pop()
```

剪枝

剪枝：广度优先搜索

```
bfs() :
    初始状态入队
    while 队列不空
        出队一个状态s
        if accept(s) :   output;
        else
            for s的每个未访问的子状态n,
                if not reject(n):
                    n入队
```

剪枝（分支限界）：广度优先搜索

bfs() :

初始化V为无穷大(最小值问题)或无穷小（最大值问题）
贪婪法得到的可行解更新V
初始化状态队列
while 队列不空
出队一个节点N
if N表示一个可行解x,　//如到达叶子状态
if f(x)优于V:　则V = f(x)
else
for 每个分支N_i,
if B(N_i)优于V:
N_i入队

剪枝（分支限界）：启发式搜索

```
function branch_and_bound (问题) :
    创建状态的优先级队列Q, 初始状态入队
    最佳解决方案 = 空
    if Q 不为空时:
        从 Q 中选择优先级最高的状态节点 P
        if P 不可行或其界比 best_solution 更差:
            剪去 P
        else if P 是叶节点:
            用 P 中的解决方案更新 best_solution
        else:
            for(P的未访问过的子节点N:
              计算N的界
              将子节点N添加到队列Q
    返回best_solution
```

N皇后

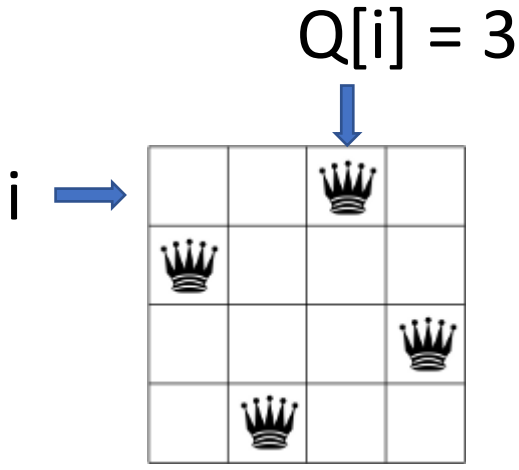
约束 剪枝

YouTube频道： hwdong

博客： hwdong-net.github.io

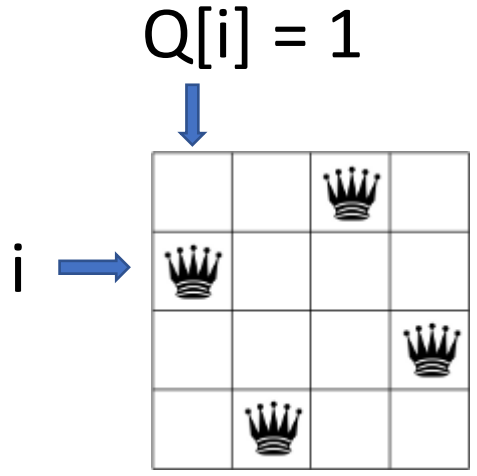
N皇后问题

- Q[i]表示第i行皇后的决策



3			

N皇后问题



- Q[i]表示第i行皇后的决策

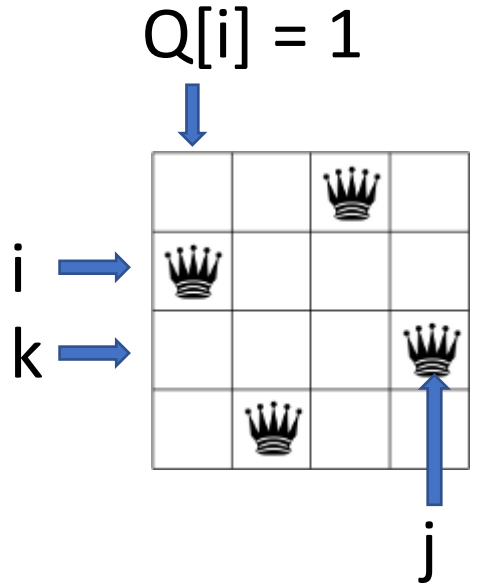
  Q[]数组记录了决策过程

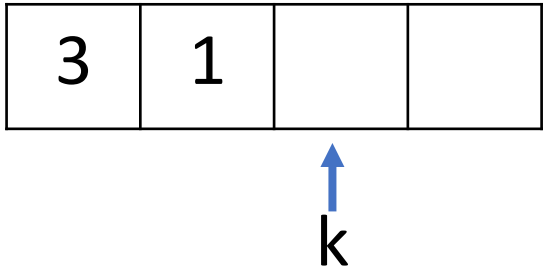
3	1		

决策过程栈

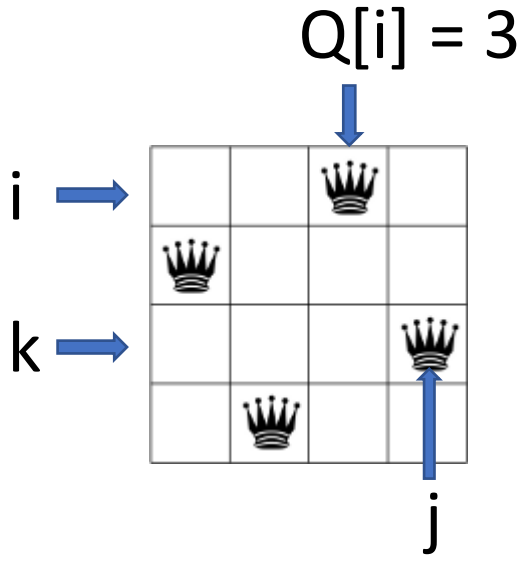
第k次决策：

```
NQueen( Q[n], k)
    if k==n+1: output Q
    for j=1 to n:
        valid= Okay(Q[k]←j)

        if valid:
              Q[k] = j
               NQueen(Q, k+1)
```





N皇后问题




```
NQueen( Q[n],k)
    if k==n+1: output Q
    for j=1 to n:
        valid=true
        for i =1 to k-1:
            if Q[i]==j or Q[i]-j==k-i or Q[i]-j==i-k:
                valid = false; break;
        if valid:
            Q[k] = j
            NQueen(Q, k+1)
```

$Q[i] \neq Q[k]$

$|Q[i]-Q[k]| \neq |i-k|$

约束剪枝

背包问题

分支限界 剪枝

YouTube频道： hwdong

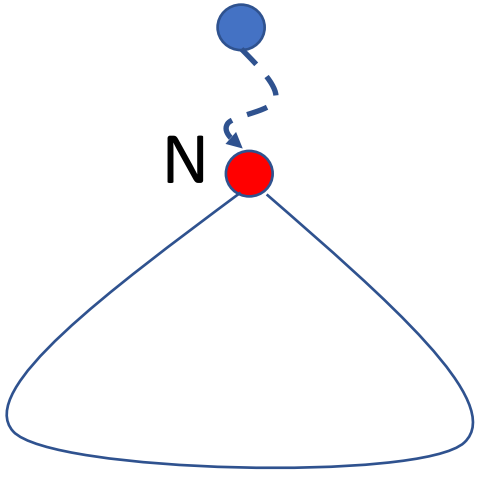
博客： hwdong-net.github.io

背包问题：估算状态的限界

按单位重量价值排序：

$v_1/w_1 >= v_2/w_2 >= \ldots >= v_n/w_n$

w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10



估算限界

$ub = cv+(W-cw)* v_{i+1}/w_{i+1}$

已装物品价值

已装物品重量

剩余物品最大单位重量价值

背包问题：估算状态的限界

```
ub = cv ; wt = cw; j=i+1
while j < =n and wt + w[j] <= W:
            ub+= v[j]
             wt += w[j]
             j += 1

//最后一物品的分数重量的价值
if j <= n:
    ub += (W - wt) * float(v[j])/ w[j]
```

背包问题：估算当前最优价值

- 1）当前解的最优价值作为最优价值V

缺点：必须等到找到可行解，才能更新V; 不适合广度优先

- 2）已装填物品的最大价值作为最优价值V。优点：随时更新

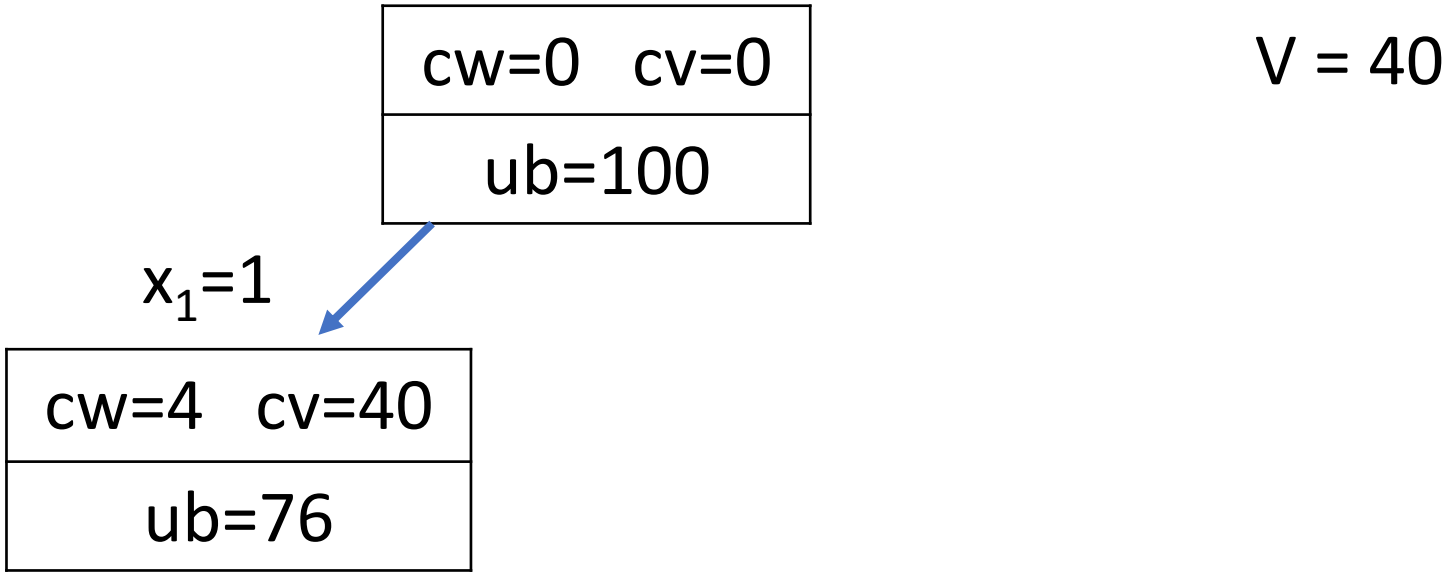
- w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10

cw=0 cv=0
ub=100

V = 0

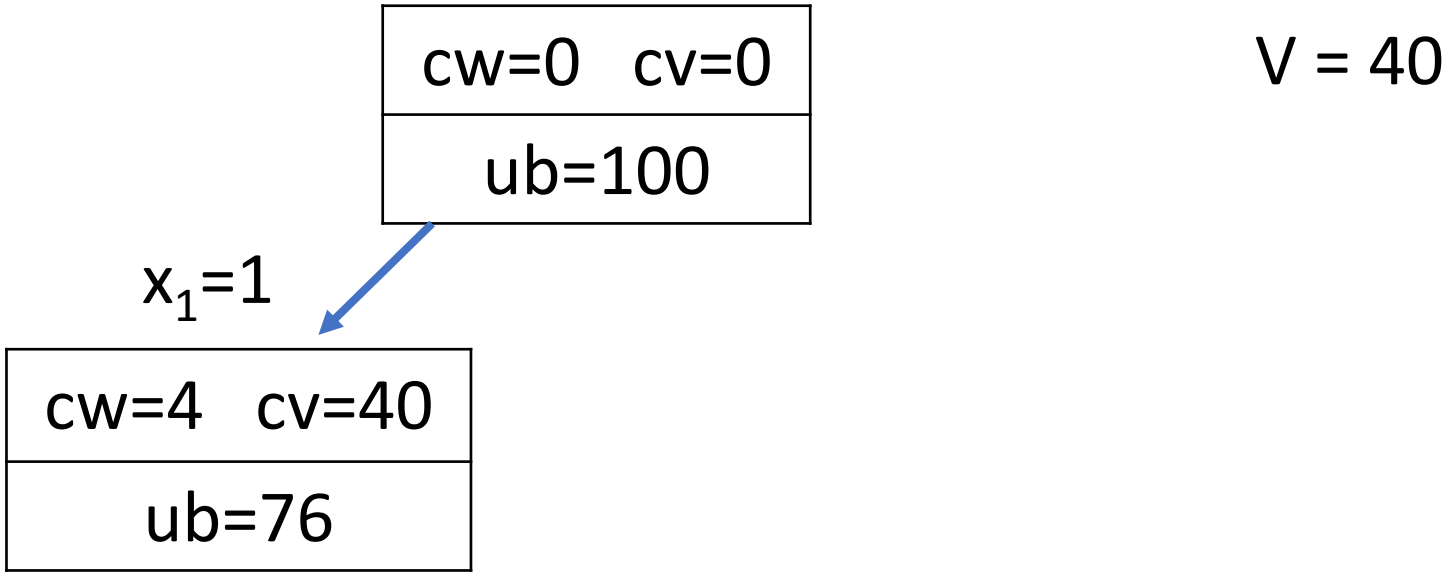
ub = 0+(10-0)* v_1/w_1 = 10*10=100

- w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10

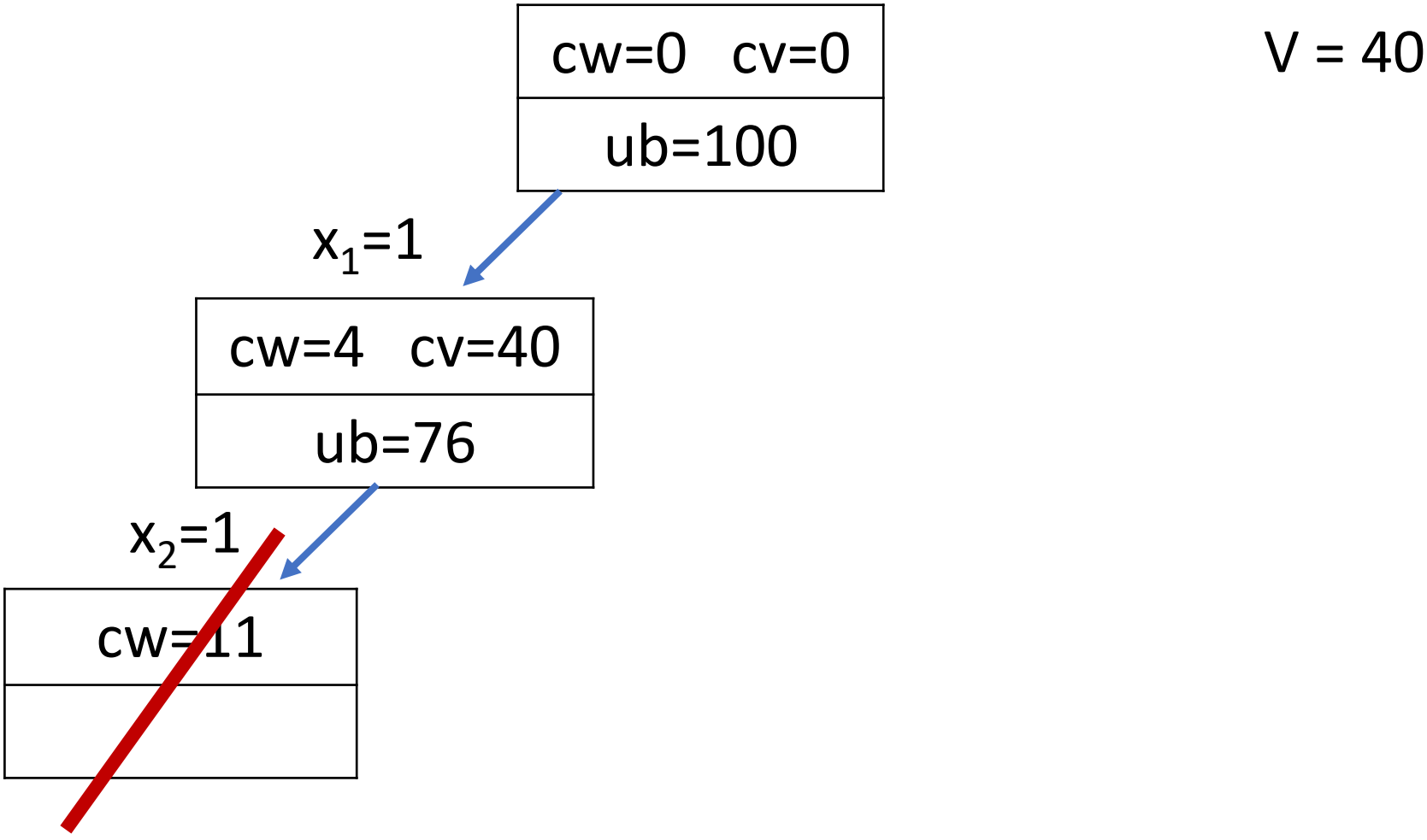


ub = 40+(10-4)* v_2/w_2 = 40+6*6=76

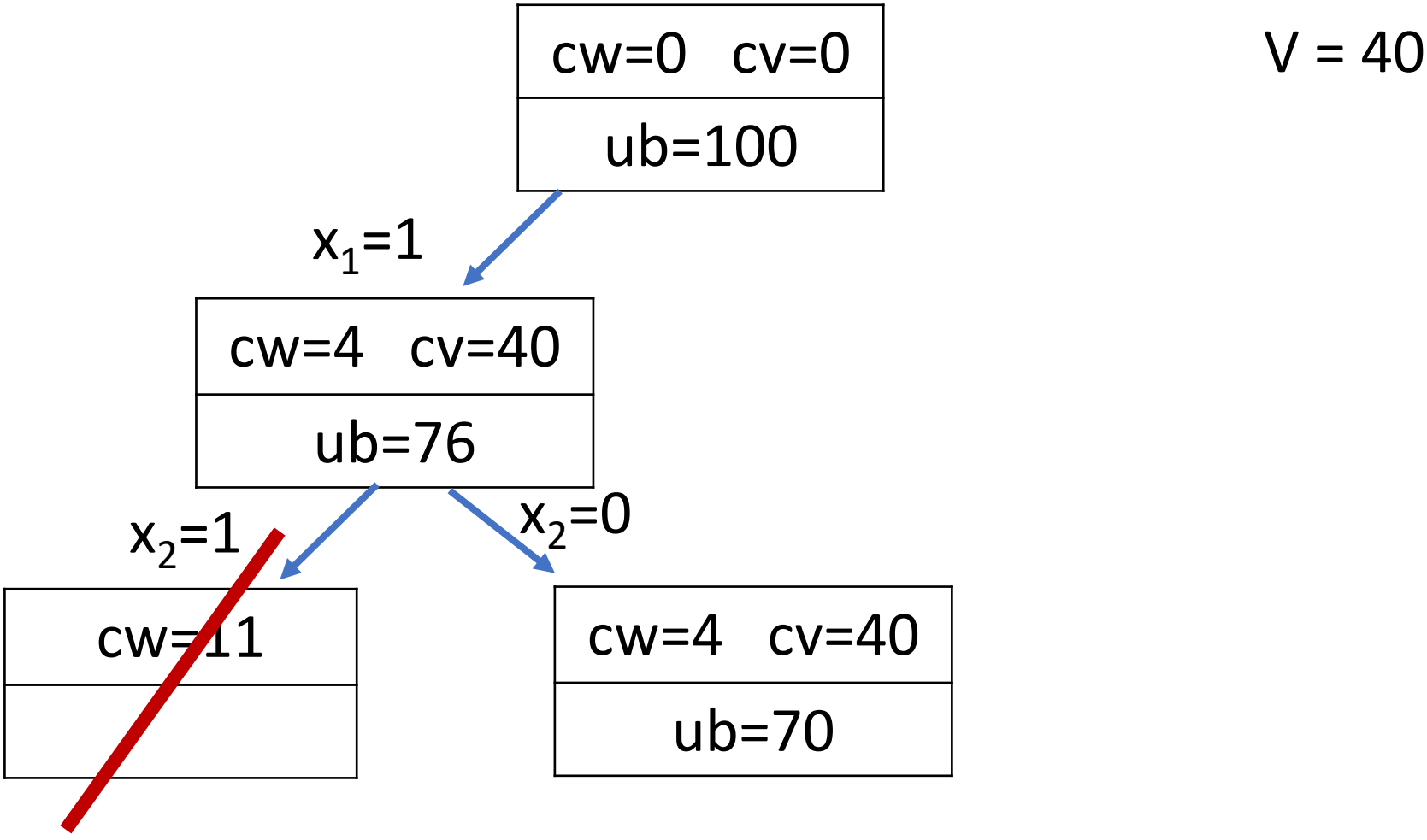
- w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10



• w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10

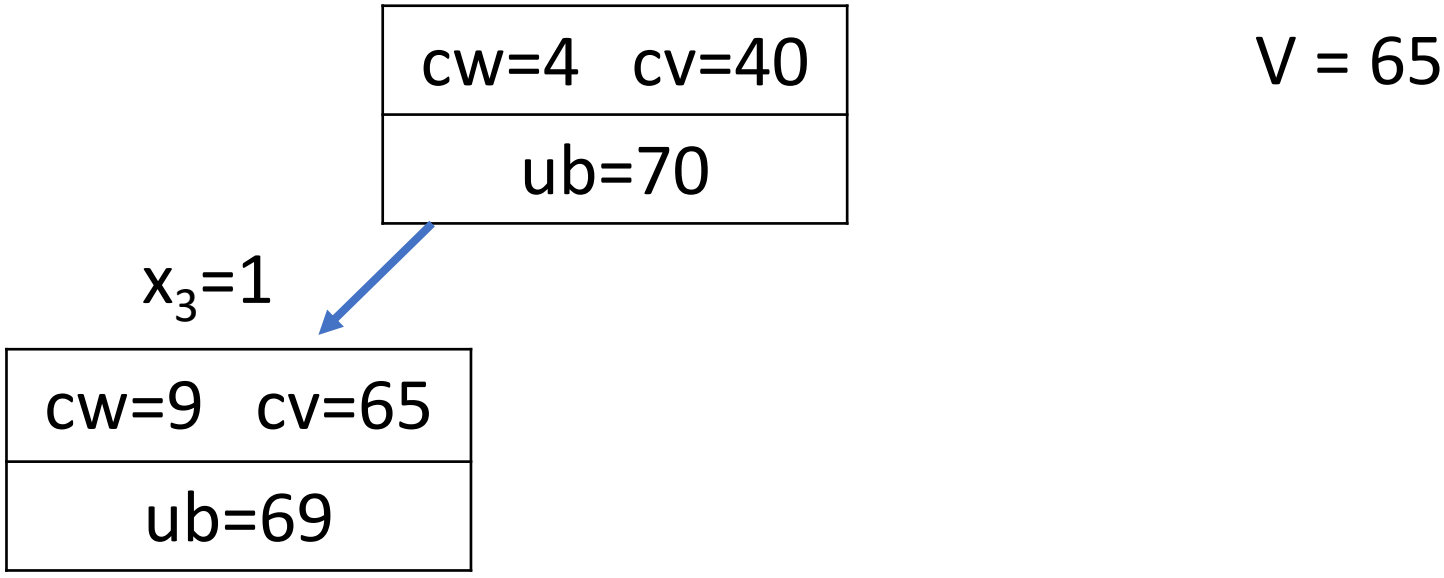


• w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10



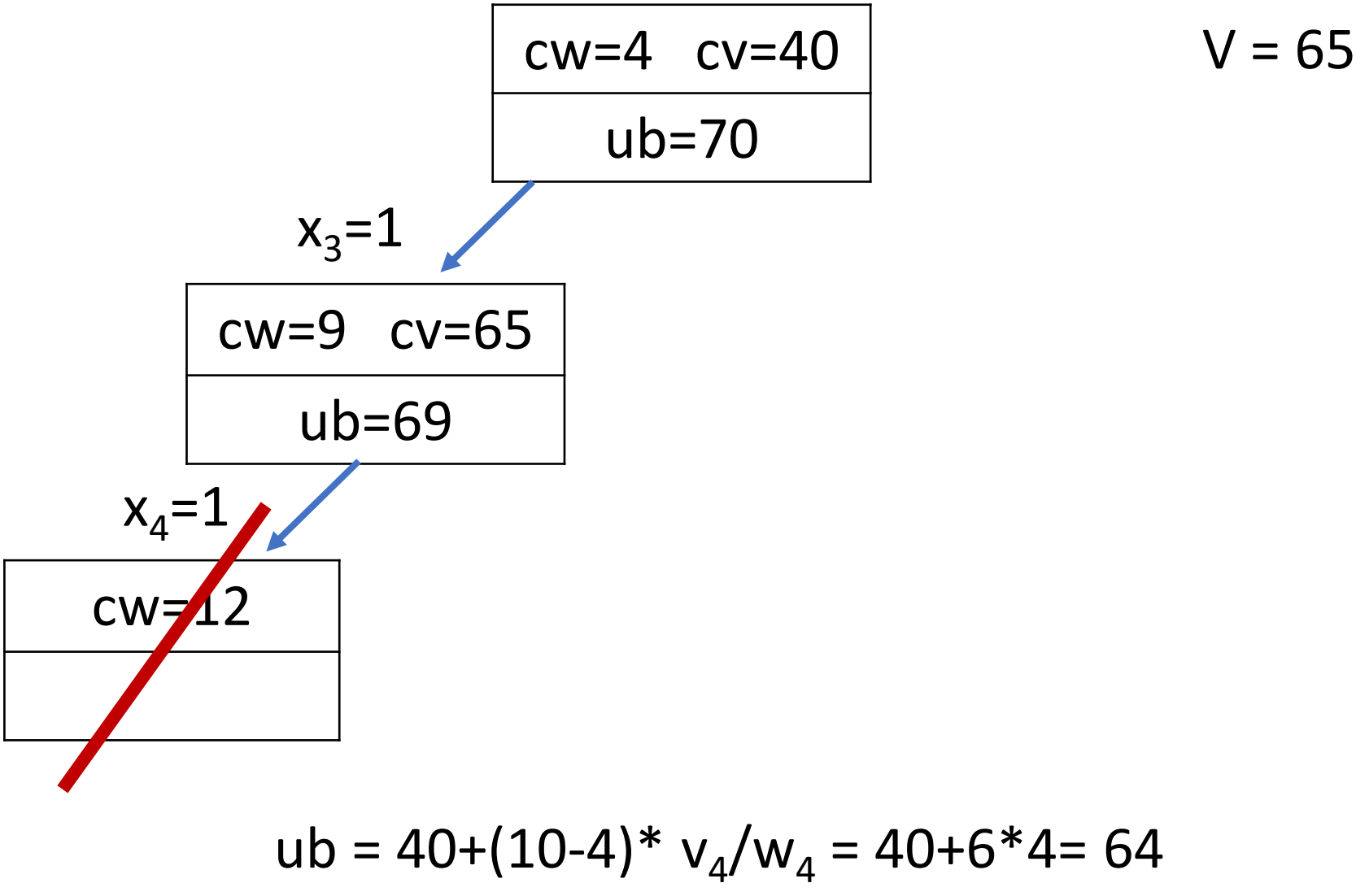
ub = 40+(10-4)* v_3/w_3 = 40+6*5=70

- w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10



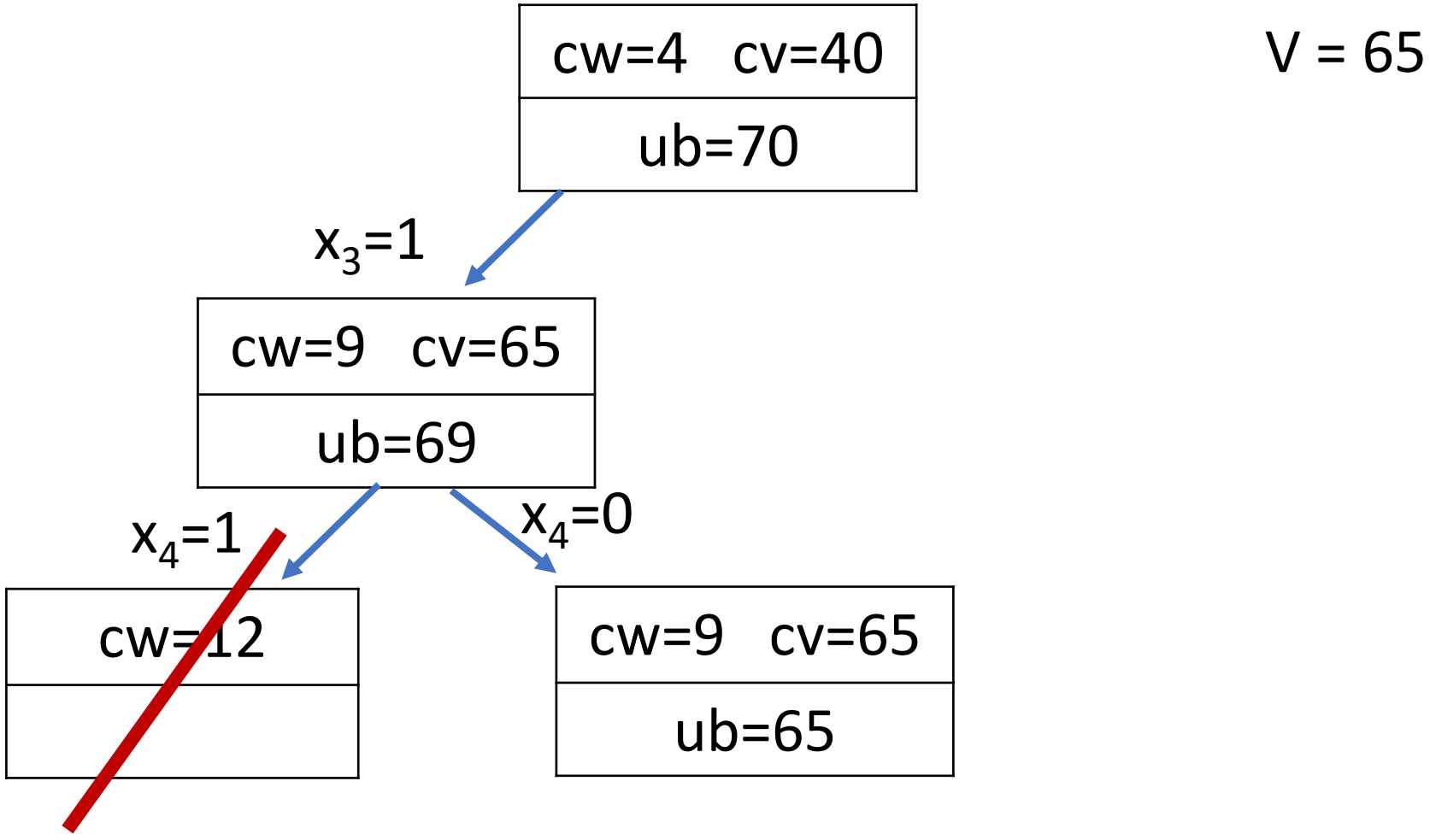
ub = 65+(10-9)* v_4/w_4 = 65+1*4= 69

• w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10

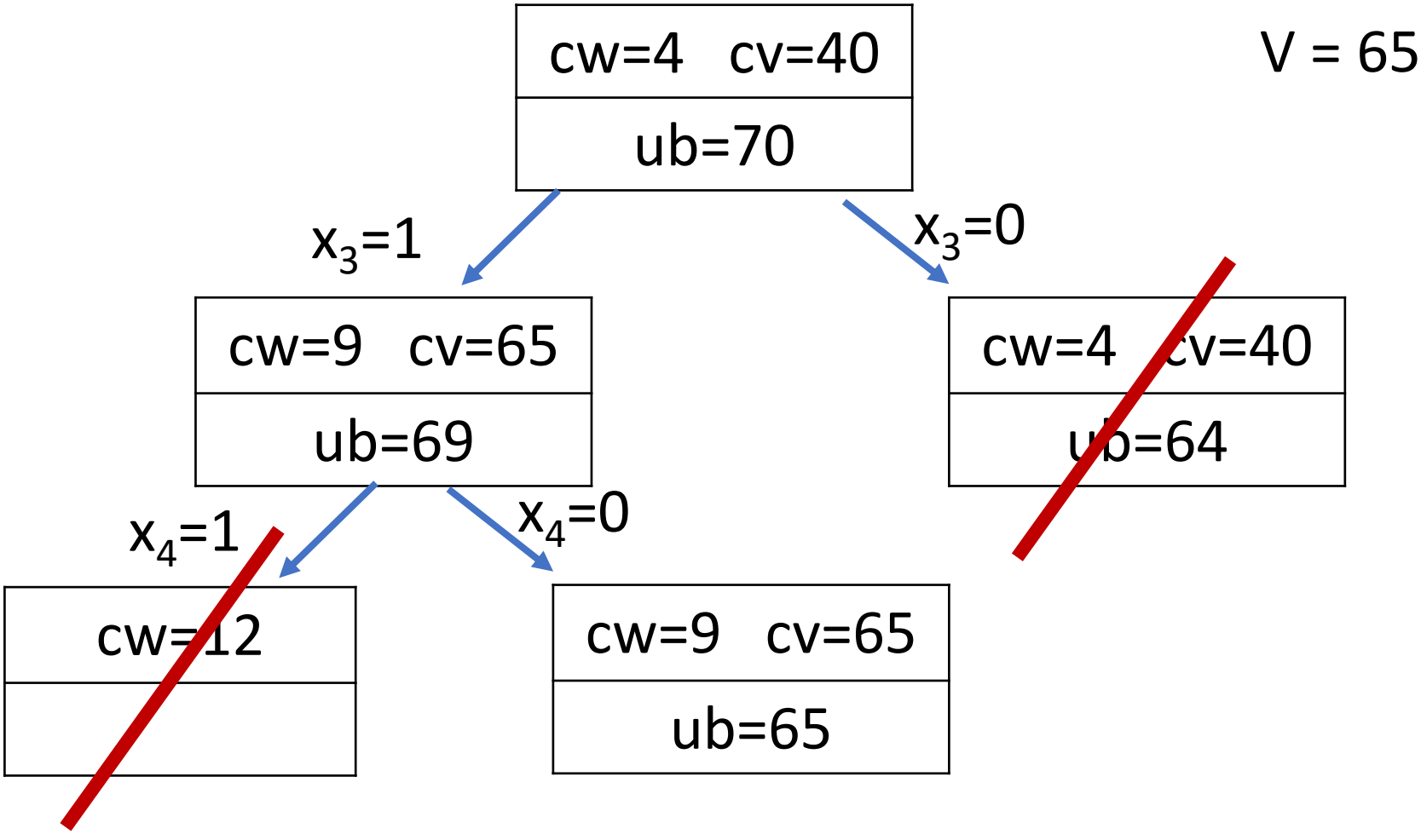


ub = 40+(10-4)* v_4/w_4 = 40+6*4= 64

• w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10

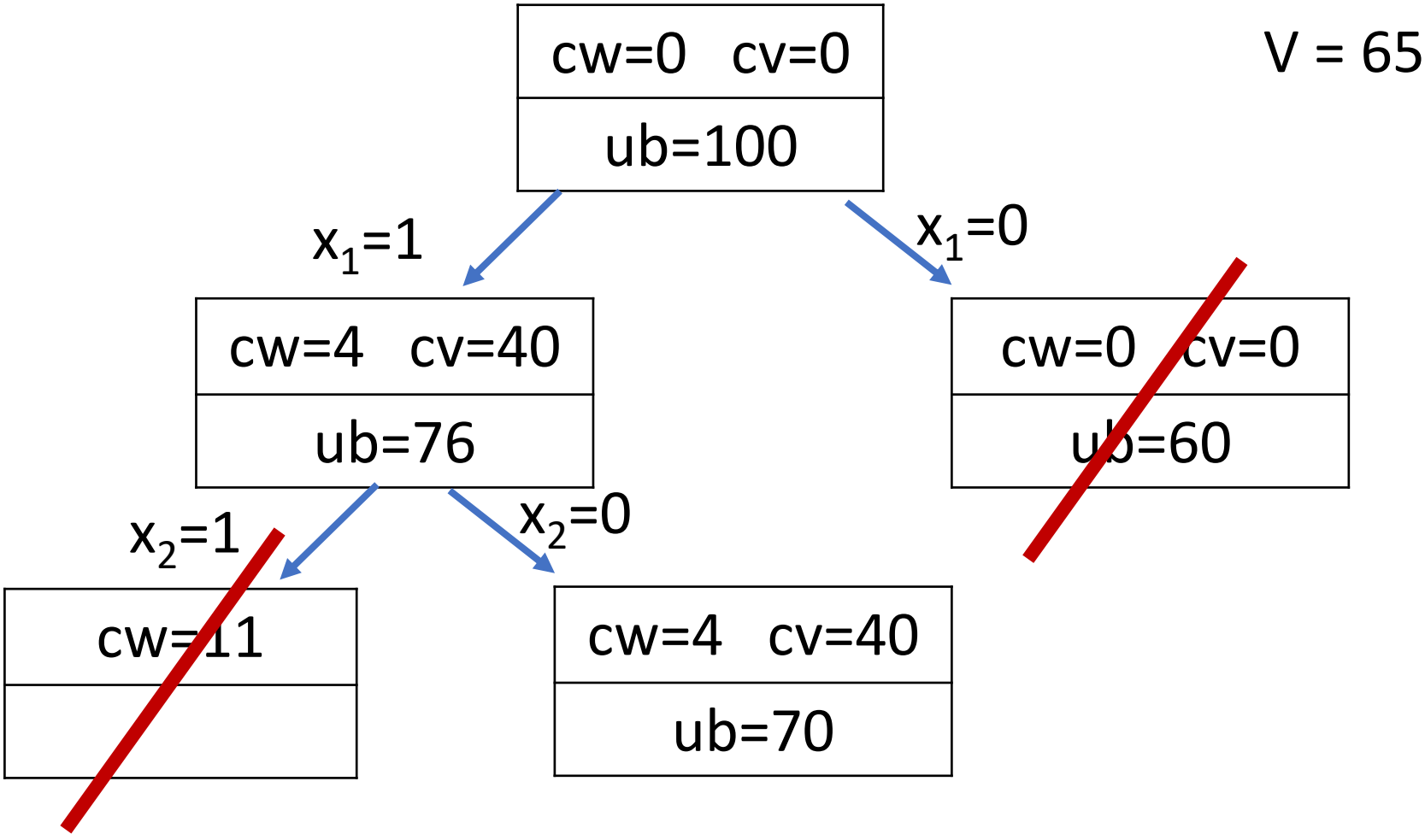


• w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10



ub = 40+(10-4)* v_4/w_4 = 40+6*4= 64

• w = (4,7,5,3) ,v = (40,42,25,12) ,W = 10



ub = 0+(10-0)* v_2/w_2 = 0+10*6=60

状态的界

```
bound(cv,cw,iW, w[], v[]) :
   if (cw+w[i] > W) return 0;
   T bound_value = cv;
   int j = i, n = w.size();
   T weight = cw;
   while j < n && weight + w[j] <= W:
      weight += w[j];
      bound_value += v[j]; j = j+1;

    if j<n:
       bound_value += (W - weight) * v[j]/w[j];
    return bound_value;
```

State

```
struct{
    Array path; //
    T cv,cw;
};
```

分支限界：深度优先搜索

```
DFS(S, V):    //S:=当前状态，V:= 启发式解的价值
  if S是目标状态：  更新V, return
  for(S的后续状态N):
      b = bound(N) //估算N状态的价值的界
      if b不优于V:
          continue   //停止该状态N的探索
      else:
          DFS(N, V): //从N状态继续探索
```

分支限界：深度优先搜索

```
knapsack_BB(W, w[], v[], S, &maxValue, &maxP)
   i = S.path.size()
   if i==w.size(): return
   b = bound(S.cv,S.cw,S.path.size(),W,w,v)
   if b<= maxValue:  return
   if S.cw+w[i] > W:
       S.path.push_back(false)
       knapsack_BB(W,w,v, S, maxValue, maxP)
   else:
```

状态

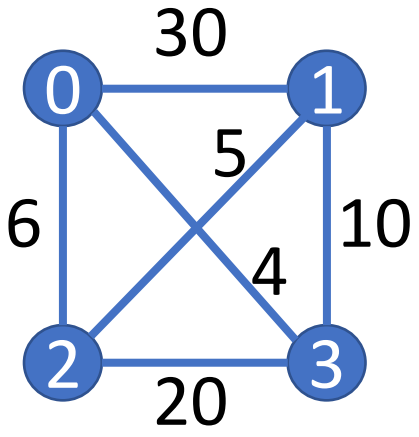
```
else:
    S.path.push_back(false);
    knapsack_BB(W, w, v,S, maxValue, maxP);
    S.path[i] = true;
    S.cv += v[i];
    S.cw += w[i];
   if (S.cv > maxValue) {
         maxValue = S.cv;
         maxP = S.path;
    }
```

广度优先遍历

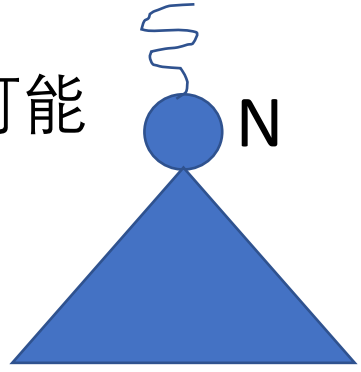
```
BFS():
   初始状态入队q;  V = 0
    while 队列q不空:
        S= q.front(); q.pop()
        if S是终止状态:
              continue;
        i= S的已考虑物品数目
        b = bound(i)
        if b<= V: continue;
        考虑物品i的放与不放的2个状态N:
            if N.cv>V:
                  V = N.cv;
               q.push(N)
```

```
knapsack_BB_BFS(W, w,v):
    queue q; State S; q.push(S);
    while !q.empty():
       S = q.front(); Q.pop();
       i = S.path.size();
       if i == w.size():    continue;
       b = bound(S.cv, S.cw, S.path.size(), W, w, v);
       if b <= maxValue:   continue;
       if (S.cw + w[i] > W) {
           auto N = S;   N.path.push_back(false);queue.push(N);
       }
      else {
          auto N = S; N.path.push_back(false);queue.push(N);
          N = S; N.path.push_back(true);   N.cv += v[i]; N.cw += w[i];
          if N.cv > maxValue:
                maxValue = N.cv;           maxP = N.path;
          queue.push(N);
     }
}
```

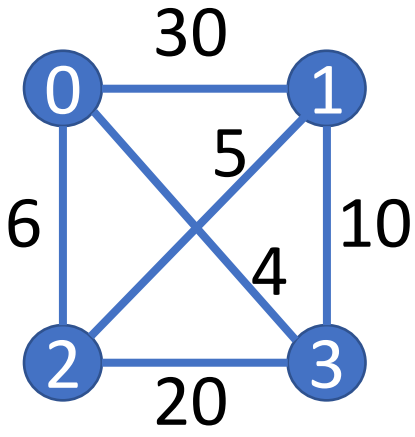
TSP-分支限界



- 这是一个最小化问题，
- 设当前最优解价值是B,
- lb(N)是状态N子树的所有可能价值的最小值

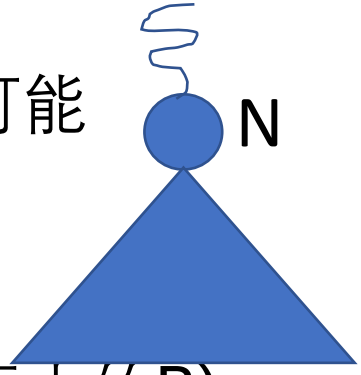


TSP-分支限界

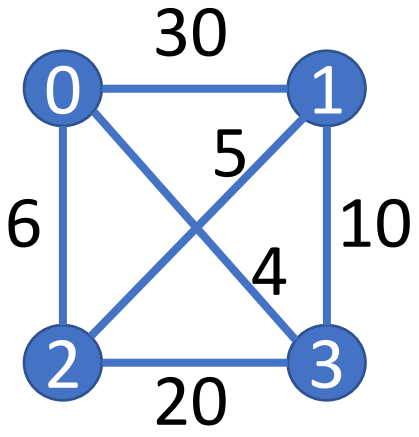


- 这是一个最小化问题，
- 设当前最优解价值是B,
- lb(N)是状态N子树的所有可能价值的最小值
- if lb(N) >B:

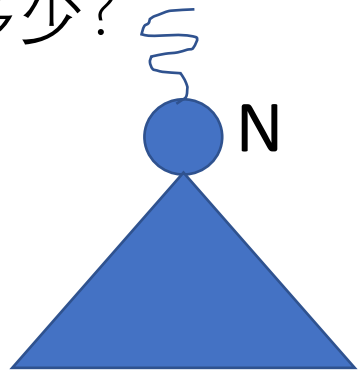
  舍弃N子树(不可能产生更小的B)
- else:可探索N



TSP-分支限界

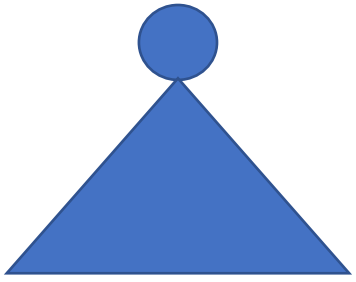


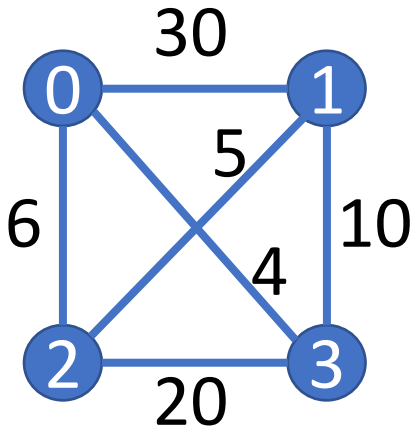
- 对任一状态N,
  其代表子树的最小的界是多少?



初始状态的(下)界

- 初始状态从顶点v_0出发
- 设路径为(v_0, v_1, v_2, v_3)

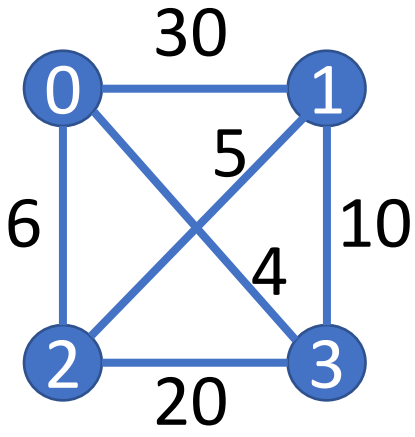




$w(v_0v_1)+ w(v_1v_2)+ w(v_2v_3)+ w(v_3v_0)$的值最小不小于?

$w(v_0v_1)+ w(v_3v_0)>=$ v_0关联的2个最小边权之和
$w(v_0v_1)+ w(v_1v_0)>=$ v_1关联的2个最小边权之和
$w(v_1v_2)+ w(v_2v_3)>=$ v_2关联的2个最小边权之和
$w(v_2v_3)+ w(v_3v_0)>=$ v_3关联的2个最小边权之和

初始状态的(下)界



$2(w(v_0v_1)+ w(v_1v_2)+ w(v_2v_3)+ w(v_3v_0))>=$

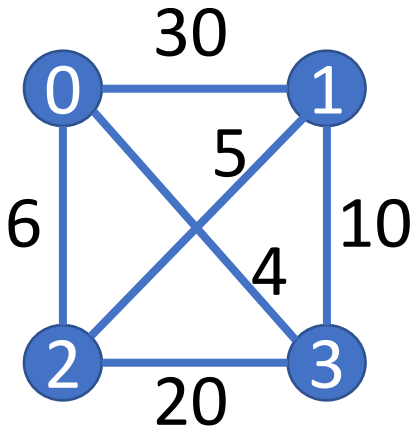
(v_0关联的2个最小边权之和+

v_1关联的2个最小边权之和+

v_2关联的2个最小边权之和+

v_3关联的2个最小边权之和)

初始状态的(下)界



$w(v_0v_1)+ w(v_1v_2)+ w(v_2v_3)+ w(v_3v_0)>=$

(v_0关联的2个最小边权之和+

v_1关联的2个最小边权之和+

v_2关联的2个最小边权之和+

v_3关联的2个最小边权之和)/2

lb = [(4+6)+(5+10)+(6+5)+(4+10)]/2 = 50/2=25

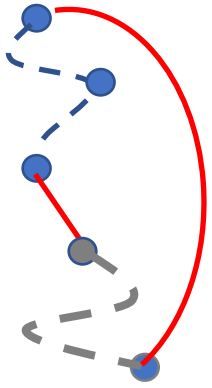
某个状态N的下界lb?

- 设N = (v_0,v_1,…,v_k)已经从顶点v_0到达了v_k,其

lb= $\sum_{i=0}^{k-1} w(v_i, v_{i+1})$

+[$\sum_{i=0,k}$ 不在路径上的最小边权

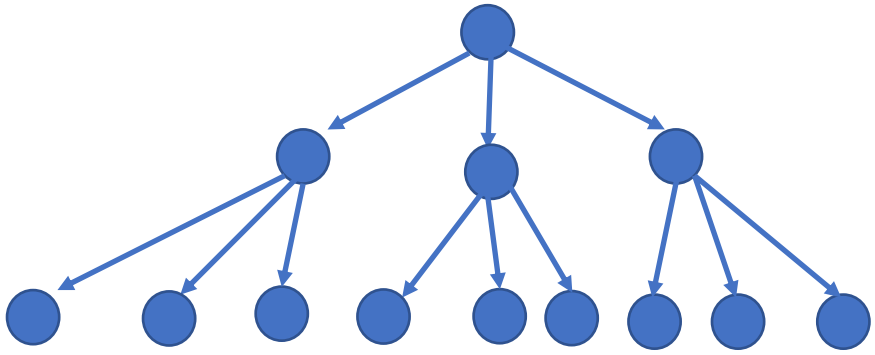
+ $\sum_{i=k+1}^{n}$不在路径上的顶点v_i关联的2个最小边权之和]/2



当前的最优价值V=？

V= ∞

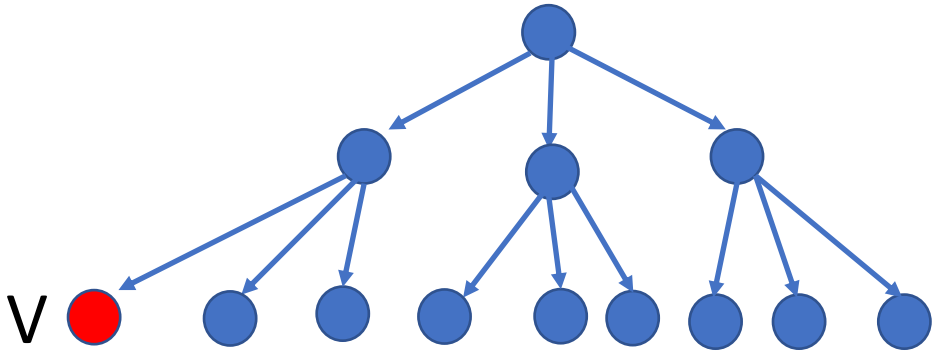
- 如果V=∞，采用广度优先搜索



当前的最优价值V=？

V= ∞

- 如果V=∞，采用广度优先搜索

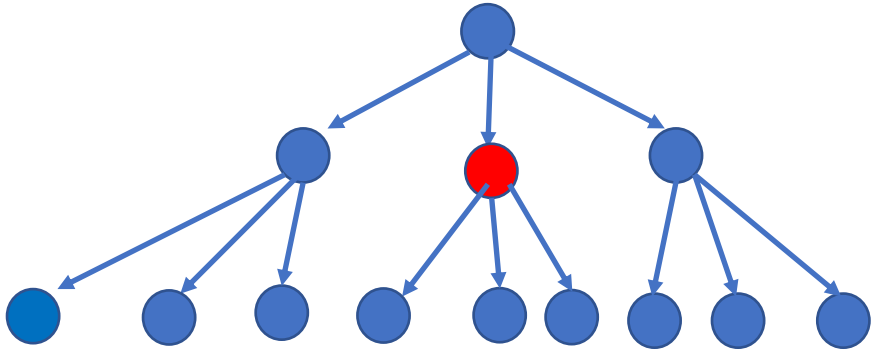


- 需要到达叶子结点层，才能开始用限界剪枝!
- 限界剪枝意义不大了！如何解决?

当前的最优价值V=？

V= ∞

- 如果V=∞，采用广度优先搜索

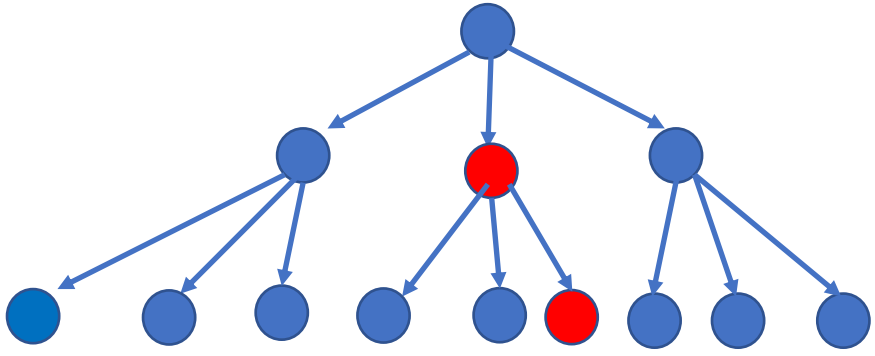


- 深度和广度搜索结合：优先队列，限界小的状态先探索

当前的最优价值V=？

V= ∞

- 如果V=∞，采用广度优先搜索

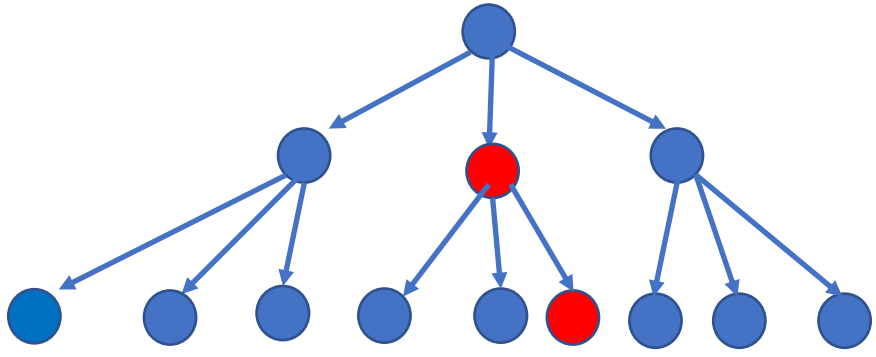


- 深度和广度搜索结合：优先队列，限界小的状态先探索

当前的最优价值V=？

$V= \infty$

- 一开始采用其他算法得到一个可行解，其价值V_0作为开始最优价值$V= V_0$

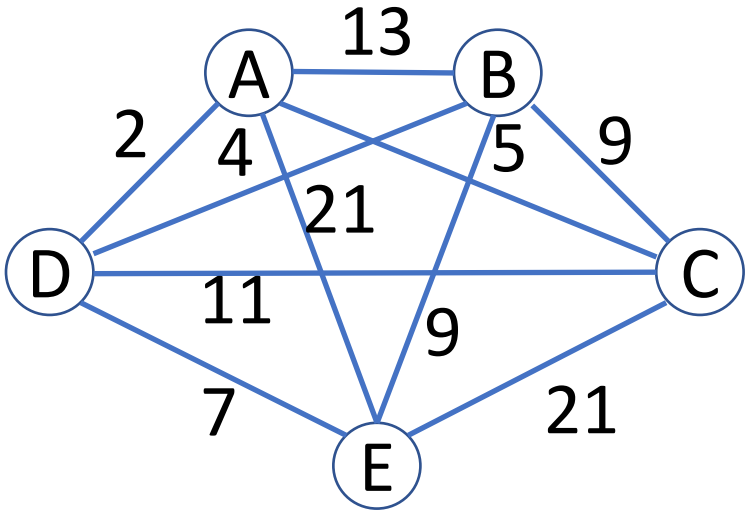


- 再广度优先搜索或者基于优先队列的广度与深度结合搜索

最近邻方法 Nearest Neighbor Method

- 随机选择出发城市
- while 还有未访问城市

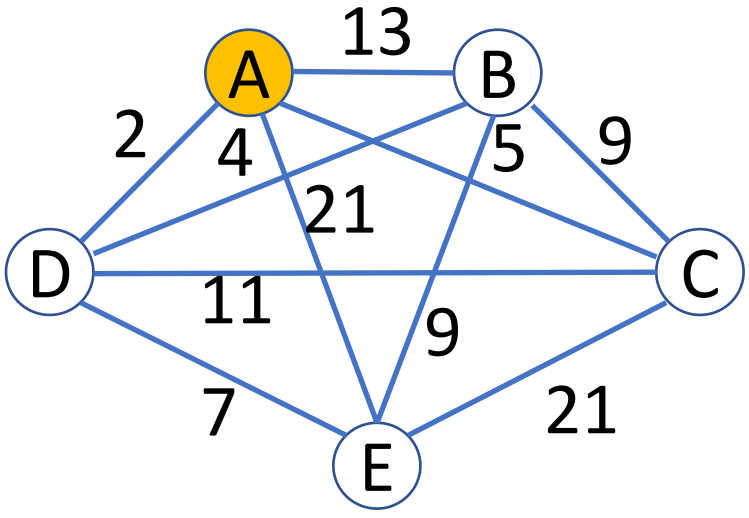
  走到距出发城市最近的未访问城市



最近邻方法 Nearest Neighbor Method

- 随机选择出发城市
- while 还有未访问城市

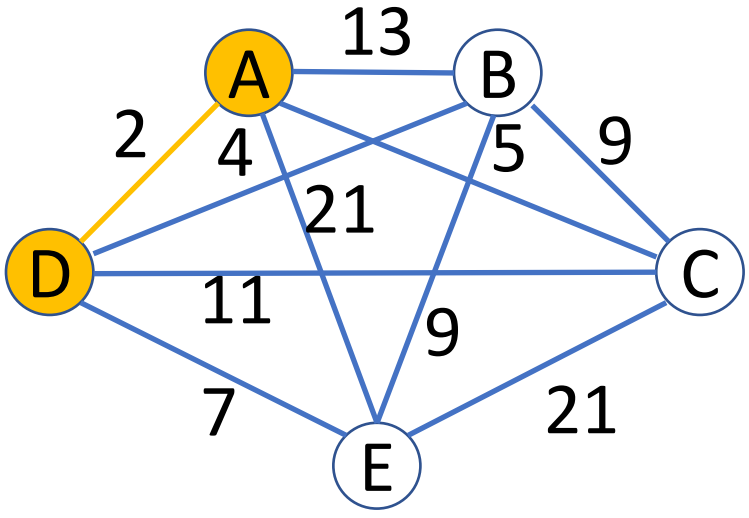
  走到距出发城市最近的未访问城市



最近邻方法 Nearest Neighbor Method

- 随机选择出发城市
- while 还有未访问城市

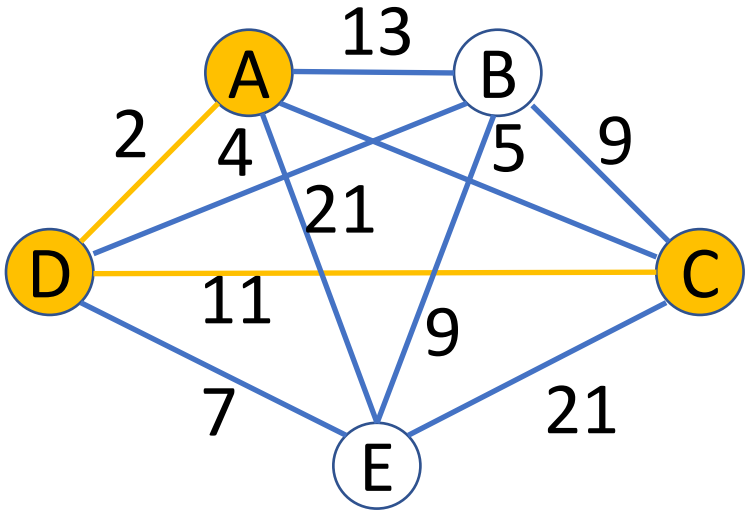
  走到距出发城市最近的未访问城市



最近邻方法 Nearest Neighbor Method

- 随机选择出发城市
- while 还有未访问城市

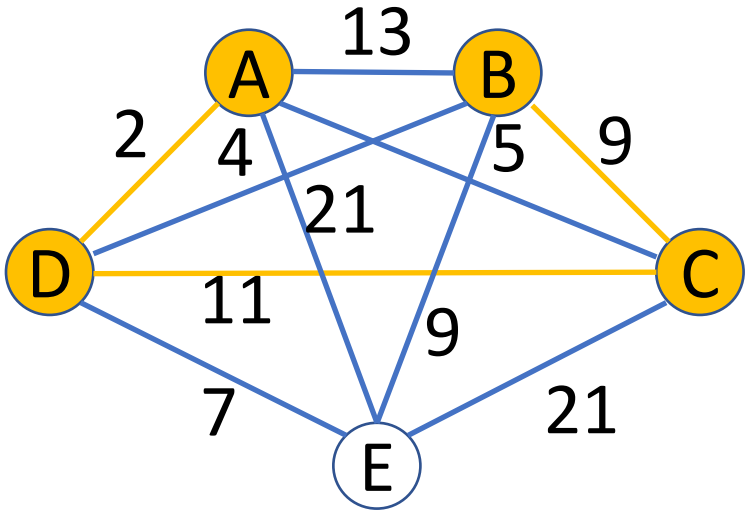
  走到距出发城市最近的未访问城市



最近邻方法 Nearest Neighbor Method

- 随机选择出发城市
- while 还有未访问城市

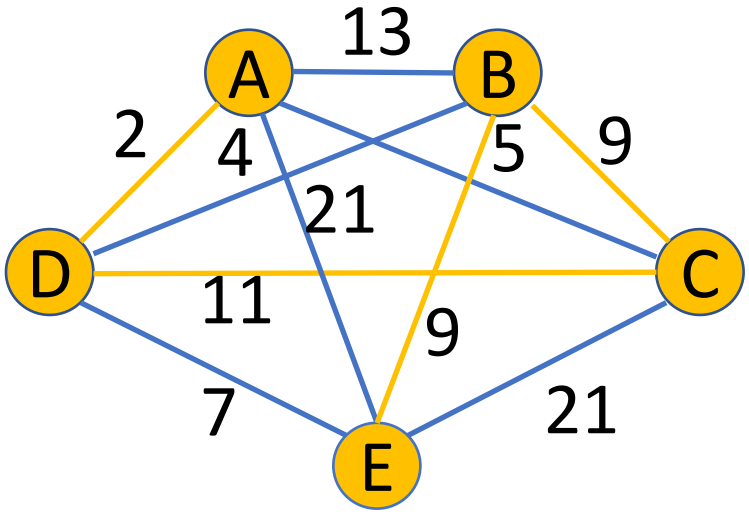
  走到距出发城市最近的未访问城市



最近邻方法 Nearest Neighbor Method

- 随机选择出发城市
- while 还有未访问城市

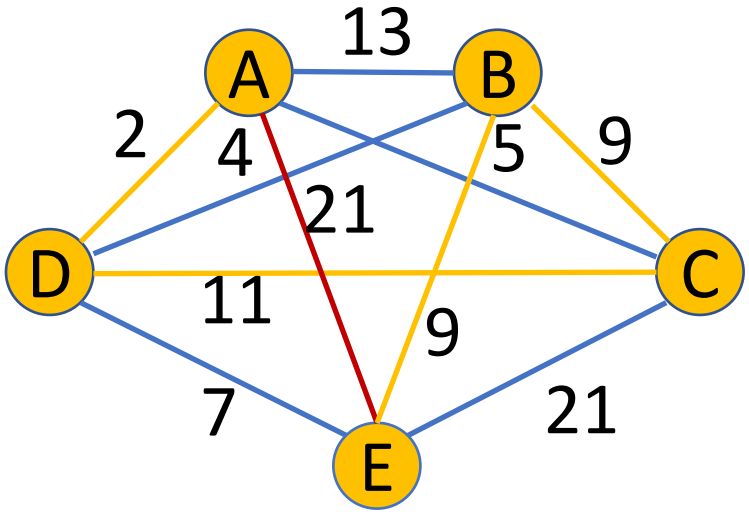
  走到距出发城市最近的未访问城市



最近邻方法 Nearest Neighbor Method

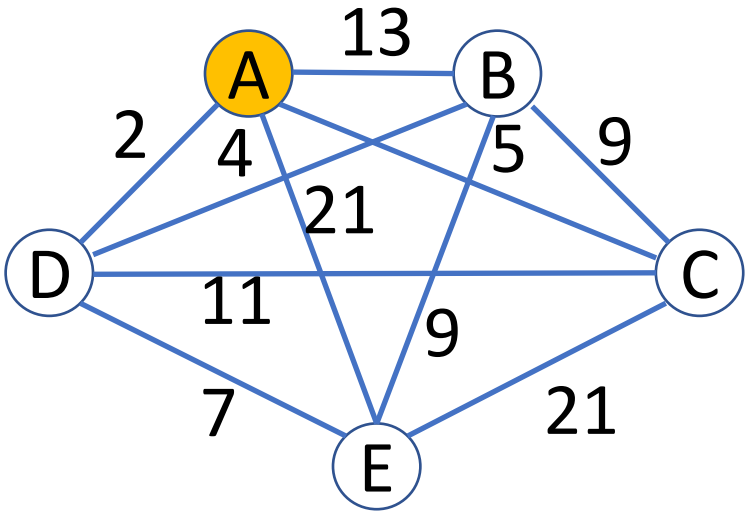
- 随机选择出发城市
- while 还有未访问城市

  走到距出发城市最近的未访问城市



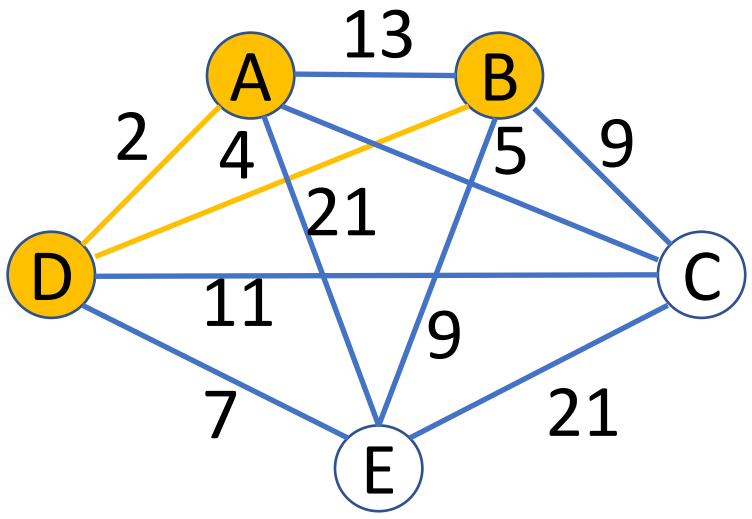
贪婪法 Greedy

- 总是选择距离当前城市最近的未访问城市



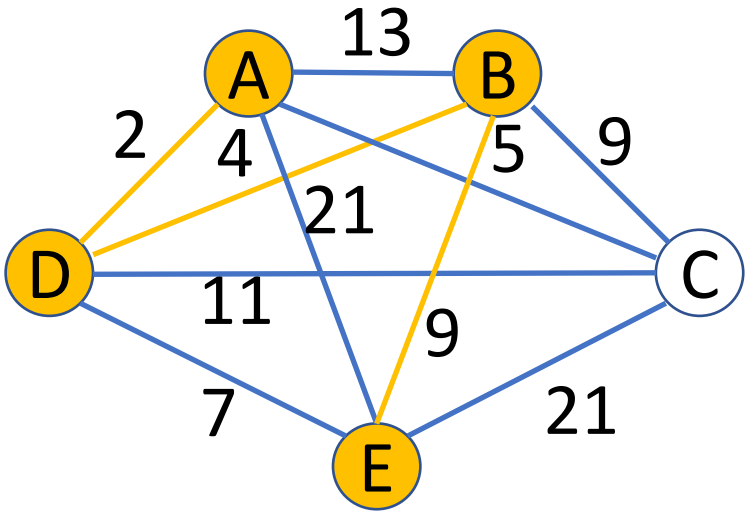
贪婪法 Greedy

- 总是选择距离当前城市最近的未访问城市



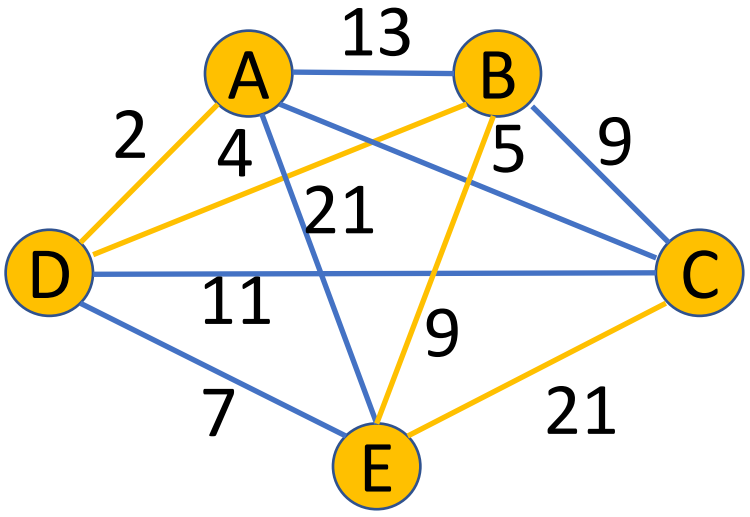
贪婪法 Greedy

- 总是选择距离当前城市最近的未访问城市



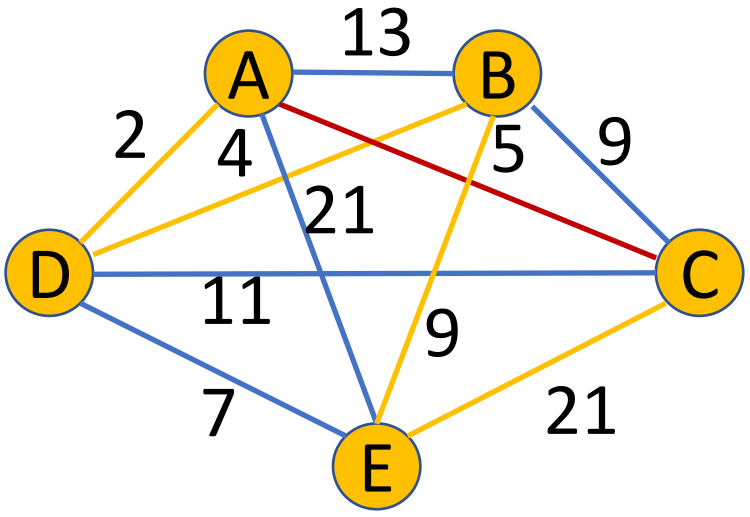
贪婪法 Greedy

- 总是选择距离当前城市最近的未访问城市

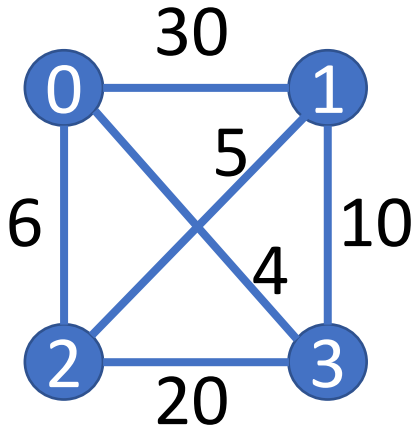


贪婪法 Greedy

- 总是选择距离当前城市最近的未访问城市



TSP用贪婪法得到一个可行解

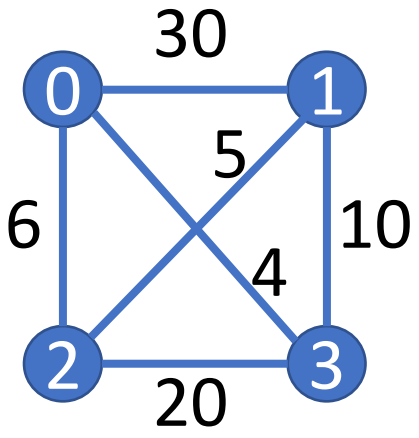


- 0->3->1->2->0
- 4+10+5+6 = 25
- 因此，可行解的价值是25，
- 作为最小值问题的上界V = 25，

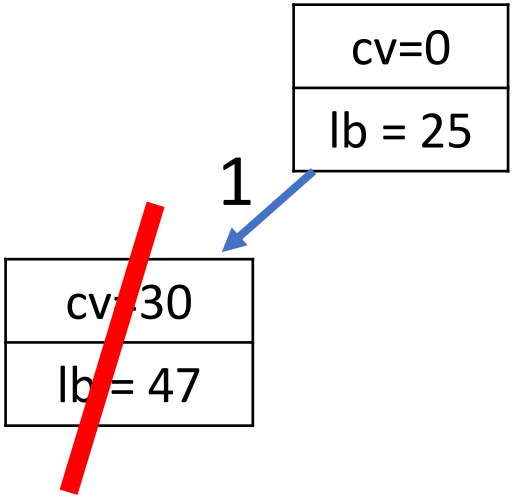
其他解的价值只有低于V，才能更优。

cv=0
lb = 25

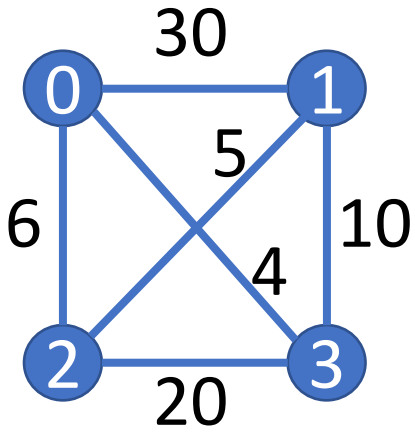
V=25



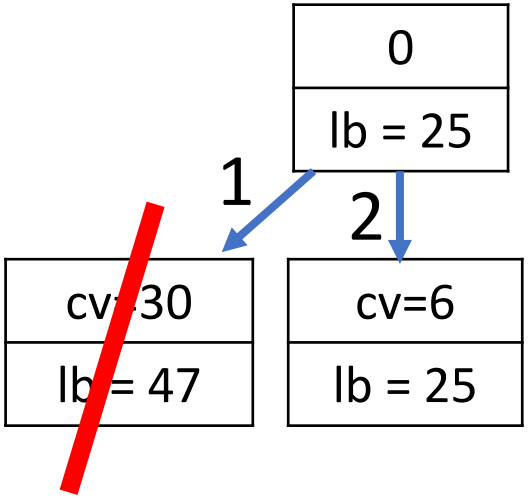
lb = [(4+6)+(5+10)+(6+5)+(4+10)]/2 = 50/2=25



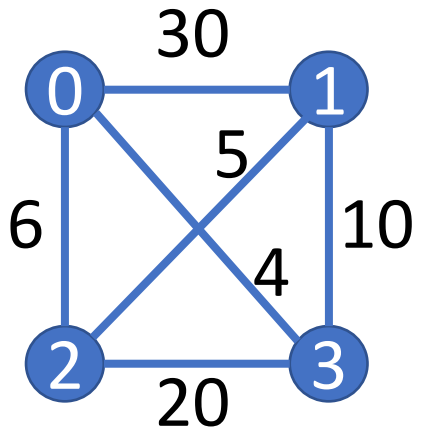
V=25



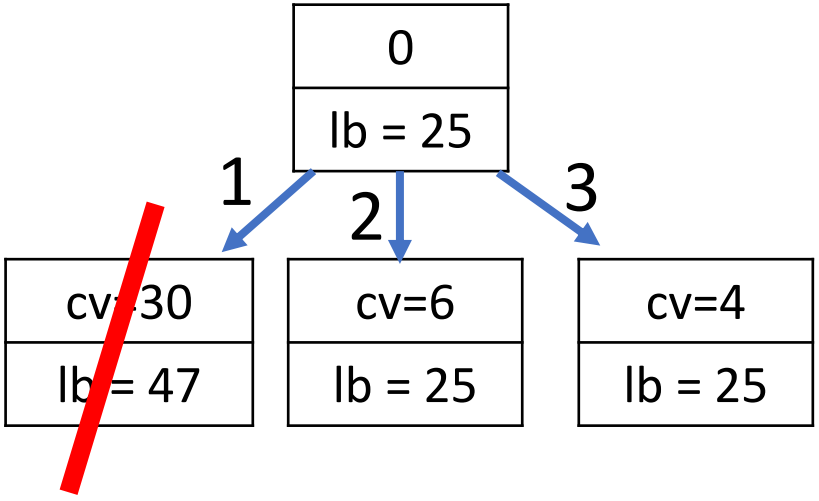
lb = 30+[(4+5)+(5+6)+(4+10)]/2 = 30+34/2=47



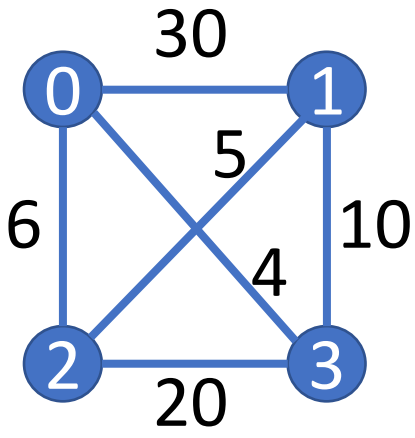
V=25



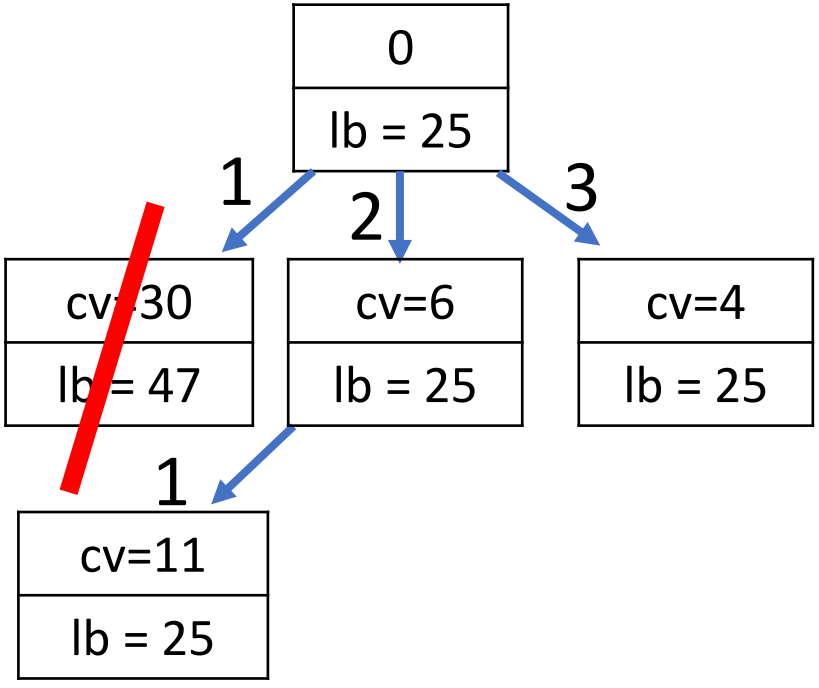
lb = 6+[(4+5)+(5+10)+(4+10)]/2 = 6+19=25



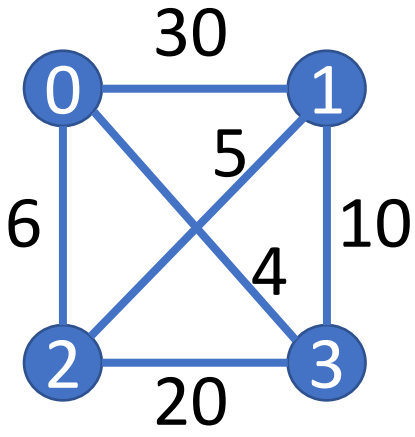
V=25



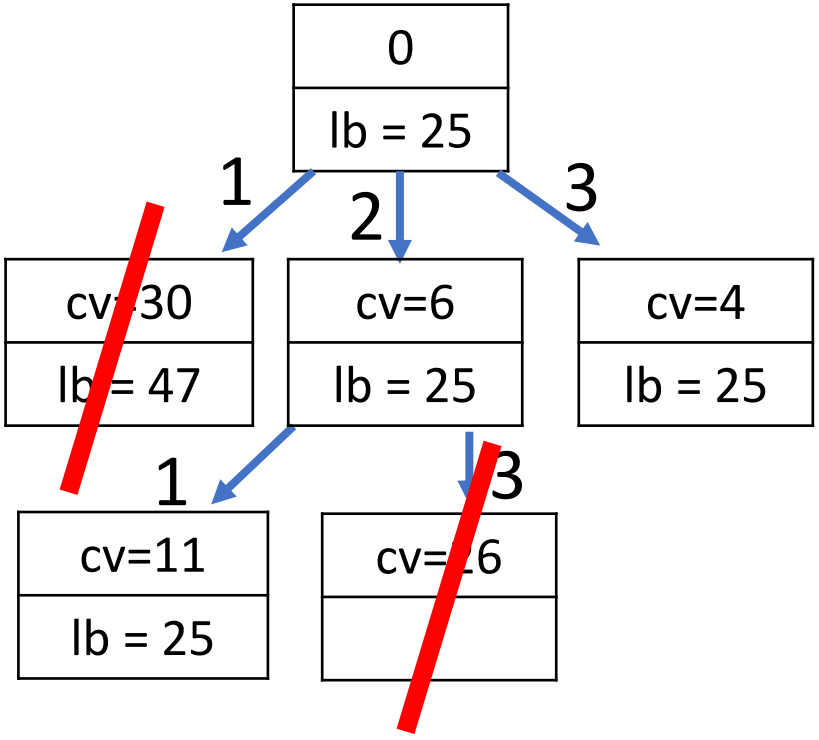
lb = 4+[(6+10)+(5+10)+(6+5)]/2 = 4+21=25



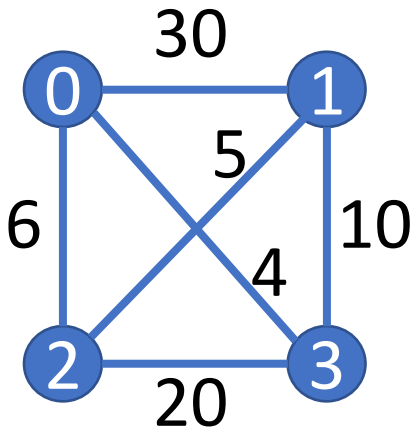
V=25

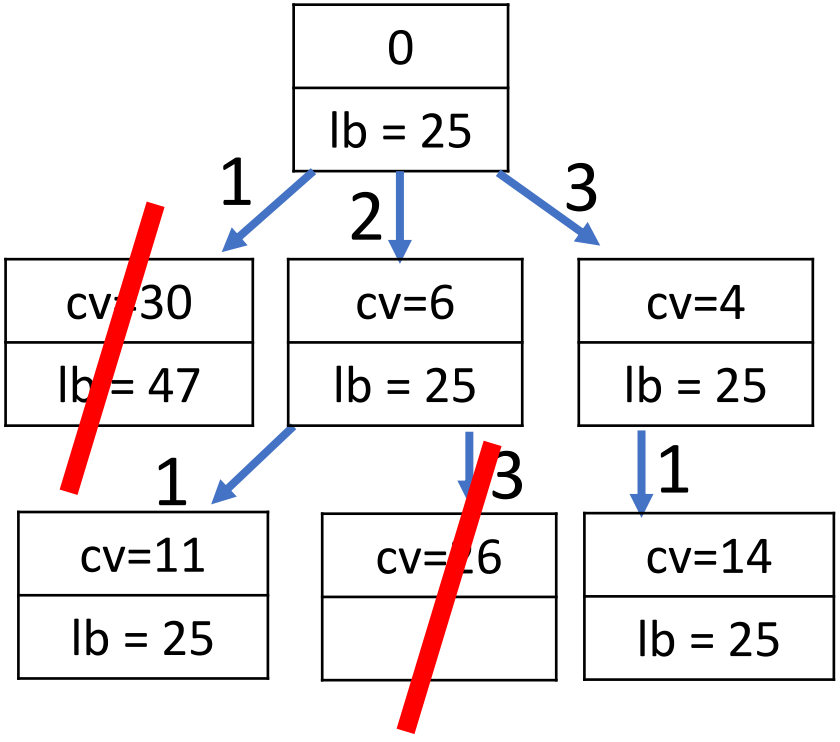


lb = 11+[(4+10)+(4+10)]/2 = 11+14=25

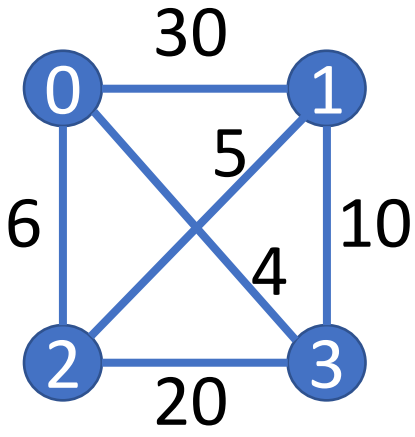

0
lb = 25
1
2
3
cv=30
lb = 47
cv=6
lb = 25
cv=4
lb = 25
1
3
cv=11
lb = 25

V=25

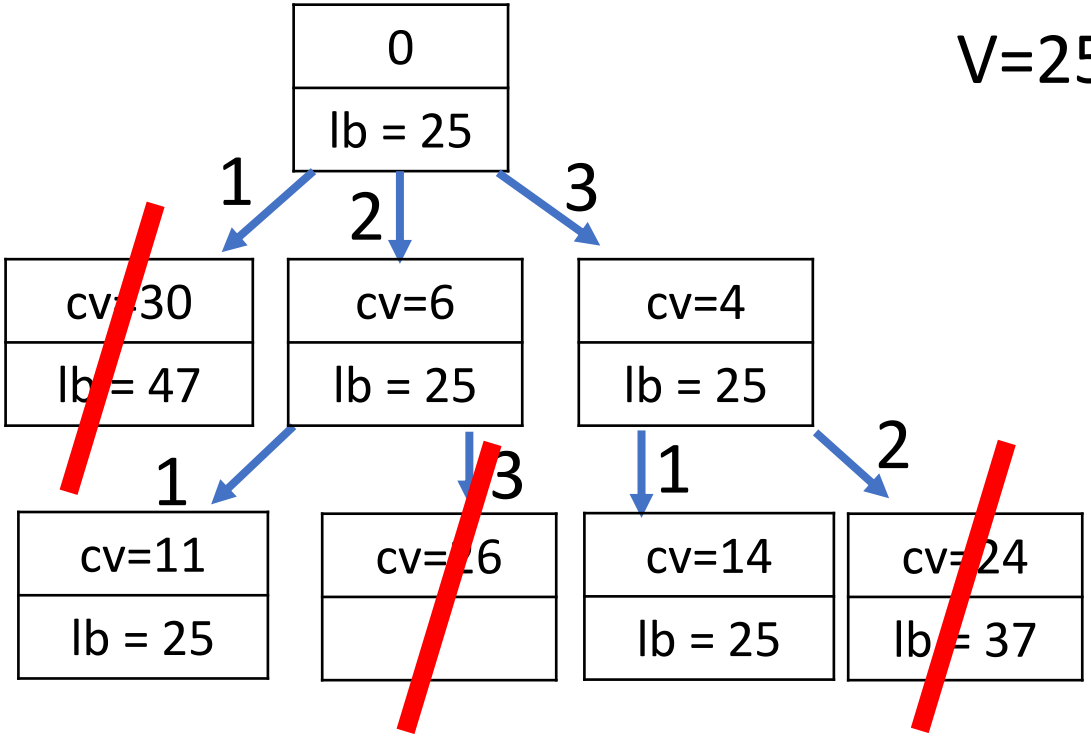

30
0
1
5
6
10
4
2
3
20



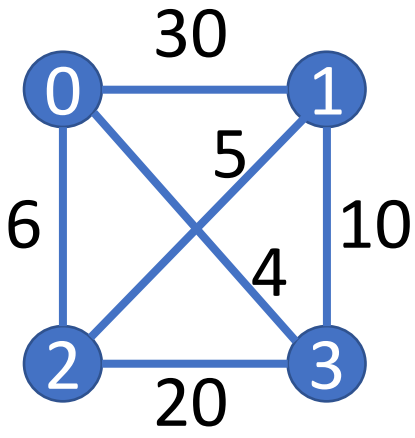
B=25



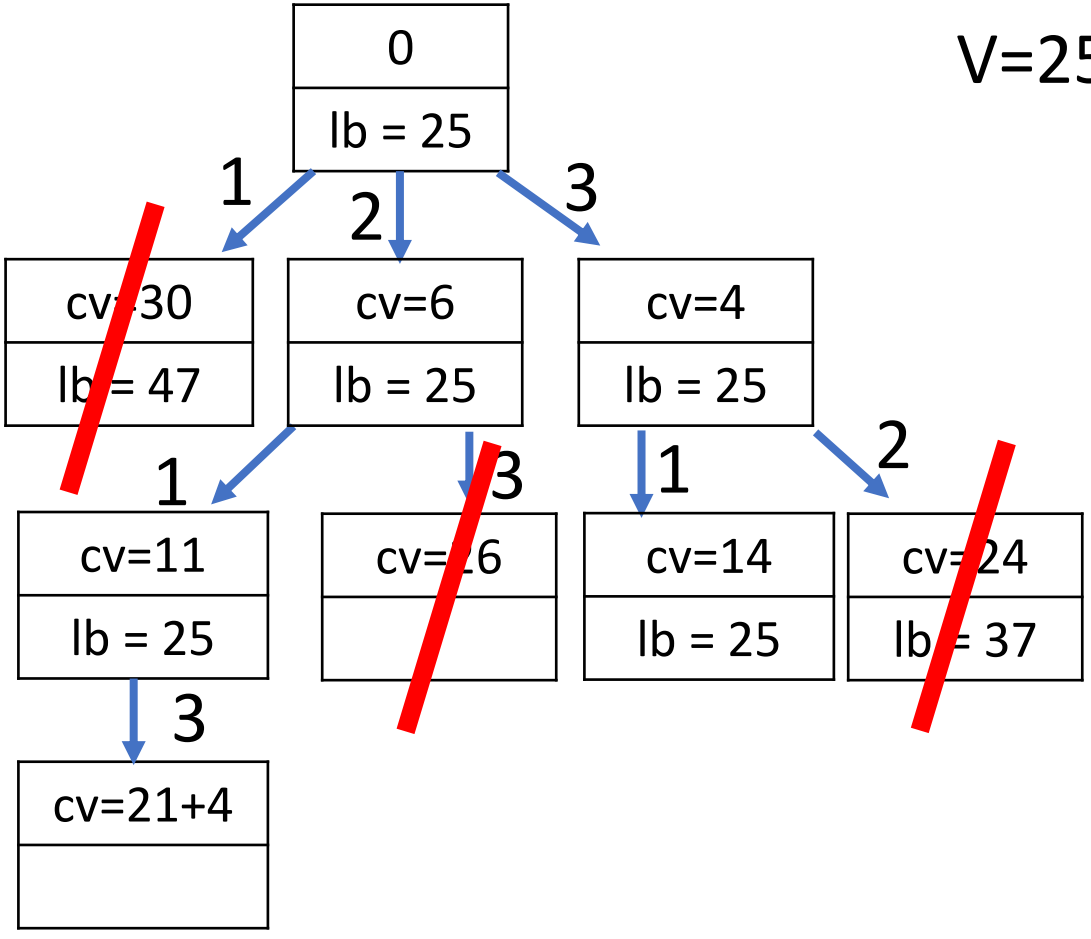
lb = 14+[(6+5)+(5+6)]/2 = 14+11=25



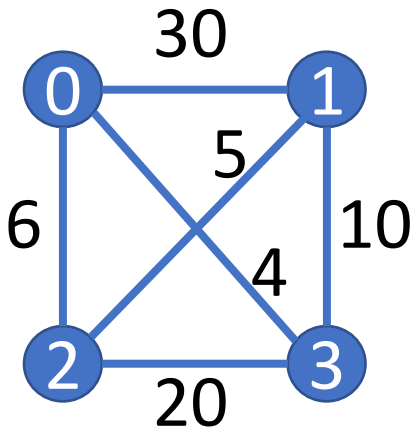
V=25

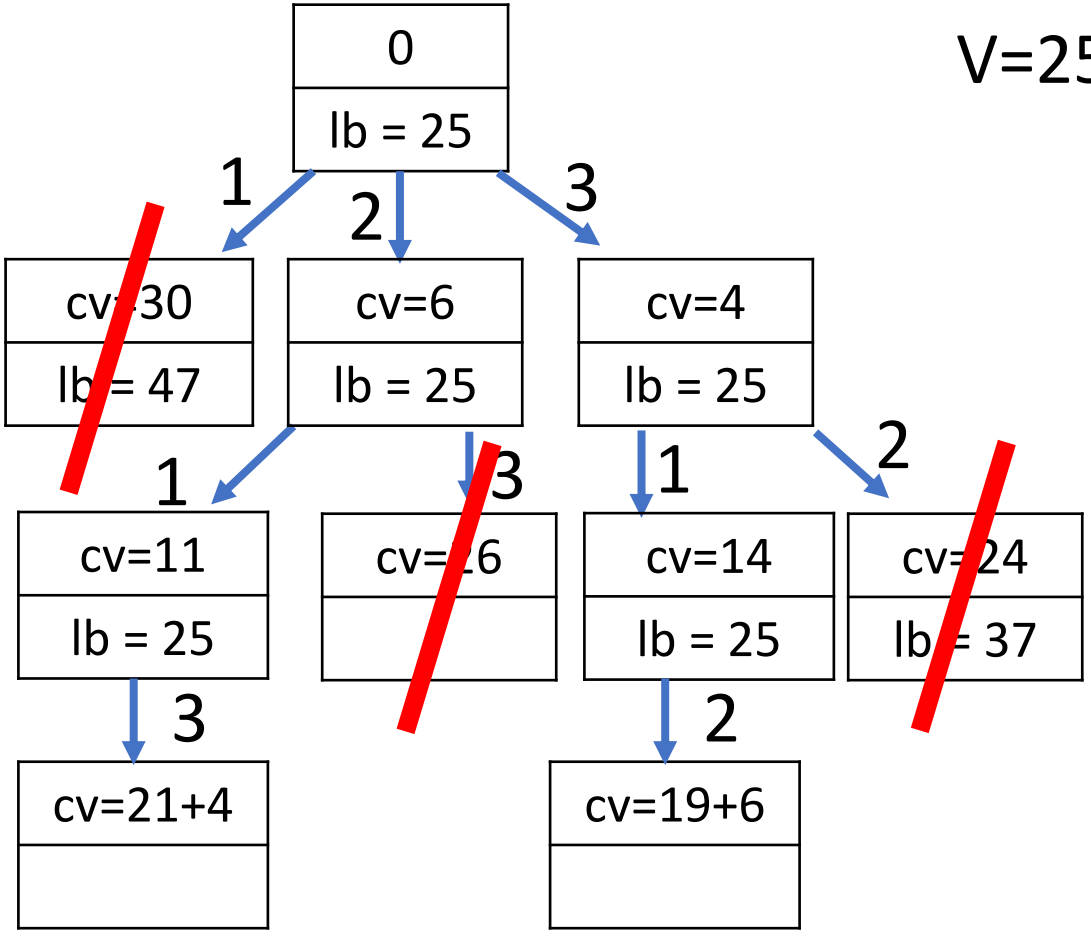


lb = 24+[(6+5)+(5+10)]/2 = 24+13=37

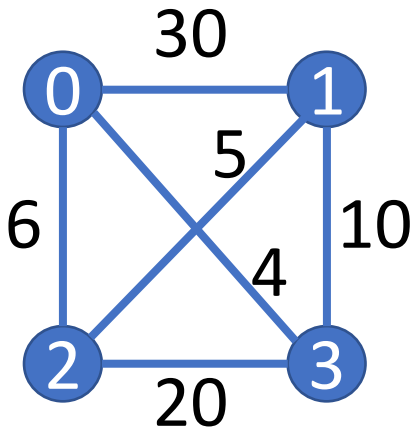

0
lb = 25
1
2
3
cv=30
lb = 47
cv=6
lb = 25
cv=4
lb = 25
1
3
1
2
cv=11
lb = 25
cv=26
cv=14
lb = 25
cv=24
lb = 37
3
cv=21+4

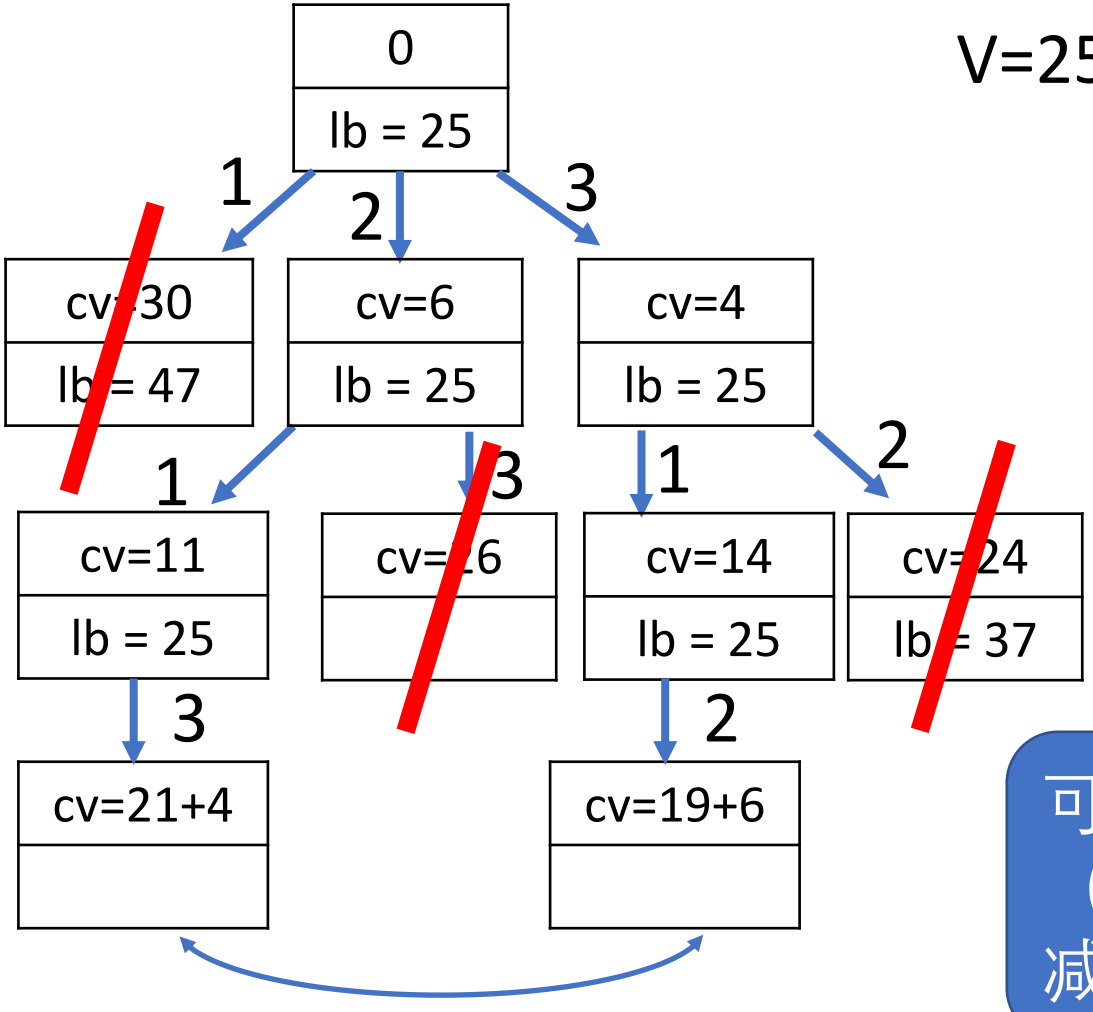
V=25


30
0
1
5
6
10
4
2
3
20

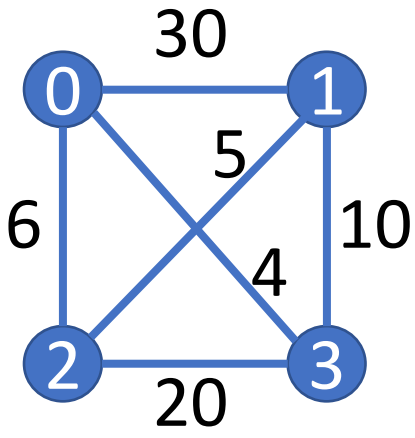

0
lb = 25
1
2
3
cv=30
lb = 47
cv=6
lb = 25
cv=4
lb = 25
1
3
1
2
cv=11
lb = 25
cv=26
cv=14
lb = 25
cv=24
lb = 37
3
2
cv=21+4
cv=19+6

V=25


30
0
1
5
6
10
4
2
3
20

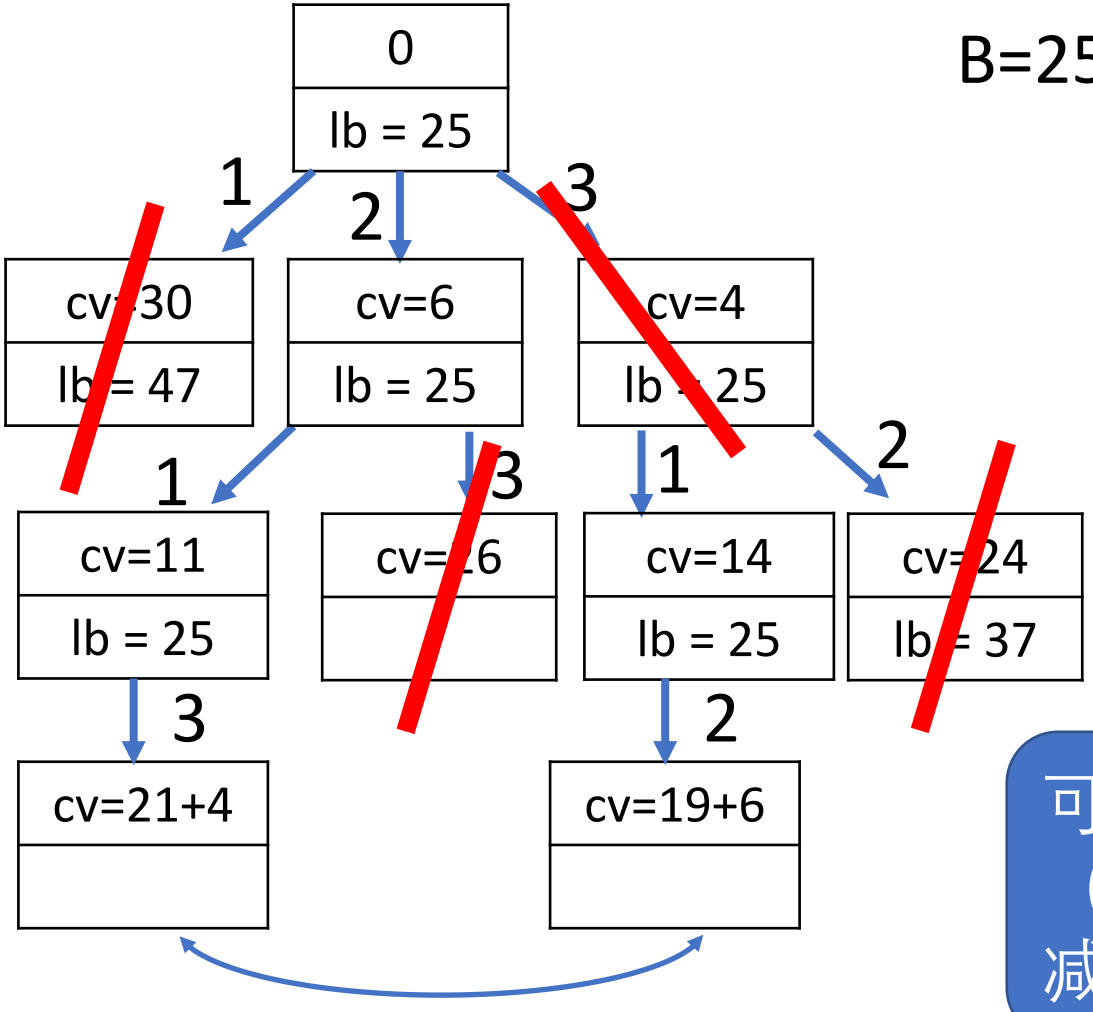


V=25

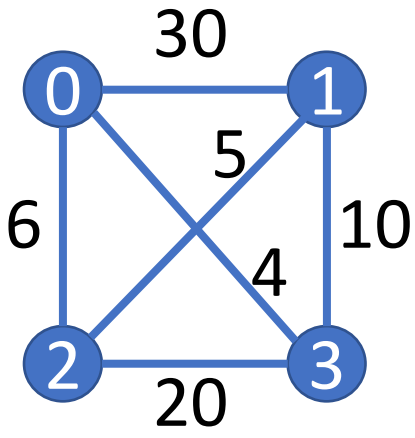


可以通过规定某2个顶点（1，3）的先后次序，减少一半搜索

这是同一条路径的正反方向



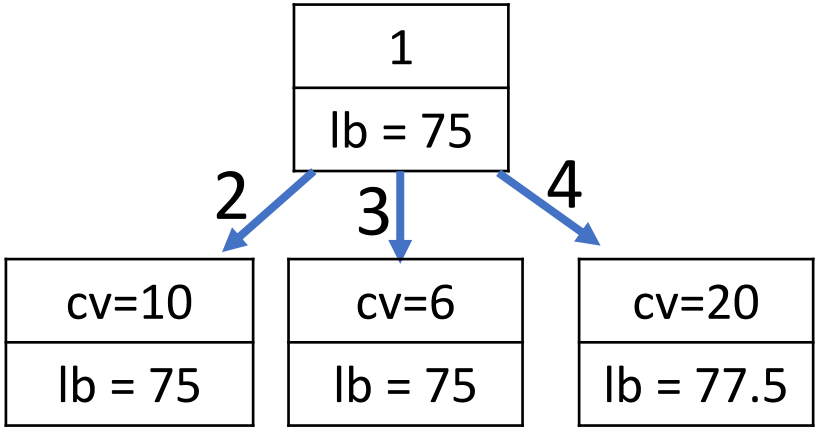
B=25

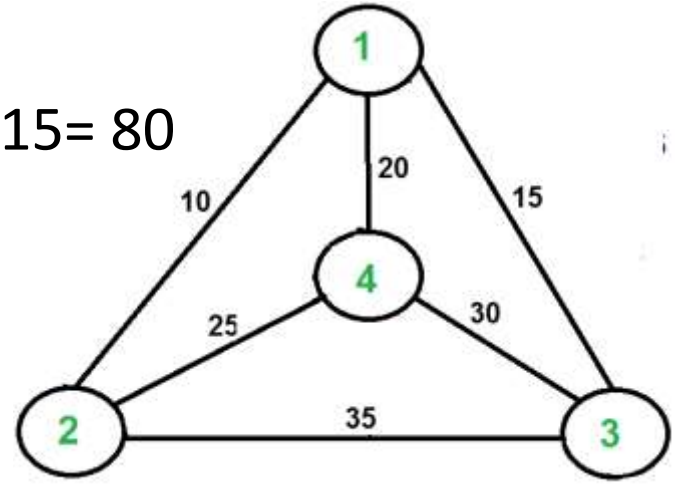


可以通过规定某2个顶点（1，3）的先后次序，减少一半搜索

这是同一条路径的正反方向

贪婪法：1-2-4-3-1， 价值 V = 10+25+30+15= 80





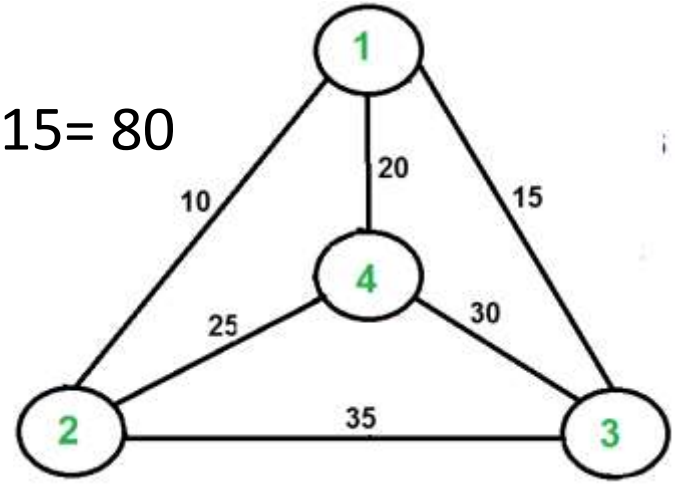
lb(1): [(10+15)+(10+25)+(30+15)+(20+25)]/2 = 150/2=75

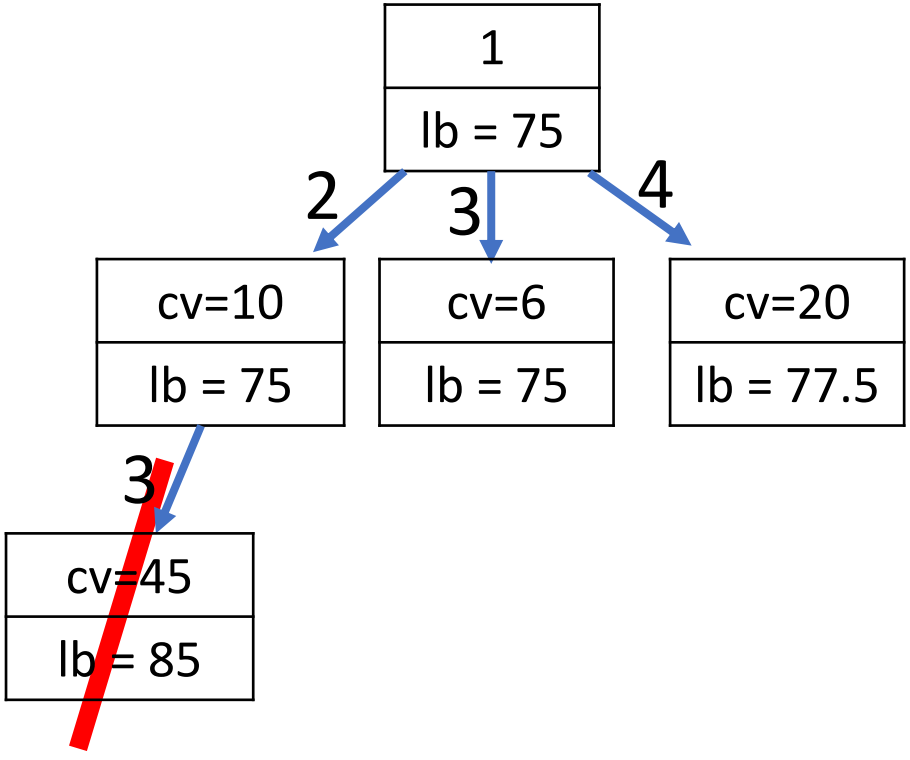
lb(1-2): 10+[(15+25)+(30+15)+(20+25)]/2 = 75

lb(1-3): 15+[(10+30)+(10+25)+(20+25)]/2 = 75

lb(1-4): 20+[(10+25)+(10+25)+(15+30)]/2 =77.5

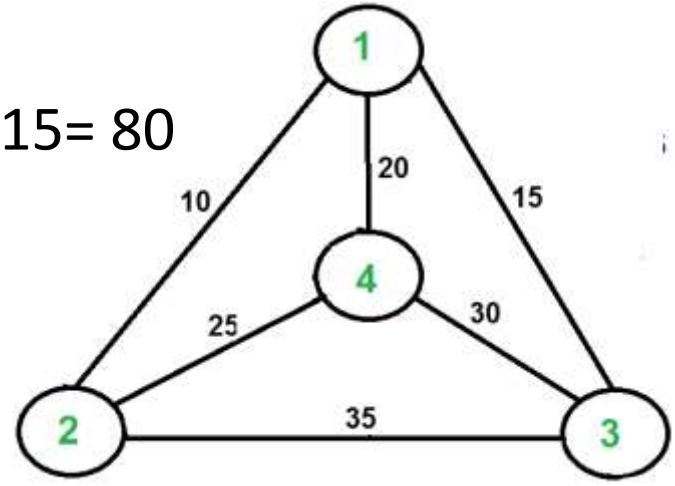
贪婪法：1-2-4-3-1， 价值 V = 10+25+30+15= 80

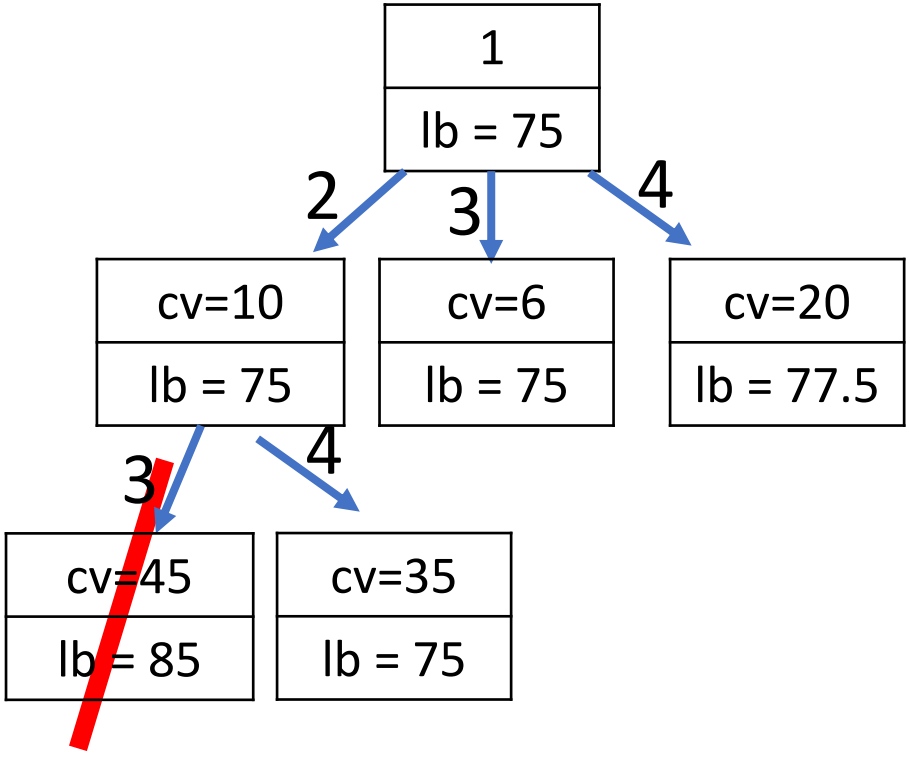




lb(1-2-3): 45+[(15+15)+(20+30)]/2 =45+40=85

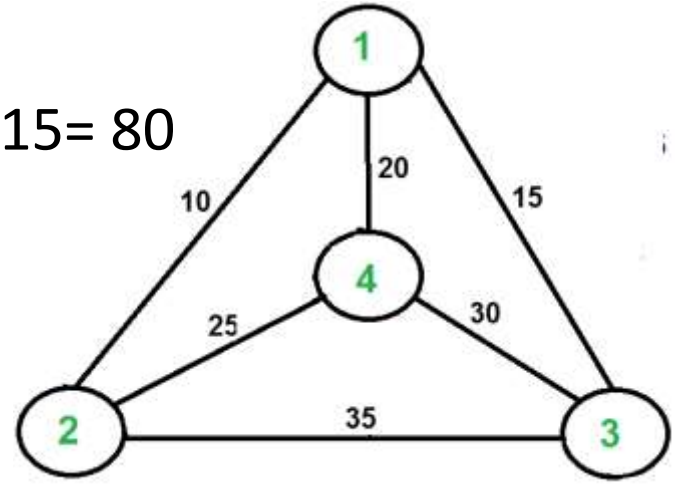
贪婪法：1-2-4-3-1， 价值 V = 10+25+30+15= 80

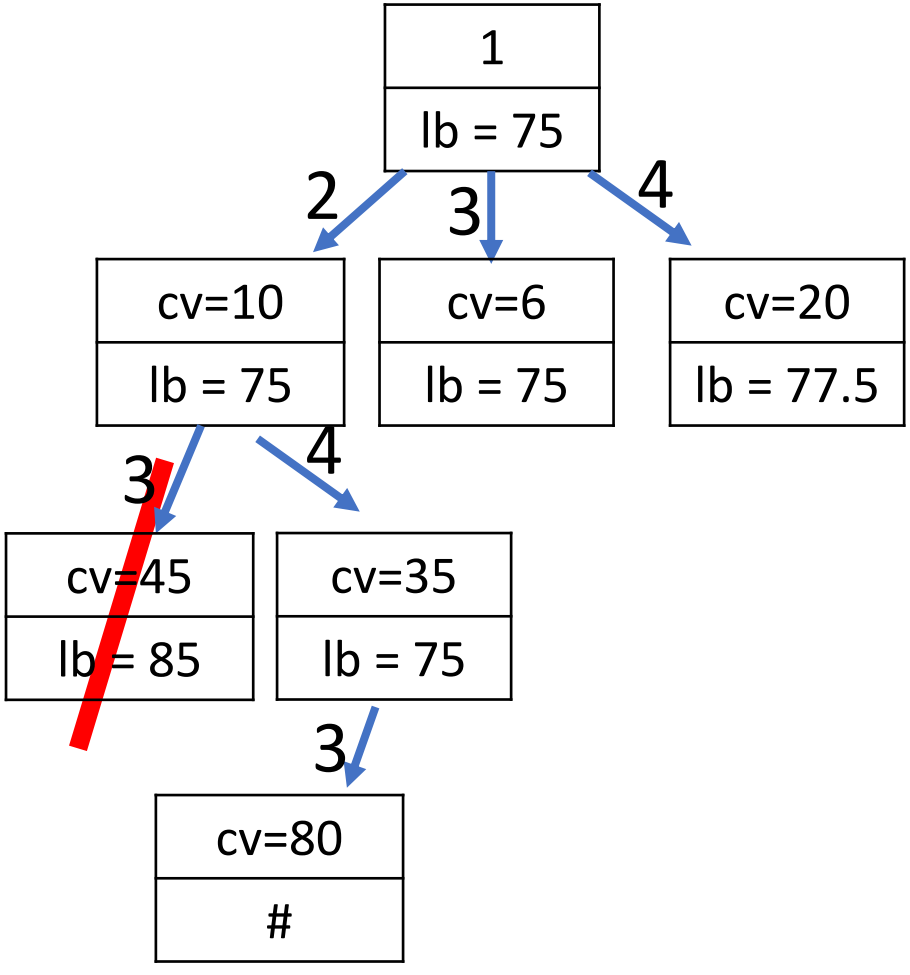




lb(1-2-4): 35+[(15+20)+(15+30)]/2 =35+40=75

贪婪法：1-2-4-3-1， 价值 V = 10+25+30+15= 80





分支限界：深度优先搜索

```
TSP_BB(G, S[], k, minCost, minPath) :
    n = S.size();
    if k==n:
        cost = pathCost(G, S);
        if cost<minCost:
            minCost = cost; minPath = S;
        return
    lb = bound(G, S, k);
    if lb > minCost  return
    for i= k to n:
         std::swap(S[i], S[k]);
         TSP_BB(G, S, k + 1, minCost, minPath);
         std::swap(S[i], S[k]);
```

```
bound(G, S, k){
        T lb = 0;
        if (k==0) {
                for (int i = 0; i < G.size(); i++) {
                        min, second= min_second(G[i]);
                        lb += (min+ second);
                }
                return lb;
        }
```

```
for  i = 0 to k-1:
        lb+=G[S[i]][S[i + 1]];
}
T lb2 = 0;
min, second = min_second(G[S[0]]);
if (min== G[S[0]][S[1]])
        lb2 += second;
else
        lb2 += min;

min, second = min_second(G[S[k]]);
if (min== G[S[k-1]][S[k]])
        lb2 += second;
else
        lb2 += min;
```

```
        for i = k + 1 to n:
                auto p = min_second(G[i]);
                lb2 += (p.first + p.second);
        }
        return lb+lb2/2;
}
```

总结：分支限界的搜索过程

- 深度优先搜索
- 基于优先队列的深度和广度优先结合的搜索
- 基于启发解（如贪婪解）的最优值的广度优先搜索。

总结：限界

对于最小值问题:

- 1）可行解的价值可用于更新当前最优值，这个当前最优值作为后续探索的上界B。
- 2）每个状态可以估算其状态树下所有目标可行解的最小值，作为状态的下界b，即该状态的可行解的价值不会低于这个下界。
- 3) 如果一个状态的下界b超过了当前最优价值(上界)B，则停止探索该状态树。

总结：限界

对于最大值问题:

- 1）可行解的价值可用于更新当前最优值，这个当前最优值作为后续探索的下界B。
- 2）每个状态可以估算其状态树下所有目标可行解的最大值，作为状态的上界b，即该状态的可行解的价值不会高于这个上界。
- 3) 如果一个状态的上界b小于当前最优价值(下界)B，则停止探索该状态树。

关注我

https://hwdong-net.github.io

Youtube频道:hwdong