第3章: 搜索

Search

李磊

第三章 搜索

- **搜索是人工智能中的一个基本问题，并与推理密切相关。搜索策略的优劣，将直接影响到智能系统的性能与推理效率。**
- **搜索是问题求解的一种基本方法 。**

- 本章内容
- 3.1 搜索的基本概念
- 3.2 问题的状态空间表示
- 3.3 一般图搜索算法
- 3.4 状态空间盲目搜索
- 3.5 状态空间的启发式搜索
- 3.6 问题归约表示法
- 3.7 与或树的盲目搜索
- 3.8 与或树启发式搜索

3.1 搜索的基本概念

- **1.什么是搜索**
  - **根据问题的实际情况不断寻找可利用的知识,构造出一条代价较少的推理路线,使问题得到圆满解决的过程称为搜索**
  - **包括两个方面:**
    - **--- 找到从初始事实到问题最终答案的一条推理路径**
    - **--- 找到的这条路径在时间和空间上复杂度最小**

3.1 搜索的基本概念

- **2.搜索的分类**
  - **按是否使用启发信息**
    - **(1) 盲目搜索(Uninformed search)**
      - **盲目搜索按预定的控制策略进行搜索，搜索过程中获得的中间信息不用来改变搜索策略。搜索总是按预定的路线进行，不考虑问题本身的特性，这种搜索有盲目性，效率不高，不利于求解复杂问题。**
      - **即，不利用领域知识来帮助搜索。**

3.1 搜索的基本概念—cont.

(2) 启发式搜索(Heuristic search, Informed search)

- 启发式搜索中利用问题领域相关的信息作为启发信息，用来指导搜索朝着最有希望的方向前进，提高搜索效率并力图找到最优解。

☞ 启发式搜索需要利用问题领域相关的信息帮助搜索，但并不是对每一类问题都容易抽取出启发信息，所以在很多情况下仍然需要盲目搜索。

3.1 搜索的基本概念—cont.

- **按问题的表示方式**

(1) 状态空间搜索

- **用状态空间法来求解问题时所进行的搜索**

(2) 与或树搜索

- **用问题归约法来求解问题时所进行的搜索**

- **其它分类方法**
  - **对抗搜索，…**

3.1 搜索的基本概念

- **3.适用情况：**
  - **不良结构或非结构化问题；难以获得求解所需的全部信息；更没有现成的算法可供求解使用。**

- **4.问题的形式化表示**
  - **搜索时首先要将问题进行形式化表示，常用的形式化表示方法有状态空间法、与或树（问题归约法）表示法等。**

3.1 搜索的基本概念

- **5.搜索策略常用评价指标：**
  1. **完备性（Completeness）**
     如果问题有解，算法就能找到，称此搜索方法是完备的。
  2. **最优性（Optimality）**
     如果解存在，总能找到最优解。
  3. **时间复杂度（Time Complexity）**
  4. **空间复杂度（Space Complexity）**

3.2 问题的状态空间表示

- **1.状态(state)**
  - **事物是运动、变化的，为描述问题的运动、变化，定义一组变量描述问题的变化特征和属性。**
  - **形式化表示：**
    - $(s_1,s_2,..s_i,\ldots,s_n)$
  - **当对每一个分量都给以确定的值时，就得到了一个具体的状态。**
  - **例：“过河问题”中我们定义了3个状态变量来描述问题的状态(M,W,B)，而(3,3,1)，(0,0,0)，(2,2,1)等都是问题的具体状态。**

3.2 问题的状态空间表示

- **2.操作符(Operator)**
  - **也称为算符，它是把问题从一种状态变换为另一种状态的手段。**
  - **操作可以是一个机械步骤，一个运算，一条规则或一个过程。**
  - **操作可理解为状态集合上的一个函数，它描述了状态之间的关系。**
  - **例：“猴子香蕉”问题中我们定义的goto(u,v)，pushBox(u,v)等。**

3.2 问题的状态空间表示

- **3.状态空间(State space)**
  - **用来描述一个问题的全部状态以及这些状态之间的相互关系。常用一个三元组表示为：**

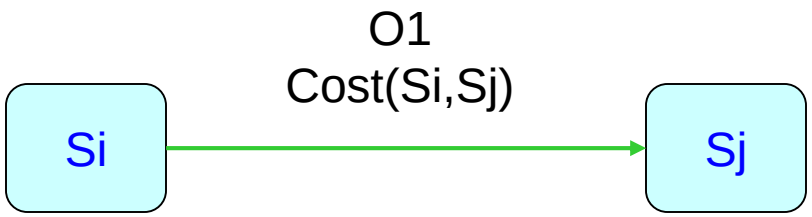
    (S, F, G)
- **其中：**
  - **S为问题的所有初始状态的集合；**
  - **F为操作（函数、规则等）的集合；**
  - **G为目标状态的集合。**

3.2 问题的状态空间表示

- **4.状态空间图**

  状态空间的有向图表示：

  - **结点（节点）：节点表示问题的状态**
  - **弧（有向边）：标记操作符；可能的路径代价。**
    - **例：下图Si,Sj为2个表示状态的节点；O1是导致状态变化的操作符；cost(Si,Sj)是从Si变化到Sj的代价（花费）。**

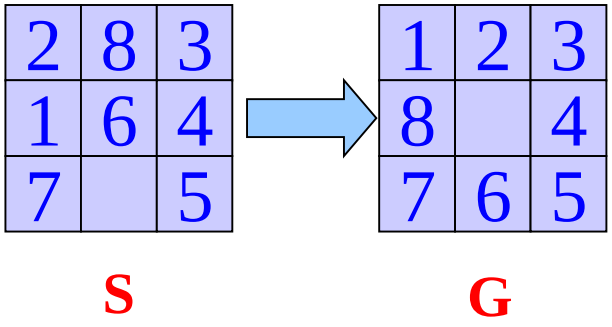


3.2 问题的状态空间表示

- **5.状态空间法求解问题的基本过程：**
  1. **首先为问题选择适当的“状态”及“操作”的形式化描述方法；**
  2. **然后从某个初始状态出发，每次使用一个满足前提条件的“操作”，且此操作产生了新的状态，递增地建立起操作序列，直到达到目标状态为止；**
  3. **此时，由初始状态到目标状态所使用的算符（操作符）序列就是该问题的一个解。**

3.2 问题的状态空间表示

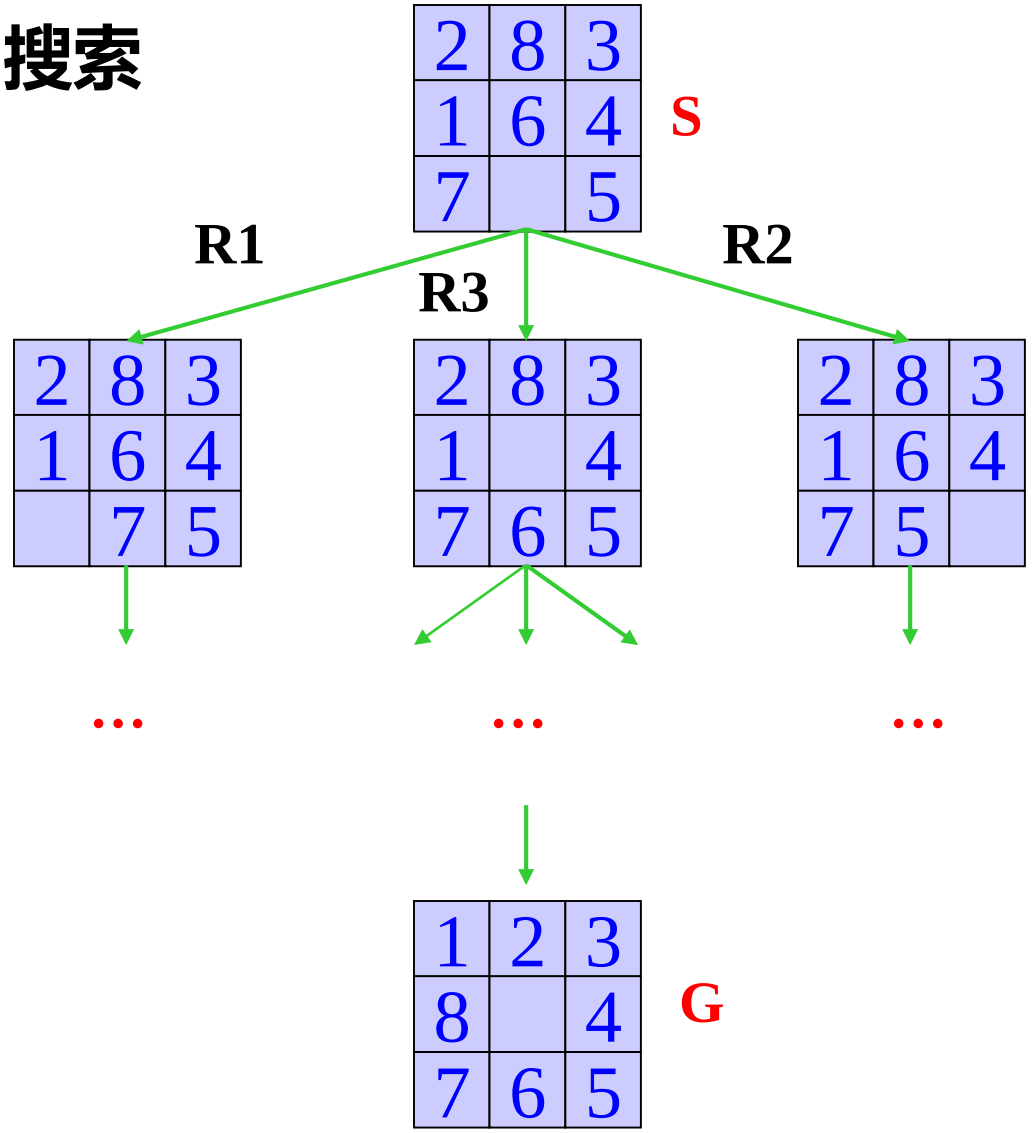
- **【例3.2.1】 8数码问题**
  - **解：**
  - **状态表示**
    - **用二维数组S[i,j]表示，**
    - **其中0<=i,j<=2；空格用0表示。**
    - **则：S[i,j]∈{0,1,2,3,4,5,6,7,8}**
    - **用i_0和j_0表示空格的下标。**



  - **定义操作符（产生式规则）：**
    - **左移 R_1 if(j_0≥1) then {S[i_0,j_0]=S[i_0,j_0-1];S[i_0,j_0-1]=0;}**
    - **右移 R_2 if(j_0≤1) then {S[i_0,j_0]=S[i_0,j_0+1];S[i_0,j_0+1]=0;}**
    - **上移 R_3 if(i_0≥1) then {S[i_0,j_0]=S[i_0-1,j_0];S[i_0-1,j_0]=0;}**
    - **下移 R_4 if(i_0≤1) then {S[i_0,j_0]=S[i_0+1,j_0];S[i_0+1,j_0]=0;}**

3.2 问题的状态空间表示

- **状态空间搜索**



3.2 问题的状态空间表示

- **【额外话题】重排九宫问题，对任意给定初始状态，可达下图所示两个目标之一，不可互换。**
  - **目标一：下图G**
  - **目标二：下图G1或G2**

目标状态一

1	2	3
8		4
7	6	5

G

目标状态二

1	2	3
4	5	6
7	8	

G1

	1	2
3	4	5
6	7	8

G2

3.2 问题的状态空间表示

- **两个目标的判定规则：**
  - **对其中任一数字n（1-8），第一行开始，每行从左往右顺序，计算（计数）n之前（或之后）所有小于n的数字的个数，设为f(n)，对所有f(n)求和。若求和结果为奇数，可达目标一；求和结果为偶数，可达目标二。空格不参与计数。**

2	8	3
1	6	4
7		5

N=f(2)+f(8)+f(3)+f(1)+f(6)+f(4)+f(7)+f(5)=0+1+1+0+3+3+5+4=17 可达目标一

2	5	3
8		4
7	6	1

N=f(2)+f(5)+f(3)+f(8)+f(4)+f(7)+f(6)+f(1)=0+1+1+3+2+4+4+0=15 可达目标一

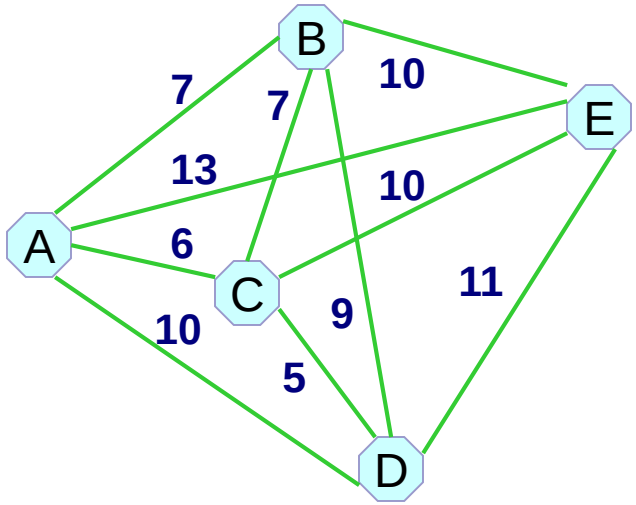
2	5	4
1	8	3
7	6	

N=f(2)+f(5)+f(4)+f(1)+f(8)+f(3)+f(7)+f(6)=0+1+1+0+4+2+5+5=18 可达目标二

3.2 问题的状态空间表示

- **【例3.2.2】 TSP问题**
  - **一个推销员从A城市出发，旅行经过B、C、D、E四个城市，然后回到A市， 其他城市只能经过一次，寻找一条花费最少的旅行路径。**
  - **解：**
  - **状态表示：**
    - **一维数组s[i]，0=<i<=5**
    - **初始状态：(A,,,,,)**
    - **目标状态：(A,x,y,z,u,A)**

      其中，x,y,z,u∈{B,C,D,E}

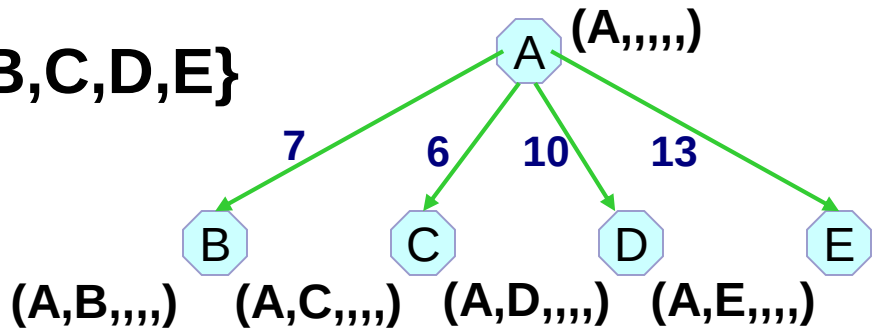


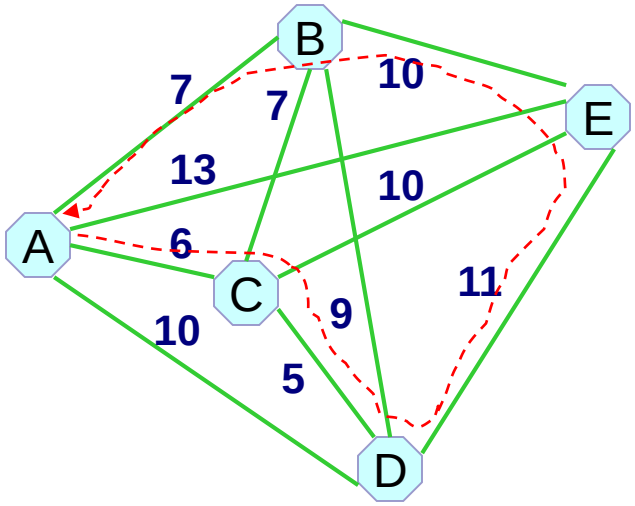
3.2 问题的状态空间表示

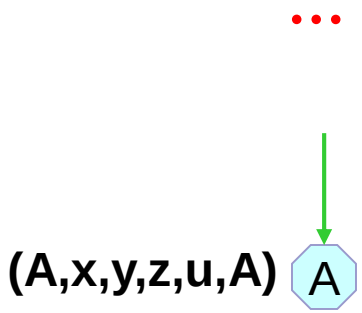
- **定义操作符**
  - **Goto(u,v)，　u,v∈{A,B,C,D,E}**
- **状态控件搜索**
  - **例：(A,C,D,E,B,A)**
  - **Cost(A,A)**

=6+5+11+10+7

=39

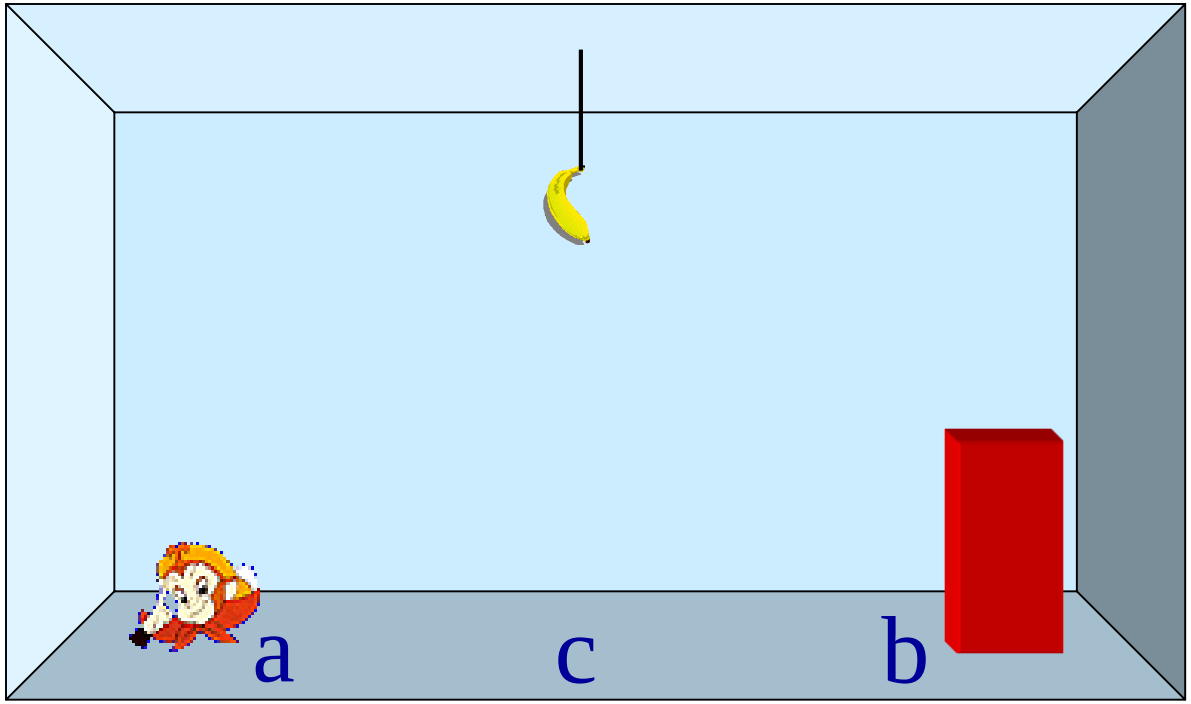






3.2 问题的状态空间表示

- **例3.2.3 猴子香蕉问题**



3.2 问题的状态空间表示

- **解：**
  - **状态表示**
    - **m -- 表示猴子的水平位置；**
    - **O -- 布尔变量，标记猴子是否在箱子上。O = 1 猴子在箱子上,O = 0 不在箱子上；**
    - **b -- 箱子的水平位置；**
    - **G -- 布尔变量，标记猴子是否已经拿到香蕉。G = 1 猴子拿到香蕉。**
    - **状态用4元组描述：（m, O, b, G）**

      初始状态：（a, 0, b, 0）

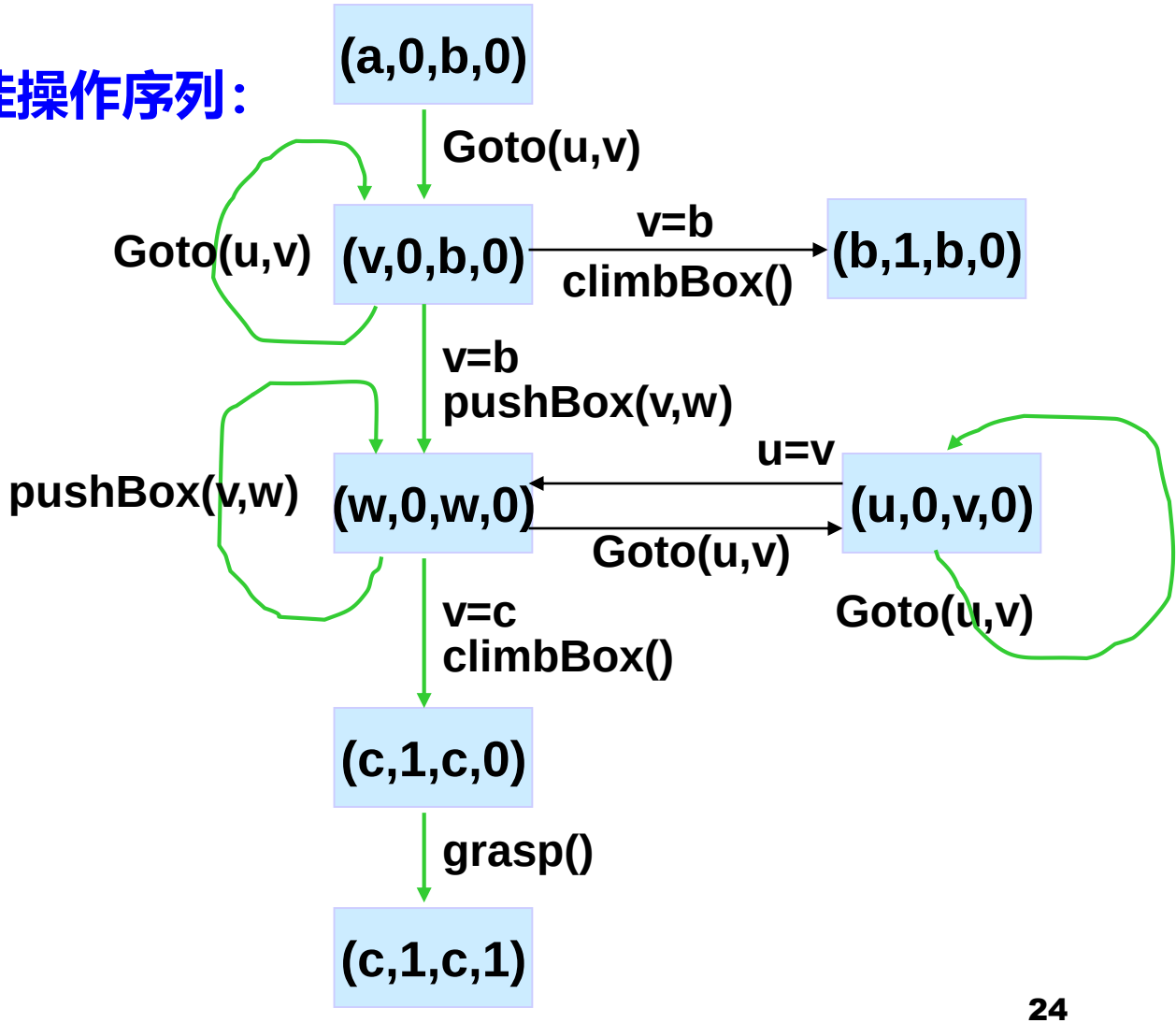
      目标状态：（c, 1, c, 1）

3.2 问题的状态空间表示

- **定义操作符**
- **Goto(u, v)：猴子从u处走到v处**
  - **条件：(m=u AND O=0)**
  - **状态变化：(u,0,b,G) ⟶ (v,0,b,G)**
- **Pushbox(v, w)：猴子推着箱子从v处移到w处**
  - **条件：(m=v AND b=v AND O=0)**
  - **状态变化：(v,0,v,G) ⟶ (w,0,w,G)**
- **Climbbox( )：猴子爬上箱子**
  - **条件：(m=b AND O=0)**
  - **状态变化：(v,0,v,G) ⟶ (v,1,v,G)**
- **Grasp( )：猴子摘取香蕉**
  - **条件：(m=c AND b=c AND O=1 AND G=0)**
  - **状态变化：(c,1,c,0) ⟶ (c,1,c,1)**

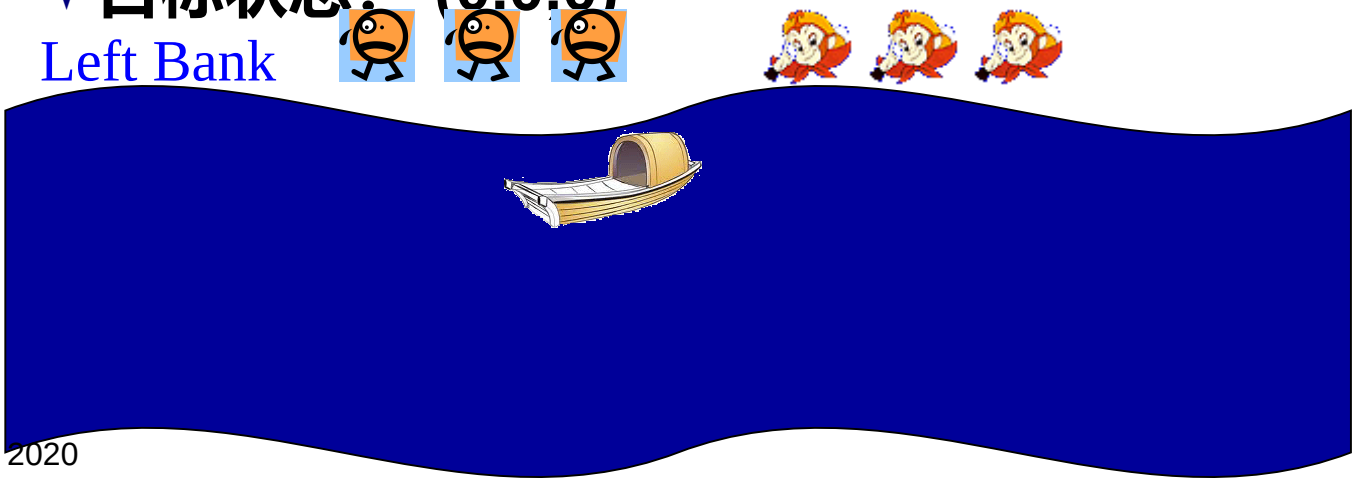
3.2 问题的状态空间表示

- **状态空间搜索**
  - **猴子拿到香蕉的最佳操作序列：**
  - goto(a,b),
  - pushBox(b,c),
  - climbBox(),
  - grasp()



3.2 问题的状态空间表示

- **例3.2.4 过河问题**
- **http://www.973.com/p45383**
  - **解：**
  - **状态表示**
    - **(m,w,B)**
    - **初始状态： (3,3,1)**
    - **目标状态： (0,0,0)**



3.2 问题的状态空间表示

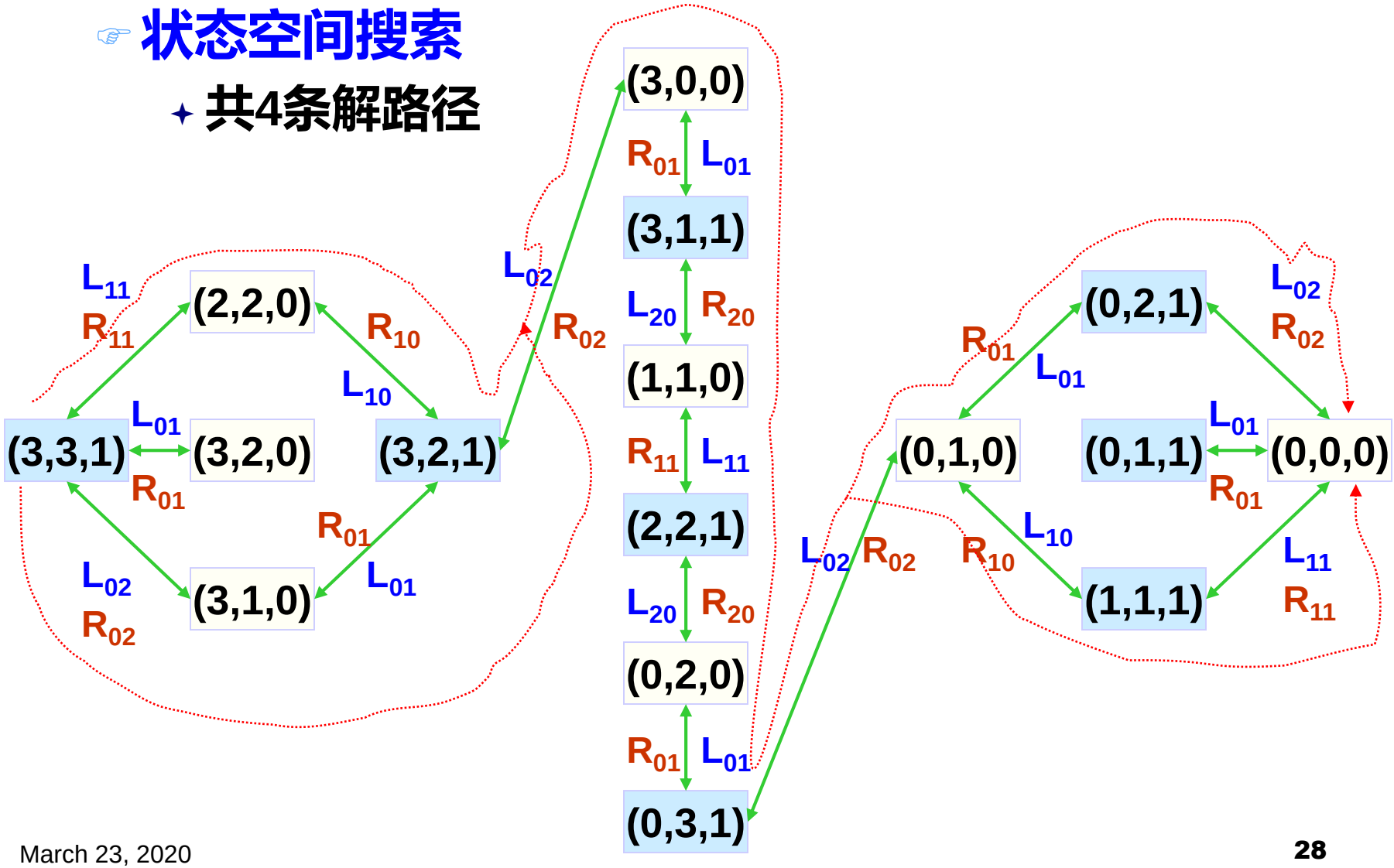
- **操作符（产生式规则）**
- **左岸往右岸运载产生式**
  - L_{10} if(M,W,1) then (M-1,W,0) (左往右运1传教士)
  - L_{01} if(M,W,1) then (M,W-1,0) (左往右运1野人)
  - L_{11} if(M,W,1) then (M-1,W-1,0) (左往右运1传教士和1野人)
  - L_{20} if(M,W,1) then (M-2,W,0) (左往右运2传教士)
  - L_{02} if(M,W,1) then (M,W-2,0) (左往右运2野人)

右岸往左岸运载产生式

- R_{10} if(M,W,0) then (M+1,W,1) (右往左运1传教士)
- R_{01} if(M,W,0) then (M,W+1,1) (右往左运1野人)
- R_{11} if(M,W,0) then (M+1,W+1,1) (右往左运1传教士和1野人)
- R_{20} if(M,W,0) then (M+2,W,1) (右往左运2传教士)
- R_{02} if(M,W,0) then (M,W+2,1) (右往左运2野人)

3.2 问题的状态空间表示

- **状态空间搜索**
  - **共4条解路径**



3.3 一般图搜索算法

- **一般图搜索（状态空间搜索）的基本思想**
  - **问题状态用图数据结构的结点表示；**
  - **从初始状态（结点）开始，对选定的结点选择满足条件的操作符，操作符作用后产生新的结点（状态）；**
  - **检查新产生的子结点中是否有目标结点：有则找到了问题的解；**
  - **否则重复上述过程直至产生目标结点，或全部结点处理完无解。**

3.3 状态空间盲目搜索

- **状态空间搜索的基本思想**
  - 先把问题的初始状态作为当前扩展节点对其进行扩展，生成一组子节点，然后检查问题的目标状态是否出现在这些子节点中。若出现，则搜索成功，找到了问题的解；若没出现，则再按照某种搜索策略从已生成的子节点中选择一个节点作为当前扩展节点。重复上述过程，直到目标状态出现在子节点中或者没有可供操作的节点为止。所谓对一个节点进行“扩展”是指对该节点用某个可用操作进行作用，生成该节点的一组子节点。

3.3.1 基本概念

- **1.扩展(Expanding)节点**
  - **对某一节点（状态），选择合适的操作符作用在节点上，使产生后继状态（子节点）的操作。**
  - **类似数据结构中的寻找邻接点，但这里的邻接点是选择操作后产生的。**
- **2. Open和Closed表**
  - **这两个表用来存放节点，Open表存放未扩展节点，Closed表存放已扩展节点和待扩展结点。两个表的结构可以相同，大致如下表：**
  - **可根据需要扩展表的结构，比如加入代价字段等。**
  - **在数据结构中，常用数组visited[ ]标记结点是否已经访问。**

Open和Closed表结构示例

编号	状态节点	父节点

3.3.2 图搜索一般过程

- **图搜索（graph search）一般过程：**
  1. **建立一个只含初始状态节点S的搜索图G，建立一个OPEN表，用来存放未扩展节点，将S放入OPEN表中；**
  2. **建立一个CLOSED表，用来存放已扩展和待扩展节点，初始为空；**
  3. **LOOP：若OPEN为空，则失败、退出；**
  4. **选择OPEN表中的第一个节点，将其移到CLOSED表中，称此节点为n节点；**
  5. **若n为目标节点，则成功、退出；**

     此解是追踪图G沿着指针从n到S这条路径得到的。

6. 扩展n节点，生成n的后继节点集合M=M1+M2+M3，其中n的后继结点分为3中情况。设M1表示图G中新结点（最新生成的）；M2在图G中已经存在，处于OPEN表中；M3在图G中已经存在，且已经在CLOSED表中：

（请对照数据结构中，搜索邻接顶点的情况）

（1）对M1型结点，加入到图G中，并放入OPEN表中，设置一个指向父节点n的指针；（DS中的未访问邻接点）

（2）对M2型结点，已经在OPEN中，确定是否需要修改父节点指针；（DS中已访问邻接点，但这个顶点的邻接点未搜索）

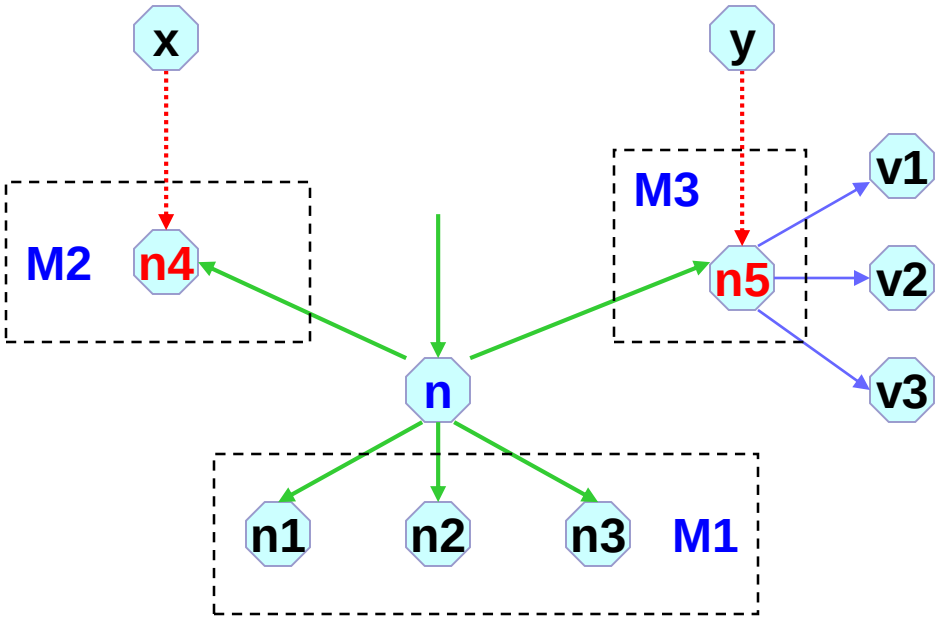
（3）对M3型结点，已经在CLOSED表中，确定是否修改其父结点指针；是否修改其后裔节点的指针；（DS中已访问邻接点，且这个顶点的邻接点都已经搜索过）

- 按某一控制策略，重新排序OPEN 表；
- goto LOOP

3.3.3 图搜索的几点说明

- **1.这是状态空间的一般搜索过程，具有通用性，后面讨论的各种搜索策略都是此过程的一个特例。不同特例的区别在于OPEN表的排序方式不同。**
- **2.扩展节点n，生成的子节点，分为三种情况：M1是图G中没有的新节点；M2是已在图G中，但没有被扩展，即在OPEN表中；M3是已在图G中，且已经被扩展生成了子节点，即已在CLOSED表中。以上情况如下图所示。**

- M1为图G中最新产生的结点；
- M2已在OPEN表中，说明有不同父节点产生，需确定谁作为其父节点；
  - 如图中的n4要确定父节点是n还是x。
- M3已在Closed表中，由不同父节点产生，需确定谁为其父节点；此外要确定其已经生成的子节点是否有效。
  - 如图中的n5要确定其父节点是n还是y；此外要确定其已经生成的子节点v1、v2和v3是否有效。



3.4 状态空间盲目搜索

- **盲目搜索按预定的控制策略进行搜索，搜索过程中获得的中间信息不用来改变搜索策略。搜索总是按预定的路线进行，不考虑问题本身的特性，这种搜索有盲目性，效率不高，不利于求解复杂问题。**

3.4.1广度优先搜索(BFS-Breadth First Search)

- **由近及远逐层访问图中顶点（典型的层次遍历）。**
- **1.节点深度：**
  - **起始节点S（根节点，图中选定起始搜索顶点）深度为0；其他节点等于父节点深度加1。**
- **2.基本思想**
  - **从初始节点S开始，依据到S的深度，逐层扩展节点并考察其是否目标节点。在第n层节点没有完全扩展之前，不对第n+1层节点进行扩展。**
  - **即：OPEN表排序策略为新产生的节点放到OPEN表的末端。**

3.BFS遍历搜索算法

从初始状态节点S出发广度优先搜索遍历图的算法bfs(S):

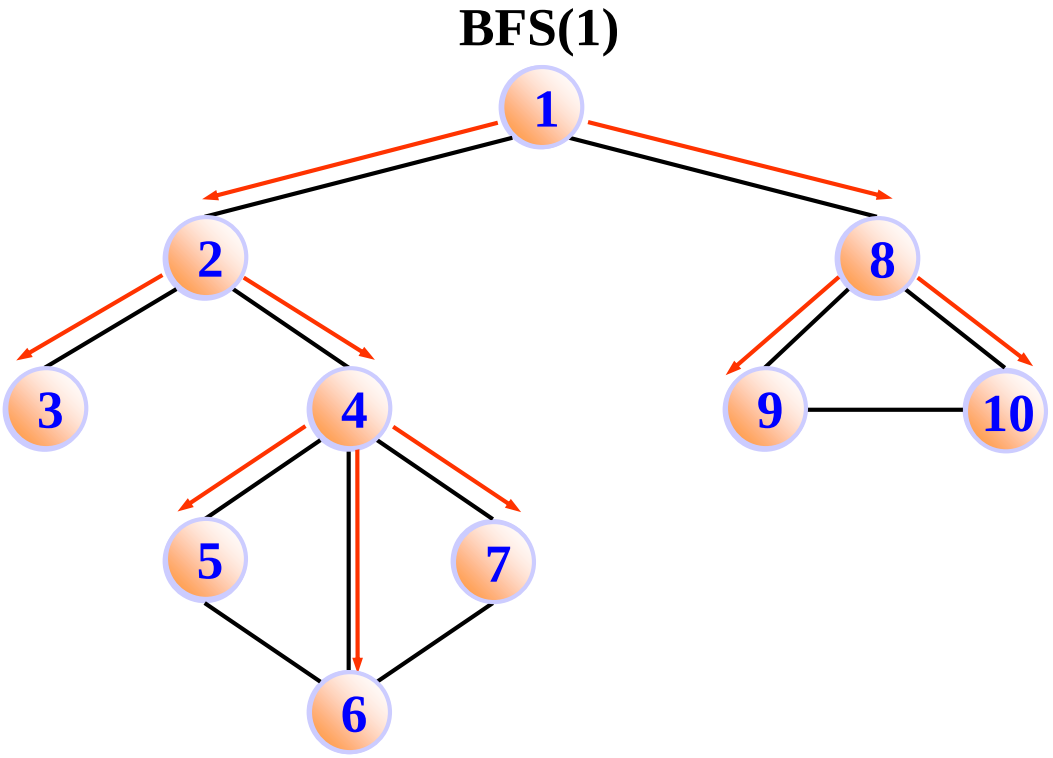
1) 访问S

2) 依次访问S的各邻接点

3) 设最近一层访问序列为v_{i1}，v_{i2}，…，v_{ik}，则依次访问v_{i1}，v_{i2}，…，v_{ik}的未被访问过的邻接点。

4) 重复（3），直到找不到未被访问的邻接点为止。

【例1】对下图进行广度优先遍历

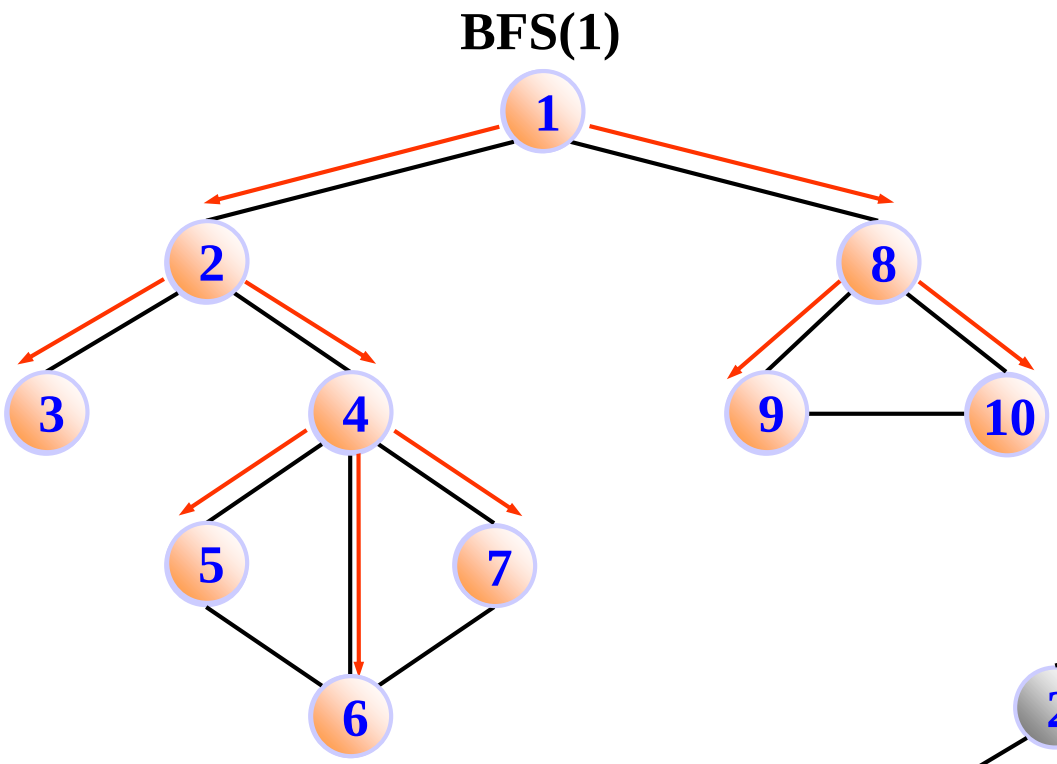


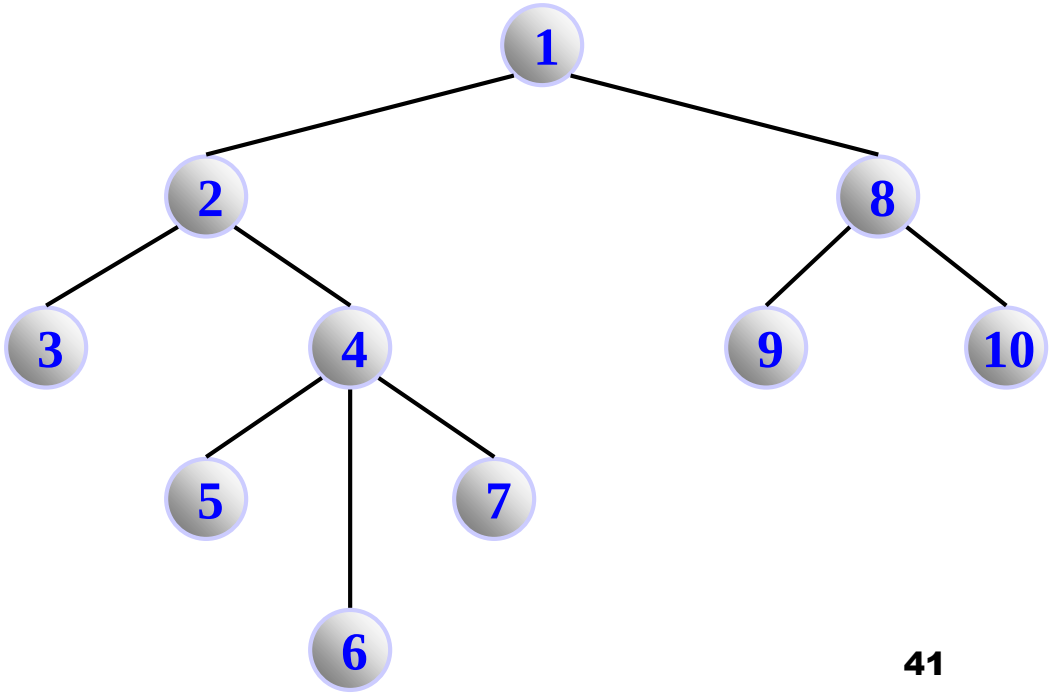
队列Q

1	2	8	3	4	9	10	5	6	7

顶点访问序列：1,2,8,3,4,9,10,5,6,7

BFS(1)生成树





执行BFS(1)的生成树

【思考问题】

① BFS(1)访问顶点的次序是不是唯一的?

【答】不一定，本题中顶点1出队后，可能先找到邻接点8，然后才找到邻接点2，由算法具体实现确定。

② 如果从BFS(7)开始遍历结果如何?

③ 从图中任意指定的顶点开始遍历呢?

④ 树可以进行广度优先搜索遍历吗?

【答】可以，树也是图，只是相对简单。

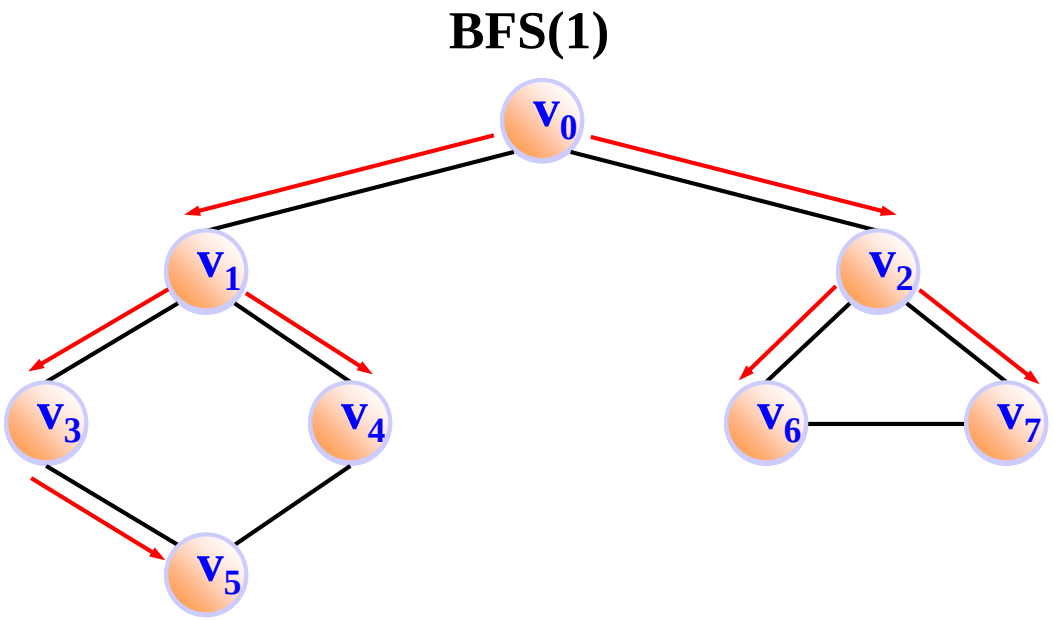
⑤ 森林怎样进行广度优先搜索遍历吗?

【答】一棵树接着一棵树遍历，直至遍历完森林中的每一棵树。

【例2】对下图进行广度优先遍历

- 顶点$v_0,v_1,v_2,v_3,v_4,v_5,v_6,v_7$的编号依次为：1,2,3,4,5,6,7,8

搜索：→

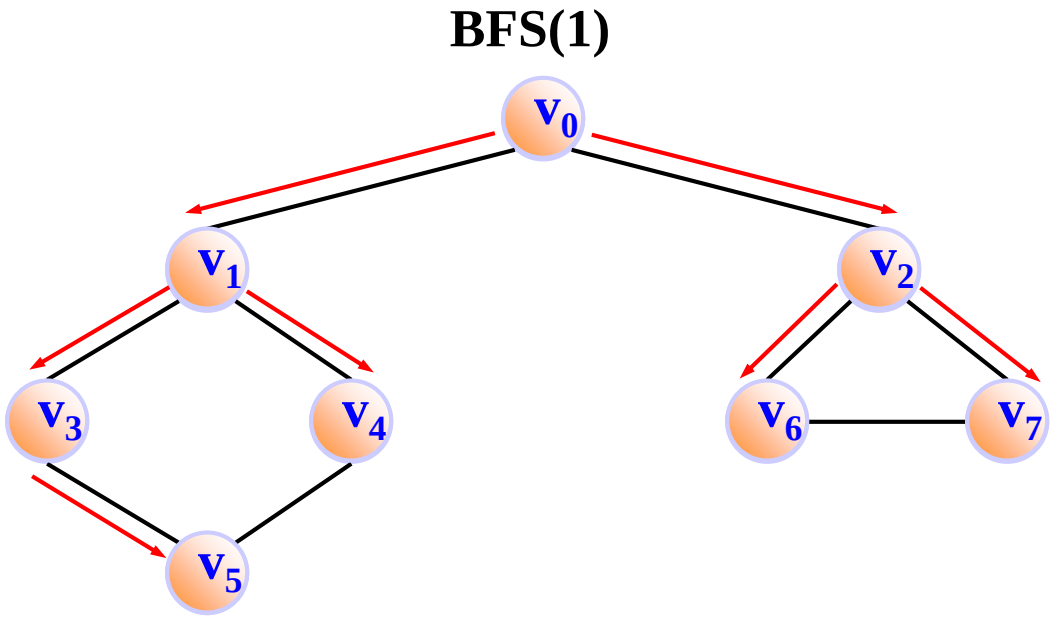


队列Q

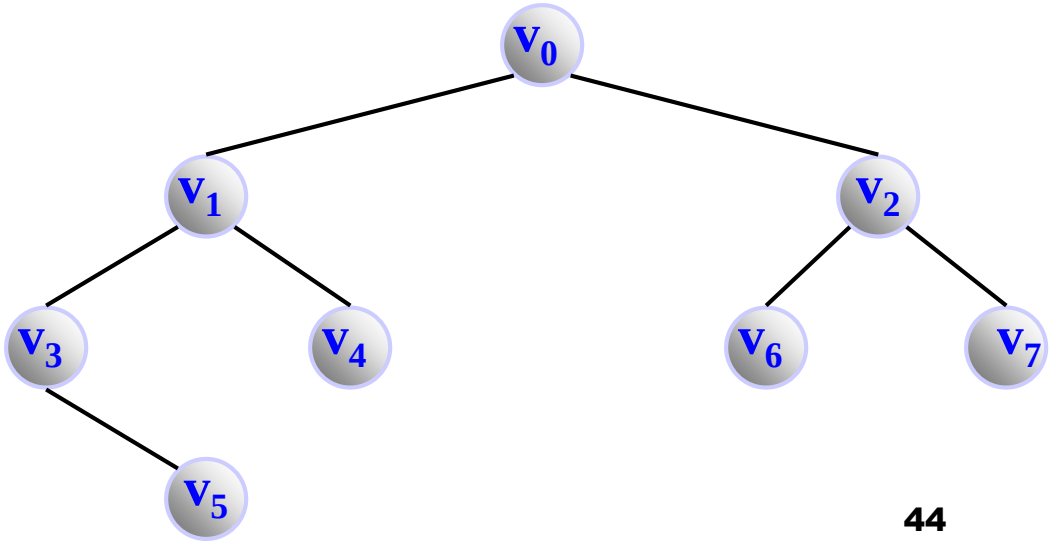
顶点访问序列：1, 2, 3, 4, 5, 7, 8, 6。

即：$v_0, v_1, v_2, v_3, v_4, v_6, v_7, v_5$

BFS(1)生成树



执行BFS(1)的生成树



【思考问题】

- 对一个图，从不同的顶点开始遍历，得到的生成树是否相同?

【答】不一定相同。

- **4.状态空间广度优先搜索**

  (1)把初始节点S0放入Open表中；

  (2)如果Open表为空，则问题无解，失败退出；

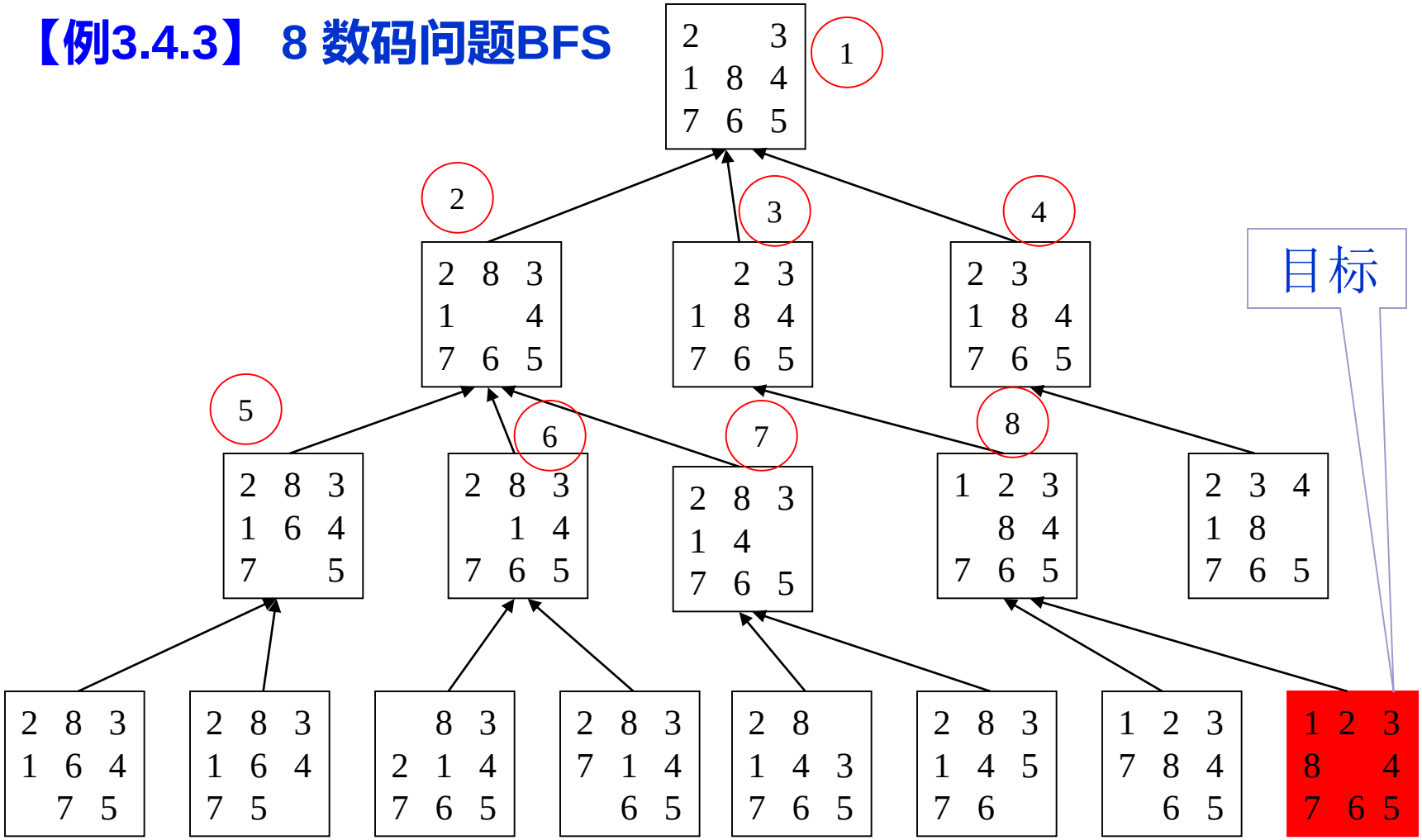
  (3)把Open表的第一个节点取出放入Closed表，并记该节点为n；

  (4)考察节点n是否为目标节点。若是，则得到问题的解，成功退出；

  (5)若节点n不可扩展，则转第(2)步；

  (6)扩展节点n，将其子节点放入Open表的尾部，并为每一个子节点设置指向父节点的指针，然后转第(2)步。

【例3.4.3】 8 数码问题BFS



- **5.广度优先搜索性能：**
  - ① **完备的**
  - ② **最优的—对搜索深度指标**
  - ③ **时间复杂度： O（a^b）**
    - **树的分枝因子（度）：树中最大的子节点数（按最坏情况考虑），设为a。**
    - **搜索深度：b**
  - ④ **空间复杂度：O（ a^b ）**

6.广度优先搜索特点

缺点

- 当目标节点距离初始节点较远时会产生许多无用的节点,搜索效率低;
- 空间是大问题(和时间相比)

优点

- 只要问题有解,则总可以得到解,而且是最短路径（深度）的解

- **应用实例：**
  - 搜索引擎的网络爬虫—网页超链接的广度优先搜索
  - 处理一个网页上的所有超链接后，再进入下一层页面处理。

3.4.2深度优先搜索(DFS-Depth First Search)

- **1.基本思想**
  - **从初始节点S开始，优先扩展最新产生的节点（最深的节点）。**
  - **即：OPEN表排序策略为新产生的节点放到OPEN表的前端，优先扩展。**

2.DFS遍历搜索算法

从初始状态顶点S出发深度优先遍历图的方法dfs (S):

1) 访问S——visit（S）；

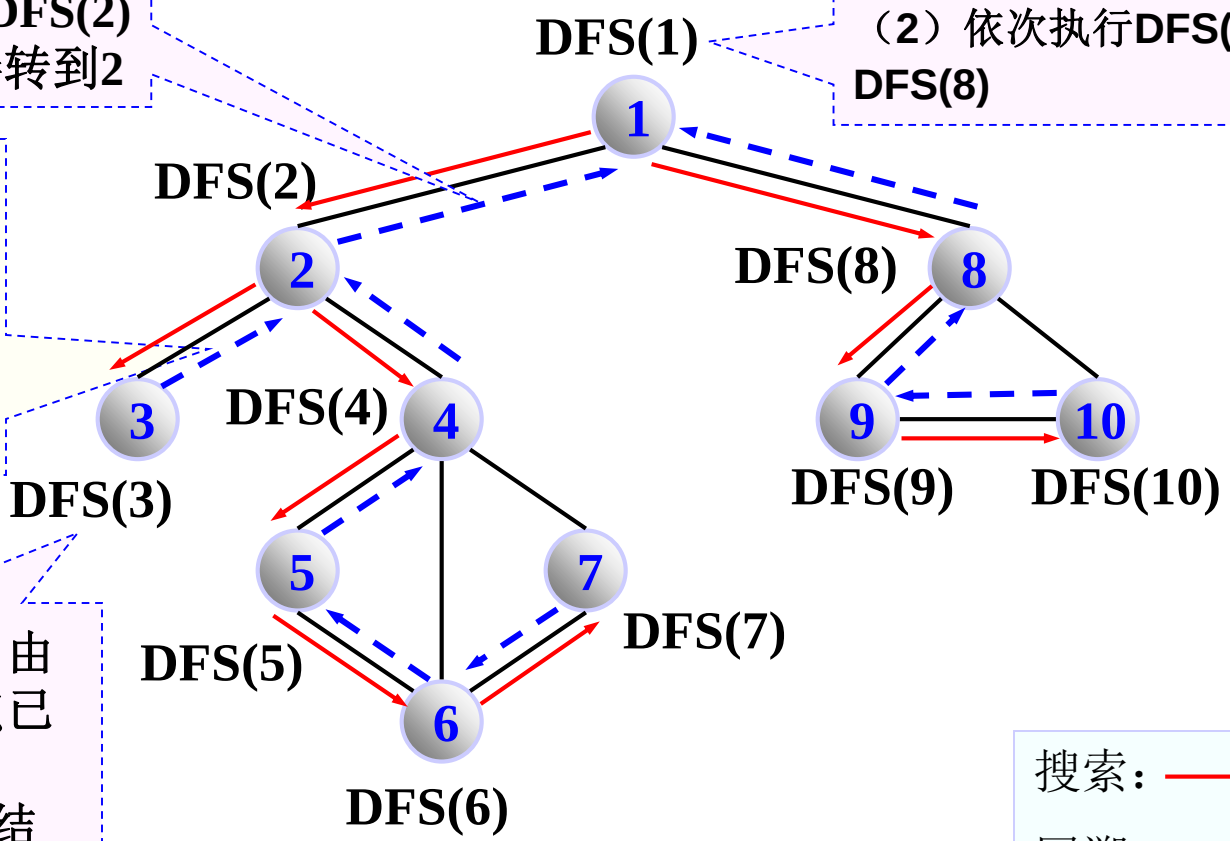
2) 依次从S的未被访问过的邻接点出发进行深度遍历.

【例1】对下图进行深度优先遍历

此箭头表示是从遍历运算DFS(1)中调用DFS(2)，即从顶点1直接转到2

DFS(1)包含如下两部分操作：
（1）访问顶点1；
（2）依次执行DFS(2)和DFS(8)

此虚箭头表示在DFS(3)执行完毕后返回到DFS(2)，即从顶点3返回到2



在访问顶点3后，由于顶点3的邻接点已全被访问，故DFS(3)执行到此结束，应返回到调用层，即返回到DFS(2)

搜索：

回溯：

顶点访问序列：1,2,3,4,5,6,7,8,9,10

【思考问题】

① DFS(1)访问顶点的次序是不是唯一的?

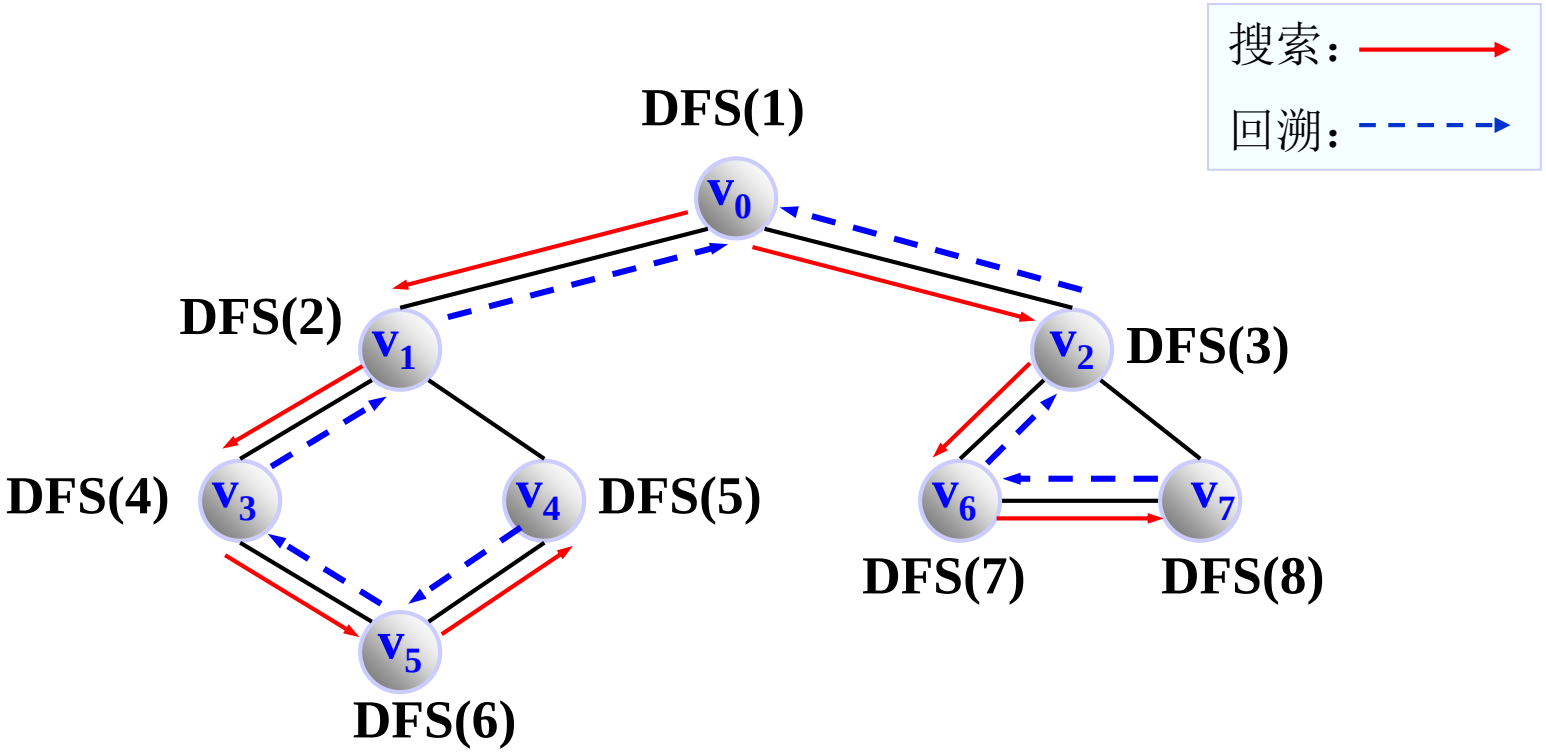
【答】不一定，本题中可能从DFS(1)先调用DFS(8)，再调用DFS(2)，由算法具体实现确定。

② 如果从DFS(7)开始遍历结果如何?

③ 从图中任意指定的顶点开始遍历呢?

【例2】对下图进行深度优先遍历

- 顶点$v_0,v_1,v_2,v_3,v_4,v_5,v_6,v_7$的编号依次为：1,2,3,4,5,6,7,8



顶点访问序列：1, 2, 4, 6, 5, 3, 7, 8。

即：$v_0, v_1, v_3, v_5, v_4, v_2, v_6, v_7$

【思考问题】

① DFS(1)访问顶点的次序是不是唯一的?

【答】不一定，本题中可能从DFS(1)先调用DFS(3)，再调用DFS(2)，由算法具体实现确定。

② 如果从DFS(4)开始遍历结果如何?

③ 从图中任意指定的顶点开始遍历呢?

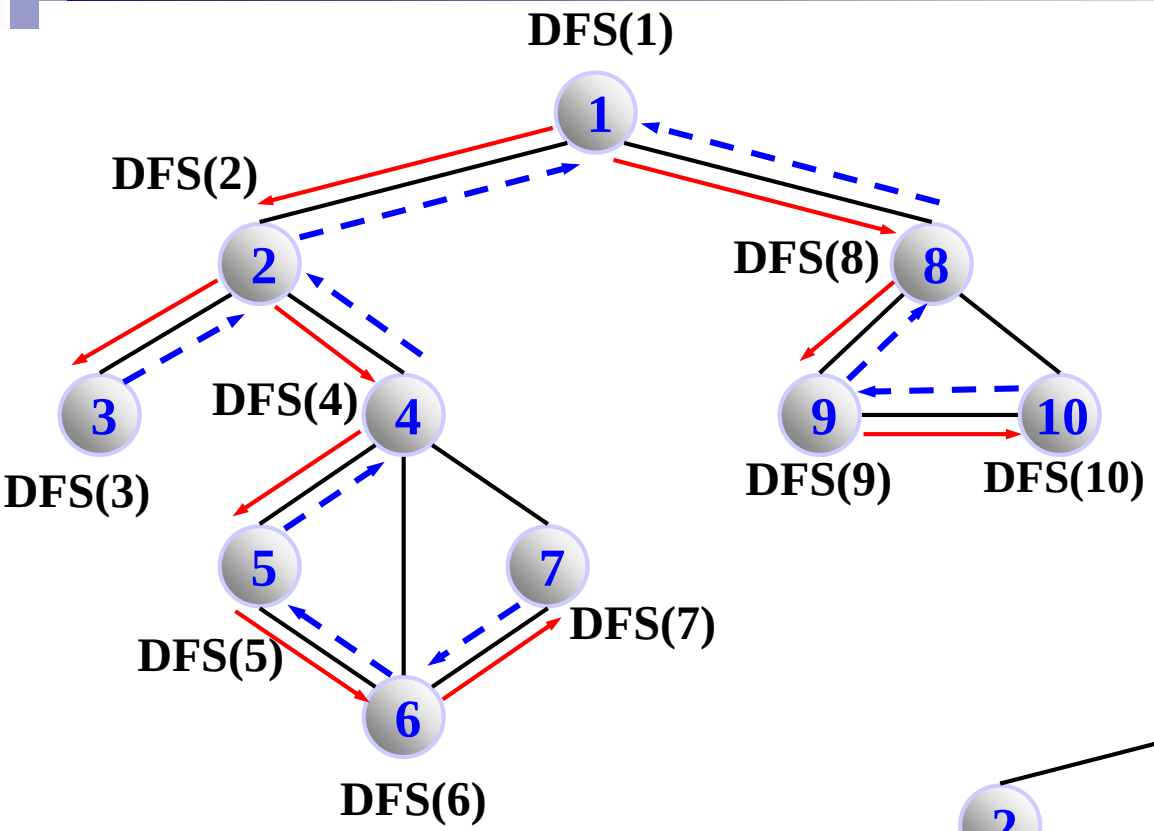
④ 树可以进行深度优先搜索遍历吗?

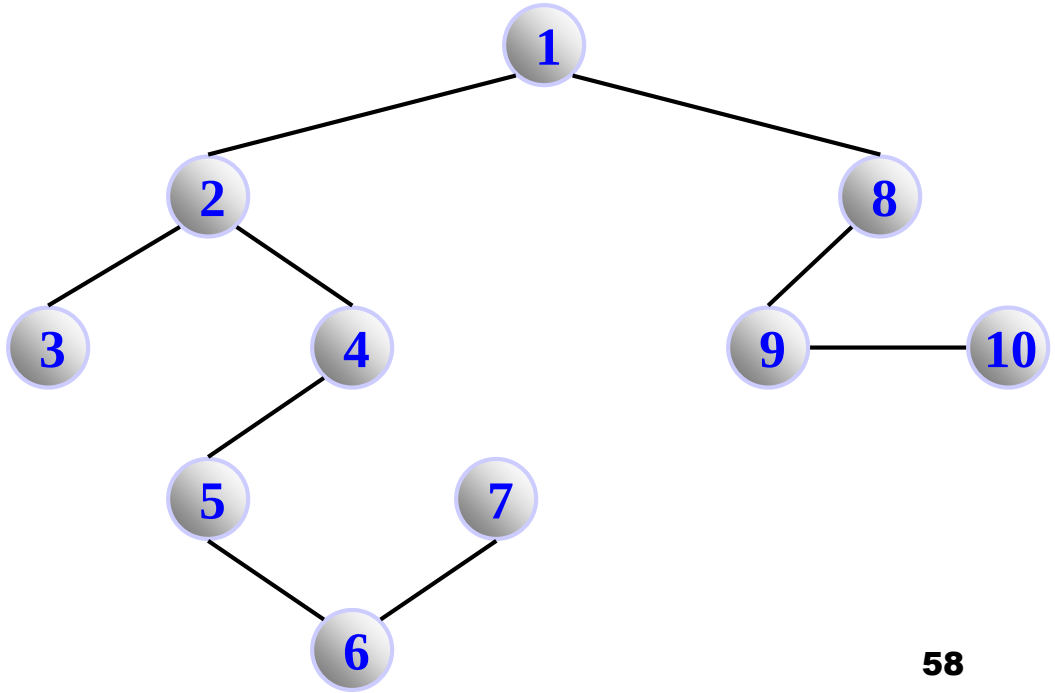
【答】可以，树也是图，只是相对简单。

⑤ 森林怎样进行深度优先搜索遍历吗?

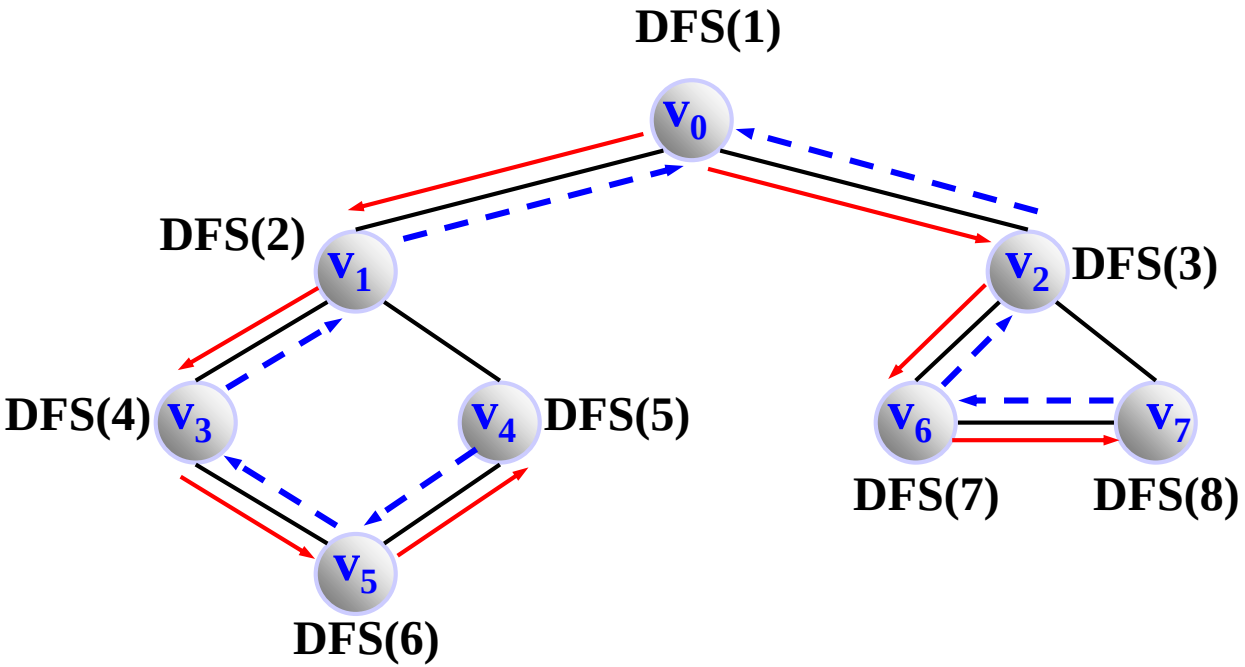
【答】一棵树接着一棵树遍历，直至遍历完森林中的每一棵树。

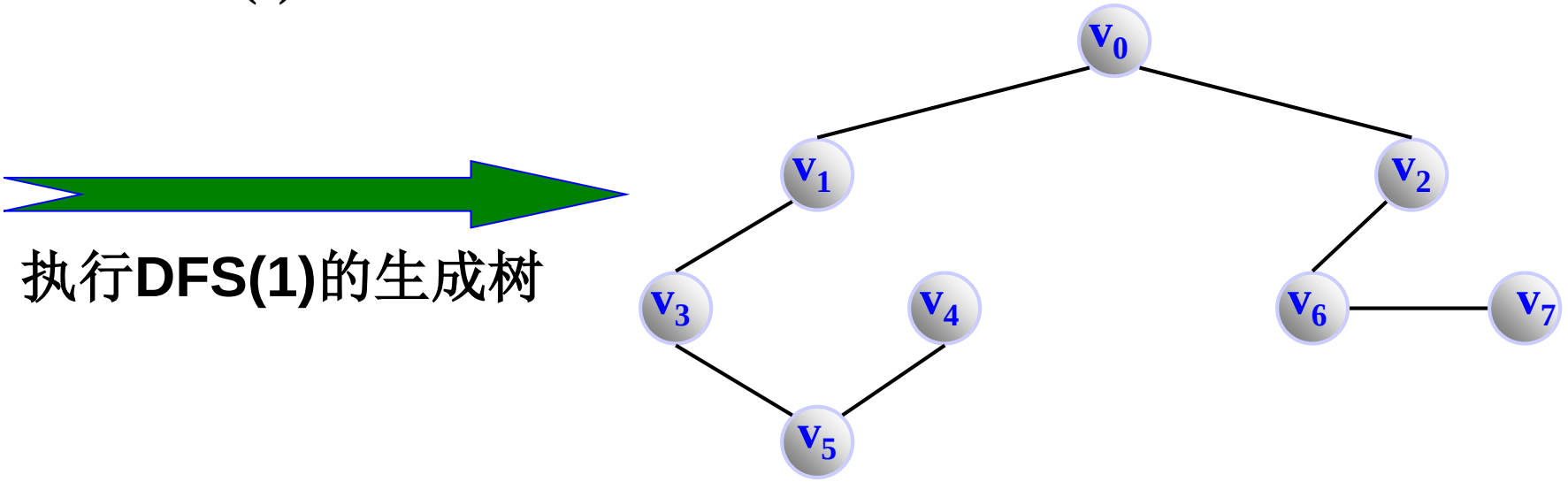
- **深度遍历生成树（DFS生成树）**
  - **对连通图和强连通图，保留深度优先遍历过程中途径的边（实例中有箭头标注的边），删除没有途径的边（没有箭头标注），则得到一棵树，叫做深度遍历生成树。**
  - **如果图是非连通，则会得到一个生成森林，每个连通分量对应一棵生成树。**


DFS(1)
1
DFS(2)
2
DFS(8)
8
3
DFS(4)
4
9
10
DFS(3)
DFS(9)
DFS(10)
5
7
DFS(7)
DFS(5)
6
DFS(6)


1
2
8
3
4
9
10
5
7
6

执行DFS(1)的生成树


DFS(1)
v0
DFS(2)
v1
v2
DFS(3)
DFS(4)
v3
v4
DFS(5)
v6
v7
DFS(7)
DFS(8)
v5
DFS(6)


v0
v1
v2
v3
v4
v6
v7
v5

执行DFS(1)的生成树

【思考问题】

- 对一个图，从不同的顶点开始遍历，得到的生成树是否相同?

【答】不一定相同。

- **3.状态空间深度优先搜索:**

  (1)把初始节点S放入OPEN表

  (2)如果OPEN表为空,则问题无解,退出

  (3)把OPEN表的第一个节点(记为节点n)取出,放入CLOSED表

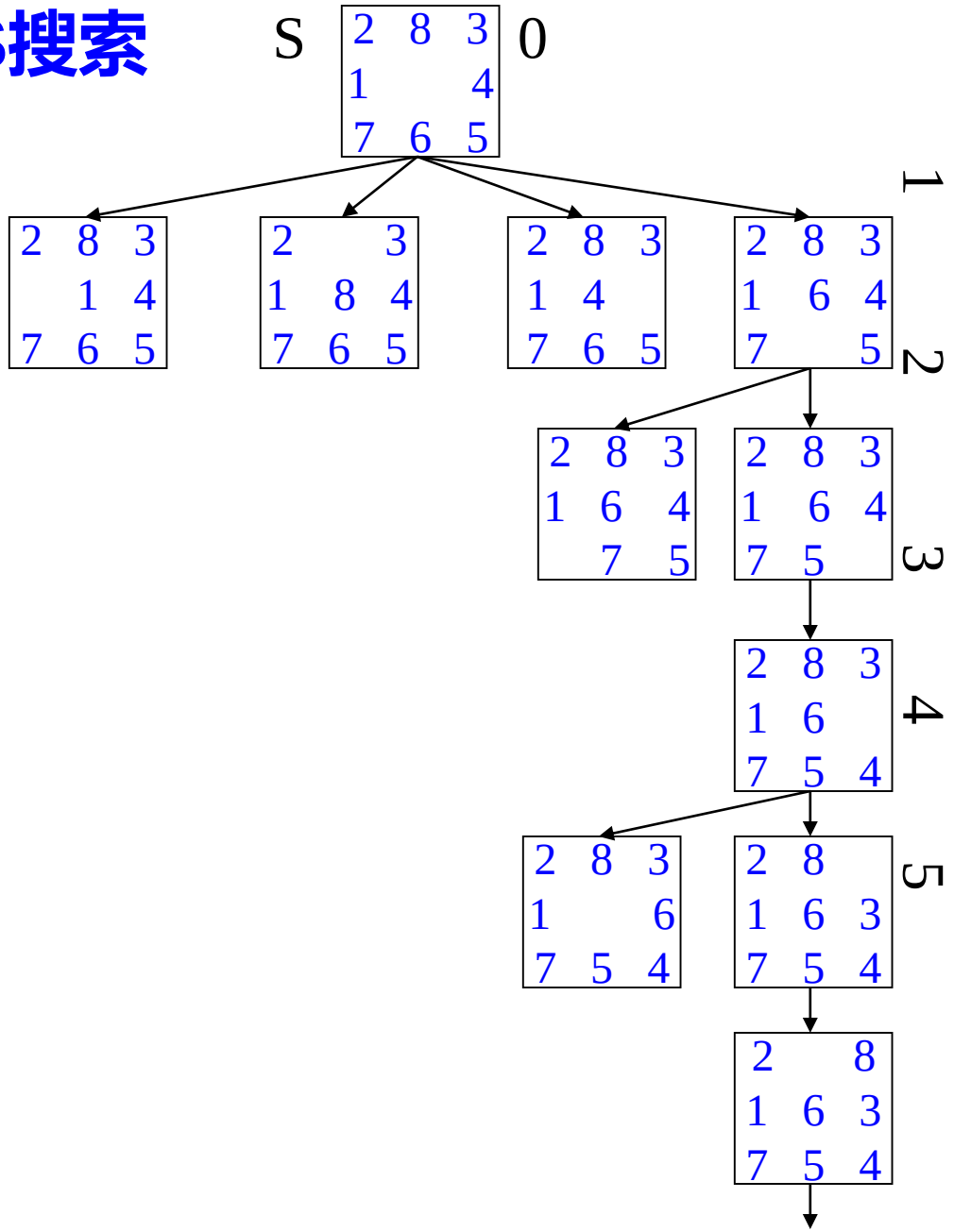
  (4)考查节点n是否为目标节点.若是,则求得了问题的解,退出

  (5)若节点n不可扩展,则转第(2)步

  (6)扩展节点n,将其子节点放入OPEN表的首部,并为每一个子节点都配置指向父节点的指针,转第(2)步

【例3.4.6】 DFS搜索

- 8数码问题



- **4.深度优先搜索性能：**
  - ① **非完备的**
    - **如果当前搜索分支为无穷分支且无解，搜索将一直持续下去而得不到解。**
  - ② **非最优的**
  - ③ **时间复杂度： O（a^b）**
    - **树的分枝因子（度）：树中最大的子节点数（按最坏情况考虑），设为a。**
    - **搜索深度：b**
  - ④ **空间复杂度：O（ a*b ）**

- **应用实例：**
  - 搜索引擎中的网络爬虫—选取一个网页，选择一个超链接，进入下一个页面，选择一个超链接，再进入下一个页面，直到一个页面没有超链接，再逐层返回处理下一个超链接。

- **注意：**
  - 这里的BFS和DFS与数据结构中的算法的唯一区别是不一定要遍历所有顶点。
  - 这里只要到达目标顶点，算法即结束。

3.4.3有界深度优先搜索

- **在深度优先搜索中，如果进入无穷且无解分支，搜索将一直持续下去，得不到问题的解，白白浪费计算机的时、空资源。为了防止出现此类情况人们提出了有界深度优先搜索策略。**
- **指定最大搜索深度dmax作为深度界限，仍按深度优先搜索方法搜索，当搜索到深度界限仍未到达目标，则视此搜索路径无解，继续搜索其他路径。**
- **有界深度优先搜索性能同深度优先搜索。**

- **有界深度搜索过程如下:**

  (1)把初始节点S放入OPEN表,置S的深度d(S)=0

  (2)如果OPEN表为空,则问题无解,退出

  (3)把OPEN表的第一个节点(记为节点n)取出,放入CLOSED表

  (4)考查节点n是否为目标节点.若是,则求得了问题的解,退出

  (5)如果节点n的深度d(节点n)=dmax,则转第(2)步

  (6)若节点n不可扩展,则转第(2)步

  (7)扩展节点n,将其子节点放入OPEN表的首部,并为每一个子节点都配置指向父节点的指针,转第(2)步

3.4.4迭代加深深度优先搜索

- **Iterative deepening Depth-first search**
- **在有界深度优先搜索中，如果深度界限设定不合适，太小则可能遗漏问题的解，太大则浪费时空资源。**
- **迭代加深搜索中，深度界限是动态变化的，从深度为1开始，找不到目标，就把深度界限加1，重新开始深度优先搜索，直到找到解或无解为止。**

- **迭代加深深度优先搜索性能：**
  - ①**完备的**
  - ②**最优的– 对于深度指标**
  - ③**时间复杂度： O（a^b）**
    - **树的分枝因子（度）：树中最大的子节点数（按最坏情况考虑），设为a。**
    - **搜索深度：b**
  - ④**空间复杂度：O（ a*b ）**

临渊羡鱼，不如退而结网。

3.5 状态空间的启发式搜索

- **启发式搜索，又叫做有信息搜索**
- **Heuristic search，Informed Search**
- **上节讨论的搜索策略按事先规定的路线进行搜索。这些策略搜索效率低下，浪费计算机时空资源，容易造成组合爆炸。可能丢掉最优解甚至全部解。**
- **本节介绍的搜索策略，在搜索过程中，针对问题自身的特性，利用问题领域的相关信息来帮助搜索，使得搜索朝着最有希望的方向前进，提高搜索效率。**

3.5.1启发性信息和估价函数

- **1.启发信息**
  - **关于问题领域的，用来帮助搜索的信息。**
- **2.启发信息按用途分类**

1) 用于决定下一个要扩展的节点

总是选择最有希望产生目标的节点（邻接点）优先扩展，即OPEN表按此希望值排序。这类启发信息使用最为广泛。

2) 用于决定产生哪些子节点

扩展一个节点时，有选择的生成子节点（选择访问邻接点），有些明显无用或没有优势的子节点不让其产生出来。

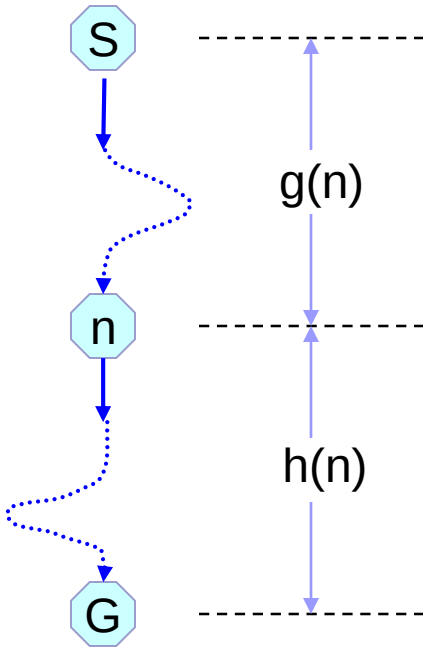
3) 用于决定从搜索图中修剪或抛弃那些节点

减小待搜索空间。

搜索图中，不访问这些顶点，或直接删除掉这些顶点。

3.5.1启发性信息和估价函数

- **3.估价函数（evaluation function）与启发函数（heuristic function）**
  - **利用启发信息，构造一个函数，对搜索路径全程的代价或希望进行评估。**
  - **如右图，当前节点为n，定义估价函数f(n)：从初始节点S开始，约束通过n，到达目标节点G的全程代价的估计值。即：**
    f(n)=g(n)+h(n)
    - **其中：g(n)为从S到n，已经走过的路径代价估计，通常即用实际花费代价，也比较容易计算。**
    - **h(n)是从n到达目标G代价的估值，这一段路径是没有走过的，必须根据问题特性，利用启发信息进行估算。搜索的启发性即体现在h(n)上，所以把h(n)叫做启发函数。**



3.5.1启发性信息和估价函数

- **估价函数的构造对多数问题不是一件容易的事。**
- **搜索的有效性取决于估价函数的构造，不当的估价函数可能失去最优解甚至全部解。此外，构造估价函数时要考虑时空代价的折中。**
- **保证有效性的优先次序是至少保证能找到解，然后保证能获得较优解，最后是获得一个最优解。**

3.5.2 A算法

- **1.什么是A算法?**
  - **又叫最好优先搜索(Best First Search)，有序搜索(Ordered Search)**
  - **在图搜索算法中，如果能在搜索的每一步都利用估价函数f(n)=g(n)+h(n)对Open表中的节点进行排序，则该搜索算法为A算法。**

- **2. A算法描述：**

  (1)把初始节点S放入Open表中，f(S)=g(S)+h(S)；

  (2)如果Open表为空，则问题无解，失败退出；

  (3)把Open表的第一个节点取出放入Closed表，并记该节点为n；

  (4)考察节点n是否为目标节点。若是，则找到了问题的解，成功退出；

  (5)若节点n不可扩展，则转第(2)步；

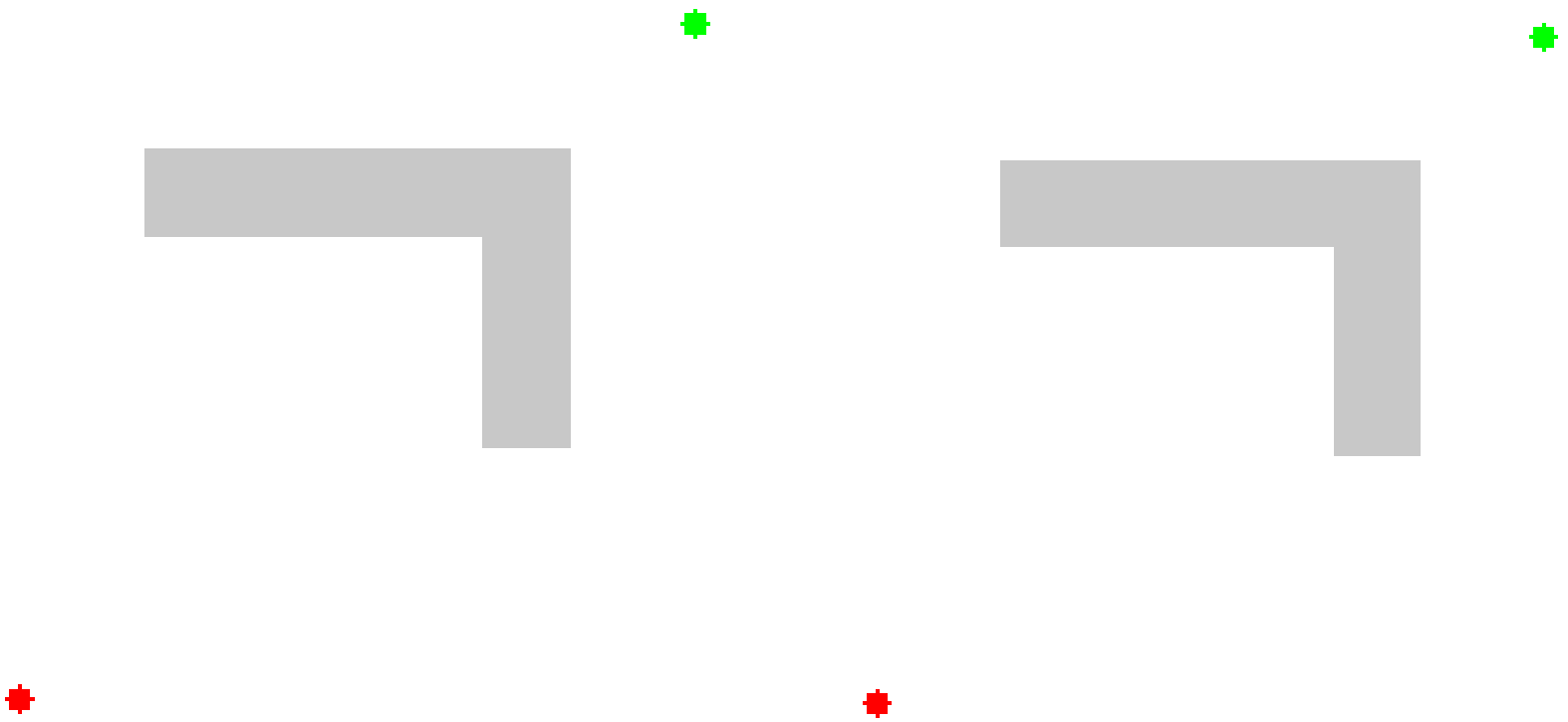
  (6)扩展节点n，生成其子节点ni(i=1, 2, …)，计算每一个子节点的估价值f(ni)(i=1, 2, …)，并为每一个子节点设置指向父节点的指针，然后将这些子节点放入Open表中；

  (7)根据各节点的估价函数值，对Open表中的全部节点按从小到大的顺序重新进行排序；

  (8)转第(2)步。

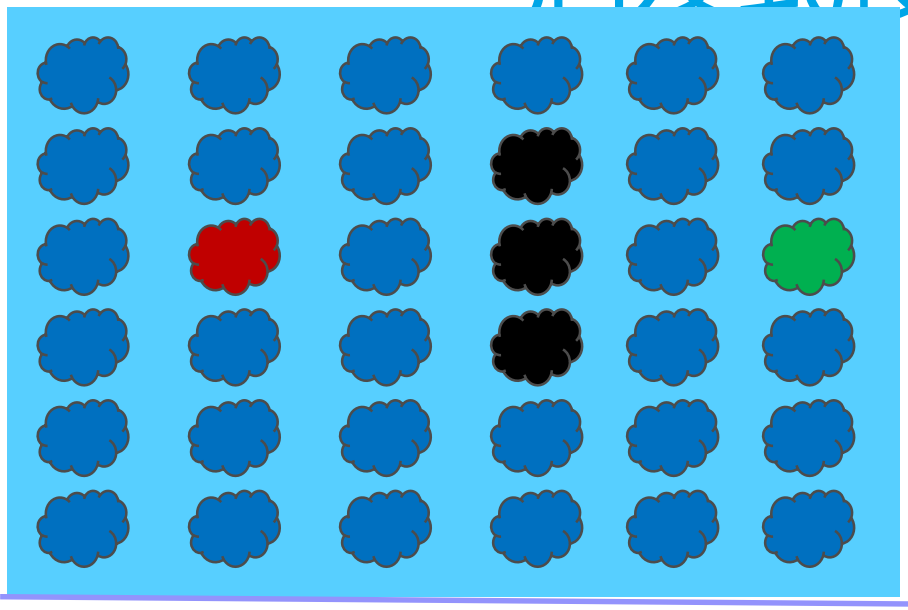
- **3. A算法性能：**
  - ①**完备的**
  - ②**非最优的**
  - ③**时间复杂度： O（a^b）**
    - **树的分枝因子（度）：树中最大的子节点数（按最坏情况考虑），设为a。**
    - **搜索深度：b**
  - ④**空间复杂度： O（a^b）**

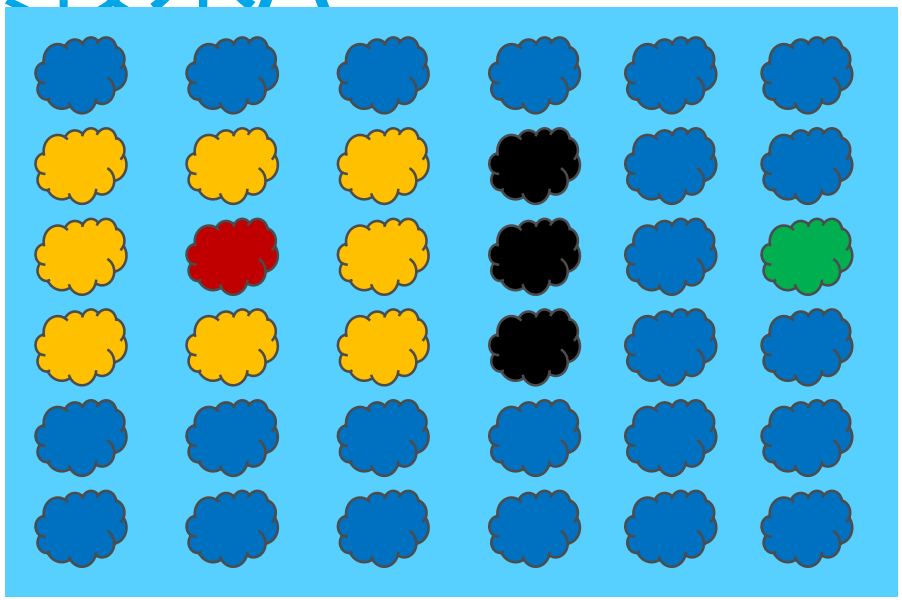
启发式搜索技术A*

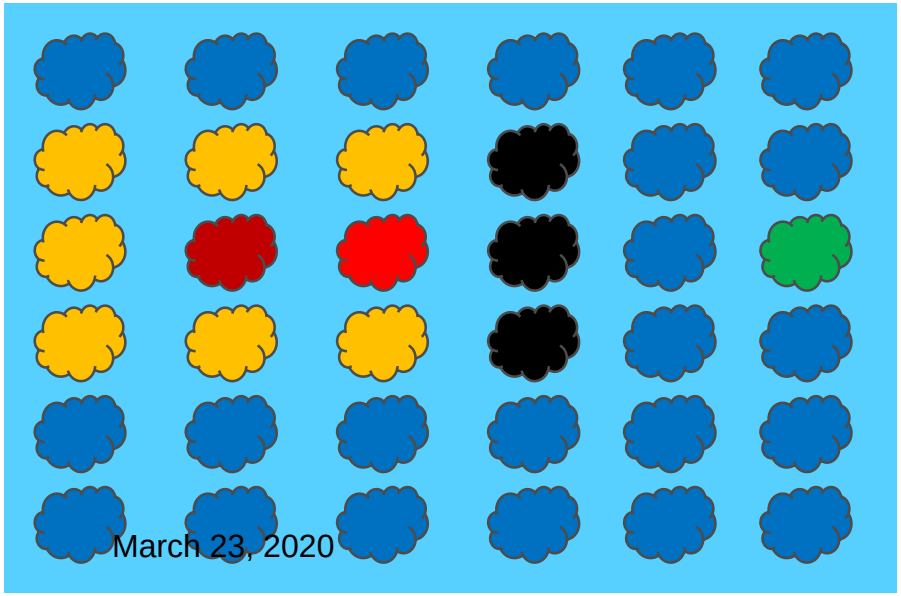


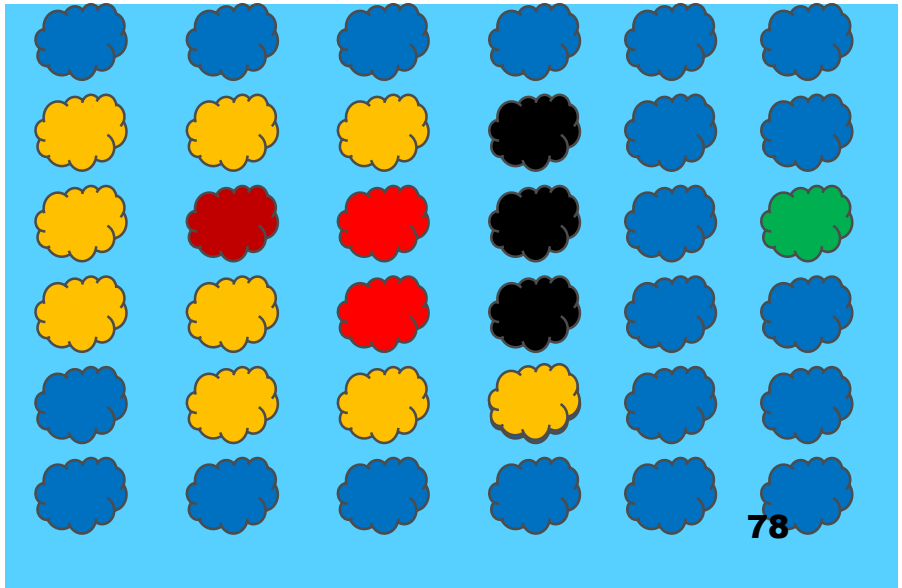
http://en.wikipedia.org/wiki/A*_search_algorithm

启发式搜索技术A*

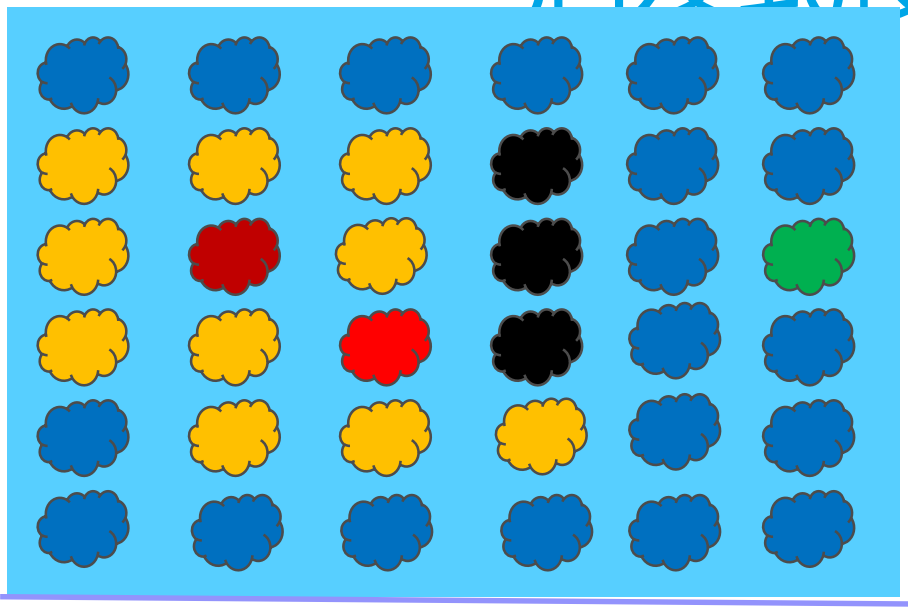


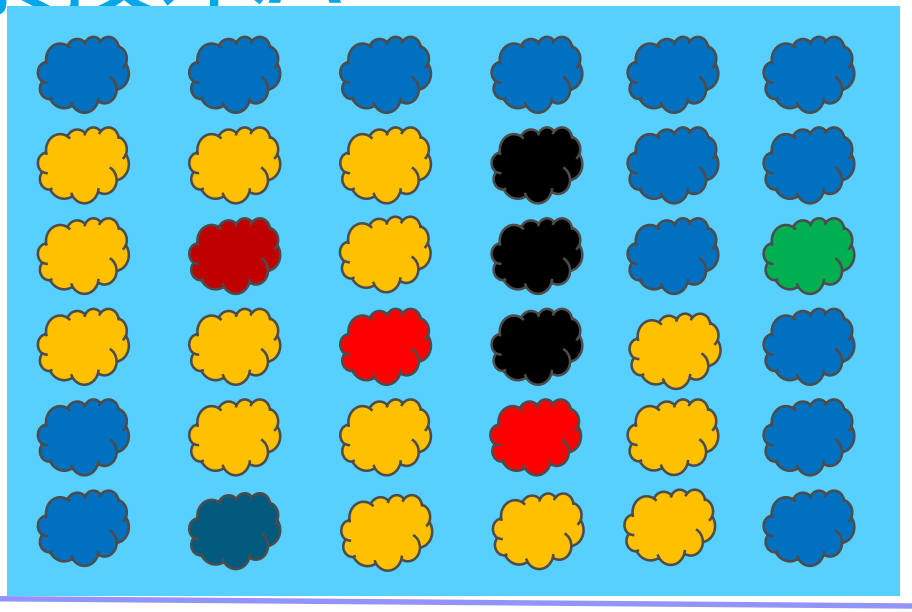


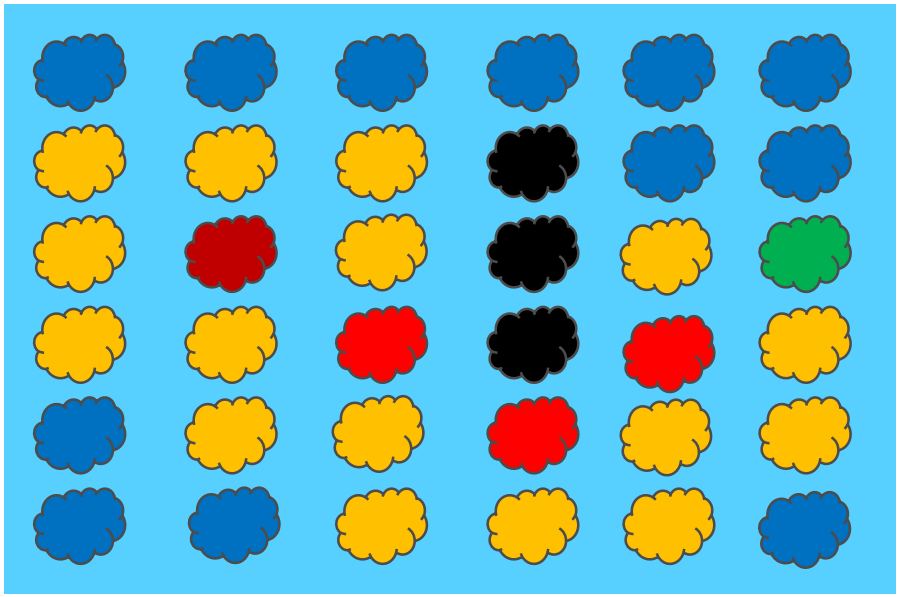


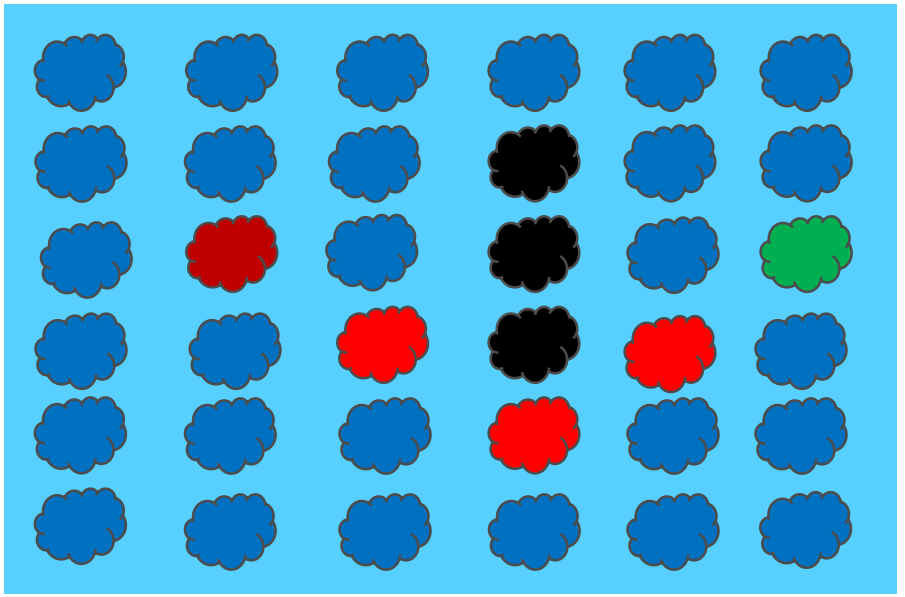


启发式搜索技术A*



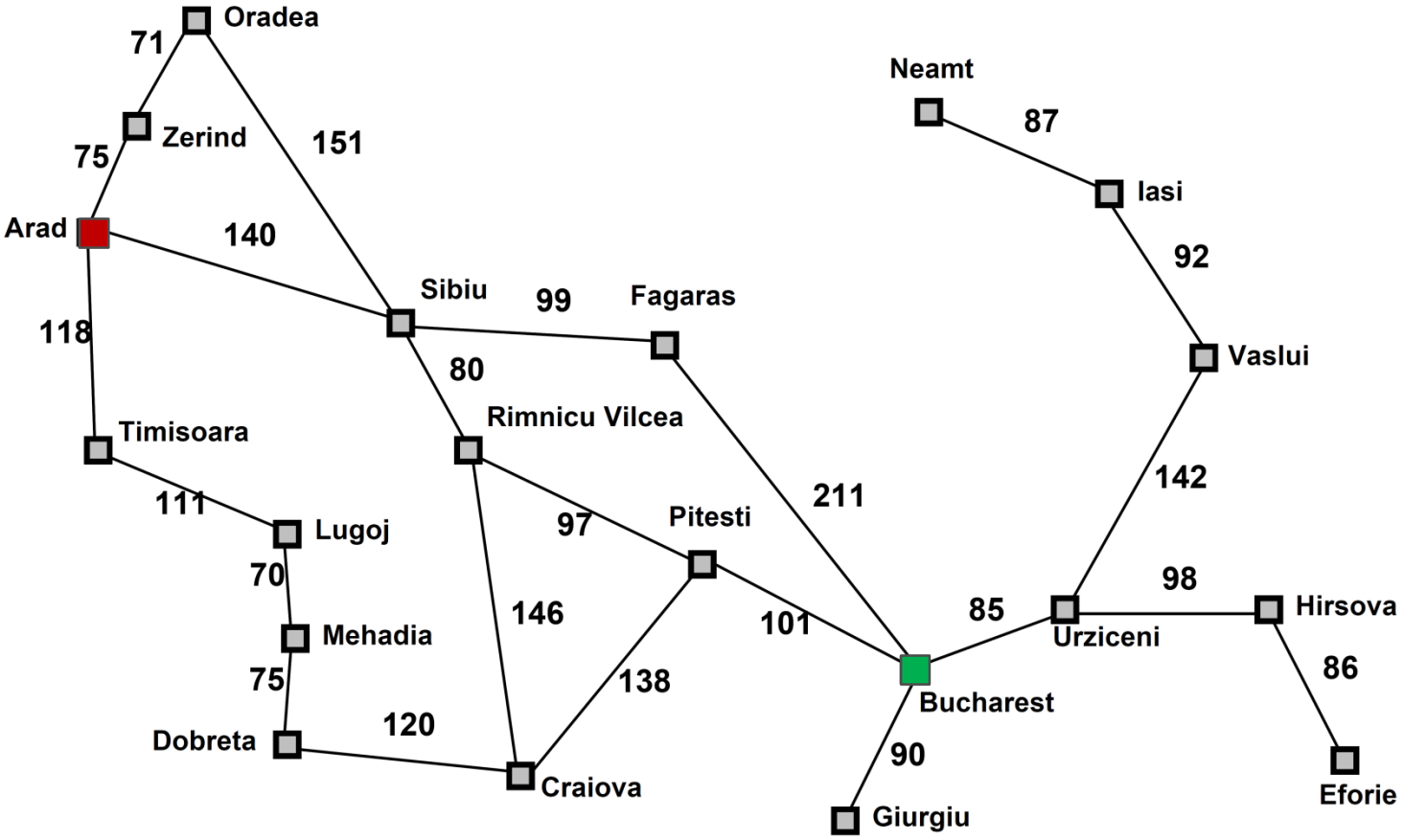






启发式搜索技术A*

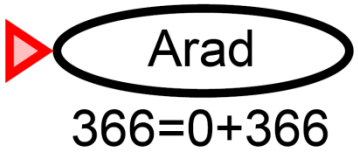
Romania with step costs in km



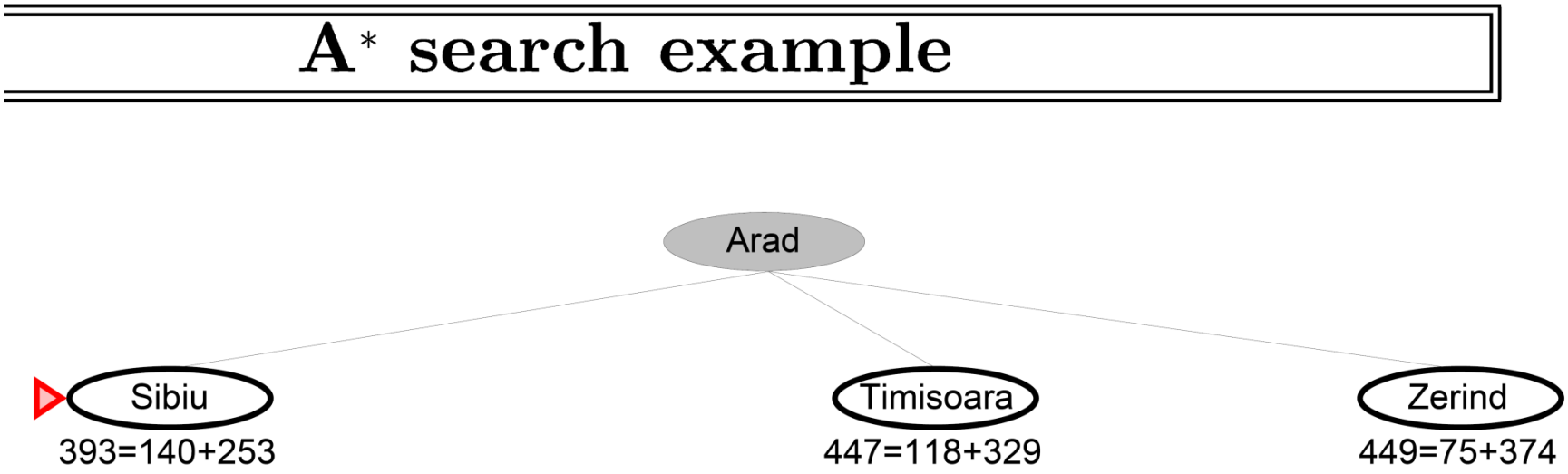
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

启发式搜索技术A*

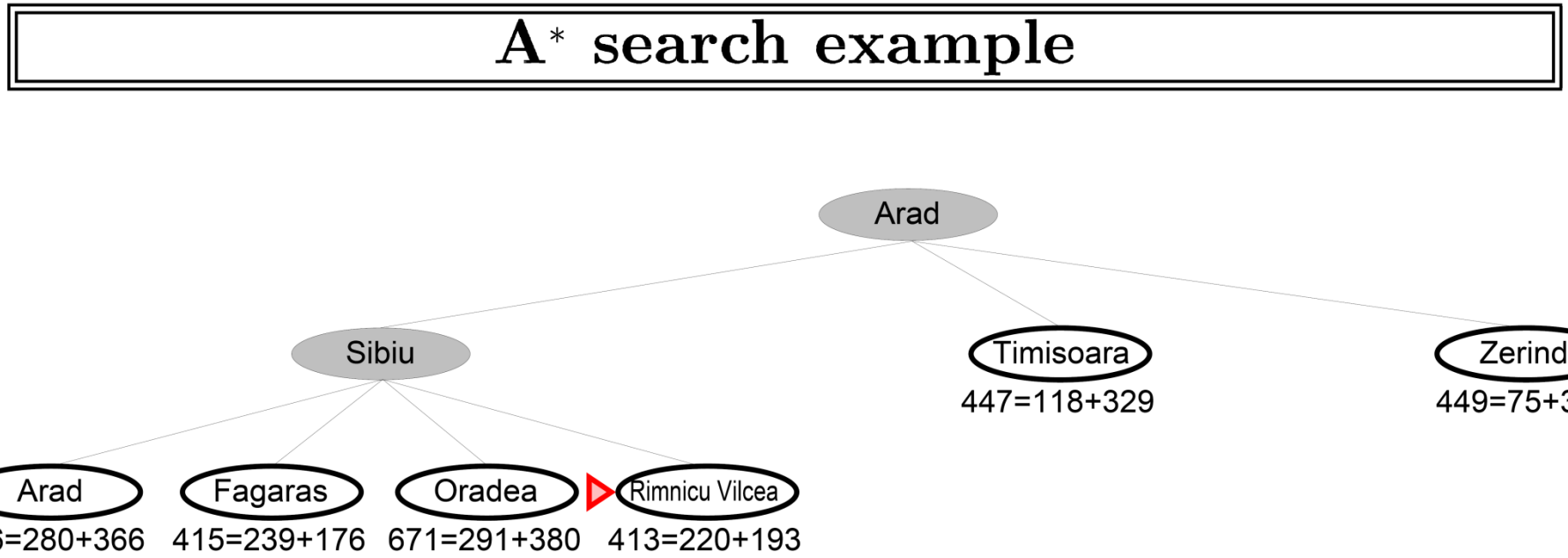
A* search example



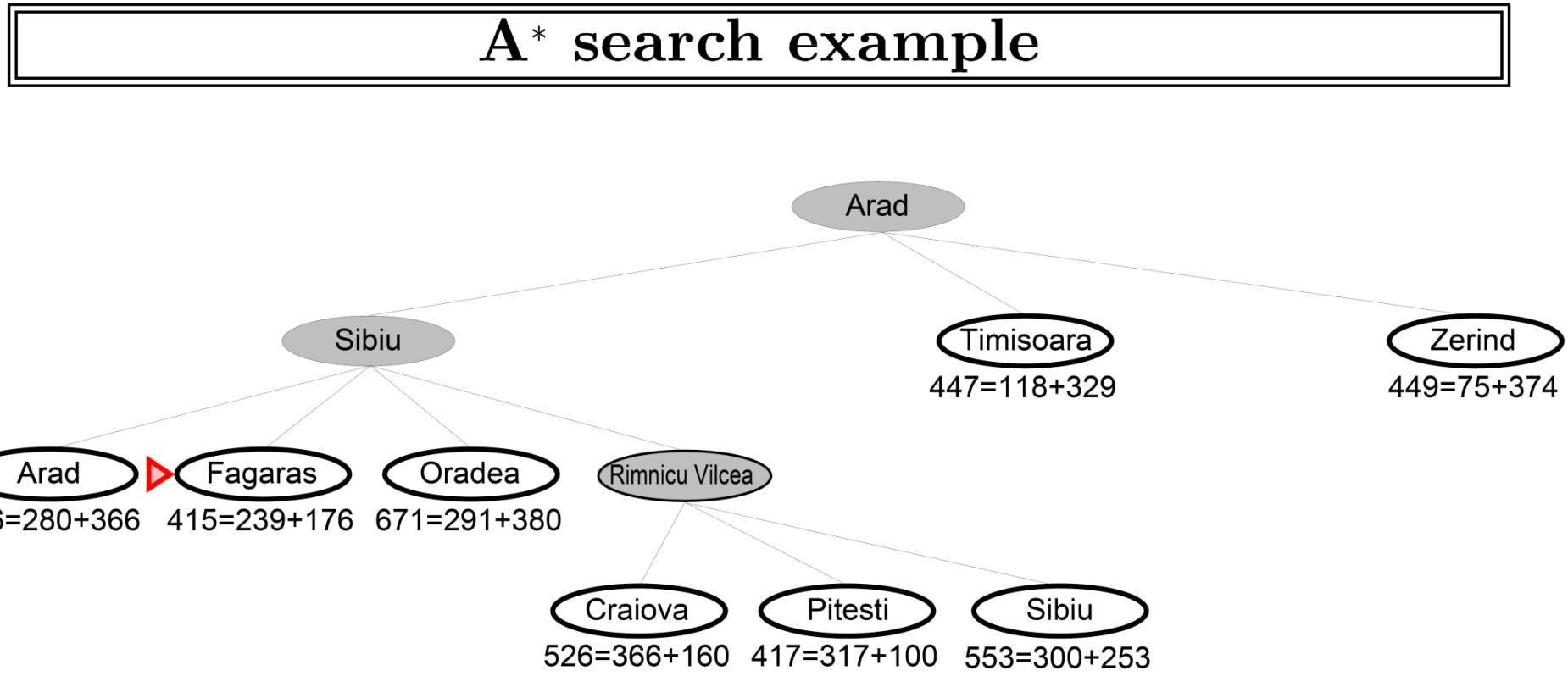
启发式搜索技术A*


A* search example
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

启发式搜索技术A*

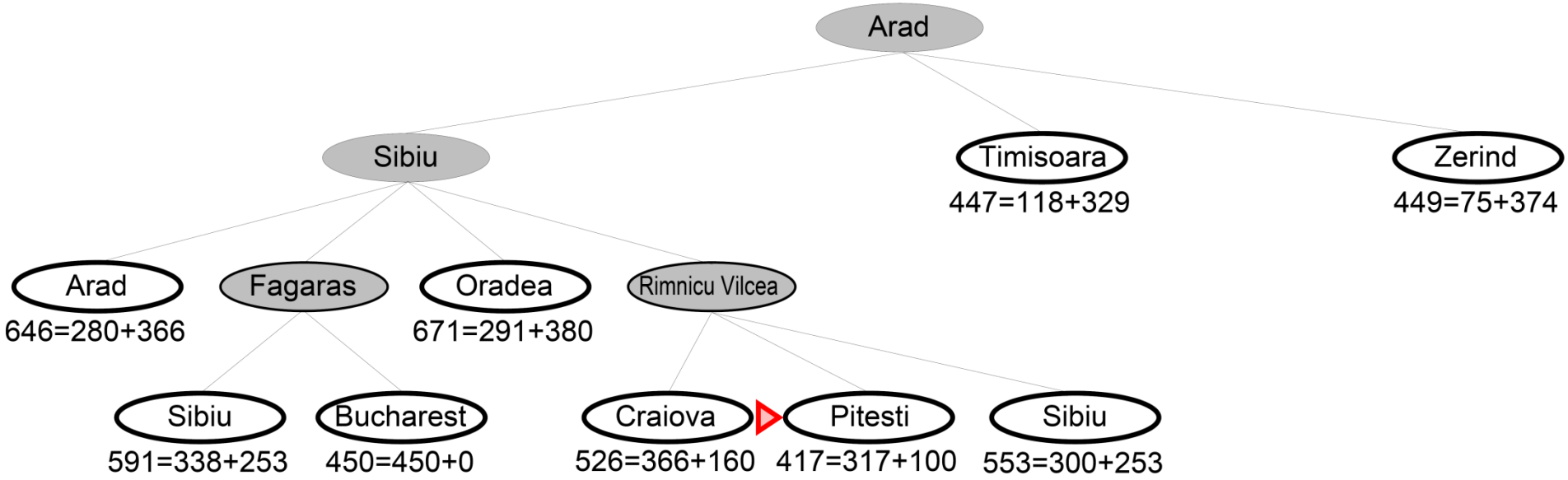

A* search example
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

启发式搜索技术A*

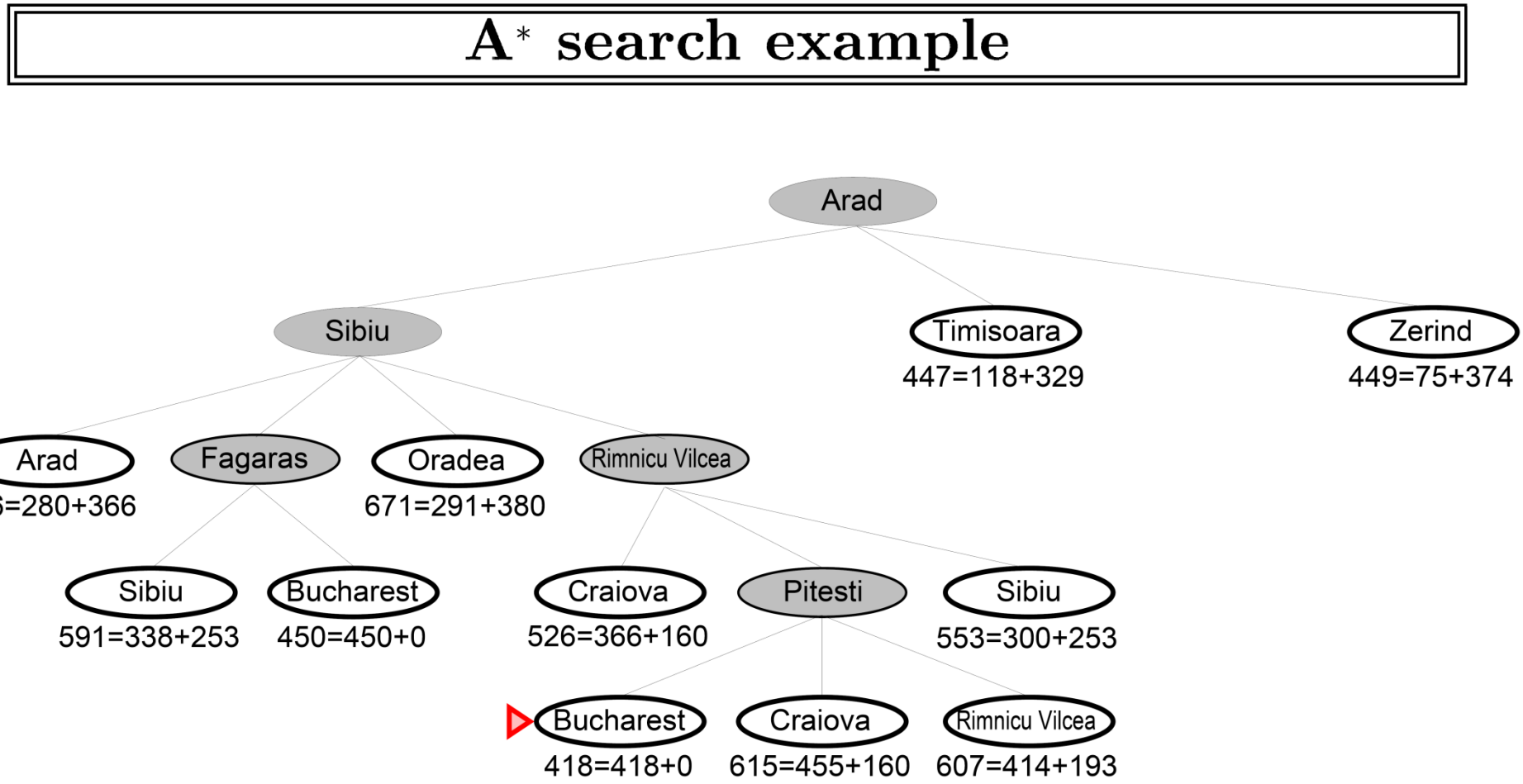

A* search example
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

启发式搜索技术A*

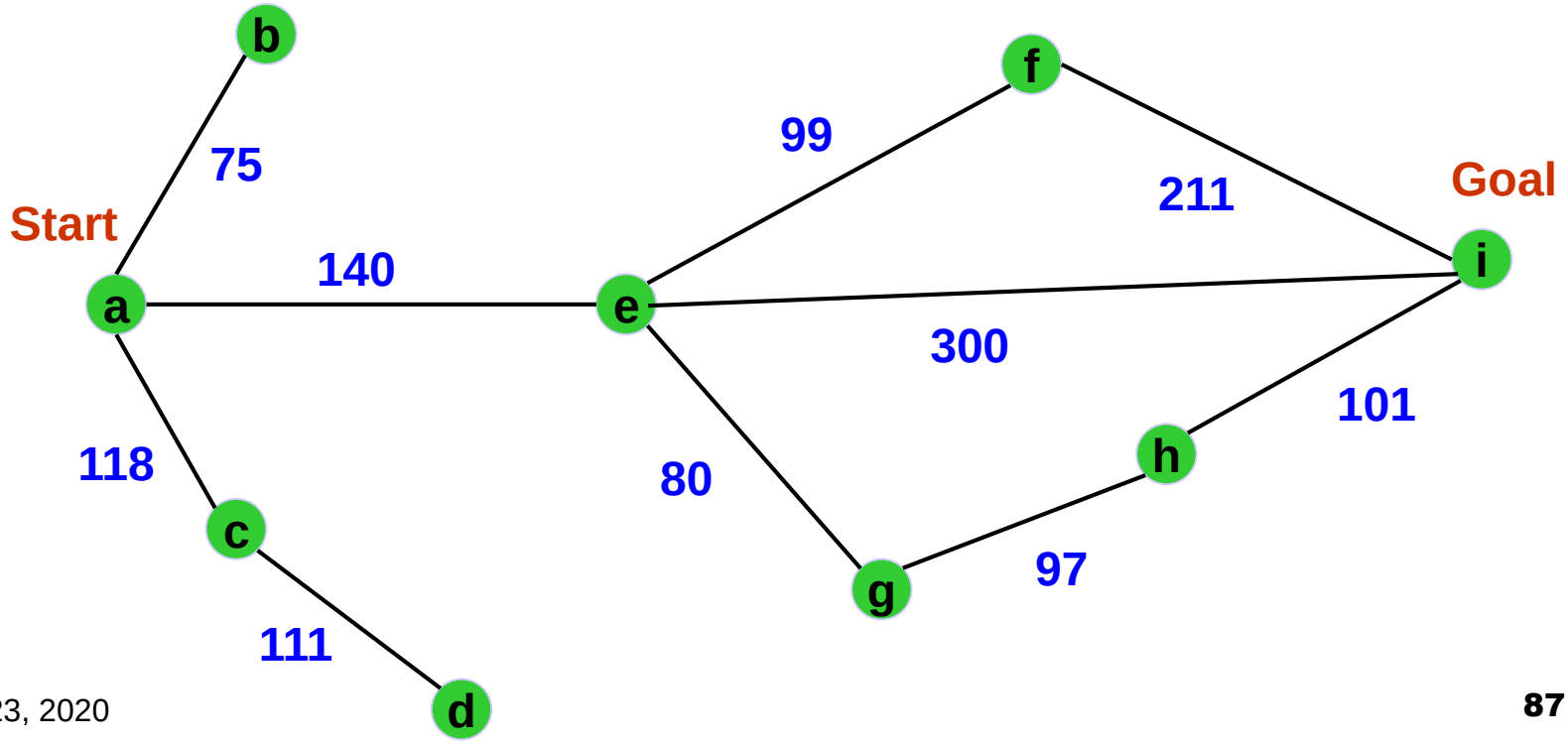
A* search example



启发式搜索技术A*


A* search example
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

- 【例3.5.1】 下图为一个城市地图，用最好优先搜索求从 a城市出发到达 i 城市的路径。g(n)用从a城市到n城市走过的实际距离。h(n)用后面的表给出，您可以设想h(n)为n到达i的直线距离。



解：用改造的OPEN表和CLOSED表表示求解过程。表中当前节点包含了从a的搜索路径，以及评估函数值。如aeg(413)，表示当前节点为g，搜索路径为从a到e，再到g，评估函数值f(g)=g(g)+h(g)。

其中g(g)为从a到e，再到g两路段的实际距离之和，即
g(i)=140+80=220

h(g)为g到i的启发函数值，查表得：
h(g)=193

所以：f(g)=g(g)+h(g)=220+193=413

其它节点处理方式相同。

当前城市	h(n)
a	366
b	374
c	329
d	244
e	253
f	178
g	193
h	98
i	0

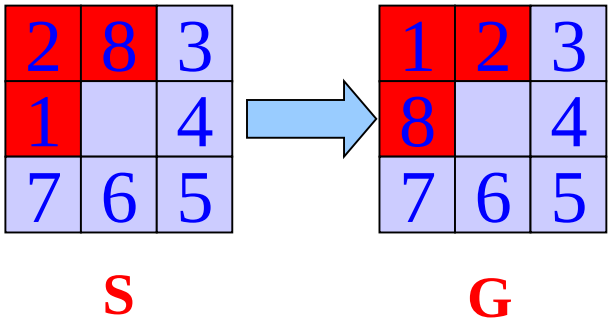
求解过程如下表，用A算法求解路径为： aeghi(418)

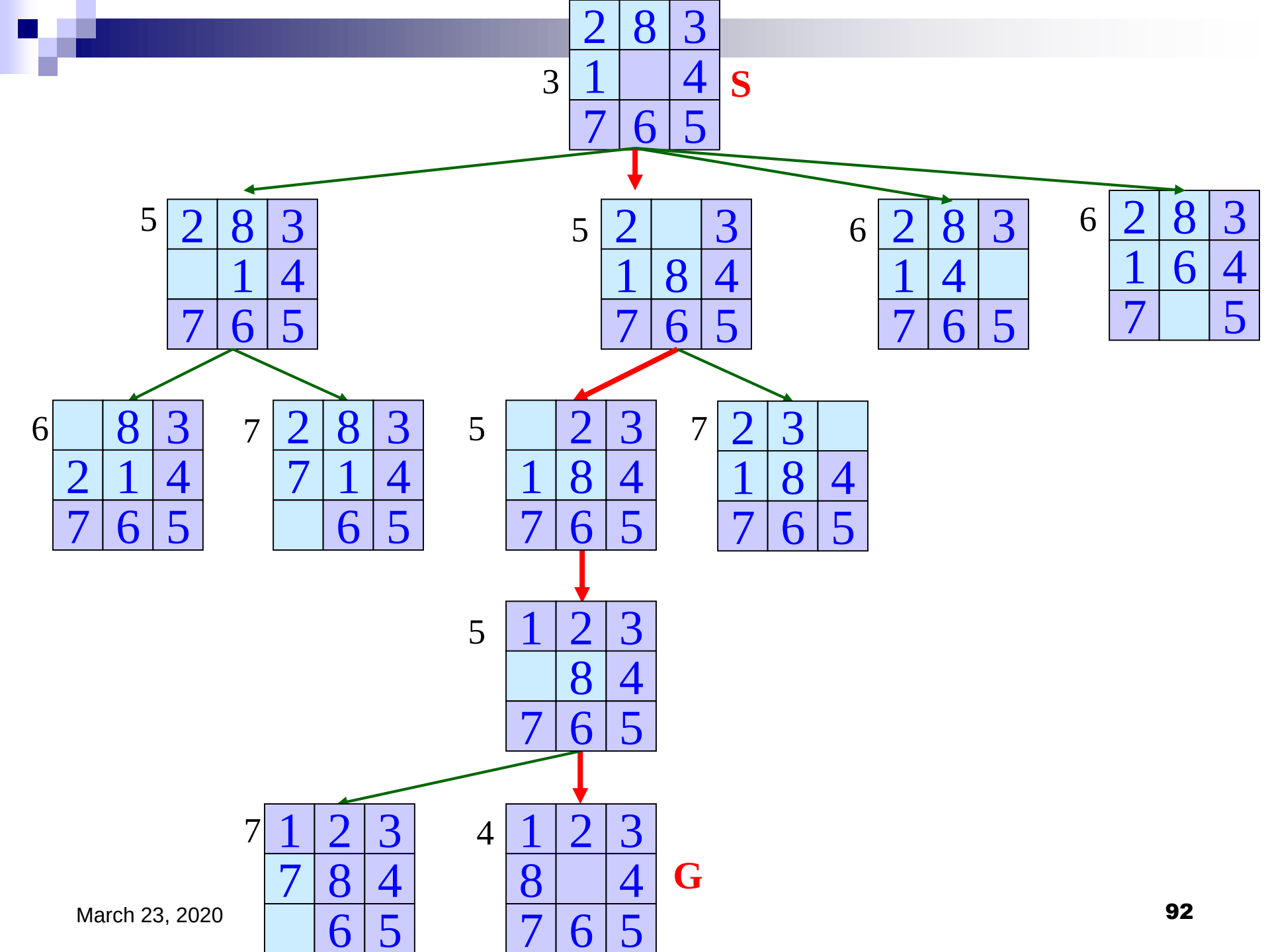
其中：实际路径代价（距离）为418。

OPEN表	当前扩展	CLOSED表
a(366)	a	a
ab(449),ac(447),ae(393)	e	a, e
ab(449),ac(447),aei(440),aef(417), aeg(413)	g	a, e, g
ab(449),ac(447),aei(440),aef(417),aegh(415)	h	a, e, g, h
ab(449),ac(447),aei(440),aef(417),aeghi(418)	f	a, e, g, h, f
ab(449),ac(447), aei(440), aeghi(418),aefi(450)	i	a, e, g, h, f, i

- 分析：本题有3条解路径：aei，全程实际代价为440；aefi，全程实际代价为450；aeghi，全程实际代价为418。
  - A搜索算法得到的解为aeghi，本题中为最优解，因为本题满足了A算法的条件。
  - 但A算法不保证能取得全局最优解。

- **【例3.5.2】 A算法求解8数码问题**
  - **解：定义评估函数: f(n)=g(n)+h(n)**
    - **g(n)=节点深度**
    - **h(n)=当前盘面与目标对比,错位数字的个数**
  - **如初始状态下: f(S)=g(S)+h(S)=0+3=3**




3
S
5
5
6
6
6
7
5
7
5
7
4
G

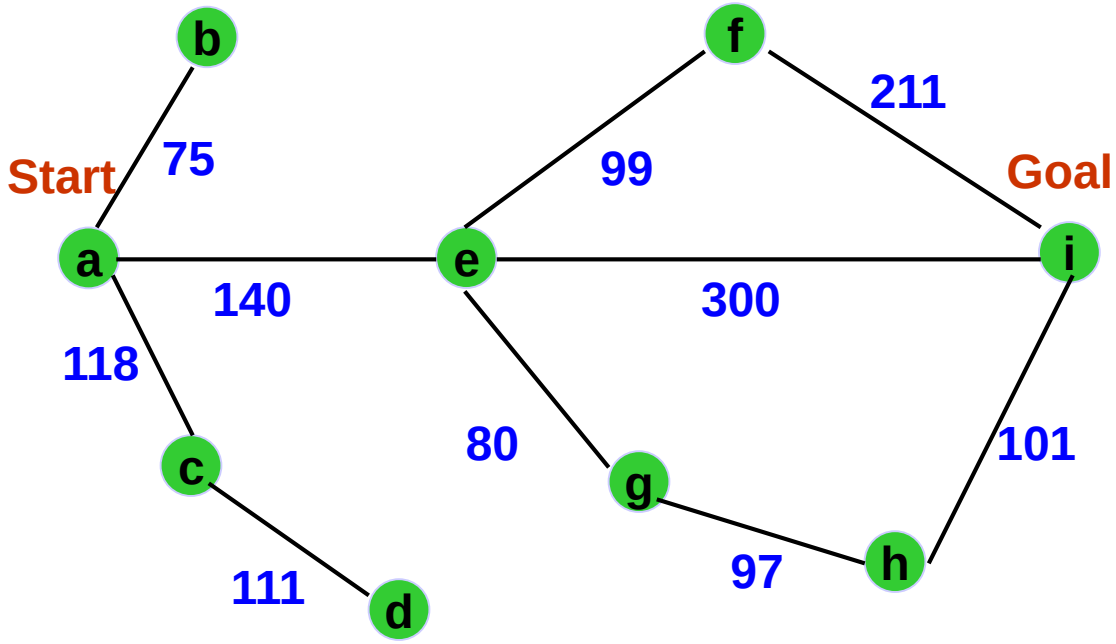
3.5.2 爬山搜索算法

人生不要
总走近路

- Hill Climbing Search
- 爬山搜索是一种贪婪(greedy)算法

- **1.什么是爬山搜索?**
  - 评估函数f(n)=g(n)+h(n)中，令g(n)=0，即f(n)=h(n)，A算法即成为爬山搜索算法。
- **2.爬山搜索优点**
  - 搜索效率高
- **3.爬山搜索问题**
  - 常常陷入局部最优，而失去全局最优解。

- 例3.5.3 用爬山搜索求解下图TSP问题。



当前城市	h(n)
a	366
b	374
c	329
d	244
e	253
f	178
g	193
h	98
i	0

□解：f(n)=h(n)，查表获得评估函数值。搜索过程见下表。解为：aei。

OPEN表	当前扩展	CLOSED表
a(366)	a	a
ab(374),ac(329),ae(253)	e	a, e
ab(374),ac(329),aei(0),aef(178), aeg(193)	i	a, e, i

□分析：本题有3条解路径：aei，全程实际代价为440；aefi，全程实际代价为450；aeghi，全程实际代价为418。

- 爬山搜索得到的解为aei，可见不是全局最优解。
- 对比前面A算法，可见爬山搜索效率较高。

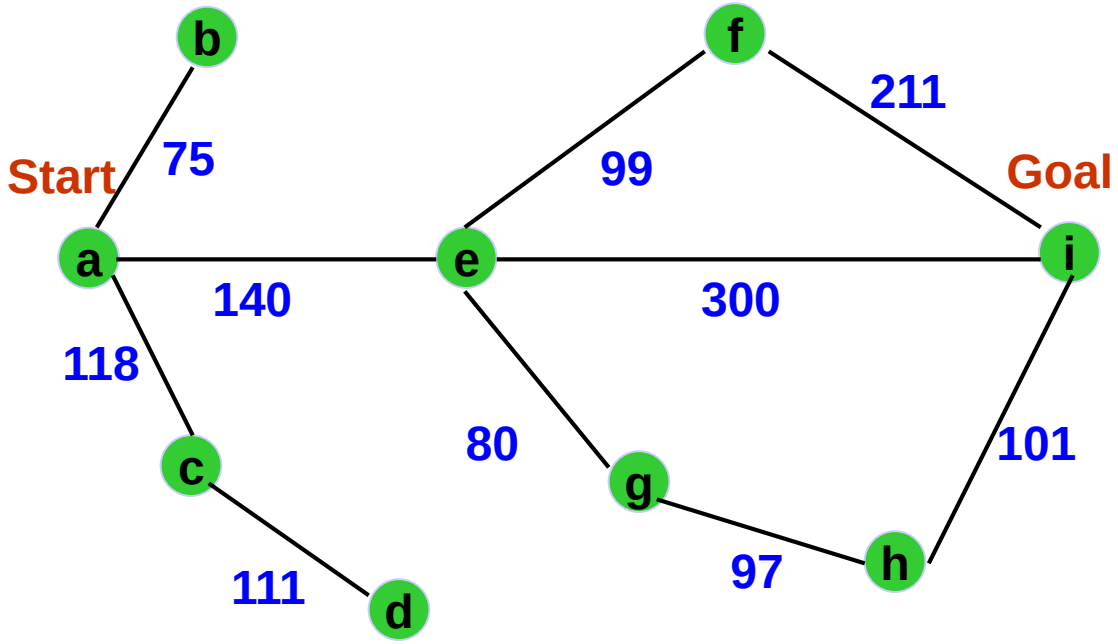
3.5.3 等代价搜索

- 等代价搜索
- Uniformed-cost search
- 即：Dijkstra算法

- **1.什么是等代价搜索?**
  - 评估函数f(n)=g(n)+h(n)中，令h(n)=0，f(n)=g(n)，且g(n)为实际路径代价，此时的A算法称为等代价搜索。
- **2.为什么属于盲目搜索?**
  - 因为h(n)=0，没有了启发性，所以属于盲目搜索。

- **3.等代价搜索性能**
  - 最优性—找到全局最优解
  - **完备性**
  - 时空复杂度同A算法
- **4.等代价搜索的问题**
  - 搜索效率低

- 【例3.5.4】 用等代价搜索求解下图TSP问题。



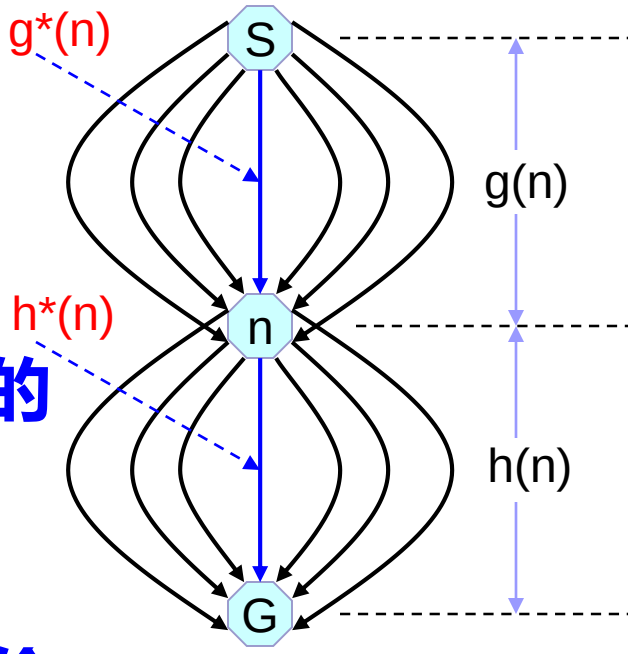
OPEN表	当前扩展	CLOSED表
a(0)	a	a
ab(75),ac(118),ae(140)	b	a,b
ac(118),ae(140)	c	a,b,c
ae(140),acd(229)	e	a,b,c,e
acd(229),aeg(220),aef(239),aei(440)	g	a,b,c,e,g
acd(229), aef(239),aei(440),aegh(317)	d	a,b,c,e,g,d
aef(239),aei(440),aegh(317)	f	a,b,c,e,g,d,f
aei(440),aegh(317),aefi(450)	h	a,b,c,e,g,d,f,h
aei(440),aefi(450),aeghi(418)	i	a,b,c,e,g,d,f,h,I

3.5.4 A* 算法

- 对A算法中的评估函数加上一些限制，使为A*算法。

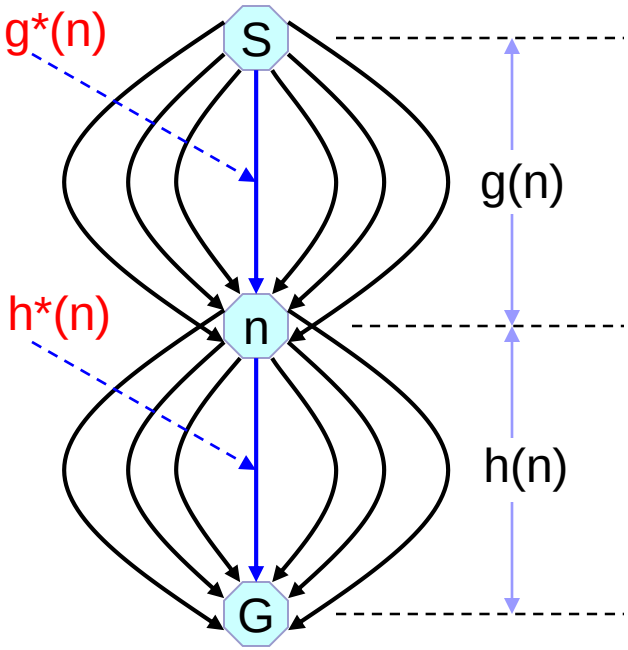
1.约束最小代价f*(n)

- f*(n)=g*(n)+h*(n)
- 为从初始状态S开始，约束经过结点n，到达目标结点G，所有解路径上的实际最小路径代价。
  - g*(n)为从S到n的实际最小代价
  - h*(n)为从n到G的估计的实际代价，
  - 若有多个目标结点， h*(n)取其中的最接近值。



2. A* 算法

- 当A算法的评估函数f(n)=g(n)+h(n)满足以下条件，即为A*算法。条件：
  - g(n)≥g*(n)，且g(n)>0
  - h(n)≤h*(n)
- 实际使用时，
  常使g(n)=g*(n)，
  即从S到n的实际最小代价。



- 3. A* 算法特点
  - 最优性
    - 保证最优解存在时能找到最优解。
  - 可采纳性
    - 算法能在有限步内终止，并能找到最优解。
- 4. A* 阅读资料
  - 一个写的很好的入门教程：http://www.policyalmanac.org/games/aStarTutorial.htm

3.5.5 f(n)=g(n)+h(n)讨论

1. **f(n)=g(n)，h(n)=0，g(n)为实际路径代价**
   - **等代价搜索 -- 效率不高，能得到最优解。**
2. **f(n)=h(n)，g(n)=0，**
   - **爬山搜索 -- 效率极高，往往陷入局部最优。**
3. **f(n)=g(n)+h(n)，g(n)>0，h(n)>0，**
   - **最好优先搜索 -- 但是设计估价函数时要充分考虑g(n)和h(n)的数量级要相当。否则当g(n)>>h(n)时，接近等代价搜索；h(n)>>g(n)时，接近贪婪搜索。**
4. **f(n)=g(n)+h(n)，g(n)>0，且$g(n) \geq g^*(n)$, $h(n) \leq h^*(n)$**
   - **A*算法 -- 为了兼顾搜索效率，在满足上述条件前提下，要尽可能取较大的h(n)值，尽可能提高搜索效率。**
5. **f(n)=0，即g(n)=0，h(n)=0，**
   - **盲目搜索。**

子曰：学而时习之，不亦说乎？有朋自远方来，不亦乐乎？人不知而不愠，不亦君子乎？

3.6 问题归约表示法

- Problem Reduction

1. 问题归约表示的基本思想

- 当一问题较复杂时，可通过分解或变换，将其转化为一系列较简单的子问题，然后通过对这些子问题的求解来实现对原问题的求解。
- 简而言之：先化简，再解决。

2. 分解

- 如果一个问题 P 可以归约为一组子问题 $P_1, P_2, \ldots, P_n$，当且仅当所有子问题 P_i 都有解时原问题 P 才有解；
- 任何一个子问题 P_i 无解都会导致原问题 P无解。
- 则称此种归约为问题的分解。
- 即分解所得到的子问题的“与”与原问题 P等价。

3.6 问题归约表示法

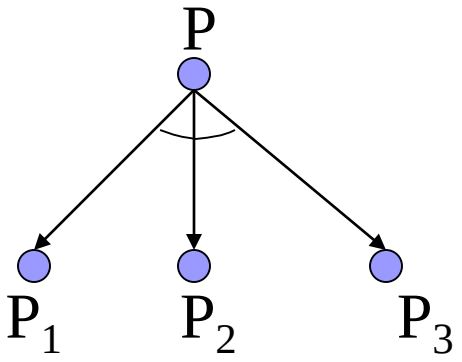
- **【例3.6.1】求方程所有的解$x^3-6x^2+11x-6=0$**
  - **解：**
  - **原问题可分解为3个一元一次方程：x-1=0, x-2=0, x-3=0；**
  - **这3个一元一次方程即为3个子问题；**
  - **同时（与关系）解了这3个方程，原问题解决。**

3.6 问题归约表示法

- **3. 等价变换**
  - 如果一个问题 P 可以归约为一组子问题 $P_1, P_2, \ldots, P_n$，并且子问题 P_i 中只要有一个有解，则原问题P就有解；
  - 只有当所有子问题 P_i 都无解时原问题 P 才无解。
  - 称此种归约为问题的等价变换，简称变换。
  - 即变换所得到的子问题的“或”与原问题 P 等价。
- 【例3.6.2】求方程一个解 $x^3-6x^2+11x-6=0$
  - 解：第一步可以等价变换为：$(x-1)(x^2-5x+6)=0$ 或 $(x-2)(x^2-4x+3)=0$ 或$(x-3)(x^2-3x+2)=0$ 三个子问题；其中任意一个可解，原问题解决。成“或”关系。

3.6 问题归约表示法

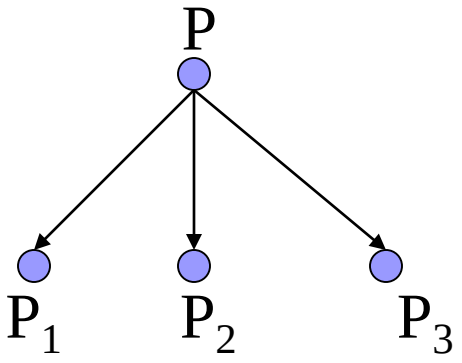
- **4. 问题归约的与或树表示**
  - **(1)与树 — 分解**
    - **原问题作父节点，分解的子问题作子节点，原问题的可解性和子问题可解性呈“与”的关系。**
    - **在图中加“连线”表示与关系。**
    - **如图： P_1, P_2, P_3 全部可解时P可解。**



与树

3.6 问题归约表示法

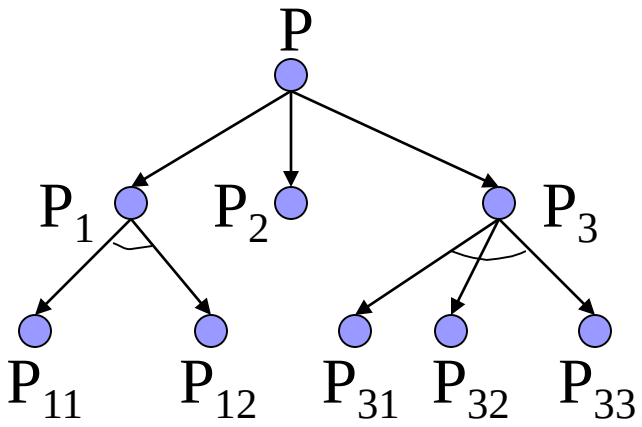
- **4. 问题归约的与或树表示**
  - **(2)或树 — 等价变换**
    - **原问题作父节点，变换的子问题作子节点，原问题的可解性和子问题可解性呈“或”的关系。**
    - **如图： P_1, P_2, P_3 中一个可解时P可解。**



或树

3.6 问题归约表示法

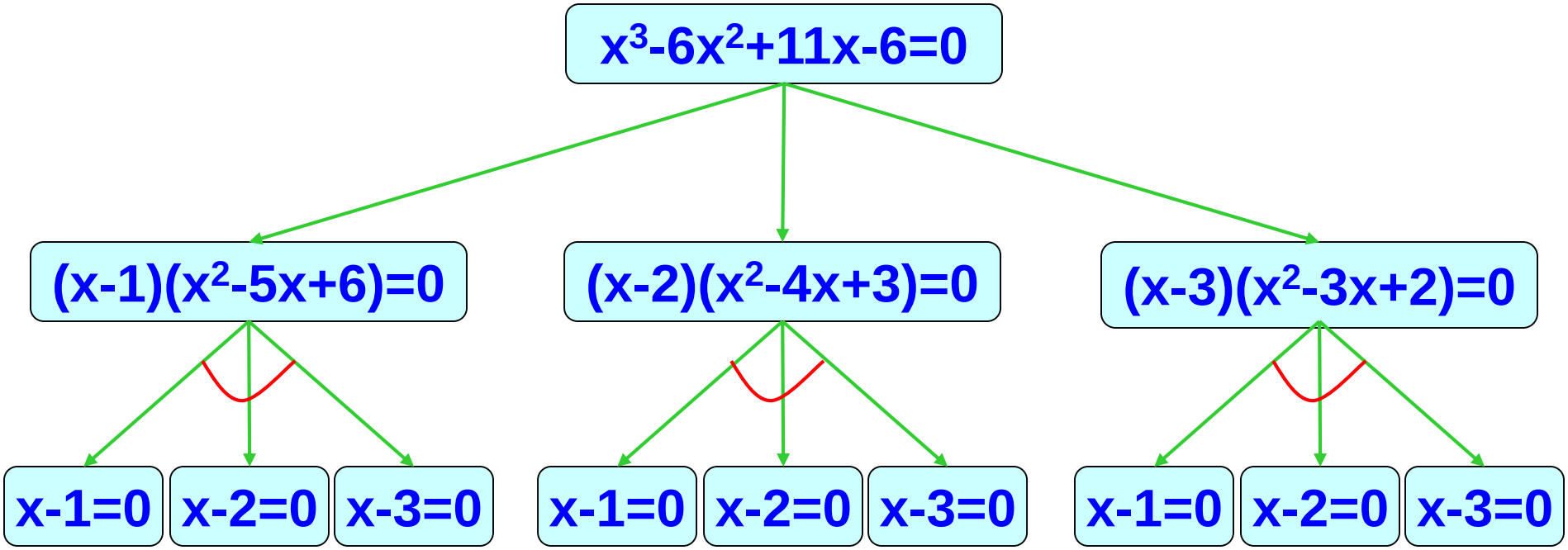
- **4. 问题归约的与或树表示**
  - **(3) 与或树**
    - **复杂问题归约时，既会用到“分解”，又会用到“变换”，用图表示，就是“与或树”，如图：**



与/或树

3.6 问题归约表示法

- **【例3.6.3 】 用与或树表示解方程 $x^3-6x^2+11x-6=0$的过程。**

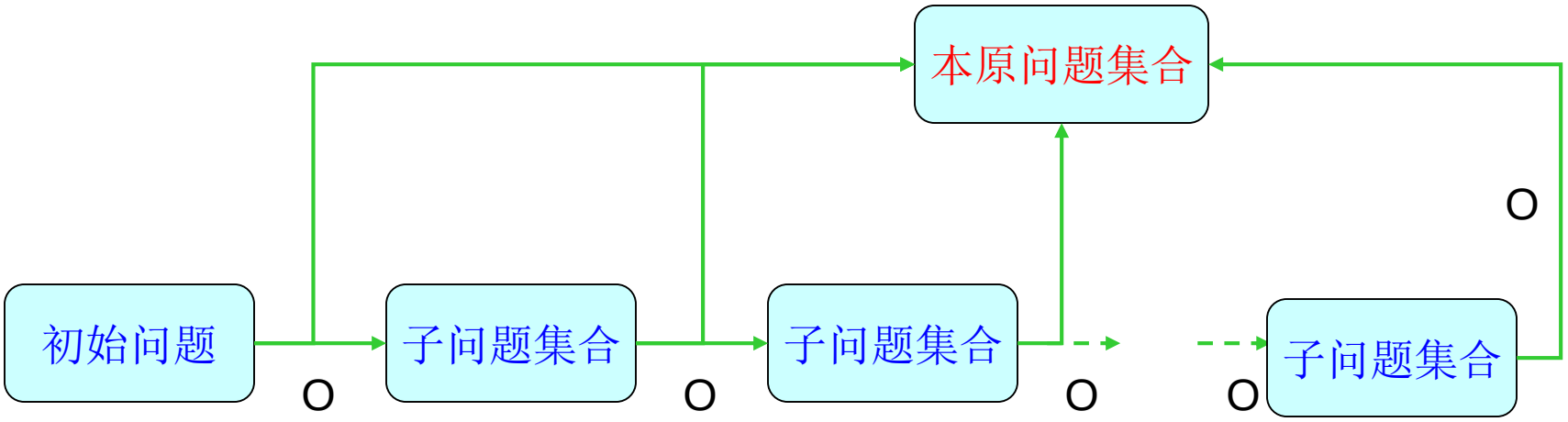


3.6 问题归约表示法

- **5. 本原问题**
  - **可以直接求出解的问题叫做本原问题。**
- **6. 问题归约的三元组描述**
  - **(1)初始问题——S**
    - **即要求解问题的数学描述。**
  - **(2)算子集合——O**
    - **通过操作算子将一个问题转换成若干个子问题。**
  - **(3)本原问题集合——P**
    - **其中的每一个问题都不需要再证明，自然成立的问题。如公理、已知事实、或已经证明过的问题。**
  - **三元组表示：（S，O，P）**
  - **也称问题归约3要素。**

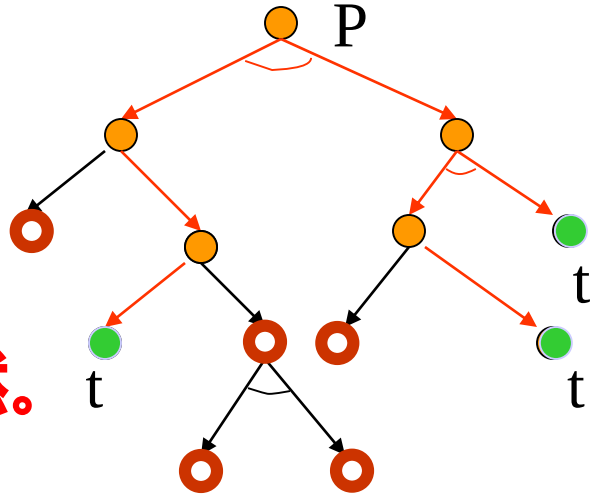
3.6 问题归约表示法

- **7. 归约目标**
  - **归约目标：产生本原问题。**
  - **问题归约过程：反复使用归约算子，产生子问题，直到所有问题都成为本原问题。过程如下图所示：**



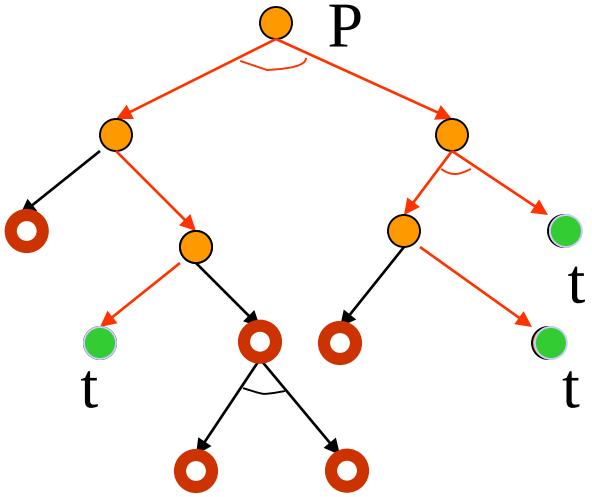
3.6 问题归约表示法

- **8. 端节点**
  - 没有子节点的节点，或叶子节点。
- **9.终止（终叶）节点**
  - 对应于本原问题描述的叶子节点。
  - 终止结点一定是端结点，反之不然。
- **10. 可解结点**
  - ①任何终止节点都是可解节点。
  - ②对有“或”子节点的节点，当其子节点中至少有一个为可解节点时，则该“或”节点就是可解节点。
  - ③对有“与”子节点的节点，只有当其子节点全部为可解节点时，该“与”节点才是可解节点。



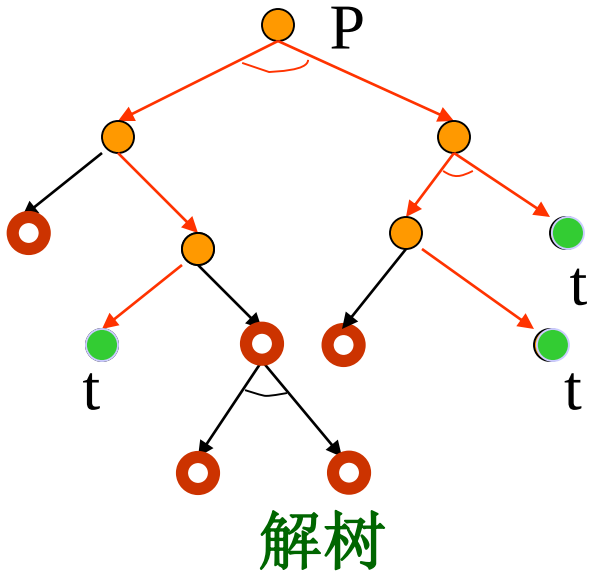
3.6 问题归约表示法

- **11. 不可解节点**
  - **①不为终止节点的端节点是不可解节点。**
  - **②对有“或”子节点的节点，若其全部子节点都为不可解节点，则该“或”节点是不可解节点。**
  - **③对有“与”子节点的节点，只要其子节点中有一个为不可解节点，则该“与”节点是不可解节点。**



3.6 问题归约表示法

- **12. 解树**
  - **由可解节点构成的，包含初始问题的树（生成树或搜索树）。**
  - **由此树可以证明初始问题可解。**
  - **例：右图中，P为初始问题，t为终止节点，红线部分构成解树。**



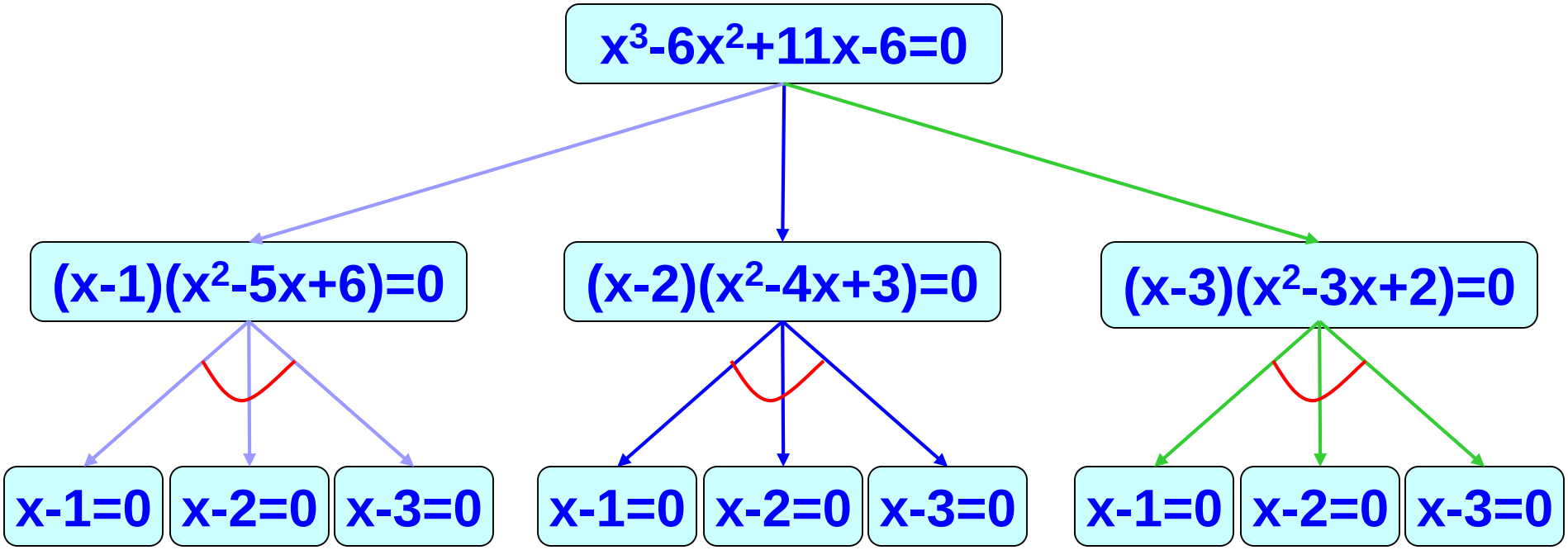
解树

问题归约求解过程实际上就是生成解树过程，即证明原始节点是可解节点的过程。

这一过程涉及到搜索的问题，对于与/或树的搜索将在后面详细讨论。

3.6 问题归约表示法

- **例3.6.4 方程 $x^3-6x^2+11x-6=0$的归约求解可有3棵解树。**



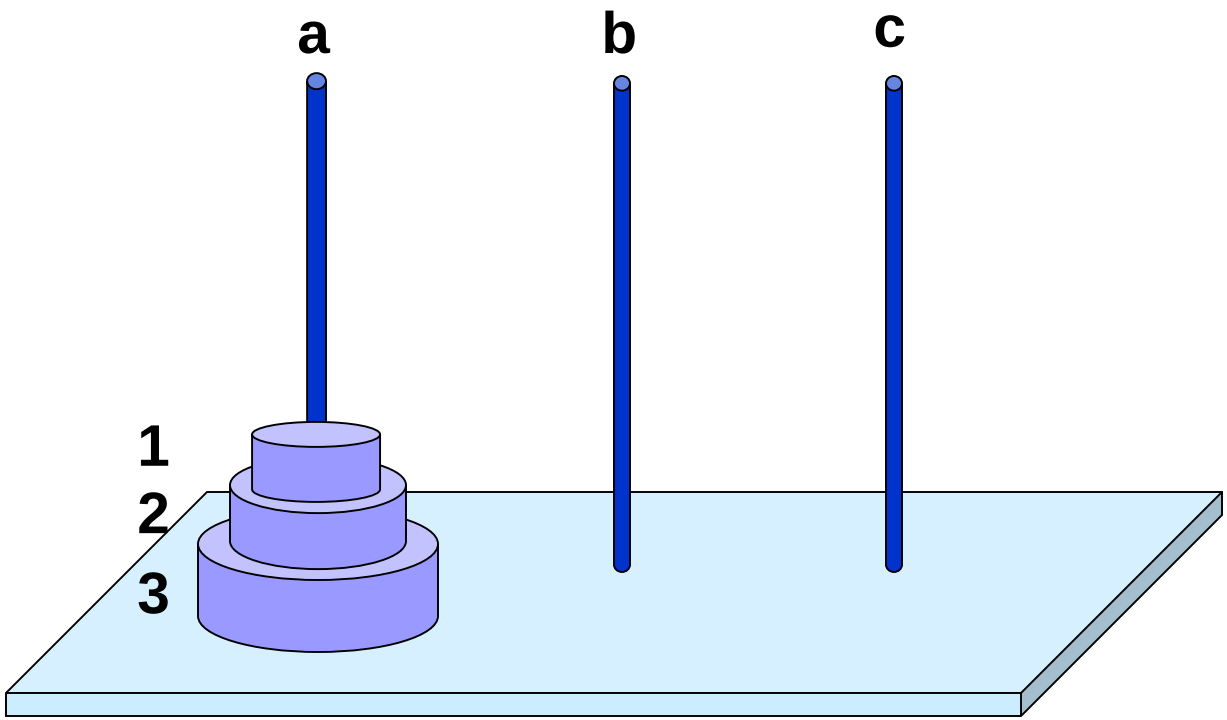
3.6 问题归约表示法

- **例3.6.5 梵塔问题（Tower of Hanoi problem）**
  - **a、b、c 3根柱子；**
  - **初始状态：在a上有3个大小不同的圆盘，1号最小，3号最大；**
  - **目标：把3个圆盘移到c上。**
  - **游戏规则：**
    - **一次只能移动一片圆盘；**
    - **任何时候大盘不能压在小盘上；**

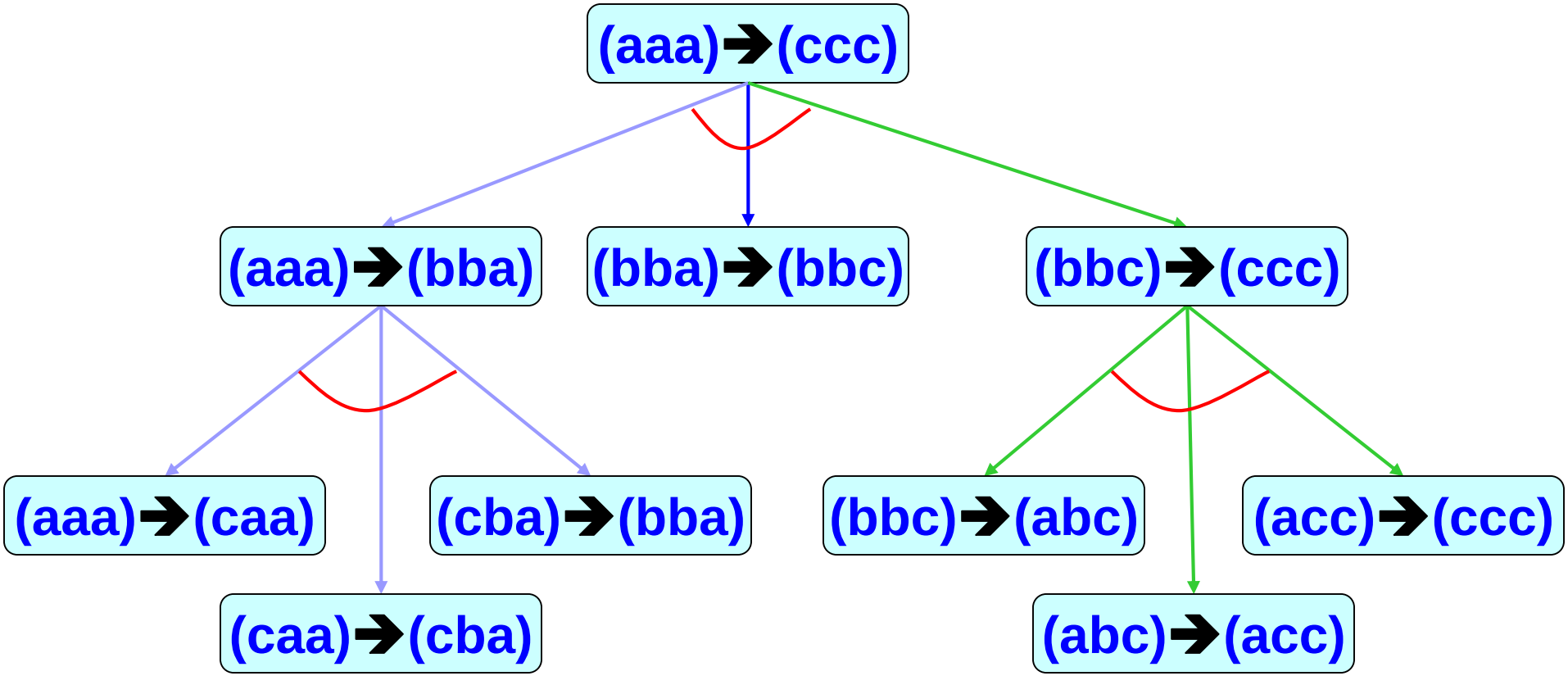
3.6 问题归约表示法

- **解：归约分析**
  - **要将所有的盘子都移动到c柱上，首先必须将3号盘移动到 c 柱，且移动之前 c 柱为空。**
  - **只有在移开 1、2 号盘后，才能移 3 号盘，且1、2盘最好不要放在 c 柱上，即1、2应移动到 b 柱子上。**
  - **完成上述移动后，关键一步是将 3 号盘从 a 柱移动到 c 柱，并继续上述过程，直至完成。**
  - **由上述分析可见，可以将三盘子的移动问题，简化为一个盘子（3 号盘），和两个盘子（1、2号盘）的移动问题，进一步分析，可将两个盘子的问题化也简为一个盘子的问题。**
  - **通过分析将初始问题归约为三个子问题：**
    - **Ⅰ. 移动1、2到 b 柱的两盘问题**
    - **Ⅱ. 移动 3 盘到 c 柱的一盘问题**
    - **Ⅲ. 移动1、2到 c 柱的两盘问题**
  - **解决了这三个子问题，则初始的三盘问题亦得到解决。**

3.6 问题归约表示法



- 梵塔问题分析
  - n个圆盘如何呢?
- 梵塔与或树表示
  - 用三元组(x,y,z)表示圆盘所在的柱子，盘号固定为1、2、3，
  - 初始状态：(aaa)—表示1、2、3号盘都在a柱上。
  - 目标状态：(ccc)—表示1、2、3号盘都在c柱上。
  - (cba)—表示1号盘在c柱，2号在b柱，3号盘在a柱。


(aaa)➔(ccc)
(aaa)➔(bba)
(bba)➔(bbc)
(bbc)➔(ccc)
(aaa)➔(caa)
(cba)➔(bba)
(bbc)➔(abc)
(acc)➔(ccc)
(caa)➔(cba)
(abc)➔(acc)

3.7 与或树的盲目搜索

- **与/或树的搜索过程即是一个不断寻找解树的过程。**

3.7.1 与或树的一般搜索过程

(1) 把原始问题作为初始节点S，并把它作为当前节点；

(2) 应用分解或等价变换操作对当前节点进行扩展；

(3) 为每个子节点设置指向父节点的指针；

(4) 选择合适的子节点作为当前节点，反复执行第(2)步和第(3)步，在此期间需要多次调用可解标记过程或不可解标记过程，直到初始节点被标记为可解节点或不可解节点为止。

- **上述搜索过程将形成一颗与/或树，这种由搜索过程所形成的与/或树称为搜索树。**

3.7.2 与/或树的广度优先搜索

- **与/或树的广度优先搜索与状态空间的广度优先搜索的主要差别是：需要在搜索过程中多次调用可解标识过程或不可解标识过程。搜索算法如下：**
- **(1)把初始节点S放入Open表中；**
- **(2)把Open表的第一个节点取出放入Closed表，并记该节点为n；**
- **(3)如果节点n可扩展，则做下列工作：**
  - **① 扩展节点n，将其子节点放入Open表的尾部，并为每一个子节点设置指向父节点的指针；**

3.7.2 与/或树的广度优先搜索

- ② 考察这些子节点中有否终止节点。若有，则标记这些终止节点为可解节点，并放入Closed表中；用可解标记过程对其父节点及先辈节点中的可解节点进行标记。如果初始节点S能够被标记为可解节点，就得到了解树，搜索成功，退出搜索过程；
- 否则，删去OPEN表中那些具有可解先辈的节点(因为其先辈节点已经可解, 故已无再考察该节点的必要),
- ③ 转第(2)步。

3.7.2 与/或树的广度优先搜索

- **(4) 如果节点n不可扩展，则作下列工作：**
  - **① 标记节点n为不可解节点；**
  - **② 应用不可解标记过程对节点n的先辈中不可解的节点进行标记。如果初始解节点S也被标记为不可解节点，则搜索失败，表明原始问题无解，退出搜索过程；**
  - **如果能确定S为不可解节点，则从Open表中删去具有不可解先辈的节点。(因为其先辈节点已不可解,故已无再考察这些节点的必要),**
  - **③ 转第(2)步。**

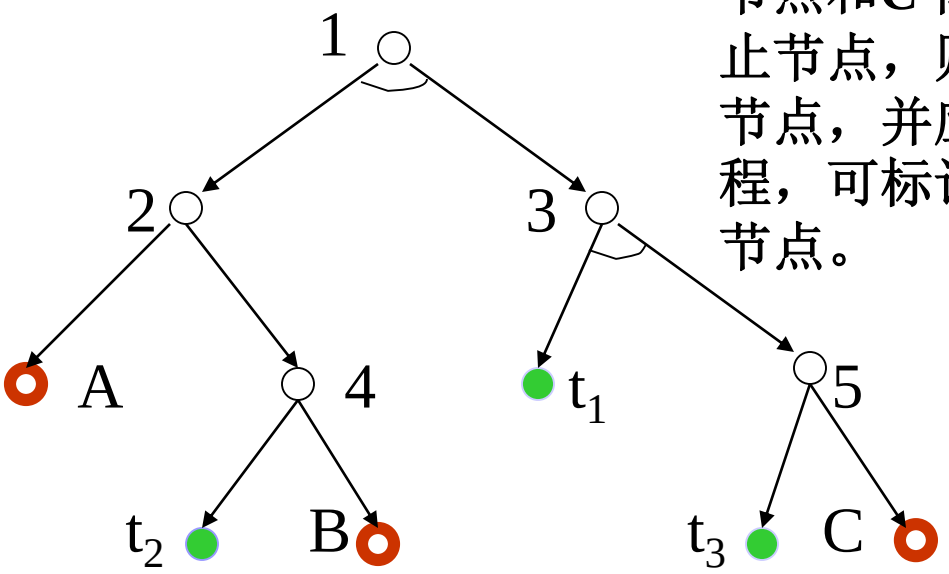
例3.7.1 设有下图所示的与/或树，节点按标注顺序进行扩展，其中标有t_1、t_2、t_3的节点是终止节点，A、B、C为不可解的端节点。

搜索过程为：

(1) 先扩展1号节点，生成2号节点和3号节点。

(2) 扩展2号节点，生成A节点和4号节点。

(3) 扩展3号节点，生成t_1节点和5号节点。由于t_1为终止节点，则标记它为可解节点，并应用可解标记过程，不能确定3号节点是否可解。



与/或树的广度优先搜索

(4) 扩展节点A，由于A是端节点，因此不可扩展。调用不可解标记过程。

(5) 扩展4号节点，生成t_2节点和B节点。由于t_2为终止节点，则标记它为可解节点，并应用可解标记过程，可标记2号节点为可解，但不能标记1号节点为可解。

(6) 扩展5号节点，生成t_3节点和C节点。由于t_3为终止节点，则标记它为可解节点，并应用可解标记过程，可标记1号节点为可解节点。

(7) 搜索成功，得到由1、2、3、4、5号节点及t_1、t_2、t_3节点构成的解树。

3.7.3 与/或树的深度优先搜索

- **与/或树的深度优先搜索和与/或树的广度优先搜索过程基本相同，其主要区别在于Open表中节点的排列顺序不同。在扩展节点时，与/或树的深度优先搜索过程总是把刚生成的节点放在Open表的首部。**
- **与/或树的深度优先搜索也可以带有深度限制dmax，其搜索算法如下：**
- **(1)把初始节点S放入Open表中；**
- **(2)把Open表第一个节点取出放入Closed表，并记该节点为n；**
- **(3)如果节点n的深度等于dmax，则转第(5)步的第①点；**
- **(4)如果节点n可扩展，则做下列工作：**
  - **① 扩展节点n，将其子节点放入Open表的首部，并为每一个子节点设置指向父节点的指针；**

3.7.3 与/或树的深度优先搜索

- ② 考察这些子节点中是否有终止节点。若有，则标记这些终止节点为可解节点，并用可解标记过程对其父节点及先辈节点中的可解解节点进行标记。如果初始解节点S能够被标记为可解节点，就得到了解树，搜索成功，退出搜索过程；
- 如果不能确定S为可解节点，则从Open表中删去具有可解先辈的节点。(因为其先辈节点已经可解, 故已无再考察该节点的必要).
- ③ 转第(2)步。

3.7.3 与/或树的深度优先搜索

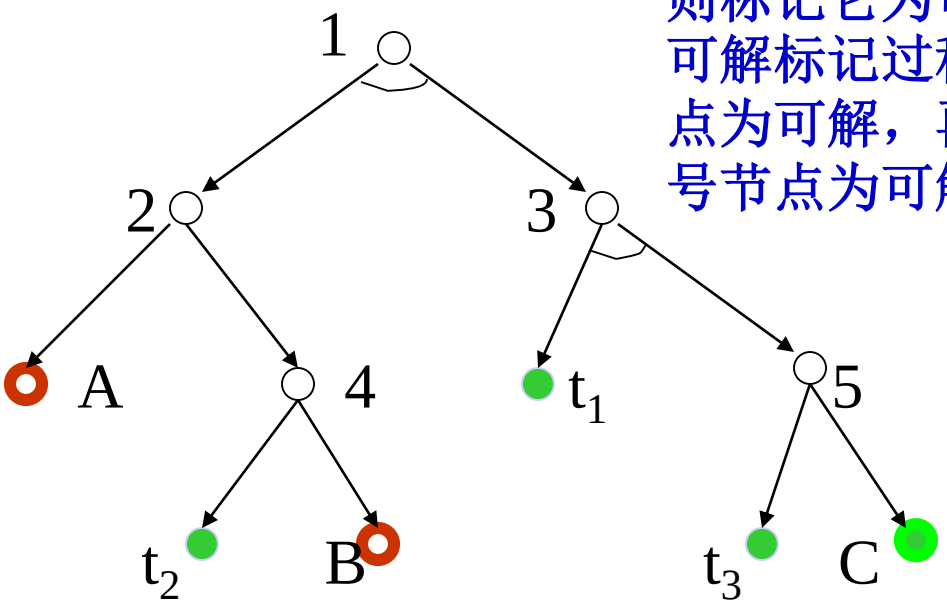
- **(5)如果节点n不可扩展，则作下列工作：**
  - **① 标记节点n为不可解节点；**
  - **② 应用不可解标记过程对节点n的先辈中不可解解的节点进行标记。如果初始解节点S也被标记为不可解节点，则搜索失败，表明原始问题无解，退出搜索过程；**
  - **如果不能确定S为不可解节点，则从Open表中删去具有不可解先辈的节点。 (因为其先辈节点已不可解, 故已无再考察这些节点的必要)**
  - **③ 转第(2)步。**

例3.7.2 对上例， 若按有界深度优先，且设dm=4，则其节点扩展顺序为：1，3，5，2，4。

搜索过程为：

(1) 先扩展1号节点，生成2号节点和3号节点。

(2)扩展3号节点，生成t_1节点和5号节点。由于t_1为终止节点，则标记它为可解节点，并应用可解标记过程，不能确定3号节点是否可解。



与/或树的有界深度优先搜索

(3)扩展5号节点，生成t_3节点和C节点。由于t_3为终止节点，则标记它为可解节点，并应用可解标记过程，可标记3号节点为可解节点，但不能标记1号为可解。

(4) 扩展2号节点，生成A节点和4号节点。

(5) 扩展4号节点，生成t2节点和B节点。由于t_2为终止节点，则标记它为可解节点，并应用可解标记过程，可标记2号节点为可解，再往上又可标记1号节点为可解。

(6) 搜索成功，得到由1、3、5、2、4号节点及t_1、t_2、t_3节点构成的解树。

3.8 与或树的启发式搜索

- **与/或树的启发式搜索过程实际上是一种利用搜索过程所得到的启发性信息寻找最优解树的过程。**
- **算法的每一步都试图找到一个最有希望成为最优解树的子树。**
- **最优解树是指代价最小的那棵解树。**
- **它涉及到解树的代价与希望树。**

3.8.1 解树的代价

- **解树的代价可按如下规则计算：**
- **(1)若n为终止节点，则其代价h(n)=0；**
- **(2)若n有“或”子节点，且子节点为n_1, n_2, … ,n_k，则n的代价为：**

$$h(n)=\min_{1\le i\le k}\{c(n,n_i)+h(n_i)\}$$

  - **其中，c(n, n_i)是节点n到其子节点n_i的边代价。**

3.8.1 解树的代价

- **(3)若n有“与”子节点，且子节点为$n_1, n_2, \ldots, n_k$，则n的代价可用和代价法或最大代价法。**
  - **若用和代价法，则其计算公式为：**
  - $$h(n)=\sum_{i=1}^{k}\{c(n,n_i)+h(n_i)\}$$
  - **若用最大代价法，则其计算公式为：**

$$h(n)=\max_{1\le i\le k}\{c(n,n_i)+h(n_i)\}$$

3.8.1 解树的代价

- **(4)若n是端节点，但又不是终止节点，则n不可扩展，其代价定义为h(n)=∞。**
- **(5)根节点的代价即为解树的代价。**

3.8.1 解树的代价

- **例3.8.1 设下图是一棵与/或树，它包括两可解树，左边的解树由S、A、t1、C及t3组成；右边的解树由S、B、t2、D及t4组成。在此与或树中，t1、t2、t3、t4为终止节点；E、F是不可解端节点；边上的数字是该边的代价。请计算解树的代价。**

解：先计算左边的解树

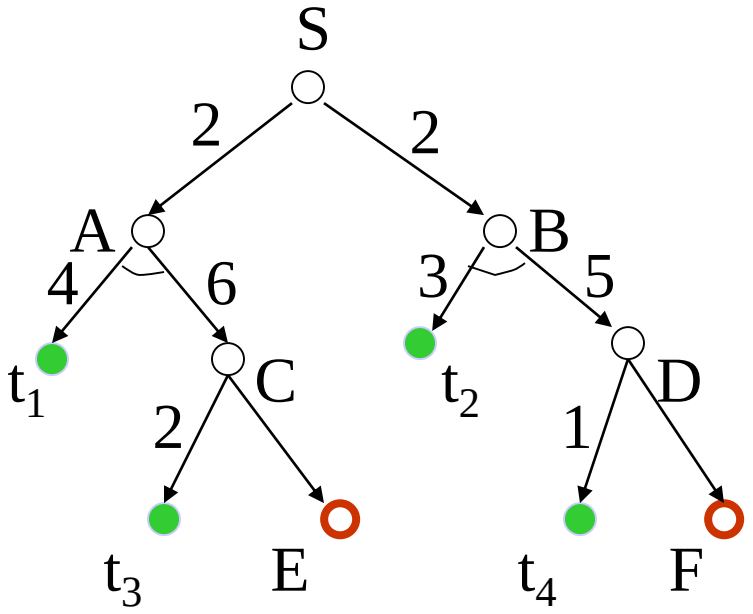
按和代价：h(S)=2+4+6+2=14

按最大代价：h(S)=(2+6)+2=10

再计算右边的解树

按和代价：h(S)=1+5+3+2=11

按最大代价：h(S)=(1+5)+2=8



与/或树的代价

3.8.2 希望树

- 希望树是指搜索过程中最有可能成为最优解树的那棵树。
- 与/或树的启发式搜索过程就是不断地选择、修正希望树的过程，在该过程中，希望树是不断变化的。

- 希望解树定义
  - (1) 初始节点S希望树T上；
  - (2) 如果n在希望树T上，n具有子节点$n_1, n_2, \ldots, n_k$，
    - ①若n为“或”节点，则n的某个子节点n_i在希望树T上的充分必要条件是：

$$\min_{1\le i\le k}\{c(n,n_i)+h(n_i)\}$$

    - ②若n是“与”节点，则n的全部子节点都在希望树T上。

3.8.3 与或树的启发式搜索

- **与/或树的启发式搜索过程如下：**

- **(1) 把初始节点S放入Open表中，计算h(S)；**
- **(2) 计算希望树T；**
- **(3) 依次在Open表中取出T的端节点放入Closed表，并记该节点为n；**

3.8.3 与或树的启发式搜索

- **(4)如果节点n为终止节点，则做下列工作：**
  - **① 标记节点n为可解节点；**
  - **② 在T上应用可解标记过程，对n的先辈节点中的所有可解解节点进行标记；**
  - **③ 如果初始解节点S能够被标记为可解节点，则T就是最优解树，成功退出；**
  - **④ 否则，从Open表中删去具有可解先辈的所有节点。**
  - **⑤ 转第(2)步。**

3.8.3 与或树的启发式搜索

- **(5) 如果节点n不是终止节点，但可扩展，则做下列工作：**
  - **① 扩展节点n，生成n的所有子节点；**
  - **② 把这些子节点都放入Open表中，并为每一个子节点设置指向父节点n的指针**
  - **③ 计算这些子节点及其先辈节点的h值；**
  - **④ 转第(2)步。**

3.8.3 与或树的启发式搜索

- **(6) 如果节点n不是终止节点，且不可扩展，则做下列工作：**
- **① 标记节点n为不可解节点；**
- **② 在T上应用不可解标记过程，对n的先辈节点中的所有不可解解节点进行标记；**
- **③ 如果初始解节点S能够被标记为不可解节点，则问题无解，失败退出；**
- **④ 否则，从Open表中删去具有不可解先辈的所有节点。**
- **⑤ 转第(2)步。**