# 人工智能

Artificial Intelligence

钟萍

http://faculty.csu.edu.cn/zhongping

# 课程内容

| 模块（64课时/32课时） |
| --- |
| 模块1：人工智能发展历史 |
| 模块2：知识表达与推理 |
| 模块3：搜索探寻与问题求解 |
| 模块4：机器学习/计算智能 |
| 模块5：深度学习 |
| 模块6：强化学习 |
| ~~模块7：人工智能博弈~~ |
| ~~模块8：人工智能伦理与安全~~ |
| ~~模块9：人工智能架构与系统~~ |
| 模块10：人工智能应用 |

# 课程内容

“101计划”第二批教材试点应用项目 (2024)

| 模块 | 大纲 |
| --- | --- |
| 模块1：人工智能发展历史 | 介绍基本概念、图灵机模型和图灵测试、人工智能主流算法（符号主义、连接主义和行为主义）、中外人工智能发展重要事件。 |
| 模块2：知识表达与推理 | 介绍知识表示方法、一阶谓词逻辑推[illegible]图谱推理和因果推理基础。 |
| 模块3：搜索探寻与问题求解 | 介绍贪婪最佳优先搜索、启发式搜索[illegible]法的性能分析、Minimax搜索、Alpha-Beta剪枝搜索和蒙特卡洛树搜索。 |
| 模块4：机器学习/计算智[illegible] | 介绍机器学习基本概[illegible]计算等。 |
| 模块5：深度学习 | [illegible]感知器模型、反[illegible] |
| 模块6：[illegible] | [illegible]强化学习中探索与利用的平衡、基于近似 |
| ~~模块7：~~[illegible] | |
| ~~模块8：~~[illegible] | |
| ~~模块9：人工智能架构与系统~~ | 介绍人工智能算法支撑技术链、人工智能芯片（GPU、XPU和类脑芯片等）和分布式深度学习优化等内容。 |
| 模块10：人工智能应用 | 介绍和实现自然语言中的机器翻译、视觉理解中的图像分类、机器人中的行为控制、科学计算以及语言大模型等具体例子。 |

**了解符号主义人工智能、连接主义人工智能和行为主义人工智能以及人工智能融合交叉等历史发展脉络**

**理解A*搜索、剪枝搜索和蒙特卡洛搜索之间的异同；掌握知识表达与推理、搜索探寻与问题求解等基本算法**

https://ebook.hep.com.cn/index.html#/detail?id=1146692759717937152&bookType=

来源：[illegible]《人工智能引论》

# 内容提要

讲授课时（6） + 实践学时（实验一 2+ 4）

# 知识表示（Knowledge Representation）

- 知识表示是问题求解的基础，是早期基于符号操作的经典人工智能研究的主要问题之一
  - GOFAI （Good Old-Fashioned Artificial Intelligence）philosophy:

    Problem Solving = Knowledge Representation + Search
- 知识表示
  - 把问题求解中所需要的对象、前提条件、算法等知识**构造为**计算机可处理的数据结构以及解释这种结构的过程

# 知识表示方法

不同类型的知识需要
不同的表示方式、
不同的表示方法需要
不同的求解技术

## ◆ 知识表示是问题求解的基础

- 问题求解是人工智能的核心问题之一
- 问题求解的目的
  - 机器自动找出某问题的正确解决策略
  - 更进一步，能够举一反三，具有解决同类问题的能力

## ◆ 智能系统中

- **知识**是对世界的描述（决定系统的能力）
- **表示**是知识的编码方式（决定系统的性能）

# 知识表示——Basic Methods

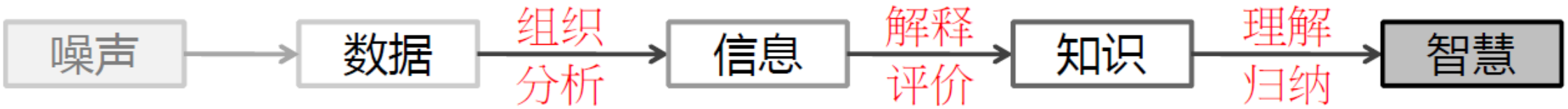
- State Space（状态空间）Representation
- Problem Reduction（问题归约）Representation
- Predicate Logic（谓词逻辑）
- Production Rules（产生式规则）
- Semantic Network （语义网络） Representation
- Frame, Script, Procedure（框架，剧本，过程）

# 知识表示方法

## ◆什么是知识？

- 数据、信息 ≠ 知识
- 知识层次


Q：具体案例？




- DATA：信息的载体和表示。用一组符号及其组合表示信息。
- INFORMATION：数据的语义。数据在特定场合下的具体含义。
- KNOWLEDGE：把有关信息关联在一起所形成的信息结构称为知识。事实&规则经过加工、整理后的信息.
- WISDOM：智慧

Q：所有的知识都是正确的？

# 知识表示方法

## ◆ 知识的属性

- 真伪性
- **相对性（相对正确性）**
- 不完全性
- **不确定性**
- **可表示性**
- 可存储性、可传递性和可处理性
- 相容性

# 知识表示方法

## ◆知识表示

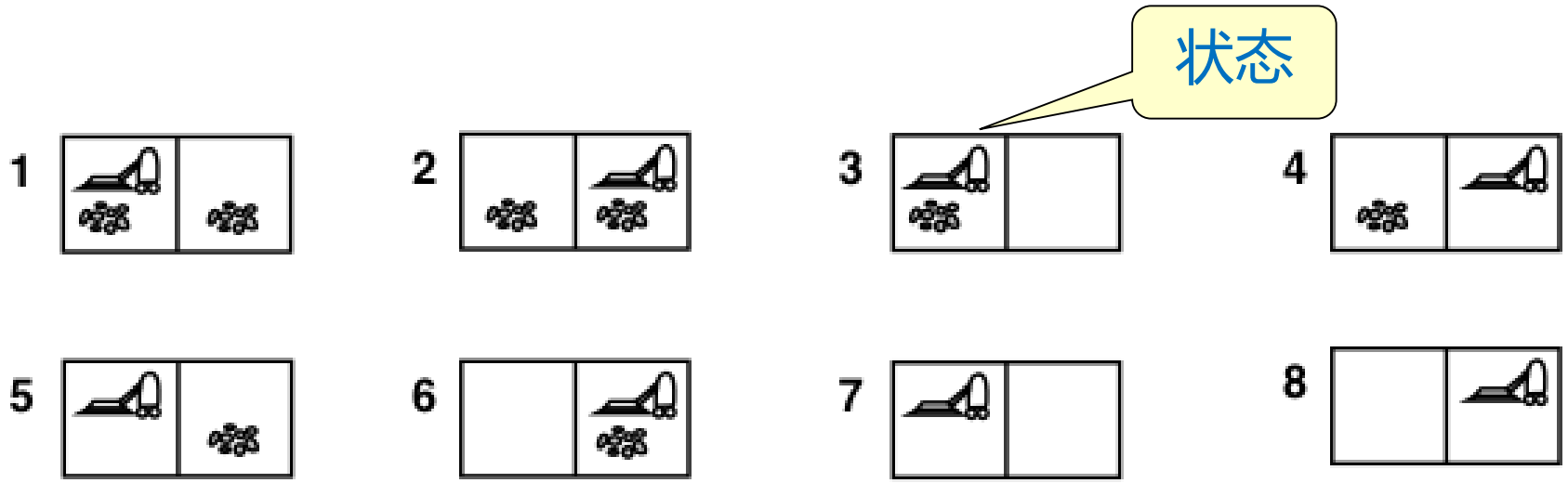
- 非形式化的自然语言描述 → 形式化的易于被计算机理解
- 符号学派的核心
  - 符号演算与机器推理
- 搜索技术是一种通用的问题求解技术

通常将待求解问题转化为某种可搜索的问题空间，然后在该空间中寻找解。

状态空间搜索技术使用“状态空间法”进行问题的符号表示，使用“搜索算法”进行推理求解。

# 真空吸尘器的世界

- 假设：吸尘器的世界只有两块地毯大小，地毯或者是脏的，或者是干净的
  - 吸尘器能做的动作只有三个
    {向左(*Left*)，向右(*Right*)，吸尘(*Suck*)}
  - 共有多少种可能的情况？
  - 怎样把环境打扫干净？
  - 跟上面讲过的几个例子有什么共同点？



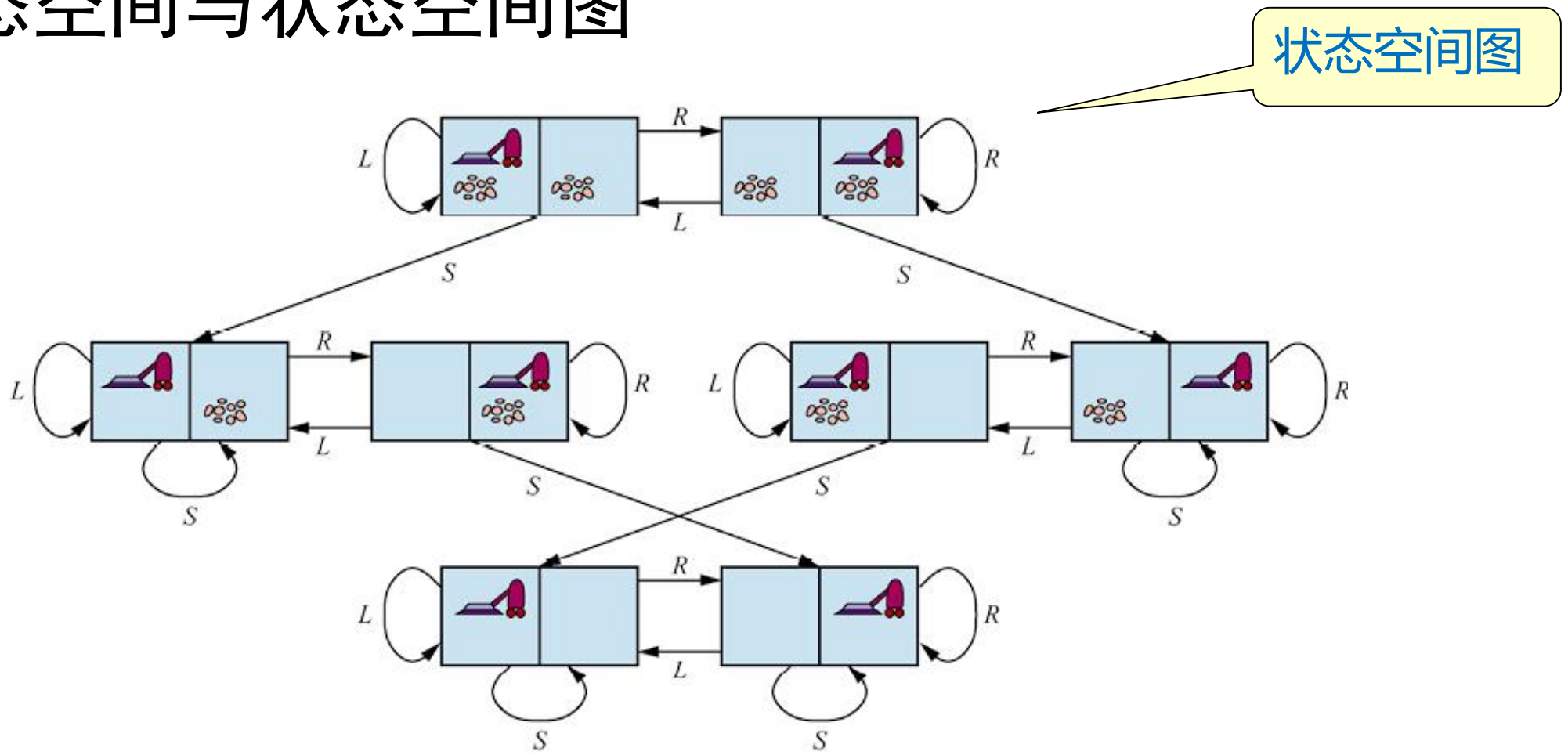
# 01 状态空间问题求解

(State Space)

# 问题求解

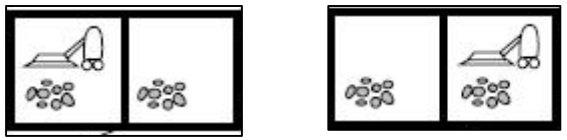
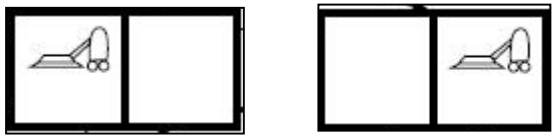
- 问题求解是人工智能的核心问题之一
- 问题求解的目的
  - 机器自动找出某问题的正确解决策略
  - 更进一步，能够举一反三，具有解决同类问题的能力
- 是从人工智能初期的智力难题、棋类游戏、简单数学定理证明等问题的研究中开始形成和发展起来的一大类技术
- 求解的手段多种多样 基本的有：搜索法、归约法、推理法、产生式等
- 其中搜索技术是问题求解的主要手段之一
  - 问题表示
  - 解的搜索

# 状态空间与状态空间图




- 状态空间图——包含了问题所有可能状态和转换关系的图
  - 图中的边是能够导致状态变化的操作（也称算子、算符）
  - 从初始状态到目标状态的路径对应的操作序列就是问题的解

# 状态空间与状态空间图

- 状态空间：一个包括问题所有可能状态及它们之间关系的图
  - 通常表示为三元组：{S, F, G}
  - S：起始状态集合
    - 例 
  - F：操作算子集合
    - 例 {向左(*Left*)，向右(*Right*)，吸尘(*Suck*)}
  - G：目标状态集合
    - 例 

# 状态空间与状态空间图

- 状态（state）
  - 在系统中决定系统状态的最小数目的变量的有序集合，对应图中的节点
  - 可以用向量表示
- 算子/动作
  - 引起状态发生变化的**操作**，对应图中的边
- 状态空间问题求解方法
  - 将实际问题转化成状态空间图来表示
  - 求解过程转化为在状态空间图中**搜索**一条从初始节点到目标节点的路径问题

# 状态空间与状态空间图

- 例：真空吸尘器世界
  - 状态（$q_1$,$q_2$,$q_3$）

    其中$q_1$表示左边地毯的状态，0表示干净，1表示肮脏

    其中$q_2$表示右边地毯的状态，0表示干净，1表示肮脏

    其中$q_3$表示吸尘器的位置，0表示在左边地毯，1表示在右边地毯

  - 算子

    向左(***Left***)：（$q_1$,$q_2$,$q_3$）=（$q_1$,$q_2$, 0）

    向右(***Right***)：（$q_1$,$q_2$,$q_3$）=（$q_1$,$q_2$,1）

    吸尘(***Suck***)}：（$q_1$, $q_2$, $q_3$）=（$q_1 \times q_3$， $q_2 \times (1-q_3)$， $q_3$）

  - 为提高求解效率，还可给算子加上执行的先决条件

# 练习1

- 请为真空吸尘器的世界问题设计节点的存储结构
  - –节点应能体现其所示的状态
  - –定义一个*Node*类实现这个节点结构
  - –定义类的成员函数*left*, *right*, *suck*来实现三种操作，并给操作加上限制条件
  - –编程实现以下功能
    - 接收输入的初始状态并显示
    - 执行用户输入的三类操作并显示结果

# 练习1——Coding

```
class CStateNode
{//真空吸尘器世界状态节点
    public:
        CStateNode();
        ~CStateNode();
        int m_lcs;//左边地毯状态0/1
        int m_rcs; //右边地毯状态0/1
        int m_vcp; //吸尘器位置0/1
        CStateNode(int lcs, int rcs, int vcp);//构造函数
        void MoveLeft();//向左操作
        void MoveRight(); //向右操作
        void Suck(); //吸尘操作};
```

# 思考题1：传教士野人问题（Missionaries& Cannibals, MC问题）

- 有三个传教士M和三个野人C过河，只有一条能装下两个人的船，在河的一方或者船上，如果野人的人数大于传教士的人数，那么传教士就会有危险，你能不能提出一种安全的渡河方法呢？
  - –状态表示？
    - 如何判断状态的合法性？
  - –算子？
    - 使用每个算子有怎样的先决条件，用后会导致状态怎样变化？
  - –画出状态空间图.
  - –该问题共有多少种不同的解?

# 思考题1解析：状态及其表示

状态可有多种表示方法：

**(左岸传教士数, 右岸传教士数, 左岸野人数, 右岸野人数, 船的位置)**

或

**(左岸传教士数, 左岸野人数, 船的位置)——（$M_L$，$C_L$，B）**

0：右岸
1：左岸

初始状态：(3, 3, 1)

目标状态：(0, 0, 0)

**合法状态：**

$0 <= M_L, C_L <= 3 \quad \& \quad (M_L = C_L \| M_L = 0 \| M_L = 3)$

# 思考题1解析：合法状态

合法状态：

$0 <= M_L, C_L <= 3$

$M_L=C_L \| M_L=0 \| M_L=3$

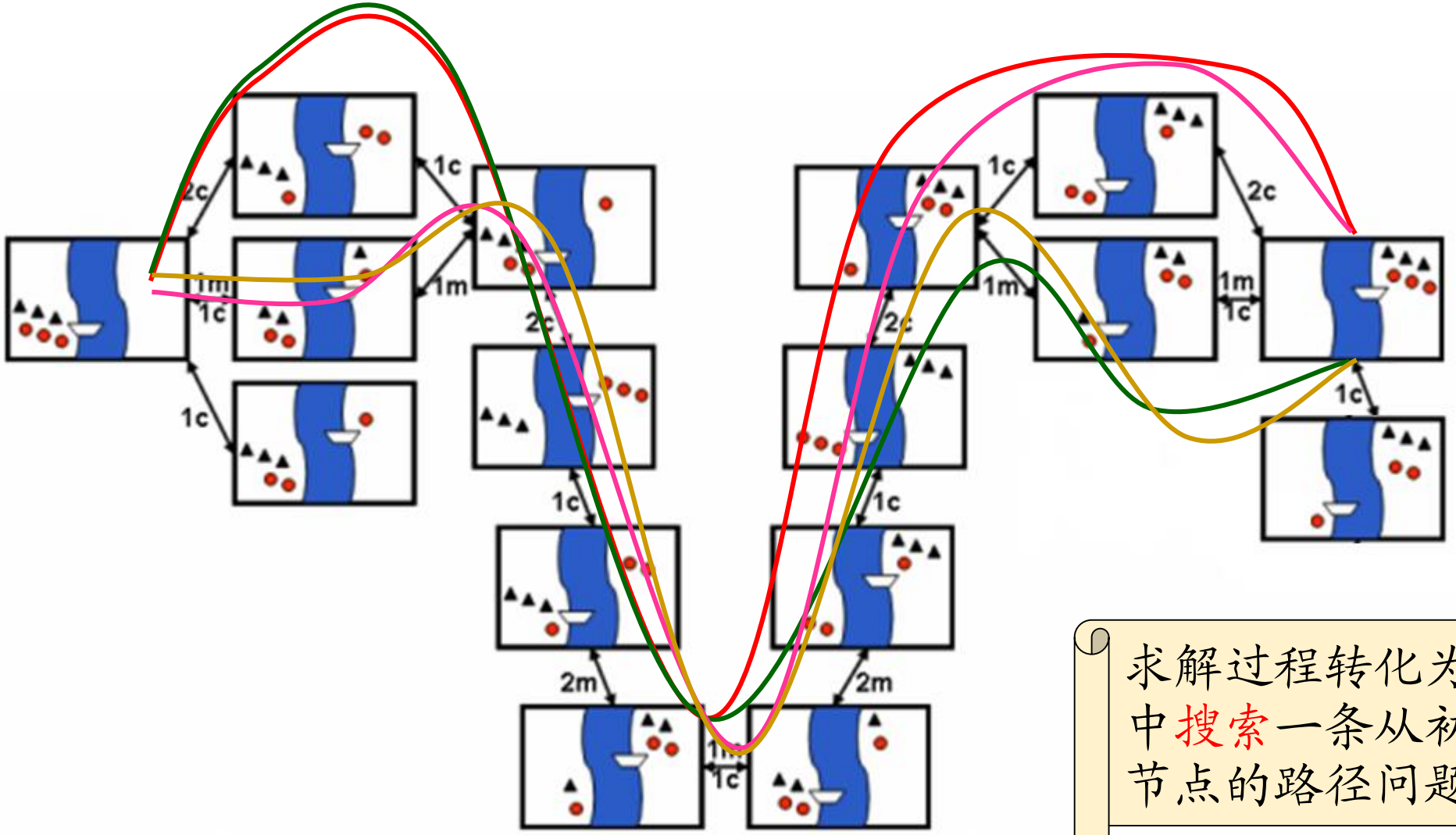
| $M_L=C_L$ | $M_L=0$ | $M_L=3$ |
|---|---|---|
| （0,0,0） | （0,0,0） | （3,0,0） |
| （1,1,0） | （0,1,0） | （3,1,0） |
| （2,2,0） | （0,2,0） | （3,2,0） |
| **（3,3,0）** | **（0,3,0）** | （3,3,0） |
| **（0,0,1）** | （0,0,1） | **（3,0,1）** |
| （1,1,1） | （0,1,1） | （3,1,1） |
| （2,2,1） | （0,2,1） | （3,2,1） |
| （3,3,1） | （0,3,1） | （3,3,1） |

# 思考题1解析：算子

- MC问题中的算子
  - –将传教士或野人运到河对岸
  - –**Move-1m1c-lr**：将一个传教士(m)一个野人(c)从左岸(l)运到右岸(r)
    - 执行后变化： $M_L=M_L-1$，$C_L=C_L-1$，$B=0$
    - 先决条件：执行操作后的状态合法
  - –所有可能操作

Move-1m1c-lr　Move-1m1c-rl　Move-2c-lr
Move-2c-rl　Move-2m-lr　Move-2m-rl
Move-1c-lr　Move-1c-rl　Move-1m-lr
Move-1m-rl

# 思考题1解析：状态空间图




求解过程转化为在状态空间图中搜索一条从初始节点到目标节点的路径问题

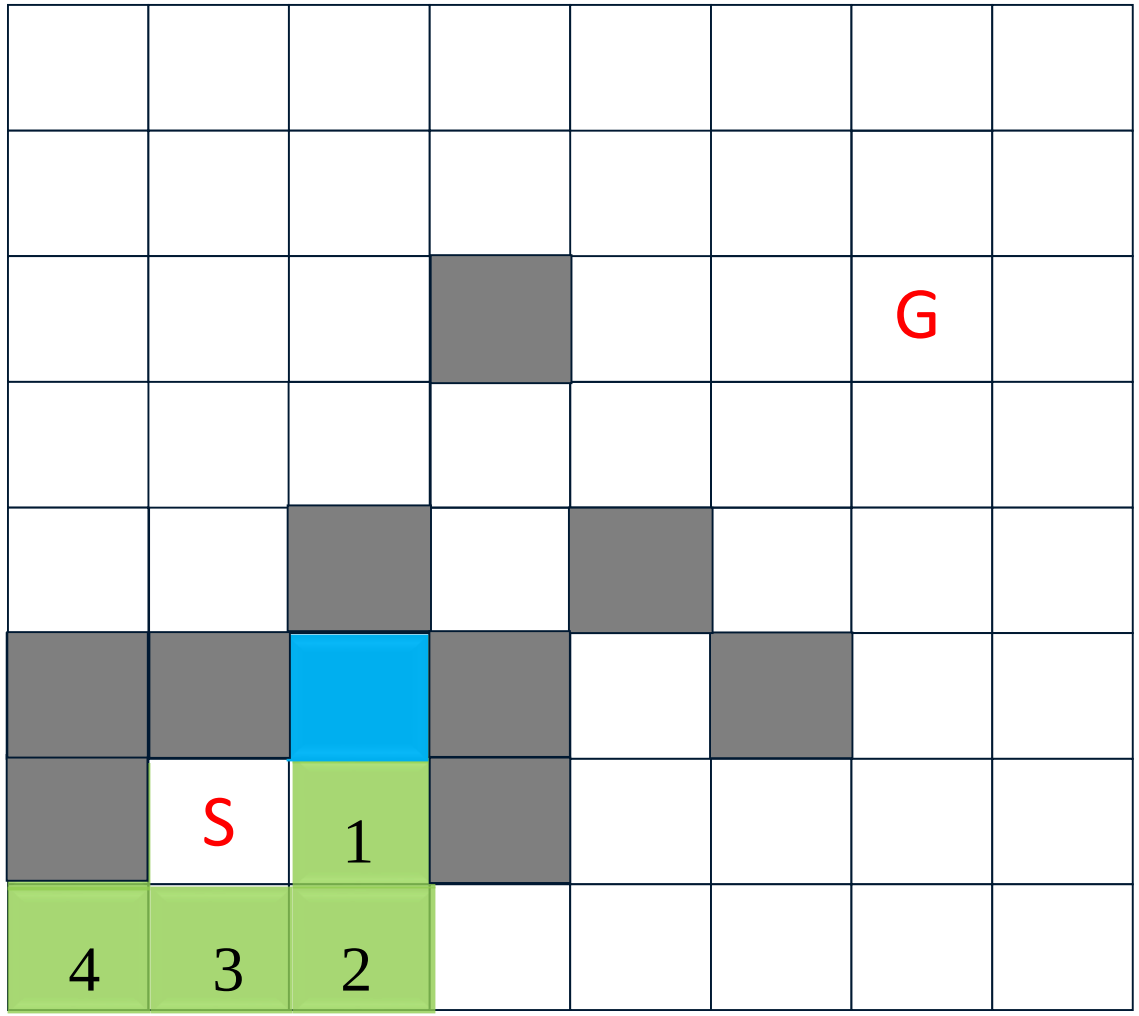
02

# 图搜索总览

(Graph Search)

# 图的搜索过程

- 有多个子节点的时候选哪个？
  - –搜索策略
- 如何避免重复搜索
  - –标记已访问过的点
- 如何回溯到其他候选点
  - –记录所有候选点



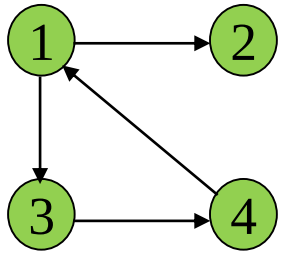
# 图结构

- 图由节点和边构成，可分为有向图和无向图
  - 图G记为G=(V,E)
  - V(G)是顶点的非空有限集
  - E(G)是边的有限集合，边是顶点的无序对或有序对
- 有向图(directed graphs)——有向图G是由两个集合V(G)和E(G)组成的
  - V(G)是顶点的非空有限集
  - E(G)是有向边（也称弧）的有限集合，弧是顶点的有序对，记为<v,w>，v,w是顶点，v为弧尾，w为弧头
- 无向图(undirected graphs)——无向图G是由两个集合V(G)和E(G)组成的
  - V(G)是顶点的非空有限集
  - E(G)是边的有限集合，边是顶点的无序对，记为（v,w）或（w,v)，并且（v,w)=(w,v)

# 图的存储结构

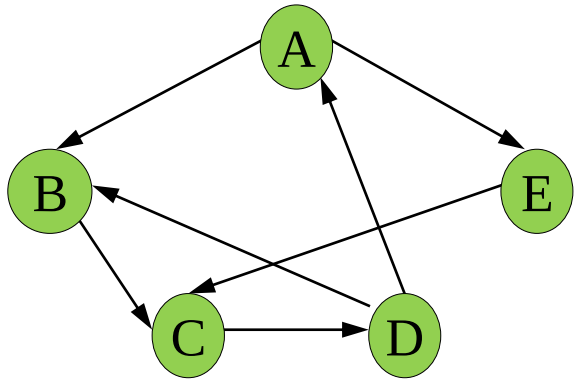
- 邻接矩阵
- 邻接表
- 十字链表
- …

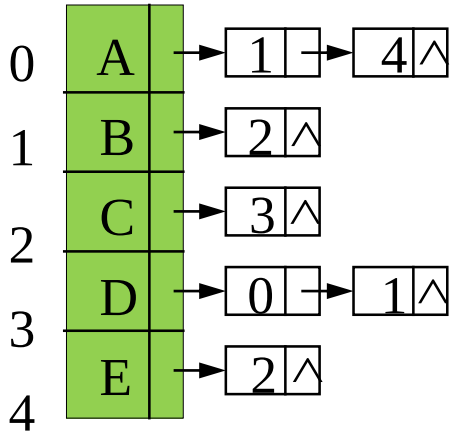
例




G1

| | ① | ② | ③ | ④ |
|---|---|---|---|---|
| ① | 0 | 1 | 1 | 0 |
| ② | 0 | 0 | 0 | 0 |
| ③ | 0 | 0 | 0 | 1 |
| ④ | 1 | 0 | 0 | 0 |



# 隐式图、显示图与搜索树

- 一般来说，状态空间图并不是事先创建好保存起来以供搜索使用，而是在搜索过程中边搜索边生成的
- 显示图
  - 完整的状态空间图，用前述的几种数据结构存储
- 隐式图
  - 仅给出初始结点、目标结点以及生成子结点的约束条件，要求按扩展规则来扩展结点，直到包含目标结点为止，不需要存储整个状态空间图
  - 实际搜索问题一般采用这种方法
- 搜索树
  - 在隐式图搜索过程中不断生长的树型结构，代表已经搜索过的状态及其之间的关系
  - 隐式图搜索只需要存储搜索树

# 图搜索中用到的数据结构

- 节点
  - 除了**存放状态本身的信息**，还需保存**指向父节点的指针**，或是何种操作可以转换为这个状态
- Open list
  - **记录候选节点**
  - 存放所有已经被生成了，但还未被扩展的节点 (open nodes)
  - 代表搜索树的前沿，也叫Fringe list
- Closed list
  - **标记已访问过的节点**
  - 存放所有已经被扩展的节点(closed nodes)

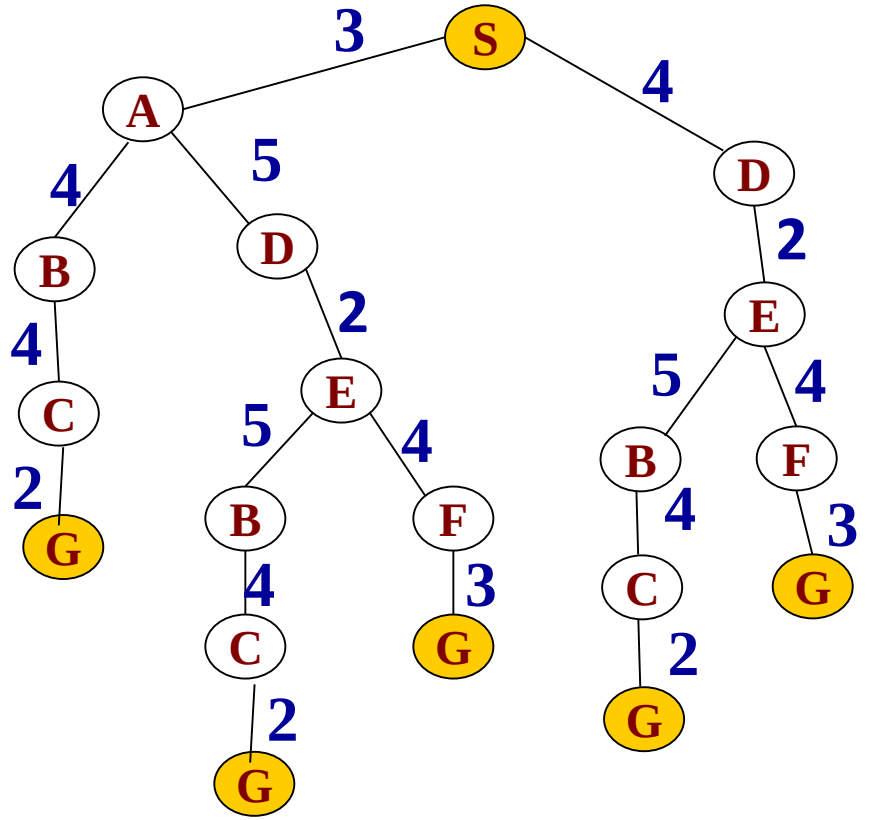
# 练习2

- 请为真空吸尘器的世界问题设计每个节点的存储结构
  - –每个节点应能体现其所示的状态
  - –应包含其父节点信息，或者说明其父节点是通过哪种操作到达该状态的
  - –定义一个类实现这个节点结构
  - –定义类的成员函数*left*, *right*, *suck*来实现三种操作，并给操作加上限制条件，每个操作应能生成一个新的状态节点
  - –编程实现以下功能
    - 根据输入生成初始状态
    - 执行操作后，显示后继状态，同时显示该状态的父节点

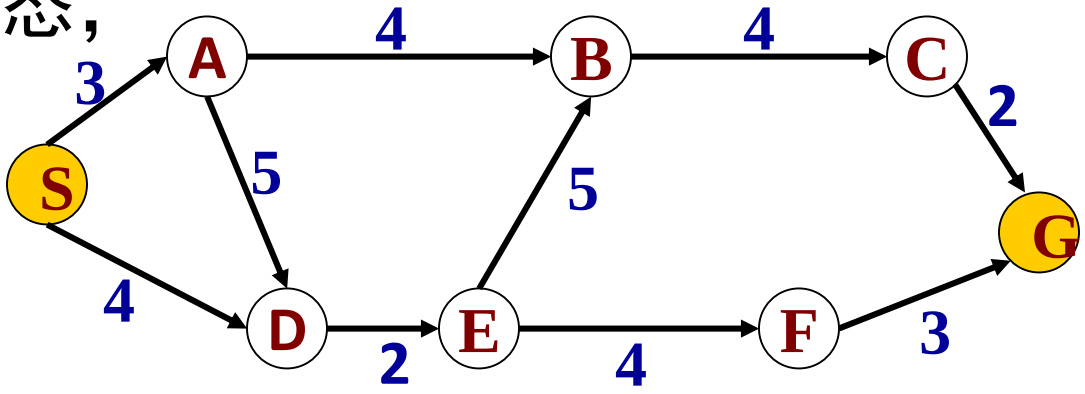
# 练习2——Coding

```
class CStateNode
{//真空吸尘器世界状态节点
    public:
        CStateNode();
        ~CStateNode();
        int m_lcs;//左边地毯状态0/1
        int m_rcs; //右边地毯状态0/1
        int m_vcp; //吸尘器位置0/1
        CStateNode * parent; //指向父节点的指针
        CStateNode(int lcs, int rcs, int vcp);//构造函数
        void MoveLeft();//向左操作
        void MoveRight(); //向右操作
        void Suck(); //吸尘操作};
```

# 图的一般搜索框架

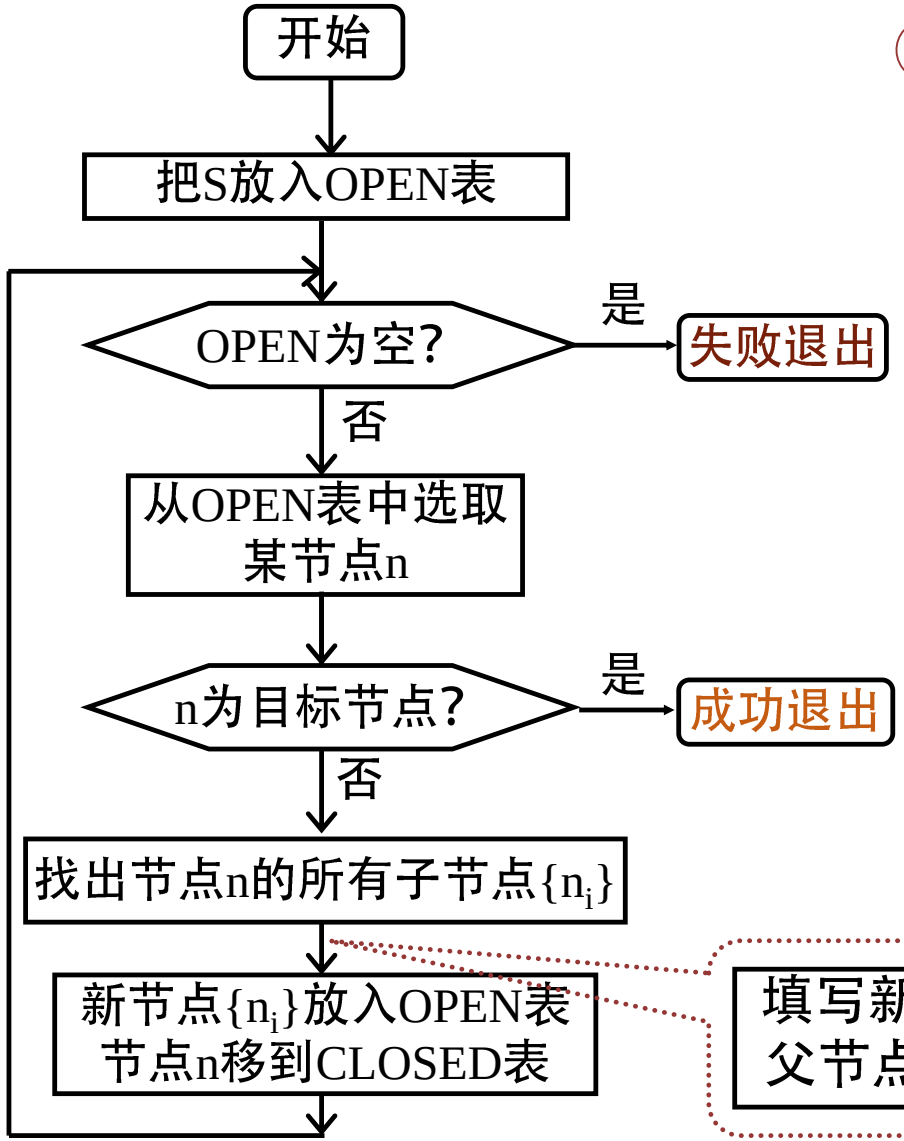



- 树搜索
  - –树——无圈连通图
  - –树搜索并非针对树的搜索，而是在搜索过程中将搜索的图看成是树，认为它没有重复状态
- 图搜索
  - –在搜索过程中会检查是否属于重复状态，避免状态循环的搜索
  - –图搜索大多比树搜索高效



# 图的一般搜索框架——搜索流程图

通用图搜索流程图




找到目标点后如何给出解路径?

初始化 *open* 表，将起始节点放入其中
Loop
　if *open* is empty return *failure*
　*Node*←remove-first (*open*)
　if *Node* is a *goal*
　　then return 从起始节点到*Node*的路径
　生成*Node* 的后继节点集合 *S*
　for all nodes *m* in *S*
　　将 *m* 以特定的顺序插入*open*
End Loop

搜索策略

# 搜索策略

- 无信息搜索（盲目搜索）
  - –宽度优先搜索
  - –深度优先搜索
  - –有界深度优搜索Depth-limited search
  - –等代价搜索Uniform-cost search
- 有信息搜索（启发式搜索）
  - –贪婪搜索
  - –A算法
  - –A*算法

# 03 盲目搜索 / 无信息搜索

# 宽度优先(BFS，Tree search)

- 搜索过程
  - 首先扩展根节点
  - 接着扩展根节点的所有后继节点
  - 然后再扩展后继节点的后继，依此类推
  - 在下一层任何节点扩展之前搜索树上的本层深度的所有节点都已经被扩展

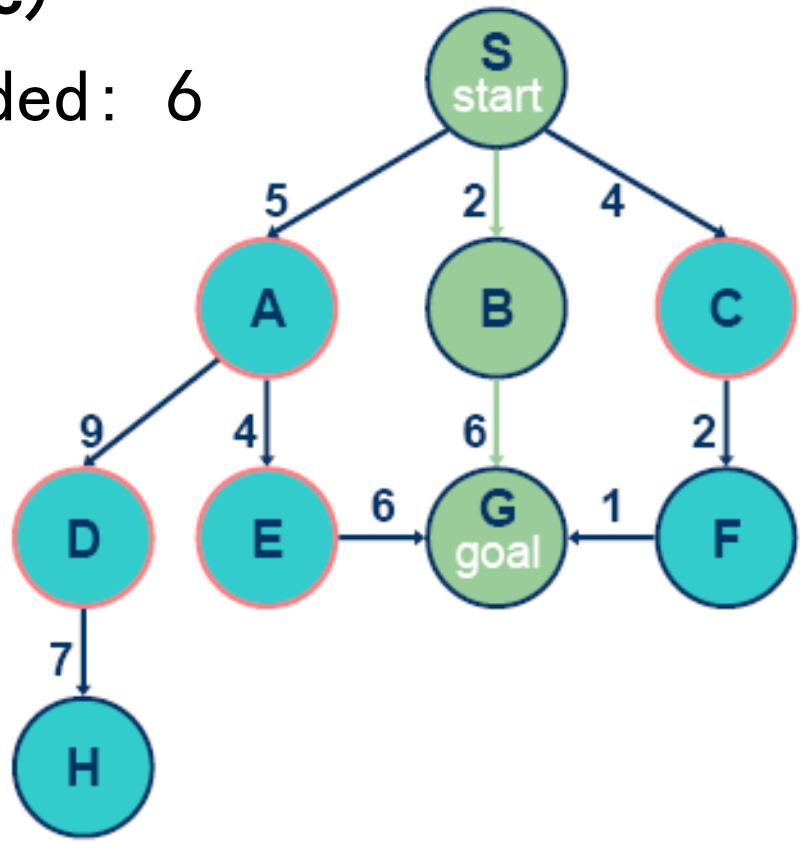
- 宽度优先搜索简单的将新扩展出来的节点顺序加入到OPEN表的后面
- OPEN表是一个FIFO的队列

# Example: BFS

**generalSearch(problem, Queue)**

# of nodes tested: 7, expanded: 6

| Expnd. node | Open list |
|---|---|
| | {S} |
| S | {A,B,C} |
| A | {B,C,D,E} |
| B | {C,D,E,G} |
| C | {D,E,G,F} |
| D | {E,G,F,H } |
| E | {G,F,H,G } |
| G | {F,H,G} |




Path: S,B,G
Cost:8

# 思考题2

- 在真空吸尘器的世界中，设初始状态为(1, 1, 0)，每个操作执行的先决条件按前面的定义，回答以下问题
  - 规定三个操作的优先级顺序为L, R, S，画出按宽度优先搜索生成的搜索树，列出每一步OPEN表的内容
  - 若操作优先级改为S, L, R，重新画出搜索树，列出每一步OPEN表的内容
  - 将前面两题的结果进行对比，两者有什么不同，哪种搜索效率更高？为什么？
  - 编程实现该问题的宽度优先搜索求解并输出最后的解（操作序列）

状态（q1,q2,q3）
其中q1表示左边地毯的状态，0表示干净，1表示肮脏
其中q2表示右边地毯的状态，0表示干净，1表示肮脏
其中q3表示吸尘器的位置，0表示在左边地毯，1表示在右边地毯

# 深度优先搜索(DFS, Tree search)

- 深度优先搜索过程：
  - 总是扩展搜索树的当前扩展分支(边缘)中最深的节点
  - 搜索直接伸展到搜索树的最深层，直到那里的节点没有后继节点
  - 那些没有后继节点的节点扩展完毕就从边缘中去掉
  - 然后搜索算法回退下一个还有未扩展后继节点的上层节点继续扩展

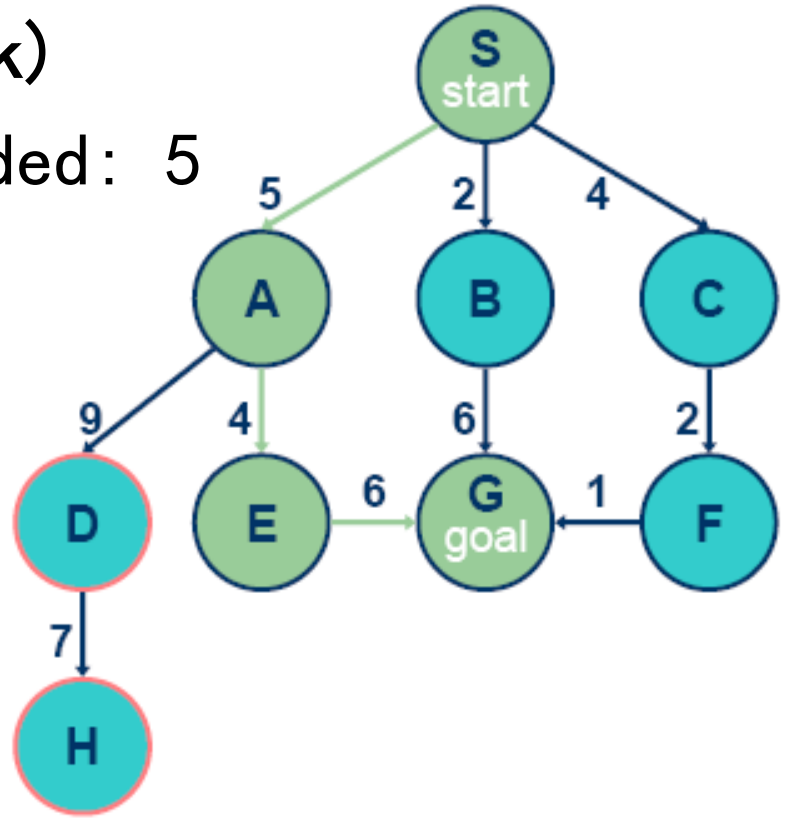
- 深度优先搜索将新扩展出来的节点顺序加入到OPEN表的前面
- OPEN表是一个LIFO的栈

# Example: DFS

**generalSearch(problem, Stack)**

# of nodes tested: 6, expanded: 5

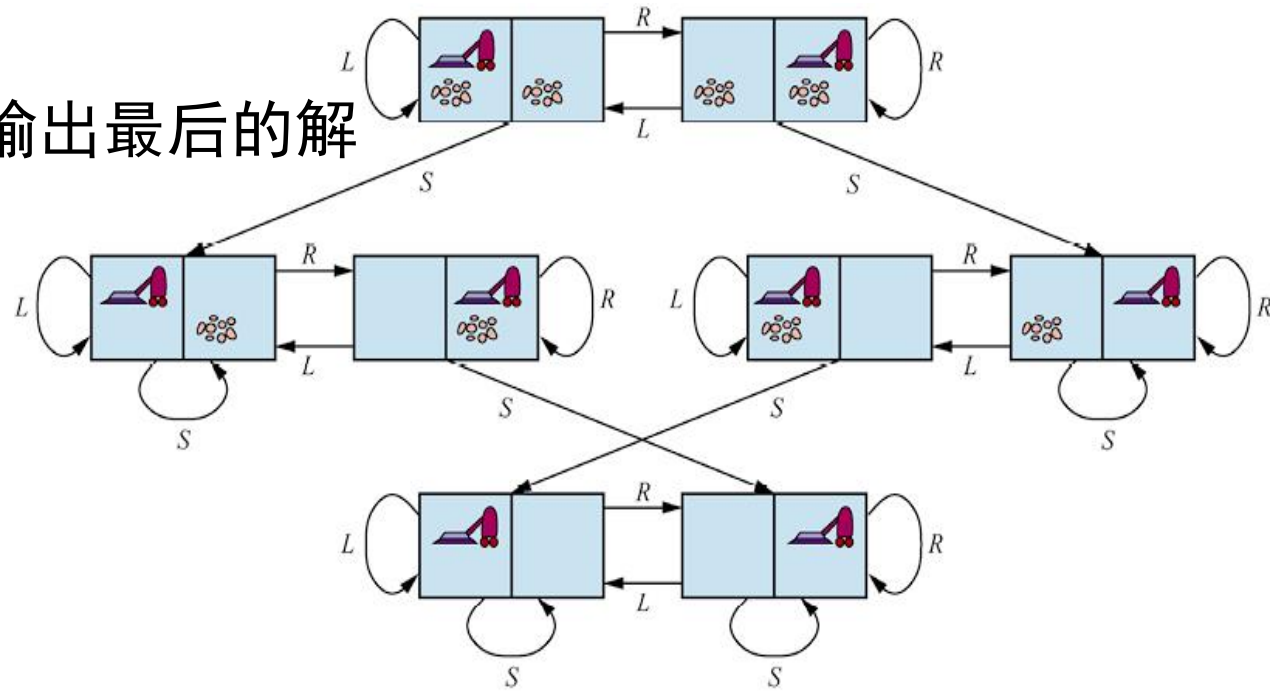
| Expnd. node | Open list |
|---|---|
| | {S} |
| S | {A,B,C} |
| A | {D,E,B,C} |
| D | {H,E,B,C} |
| H | {E,B,C} |
| E | {G, B,C } |
| G | {B,C } |




Path: S,A,E,G
Cost:15

# 思考题3

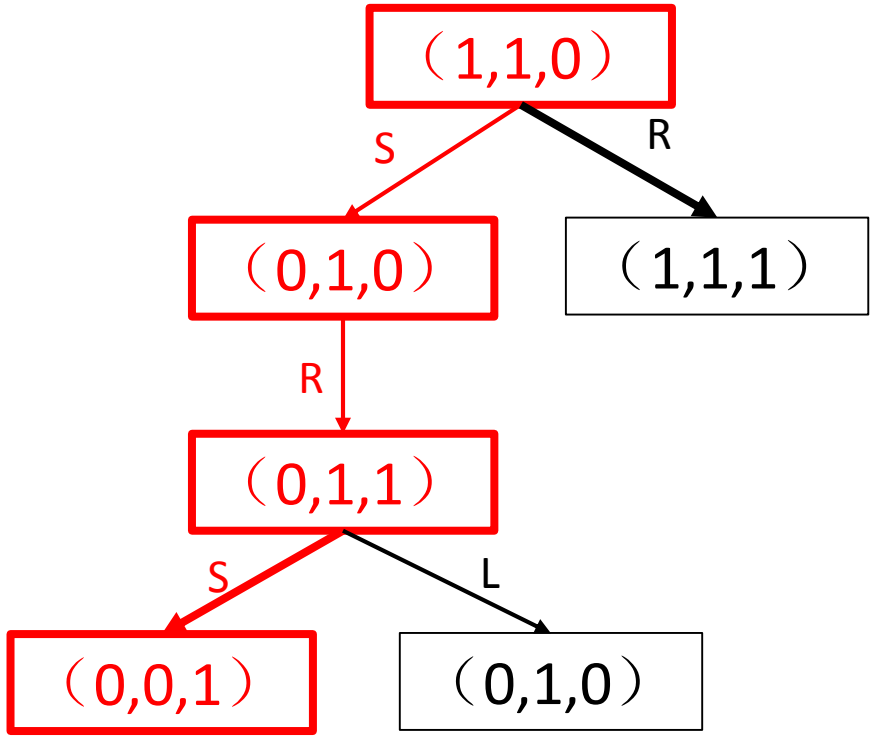
- 在真空吸尘器的世界中，设初始状态为(1, 1, 0)，每个操作执行的先决条件按前面的定义，回答以下问题
  - 规定三个操作的优先级顺序为L, R, S，试着采用深度优先搜索来解决问题，画一下搜索树，会发生什么问题，为什么会这样？列出前五步OPEN表的内容
  - 若操作优先级改为S, L, R，又会怎么样？
  - 和宽度优先相比，有什么优势和劣势？
  - 编程实现该问题的深度优先搜索求解并输出最后的解



# 思考题3解析

- 三个操作的优先级顺序为S, L, R ， 按深度优先顺序，OPEN表的变化如下
  - 扩展出6个节点
  - 做了4 次扩展操作

| 当前扩展节点 | OPEN |
|---|---|
| | {(1,1,0)} |
| (1,1,0) | {(0,1,0),(1,1,1)} |
| (0,1,0) | {(0,1,1),(1,1,1)} |
| (0,1,1) | {(0,0,1),(0,1,0),(1,1,1)} |
| (0,0,1)goal | |



# BFS vs. DFS

- 在图的一般搜索策略下，两者的算法流程相同
- 区别仅在**OPEN表中节点的排序不同**
  - 因此会选择不同的节点进行扩展
- BFS的OPEN表是一个队列，DFS是栈
- BFS的空间复杂度较大，DFS在这方面有优势
- 在假定每次扩展操作的代价都相同的情况下BFS能找到最优解，DFS则不具备最优性，且有时找不到解
  - BFS——如果每次操作代价不同会怎样？
  - DFS——如果目标节点所在的深度大大小于树的最大深度，最坏的情况会怎样？
    - 有界深度优先

# 等代价搜索(Uniform-cost search, UCS)

- BFS的扩展版本
  - 每次优先扩展最小代价的节点
- 1959年由Dijkstra提出，也叫Dijkstra算法
- 定义 $g(n)$ = 起始节点$s$ 到当前节点$n$的代价
- 修改：*open* =优先级队列(按$g$决定优先级)
- 当所有操作代价相等时UCS退化成BFS

```
function BREADTH-FIRST-SEARCH(problem) returns 一个解节点或 failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← 一个FIFO队列，其中一个元素为node
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s不在reached中 then
        将s添加到reached
        将child添加到frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns 一个解节点或failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

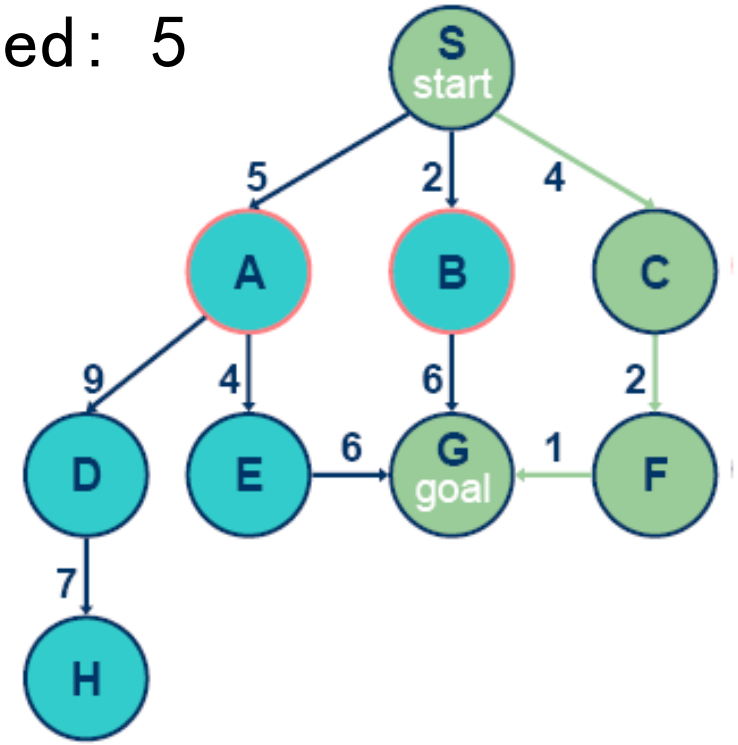
A priorty queue order by PATH-COST

# Example: UCS

**generalSearch(problem, priorityQueue)**

# of nodes tested: 6, expanded: 5

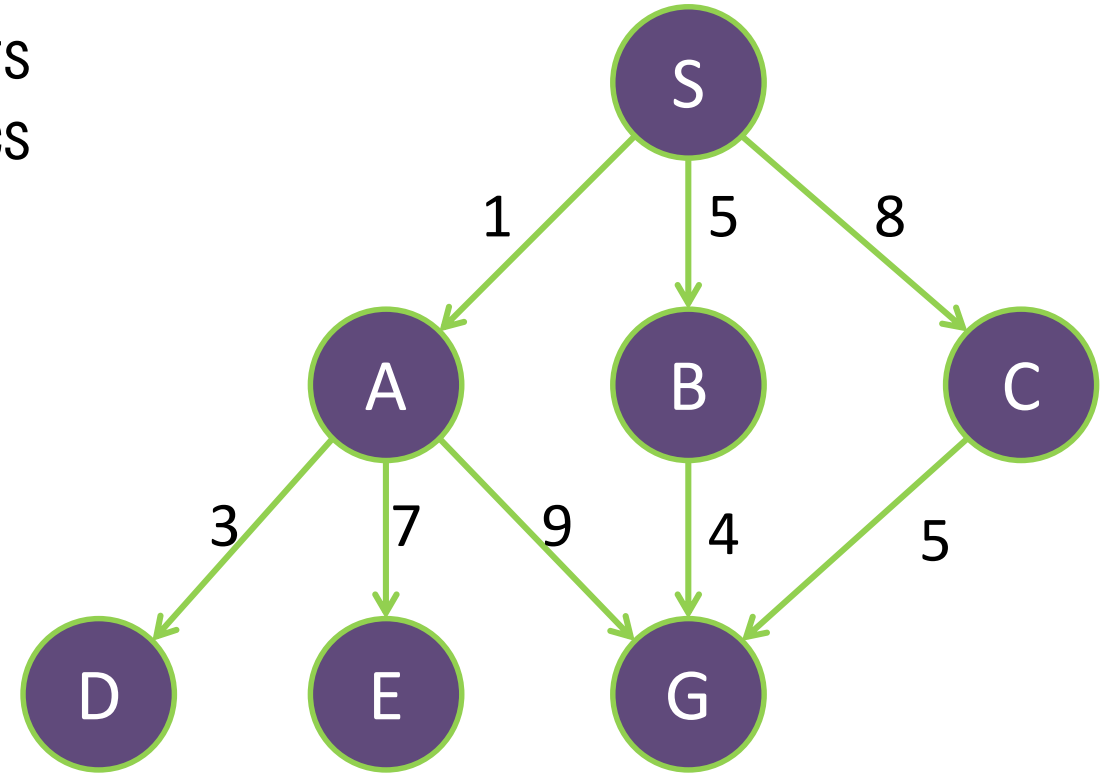
| Expnd. node | Open list |
|---|---|
| | {S:0} |
| S | {B:2,C:4,A:5} |
| B | {C:4,A:5,G:8} |
| C | {A:5,F:6,G:8} |
| A | {F:6,G:8,E:9,D:14} |
| F | {G:7,G:8,E:9,D:14} |
| G | {G:8,E:9,D:14} |




Path: S,C,F,G
Cost:7

# 练习3

- 若有如下状态空间，请写出在下列几种不同的搜索算法中节点的扩展顺序和最后的解路径
  - DFS
  - BFS
  - UCS



# 练习4

- 在吸尘器的世界中，设起始状态为（1,1,0），三个操作L,R,S的代价分别为2,2,1
  - 采用UCS搜索求解该问题，画出对应的搜索树
  - 列出每一步OPEN表的内容
  - 编程实现搜索求解过程

# 重复状态

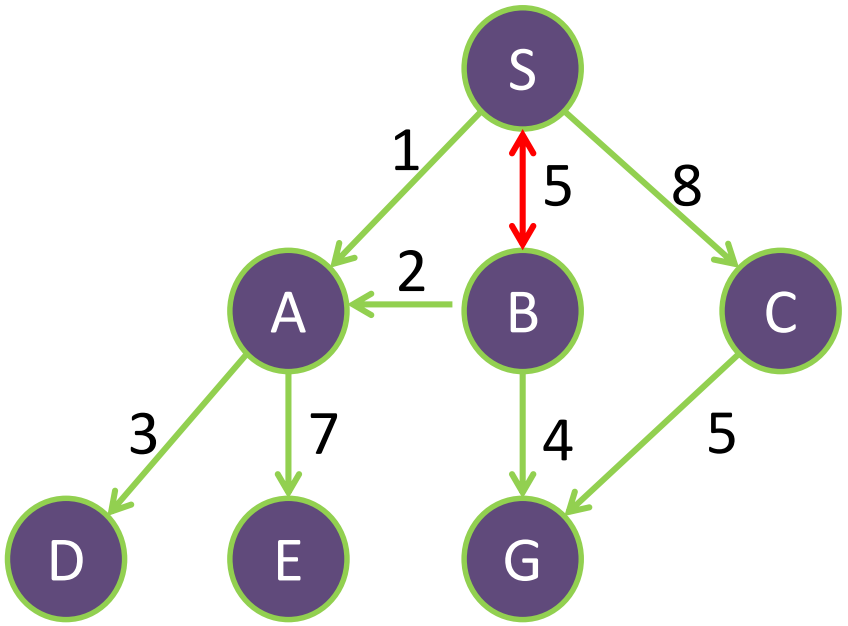
- 回顾吸尘器的世界，三个操作的优先级顺序为L,R,S
  - 宽度优先搜索
  - 深度优先搜索
- 在有重复状态的情况下
  - 宽度优先搜索能找到解，但浪费很多空间
  - 深度优先不一定能找到解
- 如何解决？
  - 加入CLOSED表记录重复状态

# 重复状态

- 加入CLOSED表来标记重复状态——图搜索

初始化 *open* 表，将起始节点放入其中
初始化 *closed* 表为空表
Loop
    if *open* 为空return *failure*
    *Node*←remove-first (*open*)
    if *Node* is a *goal*
        then return从起始节点到*Node*的路径
    **else 将*Node* 移入 *closed* 表**
    生成*Node* 的后继节点集合 *S*
    for all nodes *m* in *S*
        if *m* 不在*open* or *closed*
            将 *m* 以特定顺序插入 *open*
End Loop

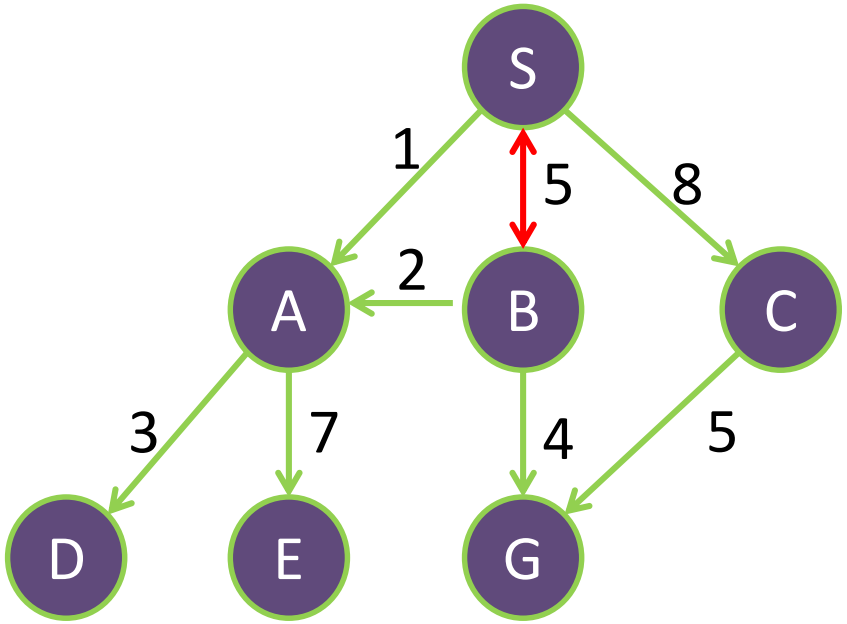
# 重复状态——例




宽度优先树搜索：

| Expnd. node | Open list |
|---|---|
| | {S} |
| S | {A,B,C} |
| A | {B,C,D,E} |
| B | {C,D,E,S,A,G} |
| C | {D,E,S,A,G,G} |
| D | {E,S,A,G,G} |
| E | {S,A,G,G} |
| S | {A,G,G,A,B,C} |
| A | {G,G,A,B,C,D,E} |
| G | {G,A,B,C,D,E} |

# 重复状态

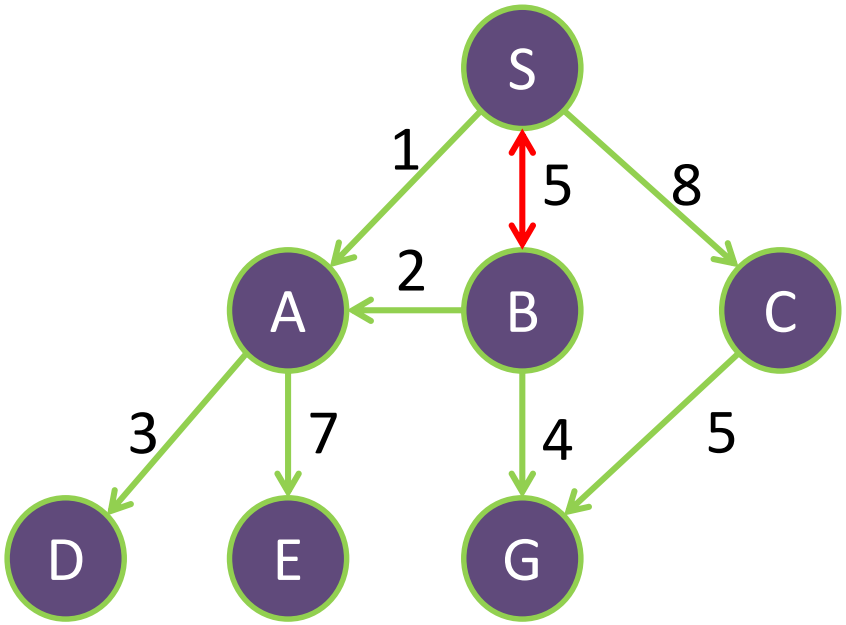



深度优先树搜索：

| Expnd. node | Open list |
|---|---|
| | {S} |
| S | {A,B,C} |
| A | {D,E,B,C} |
| D | {E,B,C} |
| E | {B,C} |
| B | {S,A,G,C} |
| S | {A,B,C,A,G,C} |
| A | {D,E,A,B,C,A,G,C} |
| D | {E,A,B,C,A,G,C} |
| E | {A,B,C,A,G,C} |
| … | … |

# 重复状态

- 图搜索




深度优先图搜索：

| Expnd. node | Open list | Closed list |
|---|---|---|
| | {S} | |
| S | {A,B,C} | {S} |
| A | {D,E,B,C} | {S,A} |
| D | {E,B,C} | {S,A,D} |
| E | {B,C} | {S,A,D,E} |
| B | {G,C} | {S,A,D,E,B} |
| G | {C} | {S,A,D,E,B} |

请自行使用宽度优先搜索求解

# 练习5

- 在真空吸尘器的世界中，设初始状态为(1,1,0)，每个操作执行的先决条件按前面的定义，规定三个操作的优先级顺序为L,R,S
  - 采用宽度优先图搜索求解，画出搜索树，列出每一步的OPEN表和CLOSED表以及最后的解
  - 采用深度优先图搜索求解，画出搜索树，列出每一步的OPEN表和CLOSED表以及最后的解
  - 编程实现上述两种搜索

# 04 启发式搜索 / 有信息搜索

# 最佳优先搜索（Best First Search）

- UCS是最佳优先搜索的特例
  - 使用优先级队列来存储扩展出的节点
- 最佳优先搜索是可以看成有信息搜索的一般框架
- 采用代价函数 $f(n)$ 对每一个扩展节点进行评价，该函数被称为**估价函数**(***evaluation function***)
- 节点在OPEN 中根据其$f$ 值来排序
  - $f$(n) 越小的节点$n$越早被扩展

# Best First Search (Tree Search)

初始化 *open* 使其包含起始节点

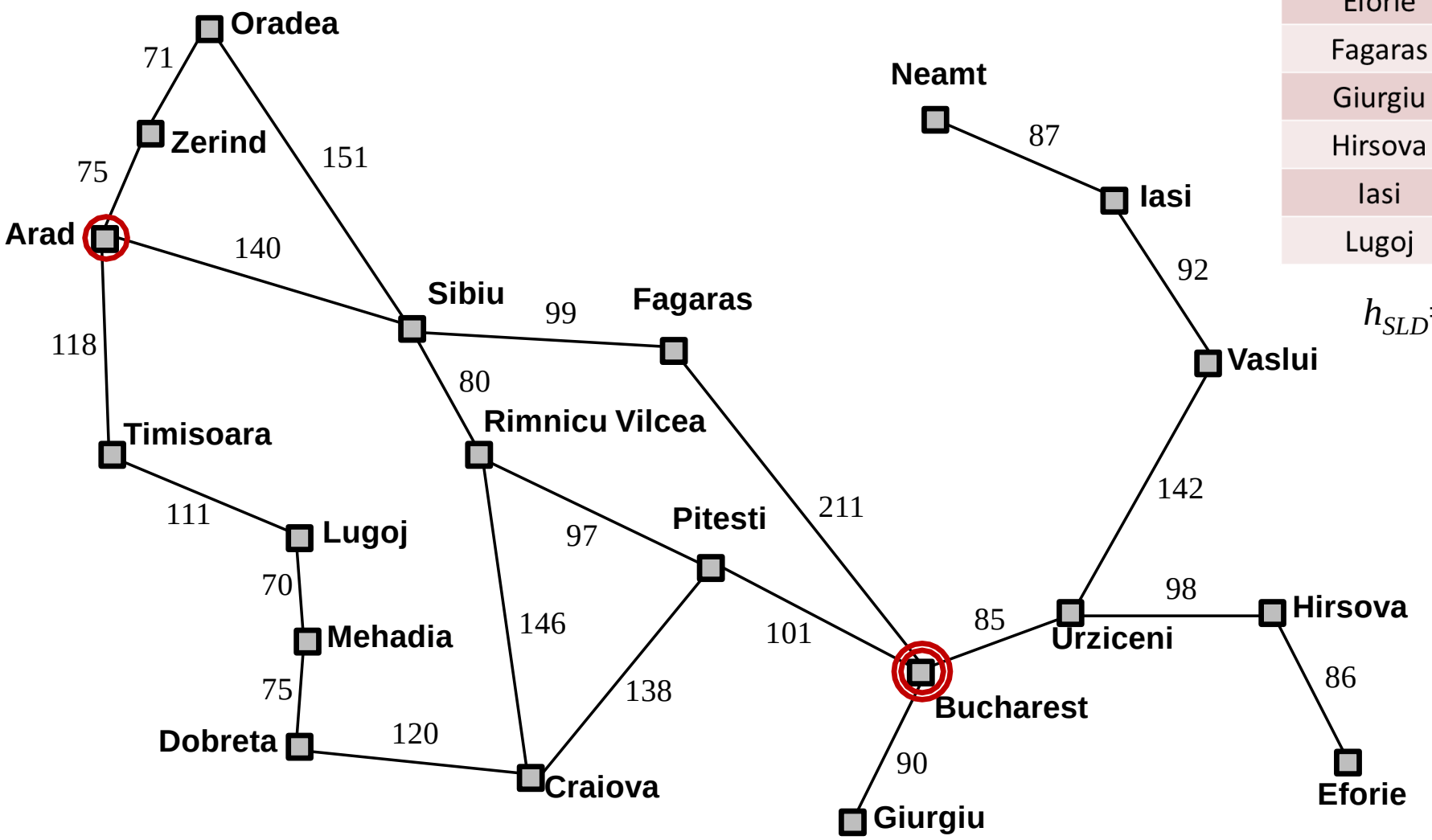
Loop
  if *open* is empty return failure
  *Node*←remove-first (*open*)
  if *Node* is a goal
    返回从起始节点到*Node* 的路径
  else {
    生成*Node*的所有后继节点
    将新生成的节点按照*f* 值从小到大的顺序加入OPEN表
    }
End Loop

*f* 的不同定义对应
不同的搜索策略

# 估价函数（Evaluation function）

- $f$ 的不同定义与算法
  - $f(n)=g(n)$——等代价搜索(UCS)
    - $g(n)$ = cost of the path from the start node to the current node $n$
  - $f(n)=h(n)$———贪婪优先搜索(Greedy best first)
    - $h(n)$——*heuristic function*（启发函数）
    - 使用了启发函数的搜索也称启发式搜索
    - $h(n)$的值是对当前状态$n$的一个估计，表示
      - 从$n$ 到目标节点的最优路径代价的估计
      - $h(n)\geq 0$，$h(n)$ 越小表示$n$越接近目标
      - If $n$ is goal then $h(n)=0$
      - 与问题相关的启发式信息都被计算为一定的$h(n)$值引入到搜索过程中
  - $f(n)=g(n)+h(n)$ ———A 算法

# 例: Romania




| City | SLD | City | SLD |
|---|---|---|---|
| Arad | 366 | Mehadia | 241 |
| Bucharest | 0 | Neamt | 234 |
| Craiova | 160 | Oradea | 380 |
| Dobreta | 242 | Pitesti | 100 |
| Eforie | 161 | Rimmicu vikea | 193 |
| Fagaras | 176 | Sibiu | 253 |
| Giurgiu | 77 | Timisoara | 329 |
| Hirsova | 151 | Urzicemi | 80 |
| Iasi | 226 | Vaslui | 199 |
| Lugoj | 244 | zerind | 374 |

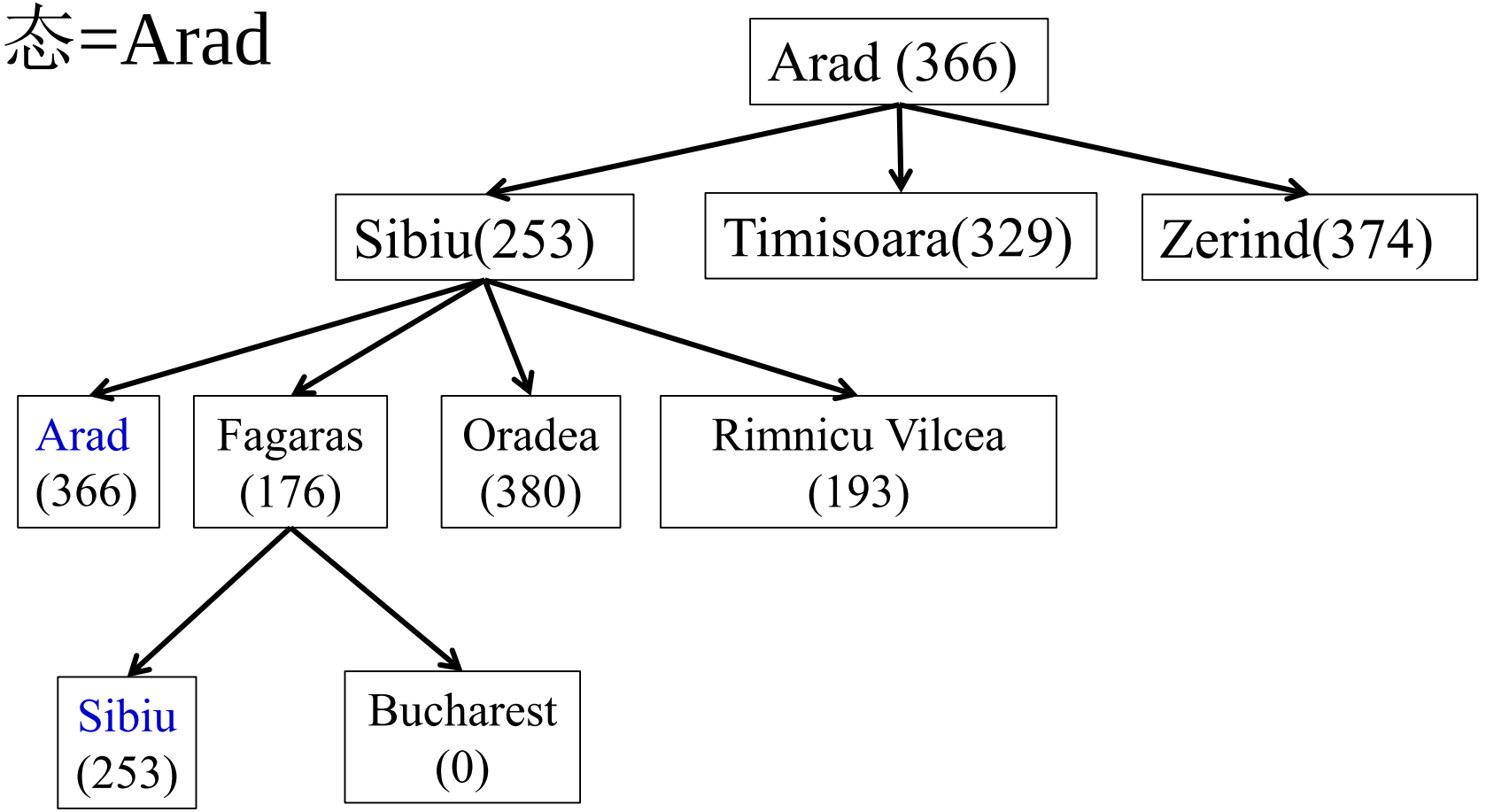
$h_{SLD}$=与目标城市间的直线距离 （km）

# 贪婪优先搜索（Greedy best first Search）

- 贪婪搜索每次选择距离目标代价最小的节点优先扩展
- 估价函数定义
  - $f(n) = h(n)$
  - $h(n)$ 估计$n$到目标节点的代价

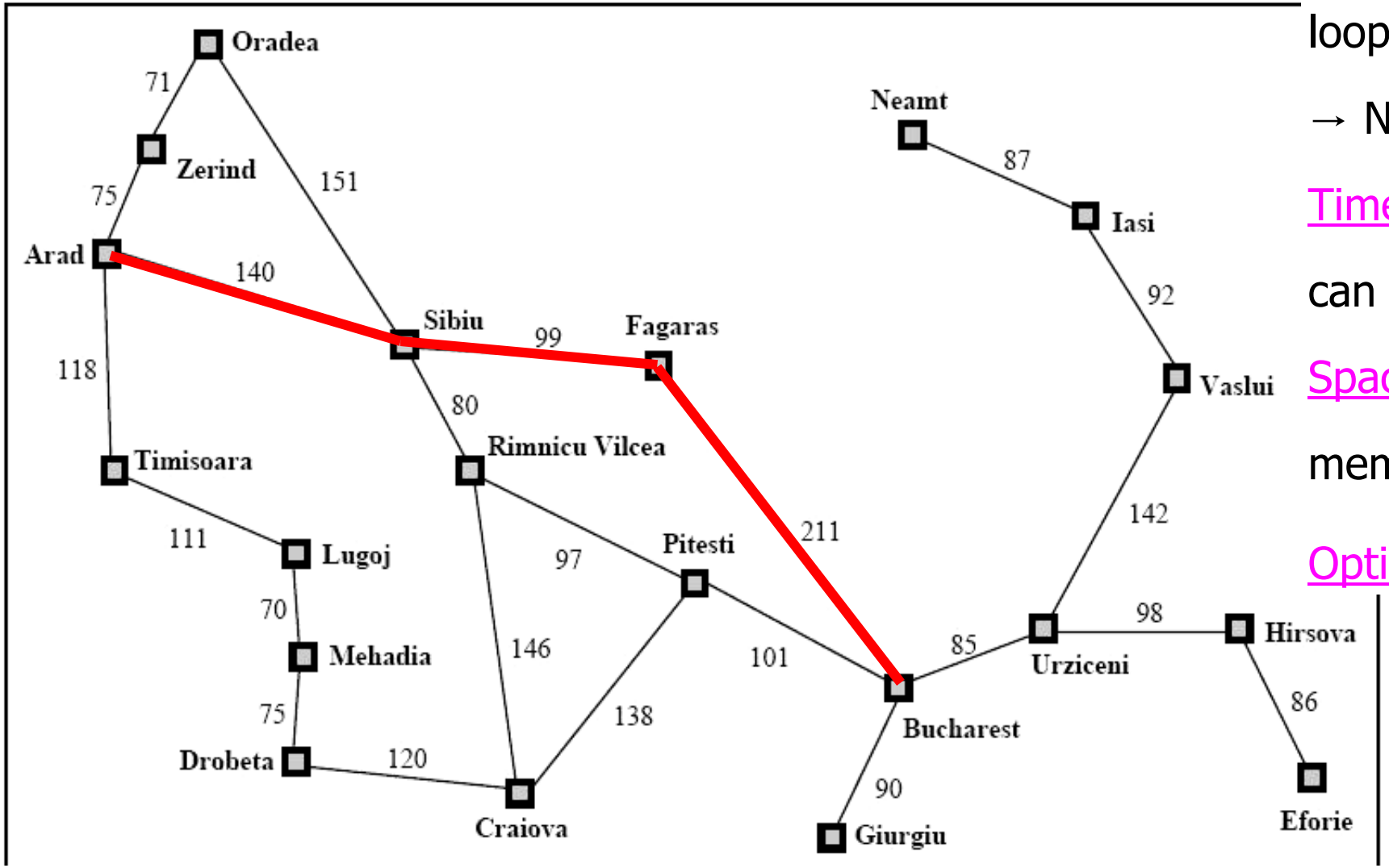
# 例

- 用贪婪搜索来求解罗马尼亚问题
  - $f(n)=h(n)= h_{SLD}(n)$
- 初始状态=Arad



# 例: Romania

不是最优路径！




Complete?? No–can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →

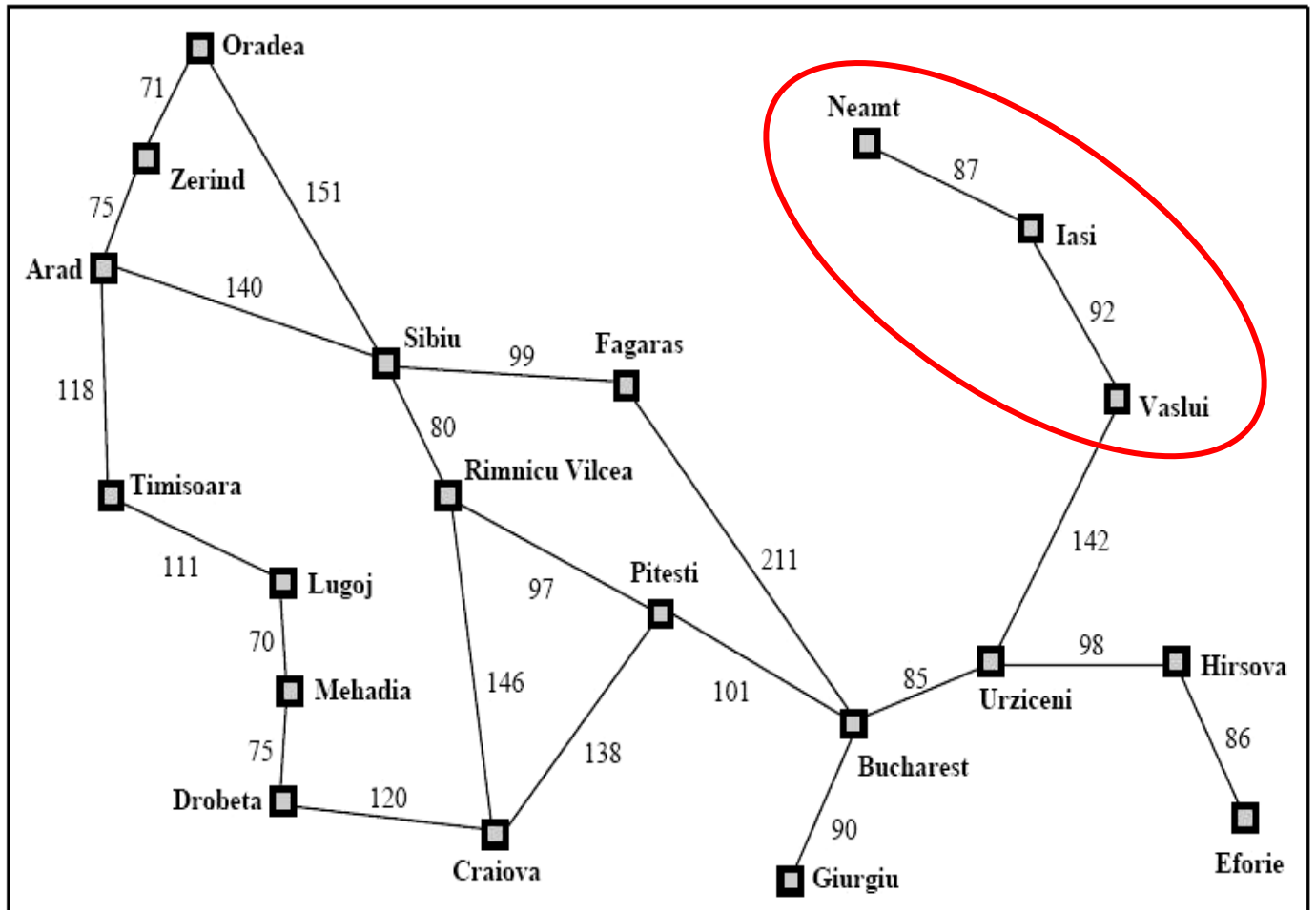
Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

# 思考题4

- 若现在的位置在*Iasi*，要到*Fagaras*去，仍使用$f(n)=h_{SLD}(n)$，采用贪婪优先树搜索
- 画出OPEN表的变化过程



# 思考题4解析——树搜索

- 显然*Neamt*到*Fagaras*的直线距离小于*Vaslui*到*Fagaras*

| Expand Node | Open List |
|---|---|
| | {*Iasi*} |
| *Iasi* | {*Neamt*, *Vaslui*} |
| *Neamt* | {*Iasi*, *Vaslui*} |
| *Iasi* | {*Neamt*, *Vaslui*} |
| *Neamt* | {*Iasi*, *Vaslui*} |
| … | … |

- 陷入死循环，无法找到解

# 贪婪优先搜索的性能

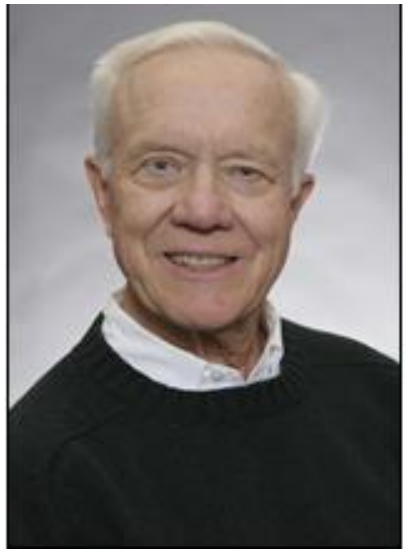
- 通常表现得很好
- 找到解的速度很快
- 不能保证找到解
  - 不合适的起点可能导致找不到解
- 不能保证最优性

# 练习7

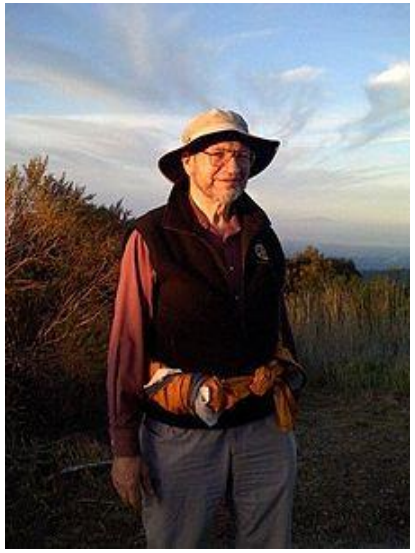
- 设计合适的结构存储地图
- 编程实现采用贪婪优先的树搜索策略搜索从Arad到Bucharest的路径

# A算法

- 1964年，尼尔逊提出一种算法以提高最短路径搜索的效率，被称为A1算法
- 1967年，拉斐尔大大改进了A1算法，称为A2算法
- A1,A2算法统称为A算法
  - $f(n)=g(n)+h(n)$



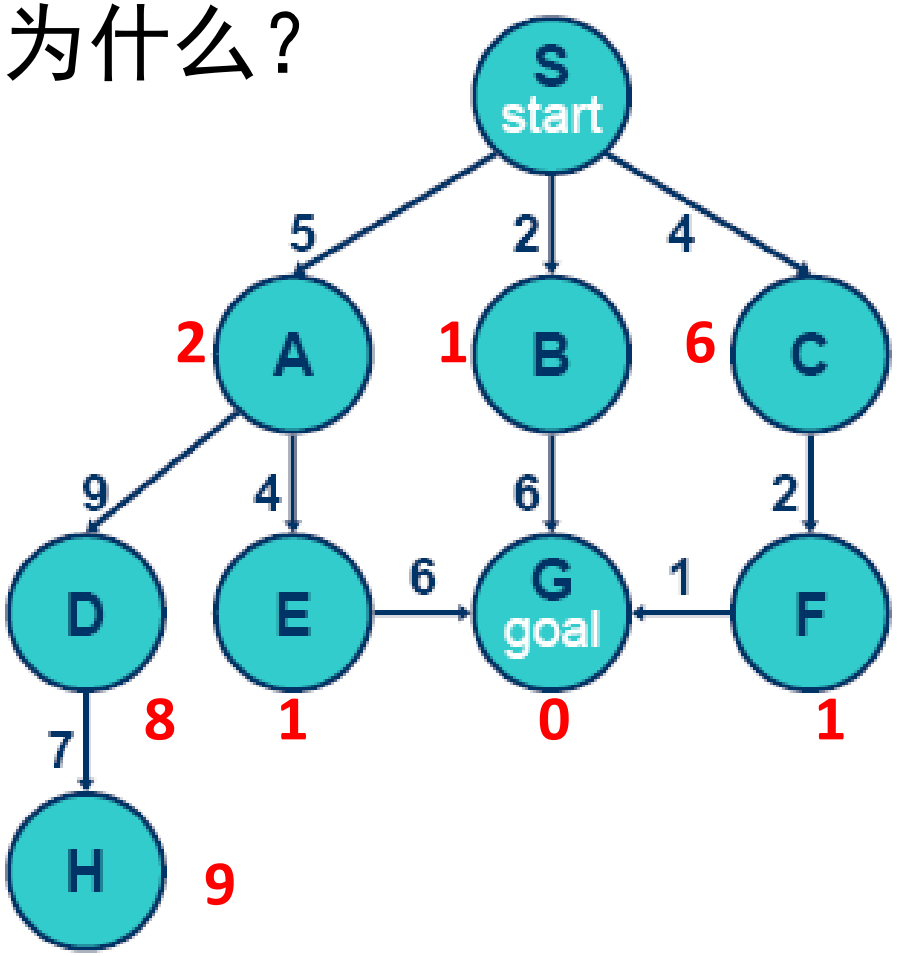
尼尔逊



拉斐尔

# 思考题5

- 使用A算法的树搜索策略对下面的图进行搜索，搜索从S到G的路径，每个节点旁红色的数字是该点的h值，请问最后找到的解是否是最优解？为什么？



# 思考题5解析

| Expand Node | Open List |
|---|---|
| | {S} |
| S | {B:2+1, A:5+2, C:4+6} |
| B | {A:5+2, G:2+6+0, C:4+6} |
| A | {G:2+6+0, C:4+6, E:5+4+1, D:5+9+8} |
| G | |

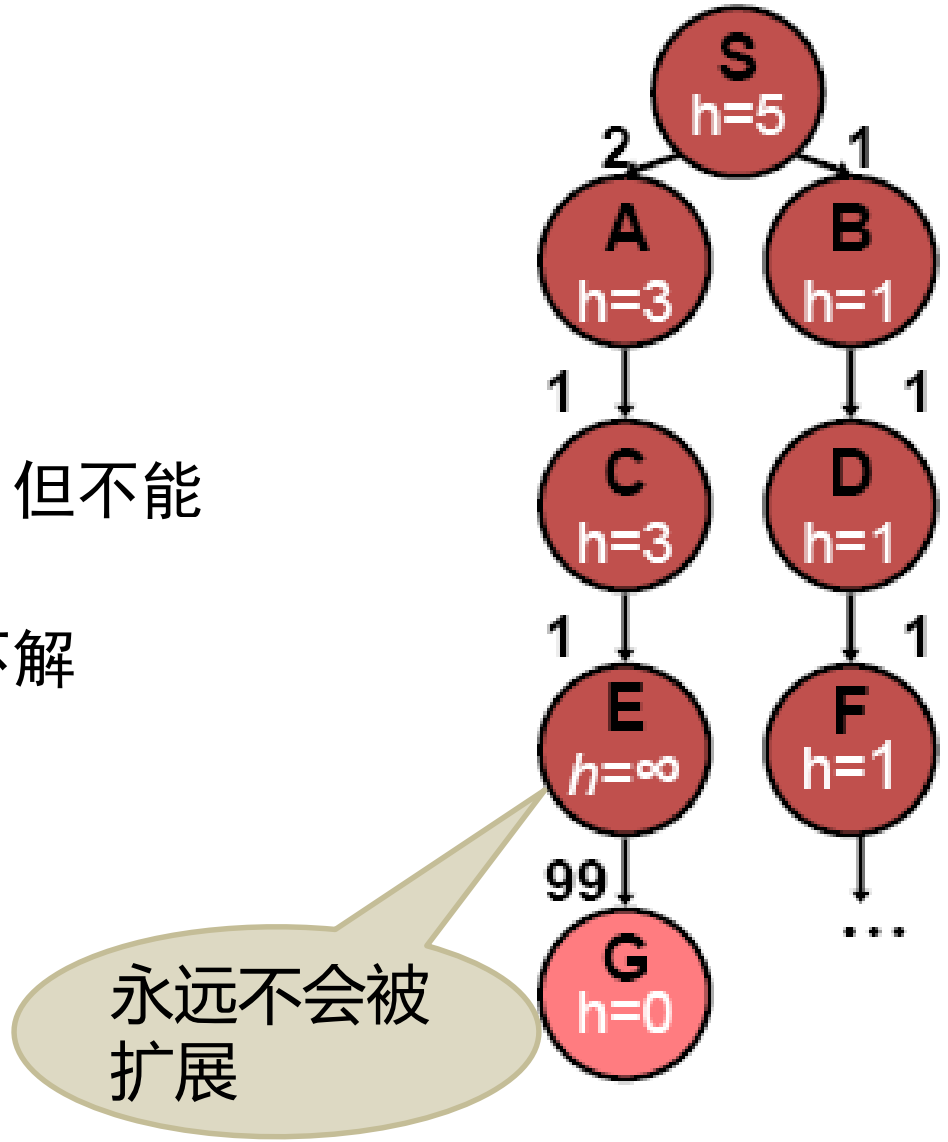
- 找到的解为SBG，cost=8，不是最优解
- 最优解为SCFG，cost=7
- C节点在最优解路径上，但由于**h值的设置**不合适，使得它在与节点G的PK中失败，直接导致无法找到最优解
- **可以看出，是否能找到最优与h的定义紧密相关**

# A算法

- 特征：
  - 估价函数

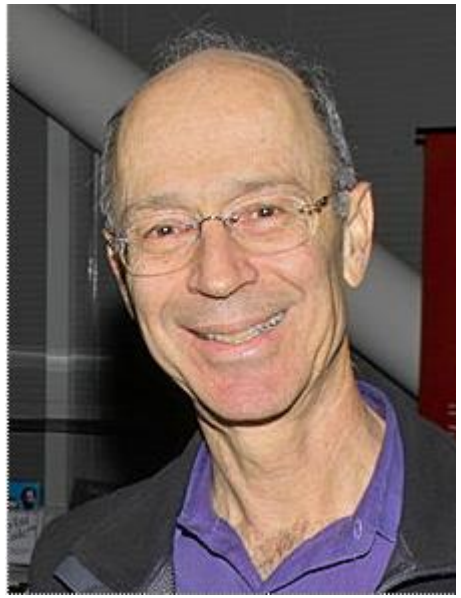
    $$f(n) = g(n) + h(n)$$

  - 对$h(n)$无限制，虽提高了算法效率，但不能保证找到最优解
  - 不合适的$h(n)$定义会导致算法找到不解
- 性能
  - 不完备，不最优



# A*算法

- 1968年，彼得.哈特对A算法进行了很小的修改，并**证明了**当估价函数满足一定的限制条件时，算法一定可以找到最优解
- 估价函数满足一定限制条件的算法称为A*算法



彼得.哈特

A*算法的限制条件

$$f(x) = g(x) + h(x)$$

$g(x)$：大于0

$h(x)$：不大于$x$到目标的实际代价$h^*(x)$

# A* search——Tree search

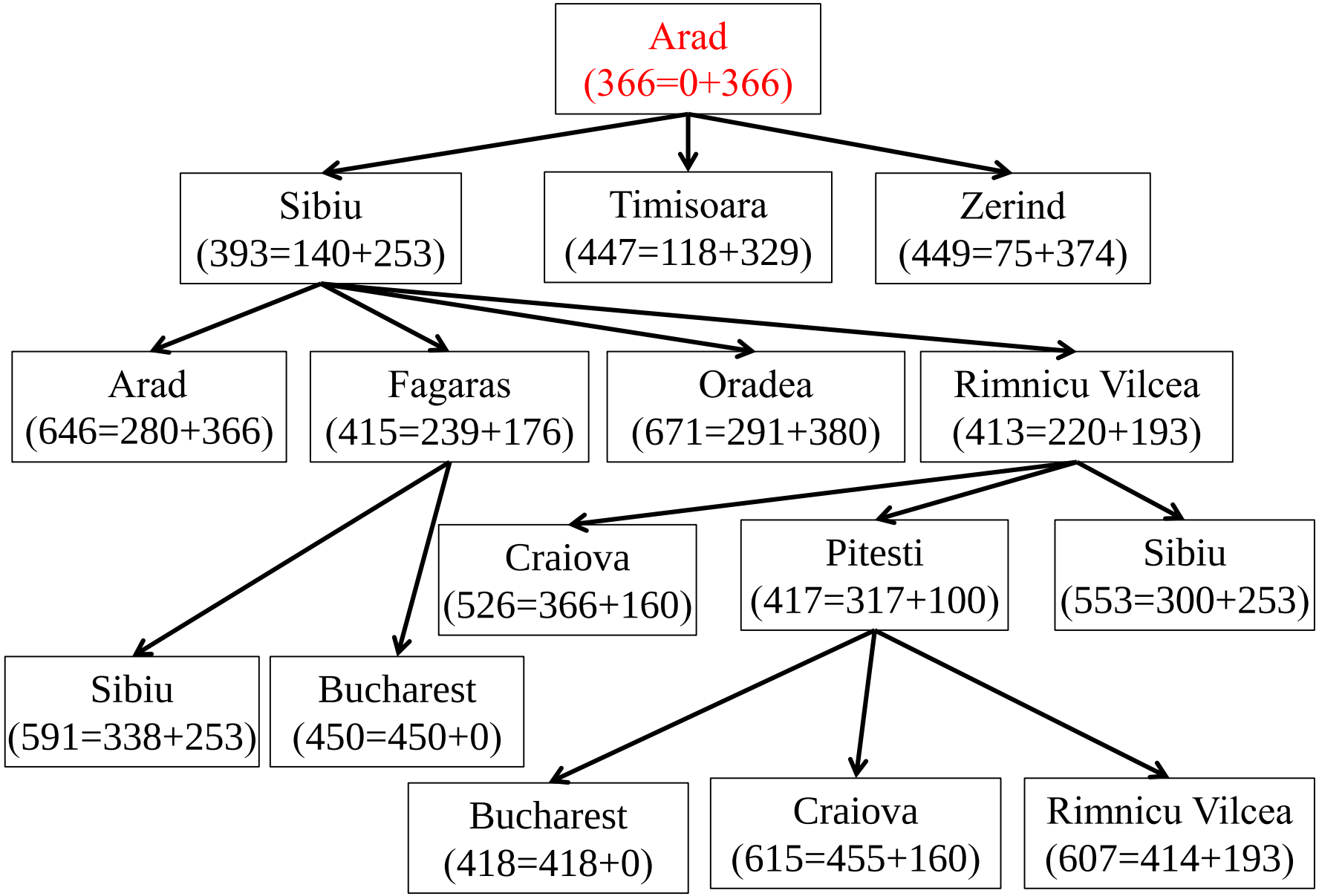
- A*算法要求其启发函数的定义是可纳的
  - –可纳性（ admissible ）
    - $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from $n$
    - Are optimistic

**可容admissible：** 专门针对启发函数而言，即启发函数不会 过高估计 从节点n到目标节点之间的实际开销代价，即小于等于实际开销。

- e.g. $h_{SLD}(n)$

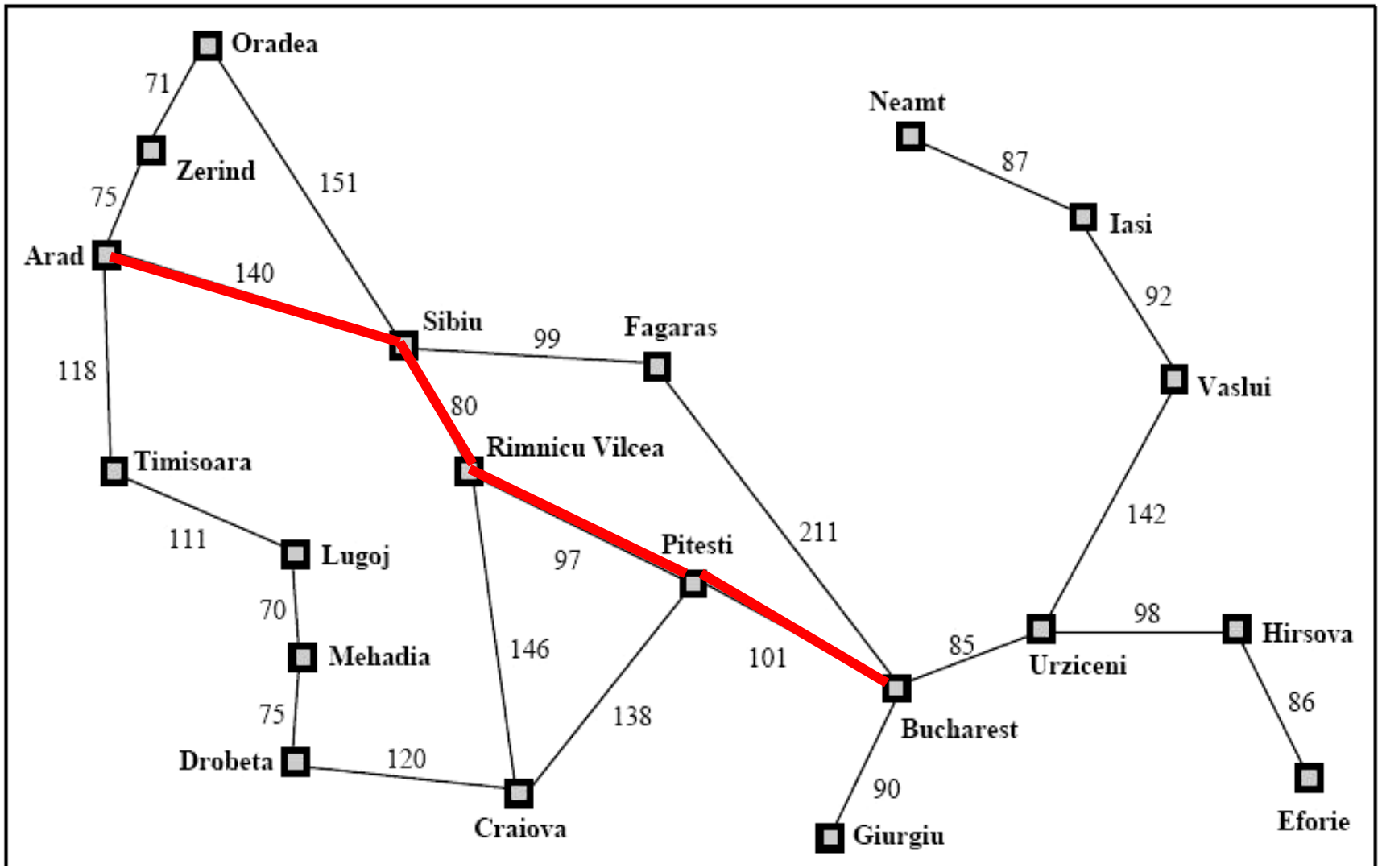
比如将两点之间的直线距离作为启发函数，从而保证其可容。
算法中的距离估算值与实际值越接近，最终搜索速度越快。

# Romania A* search example



# 例: Romania

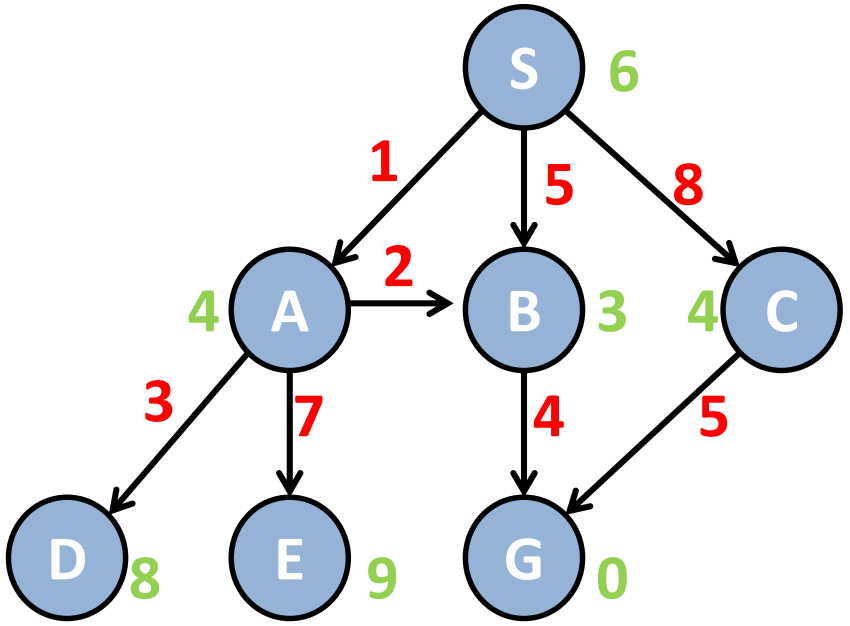
找到最优路径！



# BUT … graph search

- 若按以下算法搜索右图会发生什么？

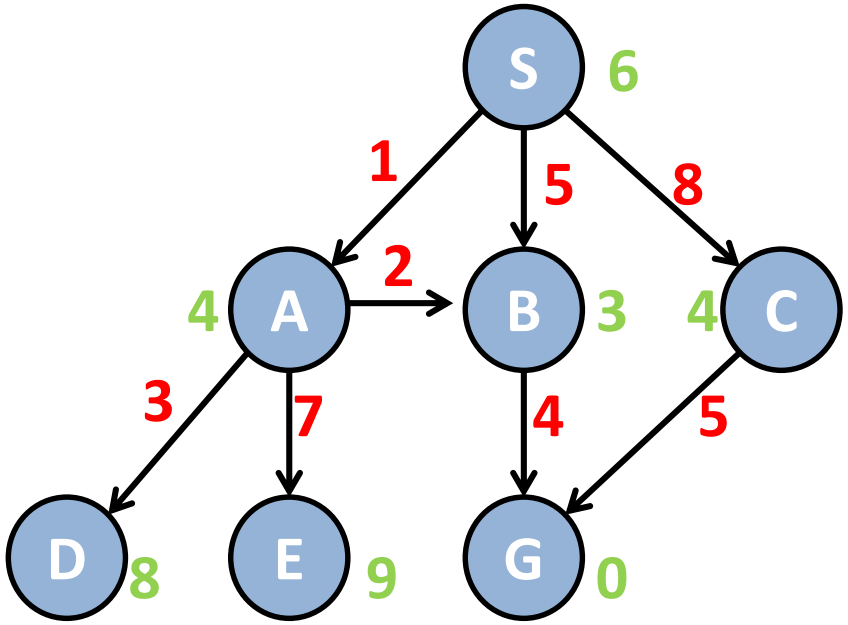
初始化 *open* 表，将起始节点放入其中
初始化 *closed* 表为空表
Loop
    if *open* 为空return *failure*
    *Node*←remove-first (*open*)
    if *Node* is a *goal*
    then return从起始节点到*Node*的路径
    else 将*Node* 移入 *closed* 表
    生成*Node* 的后继节点集合 *S*
    for **all nodes *m*** in *S*
        if *m* 不在*open* or *closed*
            将*m*按***f*** 值顺序加入OPEN表
End Loop




若忽略重复节点，则有可能无法找到最优路径

# BUT … graph search

- 在图搜索策略中要检查新生成的节点是否已经在Open或Closed表中，若是则忽略它




| Expnd. node | Open list | Closed list |
|---|---|---|
| | {S} | |
| S | {A:5, B:8, C:12} | {S} |
| A | {B:8, C:12, D:12, E:17} | {S,A} |
| B | {G:9,C:12,D:12,E:17} | {S,A,B} |
| G | {C:12,D:12,E:17} | {S,A,B} |

Path: S,B,G

Cost: 9 不是最优

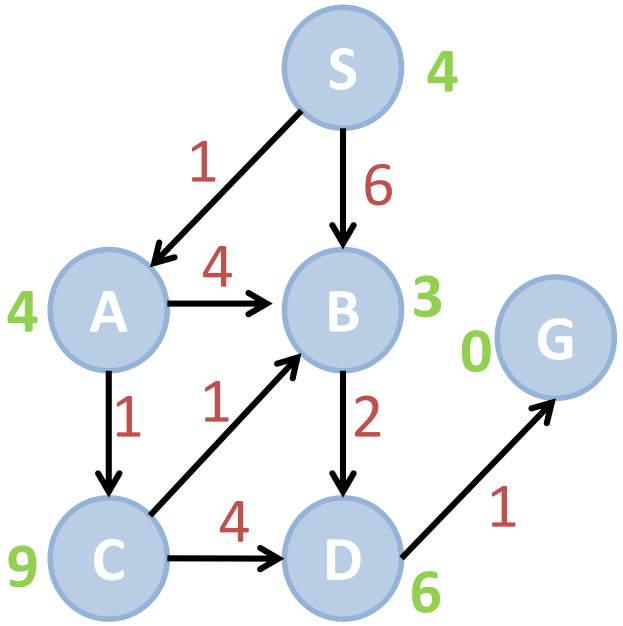
- **忽略重复节点，有可能无法找到最优路径**

# A* Search ( Graphic Search)

Algorithm A*

OPEN={*S*}
while OPEN is not empty
{ 将OPEN中*f*(*n*)最小的节点*n*移入CLOSED
if *n* is a goal node,
return success (path *p*)
**对*n* 的所有子节点 *m***
if *m* is on CLOSED 且当前代价比原来的代价低,
then remove *m* from CLOSED,
put *m* on OPEN
else if *m* is on OPEN且当前代价比原来的代价低,
修改原OPEN表中*m*的父节点指针
else if *m* is not on OPEN
Then put *m* on OPEN }
Return failure

# Example: A* graph search




| Expnd. node | Open list | Closed list |
|---|---|---|
| | {S: 4} | |
| S | {A:5, B:9} | {S} |
| A | {B:8, C:11} | {S,A} |
| B | {C:11, D:13} | {S,A,B} |
| C | {B:6, D:12} | {S,A,C} |
| B | {D:11} | {S,A,C,B} |
| D | {G:6} | {S,A,C,D} |

# A* Search ( Graphic Search)

Algorithm A*

OPEN={*S*}
while OPEN is not empty
{ 将OPEN中*f*(*n*)最小的节点*n*移入CLOSED
if *n* is a goal node,
return success (path *p*)
对*n* 的所有子节点 *m*
if *m* is on CLOSED 且当前代价比原来的代价低,
then remove *m* from CLOSED,
put *m* on OPEN
else if *m* is on OPEN且当前代价比原来的代价低,
修改原OPEN表中*m*的父节点指针
else if *m* is not on OPEN
Then put *m* on OPEN }
Return failure

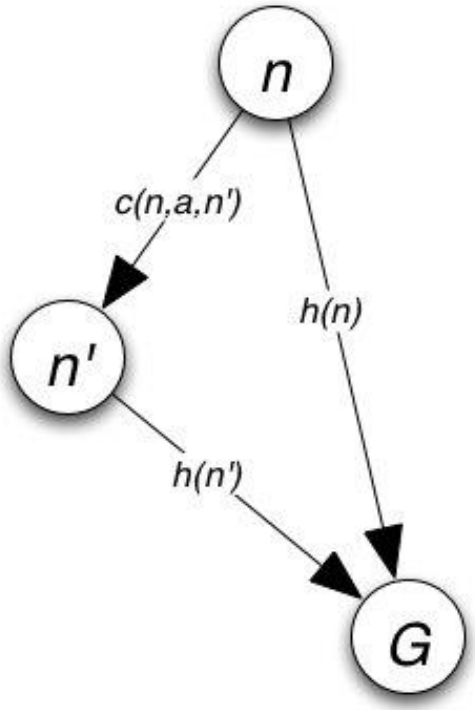
# 图搜索中$h(n)$的单调限制

- 如果启发函数满足以下条件，则称$h$满足**单调性**（一致性， consistent ）

$$h(n) \le c(n,a,n') + h(n')$$

或： $h(n) - h(n') \le c(n,a,n')$

– 其中$n'$是$n$的子节点




- If $h$ is consistent, we have

$$f(n') = g(n') + h(n')$$
$$= g(n) + c(n,a,n') + h(n')$$
$$\ge g(n) + h(n) = f(n)$$

三角形不等式：如果启发式函数$h$是一致的，

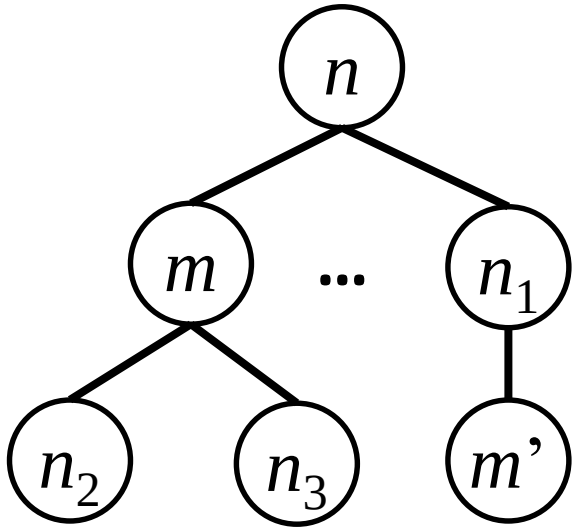
那么单个数值$h(n)$小于从$n$到$n'$的动作代价$c(n,a,n')$

加上启发函数的估计值$h(n')$ 的和。

一致的启发函数都是可容的（反过来不成立）

# 图搜索中$h(n)$的单调限制

- 若$h(n)$满足单调性限制，则沿着图中任何一条路径，路径上点的$f(n)$都是非递减的
  - 如果$h$满足单调条件，则当A*算法第一次扩展节点$n$时，该节点就已经找到了通往它的最优路径




**证明：如图**

**对任意节点$m$已经在CLOSED中，**

**若又有节点$n_1$再次扩展出$m$，记为$m'$**

**因为$n_1$比$m$后扩展**

**显然$f(n_1)>f(m)$**

**又根据单调性限制$f(m')>f(n_1)>f(m)$**

**所以第一次扩展时的$m$一定具有最小$f(m)$**

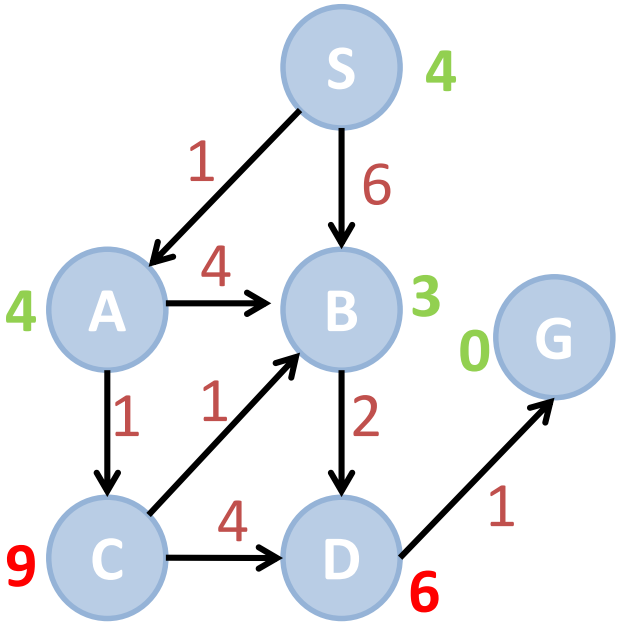
- 若**启发函数满足单调性**，则对应的A*算法称为改进的A*算法。

# 改进的A*算法( Graphic Search)

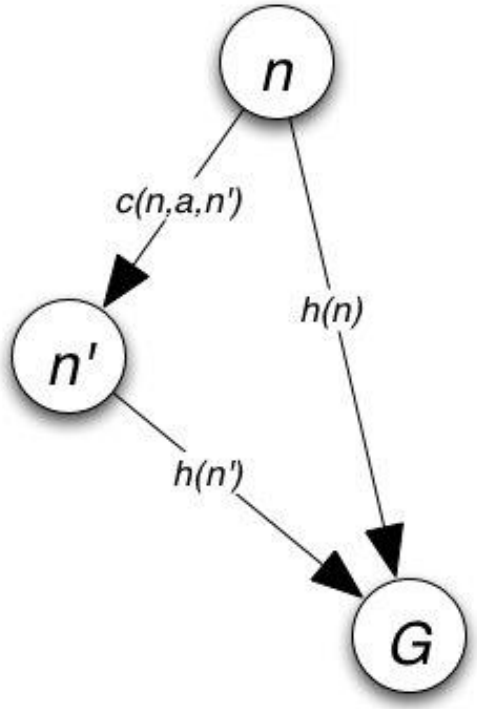
Algorithm A*

OPEN={*S*}
while OPEN is not empty
{ 将OPEN中*f*(*n*)最小的节点*n*移入CLOSED
if *n* is a goal node,
return success (path *p*)
对*n* 的所有子节点 *m*
~~if *m* is on CLOSED 且当前代价比原来的代价低,~~
~~then remove *m* from CLOSED,~~
~~put *m* on OPEN~~
~~else~~ if *m* is on OPEN且当前代价比原来的代价低,
修改原OPEN表中*m*的父节点指针
~~else~~ if *m* is not on OPEN or CLOSED
Then put *m* on OPEN }
Return failure

# Example: A* graph search



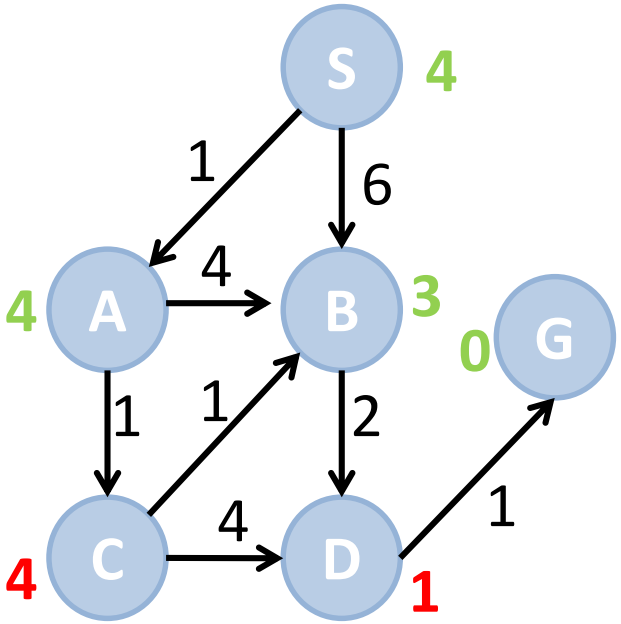

| Expnd. node | Open list | Closed list |
|---|---|---|
| | {S: 4} | |
| S | {A:5, B:9} | {S} |
| A | {B:8, C:11} | {S,A} |
| B | {C:11, D:13} | {S,A,B} |
| C | {B:6, D:12} | {S,A,C} |
| B | {D:11} | {S,A,C,B} |
| D | {G:6} | {S,A,C,D} |




$h(n) - h(n') \leq c(n,a,n')$

# 满足一致性的启发函数

- 再试试，是否当节点第一次被扩展时就已经找到了到这个节点的最优路径？

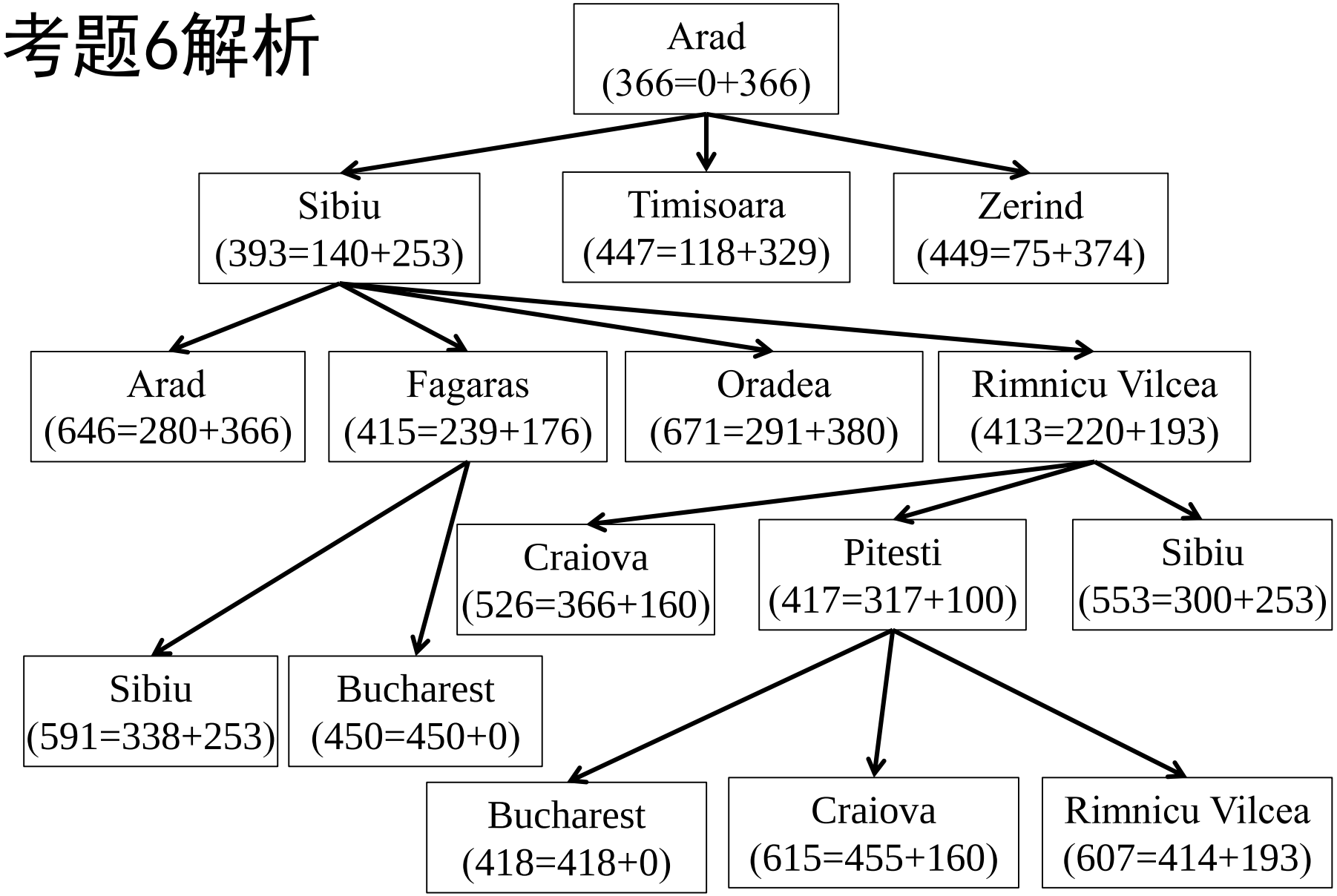



| Expnd. node | Open list | Closed list |
|---|---|---|
| | {S} | |
| S | {A:5,B:9} | {S} |
| A | {B:8,C:6} | {S,A} |
| C | {B:6,D:7} | {S,A,C} |
| B | {D:6} | {S,A,C,B} |
| D | {G:6} | **{S,A,C,B,D}** |
| G | | |

# 思考题6

- 在Romania问题中， $h(n)=h_{SLD}(n)$，是否能够满足一致性？为什么？
- 采用改进的A*图搜索求解这个问题，画出搜索树和OPEN表、CLOSED表的变化

# 思考题6解析



# 练习8

- 编程实现用改进的A*图搜索算法求解Romania问题，搜索从Arad到Bucharest的最优路径

# A*与UCS算法性能比较

- 全局路径规划问题是一个典型的图搜索问题
- 利用该问题来比较各类搜索算法的性能
  - A*
    - $h(n)$=$n$到目标$G$的曼哈顿距离

      $=|x_n-x_G|+|y_n-y_G|$
    - $f(n)=g(n)+h(n)$
  - UCS
    - $f(n)=g(n)$

# 估价函数对算法的影响

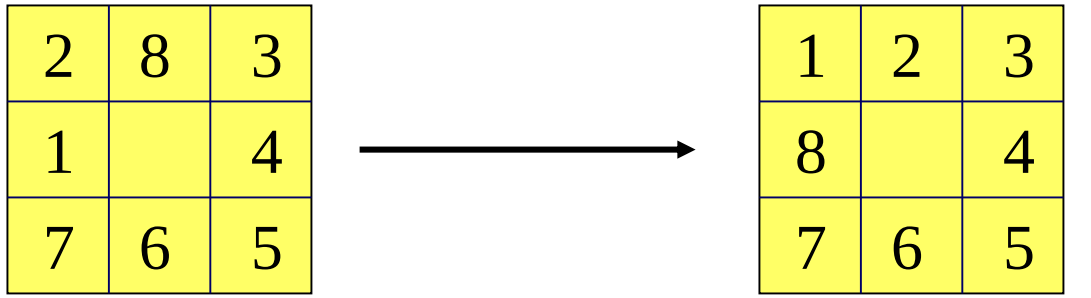
- 不同的估价函数对算法的效率可能产生极大的影响
- 不同的估价函数甚至产生不同的算法

$$f(x) = g(x) + h(x)$$

  - $h(x)=0$：uniform cost search，非启发式算法
  - $g(x)=0$：greedy search，无法保证找到解
  - 即使采用同样的形式$f(x) = g(x) + h(x)$，不同的定义和不同的值也会影响搜索过程

# 例：

- 八数码难题




（初始状态）　　（目标状态）

操作：空格上移，空格下移，空格左移，空格右移

# 利用A*算法求解八数码问题

- 估价函数的定义
  - $f(x) = g(x) + h(x)$
  - $g(x)$：从初始状态到$x$需要进行的移动操作的次数
  - $h(x)$：？ =$x$状态下错放的棋子数是可纳的


| | | |
|---|---|---|
| 2 | 8 | 3 |
| 7 | 1 | 4 |
| | 6 | 5 |

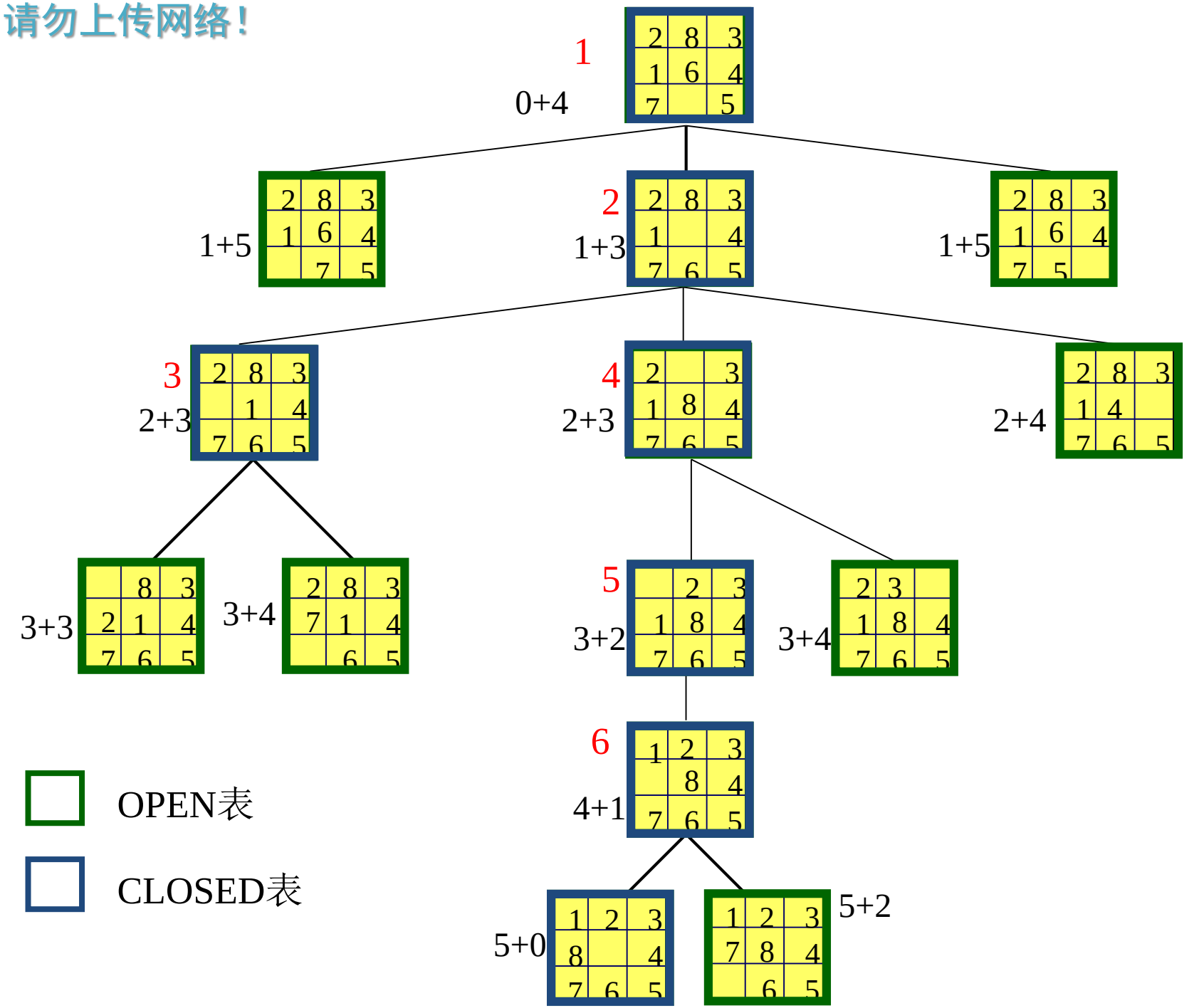
$h(x)=4$

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 7 | 8 | 4 |
| | 6 | 5 |

$h(x)=2$

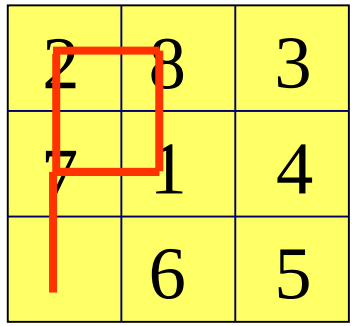
| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 8 | 4 |
| 7 | 6 | 5 |

$h(x)=1$

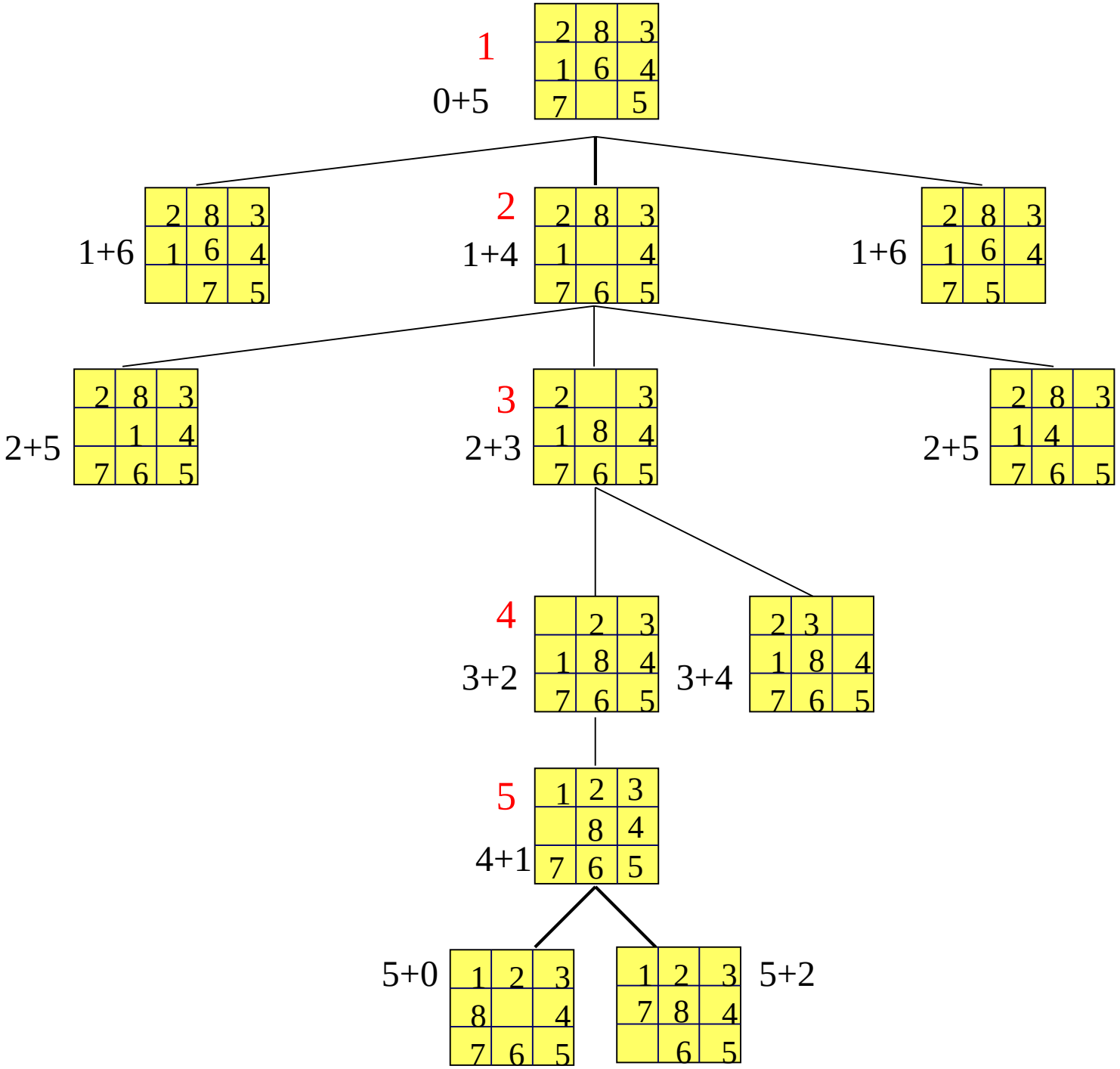

1
0+4
2 8 3
1 6 4
7 5
1+5
2 8 3
1 6 4
7 5
2
1+3
2 8 3
1 4
7 6 5
1+5
2 8 3
1 6 4
7 5
3
2+3
2 8 3
1 4
7 6 5
4
2+3
2 3
1 8 4
7 6 5
2+4
2 8 3
1 4
7 6 5
3+3
8 3
2 1 4
7 6 5
3+4
2 8 3
7 1 4
6 5
5
3+2
2 3
1 8 4
7 6 5
3+4
2 3
1 8 4
7 6 5
6
4+1
1 2 3
8 4
7 6 5
5+0
1 2 3
8 4
7 6 5
1 2 3
7 8 4
6 5
5+2
OPEN表
CLOSED表

# 思考题7

- 八数码问题中估价函数如下定义
  - $f(x) = g(x) + h(x)$
  - $g(x)$：从初始状态到$x$需要进行的移动操作的次数
  - $h(x)$：所有棋子与目标位置的曼哈顿距离之和
    - 该定义是否是可纳的？
    - 画出搜索树，与前面采用放错棋子数的算法对比。
    - 对结果进行分析，提出你自己的看法。




$h(x)=2+1+1+2=6$


1
0+5
2 8 3
1 6 4
7 5
2
1+4
2 8 3
1 4
7 6 5
1+6
2 8 3
1 6 4
7 5
1+6
2 8 3
1 6 4
7 5
3
2+3
2 3
1 8 4
7 6 5
2+5
2 8 3
1 4
7 6 5
2+5
2 8 3
1 4
7 6 5
4
3+2
2 3
1 8 4
7 6 5
3+4
2 3
1 8 4
7 6 5
5
4+1
1 2 3
8 4
7 6 5
5+0
1 2 3
8 4
7 6 5
1 2 3
7 8 4
6 5
5+2

# Dominance(占优)

- If $h_2(n) \geq h_1(n)$ for all $n$ (两者均可纳) then $h_2$ dominates $h_1$
  - *If $h(n) = h^*(n)$ for all $n$,*
    - 只扩展最优路径上的点
  - *If $h(n) = 0$ for all $n$,*
    - A* 退化为UCS
- $h$越接近$h^*$, 扩展的节点越少

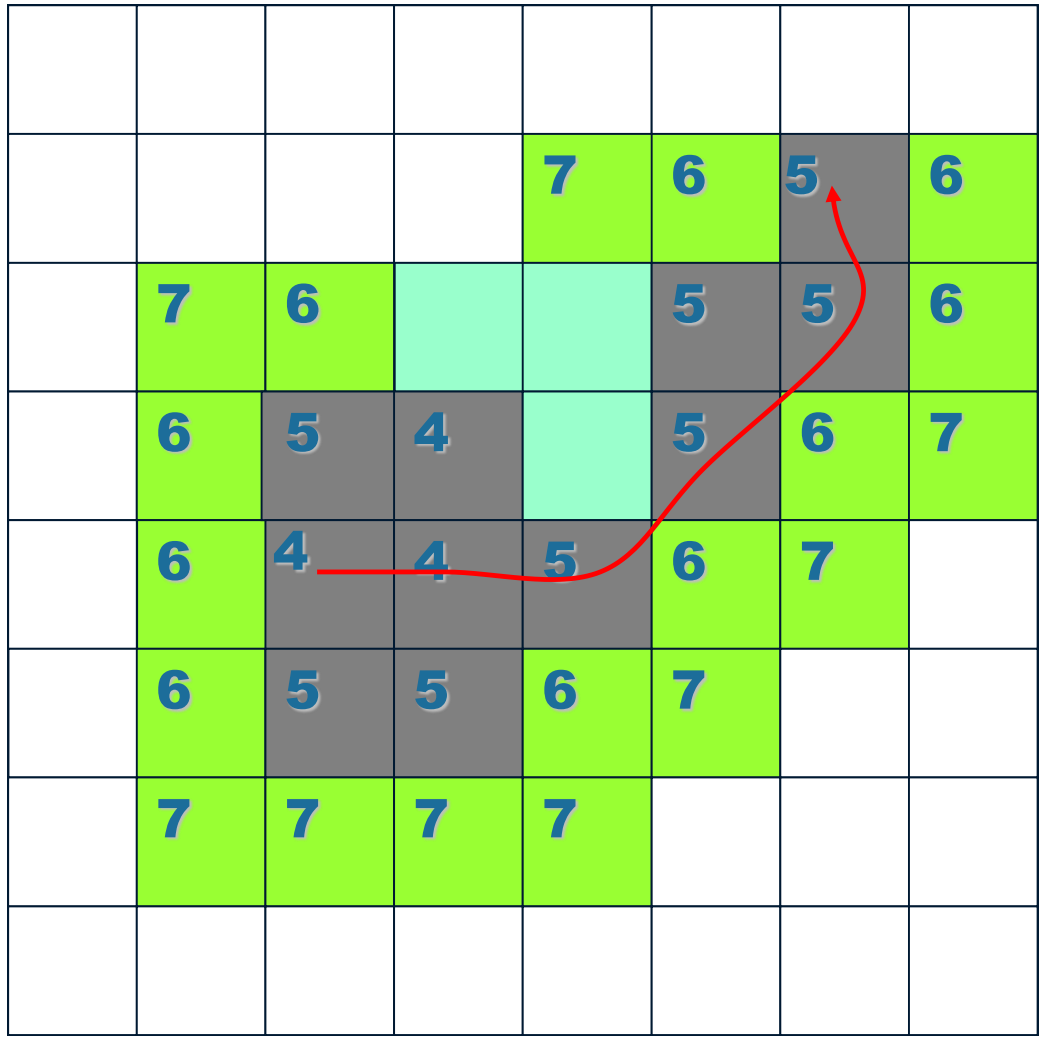
# Example: 8 puzzle

- Typical search costs (average number of nodes expanded):
  - $d$=12
    - $A^*(h_1)$ = 227 nodes
    - $A^*(h_2)$ = 73 nodes
  - $d$=24
    - $A^*(h_1)$ = 39,135 nodes
    - $A^*(h_2)$ = 1,641 nodes

# 启发函数的设计

- Most important in A*
- 一般可以问题对应的松弛问题(relaxed problem)最优解的代价作为原始问题的启发函数
  - relaxed problem——与原问题相比，该问题的限制条件更少

# 例：Simple Maze




- Relaxation in Maze
  - Move from (x,y) to (x’,y’) is illegal
    - If |x-x’| > 1
    - Or |y-y’| > 1
    - Or (x’,y’) contains a wall

  Otherwise, it’s legal.

# 松弛问题的可纳性

- Why does this work?
  - –所有在原始问题中合法的路径在松弛问题中仍是合法的
  - –松弛问题中最优解的代价一定不大于原始问题中最优解的代价

# 8-puzzle的松弛问题

- Tile move from (x,y) to (x’,y’) is illegal
  - If |x-x’| > 1 or |y-y’| > 1
  - Or (|x-x’| ≠ 0 and |y-y’| ≠ 0)
  - Or (x’,y’) contains a tile

  Otherwise, it’s legal

$h_1$(s): 放错的棋子数

假设对8-puzzle放松限制，棋子可以移动到任意位置，则$h_1(n)$ 对应的是该问题的最优路径代价

$h_2$(s): 曼哈顿距离之和

如果将规则放松到可以移动到任何相邻的格子，则$h_2(n)$对应的是该问题的最优路径代价