

# 人工智能技术

# Artificial Intelligence

——人工智能: 经典智能+计算智能+机器学习

AI: Classical Intelligence + Computing Intelligence + Machine Learning

王鸿鹏、杜月、王润花、许丽

南开大学人工智能学院





## Section II

# 第二部分 经典智能 Classical Intelligence

## ——第四章：问题求解 Chapter 4: Problem-Solving



# 问题求解 Problem-Solving

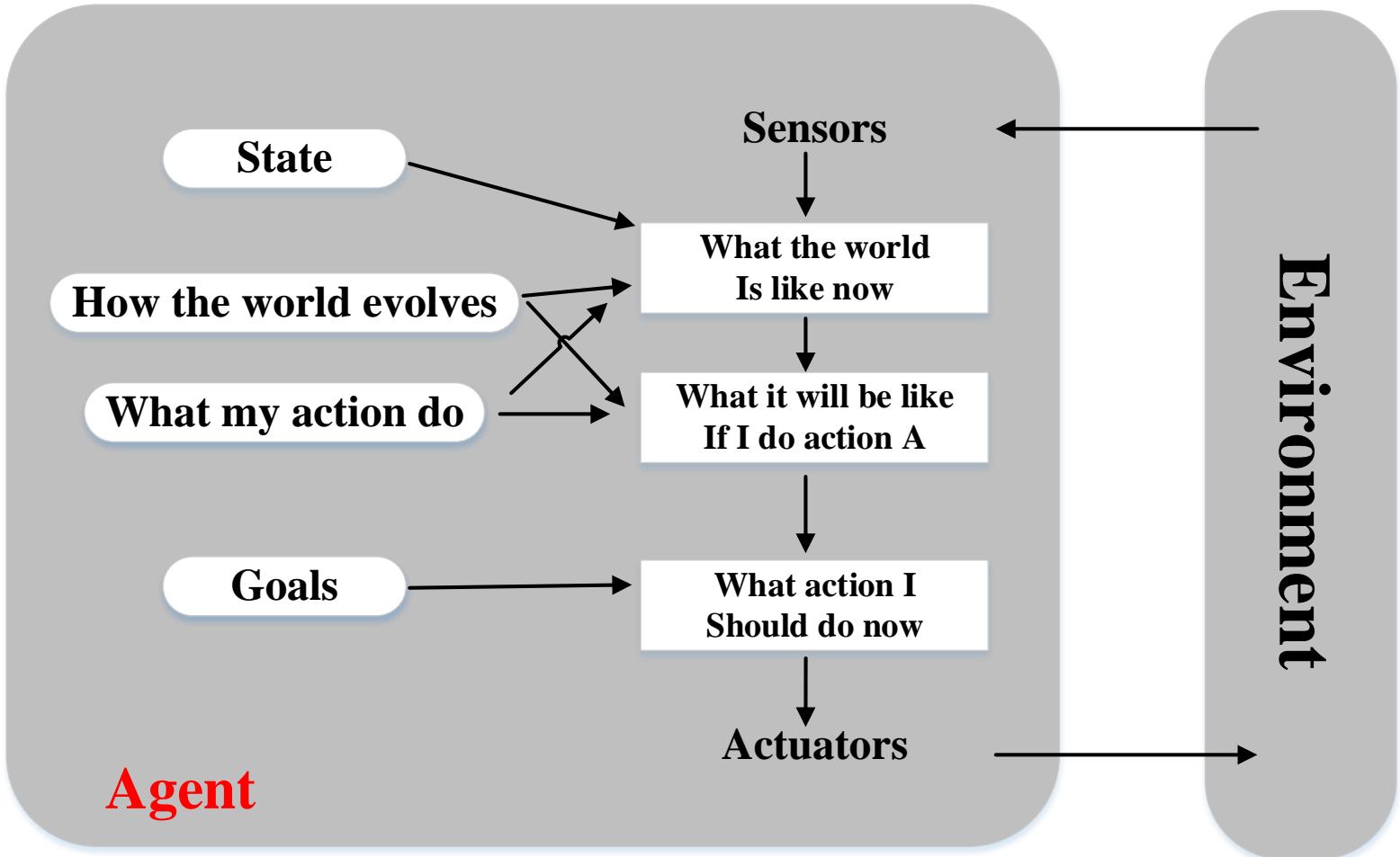
- 问题求解是人工智能的核心问题之一
- 问题求解的目的
  - 机器自动找出某问题的正确解决策略
  - 更进一步，能够举一反三，具有解决同类问题的能力
- 是从人工智能初期的智力难题、棋类游戏、简单数学定理证明等问题的研究中开始形成和发展起来的一大类技术
- 求解的手段多种多样
- 其中搜索技术是问题求解的主要手段之一
  - 问题表示
  - 解的搜索



# 问题求解 Agent

- An **agent** is anything that can be viewed as **perceiving** its **environment** through **sensors** and **acting** upon that environment through **actuators**
- 性能度量：考虑Agent行动的后果。通过性能度量，对作用于环境的行动序列进行评估
- 任务环境：PEAS描述
  - Performance (性能)
  - Environment (环境)
  - Actuators (执行器)
  - Sensors (传感器)

# 基于目标的Agent





# Methodologies of Solving Problems 1

搜索求解

Solving Problems by Searching



# 什么是搜索 Searching

- 搜索是人工智能的一个基本问题。
- 理论上有解的问题，在现实世界中由于各种约束（主要是对时空资源的约束）而未必能得到解（或者最优解）。搜索策略最关心的问题就是能否尽可能快地得到（有效或最优）解。
- 搜索就是利用已知条件（知识）寻求解决问题的办法的过程。

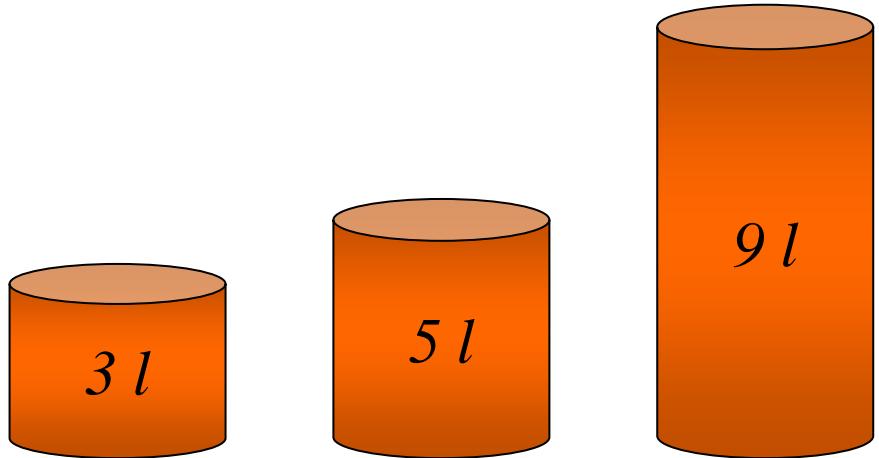


# 搜索方法适用的问题

- 两类问题
  - 结构不良或非结构化的问题；
  - 结构比较好，理论上也有算法可依，但问题本身的复杂性超过了计算机在时间、空间上的局限性的问题。
- 对于这两类问题，往往无法用某些巧妙的算法来获取其精确解，而只能利用已有知识，一步步摸索着前进。这个过程中就存在着如何寻找可用知识，确定出开销尽可能少的一条推理路线的问题。
- 根据问题实际情况寻找可用知识，并以此构造出一条代价较小的推理路线，使得问题获得圆满解决的过程就是搜索。

## 例题1

Example: Measuring problem!



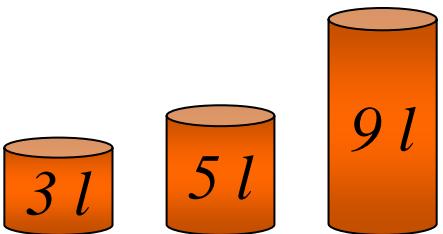
- **Problem:** Using these three buckets, measure 7 liters of water.

# 例题1

- **Problem:** Using these three buckets, measure 7 liters of water.

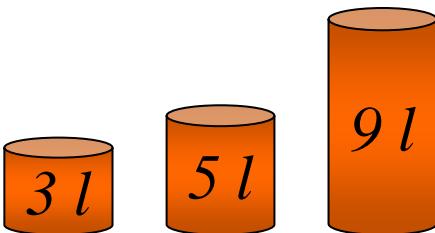
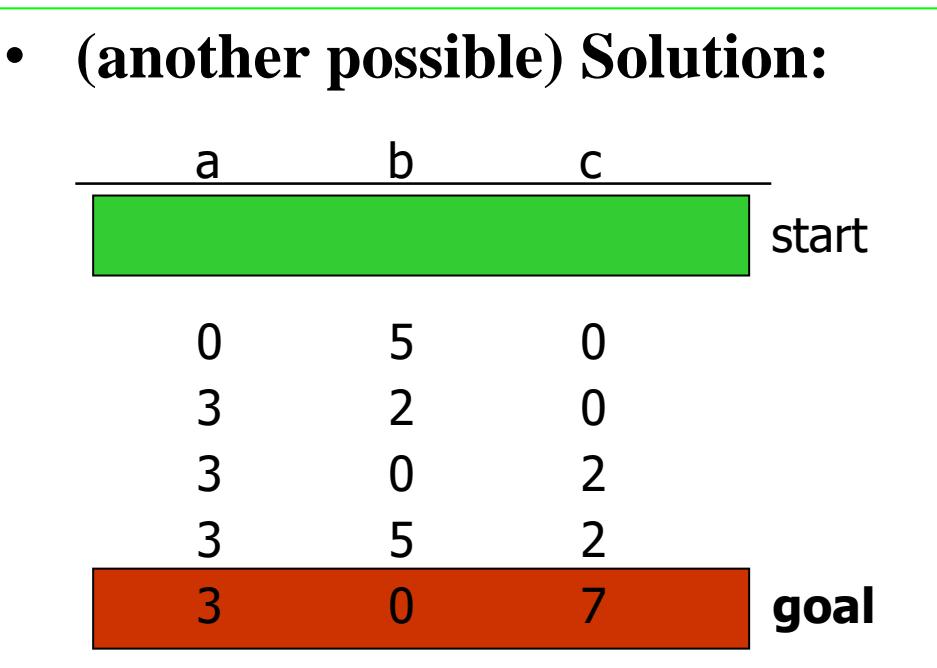
- **(one possible) Solution:**

a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	7	goal



## 例题1

- **Problem:** Using these three buckets, measure 7 liters of water.



# 例题1

Which solution do we prefer?

- **Solution 1:**

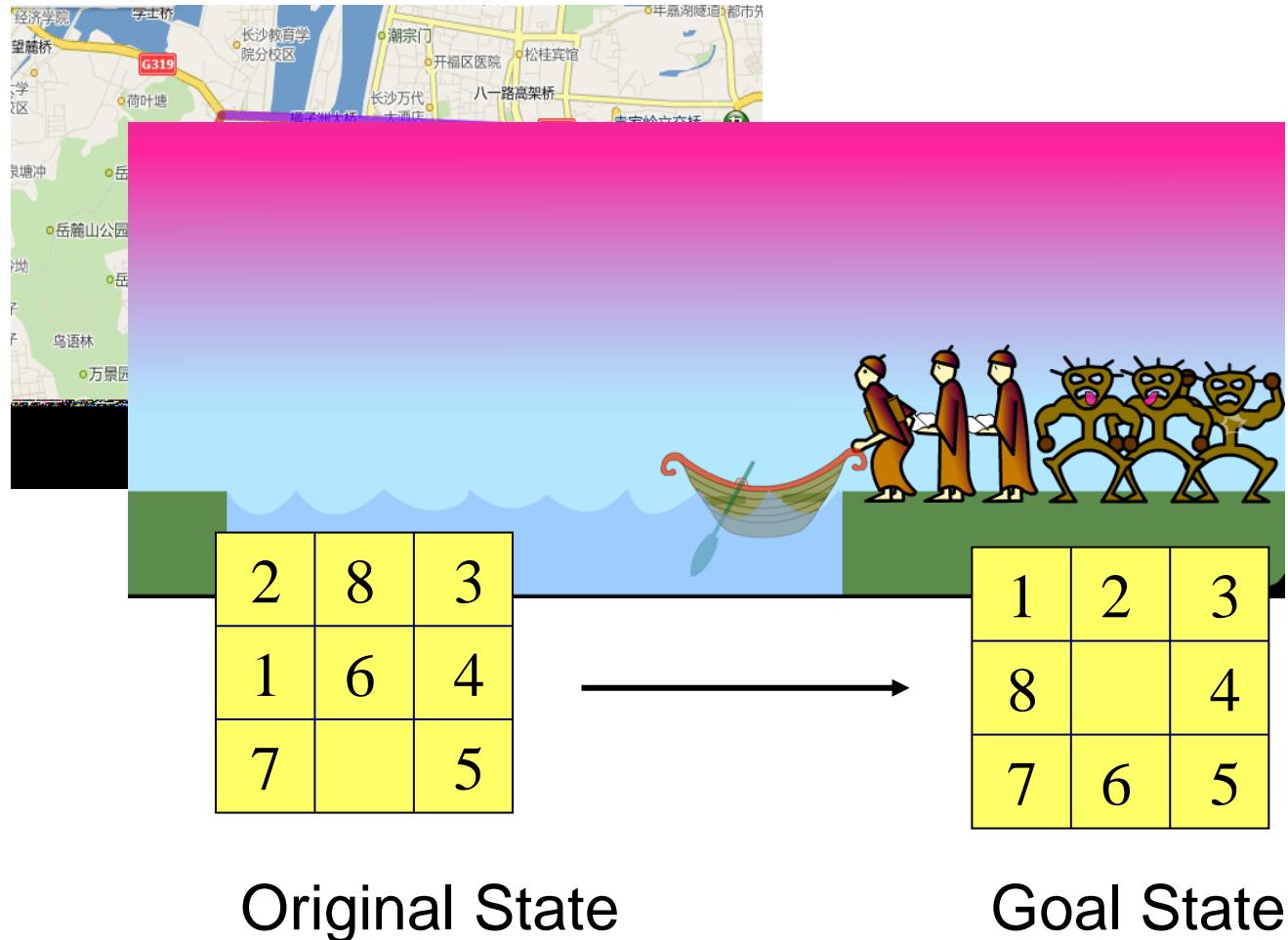
a	b	c	
0	0	0	start
3	0	0	
0	0	3	
3	0	3	
0	0	6	
3	0	6	
0	3	6	
3	3	6	
1	5	6	
0	5	7	goal

- **Solution 2:**

a	b	c	
0	0	0	start
0	5	0	
3	2	0	
3	0	2	
3	5	2	
3	0	7	goal

# 很多问题可以转化为搜索问题

- 导航
- 游戏
- 分配
- 调度
- 路径规划



# 图搜索

- 状态空间法的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
- 图的搜索
  - 无信息搜索（盲目搜索）
    - 宽度优先搜索
    - 深度优先搜索
  - 有信息搜索（启发式搜索）
    - A算法
    - A\*算法

图的一般搜索策略

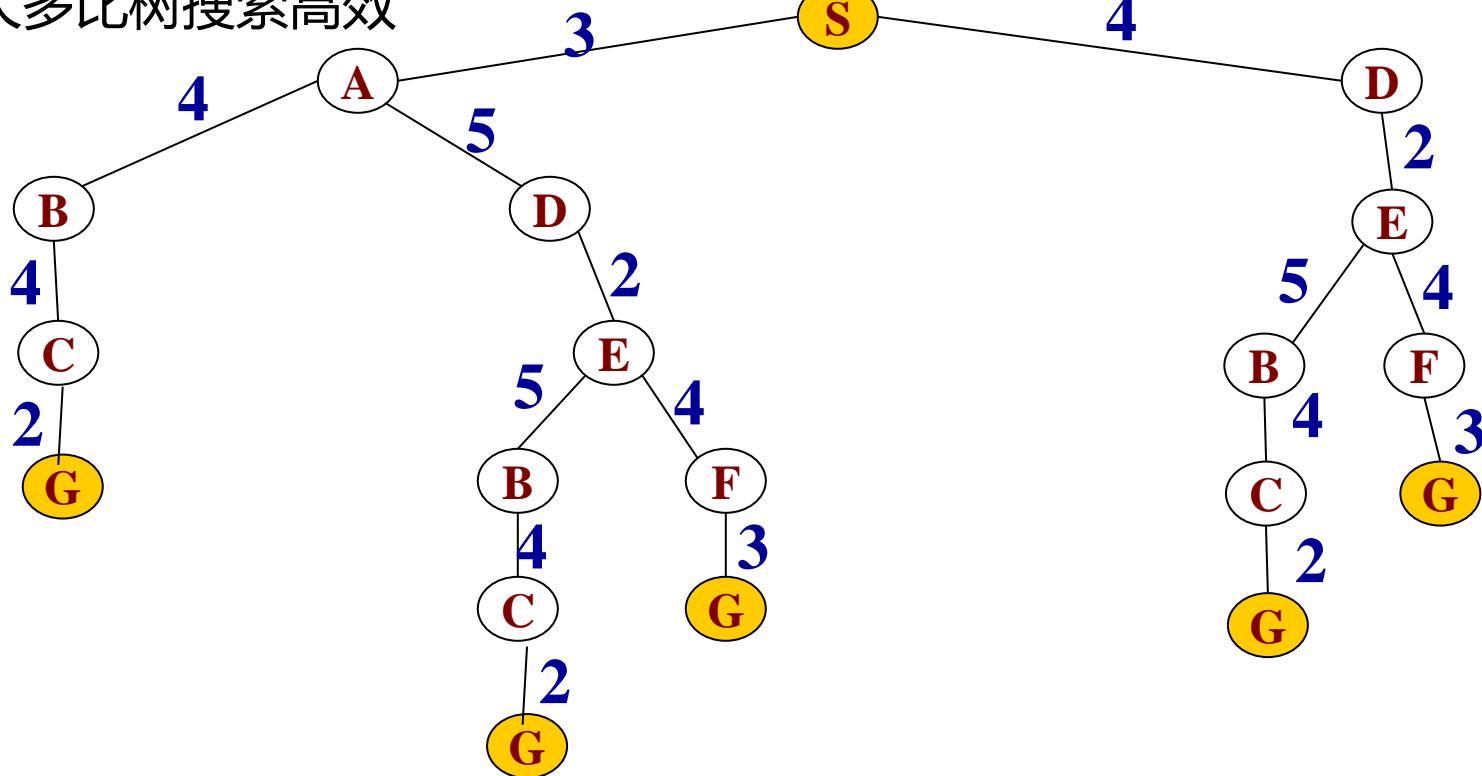


# 一般图搜索的假设

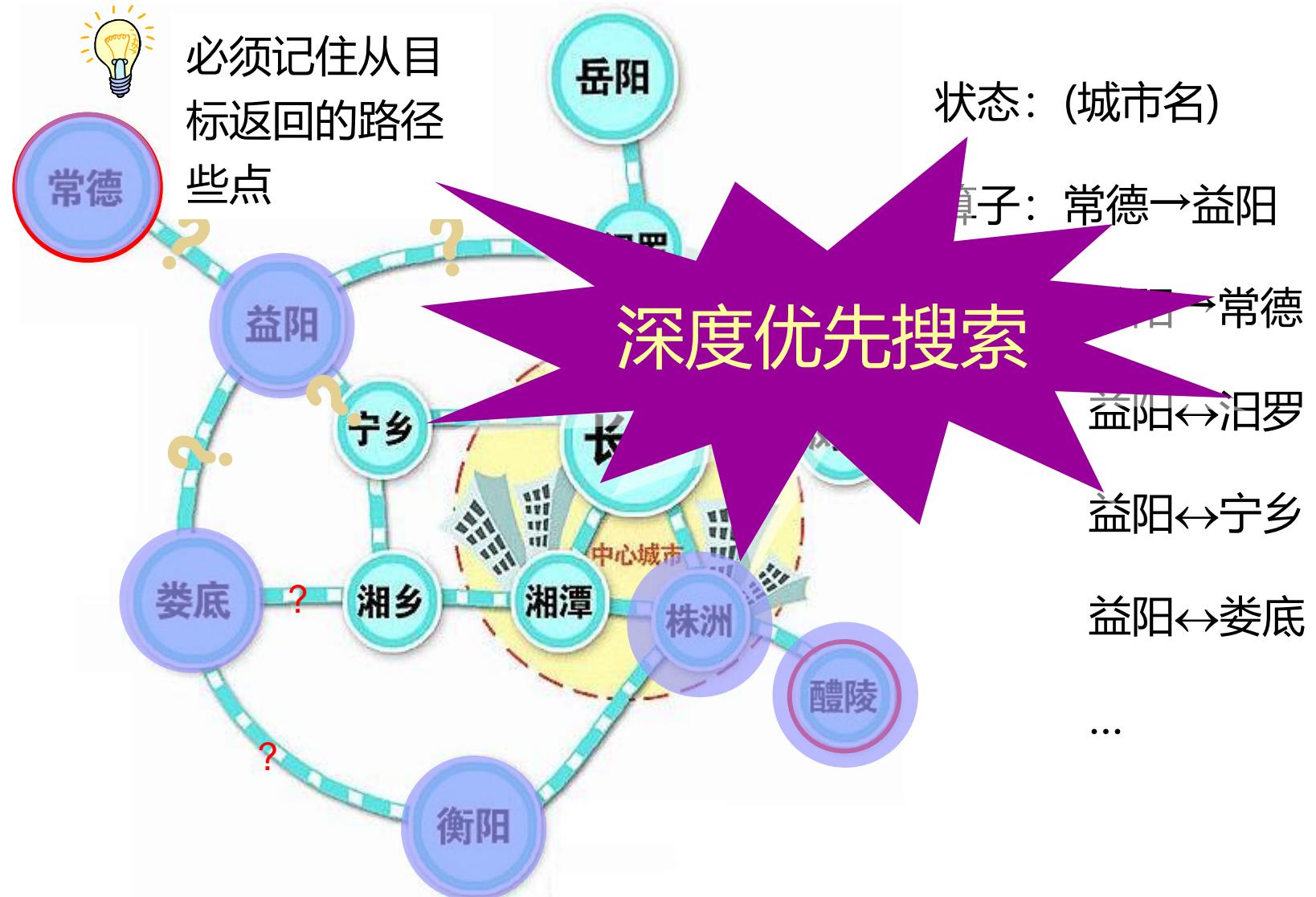
- Fully observable
- Deterministic
- Static
- Discrete
- Single agent

# 树搜索与图搜索

- 树是无圈连通图，每个节点只有一个父节点
- 树搜索不检查重复状态
- 图搜索大多比树搜索高效



# 图的搜索过程



# 图的搜索过程



必须记住下一步还可以走哪些点

OPEN表(记录还没有扩展的点)



必须记住哪些点走过了

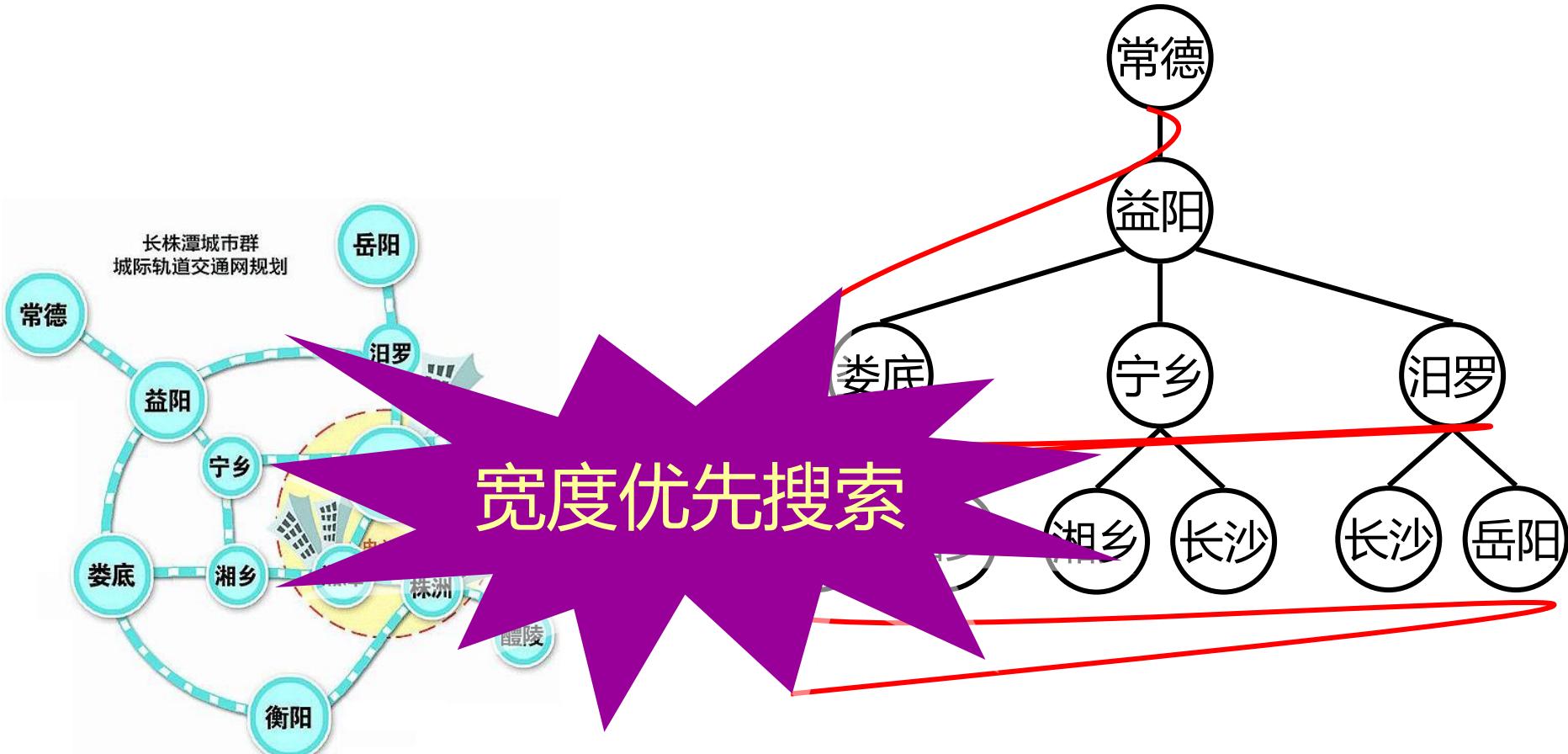
CLOSED表(记录已经扩展的点)



必须记住从目标返回的路径

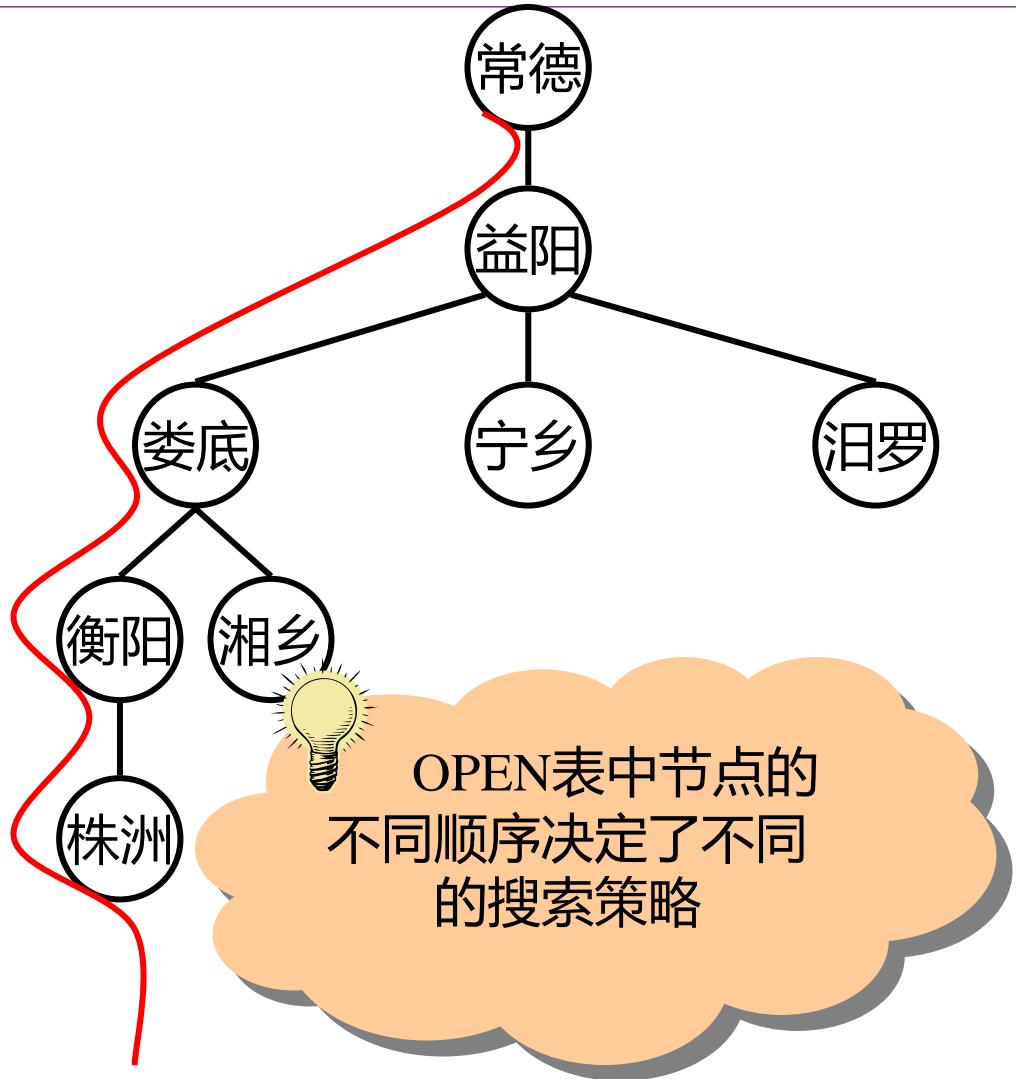
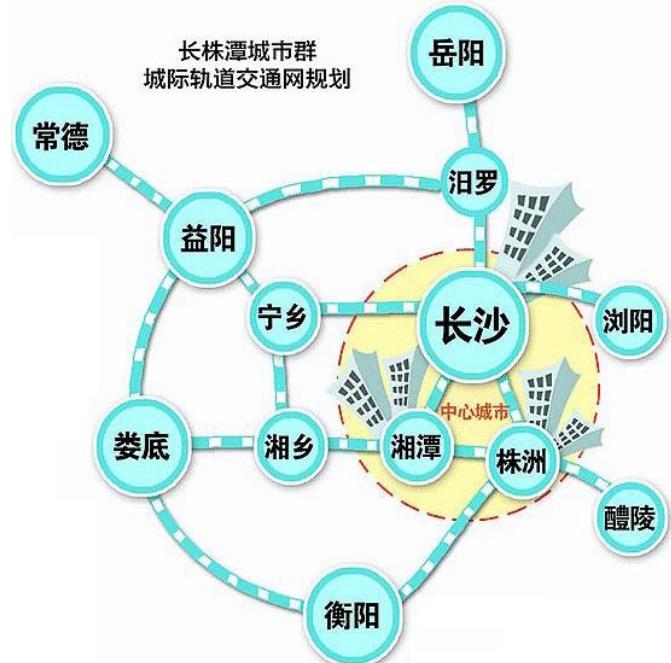
每个表示状态的节点结构中必须有指向父节点的指针

# 图的搜索过程



OPEN: 常德 益阳 娄底 宁乡 汝罗 衡阳 湘乡 长沙 岳阳

# 图的搜索过程



OPEN: 常德 益阳 娄底 衡阳 株洲 湘乡 宁乡 汝罗

# 图搜索策略

- OPEN 表：存放所有待扩展（即已访问但未扩展）的节点的集合
- CLOSED 表：存放已扩展节点
- 如果新生成的节点在CLOSE 表或者OPEN 表中，它被丢弃而不是加入 OPEN 表中

状态节点	父节点

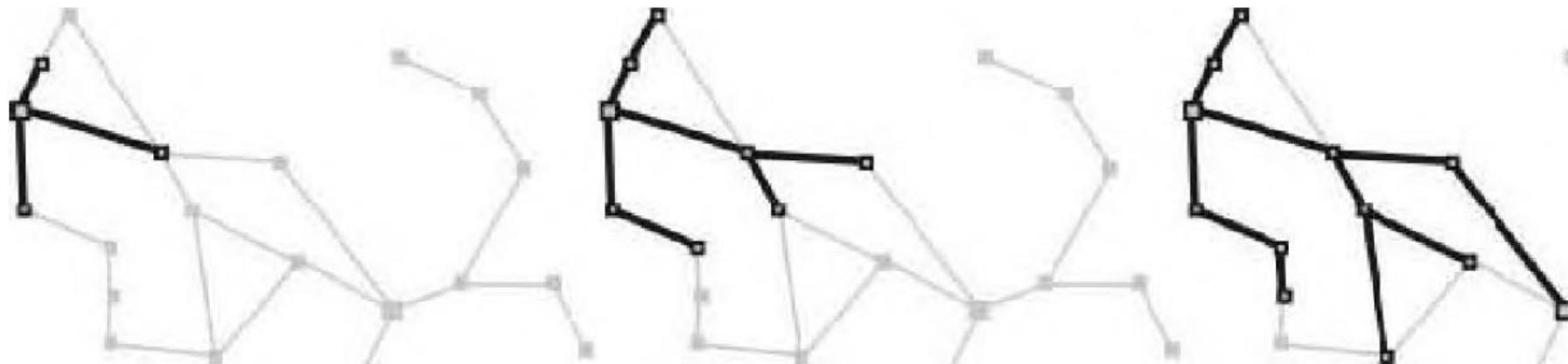
OPEN表

编号	状态节点	父节点

CLOSED表

# 图搜索策略

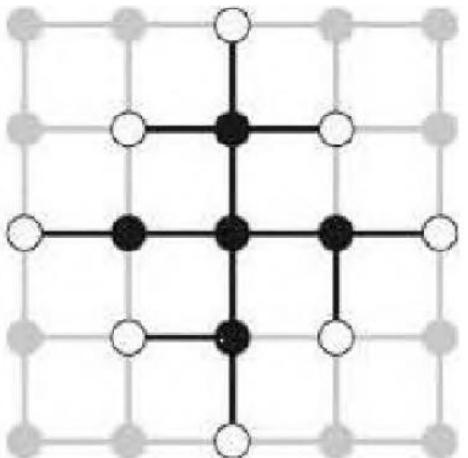
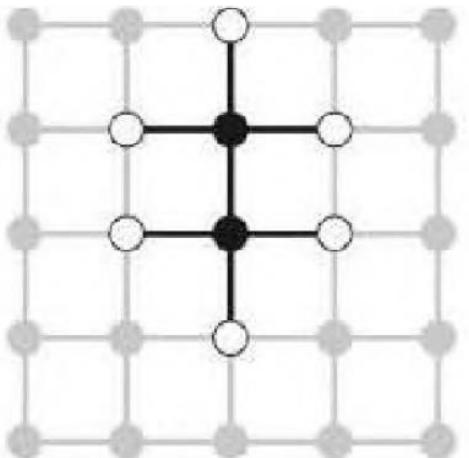
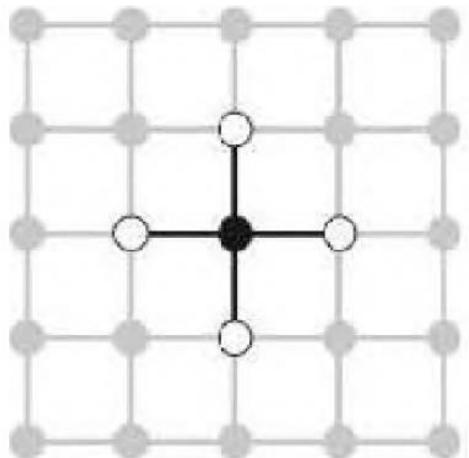
- 通过图搜索算法构造的搜索树中每个状态至多包含一个副本，可以在状态空间中生长一棵树



- OPEN表（边缘）将状态空间图划分成了已探索区域和未探索区域，从初始状态出发至任一未被探索状态都必须通过OPEN表中的节点

# 图搜索策略

- OPEN表（边缘）将状态空间图划分成了已探索区域和未探索区域，从初始状态出发至任一未被探索状态都必须通过OPEN表中的节点
- 每个步骤要么将一个状态从边缘变为已探索区域（从OPEN表进入CLOSE表），要么将未探索区域变为边缘（新扩展节点进入OPEN表）



# 图搜索策略

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial stale of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```



# 图搜索策略

- 图搜索策略是一种在图中寻找路径的方法。初始节点和目标节点分别代表初始状态集合和满足终止条件的状态集合。求得把一个状态变换为另一个状态的行动序列问题就等价于求得图中的一条路径问题。
- 图搜索过程
  1. 建立一个只含有起始节点S的搜索图G，把S放到一个叫做OPEN的未扩展节点表中。
  2. 建立一个叫做CLOSED的已扩展节点表，其初始为空表。
  3. LOOP：若OPEN表是空表，则失败退出。
  4. 选择OPEN表上的第一个节点，把它从OPEN表移出并放进CLOSED表中，称此节点为节点n。



# 图搜索策略

- 图搜索过程 (续)

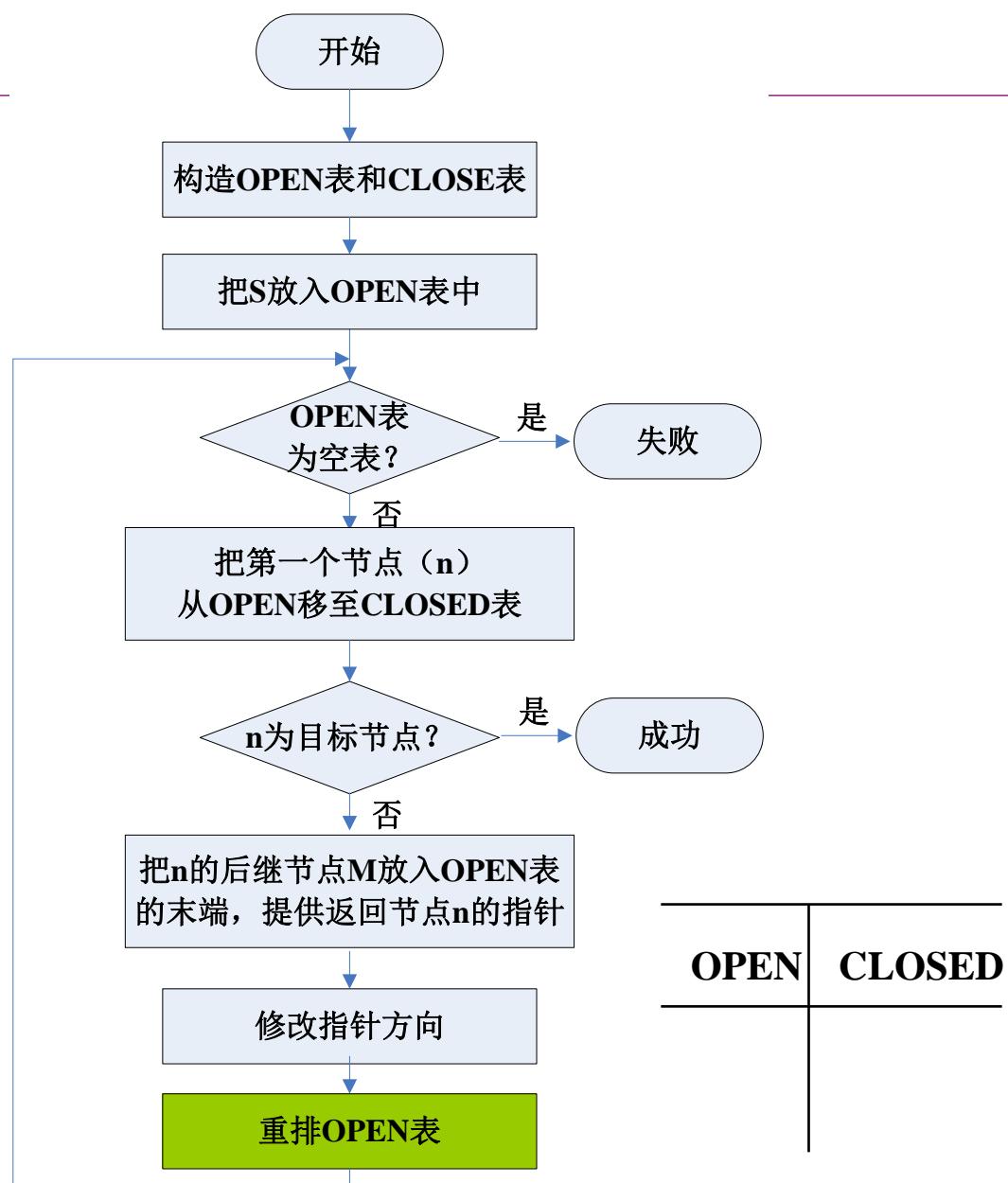
5. 若n为一目标节点，则有解并成功退出，此解是追踪图G中沿着指针从n到S这条路径而得到的（指针将在第(7)步中设置）。
6. 扩展节点n，同时生成不是n的祖先的那些后继节点的集合M。把M的这些成员作为n的后继节点添入图G中
7. 对那些未曾在OPEN表或CLOSED表中出现过的M成员，设置一个通向n的指针，把M的这些成员加进OPEN表。
8. (可选) 对已经在OPEN或CLOSED表上的每一个M成员，确定是否需要更改通到n的指针方向。对已在CLOSED表上的每个M成员，确定是否需要更改图G中通向它的每个后裔节点的指针方向。
9. 按某一任意方式或某个探试值，重排OPEN表
10. GO LOOP。

# 图搜索策略



子节点      子节点

节点数据结构图



- 4种途径评价搜索求解算法的性能
  - 完备性：当问题有解时，算法是否能保证找到解？
  - 最优性：搜索策略是否能找到最优解？
  - 时间复杂度：找到解需要多长时间？
  - 空间复杂度：在执行搜索的过程中需要多少内存？
- 通常考虑
  - 状态空间图的大小： $|V|+|E|$
  - b：分支因子（任意节点的最多后继数）
  - d：目标节点所在的最浅深度（从根节点到目标状态的步数）
  - m：状态空间中任意路径的最大长度

- 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
  - 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Frontier list
- Closed list
  - 存放所有已经被扩展的节点(closed nodes)
  - Not necessary for tree search



# 搜索分类

- 盲目搜索（无信息的搜索）Uninformed

盲目搜索是按照**预定的控制策略**进行搜索，在搜索的过程中获得的中间信息不被用来改进控制策略。这种搜索方法不考虑问题本身的特性，仅仅是教条地按照预定路线前进，具有盲目性，效率也不高，不适于复杂问题的求解。

- 启发式搜索（有信息的搜索）Informed

启发式搜索在搜索中加入了与问题有关的**启发式信息**，用以指导搜索朝着最有希望的方向前进，加速问题的求解过程并找到最优解。

## 盲目搜索

### Blind Search Strategies

1. Breadth-First search
2. Depth-First search
3. Uniform-Cost search

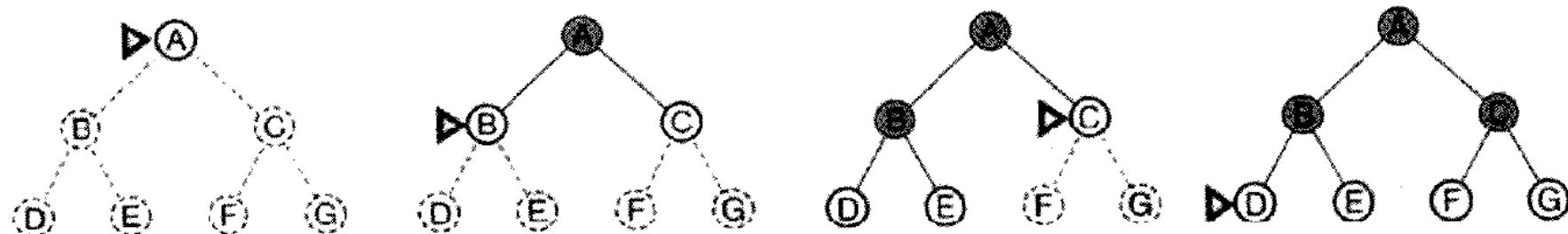


# 盲目搜索（无信息的搜索）

- 无须重新安排OPEN表的搜索叫做无信息搜索或盲目搜索。
- 盲目搜索以节点扩展的次序来分类：
  - 宽度优先搜索
  - 深度优先搜索
  - 有界深度优先搜索
  - 等代价搜索
  - .....
- 盲目搜索只适用于求解比较简单的问题。
- 深度优先搜索中的**有界深度优先搜索**具有一定的启发性。

# 宽度优先搜索 (Breadth-first Search, BFS)

- 宽度优先搜索(Breadth-first search)又称为**广度优先搜索**。这种搜索是逐层进行的，在对下一层的任一节点进行考察之前，必须完成本层所有节点的搜索。
- 宽度优先搜索的基本思想是：从初始节点S开始，逐层地对节点进行扩展并考察它是否为目标节点，在第n层地节点没有全部展开并考察之前，不对n+1层地节点进行扩展
- OPEN表中的节点总是按进入的先后顺序排列，先进入的节点排在前面，后进入的节点排在后面。

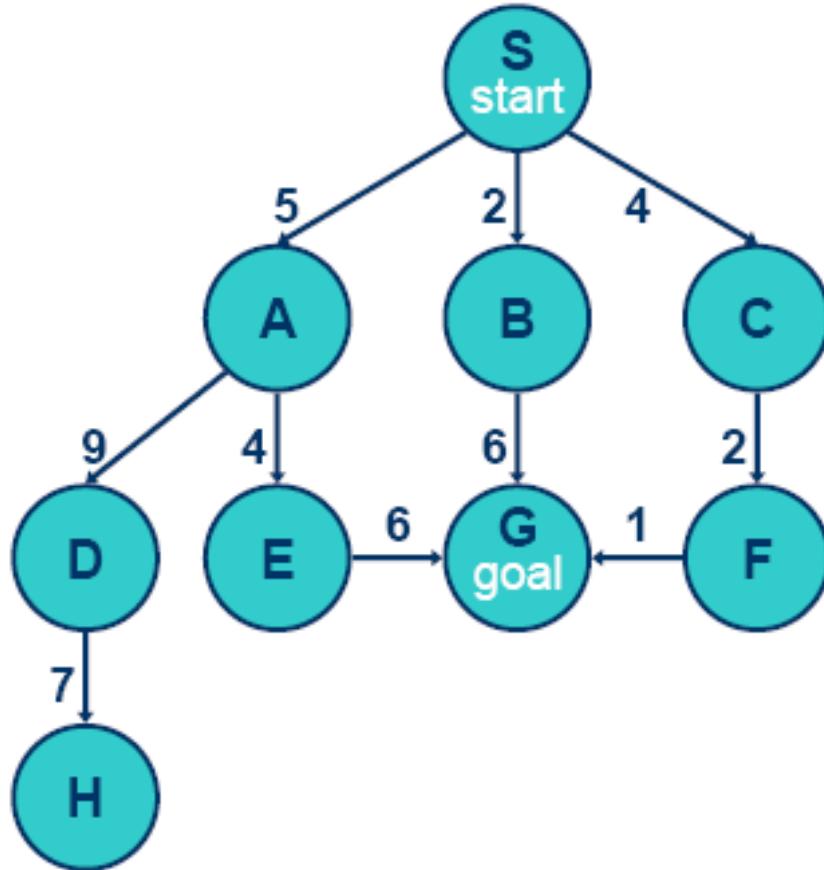


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# of nodes tested: 0, expanded: 0

Expnd. node	Open list
	{S}

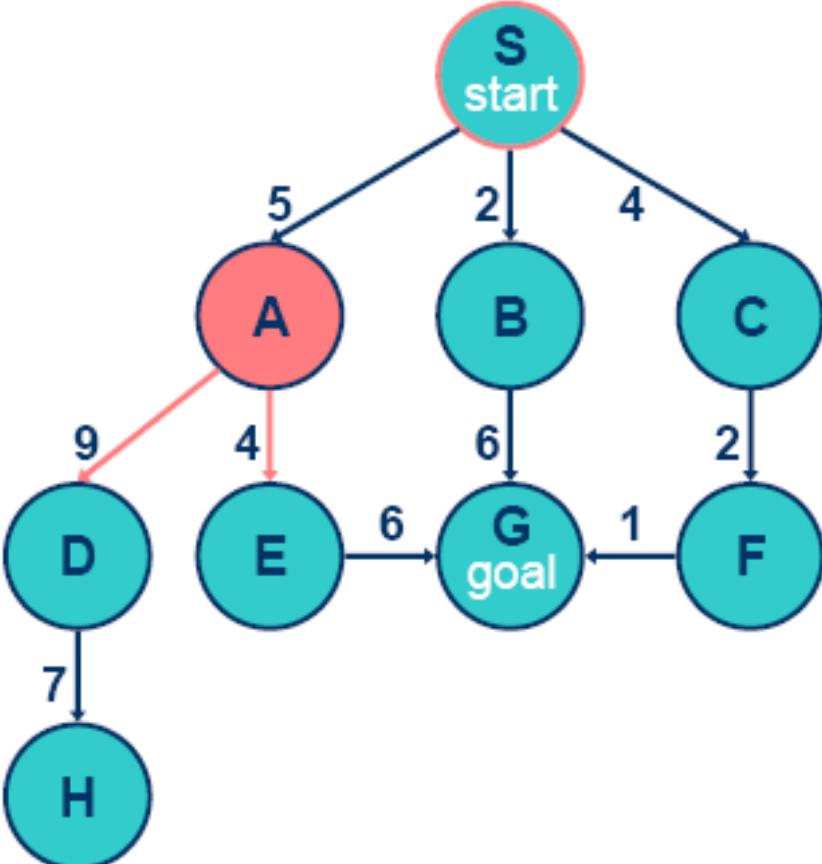


# 举例 (Example: BFS)

**generalSearch(problem, Queue)**

# of nodes tested: 2, expanded: 2

Expnd. node	Open list
	{S}

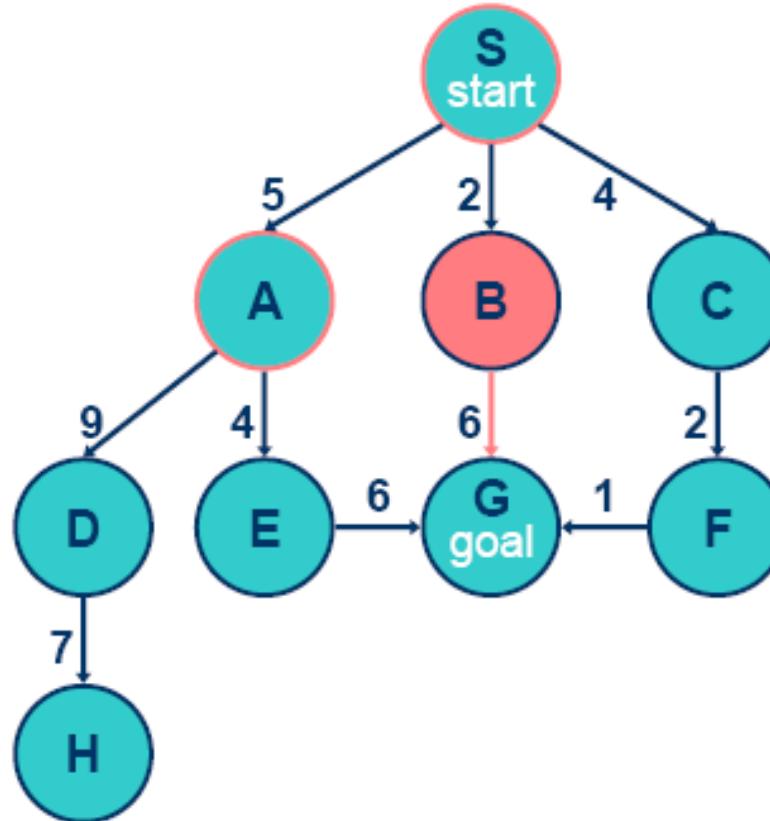


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# of nodes tested: 3, expanded: 3

Exnd. node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}

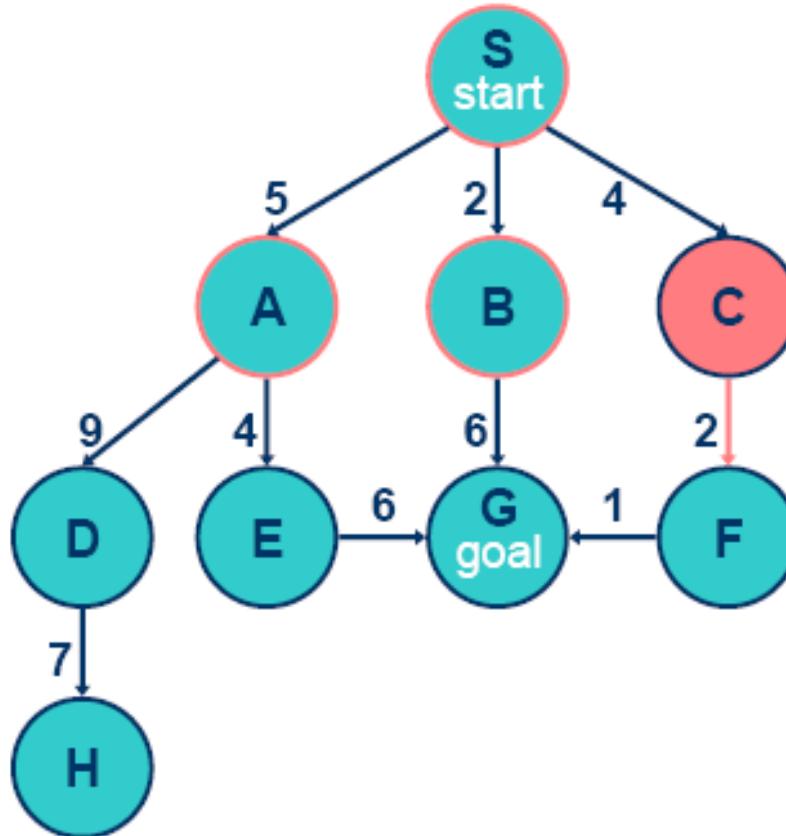


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# of nodes tested: 4, expanded: 4

Exnd. node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}

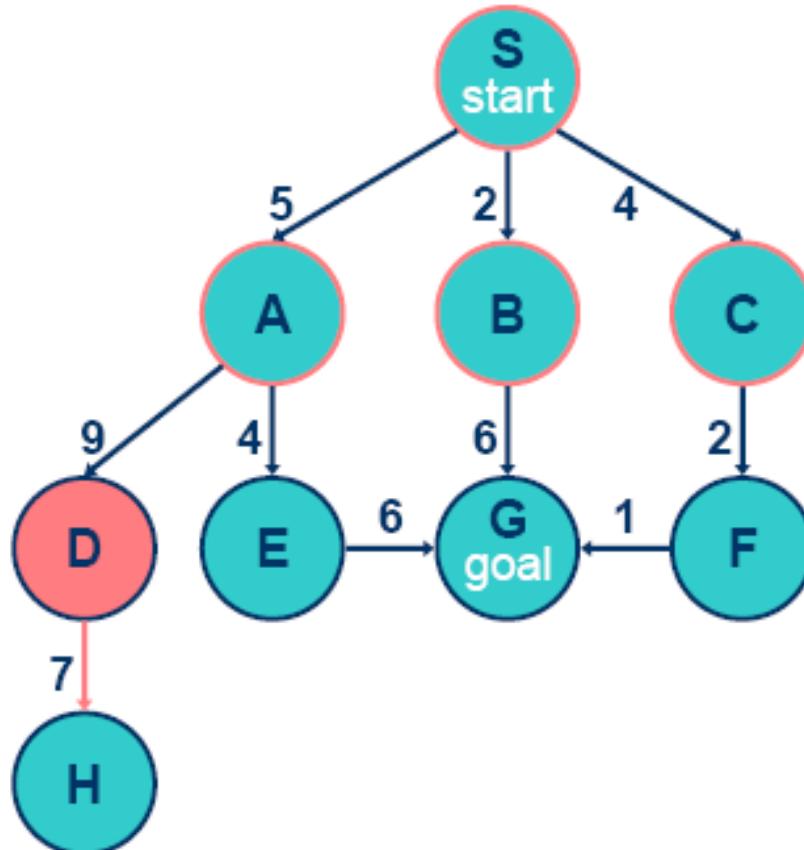


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# of nodes tested: 5, expanded: 5

Exnd. node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D not goal	{E,G,F,H }

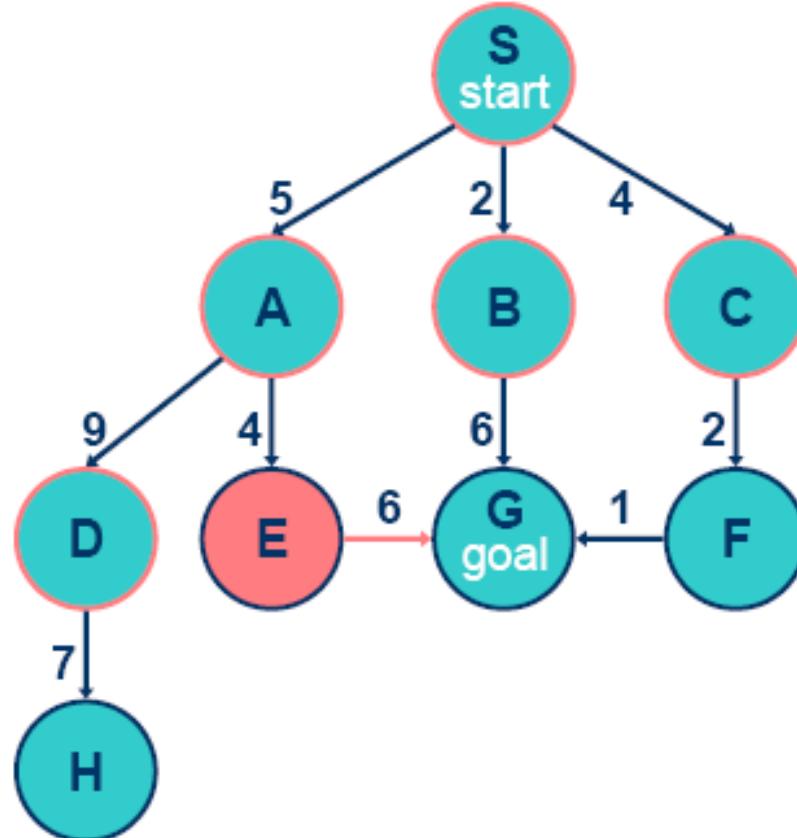


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# of nodes tested: 6, expanded: 6

Exnd. 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 not goal	{G,F,H,G }

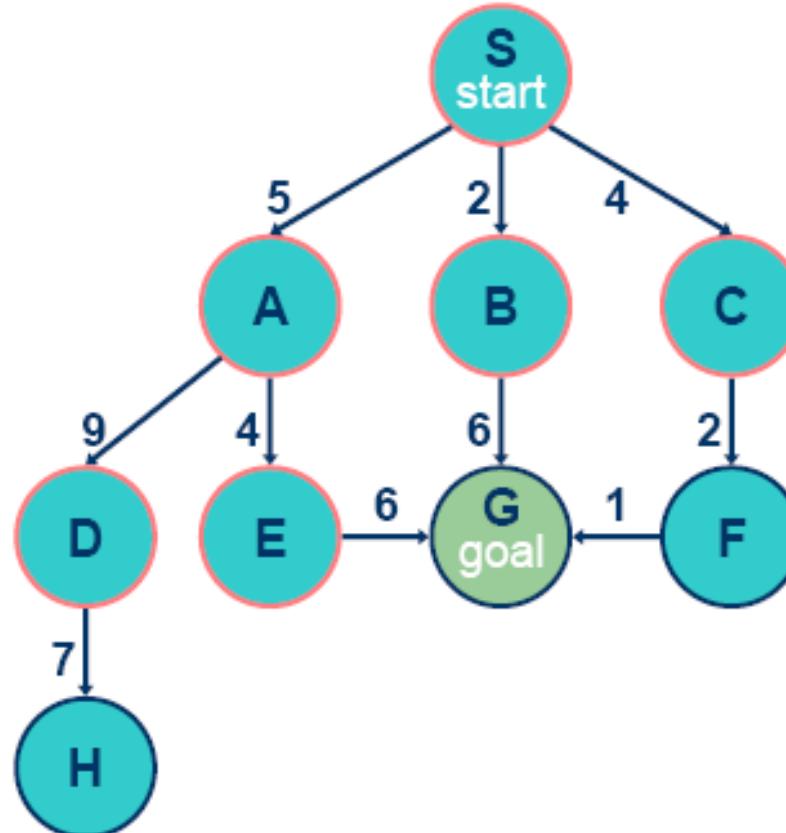


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# 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 goal	{F,H,G} no expand

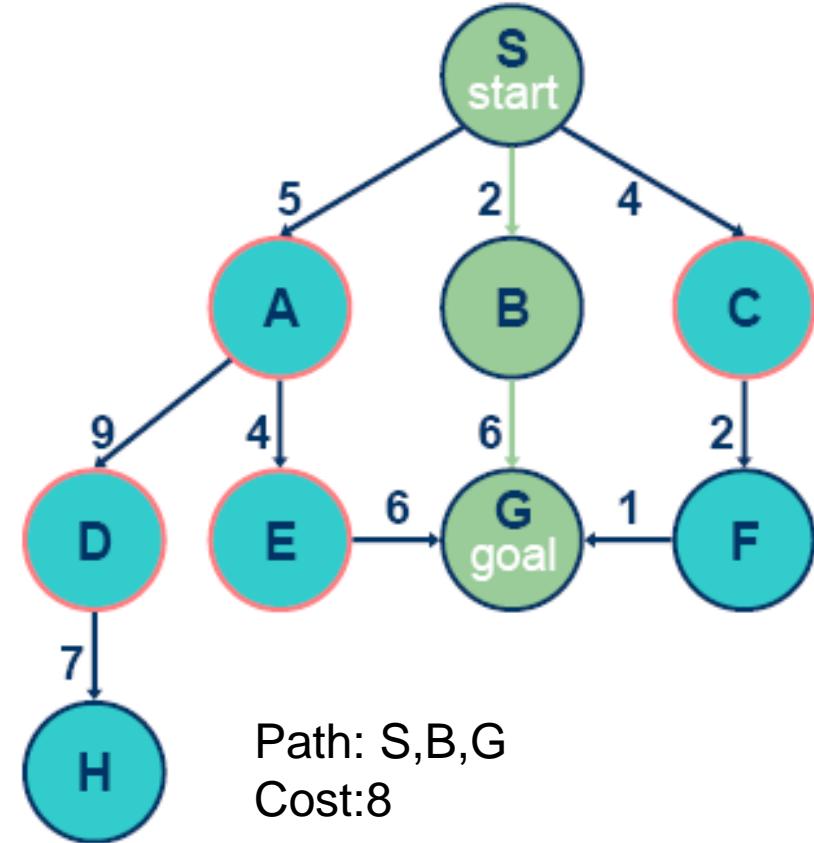


# 举例 (Example: BFS)

**generalSearch(problem, Stack)**

# 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}



# 宽度优先搜索算法性能

- 完备性：完备
- 最优性：当分支因子 $b$ 有限时，如果路径代价是基于节点深度的非递减函数，宽度优先搜索是最优的
- 时间复杂度： $O(b^d)$
- 空间复杂度： $O(b^d)$

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^7$	1.1 seconds	1 gigabyte
8	$10^9$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes



# 宽度优先搜索算法性能

- 完备性 (Completeness)
  - *Does it always find a solution if one exists?*
  - YES
    - If shallowest goal node is at some finite depth  $d$
- 时间复杂度 (Time complexity)
  - 假设每个状态有  $b$  个后继, 目标节点所在深度为  $d$ 
    - 根节点有  $b$  个后继, 这  $b$  个节点每个又有  $b$  个后继, 即  $b^2$
    - 最坏的情况: 扩展除  $d$  层最后一个节点外的所有节点
    - 总共扩展操作的次数:
$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$



# 宽度优先搜索算法性能

- 空间复杂度 (Space complexity)
  - 每个节点都需要存储
- 最优度 (Optimality)
  - 如果每步扩展的代价相同时，宽度优先搜索能找到最优解

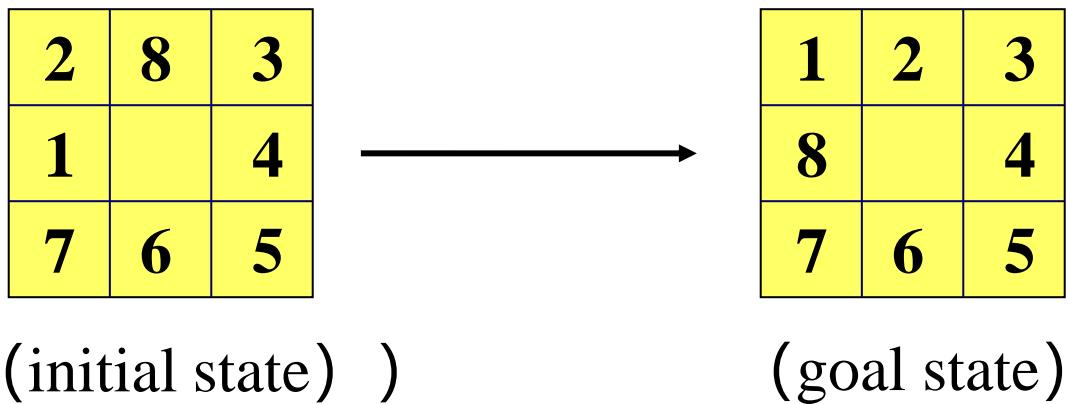
# 宽度优先搜索算法性能

- Two lessons
  - 指数级的时间消耗，内存消耗是比执行时间消耗更大的问题
  - 指数级复杂度的搜索问题无法通过无信息搜索完成，除非其搜索空间非常小
- 假设每个节点有 $b=10$ 个后继节点，目标节点在 $d$ 层

Depth	Nodes	Time	Memory
2	110	1.1 milliseconds	107 kilobytes
4	11,110	111 milliseconds	10.6 megabytes
6	$10^6$	11 seconds	1 gigabytes
8	$10^8$	19 minutes	103 gigabytes
10	$10^{10}$	31 hours	10 terabytes
12	$10^{12}$	129 days	1 petabytes
14	$10^{14}$	35 years	99 petabytes
16	$10^{16}$	3,500 years	10 exabytes

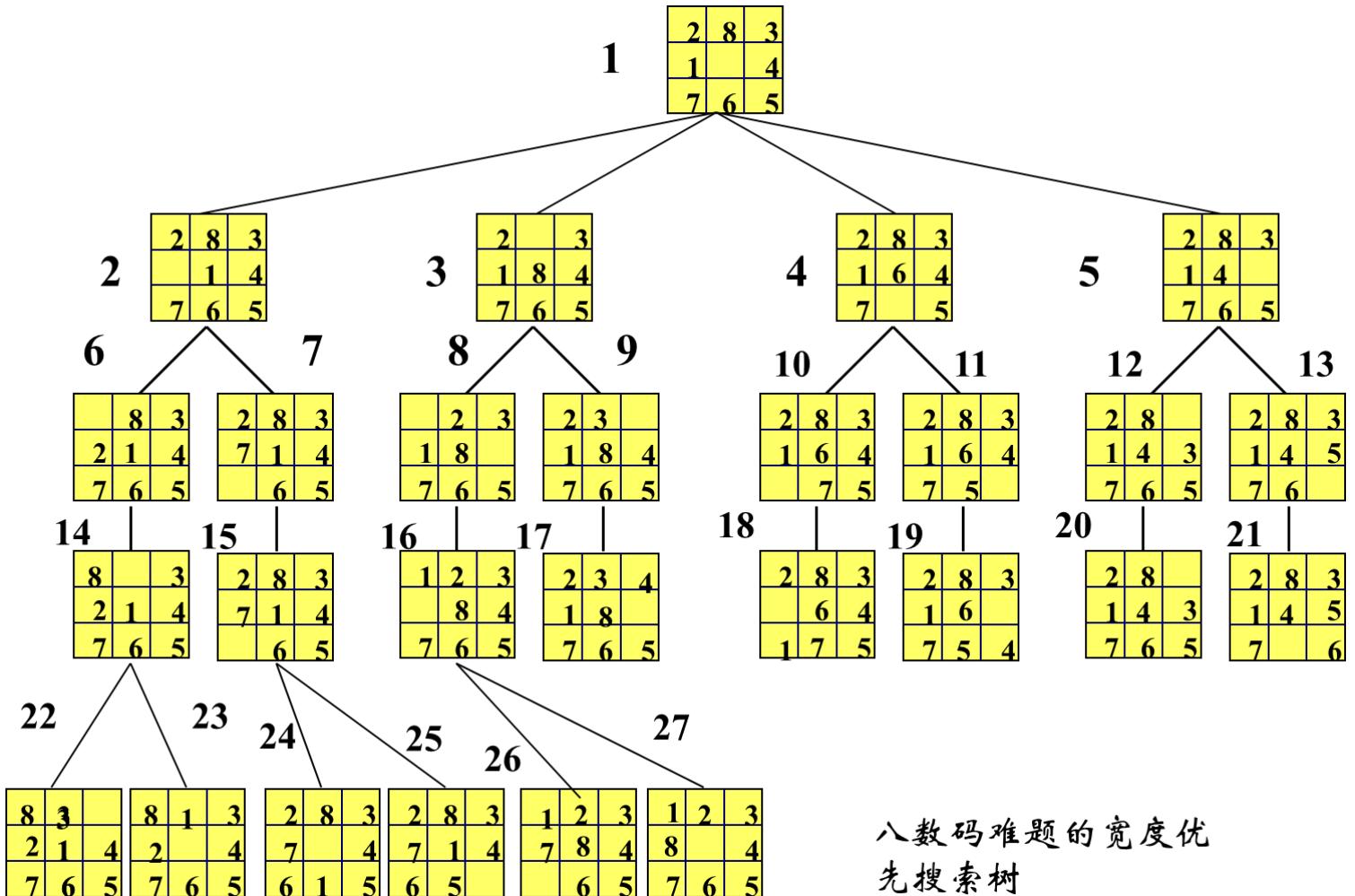
# 举例：8数码问题

- Example: 8-puzzle problem



规定：将牌移入空格的顺序为：从空格左边开始顺时针旋转。不许斜向移动，也不返回先辈节点。  
从下图可见，要扩展26个节点，共生成46个节点之后才求得解（目标节点）。

# 举例：8数码问题





# 深度优先搜索 (Depth-first search, DFS)

- 深度优先搜索(depth-first search)也是一种经典的无信息搜索策略。它的特点是优先扩展最新产生的节点。
- 深度优先搜索的基本思想是：从初始节点S开始，在其子节点中选择一个节点进行考察，若不是目标节点，则再在该子节点中选择一个进行考察，一直如此向下搜索。当到达某个节点，且该子节点既不是目标节点，又不能继续扩展时，才选择其兄弟节点进行考察。
- 深度优先搜索使用FILO队列（栈）



# 深度优先搜索 (Depth-first search, DFS)

- 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
- Implementation: *open* is a LIFO queue (=stack)
  - 防止搜索过程沿着无益的路径扩展下去，往往给出一个节点扩展的最大深度——**深度界限**。
  - 与宽度优先搜索算法最根本的不同在于：将扩展的后继节点放在OPEN表的**前端**。

# 深度优先搜索 (Depth-first search, DFS)

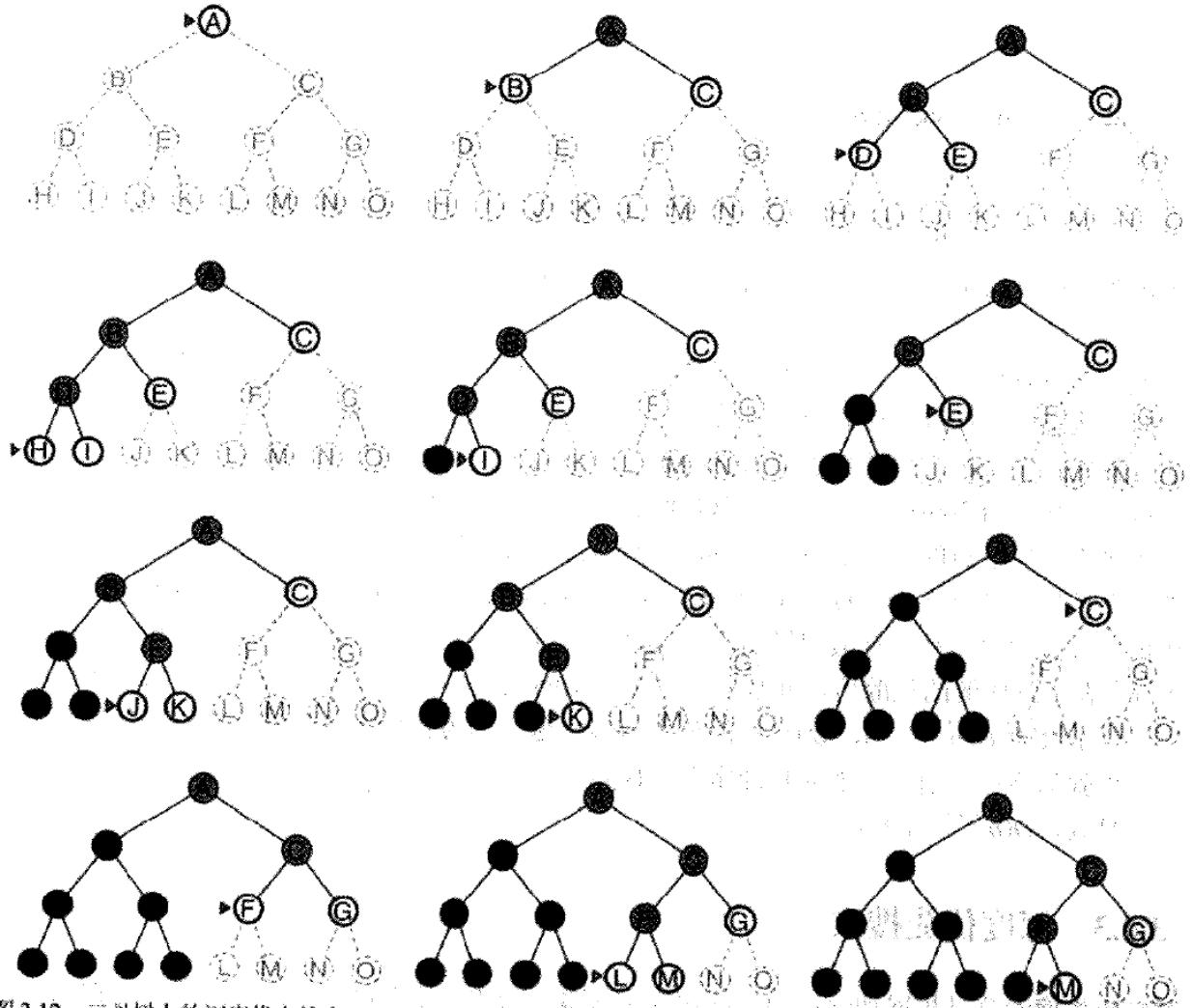


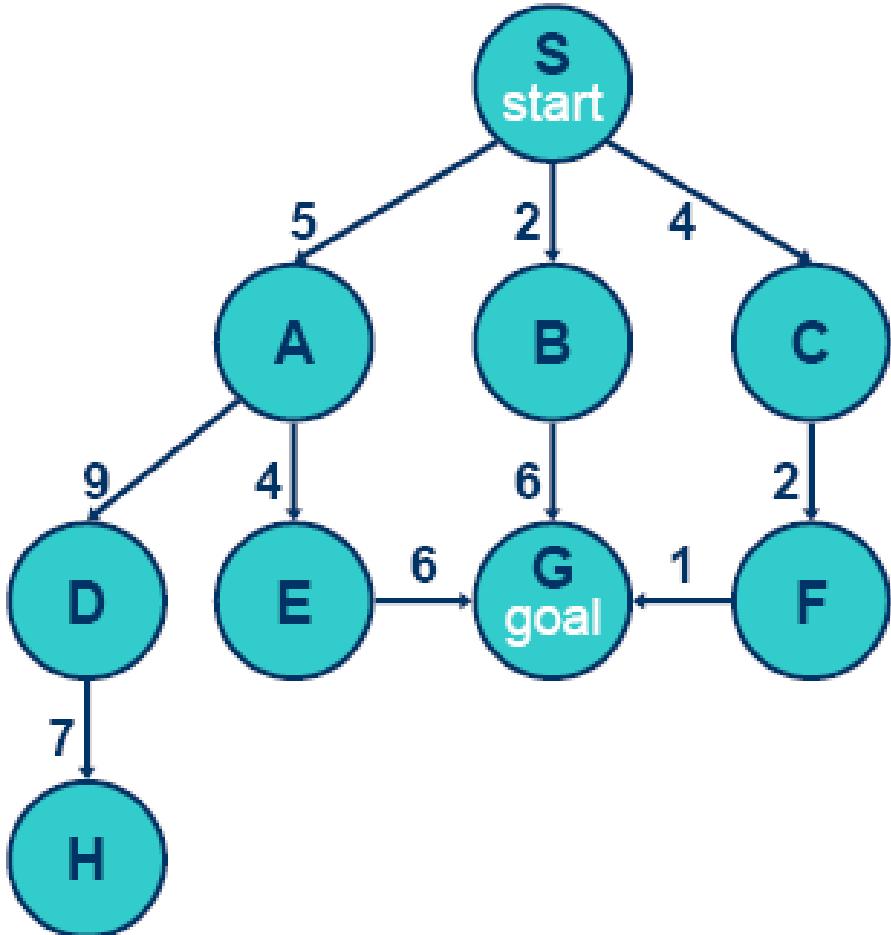
图 2.19 例 2.19 深度优先搜索 (Depth-first search, DFS)

# 举例 (Example: DFS)

**generalSearch(problem, Stack)**

# of nodes tested: 0, expanded: 0

Exnd. node	Open list
	{S}

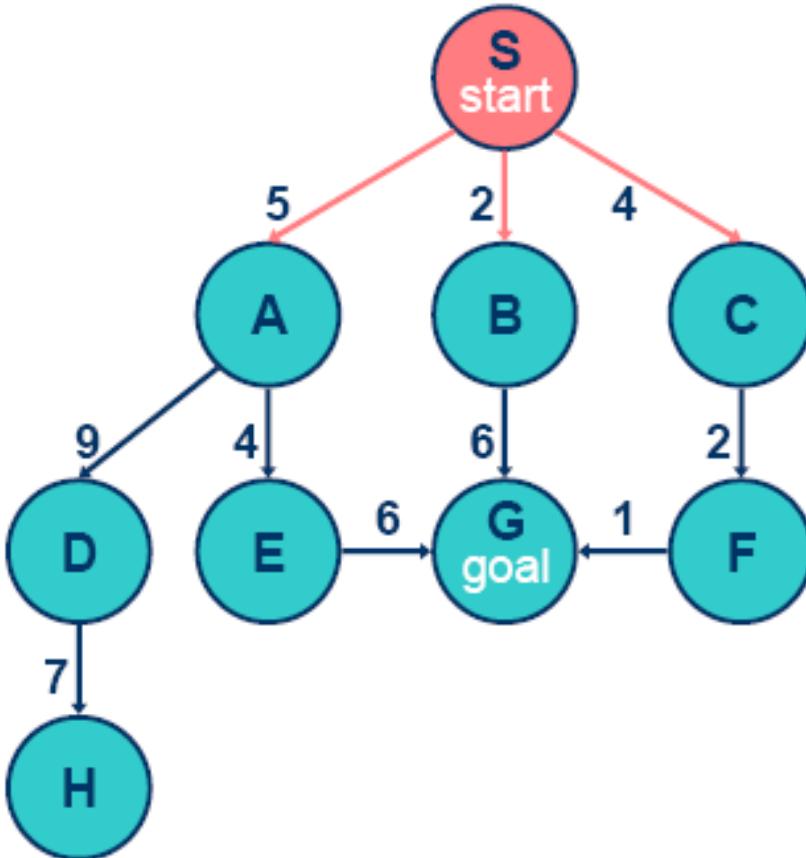


# 举例 (Example: DFS)

**generalSearch(problem, Stack)**

# of nodes tested: 1, expanded: 1

Expnd. node	Open list
	{S}
S not goal	{A,B,C}

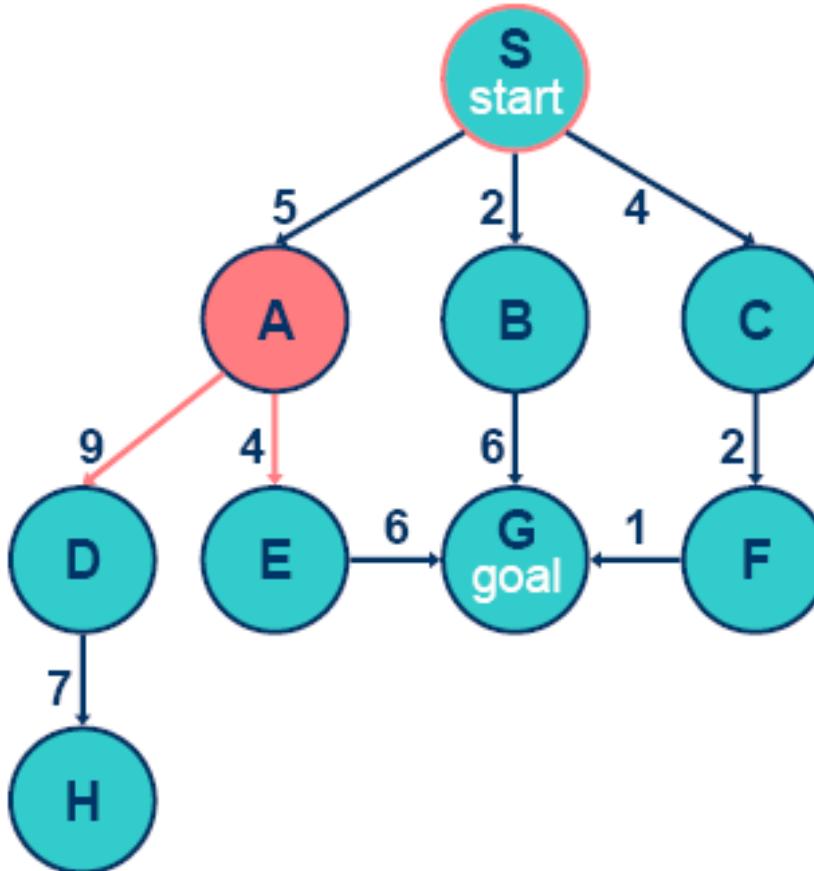


# 举例 (Example: DFS)

**generalSearch(problem, Stack)**

# of nodes tested: 2, expanded: 2

Exnd. node	Open list
	{S}
S	{A,B,C}
A not goal	{D,E,B,C}

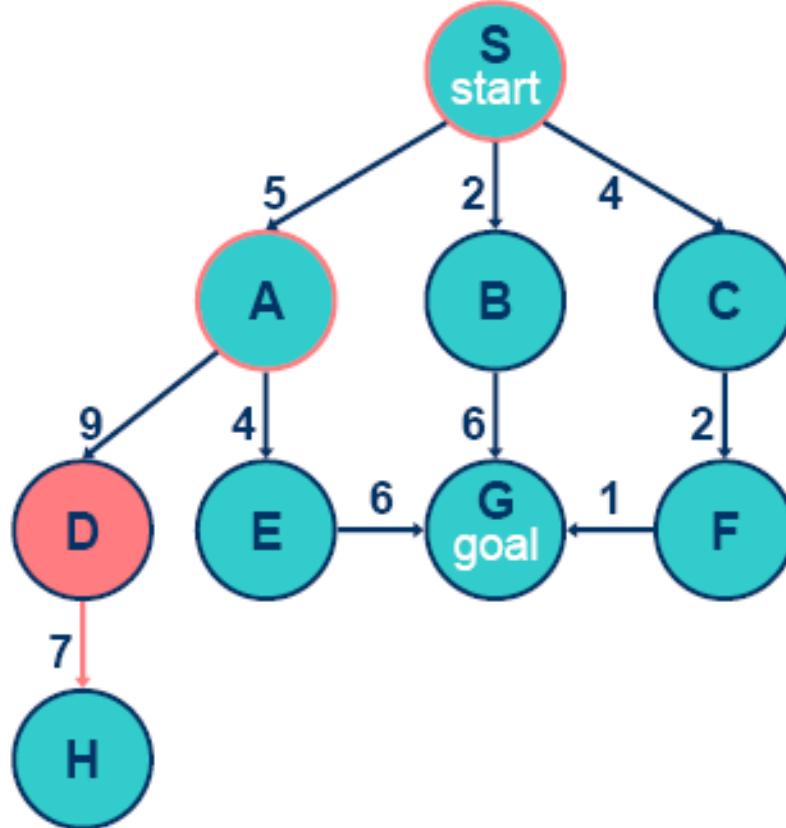


# 举例 (Example: DFS)

**generalSearch(problem, Stack)**

# of nodes tested: 3, expanded: 3

Exnd. node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D not goal	{H,D,E,B,C}

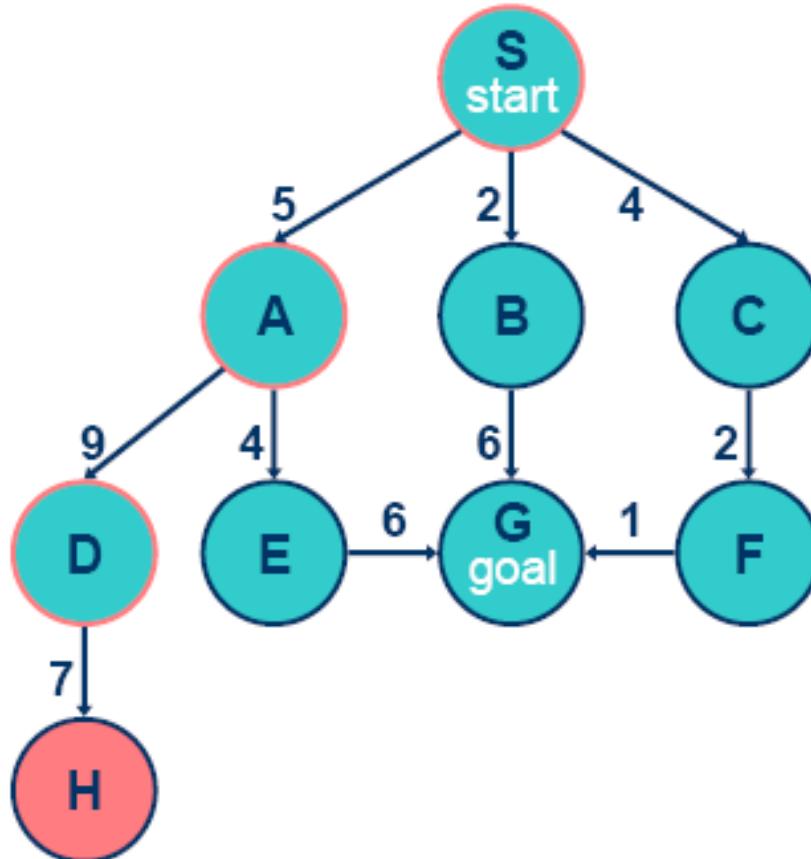


# 举例 (Example: DFS)

**generalSearch(problem, Stack)**

# of nodes tested: 4, expanded: 4

Exnd. node	Open list
	{S:0}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H not goal	{E,B,C}

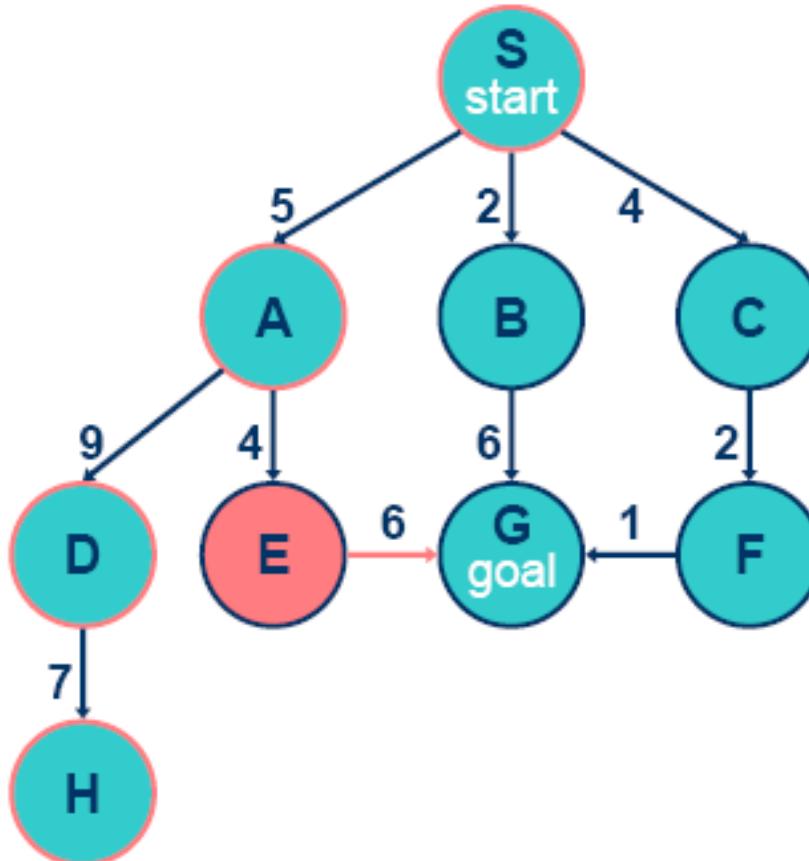


# 举例 (Example: DFS)

**generalSearch(problem, Stack)**

# of nodes tested: 5, expanded: 5

Exnd. node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E not goal	{G, B,C }

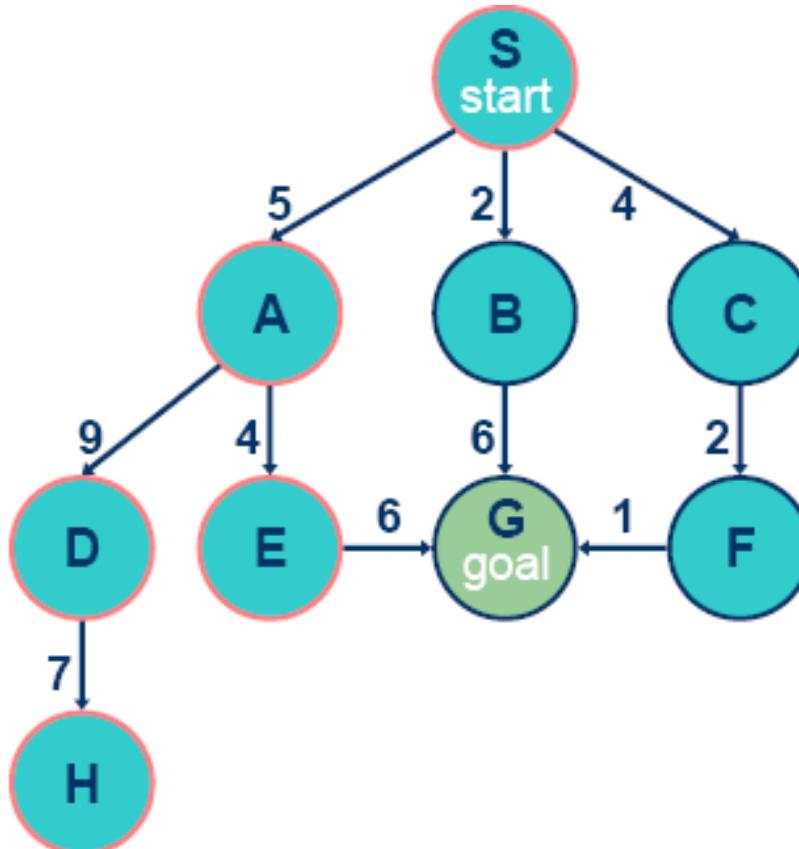


# 举例 (Example: DFS)

**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 goal	{B,C } no expand

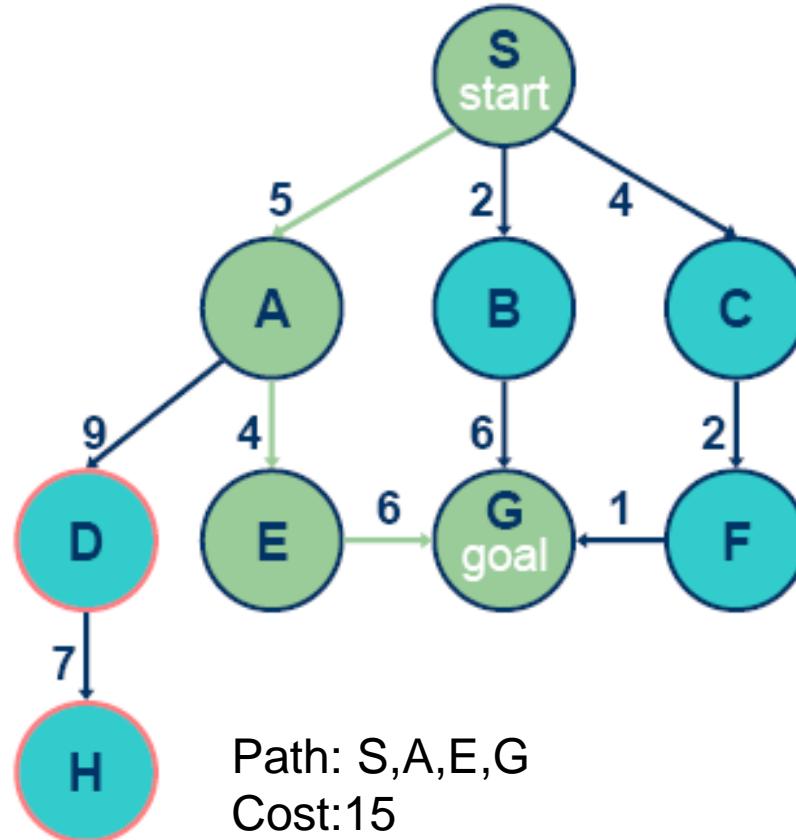


# 举例 (Example: DFS)

**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 }





# 深度优先搜索算法性能

- 完备性：图搜索完备/树搜索不完备
- 最优性：非最优
- 时间复杂度： $O(b^m)$ 
  - M是任一节点的最大深度
- 空间复杂度： $O(b^m)$ 
  - 深度优先搜索只需要存储一条从根节点到叶节点的路径，以及该路径上每个节点的所有未被扩展的兄弟节点

# 深度优先搜索算法性能

- Completeness
  - *Does it always find a solution if one exists?*
  - NO
    - *unless search space is finite and no loops are possible.*
- Time complexity
  - 假设每个状态有  $b$  个后继, 目标节点所在深度为  $d$ , 搜索树的最大深度为  $m$

$$O(b^m)$$

Terrible if  $m$  is much larger than  $d$  (depth of optimal solution)



# 深度优先搜索算法性能

- Space complexity
  - 深度优先搜索对内存的需求比较适中
  - 只需要保存从根到叶的单条路径，包括在这条路径上每个节点的未扩展的兄弟节点
  - 当搜索过程到达了最大深度的时候，所需要的内存最大
  - 假定每个节点的分支系数为 $b$ ，最大深度为 $m$ 的搜索树
    - 保存在内存中的节点的数量包括到达深度 $m$ 时所有未扩展的节点以及正在被考虑的节点。
    - 每个层次上都有 $(b-1)$ 个未扩展的节点，总的内存需要量为 $m(b-1)+1$ 。因此深度优先搜索的空间复杂度是 $b$ 的线性函数 $O(bm)$
- Optimality
  - No



# 有界深度优先搜索

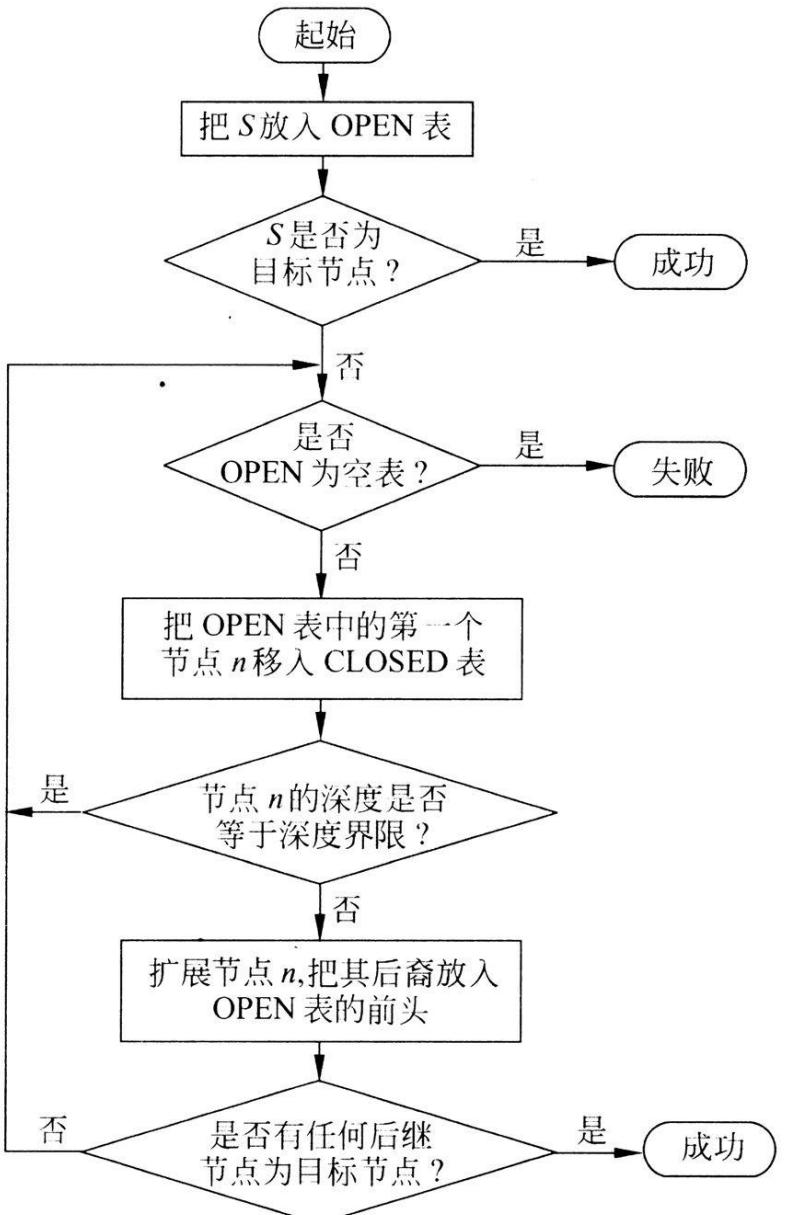
- 有界深度优先搜索(depth-limited search)策略的提出，是为了解决深度优先搜索不完备，有可能陷入无穷分支的死循环而得不到解得问题。
- 有界深度优先搜索的基本思想是：对深度优先搜索引入搜索深度之界限l；当搜索深度达到了深度界限，而尚未出现目标节点时，就换一个分支进行搜索。
- 有界深度优先搜索可以看做特殊的深度优先搜索，深度  $l = \infty$



# 有界深度优先搜索

- 完备性:  $l < d$ , 不完备
- 最优性:  $l > d$ , 非最优
- 时间复杂度:  $O(b^l)$
- 空间复杂度:  $O(bl)$

# 有界深度优先搜索



# 有界深度优先搜索

## ● 有界深度优先搜索的递归实现

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```



# 一致代价搜索 (Uniform Cost Search , UCS)

- 解决哪类问题?

有些问题并不要求有应用算符序列为最少的解，而是要求具有某些特性的解。搜索树中每条连接弧线上的有关代价以及随之而求得的具有**最小代价**的解答路径。

- **一致代价搜索算法**(uniform cost search)优先扩展路径消耗 $g(n)$ 最小的节点，而不是深度最浅的节点，从而实现代价最小的最优搜索。
- 一致代价搜索算法可以看做是宽度优先算法的推广，如果所有的连接弧线具有相等的代价，那么等代价算法就简化为宽度优先搜索算法。



# 一致代价搜索 (Uniform Cost Search , UCS)

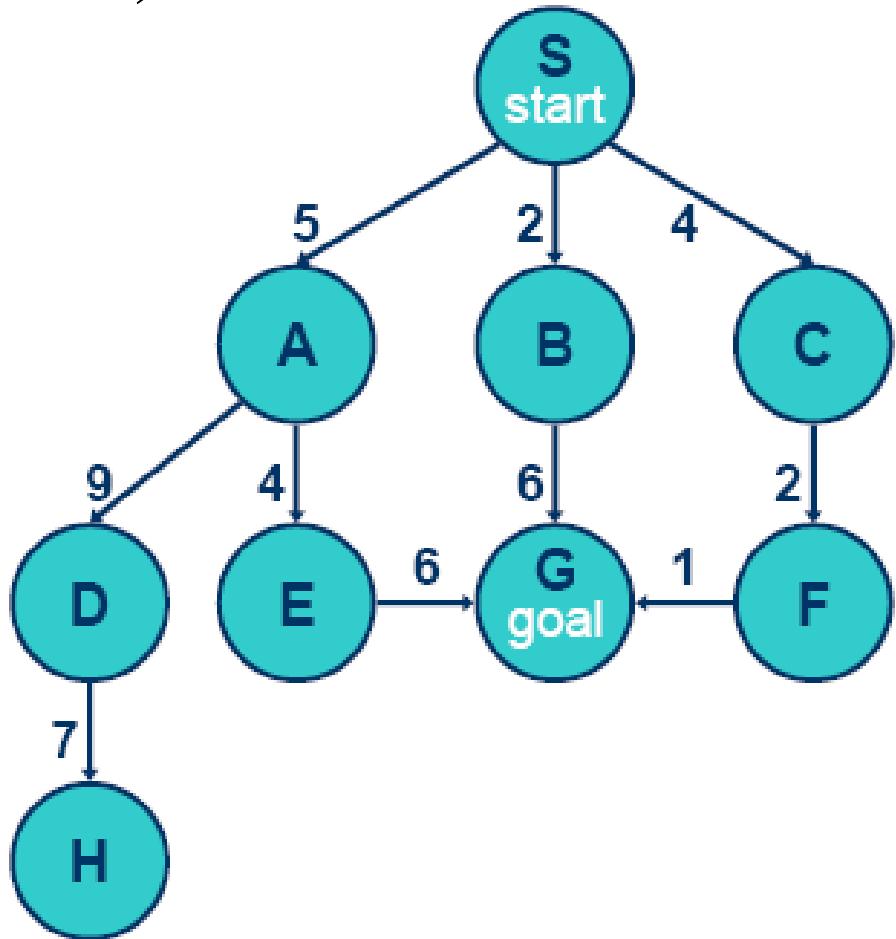
- BFS的扩展版本
  - Expand node with *lowest path cost*
- 1959年由Dijkstra提出，也叫Dijkstra算法
- Let  $g(n) = \text{cost of path from start node } s \text{ to current node } n$
- Implementation:  $\text{open} = \text{Priority Queue(queue ordered by } g\text{)}$ .
- UCS is the same as BFS when all step-costs are equal.

# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 0, expanded: 0

Expnd. node	Open list
	{S}

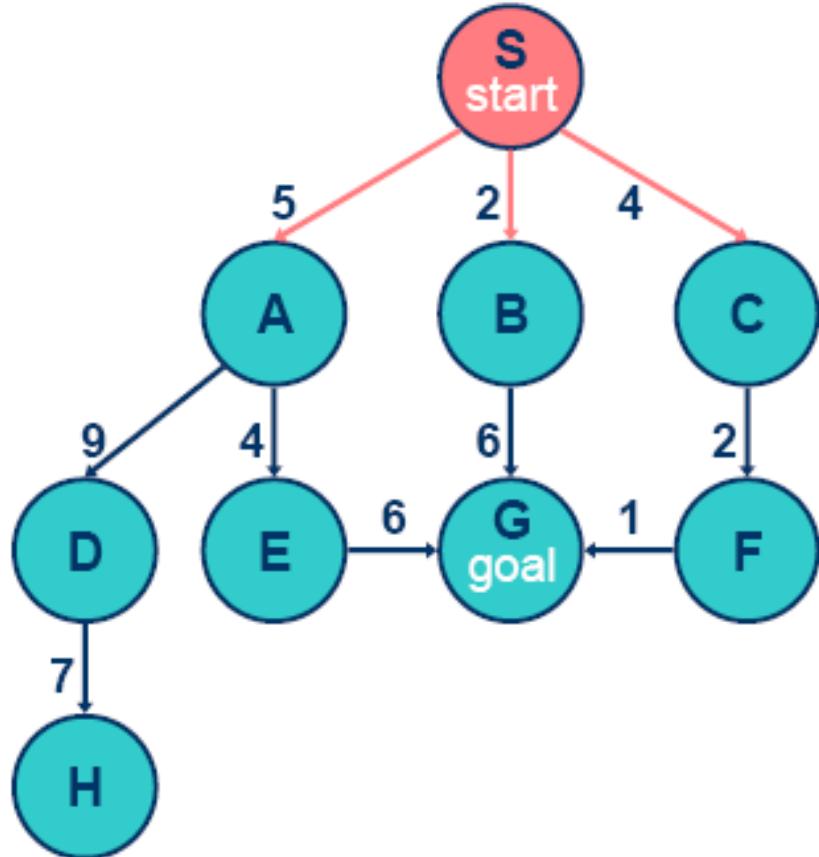


# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 1, expanded: 1

Expnd. node	Open list
	{S:0}
S not goal	{B:2,C:4,A:5}

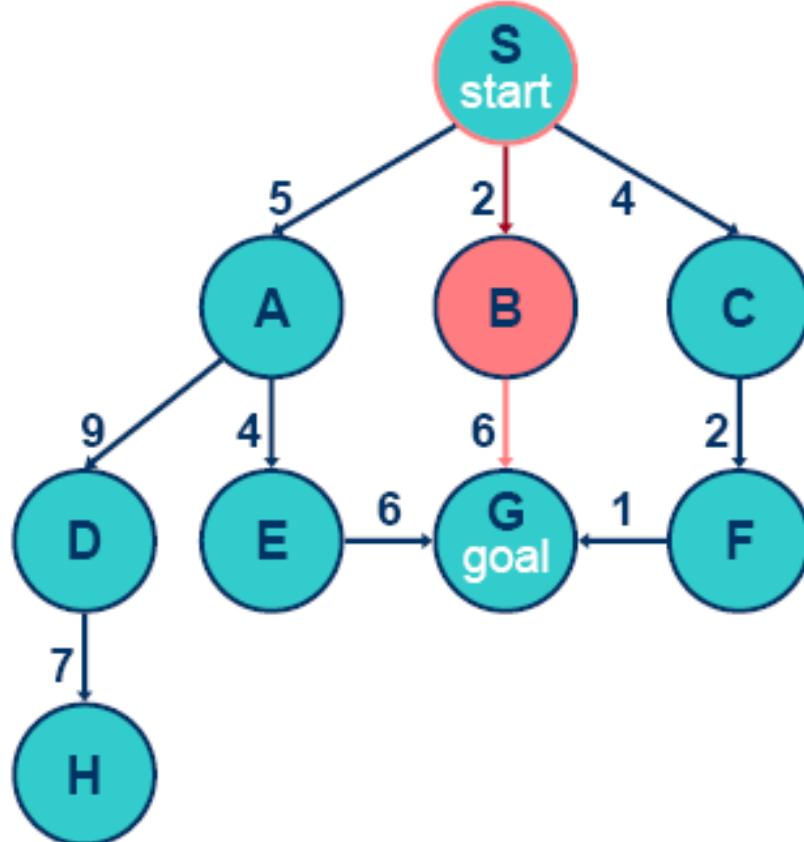


# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 2, expanded: 2

Exnd. node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}

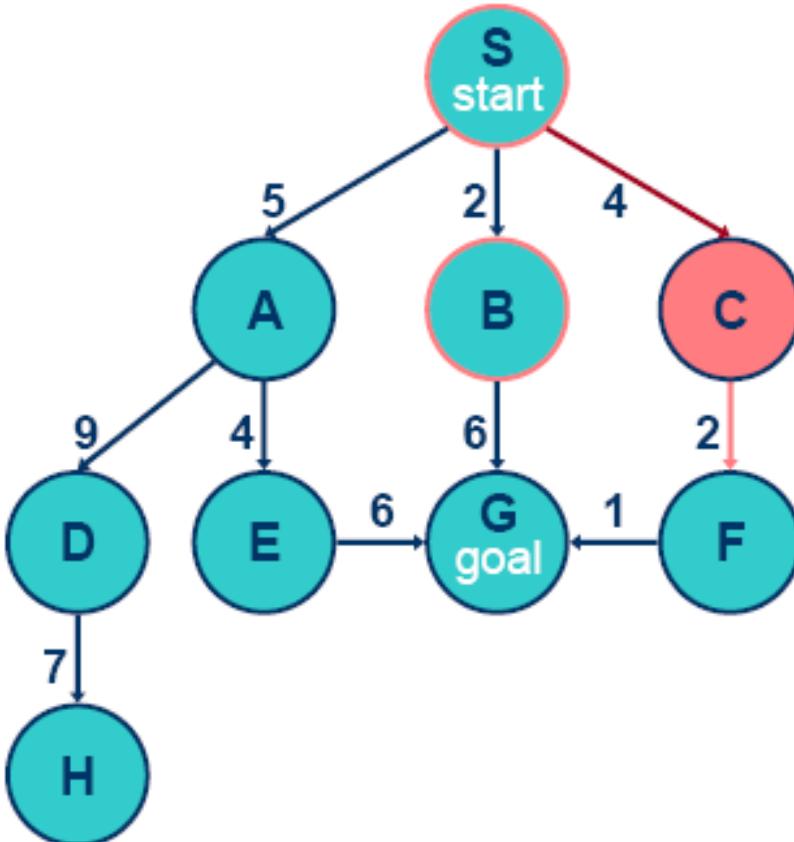


# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 3, expanded: 3

Exnd. node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}

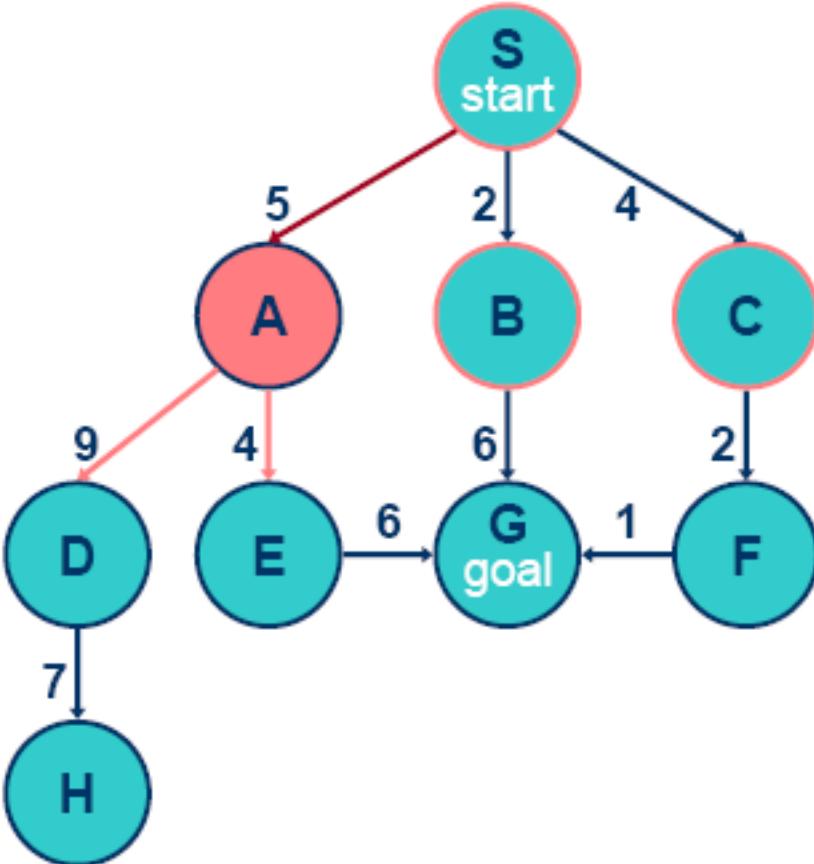


# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 4, expanded: 4

Exnd. 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 not goal	{F:6,G:8,E:5+4, D:5+9}

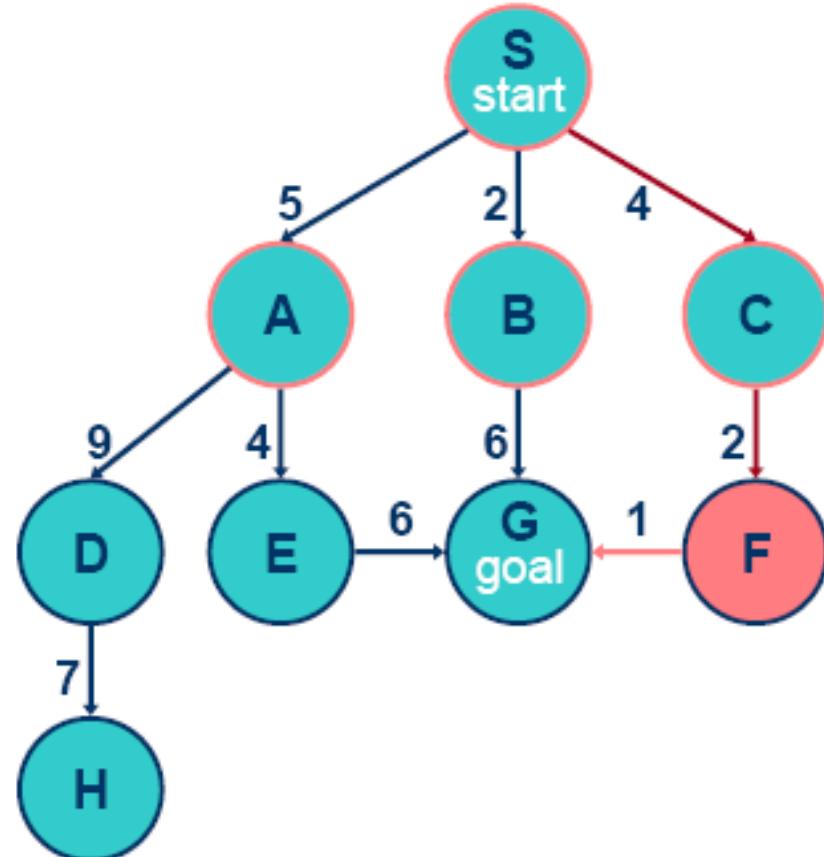


# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 5, expanded: 5

Exnd. 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 not goal	{G:4+2+1,G:8,E:9 ,D:14}

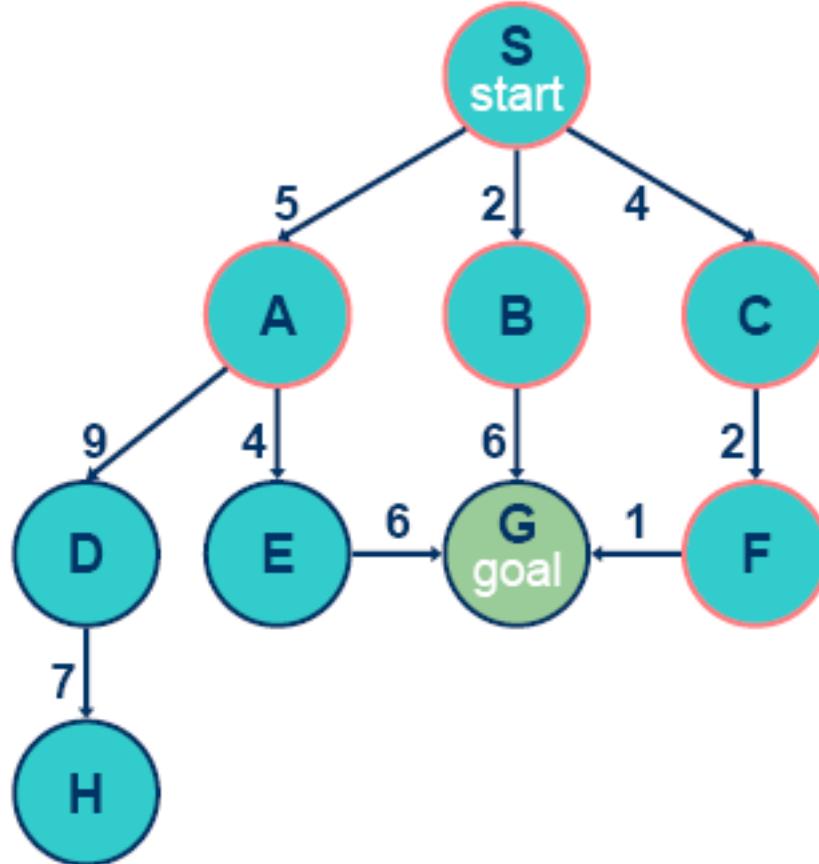


# 举例 (Example: UCS)

**generalSearch(problem, priorityQueue)**

# of nodes tested: 6, expanded: 5

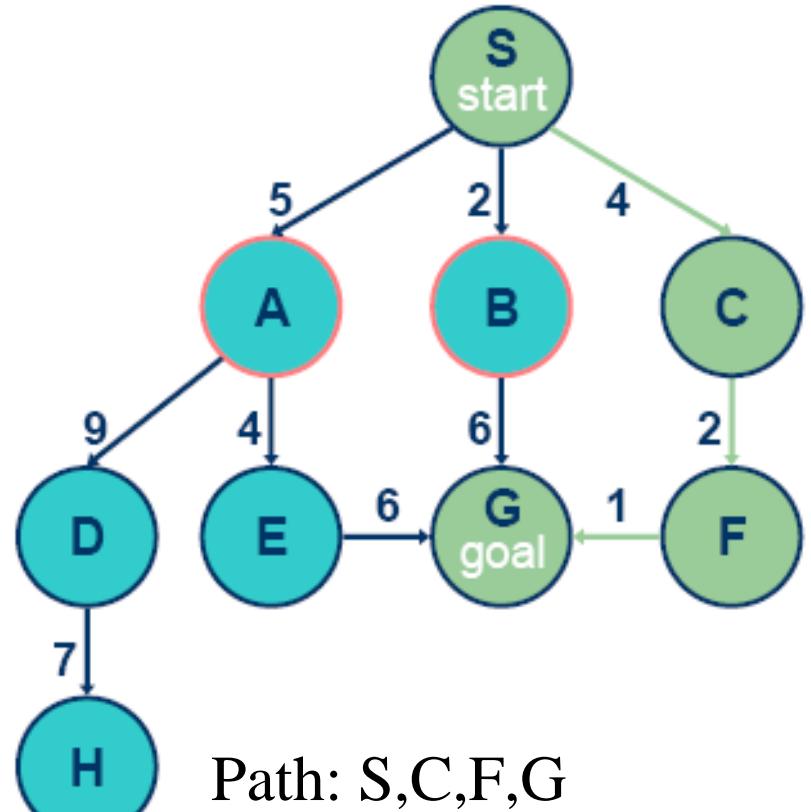
Exnd. node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9, D:14}
F	{G:7,G:8,E:9,D:14}
G goal	{G:8,E:9,D:14} no expand



## 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}





# 等代价搜索算法性能

- 最优性：最优
- 完备性：在每一步的代价都大于等于某个小的正值常数的前提下，完备
- 时间复杂度： $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ 
  - $C^*$ 表示最优解的代价， $\varepsilon$ 是每个行动的代价
- 空间复杂度： $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$

# 等代价搜索算法性能

- Completeness:
  - YES, if step-cost  $> \varepsilon$  (a small positive constant)
- Time complexity:
  - Assume  $C^*$  the cost of the optimal solution.
  - Assume that every action costs at least  $\varepsilon$
  - Worst-case:  $O(b^{C^*/\varepsilon})$
- Space complexity:
  - Idem to time complexity
- Optimality:
  - nodes expanded in order of increasing path cost
  - YES, if complete.



# Dijkstra算法原理

- Dijkstra(迪杰斯特拉)算法是典型的单源**最短路径**算法，用于计算一个节点到其他所有节点的最短路径。
- Dijkstra算法的主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法是很有代表性的最短路径算法，在很多专业课程中都作为基本内容有详细的介绍，如数据结构，图论，运筹学等等。
- Dijkstra一般的表述通常有两种方式，一种用永久和临时标号方式，一种是用OPEN, CLOSE表的方式。
- 注意该算法要求图中不存在负权回路。



# Dijkstra算法原理

- a) 初始时，集合S只包含源点，即 $S = \{v\}$ ,  $D(v)=0$ 。集合U包含除v外的其它顶点，即:  $U=\{\text{其余顶点}\}$ ，若v与U中顶点u有边，则 $\langle u,v \rangle$ 正常有权值 $D(u)=W(v,u)$ ，否则权值为 $\infty$ 。（ $W(v,u)$ 表示v到u的权值， $D(u)$ 表示最短路径值）
- b) 从U中选取一个距离v最小的顶点k，把k加入集合S中（该选定距离 $D(k)$ 就是v到k的最短路径长度）。

$$k = \operatorname{argmin}(D(i), i \in U)$$

- a) 以k为新考虑的中间点，修改U中各顶点的距离：若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值。

$$D(u)=\min\{D(u), D(k)+W(k,u)\}$$

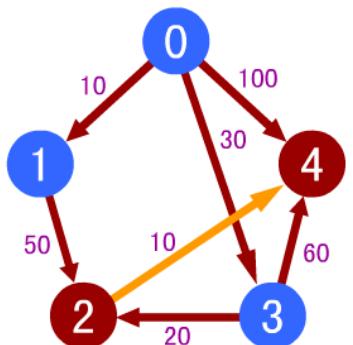
- a) 重复步骤b和c直到所有顶点都包含在S中。

# Dijkstra算法演示

bobo

## 最短路径

循环	红点集S	K	D{0}	D{1}	D{2}	D{3}	D{4}	P{0}	P{1}	P{2}	P{3}	P{4}
初始化	{2}	-	$\infty$	$\infty$	0	$\infty$	10	-1	-1	-1	-1	2



请输入源点:   动画演示中



# 盲目搜索方法比对

	宽度优先	深度优先	有界深度优先	等代价
完备性	Yes(条件)	No	No	Yes(条件)
最优化	Yes(条件)	No	No	Yes
时间复杂度	$O(b^d)$	$O(b^m)$	$O(b^l)$	$O(b^{1+ C^* /\varepsilon})$
空间复杂度	$O(b^d)$	$O(bm)$	$O(bl)$	$O(b^{1+ C^* /\varepsilon})$

# 思考题

- Genetics Professor
  - Wanting to name her new baby girl
  - Using only the letters D,N & A
- Suppose you have the letters ordered alphabetically (A, D, N) and you start writing down possibilities:  
**A, D, N, AA, AD, AN, DA, DD, DN, NA, ND, NN, ...**
  - How many strings of four or fewer letters are there where the letters are D, N or A?
  - In the above possibilities, are you searching in a depth first or breadth first way?
  - What are the next three possible names you would write down?
  - How many possibilities will you write down before getting to the name ANNA?



# 从盲目搜索到启发式搜索

- We have looked at different blind search strategies
  - Uninformed search methods lack problem-specific knowledge
  - inefficient in many cases
- But, if we know about the problem we are solving, we can be even cleverer...
  - Using problem-specific knowledge can dramatically improve the search speed
- informed search algorithms
  - use problem specific heuristics(启发式信息)

## Solving Problems by Searching 2:

# 启发式搜索

(Heuristic Search Strategies)

1. Best-first search
2. Greedy Search
3. Algorithm A
4. Algorithm A\*
5. Hill climbing



# 启发式搜索(Heuristic search/ Informed search)

- 启发式搜索(Heuristically Search)又称为有信息搜索(Informed Search)，它是利用问题拥有的启发信息来引导搜索，达到减少搜索范围、降低问题复杂度的目的，这种利用启发信息的搜索过程称为启发式搜索。
- 启发式搜索运用启发信息，引用某些准则或经验来重新排列 $\textcolor{blue}{OPEN}$ 表中节点的顺序，使搜索沿着某个被认为最有希望的前沿区段扩展。
- 用符号 $f$ 来标记估价函数/评价函数(Evaluation Function)，用 $f(n)$ 表示节点 $n$ 的估价函数值， $f$ 是从起始节点约束地通过节点 $n$ 而到达目标节点的最小代价路径上的一个估算代价。
- 估价函数 $f$ 值越小，意味着改节点位于最优解路径上的“希望”越大，最后找到的最优路径即平均综合指标为最小的路径。正确选择估价函数/评价函数 $f(n)$ ，对于寻求最小代价路径至关重要。

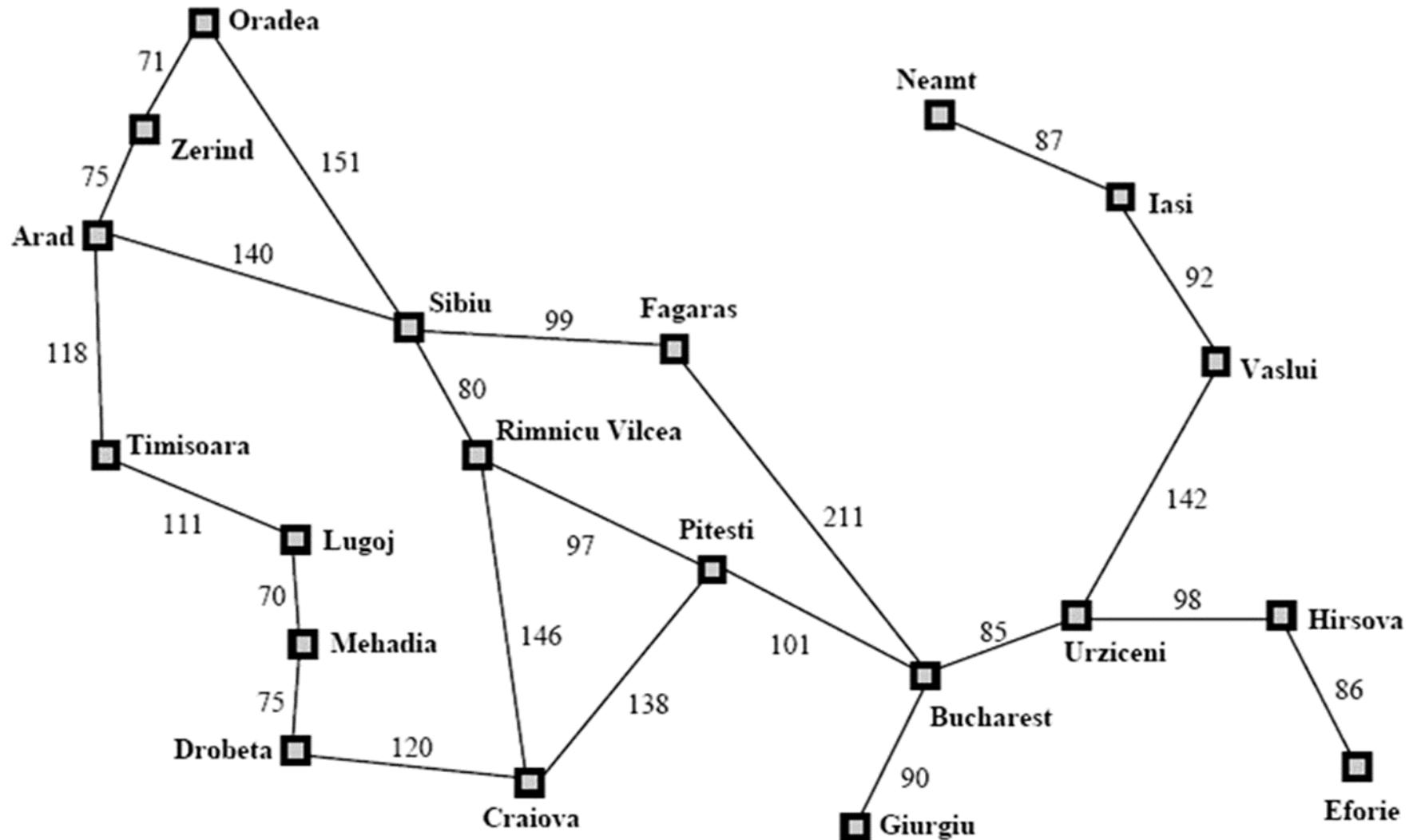
# 启发函数(Heuristic function)

## ➤ 启发函数

$h(n)$  = 节点  $n$  到目标节点的最小代价路径的代价估计值

- 利用 $h(n)$ 来决定节点的扩展顺序
- $h(n)$ 的值是对当前状态 $n$ 的一个估计，表示：
  - the "goodness" of node n
  - how close node n is to a goal
  - the cost of minimal cost path from node n to a goal
- $h(n) \geq 0$
- $h(n)$ 越小表示 $n$ 越接近目标，若 $n$ 为目标则 $h(n) = 0$
- 与问题相关的启发式信息都被计算为一定的 $h(n)$ 的值引入到搜索过程中。

# 例1：Romania



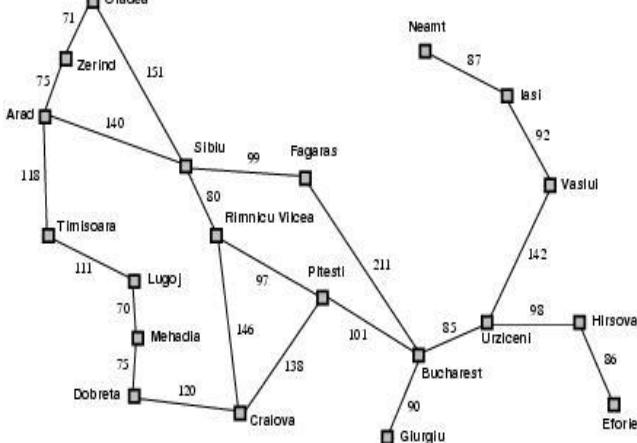
# 例1：Romania

- ❖ On holiday in Romania; currently in Arad
  - ❖ Flight leaves tomorrow from Bucharest
- ❖ Formulate goal
  - ❖ Be in Bucharest
- ❖ Formulate problem
  - ❖ States: various cities
  - ❖ Actions: drive between cities
- ❖ Find solution
  - ❖ Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, ...

# 例1：Romania

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

- $h_{SLD}$ =straight-line distance heuristic.
- $h_{SLD}$  can **NOT** be computed from the problem description itself





# 启发式搜索的一般策略：最佳优先搜索/有序搜索

最佳优先搜索(Best-first Search)  $\leftrightarrow$  有序搜索(Ordered Search)

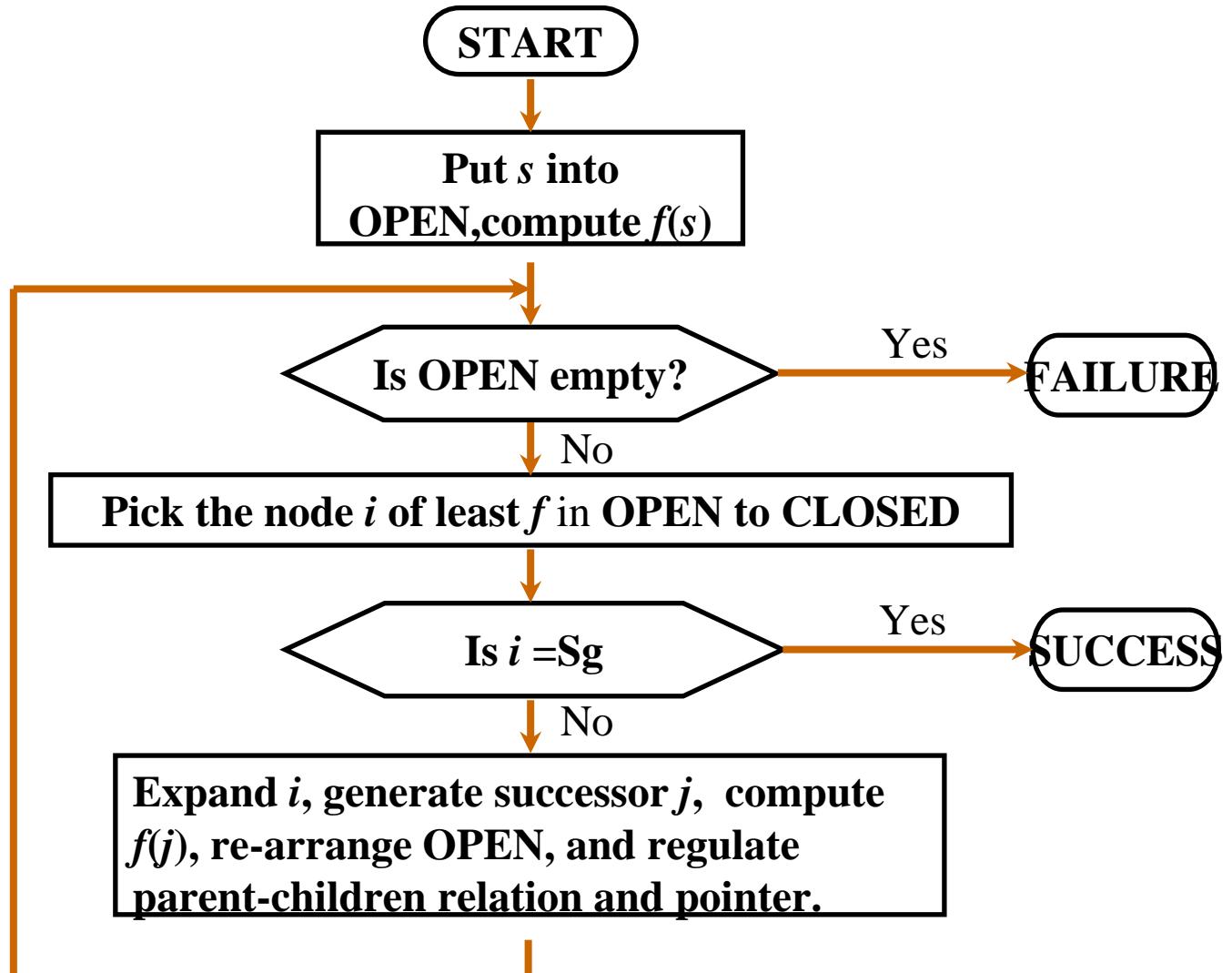
- 最佳优先搜索是一般Tree-Search和Graph-Search算法的一个实例，节点是基于评价函数 $f(n)$ 值被选择扩展的。评价函数被看作是代价估计，因此评价值最低的节点被选择首先进行扩展。
- 最佳优先搜索的实现与一致代价搜索(Uniform Cost Search)类似，不过最佳优先是根据评价函数 $f$ 值而不是路径消耗 $g$ 值对优先级队列排队。一致代价搜索可以看作是最佳优先图搜索算法的一个特例。



# 最佳优先搜索算法流程

- ① 把起始节点S放到OPEN表中，计算 $f(s)$ 并把其值与节点S联系起来。
- ② 如果OPEN是空表，则失败退出，无解。
- ③ 从OPEN表中选择一个 $f$ 值最小的节点i。如果有几个节点合格，当其中有一个为目标节点时，则选择此目标节点，否则就选择其中任一节点作为节点i。
- ④ 把节点i从OPEN表中移出，并把它放入CLOSED的已扩展节点表中。
- ⑤ 如果i是一个目标节点，则成功退出，求得一个解。
- ⑥ 扩展节点i，生成其全部后继节点。对于i的每一个后继节点j：
  - a) 计算 $f(j)$ 。
  - b) 如果j即不在OPEN表中，又不在CLOSED表中，则用估价函数 $f$ 把它填入OPEN表。从j加一指向其父节点i的指针，以便一旦找到目标节点时记住一个解答路径。
  - c) 如果j已在OPEN表或CLOSED表中，则比较刚刚对j计算过的 $f$ 值和前面计算过的该节点在表中的 $f$ 值。如果新的 $f$ 值较小，则
    - I. 以此新值取代旧值。
    - II. 从j指向i，而不是指向它的父节点。
    - III. 如果节点j在CLOSED表中，则把它移回OPEN表。
- ⑦ 转向②，即 GO TO (2)。

# 最佳优先搜索算法流程





# 最佳优先搜索的有效性辨别

- 与盲目搜索方法相比，有序搜索的目的在于减少被扩展的节点数。
- 有序搜索的有效性直接取决于估价函数  $f$  的选择。这将敏锐的辨别出有希望的节点和没有希望的节点。如果辨别不准确，就可能失去一个最好的解甚至全部的解。
- 如果没有适用的准确的希望量度，那么  $f$  的选择将涉及两个方面的内容：一方面是一个时间和空间之间的折衷方案；另一方面是保证有一个最优的解或任意解。

## 例2：八数码难题

2	8	3
1		4
7	6	5

$$f(n) = d(n) + W(n)$$

- $d(n)$  是搜索树中节点  $n$  的深度，即从初始节点到节点  $n$  所需要进行的操作次数
- $W(n)$  用来计算节点  $n$  相对于目标棋局错放的旗子个数



# 搜索问题分类

- 节点希望度量以及某个估价函数的合适程度取决于手头的问题情况。根据所要求的解答类型，可以把问题分为下列三种情况：
  1. 假设该状态空间含有几条不同代价的解答路径，其问题是要求得到最优（即最小代价）解答。这种情况的代表性例子为A\*算法。
  2. 与第一种情况相似，但有一个附加条件：此类问题是比较难的，如果按第一种情况加以处理，则搜索过程很可能在找到解答之前就超过了时间和空间界限。在这种情况下，关键问题是：1) 如何通过适当的搜索试验找到好的（但不是最优的）解答；2) 如何限制搜索试验的范围和所产生的解答与最优解答的差异程度。推销员旅行问题
  3. 不考虑解答的最优化；或者只存在一个解，或者任意一个解与其他解一样好。这样，问题是如何使搜索试验的次数最少，而不像第二种情况那样试图使某些搜索试验和解答代价的综合指标最小。定理证明问题

# 估价函数选择

- ◆ Different ways of defining the function  $f$ . This leads to different search algorithms

- $f(n) = g(n)$  uniform cost algorithm
- $f(n) = h(n)$  greedy algorithm
- $f(n) = g(n) + h(n)$  A algorithm
- $f^*(n) = g^*(n) + h^*(n)$  A\* algorithm

$g(n)$  = cost of the path from the start node to an open node  $n$   
 $h(n)$  = estimates the distance remaining to a goal



# Methodologies of Heuristic Search 1:

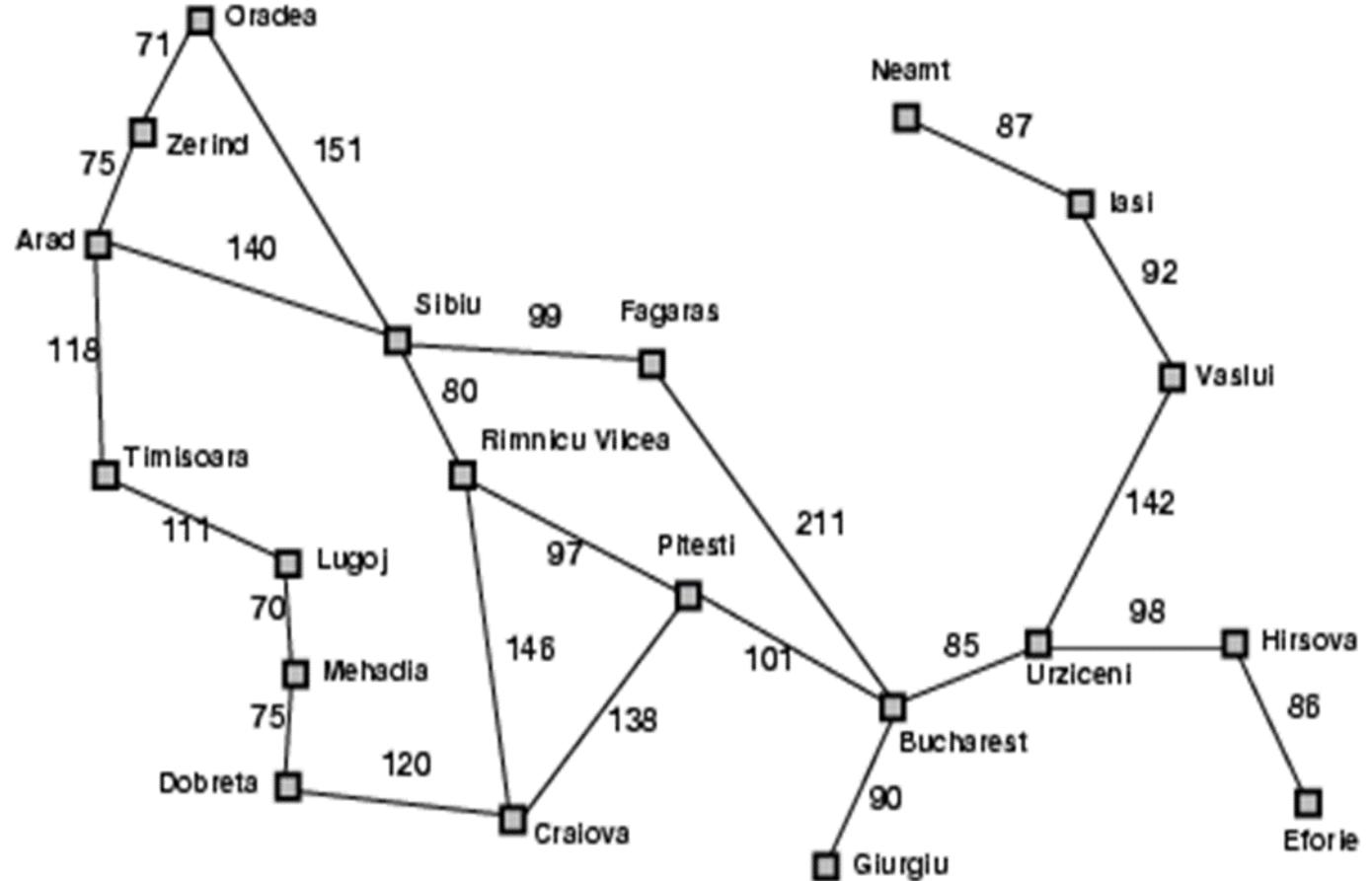
贪婪最佳优先搜索  
(Greedy Best-first Search)

# Greedy Search

---

- ➊ In greedy search, the idea is to expand the node with the smallest estimated cost to reach the goal.
- ➋ We use a heuristic function
  - ➌  $f(n) = h(n)$

# 例3：Greedy Search



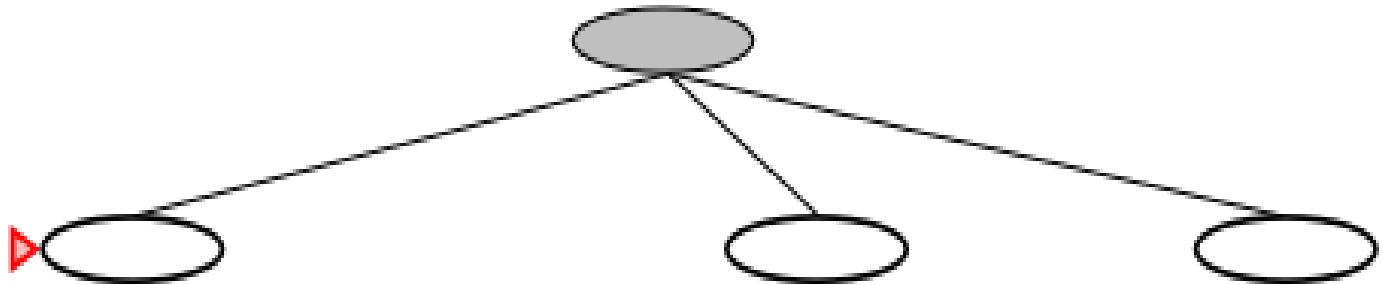
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu vilcea	193
Sibiu	253
Timisoara	329
Urzicemini	80
Vaslui	199
zerind	374

## 例3：Greedy Search

- ➊ Assume that we want to use greedy search to solve the problem of travelling from Arad to Bucharest.
  - ➊  $f(n)=h(n)$
- ➋ The initial state=Arad

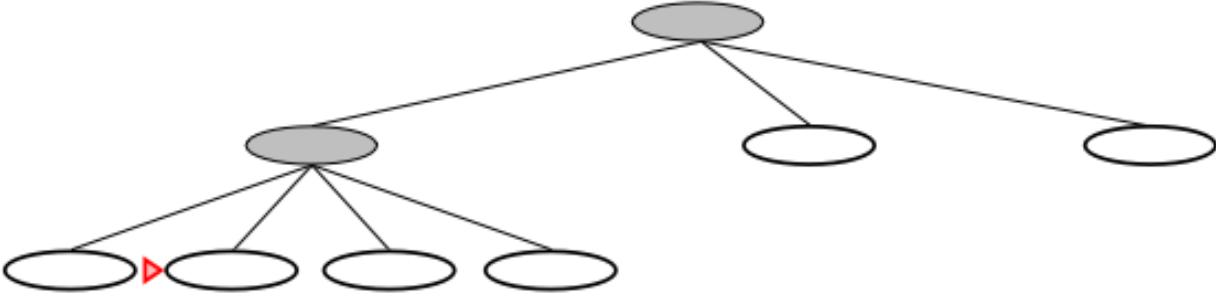
Arad (366)  


## 例3：Greedy Search



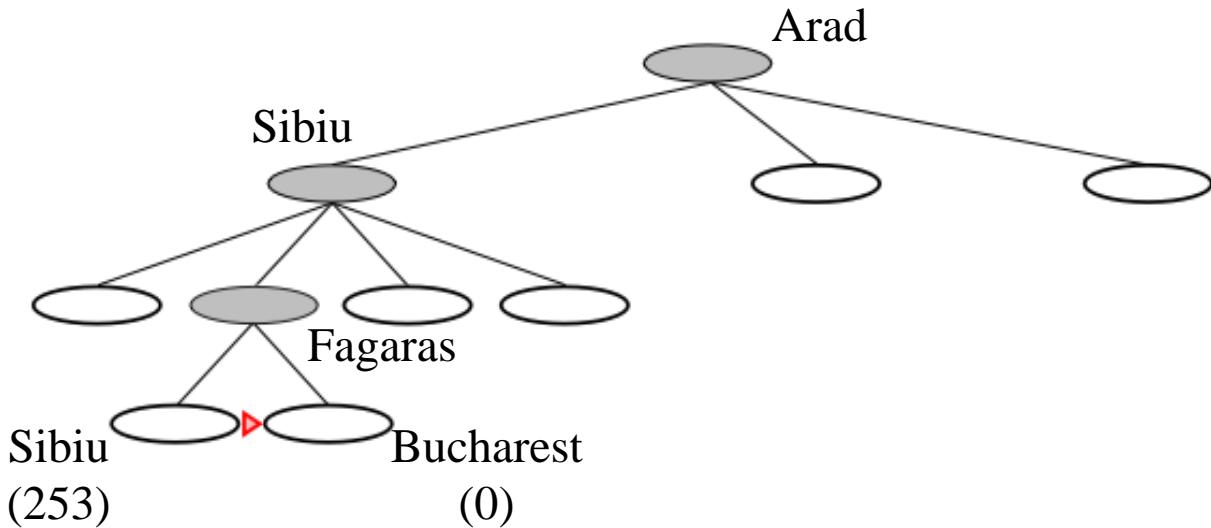
- ◆ The first expansion step produces:
  - Sibiu, Timisoara and Zerind
- ◆ Greedy best-first will select Sibiu.

## 例3：Greedy Search



- ◆ If Sibiu is expanded we get:
  - Arad, Fagaras, Oradea and Rimnicu Vilcea
- ◆ Greedy best-first search will select: Fagaras

## 例3：Greedy Search



- ➊ If Fagaras is expanded we get:
  - Sibiu and Bucharest
- ➋ Goal reached !!
  - Yet not optimal (see Arad, Sibiu, Rimnicu Vilcea, Pitesti)



# Greedy Search, evaluation

---

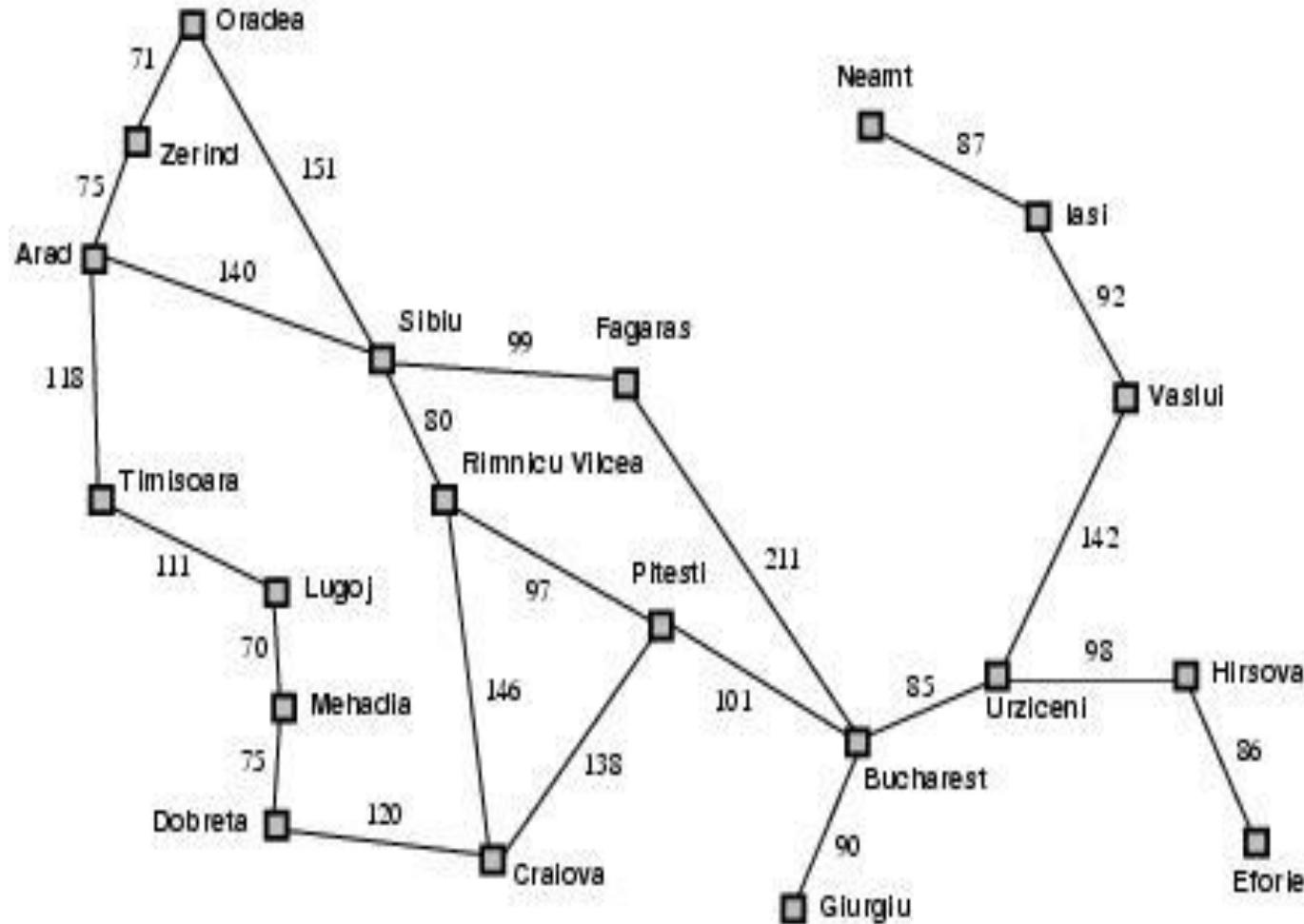
- ◆ Often perform very well
- ◆ They tend to find good solutions quickly
- ◆ not always optimal ones.

# Greedy Search, evaluation

- ❖ Completeness:

NO

- Check on repeated states
- Minimizing  $h(n)$  can result in false starts,  
e.g. *Iasi* to *Fagaras*.





# Greedy Search, evaluation

- ➊ Completeness: NO ( DF-search)
- ➋ Time complexity?
  - Worst-case DF-search  
(with  $m$  is maximum depth of search space)
  - Good heuristic can give dramatic improvement.

# Greedy Search, evaluation

---

- ❖ Completeness: NO (cfr. DF-search)
- ❖ Time complexity:
- ❖ Space complexity:
  - Keeps all nodes in memory

# Greedy Search

---

- ❖ Completeness: NO (cfr. DF-search)
- ❖ Time complexity:
- ❖ Space complexity:
- ❖ Optimality? NO
  - ▣ Same as DF-search

# 贪婪有序搜索的算法性能 (未完成)

## ● 完备性 (Completeness)

- Does it always find a solution if one exists?
- NO

## ● 时间复杂度 (Time complexity)

$$O(b^m)$$

## ● 空间复杂度

$$O(b^m)$$

m是搜索空间的最大深度



## Methodologies of Heuristic Search 2:

# A<sup>\*</sup>搜索 (A<sup>\*</sup> Search)

# A算法



尼尔逊



拉斐尔

- 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，拉斐尔改进了A1算法，称为A2算法

## ◆ 特征

- 估价函数

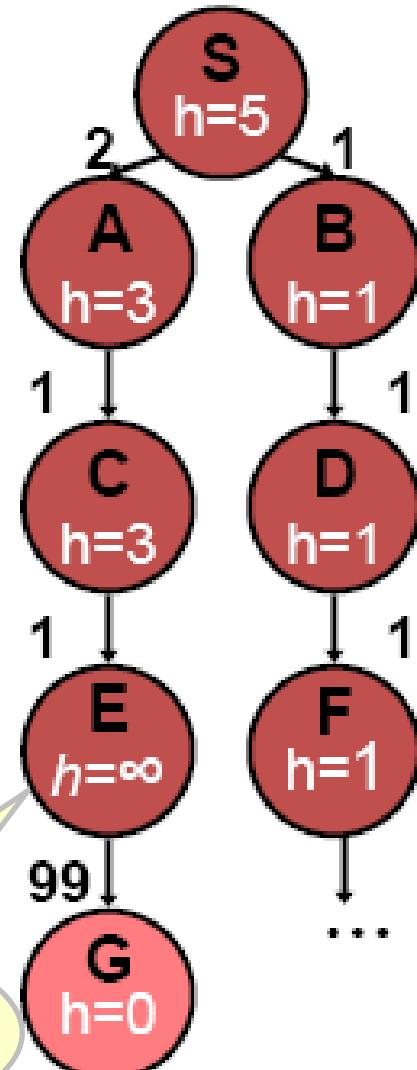
$$f(x) = g(x) + h(x)$$

- 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

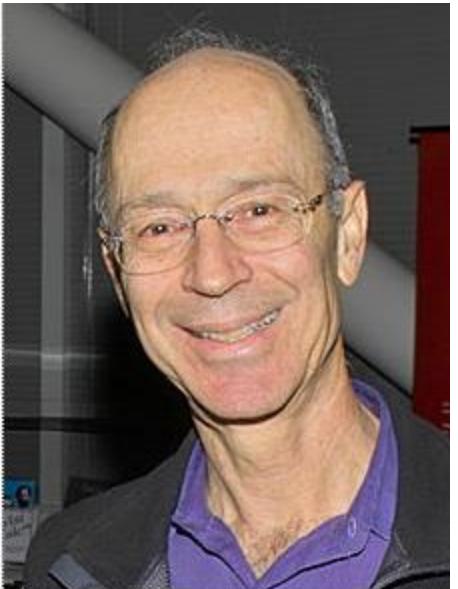
## ◆ 性能

- 不完备，不最优

永远不会被扩展



# A\*算法



彼得·哈特

- 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

## A\*算法的限制条件

$$f(x) = \underline{g(x)} + \underline{h(x)}$$

大于0 不大于 $x$ 到目标的实际代价 $h^*(x)$



# A\*算法

A\* (A-Star)算法是一种典型的启发式搜索算法, 是一种静态路网中求解最短路径最有效的方法。

在启发式搜索中, 对位置的估价是十分重要的, 启发中的估价是用估价函数表示的:

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

A\*算法是一个可采纳的最好优先算法。A\*算法的估价函数可表示为:

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

## ◆ A\*算法要求其启发函数的定义是可纳的

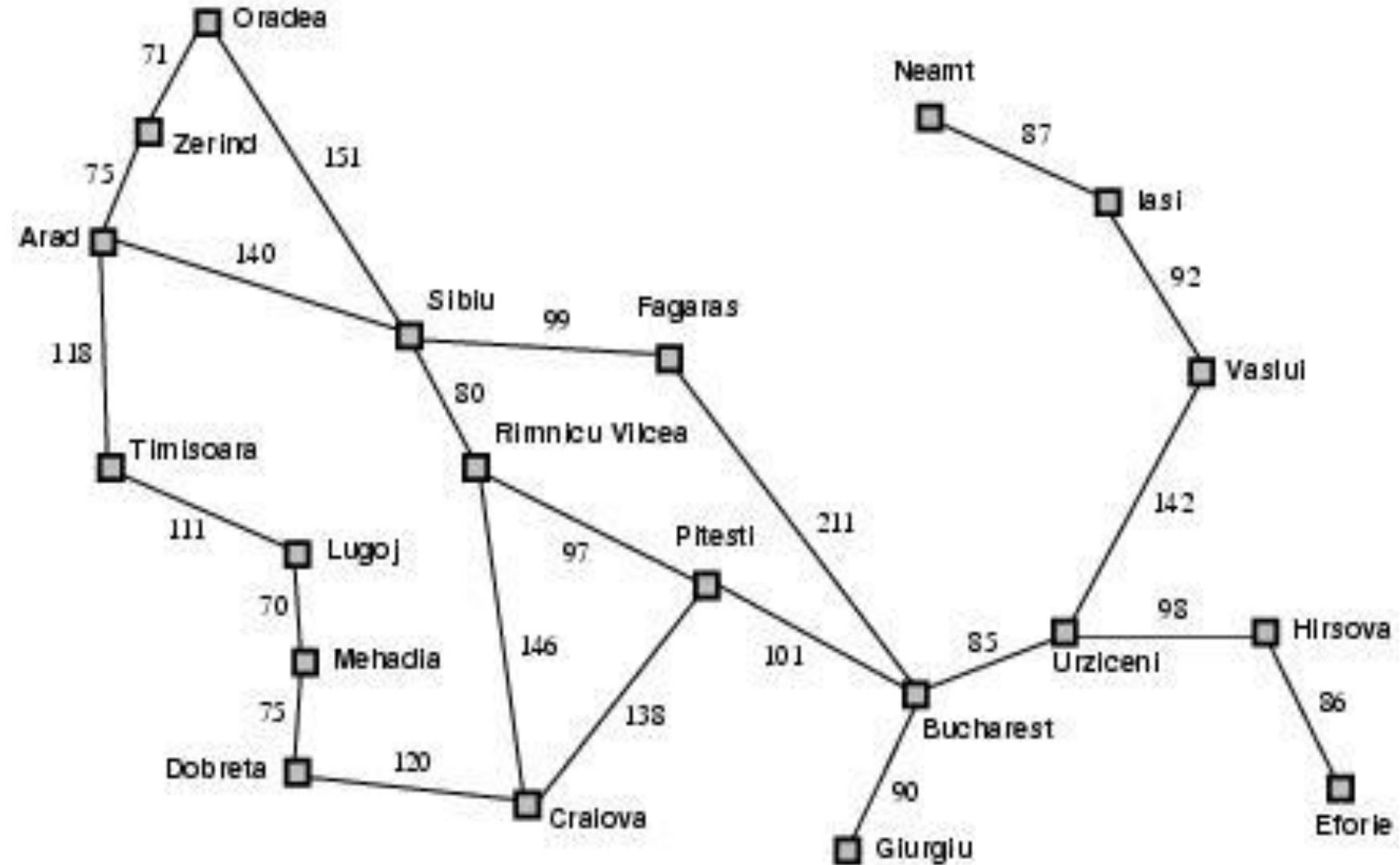
### ■ 可纳性

- A heuristic is admissible if it never overestimates the cost to reach the goal
- Are optimistic

### ■ Formally:

- $h(n) \leq h^*(n)$  where  $h^*(n)$  is the true cost from  $n$
- $h(n) \geq 0$  so  $h(G)=0$  for any goal  $G$ .
  - e.g.  $h_{SLD}(n)$  never overestimates the actual road distance

## 例4：A\*算法 Romania



## 例4：A\*算法 Romania

(1) The initial state

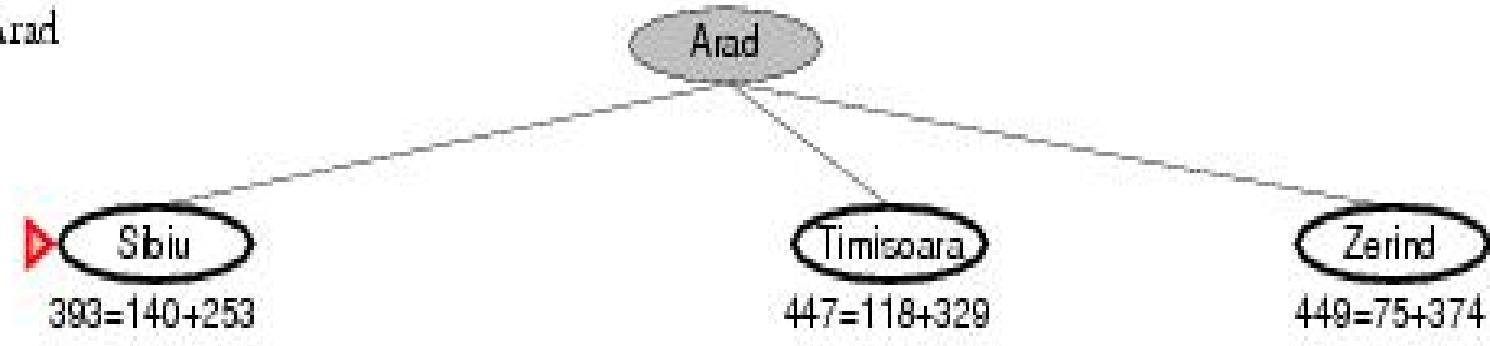


❖ Find Bucharest starting at Arad

$$\blacksquare f(\text{Arad}) = g(\text{??}, \text{Arad}) + h(\text{Arad}) = 0 + 366 = 366$$

## 例4：A\*算法 Romania

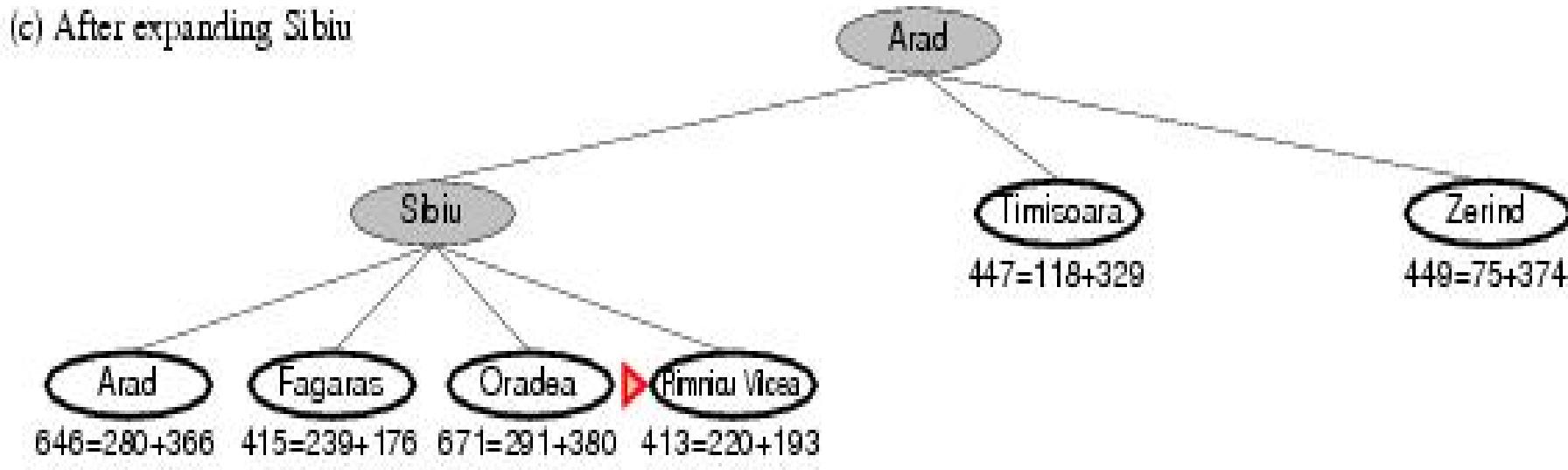
After expanding Arad



- ❖ Expand Arad and determine  $f(n)$  for each node
  - $f(\text{Sibiu}) = g(\text{Arad}, \text{Sibiu}) + h(\text{Sibiu}) = 140 + 253 = 393$
  - $f(\text{Timisoara}) = g(\text{Arad}, \text{Timisoara}) + h(\text{Timisoara}) = 118 + 329 = 447$
  - $f(\text{Zerind}) = g(\text{Arad}, \text{Zerind}) + h(\text{Zerind}) = 75 + 374 = 449$
- ❖ Best choice is Sibiu

## 例4：A\*算法 Romania

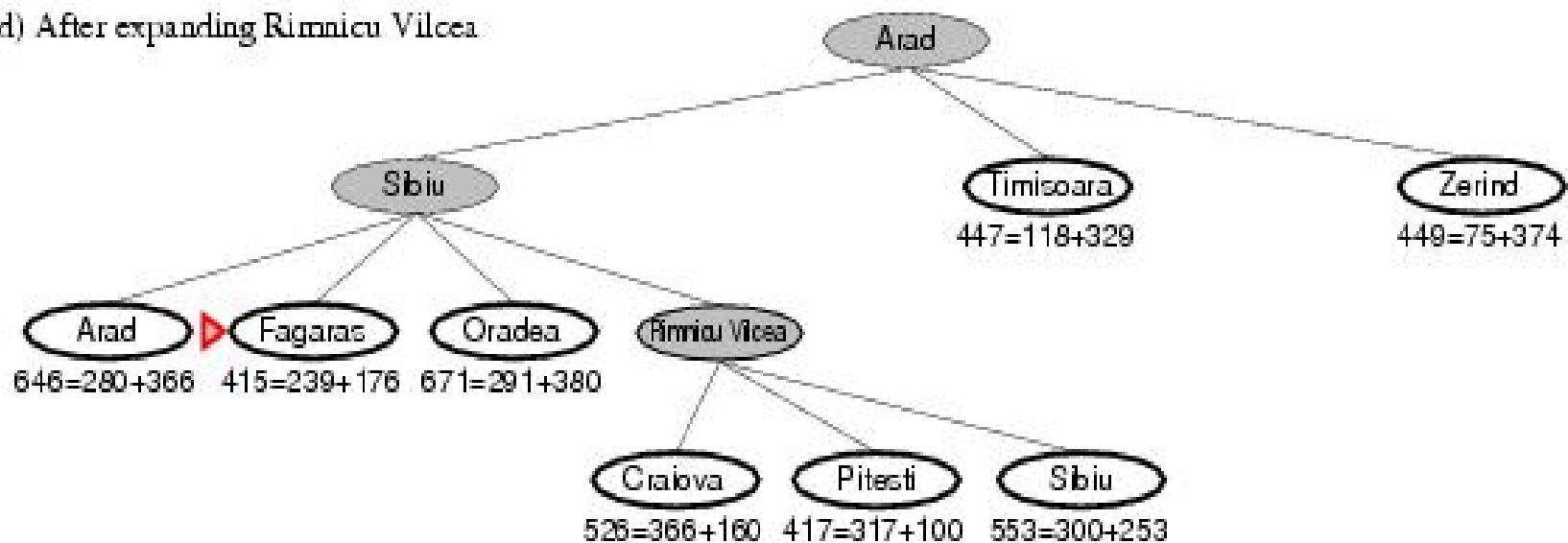
(c) After expanding Sibiu



- ❖ Expand Sibiu and determine  $f(n)$  for each node
  - ❖  $f(\text{Arad}) = g(\text{Sibiu}, \text{Arad}) + h(\text{Arad}) = 280 + 366 = 646$
  - ❖  $f(\text{Fagaras}) = g(\text{Sibiu}, \text{Fagaras}) + h(\text{Fagaras}) = 239 + 179 = 415$
  - ❖  $f(\text{Oradea}) = g(\text{Sibiu}, \text{Oradea}) + h(\text{Oradea}) = 291 + 380 = 671$
  - ❖  $f(\text{Rimnicu Vilcea}) = g(\text{Sibiu}, \text{Rimnicu Vilcea}) + h(\text{Rimnicu Vilcea}) = 220 + 192 = 413$
- ❖ Best choice is Rimnicu Vilcea

# 例4：A\*算法 Romania

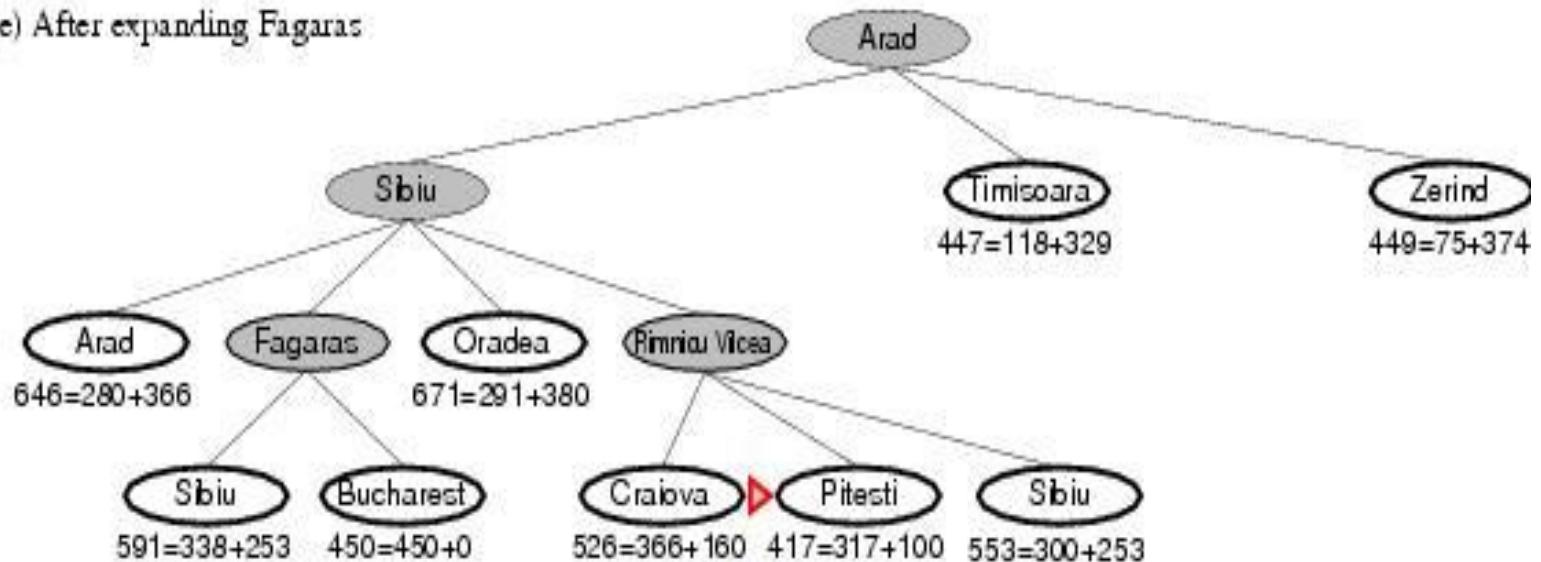
(d) After expanding Rimnicu Vilcea



- Expand Rimnicu Vilcea and determine  $f(n)$  for each node
  - $f(\text{Craiova}) = g(\text{Rimnicu Vilcea}, \text{Craiova}) + h(\text{Craiova}) = 360 + 160 = 526$
  - $f(\text{Pitesti}) = g(\text{Rimnicu Vilcea}, \text{Pitesti}) + h(\text{Pitesti}) = 317 + 100 = 417$
  - $f(\text{Sibiu}) = g(\text{Rimnicu Vilcea}, \text{Sibiu}) + h(\text{Sibiu}) = 300 + 253 = 553$
- Best choice is Fagaras

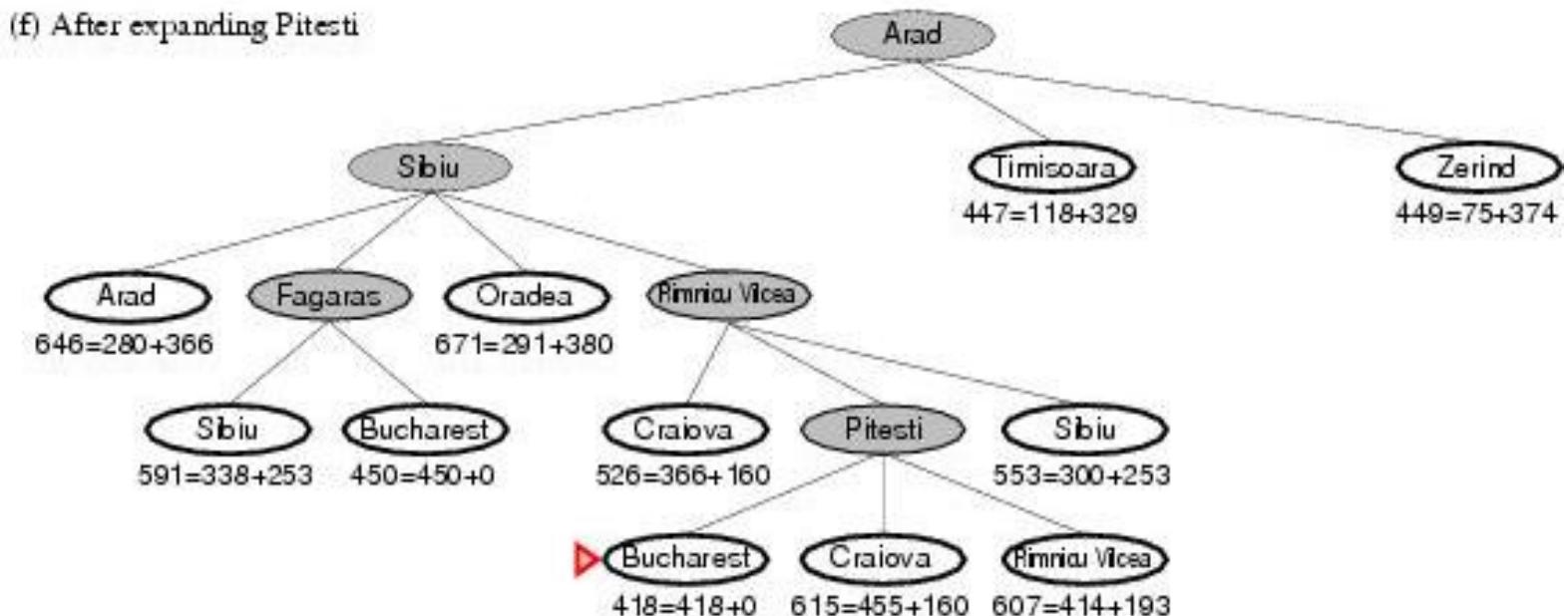
# 例4：A\*算法 Romania

(e) After expanding Fagaras



- ➊ Expand Fagaras and determine  $f(n)$  for each node
  - $f(\text{Sibiu}) = g(\text{Fagaras}, \text{Sibiu}) + h(\text{Sibiu}) = 338 + 253 = 591$
  - $f(\text{Bucharest}) = g(\text{Fagaras}, \text{Bucharest}) + h(\text{Bucharest}) = 450 + 0 = 450$
- ➋ Best choice is Pitesti !!!

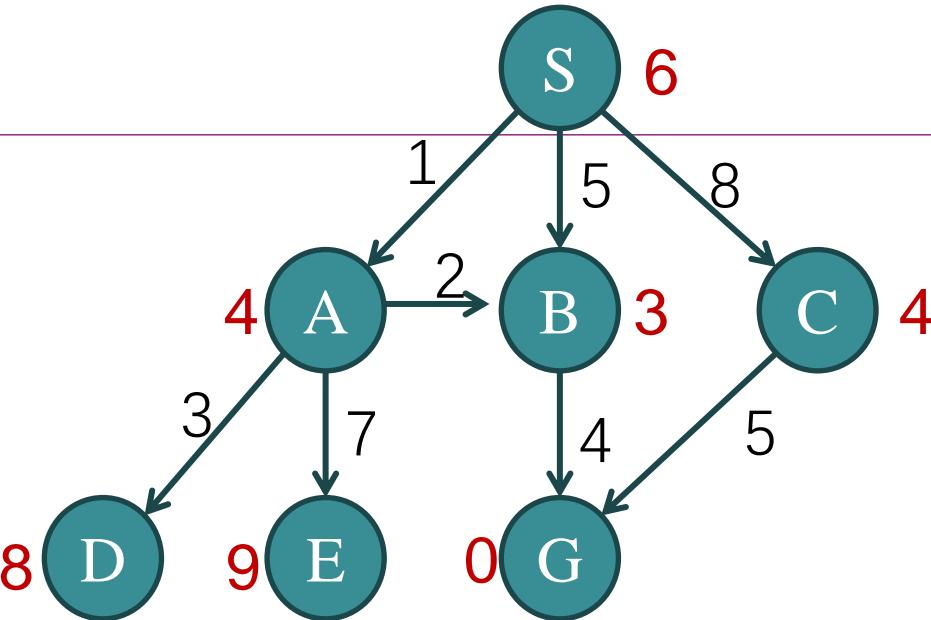
# 例4：A\*算法 Romania



- ◆ Expand Pitesti and determine  $f(n)$  for each node
  - $f(\text{Bucharest}) = g(\text{Pitesti}, \text{Bucharest}) + h(\text{Bucharest}) = 418 + 0 = 418$
- ◆ Best choice is Bucharest !!!
  - Optimal solution (only if  $h(n)$  is admissible)

# 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
- 若忽略重复节点，则有可能无法找到最优路径



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

## Algorithm A\*

OPEN={ s, nil }

while OPEN is not empty

    remove from OPEN the node  $\langle n, p \rangle$  with minimum  $f(n)$

    place  $\langle n, p \rangle$  on CLOSED

    if  $n$  is a goal node,

        return success (path  $p$ )

    for each edge  $e$  connecting  $n$  &  $m$  with cost  $c$

        if  $\langle m, q \rangle$  is on CLOSED and  $\{p|e\}$  is cheaper than  $q$ ,  
        then remove  $m$  from CLOSED,

            put  $\langle m, \{p|e\} \rangle$  on OPEN

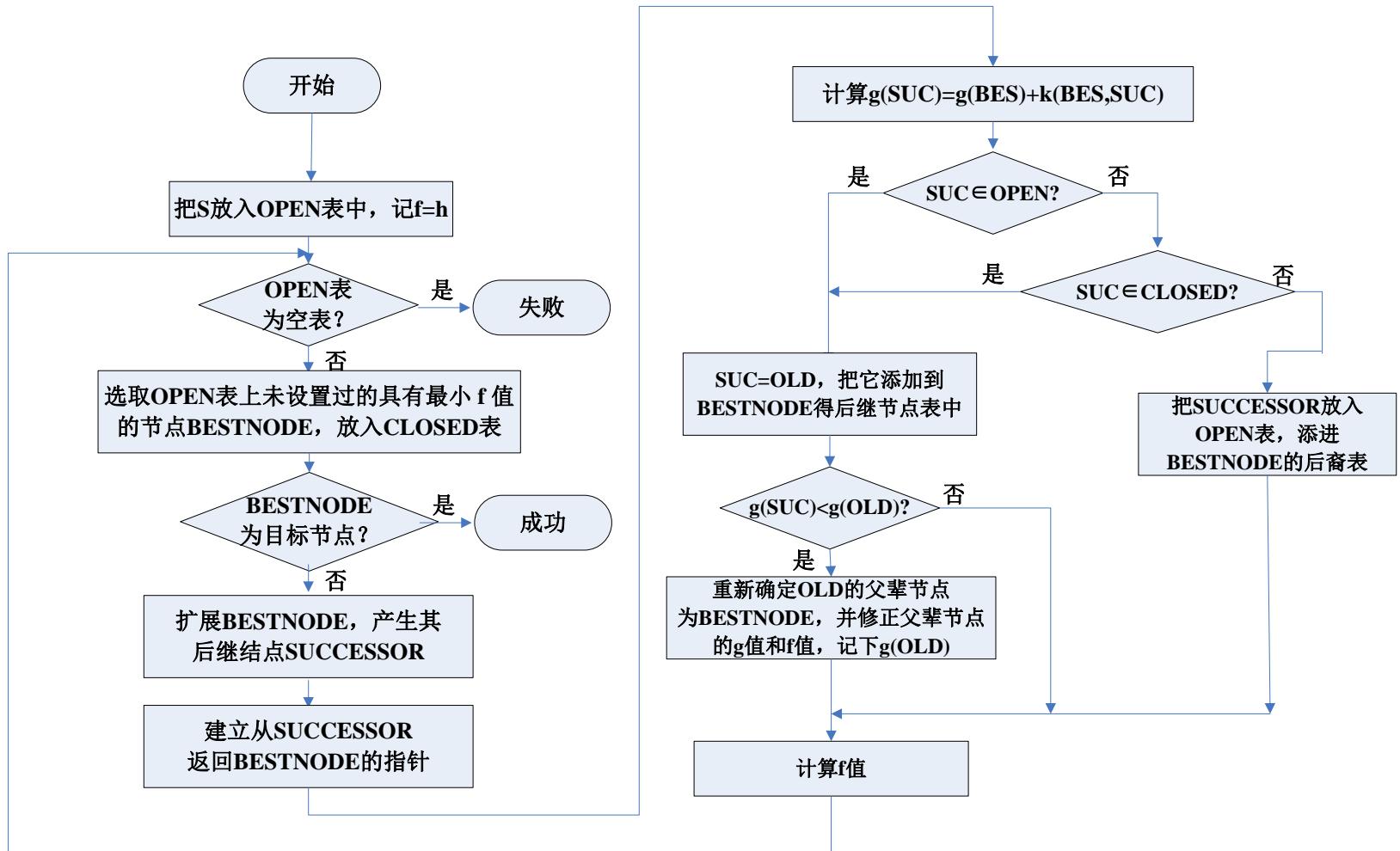
        else if  $\langle m, q \rangle$  is on OPEN and  $\{p|e\}$  is cheaper than  $q$ ,  
            then replace  $q$  with  $\{p|e\}$

        else if  $m$  is not on OPEN

            Then put  $\langle m, \{p|e\} \rangle$  on OPEN

Return failure

# A\*算法流程



# 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

■  $h(x)$ : 在 $x$ 状态下错放的棋子数  
满足限制条件

错放的棋子越  
少越好!



2	8	3
7	1	4
	6	5

$$h(x)=4$$

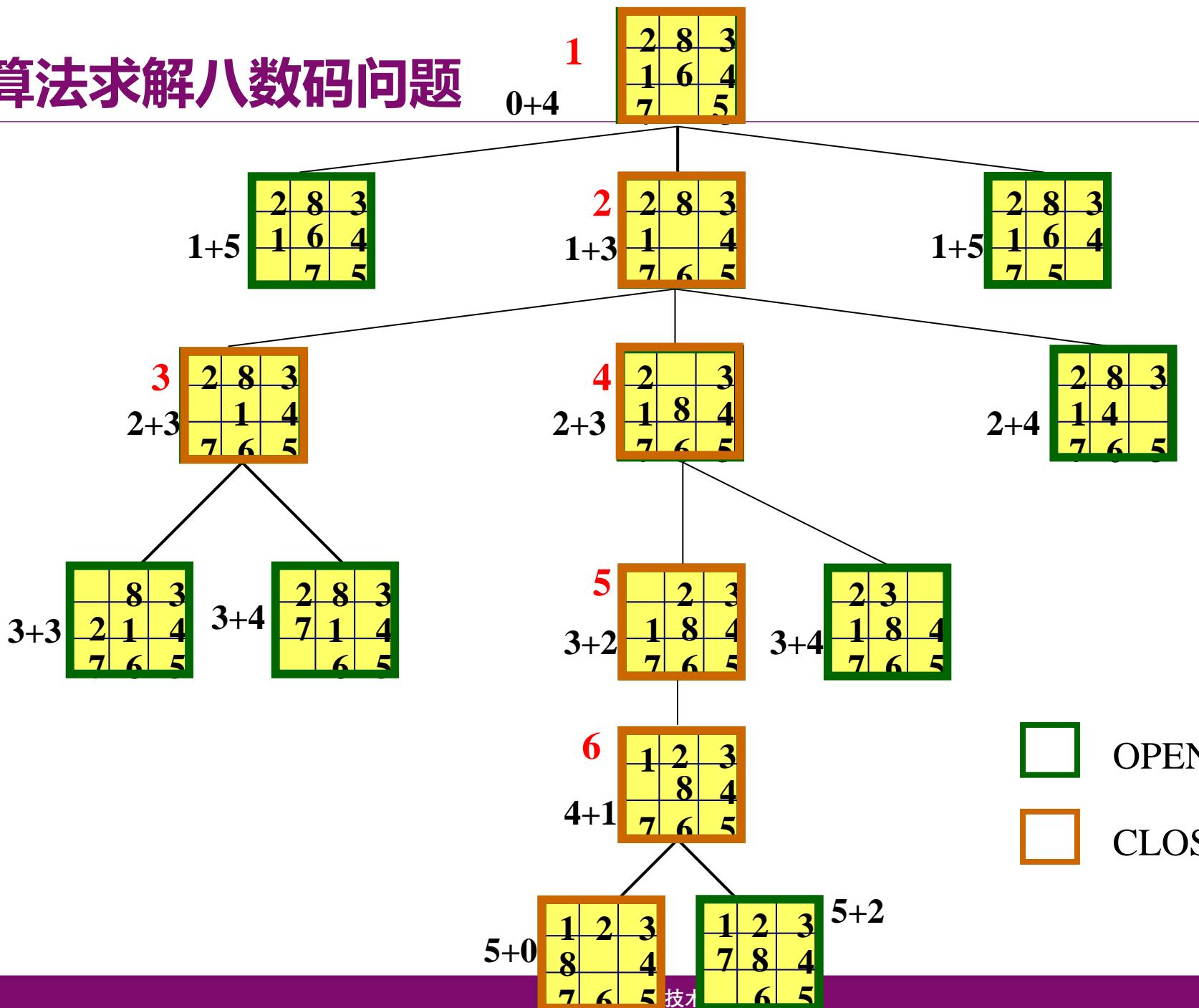
1	2	3
7	8	4
	6	5

$$h(x)=2$$

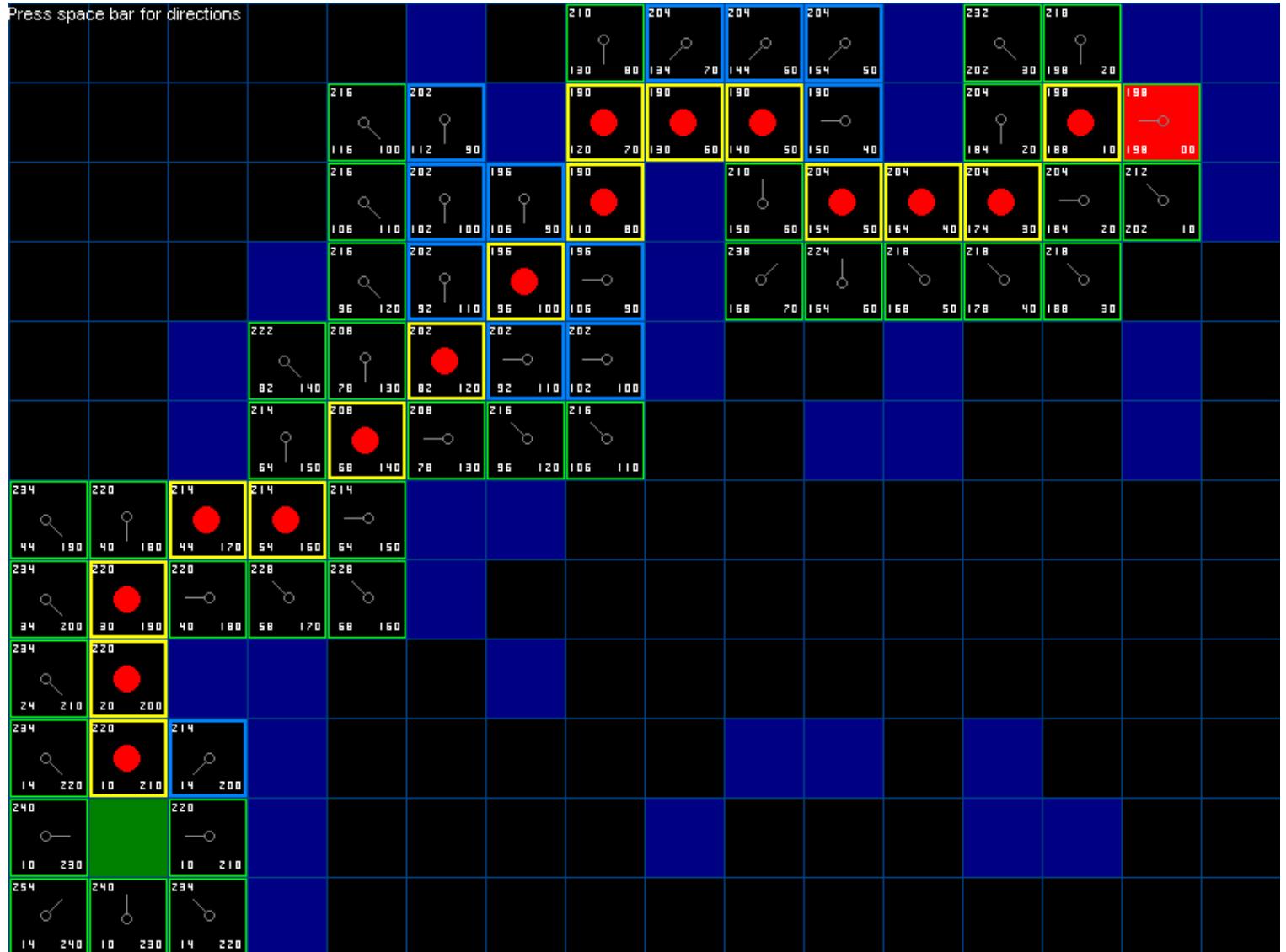
1	2	3
	8	4
7	6	5

$$h(x)=1$$

# 利用A\*算法求解八数码问题



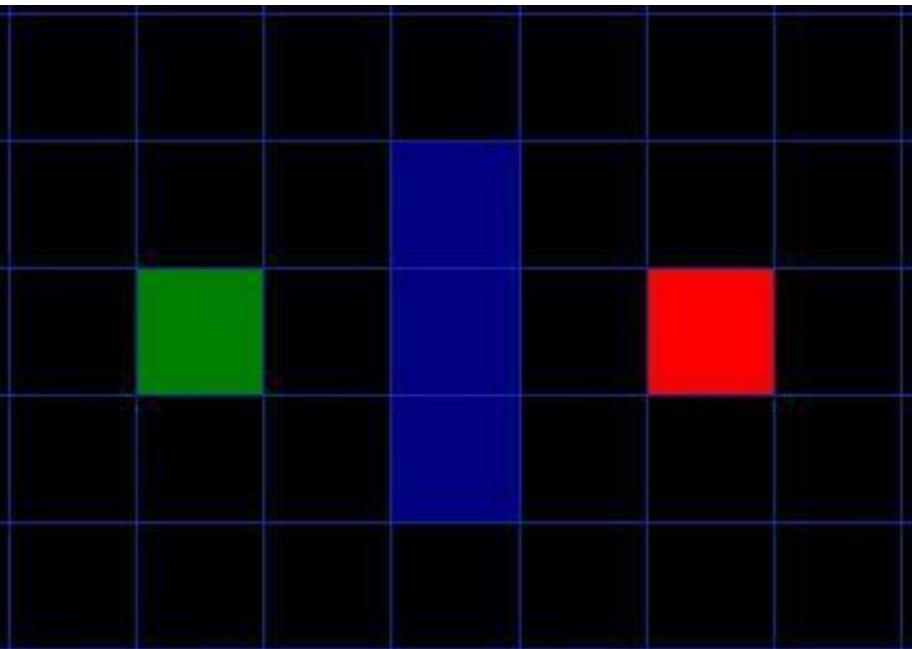
# 例5：A\*算法 Path Planning in a grid map



# 例5：A\*算法 Path Planning in a grid map 详解

## 1. 简化搜索区域

- 把搜索区域划分成了方形网格。
- 用二维数组保存方形网格。
  - 数组的每一个元素是网格的一个方块
  - 方块被标记为可通过的和不可通过的
  - 路径被描述为从A到B经过的方块的集合
  - 一旦路径被找到，就从一个方格的中心走向另一个，直到到达目的地



绿色的是起点A  
红色是终点B  
蓝色是中间的墙。



# 例5：A\*算法 Path Planning in a grid map 详解

## 2. 开始搜索

- 从点A开始，并且把它作为待处理点存入一个“开启列表”。
- 寻找起点周围所有可到达或者可通过的方格，跳过有墙或其他无法通过地形的方格。也把他们加入开启列表。为所有这些方格保存点A作为“父方格”。
- 从开启列表中删除点A，把它加入到一个“关闭列表”，列表中保存所有不需要再次检查的方格。
- 选择开启列表中的临近方格，即F值最低的。



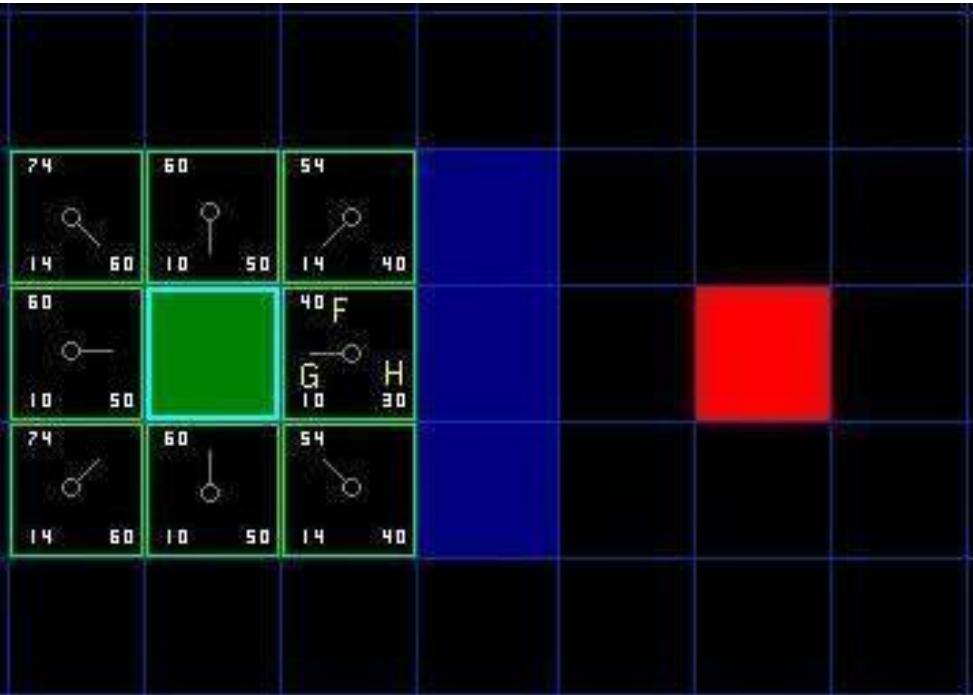
# 例5：A\*算法 Path Planning in a grid map 详解

## 3. 路径评分

- 路径评分公式
  - $F = G + H$
  - $G$  = 从起点A，沿着产生的路径，移动到网格上指定方格的移动耗费
  - $H$  = 从网格上那个方格移动到终点B的预估移动耗费
- 路径是通过反复遍历开启列表并且选择具有最低F值的方格来生成的

## 例5：A\*算法 Path Planning in a grid map 详解

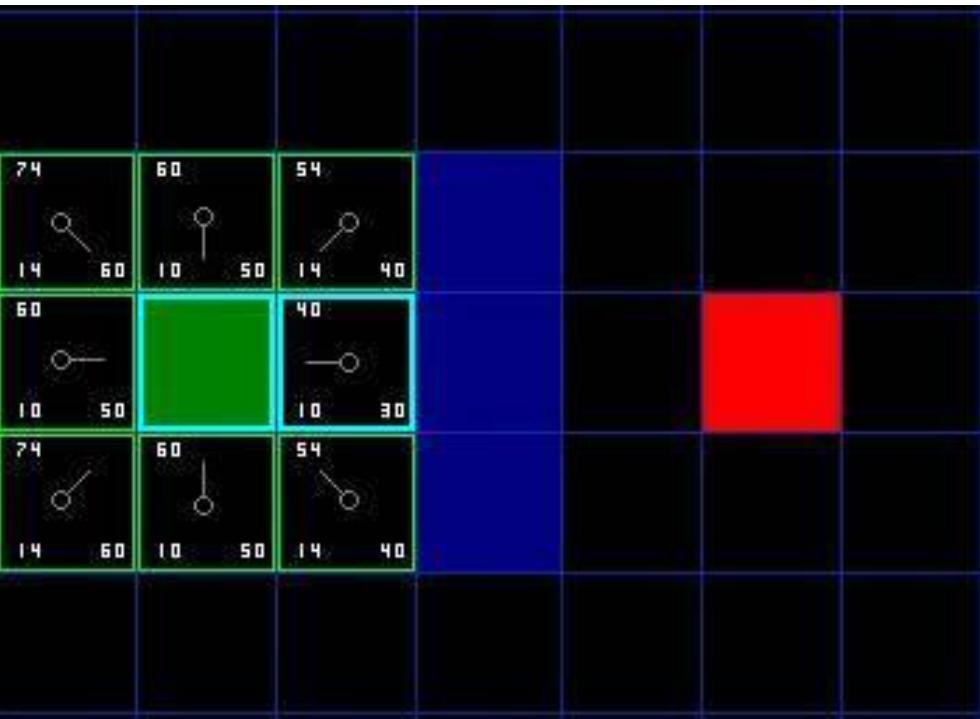
- 水平或者垂直移动的耗费为10，对角线方向耗费为14
- H值可以用不同的方法估算。使用的方法被称为曼哈顿方法，它计算从当前格到目的格之间水平和垂直的方格的数量总和
- 左上角为F，左下角为G，右下角为H



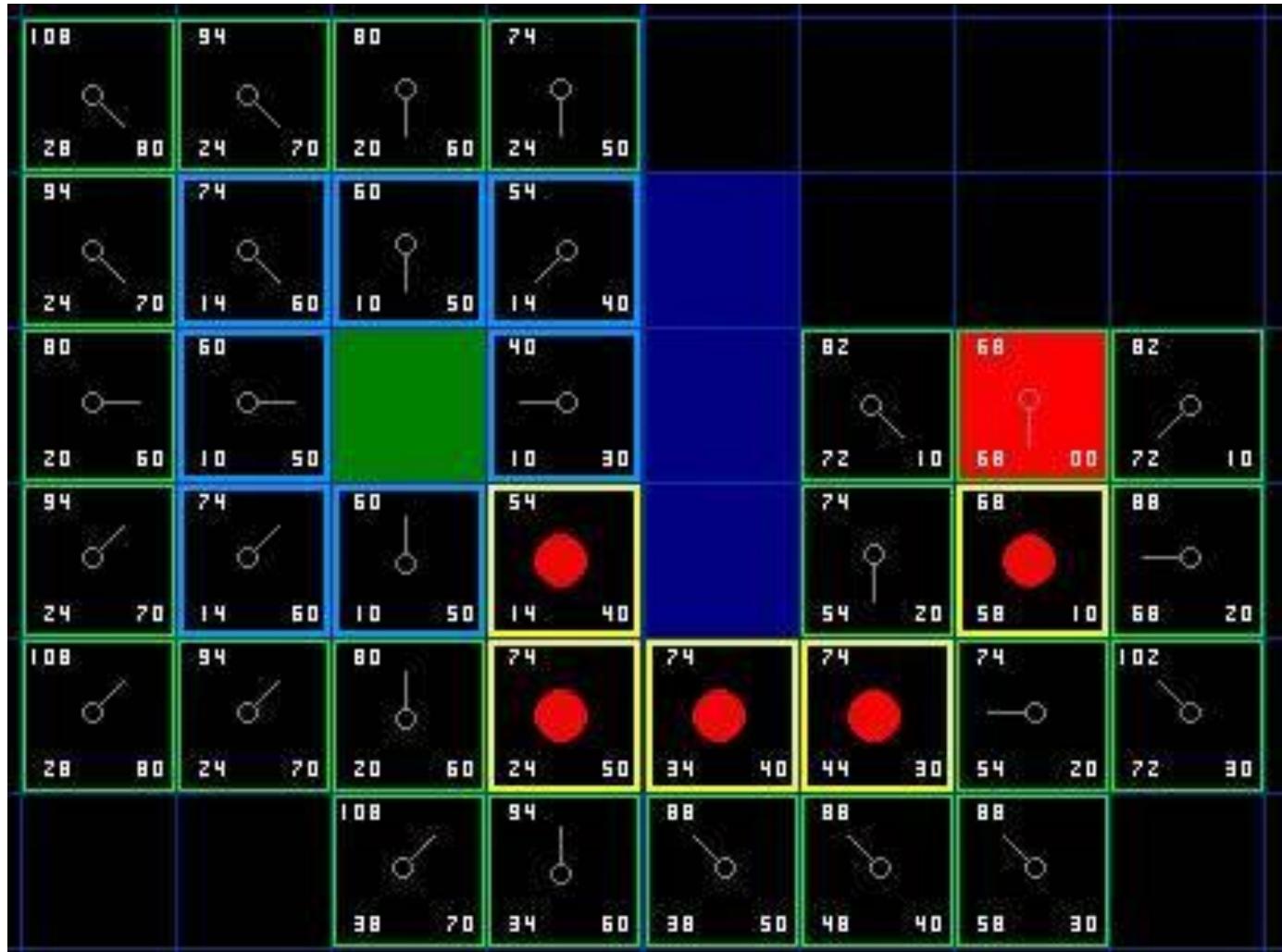
# 例5：A\*算法 Path Planning in a grid map 详解

## 4. 继续搜索

- 从开启列表中选择F值最低的方格，把它从开启列表中删除，然后添加到关闭列表中。
- 检查所有相邻格子。跳过那些已经在关闭列表中的或者不可通过的，添加到开启列表。把选中的方格作为新的方格的父节点。
- 如果某个相邻格已经在开启列表里了，检查现在的这条路径是否更好。



# 例5：A\*算法 Path Planning in a grid map 详解





# 例5：A\*算法 Path Planning in a grid map 详解

## 5. 算法总结

- 把起始格添加到开启列表。
- 重复如下的工作：
  - 寻找开启列表中F值最低的格子。
  - 把它切换到关闭列表。
  - 对相邻的8格中的每一个进行检查（检查过程见下页）。
- 保存路径。从目标格开始，沿着每一格的父节点移动直到回到起始格。这就是你的路径。



# 例5：A\*算法 Path Planning in a grid map 详解

## 5. 算法总结

- 如果它不可通过或者已经在关闭列表中，略过它。反之如下。
- 如果它不在开启列表中，把它添加进去。把当前格作为这一格的父节点。记录这一格的F,G,和H值。
- 如果它已经在开启列表中，用G值为参考检查新的路径是否更好。更低的G值意味着更好的路径。如果是这样，就把这一格的父节点改成当前格，并且重新计算这一格的G和F值。如果你保持你的开启列表按F值排序，改变之后你可能需要重新对开启列表排序。
- 停止，当你把目标格添加进了开启列表，这时候路径被找到，或者没有找到目标格，开启列表已经空了。这时候，路径不存在。



# A\*算法的最优性

保证找到最短路径（**最优解**的）条件，关键在于估价函数 $h(n)$ 的选取：

1. 估价值 $h(n) \leq n$ 到目标节点的距离实际值，这种情况下，搜索的点数多，搜索范围大，效率低。但能得到**最优解**。
2. 如果 估价值>实际值, 搜索的点数少，搜索范围小，效率高，但不能保证得到**最优解**。
3. 估价值与实际值越接近，估价函数取得就越好。



# 保证最优性的条件：可采纳性和一致性

- 采纳入发式
  - 不会过高估计到达目标的代价
  - $f(n)$ 不会超过经过节点n的解的实际代价
- 一致性（单调性）
  - 对于每个节点n和通过任一行动a生成的n的每个后续节点n'，从节点n到达目标的估计代价不大于从n到n'的单步代价与从n'到达目标的估计代价之和

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



# A\*算法的最优性

- 如果 $h(n)$ 是一致的，沿着任何路径的 $f(n)$ 的值是非递减的
- 若A\*选择扩展节点 $n$ ，就已经找到了到达节点 $n$ 的最优路径



# A\*算法的算法性能

- 完备性：完备
- 最优性：最优
- 时间复杂度： $O(b^{\varepsilon d})$ 
  - $\varepsilon$ 是启发式的相对错误
  - $d$ 是最优解所在深度
- 空间复杂度：指数级
  - 在内存保留所有已经生成的节点



# 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

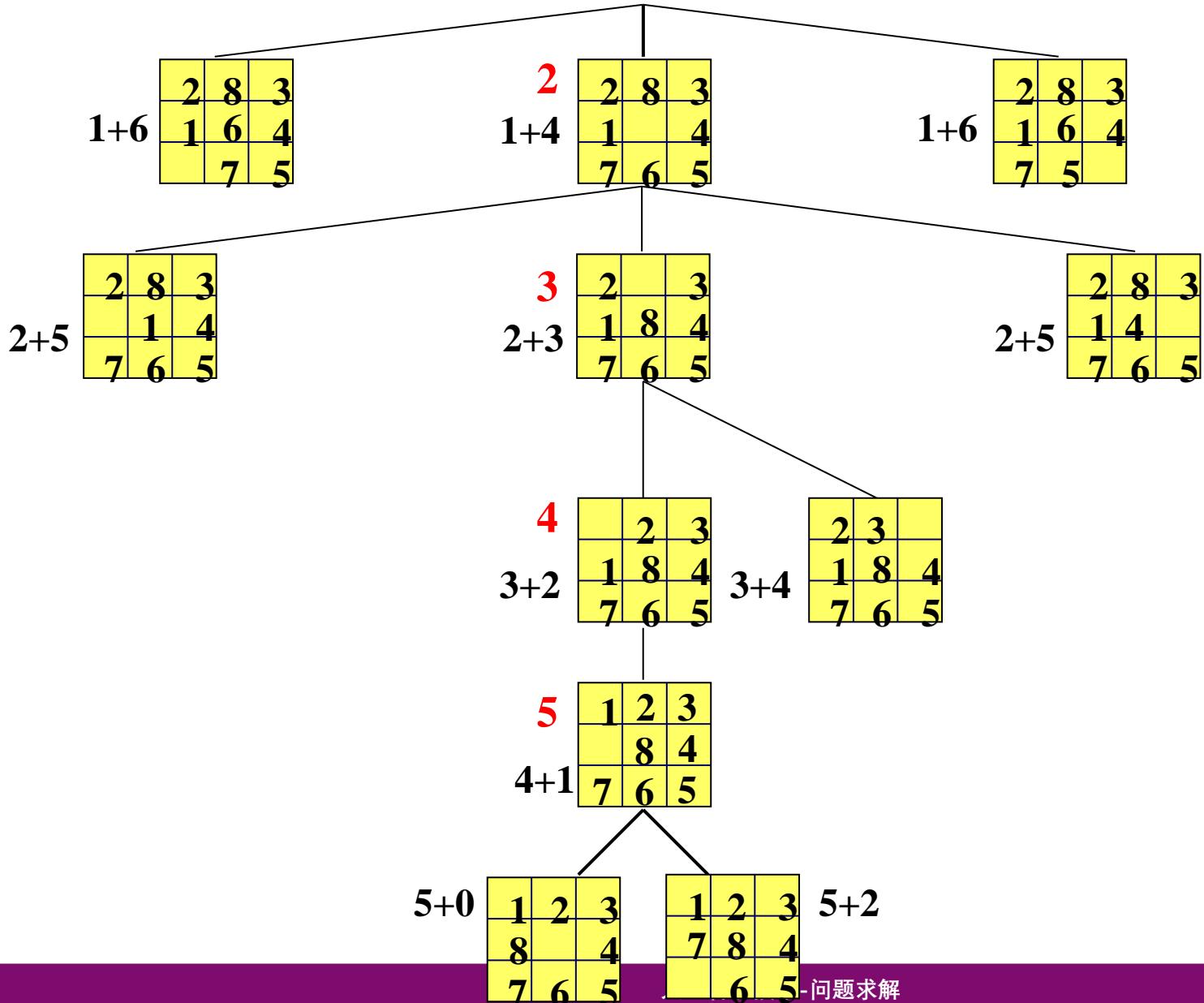
# 估价函数对算法的影响示例

## 例：八数码问题

- $f(x) = g(x) + h(x)$
- $g(x)$ : 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
  - 曼哈顿距离: 两点之间水平距离和垂直距离之和
  - 仍满足估价函数的限制条件

# 估价函数对算法的影响示例

2	8	3
1	6	4
7	5	



# Local Search

---

- ➊ Local search methods work on complete state formulations.
- ➋ They keep only a small number of nodes in memory.
- ➌ Local search is useful for solving optimization problems:
  - ▣ Often it is easy to find a solution
  - ▣ But hard to find the best solution
- ➍ Algorithm goal:
  - ▣ find optimal configuration

# Local Search

---

- ◆ Hill climbing 爬山法
- ◆ Gradient descent 梯度下降法
- ◆ Simulated annealing 模拟退火法

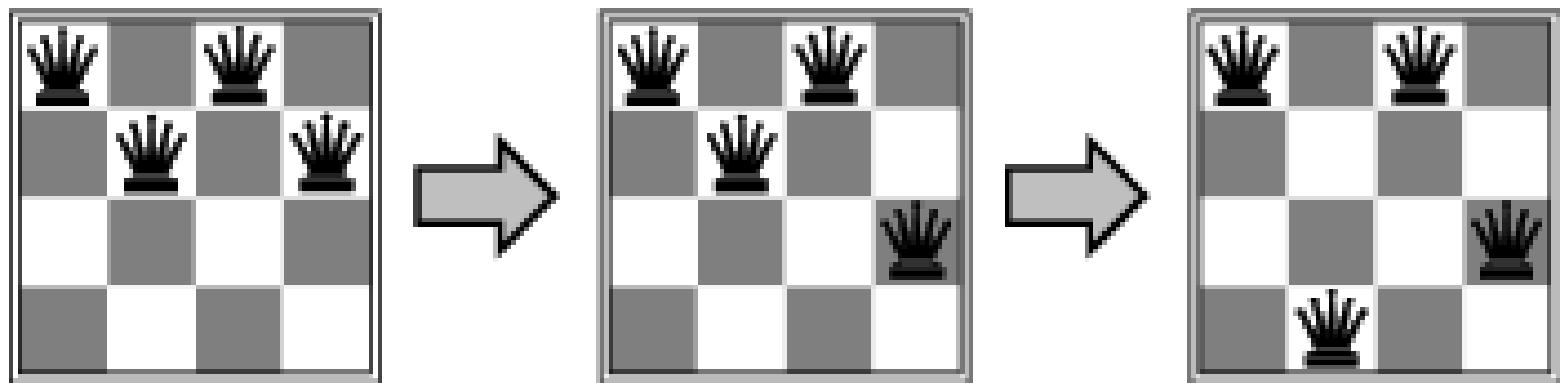


# Methodologies of Heuristic Search 3:

## 超越经典搜索

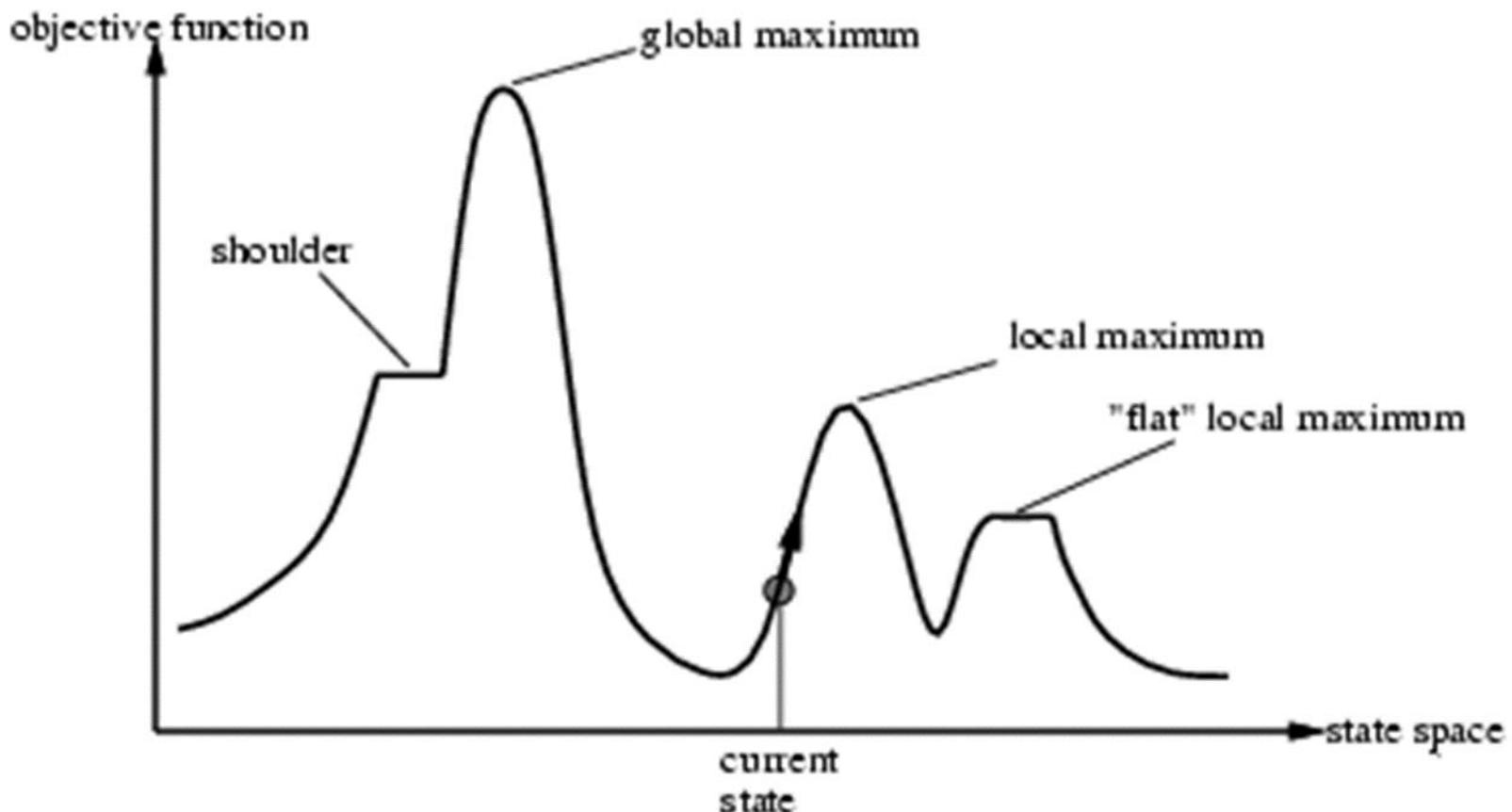
## 6.1 局部搜索算法与最优化问题

- 局部搜索(local search)算法
  - 从单个当前节点出发，移动到它的邻近状态
  - 不关心路径，不需要保留搜索路径
- 局部搜索算法的优点
  - 只使用很少的内存（一般是常数）
  - 在系统化算法不适用的很大/连续状态空间找到合理的解



## 6.1 局部搜索算法与最优化问题

- 状态空间地形图





## 6.1 局部搜索算法与最优化问题

- 爬山法
- 模拟退火搜索
- 局部束搜索
- 遗传算法

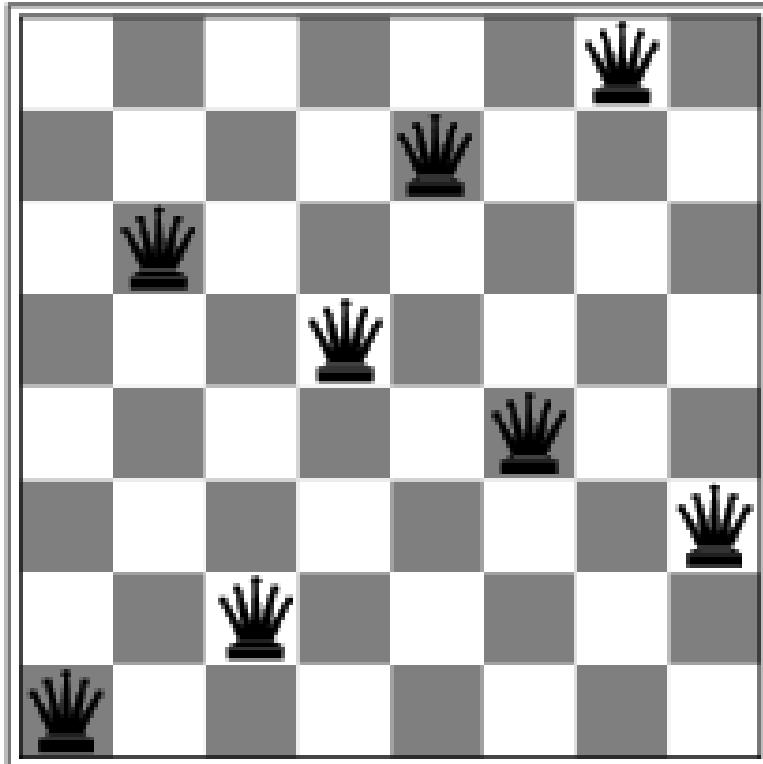
# 1、爬山法

- 爬山法(Hill-climbing search)
  - 通过简单的循环，不断向值增加的方向持续移动
  - 节点只需要记录当前状态和目标值
  - 爬山法不会考虑与当前状态不相邻的状态

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                  neighbor, a node
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor  $\leftarrow$  a highest-valued successor of current
    if VALUE[neighbor]  $\leq$  VALUE[current] then return STATE[current]
    current  $\leftarrow$  neighbor
```

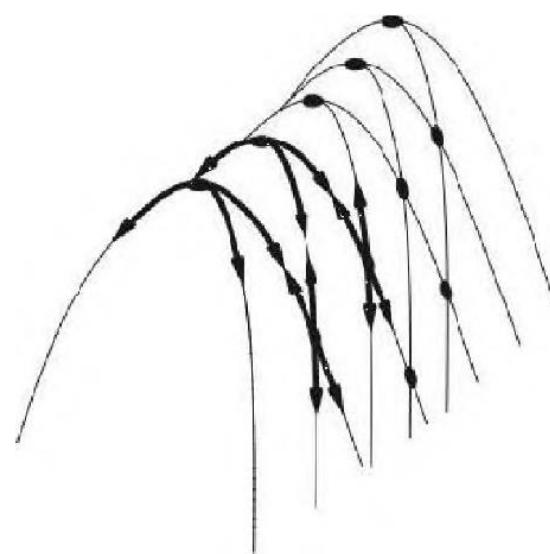
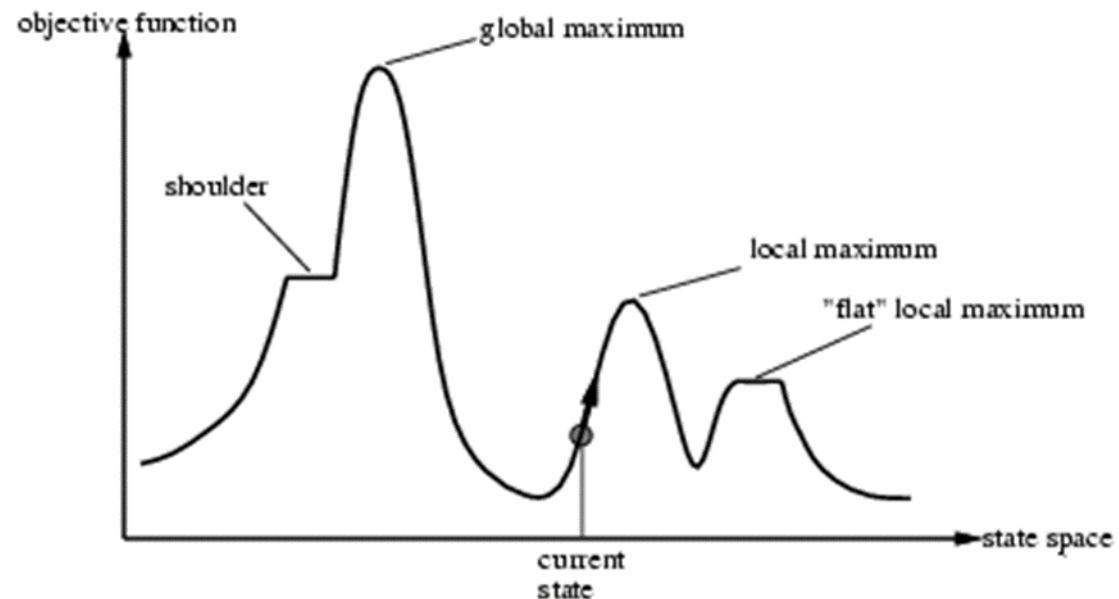
# 1、爬山法

- 八皇后问题：每列放置一个皇后
- 启发式函数 $h$ ：形成相互攻击的皇后对数量
  - 图中： $h=17$
  - 后继： $h=12$ （最好）
- 如多个后继同是最小值，爬山法会在最佳后继集合中随机选择一个进行扩展



# 1、爬山法

- 局部极大值：一个比它的每个邻接节点都高的顶峰，但是比全局最大值小。
- 山脊：造成一系列局部极大值
- 高原：在状态空间地形图上的一块平原区域





# 1、爬山法

- 爬山法的改进
  - 在平坦位置允许侧向移动，并限制连续平移次数
  - 随机爬山法：在上山移动中随机的选择下一步，被选中的概率随着上山移动的陡峭程度而不同（例如随机生成后继节点直到生成一个优于当前节点的后继——首选爬山法）
  - 随机重启爬山法（random restart hill climbing）：通过随机生成初始状态引导爬山搜索，直到找到目标
- 爬山法是否成功严重依赖于状态空间地形图的形状

## 2、模拟退火搜索

- 爬山法与随机行走法的结合——模拟退火搜索 (Simulated annealing search)

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  VALUE[next] - VALUE[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```



### 3、局部束搜索

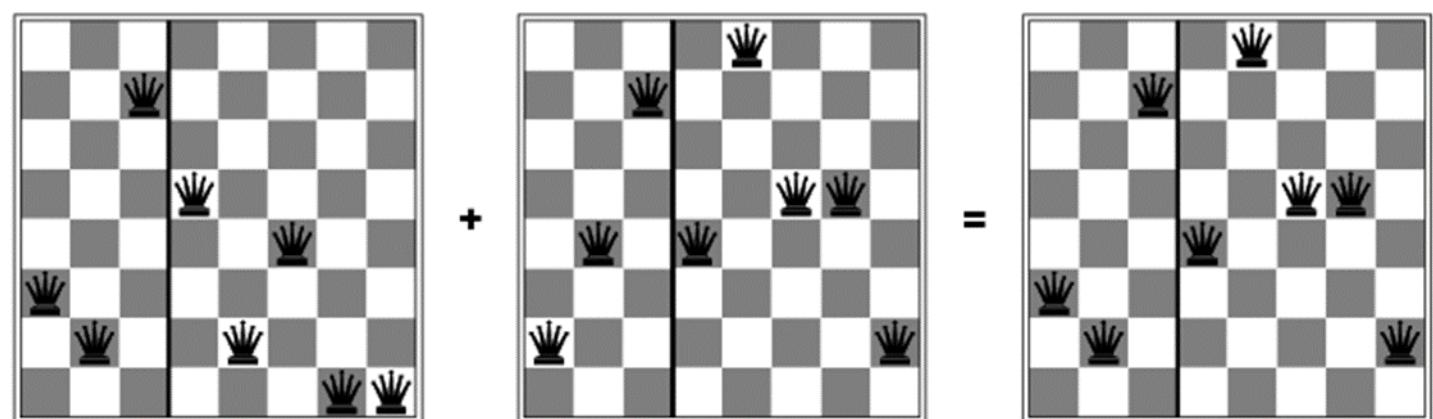
- 局部束搜索(local beam search)从 $k$ 个随机生成的状态开始，同时进行 $k$ 个搜索
- 在每一步， $k$ 个状态的所有后继状态全部生成，如果其中存在目标状态，搜索结束；否则从整个后继列表中选择 $k$ 个最优的后继，重复上述过程
- 随机束搜索(stochastic beam search)：在后继列表中，随机选择 $k$ 个后继状态。其中，选择给定后继状态的概率是状态值的递增函数。



## 4、遗传算法

- 遗传算法(genetic algorithm, GA)是随机束搜索的一个变形，它通过把两个父状态结合生成后继，而不是通过修改单一状态进行
  - 种群：k个随机生成的初始状态
  - 个体：每一个状态，用有限长度的字符串表示
  - 适应度函数：目标函数

## 4、遗传算法



## 4. 遗传算法

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population  $\leftarrow$  empty set
    loop for i from 1 to SIZE(population) do
      x  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      y  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      child  $\leftarrow$  REPRODUCE(x, y)
      if (small random probability) then child  $\leftarrow$  MUTATE(child)
      add child to new_population
    population  $\leftarrow$  new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

---

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n  $\leftarrow$  LENGTH(x)
  c  $\leftarrow$  random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

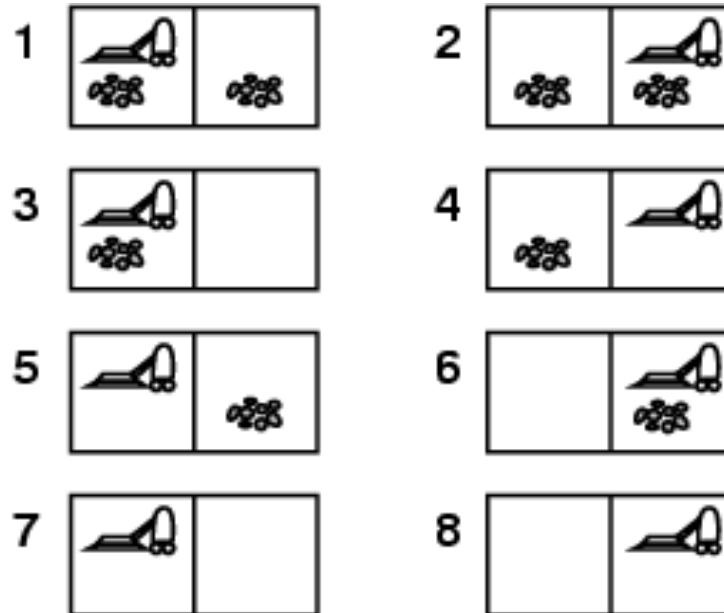


## 6.2 使用不确定动作的搜索

- 部分可观察或不确定的环境
  - 在部分可观察环境中，感知信息用于缩小Agent可能的状态范围，使Agent更容易达到目标
  - 在不确定环境中，感知信息告诉Agent某一行动的实际后果
  - Agent未来的行动依赖于未来的感知信息，问题的解不是一个行动序列，而是一个应急规划（策略）
  - 应急规划描述了根据接收的感知信息决定行动

## 6.2 使用不确定动作的搜索

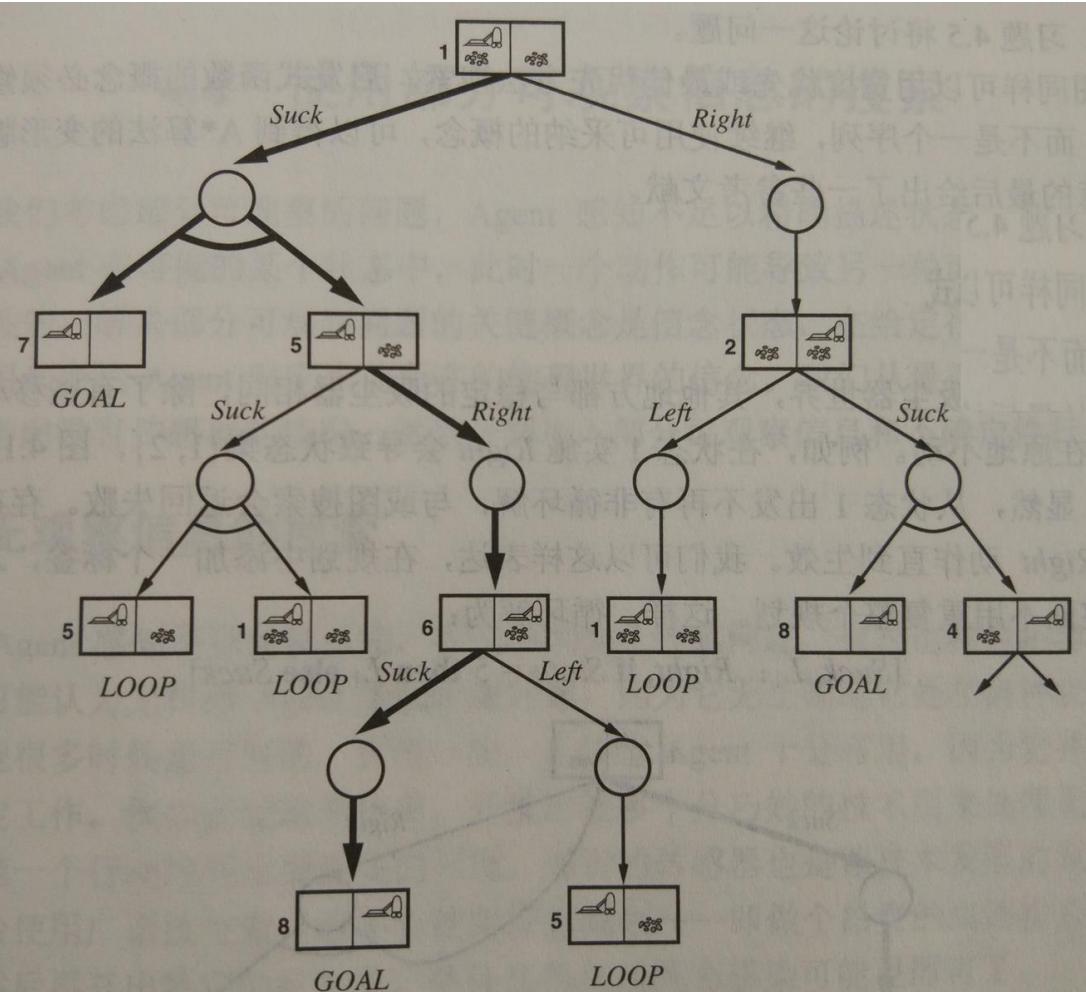
- 吸尘器世界
  - 八个状态
  - 三种行动: Left, Right, Suck
  - 目标状态: 7,8
- 不稳定的吸尘器世界
  - 脏区域吸尘可使该区域变干净, 有时也会同时清洁邻近区域
  - 干净区域吸尘, 也许会使脏东西掉到地毯上
  - 转移模型和解的推广



*[Stick, if State = 5 then [Right, Suck] else*

# 与或树搜索

- 与或树问题的解：  
一颗子树
  - 每个叶子节点都有目标节点
  - 或节点上规范一个行动
  - 与节点上包括所有可能后果



# 与或树搜索（深度优先递归算法）

```

function AND-OR-GRAFH-SEARCH(problem) returns a conditional plan, or failure
    OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
    if problem.GOAL-TEST(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.ACTIONS(state) do
        plan  $\leftarrow$  AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan  $\neq$  failure then return [action | plan]
    return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
    for each si in states do
        plani  $\leftarrow$  OR-SEARCH(si, problem, path)
        if plani = failure then return failure
    return [if s1 then plan1 else if s2 then plan2 else ... if sn-1 then plann-1 else plann]

```

Q&A

THANKS!

