

# 第四章 搜索技术

## ——1.通过搜索进行问题求解



计算机学院 赵曼



- 0 搜索技术概述
- 1 问题求解 Agent
- 2 问 题 实 例
- 3 通 过 搜 索 求 解
- 4 无 信息搜索策略
- 5 有 信息搜索策略



# 引言：搜索技术概述

## 农夫过河问题

- 有这样一个问题，农夫、狼、羊、白菜刚开始在河的岸边，现在有一条船，要使他们全部过河，最终到达河的另一岸，但在过河时有三个限制条件：
  - 只有农夫能开船
  - 船上除了农夫外只能放一样物品
  - 没有农夫看管时，狼会吃羊，羊会吃白菜请问如果你是农夫，你该如何制定过河的策略？
- 在所有策略中，究竟哪一种策略能够使农夫过河，又有哪一种策略使得过河的步骤最少，这就是搜索要解决的问题，**搜索技术就是用搜索方法寻求问题解答的技术**

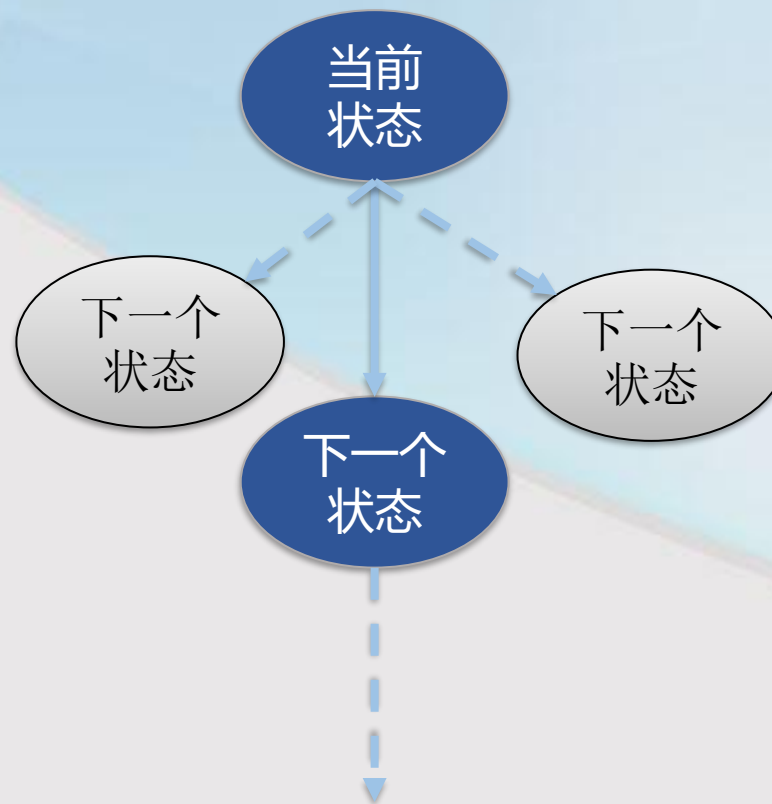


# 搜索

- 搜索是实现人工智能的一个重要组成部分，它能使机器像人一样思考，它会影响智能系统的性能和运行效率
- 搜索技术在人工智能领域有非常多的应用，在现实中能够解决许多问题，例如：
  - 在农夫过河问题中找到最优过河策略
  - 在国际象棋中，如何选择最优的下棋策略
  - 使用搜索引擎时，按照关键字呈现搜索结果

## 搜索什么

- 许多复杂的问题可以逐步解决，每走一步，问题就达到新的状态
- 当前状态和下一状态之间存在联系，做成图，就成为状态转移图
- 搜索的目标，就是在状态转移图中寻找最优的路线



# 如何搜索

- 盲目搜索
  - 对一个问题，尝试所有可能的方案，直到找到最优方案。
- 启发式搜索
  - 在搜索中加入了与问题有关的启发性信息，使得机器能够更“聪明”地实现搜索，降低尝试的次数，每一步都尽可能选择最优路径，以最（较）快的速度找到问题的解。
- 博弈搜索
  - 许多问题需要两人或者多人参与，在有对弈者参与的状态空间下进行的搜索称为博弈搜索，这种情况不仅要考虑自己每一步的策略，同时还要考虑对方可能执行的操作。





1

## 问 题 求 解 Agent

- 1.1 人工智能中问题求解
- 1.2 简单的问题求解智能体的算法
- 1.3 相关术语
- 1.4 问题形式化的五个要素

2

问 题 实 例

3

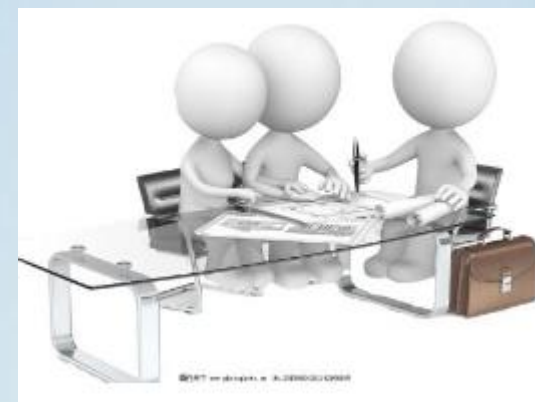
通 过 搜 索 求 解

4

无信息搜索策略

5

有信息搜索策略





# Problem solving in AI 人工智能中的问题求解

## □ The solution 解

is a sequence of actions to reach the goal.

是一个达到目标的动作序列。

## □ The process 过程

look for the sequence of actions, which is called search.

寻找该动作序列，称其为搜索。

## □ Problem formulation 问题形式化

given a goal, decide what actions and states to consider.

给定一个目标，决定要考虑的动作与状态。

## □ Why search 为何搜索

Some NP-complete or NP-hard problems, can be solved only by search.

对于某些NP完或者NP难问题，只能通过搜索来解决。

## □ Problem-solving agent 问题求解智能体

is a kind of goal-based agent to solve problems through search.

是一种基于目标的智能体，通过搜索来解决问题。

# Algorithm of Simple Problem Solving Agents 简单的问题求解智能体算法

**function** SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **return** an action

**static:** *seq*, an action sequence, initially empty

*state*, some description of the current world state

*goal*, a goal, initially null

*problem*, a problem formulation

*action*, the most recent action, initially none

*state*  $\leftarrow$  UPDATE-STATE(*state*, *percept*)

**if** *seq* is empty **then**

*goal*  $\leftarrow$  FORMULATE-GOAL(*state*)

*problem*  $\leftarrow$  FORMULATE-PROBLEM(*state*, *goal*)

*seq*  $\leftarrow$  **SEARCH**(*problem*)

**if** *seq* = failure **then return** a null action

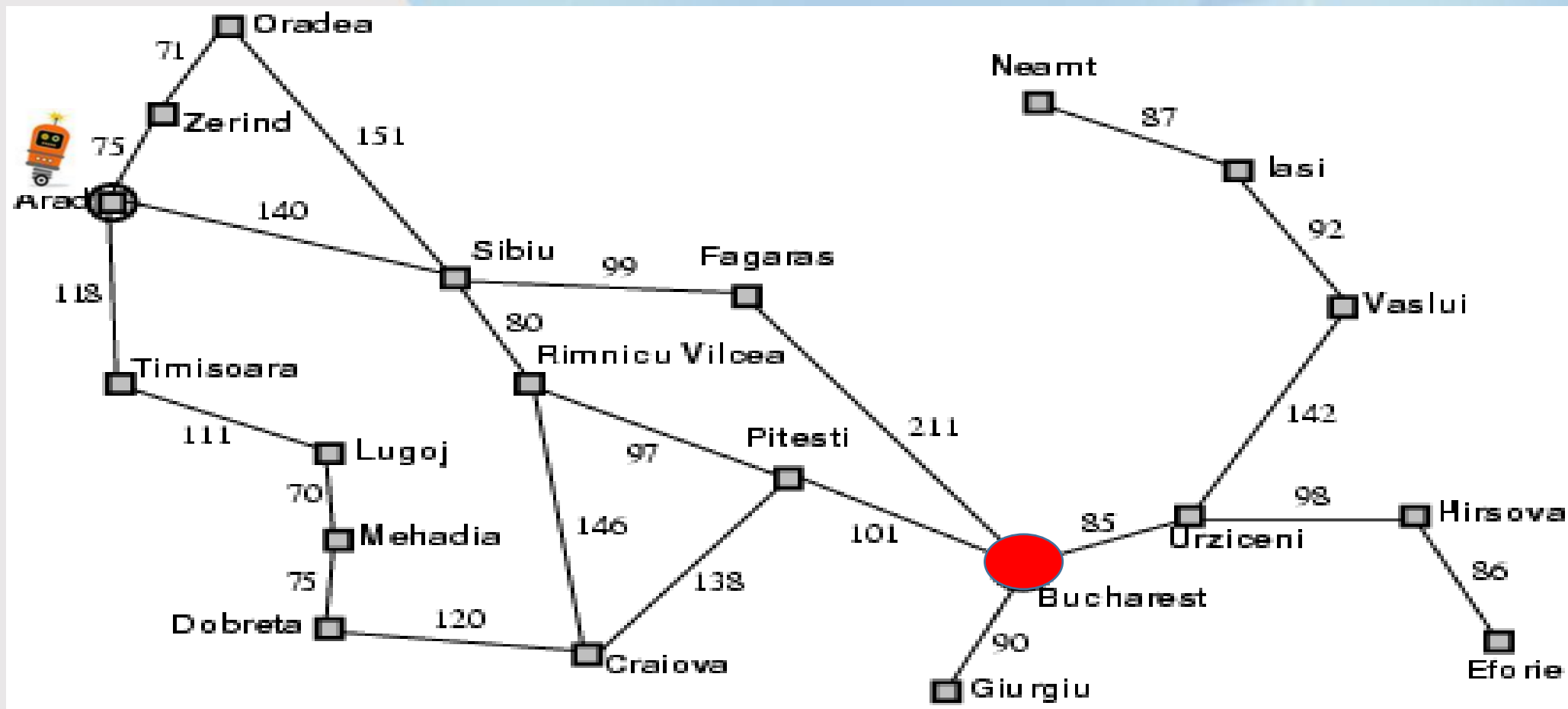
*action*  $\leftarrow$  FIRST(*seq*)

*seq*  $\leftarrow$  REST(*seq*)

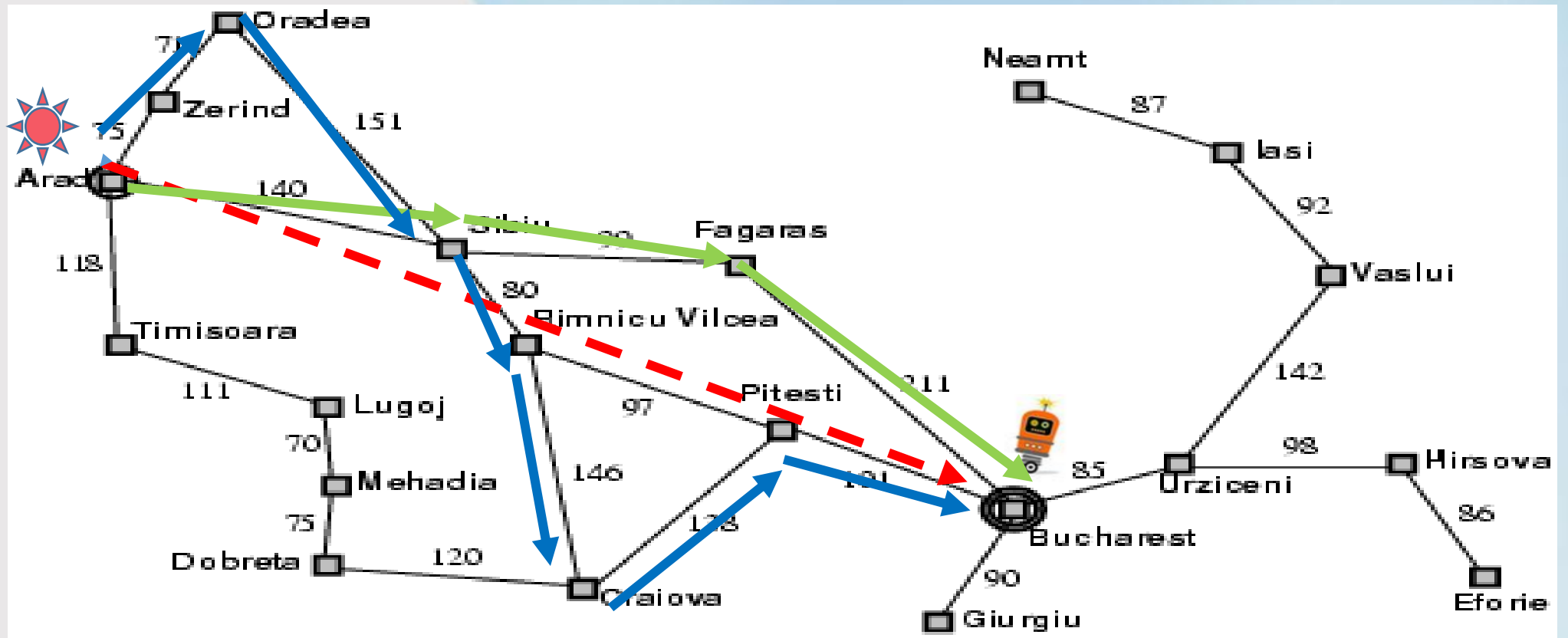
**return** *action*



## 例子:罗马尼亚Romania度假问题



# 例子:罗马尼亚Romania度假问题



## Related Terms 相关术语

### □ State space 状态空间

The state space of the problem is formally defined by: Initial state, actions and transition model.

问题的状态空间可以形式化地定义为：初始状态、动作和转换模型。。

### □ Graph 图

State space forms a graph, in which nodes are states, and links are actions.

状态空间形成一个图，其中节点表示状态、链接表示动作。

### □ Path 路径

A path in the state space is a sequence of states connected by a sequence of actions

状态空间的一条路径是由一系列动作连接的一个状态序列

# Five Items to Formulate a Problem 形式化的五个要素

## □ 1) Initial state 初始状态

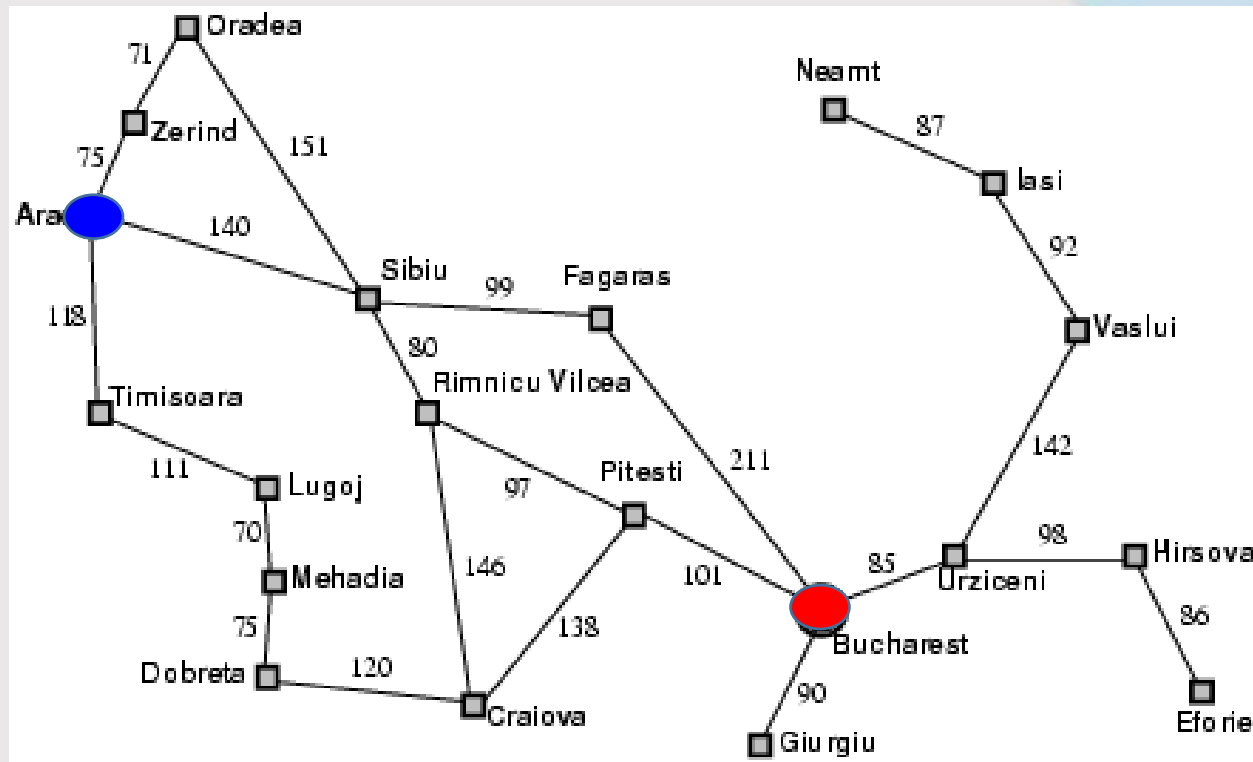
The agent starts in.

即智能体出发时的状态。

E.g., the initial state for the agent  
in Arad may be described as:

例如，该智能体位于Arad的  
初始状态可以记作：

*In(Arad).*



# Five Items to Formulate a Problem 形式化的五个要素

## 2) Actions 动作

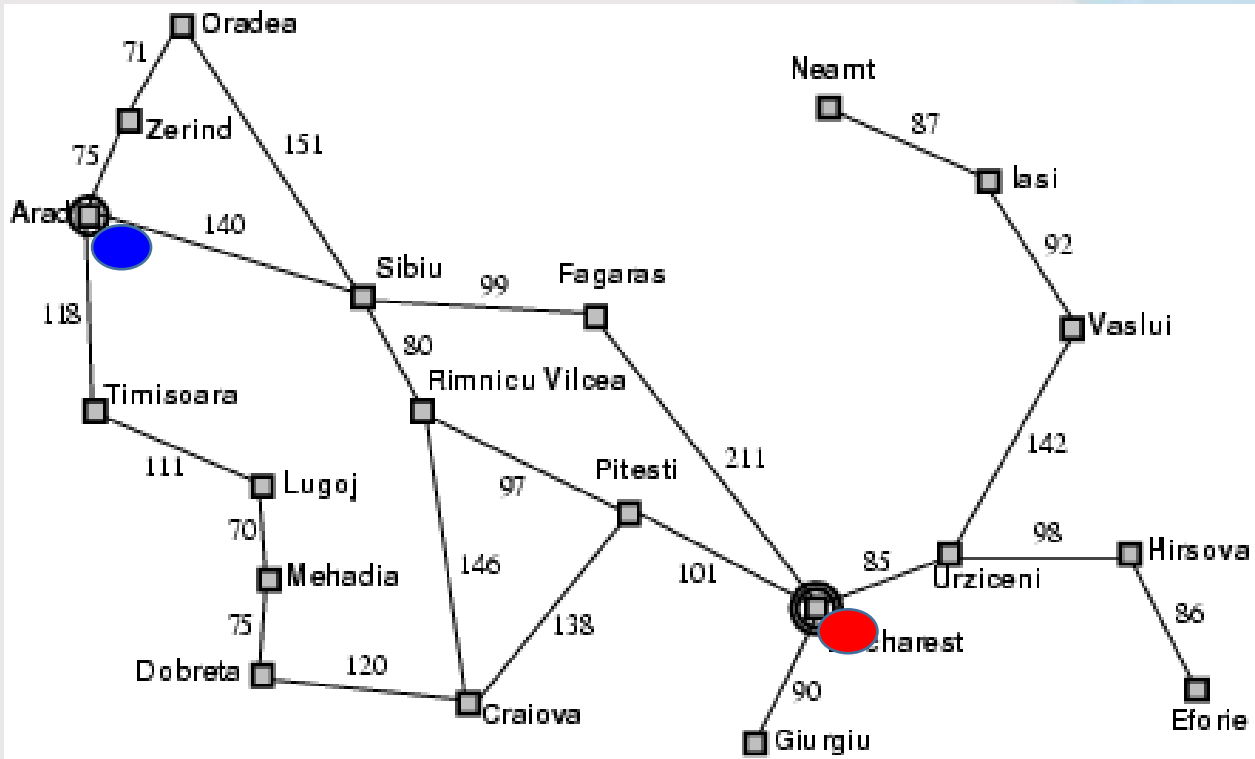
- A description of the possible actions available to the agent.

描述该智能体可执行的动作。

- **ACTION(s)** returns the actions that can be executed in **s**. E.g.,

**ACTION(s)**返回s状态下可执行的动作序列。例如，

$\{Go(Zerind), Go(Sibiu), Go(Timisoara)\}$ .





# Five Items to Formulate a Problem 形式化的五个要素

## 3) Transition model 转换模型

(后继函数)

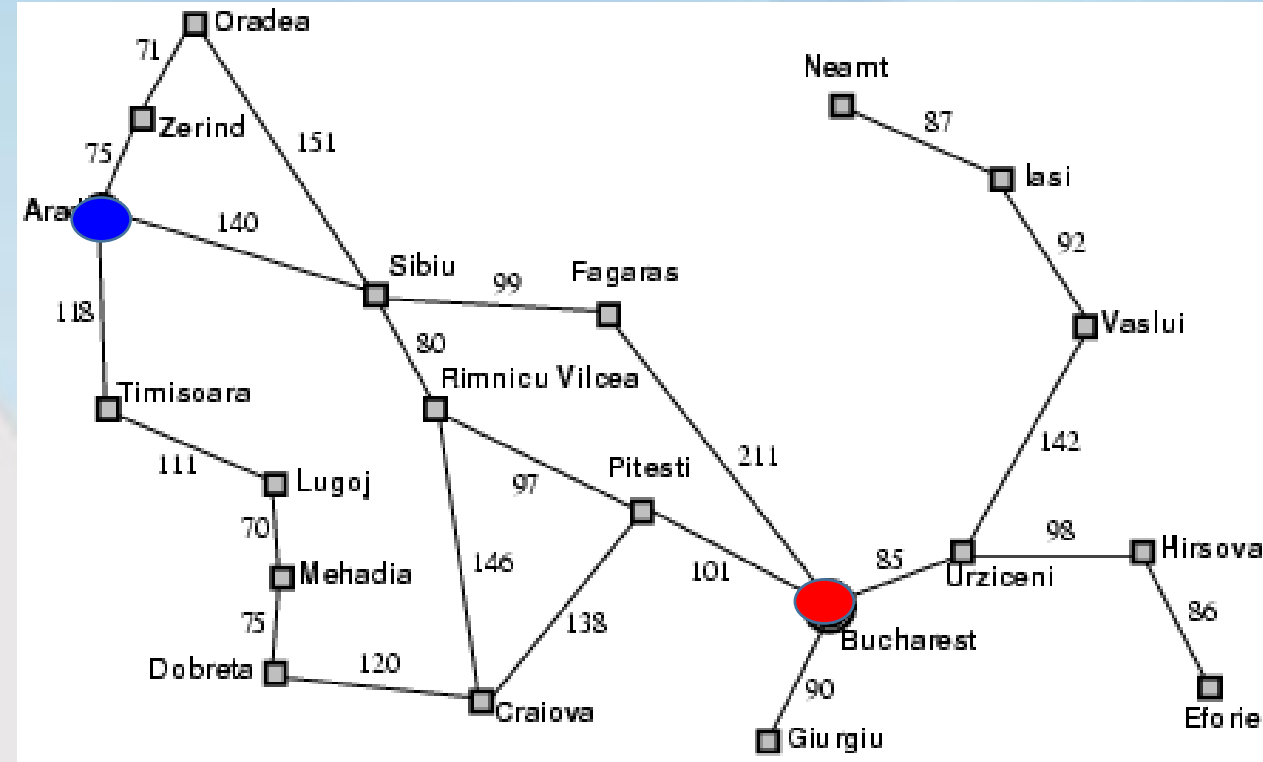
- A description of what each action does.

描述每个动作做什么。。

- **RESULT(s, a)** returns the state from doing action **a** in **s** .. E.g.,

**RESULT(s, a)** 返回 **s**, 动作 **a** 之后的状态。例如,

$$RESULT(In(Arad), Go(Zerind)) = In(Zerind)$$



# Five Items to Formulate a Problem 形式化的五个要素

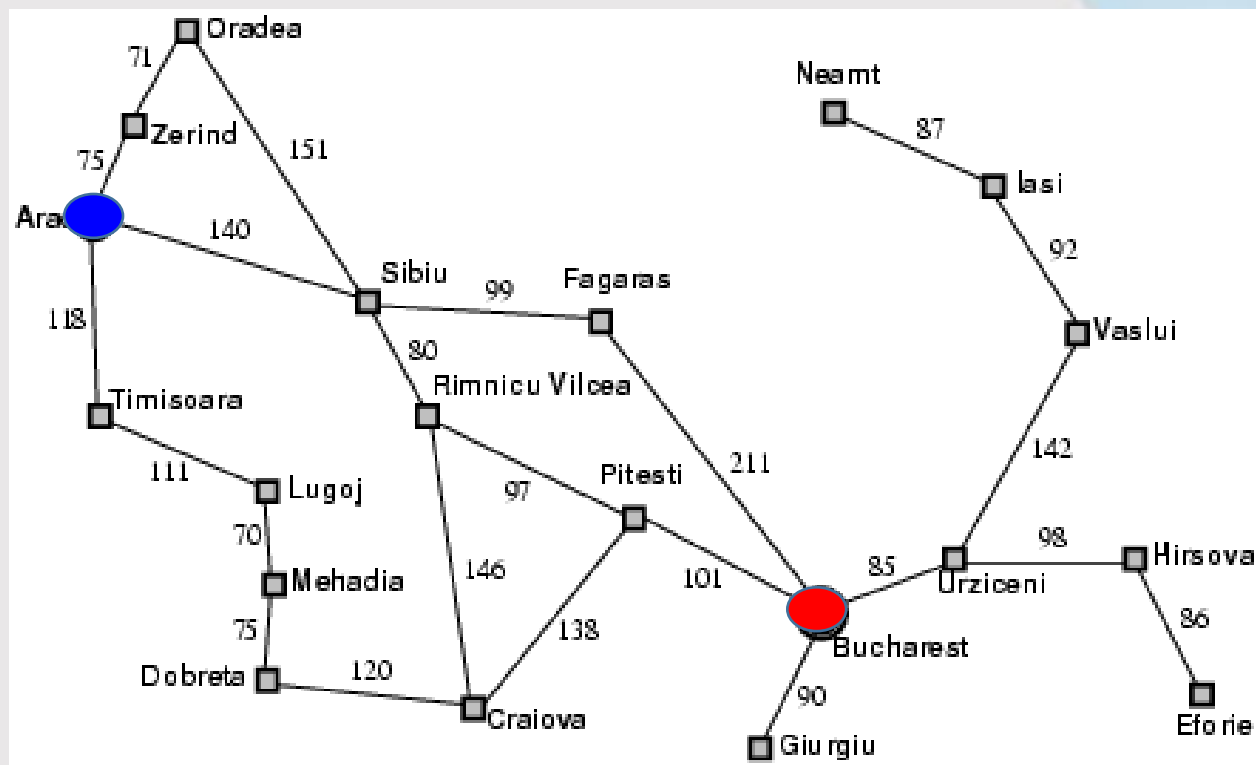
## 4) Goal test 目标测试

- To determine whether a given state is a goal state.

确定一个给定的状态是否是目标状态。

E.g., the agent's goal in Bucharest is the singleton set:

例如：智能体在 Bucharest 的目标是单元集合：  
 $\{In(Bucharest)\}$ .



# Five Items to Formulate a Problem 形式化的五个要素

## □ 5) Path cost 路径代价

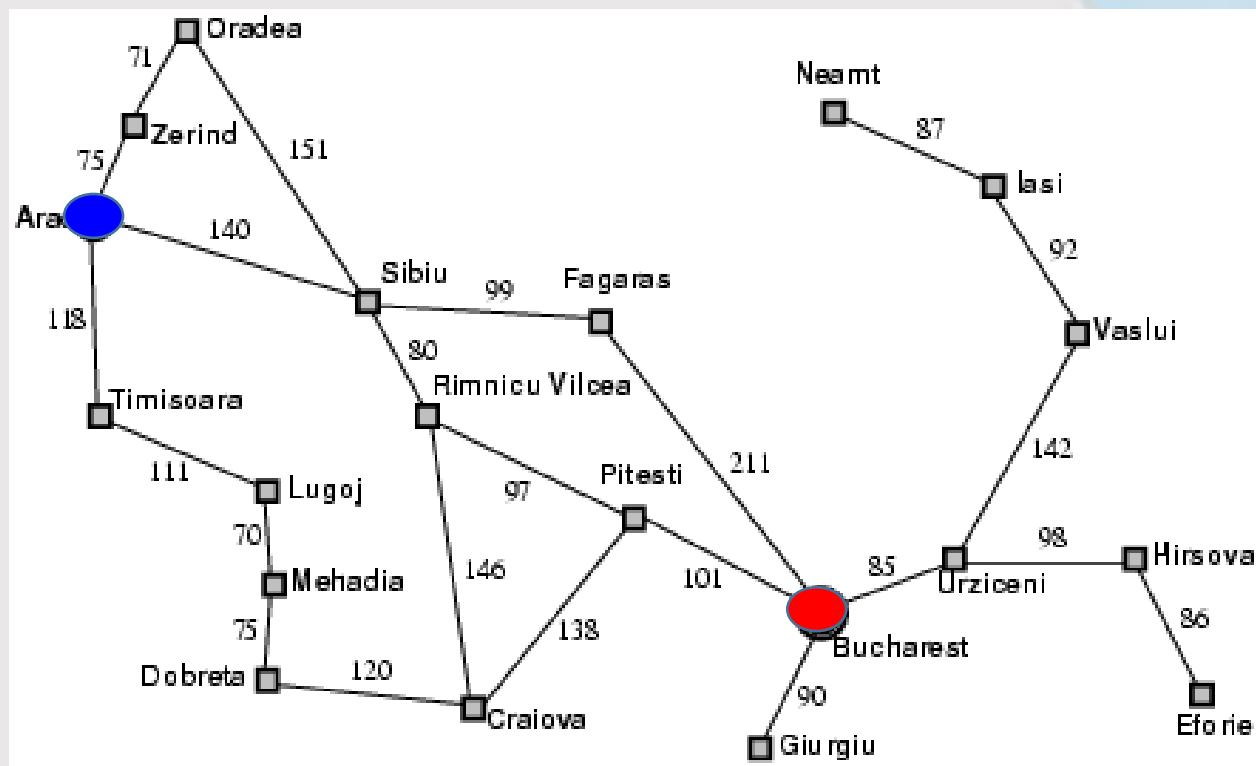
- To assign a numeric cost to each path.

即每条路径所分配的一个数值代价。

E.g., step cost of taking action **a** in state **s** to reach state **s'** is denoted by:

例如：状态 **s** 下执行动作 **a** 到达状态 **s'** 的步骤代价表示：

$$c(s, a, s').$$





1

问 题 求 解 Agent

2

问 题 实 例

2.1 Example1: Vacuum cleaner world

2.2 Example2: 8 puzzle

2.3 Example3: 8 queens problem

3

通 过 搜 索 求 解

4

无信息搜索策略

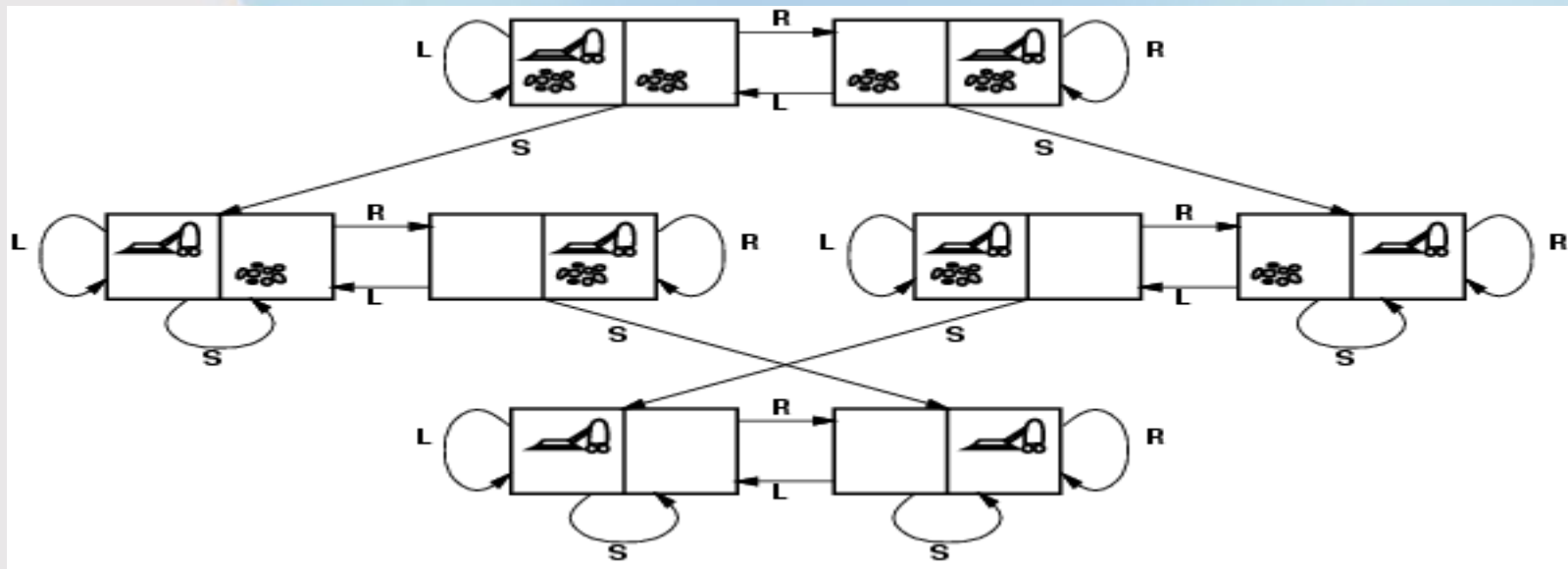
5

有信息搜索策略

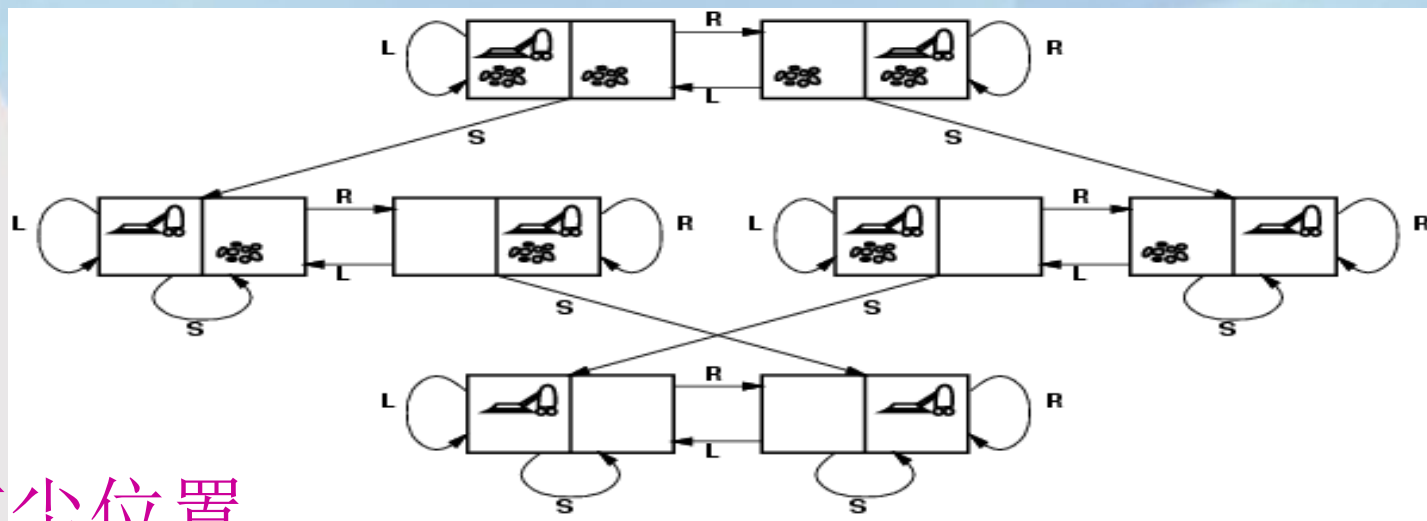


# 真空吸尘器世界状态空间图

- 状态?
- 动作?
- 转换模型?
- 目标测试?
- 路径代价?



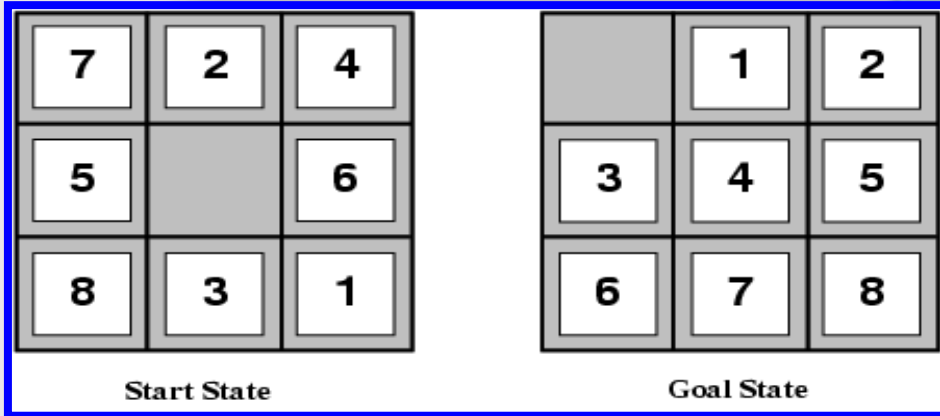
# 真空吸尘器世界状态空间图



- 状态: 机器人的位置及灰尘位置
- 动作: 向左, 向右, 吸
- 转换模型: 该动作应有的预期效果的预期效果 (除无效动作)
- 目标测试: 所有位置都干净
- 路径代价: 每做一个动作计代价为1



# 滑块游戏: 八数码问题



The 8 -puzzle belongs to the family of sliding block puzzles, this family is known be NP - complete.

8数码难题属于滑块数码难题家族，这个难题家族，这个被认为是 NP-完的。

8-puzzle:  $3 \times 3$  board with 8 numbered tiles and a blank space.

8数码难题:  $3 \times 3$ 棋盘上有8个数字棋子和一个空格。

A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state. 与空格相邻的滑块可以移向该空格，目的是达到一个指定的目标状态。



3x3 sliding puzzle.



7x7 sliding block puzzle



15-puzzle



Word puzzle



华容道



# 滑块游戏: 八数码问题

- 状态?
- 动作?
- 转换模型?
- 目标测试?
- 路径代价?

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- 状态: 8个数字块的放置情况
- 动作: 空格向上, 向下, 向左, 向右移动
- 转换模型: 移动后, 产生数字块和空格的互换
- 目标测试: 目标状态 (给定的)
- 路径代价: 每移动一步代价为1

## $(N^2-1)$ -数码的问题空间与运行时间:

- 8-puzzle  $\rightarrow 9! = 362,880$  states
- 15-puzzle  $\rightarrow 16! \sim 2.09 \times 10^{13}$  states
- 24-puzzle  $\rightarrow 25! \sim 10^{25}$  states

8-puzzle  $\rightarrow 362,880$  states

0.036 sec

15-puzzle  $\rightarrow 2.09 \times 10^{13}$  states

$\sim 55$  hours

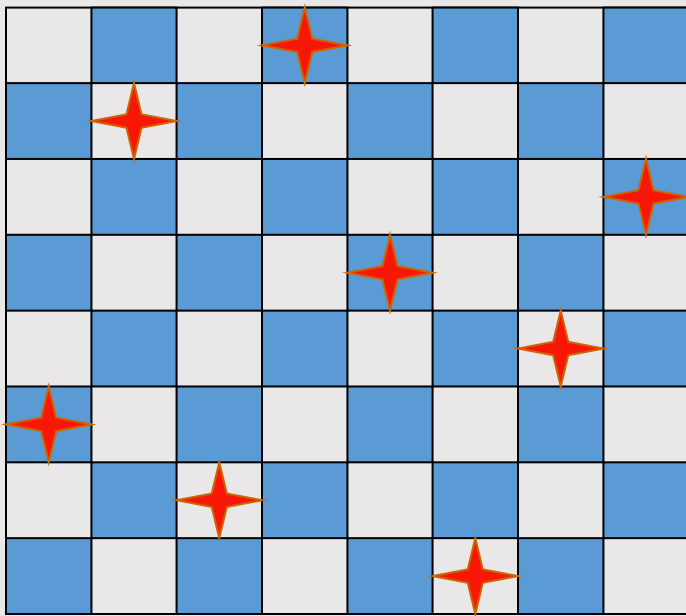
24-puzzle  $\rightarrow 10^{25}$  states  
 $> 10^9$  years

100 millions states/sec

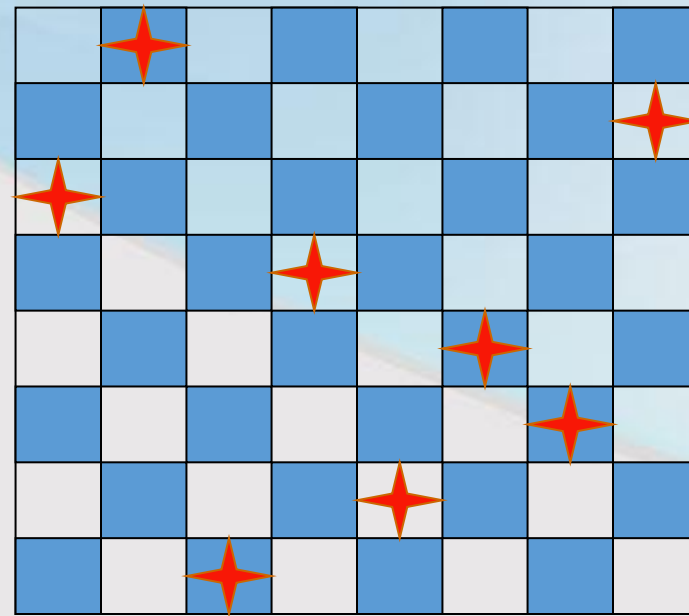
实际可达到的空间为1/2



# 八皇后问题



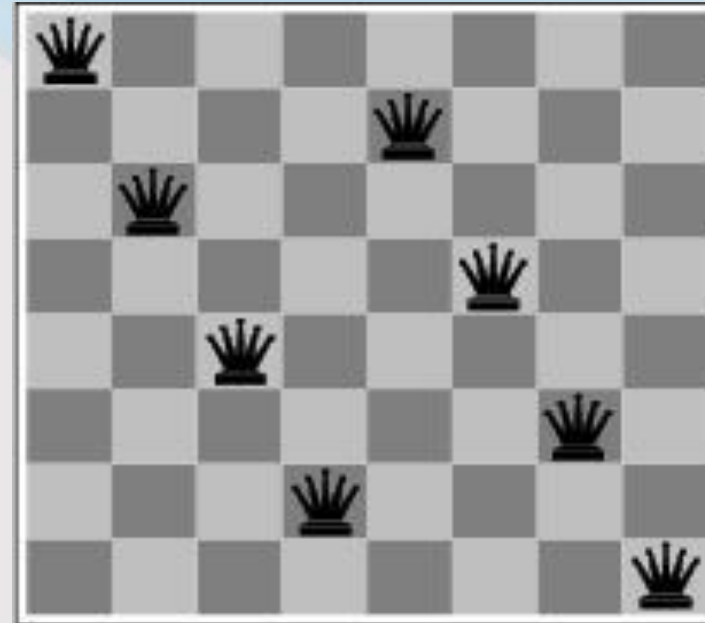
一个解



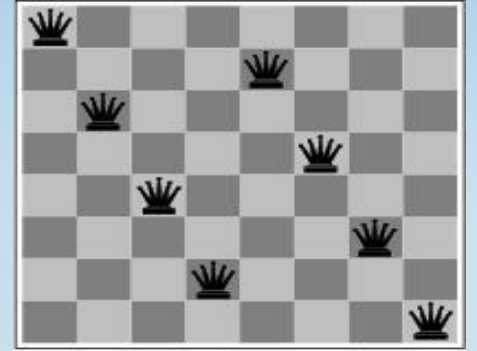
不是一个解

# 八皇后问题

- 状态??
- 初始状态??
- 行动??
- 转换模型??
- 目标测试??
- 路径代价??



# 八皇后问题



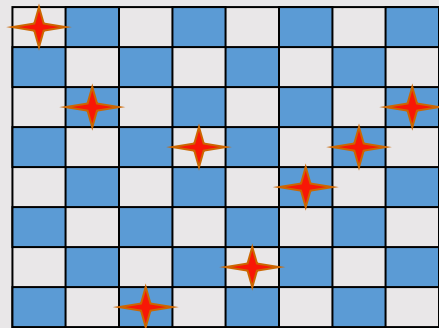
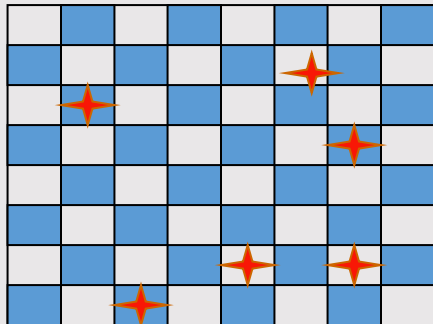
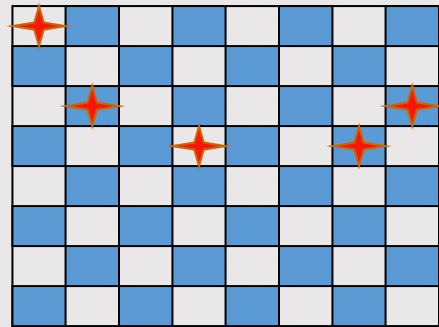
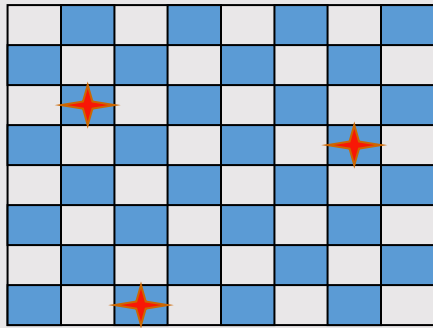
## 增量形式

vs.

## 完全状态形式化

- |          |              |     |             |
|----------|--------------|-----|-------------|
| • 状态??   | 棋盘上皇后的摆放位置   |     |             |
| • 初始状态?? | 空            | vs. | 8个放满        |
| • 行动??   | 放置一个棋子       | vs. | 移动一个棋子      |
| • 转换模型?? | 将一个棋子在空格     | vs. | 移动一个棋子到另一空格 |
| • 目标测试?? | 所有皇后互相不攻击    |     |             |
| • 路径代价?? | 无意义（或每步代价为1） |     |             |

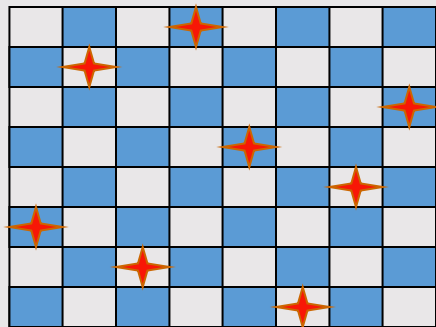
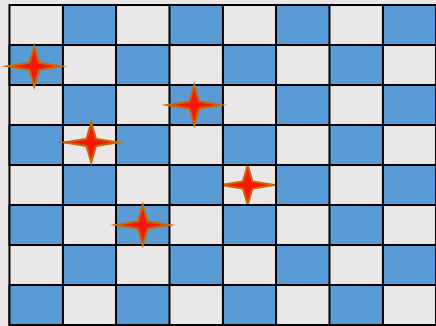
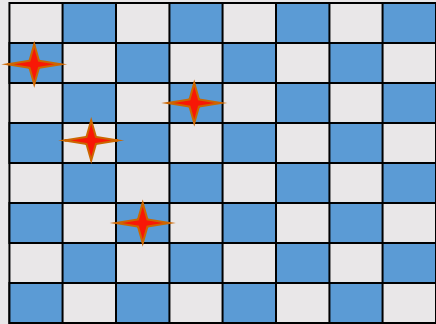
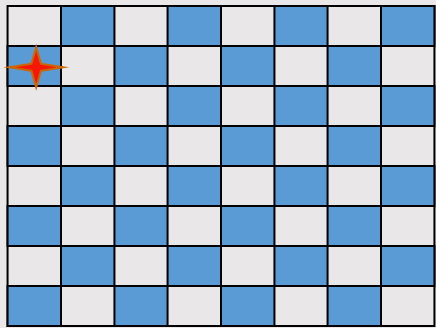
# 八皇后问题：增量形式化-1



- 状态: 0~8个皇后摆放在棋盘上
- 初始状态: 空棋盘
- 后继函数: 把一个皇后放在棋盘上的任一位置
- 路径耗散: 无关
- 目标测试: 8个皇后无冲突的放在棋盘上

→  $64 \times 63 \times \dots \times 57 \sim 3 \times 10^{14}$  states

# 八皇后问题：增量形式化-2



- 状态: 摆放 $k$ 个皇后 ( $0 \leq k \leq 8$ ) 的安排, 要求最左边 $k$ 列里每列一个皇后, 保证没有一个皇后能相互攻击;
- 初始状态: 盘上没有皇后
- 模型转换: 把一个皇后添加到最左边空列的任何格子内, 只要该位子不被攻击;
- 路径耗散: 无意义
- 目标: 8个皇后都在盘上

→ 2,057 states



## N皇后问题启示:

- 特点: 一个解是最终的节点而不是一条路径;
- 状态空间的数目: 100个皇后问题初始状态有 $10^{400}$ 个状态, 改进后为 $10^{52}$ , 但仍然是很大;
- 但是技术上是存在解决n很大的皇后问题;
- 通过对状态空间分布的研究可以很好的解决。



## 四个基本步骤:

### 1、目标形式化 (Goal formulation)

- 成功的状态描述

### 2、问题形式化 (Problem formulation)

- 根据所给的目标考虑行动和状态的描述

### 3、搜索 (Search)

- 通过对行动序列代价计算来**选取**最佳的行动序列.

### 4、执行 (Execute)

- 给出“解”执行行动.

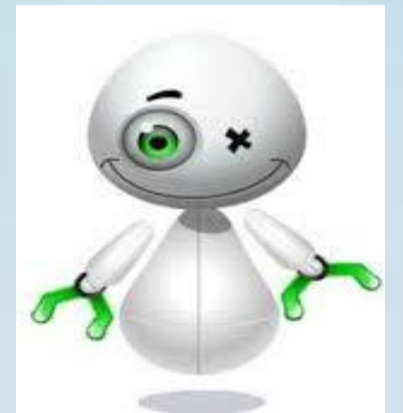
# 问题求解智能体



问题求解：形式化——搜索——执行

# 问题形式化

- 一个**问题**可以由四个组成部分来**形式化**:
  - 1、初始状态
  - 2、动作
  - 3、转换模型或后续函数
  - 4、目标测试
  - 5、路径代价
- 一个**解**是从初始状态到目标状态的动作系列  
解、代价、耗散值、最优解、无解.....





1

问题求解 Agent

2

问题实例

3

通过搜索求解

3.1 问题求解的方法

3.2 采用树搜索方法

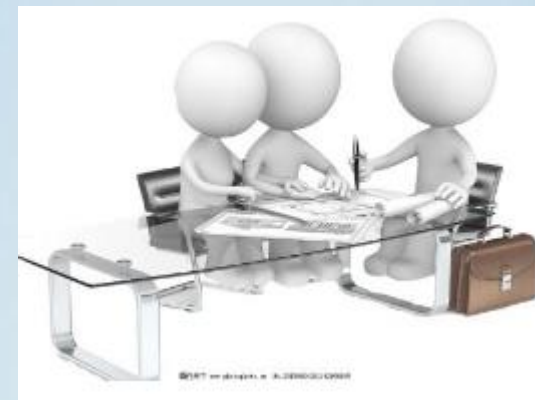
3.3 采用图搜索方法

4

无信息搜索策略

5

有信息搜索策略



## 3.1 问题求解

### ➤ 我们怎么找到解决问题的方法？

- 搜索状态空间（记住空间的复杂性取决于状态表示）
  - 通过显式生成树搜索
    - ✓ 根=初始状态。
    - ✓ 节点和叶子通过后继函数生成。
  - 在一般的搜索生成图（相同的状态，通过多条路径）

### ➤ 基本思想：

通过产生已经探索到的状态的后续状态的方法来离线地进行状态空间的模拟搜索。

## 3.2 树搜索方法

基本思想：

- 通过产生已经探索到的状态的后续状态的方法来离线地进行状态空间的模拟搜索。

function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize search tree to the *initial state of the problem*

loop do

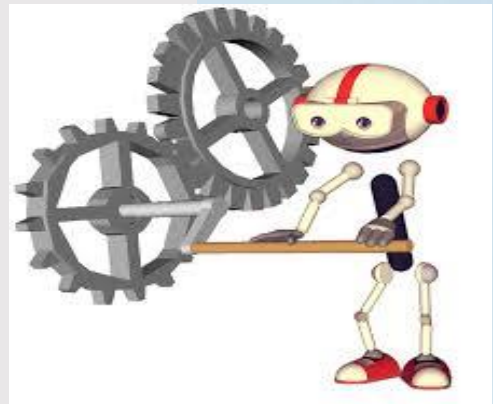
if no candidates for expansion then return *failure*

choose leaf node for expansion according to *strategy*

if node contains goal state then return *solution*

else expand the node and add resulting nodes to the search tree

enddo





# 罗马尼亚问题的树搜索过程

function TREE-SEARCH(*problem, strategy*) return a solution or failure

Initialize search tree to the *initial state of the problem*

do

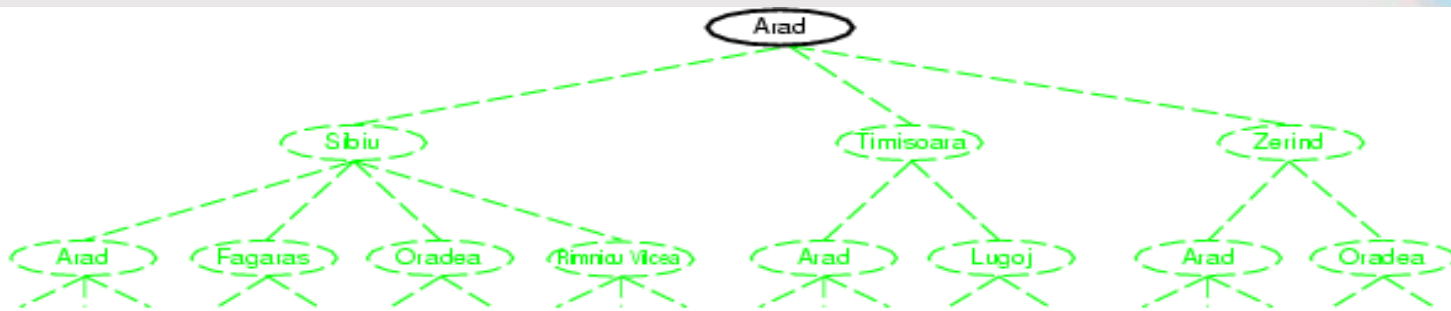
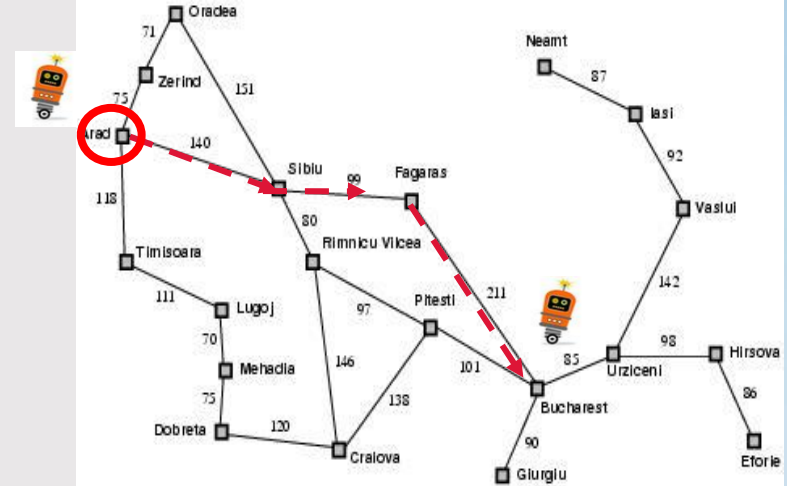
if no candidates for expansion then return *failure*

choose leaf node for expansion according to *strategy*

if node contains goal state then return *solution*

else expand the node and add resulting nodes to the search tree

enddo

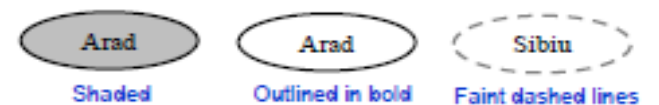


注：

阴影：表示该节点已被扩展。

粗实线：表示该节点已被生成，但尚未扩展。

浅虚线：表示该节点尚未生成。





# 罗马尼亚问题的树搜索过程

**function** TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

**do**

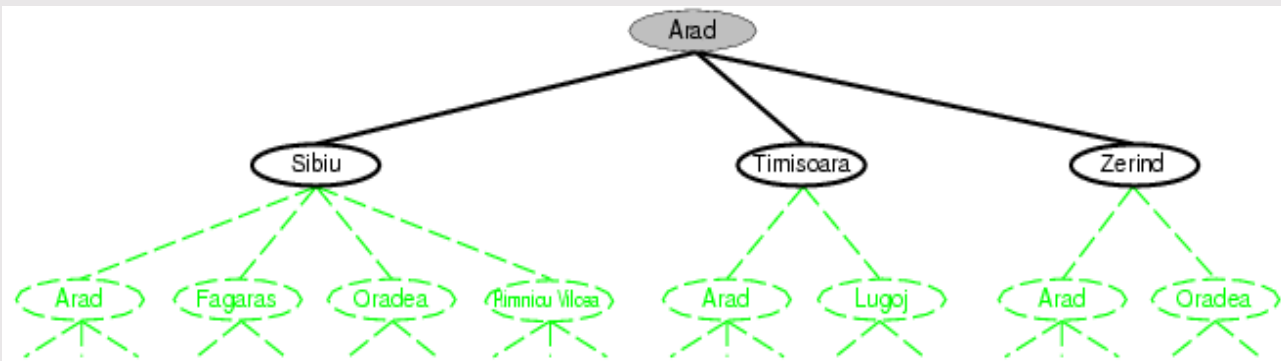
**if** no candidates for expansion **then return** failure

**choose** leaf node for expansion according to *strategy*

**if** node contains goal state **then return** solution

**else** expand the node and add resulting nodes to the search tree

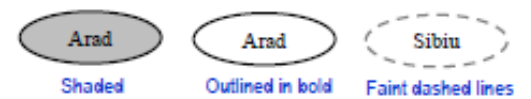
**enddo**



注：

阴影：表示该节点已被扩展。  
粗实线：表示该节点已被生成，但尚未扩展。

浅虚线：表示该节点尚未生成。



## 罗马尼亚问题的树搜索过程

**function** TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

**do**

**if** no candidates for expansion **then return** *failure*

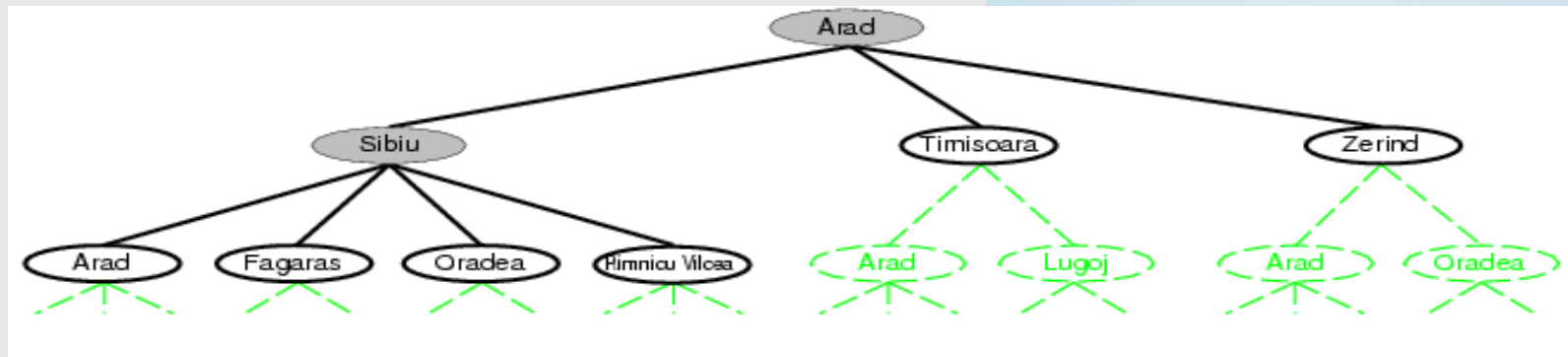
**choose** leaf node for expansion according to *strategy*

**if** node contains goal state **then return** *solution*

**else** expand the node and add resulting nodes to the search tree

**enddo**

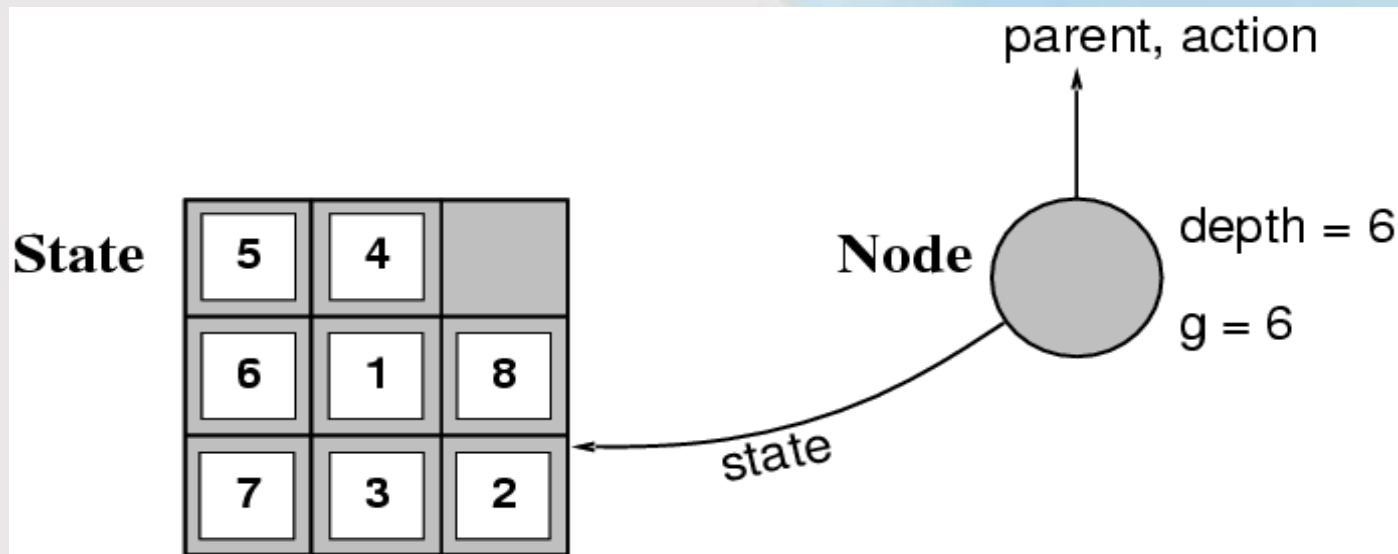
Determines search process!!



# 状态与节点

- 一个状态是一个物理配置的表示，而一个节点是一个数据结构，它的组成部分。

它包含：状态、父亲节点、动作、路径代价和深度



# 树搜索算法实现:

function TREE-SEARCH(*problem*, *fringe*) return **a solution** or **failure**

*fringe*  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)

**loop do**

if EMPTY?(*fringe*) then **return failure**

*node*  $\leftarrow$  REMOVE-FIRST(*fringe*)

if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds

then **return SOLUTION**(*node*)

*fringe*  $\leftarrow$  INSERT-ALL(**EXPAND**(*node*, *problem*), *fringe*)

注：该 *fringe*（亦称 *open list*）：一种数据结构，用于存储所有的叶节点。

```
function EXPAND(node, problem) return a set of nodes
    successors  $\leftarrow$  the empty set
    for each  $\langle$ action, result $\rangle$  in SUCCESSOR-FN[problem](STATE[node]) do
        s  $\leftarrow$  a new NODE
        STATE[s]  $\leftarrow$  result
        PARENT-NODE[s]  $\leftarrow$  node
        ACTION[s]  $\leftarrow$  action
        PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s]  $\leftarrow$  DEPTH[node]+1
        add s to successors
    return successors
```

### 3.3 通用图搜索算法

**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure

initialize the ***fringe*** using the initial state of *problem*

initialize the ***explored*** to be empty

**loop do**

if the *fringe* empty **then return** failure

**choose** a leaf node and remove it from the *fringe*

if the node contains a goal state **then return** the corresponding solution

add the node to the *explored*

expand the chosen node, adding the resulting nodes to the *fringe*

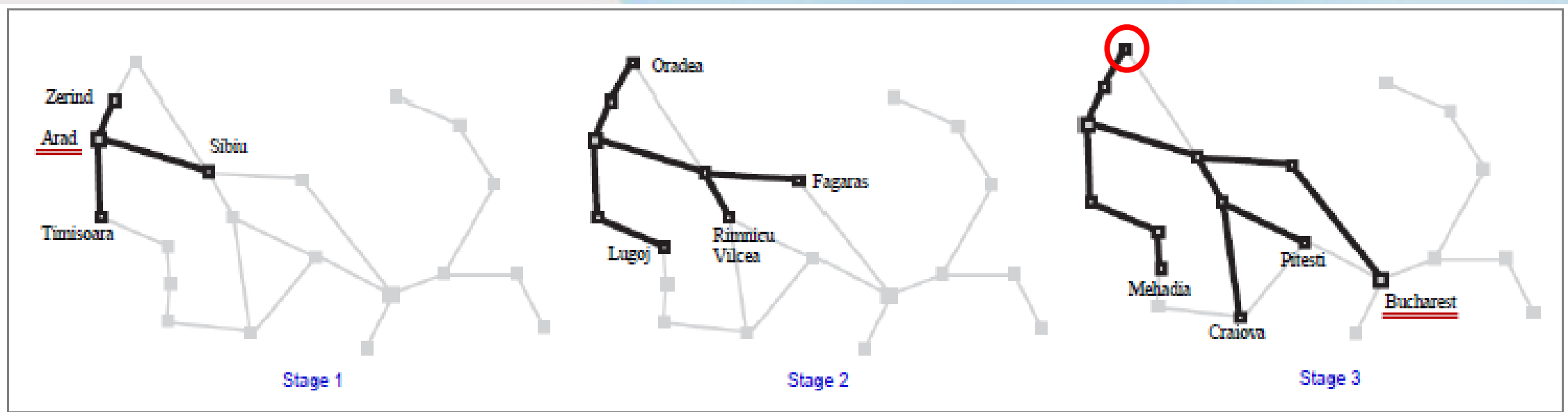
only if not in the *fringe* or *explored*

注：该 *explored*（亦称 *closed list*）：一种数据结构，用于存储每个已扩展节点。

该算法中采用，将在 *explored* 或 *fringe* 中的曾出现过的节点丢弃的方法。

# 罗马尼亚问题的图搜索过程

通过图搜索在该罗马尼亚地图上生成一系列搜索路径。



每个路径在每个阶段 通过每一步加以扩展。

注意在第 3 阶段，最北部城市 (Oradea) 的后继节点均已出现过，所以，相当于成为一个dead end.





1 问题求解 Agent

2 问题实例

3 通过搜索求解

4 无信息搜索策略

- 4.1 宽度优先搜索
- 4.2 代价一致搜索
- 4.3 深度优先搜索
- 4.4 深度优先改进
- 4.5 双向搜索
- 4.6 无信息搜索的比较

5 有信息搜索策略



# 盲目搜索策略

- 无信息搜索又名**盲目搜索**：
  - 在搜索时，只有问题定义信息可用。
  - 在搜索时，当有策略可以确定一个非目标状态比另一种更好的搜索，称为**有信息的搜索**。
- 盲目搜索策略仅利用了问题定义中的信息，所有的搜索策略是由节点扩展的顺序加以区分。
  - 宽度优先搜索
  - 深度优先搜索
  - 迭代深度搜索
  - 代价一致搜索
  - 深度有限搜索
  - 双向搜索



# 搜索策略评价

- 搜索策略指节点扩展顺序的选择。
- 搜索策略的**性能**由下面四个方面来评估：
  - **完备性**: 如果问题的解存在时它总能找到解。
  - **时间复杂性**: 产生的节点个数。
  - **空间复杂性**: 搜索过程中内存中的最大节点数。
  - **最优性**: 它总能找到一个代价最小的解。

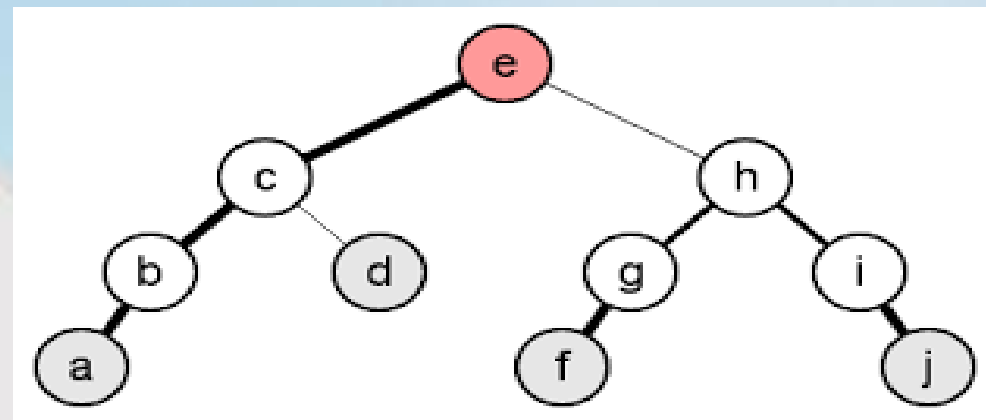


# 搜索策略评价

- **问题难度**由时间和空间复杂度的定义来度量的:

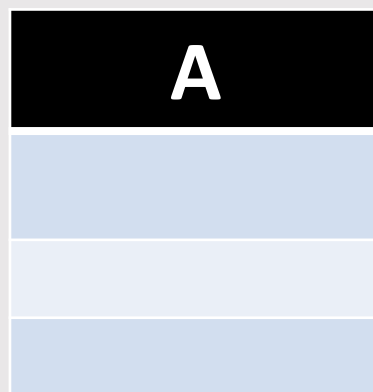
✓时间和空间复杂度根据下面三个量来表达:

- $b$ : 搜索树的最大分支数
- $d$ : 最小代价解所在的深度
- $m$ : 状态空间的最大深度(可能是 $\infty$ )

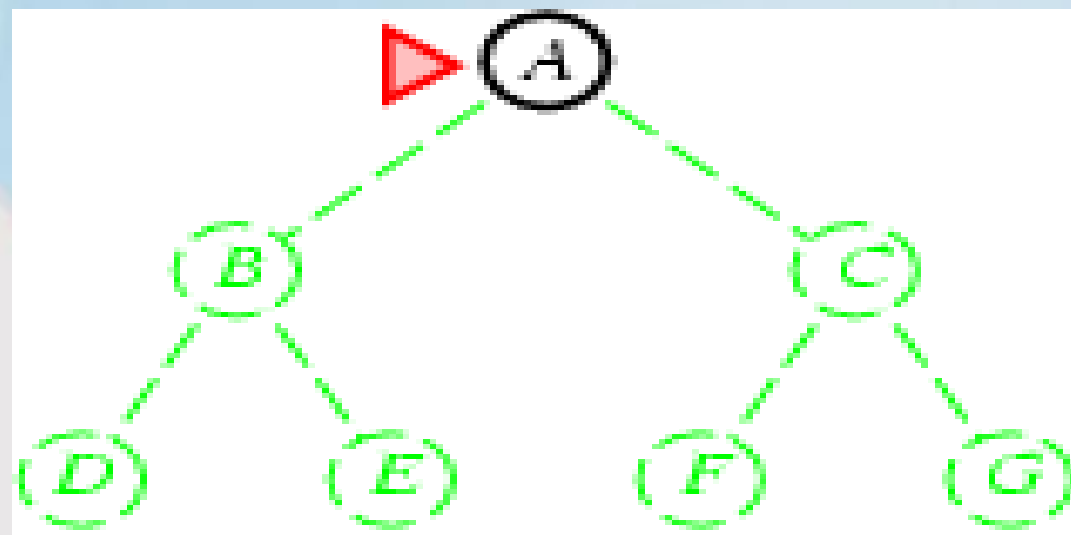


## 4.1 宽度优先搜索

- 优先扩展最浅层的未扩展节点
- 实现方法:
  - **Fringe**表采用先进先出队列（**FIFO queue**），即新的后续节点总是放在队列的末尾

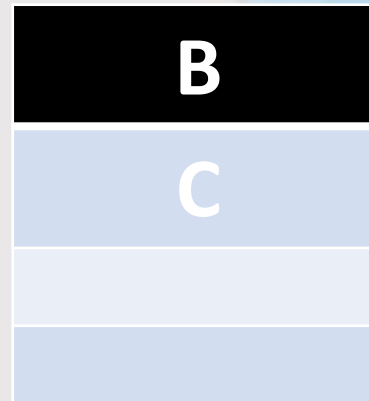


Fringe表

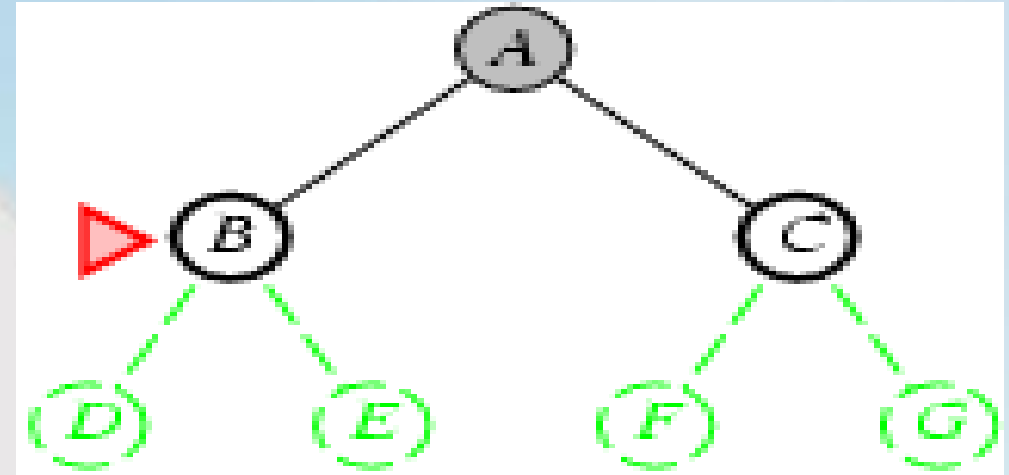


# 宽度优先搜索

- 优先扩展最浅层的未扩展节点
- 实现方法:
  - **Fringe**表采用先进先出队列（**FIFO queue**），即新的后续节点总是放在队列的末尾



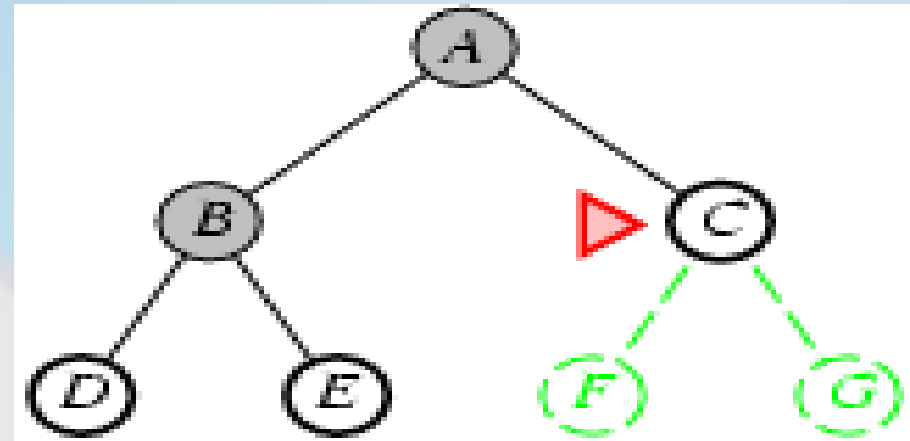
Fringe表



# 宽度优先搜索

- 优先扩展最浅层的未扩展节点
- 实现方法:
  - **Fringe**表采用先进先出队列（ **FIFO queue**），即新的后续节点总是放在队列的末尾

C
D
E

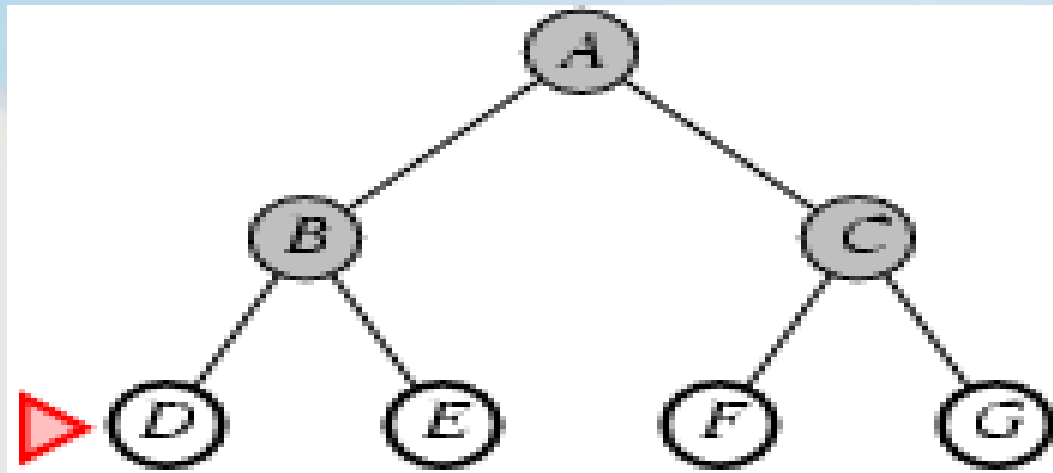




# 宽度优先搜索

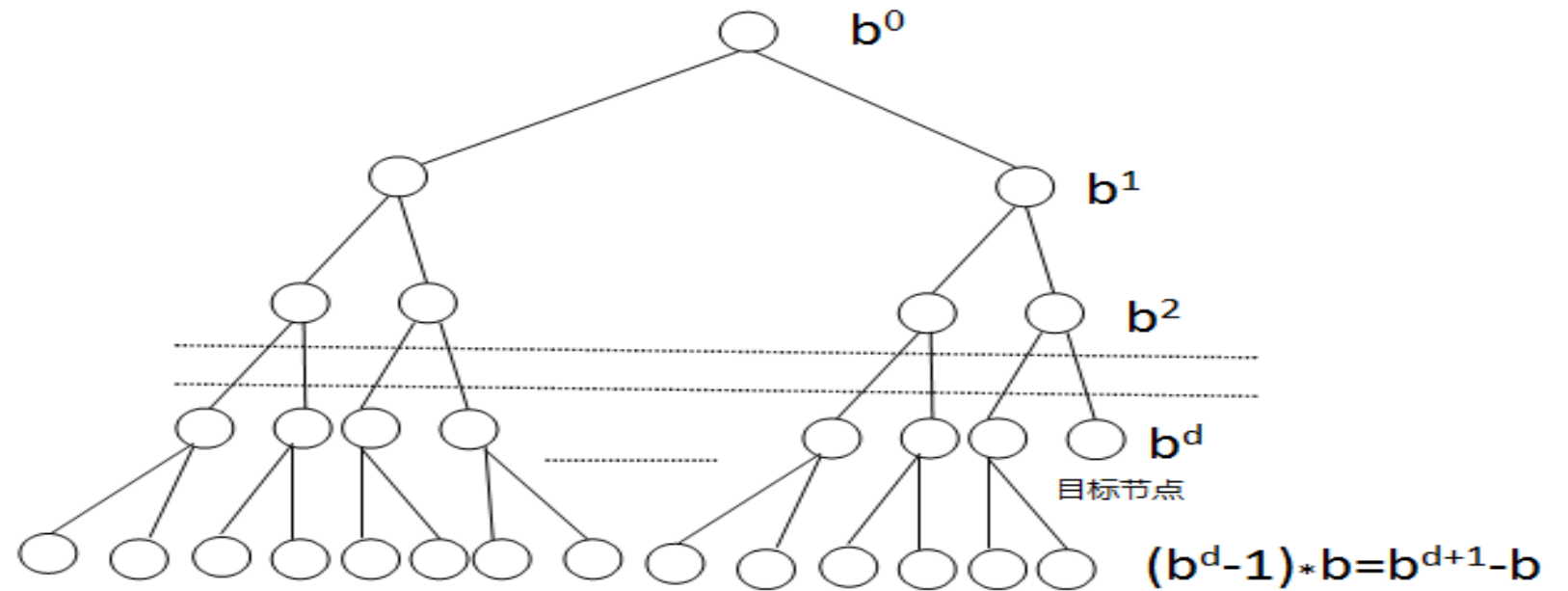
- 优先扩展最浅层的未扩展节点
- 实现方法:
  - **Fringe**表采用先进先出队列（**FIFO queue**），即新的后续节点总是放在队列的末尾

D
E
F
G



# 宽度优先搜索的性能指标

- 时间?  $1+b+b^2+b^3+\dots +b^d + (b^d-1)*b = \underline{O(b^{d+1})}$



# 宽度优先搜索的性能指标

- 完备性? Yes (只要  $b$  是有限的)
- 时间?  $1+b+b^2+b^3+\dots +b^d + b(b^d-1) = O(b^{d+1})$
- 空间?  $O(b^{d+1})$
- 最优性? Yes (只要单步代价是一样的)



Assume: branch factor  $b=10$ , 1 million nodes/s, 1 Kbytes/node:

DEPTH2	NODES	TIME	MEMORY
2	110	0.11 milliseconds	107 kilobyte
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabytes
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabytes
14	$10^{14}$	3.5 years	1 exabyte
16	$10^{16}$	350 years	10 exabyte

- 空间是一个比时间更严重的问题.
- 指数复杂性的搜索问题不能通过无信息搜索的方法求解。（除了最小的实例）

## 4.2一致代价搜索

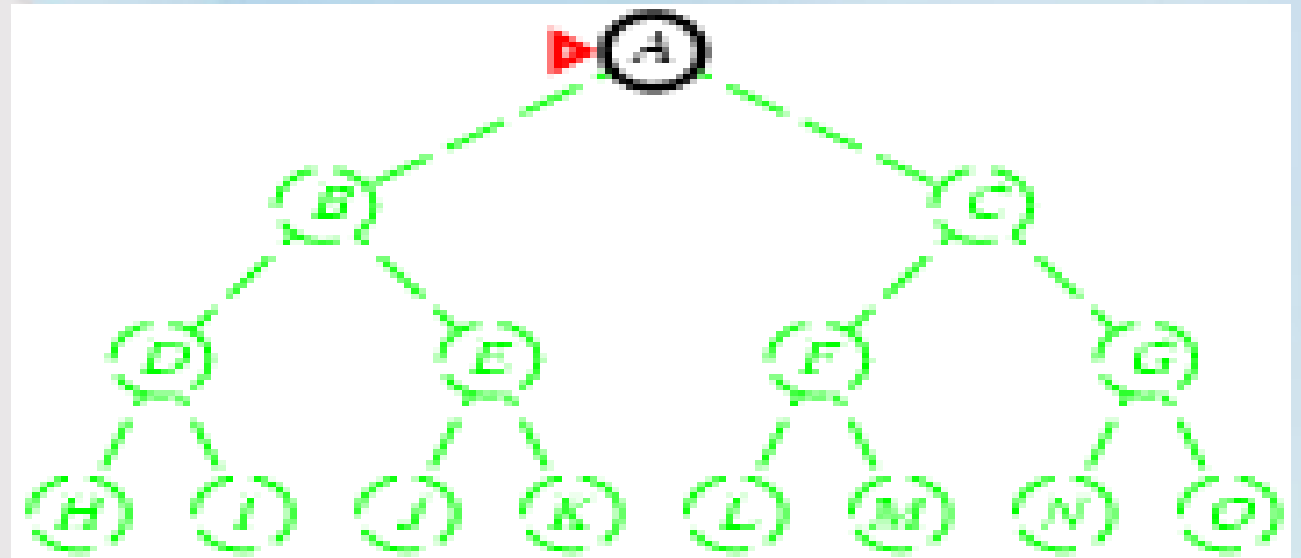
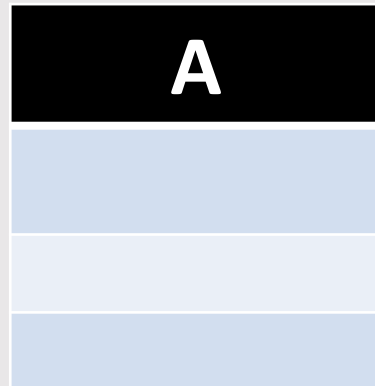
- 优先扩展具有**最小代价**的未扩展节点
- 实现: *fringe* 是根据路径代价排序的队列
- 在单步代价相等时与宽度优先搜索一样
- 完备性? Yes, 只要单步代价不是无穷小
- 时间? 代价小于最优解的节点个数,  $O(b^{\text{ceiling}(C^*/\epsilon)})$
- 空间? 代价小于最优解的节点个数,  $O(b^{\text{ceiling}(C^*/\epsilon)})$
- 最优性? Yes - 节点是根据代价排序扩展的



注:  $C^*$  最优解的代价  
 $\epsilon$  是至少每个动作的代价  
ceiling取上整

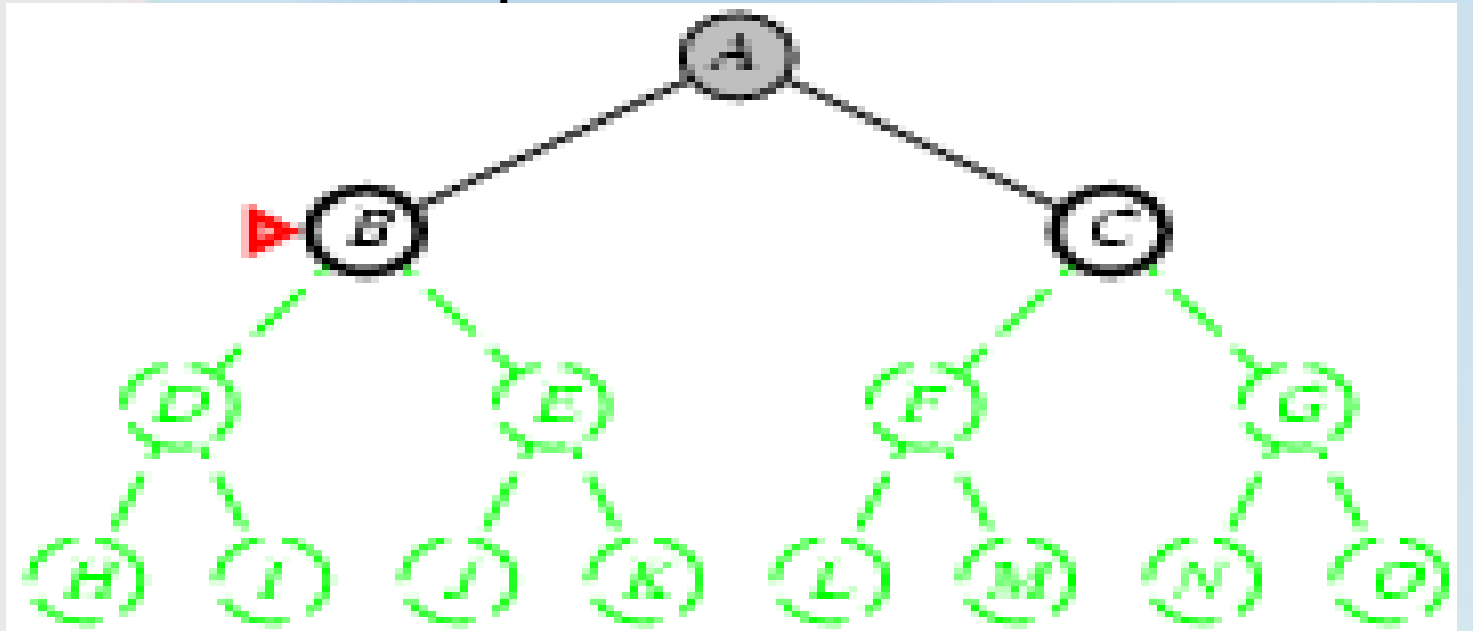
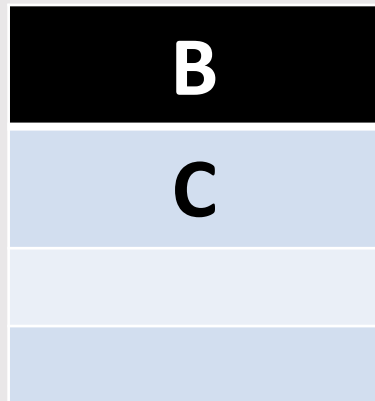
## 4.3 深度优先搜索

- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)



# 深度优先搜索

- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

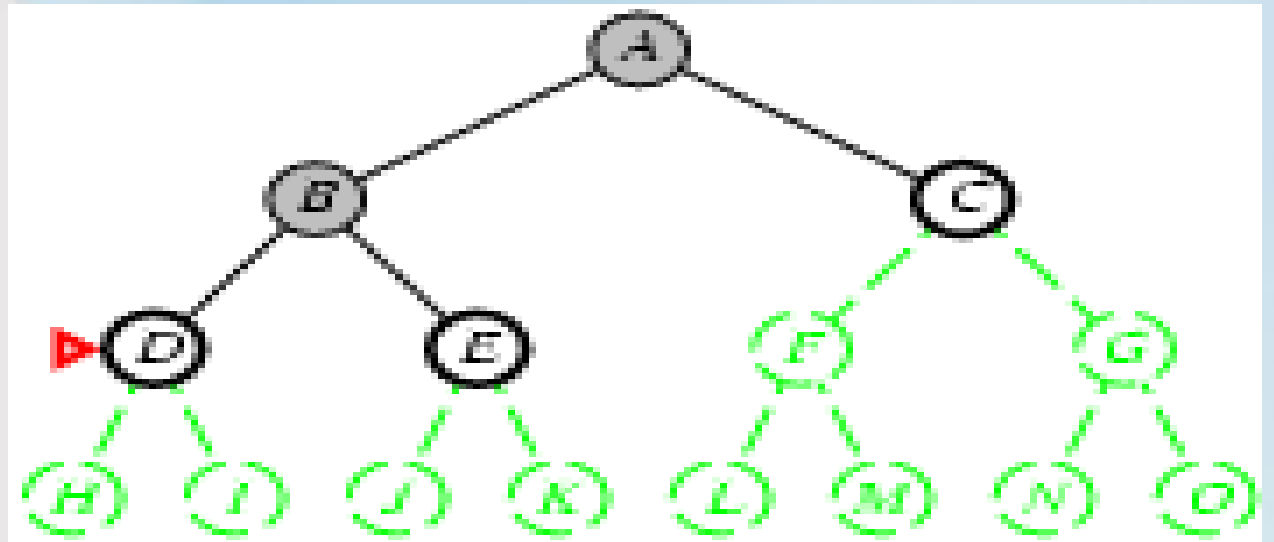




# 深度优先搜索

- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

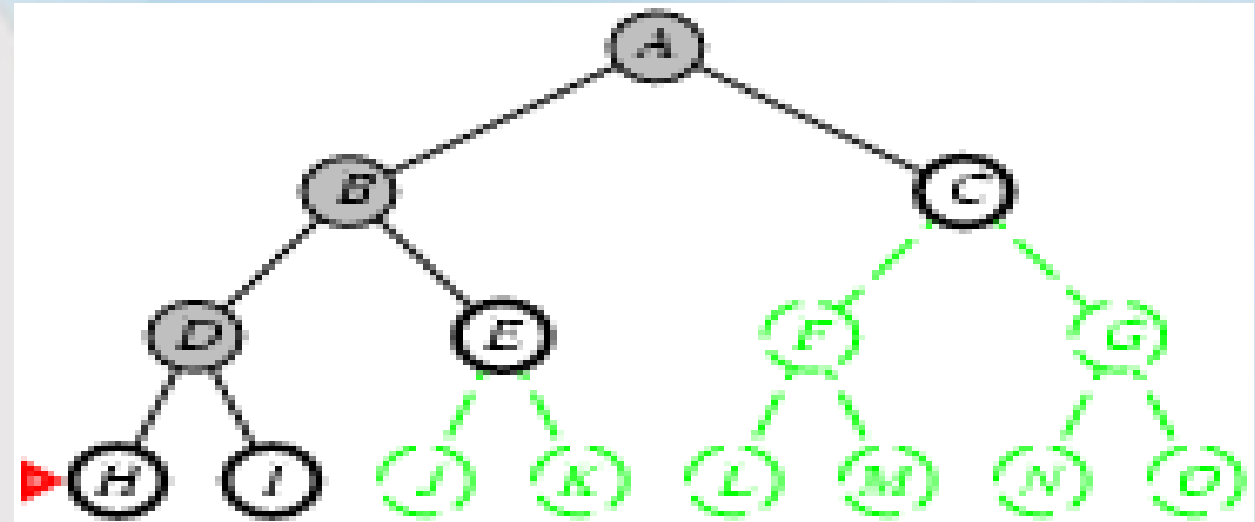
D
E
C



# 深度优先搜索

- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

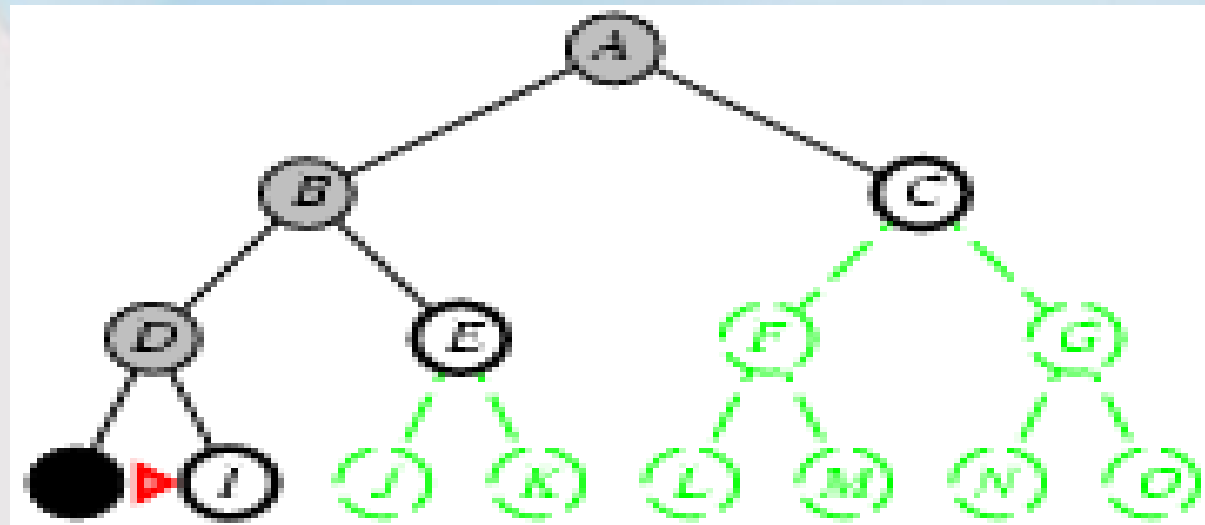
H
I
E
C



# 深度优先搜索

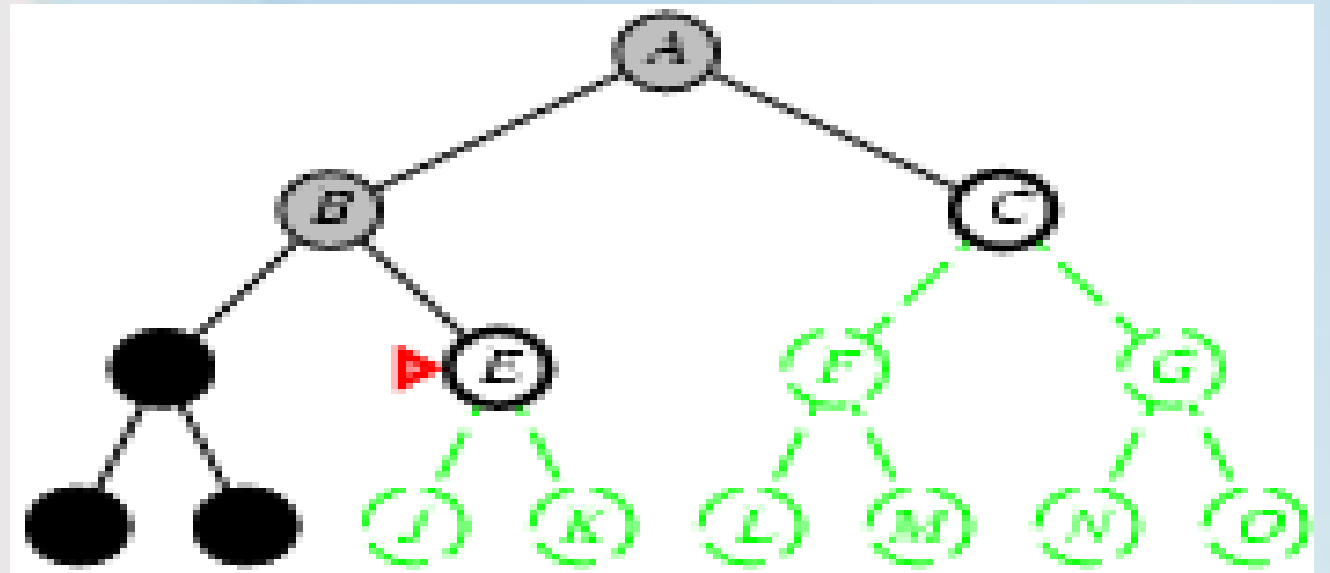
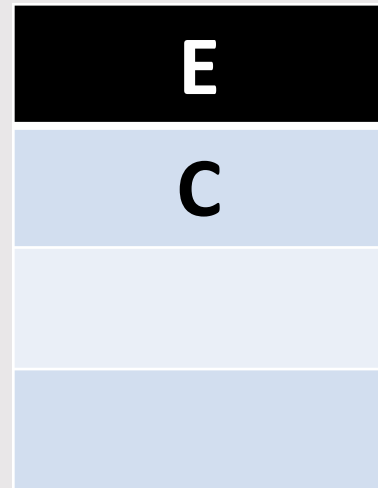
- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

I
E
C



# 深度优先搜索

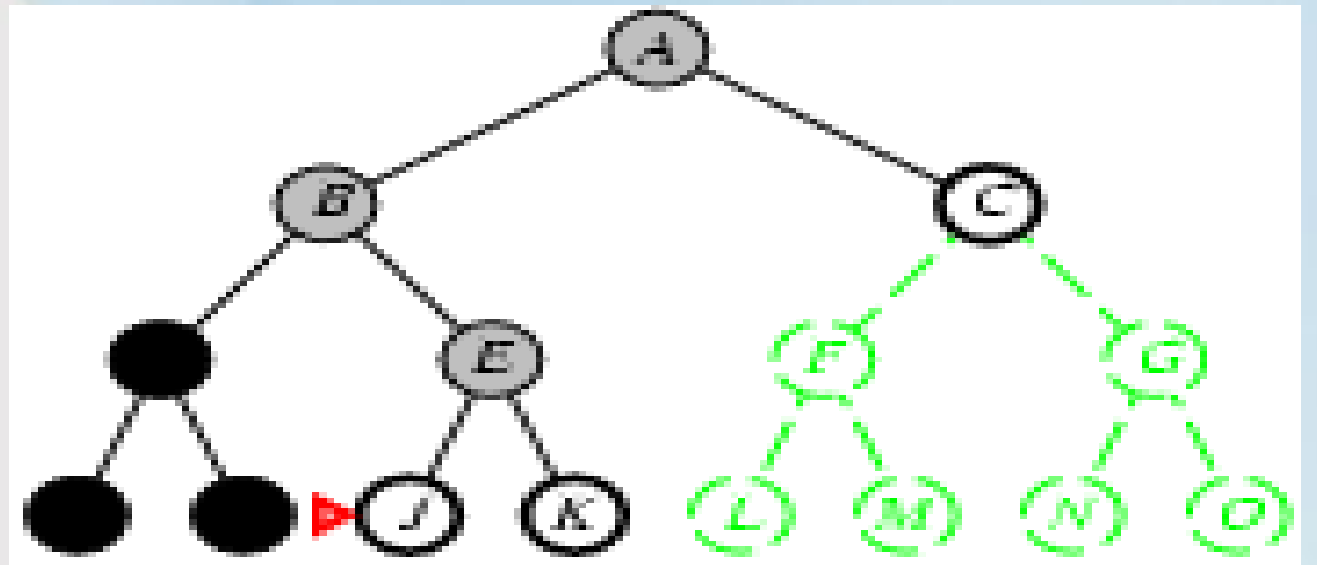
- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)



# 深度优先搜索

- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

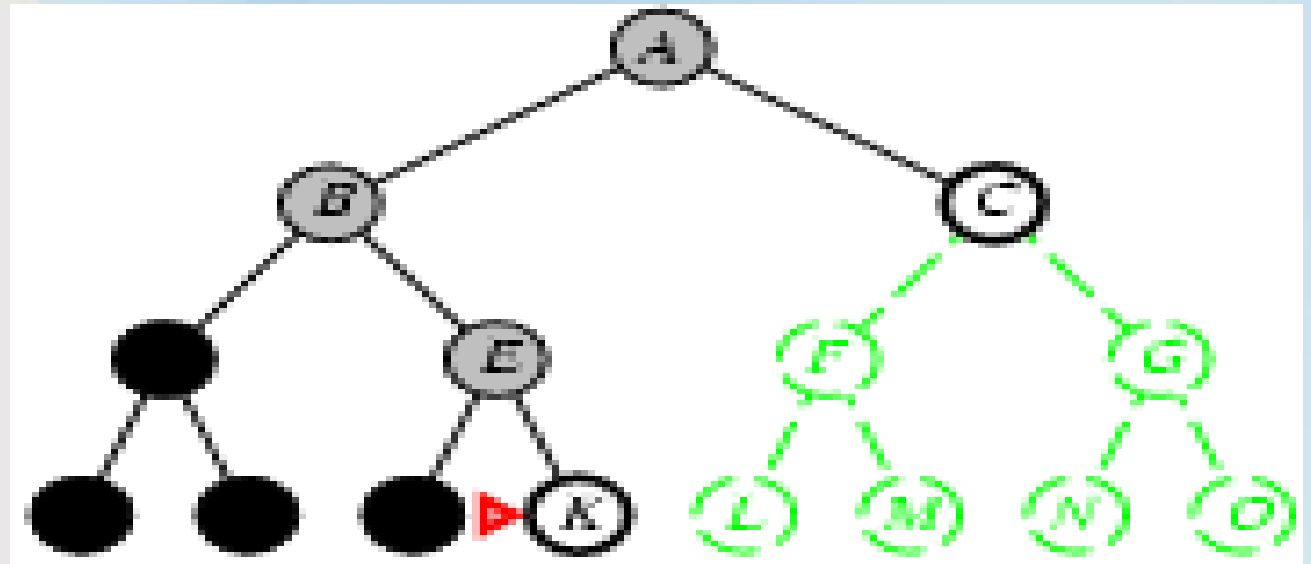
J
K
C



# 深度优先搜索

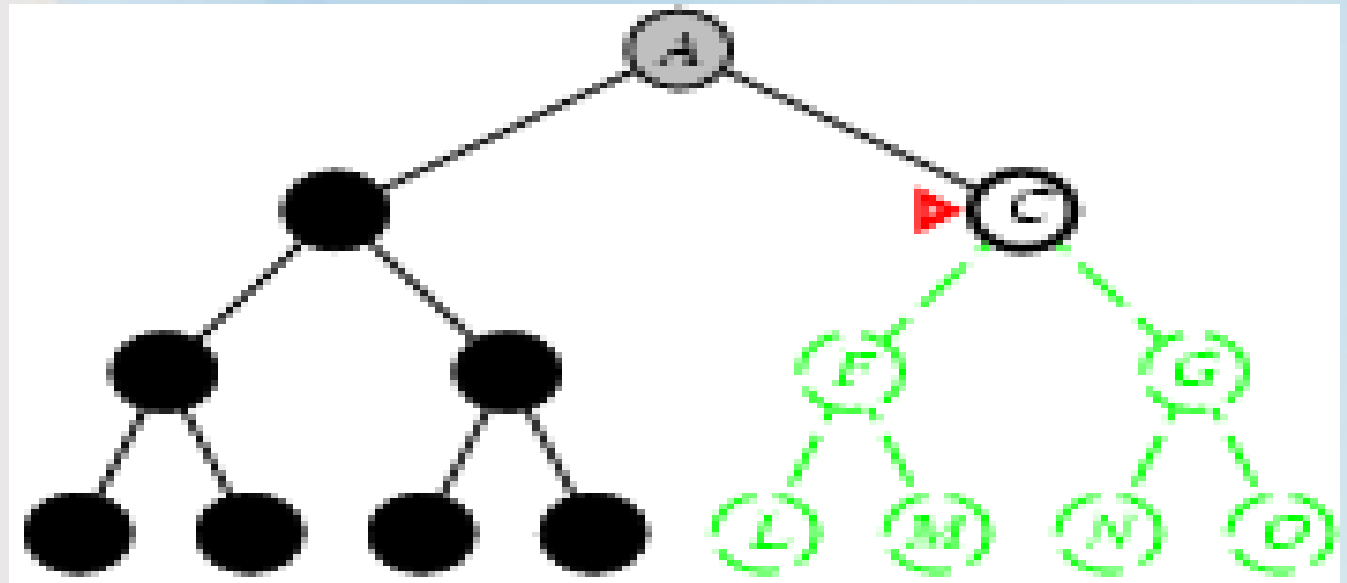
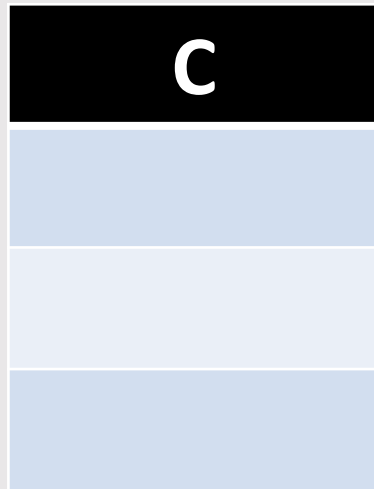
- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

K
C



# 深度优先搜索

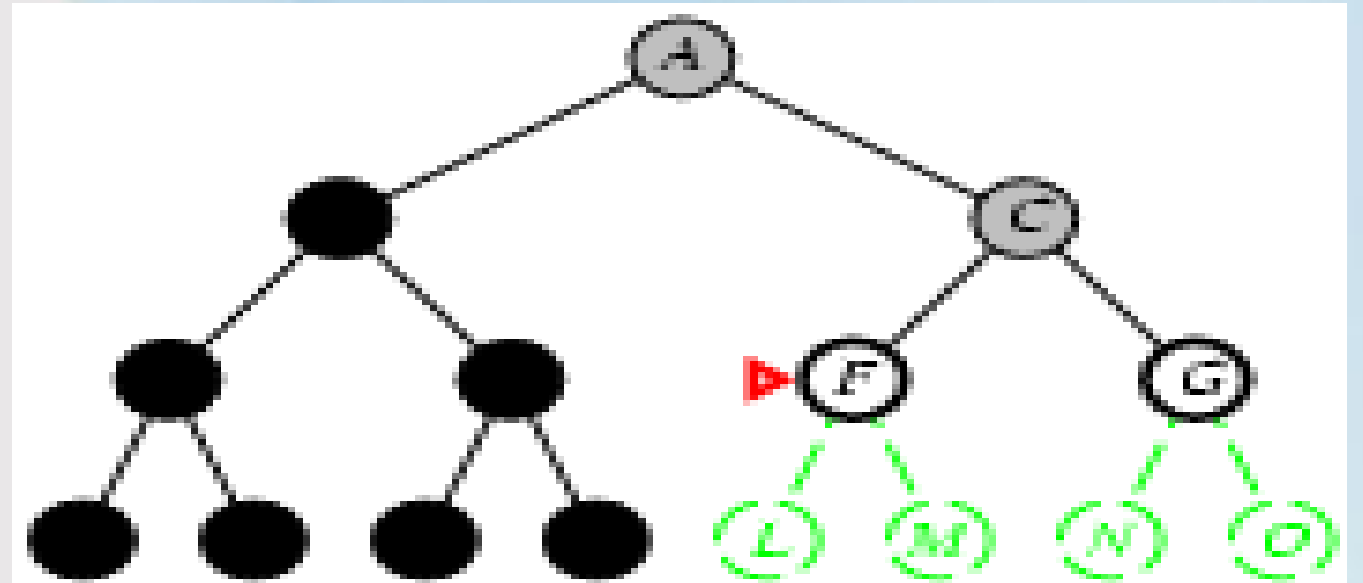
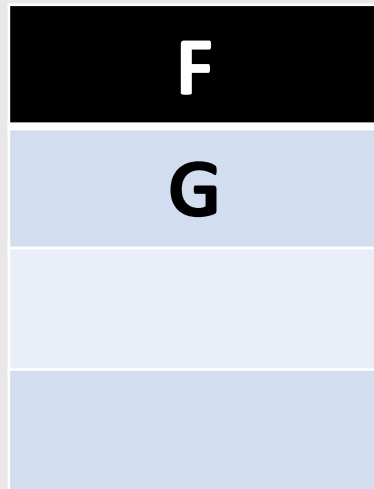
- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)





# 深度优先搜索

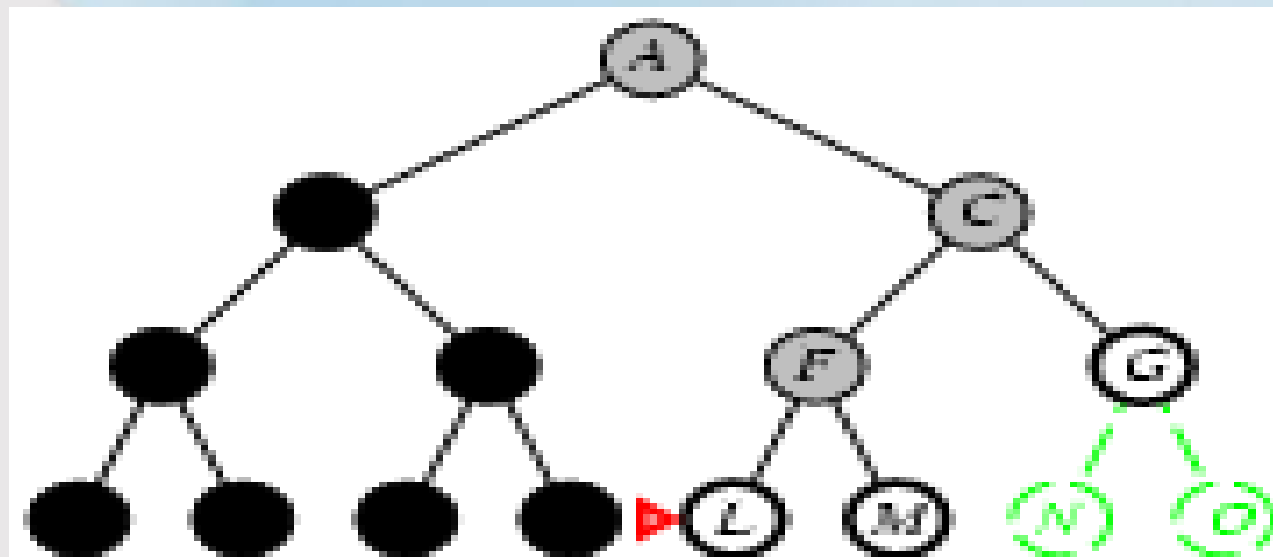
- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)



# 深度优先搜索

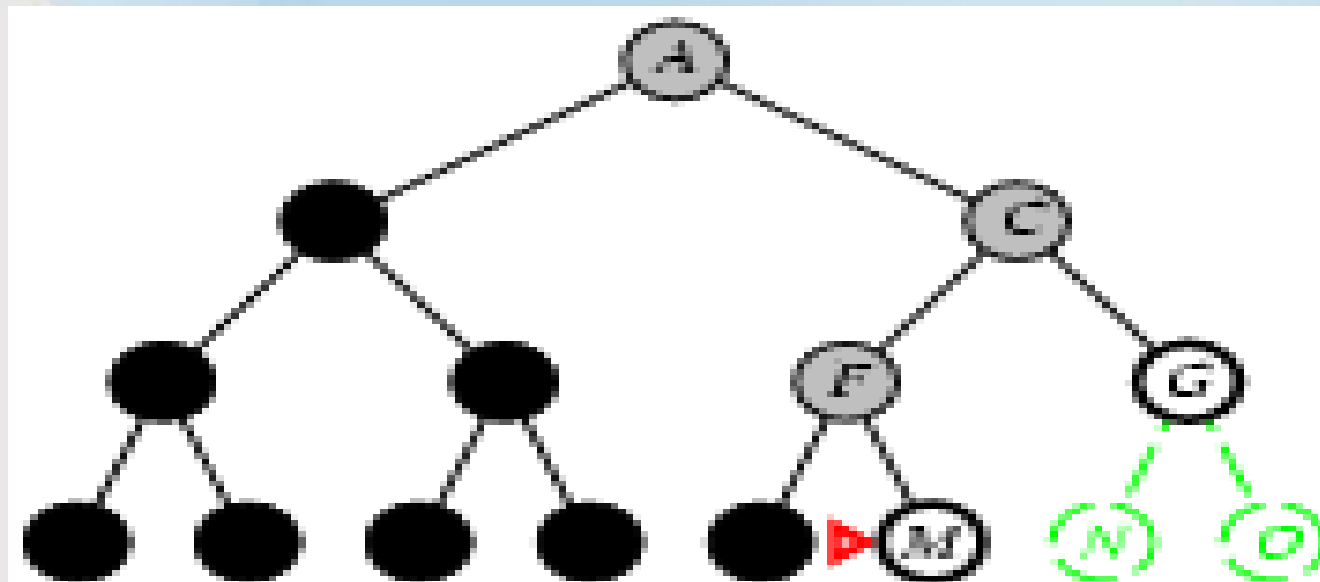
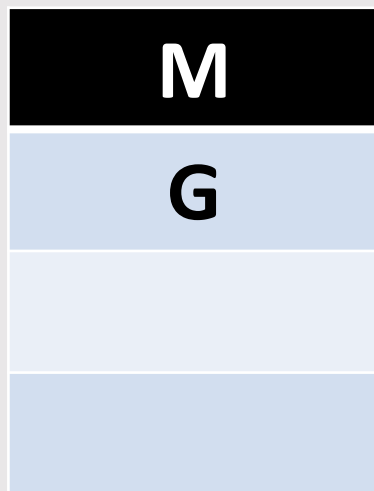
- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)

L
M
G



# 深度优先搜索

- 扩展最深层的未扩展节点
- 实现: *fringe* = 后进先出队列 (LIFO queue)



# 深度优先搜索的性能指标

- 完备性? No: 在无限状态空间中不能保证找到解
- 
- 时间?  $O(b^m)$
- 
- 空间?  $O(bm)$ , i.e., 线性空间!
- 
- 最优性? No



## 4.4 深度优先搜索改进

### 4.4.1 有深度限制的深度优先搜索

- 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
```

# 深度有限搜索的性质

- 完备性? No
- 
- 时间?  $O(b^l)$
- 
- 空间?  $O(bl)$ , i.e., 线性空间!
- 
- 最优性? No
- 



## 4.4.2 迭代深入搜索

**function** ITERATIVE-DEEPENING-SEARCH( *problem*) **returns** a solution, or failure

inputs: *problem*, a problem

**for** *depth*  $\leftarrow$  0 **to**  $\infty$  **do**

*result*  $\leftarrow$  DEPTH-LIMITED-SEARCH( *problem*, *depth*)

**if** *result*  $\neq$  cutoff **then return** *result*





# 迭代深入搜索 $l=0$

Limit = 0

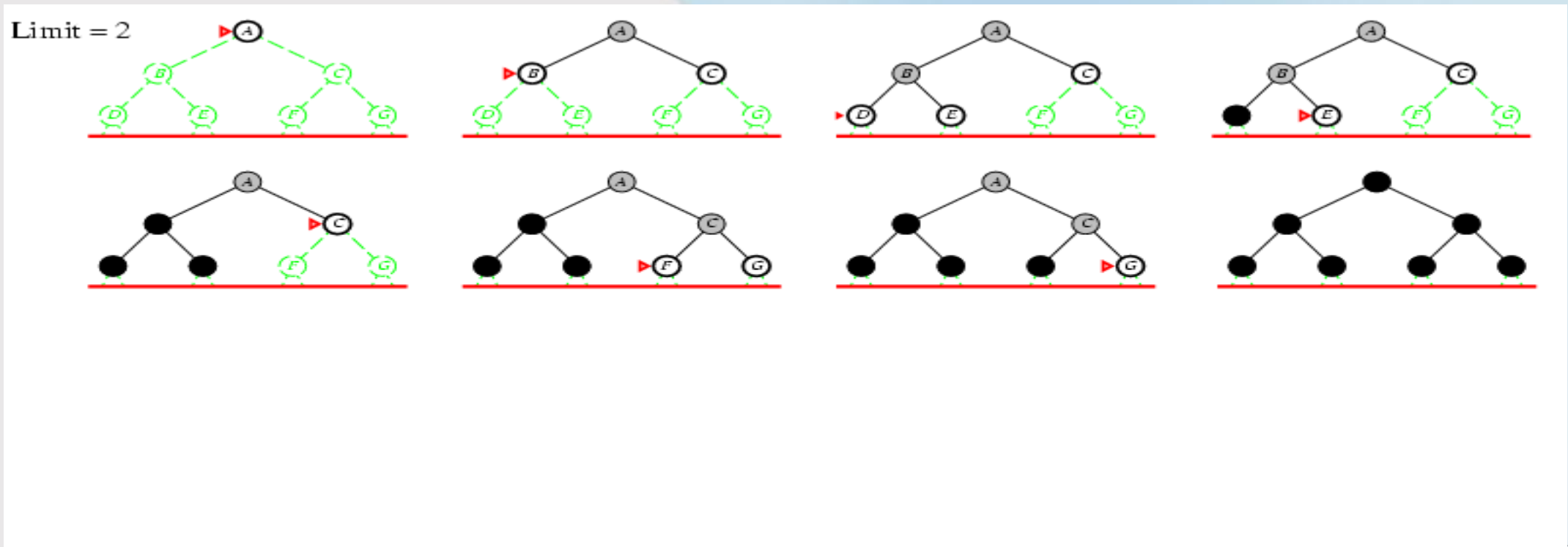


# 迭代深入搜索 $l=1$

Limit = 1

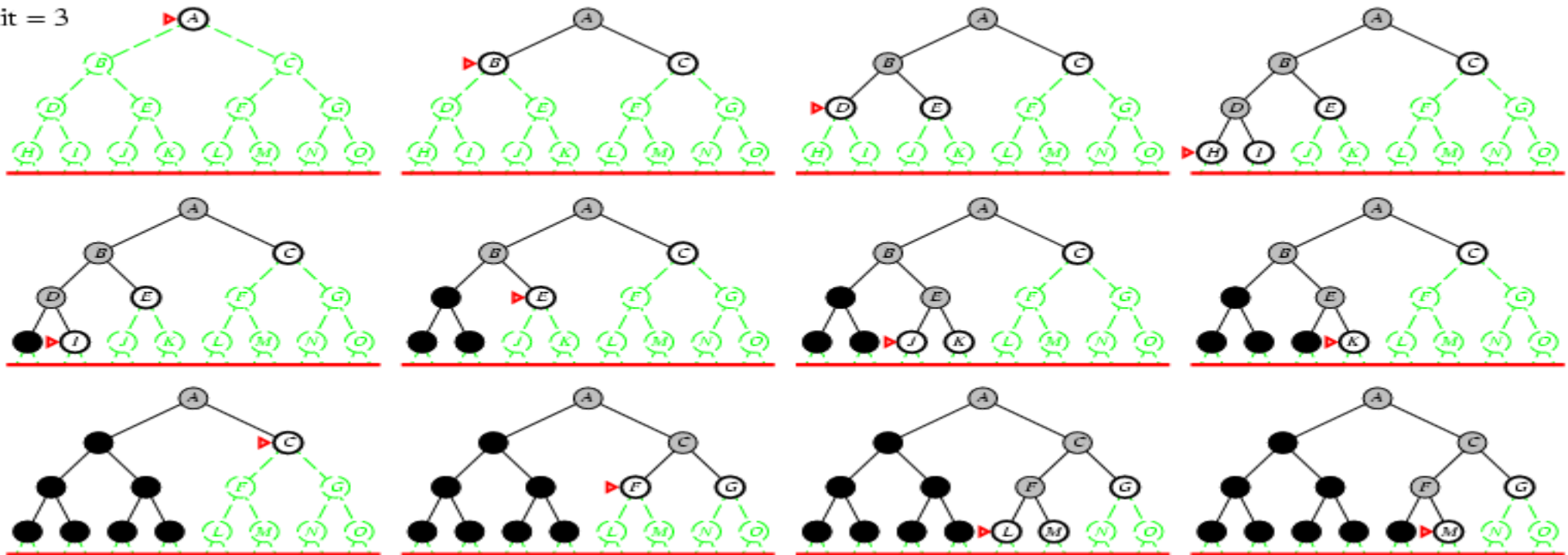


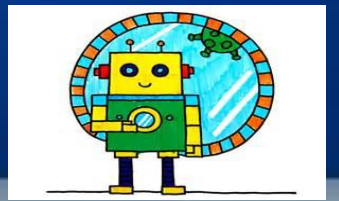
# 迭代深入搜索 $l=2$



# 迭代深入搜索 $l=3$

Limit = 3





# 迭代深入搜索性能分析

- 深度有限搜索（Deep limited search, DLS）搜索到d层时产生的节点数：

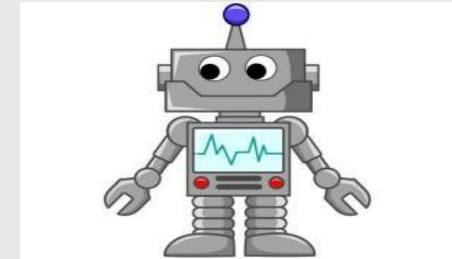
$$N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$

- 迭代深入搜索（iterative deepening search, IDS）搜索到d层时产生的节点数：

$$N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

注： 宽度优先搜索  $1+b+b^2+b^3+\dots +b^d + (b^d-1)*b = \underline{O(b^{d+1})}$

# 迭代深入搜索性能分析



- 对于分支数  $b = 10$ , 深度  $d = 5$  的问题

深度有限:  $N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$

$$N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$$

迭代深度有限:  $N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- 超出比率 =  $(123,456 - 111,111)/111,111 = 11\%$

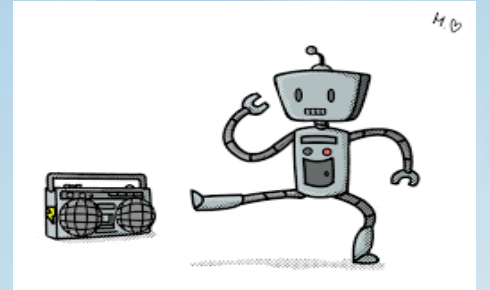
而宽度优先:  $1+b+b^2+b^3+\dots +b^d + (b^d-1)*b = \underline{O(b^{d+1})}$

$$N = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 + 10(100000-1) = 1,111,101$$

超出比达: 88.9%

# 迭代深入搜索的性质

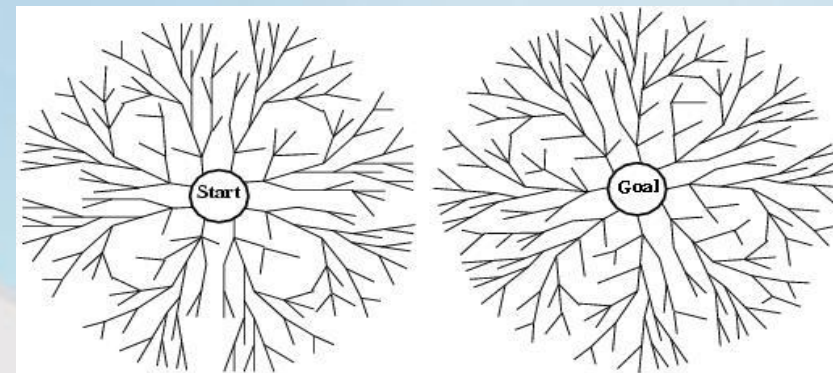
- 完备性? **Yes**
- 
- 时间?  $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- 
- 空间?  $O(bd)$
- 
- 最优性? **Yes**, 只要单步代价相等





## 4.5 双向搜索

- 从初始状态和目标状态同时出发：
  - 原理:  $b^{d/2} + b^{d/2} \leq b^d$
- 检查当前节点是否是其他fringe表的节点。
- 空间复杂度仍然是最大的问题。
- 如果双向都采用宽度优先则算法是完备的和最优性的。

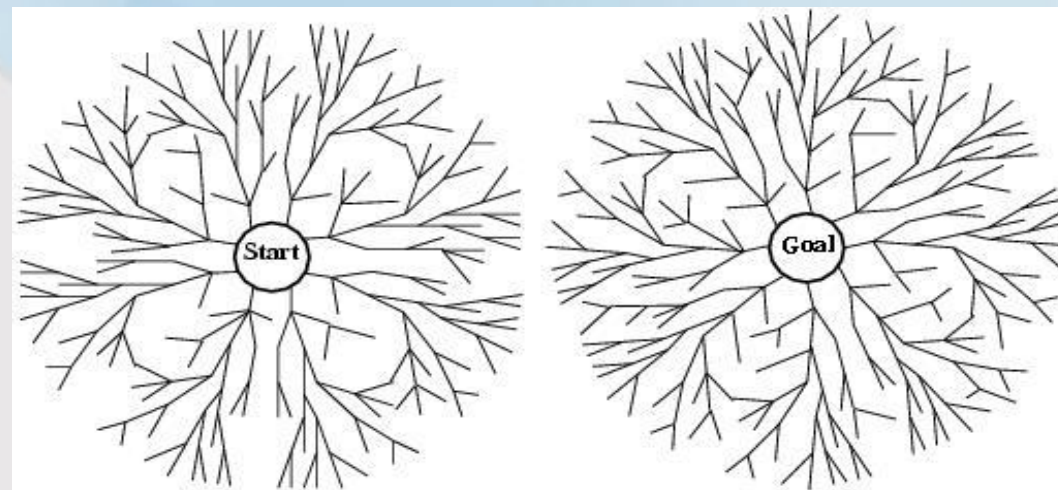


比较  $b=10$  以及  $d=6$  的问题扩展节点数:

$$N(\text{BiD}) = 2 * (10 + 100 + 1000) = 2220$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 + 1000000 = 1111110$$

# 如何反向搜索？



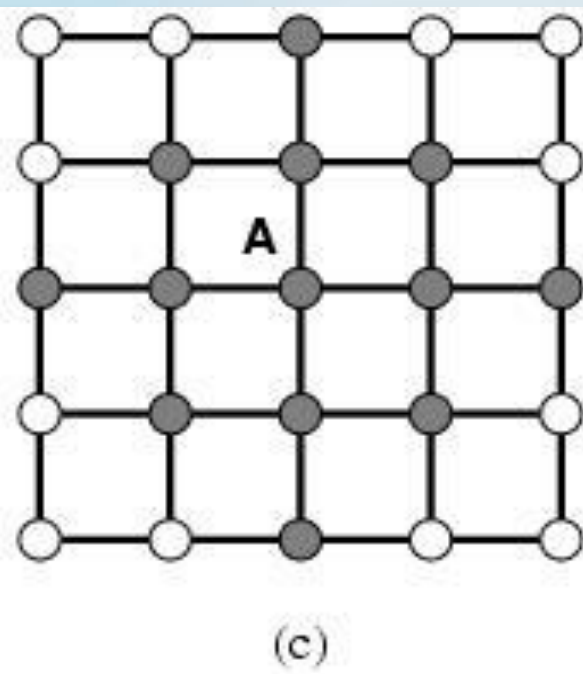
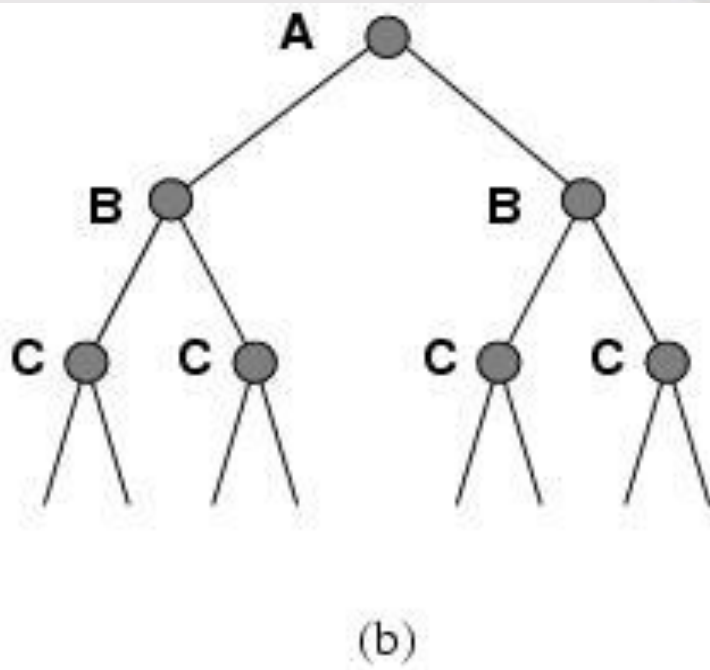
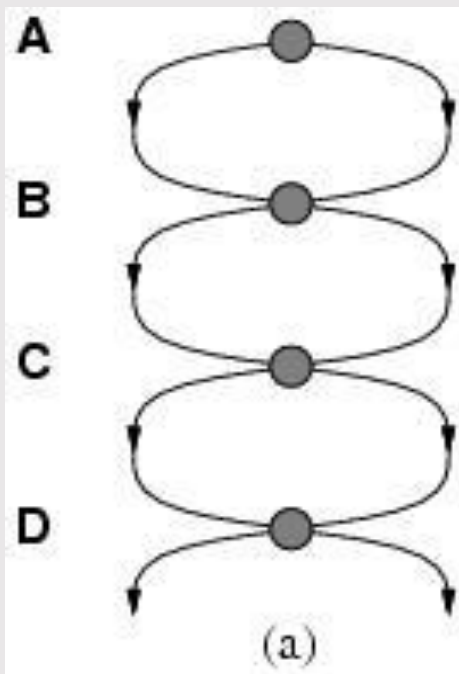
- 每个节点的前状态应是有效的可计算。
- 行动是容易可逆的。

## 4.6 无信息搜索算法性能评价小结

评价	宽度优先	代价一致	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
完备性	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
时间复杂度	$b^{d+1}$	$b^{C^*/\epsilon}$	$b^m$	$b^l$	$b^d$	$b^{d/2}$
空间复杂度	$b^{d+1}$	$b^{C^*/\epsilon}$	$bm$	$bl$	$bd$	$b^{d/2}$
最优性	YES*	YES*	NO	NO	YES	YES

# 重复状态

- 如果不能检测重复的状态，会导致把一个线性空间问题变成指数级的问题（不可解）。



# 实现： 图搜索算法

// *explored* 存储所有已扩展的节点

**function** GRAPH-SEARCH( *problem*, *fringe*) **return** a solution or failure

*explored*  $\leftarrow$  an empty set

*fringe*  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)

**loop do**

**if** EMPTY?(*fringe*) **then return** failure

*node*  $\leftarrow$  REMOVE-FIRST(*fringe*)

**if** GOAL-TEST[*problem*] applied to STATE[*node*] succeeds

**then return** SOLUTION(*node*)

**if** STATE[*node*] is not in *explored* **then**

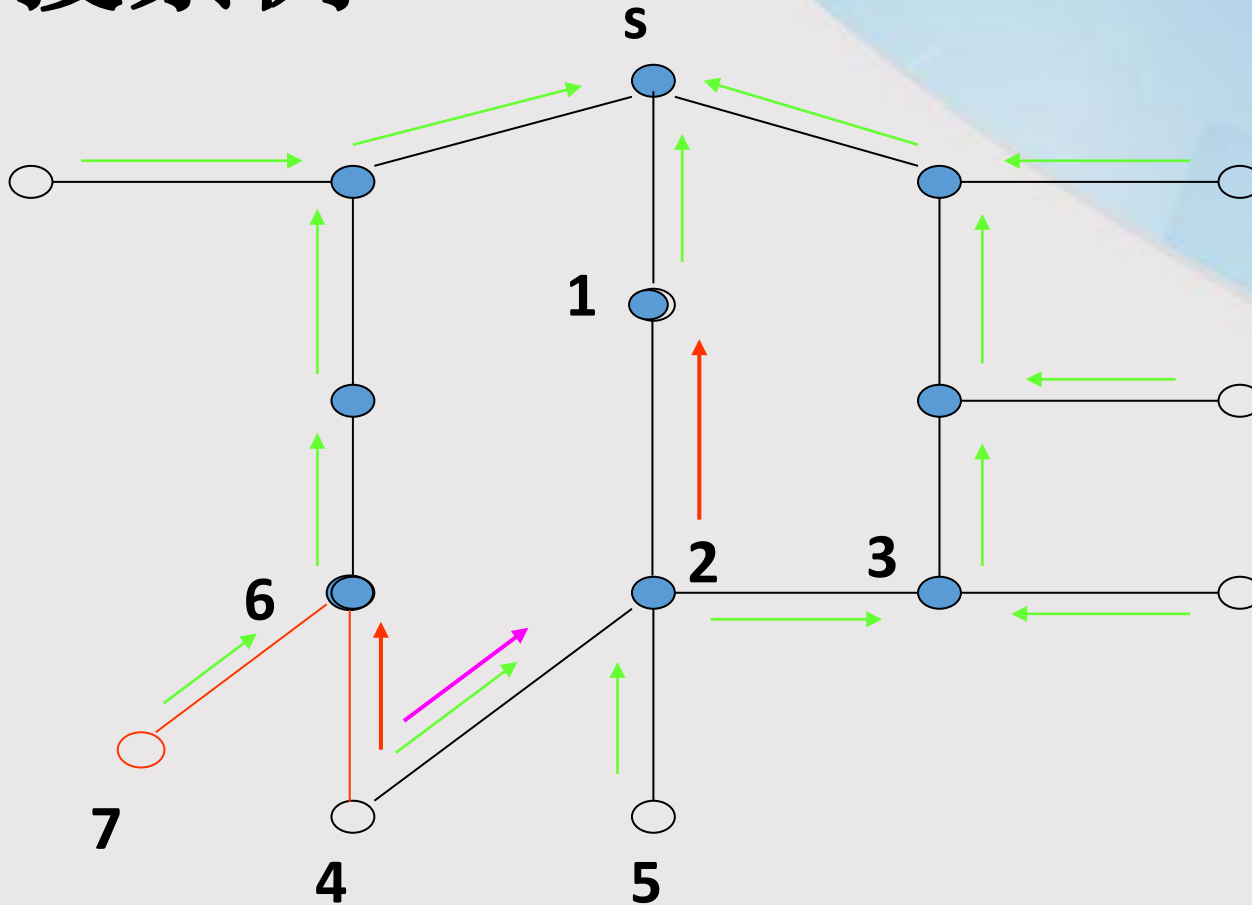
        add STATE[*node*] to *explored*

*fringe*  $\leftarrow$  INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

fringe(1,4,7,5,...)

explored (s,2,3, 6,...)

## 图搜索例



cost(4)=5 cost(2)=4

Expand 6,  $\Rightarrow$  7, 4

cost(4)=4

Modify 4 pointer, point to 6

Expand 1,  $\Rightarrow$  2

cost(2)=2

Modify 2 pointer, Point to 1

cost(4)=3

Modify 4 pointer, Point to 2



# 图搜索

- 完备性： 是的
- 最优性：
  - 图搜索截断了新路径，这可能导致局部最优解。
  - 当单步代价相同宽度搜索或代价一致的搜索时是最优的。
- 时间和空间复杂度：
  - 与状态空间的大小成比例的（可能远远小于 $O(b^d)$ ）
  - 深度和迭代深度搜索算法不再是线性空间。（因为，需要保存explored表）。



## 无信息搜索课堂思考题：

- 传教士和野人问题M-C问题 (**Missionaries & Cannibals Problem**)

- ✓ **已知：** 传教士人数 $M=3$ ，野人人数 $C=3$ ，一条船一次可以装载不超过2人 $K \leq 2$ 。
- ✓ **条件：** 任何情况下，如果传教士人数少于野人人数则有危险。
- ✓ **问题：** 传教士为了安全起见，应如何规划摆渡方案，使得任何时刻，河两岸以及船上的野人数目总是不超过传教士的数目。

即求解传教士和野人从左岸全部摆渡到右岸的过程中，任何时刻满足：

$M(\text{传教士数}) \geq C(\text{野人数})$  和  $M+C \leq k$  的摆渡方案。

- ✓ **要求：**
  - (1) 形式化该问题，并计算状态空间大小；
  - (2) 应用无信息搜索算法求解；（*考虑重复状态？*）
  - (3) 这个问题状态空间很简单，你认为是什么导致人们求解它和困难？

- 问题形式化:

用一个三元组( $m, c, b$ )来表示河岸上的状态, 其中 $m$ 、 $c$ 分别代表某一岸上传教士与野人的数目,  $b=1$ 表示船在这一岸,  $b=0$ 则表示船不在。

- ✓ 条件是: 两岸上 $M \geq C$ , 船上 $M+C \leq 2$ 。

说明: 由于传教士与野人的总数目是一常数, 所以只要表示出河的某一岸上的情况就可以了, 为方便起见, 我们选择传教士与野人开始所在的岸为所要表示的岸, 并称其为左岸, 另一岸称为右岸。显然仅用描述左岸的三元组就足以表示出整个情况了。

- ✓ 综上, 我们的状态空间可表示为:  $(ML, CL, BL)$ , 其中 $0 \leq ML, CL \leq N$ ,  $BL \in \{0, 1\}$ 。

状态空间的总状态数为 $(N+1) \times (N+1) \times 2$ ,

- ✓ 问题的初始状态是 $(N, N, 1)$ , 目标状态是 $(0, 0, 0)$ 。

## 转换模型

- 该问题主要有两种操作：从**左岸划向（条件）**右岸和从**右岸划向**左岸，以及每次摆渡的传教士和野人**个数变化（行动）**。

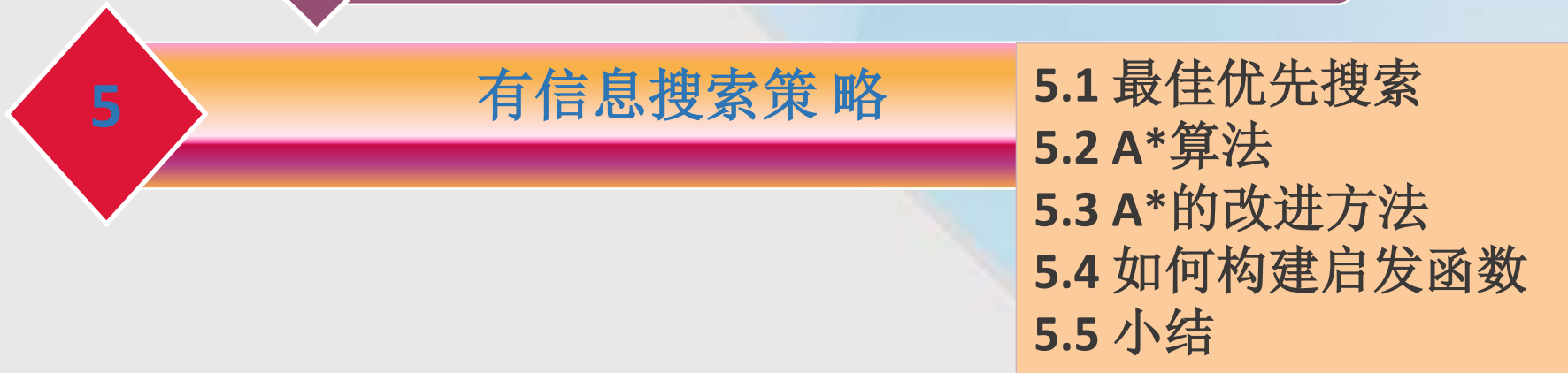
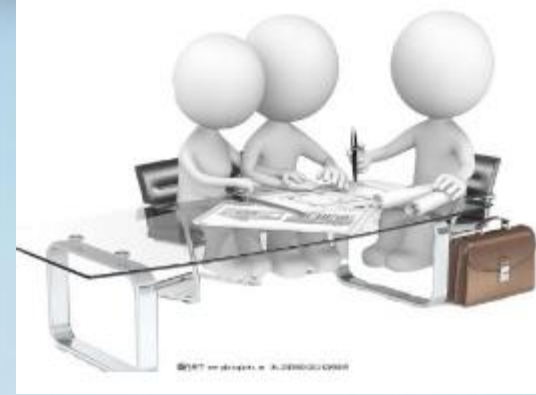
我们可以使用一个2元组（BM，BC）来表示每次摆渡的传教士和野人个数，我们用i代表每次过河的总人数， $i = 1 \sim k$ ，则每次有BM个传教士和 $BC = i - BM$ 个野人过河，其中 $BM = 0 \sim i$ ，而且当 $BM \neq 0$ 时需要满足 $BM \geq BC$ 。则

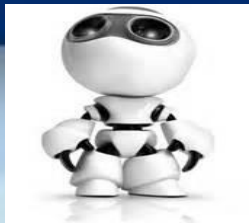
- ✓从左到右的操作为：（ML-BM，CL-BC，B = 1）
- ✓从右到左的操作为：（ML+BM，CL+BC，B = 0）

因此，当 $N=3$ ， $K=2$ 时，满足条件的（BM，BC）有：

（0,1）、（0,2）、（0,3）、（1,0）、（1,1）、（2,0）、  
（2,1）、（2,2）、（3,0）、（3,1）、（3,2）、（3,3）。

由于从左到右与从右到左是对称的，所以此时一共有24种操作。





# 回顾:树搜索

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

- 搜索策略就是节点扩展顺序的**选择**

# 有信息搜索

- **无信息搜索** 又名盲目搜索:

- 在搜索时, 只有问题定义信息可用。
- 盲目搜索策略仅利用了问题定义中的信息。

- **有信息搜索:**

- 在搜索时, 当有**策略**可以确定一个非目标状态比另一种更好的搜索, 称为有信息的搜索。



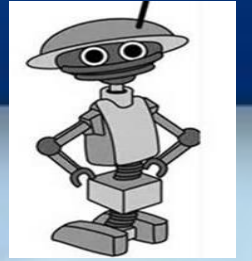


## 5.1 最佳优先搜索

- **思想:** 使用一个评估函数  $f(n)$  给每个节点估计他们的希望值。 优先扩展最有希望的未扩展节点。
- **实现:** **fringe**表中节点根据评估值从大到小排序
- **最佳优先**搜索策略有:
  - 贪婪最佳优先搜索
  - $A^*$  搜索







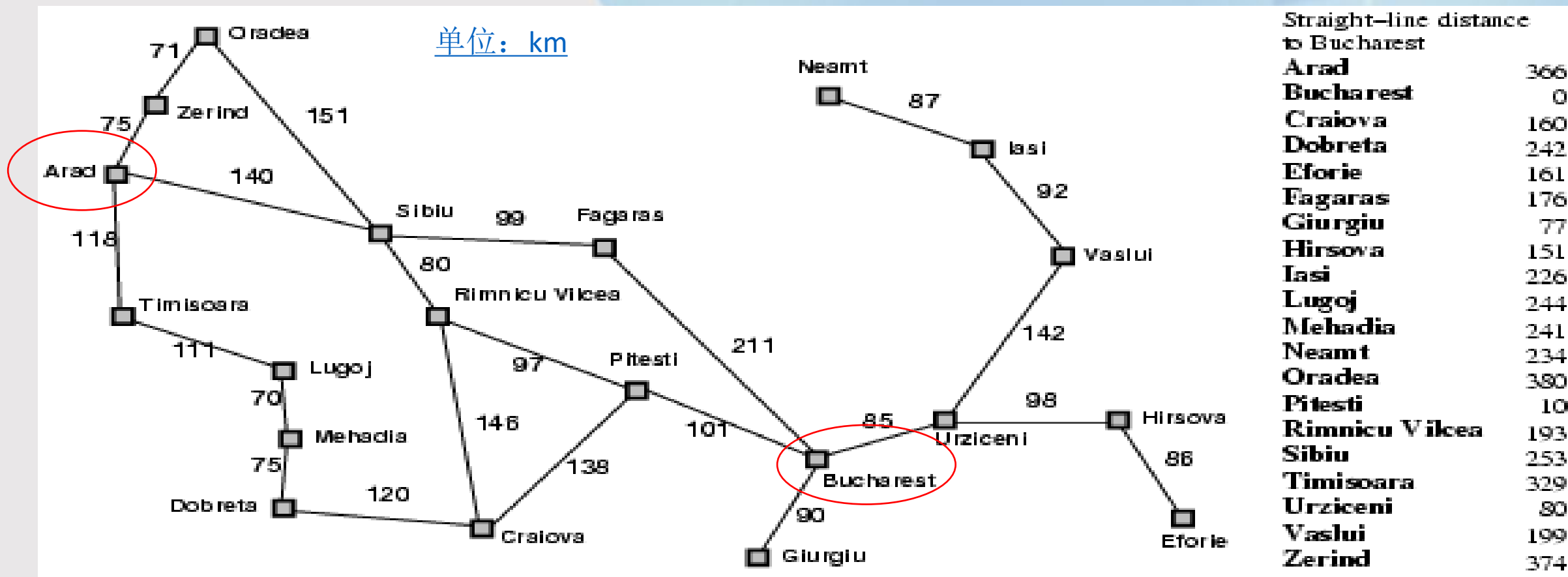
# 贪婪最佳优先搜索

- 评估函数:  $f(n) = h(n)$  (**h**euristic, 启发函数)  
= **估计**从节点n到目标的代价

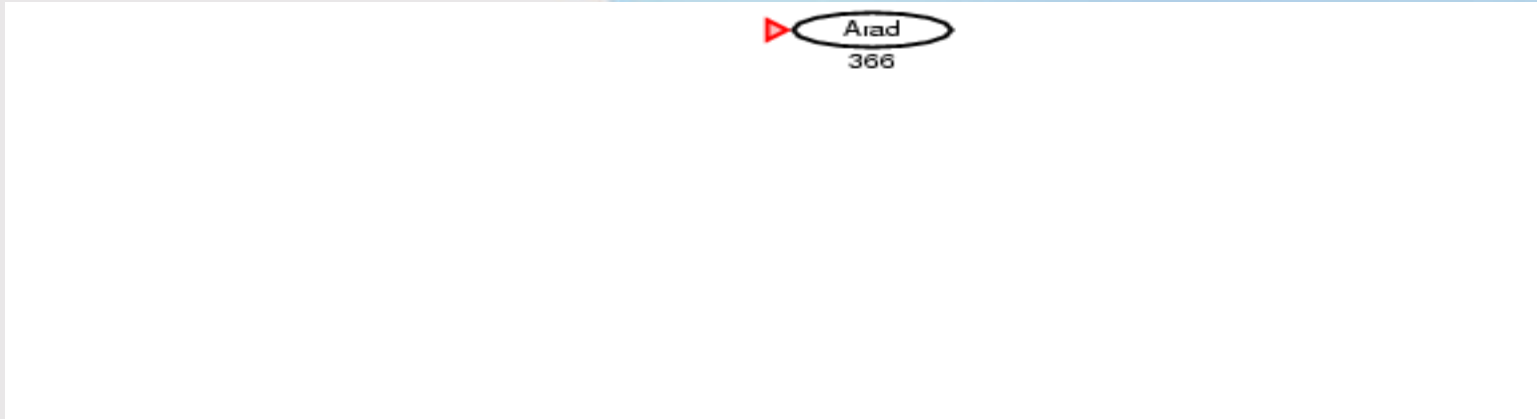
例如:  $h_{SLD}(n)$  = 从节点 n 到 Bucharest 的  
直线距离

- **贪婪最佳优先搜索** 优先扩展看上去更**接近目标**的节点（启发式函数评估出来的）。

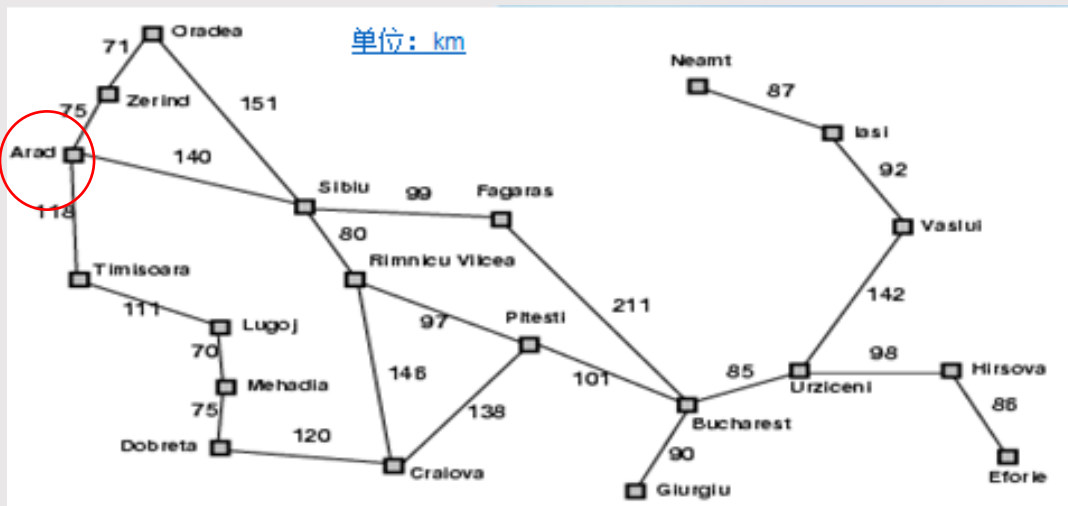
# 例：罗马尼亚问题



# 贪婪最佳优先搜索示例



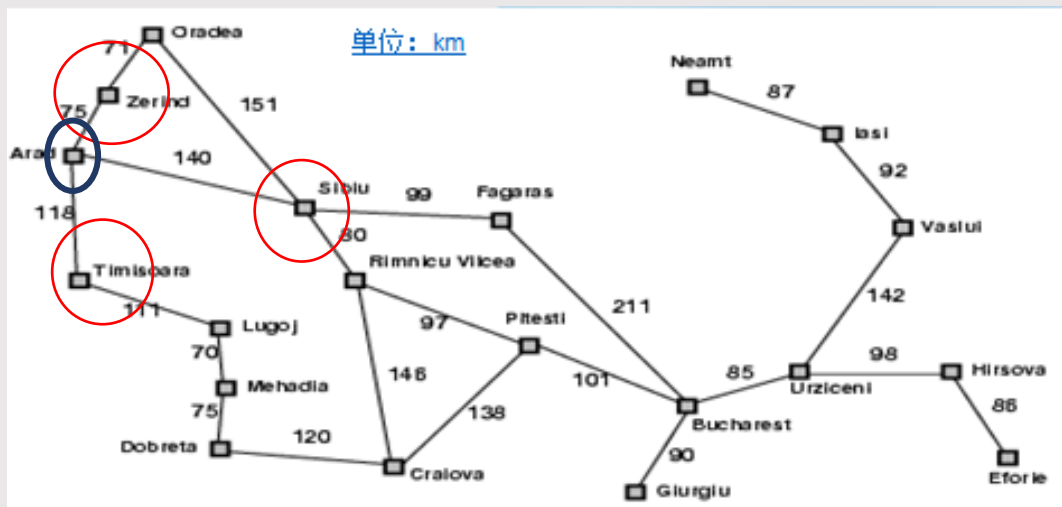
Straight-line distance to Bucharest	
Arad	366
<b>Bucharest</b>	<b>0</b>
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



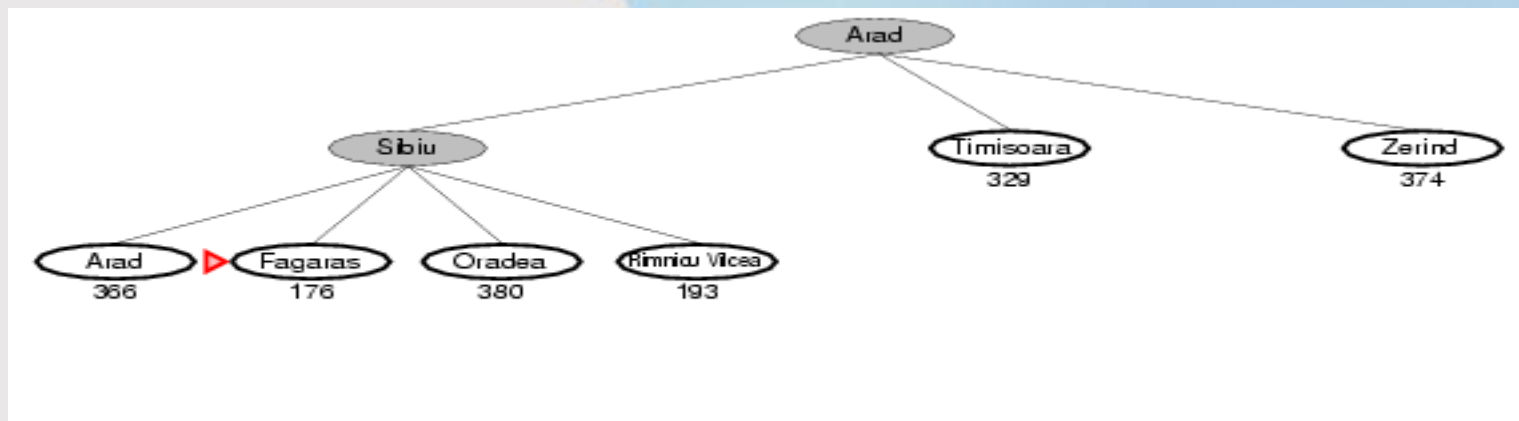
# 贪婪最佳优先搜索示例



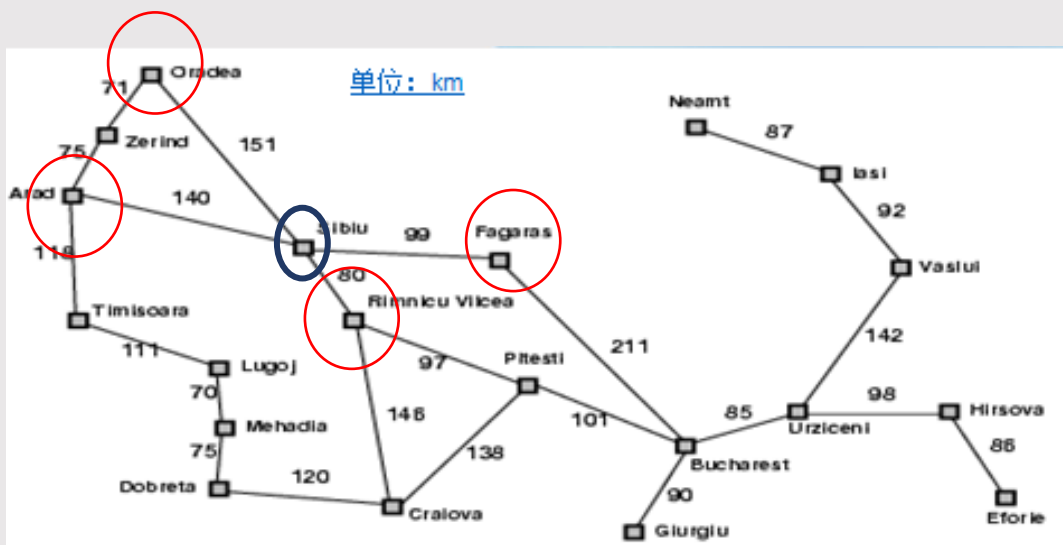
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



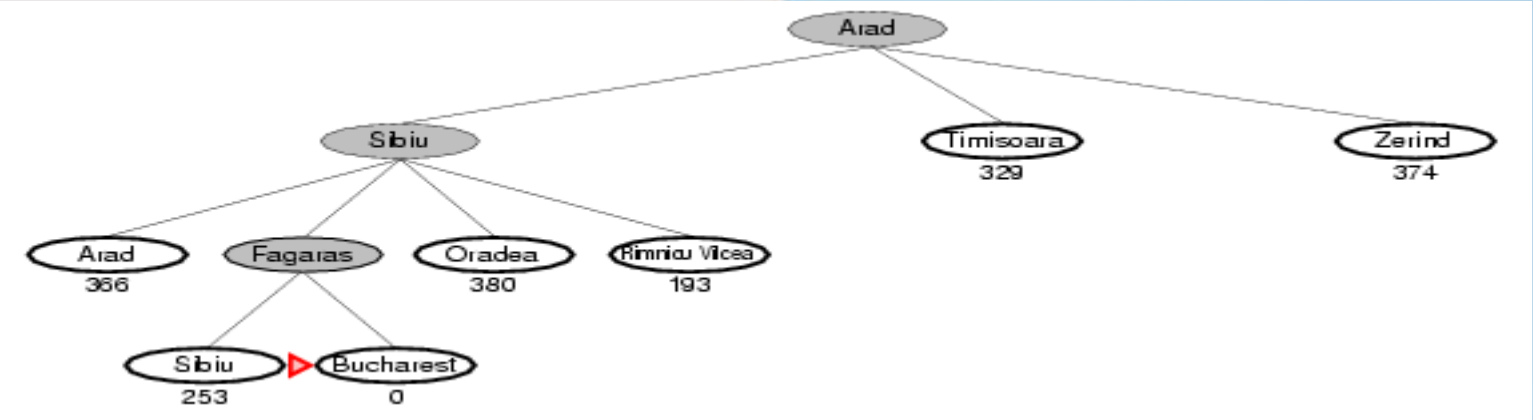
# 贪婪最佳优先搜索示例



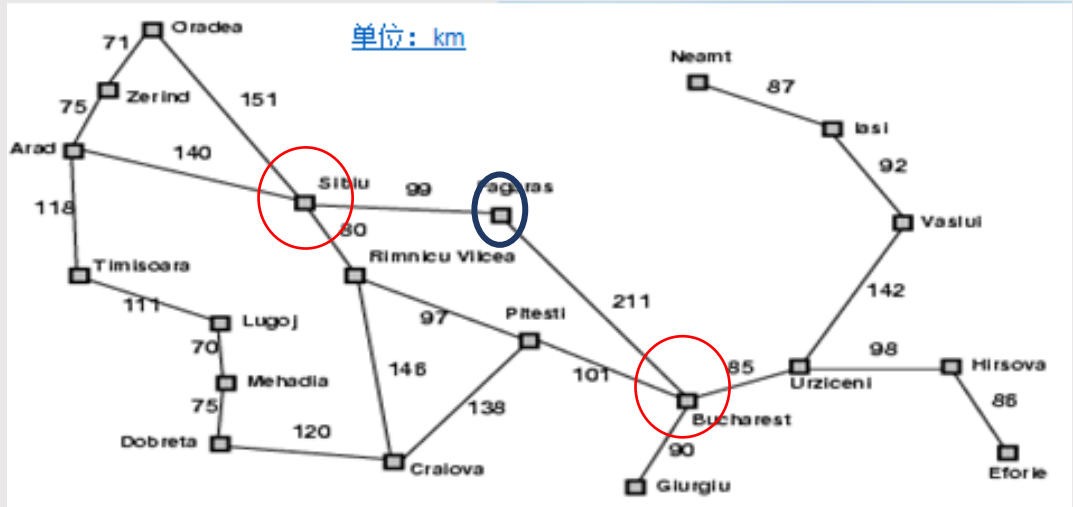
Straight-line distance to Bucharest	
Arad	366
<b>Bucharest</b>	0
Craiova	160
Dobreta	242
Eforie	161
<b>Fagaras</b>	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
<b>Oradea</b>	380
Pitesti	101
<b>Rimnicu Vilcea</b>	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



# 贪婪最佳优先搜索示例



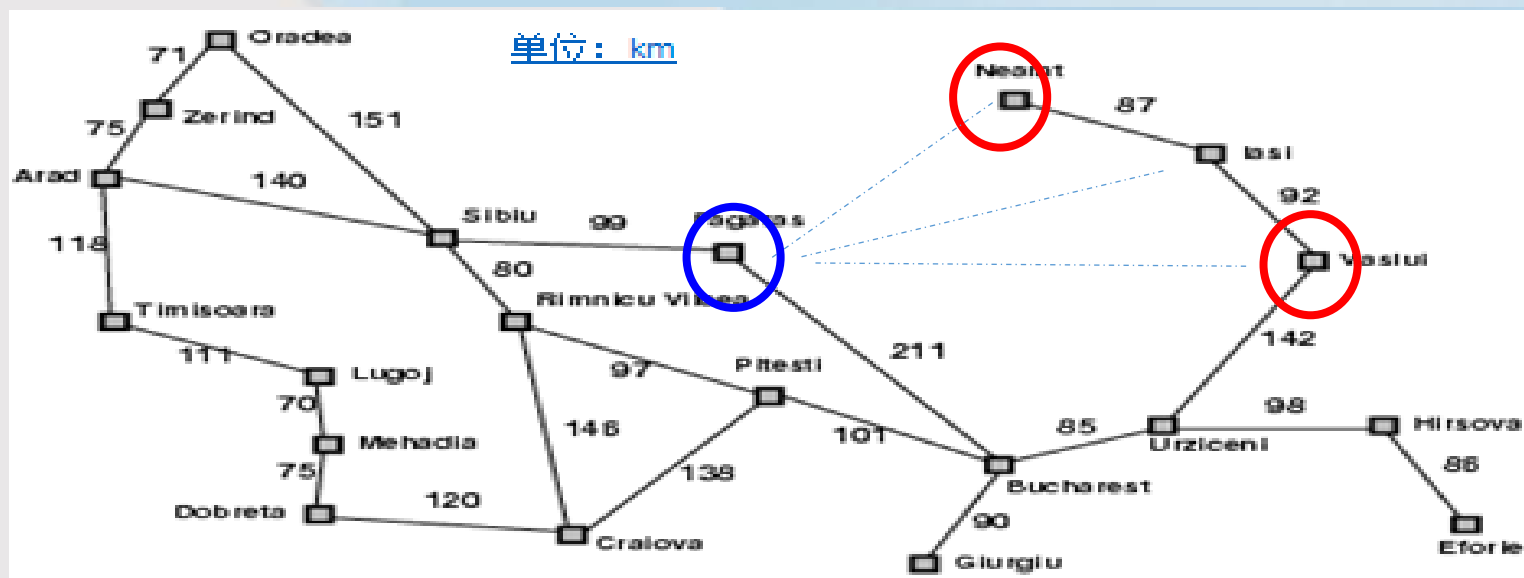
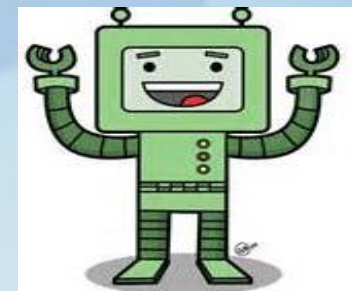
Straight-line distance to Bucharest	
Arad	366
<b>Bucharest</b>	<b>0</b>
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
<b>Sibiu</b>	<b>253</b>
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



路径代价:  $140+99+211=450$

# 贪婪最佳优先搜索的属性

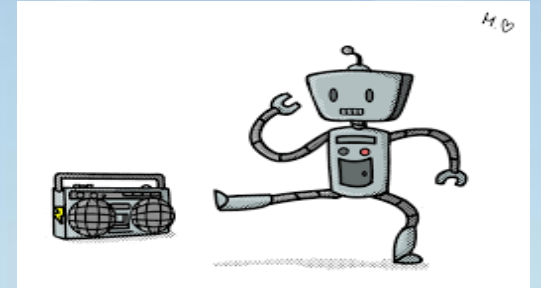
- 完备性? No – 可能陷于死循环当中,  
比如, IASI → Neamt → Iasi → Neamt →
- 时间?  $O(b^m)$ , 但一个好的启发式函数能带来巨大改善
- 空间?  $O(b^m)$
- 最优性? No






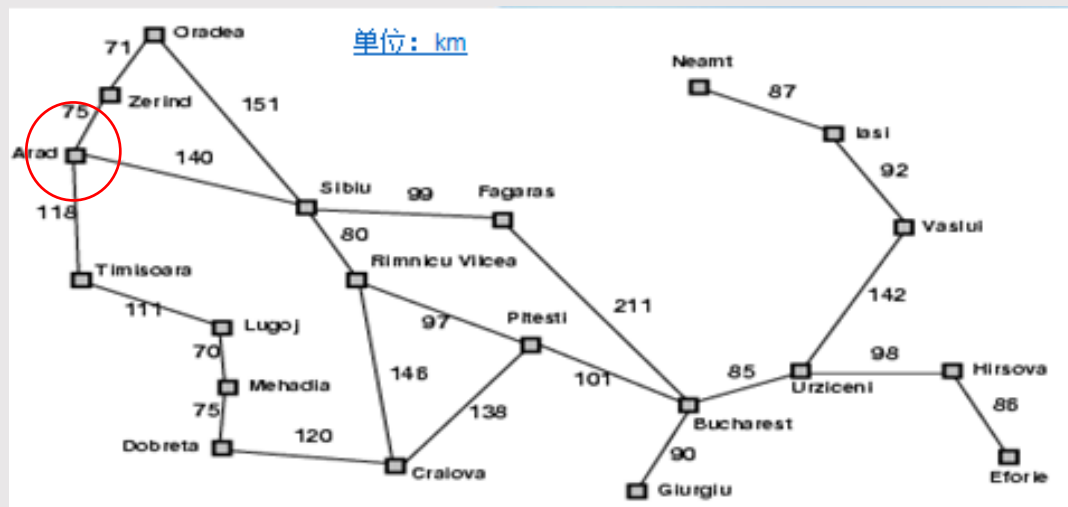
## 5.2 A\* 搜索

- **思想:** 避免扩展代价已经很高的节点。
- 评估函数  $f(n) = g(n) + h(n)$ 
  - $g(n)$  = 到达节点 $n$ 已经发生的实际代价
  - $h(n)$  = 从节点 $n$ 到目标的代价估计值
  - $f(n)$  = 评估函数, 估计从初始节点出发, 经过节点 $n$ , 到目标的路径代价的估计



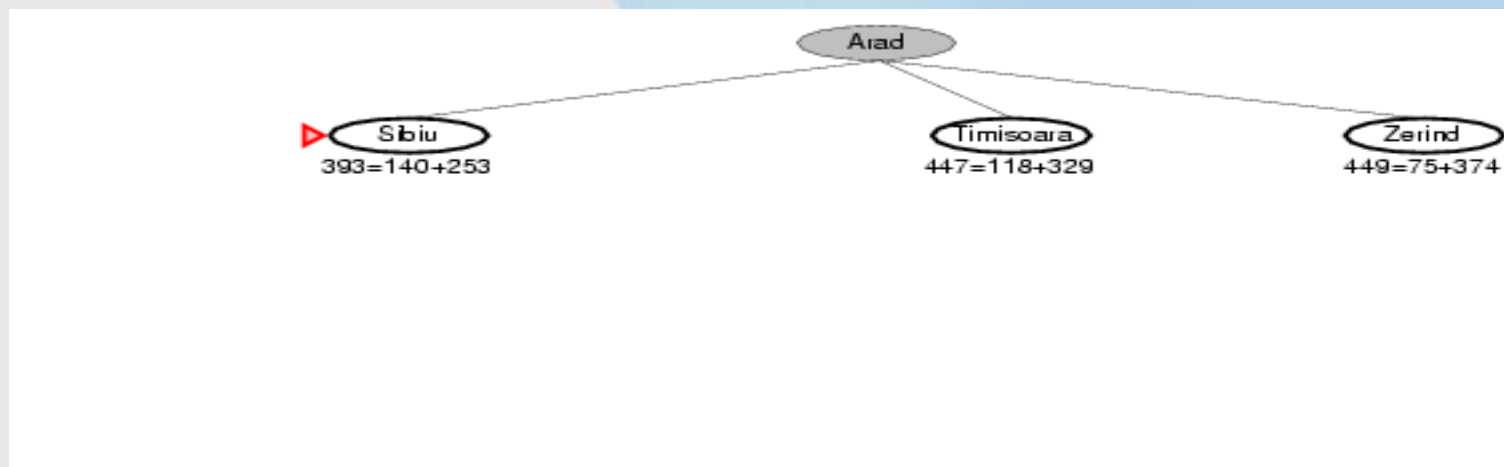
# A\* 搜索示例


 Arad  
 $366 = 0 + 366$

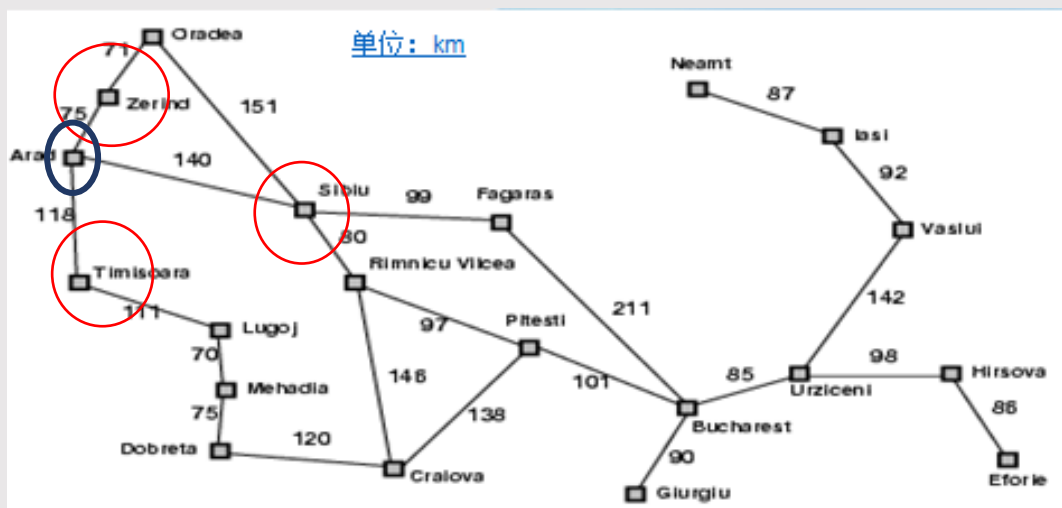


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

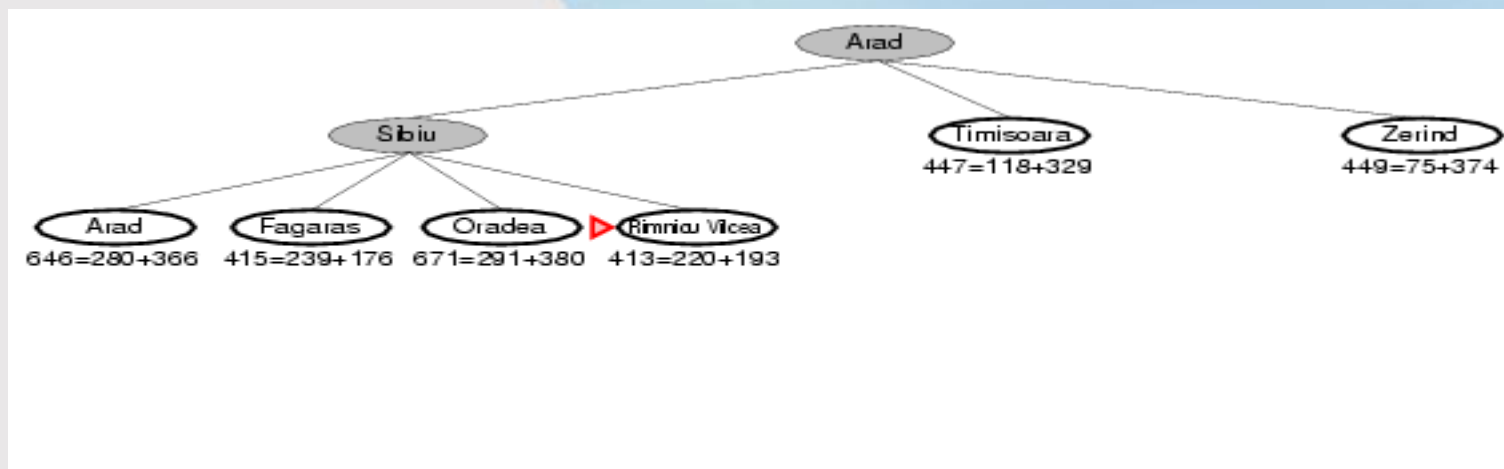
# A\*搜索示例



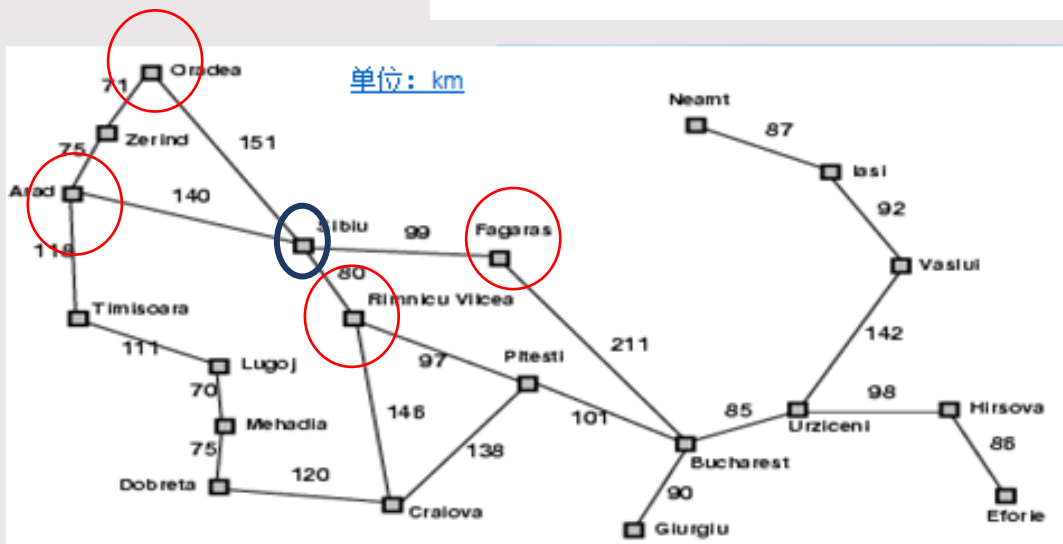
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



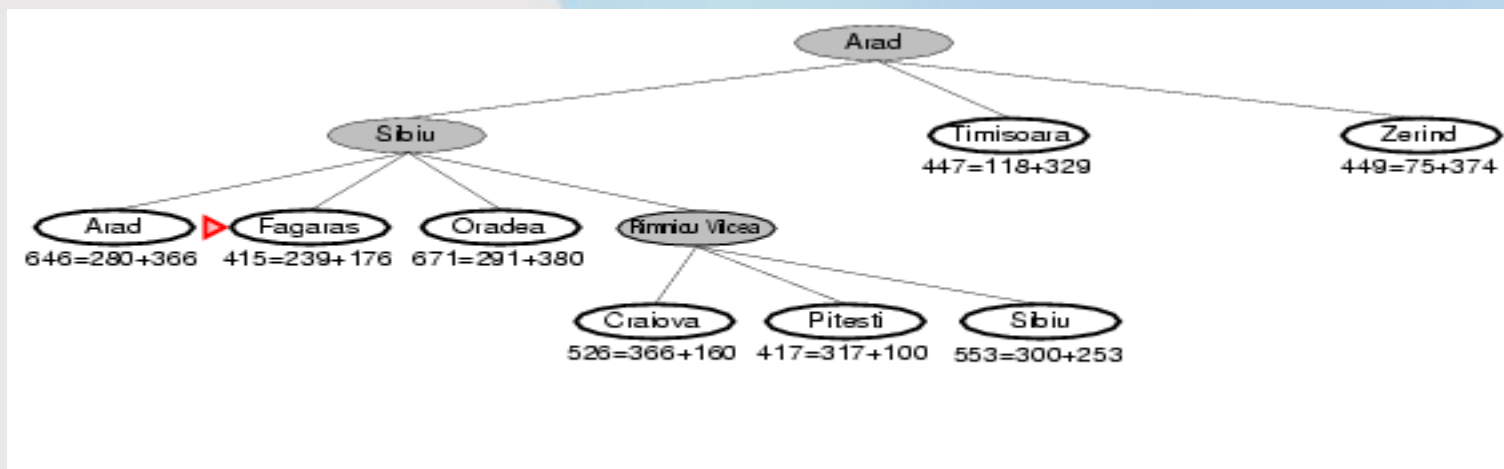
# A\*搜索示例



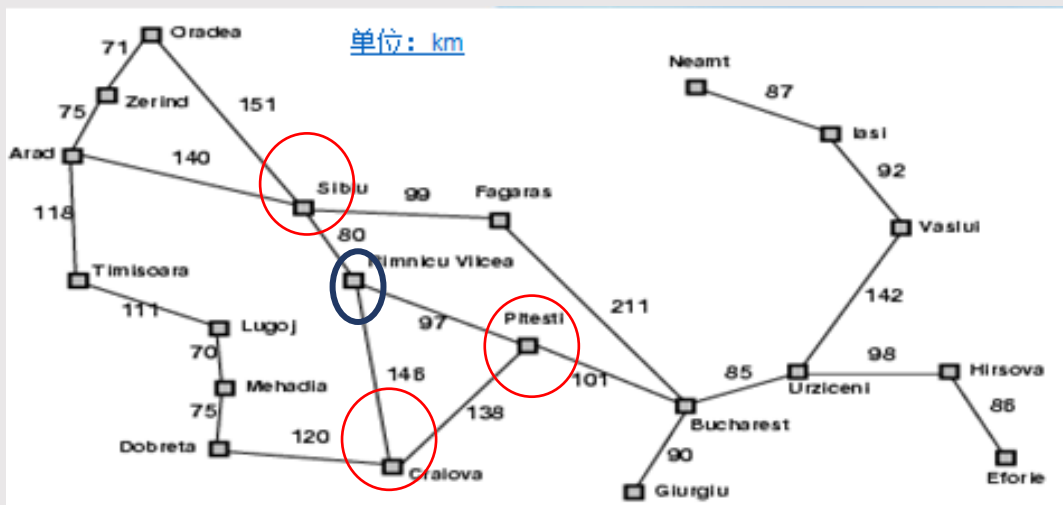
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	101
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



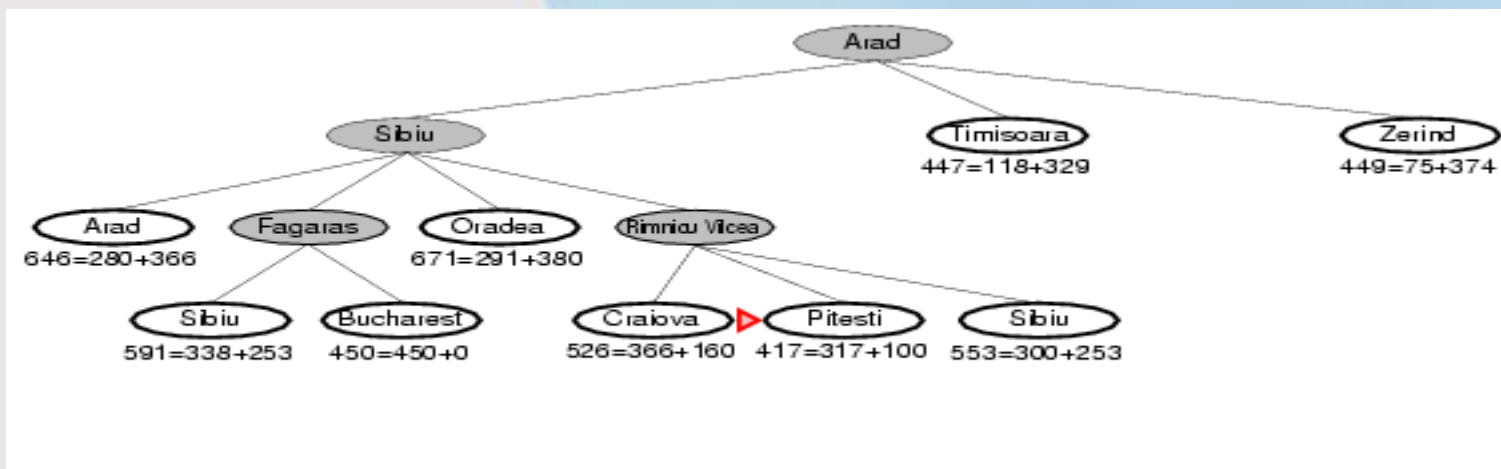
# A\*搜索示例



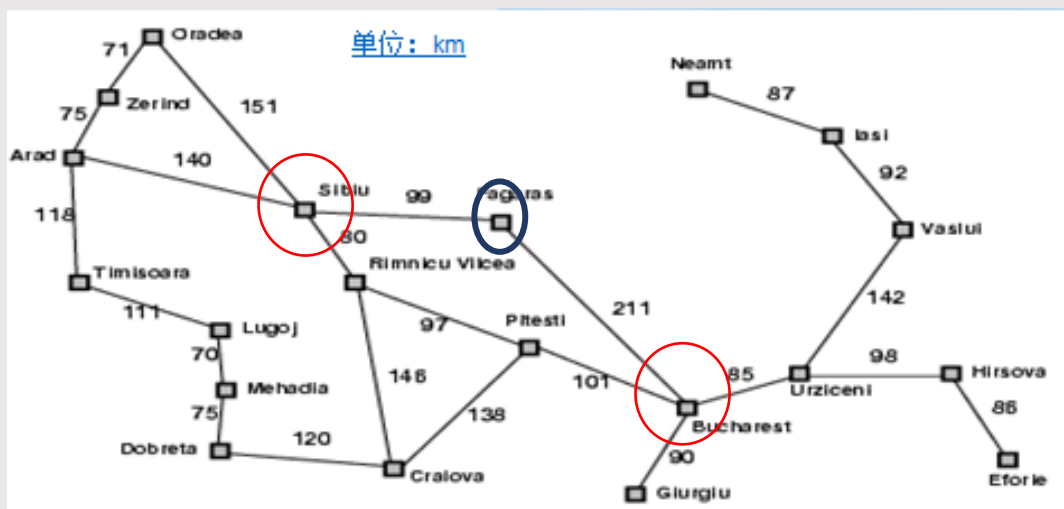
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	101
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



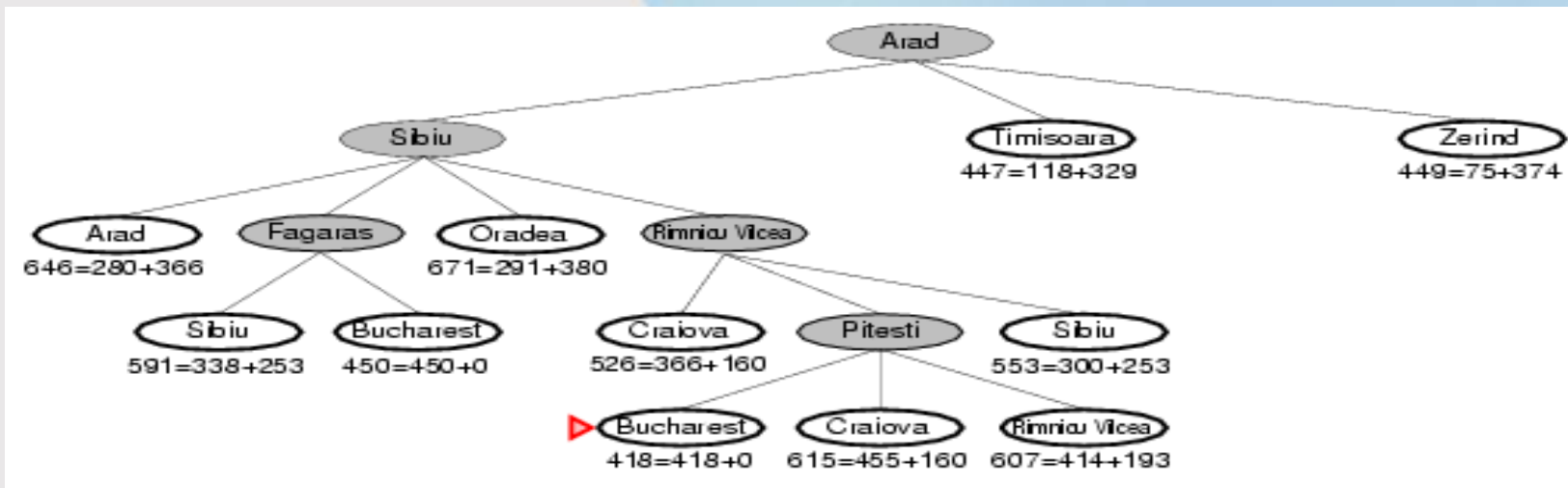
# A\*搜索示例



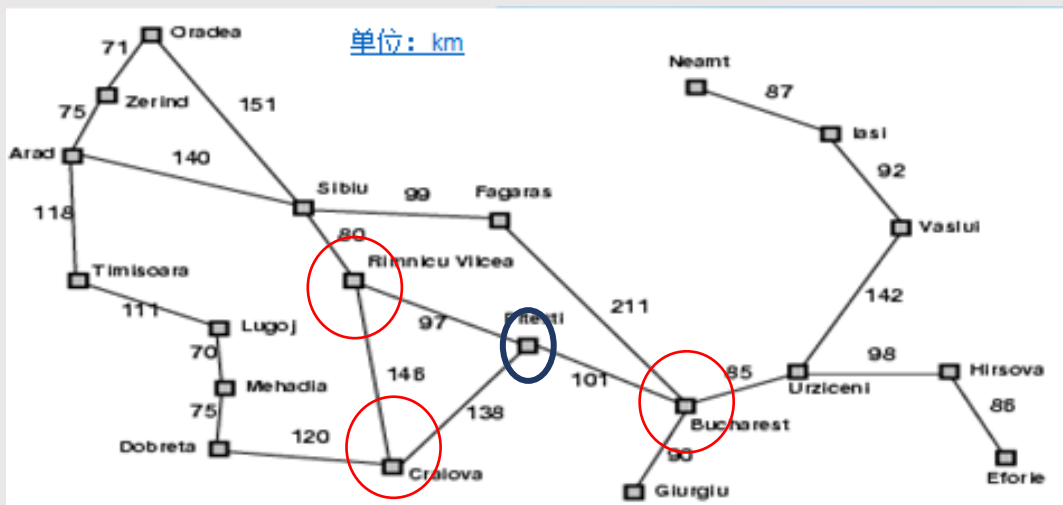
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



# A\*搜索示例



Straight-line distance to Bucharest	
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	101
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374





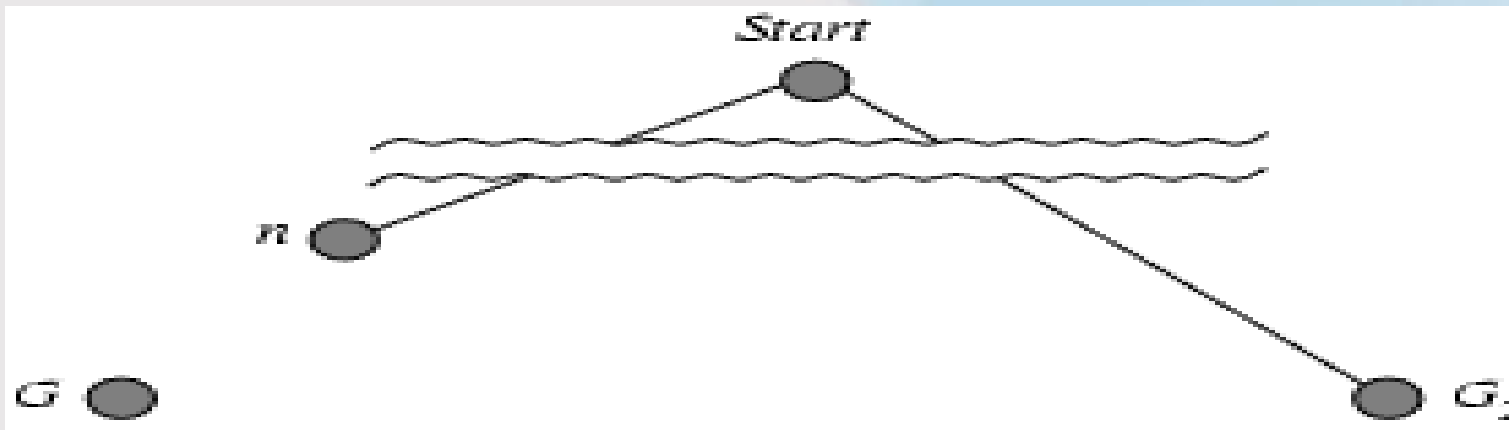
# 可采纳的启发式函数（admissible heuristic）

- 如果启发式函数 $h(n)$ 对于任意的节点 $n$ 都满足  $h(n) \leq h^*(n)$ ，这里  $h^*(n)$ 是指从节点 $n$ 到达目标的真正代价，则称 $h(n)$ 是可采纳的（admissible heuristic）。
- 定理：如果 $h(n)$ 是可采纳的，则  $A^*$  树搜索算法是具有最优性。

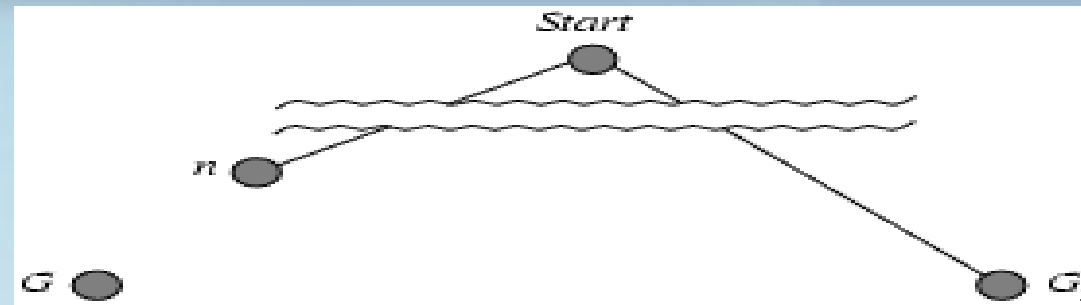


# A\* 的最优性证明

- 假设某次优目标节点  $G_2$  已经产生并在 fringe 表中排队，设  $n$  是 fringe 表中到达最优目标节点  $G$  的最短路径上的一个未扩展节点



# A\* 的最优性



①  $f(G_2) = g(G_2)$

②  $f(G) = g(G)$

③  $g(G_2) > g(G)$

④  $f(G_2) > f(G)$

⑤  $h(n) \leq h^*(n)$

⑥  $g(n) + h(n) \leq g(n) + h^*(n)$

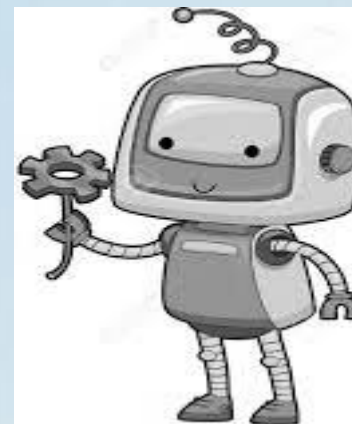
⑦  $f(n) \leq f(G)$

⑧  $f(G_2) > f(n)$

所以 A\* 决不会在选择节点n之前选择次优节点  $G_2$  扩展

# A\*搜索算法的性质

- 完备性? Yes
- 时间? 指数级
- 空间? 将所有产生的节点存储在内存中
- 最优性? Yes



## 5.3 A\*的改进方法

存储受限制的启发式搜索：

**思想：**迭代加深的A\*算法，用一个f-cost代替有限制的深度优先中的d作为截断值，进行剪枝。

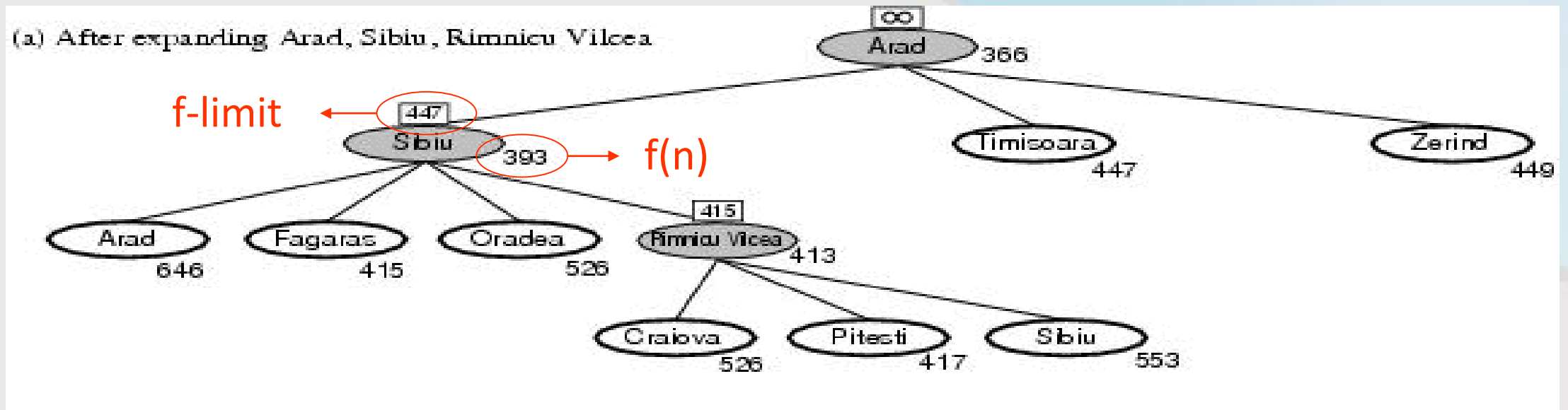
### ✓递归最佳优先搜索（RBFS）

- 尝试模拟标准的最佳优先搜索而只需线性空间。
  1. 记录当前节点的祖先可得到的最佳可替换路径的f值。
  2. 如果当前的f值超过了这个限制，则递归将转回到替换路径。
  3. 向上回溯改变f值到它的孩子的最佳f值
  4. 重复扩展这个上个节点，因为仍有可能存在较优解。

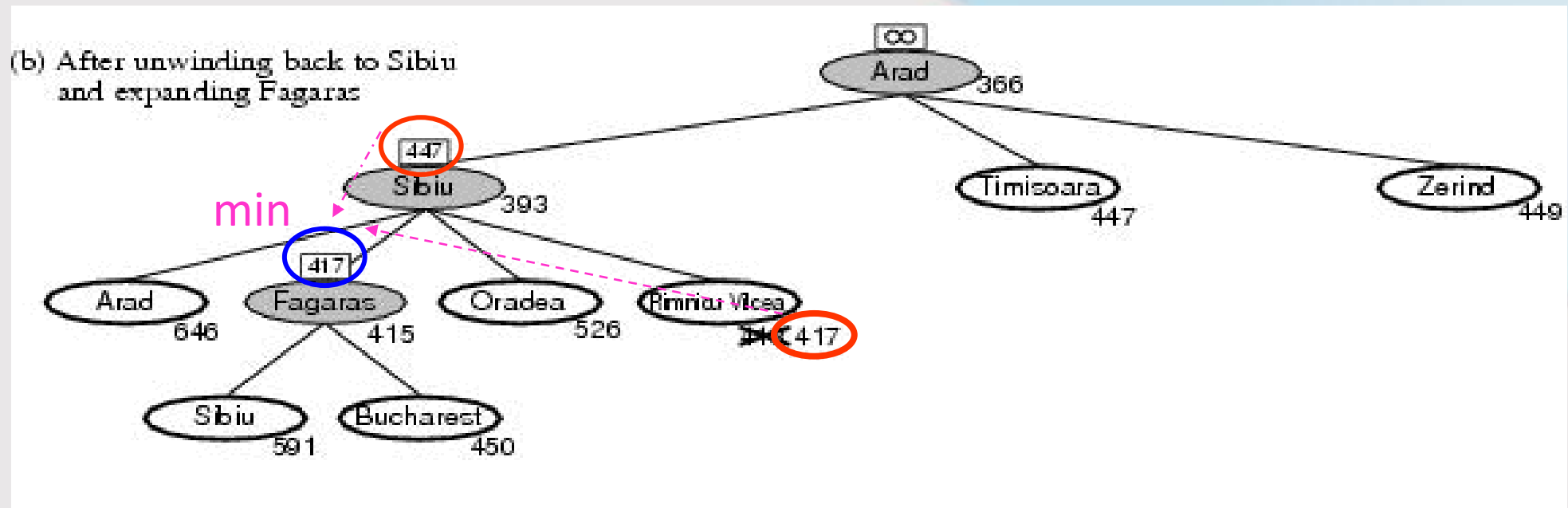
### ✓（简单）存储限制的A\*

- 当内存满了的时候删除最坏的节点

## 递归最佳优先搜索例子

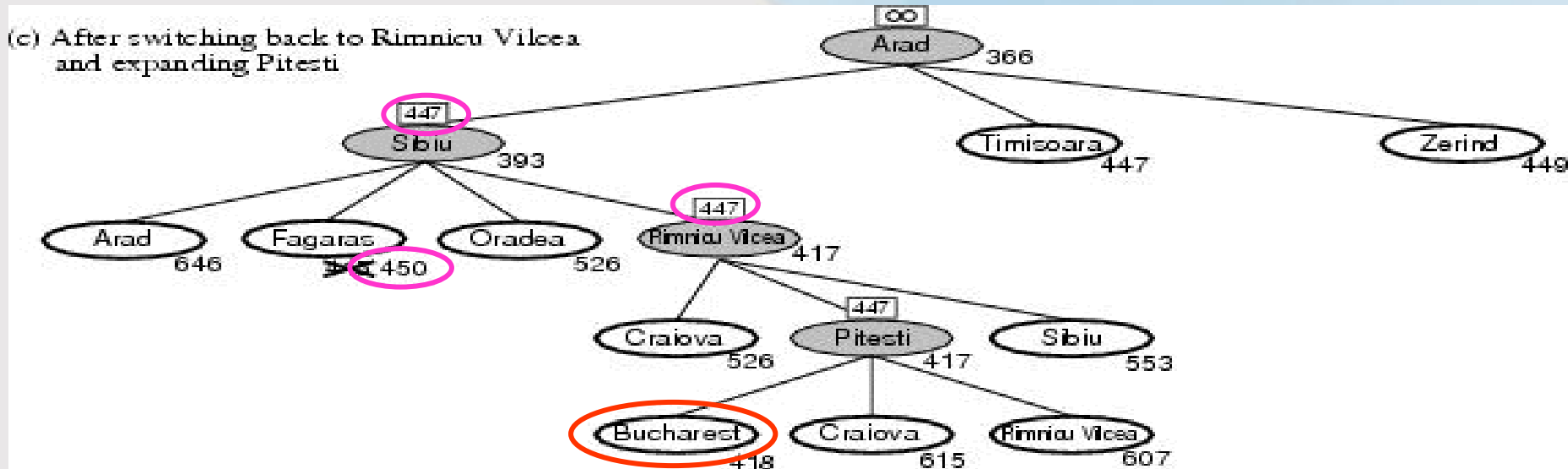


## 递归最佳优先搜索例子





## 递归最佳优先搜索例子



性能:

同A\*一样, 如果h是可采纳的, 则有最优性;

空间复杂度却是线性的 $O(bd)$ ;

时间复杂度难以描述: 取决于h的精确和最佳路径变换的次数。

**function** RECURSIVE-BEST-FIRST-SEARCH(*problem*) **return** a solution or failure

**return** RFBS(*problem*, MAKE-NODE(INITIAL-STATE[*problem*]),  $\infty$ )

**function** RFBS( *problem*, *node*, *f\_limit*) **return** a solution or failure and a new *f*-cost limit

if GOAL-TEST[*problem*](STATE[*node*]) then **return** *node*

*successors*  $\leftarrow$  EXPAND(*node*, *problem*)

if *successors* is empty then **return** failure,  $\infty$

for each *s* in *successors* do

$f[s] \leftarrow \max(g(s) + h(s), f[node])$

repeat

*best*  $\leftarrow$  the lowest *f*-value node in *successors*

if  $f[best] > f\_limit$  then **return** failure,  $f[best]$

*alternative*  $\leftarrow$  the second lowest *f*-value among *successors*

*result*,  $f[best] \leftarrow$  RBFS(*problem*, *best*, min(*f\_limit*, *alternative*))

if *result*  $\neq$  failure then **return** *result*

# (简化) 存储限制 A\*算法

- 思想：利用所有可以用的内存；  
扩展所有最佳节点直到内存满了；  
当内存满了，摘掉最差的那些叶子；  
将该节点值备份到父节点中；

如果所有的节点都有相同的 $f$ 值怎么办？

某些节点被选出了扩展或被删除；

简化的存储限制通过扩展新节点和删除最老的节点来简化处理。

(这是什么思想？深度还是宽度？)

性能：如果有解，简化的存储限制是完备的，也有最优性。

# 可采纳的启发式函数分析

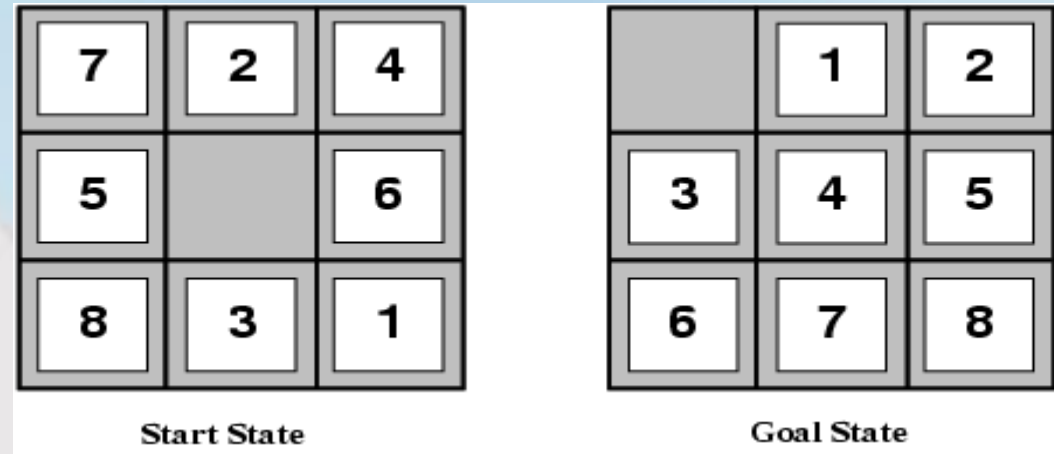
比如, 对于 8-数码问题:

- $h_1(n)$  = 错放的数字块个数
- $h_2(n)$  = 状态 $n$ 到目标状态的曼哈顿距离  
(i.e., 所有数字块到目标位置的曼哈顿距离的和)

7	2	4
5		6
8	3	1
Start State		

	1	2
3	4	5
6	7	8
Goal State		

# 可采纳的启发式函数分析



- $h_1(S) = ?$  8
- $h_2(S) = ?$   $3+1+2+2+2+3+3+2 = 18$

# 占优势的启发式函数-启发性（控制能力）

- 有效分支因子 $b^*$

- 如果A\*算法生成的总节点数是N，在解深度为d的一致树中所必须的分支因子 $b^*$ （含有N+1个节点）既有：

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

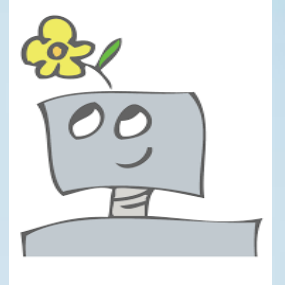
- 对于足够难的问题，该度量值是相当稳定的。
  - 因此，在小规模问题集合上实验测量出的 $b^*$ 值，可以为研究启发式的有效性提供很好的指导。
  - 一个好启发式的 $b^*$ 是接近1的。



# 占优势的启发式函数-启发性（控制能力）

- 为了测试 $h_1$ 、 $h_2$ 的质量和控制在随机产生了1200个八数码问题，他们的解长度为2到24，分别用迭代深度优先、 $A^*(h_1)$ 、 $A^*(h_2)$ 求解得到 $b^*$ 与实际代价的测试结果如下：

$d$	Search Cost			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26



- 在两个启发式均为可采纳的情况下：对于所有 $h_2(n) \geq h_1(n)$ 则 $h_2$ 的启发性（控制能力）要优于 $h_1$ 。



## 5.4 如何构建启发函数

- 可采纳的启发式可以源自简化版问题的一个 精确解。

称为**松弛问题**：对原定问题的动作的约束放宽，以松弛化问题的最优解的代价来定义原问题的一个可采纳启发式函数。

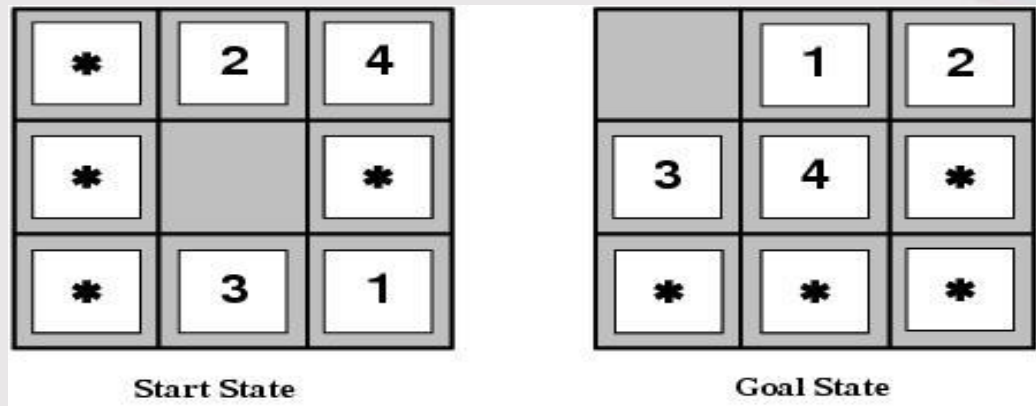
- 1)对于 $h_1$ 来说松弛8数码问题：一个牌可以移到任意的位置。  
因此， $h_1$ 是这个问题的最短路径解。
- 2)对于 $h_2$ 来说松弛8数码问题：一个牌可以移到任一相邻的位置。因此， $h_2$ 是这个问题的最短解。

明显的，一个松弛问题的最优解一定不会大于真实问题的最优解。

如：**ABSolver** 程序给出了魔方游戏第一个有用的启发式。

# 如何构建启发函数

- 可采纳的启发式可以从原问题的子问题的一个解得到。
- 这个代价是真实问题代价的下界。
- 将每个可能的子问题实例的精确解存储起来作为一个**模式数据库**。
  - 用这个模式数据库的构建一个完整的启发式



# 如何构建启发函数

- 另外一种方法就是从**经验中学习**:
  - 经验=求解大量的8数码游戏。
  - 例如：8数码的 $f(n)=g(n)+p(n)+3s(n)$ 效率最高。  
p:曼哈顿距离 s: 是否跟在正确的牌后

## 5.5 通过搜索进行问题求解——有信息的搜索

- 最佳优先搜索:  $f(n)=g(n)+h(n)$
- 贪婪最佳优先搜索:  $f(n)=h(n)$
- A\* 搜索: 其中  $h(n) \leq h^*(n)$  是可采纳的。
- A\* 的改进
- 启发函数的启发能力:
- 可采纳启发式设计: 松弛问题 模式数据库 学习机制

# 作业：编程实现：罗马尼亚度假问题

## 搜索策略

- 分别采用：宽度优先、深度优先、贪婪算法和A\*算法实现。
- 并能比较算法性能。

## 编程实现

- 编程语言自选
- 输入罗马尼亚简化地图

## 运行结果

- 图形化界面包含：  
地图、算法选择、耗散值、生成节点数统计、运行时间、路径搜索动态\*示意等

