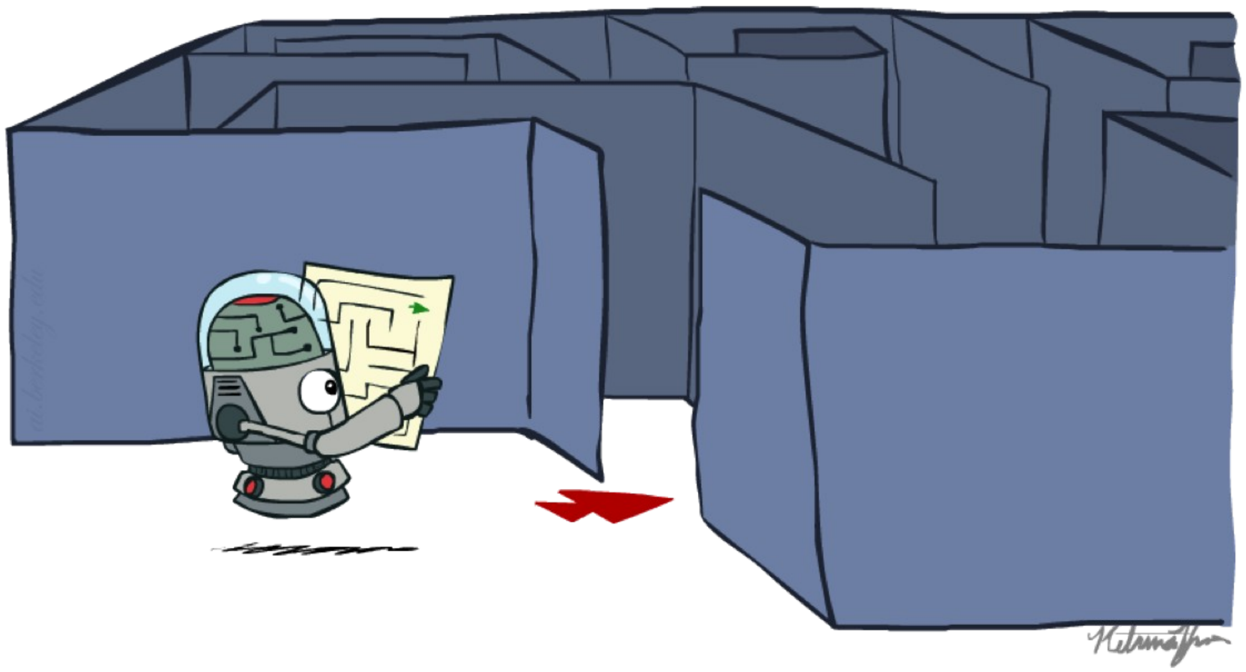
Artificial Intelligence

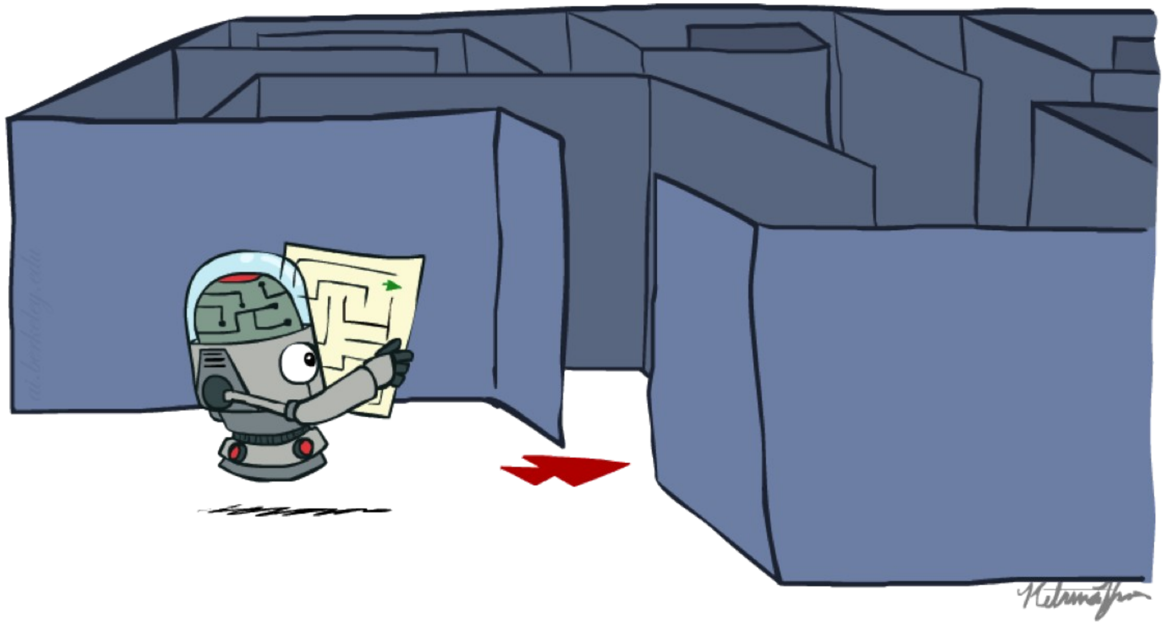
Search



[slides adapted from Dan Klein, Pieter Abbeel]

目录

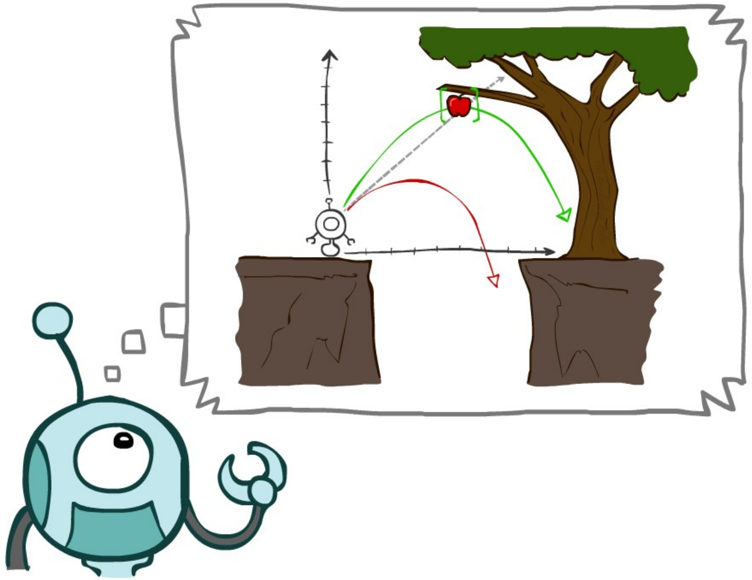
- 3.1 问题求解 Agent
- 3.2 问题形式化
- 3.3 搜索算法
- 3.4 无信息搜索策略



3.1 问题求解 Agent

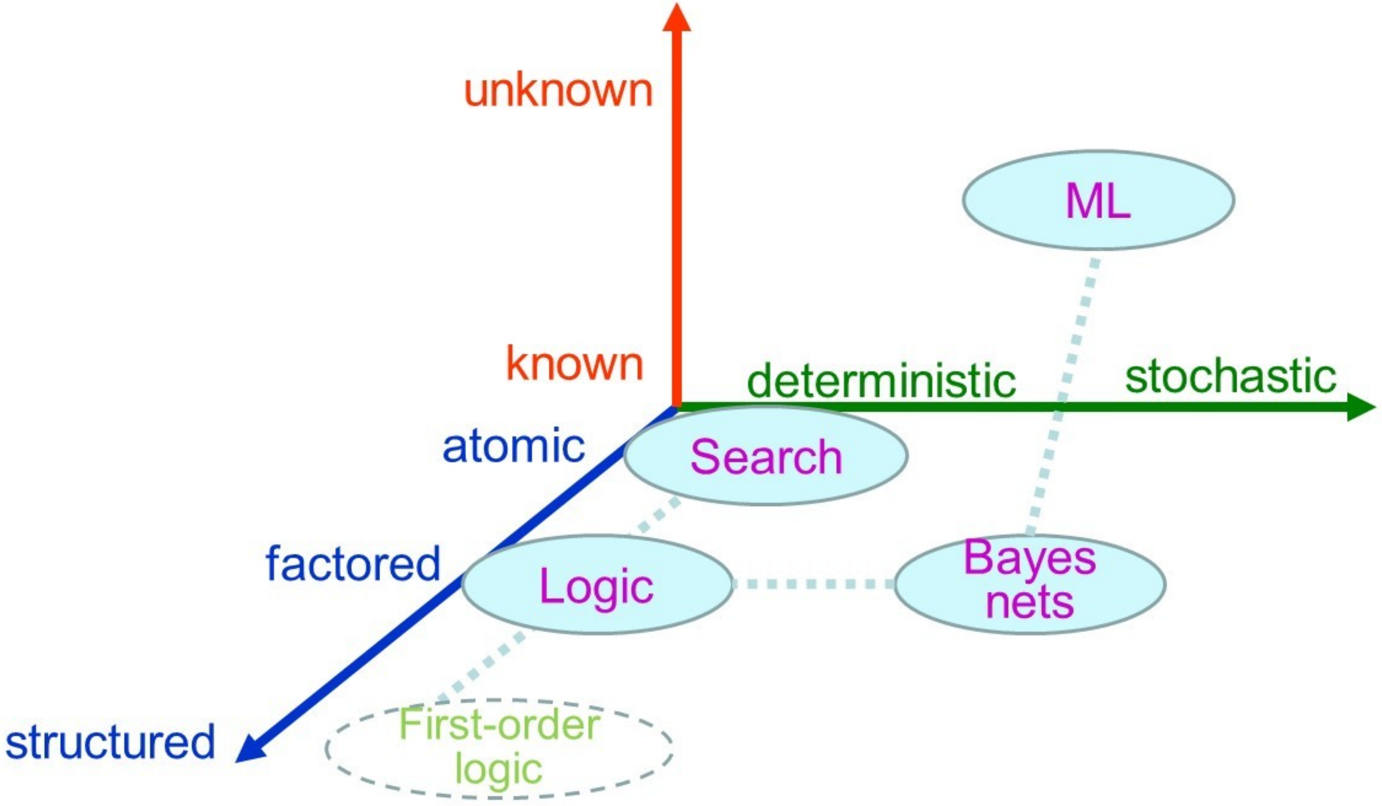
问题求解 Agent

- 为了达到目标，寻找一组行动序列的过程被称为**搜索**
- 搜索算法：问题 -> 问题的解（行动序列）



3.1 问题求解 Agent

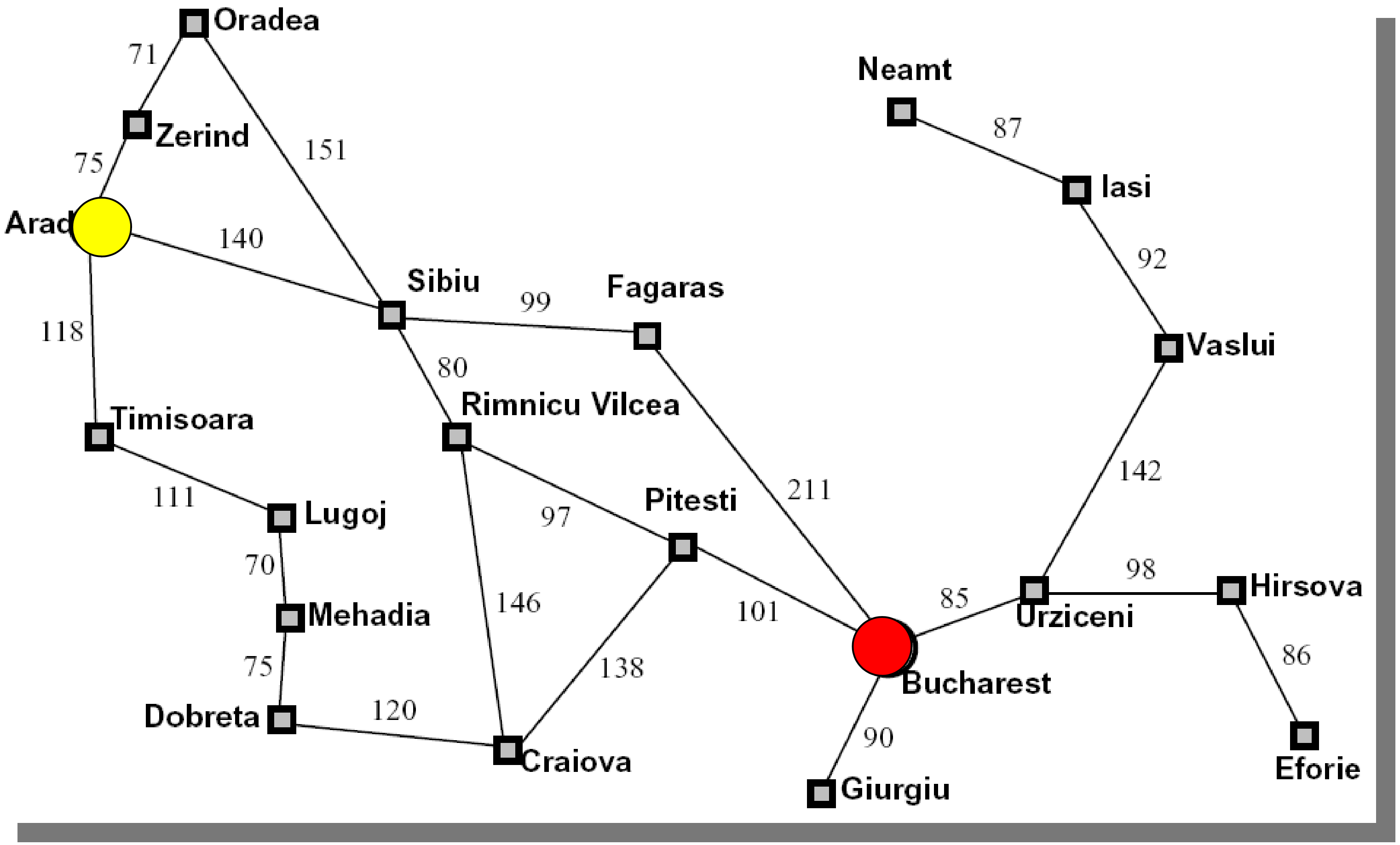
- 搜索算法一般假定：
  - 已知的、确定性的
  - 状态和动作是离散的
  - 完全可观察的
  - 任务较简单
- 通常有一个明确的目标
- 以最小的代价找到目标



Example: Traveling in Romania

问题导入

- 罗马尼亚问题：
  一个 **Agent** 如何从罗马尼亚的 **Arad** 走到 **Bucharest** ？



搜索进行问题求解的过程可分解为三个阶段：

形式化、搜索、执行

搜索：问题 -> 解（行动序列）

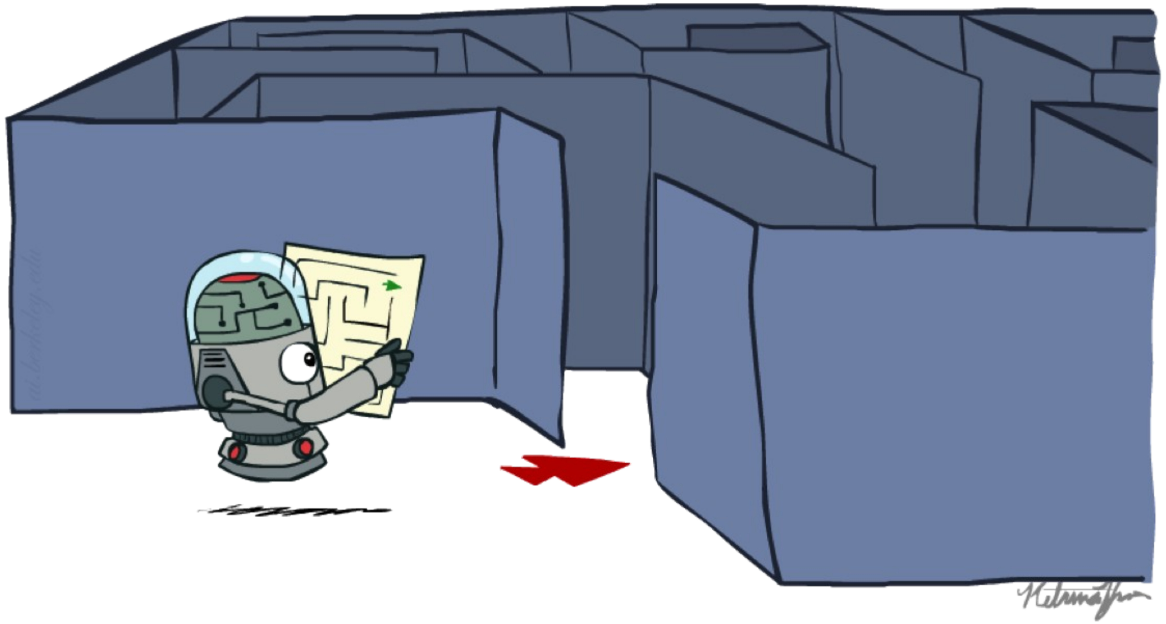
Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns 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

    state ← UPDATE-STATE(state, percept) // What is the current state?
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)   // 目标形式化
        problem ← FORMULATE-PROBLEM(state, goal)   // 问题形式化
        seq ← SEARCH( problem)  // 搜索
    action ← FIRST(seq)   // 执行
    seq ← REST(seq)
    return action
```

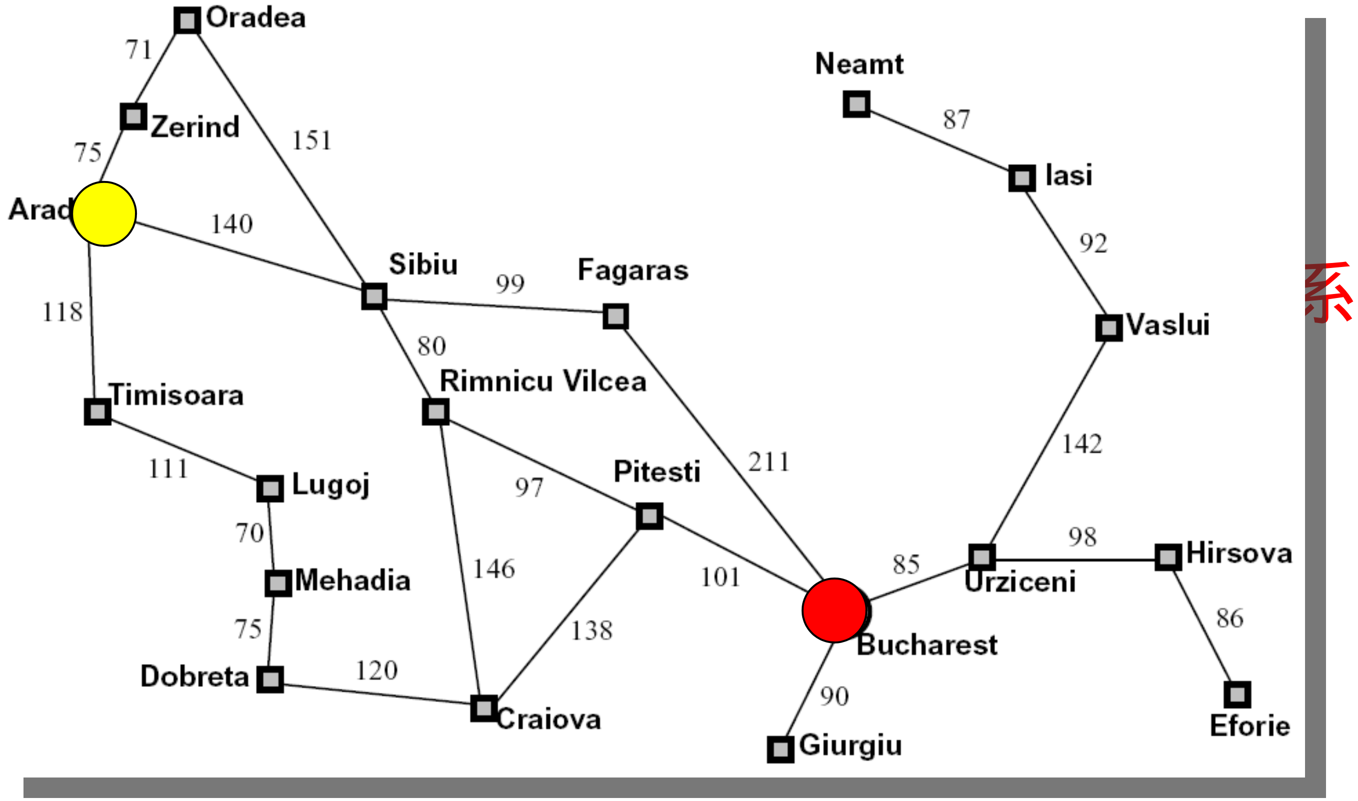
目录

- 3.1 问题求解 Agent
- 3.2 问题形式化
- 3.3 搜索算法
- 3.4 无信息搜索策略



问题形式化描述

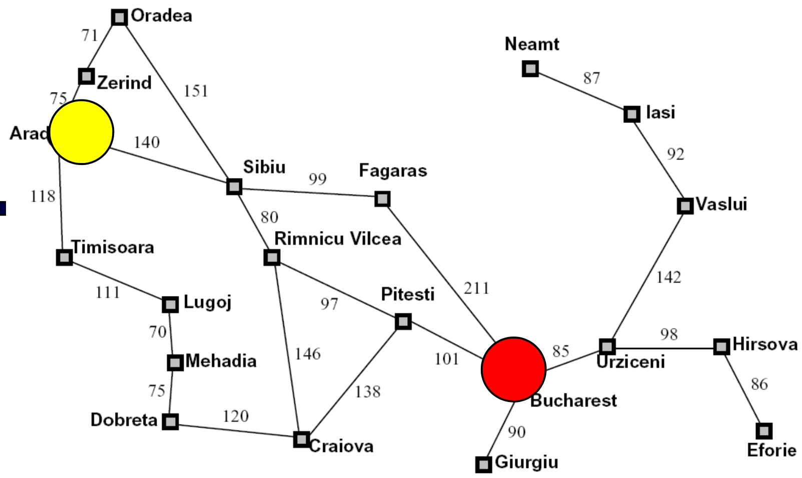
- 一个问题可以用 5 个组成部分形式化地描述：
  - Agent 的初始状态　s_0
  - Agent 的可能行动　ACTION(s)
  - 转移模型　RESULT(s,a)
  - 目标测试　s==s_g
  - 路径耗散　$c(s,a,s')$



状态空间图：罗马尼亚地图

系

形式化描述



- 一个问题用 5 个部分进行形式化描述（以罗马尼亚问题为例

1 ）初始状态 ： In Arad

2 ）行动 ： ACTIONS(s) ，给定一个状态 s ， ACTIONS(s) 返回状态 s 下可以执行的动作的集合，例如状态 s 为“ In Arad” ， ACTIONS(s) 返回的行动为 { Go (Sibiu), Go (Timisoara), Go (Zerind) }

3 ）转移模型 ： RESULT(s ， a) ，在状态 s 下，执行 a 动作后，达到的状态。

RESULT(In (Arad) ， Go (Sibiu)) = In (Sibiu)

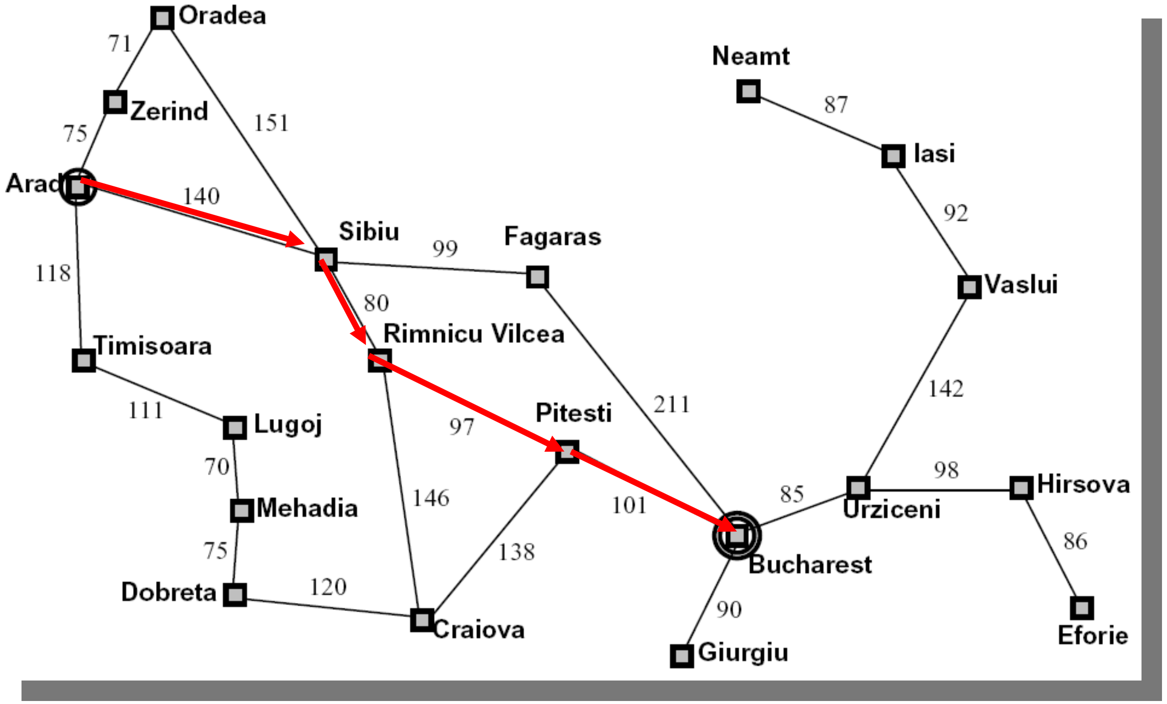
4 ）目标测试 ：确定给定的状态是不是目标状态。目标状态集为 { In (Bucharest) }

5 ）路径耗散 ：路径耗散函数为每条路径赋一个耗散值，即边权。

罗马尼亚案例中，路径耗散可用公里数表示的路径长度。从状态 s 采用行动 a 走到状态

形式化描述

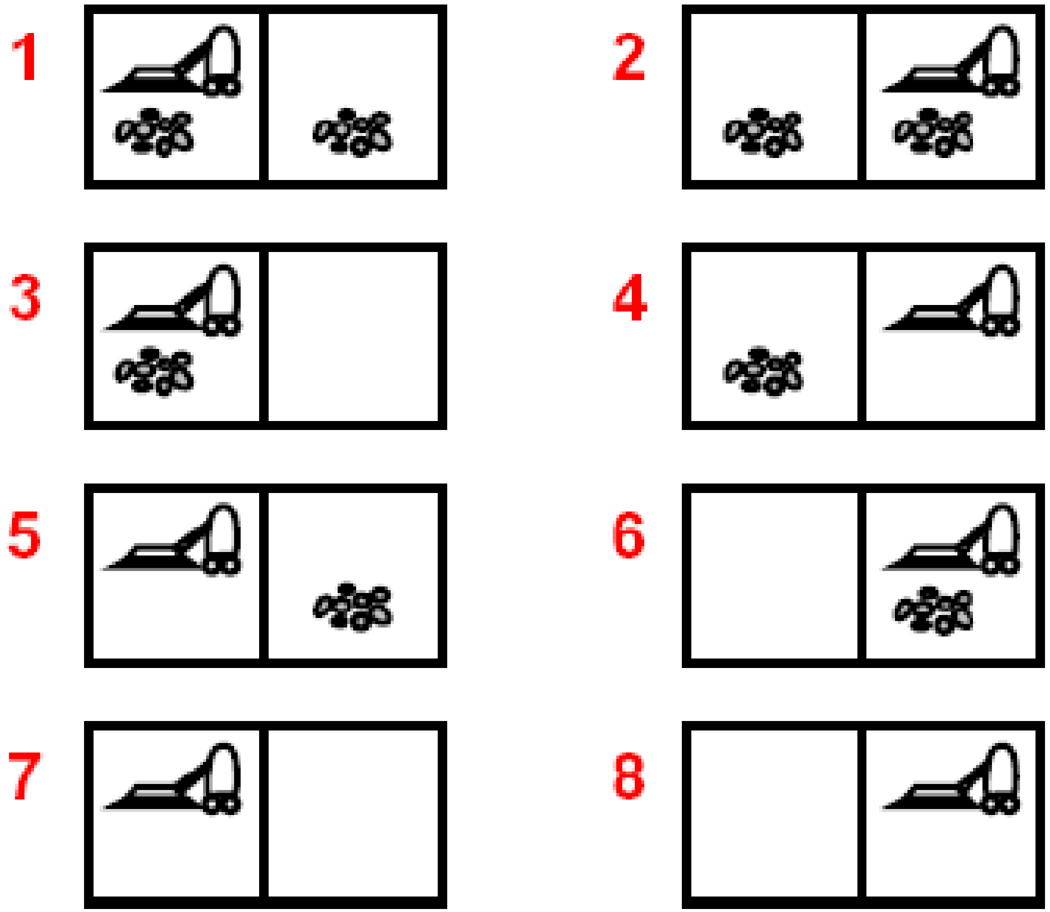
- 问题的解：
  - 从初始状态到目标状态的一组行动序列
  - 解的质量由路径耗散函数度量
  - 路径耗散值最小的解为最优解



状态空间图：罗马尼亚地图

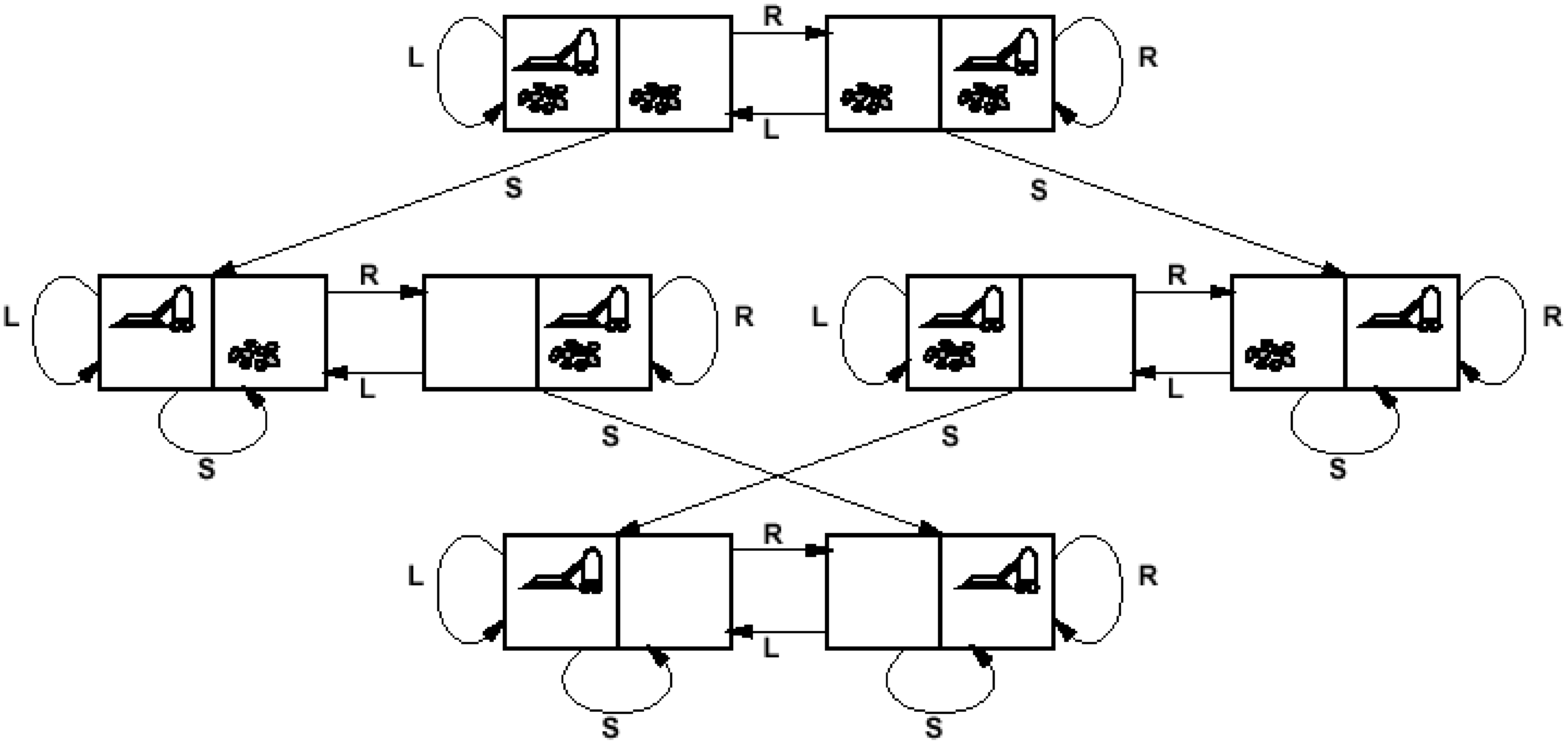
举例：真空吸尘器世界

- Simple world
  - 8 States (状态



问题形式化— Vacuum World

问

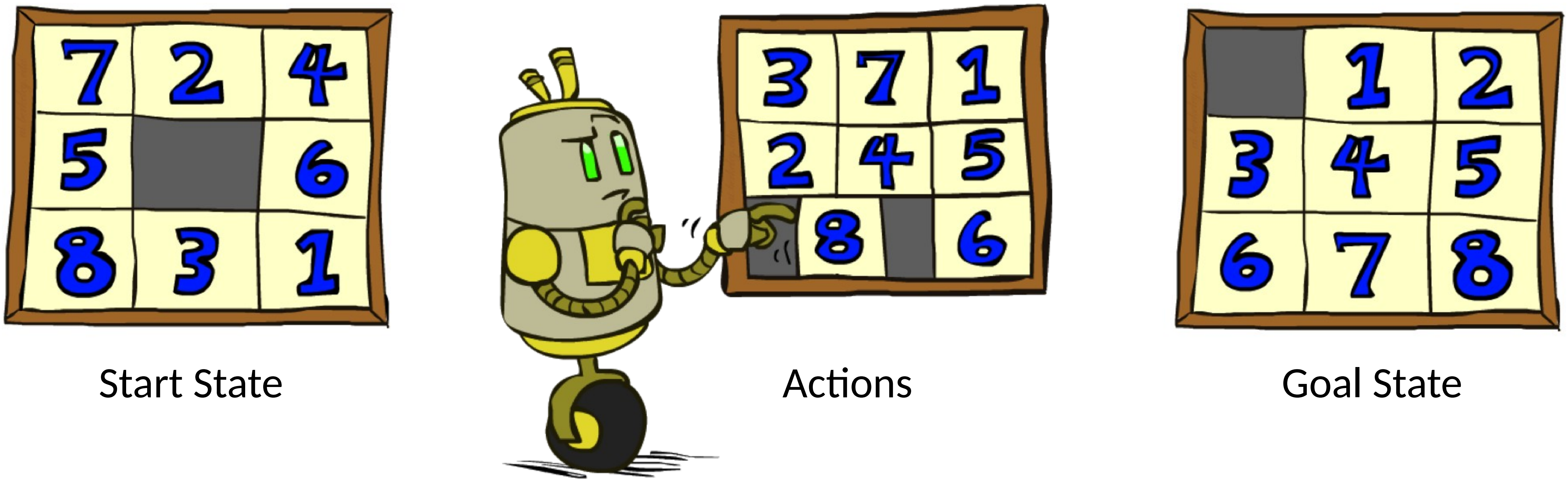


解

问题形式化—相关实例 ：8-puzzle

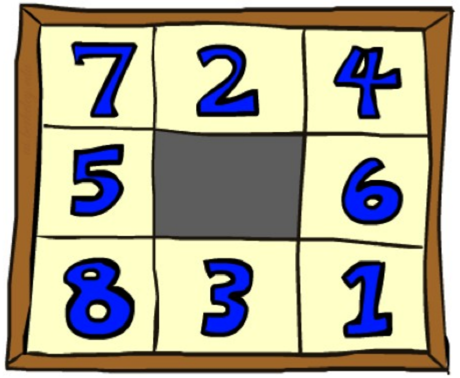
八数码问题游戏：

一个 **3x3** 的棋盘中有 **8** 个数字棋子和一个空格，与空格相邻棋子可滑动到空格中。游戏目标要达到右侧给出的指定状态。

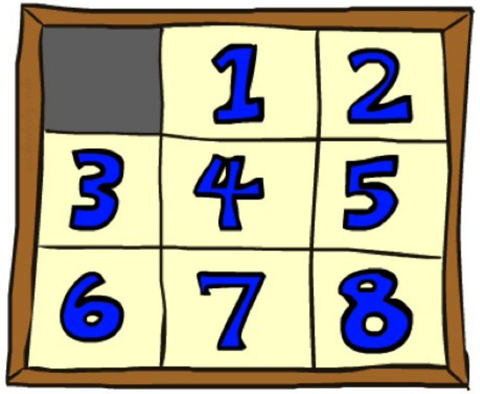


问题形式化—相关实例 ：8-puzzle

八数码问题游戏：
　　一个 **3x3** 的棋盘中有 **8** 个数字棋子和一个空格，与空格相邻棋子可滑动到空格中。游戏目标要达到右侧给出的指定状态。



start state



goal state

- Start state: √
- Actions ：
- Transform model:
- Goal test: √
- Path cost:

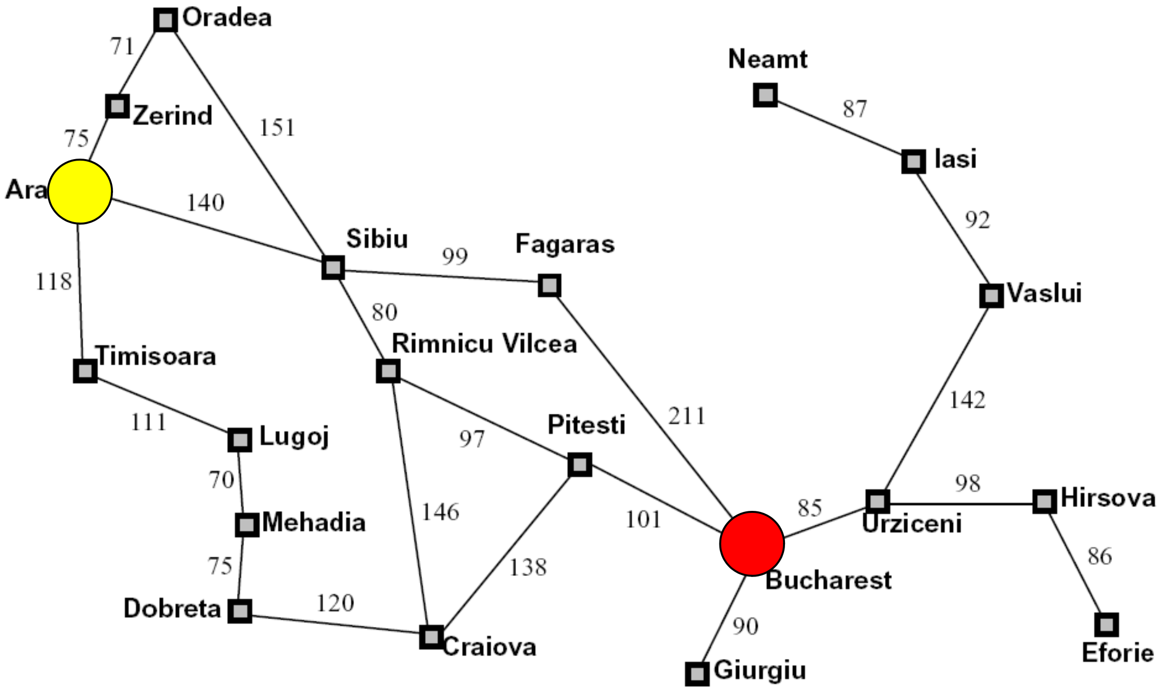
状态空间：9 ！

给定初始状态后，可能的状态共有 9!/2=181440 个

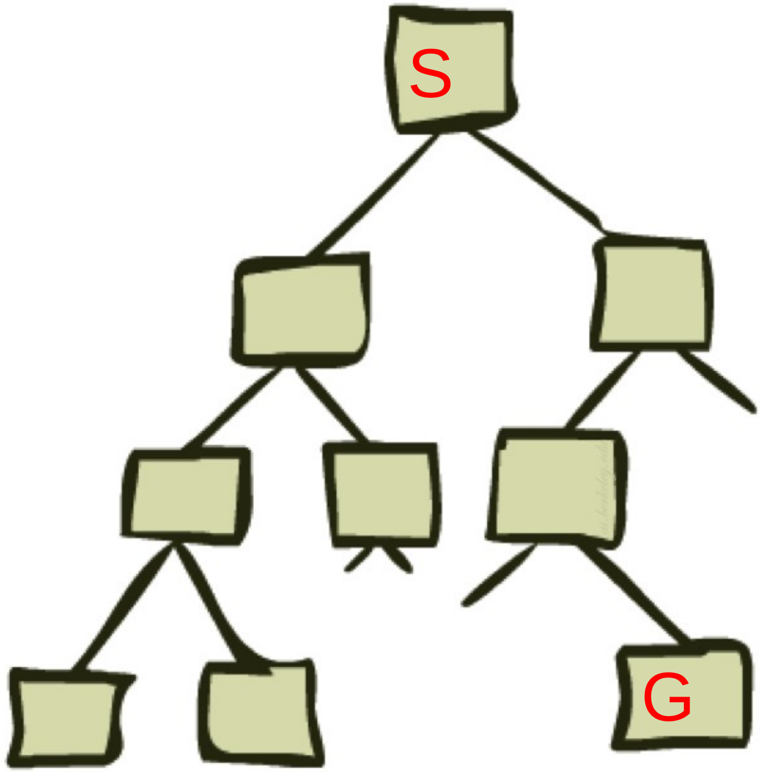
So, we need a principled way to look for a solution in these huge search spaces…

状态空间图和搜索树

罗马尼亚问题



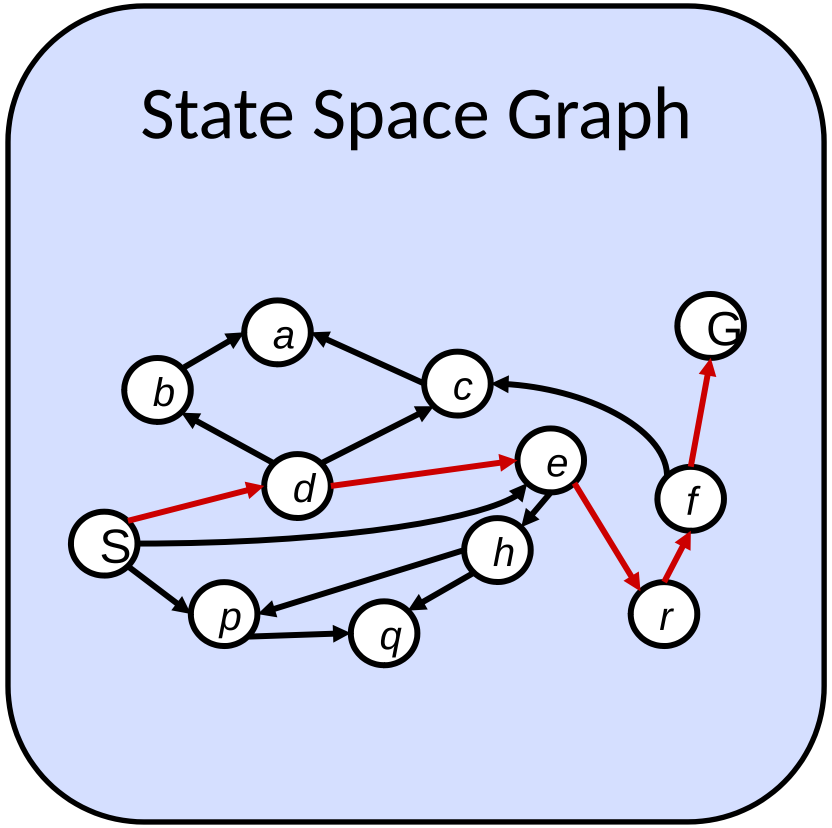
状态空间图

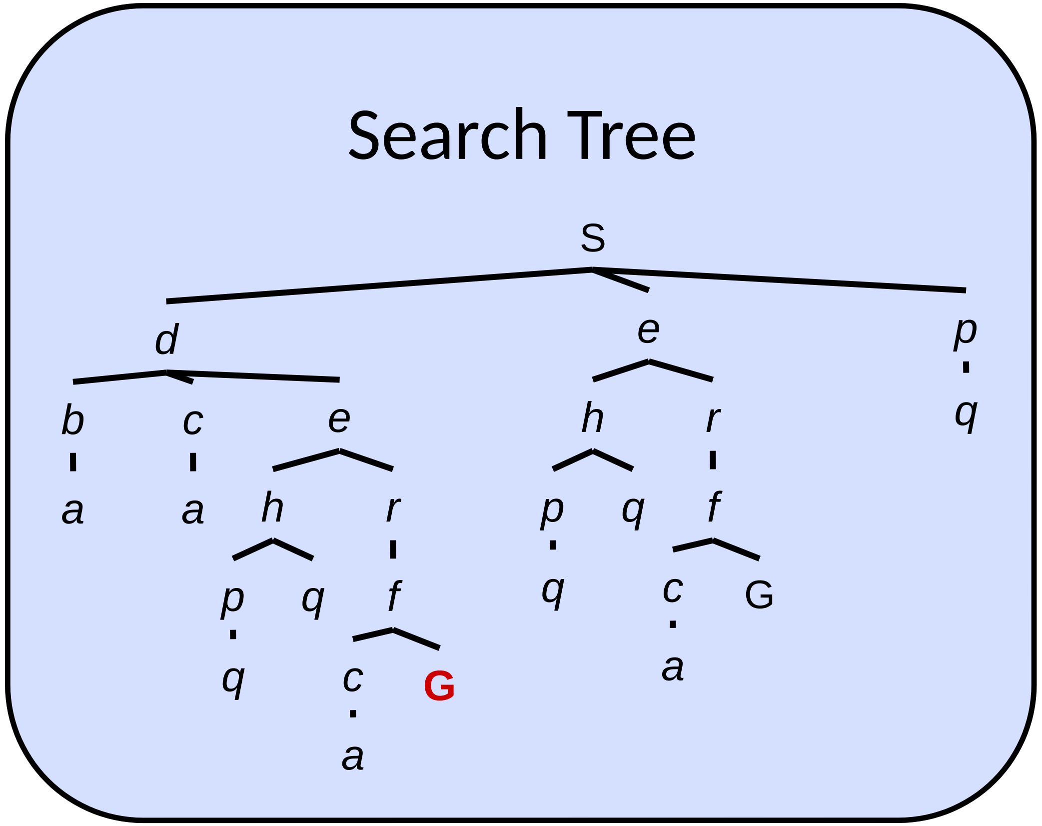


搜索树

状态空间图：对搜索问题的数学表达，，每个状态只出现一次； 搜索树：根结点是初始状态，状态可能会出现多

状态空间图和搜索树



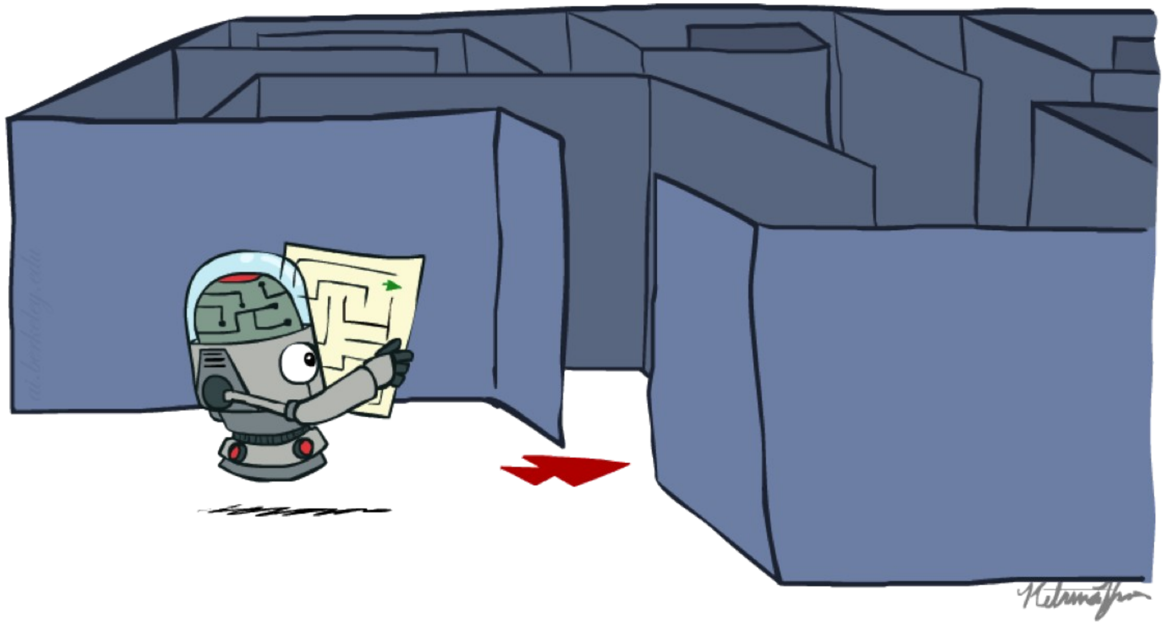


结点 = 状态

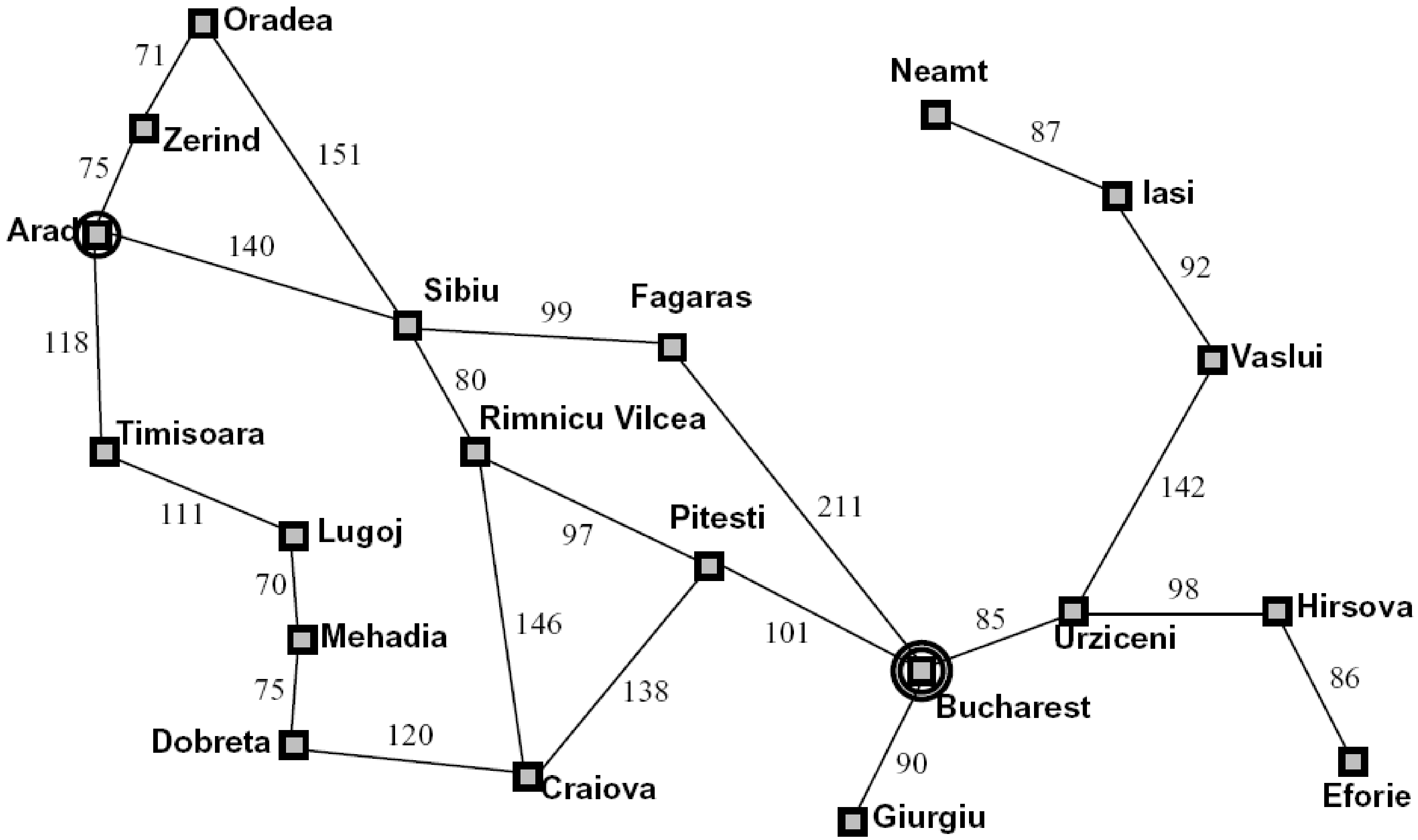
边 = 行动

目录

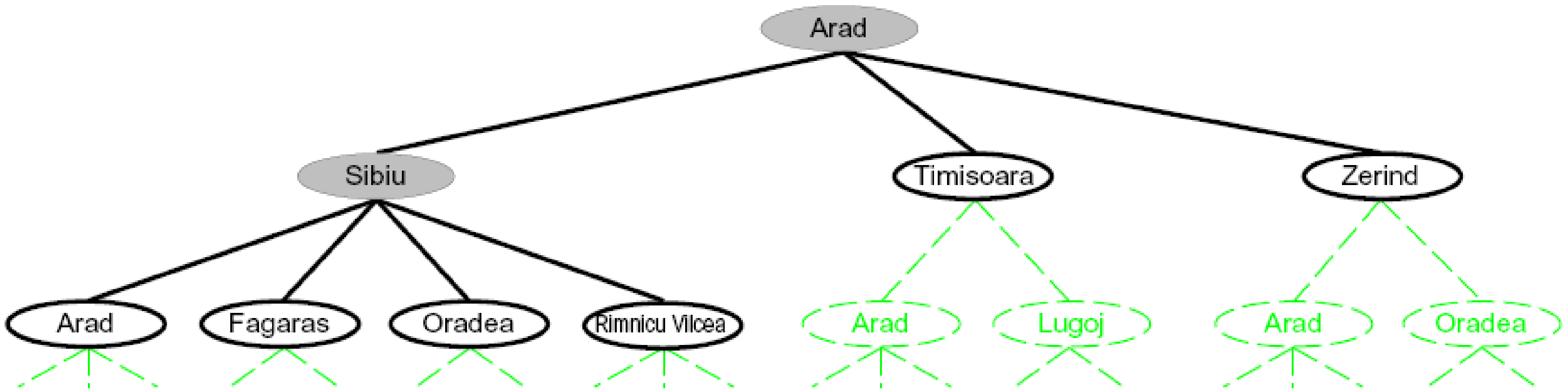
- 3.1 问题求解 Agent
- 3.2 问题形式化
- 3.3 搜索算法
- 3.4 无信息搜索策略



Search Example: Romania



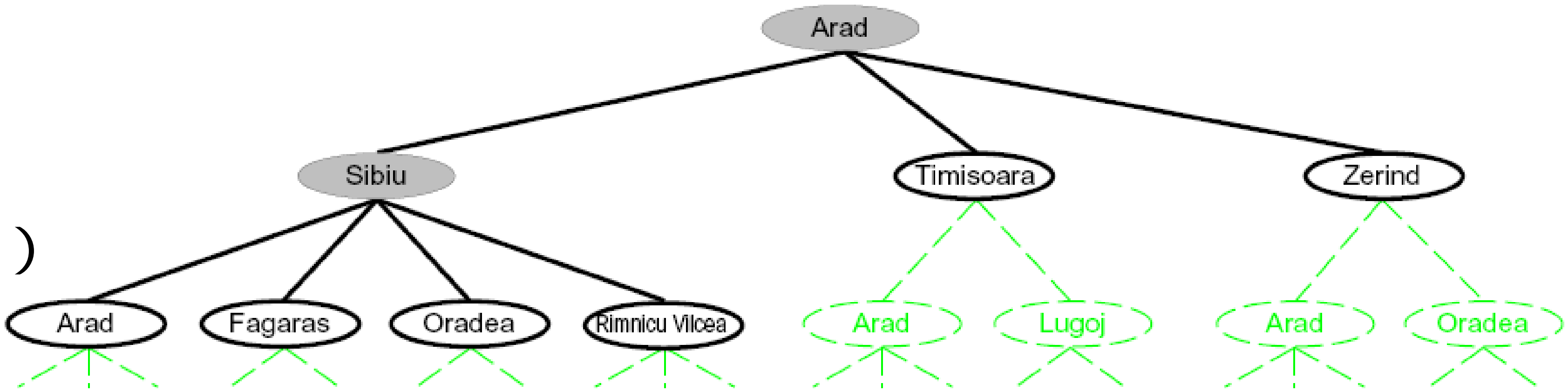
Searching with a Search Tree



- 搜索：
  - 选择将要扩展的状态（树的结点）
  - 边缘：给定时间点，所有待扩展的叶结点的集合

General Tree Search

- 结点类型：
  - 已扩展过的
  - 已生成但未被扩展（边缘）
  - 未生成的



- 问题： 如何选择将要扩展哪个状态？

搜索策略 (*strategy*): 确定结点扩展的顺序

General Tree Search

- 结点类型：
  - 已扩展过的
  - 已生成但未被扩展（边缘）
  - 未生成的

```
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
    end
```

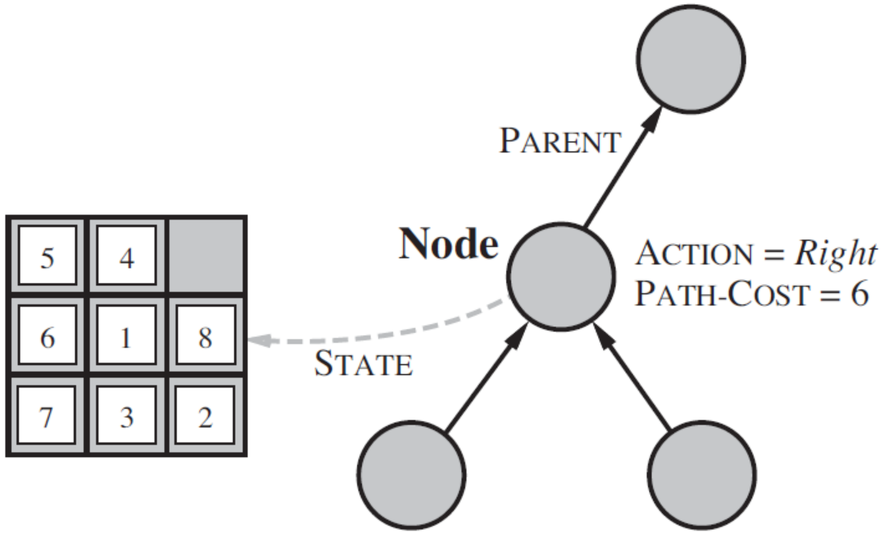
- 问题： 如何选择将要扩展哪个状态？

搜索策略 (*strategy*)：确定结点扩展的顺序

例如：随机选择结点扩展

如何找到解的路径（行动序列）？

- 搜索算法用一个数据结构来记录搜索树的构造过程。
- 对于树中的每个结点 n ，定义四个元素：
  - n.State: 对应状态空间中的状态；
  - n.Parent: 搜索树中产生该结点的结点（父结点）
  - n.Action: 父结点生成该结点时所采取的行动
  - n.Path-cost: 从初始状态到该结点的路径耗散

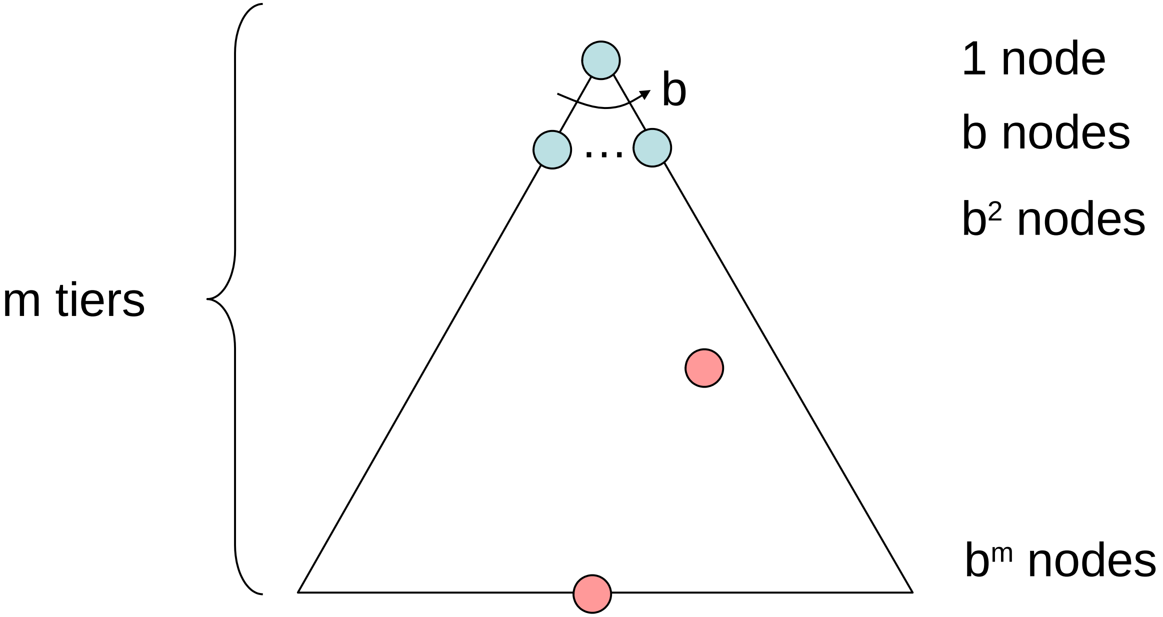


- 搜索找到问题的目标状态时，借助 **Parent** 可以找到解的路径

问题求解算法的性能

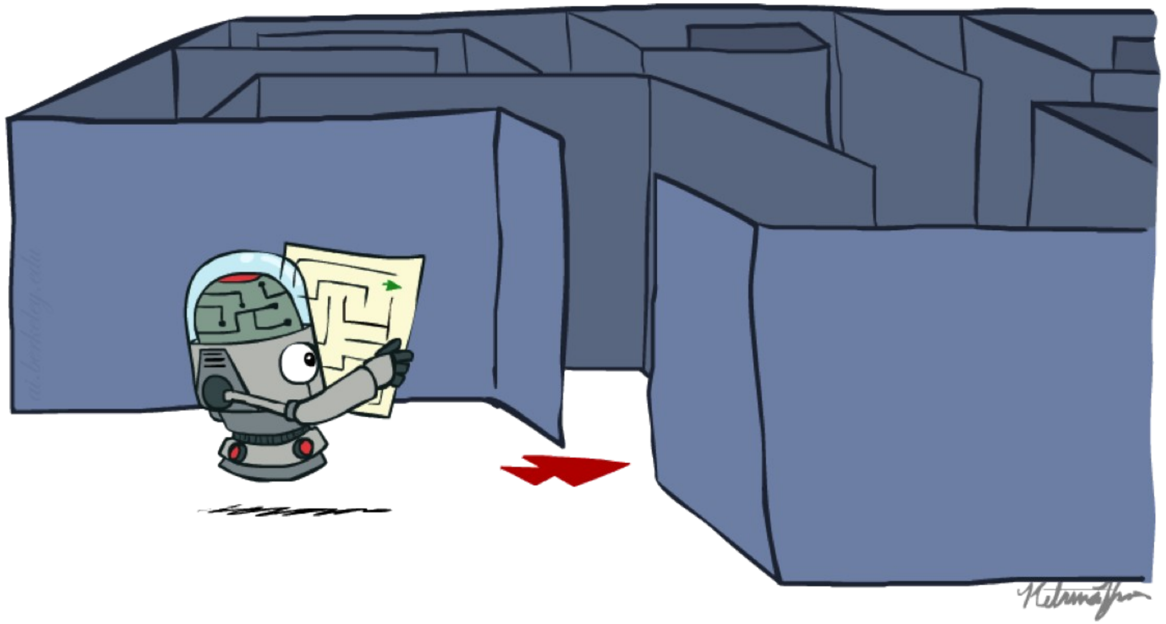
- 完备性： 算法是否能保证找到解？
- 最优性：搜索策略能否找到最优解？
- 时间复杂度：找到解需要花费多长时间？
- 空间复杂度：执行搜索的过程中需要多少内存？

- 搜索树：
  - b ：分支因子
  - m ：最大深度
  - solutions at various depths



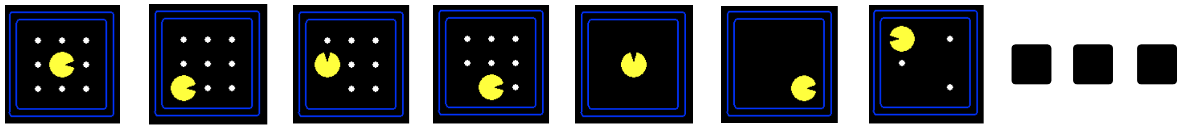
- 整个树中结点个数：
  - $1 + b + b^2 + ... b^m = O(b^m)$

目录

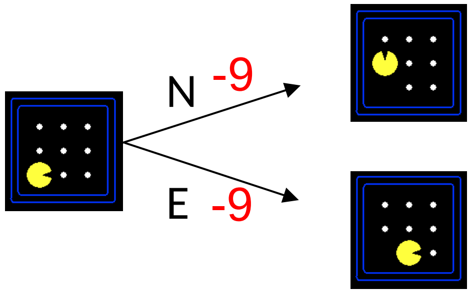


Search Problems

- A search problem consists of:



  - A state space *S*
  - An initial state s_0
  - Actions *A*(*s*) in each state
  - Transition model *Result*(*s*,*a*)
  - A goal test *G*(*s*)
    - *s* has no dots left
  - Action cost *c*(*s*,*a*,*s'*)
    - +1 per step; -10 food; -500 win; +500 die; -200 eat ghost

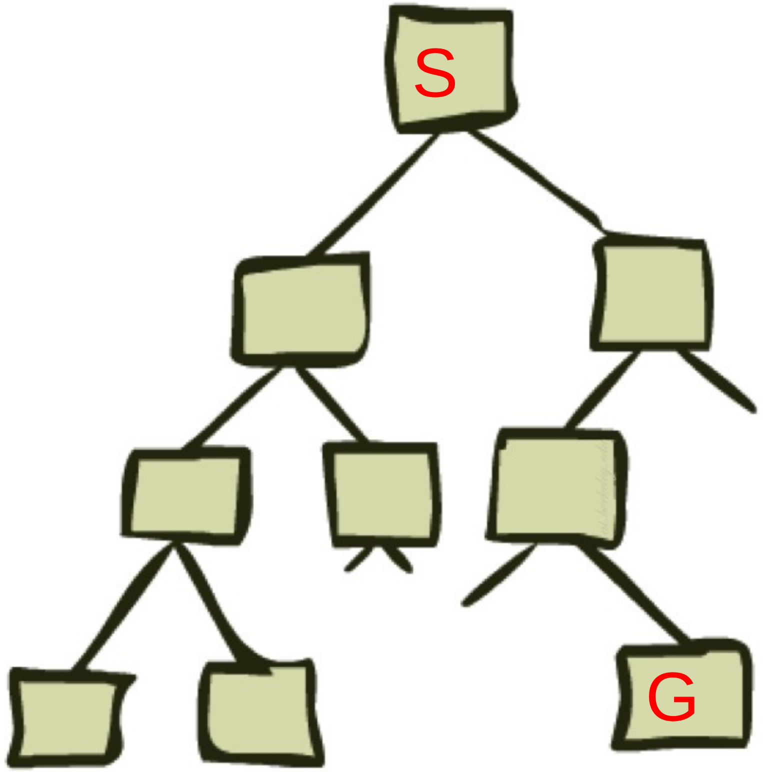


- A solution is an action sequence that reaches a goal state
- An optimal solution has least cost among all solutions

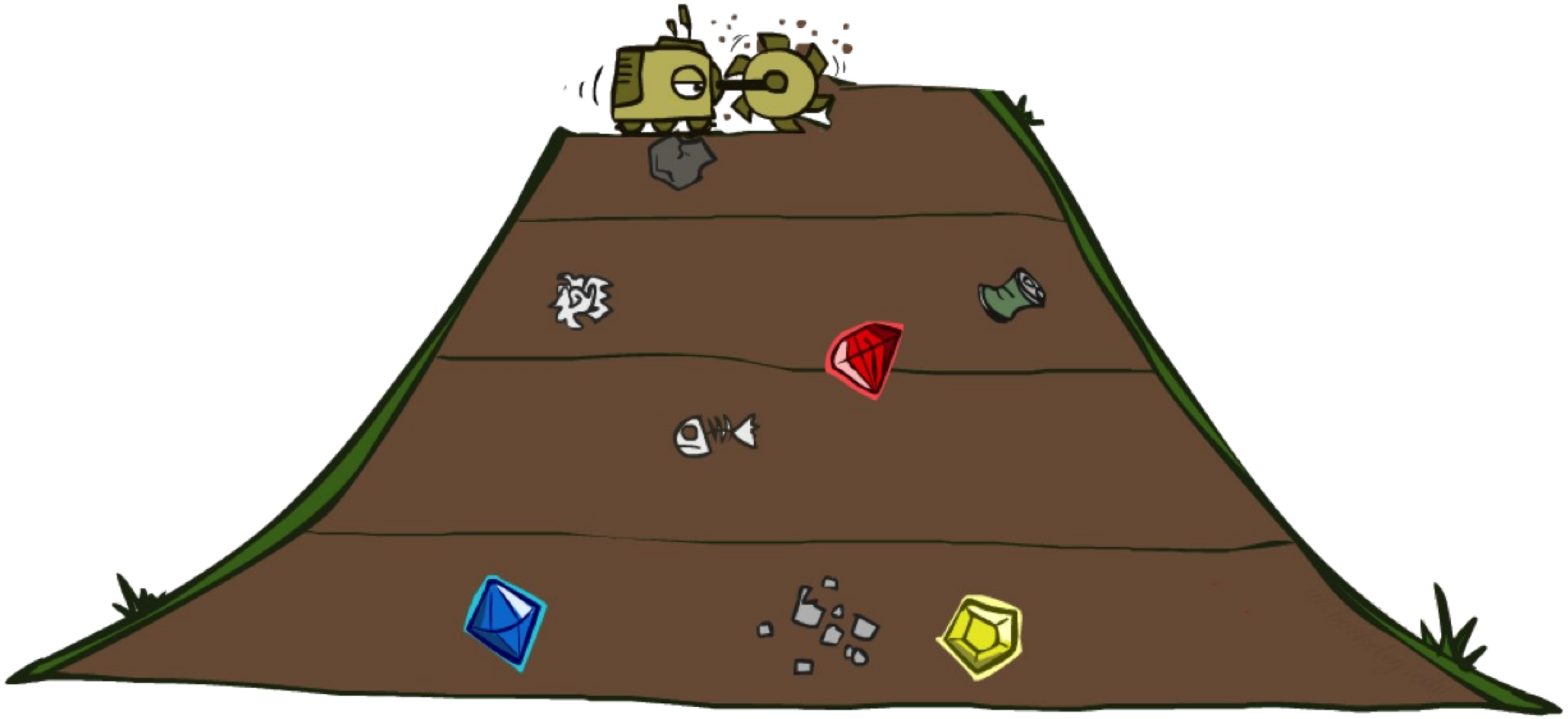
https://inst.eecs.berkeley.edu/~cs188/sp22

3.4 无信息搜索策略

- 5 种无信息搜索（盲目搜索）：
  - 宽度优先搜索 Breadth-first
  - 一致代价搜索 Uniform-cost
  - 深度优先搜索 Depth-first
  - 深度受限搜索 Depth-limited
  - 迭代加深的深度优先搜索 Iterative-deepening

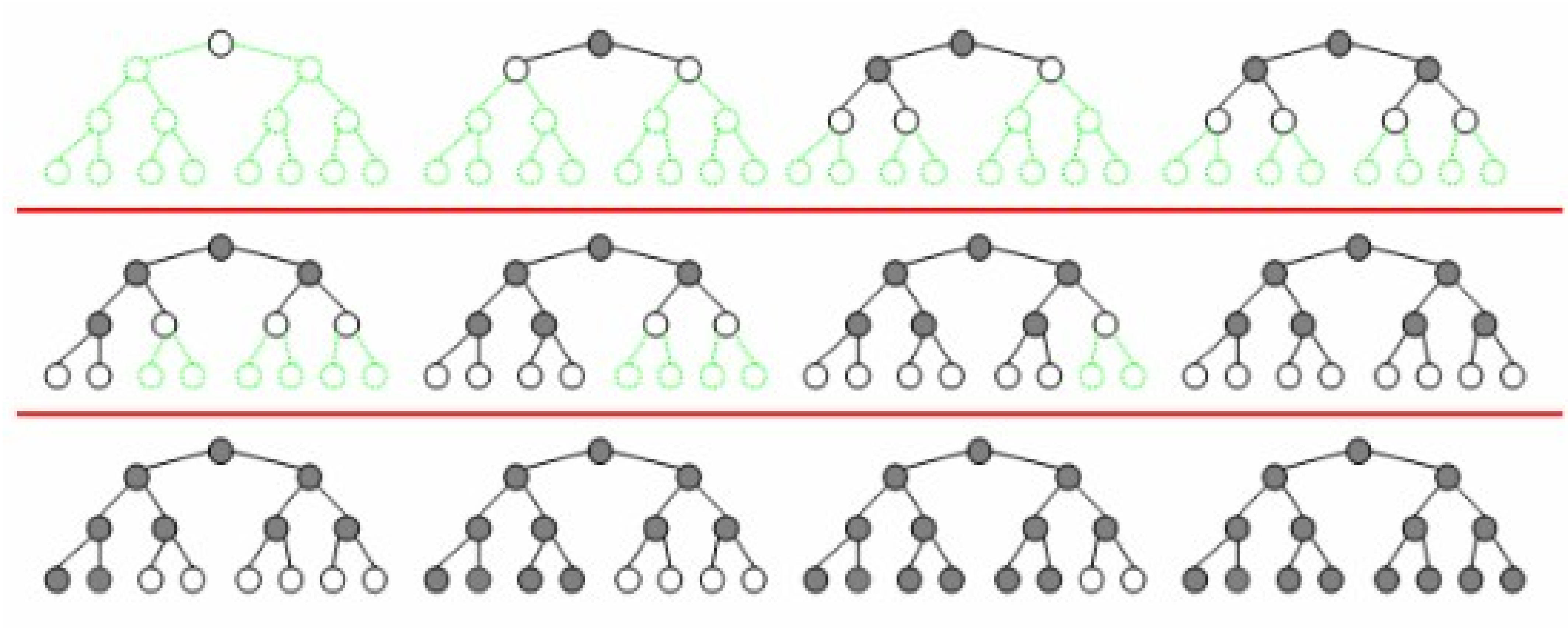


3.4.1 宽度优先搜索（ BFS)



3.4.1 宽度优先搜索（ BFS)

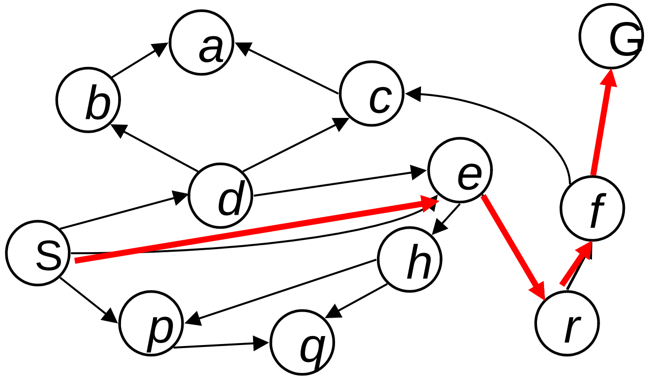
BFS 是一种简单的搜索策略。它从根节点开始，按搜索深度逐层扩展结点，直到找到目标结点为止，其搜索顺序如下图所示：



宽度优先搜索（ BFS)

搜索策略：先扩展深度最浅的结点

实现： 边缘是 FIFO 队列



FIFO 队列：

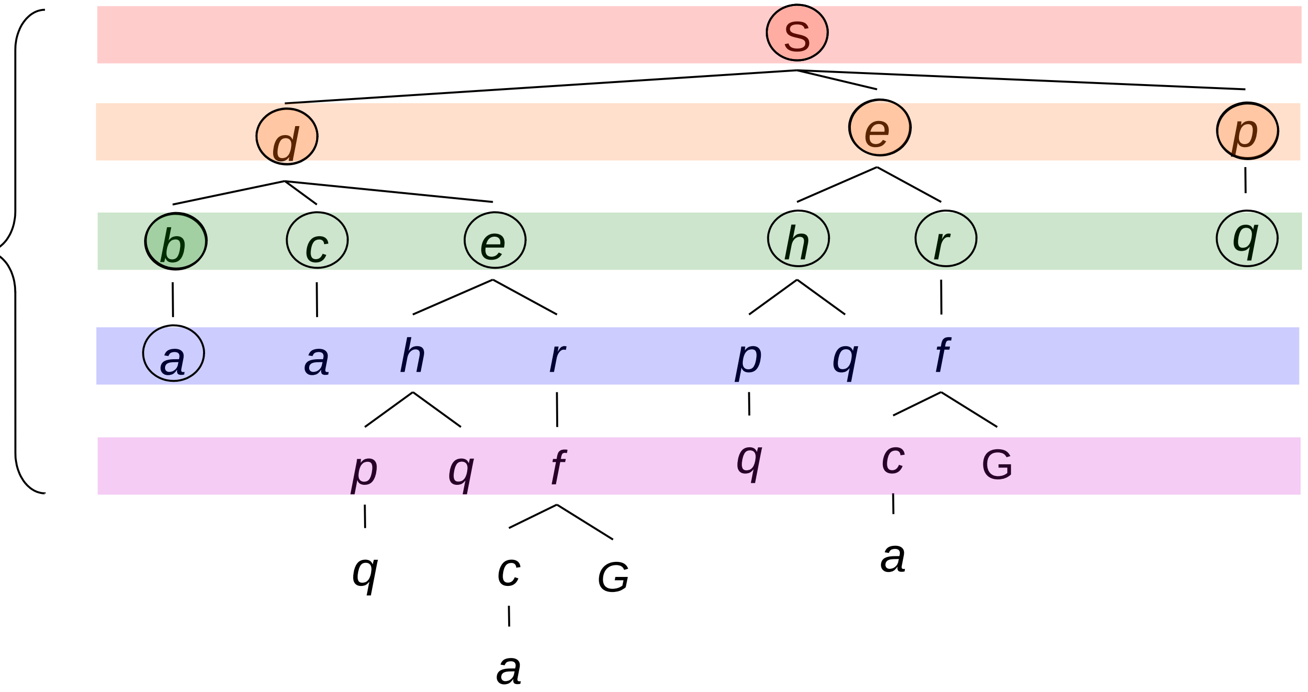
先进先出队列；

新结点加入到队尾，浅层的老结点会在深层的新结点之前被扩展

边缘队列：

S
d e p
e p b c e
p b c e h r
b c e h r q
c e h r q a
…

Search Tiers

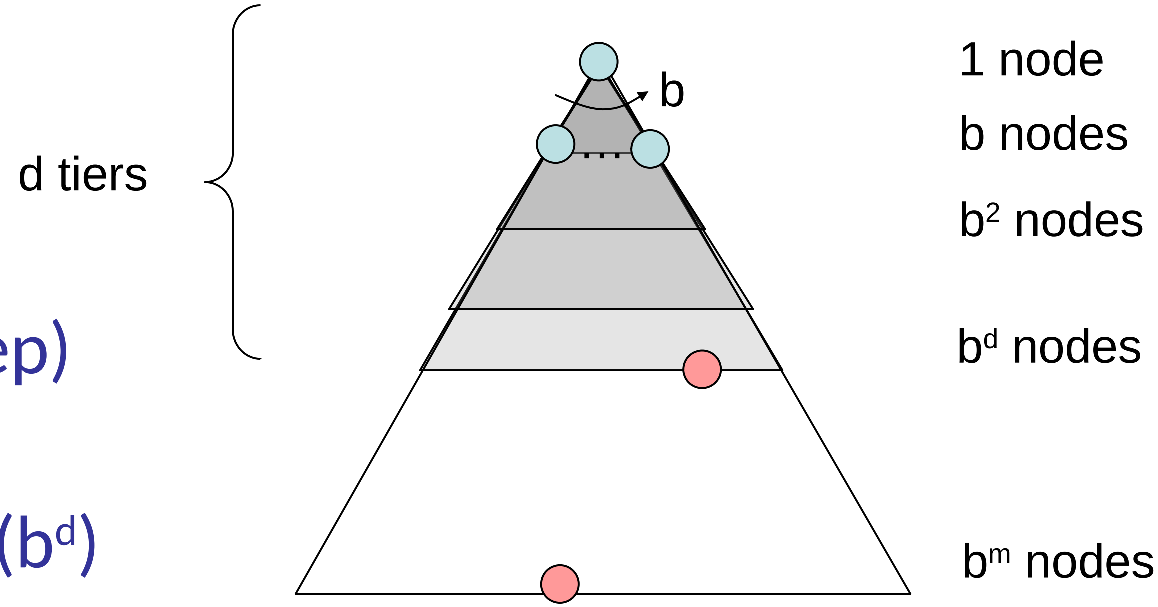


搜索序列：S d e p b c…p q f q c G

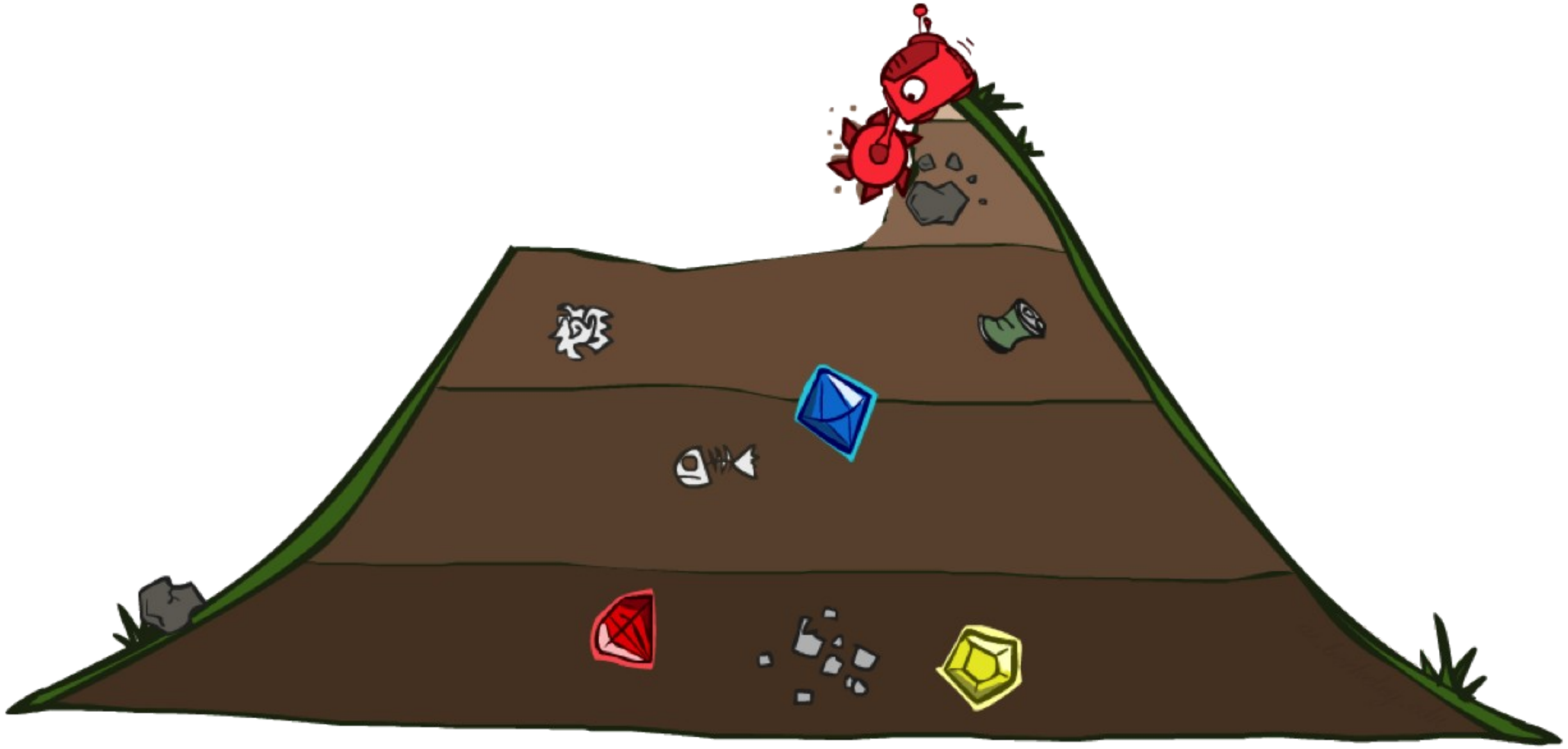
解序列：S e r f G

宽度优先搜索（ BFS ）的性能

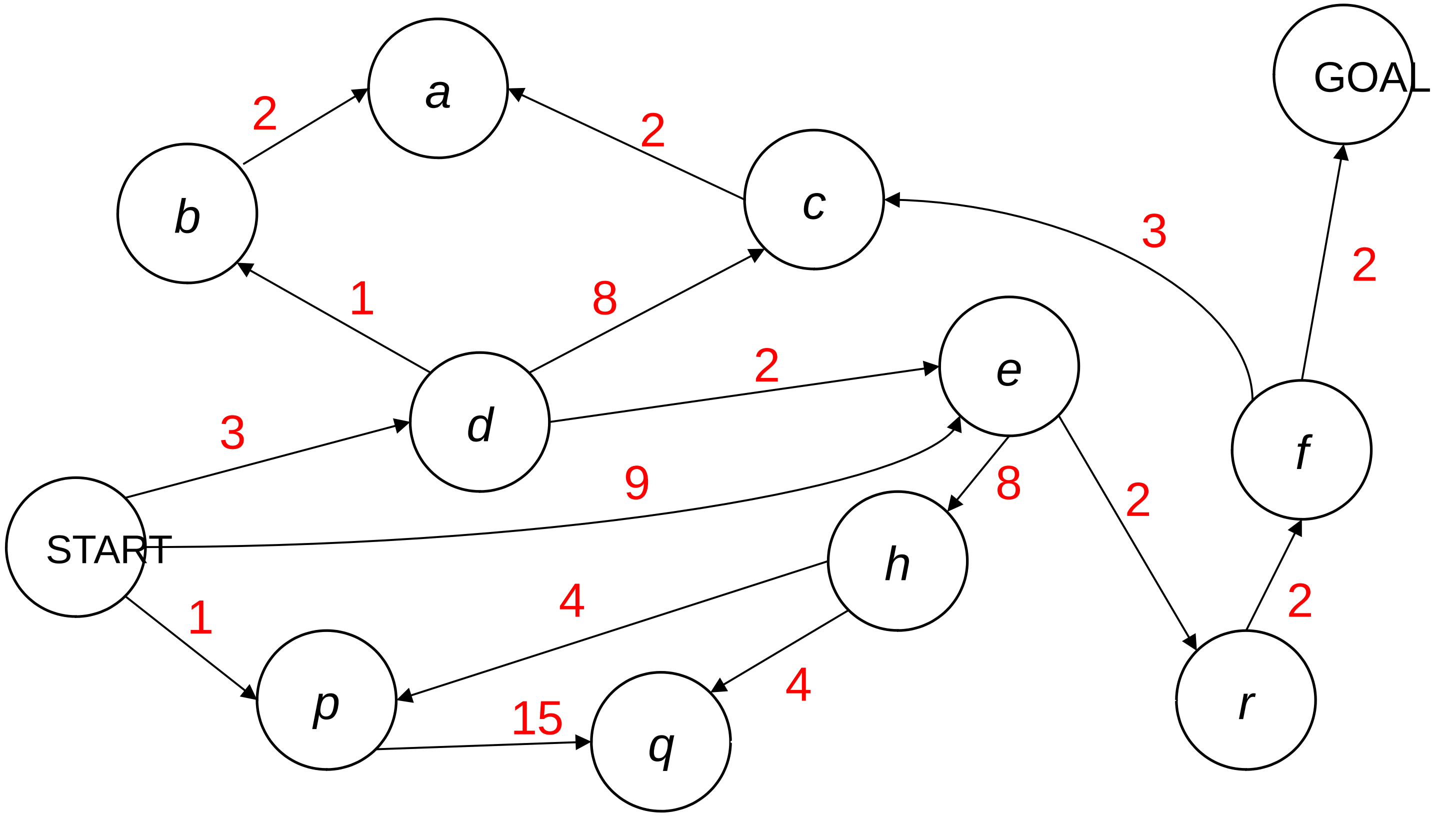
- 完备性？Yes (if *b&d* are finite)
- 最优性？Yes (only if cost = 1 per step)
- 时间复杂度？$1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- 空间复杂度？$O(b^d)$ (keeps every node in memory)



3.4.2 一致代价搜索



3.4.2 一致代价搜索



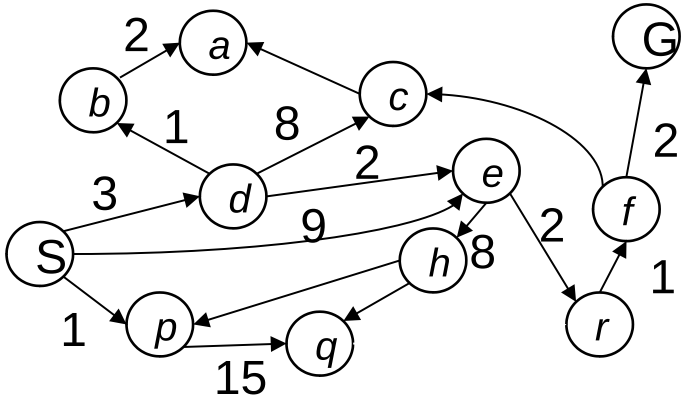
状态空间图（边上有代价值）

一致代价搜索

代价函数： *g(n) = cost from root to state n*

搜索策略：先扩展 *g(n)* 最小的结点

实现： 边缘是优先队列（按照 *g(n)* 从小到大排序）



边缘队列

S(0)

p(1) d(3) e(9)

d(3) e(9) q(16)

b(4) e(5) e(9) c(11) q(16)

e(5) a(6) e(9) c(11) q(16)

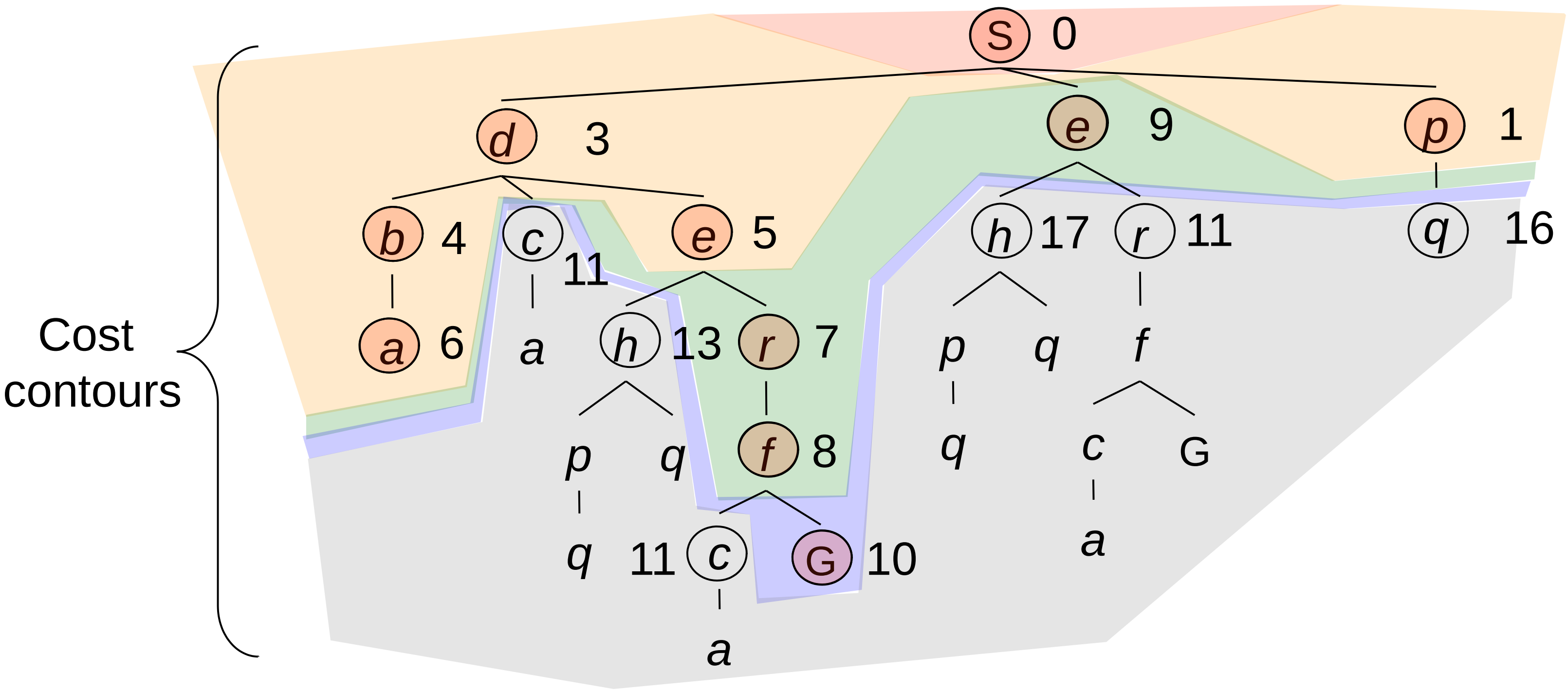
a(6) r(7) e(9) c(11) h(13) q(16)

r(7) e(9) c(11) h(13) q(16)

f(8) e(9) c(11) h(13) q(16)

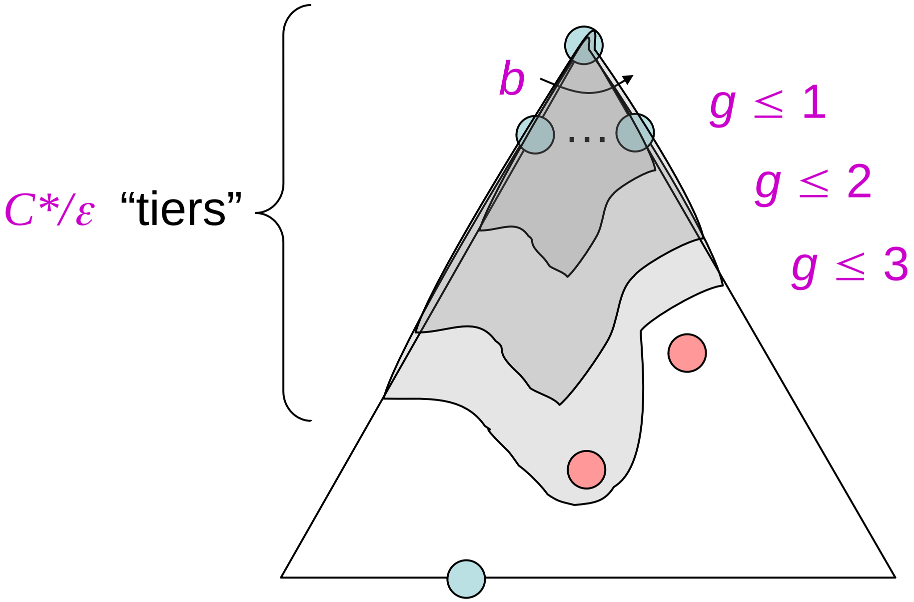
e(9) G(10) c(11) h(13) q(16)

G(10) c(11) r(11) h(13) q(16) h(17)



一致代价搜索（UCS) 的性能

- UCS 扩展了哪些结点？
  - 假定最优解的代价是 C^* ，每一步的代价至少是 □
  - “effective depth” is roughly C^*/ε
  - 扩展了 cost< C^* 的所有结点！
  - 时间复杂度？ $O(b^{C^*/\varepsilon})$



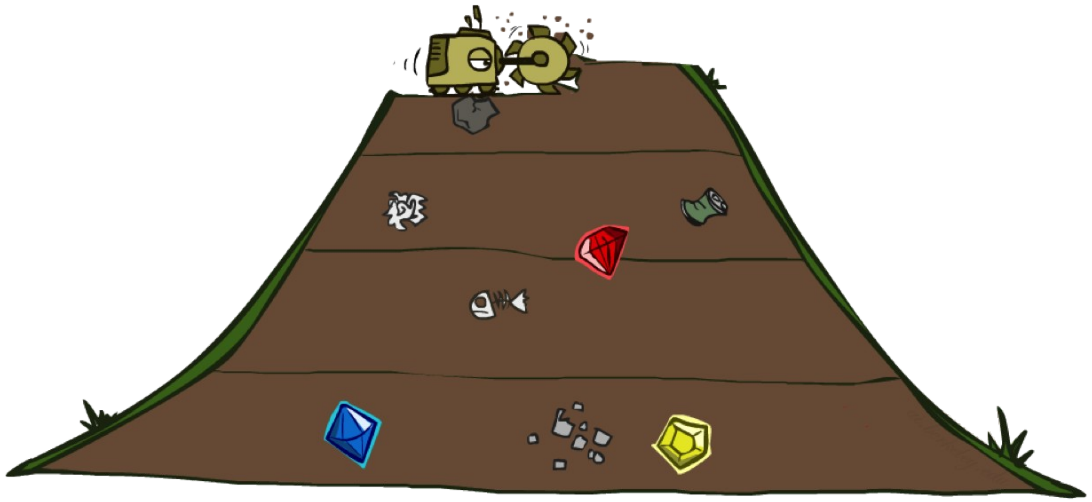
- 空间复杂度？ $O(b^{C^*/\varepsilon})$

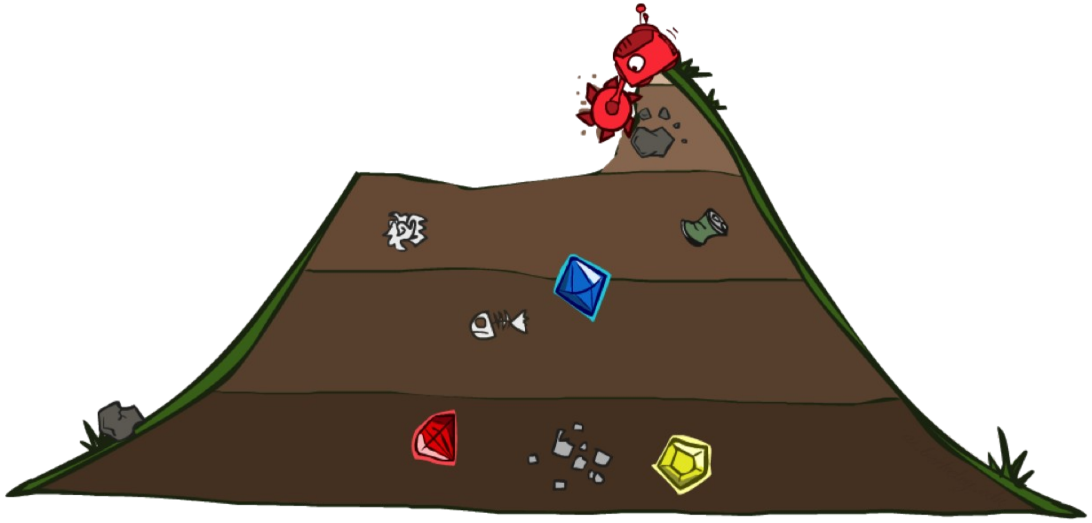
- 完备性？Yes, Assuming C^* is finite and $\varepsilon > 0$

- 最优性？Yes

宽度优先搜索 vs. 一致代价搜索

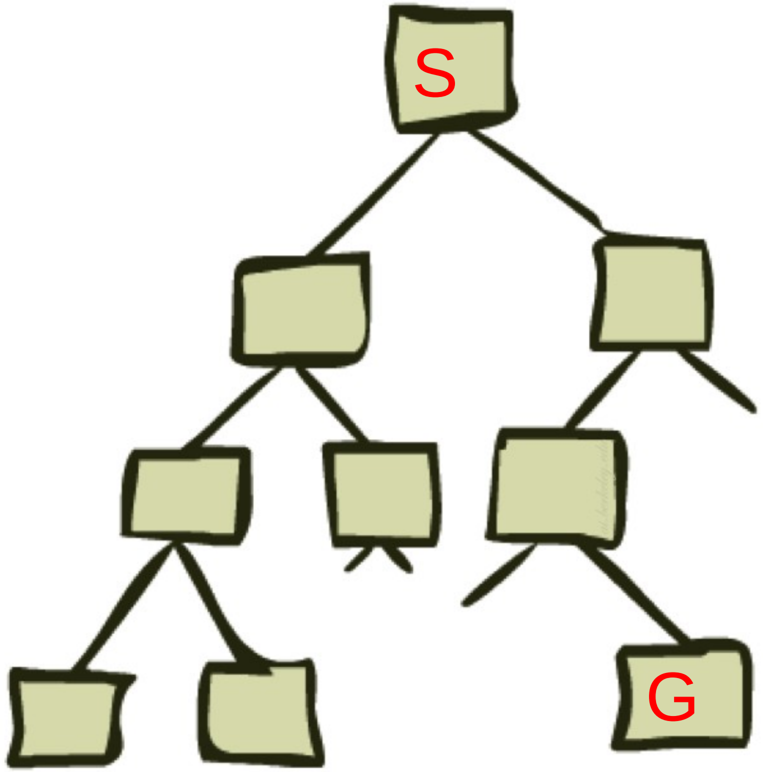
	宽度优先搜索	一致代价搜索
边缘队列	FIFO	优先队列（路径耗散）
最优性	所有边耗散相同时	最优解





3.4 无信息搜索策略

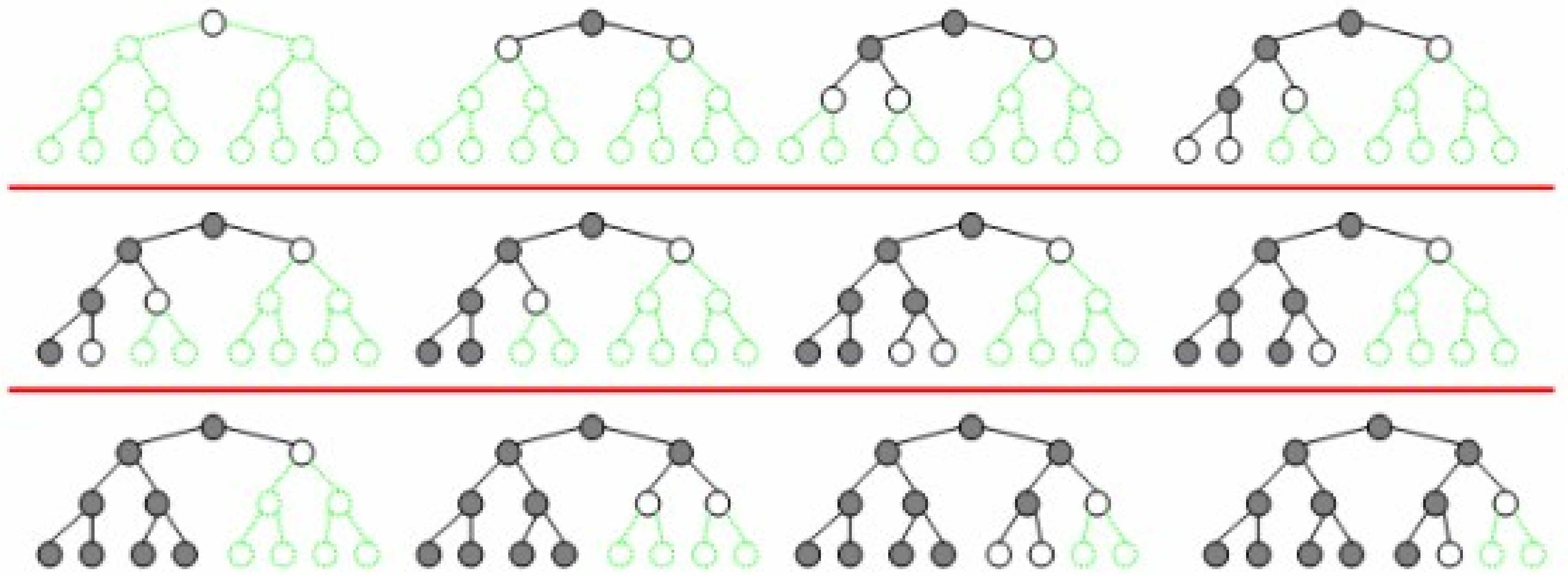
- 5 种无信息搜索（盲目搜索）：
  - 宽度优先搜索 Breadth-first
  - 一致代价搜索 Uniform-cost
  - <u>深度优先搜索</u> Depth-first
  - 深度受限搜索 Depth-limited
  - 迭代加深的深度优先搜索 Iterative-deepening



3.4.3 深度优先搜索（DFS）

深度优先搜索（ DFS ）

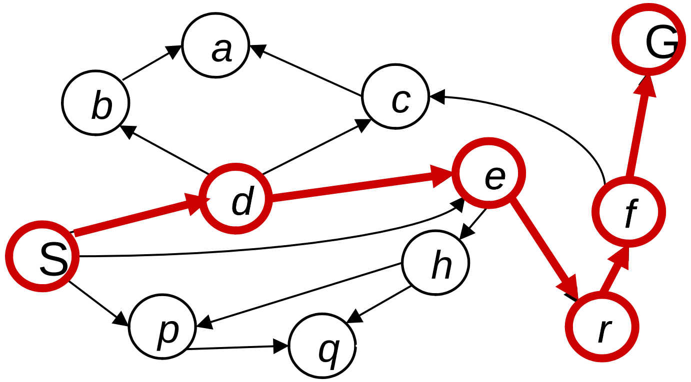
DFS 总是优先扩展当前搜索树中最深的结点。当到达搜索树中边缘无后继的结点时，该算法回溯到次深未扩展的结点继续搜索，直到搜索到目标状态为止。



深度优先搜索（DFS）

搜索策略：先扩展深度最深的结点

实现： 边缘是 LIFO 栈



LIFO 栈：

后进先出队列；

最新生成的结点最早被选择被扩展

边缘队列

S

d e p

b c e e p

a c e e p

c e e p

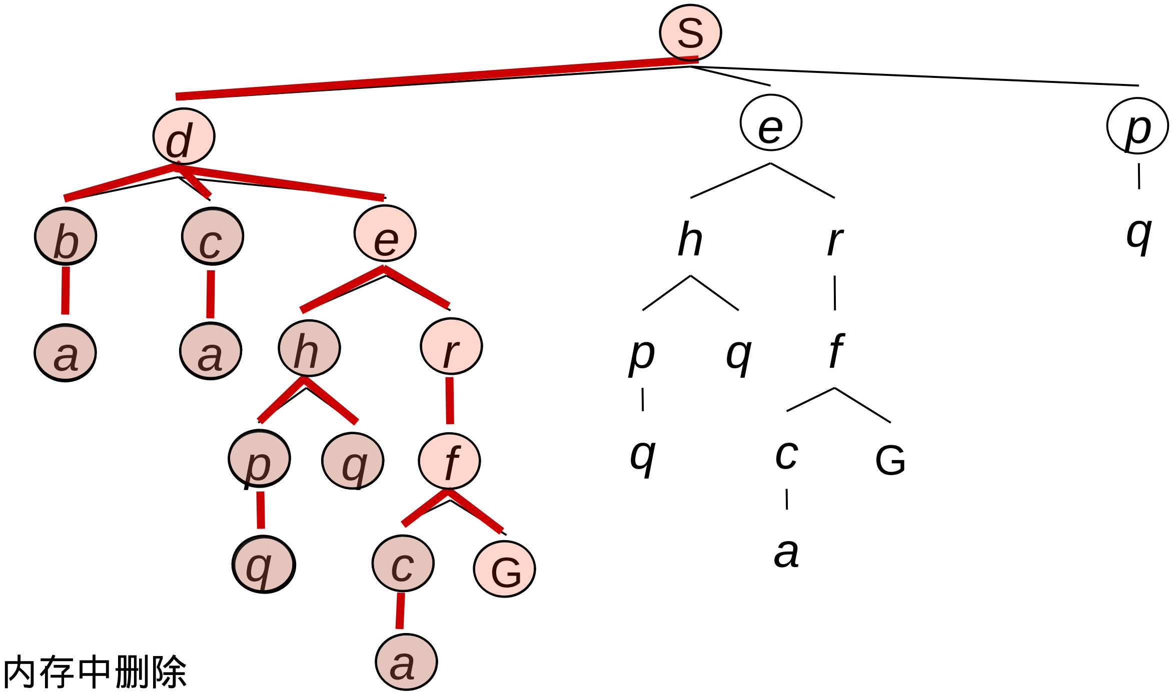
a e e p

e e p

h r e p

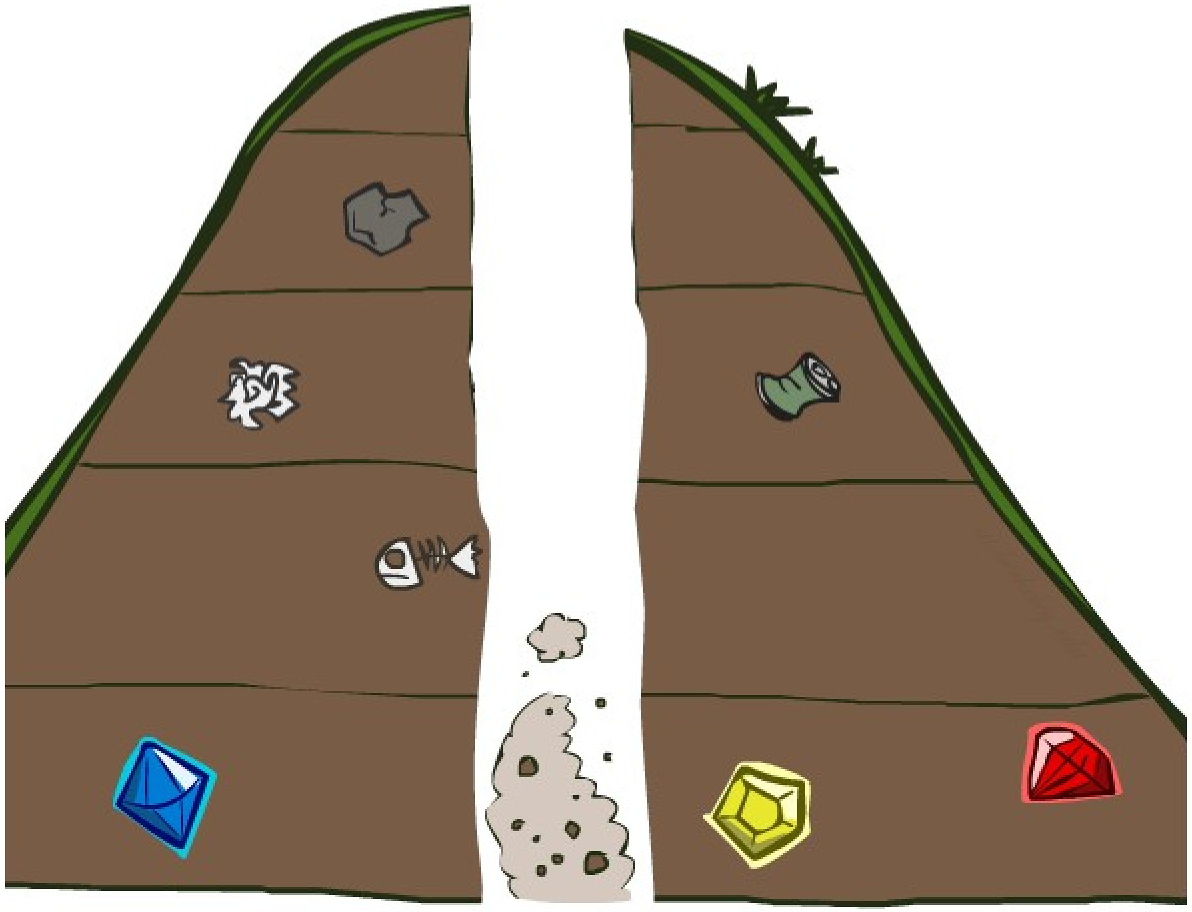
p q r e p

……



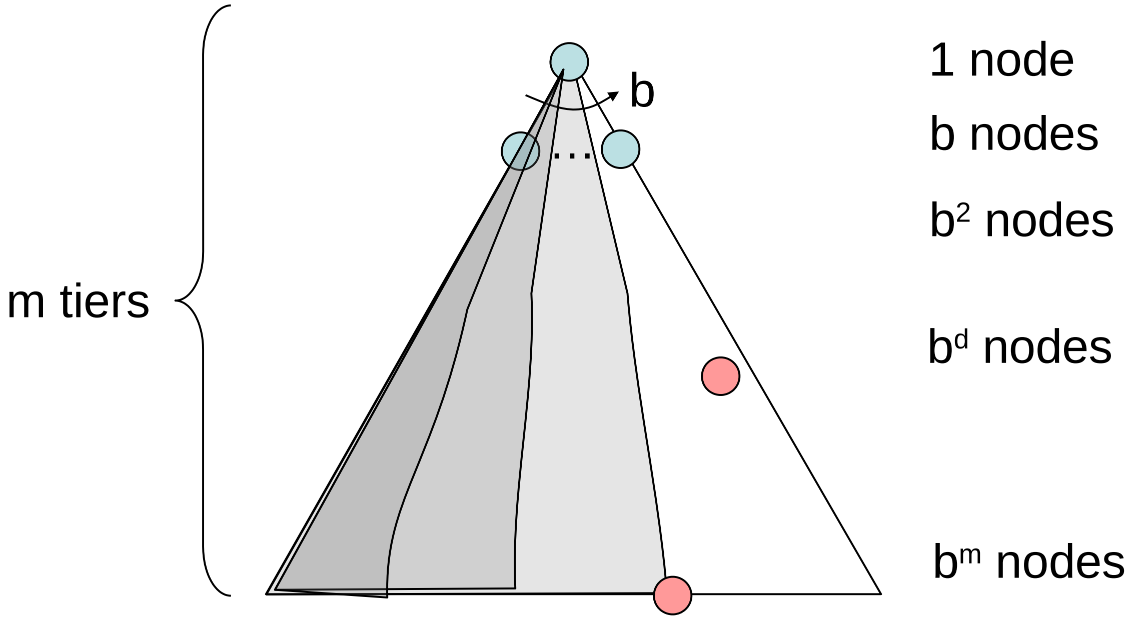
已扩展且在边缘中没有后代的结点可以从内存中删除

Search Algorithm Properties



深度优先搜索（DFS) 的性能

- 完备性？No complete in finite spaces
- 最优性？No
- 时间复杂度？$O(b^m)$
- 空间复杂度？$O(bm)$ linear space!



深度优先搜索只要存储一条从根结点到叶子结点的路径，以及该路径上每个结点所有未被扩展的兄弟结点即可。

已扩展并且在边缘中没有后代的结点可以从内存中删除

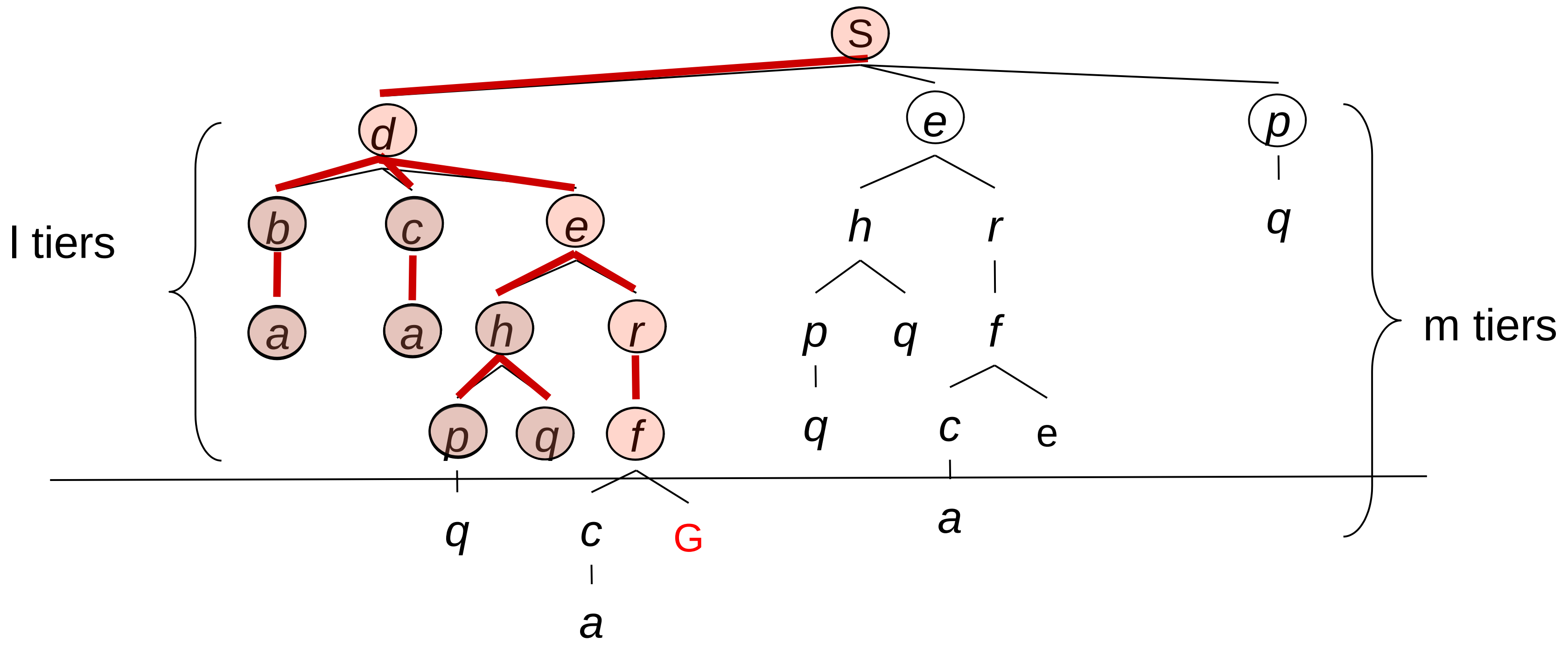
3.4.4 深度受限搜索

= DFS with depth limit *l*

- 深度为 l 的结点被当做最深层结点（没有后继结点）来对待。

- 避免 DFS 在无限状态空间下搜索失败，解决了无穷路径问题

- 算法的性能：
  - 时间复杂度 O(b^l)
  - 空间复杂度 O(bl)
  - l<m: 不是完备的
  - 不是最优的

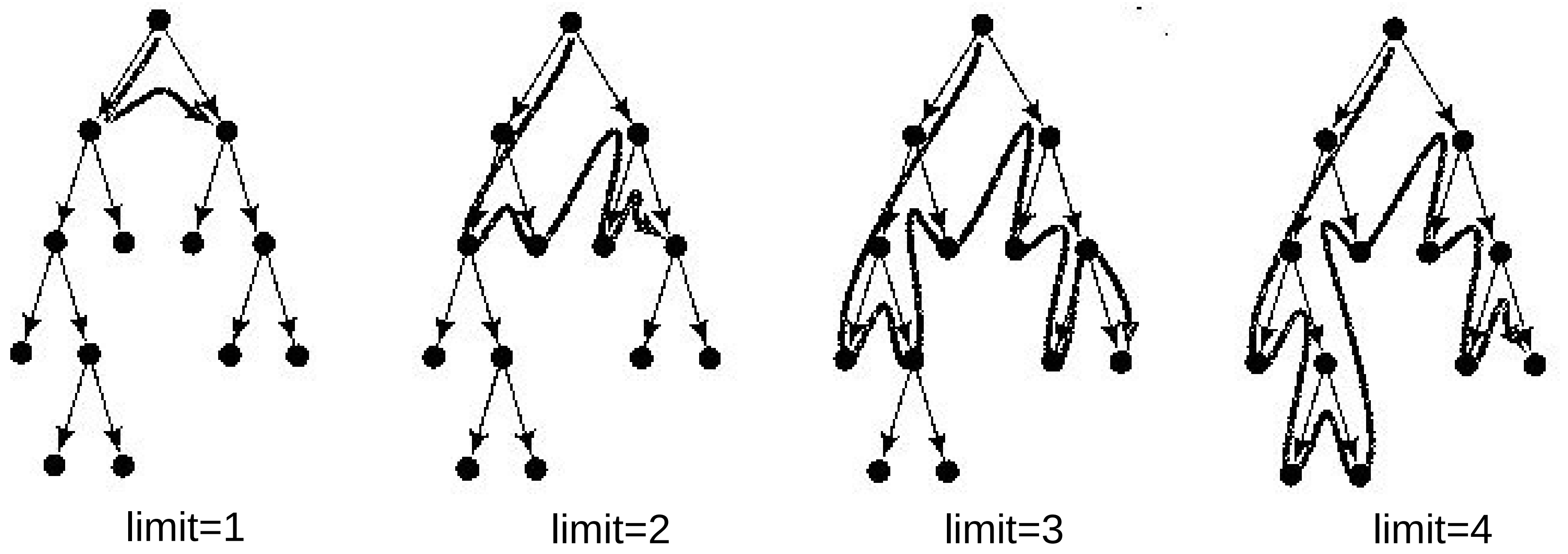


3.4.5 迭代加深的深度优先搜索（ IDS ）

- 在深度受限搜索的基础上，逐步增加深度限制。该算法结合了深度优先和广度优先的优点。

- 算法原理：设最大深度 limit ，开始 limit 设为 1 ，深度优先搜索，如果没有找到目标，则 limit 加一，再次深度优先搜索，以此类推直到找到目标。

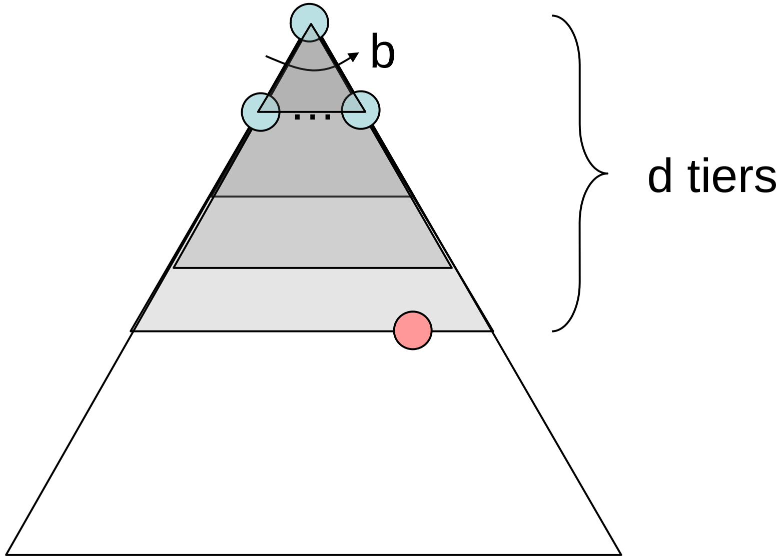
3.4.5 迭代加深的深度优先搜索（ IDS ）



优势：既可以避免陷入深度无限的分支，同时还可以找到深度最浅的目标解，从而在每一步代价一致的时候找到最优解，再加上其优越的空间复杂度，常常作为首选的无信息搜索策略。

迭代加深的深度优先搜索（ IDS ）

- IDS = DFS + BFS
- 完备性？Yes （分支因子 b 有限时）
- 最优性？Yes, if step cost = 1
- 时间复杂度？$O(b^d)$
- 空间复杂度？$O(bd)$



Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes [a]	Yes [a,b]	No	No	Yes [a]
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes [c]	Yes	No	No	Yes [c]

搜索策略比较：

b 指分支因子，*d* 指最浅解的深度，*m* 指搜索树的最大深度，*l* 是深度界限。

右上角标的含义： a 指当 *b* 有限时算法是完备的，b 指若对正数□有单步代价□□ □，则是完备的。 *c* 单步代价相同时算法最优。