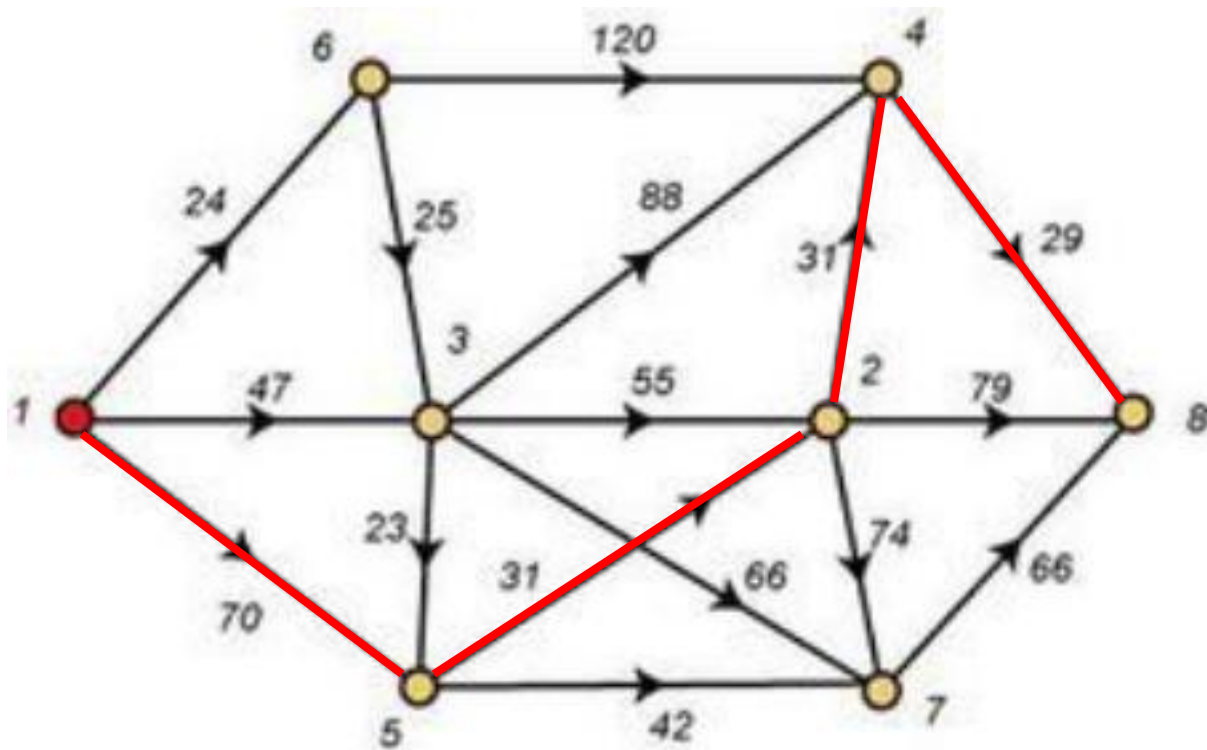


搜索问题

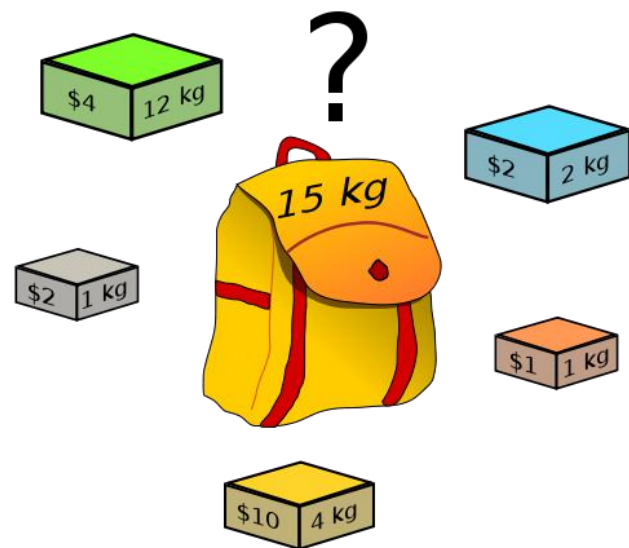
无信息搜索/启发式搜索

问题0 最短路径



问题1 0/1背包

- ◆ 有一背包，最多可承重 W
- ◆ 有 n 个货物
 - ◆ 各自的重量分别为 w_0, \dots, w_{n-1}
 - ◆ 各自的价值分别为 v_0, \dots, v_{n-1}
- ◆ 选择哪些货物放入背包，在不超重的情况下，使得背包内货物总价值最大？



问题2 青蛙跳河



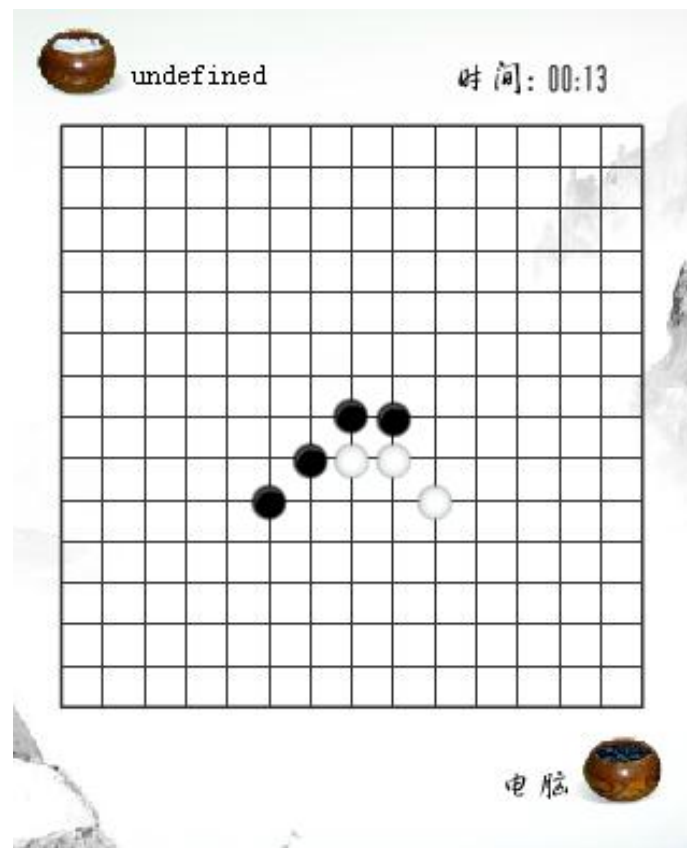
目标：请让左右两边河墩上的青蛙交换位置。

规则：

1. 前面的一个河墩为空，青蛙可直接跳过去；
2. 前面河墩隔一个是空的，不论隔的是哪种颜色的青蛙，可从它头上跳过去到达空位。

问题3 五子棋

- 设计一个人机对抗的五子棋游戏



- ◇ 旅行推销员问题
- ◇ 骑士聚会问题
- ◇ 象棋
- ◇ 围棋
- ◇ 各种电脑游戏

有没有通用的
解决办法？

搜索问题

- ◆ 搜索概述

- ◆ 盲搜

 - ◆ 深度优先，广度优先等

- ◆ “不盲的”搜

 - ◆ 爬山算法，遗传算法等

- ◆ 对抗搜索

 - ◆ 游戏

搜索

- ◆ 不是通过对问题的分析找出问题的计算公式进而计算，而是根据问题的性质，找出问题的状态模型，确定包括目标状态在内的各种状态的生成方法，按一定的规律逐一生成各种可能的状态，从中寻找符合要求的目标状态
 - ◆ 程序设计中一种常用策略
 - ◆ 求解难以直接计算的（组合优化）问题
 - ◆ 适应广泛

搜索：模型定义

- ◇ 搜索问题一般通过以下几个属性来定义：
 - ◇ 状态
 - ◇ 初始状态
 - ◇ 动作
 - ◇ 转移模型
 - ◇ 目标测试
 - ◇ 路径成本（可选）

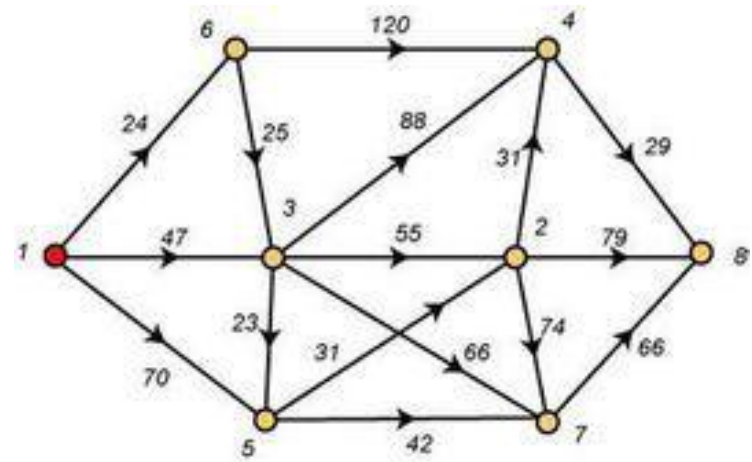


思考：

$s = \langle 1, 3, 4 \rangle$

接收到动作 $\text{move}(4, 2)$,

$s' = ?$



- ◆ **状态**：一个节点序列, 如 $\langle v_1, v_2, v_3 \rangle$, 表示经过的路径；序列最后一个节点表示当前的位置。
- ◆ **初始状态**：序列只有一个节点，表示出发位置。
- ◆ **动作**： $\text{move}(v_i, v_j)$ ，从一个节点移动到另一个节点。图中须有一条边从 v_i 指向 v_j 。
- ◆ **转移模型**：状态 s , 接受到动作 $\text{move}(v_i, v_j)$ ，若 s 的最后一个节点等于 v_i , 则转移到状态 s' , s' 是在 s 的末尾加上 v_j ；否则不发生状态改变。
- ◆ **目标测试**：有一个目标节点 v_e , 一个状态是目标状态当且仅当其末尾的节点等于 v_e 。
- ◆ **路径成本**：一个状态的路径成本等于其序列中每两个前后相邻的节点路径之和。



思考：

$s = \langle 0, 1, 2, 3 \rangle$

接受到动作 $\text{set}(1, 2)$ ， $s' = ?$ s' 是目标吗？

Q			
	Q		
		Q	
			Q

- ◆ **状态：** 一个四维向量 $\langle x_1, x_2, x_3, x_4 \rangle$ ，表示第 i 行皇后放在哪一列。向量中每项取值范围是 $[-1, 3]$ ，其中 -1 表示该行没有皇后。比如上图的状态可描述为 $\langle 0, 1, 2, 3 \rangle$ 。
- ◆ **初始状态：** 可以是任意一个状态。
- ◆ **动作：** $\text{set}(i, j)$ ，将第 i 行的皇后放置在第 j 列。
- ◆ **转移模型：** 状态 s 接收到动作 $\text{set}(i, j)$ ，则将 s 中第 i 项的值修改为 j ，由此生成一个新的状态 s'
- ◆ **目标测试：** 棋盘上有 4 个皇后，且所有皇后彼此不能攻击。

状态模型
7维向量 $\langle x_0, \dots \rangle$
每一项取值 0, 1, -1
1 绿色
0 棕色
-1 空位

初始状态 $\langle 1, 1, 1, -1, 0, 0, 0 \rangle$

目标
 $\langle 000-1111 \rangle$

动作
 $\text{jump}(\text{crntPos}, \text{nextPos})$

练习

状态、初始状态、动
测试。

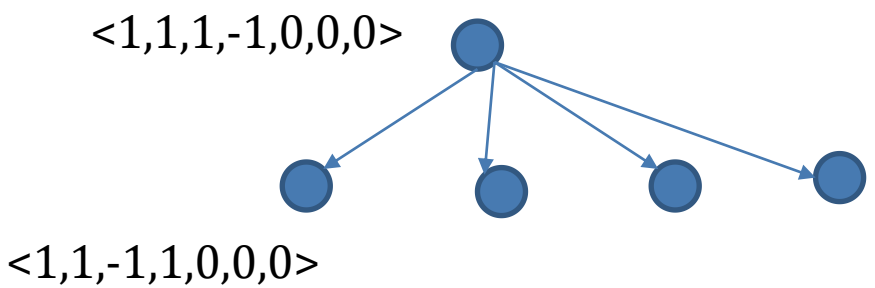


目标：请让左右两边河墩上的青蛙交换位置。

规则：

1. 前面的一个河墩为空，青蛙可直接跳过去；
2. 前面河墩隔一个是空的，不论隔的是哪种颜色的青蛙，可从它头上跳过去到达空位。

状态模型
7维向量 $\langle x_0, \dots \rangle$
每一项取值 0, 1, -1
1 绿色
0 棕色
-1 空位



初始状态 $\langle 1,1,1,-1,0,0,0 \rangle$

目标
 $\langle 000-1111 \rangle$

动作
 $\text{jump}(\text{crntPos}, \text{nextPos})$

搜索的过程

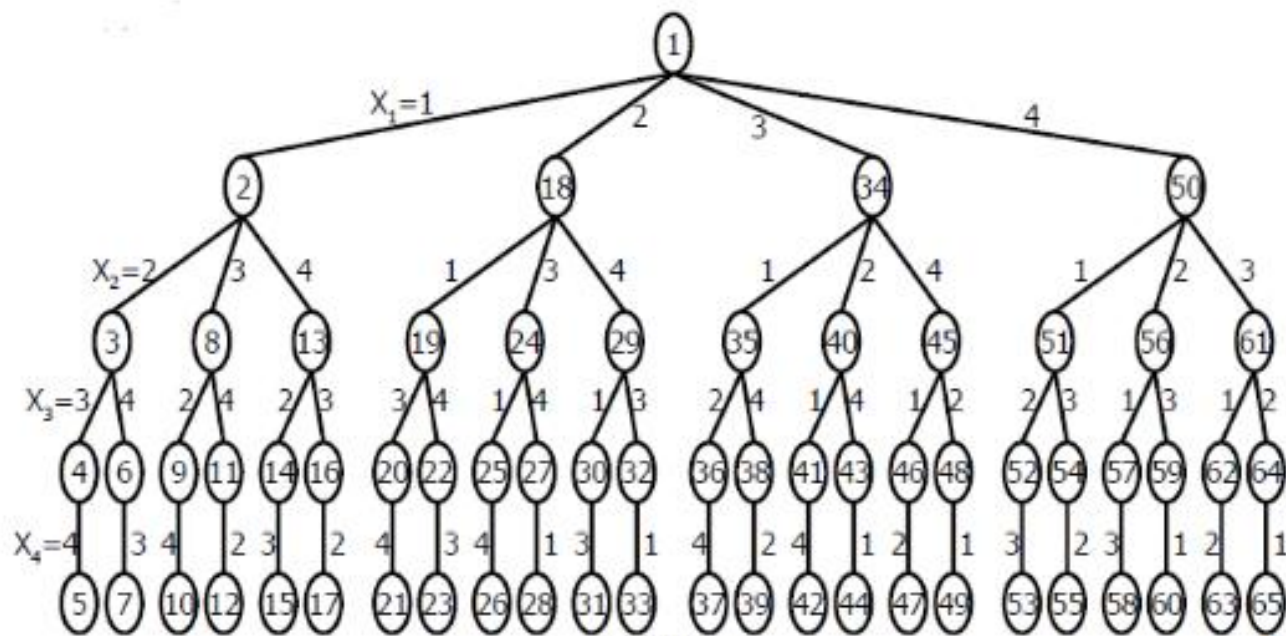
◇ 搜索基本过程

- ◇ 一个问题所有可能的状态构成了该问题的状态空间(离散)
- ◇ 每一个具体的状态是该空间的一个节点
- ◇ 搜索过程可以看成对由表示状态的节点和表示状态转移（动作）的弧所构成的有向图遍历

◇ 搜索方案 (solution)

- ◇ 从初始状态到目标状态的一个动作序列

4皇后问题的状态空间 (4叉树)



注意：该状态空间已经去除了部分非目标状态，
比如同一行只能有一个皇后。



树搜索(Tree-Search)

```
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
```

基本思路：通过生成已有状态的后继状态来遍历状态空间，寻找解决方案（solution）



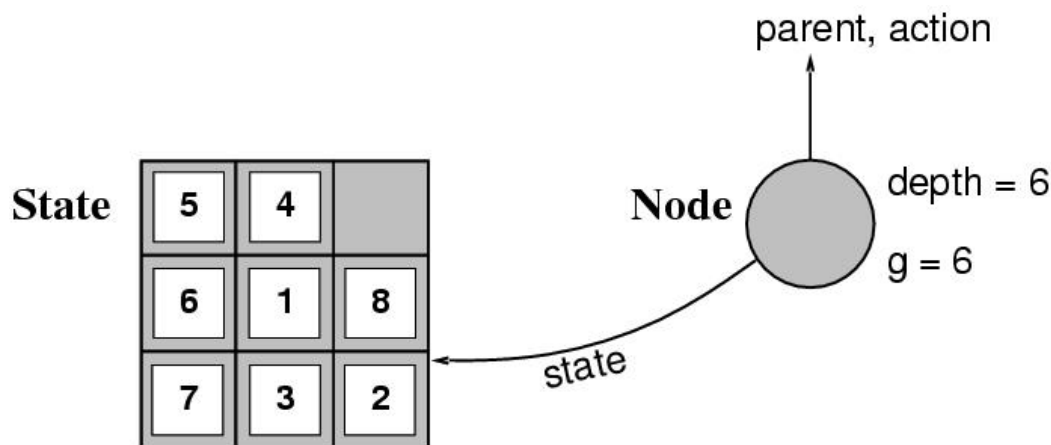
树搜索的通用框架

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

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

实现细节: states vs. nodes

- ◆ **state** 是对物理状况的一个描述
- ◆ **node** 是一个数据结构，构成搜索树的基本单元，可能包括：
 - ◆ state, parent node, action, path cost, depth



盲搜

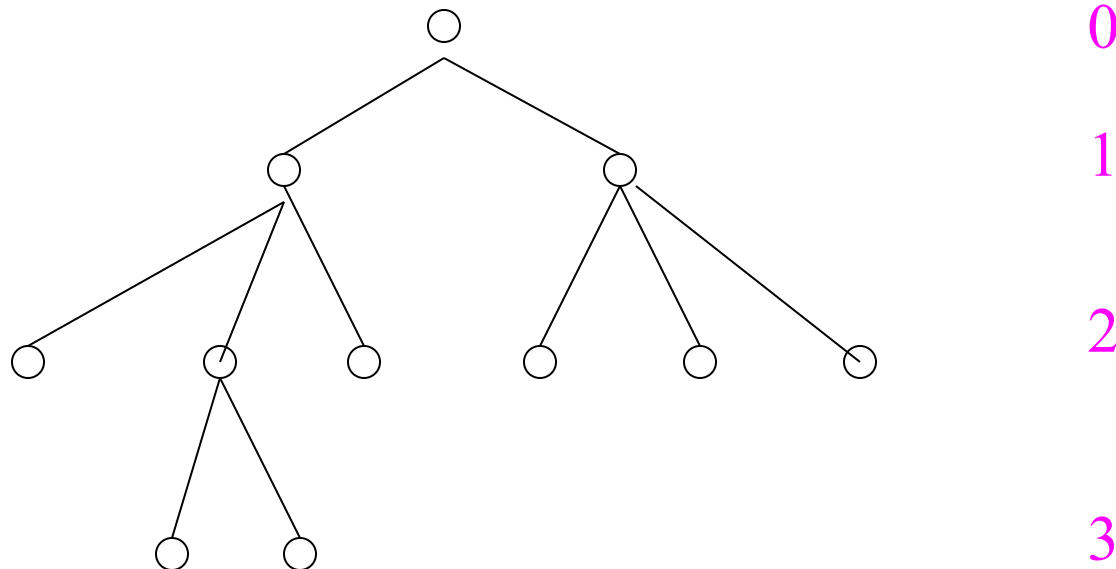
- ◆ 概念：blind search or uninformed search
 - ◆ 除问题定义之外，搜索策略不能获得状态的其他信息，只能通过产生新的状态并检测目标状态来解决问题。
- ◆ 策略分类
 - ◆ 根据状态空间中节点展开的顺序来对盲搜策略分类
 - ◆ 广度优先搜索，深度优先搜索，有深度限制的搜索，一致代价搜索，迭代扩展深优搜索，双向搜索

一些基本概念

◆ 节点深度:

根节点深度=0

其它节点深度=父节点深度+1



一些基本概念 (续1)

◇ 路径

设一节点序列为 (n_0, n_1, \dots, n_k) , 对于 $i=1, \dots, k$, 若节点 n_{i-1} 具有一个后继节点 n_i , 则该序列称为从 n_0 到 n_k 的路径。

◇ 路径的耗散值

一条路径的耗散值等于连接这条路径各节点间所有耗散值的总和。用 $C(n_i, n_j)$ 表示从 n_i 到 n_j 的路径的耗散值。

一些基本概念（续1）

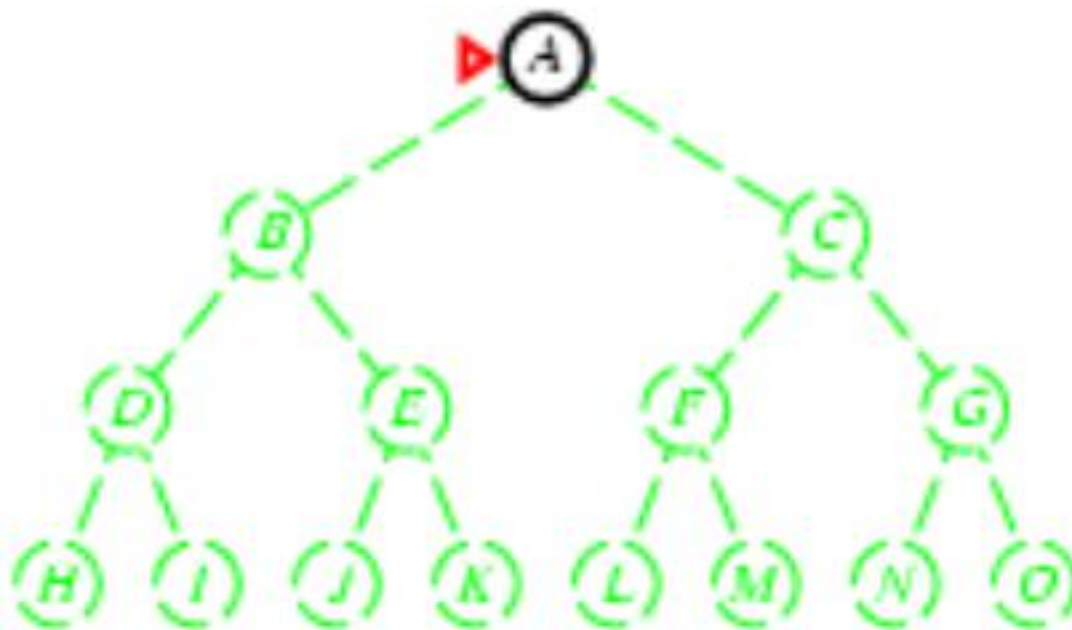
◇ 扩展一个节点

生成出该节点的所有后继节点，并给出它们之间的耗散值。这一过程称为“扩展一个节点”。

深度优先搜索

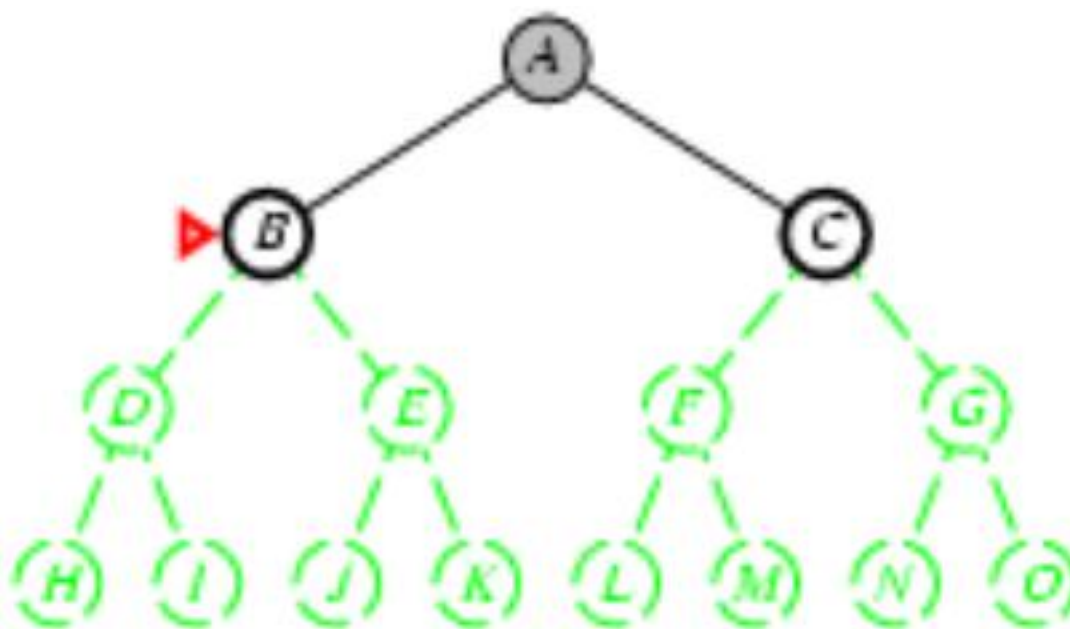
Depth-first search (DFS)

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



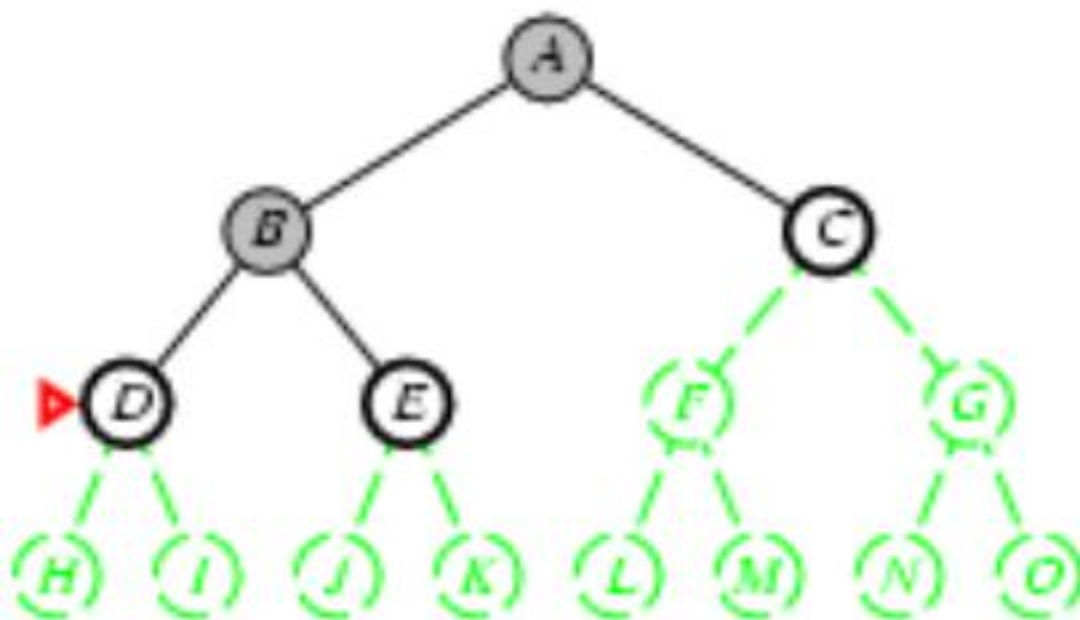
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



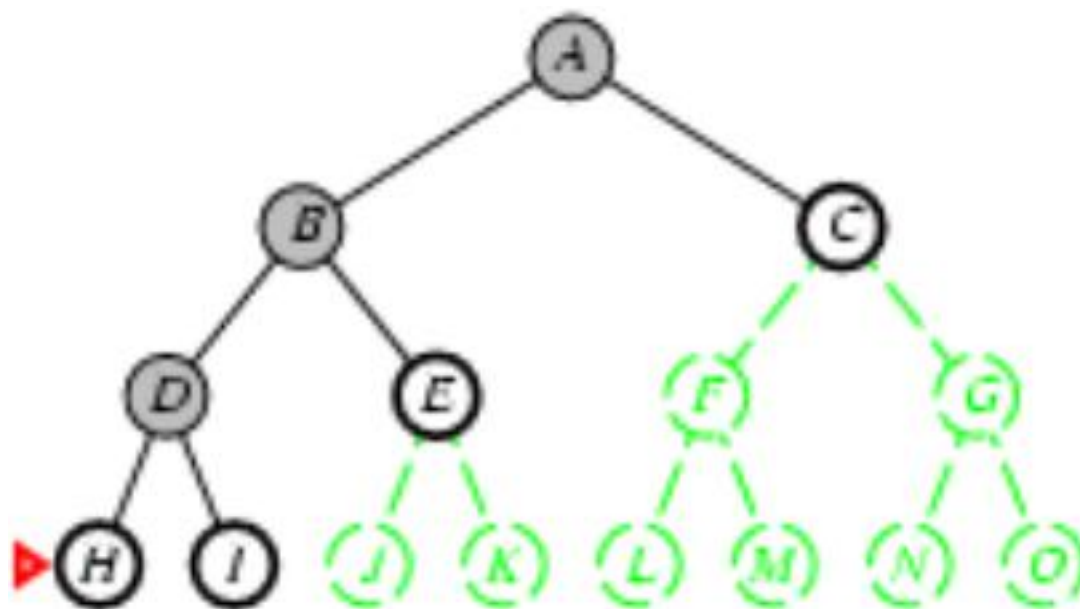
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



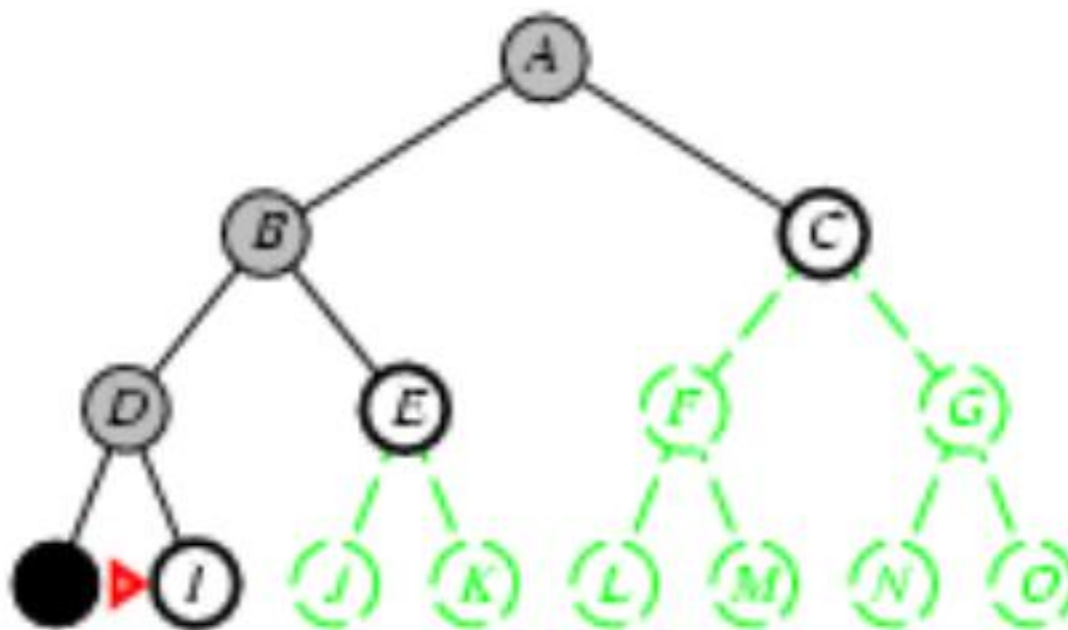
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



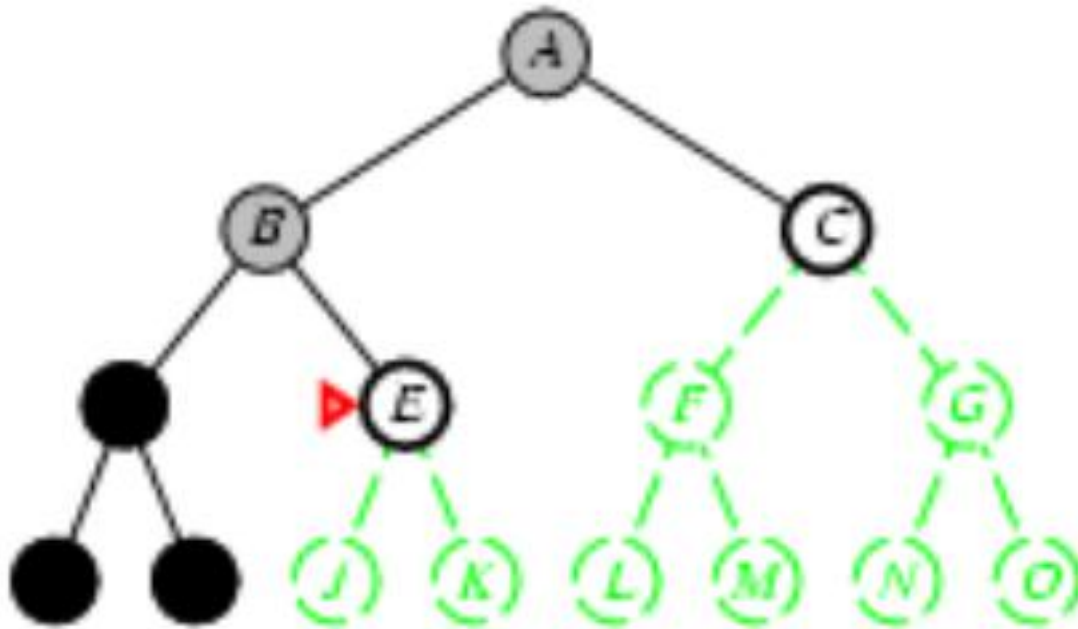
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



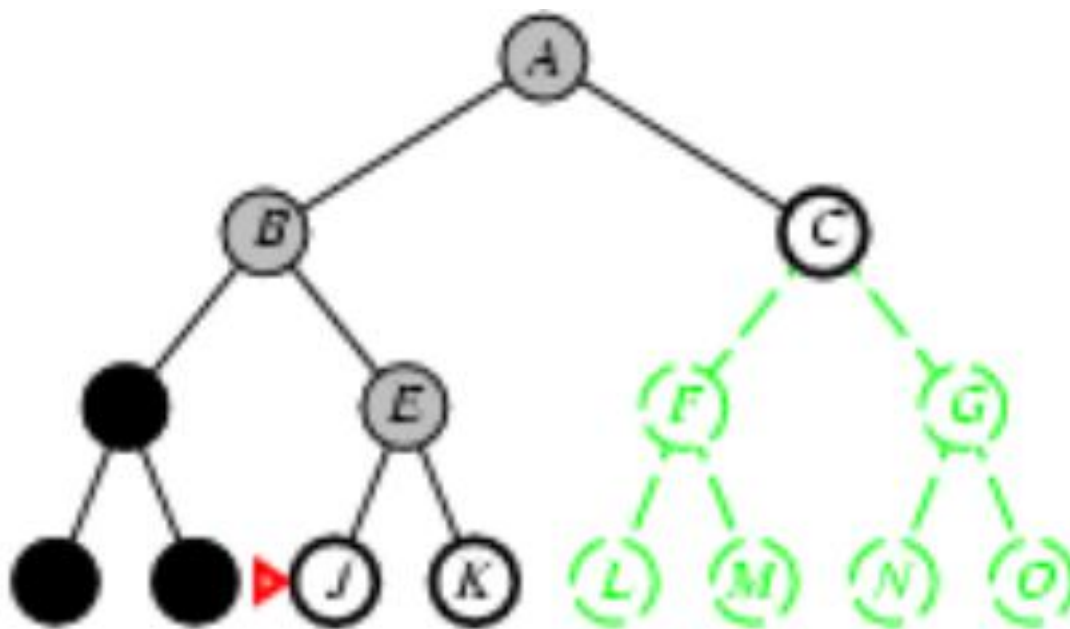
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



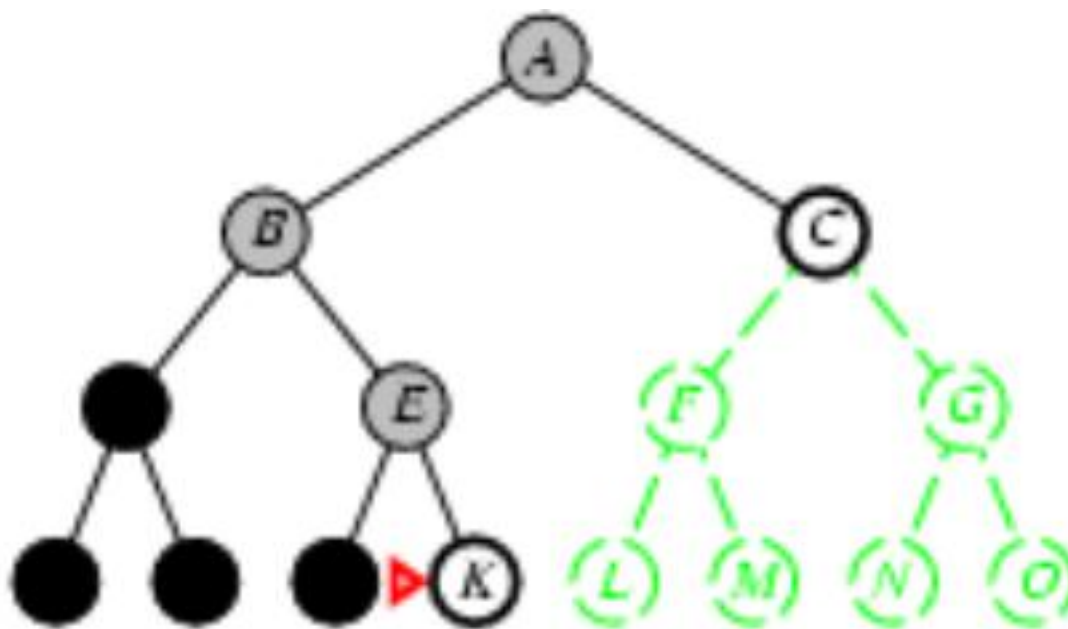
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



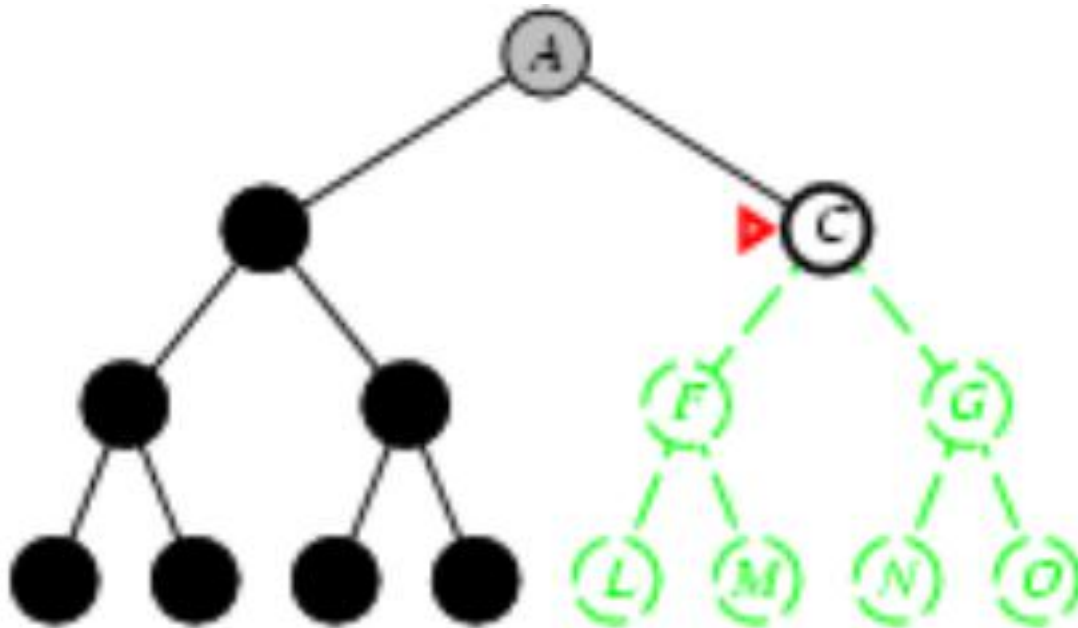
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



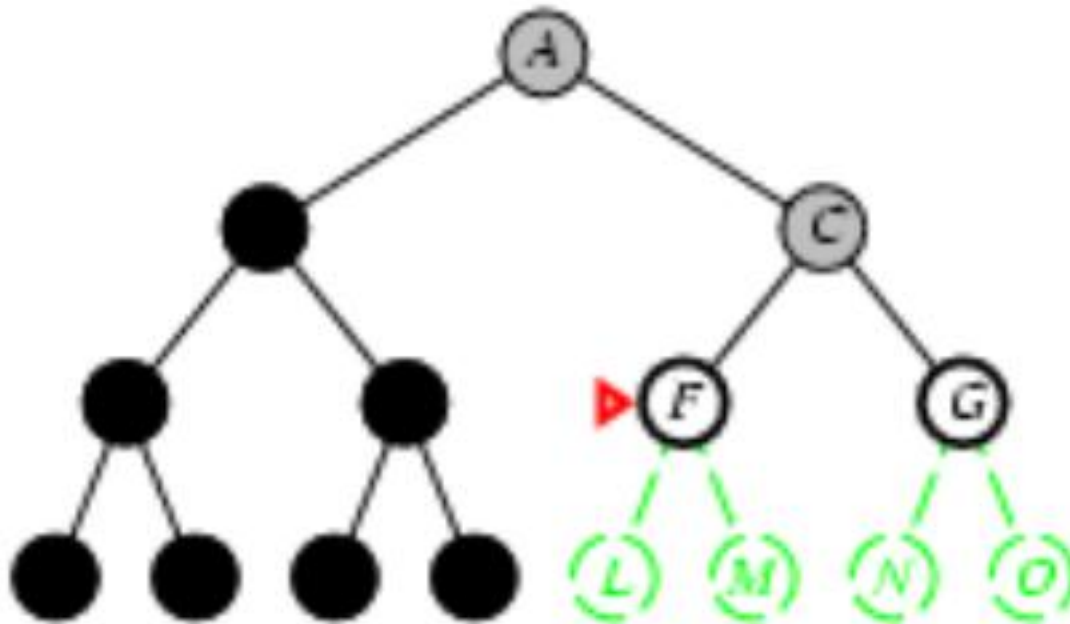
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



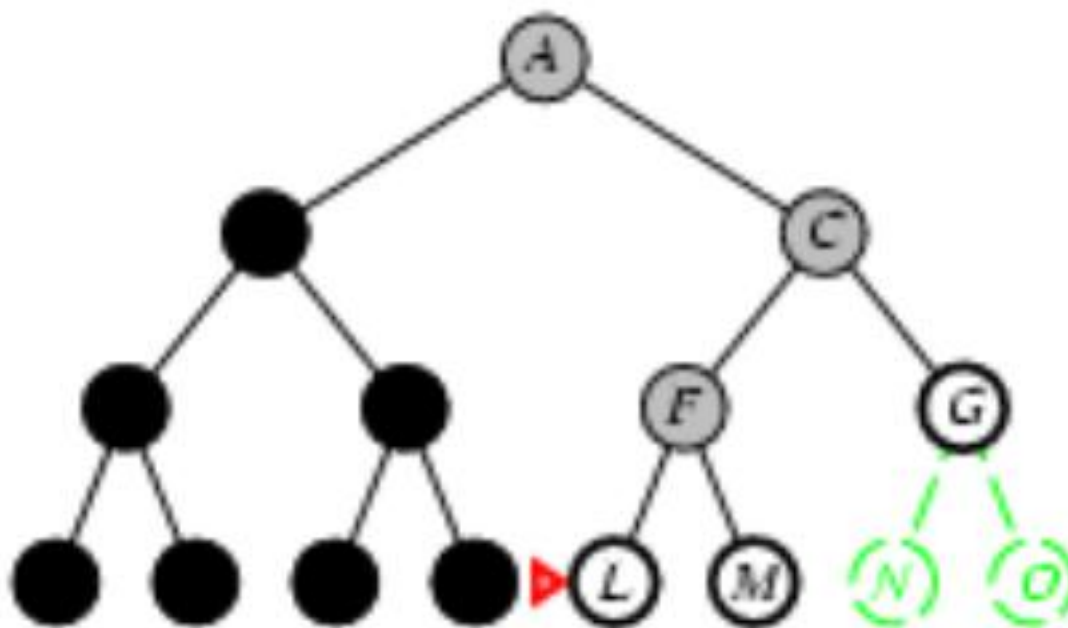
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



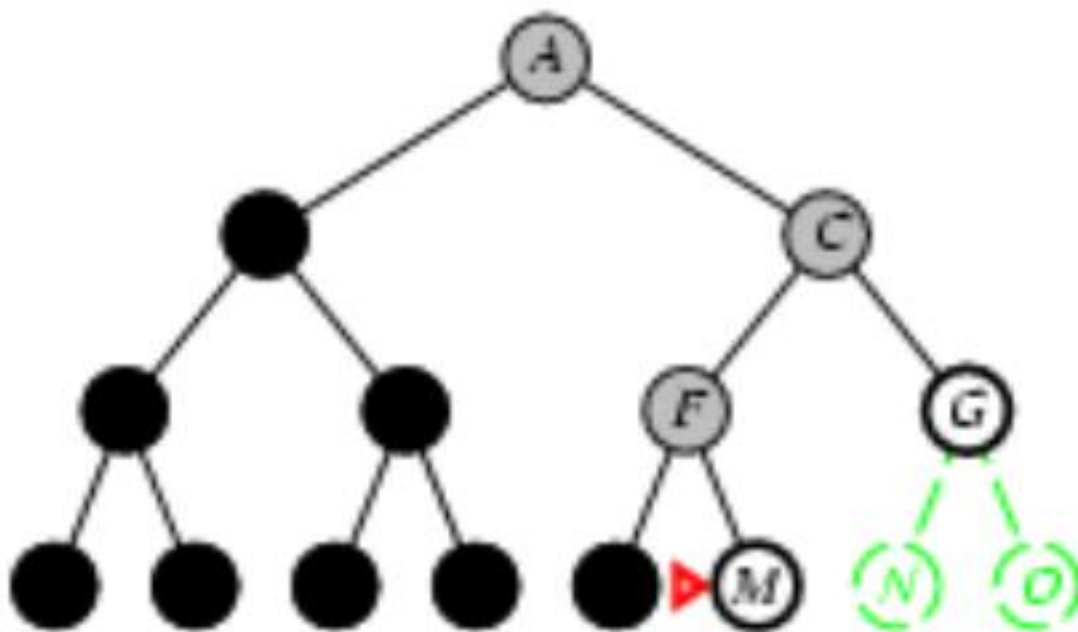
Depth-first search

- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



Depth-first search

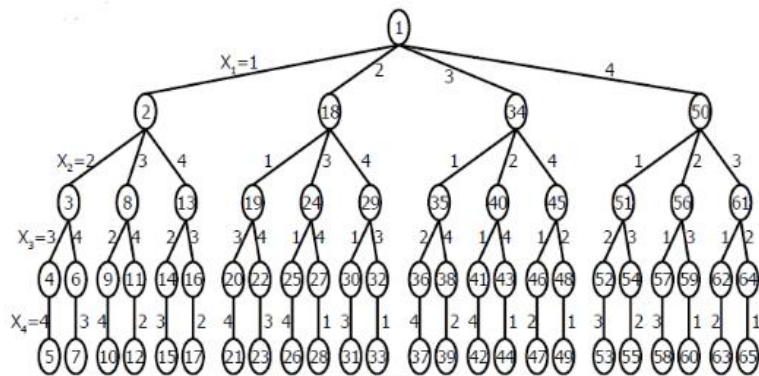
- ◆ 优先扩展和搜索在搜索树中处于最低层的节点



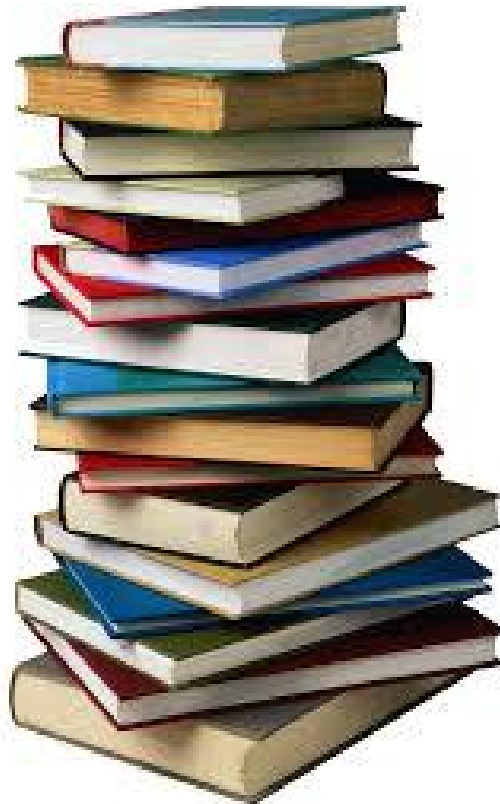
8皇后深度优先搜索的简单实现

```
for (i1=1;i1<=8;i1++){
    for (i2=1;i2<=8;i2++){
        for (i3=1;i3<=8;i3++){
            for(i4=1;i4<=8;i4++){
                for(i5=1;i5<=8;i5++){
                    for(i6=1;i6<=8;i6++){
                        for (i7=1;i7<=8;i7++){
                            for (i8=1;i8<=8;i8++){
                                if (isValid(i1,i2,i3,i4,i5,i6,i7,i8)){
                                    Num ++;
                                    printf("方案%d: %d %d %d %d %d %d %d %d\n", Num, i1, i2, i3, i4, i5, i6, i7, i8);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

思考：这个多层循环如何搜索右侧的状态空间树？



栈 stack



栈 stack

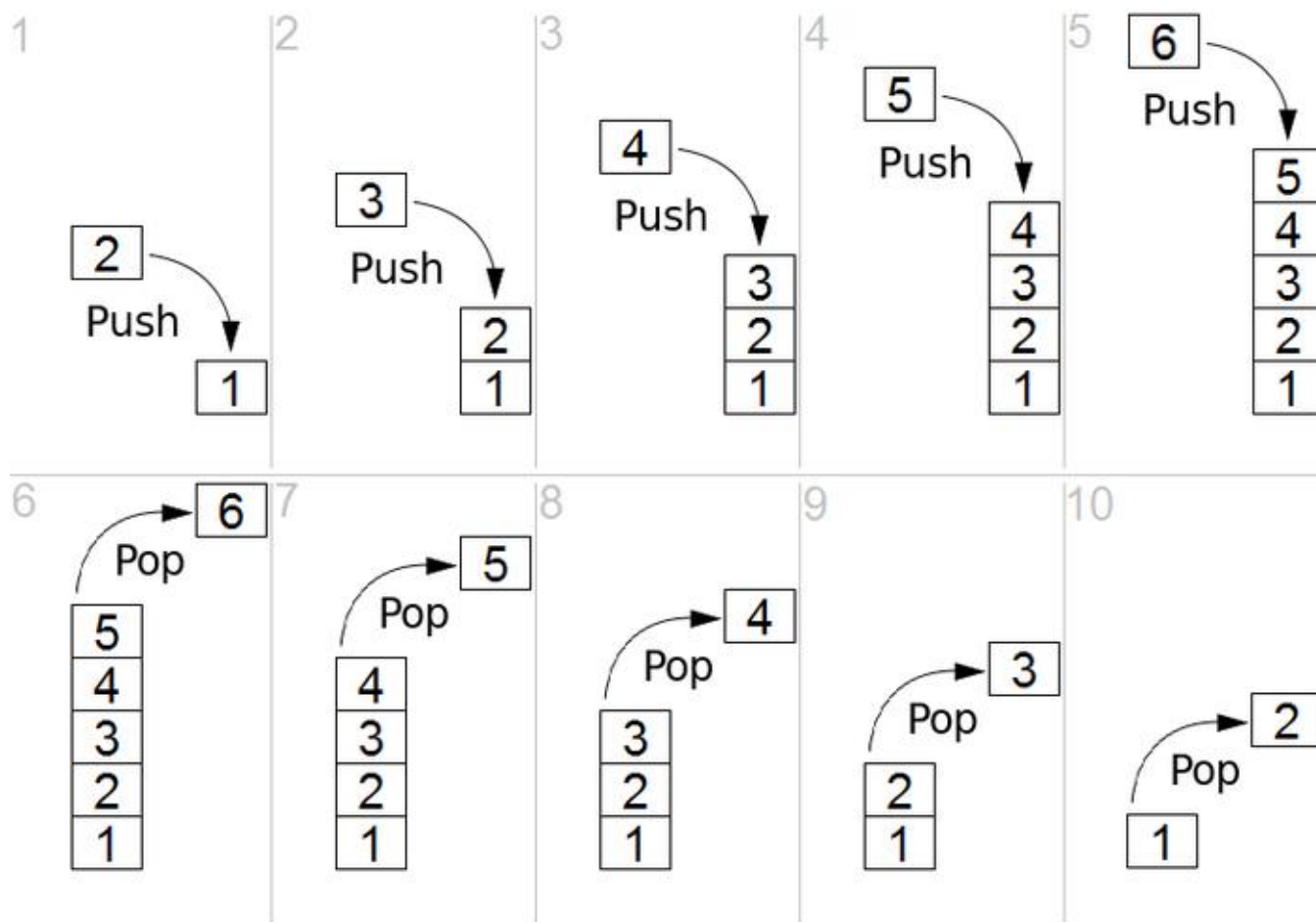
◆ 特点：last in, first out (LIFO)

◆ 操作

◆ pop

◆ push

◆ top





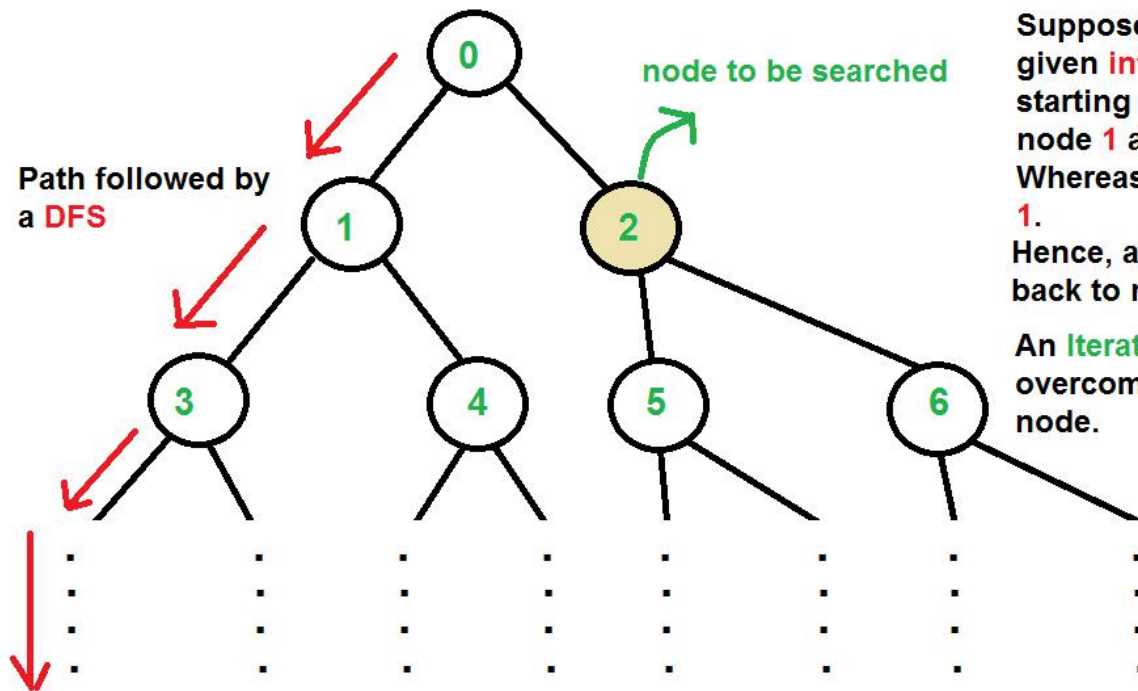
基于栈的DFS算法

1. 将初始节点压入栈内
2. 检查栈是否为空。当栈为空时停止搜索，否则弹出栈顶节点 w
3. 检查 w 是否为目标节点。如果是，则输出结果，如果需要得到其他解则转第2步，否则终止搜索。
4. 如果 w 不是目标节点，则扩展该节点，并将生成的新节点压入栈中
5. 转到第2步

分析DFS

- ◆ 可能在初始阶段误入歧途，直到搜索到该路径的终点，然后再对其他路径进行搜索
 - ◆ 对于无限状态空间的问题，可能无法结束
 - ◆ 状态空间有限，可能由于进入很深的无效路径而效率很低
 - ◆ 搜索路径或者结果不唯一时，不能保证找到的结果是所有可能结果中具有最短搜索路径的
- ◆ 适用情况
 - ◆ 有限的状态空间
 - ◆ 高度较为均衡的搜索树
 - ◆ 需要对有限状态空间完全遍历的搜索
 - ◆ 不要求寻找搜索步数最少的解的问题

分析DFS



Suppose, we want to find node- '2' of the given **infinite** undirected graph/tree. A **DFS** starting from node- 0 will dive left, towards node 1 and so on.

Whereas, the node 2 is just adjacent to node 1.

Hence, a DFS wastes a lot of time in coming back to node 2.

An **Iterative Deepening Depth First Search** overcomes this and quickly find the required node.

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-  
ure  
  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
```

Since iterative deepening visits states multiple times, it may seem wasteful, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level, so it does not matter much if the upper levels are visited multiple times

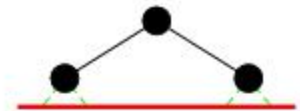
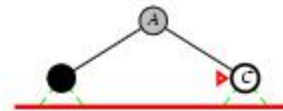
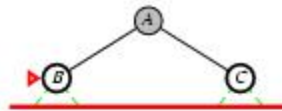
Iterative deepening DF search / =0

Limit = 0



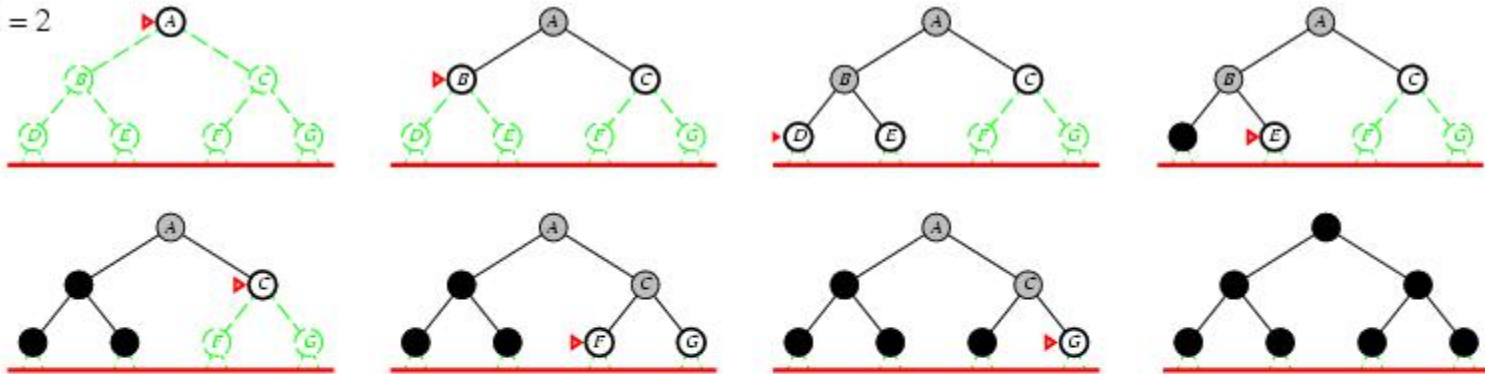
Iterative deepening DF search / =1

Limit = 1



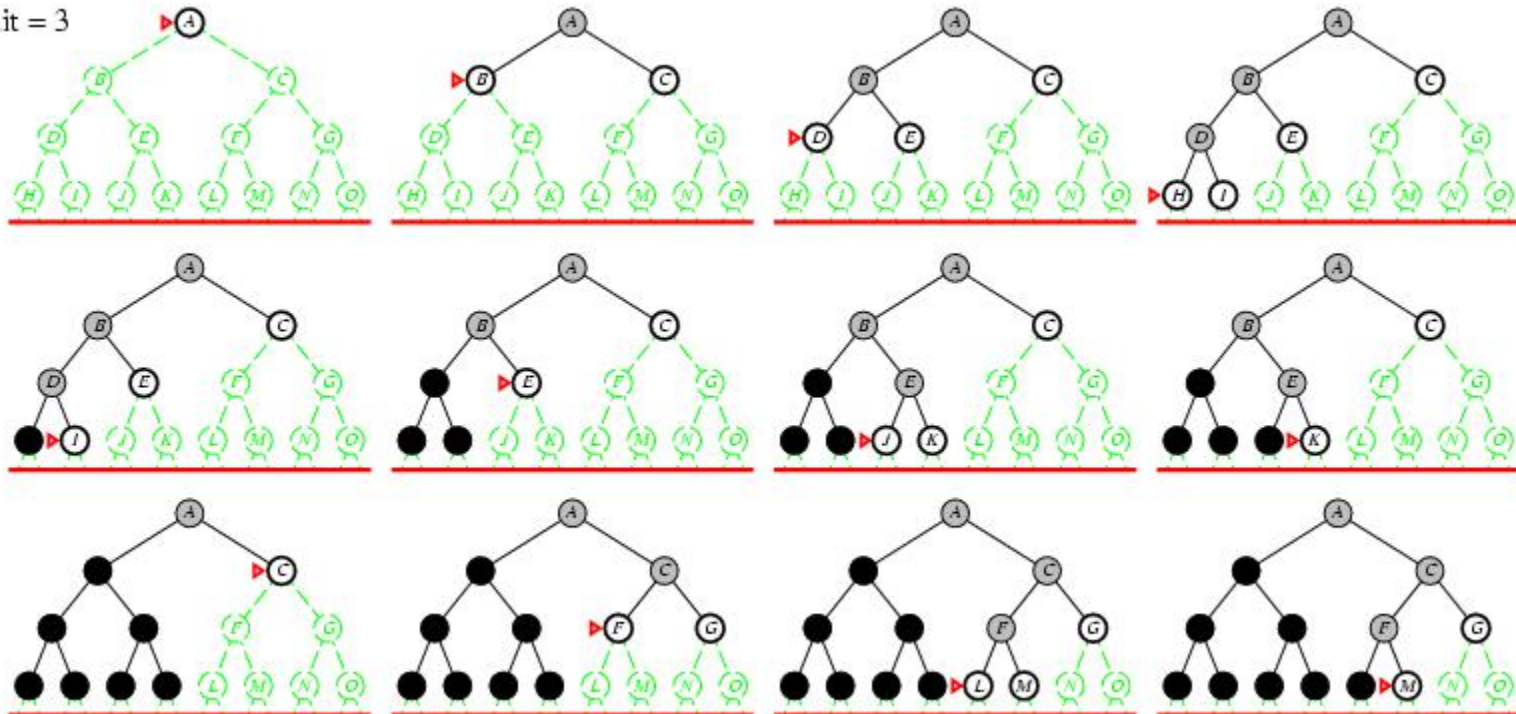
Iterative deepening DF search / =2

Limit = 2



Iterative deepening DF search / =3

Limit = 3



Iterative deepening search

- ◆ Number of nodes generated in a **depth-limited search** (DLS) to depth d with branching factor b :

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

- ◆ Number of nodes generated in **an iterative deepening search** (IDS) to depth d with branching factor b :

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

- ◆ For $b = 10, d = 5$,
 - ◆ $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
 - ◆ $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
- ◆ Overhead = $(123,456 - 111,111)/111,111 = 11\%$

重复节点的判断

- ◆ 八皇后问题在搜索过程中不会构成环，即搜索过程中不会出现重复的节点。
- ◆ 有些问题，其状态空间所构成的是带有圈的图，在搜索过程中可能生成出重复的节点。如果不处理，搜索会陷入圈中无法进展。
 - ◆ 例如八数码问题

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

目标状态

2	8	3
1	6	4
	7	5

2	8	3
1		4
7	6	5

2	8	3
1	6	4
7	5	

2	8	3
	1	4
7	6	5

2		3
1	8	4
7	6	5

2	8	3
1	6	4
7		5

2	8	3
1	4	
7	6	5

2	8	3
1		4
7	6	5

2	8	
1	4	3
7	6	5

2	8	3
1	4	5
7	6	

八数码：初始状态和
部分搜索过程

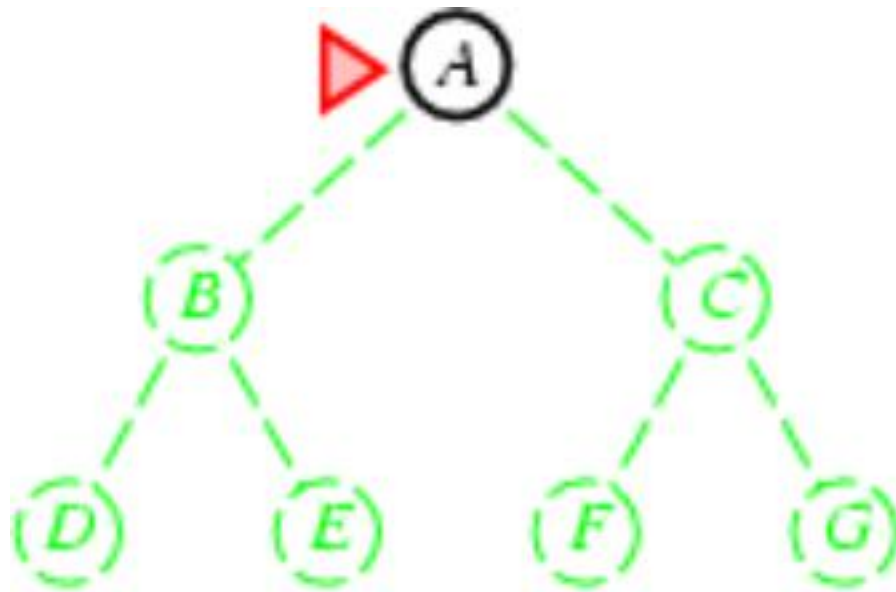
重复节点的判断

- ◆ 如何进行两个节点状态之间的比较
 - ◆ 大部分问题，比较容易，逐一比较状态的属性即可
 - ◆ 部分问题，状态描述比较复杂，需要选择适当的方法或采用适当编码技术
- ◆ 如何确定哪些节点需要比较
 - ◆ 对于状态空间小的问题，比较效率影响不大
 - ◆ 状态空间大，比较效率非常重要，尽可能避免不必要的比较

广度优先搜索 (BFS)

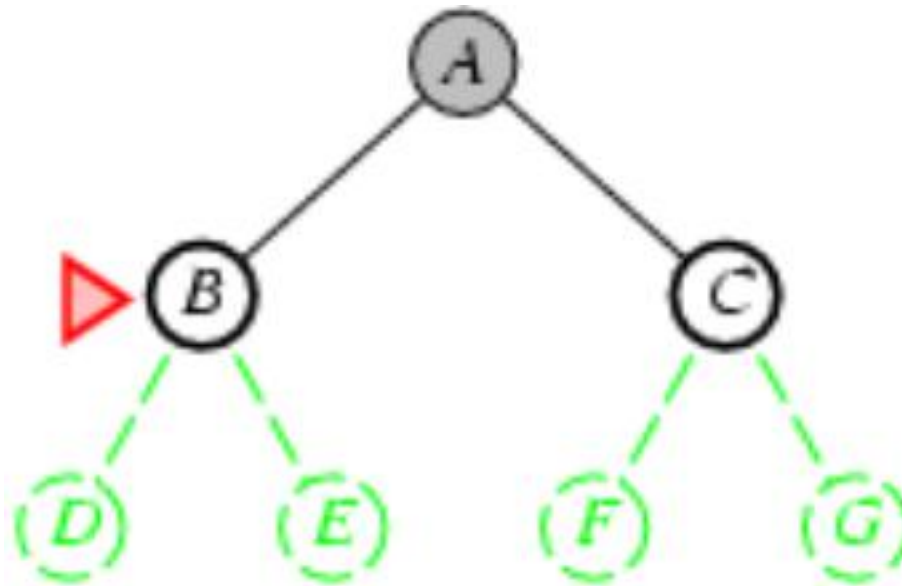
- ◆ 节点的生成和扩展是按层次顺序进行的，处于搜索树较浅层次的节点总是先被检查和扩展
- ◆ 节点搜索的先后顺序与它们生成顺序一致
- ◆ 一般使用队列作为存储

Breadth-first search



状态队列: A

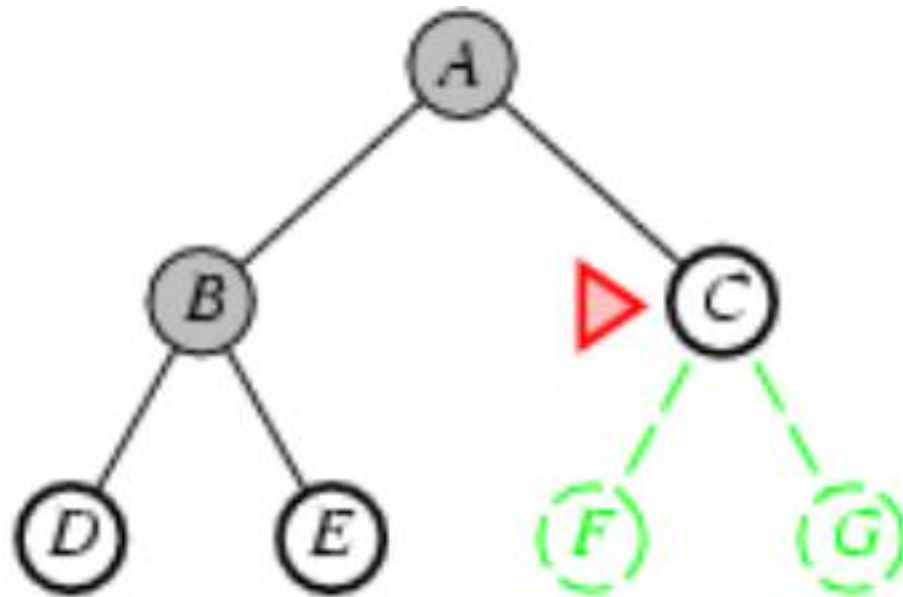
Breadth-first search



状态队列: C, B, ~~A~~

↑
front

Breadth-first search



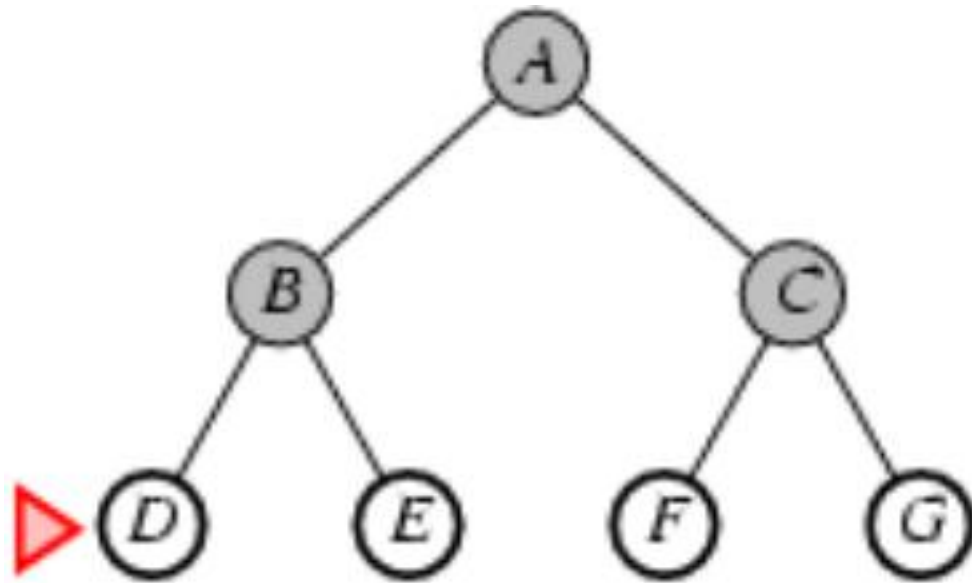
状态队列: E, D, C, ~~B~~, ~~A~~

↑
front

状态队列: G, F, E, D, ~~C~~, ~~B~~, ~~A~~

↑
front

Breadth-first search



状态队列: G, F, E, D, ~~C~~, ~~B~~, ~~A~~

G, F, E, ~~D~~, ~~C~~, ~~B~~, ~~A~~

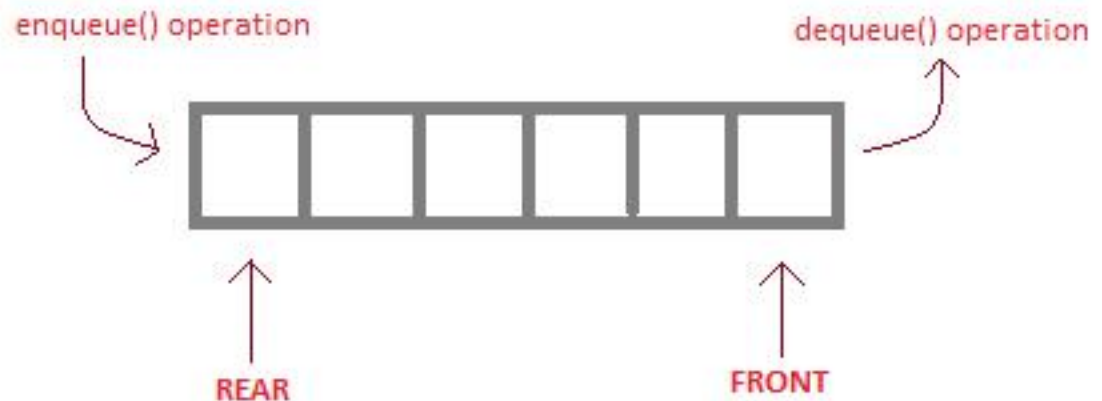
G, F, ~~E~~, ~~D~~, ~~C~~, ~~B~~, ~~A~~

队列 queue



队列 queue

- ◆ 特点：先进先出 first in first out (FIFO)
- ◆ 操作
 - ◆ enqueue
 - ◆ dequeue



`enqueue()` is the operation for adding an element into Queue.

`dequeue()` is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

广度优先搜索算法

1. 选择初始节点并保存到队列的尾部
2. 判断队列是否为空，当队列为空时，停止搜索，否则从队列的头部中取出节点 h （即出队操作）
3. 检查 h 是否为目标节点。如果 h 是目标节点，则输出结果。如果要得到其他解则转到第2步，否则终止搜索
4. 如果 h 不是目标节点，则扩展该节点，将所生成的新节点保存到队列的尾部（即入队操作）
5. 转到第2步

- ◆ 与DFS算法非常相似，所不同的只是使用队列作为节点的存储结构
- ◆ 搜索按节点生成的层次进行
- ◆ 保证搜索到的结果具有最短的搜索路径

八皇后问题的BFS解法

- ◆ 与八皇后的DFS算法非常类似，所不同的是节点的存取放式

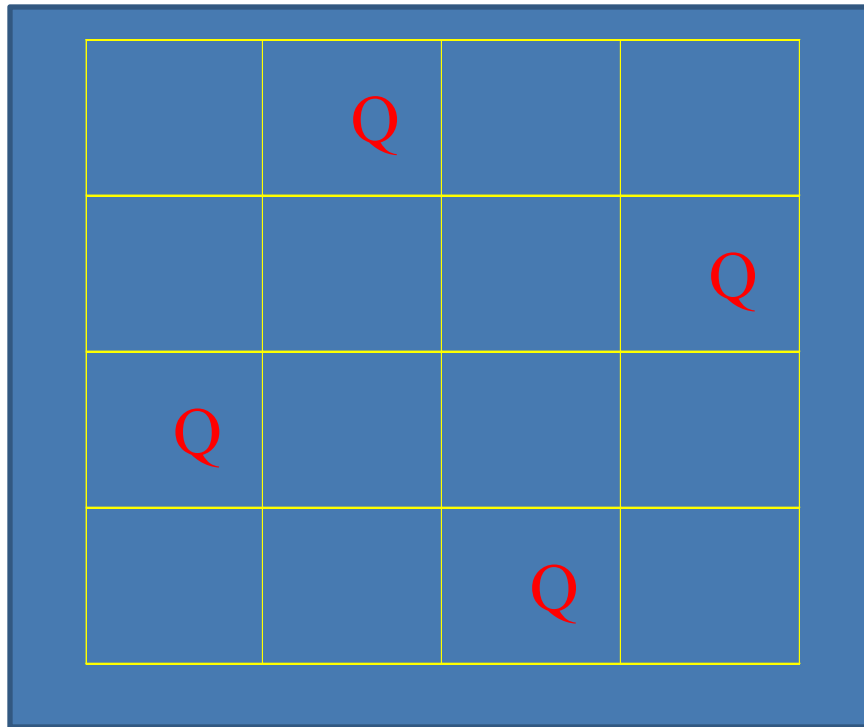


八皇后问题BFS算法分析

- ◆ 分析队列`queue<QueenState>`的大小
- ◆ 可见与DFS相比，BFS使用的内存空间较大，因为对各层节点都同时展开
 - ◆ DFS：空间复杂度 $O(d)$ ， d 为搜索树的深度
 - ◆ BFS：空间复杂度 $O(n)$ ， n 为搜索树的节点个数
- ◆ 对于复杂、状态空间大的问题，使用BFS时有可能由于组合爆炸而产生大量的内存消耗，以致超出计算平台资源的限制，引起搜索失败

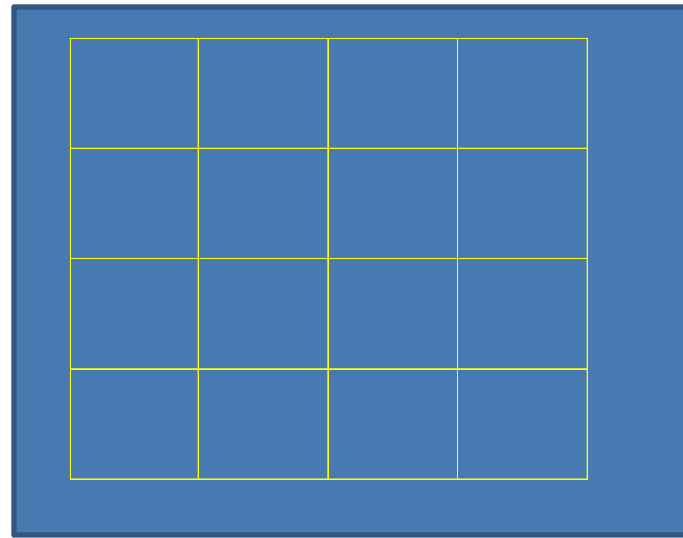
回溯

◇ 例：皇后问题

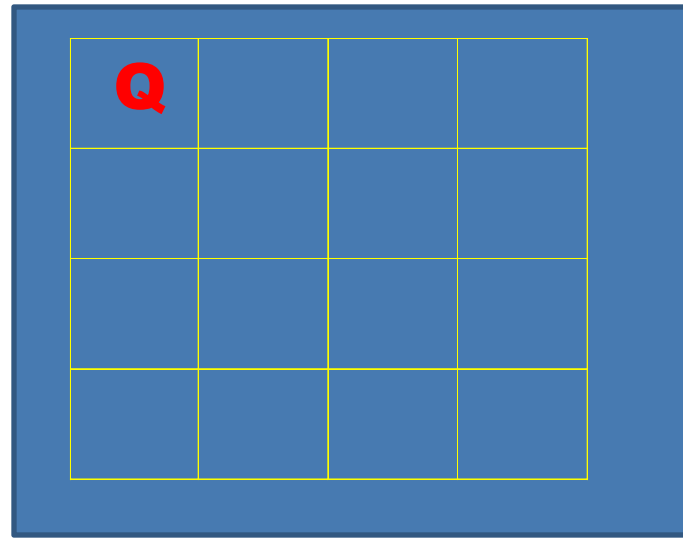
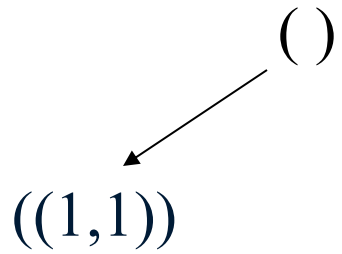


采用逐行放皇后的搜索策略

()



采用逐行放皇后的搜索策略



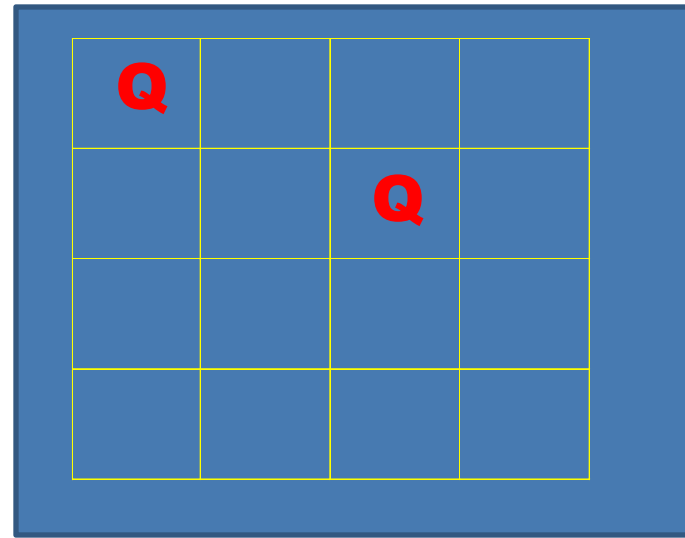
采用逐行放皇后的搜索策略

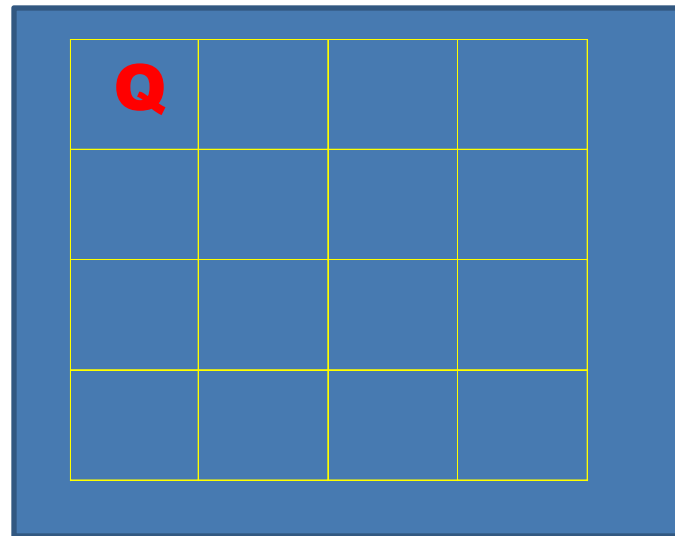
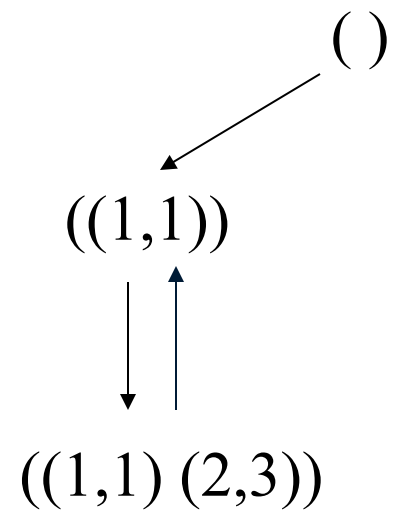
$()$
↙
 $((1,1))$

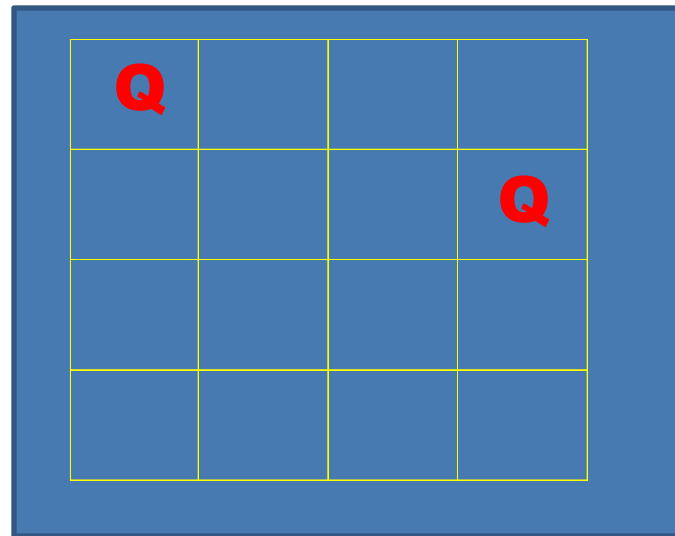
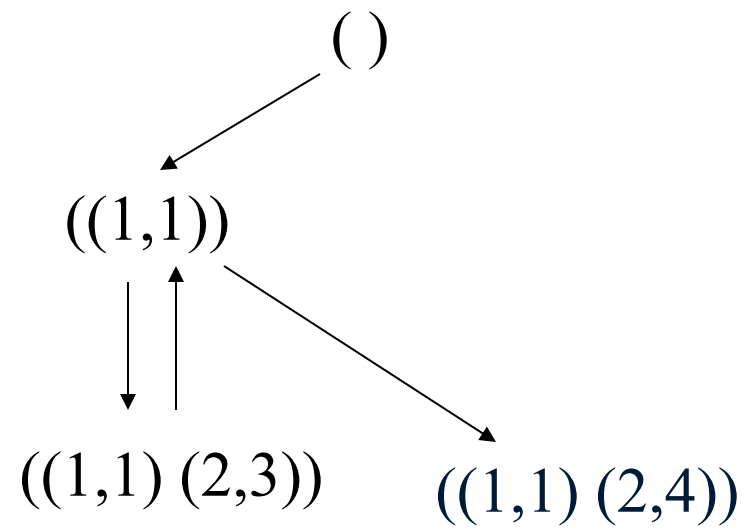


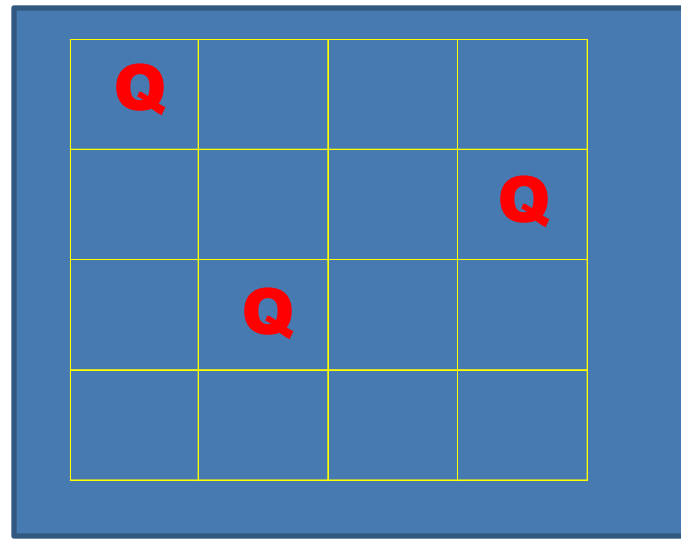
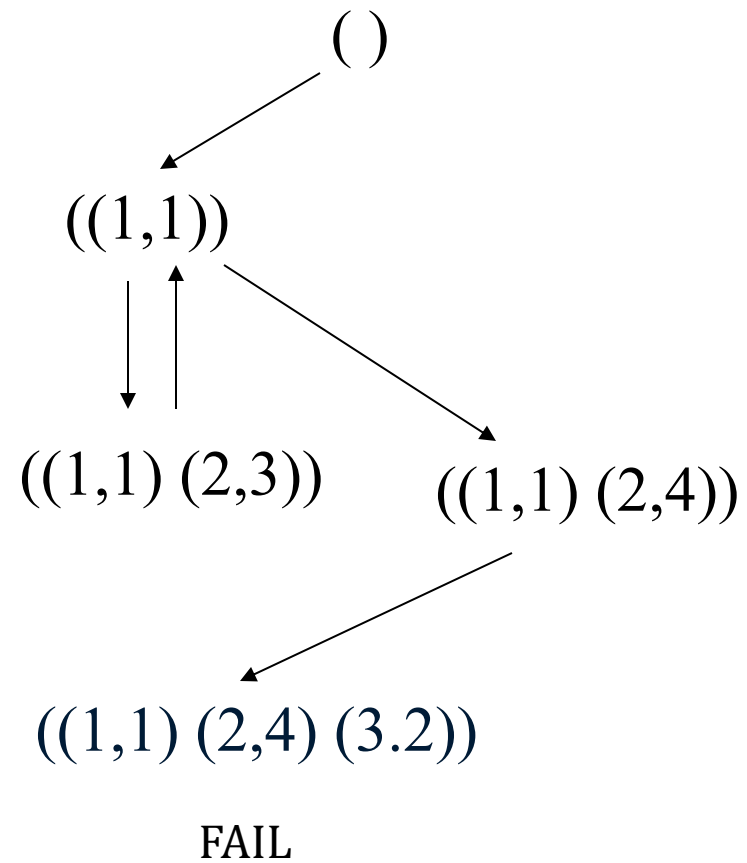
$((1,1) (2,3))$

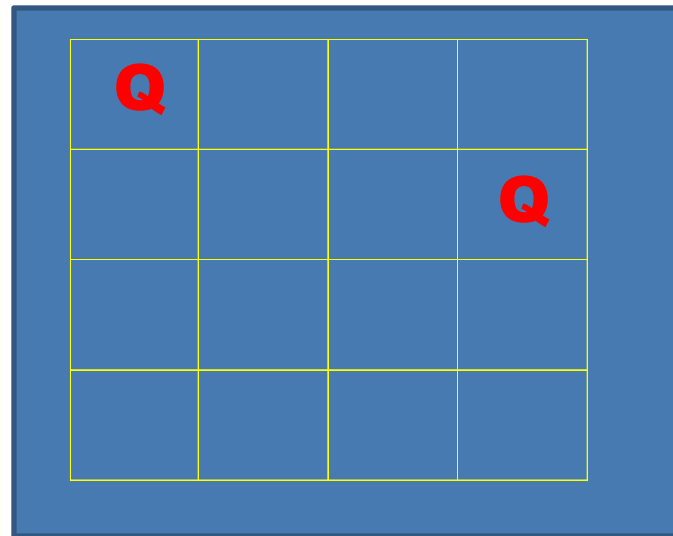
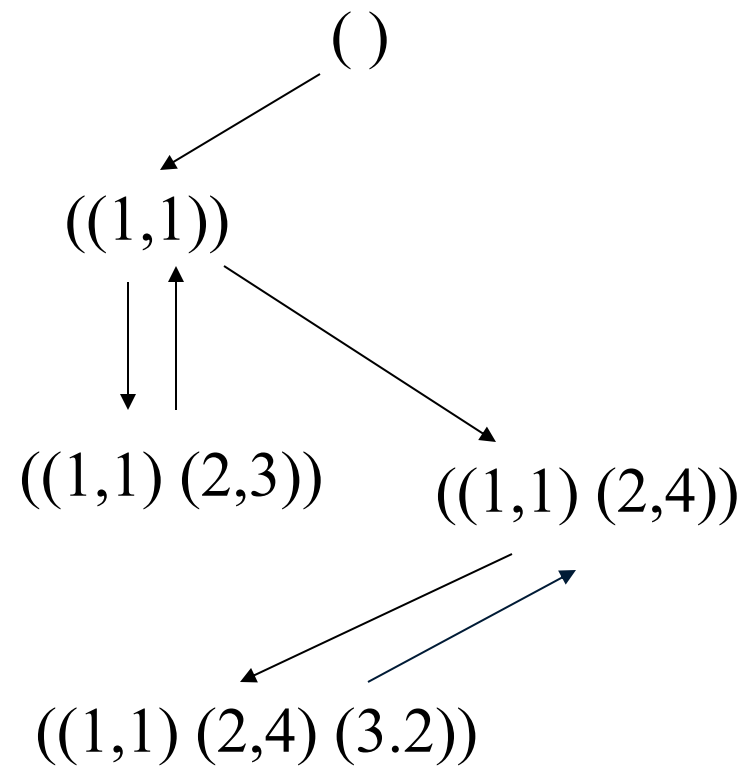
FAIL

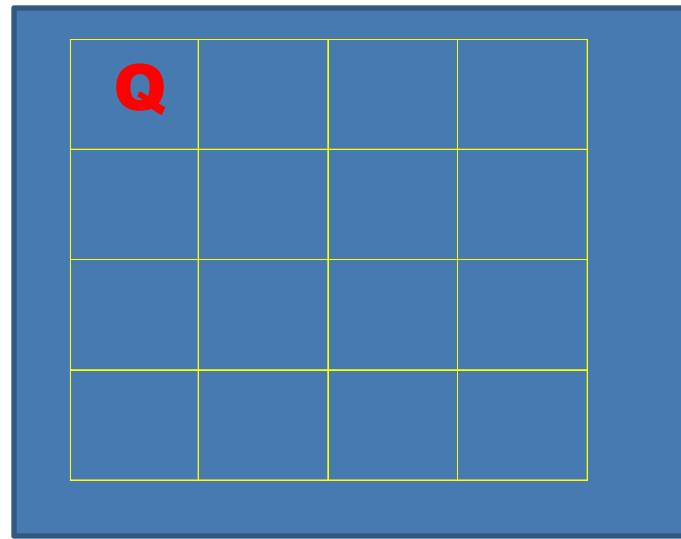
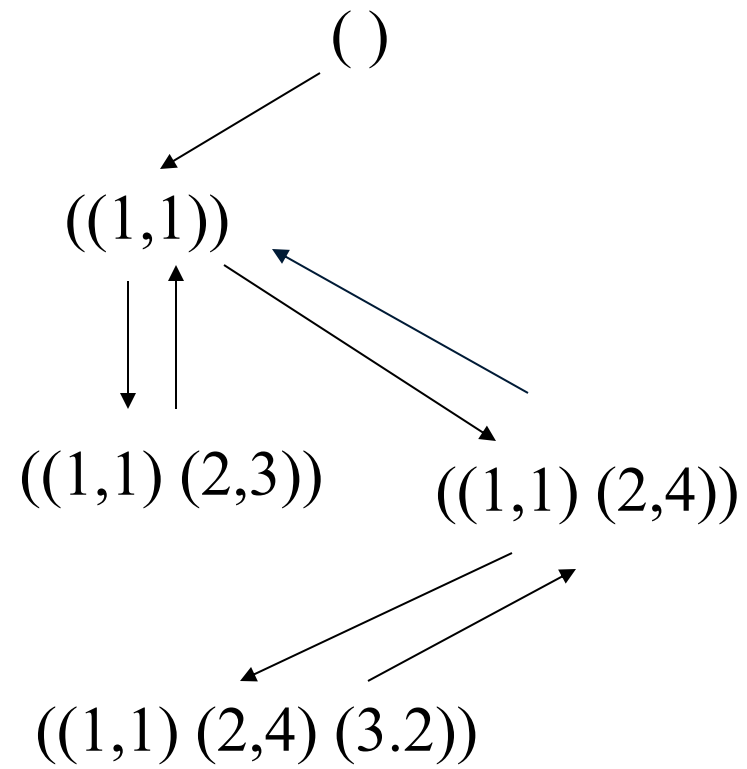


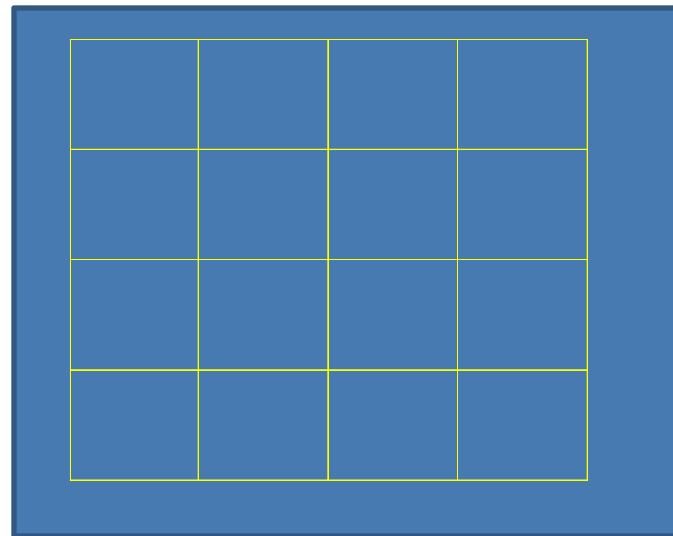
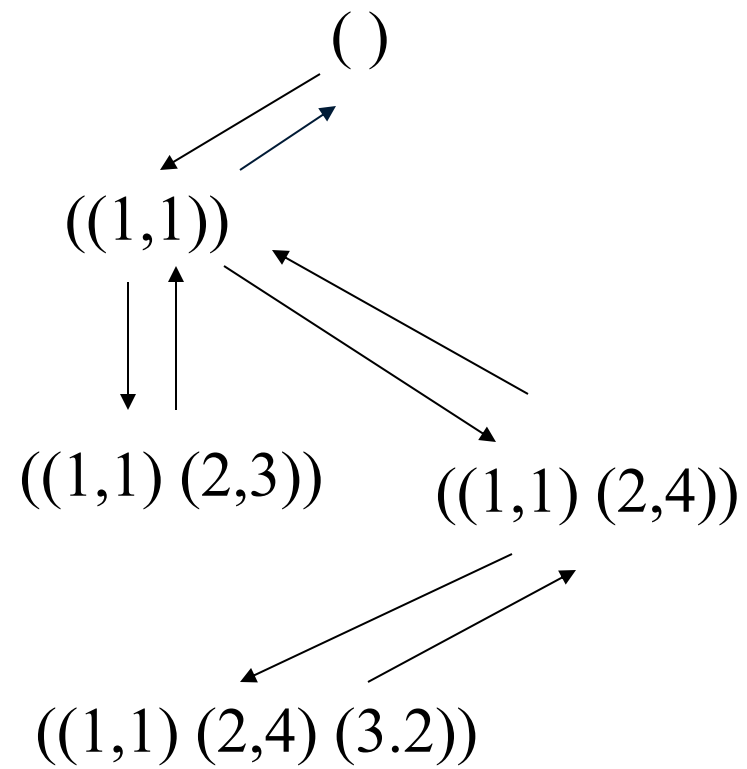


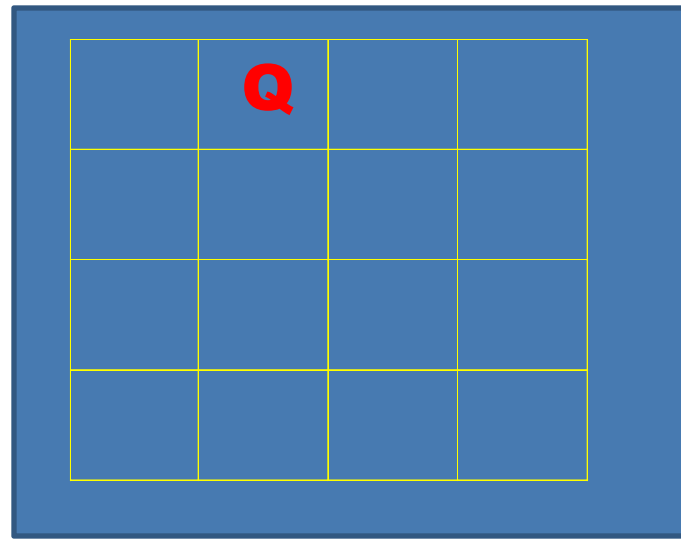
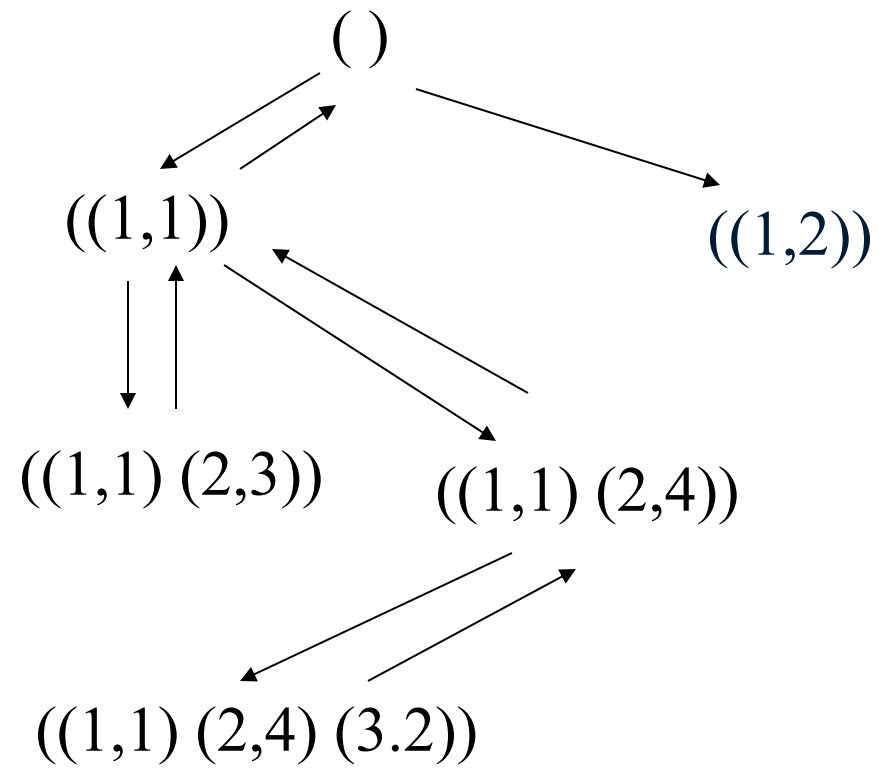


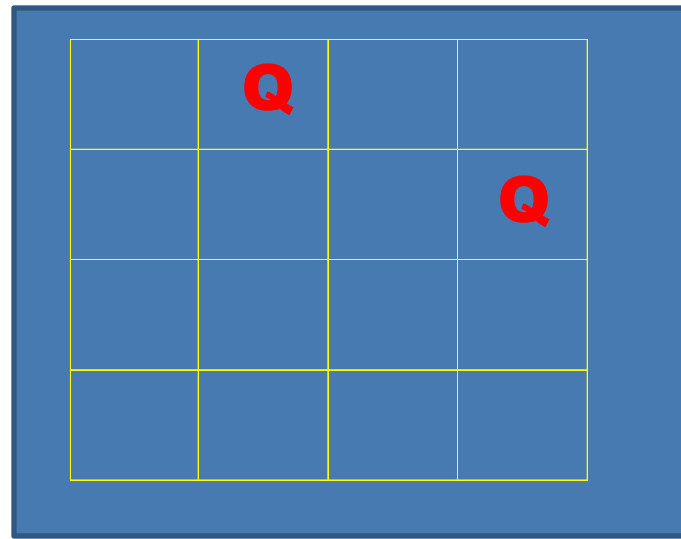
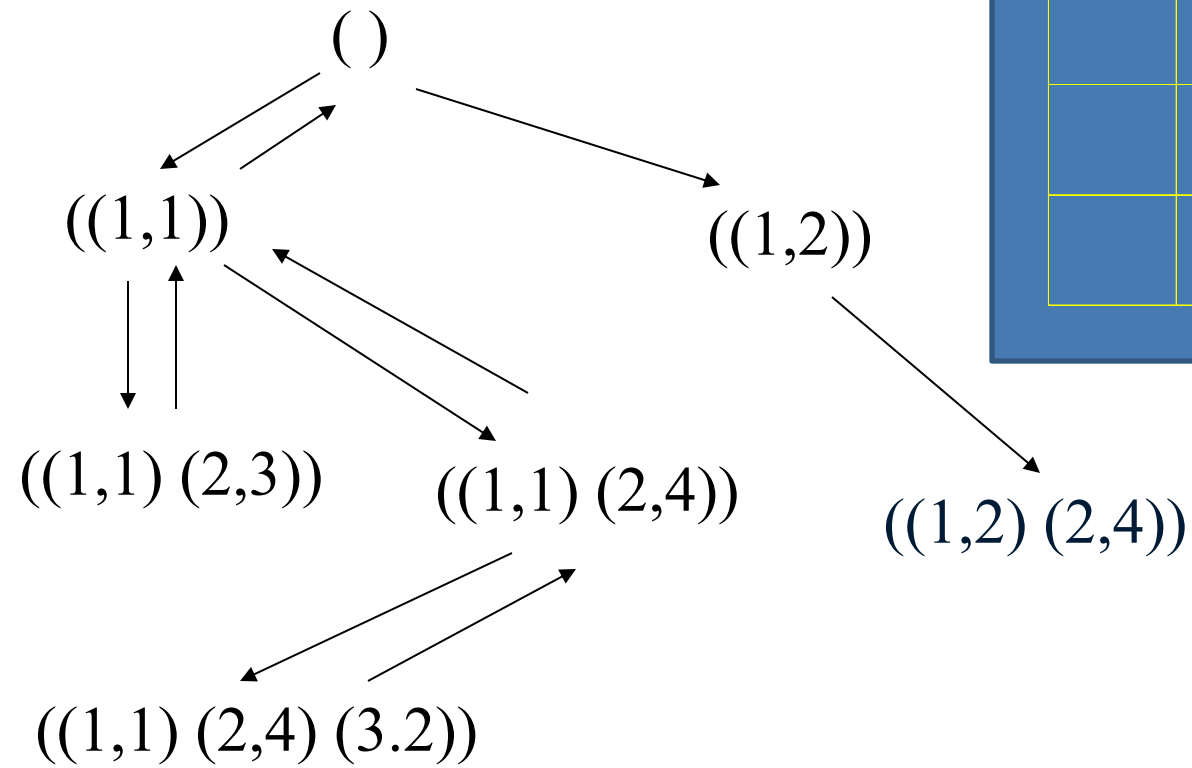


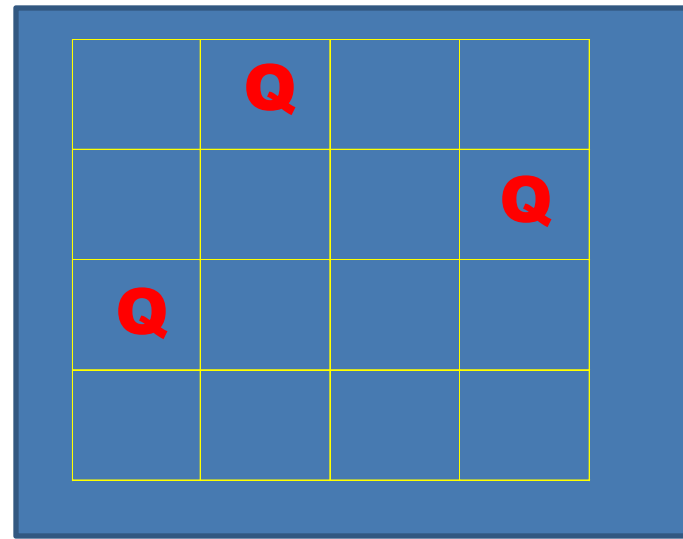
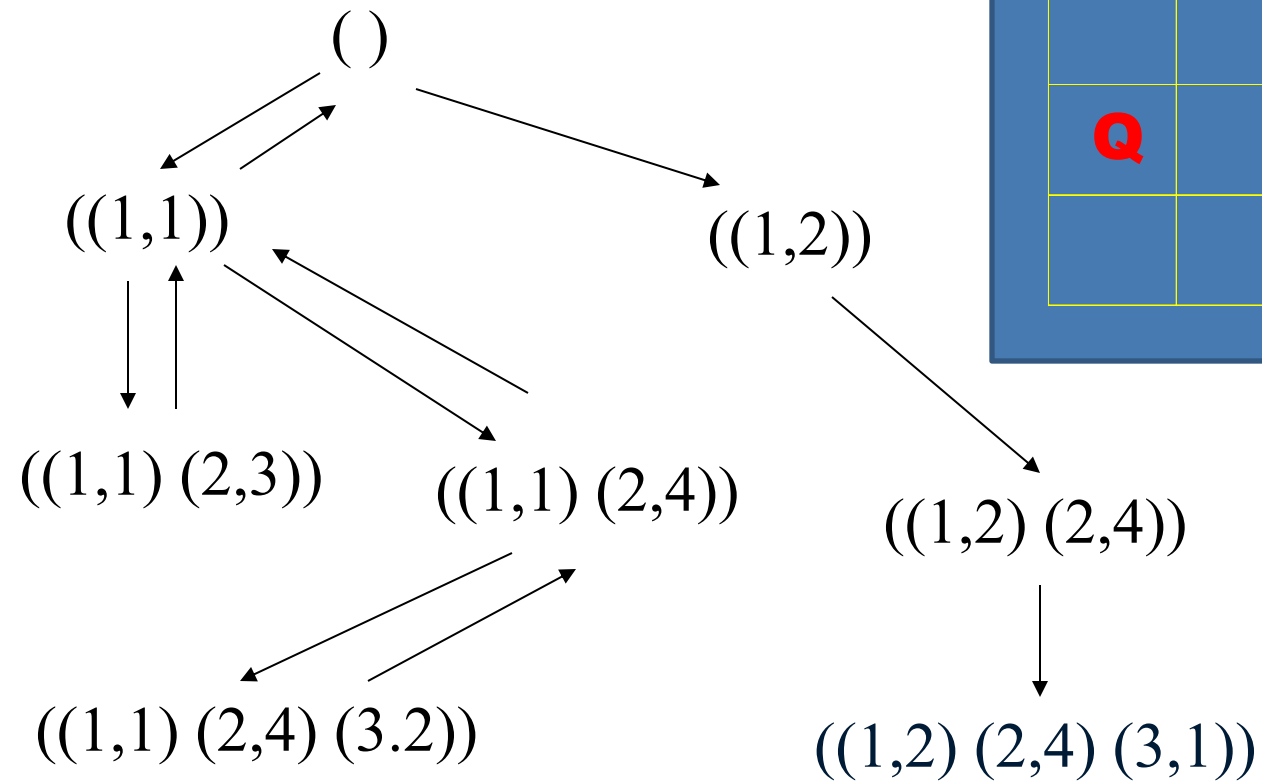


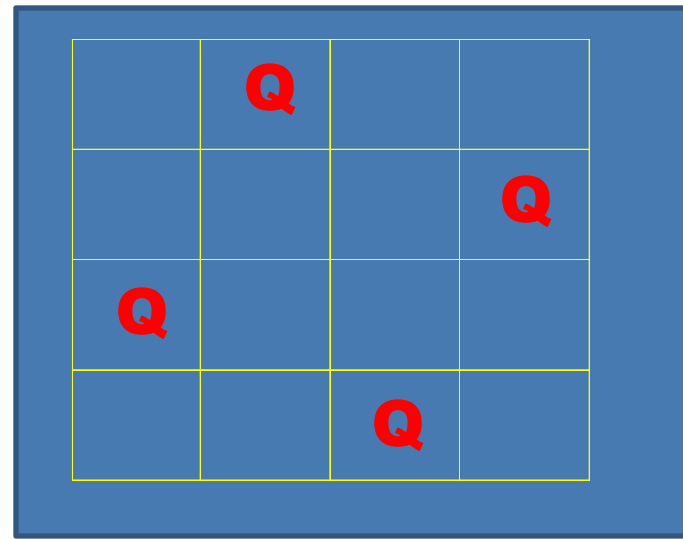
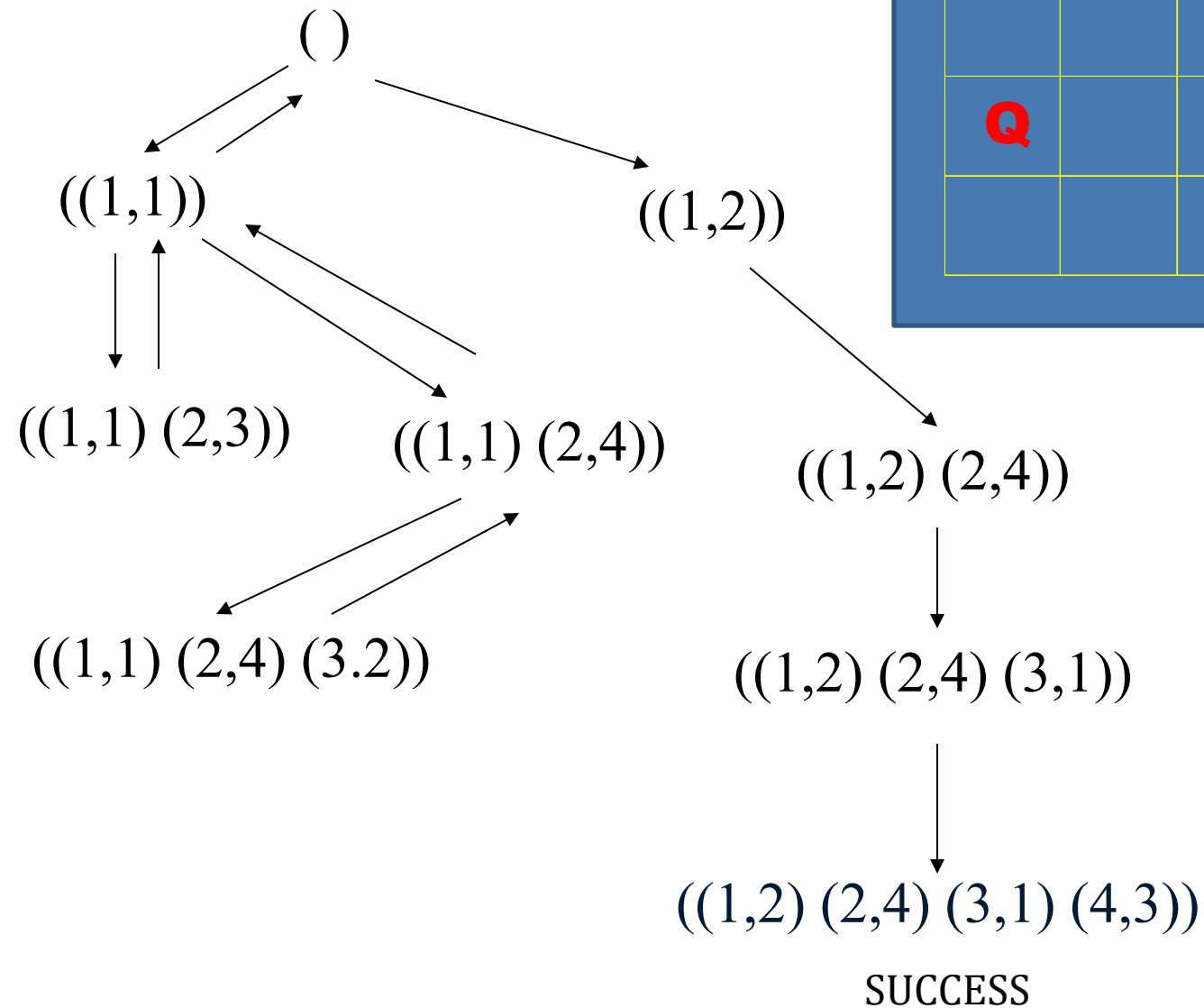




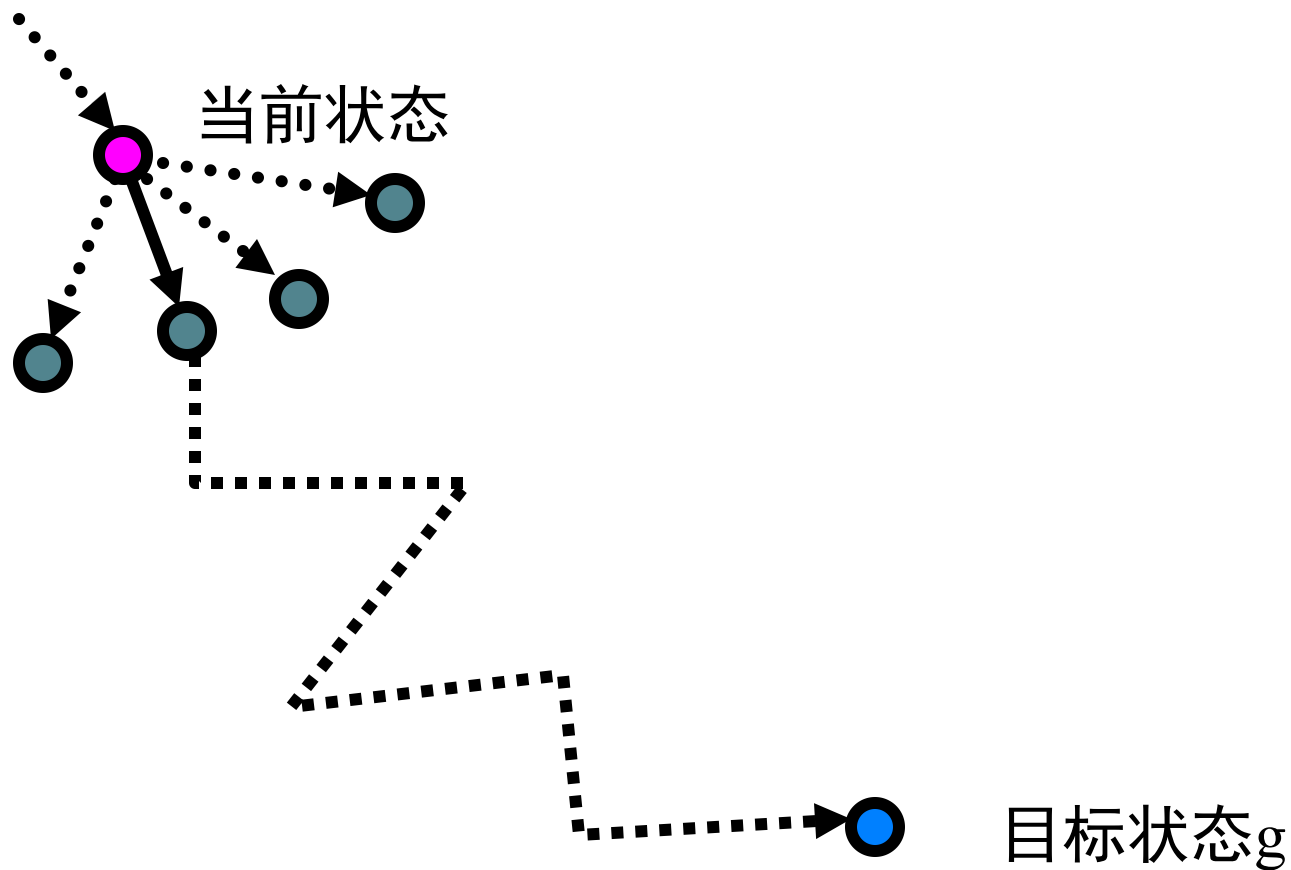








递归的思想



回溯搜索

- ◆ 回溯搜索是深度优先搜索的另一种搜索方式，与前面一般的DFS搜索略有不同
 - ◆ 对节点扩展时，每次只产生一个后继结点，并继续对这个节点进行回溯搜索
 - ◆ 当一条路径搜索完成后，再返回到生成当前节点的那个节点，看是否可以从该节点再生成新的节点
 - ◆ 重复上过程，直到找到搜索目标或者搜索全部状态空间

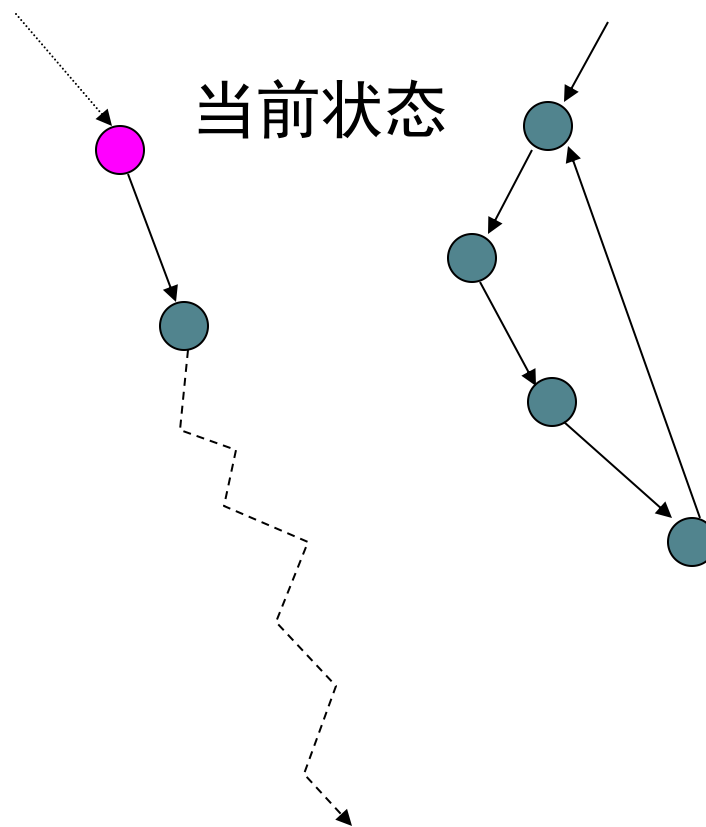
回溯搜索的编程

◇ 编程实现简单

- ◇ 对当前节点的所有后继节点的遍历通过搜索中回溯机制，即函数的递归调用和返回完成
- ◇ 不需要保存后继节点的栈结构

递归存在的问题

- 问题：
 - 深度问题
 - 死循环问题



Informed Search

- ◆ “Informed Search”指在搜索过程中将使用问题定义之外的额外信息、知识，来寻找更高效的解决方案。
- ◆ 基本思路是引入一个评估函数 $f(n)$ 来评估节点，选择分值最高或最低的节点进行扩展——贪心策略
- ◆ 对评估函数的设计可以加入领域知识
 - ◆ 例：五子棋策略优先连四或连三

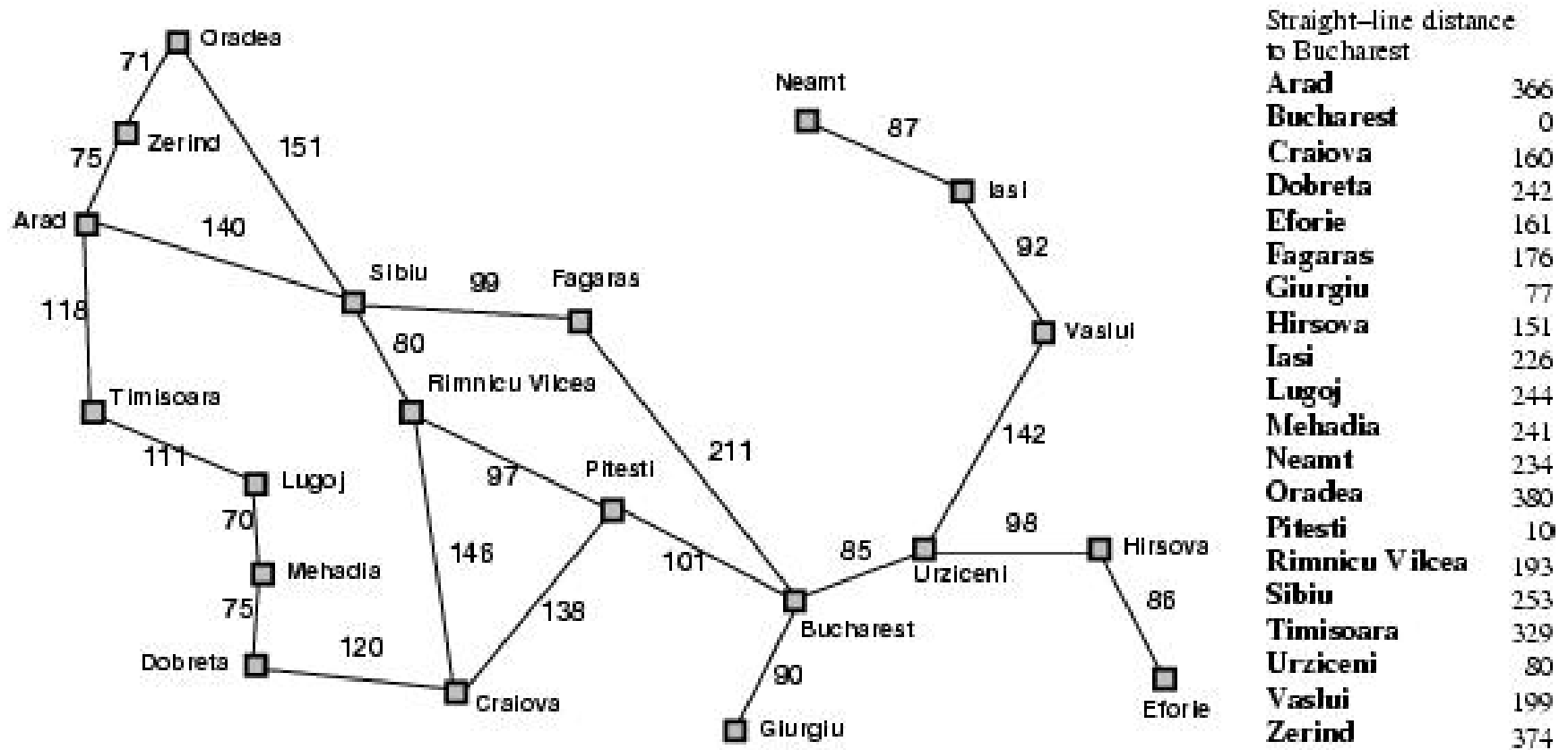


Best-first search

- ◆ 思路: 使用一个评估函数 $f(n)$ 来计算每个待扩展节点的分值
 - ◆ 评估函数通常是一个估计
 - 在待扩展节点中, 选择分值最高或最低的进行扩展
- ◆ 实现方法:
将工作队列 (fringe) 中的节点按评估函数分值排序
- ◆ 扩展方法:
 - ◆ greedy best-first search
 - ◆ A* search

例：最短路径

下图为罗马尼亚地图，路径上数字为城市间距离



Greedy best-first search

- ◆ 评估函数 $f(n) = h(n)$ (**h**euristic)
- ◆ = 估计从n到目标节点之间的代价
- ◆ 比如, $h_{SLD}(n)$ = 从n到目标Bucharest之间直线距离
- ◆ Greedy best-first search 优先扩展**看上去**离目标最近的节点

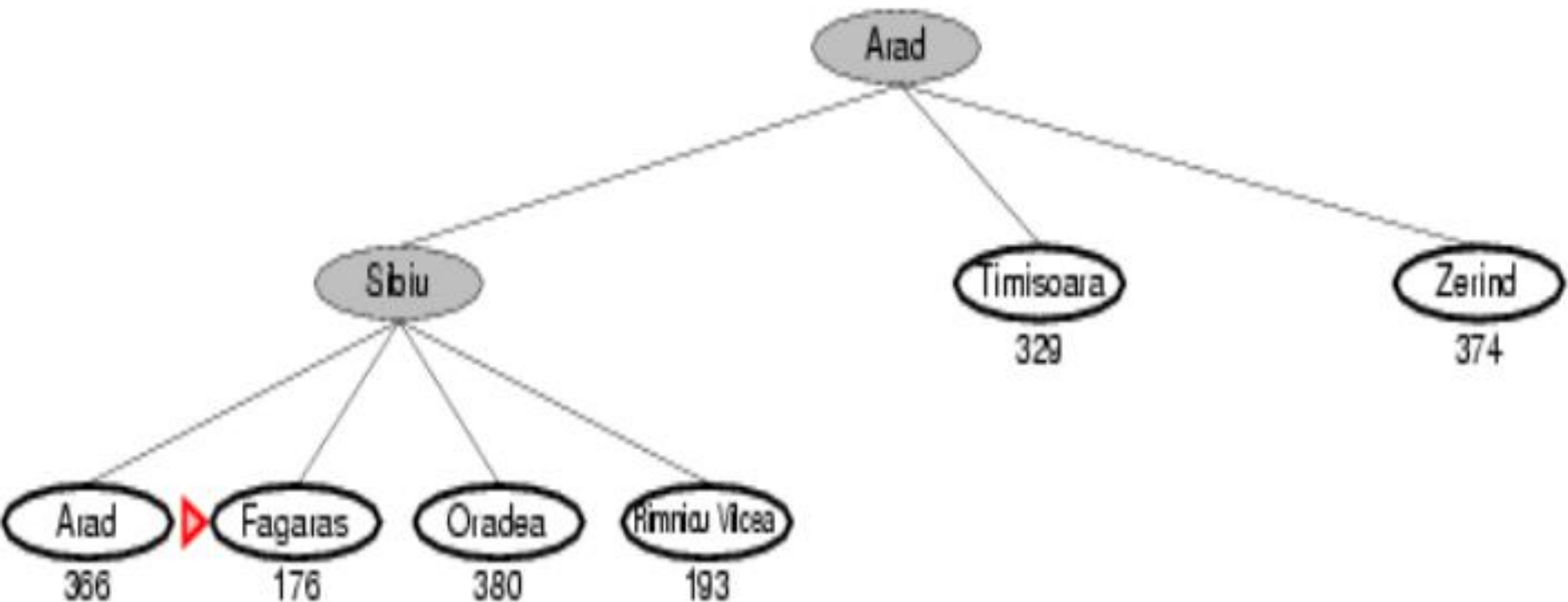
Greedy best-first search example



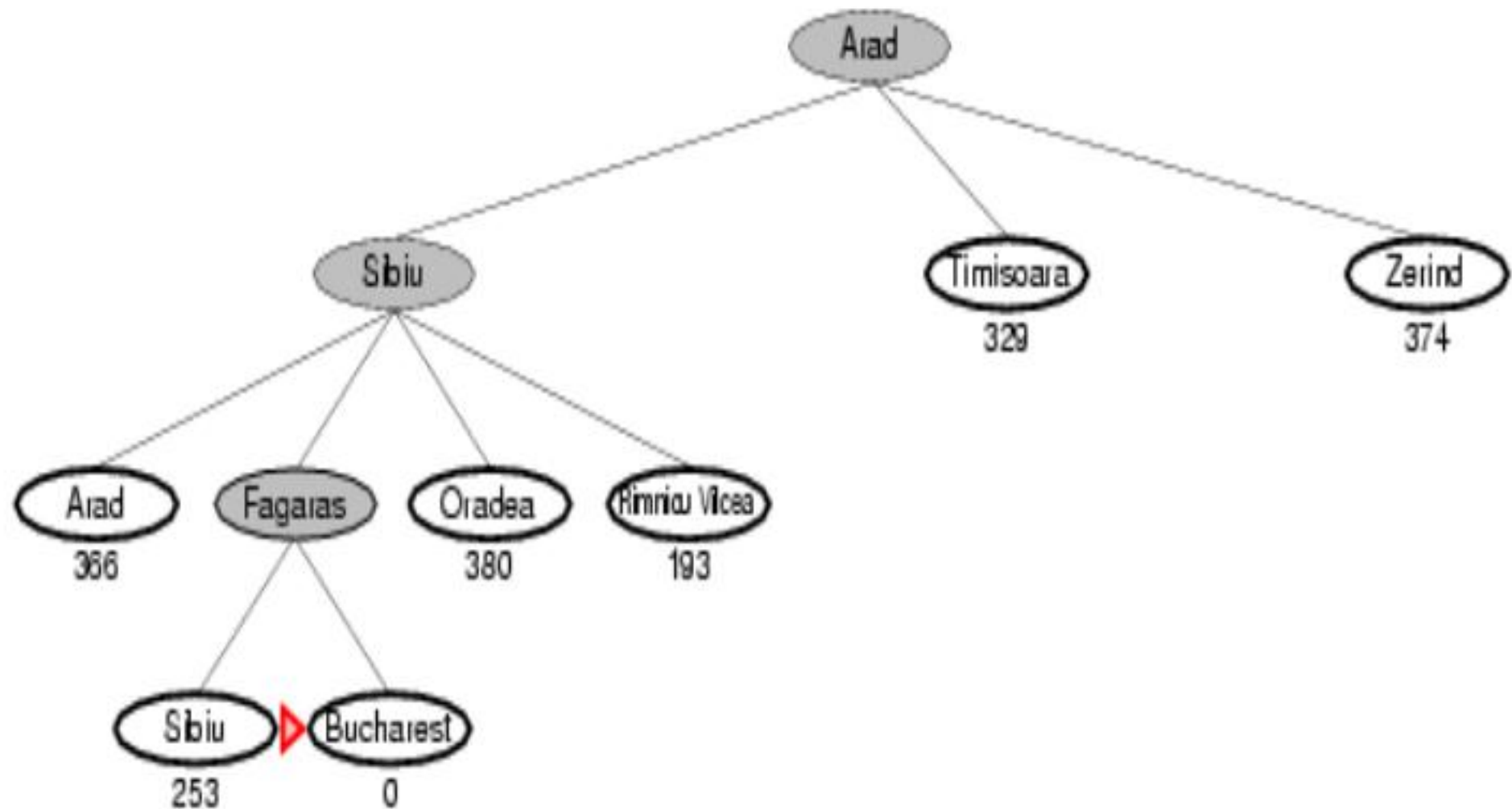
Greedy best-first search example



Greedy best-first search example



Greedy best-first search example



A* search

- ◆ 思路: 避免扩展已经很昂贵的路径
- ◆ 评估函数 $f(n) = g(n) + h(n)$
- ◆ $g(n)$ = 到达节点 n 的代价
- ◆ $h(n)$ = 从 n 到目标的代价估计值
- ◆ $f(n)$ = 估计的总代价

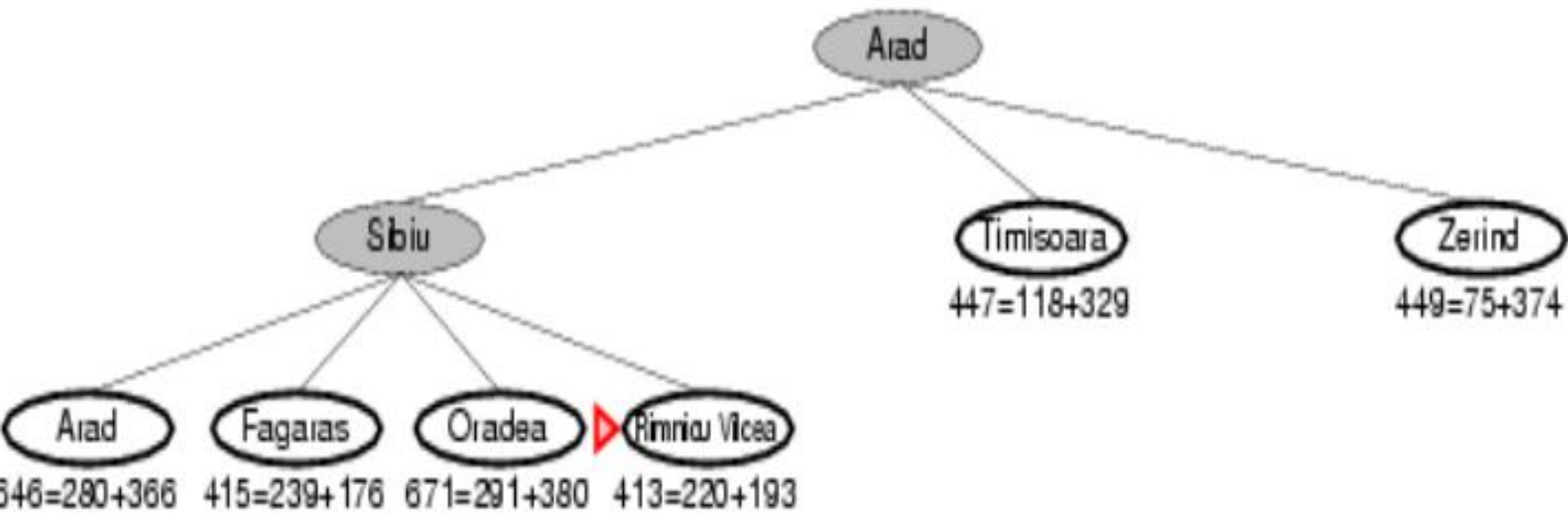
A* search example



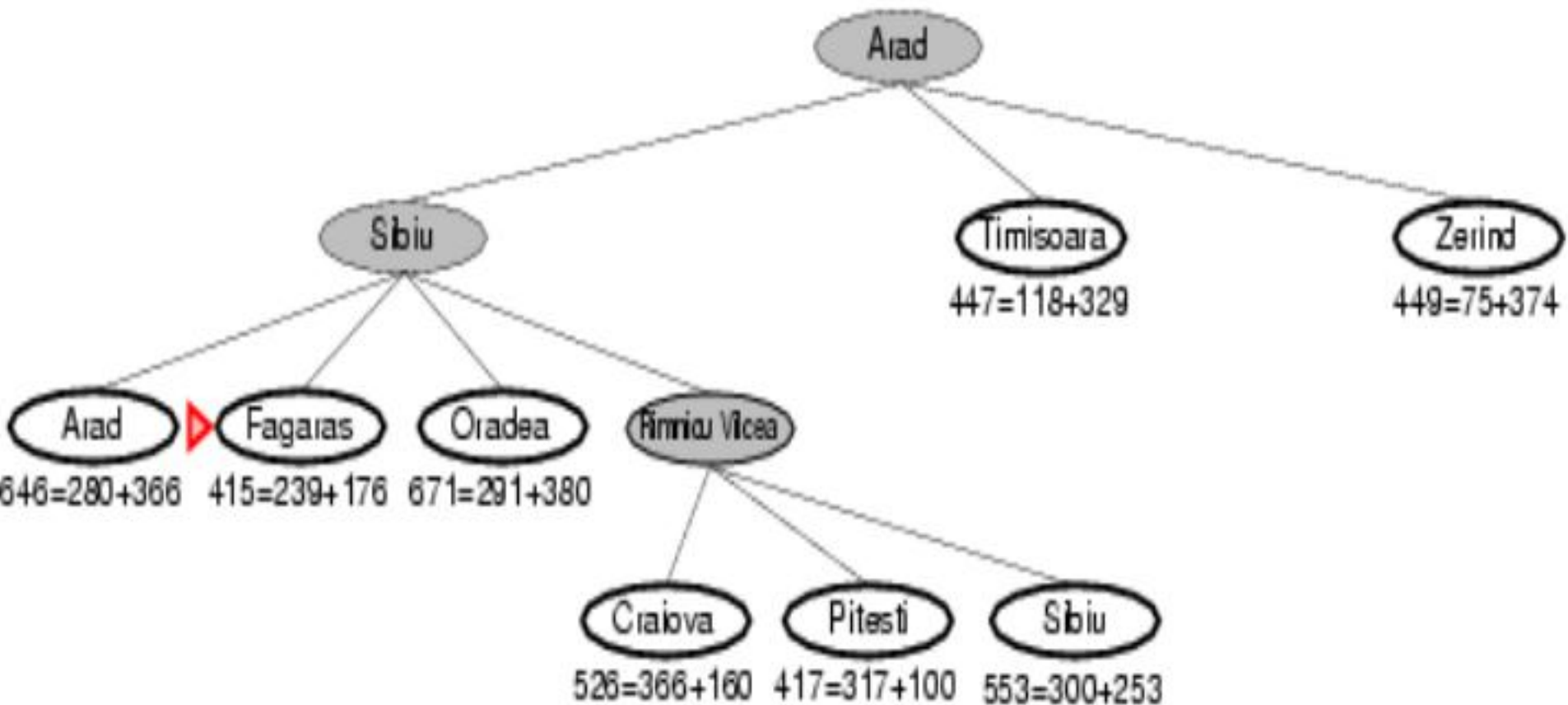
A* search example



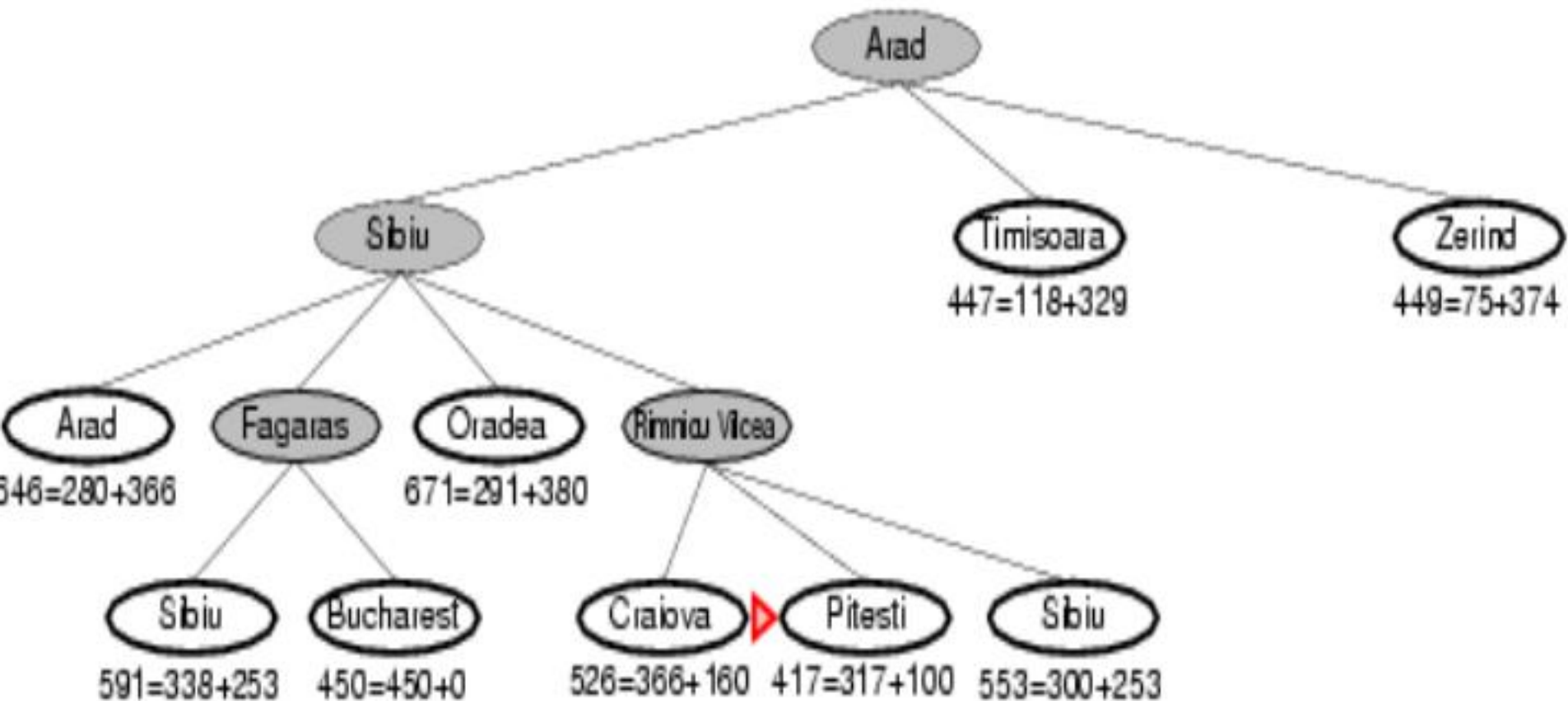
A* search example



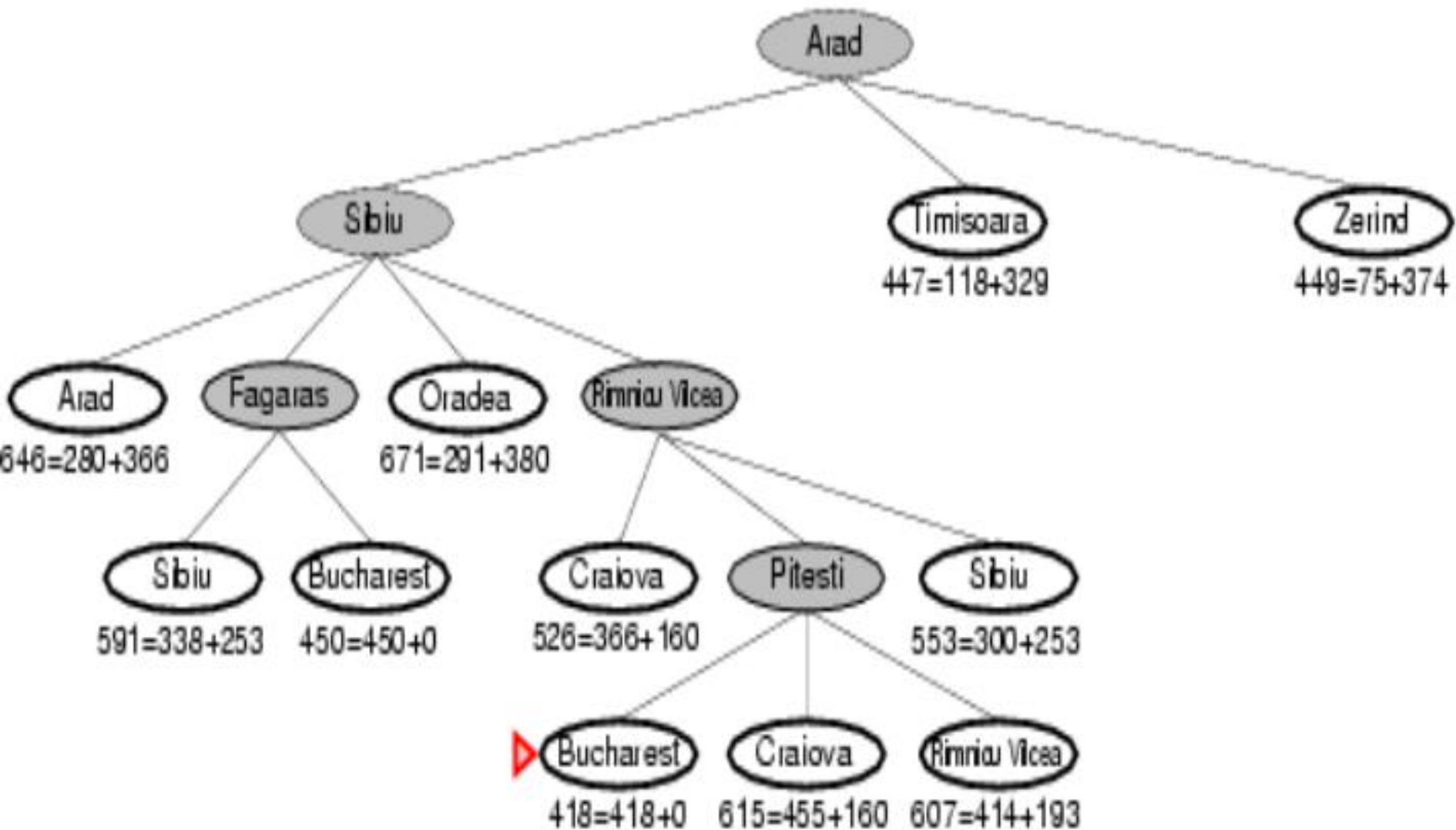
A* search example



A* search example



A* search example



Admissible heuristics

- ◆ A heuristic $h(n)$ is **admissible** if for every node n , $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n .

An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**

Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

- ◆ **Theorem:** If $h(n)$ is admissible, A^* using TREE-SEARCH is optimal

局部搜索算法

Local search algorithms

- ◆ 在很多优化问题中，我们不关心到达目标的路径，目标状态本身就是要求的解决方案
- ◆ 状态空间= 一组完整的配置 (configurations)
 - ◆ 比如n皇后问题
- ◆ 问题目标是寻找满足约束条件的配置
- ◆ 对于这类问题，我们可以采用局部搜索算法
- ◆ 思想：不断尝试优化当前的状态，力图构造目标状态

例: n -queens

- ◆ Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal



爬山算法

Hill-climbing search

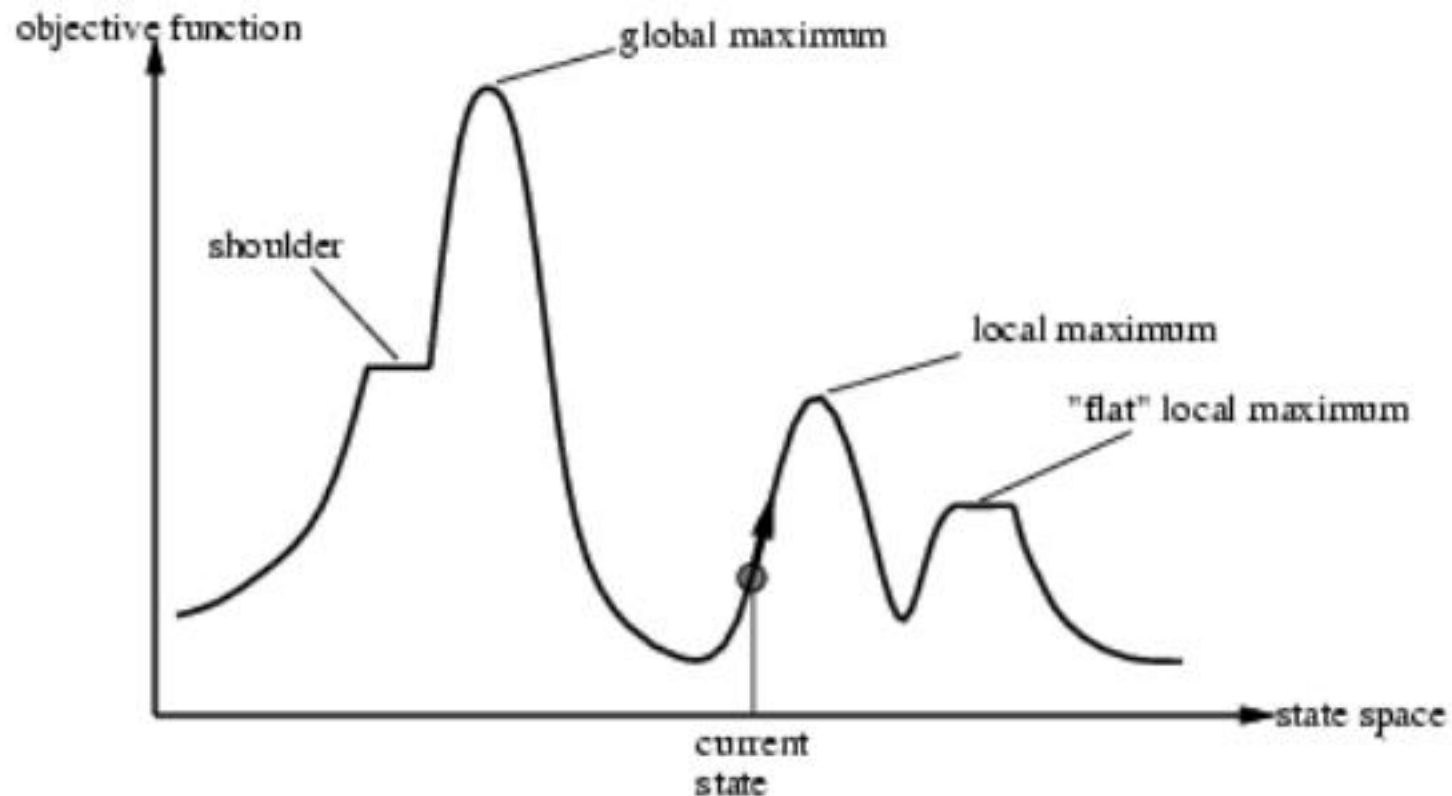
◇ "就像一个健忘的人在浓雾中爬珠峰"

```
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 ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
    neighbor ← a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
```


Hill-climbing search

- ❖ 存在的问题: 依赖于最初状态, 可能陷入局部最优



讨论

- ◆ 效率高，采用基本的爬山算法可以解决128皇后
- ◆ 文献中记载，可以解决3000, 000皇后
 - ◆ Rok Sasic and Jun Gu. 3,000,000 Queens in less than one minute. SIGART Bull. 1991
- ◆ 一般难以找到全局最优解
- ◆ 不用保存状态队列，每次仅在当前状态的局部范围内进行搜索，因此被归入局部搜索方法
- ◆ 受制于问题状态空间的形态
- ◆ 若有多个邻居具有相同的最高价值，则从中随机选择一个邻居继续搜索（注意重复）

改进策略

- ◆ 多次random restart
 - ◆ 若每次找到最优解的概率为 p ，则平均需要 $1/p$ 次restart就能找到最优解

改进策略 续

- ◆ 允许用价值相同的邻居取代当前状态
sideways move

- ◆ 要注意避免死循环
- ◆ 一般要限制连续sideways move

