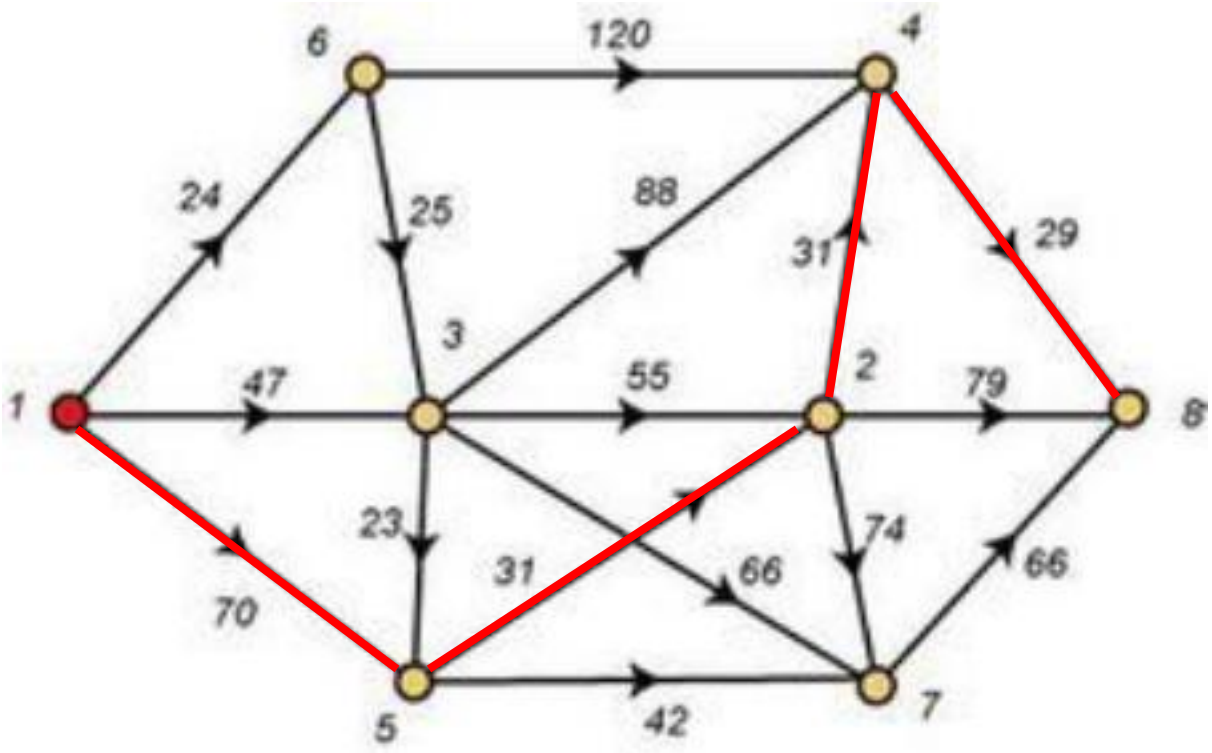
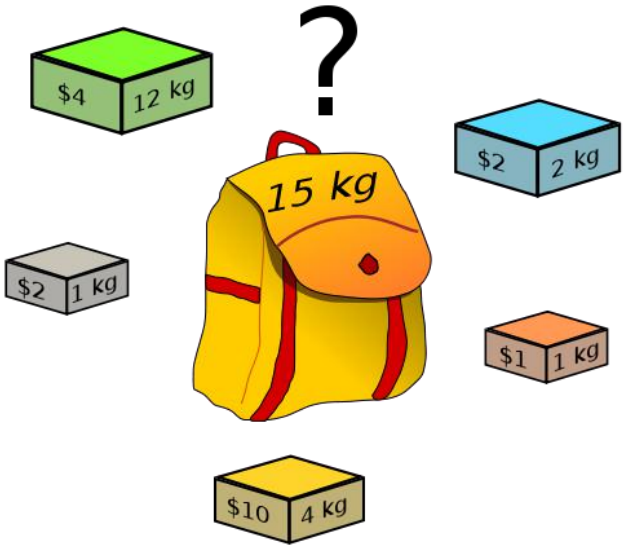
搜索问题

遍历搜索

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$w_0, ..., w_{n-1}$
  - 各自的价值分别为$v_0, ..., v_{n-1}$
- 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

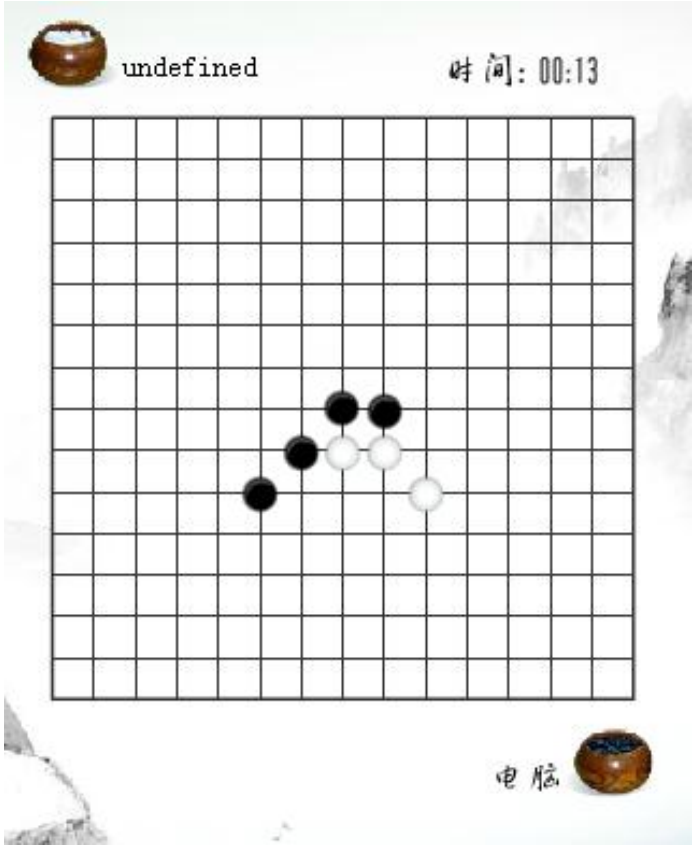
目标：请让左右两边河墩上的青蛙交换位置。

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

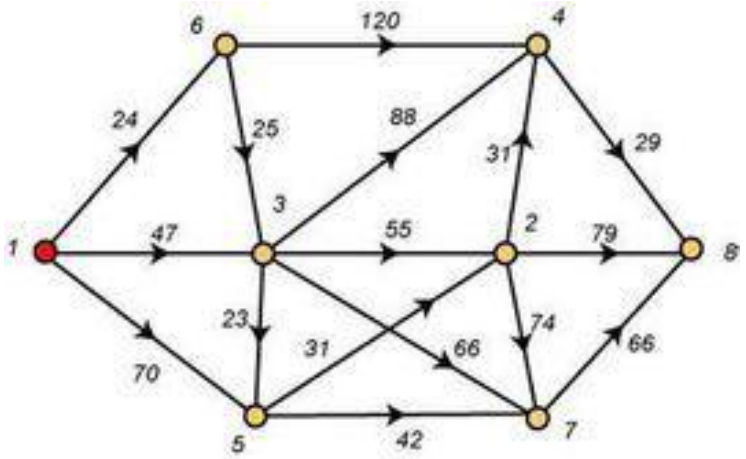
- 搜索问题一般通过以下几个属性来定义:
  - 状态
  - 初始状态
  - 动作
  - 转移模型
  - 目标测试
  - 路径成本（可选）

思考：

s=<1,3,4>

接收到动作 move(4,2)，

s’ =?

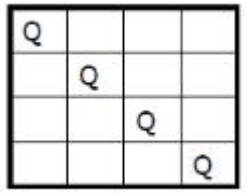


- **状态**：一个节点序列，如<v_1, v_2, v_3>,表示经过的路径；序列最后一个节点表示当前的位置。
- **初始状态**：序列只有一个节点，表示出发位置。
- **动作**：move（v_i, v_j），从一个节点移动到另一个节点。图中须有一条边从v_i指向 v_j。
- **转移模型**：状态s，接受到动作move（v_i, v_j），若s的最后一个节点等于v_i,则转移到状态s'，s'是在s的末尾加上v_j；否则不发生状态改变。
- **目标测试**：有一个目标节点 v_e，一个状态是目标状态当且仅当其末尾的节点等于v_e。
- **路径成本**：一个状态的路径成本等于其序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

思考：

s=<0,1,2,3>

接受到动作 set(1,2)，s’ =? s’ 是目标吗?



- **状态**：一个四维向量<x_1, x_2, x_3, x_4>，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<0, 1, 2, 3>。
- **初始状态**：可以是任意一个状态。
- **动作**：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 <x0,....>
每一项取值 0, 1, -1
1 绿色
0 棕色
-1 空位

初始状态 <1,1,1,-1,0,0,0>

目标
<000-1111>

动作
jump(crntPos, 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 <x0,....>

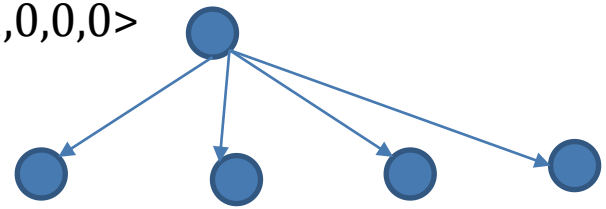
每一项取值 0, 1, -1

1 绿色

0 棕色

-1 空位

<1,1,1,-1,0,0,0>



<1,1,-1,1,0,0,0>

初始状态 <1,1,1,-1,0,0,0>

目标

<000-1111>

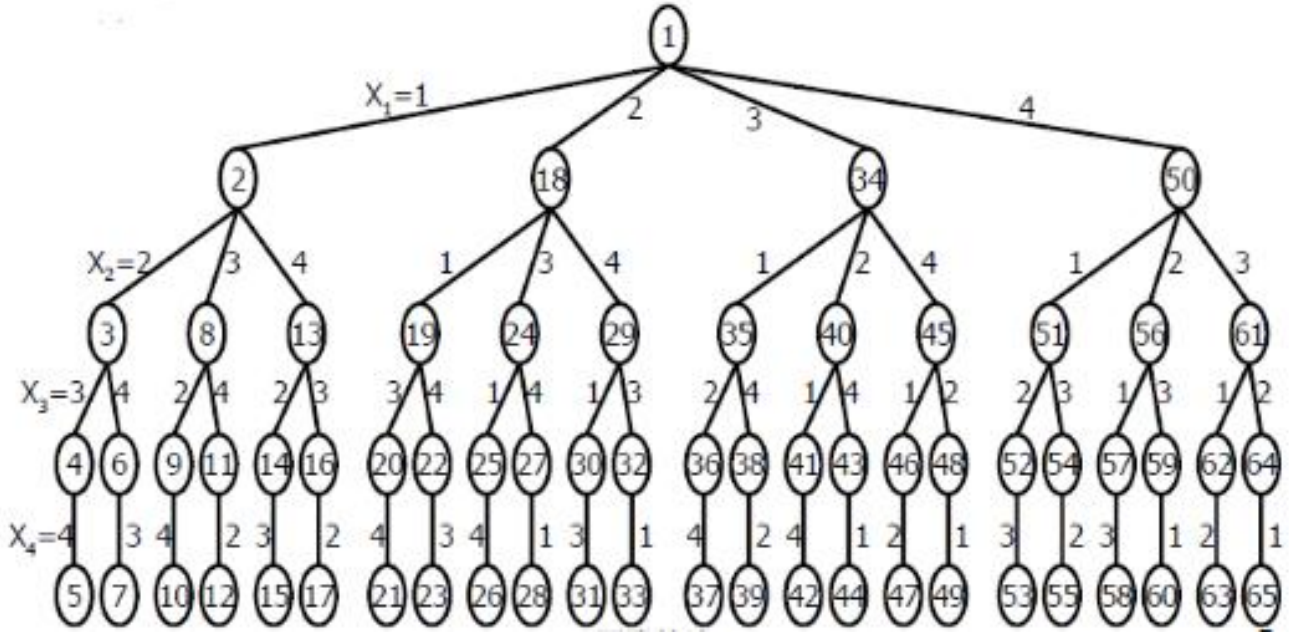
动作

jump(crntPos, 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）

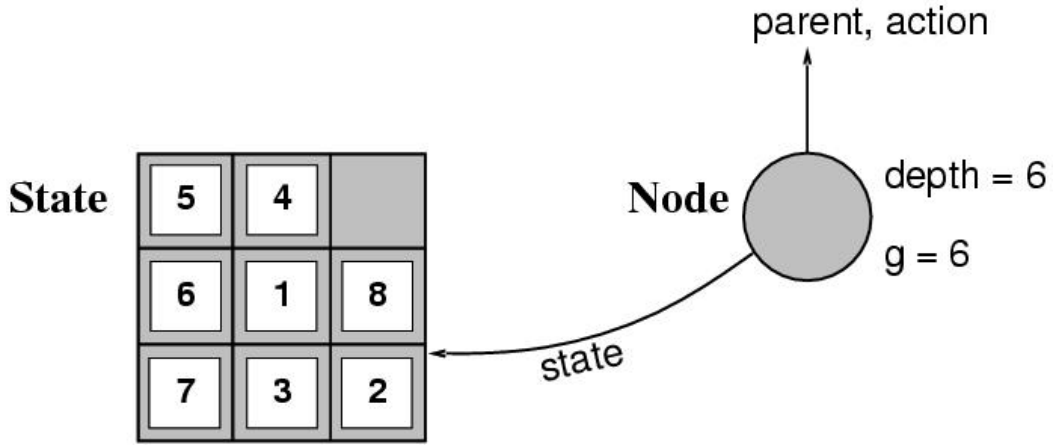
树搜索的通用框架

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

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

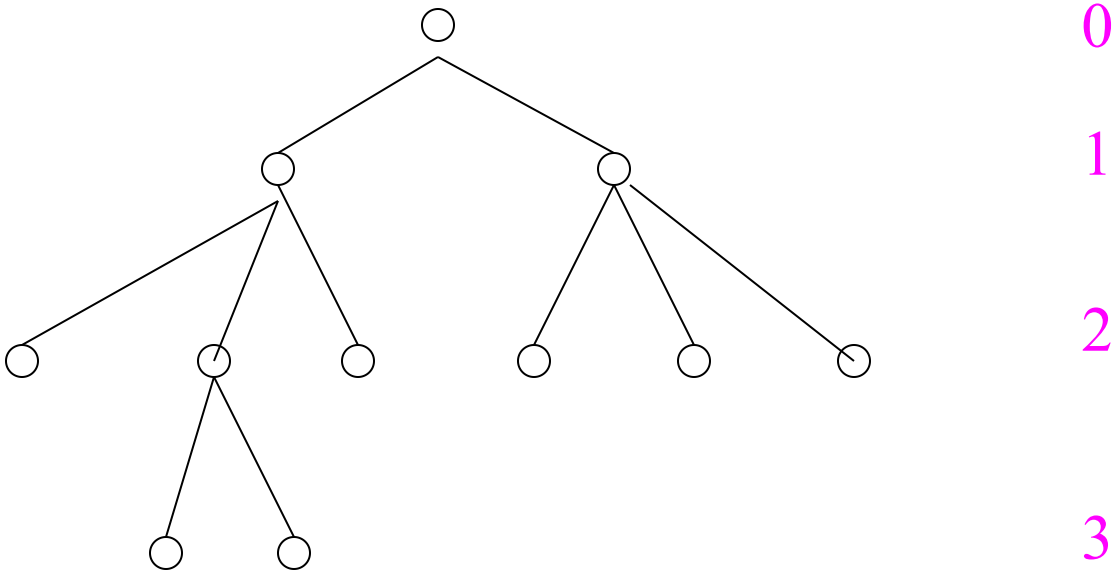
- 概念：blind search or uninformed search
  - 除问题定义之外，搜索策略不能获得状态的其他信息，只能通过产生新的状态并检测目标状态来解决问题。
- 策略分类
  - 根据状态空间中节点展开的顺序来对盲搜策略分类
    - 广度优先搜索，深度优先搜索，有深度限制的搜索，一致代价搜索，迭代扩展深优搜索，双向搜索

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$(n_0, n_1,...,n_k)$，对于i=1,...,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）

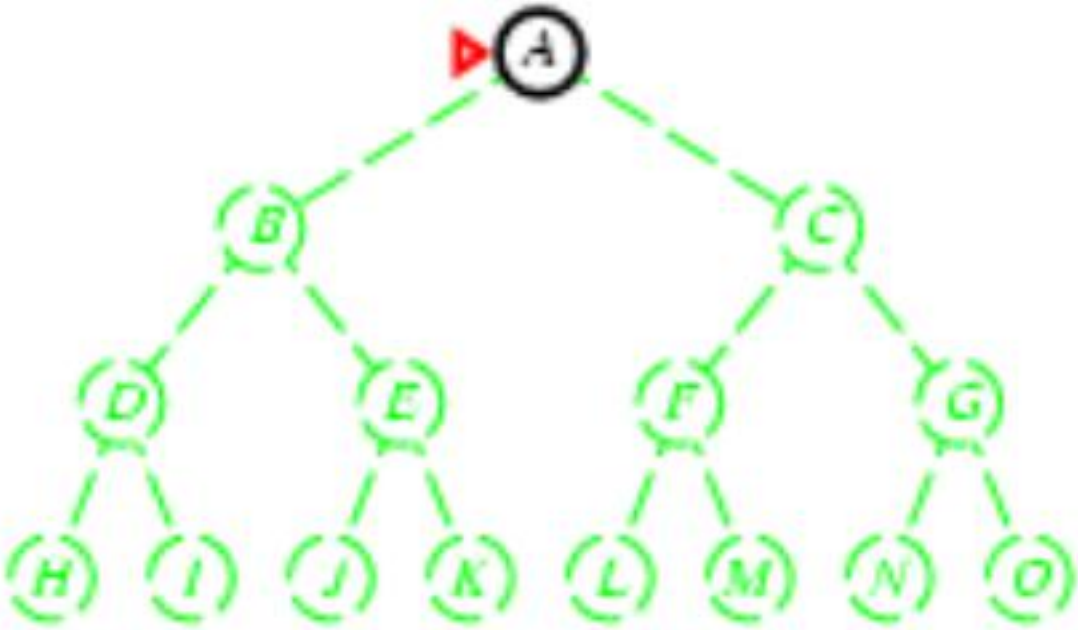
- 扩展一个节点

  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

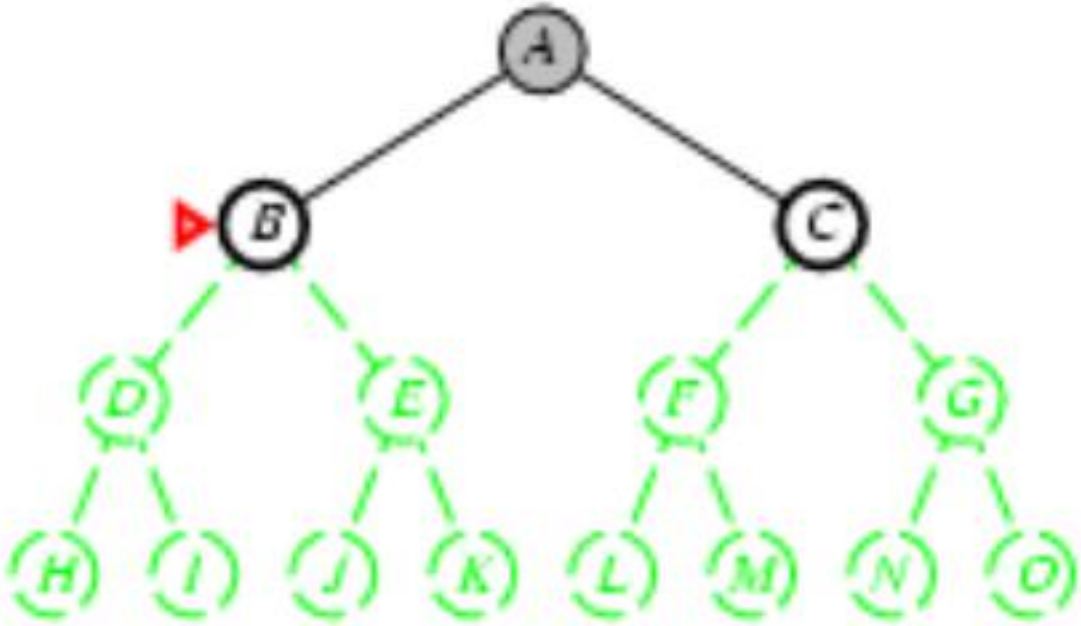
Depth-first search （DFS）

- 优先扩展和搜索在搜索树中处于最低层的节点



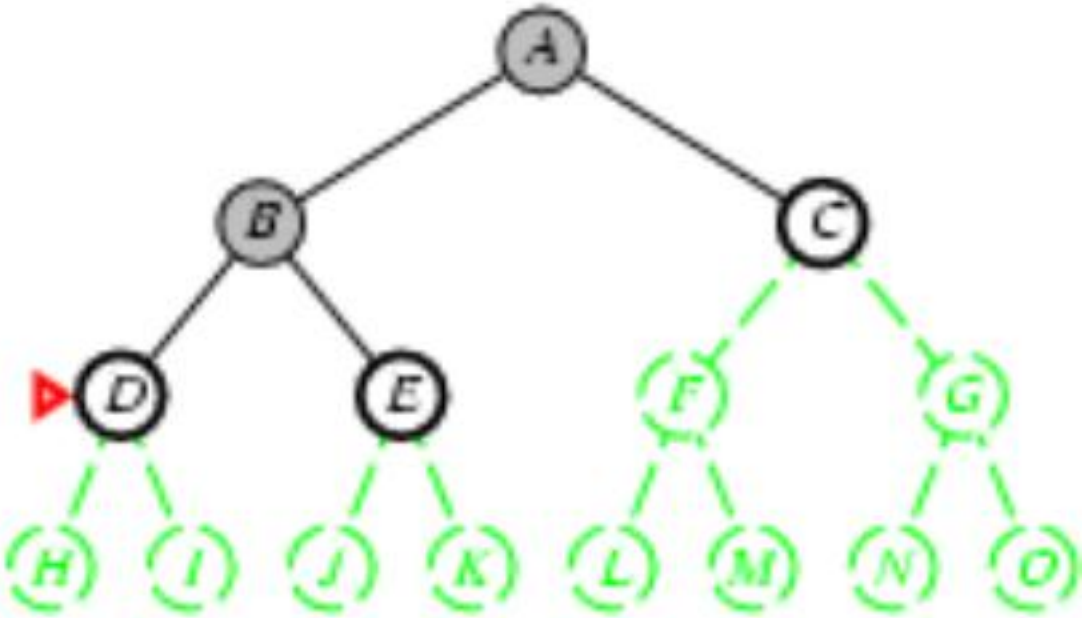
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



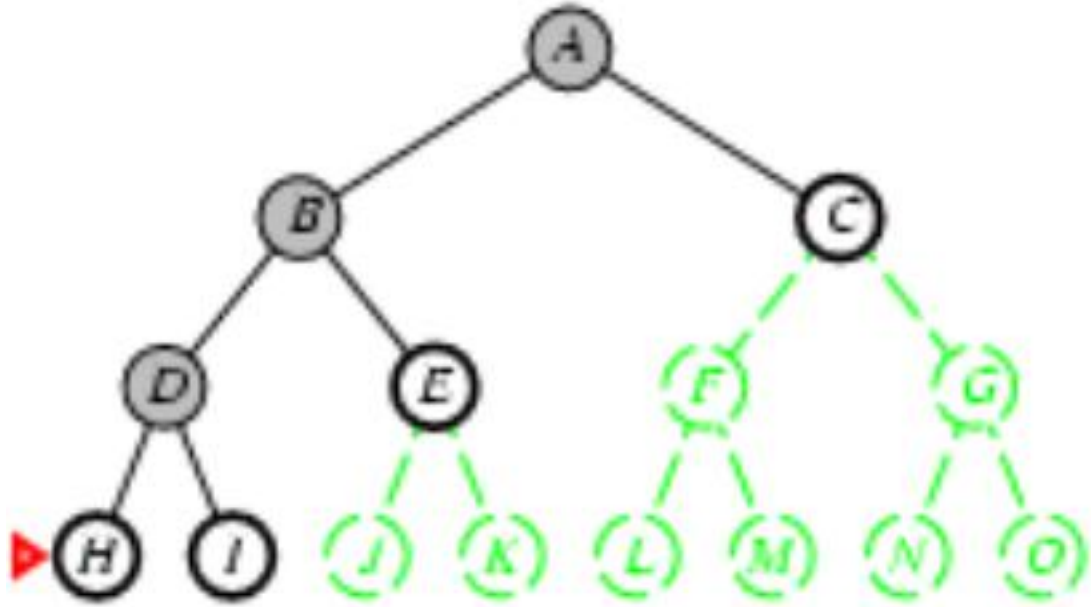
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



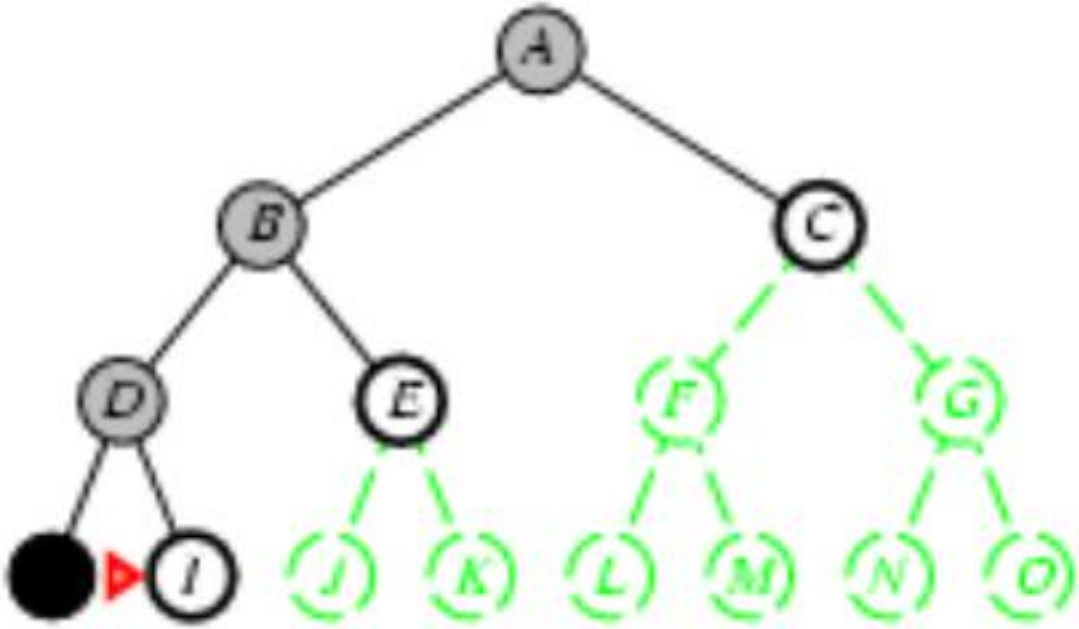
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



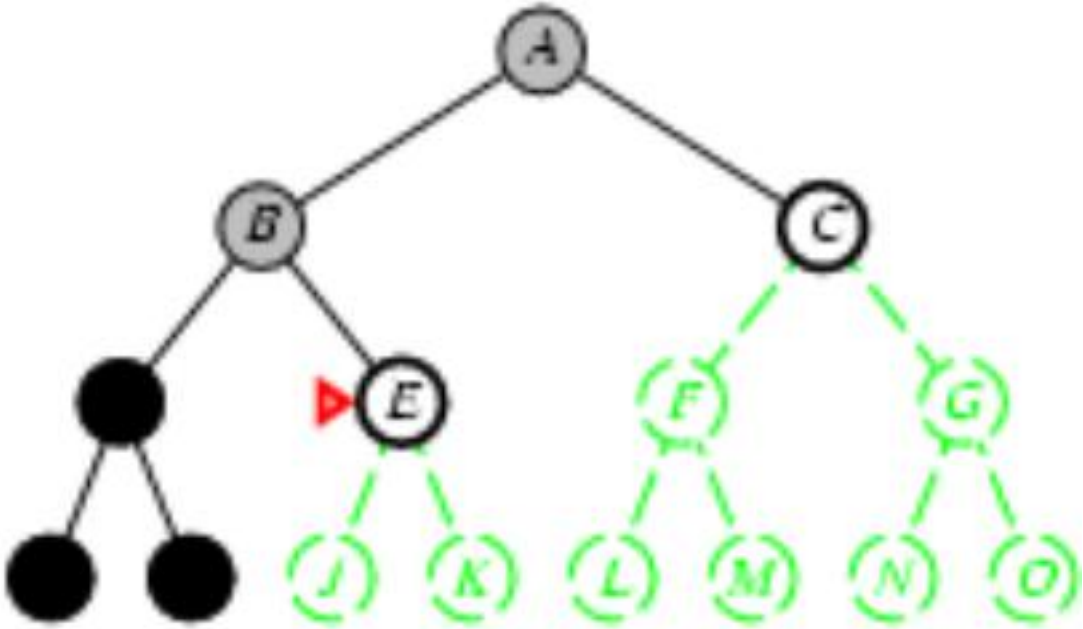
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



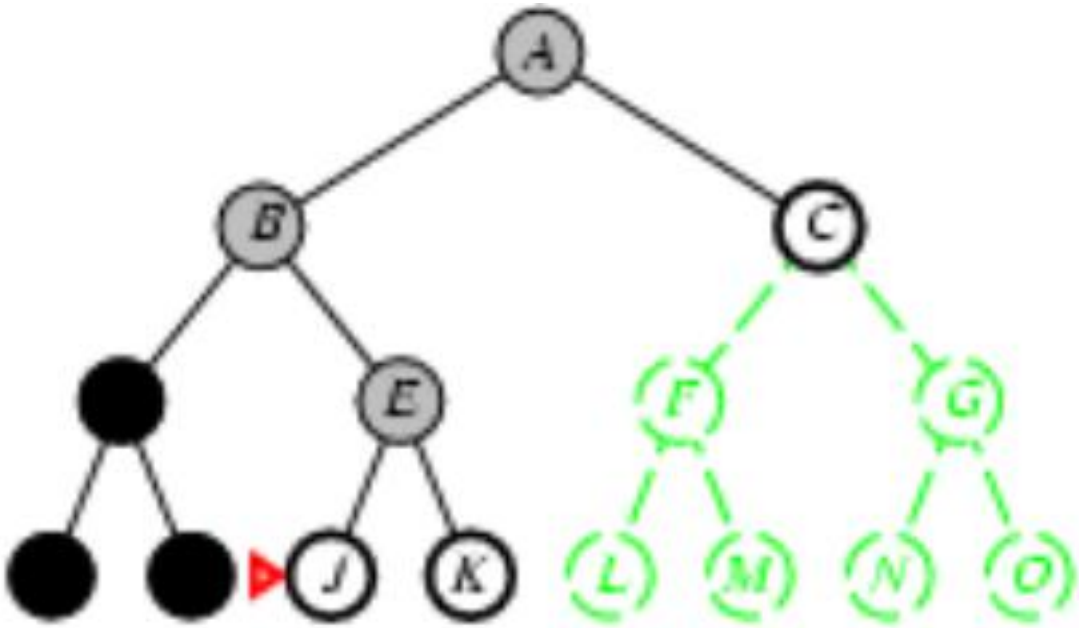
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



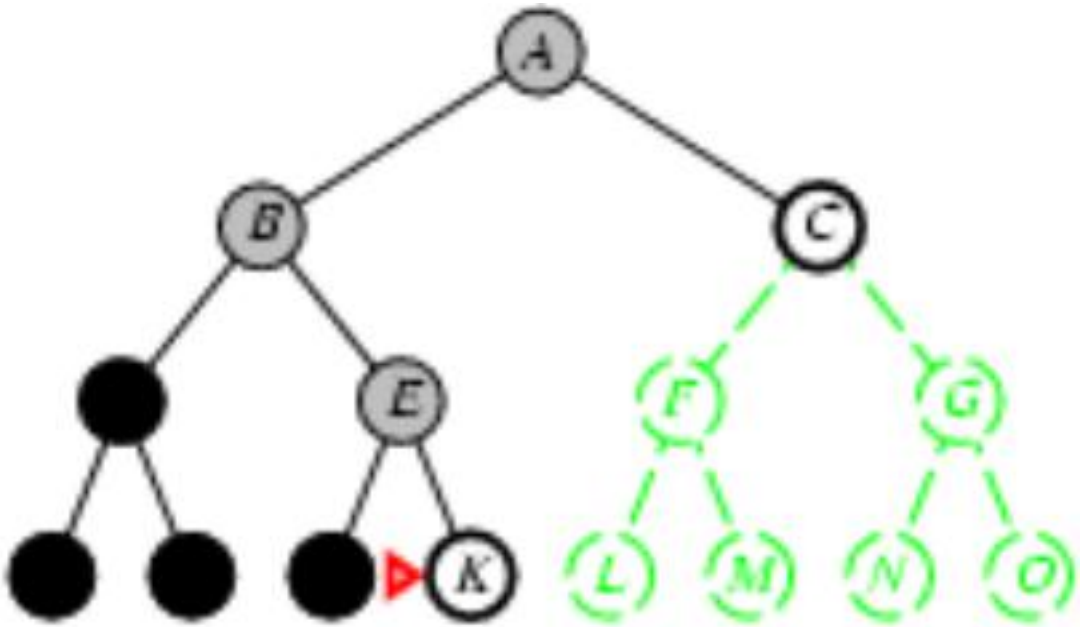
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



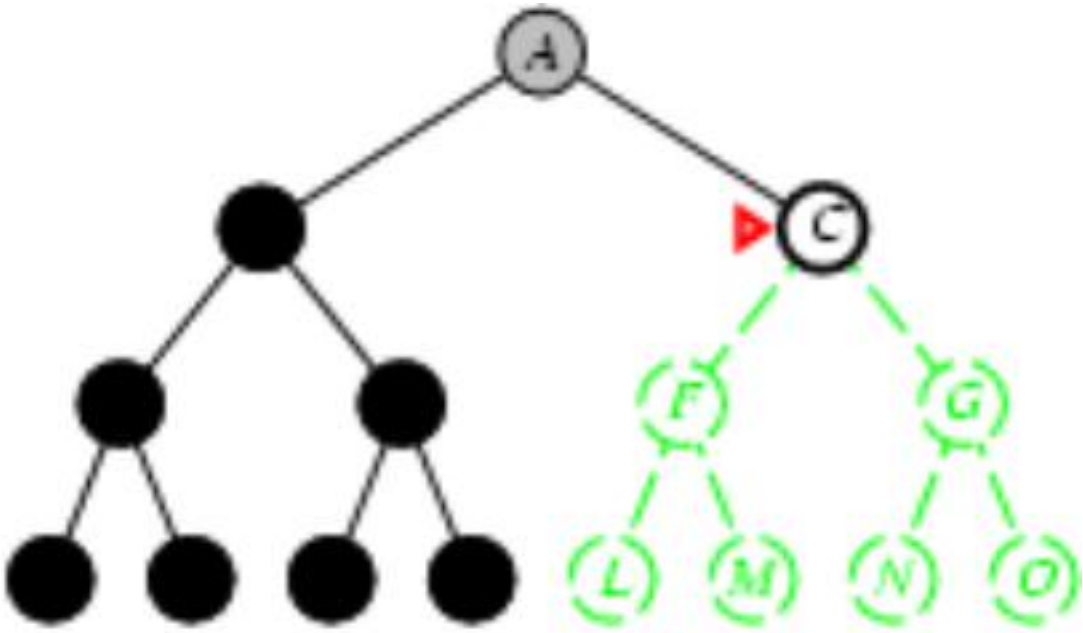
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



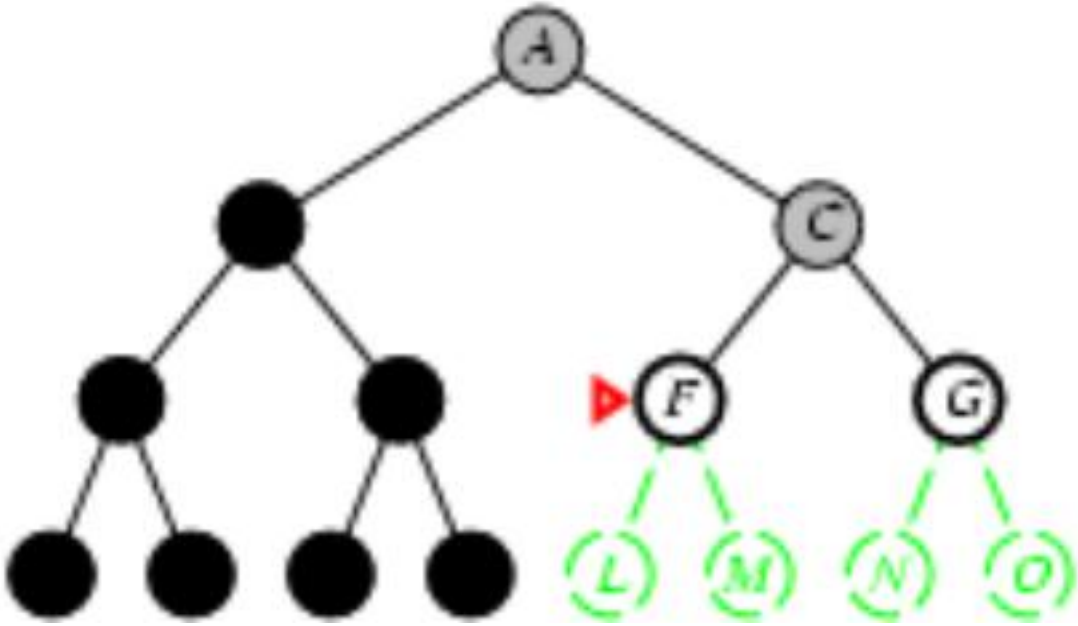
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



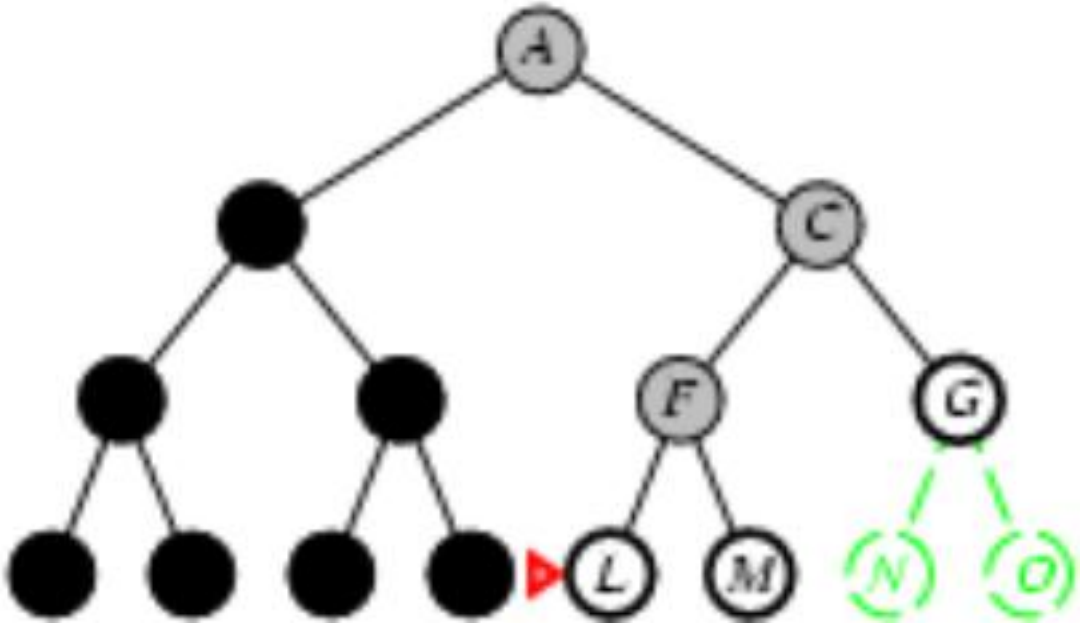
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



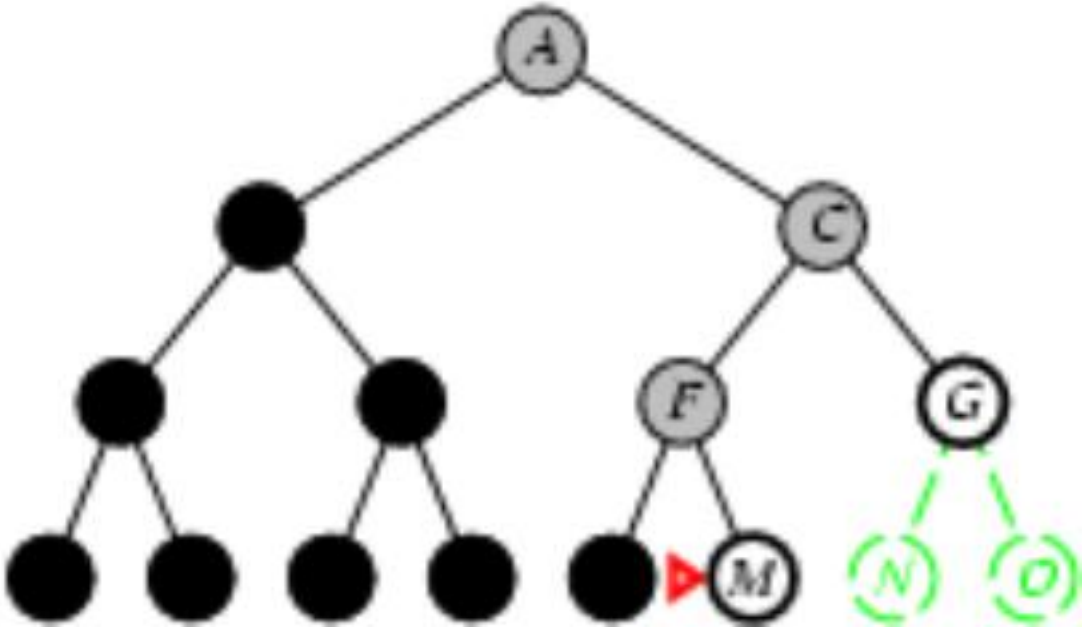
Depth-first search

- 优先扩展和搜索在搜索树中处于最低层的节点



Depth-first search

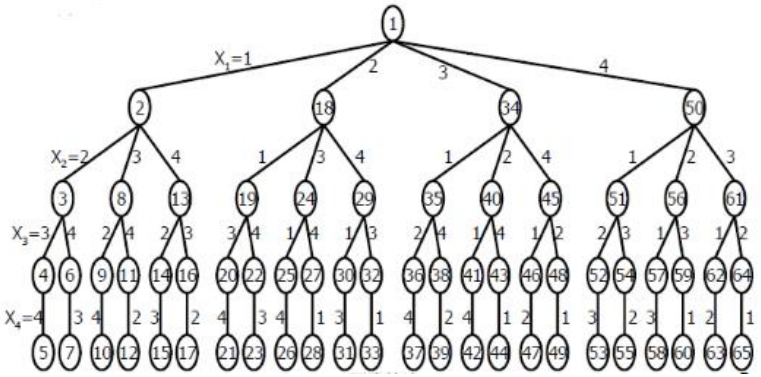
- 优先扩展和搜索在搜索树中处于最低层的节点



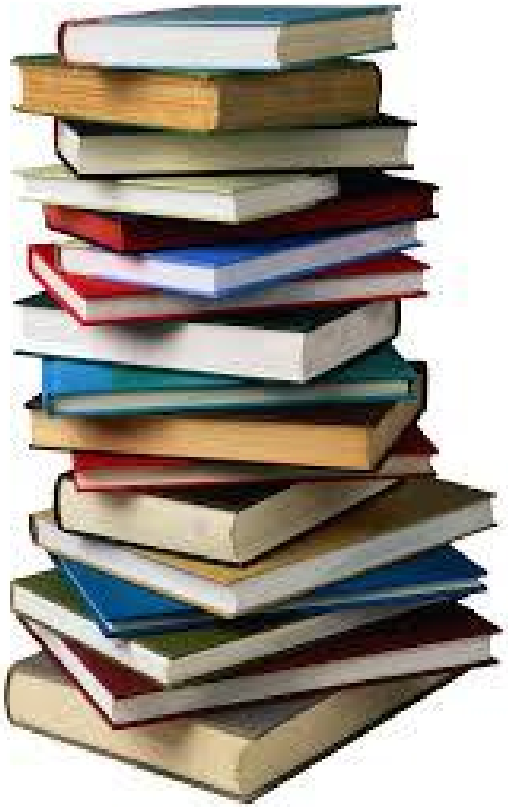
8皇后深度优先搜索的简单实现

```
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?

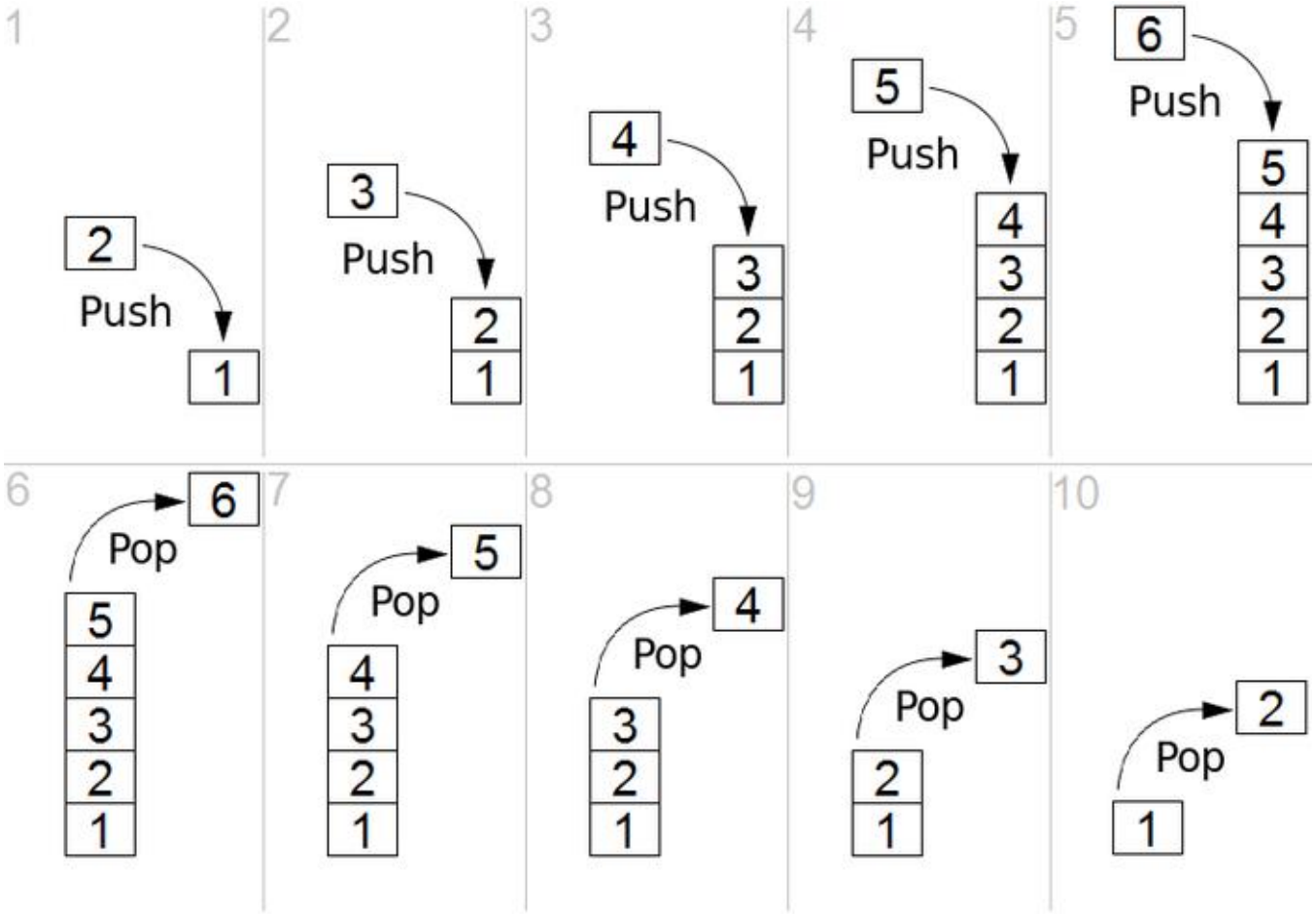


栈 stack



栈 stack

- 特点： last in, first out (LIFO)
- 操作
  - pop
  - push
  - top



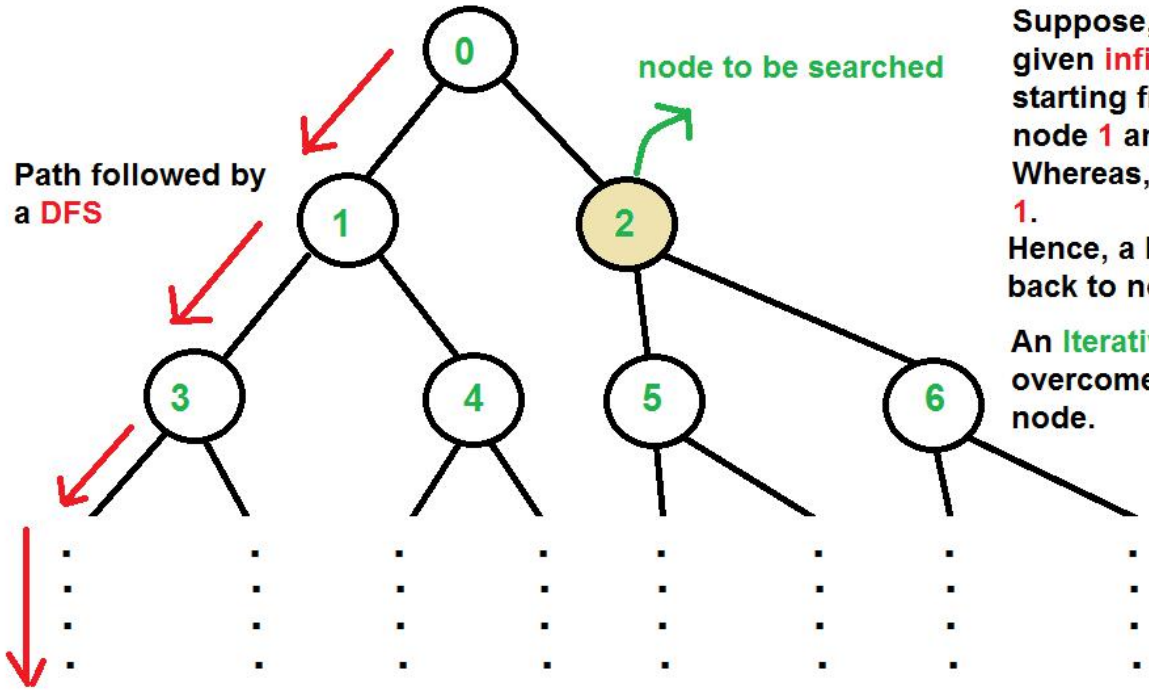
基于栈的DFS算法

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 failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

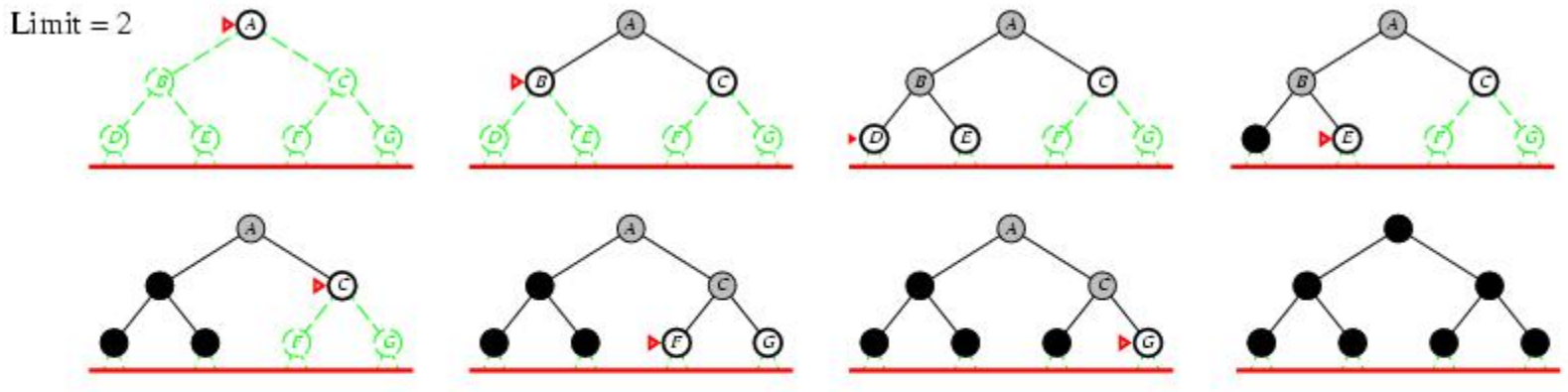
Since iterative deepening visits states multiple times, it may seem wasteful, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level, so it does not matter much if the upper levels are visited multiple times

Iterative deepening DF search *l* =0

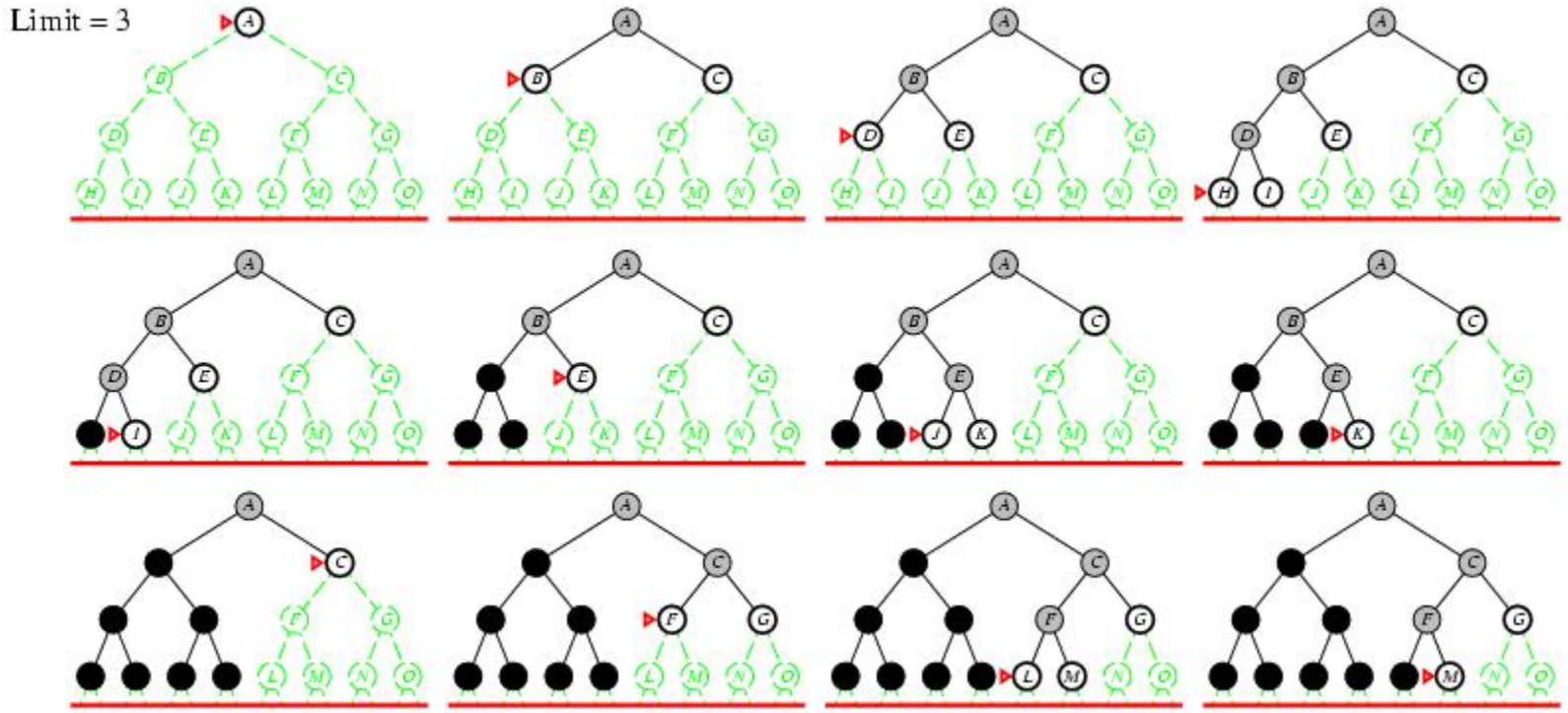
Limit = 0

Iterative deepening DF search *l* =1

Iterative deepening DF search *l* =2



Iterative deepening DF search *l* =3



Iterative deepening search

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

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

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

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

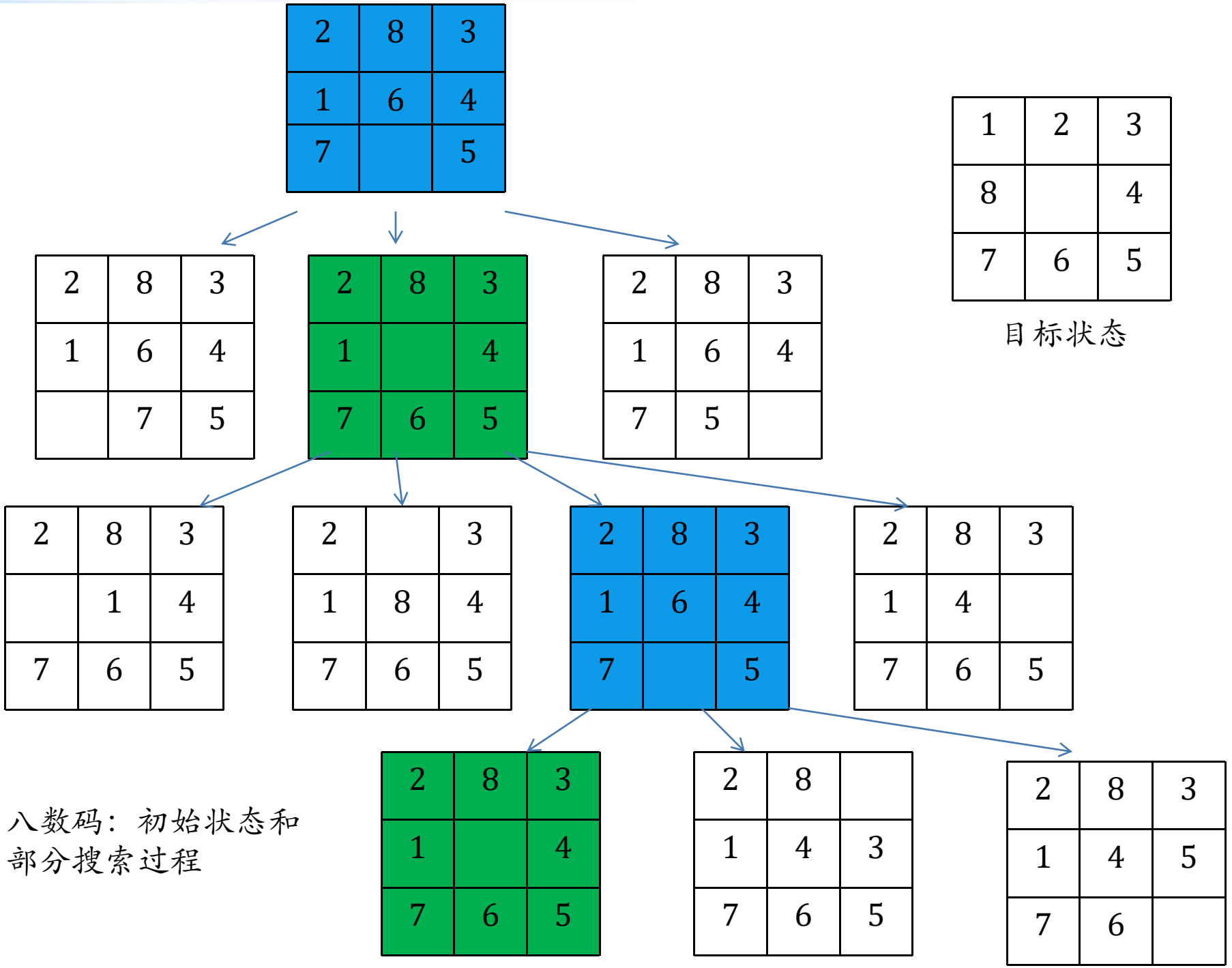
- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- 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 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 5
7 6
2 8
1 4 3
7 6 5
八数码：初始状态和
部分搜索过程

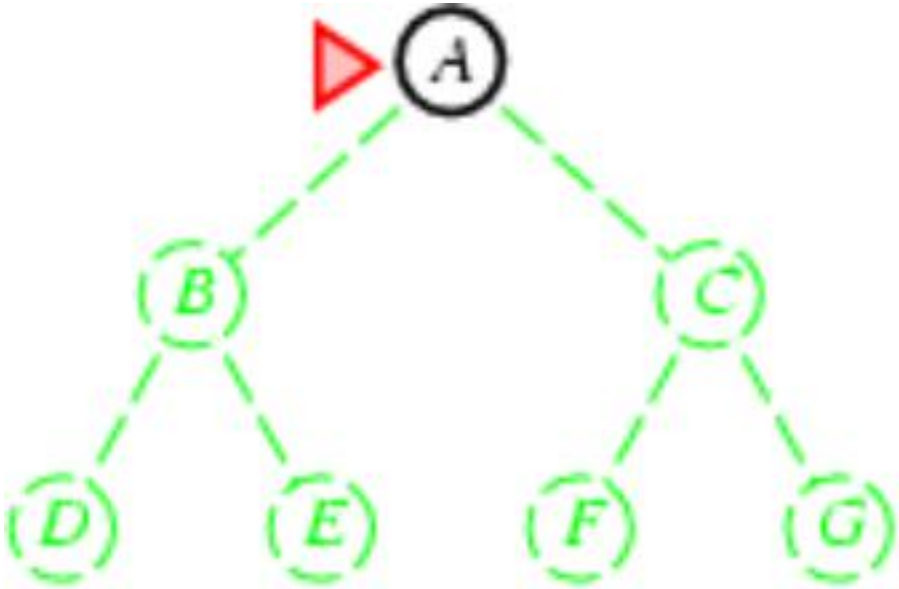
重复节点的判断

- 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）

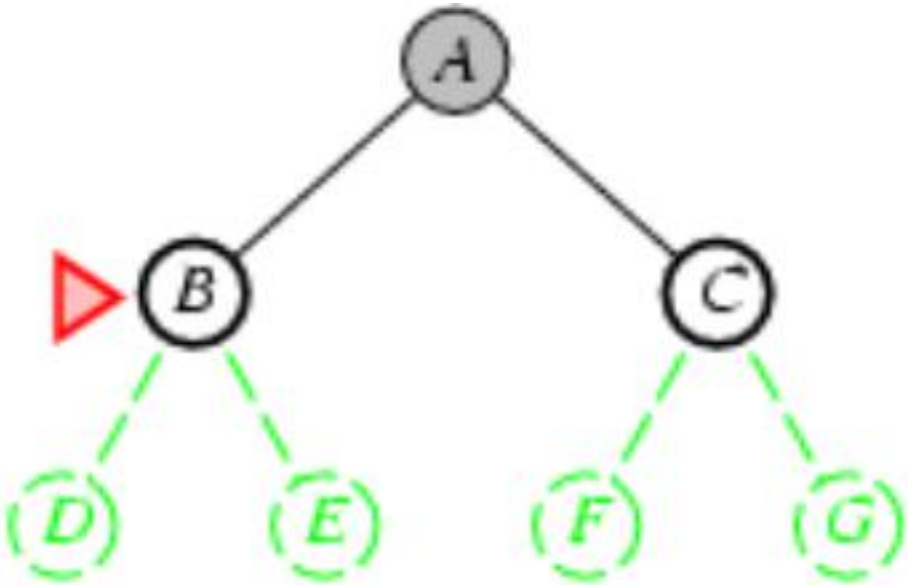
- 节点的生成和扩展是按层次顺序进行的，处于搜索树较浅层次的节点总是先被检查和扩展
- 节点搜索的先后顺序与它们生成顺序一致
- 一般使用队列作为存储

Breadth-first search



状态队列：A

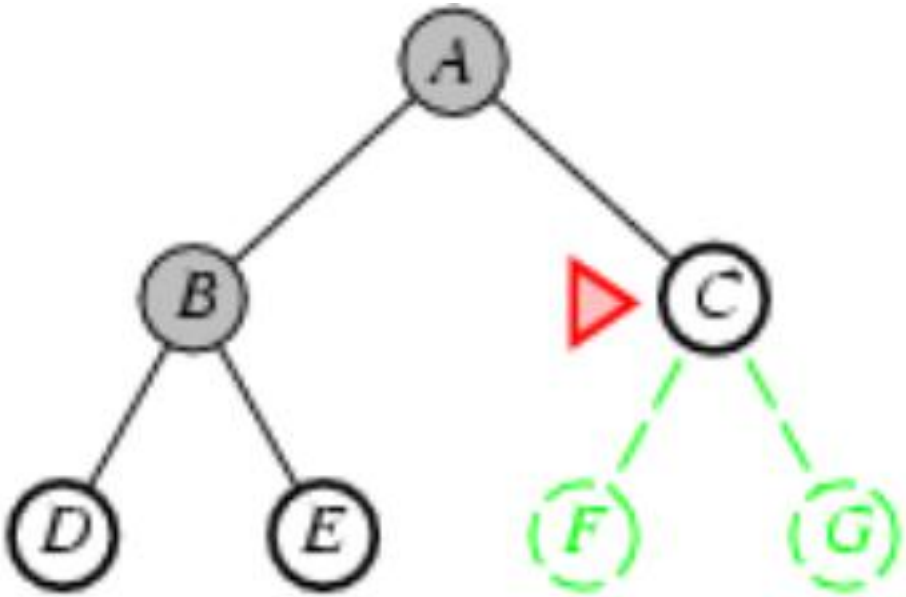
Breadth-first search



状态队列： C, B, ~~A~~

front

Breadth-first search



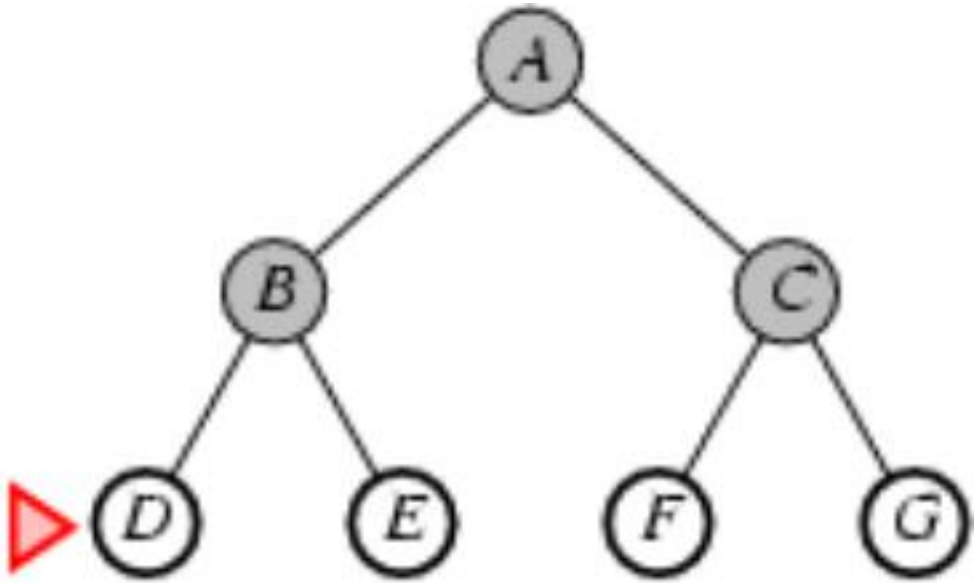
状态队列： E, D, C, ~~B~~, ~~A~~

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~~

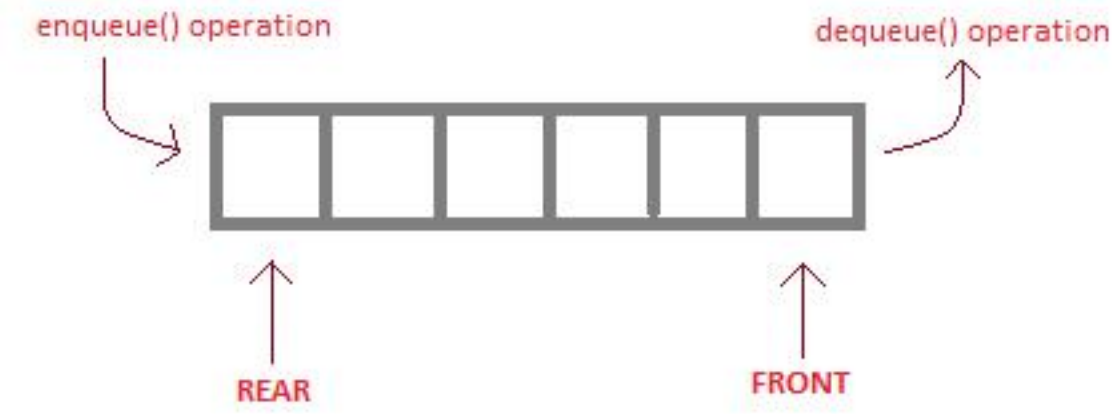
G, F, E, ~~D, C, B~~, ~~A~~

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

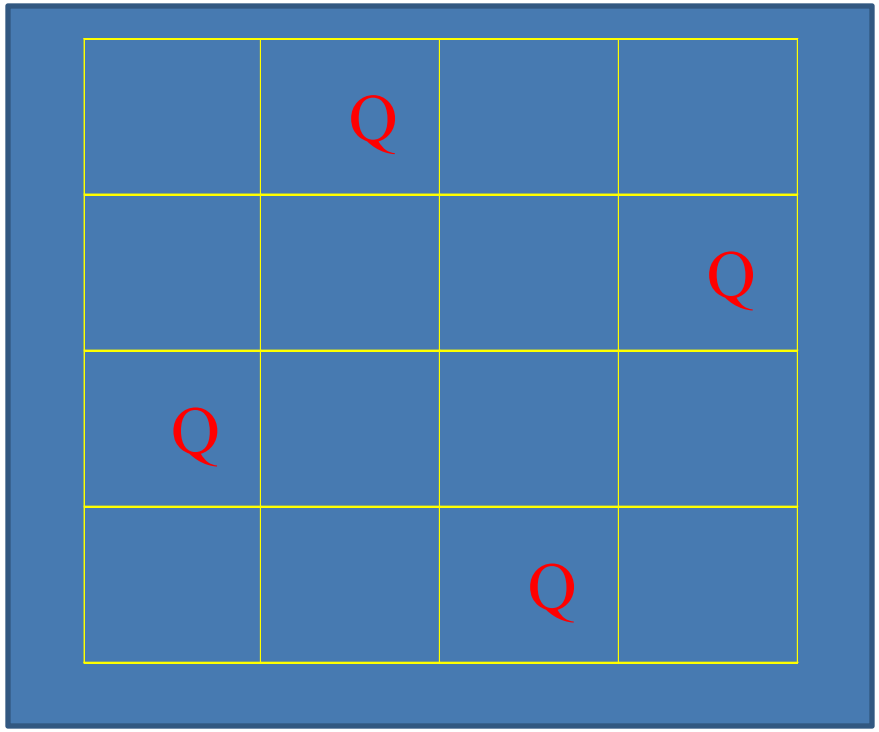
八皇后问题BFS算法分析

- 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

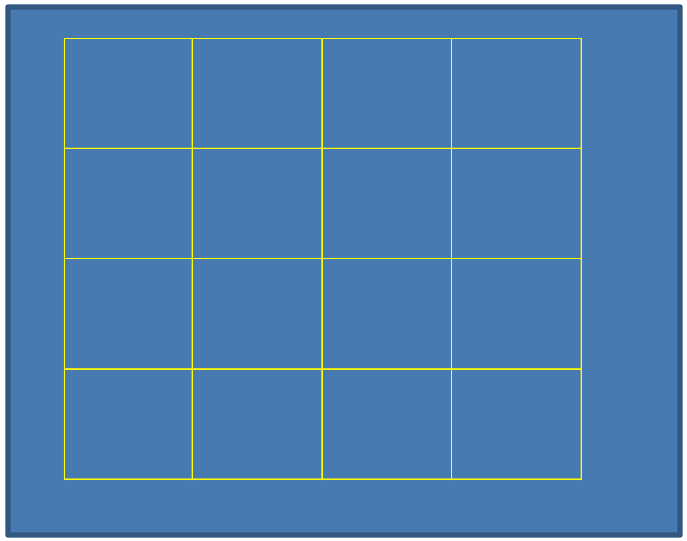
回溯

- 例：皇后问题

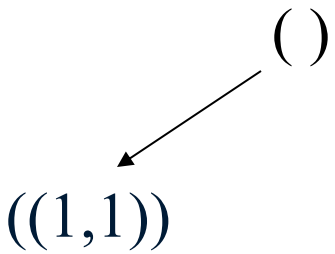


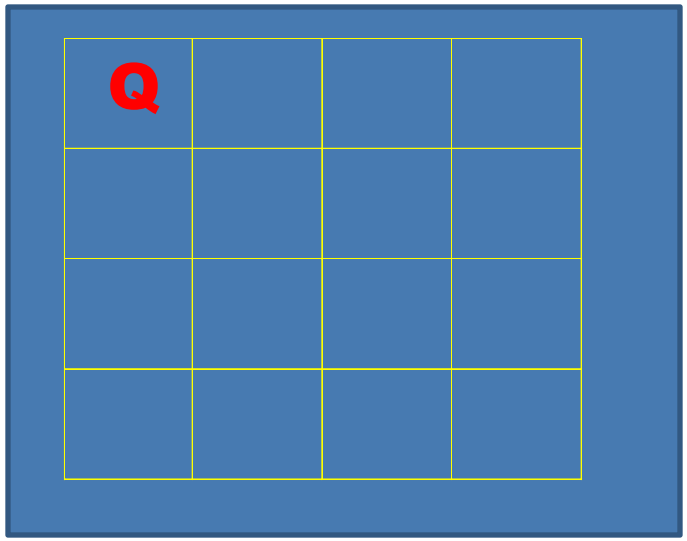
采用逐行放皇后的搜索策略

( )

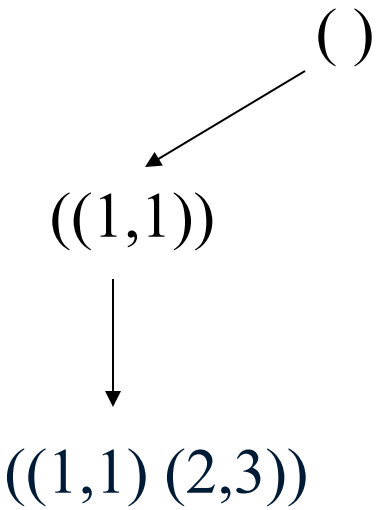


采用逐行放皇后的搜索策略

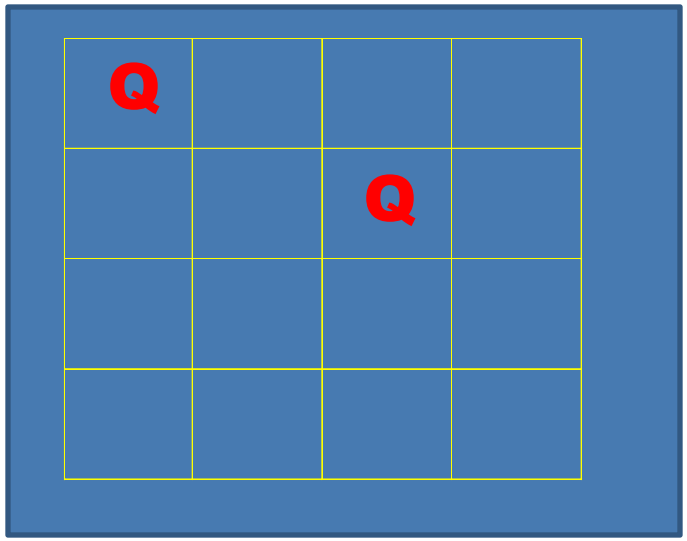


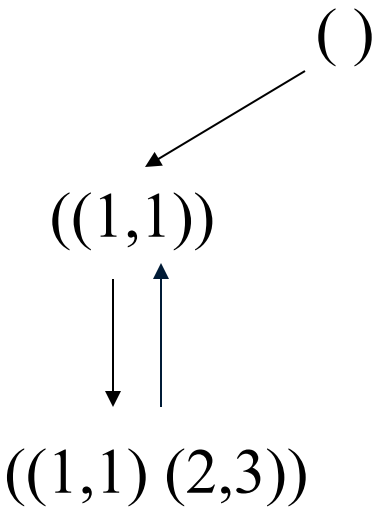


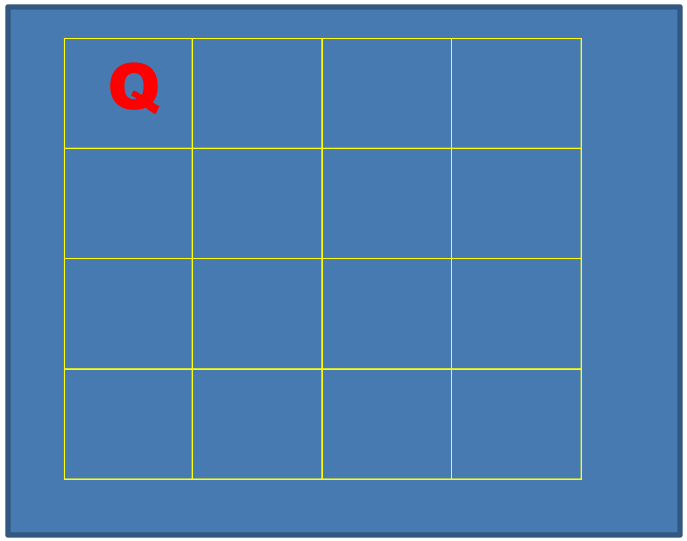
采用逐行放皇后的搜索策略

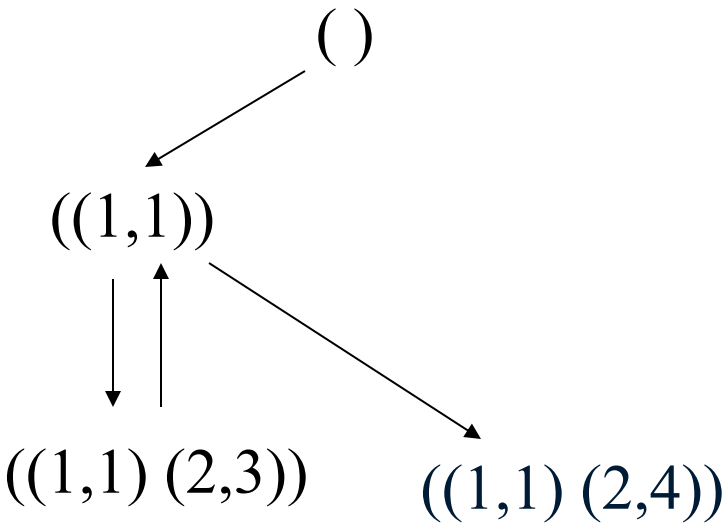


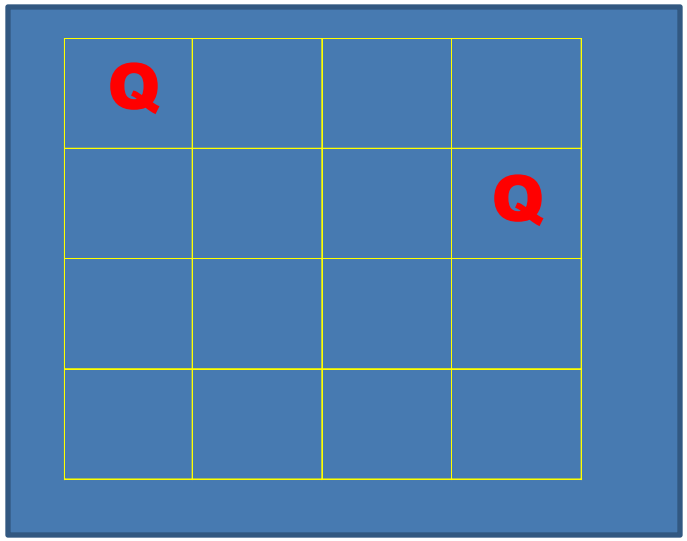
FAIL

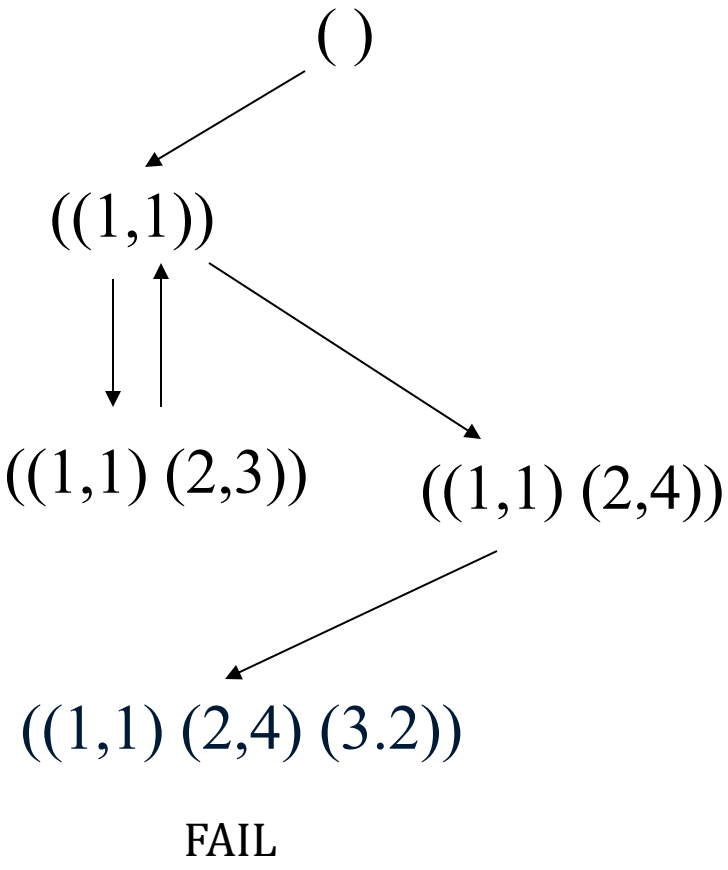


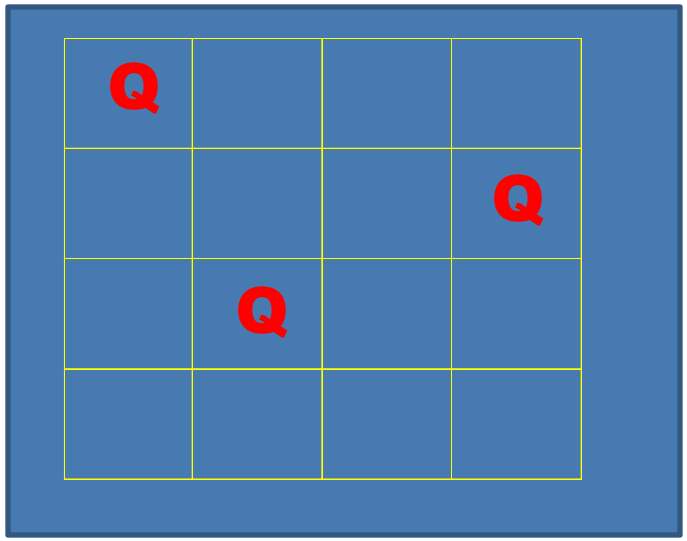

( )
((1,1))
((1,1) (2,3))

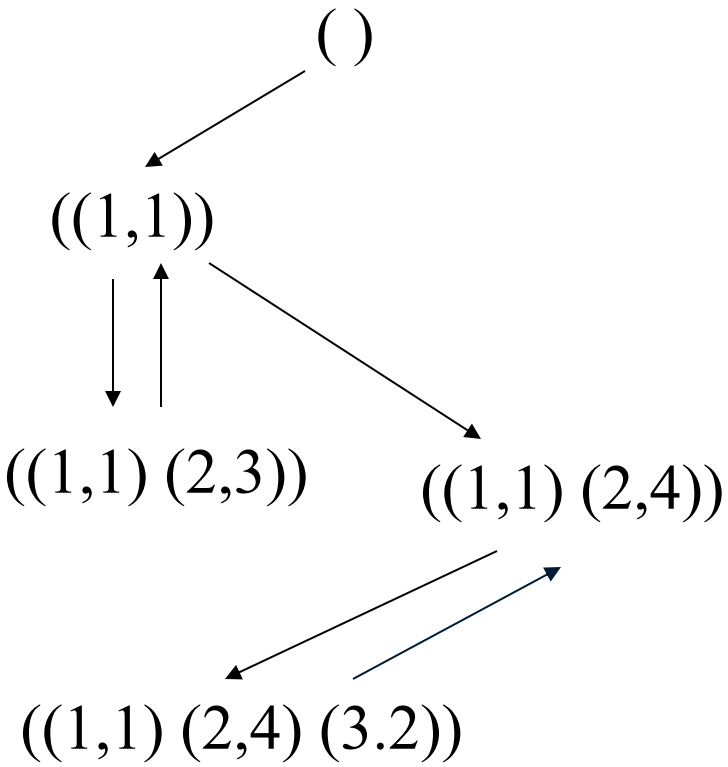

Q

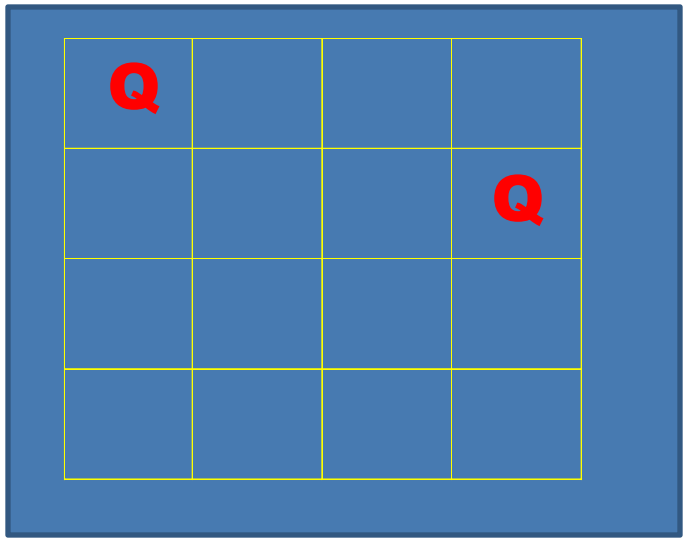

( )
((1,1))
((1,1) (2,3))
((1,1) (2,4))

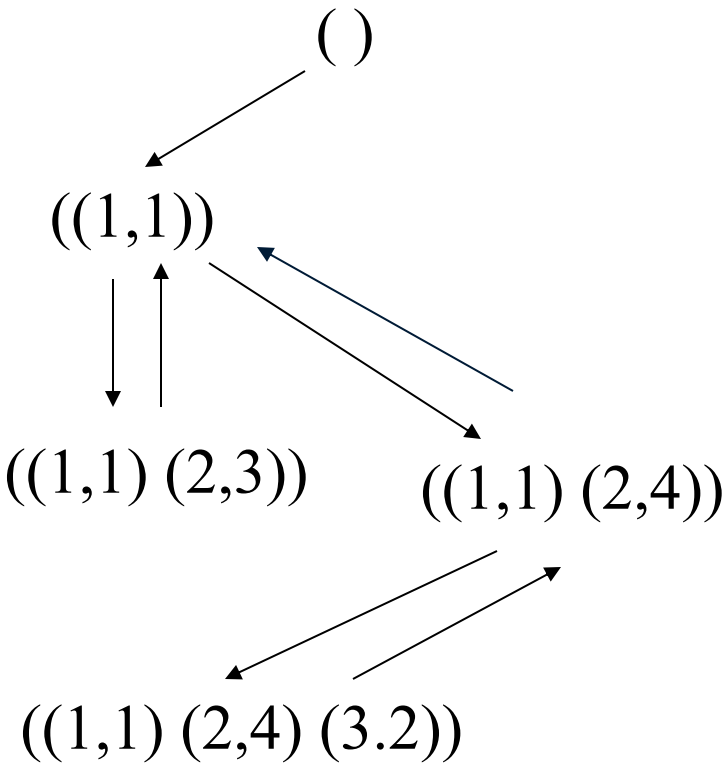

Q
Q

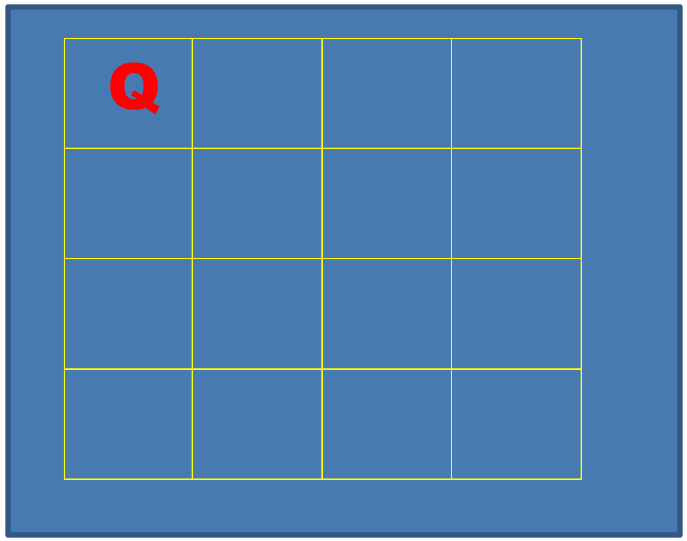

( )
((1,1))
((1,1) (2,3))
((1,1) (2,4))
((1,1) (2,4) (3.2))
FAIL

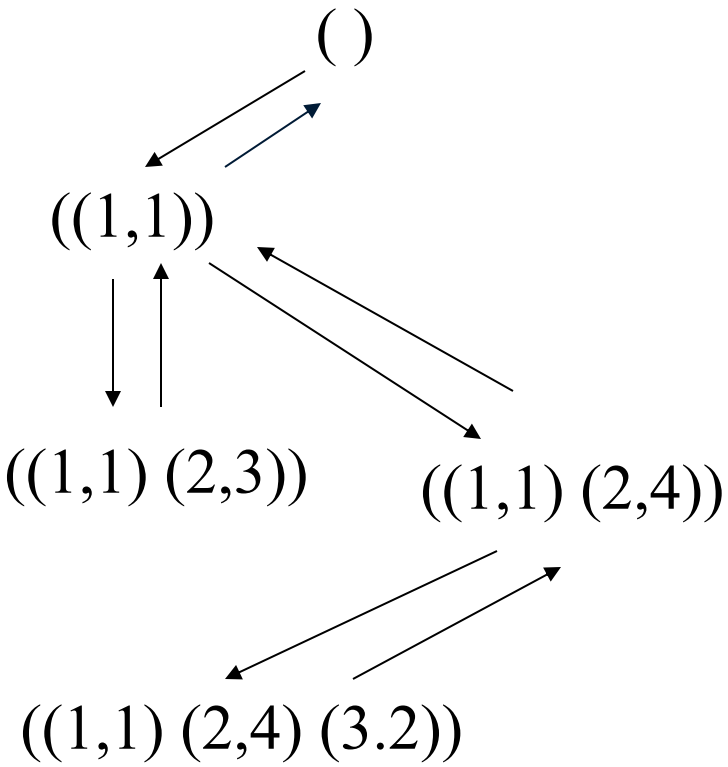

Q
Q
Q

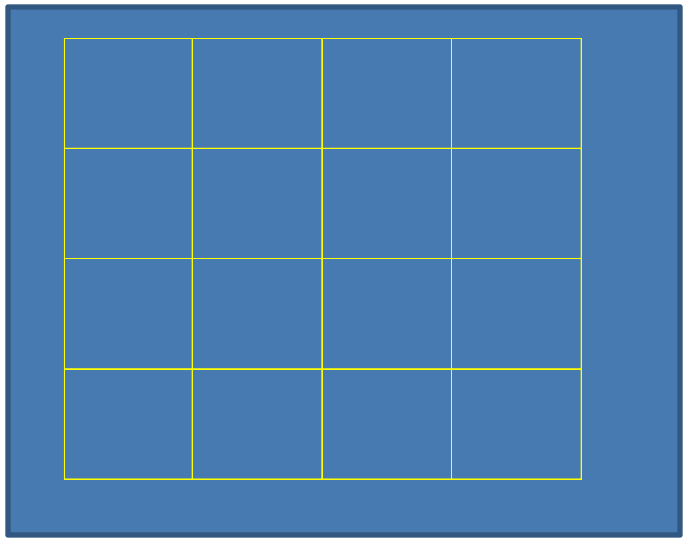

( )
((1,1))
((1,1) (2,3))
((1,1) (2,4))
((1,1) (2,4) (3.2))

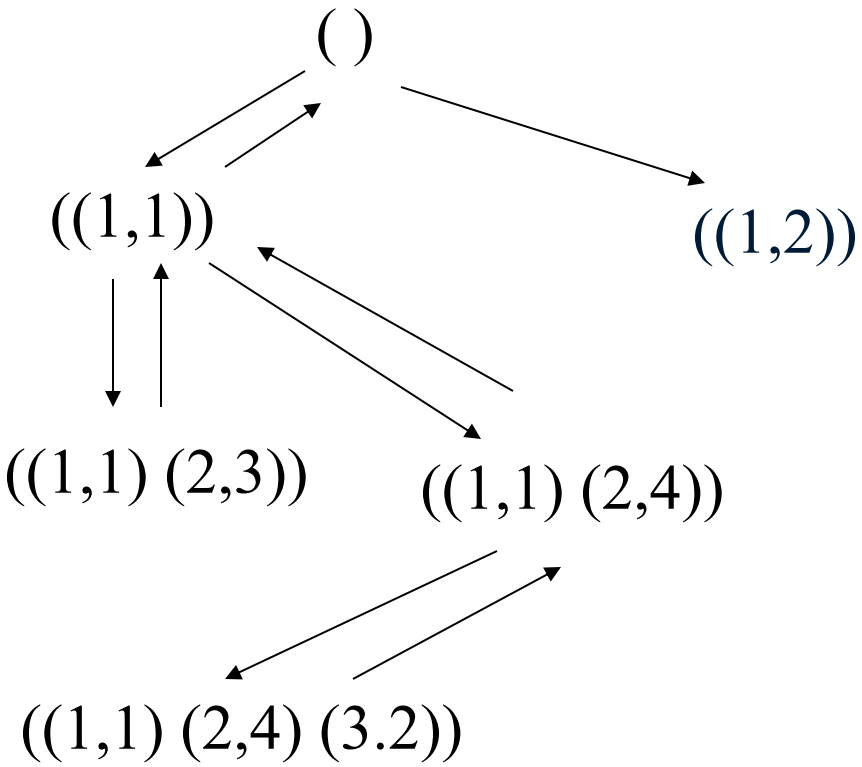

Q
Q

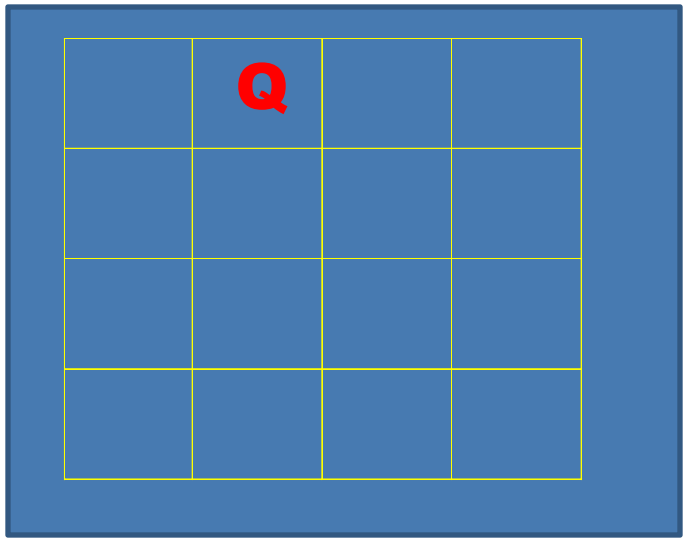

( )
((1,1))
((1,1) (2,3))
((1,1) (2,4))
((1,1) (2,4) (3.2))

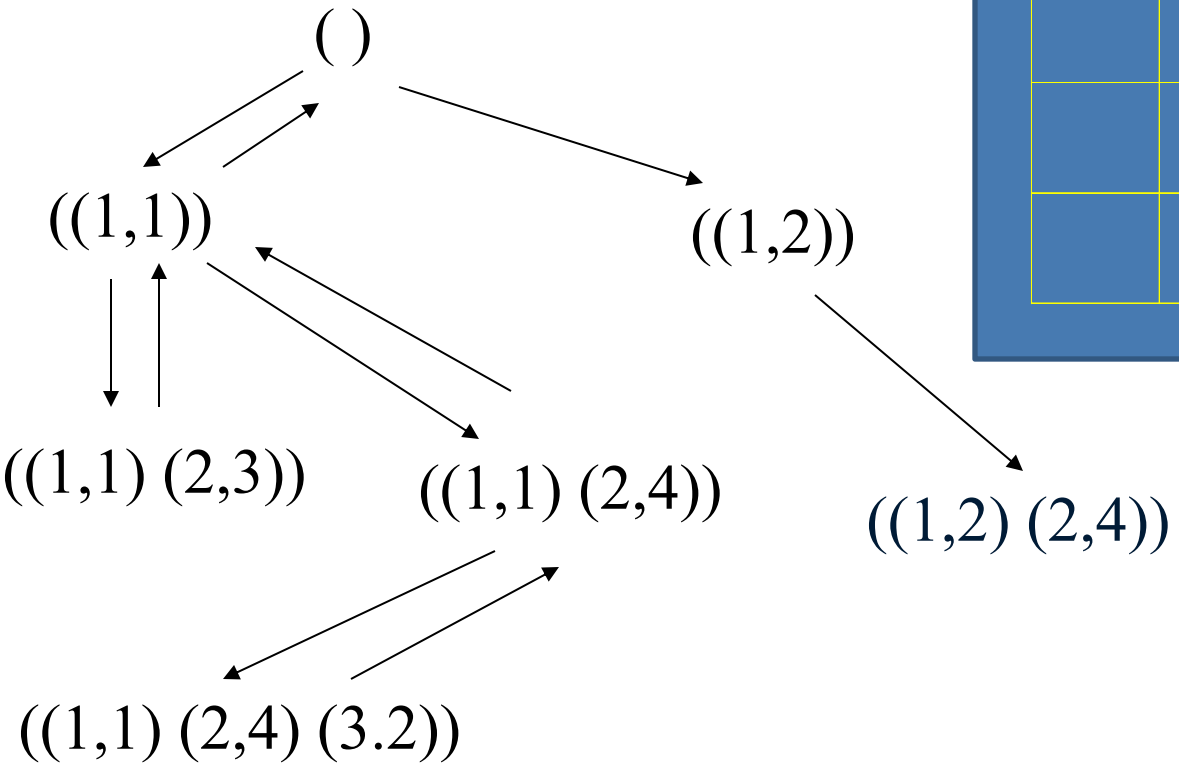

Q

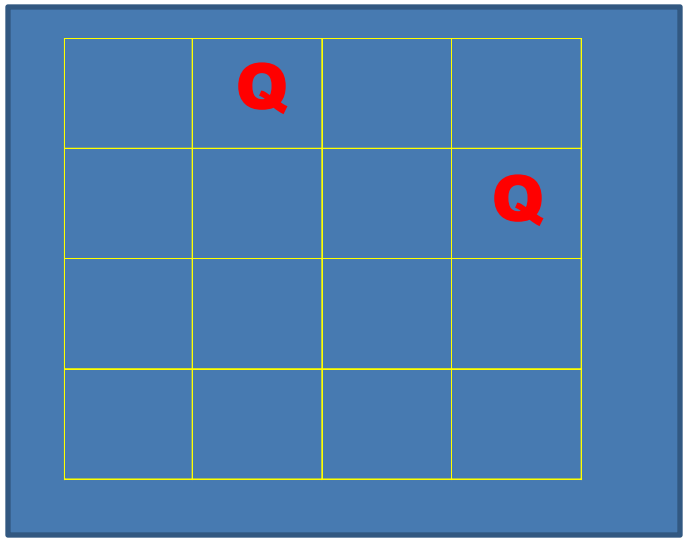

( )
((1,1))
((1,1) (2,3))
((1,1) (2,4))
((1,1) (2,4) (3.2))

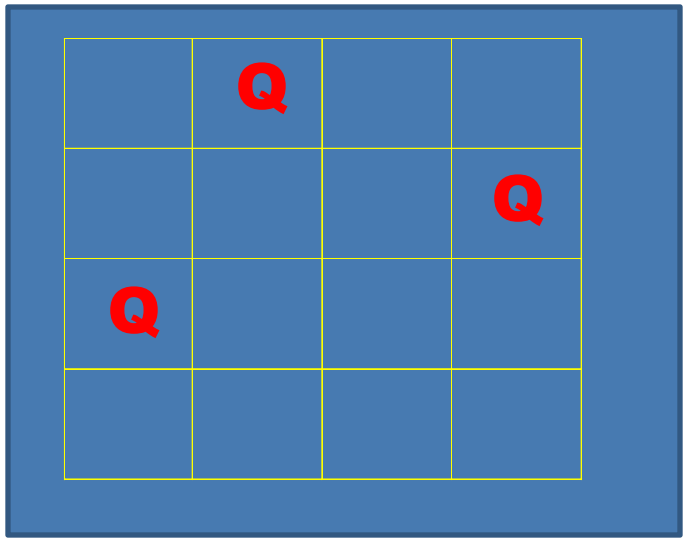



( )
((1,1))
((1,2))
((1,1) (2,3))
((1,1) (2,4))
((1,1) (2,4) (3.2))


Q


( )
((1,1))
((1,2))
((1,1) (2,3))
((1,1) (2,4))
((1,2) (2,4))
((1,1) (2,4) (3.2))


Q
Q



( )

((1,1))

((1,2))

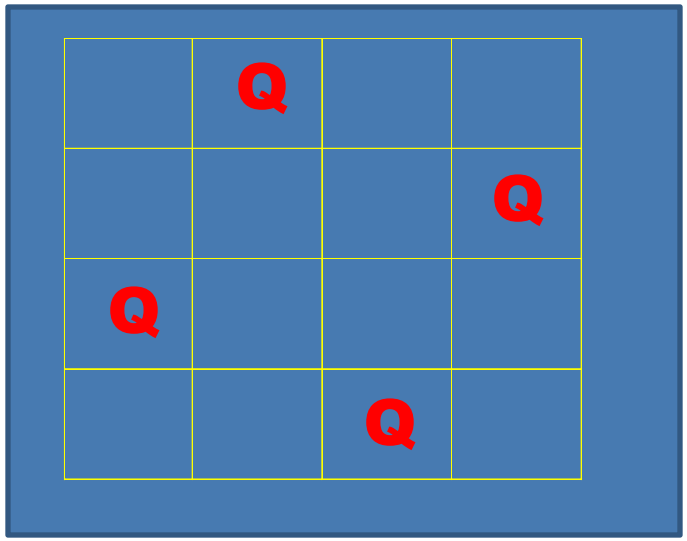
((1,1) (2,3))

((1,1) (2,4))

((1,2) (2,4))

((1,1) (2,4) (3.2))

((1,2) (2,4) (3,1))



( )

((1,1))

((1,2))

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

((1,1) (2,4))

((1,2) (2,4))

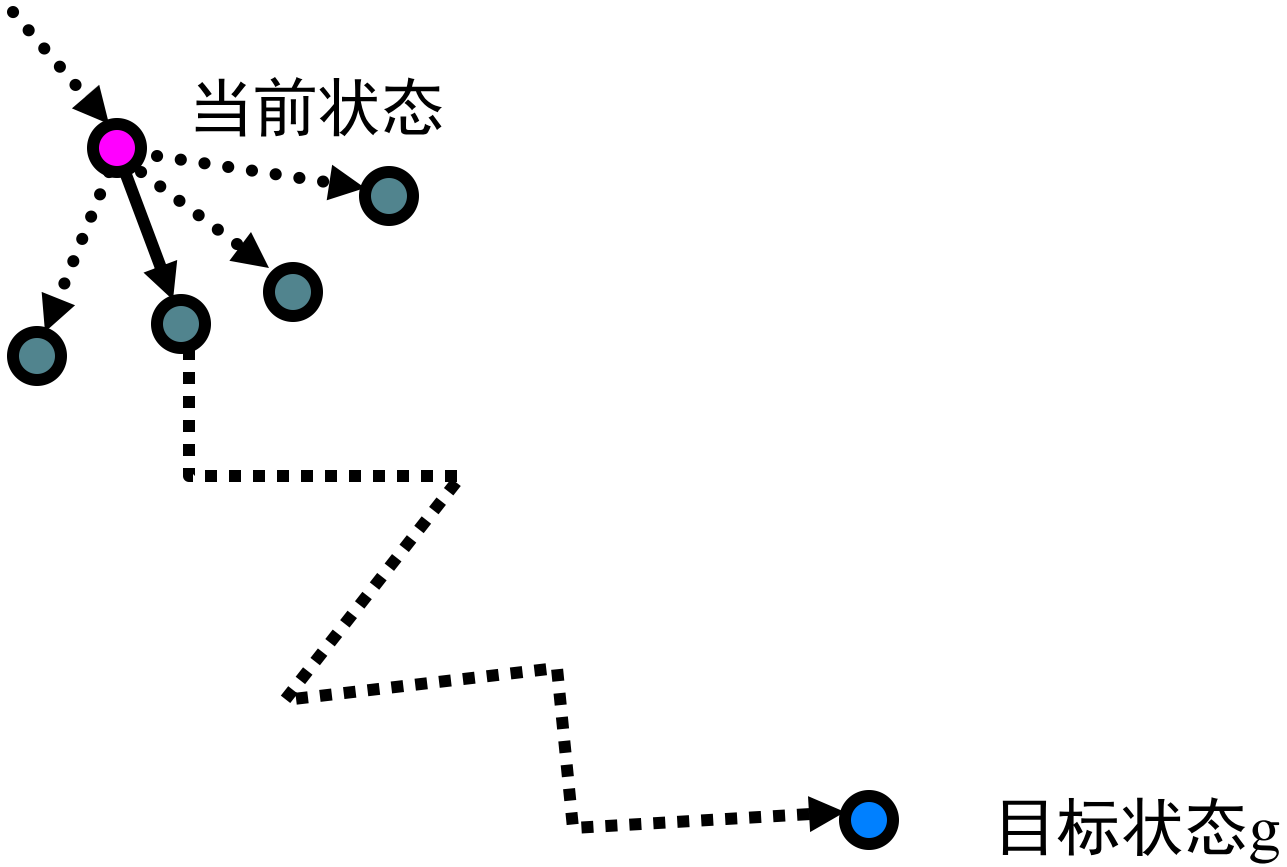
((1,1) (2,4) (3.2))

((1,2) (2,4) (3,1))

((1,2) (2,4) (3,1) (4,3))

SUCCESS

递归的思想



回溯搜索

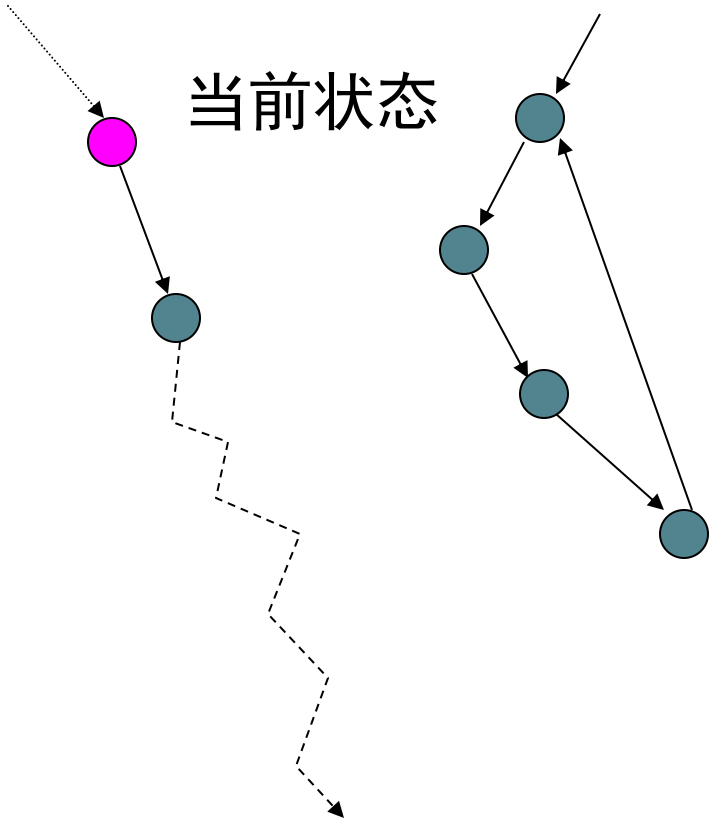
- 回溯搜索是深度优先搜索的另一种搜索方式，与前面一般的DFS搜索略有不同
  - 对节点扩展时，每次只产生一个后继结点，并继续对这个节点进行回溯搜索
  - 当一条路径搜索完成后，再返回到生成当前节点的那个节点，看是否可以从该节点再生成新的节点
  - 重复上过程，直到找到搜索目标或者搜索完全部状态空间

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



递归 VS. 图搜索

- 递归搜索：只保留从初始状态到当前状态的一条路径。
- 图搜索：保留所有已经搜索过的路径。