第 5 章　搜索求解策略

教材(仅供参考):

王万良《人工智能导论》（第4版）

https://www.icourse163.org/course/ZJUT-1002694018

社区资源: https://github.com/Microsoft/ai-edu

第5章 搜索求解策略

- 描述性的知识创造了多种可能的选择
  - 那么应该使用哪条知识呢？
  - 如何使用呢？
- 搜索就是对可能的选择的探索，是探索知识的一种主要方法

第5章 搜索求解策略

- 在求解一个问题时，涉及到两个方面：一是该问题的表示，如果一个问题找不到一个合适的表示方法，就谈不上对它求解。另一方面则是选择一种相对合适的求解方法。由于绝大多数需要人工智能方法求解的问题缺乏直接求解的方法，因此，搜索为一种求解问题的一般方法。
- 下面首先讨论搜索的基本概念，然后着重介绍状态空间知识表示和搜索策略，主要有回溯策略、宽度优先搜索、深度优先搜索等盲目的图搜索策略，以及A及A*搜索算法等启发式图搜索策略。

第5章 搜索求解策略

- 5.1 搜索的概念
- 5.2 状态空间的搜索策略
- 5.3 盲目的图搜索策略
- 5.4 启发式图搜索策略

第5章 搜索求解策略

- ✓ **5.1 搜索的概念**
- **5.2 状态空间的搜索策略**
- **5.3 盲目的图搜索策略**
- **5.4 启发式图搜索策略**

5.1 搜索的概念

- 问题求解：
- 问题的表示。
- 求解方法。
- 问题求解的基本方法：搜索法、归约法、归结法、推理法及产生式等。

5.1 搜索的概念

- 问题求解：
  - 解决这些问题有一些通用的搜索算法
    - 无信息的（除了问题本身，没有任何其他信息）
    - 有信息的（利用给定的知识引导，能够更有效的找到解）
  - 为达到目标，寻找这样的行动序列的过程称为搜索
  - 一个问题可以用5个组成部分形式描述
    - 1.Agent的初始状态
    - 2.描述Agent的可能的行动
    - 3.对每个行动的描述
    - 4.目标测试
    - 5.路径耗散

5.1 搜索的概念

- 1 吸尘器问题
  - 状态　：由Agent位置和灰尘位置确定，有n个位置，而且是有或没有灰尘，世界状态有$n*2^n$
  - 初始行动：任何状态都能被设计成初始状态
  - 行动　：左、右、前、后、吸
  - 转移模型：最左边不能再左，最右边不能再右
  - 目标测试：检测所有位置是否干净
  - 路径耗散：路径代价值为1

5.1 搜索的概念

- 2 Web站点和车载系统导航
  - 状态　：地点(如机场)和当前时间
  - 初始状态：用户在咨询时确定
  - 行动　：乘坐一航班任意舱位从现有地点起飞
  - 转移模型：飞行目的地当做当前地点和飞行抵达时间作为当前时间
  - 目标测试：是否到达了用户描述的目的地
  - 路径耗散：金钱、时间、舱位等级等等

5.1.1 搜索的基本问题与主要过程

- **搜索中需要解决的基本问题：**

（1）是否一定能找到一个解。

（2）找到的解是否是最佳解。

（3）时间与空间复杂性如何。

（4）是否终止运行或是否会陷入一个死循环。

5.1.1 搜索的基本问题与主要过程

- **搜索的主要过程：**

(1) 从初始或目的状态出发，并将它作为当前状态。

(2) 扫描操作算子集，将适用当前状态的一些操作算子作用于当前状态而得到新的状态，并建立指向其父结点的指针 。

(3) 检查所生成的新状态是否满足结束状态，如果满足，则得到问题的一个解，并可沿着有关指针从结束状态反向到达开始状态，给出一解答路径；否则，将新状态作为当前状态，返回第(2)步再进行搜索。

5.1.2 搜索策略

1. 搜索方向：

(1) **数据驱动**：从初始状态出发的正向搜索。

正向搜索——从问题给出的条件出发。

(2) **目的驱动**：从目的状态出发的逆向搜索。

逆向搜索：从想达到的目的入手，看哪些操作算子能产生该目的以及应用这些操作算子产生目的时需要哪些条件。

(3) 双向搜索

双向搜索：从开始状态出发作正向搜索，同时又从目的状态出发作逆向搜索，直到两条路径在中间的某处汇合为止。

5.1.2 搜索策略

- **2. 盲目搜索与启发式搜索：**

（1）**盲目搜索**：在不具有对**特定问题**的任何有关信息的条件下，按固定的步骤（依次或随机调用操作算子）进行的搜索。

（2）**启发式搜索**：考虑特定问题领域可应用的知识，动态地确定调用操作算子的步骤，优先选择较适合的操作算子，尽量减少不必要的搜索，以求尽快地到达结束状态。

第5章 搜索求解策略

- 5.1 搜索的概念
- ✓ 5.2 状态空间的搜索策略
- 5.3 盲目的图搜索策略
- 5.4 启发式图搜索策略

5.2 状态空间的搜索策略

- **5.2.1 状态空间表示法**
- **5.2.2 状态空间的图描述**

5.2.1 状态空间表示法

- **状态空间图**：对一个问题的表示
  - 通过问题表示，人们可以探索和分析通往解的可能的可替代路径
  - 特定问题的解将对应状态空间图中的一条路径
    - 搜索一个问题的任意解
    - 一个最短（最优）的解

5.2.1 状态空间表示法

- **状态**：表示系统状态、事实等叙述型知识的一组变量或数组：

$$Q = [q_1, q_2, \cdots, q_n]^T$$

- **操作**：表示引起状态变化的过程型知识的一组关系或函数：

$$F = \{f_1, f_2, \cdots, f_m\}$$

5.2.1 状态空间表示法

- **状态空间**：利用状态变量和操作符号，表示系统或问题的有关知识的符号体系，状态空间是一个四元组：

$$(S, O, S_0, G)$$

S：状态集合。

O：操作算子的集合。

S_0：包含问题的初始状态是 S 的非空子集。

G：包含问题的目的状态，可以是若干具体状态或满足某些性质的路径信息描述。

5.2.1 状态空间表示法

- **求解路径：**从 S_0 结点到 G 结点的路径。

- **状态空间的一个解：**一个有限的操作算子序列。

$$S_0 \xrightarrow{O_1} S_1 \xrightarrow{O_2} S_2 \xrightarrow{O_3} \cdots \xrightarrow{O_k} G$$

$O_1, \cdots, O_k$ ：状态空间的一个解。

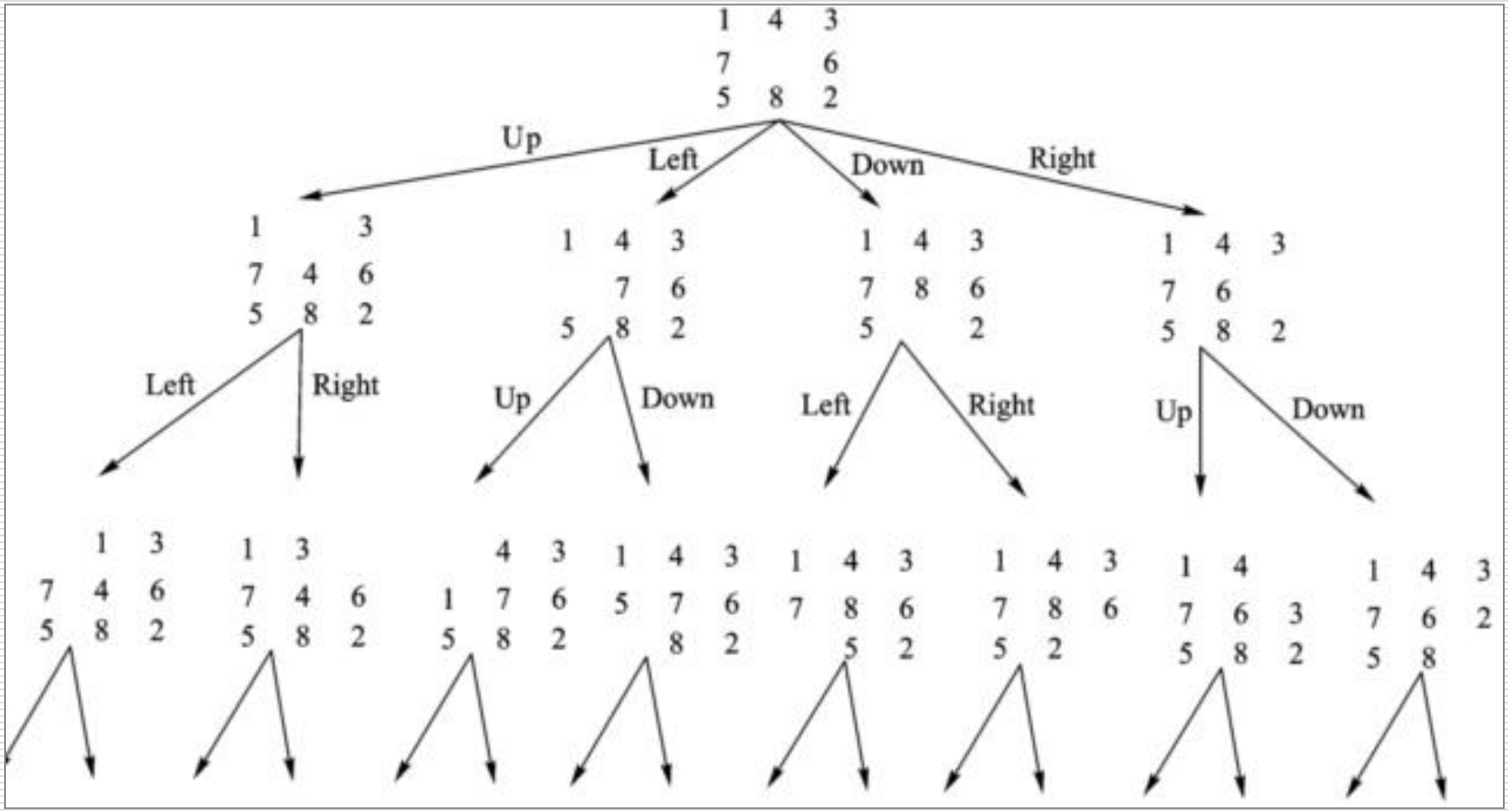
5.2.1 状态空间表示法

- **例5.1　八数码问题的状态空间。**

状态集 S ：所有摆法　　操作算子：

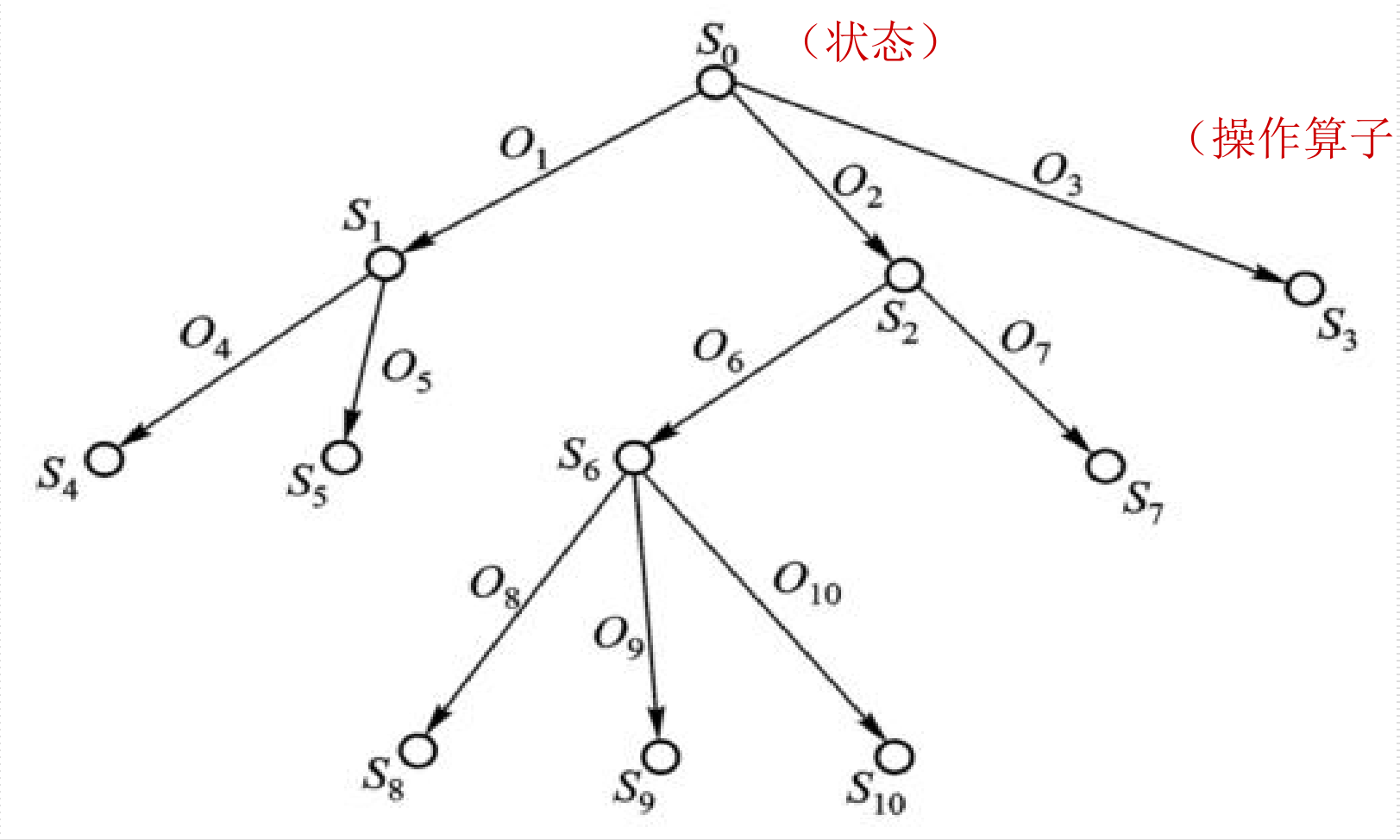
将空格向上移Up
将空格向左移Left
将空格向下移Down
将空格向右移Right

5.2.2 状态空间的图描述



八数码状态空间图

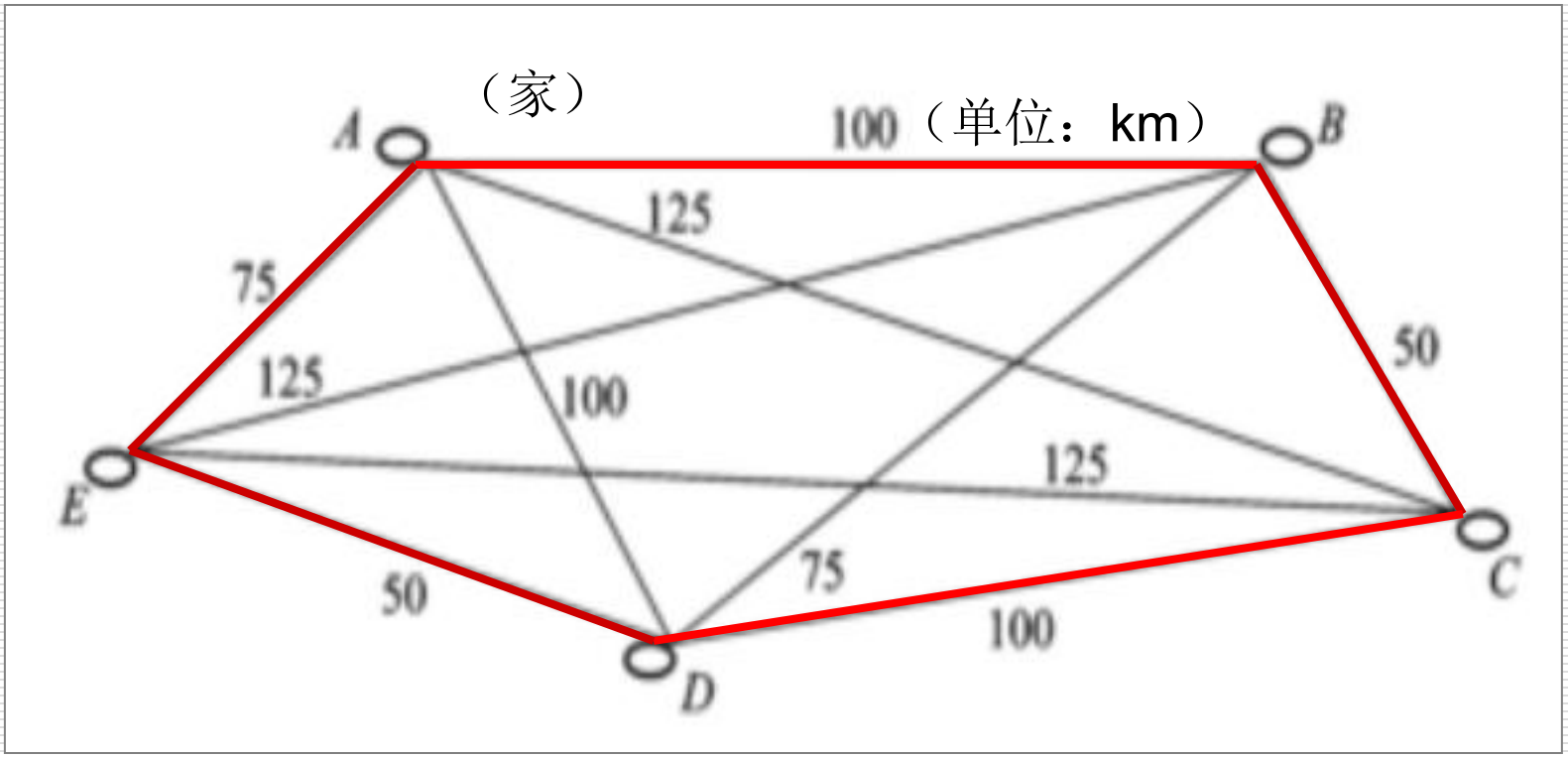
5.2.2 状态空间的图描述



状态空间的有向图描述

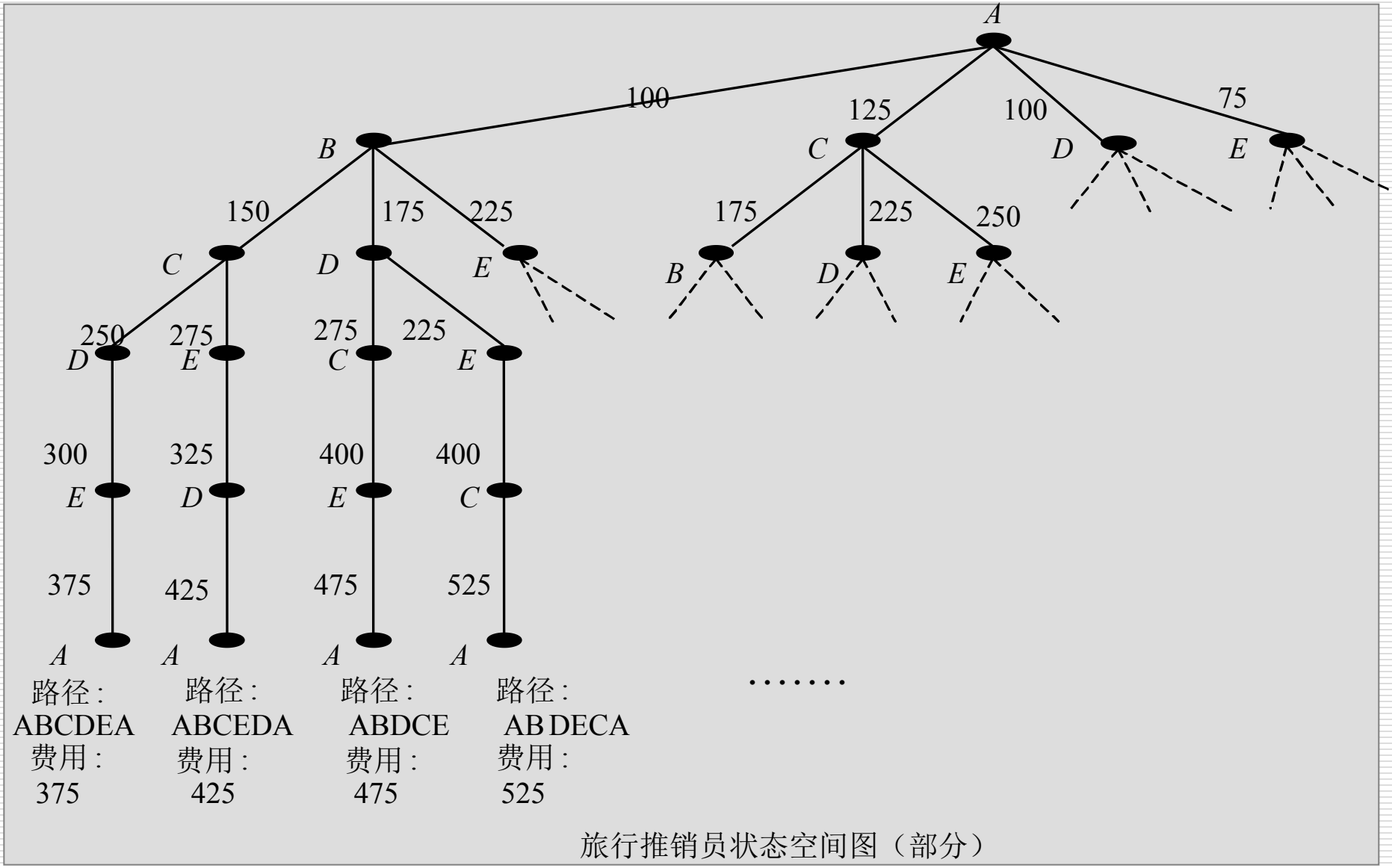
5.2.2 状态空间的图描述

例5.3 旅行商问题（traveling salesman problem, TSP）或邮递员路径问题。



可能路径：费用为375的路径（*A*，*B*，*C*，*D*，*E*，*A*）

5.2.2 状态空间的图描述



旅行推销员状态空间图（部分）

第5章 搜索求解策略

- 5.1 搜索的概念
- 5.2 状态空间知识表示方法
- ✓ 5.3 盲目的图搜索策略
- 5.4 启发式图搜索策略

5.3 盲目的图搜索策略

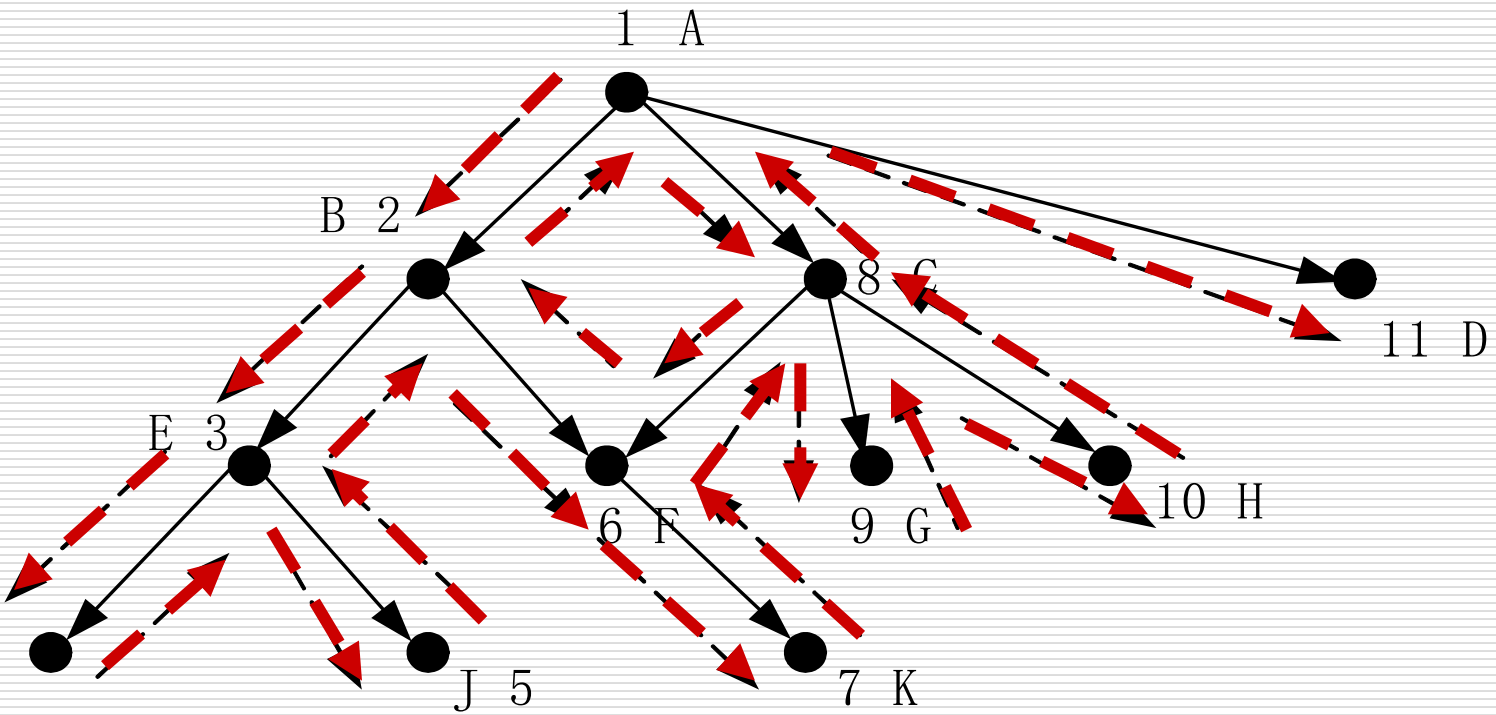
- **5.3.1 回溯策略**
- **5.3.2 宽度优先搜索策略**
- **5.3.3 深度优先搜索策略**

5.3.1 回溯策略

- **带回溯策略的搜索：**

从初始状态出发，不停地、试探性地寻找路径，直到它到达目的或“不可解结点”，即“死胡同”为止。若它遇到不可解结点就回溯到路径中最近的父结点上，查看该结点是否还有其他的子结点未被扩展。若有，则沿这些子结点继续搜索；如果找到目标，就成功退出搜索，返回解题路径。

5.3.1 回溯策略



回溯搜索示意图

5.3.1 回溯策略

- **回溯搜索的算法**

(1) PS（path states）表：保存当前搜索路径上的状态。如果找到了目的，PS就是解路径上的状态有序集。

(2) NPS（new path states）表：新的路径状态表。它包含了等待搜索的状态，其后裔状态还未被搜索到，即未被生成扩展 。

(3) NSS（no solvable states）表：不可解状态集，列出了找不到解题路径的状态。如果在搜索中扩展出的状态是它的元素，则可立即将之排除，不必沿该状态继续搜索。

5.3.1 回溯策略

- 图搜索算法（深度优先、宽度优先、最好优先搜索等）的回溯思想（P112）：

（1）用未处理状态表（NPS）使算法能返回（回溯）到其中任一状态。

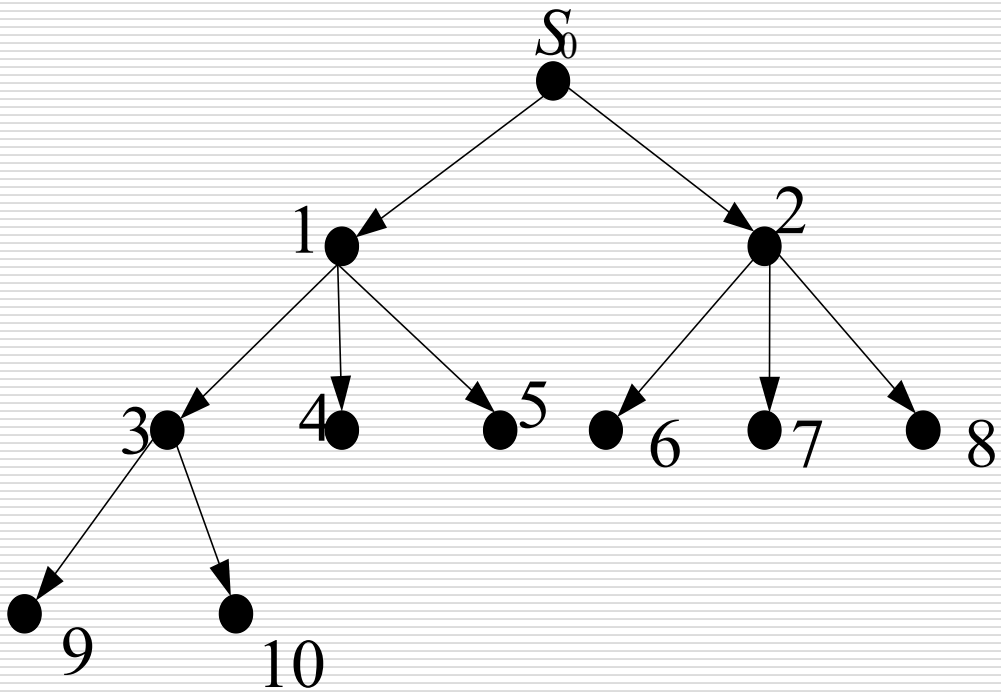
（2）用一张“死胡同”状态表（NSS）来避免算法重新搜索无解的路径。

（3）在PS 表中记录当前搜索路径的状态，当满足目的时可以将它作为结果返回。

（4）为避免陷入死循环必须对新生成的子状态进行检查，看它是否在该三张表中 。

5.3.2 宽度优先搜索策略

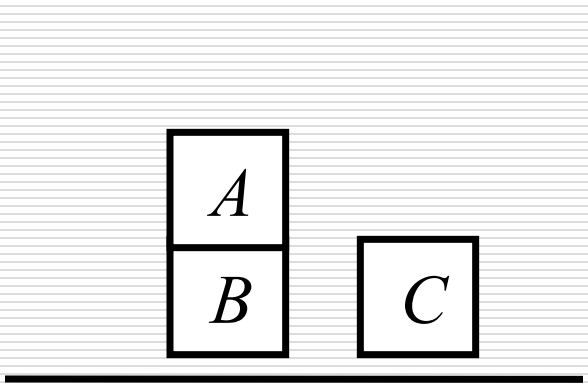
- open表（NPS表）：已经生成出来但其子状态未被搜索的状态。
- closed表（ PS表和NSS表的合并）：记录了已被生成扩展过的状态。



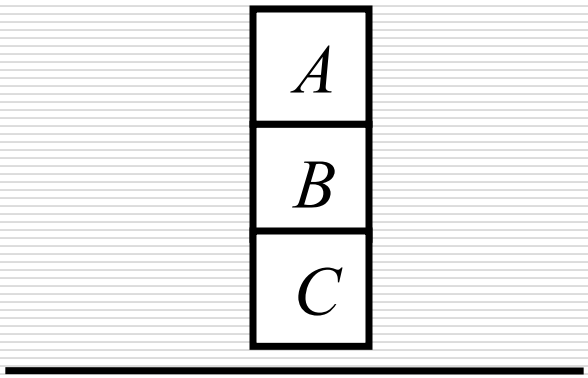
宽度优先搜索法中状态的搜索次序

5.3.2 宽度优先搜索策略

- **例5.4** 通过搬动积木块，希望从初始状态达到一个目的状态，即三块积木堆叠在一起。



(a) 初始状态



(b) 目的状态

积木问题

5.3.2 宽度优先搜索策略

- 操作算子为*MOVE*（*X*，*Y*）：把积木*X*搬到*Y*（积木或桌面）上面。

  MOVE（*A*，*Table*）：“搬动积木*A*到桌面上”。

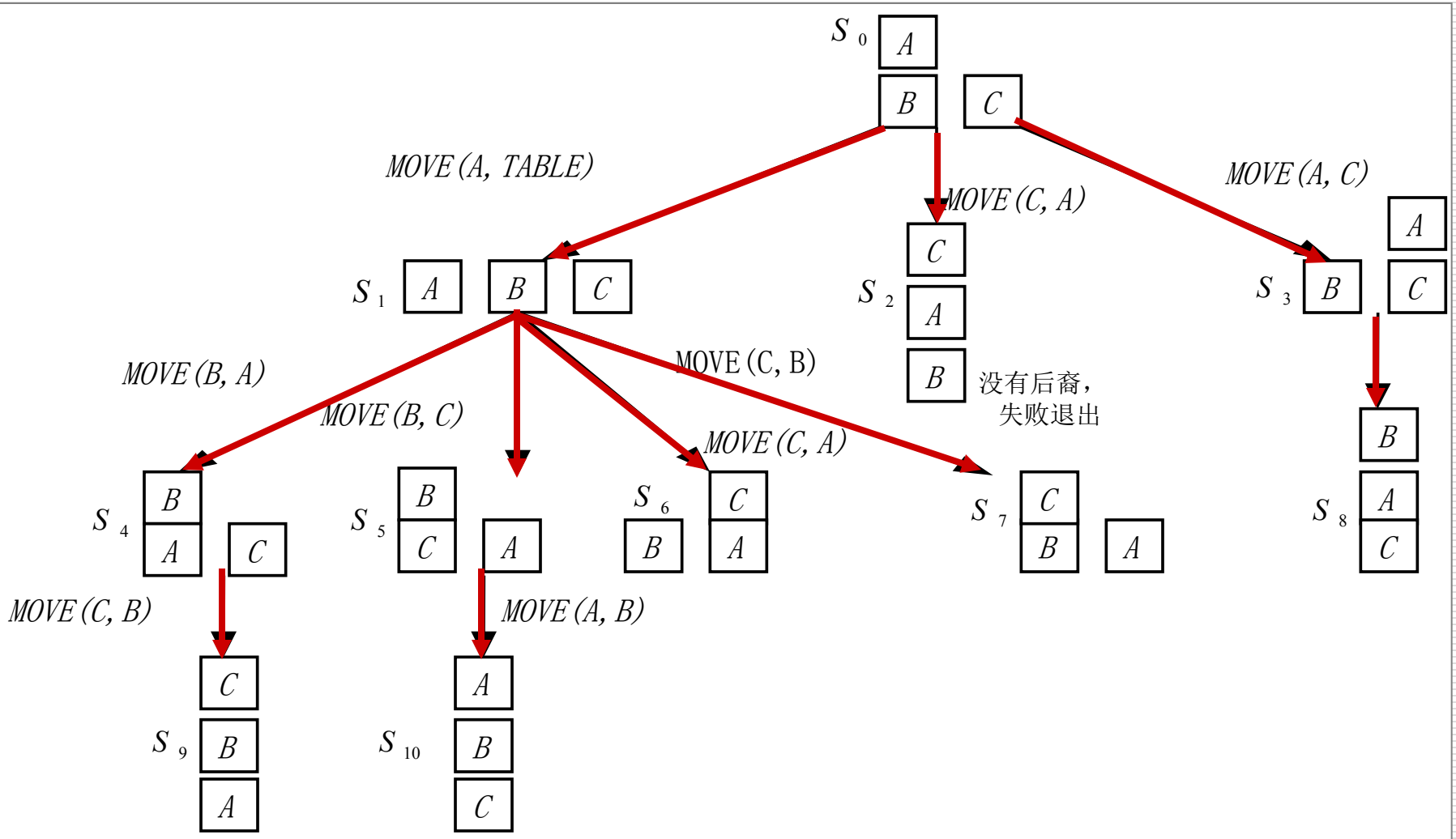
- 操作算子可运用的先决条件：

（1）被搬动积木的顶部必须为空。

（2）如果 *Y* 是积木，则积木 *Y* 的顶部也必须为空。

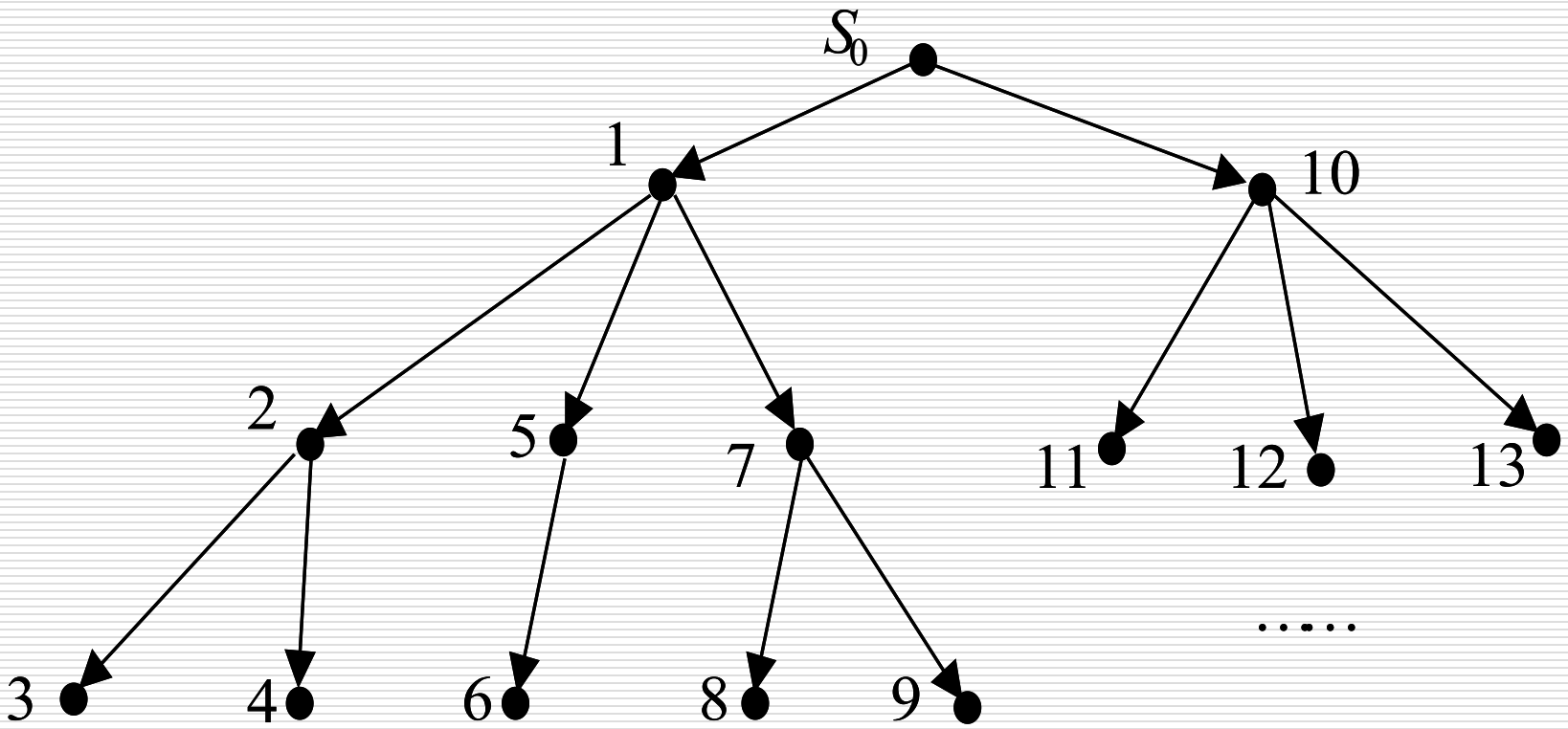
（3）同一状态下，运用操作算子的次数不得多于一次。

5.3.2 宽度优先搜索策略



积木问题的宽度优先搜索树

5.3.3 深度优先搜索策略



深度优先搜索法中状态的搜索次序

5.3.3 深度优先搜索策略

- 在深度优先搜索中，当搜索到某一个状态时，它所有的子状态以及子状态的后裔状态都必须先于该状态的兄弟状态被搜索。
- 为了保证找到解，应选择合适的深度限制值，或采取不断加大深度限制值的办法，反复搜索，直到找到解。

5.3.3 深度优先搜索策略

- 深度优先搜索并不能保证第一次搜索到的某个状态时的路径是到这个状态的最短路径。
- 对任何状态而言，以后的搜索有可能找到另一条通向它的路径。如果路径的长度对解题很关键的话，当算法多次搜索到同一个状态时，它应该保留最短路径。

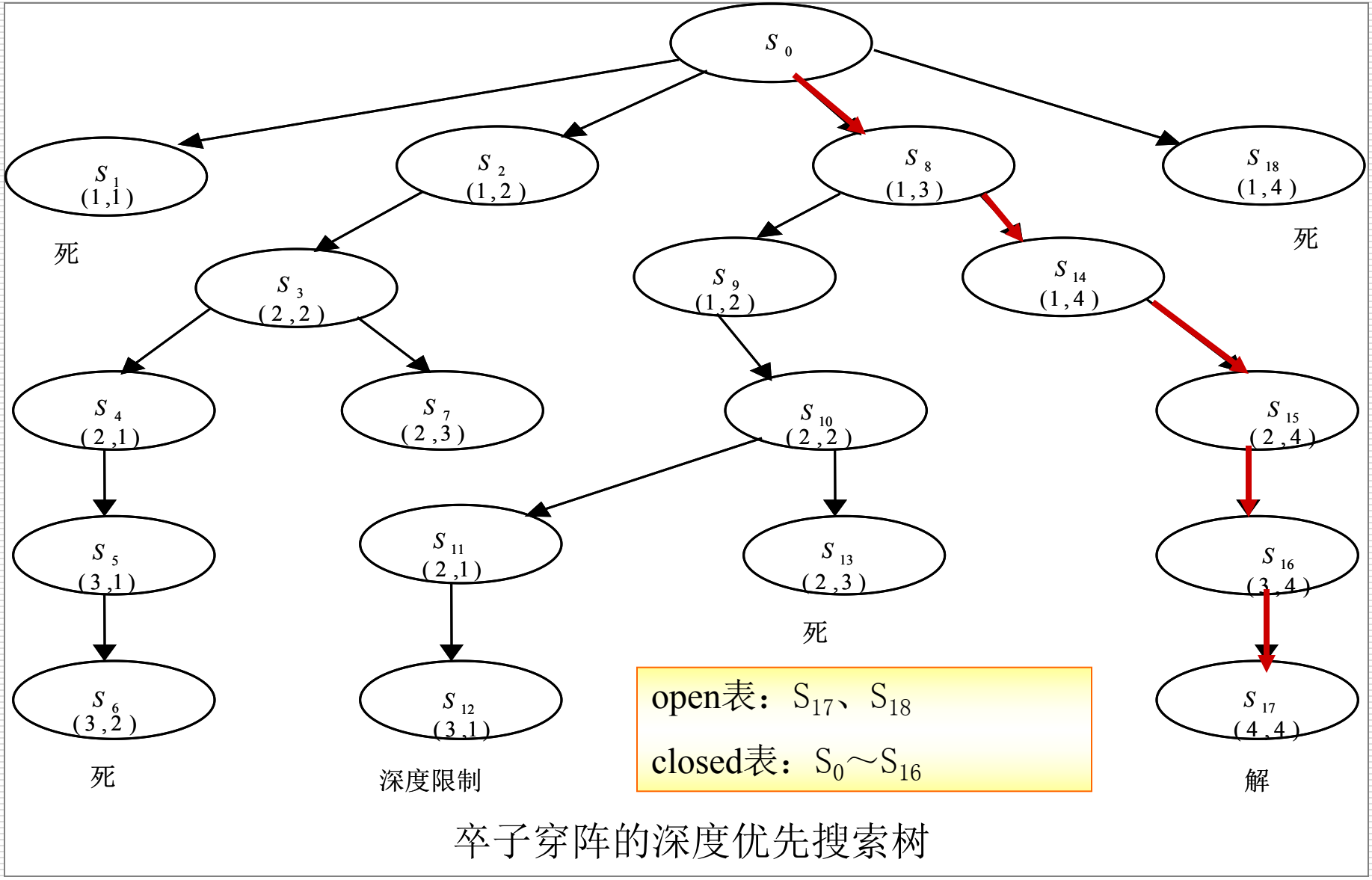
5.3.3 深度优先搜索策略

- **例5.5　卒子穿阵问题**，要求一卒子从顶部通过下图所示的阵列到达底部。卒子行进中不可进入到代表敌兵驻守的区域（标注1），并不准后退。假定深度限制值为5。

行	1	2	3	4	列
1	1	0	0	0	
2	0	0	1	0	
3	0	1	0	0	
4	1	0	0	0	

阵列图

5.3.3 深度优先搜索策略



卒子穿阵的深度优先搜索树

第5章 搜索求解策略

- 5.1 搜索的概念
- 5.2 状态空间知识表示方法
- 5.3 盲目的图搜索策略
- ✓ 5.4 启发式图搜索策略
- 5.5 与/或图搜索策略

5.4 启发式图搜索策略

- **5.4.1 启发式策略**
- **5.4.2 启发信息和估价函数**
- **5.4.3 *A*搜索算法**
- **5.4.4 *A**搜索算法及其特性分析**

- **MMRPG 游戏**
  - 魔兽世界、仙剑奇侠传
  - 有一个非常重要的功能，那就是人物角色自动寻路。
    - 当人物处于游戏地图中的某个位置的时候，我们用鼠标点击另外一个相对较远的位置，人物就会自动地绕过障碍物走过去。玩过这么多游戏，不知你是否思考过，这个功能是怎么实现的呢？

- 实际上，像出行路线规划、游戏寻路，这些真实软件开发中的问题，一般情况下，都不需要非得求最优解（也就是最短路径）。
- 在权衡路线规划质量和执行效率的情况下，只需要寻求一个次优解就足够了。
- 那如何快速找出一条接近于最短路线的次优路线呢？

5.4.1 启发式策略

- “**启发**”（heuristic）：关于发现和发明操作算子及搜索方法的研究。
- 在状态空间搜索中，**启发式**被定义成一系列操作算子，并能从状态空间中选择最有希望到达问题解的路径。
- **启发式策略**：利用与问题有关的启发信息进行搜索。
  - 在搜索过程中，启发式算法被定义成一系列额外的规则
  - 经验法则
  - 利用一些特定的知识
  - “高手怎么下，我也怎么下”

5.4.1 启发式策略

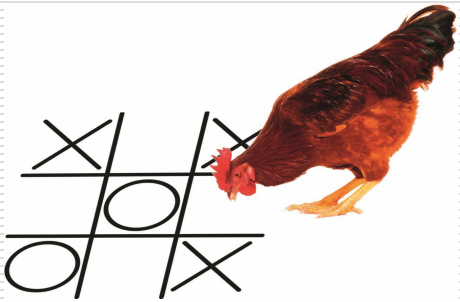
- **运用启发式策略的两种基本情况：**

（1）一个问题由于在问题陈述和数据获取方面固有的模糊性，可能会使它没有一个确定的解。

（2）虽然一个问题可能有确定解，但是其状态空间特别大，搜索中生成扩展的状态数会随着搜索的深度呈指数级增长。

5.4.1 启发式策略

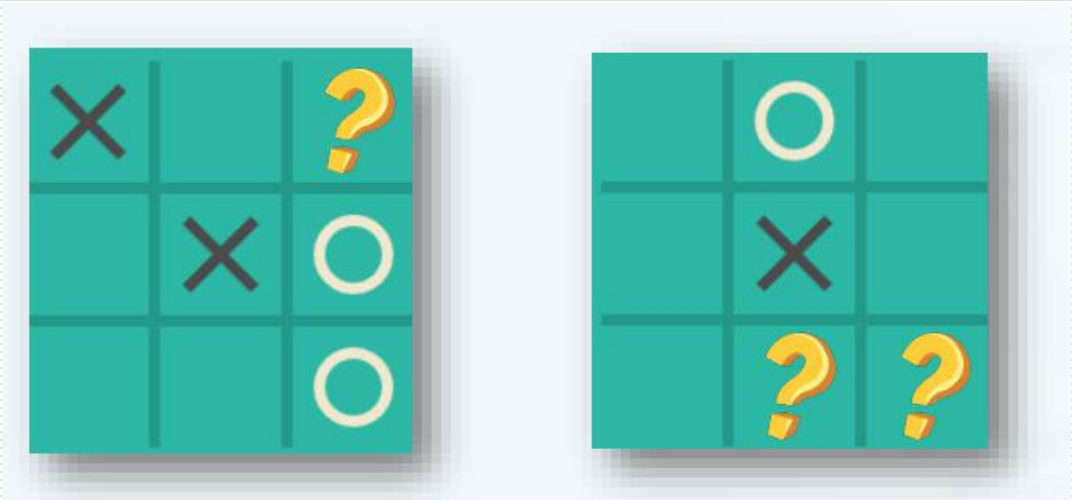
- **例5.6 井字棋**。在九宫棋盘上，从空棋盘开始，双方轮流在棋盘上摆各自的棋子 × 或 ○ （每次一枚），谁先取得三子一线（一行、一列或一条对角线）的结果就取胜



- × 和 ○ 能够在棋盘中摆成的各种不同的棋局就是问题空间中的不同状态。
- 9个位置上摆放{空， × ， ○}有 3^9 种棋局。
- 可能的走法： $9\times8\times7\times\cdots\times1$。

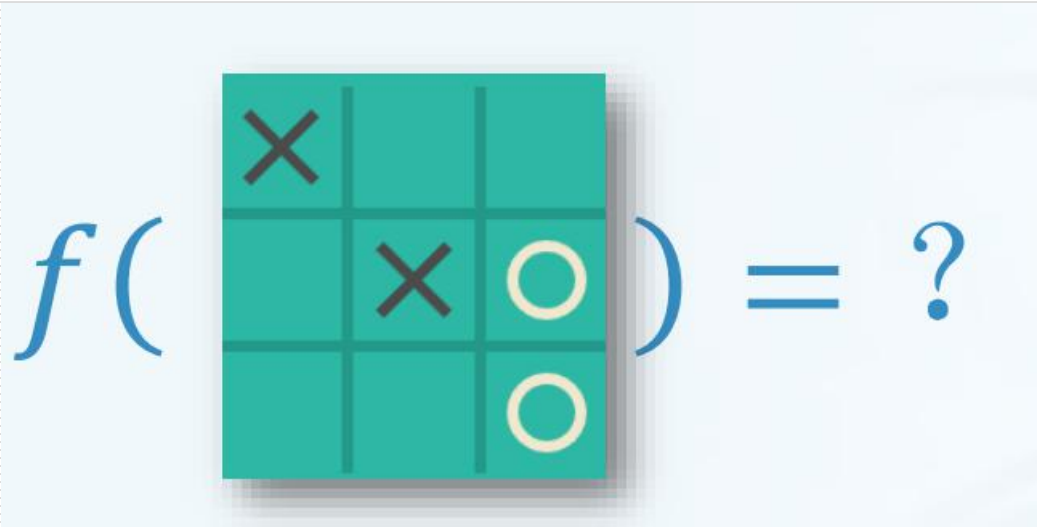
5.4.1 启发式策略

- 如何判断是否有利
  - 当前局面如何
  - 接下来走哪一步棋是最有利的



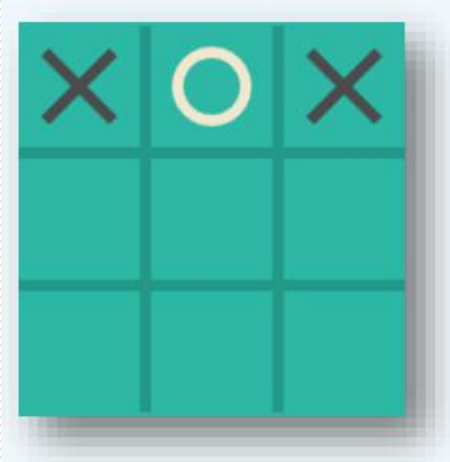
5.4.1 启发式策略

- 估值函数
  - 对每一种局面给出一个估值

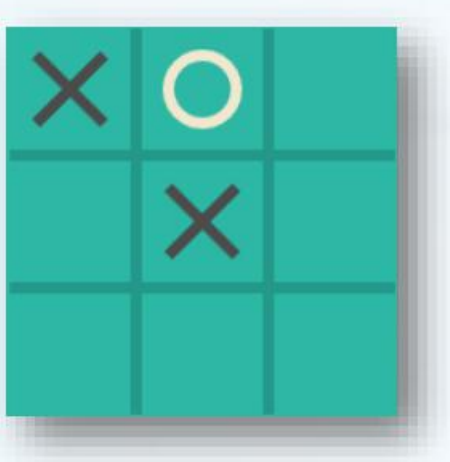


5.4.1 启发式策略

- 井字棋的估值函数
  - 玩家X还存在可能性的行、列、斜线数减去
  - 玩家O还存在可能性的行、列、斜线数



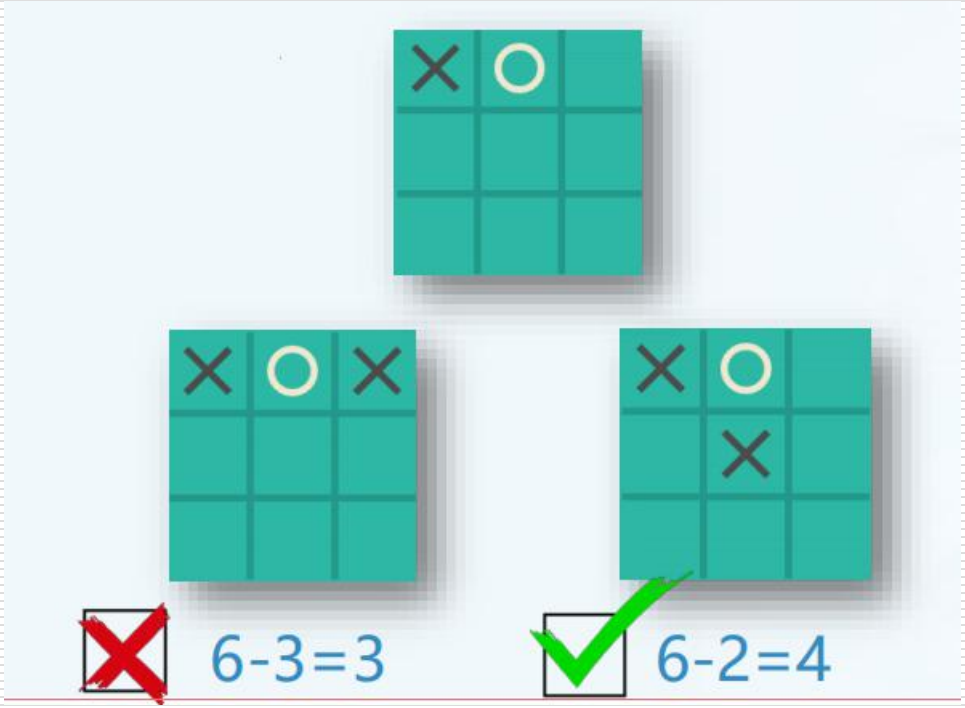
6-3=3



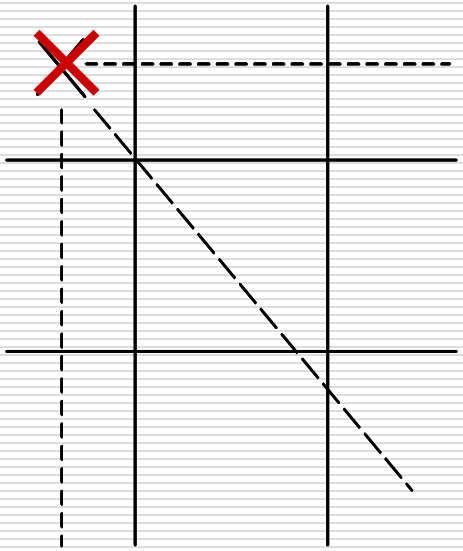
6-2=4

5.4.1 启发式策略

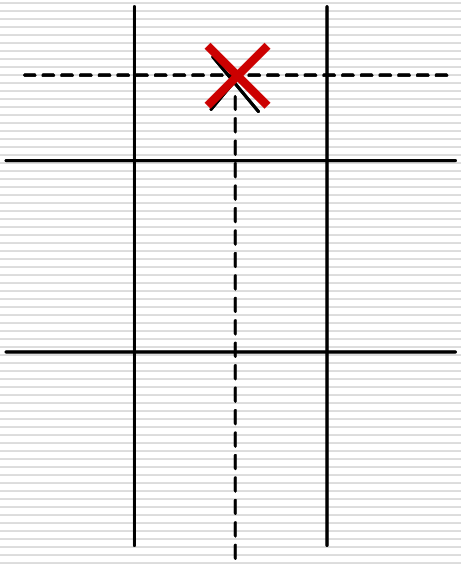
- 选取策略
  - 根据估值函数选择最优策略
  - 最佳行动就是能够使得下一个状态的评估值最大的行动



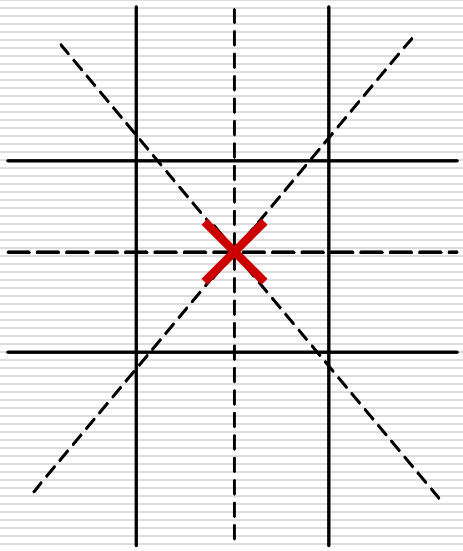
5.4.1 启发式策略



赢的几率③



赢的几率②



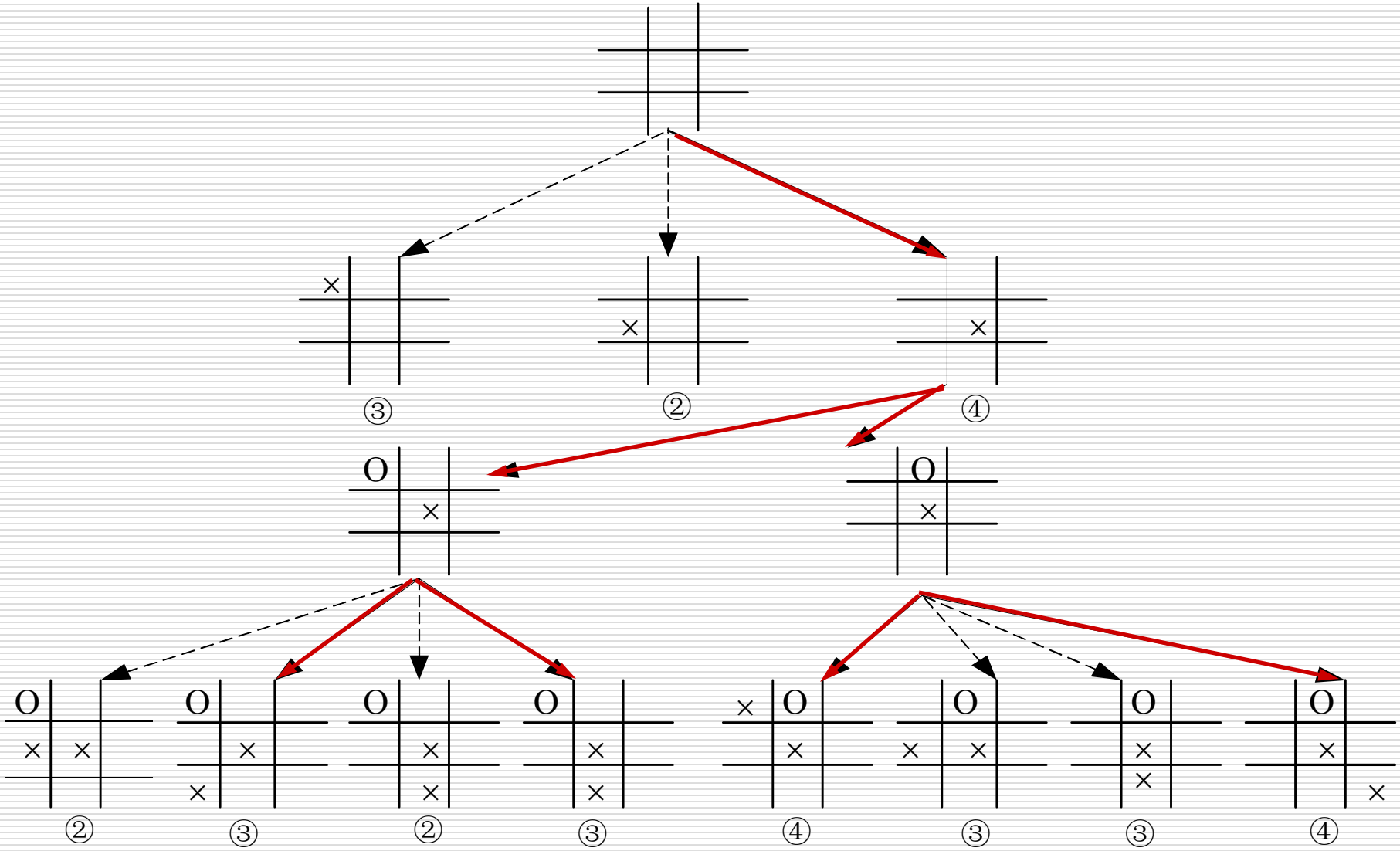
赢的几率④

启发式策略的运用

5.4.1 启发式策略

- 选取策略
  - 根据估值函数选择最优策略
  - 最佳行动就是能够使得下一个状态的评估值最大的行动
  - 相对重要的是，更有利的棋局（较好的棋局）被赋予了更高的启发值

5.4.1 启发式策略



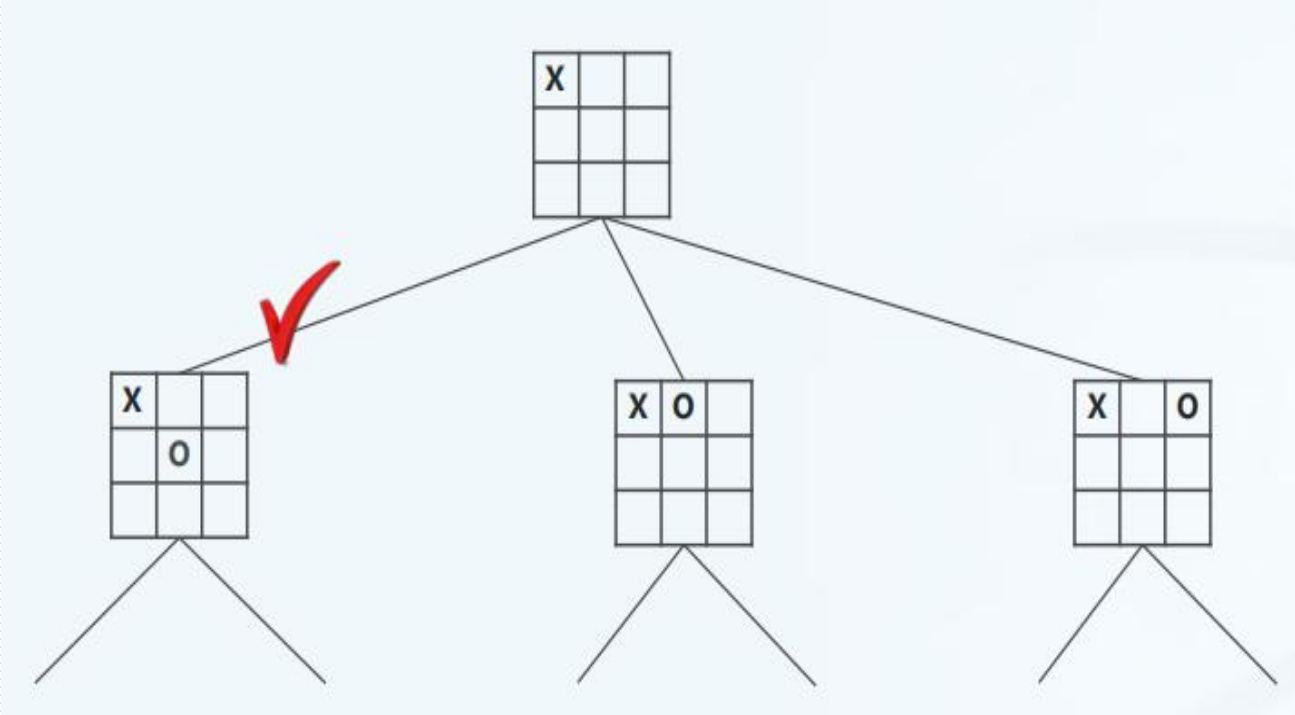
启发式搜索下缩减的状态空间

5.4.1 启发式策略

- 特点
  - 它常能发现很不错的解，但也没办法证明它不会得到较坏的解
  - 它通常可在合理时间解出答案，但也没办法知道它是否每次都可以这样的速度求解

5.4.1 启发式策略

- 井字棋
  - 当X下在角落的时候，O必须下在中心



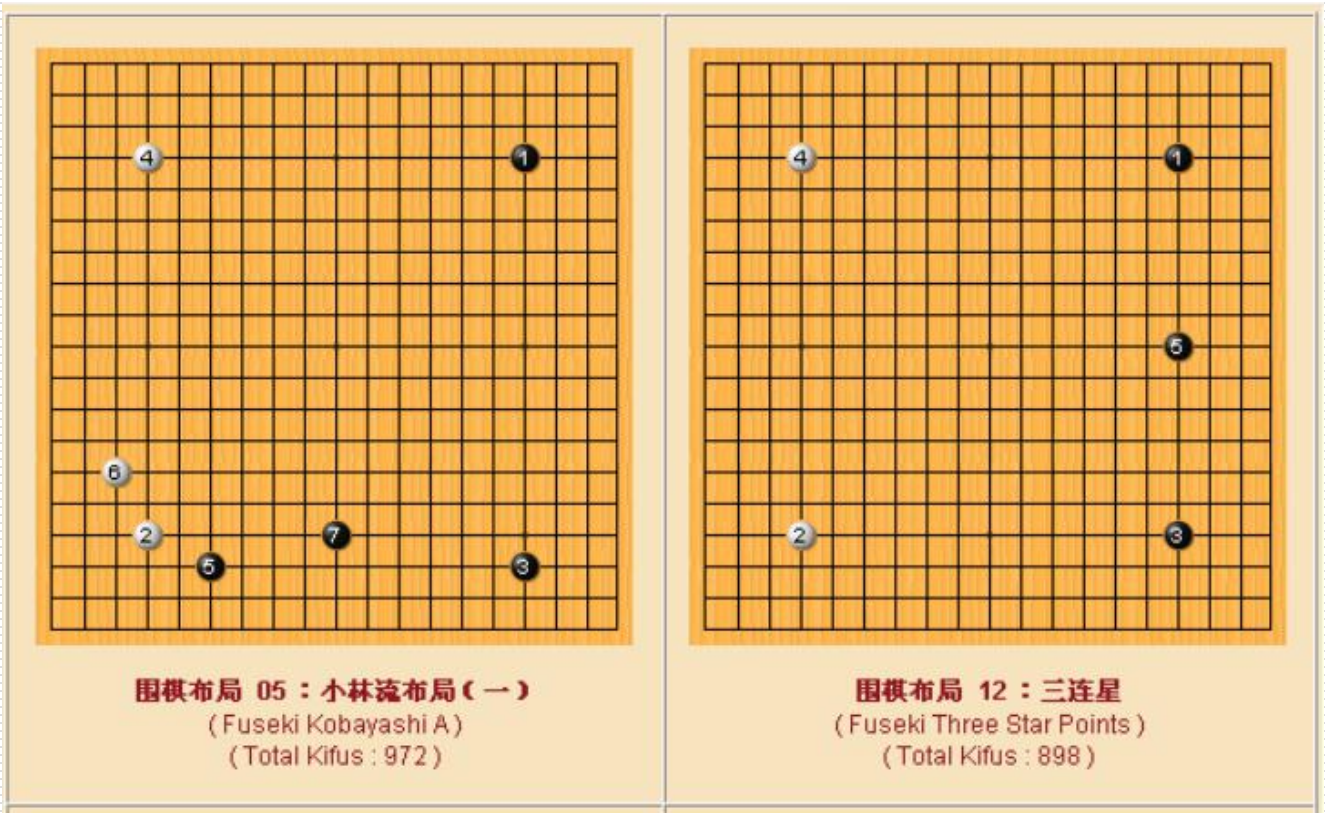
  - 井字棋的最优策略竟是先占角

5.4.1 启发式策略

- 进一步改进
  - 采用多步搜索策略
  - 提高搜索的深度
  - 尽量接近搜索过程的终止状态
  - 搜索配合剪枝提高效率

5.4.1 启发式策略

- 围棋
  - 根据经验积累出很多布局定式



5.4.2 启发信息和估价函数

- 在具体求解中，能够利用与该问题有关的信息来简化搜索过程，称此类信息为**启发信息**。
- **启发式搜索**：利用启发信息的搜索过程。

5.4.2 启发信息和估价函数

- 求解问题中能利用的大多是非完备的启发信息：

（1）求解问题系统不可能知道与实际问题有关的全部信息，因而无法知道该问题的全部状态空间，也不可能用一套算法来求解所有的问题。

（2）有些问题在理论上虽然存在着求解算法，但是在工程实践中，这些算法不是效率太低，就是根本无法实现。

＃字棋：9！，西洋跳棋：10^{78}，国际象棋：10^{120}，围棋：10^{761}。

假设每步可以搜索一个棋局，用极限并行速度（10^{-104}年/步）来处理，搜索一遍国际象棋的全部棋局也得10^{16}年即1亿亿年才可以算完！

5.4.2 启发信息和估价函数

- 启发信息的分类：

（1）陈述性启发信息

（2）过程性启发信息

（3）控制性启发信息

- 利用控制性的启发信息的情况：

（1）没有任何控制性知识作为搜索的依据，因而搜索的每一步完全是随意的。

（2）有充分的控制知识作为依据，因而搜索的每一步选择都是正确的，但这是不现实的。

5.4.2 启发信息和估价函数

- 估价函数的任务就是估计待搜索结点的“有希望”程度，并依次给它们排定次序（在open表中）。

- 估价函数 $f(n)$ ：从初始结点经过 n 结点到达目的结点的路径的最小代价估计值，其一般形式是

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

- 一般地，在$f(n)$ 中，$g(n)$的比重越大，越倾向于宽度优先搜索方式，而 $h(n)$的比重越大，表示启发性能越强。

5.4.2 启发信息和估价函数

- **例5.7 八数码的估价函数**设计方法有多种，并且不同的估价函数对求解八数码问题有不同的影响。

  - 最简单的估价函数：取一格局与目的格局相比，其位置不符的将牌数目。
  - 较好的估价函数：各将牌移到目的位置所需移动的距离的总和。
  - 第三种估价函数：对每一对逆转将牌乘以一个倍数。
  - 第四种估价函数：克服了仅计算将牌逆转数目策略的局限，将位置不符将牌数目的总和与3倍将牌逆转数目相加。

5.4.3 *A*搜索算法

- **启发式图搜索法的基本特点：**如何寻找并设计一个与问题有关的$h(n)$及构出$f(n)=g(n)+h(n)$，　然后以$f(n)$的大小来排列待扩展状态的次序，每次选择$f(n)$值最小者进行扩展。

- open表：保留所有已生成而未扩展的状态。
- closed表：记录已扩展过的状态。
- 进入open表的状态是根据其估值的大小插入到表中合适的位置，每次从表中优先取出启发估价函数值最小的状态加以扩展。

5.4.3 *A*搜索算法

- 一般启发式图搜索算法（简记为*A*）

```
procedure heuristic_search
open：=[start]；closed：=[ ]；f(s)：=g(s)+h(s)；      *初始化
while open≠[ ] do
begin
   从open表中删除第一个状态，称之为n；
   if n=目的状态then return(success)；
   生成n的所有子状态；
  if n没有任何子状态then continue；
 for n的每个子状态do
    case子状态is not already on open表or closed表；
       begin
           计算该子状态的估价函数值；
           将该子状态加到open表中；
        end；
```

5.4.3 *A*搜索算法

case子状态is already on open表：
if该子状态是沿着一条比在open表已有的更短路径而到达
then 记录更短路径走向及其估价函数值；
case子状态is already on closed表：
if该子状态是沿着一条比在closed表已有的更短路径而到达then
begin
将该子状态从closed表移到open表中；
记录更短路径走向及其估价函数值；
end；
case end；
将n放入closed表中；
根据估价函数值，从小到大重新排列open表；
end；*open表中结点已耗尽
return(failure)；
end.

5.4.3 *A*搜索算法

- 每次重复时，*A*搜索算法从open表中取出第一个状态，如果该状态满足目的条件，则算法返回到该状态的搜索路径。
- 如果open表的第一个状态不是目的状态，则算法利用与之相匹配的一系列操作算子进行相应的操作来产生它的子状态。如果某个子状态已在open表（或closed表）中出现过，即该状态再一次被发现时，则通过刷新它的祖先状态的历史记录，使算法极有可能找到到达目的状态的更短的路径
- 接着，*A*搜索算法open表中每个状态的估价函数值，按照值的大小重新排序，将值最小的状态放在表头，使其第一个被扩展。

5.4.3 A搜索算法

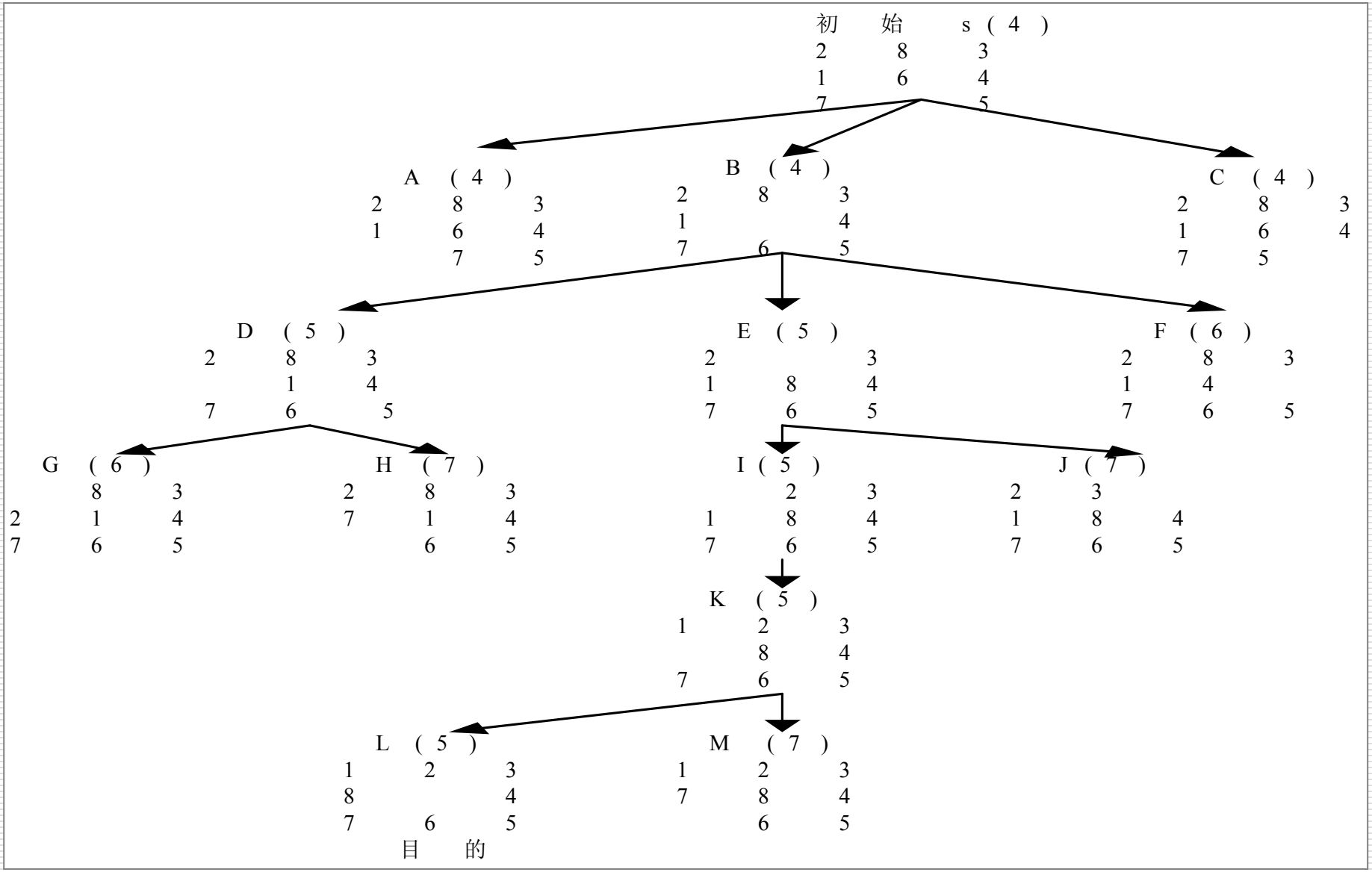
⑩ **例5.8** 利用A搜索算法求解八数码问题的搜索树，其估价函数定义为

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

- $d(n)$：状态的深度，每步为单位代价。
- $w(n)$：以“不在位”的将牌数作为启发信息的度量。

- $h^*(n)$：为状态 n到目的状态的最优路径的代价。
- $w(n)=h(n)\le h^*(n)$：A搜索算法$\rightarrow A^*$搜索算法。

5.4.3 *A*搜索算法


初 始 s (4)
2 8 3
1 6 4
7 5
A (4)
2 8 3
1 6 4
7 5
B (4)
2 8 3
1 4
7 6 5
C (4)
2 8 3
1 6 4
7 5
D (5)
2 8 3
1 4
7 6 5
E (5)
2 3
1 8 4
7 6 5
F (6)
2 8 3
1 4
7 6 5
G (6)
8 3
2 1 4
7 6 5
H (7)
2 8 3
7 1 4
6 5
I (5)
2 3
1 8 4
7 6 5
J (7)
2 3
1 8 4
7 6 5
K (5)
1 2 3
8 4
7 6 5
L (5)
1 2 3
8 4
7 6 5
M (7)
1 2 3
7 8 4
6 5
目 的

5.4.3 *A*搜索算法

- open表和closed表内状态排列的变化情况

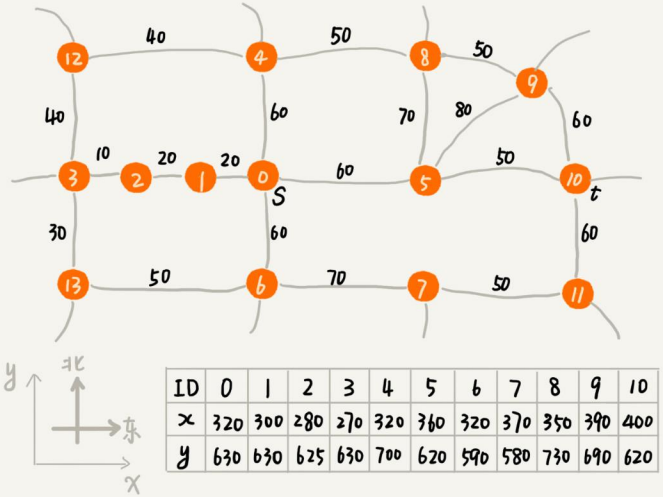
Open 表	Closed 表
初始化：(s(4))	(...)
一次循环后：	
(B(4) A(6) C(6) F(6))	(s(4))
二次循环后：	
(D(5) E(5) A(6) C(6) F(6))	(s(4) B(4))
三次循环后：	
(E(5) A(6) C(6) F(6) G(6) H(7))	(s(4) B(4) D(5))
四次循环后：	
(I(5) A(6) C(6) F(6) G(6) H(7) J(7))	(s(4) B(4) D(5) E(5))
五次循环后：	
(K(5) A(6) C(6) F(6) G(6) H(7) J(7))	(s(4) B(4) D(5) E(5) I(5))
六次循环后：	
(L(5) A(6) C(6) F(6) G(6) H(7) J(7) M(7))	(s(4) B(4) D(5) E(5) I(5) K(5))
七次循环后：	
L为目的状态，则成功推出，结束搜索	(s(4) B(4) D(5) E(5) I(5) K(5) L(5))

5.4.4 *A**搜索算法及其特性分析

- 定义h*(n)为状态n到目的状态的最优路径的代价，则当A搜索算法的启发函数h(n)小于等于h*(n)，即满足
- $h(n) \le h^*(n)$，　对所有节点n
- 时，被称为A*算法

5.4 A*搜索策略

- 一个真实的地图，每个顶点在地图中的位置，我们用一个二维坐标（x，y）来表示，其中，x 表示横坐标，y 表示纵坐标。



ID	0	1	2	3	4	5	6	7	8	9	10
x	320	300	280	270	320	360	320	370	350	390	400
y	630	630	625	630	700	620	590	580	730	690	620

  - Dijkstra 算法
    - 优先级队列，来记录已经遍历到的顶点以及这个顶点的路径长度
    - 尽管找的是从 s 到 t 的路线，但是最先被搜索到的顶点依次是 1，2，3
    - 没有考虑到这个顶点到终点的距离

5.4 A*搜索策略

- 启发函数（h(i)）
  - 欧几里得距离：顶点跟终点之间的直线距离
  - 曼哈顿距离（Manhattan distance）：两点之间横纵坐标的距离之和。

```
int hManhattan(Vertex v1, Vertex v2) { // Vertex表示顶点，后面有定义
  return Math.abs(v1.x - v2.x) + Math.abs(v1.y - v2.y);
}
```

  - 估价函数
    - 通过两者之和 f(i)=g(i)+h(i)，来判断哪个顶点该最先出队列

5.4 A*搜索策略

- 图 **Graph** 类的定义

```
private class Vertex {
  public int id; // 顶点编号ID
  public int dist; // 从起始顶点，到这个顶点的距离，也就是g(i)
  public int f; // 新增：f(i)=g(i)+h(i)
  public int x, y; // 新增：顶点在地图中的坐标（x，y）
  public Vertex(int id, int x, int y) {
    this.id = id;
    this.x = x;
    this.y = y;
    this.f = Integer.MAX_VALUE;
    this.dist = Integer.MAX_VALUE;
  }
}
// Graph类的成员变量，在构造函数中初始化
Vertex[] vertexes = new Vertex[this.v];
// 新增一个方法，添加顶点的坐标
public void addVetex(int id, int x, int y) {
  vertexes[id] = new Vertex(id, x, y)
}
```

5.4 A*搜索策略

- A* 算法的代码实现的主要逻辑

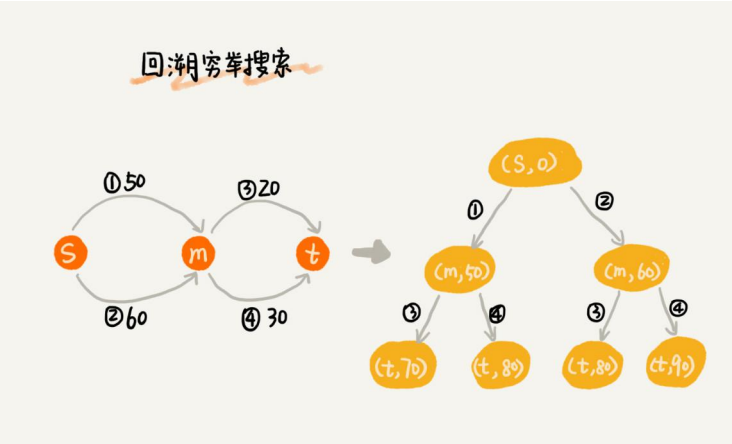
```
public void astar(int s, int t) { // 从顶点s到顶点t的路径
  int[] predecessor = new int[this.v]; // 用来还原路径
  // 按照vertex的f值构建的小顶堆，而不是按照dist
  PriorityQueue queue = new PriorityQueue(this.v);
  boolean[] inqueue = new boolean[this.v]; // 标记是否进入过队列
  vertexes[s].dist = 0;
  vertexes[s].f = 0;
  queue.add(vertexes[s]);
  inqueue[s] = true;
  while (!queue.isEmpty()) {
    Vertex minVertex = queue.poll(); // 取堆顶元素并删除
    for (int i = 0; i < adj[minVertex.id].size(); ++i) {
      Edge e = adj[minVertex.id].get(i); // 取出一条minVetex相连的边
      Vertex nextVertex = vertexes[e.tid]; // minVertex-->nextVertex
      if (minVertex.dist + e.w < nextVertex.dist) { // 更新next的dist,f
        nextVertex.dist = minVertex.dist + e.w;
        nextVertex.f
            = nextVertex.dist+hManhattan(nextVertex, vertexes[t]);
        predecessor[nextVertex.id] = minVertex.id;
        if (inqueue[nextVertex.id] == true) {
          queue.update(nextVertex);
        } else {
          queue.add(nextVertex);
          inqueue[nextVertex.id] = true;
        }
      }
      if (nextVertex.id == t) { // 只要到达t就可以结束while了
        queue.clear(); // 清空queue，才能推出while循环
        break;
      }
    }
  }
  // 输出路径
  System.out.print(s);
  print(s, t, predecessor); // print函数请参看Dijkstra算法的实现
}
```

5.4 A*搜索策略

- **A* 算法 与Dijkstra 算法的主要区别**
  - **优先级队列构建的方式不同。A* 算法是根据 f 值（ f(i)=g(i)+h(i)）来构建优先级队列，而 Dijkstra 算法是根据 dist 值（ g(i)）来构建优先级队列；**
  - **A* 算法在更新顶点 dist 值的时候，会同步更新 f 值；**
  - **循环结束的条件不一样。Dijkstra 算法是在终点出队列的时候才结束，A* 算法是一旦遍历到终点就结束。**

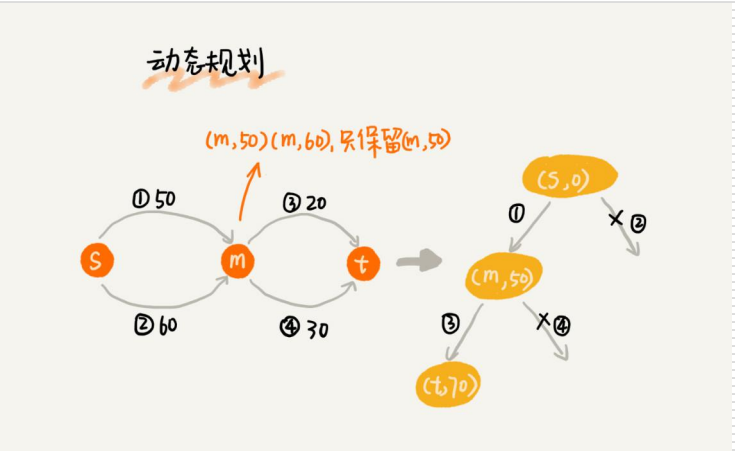
5.4 A*搜索策略

- 尽管 A* 算法可以更加快速的找到从起点到终点的路线，但是它并不能像 Dijkstra 算法那样，找到最短路线。为什么？
  - 最简单的方法是，通过回溯穷举所有从 s 到达 t 的不同路径，然后对比找出最短的那个。不过很显然，回溯算法的执行效率非常低，是指数级的。



5.4 A*搜索策略

- 尽管 A* 算法可以更加快速的找到从起点到终点的路线，但是它并不能像 Dijkstra 算法那样，找到最短路线。为什么？
  - Dijkstra 算法利用动态规划的思想，对回溯搜索进行了剪枝，只保留起点到某个顶点的最短路径，继续往外扩展搜索。动态规划相较于回溯搜索，只是换了一个实现思路，但它实际上也考察到了所有从起点到终点的路线，所以才能得到最优解。



5.4 A*搜索策略

- 尽管 A* 算法可以更加快速的找到从起点到终点的路线，但是它并不能像 Dijkstra 算法那样，找到最短路线。为什么？
  - A* 算法之所以不能像 Dijkstra 算法那样，找到最短路径，主要原因是两者的 while 循环结束条件不一样。
  - A* 算法利用贪心算法的思路，每次都找 f 值最小的顶点出队列，一旦搜索到终点就不再继续考察其他顶点和路线了。所以，它并没有考察所有的路线，也就不可能找出最短路径了

5.4 A*搜索策略

- 如何借助 A* 算法解决今天的游戏寻路问题？
  - 把地图，抽象成图
  - 把整个地图分割成一个一个的小方块。在某一个方块上的人物，只能往上下左右四个方向的方块上移动。
    - 把每个方块看作一个顶点。
    - 两个方块相邻，就在它们之间，连两条有向边，并且边的权值都是 1。
    - 在一个有向有权图中，找某个顶点到另一个顶点的路径问题。
    - 将地图抽象成边权值为 1 的有向图之后，就可以套用 A* 算法，来实现游戏中人物的自动寻路功能了。

5.4.4 *A**搜索算法及其特性分析

- 如果某一问题有解，那么利用*A**搜索算法对该问题进行搜索则一定能搜索到解，并且一定能搜索到最优的解而结束。
- 上例中的八数码*A*搜索树也是*A**搜索树，所得的解路（*s*，*B*，*E*，*I*，*K*，*L*）为最优解路，其步数为状态*L*（5）上所标注的5 。

5.4.4 A^*搜索算法及其特性分析

1. 可采纳性

当一个搜索算法在最短路径存在时能保证找到它，就称它是可采纳的。

2. 单调性

搜索算法的单调性：在整个搜索空间都是局部可采纳的。一个状态和任一个子状态之间的差由该状态与其子状态之间的实际代价所限定。

3. 信息性

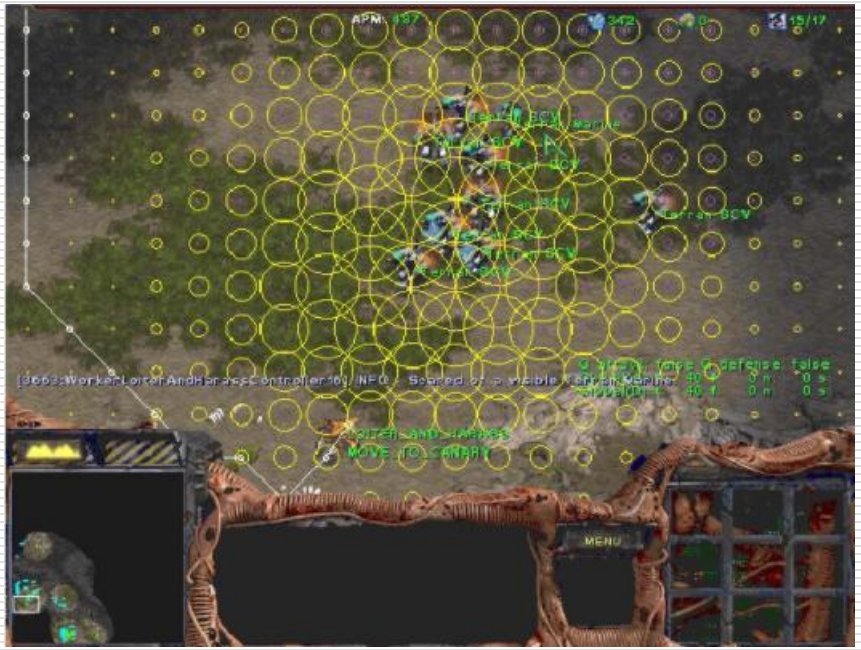
在两个A^*启发策略的h_1和h_2中，如果对搜索空间中的任一状态n都有$h_1(n) \le h_2(n)$，就称策略h_2比h_1具有更多的信息性。

5.4.4 *A**搜索算法及其特性分析

- h(n)的信息性通俗点说其实就是在估计一个节点的值时的约束条件，如果信息越多或约束条件越多则排除的节点就越多，估价函数越好或说这个算法越好。
- 这就是为什么广度优先算法的不甚为好的原因了，因为它的h(n)=0，没有一点启发信息。但在游戏开发中由于实时性的问题，h(n)的信息越多，它的计算量就越大，耗费的时间就越多。就应该适当的减小h(n)的信息，即减小约束条件。但算法的准确性就差了，这里就有一个平衡的问题。

5.4.4 A*搜索算法及其特性分析

- 应用
  - 视频游戏
  - 路径/路由规划问题
  - 资源规划问题
  - 机器人移动规划
  - 语言分析机器翻译
  - 语音识别
  - ……

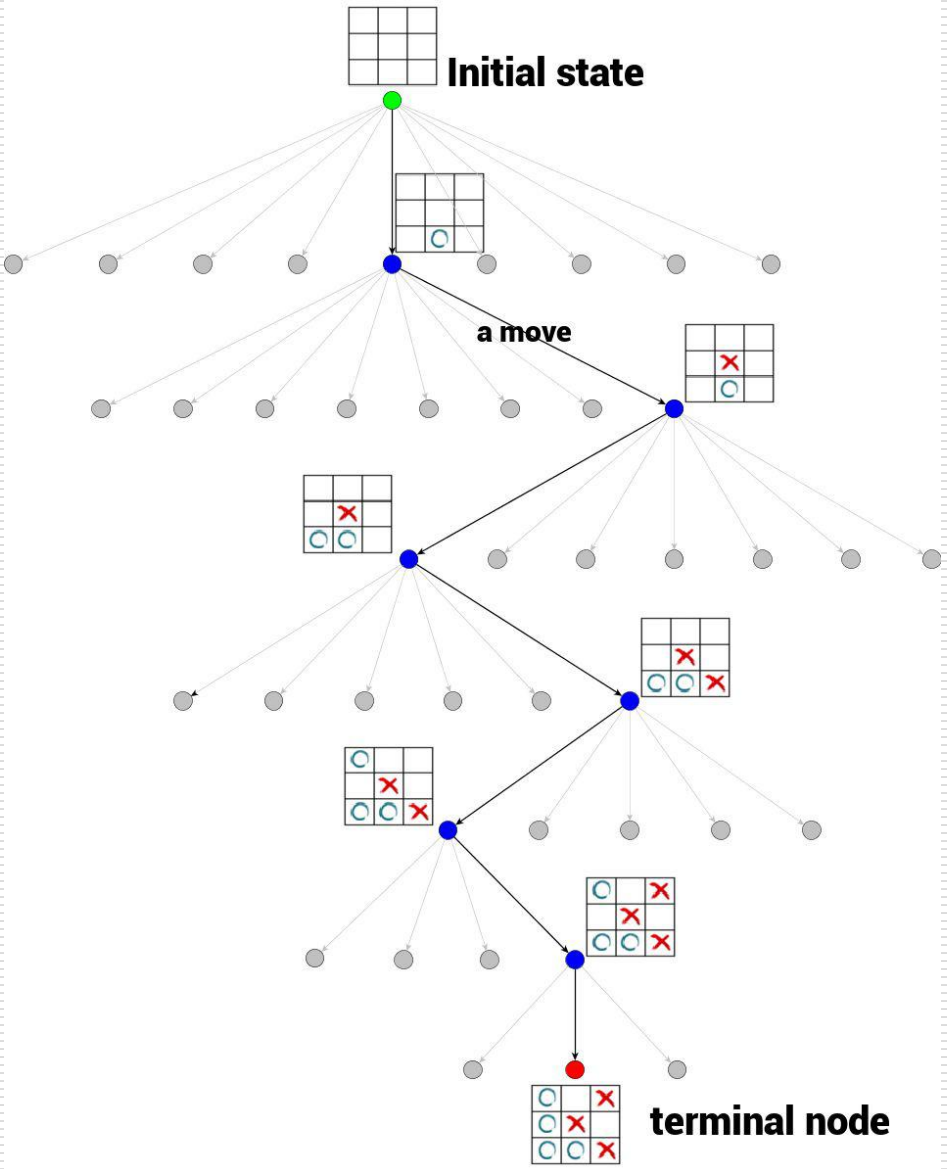


5.4.1 启发式策略

- 启发式算法
  - 重点在于如何设计并实现启发函数，使我们能够更快地获得较优解
- 进一步改进
  - 改进估值函数
  - 针对不同棋类添加不同的估值规则
  - 通过神经网络等方式让AI自己学得策略

博弈树

- 博弈树
  - 一种树结构，其中每一个节点表征博弈的确定状态
  - 从一个节点向其子节点的转换被称为一个行动（move）
  - 节点的子节点数目被称为分支因子（branching factor）
  - 端节点（terminal nodes）
    - 表示博弈无法再继续进行
    - 状态可以被评估，并总结博弈的结果。

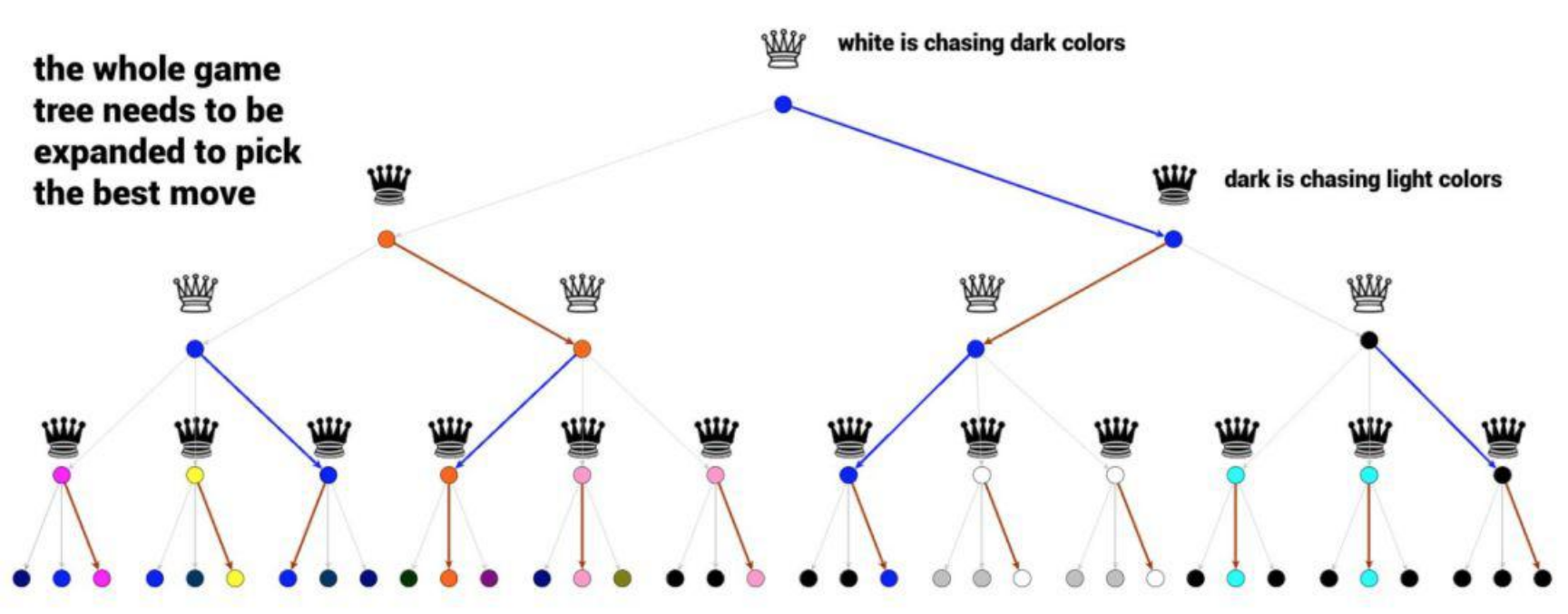


博弈树

- 博弈树是一种递归的数据结构
  - 当你选择了一个最佳行动并到达一个子节点的时候，这个子节点其实就是其子树的根节点
  - 在每一次（以不同的根节点开始），将博弈看成由博弈树表征的“寻找最有潜力的下一步行动”问题的序列
- 什么是最有潜力的下一步行动？
  - 极小极大（minimax）策略和 alpha-beta 剪枝算法

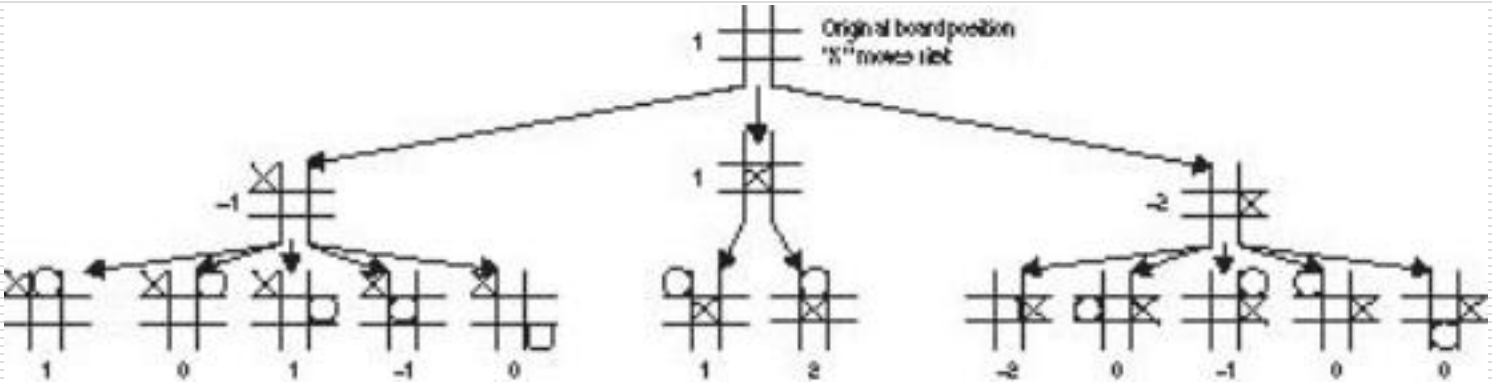
博弈树的极小极大算法

- 在假定你的对手执行最佳行动的前提下，最大化你的收益
  - 以放弃最优策略为代价，从而最小化了风险



博弈树的极小化极大评估

- 井字棋
  - 两个有经验的玩家之间的博弈往往以平局结束
  - X需要的不是遵循最快的路径取得胜利，而是在即使O阻塞了这条路径的情况下找到通往胜利的路径
- 二人博弈
  - Max：试图最大化启发式评估的玩家
  - Min：试图最小化启发式评估的玩家

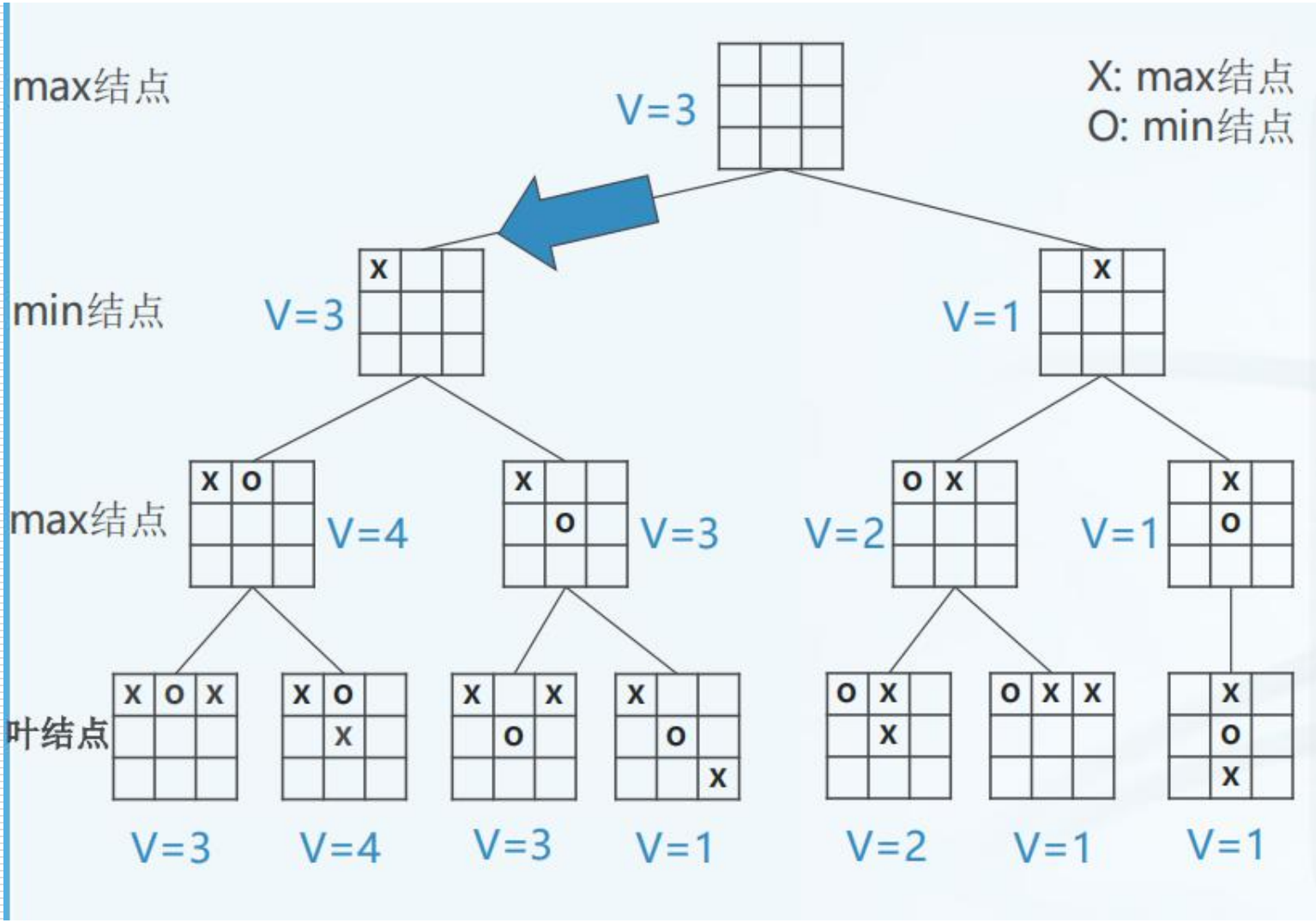


博弈树的极小化极大评估

- 构建过程
  - 构建决策树
  - 将评估函数应用于叶子结点
  - 自底向上计算每个结点的minimax值
  - 从根结点选择minimax值最大的分支，作为行动策略

博弈树的极小化极大评估

- 井字棋



博弈树的极小化极大评估

- 极小极大算法的最大弱点
  - 需要展开整个博弈树。对于有高分支因子的博弈（例如围棋或国际象棋），该算法将导致巨大的博弈树，使得计算无法进行。
  - 仅在确定的阈值深度 d 内展开博弈树
    - 无法保证在阈值深度 d 处的任何节点是否端节点
  - 通过 alpha-beta 剪枝算法来修剪博弈树

具有alpha-beta剪枝的极小化极大算法

- 相比单独使用极小化极大算法，α-β剪枝通常只需要检查大约一半的节点
- α-β剪枝基本原则
  - 在发现一个走子方式很差以后，将彻底放弃这种走子方式，不会花费额外的资源来发现这到底有多糟糕
  - 假设α为下界，β为上界，对于α ≤ N ≤ β:
    - 若 α ≤ β 则N有解
    - 若 α > β 则N无解
- 目前已被用于多个成功的博弈引擎例如 Stockfish——AlphaZero 的主要对手之一
  - （https://github.com/mcostalba/Stockfish）

具有alpha-beta剪枝的极小化极大算法

- 定义极大层的下界为alpha，极小层的上界为beta，alpha-beta剪枝规则描述如下：
  - （1）alpha剪枝。若任一极小值层结点的beta值不大于它任一前驱极大值层结点的alpha值，即alpha(前驱层) >= beta(后继层)，则可终止该极小值层中这个MIN结点以下的搜索过程。这个MIN结点最终的倒推值就确定为这个beta值。
  - （2）beta剪枝。若任一极大值层结点的alpha值不小于它任一前驱极小值层结点的beta值，即alpha(后继层) >= beta(前驱层)，则可以终止该极大值层中这个MAX结点以下的搜索过程，这个MAX结点最终倒推值就确定为这个alpha值。

具有alpha-beta剪枝的极小化极大算法

- 定义极大层的下界为alpha，极小层的上界为beta，alpha-beta剪枝规则描述如下：
  - alpha剪枝。若任一极小值层结点的beta值不大于它任一前驱极大值层结点的alpha值，即alpha(前驱层) >= beta(后继层)，则可终止该极小值层中这个MIN结点以下的搜索过程。这个MIN结点最终的倒推值就确定为这个beta值。
  - beta剪枝。若任一极大值层结点的alpha值不小于它任一前驱极小值层结点的beta值，即alpha(后继层) >= beta(前驱层)，则可以终止该极大值层中这个MAX结点以下的搜索过程，这个MAX结点最终倒推值就确定为这个alpha值。

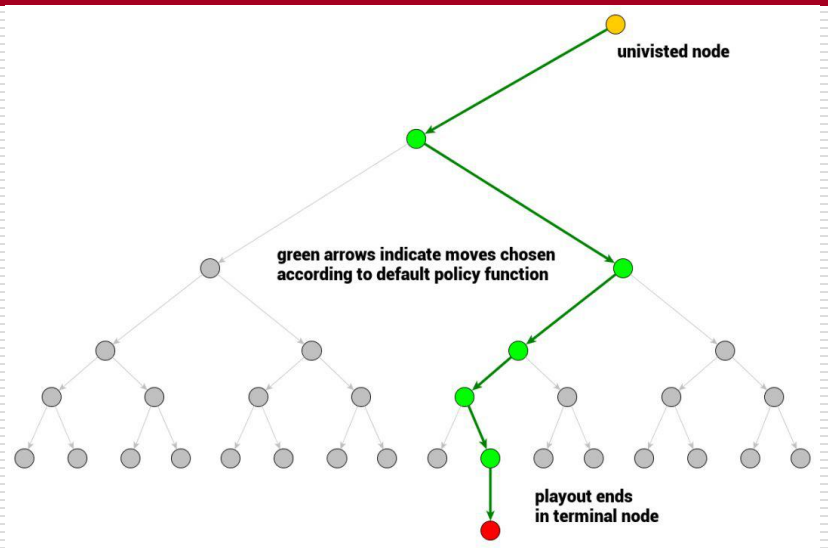
```
val = alpha_beta_valuation(board, next_player, player, alpha, beta)
board[move] = 0pen_token
if player == 0_token:   # 当前玩家是0,是Max玩家(记号是1)
    if val > alpha:
        alpha = val
    if alpha >= beta:
        return beta  # 直接返回当前的最大可能取值beta, 进行剪枝
else:  # 当前玩家是X,是Min玩家(记号是-1)
    if val < beta:
        beta = val
    if beta <= alpha:
        return alpha  # 直接返回当前的最小可能取值alpha, 进行剪枝
```

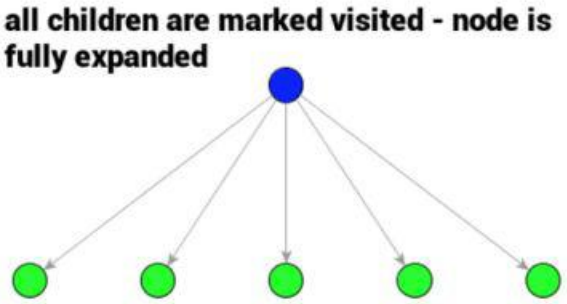
蒙特卡洛树搜索

- 给出一个“游戏状态”并选择“胜率最高的下一步”
- 多次模拟博弈，并尝试根据模拟结果预测最优的移动方案。
  - 主要概念是搜索，即沿着博弈树向下的一组遍历过程
    - 单次遍历的路径会从根节点（当前博弈状态）延伸到没有完全展开的节点，未完全展开的节点表示其子节点至少有一个未访问到。
    - 遇到未完全展开的节点时，它的一个未访问子节点将会作为单次模拟的根节点，随后模拟的结果将会反向传播回当前树的根节点并更新博弈树的节点统计数据。
    - 一旦搜索受限于时间或计算力而终止，下一步行动将基于收集到的统计数据进行决策。

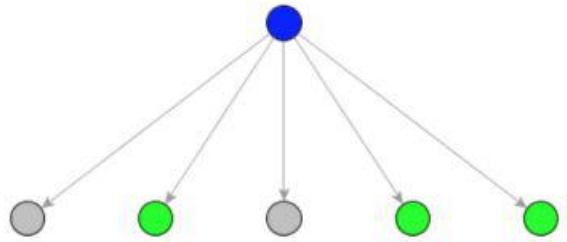
蒙特卡洛树搜索

- rollout 策略函数
  - 服从均匀分布的随机采样
- 博弈树的展开节点、完全展开节点和访问节点





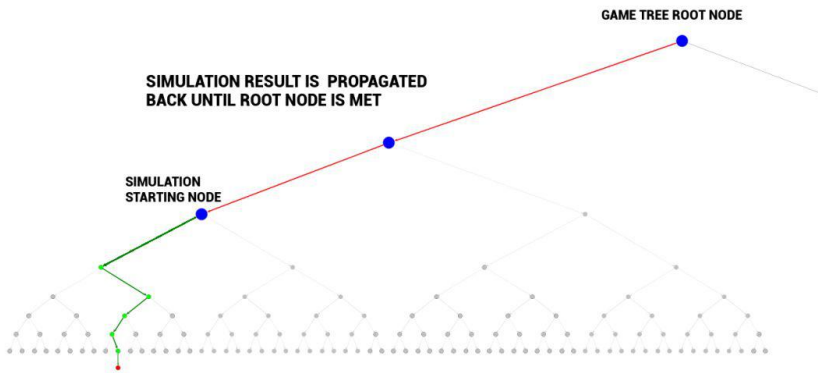
simulation/game state evaluation has been computed in all green nodes, they are marked visited

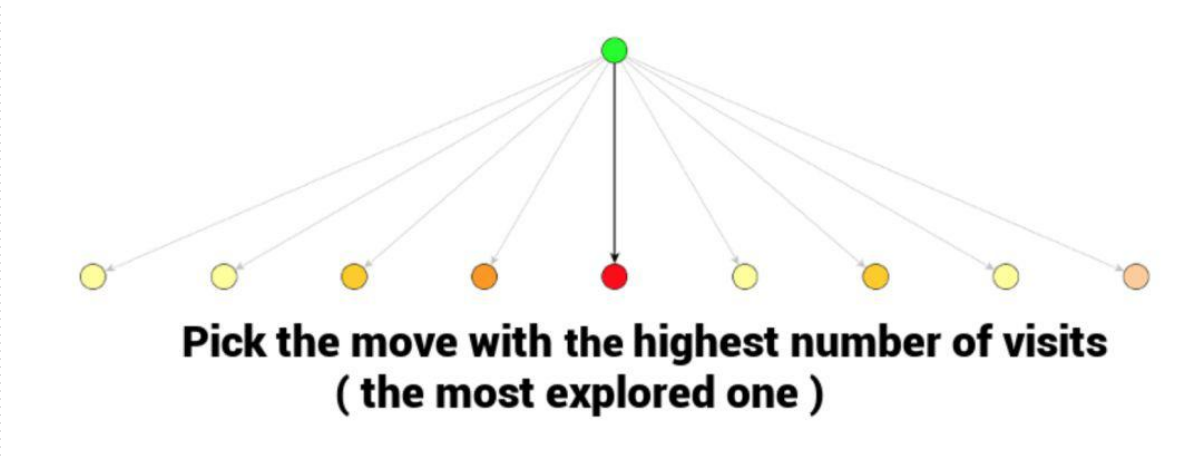


there are two nodes from where no single simulation has started - these nodes are unvisited, parent is not fully expanded

蒙特卡洛树搜索

- 反向传播
  - 将模拟结果传播回去
- 节点的统计数据
- 博弈树遍历
- 终止蒙特卡洛树搜索





- 井字棋（Tic-Tac-Toe）的实现示例：
  https://github.com/int8/monte-carlo-tree-search

总结

- 搜索与AI
  - 搜索方法在AI系统中几乎无处不在，常常是系统内核与外部模块的骨架
  - 一个自主机器人使用搜索：
    - 决定该采取的动作和执行哪一个传感操作
    - 快速的预测可能发生的碰撞
    - 轨道的规划
    - 将由大量传感器得到的数字设计翻译为符号的描述
    - 诊断一些预期的事情为何没有发生
    - ……
  - 许多搜索可能会同时发生并持续进行

THE END