第十一周 搜索问题

李泽榉，复旦大学生物医学工程与技术创新学院

课件内容参考浙江大学吴飞老师《人工智能引论》以及斯坦福CS221

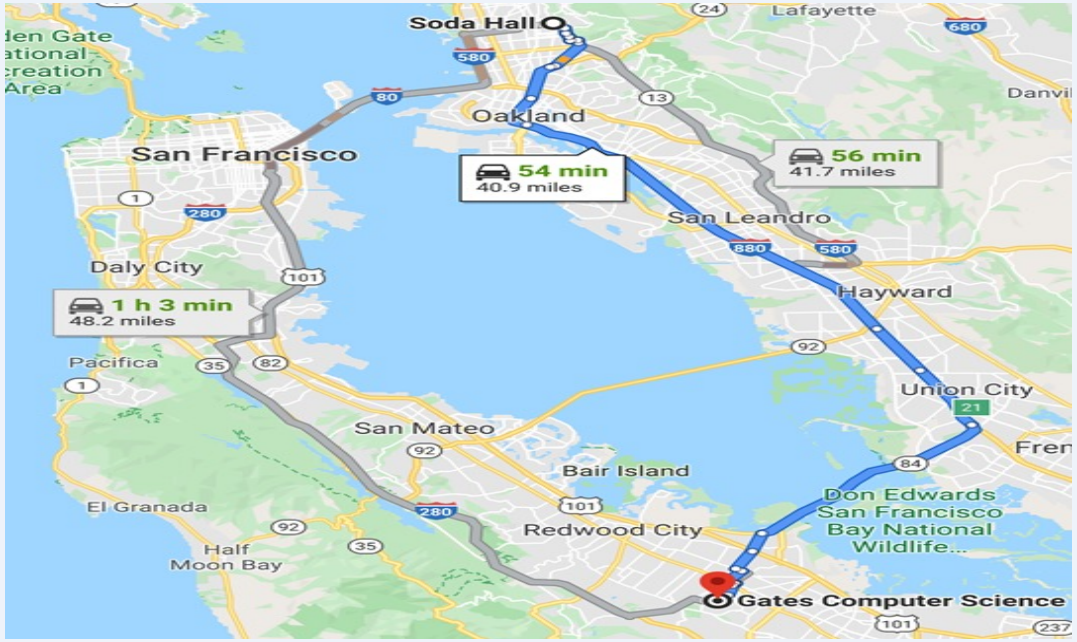
搜索问题是什么

搜索：从庞大解空间中搜索得到与问题相对应、满足一定约束条件的最佳答案

简而言之：寻找一条从**初始状态**（问题）到**目标状态**（答案）的**最佳路径**

例① 地图最短路径

从 A 点到 B 点行驶时间最短的路线

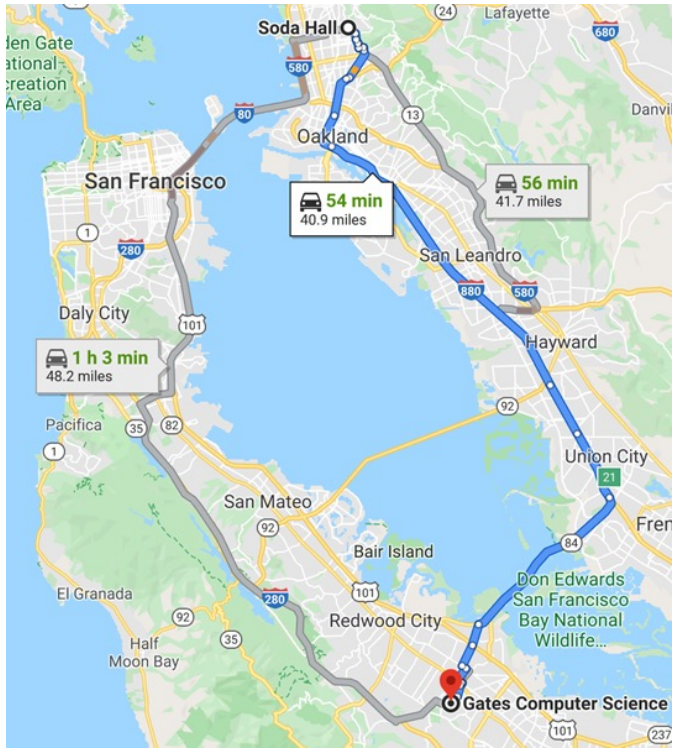


例② 数独求解

从空白格填到满足规则的完整方案

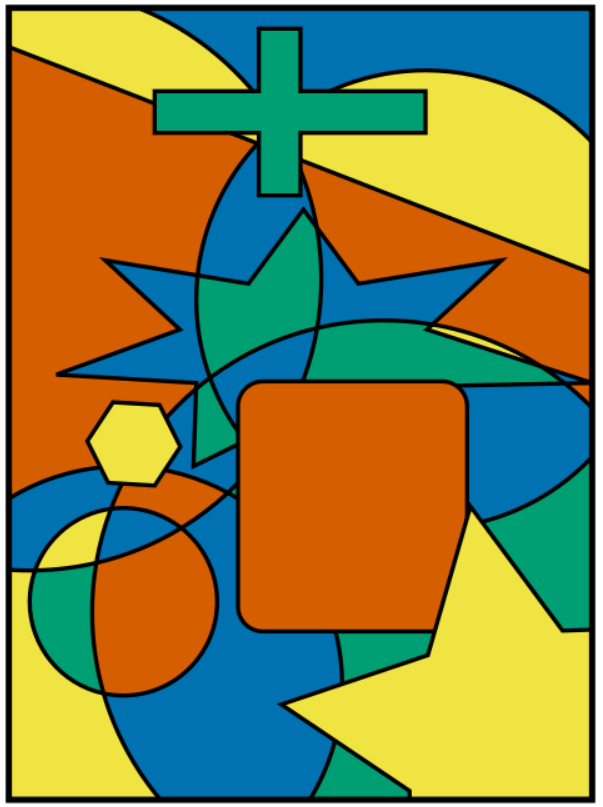
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

常见的搜索问题



路径规划

动作规划：机器臂动作规划



约束满足：地图染色问题

搜索问题应用：路径规划

路径规划是指在给定的环境中找到从起点到终点的最佳路径的过程。它在现实生活中有着广泛的应用，包括**计算机网络**、**物流配送**、**机器人导航**等领域。

- **确定目的**：不同的问题有不同的目的，如：选择**最短路径** vs. 选择**用时最短路径**
- **选择行动方式**：针对不同的目的，在每个分歧点选择**最佳行动方案**

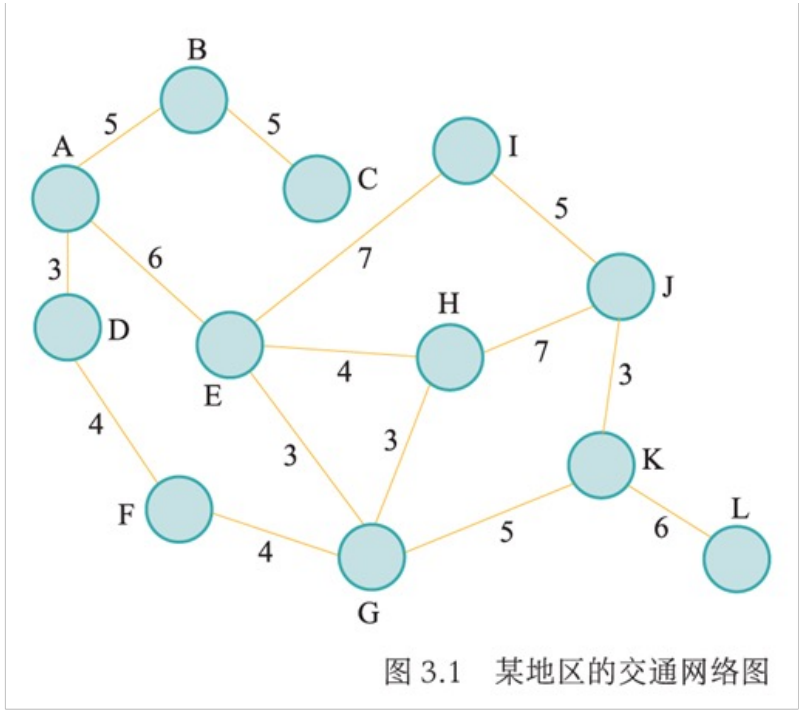


图 3.1　某地区的交通网络图

搜索问题应用：动作规划

Shakey the Robot 诞生于 1960 年代，是**第一个能够感知和推理周围环境的移动机器人**。

这个早期的机器人成为后来构建机器人的**原型**，并对现代**机器人技术**和**人工智能技术**产生了重大影响。

1970 年 11 月 20 日，《生活》杂志将 Shakey 称为**"第一个电子人"**。

其运动规划问题即是典型的动作规划（搜索）问题。

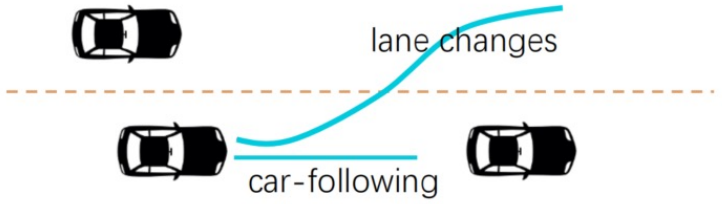
Charles Rosen与他制造的“第一个电子人”Shakey

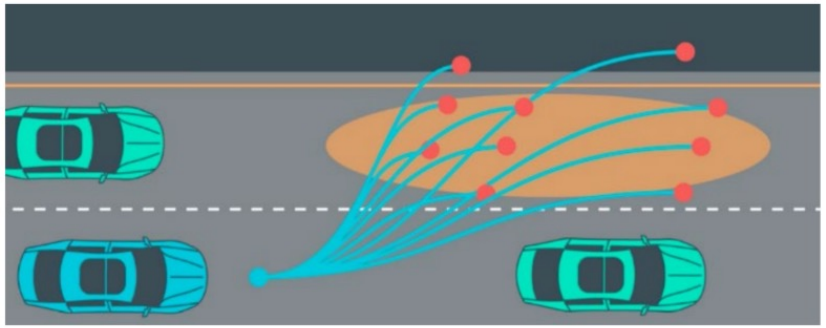
搜索问题应用：动作规划

动作规划：将上游行为决策的指令解释成一条带有**时间信息的轨迹曲线**，最后给底层的反馈控制。应用于**机械臂**、**扫地机器人**、**无人驾驶**等领域。

动作规划可以被拆分成两个子问题：

- **轨迹规划**（Trajectory Planning）：二维平面上的**优化轨迹**问题
- **速度规划**（Speed Planning）：选定轨迹后，以何种**速度**行驶该轨迹





搜索问题应用：约束满足

约束满足：对于一组对象，需要满足一些**限制或条件**。问题单元表示为变量上**有限条件的同质集合**。常用**搜索方法**求解。

- **四色定理**：平面地图可用**四种颜色**染色，使任意邻接区域颜色不同
- **数独问题**：每行、每列、每宫均须包含 1～9，不重复、不缺漏

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

搜索 vs. 检索

- 本周的"搜索" ≠ 搜索引擎（百度 / 谷歌）的搜索
- 搜索引擎的"搜索"本质是**检索**——从大数据库中匹配关键词；之所以叫"搜索"，是因为答案无法预先存储，用户有"搜到答案"的体验
- 本课"搜索" = **搜索类优化**：从众多候选方案中，通过不断搜索找到最优解（方案无法预存，只能动态探索）

目录

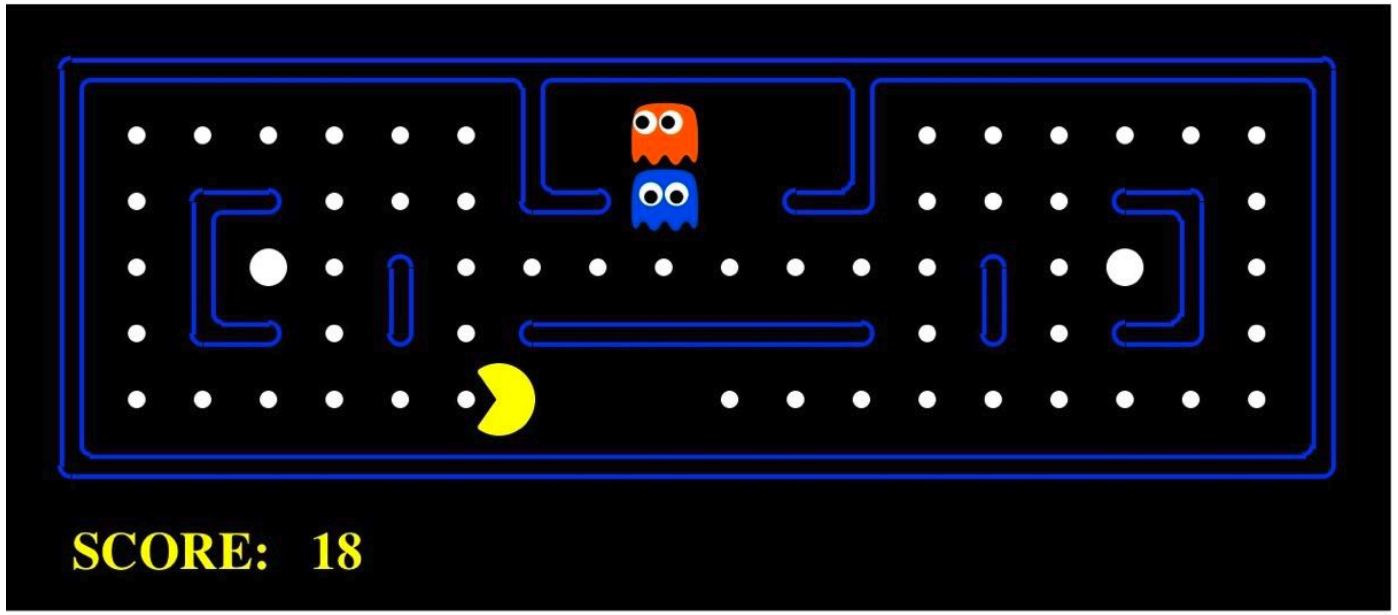
搜索基本概念

搜索问题的四要素

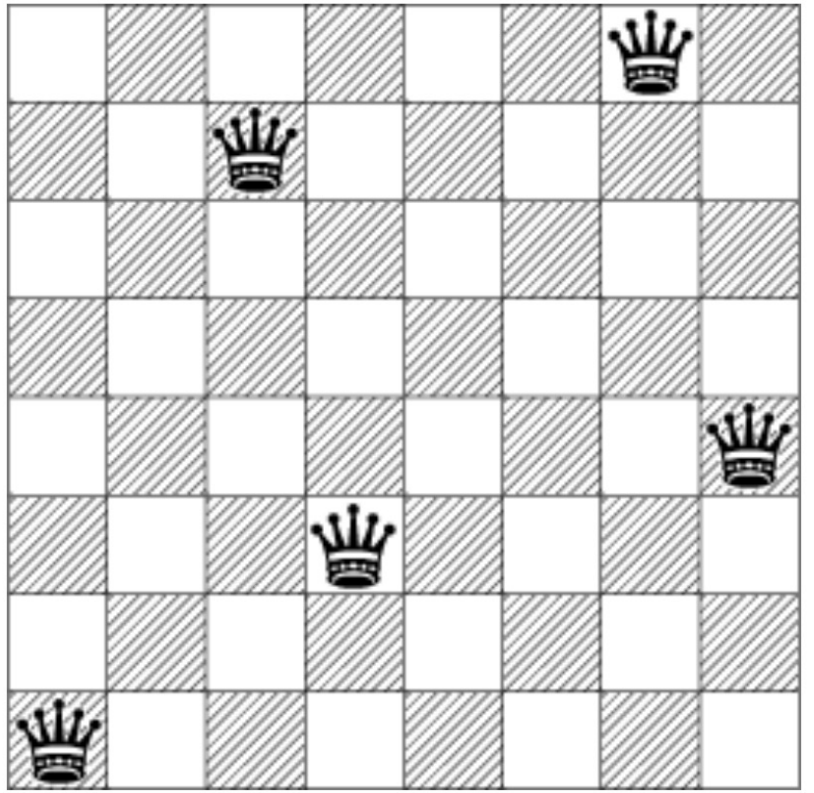
- 状态/状态空间
  - 状态描述一个具体的场景
  - 状态空间包含了所有的可能状态
- 后继函数（动作、损耗）
  - 状态通过动作选择而产生连接的关系
  - 动作空间表示某一个状态下可以采取的动作集合
- 开始状态
  - 问题开始的状态
- 结束测试
  - 问题结束的条件

搜索基本概念

典型问题：



吃豆人游戏

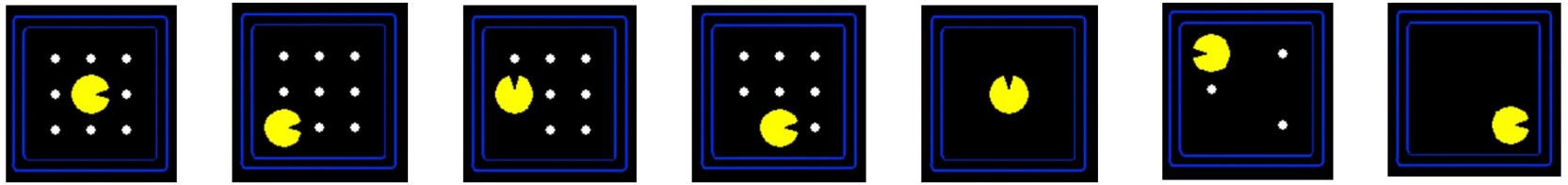


八皇后问题

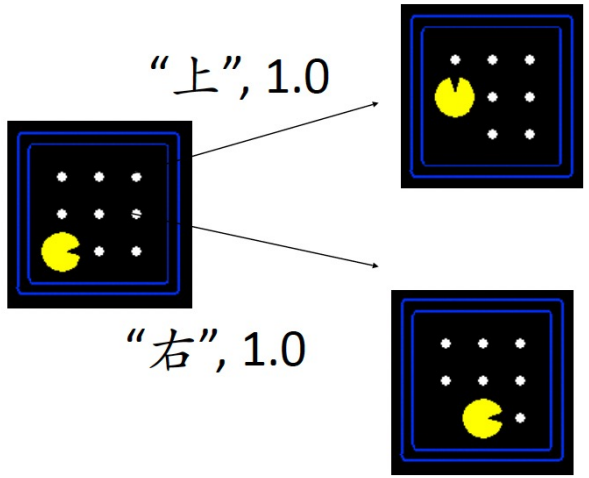
搜索基本概念

吃豆人游戏

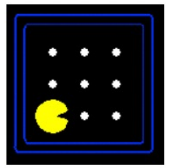
- 状态空间



- 后继函数
  - 动作空间：上，下，左，右
  - 损耗：单步损耗为 1



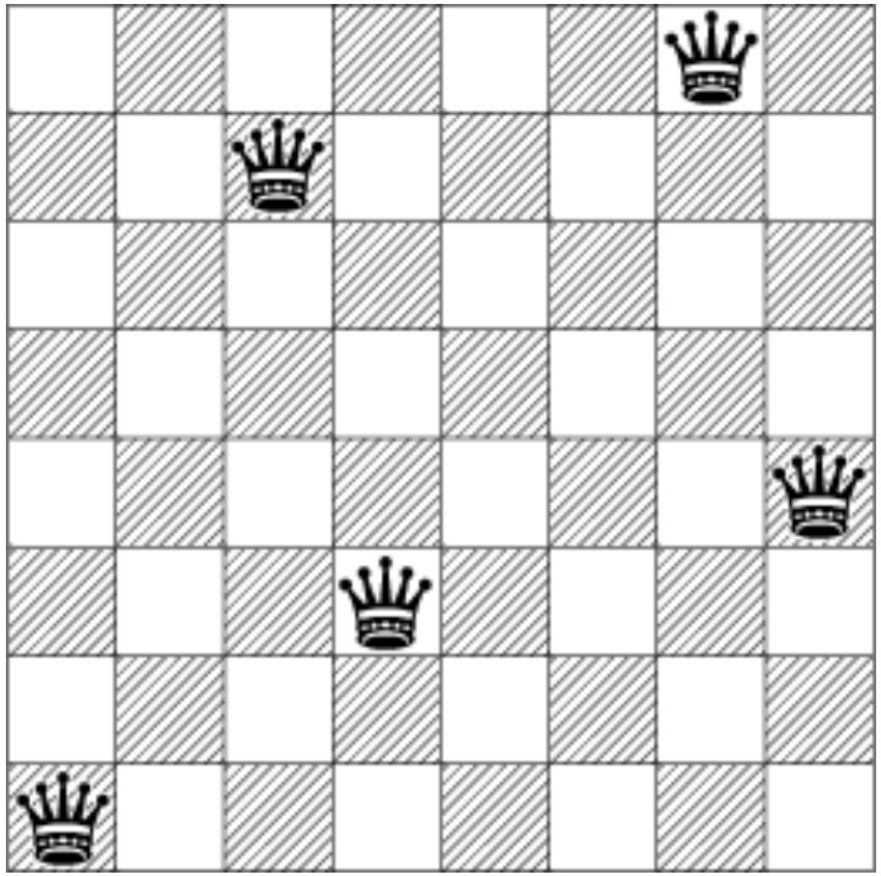
- 开始状态

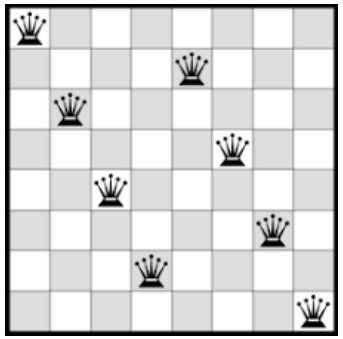


- 目标测试
  - 移动到某一个特定位置

搜索基本概念

八皇后问题



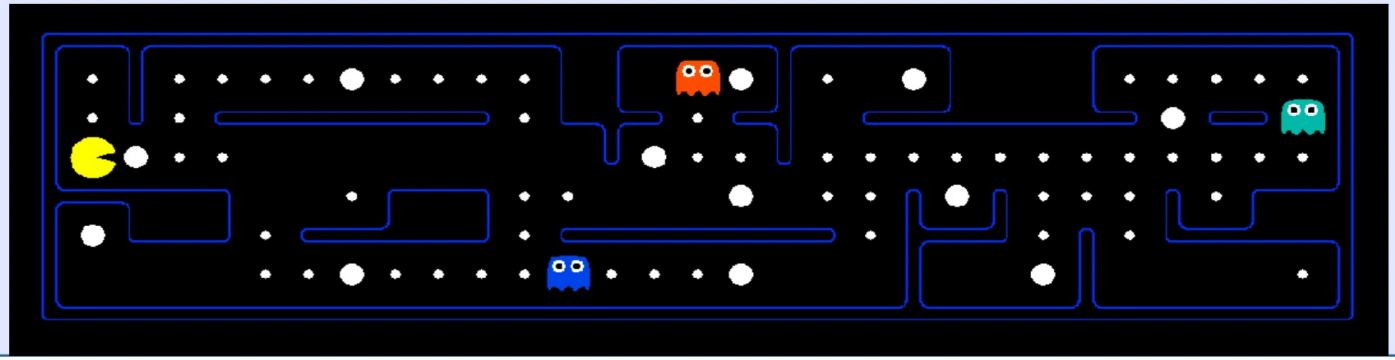


- 状态空间
  - 8个皇后摆在棋盘上的位置集合
- 后继函数
  - 增加一个皇后到棋盘上
- 开始状态
  - 空白的棋盘
- 目标测试
  - 8个皇后在棋盘上，两两不攻击

搜索基本概念

状态空间

全局状态空间：建模环境中包含的每一个可描述变量（细节）



搜索状态空间：仅需要建模解决特定搜索问题需要的变量

- 问题： 路径寻址
  - 状态：(x,y) 小黄人的位置
  - 动作： 上下左右
  - 后继： 根据动作更新位置
  - 目标： 当前状态是否目标位置
- 问题： 吃完所有豆子
  - 状态：{(x,y), 豆子分布情况}
  - 动作： 上下左右
  - 后继： 根据动作状态
  - 目标： 所有豆子都被吃了

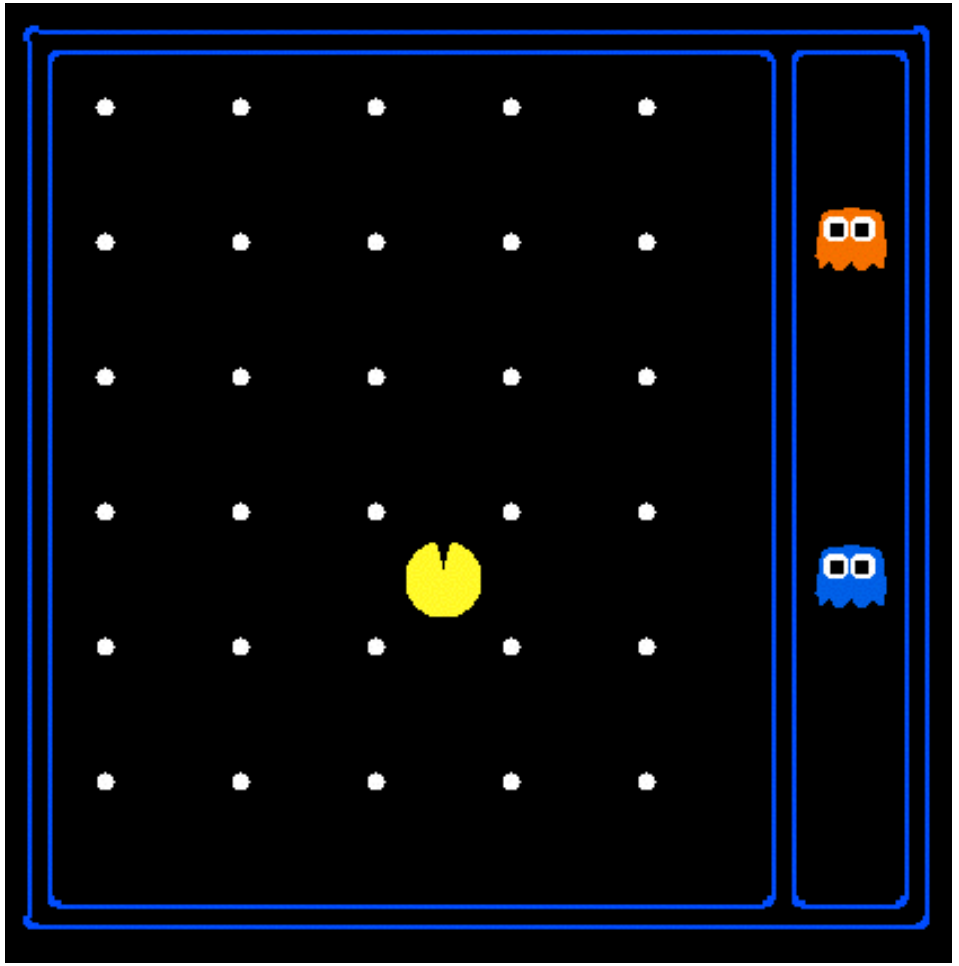
搜索基本概念

- 全局状态空间:
  - 小黄人活动空间：120
  - 豆子个数：30
  - 鬼怪活动空间：12
  - 小黄人动作：上下左右
- 全局状态空间:

  120x(2^{30})x(C_{12}^{2})x4
- 路径寻址问题的状态空间:

  120
- 吃完所有豆子问题的状态空间:

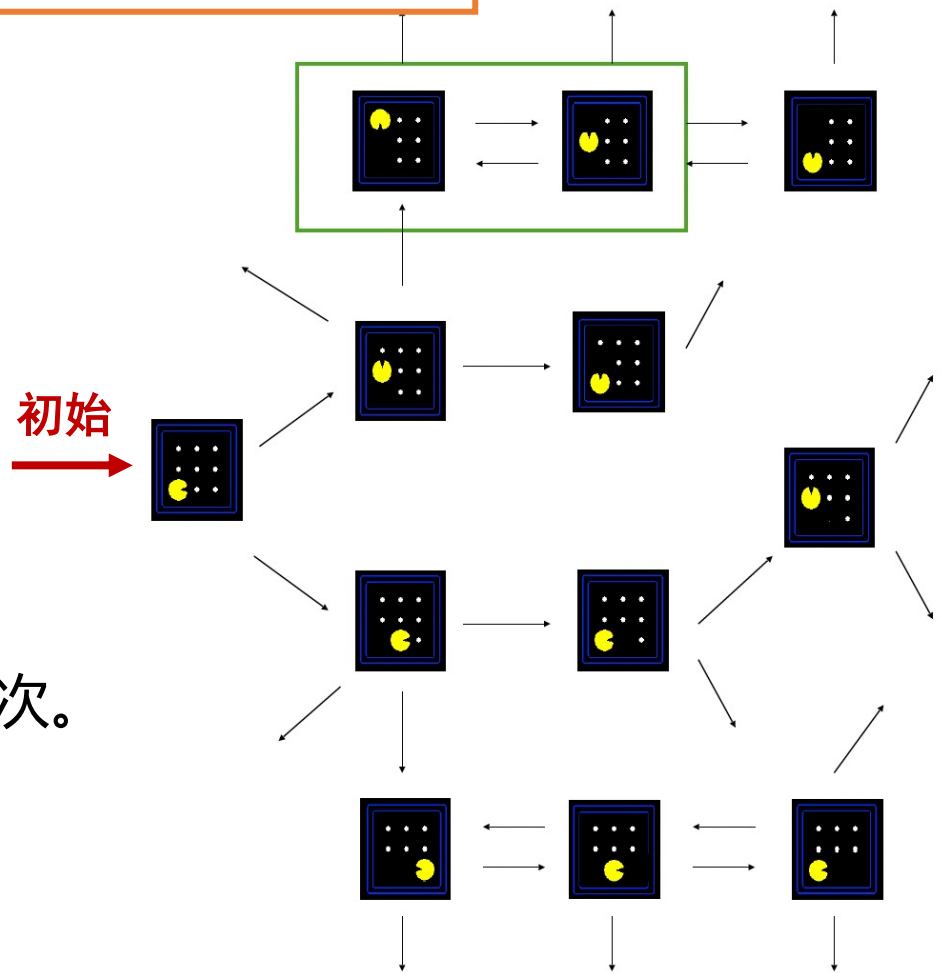
  120x(2^{30})



搜索基本概念

状态空间图

- 一种针对搜索问题的数学表示形式
  - 节点：抽象的状态
  - 边：后继函数（状态执行动作的结果）
  - 目标检测：目标节点的集合（一个/多个）

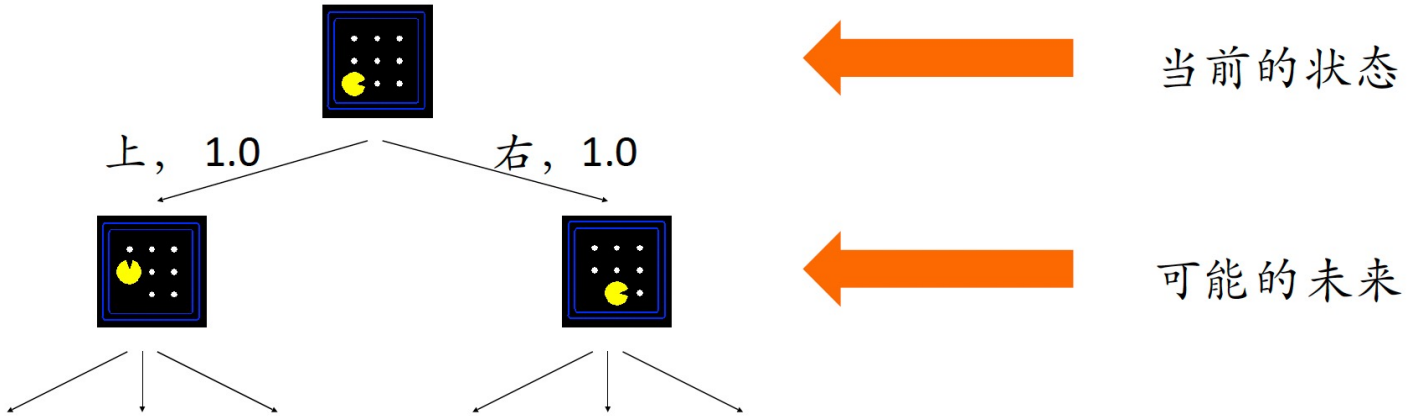


- 在状态空间图中，每一个问题中的状态只出现一次。
- 状态空间图中的节点与问题中的状态一一对应。

搜索基本概念

搜索树

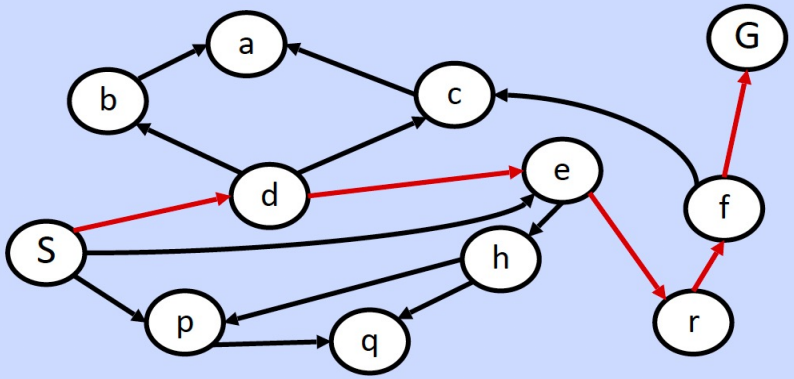
- 树是有向无环图，有根节点，边的个数是节点个数减一
- 开始状态是根节点
- 孩子节点对应父亲节点的后继状态
- 节点对应状态，并且包含了从开始状态达到当前状态的路径
- 路径表示一个动作序列



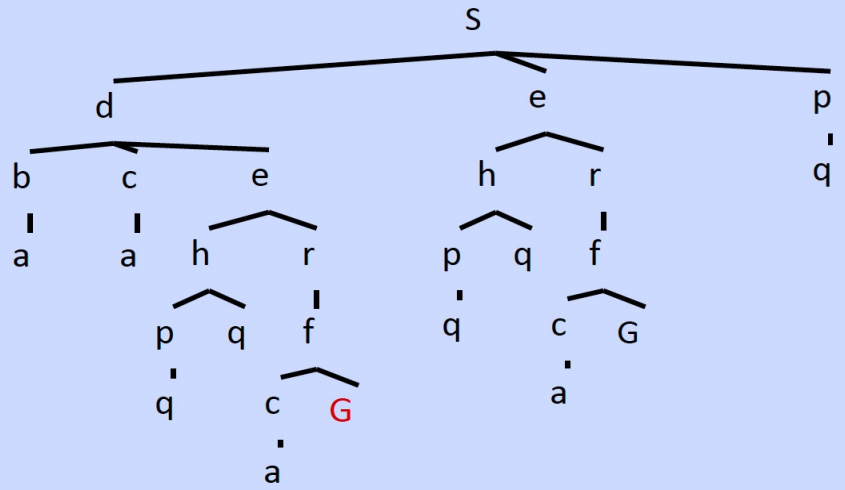
搜索基本概念

状态图——搜索树

状态空间图



搜索树

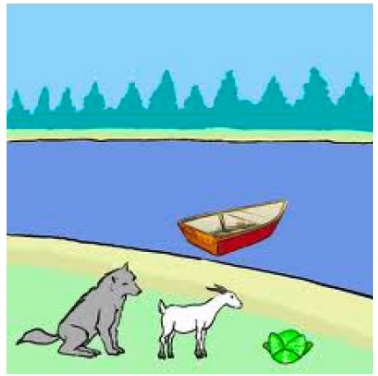


状态a，在树中的不同位置出现。它实际上代表了不同的从s到a的方式

- 搜索树中的一个节点对应状态空间图中的一条完整路径
- 状态空间图中的节点可以对应到搜索树中的多个状态节点

搜索基本概念

过河问题：一位农夫带着一头狼，一只羊和一筐白菜过河，河边有一条小船，农夫划船每次只能载狼、羊、白菜三者中的一个过河。农夫不在旁边时，狼会吃羊，羊会吃白菜。问农夫该如何过河。



状态对象：Farmer Cabbage Goat Wolf

后继函数：$F\triangleright$ $F\triangleleft$

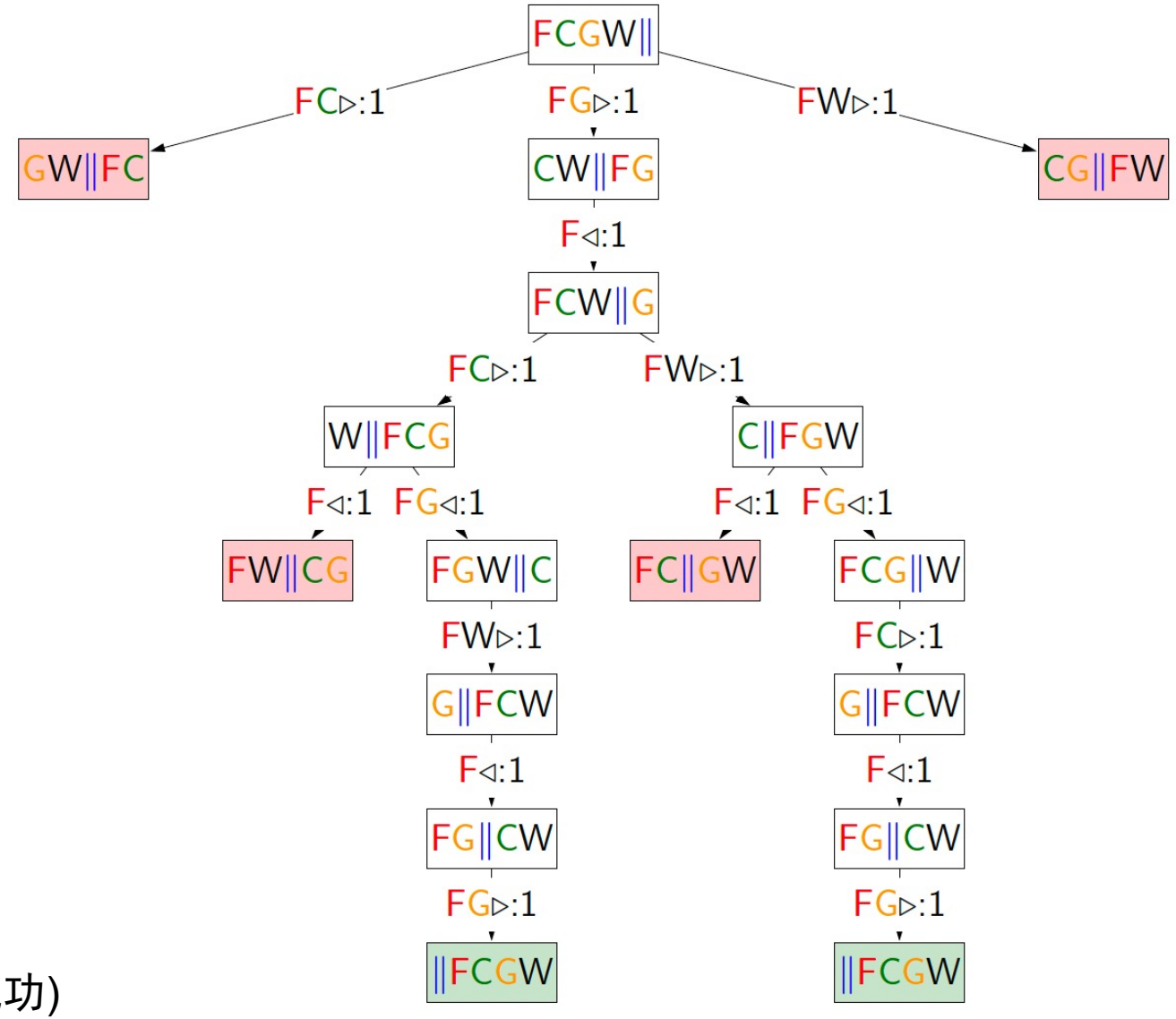
$FC\triangleright$ $FC\triangleleft$

$FG\triangleright$ $FG\triangleleft$

$FW\triangleright$ $FW\triangleleft$

开始状态：FCGW||

目标检测：狼和山羊/山羊和白菜在同一边 (失败)
农夫、狼、羊、白菜都在河的另一边(成功)



目录

搜索算法分类

搜索问题的四要素

- 状态/状态空间
  - 状态描述一个具体的场景
  - 状态空间包含了所有的可能状态
- 后继函数（动作、损耗）
  - 状态通过动作选择而产生连接的关系
  - 动作空间表示某一个状态下可以采取的动作集合
- 开始状态
  - 问题开始的状态
- 结束测试
  - 问题结束的条件

无信息搜索算法是指除了搜索四要素信息之外，没有其他任何信息来源的搜索算法，主要有：

- 回溯搜索
- 深度优先搜索
- 广度优先搜索
- 代价一致搜索（+动态规划）

启发式搜索算法是指在搜索的过程中，通过一些启发式的信息指导下一步搜索方向的搜索算法，主要有：

- 贪婪最优搜索
- A*搜索

博弈搜索算法是指两个（多个）智能体通过竞争对抗，最大化自身利益而进行的搜索算法，主要有：

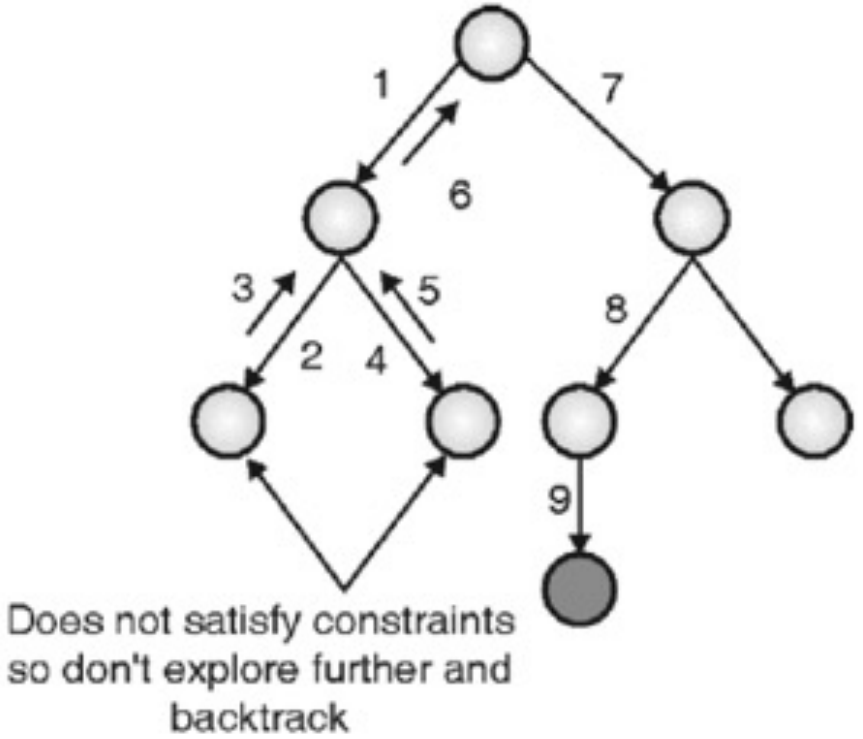
- Minimax搜索
- Alpha-beta剪枝
- 蒙特卡洛树

回溯搜索 （Backtracking search）

核心思想：无法满足条件时回溯

如果每个状态执行b个动作，D表示最大搜索深度：

- 空间复杂度：O(D)
- 时间复杂度：O(b^D)



回溯搜索 （Backtracking search）

算法伪代码：

Algorithm: backtracking search

```
def backtrackingSearch(s, path):
    If IsEnd(s): update minimum cost path
    For each action a ∈ Actions(s):
        Extend path with Succ(s, a) and Cost(s, a)
        Call backtrackingSearch(Succ(s, a), path)
    Return minimum cost path
```

回溯搜索 （Backtracking search）

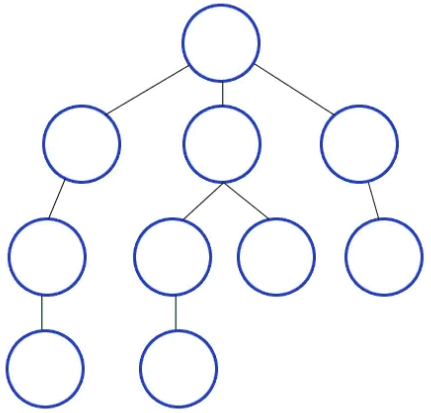
代码示例：

```
def backtrackingSearch(problem):
    best = {
        'cost': float('inf'),
        'history': None
    }
    def recurse(state, history, totalCost):
        # at state, having undergone history, with totalCost so far
        if problem.isEnd(state):
            # update the best solution
            if totalCost<best['cost']:
                best['cost'] = totalCost
                best['history'] = history
            return
        for action, newState, cost in problem.succAndCost(state):
            recurse(newState, history+[(action, newState, cost)], totalCost+cost)
    recurse(problem.startState(), [], 0)
    return (best['cost'], best['history'])
```

深度优先（Depth first search）

Assumption: zero action costs

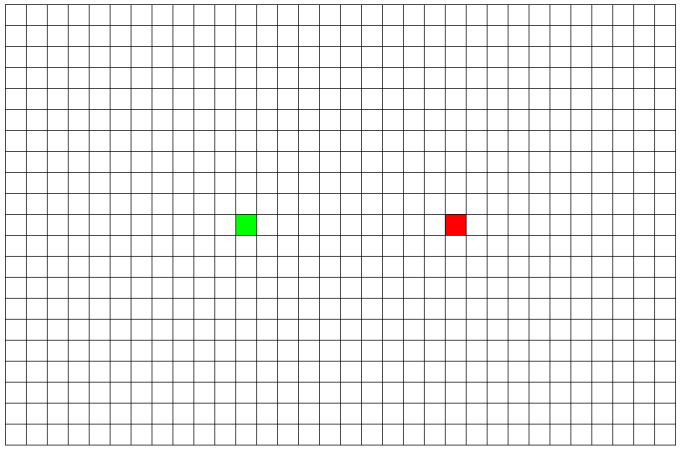
Assume action costs $\text{Cost}(s, a) = 0$.



核心思想：当找到第一个终止状态时：回溯搜索+暂停

如果每个状态执行b个动作，D表示最大搜索深度：

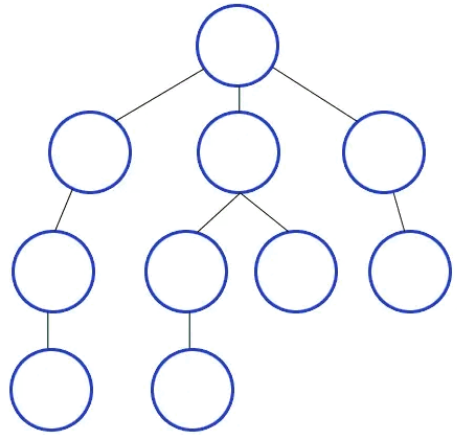
- 空间复杂度：O(D)
- 时间复杂度：O(b^D)



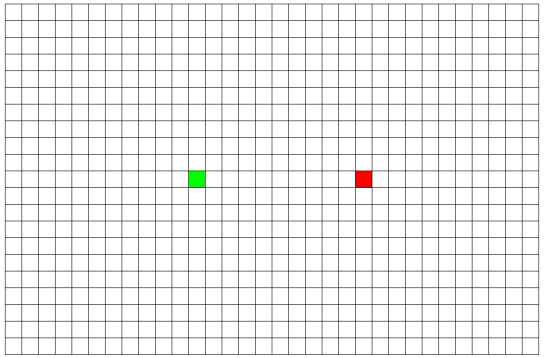
广度优先（Breath first search）

Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.



核心思想：按深度增加的顺序探索所有节点



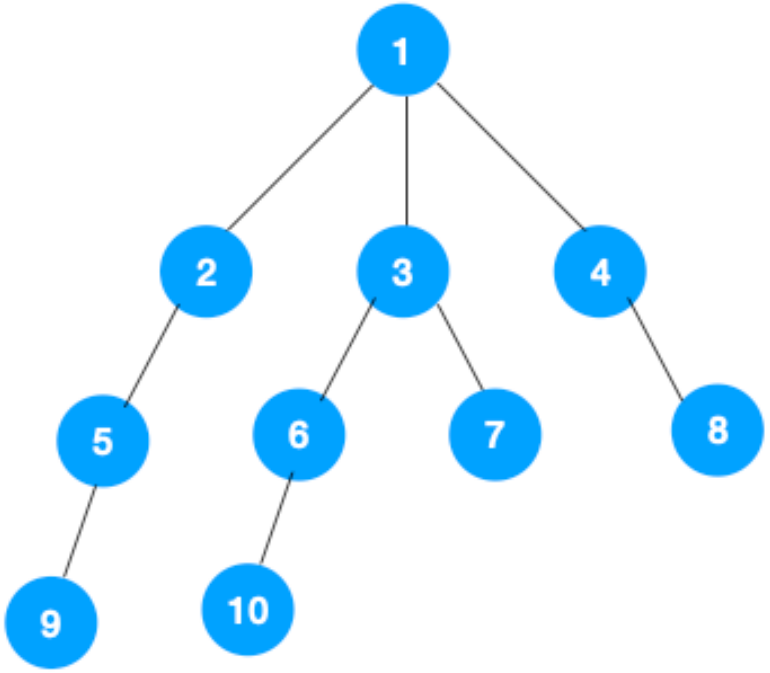
如果每个状态执行b个动作，结果产生d个动作：

- 空间复杂度：O(b^d) (相较之前变得更差了！)
- 时间复杂度：O(b^d) (更好，取决于d而不是D)

迭代加深DFS（DFS-ID）

Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.



核心思想：

- 修改 DFS 以在最大深度处停止
- 主动设定 DFS 最大深度

如果每个状态执行b个动作，结果产生d个动作：

- 空间复杂度：O(d) (变小了！)
- 时间复杂度：$O(b^d)$ (=BFS)

§设定最大搜索深度为 1，运行深度优先搜索算法，未发现目标
§设定最大搜索深度为 2，运行深度优先搜索算法，未发现目标
§设定最大搜索深度为 3，.....

各搜索算法复杂度比较

每个状态所需的动作数 b， 解决深度 d， 最大深度 D：

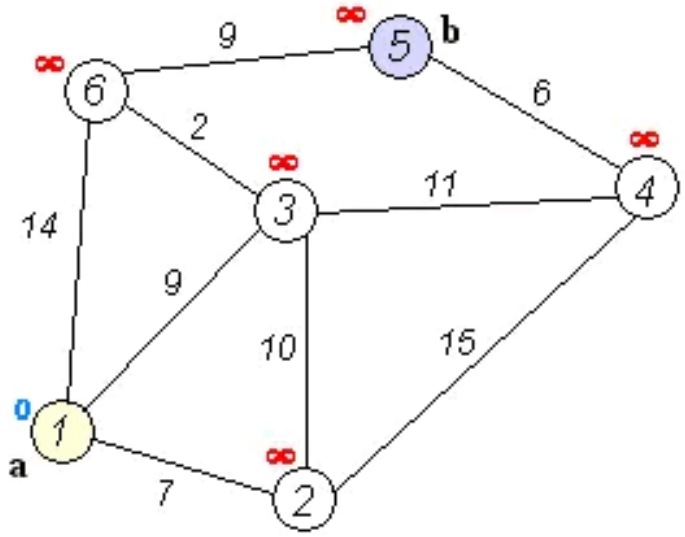
Algorithm	Action costs	Space	Time
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

解决深度 d：找到最优解的深度；**Action costs**：每一步操作的代价。**DFS** 标注 zero，不显式考虑动作成本，只关注路径是否存在；**BFS/DFS-ID** 标注 constant ≥ 0，可处理固定非零成本，优化步数；**Backtracking** 标注 any，支持任意成本（但通常不保证最优性）。

代价一致搜索（Uniform Cost Search）

核心思想：

- 按照路径代价高低来决定扩展节点的次序。
- 主要关心搜索路径的**总代价**（action cost ≠0），不关心路径的**步数**（action cost=0）。选择从根节点出发路径最短的节点，因此被称为**代价最优最先**

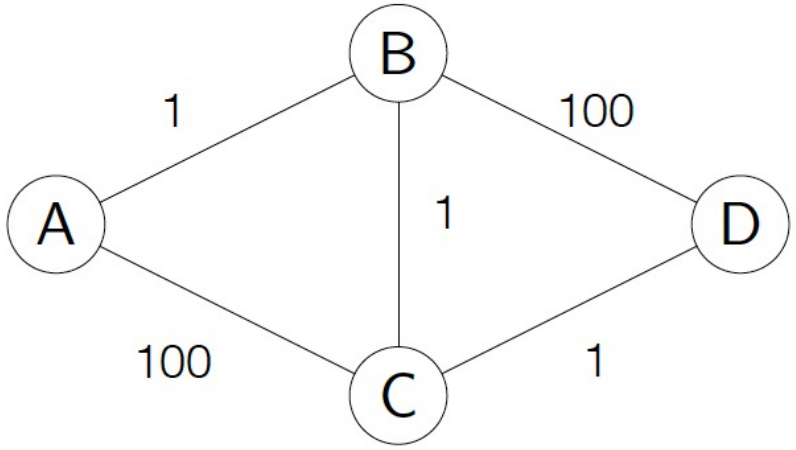


如果每个状态执行b个动作，结果产生d个动作：

- 空间复杂度：O(d) (变小了！)
- 时间复杂度：O(b^d) (=BFS)

代价一致搜索（Uniform Cost Search）

Example: UCS example



Start state: A, end state: D

Minimum cost path:

$A \rightarrow B \rightarrow C \rightarrow D$ with cost 3

动态规划

动态规划是解决多阶段决策过程的一种数学方法。主要思想是把多阶段问题变换为一系列相互联系的单阶段问题，然后逐个加以解决。

什么样的问题可以用**动态规划**来解？：

（1）重复子问题

- 递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次，这种性质称为子问题的重叠性质。
- 动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
- 通常不同的子问题个数随问题的大小呈多项式增长，用动态规划算法只需要多项式时间，从而获得较高的解题效率

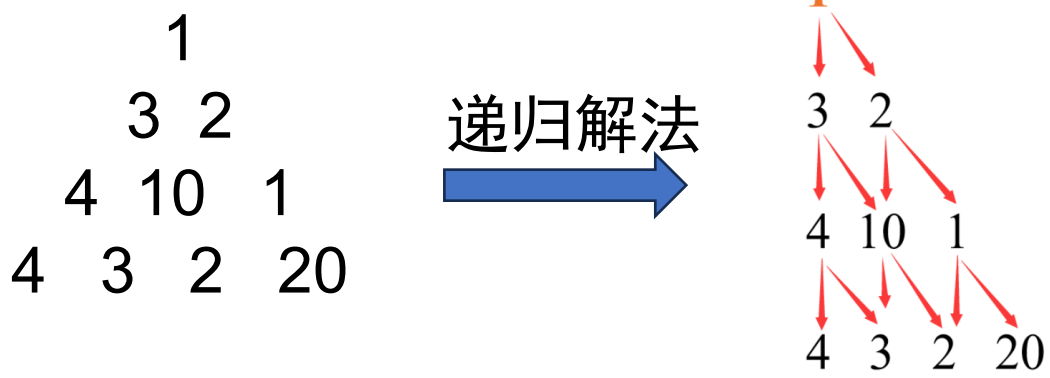
（2）最优子结构

- 一个问题的最优解包含着其子问题的最优解，这种性质称为最优子结构性质。
- 分析问题的最优子结构性质：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。
- 利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。
- **最优子结构是一个问题能用动态规划算法求解的前提**

引用：https://blog.csdn.net/qq_43466788/article/details/109627650

例子说明

如图所示，在数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字和最大。



方法1：经典递归解法

如果我们采用暴力法来解决这个问题，从第一层开始每一层都有两个选择，n层情况下时间复杂度为 2^n

引用：https://blog.csdn.net/qq_43466788/article/details/109627650

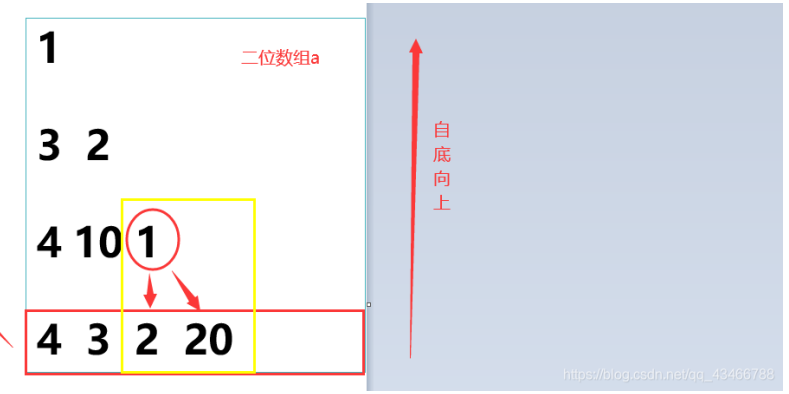
例子说明

如图所示，在数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字和最大。

```
        1
      3   2
    4   10   1
  4   3   2   20
```

动态规划法

p[i][j]	0	1	2	3
0	24			
1	16	23		
2	8	13	21 =a[2][2]+Max(p[3][2],p[3][3])	
3	4	3	2	20



方法2：动态规划法

思路1：自下而上

$$\text{递归方程 } p[i][j] = \begin{cases} a[i][j] & \text{当} i = n\text{时} \\ \max\{p[i+1][j], p[i+1][j+1]\} + a[i][j] & \text{当} 1 <= i < n\text{时} \end{cases}$$

引用：https://blog.csdn.net/qq_43466788/article/details/109627650

例子说明

如图所示，在数字三角形中寻找一条从顶部到底边的路径，使得路径上所经过的数字和最大。

```
        1
      3   2
    4   10   1
  4   3   2   20
```

动态规划法

p[i][j]	1	2	3	4
1	1			
2	4	3		
3	8	14	4	
4	12	17	16	24

```
1
3   2
4   10   1
4   3   2   20
```


方法2：动态规划法

思路2：自上而下

递归方程 $$p[i][j] = \begin{cases} a[1][1] & 当i = 1时 \\ \max\{p[i-1][j-1], p[i-1][j]\} + a[i][j] & 当1 < i <= n时 \end{cases}$$

最终时间复杂度由2^n变为n^2

练习时间

3. （5 分）在解决迷宫寻路问题时，广度优先搜索（BFS）和深度优先搜索（DFS）是两种常用的盲目搜索策略。假设有一个树形结构的状态空间，起始节点为根节点，目标节点位于某一深度位置。以下关于 BFS 和 DFS 的描述，哪一项是正确的？

A. DFS 总是能找到从起点到目标的最短路径

B. BFS 在找到目标节点时，保证该路径是最短的（步数最少）

C. 在内存使用方面，BFS 通常比 DFS 更节省空间

D. DFS 按层次遍历，能保证先找到较浅层的目标节点

练习时间

3. （5 分）在解决迷宫寻路问题时，广度优先搜索（BFS）和深度优先搜索（DFS）是两种常用的盲目搜索策略。假设有一个树形结构的状态空间，起始节点为根节点，目标节点位于某一深度位置。以下关于 BFS 和 DFS 的描述，哪一项是正确的？

A. DFS 总是能找到从起点到目标的最短路径

B. BFS 在找到目标节点时，保证该路径是最短的（步数最少）

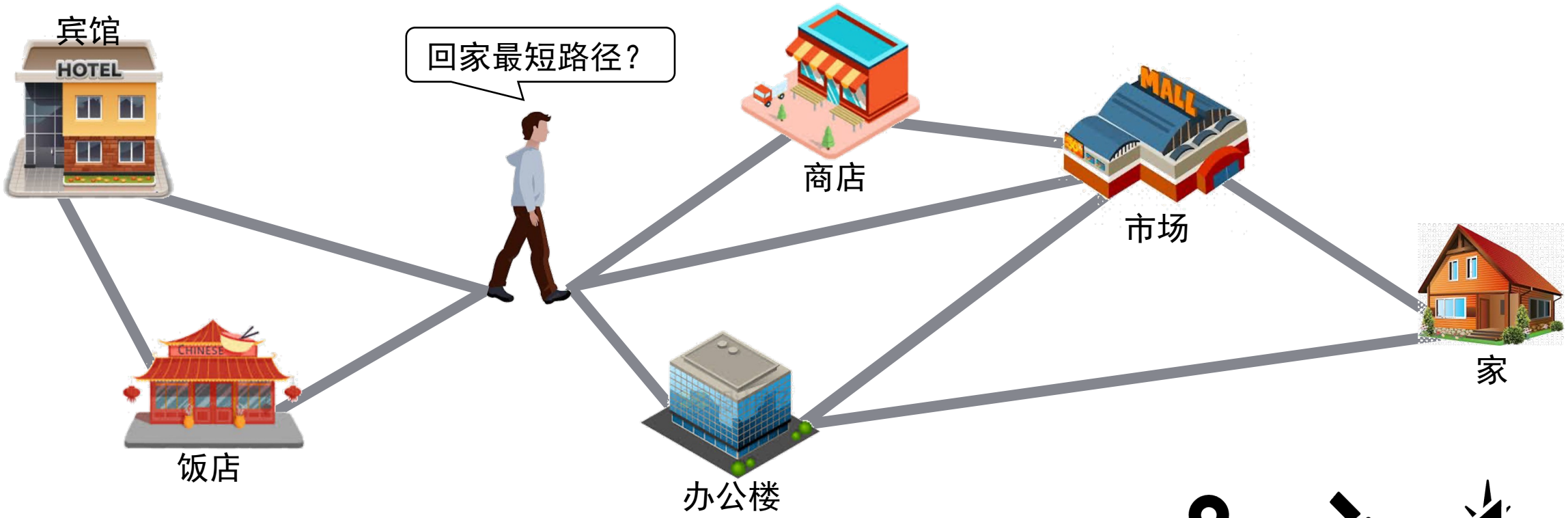
C. 在内存使用方面，BFS 通常比 DFS 更节省空间

D. DFS 按层次遍历，能保证先找到较浅层的目标节点

答案：B

目录

State的探索



无信息搜索：

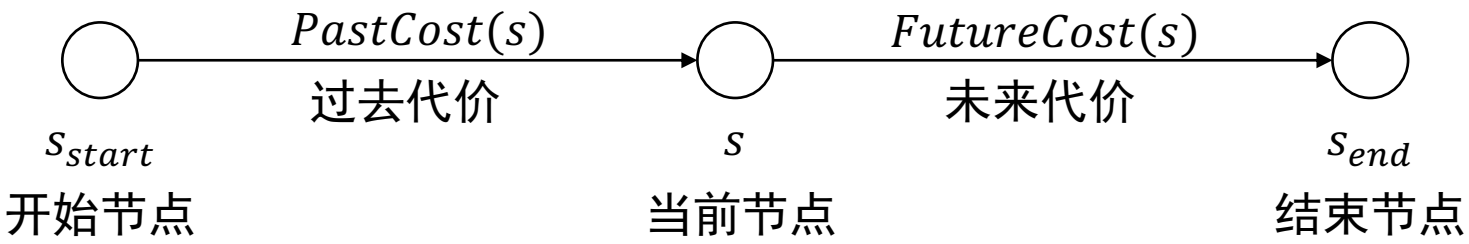
- 回溯，广度优先，深度优先：“漫无目的”地遍历
- 代价一致搜索：看一步走一步

启发式搜索：有导航的辅助

启发式信息：

指在搜索算法中，用于指导和优化搜索过程的额外信息，通常用于帮助算法更快地找到最优解。启发式信息通常由启发式函数（heuristic function）提供，它根据当前状态对达到目标状态的“代价”或“距离”进行估计

State的探索



最理想状态：同时探索PastCost(s)+FutureCost(s)

代价一致搜索：探索遍历PastCost(s)的所有状态

贪婪最佳优先搜索：只探索h(s)

A*搜索：同时探索PastCost(s)+h(s)

定义：启发式函数

启发式函数h(s)，评估从某个状态 n 到目标状态的预估代价或距离。引导搜索方向，降低复杂度

使用h(s)而不是直接使用FutureCost(s)的原因：

未来代价一般是未知的，如果已知，那么直接挑选未来代价最小的道路即可，搜索问题已经结束。因此一般使用具有一定指引性信息的启发函数h(s)，可以一定程度上指引搜索方向，压缩搜索空间。

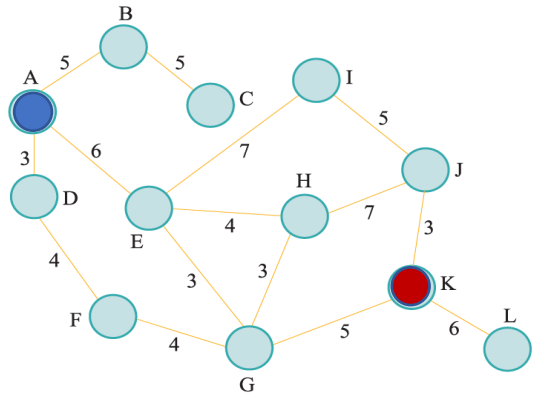
贪婪最佳优先搜索：只关注未来，目的性强

两个重要函数：评价函数与启发函数

辅助信息	所求解问题之外、与所求解问题相关的特定信息或知识。	
评价函数（evaluation function）f(n)	从当前节点n出发，根据评价函数来选择后续结点	下一个结点是谁？
启发函数（heuristic function）h(n)	计算从结点n到目标结点之间所形成路径的最小代价值，这里将两点之间的直线距离作为启发函数	完成任务还需要多少代价？

贪婪最佳优先搜索(Greedy best-first search)：评价函数f(n)=启发函数h(n)

将启发函数作为评价函数的搜索过程



求取从城市A到城市K之间一条行驶时间最短路线

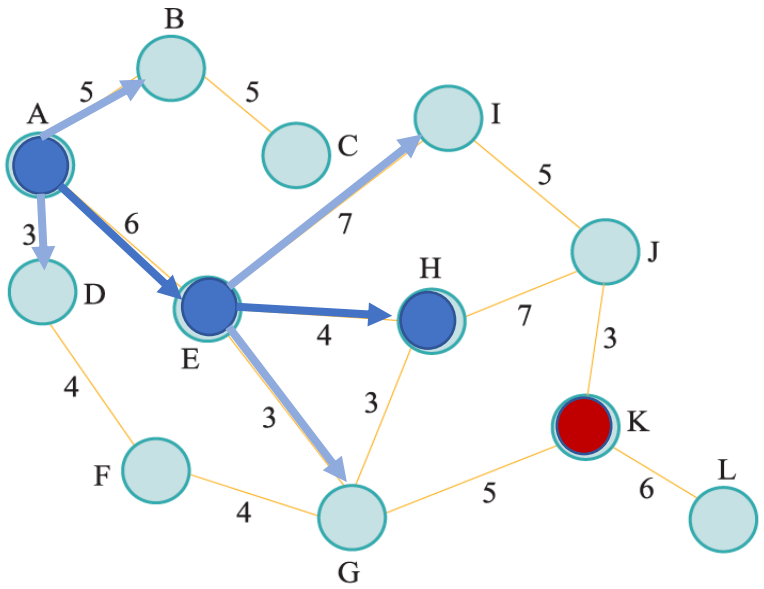
注意下表里的值是直线距离，这是一种额外信息，与左图中的代价无关

表 3.2　每个城市到目标城市的直线距离（启发函数取值）

状态	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
h(*n*)	13	10	6	12	7	8	5	3	6	3	0	6

每个城市到目标城市（即城市K）的直线距离（启发函数取值）

贪婪最佳优先搜索——搜索过程



求取从城市A到城市K之间一条行驶时间最短路线

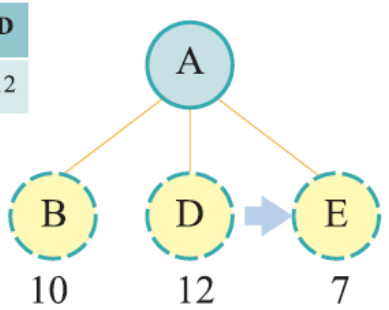
表 3.2　每个城市到目标城市的直线距离（启发函数取值）

状态	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
$h(n)$	13	10	6	12	7	8	5	3	6	3	0	6

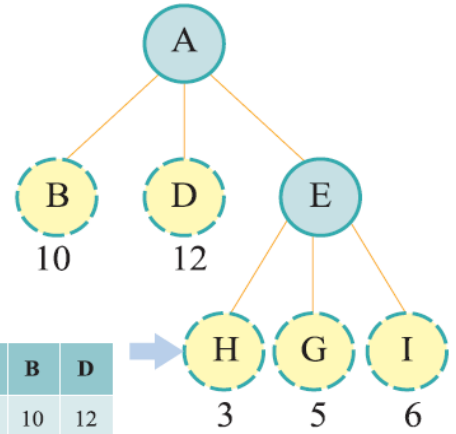
每个城市到目标城市（即城市K）的直线距离（启发函数取值）

扩展节点A

边缘结点	E	B	D
评价值	7	10	12

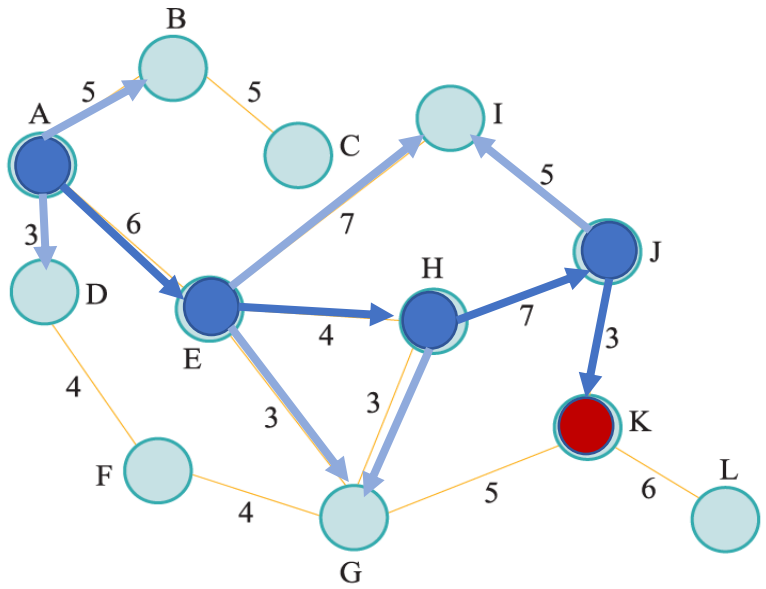


扩展节点E



边缘结点	H	G	I	B	D
评价值	3	5	6	10	12

贪婪最佳优先搜索——搜索过程



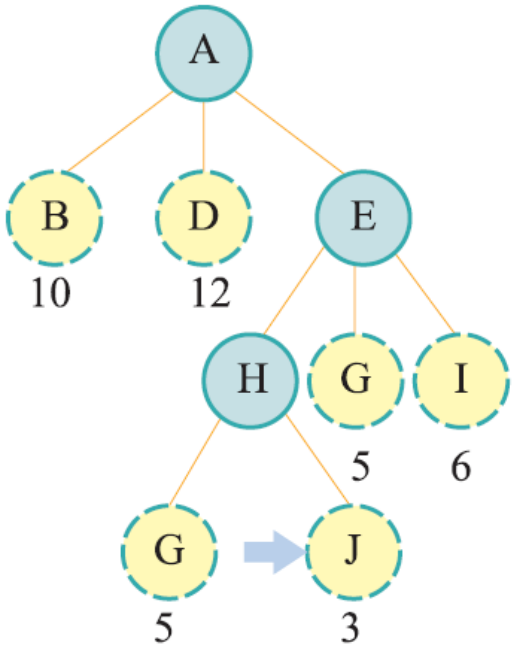
求取从城市A到城市K之间一条行驶时间最短路线

表 3.2　每个城市到目标城市的直线距离（启发函数取值）

状态	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
h(*n*)	13	10	6	12	7	8	5	3	6	3	0	6

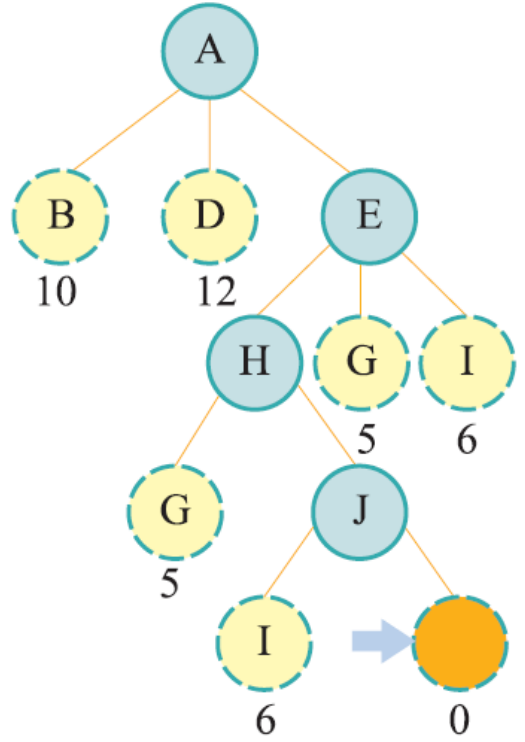
每个城市到目标城市（即城市K）的直线距离（启发函数取值）

扩展节点H



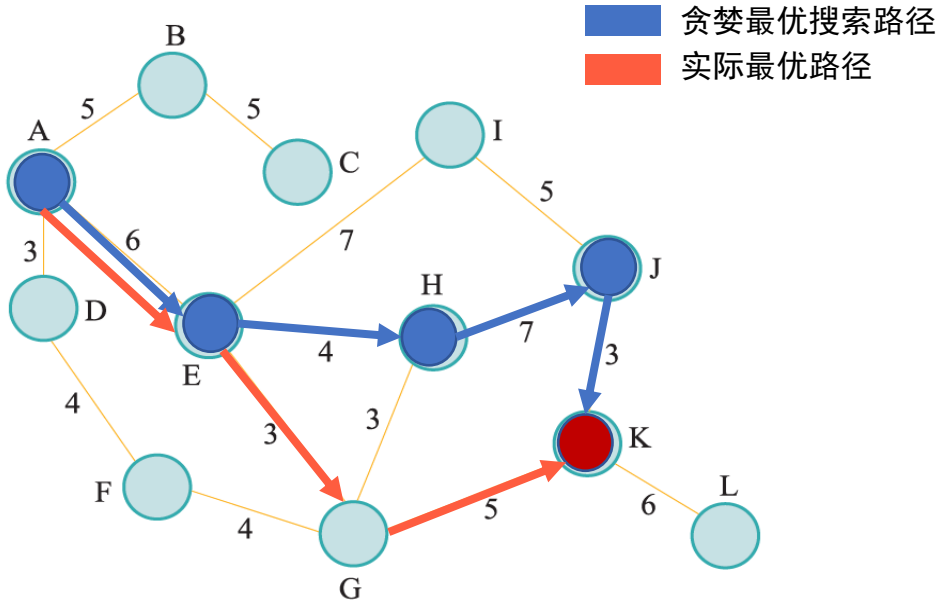
边缘结点	J	G	G	I	B	D
评价值	3	5	5	6	10	12

扩展节点J



边缘结点	K	G	G	I	I	B	D
评价值	0	5	5	6	6	10	12

贪婪最佳优先搜索：欲速则不达



求取从城市A到城市K之间一条行驶时间最短路线

贪婪算法找到的路径 $A \rightarrow E \rightarrow H \rightarrow J \rightarrow K$ **（长度为20）**

实际上最短路径为 $A \rightarrow E \rightarrow G \rightarrow K$ **（长度为14）**

贪婪最佳优先搜索：为什么会失败 欲速则不达

- 贪婪最佳优先搜索算法采用了排除环路的剪枝方法（已选择节点不在后续可选节点中），因此它是完备的，即总是可以找到一个解。但是该算法并不是最优的。
- 贪婪最佳优先搜索算法的问题在于其总是贪婪地扩展与目标结点之间代价最小（直线距离最近）的结点，而不考虑从初始结点到达该结点所需的代价。
- 在上述搜索步骤中，虽然第四步扩展的J结点距离目标结点的估计代价很小（取值为3），但从起始结点到达该结点的路径A → E → H → J代价为17，已经超过了实际上最短路径的代价（值为14），因此路径A → E → H → J并不是一个好的探索方向。

A*搜索

两个重要函数：评价函数与启发函数

辅助信息	所求解问题之外、与所求解问题相关的特定信息或知识。	
评价函数（evaluation function）f(n)	从当前节点n出发，根据评价函数来选择后续结点。	下一个结点是谁?
启发函数（heuristic function）h(n)	计算从结点n到目标结点之间所形成路径的最小代价值，这里将两点之间的直线距离作为启发函数。	完成任务还需要多少代价?

A*搜索(Greedy best-first search)：评价函数f(n) = 过去代价g(n) + 启发函数h(n)

（当前最小代价）（后续估计最小代价）

在评价函数中考虑从起始结点到当前结点的路径代价，评价函数和启发函数各司其职

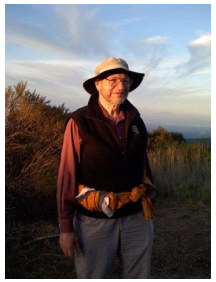
A： 第一个字母，代表先进性
Algorithm缩写，代表算法的一种

*：star，代表启发式信息加权下的总能找到最优解

由Nils John Nilsson、Bertram Raphael和Peter E. Hart与1968年提出

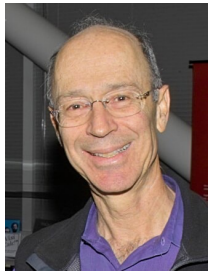
Nils John Nilsson

提出启发式信息，贪婪搜索



Bertram Raphael

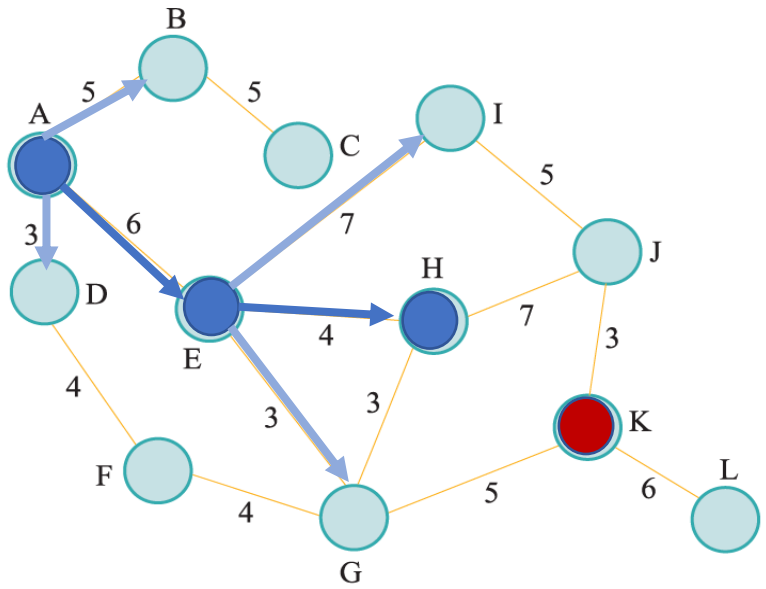
将过去代价g(n)引入评价函数



Peter E. Hart

提出启发式函数的可容性和一致性，以证明A*算法最优性

A*搜索——搜索过程



求取从城市A到城市K之间一条行驶时间最短路线

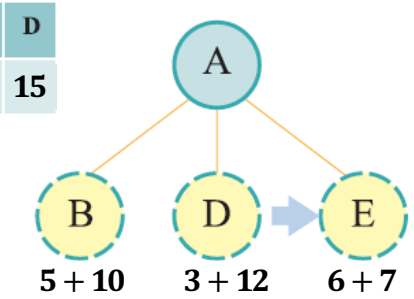
表 3.2　每个城市到目标城市的直线距离（启发函数取值）

状态	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
h(*n*)	13	10	6	12	7	8	5	3	6	3	0	6

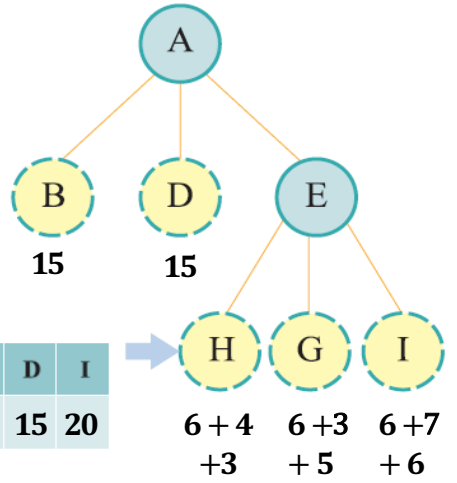
每个城市到目标城市（即城市K）的直线距离（启发函数取值）

扩展节点A

边缘结点	E	B	D
评价值	13	15	15

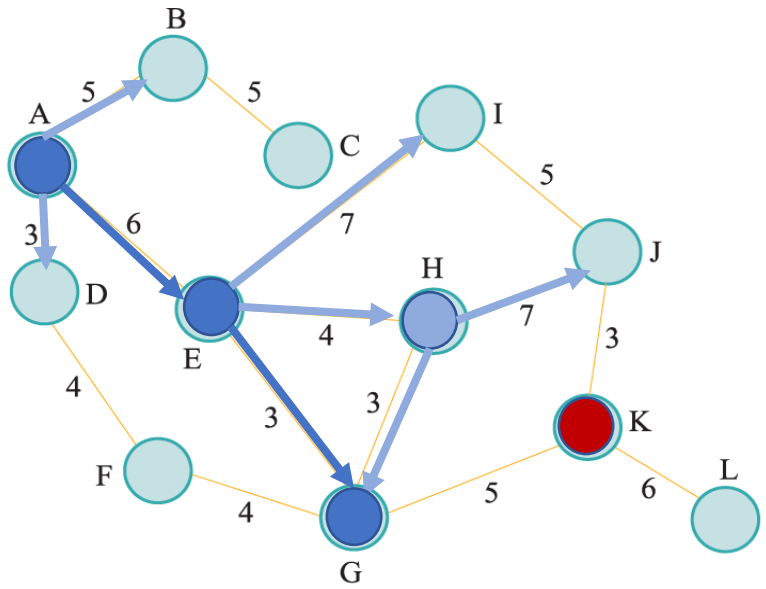


扩展节点E



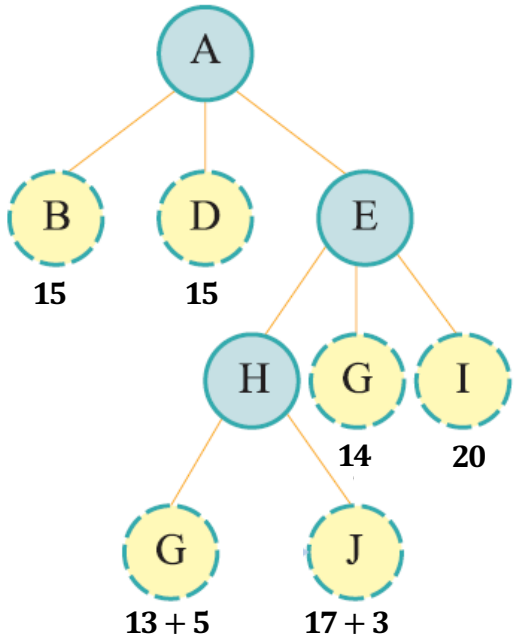
边缘结点	H	G	B	D	I
评价值	13	14	15	15	20

A*搜索——搜索过程



求取从城市A到城市K之间一条行驶时间最短路线

扩展节点H



因为此时A → E → G的评价函数值最小，因此最小评价函数路径由：

A → E → H

变为：

A → E → G

故启发式算法有时需要**回退**，中间结果不代表最后结果。

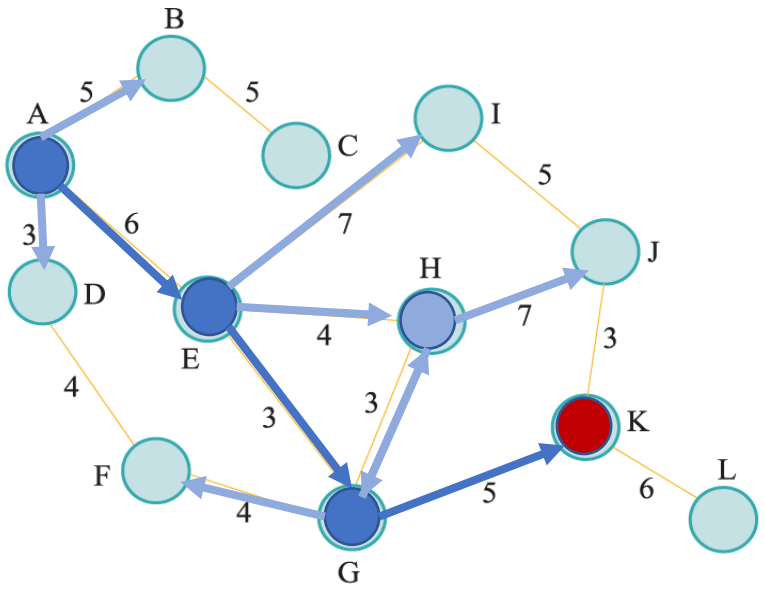
表 3.2　每个城市到目标城市的直线距离（启发函数取值）

状态	A	B	C	D	E	F	G	H	I	J	K	L
$h(n)$	13	10	6	12	7	8	5	3	6	3	0	6

每个城市到目标城市（即城市K）的直线距离（启发函数取值）

边缘结点	G	B	D	G	I	J
评价值	14	15	15	18	20	20

A*搜索——搜索过程



求取从城市A到城市K之间一条行驶时间最短路线

扩展节点 **G**

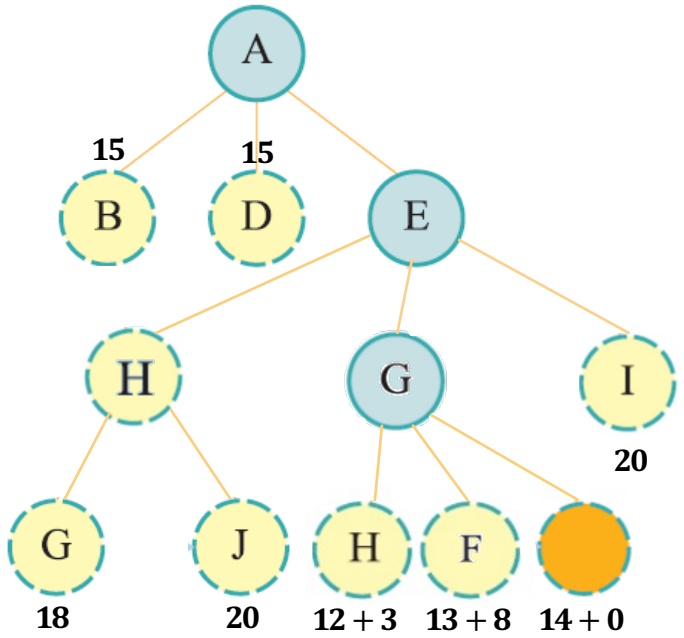


表 3.2　每个城市到目标城市的直线距离（启发函数取值）

状态	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
$h(n)$	13	10	6	12	7	8	5	3	6	3	0	6

每个城市到目标城市（即城市K）的直线距离（启发函数取值）

边缘结点	K	B	D	H	G	I	J	F
评价值	14	15	15	15	18	20	20	21

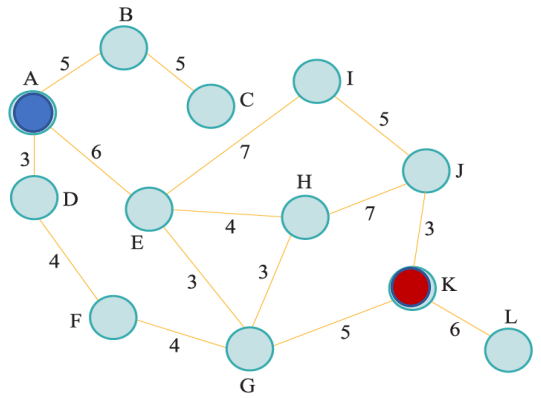
因此A*算法选择的最短路径为A → E → G → K，与实际最优路径一致

A*搜索性能分析

A*算法的完备性和最优性取决于搜索问题和启发函数的性质。

启发函数的性质：

- **可容性**
- **一致性**



求取从城市A到城市K之间一条行驶时间最短路线

接下来使用的一些符号解释：

$h(n)$：结点n的启发函数取值

$g(n)$：从起始结点到结点n所对应路径的代价

$f(n)$：结点n的评价函数取值

$h^*(n)$：从结点n出发到达终止结点的最小代价

$c(n,a,n')$：从结点n执行动作a到达结点n'的单步代价

在之前的问题中这些符号可理解为：

$h(n)$：城市n到目标城市K的直线距离

$g(n)$：从起始城市A到城市n所经过的路径长度

$f(n)$：城市 n 的评价函数取值（之前路径长度+到K直线距离）

$h^*(n)$：从城市n出发到达目标城市K的最短路径

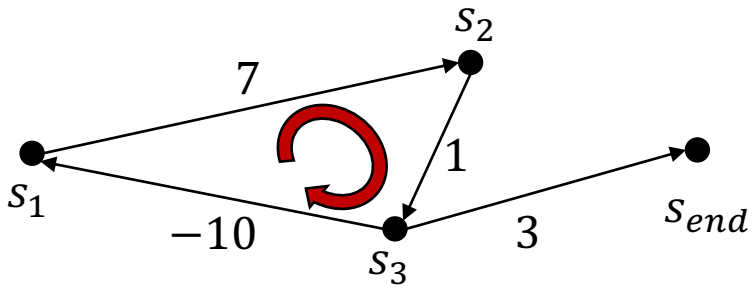
$c(n,a,n')$：从城市n执行动作a到达城市n'的路径长度

A*搜索性能分析：完备性

如何确保A*能够保证在存在解的情况下总能找到一个解？ → **完备性**

如果所求解问题和启发函数满足以下条件，则A*算法是完备的：

- 搜索树中分支数量是有限的，即每个结点的后继结点数量是有限的。
  - 确保搜索空间有限，算法不会陷入无限搜索。
- 单步代价的下界是一个正数。
  - 如果为负数，A*可能陷入无限循环。

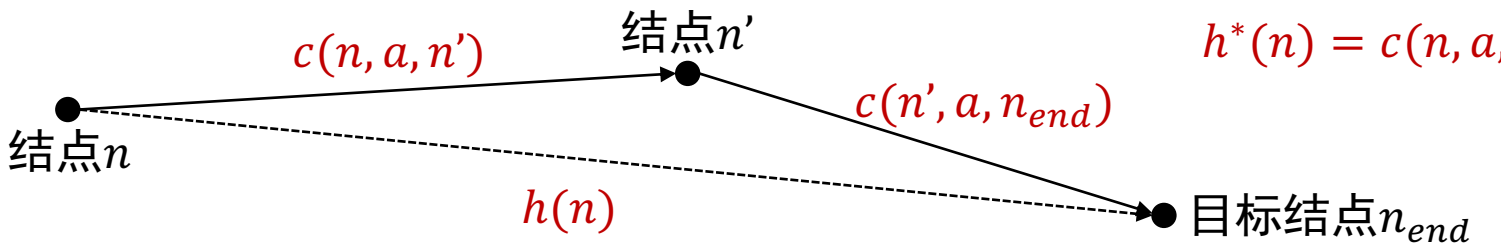


- 启发函数有下界。
  - 如果启发函数没有下界，要么其不满足可容性，要么该系统中不存在可行解（即代价→∞）。

A*搜索性能分析：可容性

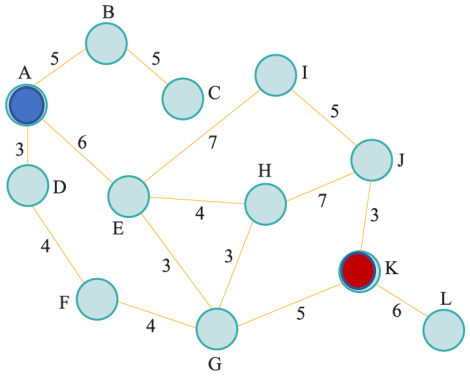
A*算法的最优性取决于搜索问题和启发函数的性质。

启发函数的性质——**可容性**：



可容性指满足条件：**对于任意结点n，有$h(n) \le h^*(n)$，如果n是目标结点，则有$h(n)=0$。**

启发函数不会过高估计从结点n到终止结点所应该付出的代价（即**估计代价小于等于实际代价**）。



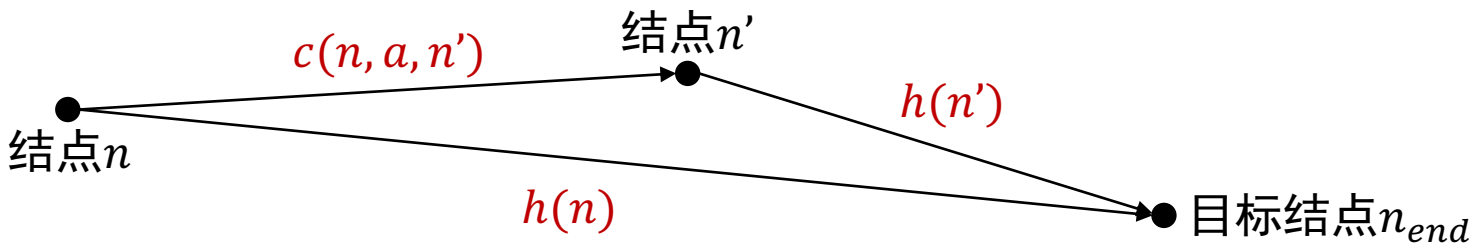
求取从城市A到城市K之间一条行驶时间最短路线

在寻找最短路径问题中，状态城市与目标城市之间的距离不可能小于二者的直线距离，因此到目标城市的直线距离这个启发函数是具备可容性的。

A*搜索性能分析：一致性

A*算法的最优性取决于搜索问题和启发函数的性质。

启发函数的性质——一致性：



启发函数的一致性指满足条件 $\boldsymbol{h(n) \leq c(n,a,n') + h(n')}$。

如果将单步代价$c(n,a,n')$重新定义为n和n'在二维平面上的欧氏距离，则根据欧氏距离三角不等式原则，城市n到目标城市n_{end}之间直线距离一定小于等于从城市n到其相邻城市n'的直线距离与城市n'到目标城市n_{end}之间直线距离之和。故此时，若将当前城市与目标城市n_{end}之间直线距离作为启发函数值，则启发函数满足一致性条件。

A*搜索性能分析：可容性与一致性

一致性必然导致可容性。

一致性是一个比可容性更严格的性质，即一个启发函数如果满足一致性条件，那么该启发函数必然满足可容性条件。

证明：

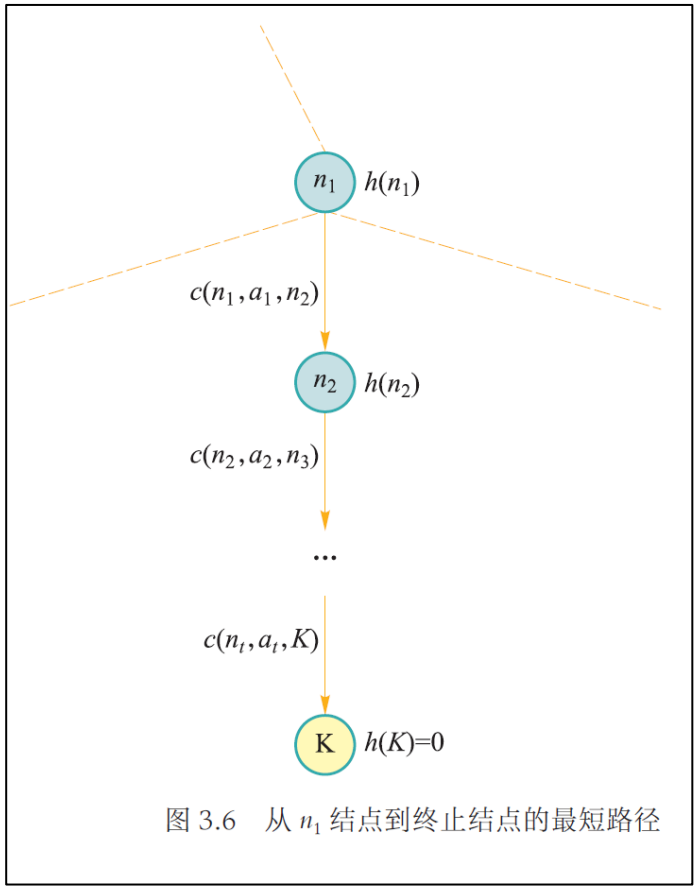


图 3.6　从 n_1 结点到终止结点的最短路径

如图中以n_1为根结点的子树，假设从该结点到达终止结点代价最小的路径为$n_1 \to n_2 \to \cdots \to \mathrm{K}$。由于$h(\mathrm{K}) = 0$且启发函数满足一致性条件，因此存在：

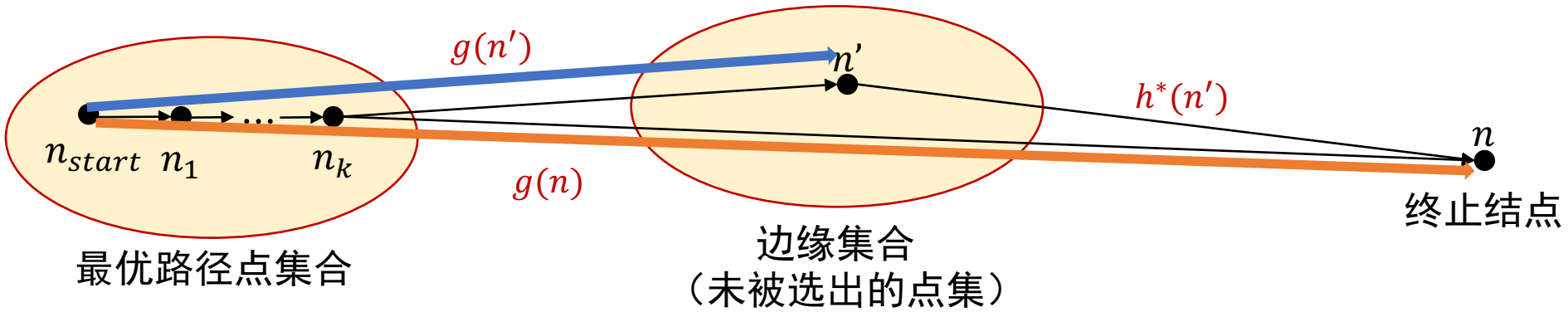
$$
\begin{aligned}
h(n_1) &\le h(n_2) + c(n_1, a_1, n_2) \\
&\le h(n_3) + c(n_2, a_2, n_3) + c(n_1, a_1, n_2) \\
&\le \cdots \\
&\le c(n_1, a_1, n_2) + c(n_2, a_2, n_3) + \cdots + c(n_t, a_t, \mathrm{K}) \\
&= h^*(n_1)
\end{aligned}
$$

即满足一致性条件的启发函数一定满足可容性条件。

A*搜索性能分析：最优性

如何确保A*能够保证在存在解的情况下总能找到最优的那一个解？ → **最优性（找到的就是是最好的）**

如果启发函数是可容的，那么A*算法满足最优性

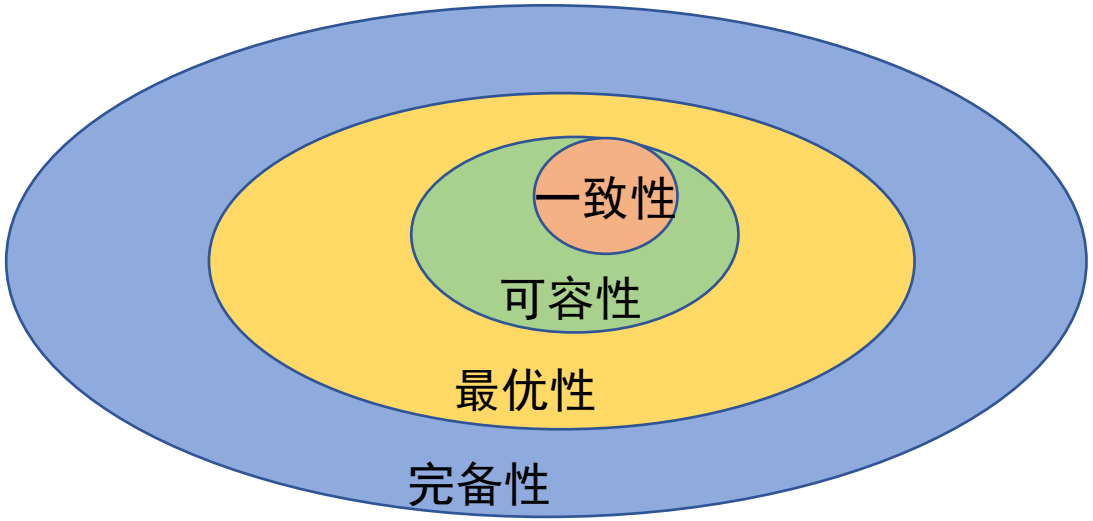


假设A*算法找到的第一个终止结点为n。对于此时边缘集合中任意结点n'，根据算法每次扩展时的策略，即选择评价函数取值最小的边缘结点进行扩展，有$f(n) \le f(n')$。由于A*算法对评价函数定义为$f(n) = g(n) + h(n)$，且$h(n) = 0$，有$f(n) = g(n) + h(n) = g(n)$，$f(n') = g(n') + h(n')$，综合可得

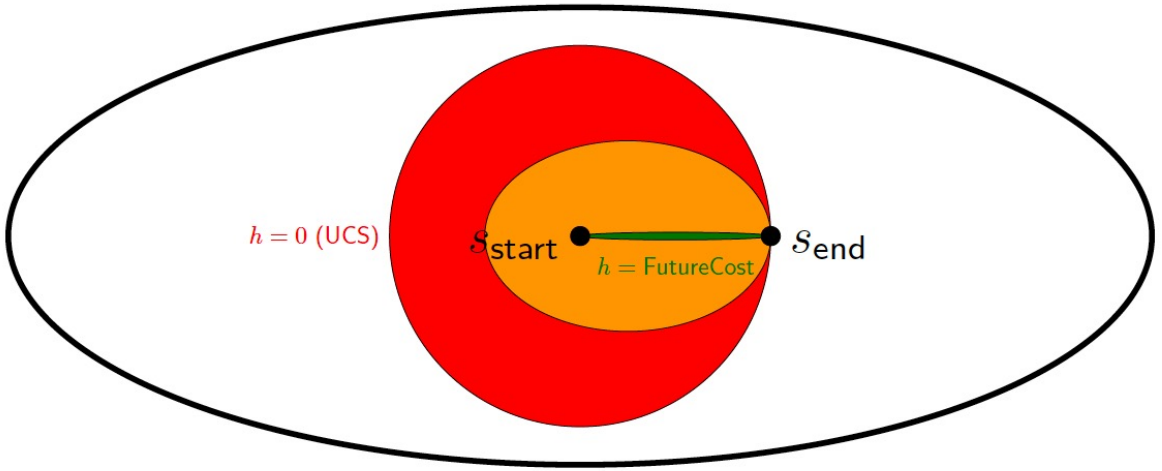
$$g(n) \le g(n') + h(n') \le g(n') + h^*(n')$$

其中$g(n)$为从起始结点到结点n所对应路径的代价，$g(n') + h^*(n')$表示从初始结点出发经过结点n'到达终止结点的最小代价。也就是说此时扩展其他任何一个边缘结点都不可能找到比结点n所对应路径代价更小的路径，因此结点n对应的是一条最短路径，即算法满足最优性。

A*搜索性能分析



各性质之间的关系



搜索空间随启发信息强度的变化趋势

- 完备性是前提，即搜索问题首先一定要有解。
- 一致性一定导致可容性，二者都可以推导出最优性。

- 当启发式信息不起作用时，即h(s)=0时，A*搜索等同于代价一致搜索（UCS）。（红色区域）
- 当启发式信息等同于未来代价时，即h(s)=FutureCost(s)时，A*搜索会直接沿着最小代价路径前进。（绿色区域）
- 通常启发式信息介于上述两种情况之间。（橙色区域）

A*搜索可视化

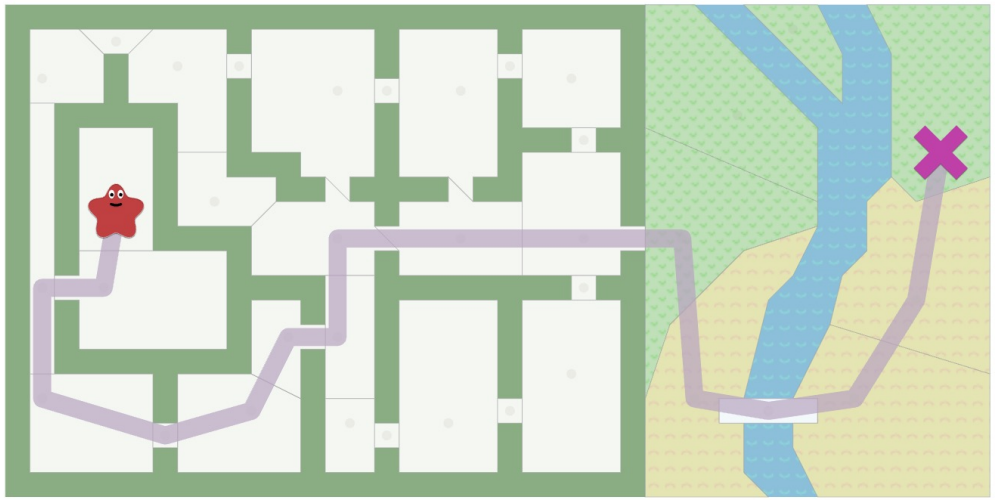
Introduction to the A* Algorithm

from Red Blob Games

Home Blog Links Twitter About Search

Created 26 May 2014, updated Aug 2014, Feb 2016, Jun 2016, Jun 2020

In games we often want to find paths from one location to another. We're not only trying to find the shortest distance; we also want to take into account travel time. Move the blob (start point) and cross (end point) to see the shortest path.



https://www.redblobgames.com/pathfinding/a-star/introduction.html

目录

Minimax搜索

对抗搜索（adversarial search）或博弈搜索（game search）：

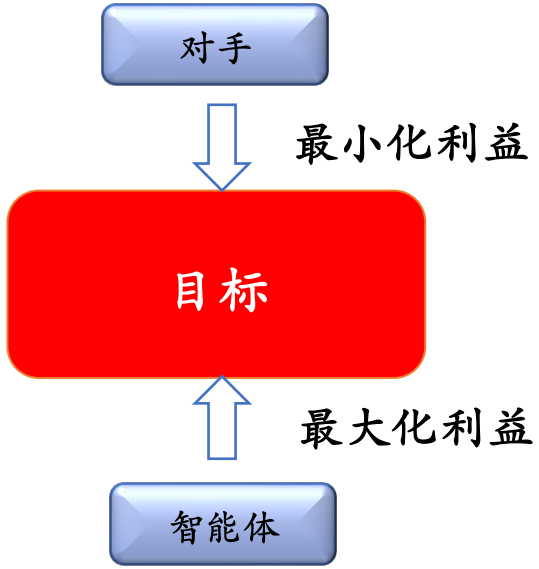
两个智能体同处于一个竞争的环境，智能体之间通过竞争来实现各自相反目标，即一方想**最大化自身的利益**，而另一方则想**最小化对手的利益**。通俗地说，对抗搜索的过程就是两个智能体各自选择对自己有利的策略。

狭路相逢勇者胜

勇者相逢智者胜

智者相逢德者胜

德者相逢道者胜



Minimax搜索——若干定义

- **状态**：状态s包括当前的游戏局面和当前行动的智能体。初始状态s_0包括初始游戏局面和初始行动的玩家。由于本节讨论的问题假设两个竞争对手轮流行动，因此第i步行动的玩家是确定的，函数player(s)给出状态s下行动的智能体。

- **动作**：给定状态s，动作指的是player(s)在当前局面下可以采取的操作a，记动作集合为actions(s)。

- **状态转移**：给定状态s和动作$a \in \text{actions}(s)$，状态转移函数result(s,a)决定了在s状态采取a动作后所得后继状态。

- **终局状态检测**：终止状态检测函数terminal_test(s)用于测试游戏是否在状态s结束。

- **终局得分**：终局得分utility(s,p)表示在终局状态s时玩家p的得分。在二人零和博弈中，两名玩家的终局得分之和应该是固定的，因此算法只需记录其中一人的终局得分为utility(s)，则另一人的得分可按照零和原则相应算出。

Minimax搜索——搜索策略

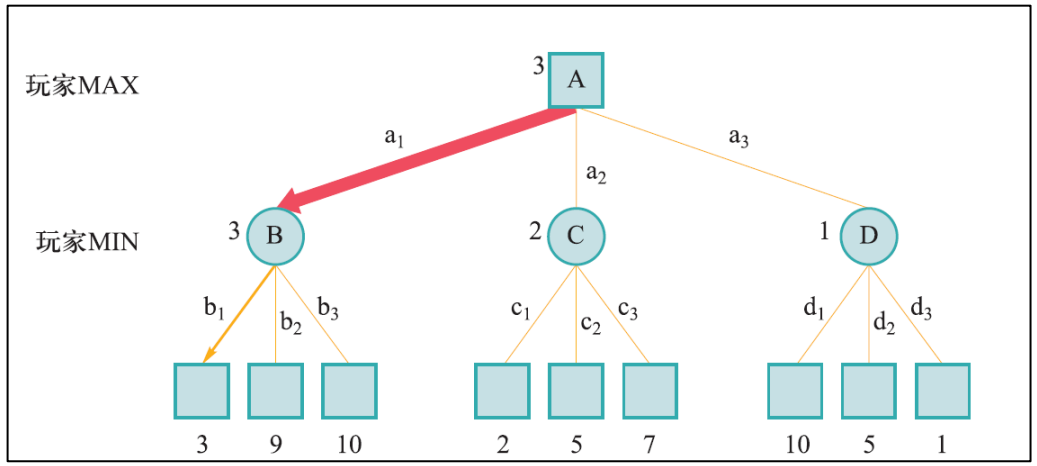
由于每个玩家都想要最大化自己的得分，因此：

玩家1行动时：选择**最大化**终局得分的搜索树分支（MAX）

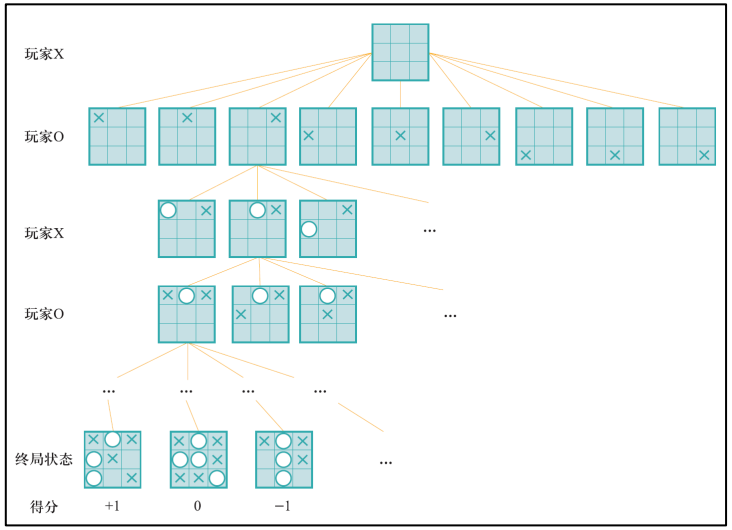
玩家0行动时：选择**最小化**终局得分的搜索树分支（MIN）

这也就是最小最大算法的名字由来。

$$\text{minimax}(s) = \begin{cases} \text{utility}(s), & \text{if terminal\_test}(s) \\ \max_{a \in \text{actions}(s)} \text{minimax}\big(\text{result}(s,a)\big), & \text{if player}(s) = MAX \\ \min_{a \in \text{actions}(s)} \text{minimax}\big(\text{result}(s,a)\big), & \text{if player}(s) = MIN \end{cases}$$



玩家MAX和玩家MIN在最优策略下采取动作的示意图



井字棋所对应的对抗搜索树

Minimax搜索——算法流程

算法 3.2 通过最小最大搜索求解最优动作的算法

函数： MinimaxDecision
输入： 当前的盘面状态 s
输出： 玩家 MAX 行动下，当前最优动作 a^*

```
a* ← argmax_{a∈actions(s)} MinValue(result(s, a))
```

函数： MaxValue
输入： 当前的盘面状态 s
输出： 玩家 MAX 行动下，当前状态的得分 $v = \text{minimax}(s, \text{MAX})$

```
if terminal_test(s) then return utility(s)
v ← −∞
foreach a ∈actions(s) do
    v ← max(v, MinValue(result(s, a)))
end
```

函数： MinValue
输入： 当前的盘面状态 s
输出： 玩家 MIN 行动下，当前状态的得分 $v = \text{minimax}(s, \text{MIN})$

```
if terminal_test(s) then return utility(s)
v ← +∞
foreach a ∈actions(s) do
    v ← min(v, MaxValue(result(s, a)))
end
```

Minimax搜索就是在计算搜索树中每个结点分数之后，每一步由玩家根据自己的角色来选择使得分数最大或分数最小的行动。

- 算法3.2给出了Minimax算法的流程，该算法假设玩家MAX为初始行动玩家。这个假设并不会使问题失去一般性，如果实际上初始行动的是玩家MIN，则可以通过将终局得分取相反数的方法来交换玩家MIN和玩家MAX。函数MinValue和MaxValue递归求解了每个搜索树结点的分数，函数MinimaxDecision根据每个结点的分数求解玩家MAX在当前盘面下所采取的最优动作。
- Minimax算法对游戏搜索树执行了一个完整的**深度优先搜索**。因此，如果搜索树是有限的，那么Minimax算法必然能在有限时间内终止；如果对手总是理性地按照最优策略来交替采取行动，那么Minimax算法得到的解是最优的。
- 如果m是游戏树的最大深度，在每个结点最多存在b种有效走法，那么Minimax算法的时间复杂度为$O(b^m)$，空间复杂度为$O(bm)$。

拓展阅读

突破组合爆炸之难与陷入地平线问题之困

香农之数：在象棋比赛中，深蓝需要判断某一时刻棋局落子对整个棋局胜负会带来怎样的影响。由于可选棋局众多，尽管深蓝平均每秒能够对1亿个棋局进行判断评估，还是无法在规定时间内计算得到当前棋局对胜负的潜在影响。1950年，香农（Claud Shannon）发表了一篇有关国际象棋编程的论文。在这篇论文中，香农估算国际象棋比赛中落子选择从第一次移动时的20种会增加到第二次移动时的400种，在第六次移动时可能的落子选择达到1.19亿种。香农甚至估算认为国际象棋的落子总数为10的120次方种，远远超出了10的82次方这一宇宙原子总数，这就是国际象棋中“组合爆炸”难题。

剪枝--眼不见心不烦：为了从海量可能落子中选择一种合适落子，以克服组合爆炸挑战，深蓝采用了由1971年图灵奖获得者约翰·麦卡锡 (John McCarthy)发明的“**阿尔法-贝塔（Alpha-Beta）”剪枝搜索算法**。

上帝之落子：2016年3月，AlphaGo在首尔以4:1比分战胜了李世石。李世石在第四局78步给出了惊人落子，扭转了乾坤。AlphaGo无法应对李世石这一步神来落子是搜索算法固有存在地平线问题（horizon problem）所致，即搜索算法无法看到远方地平线以下的景象。这可理解为“计算机暴力搜索计算”与“人类直觉顿悟”之间的差异，突破地平线之困依靠搜索之能还相距甚远。

alpha-beta剪枝

Alpha-Beta 剪枝搜索算法在Minimax算法中可减少被搜索的结点数，即在保证得到与原Minimax算法同样的搜索结果时，剪去了不影响最终结果的搜索分枝。

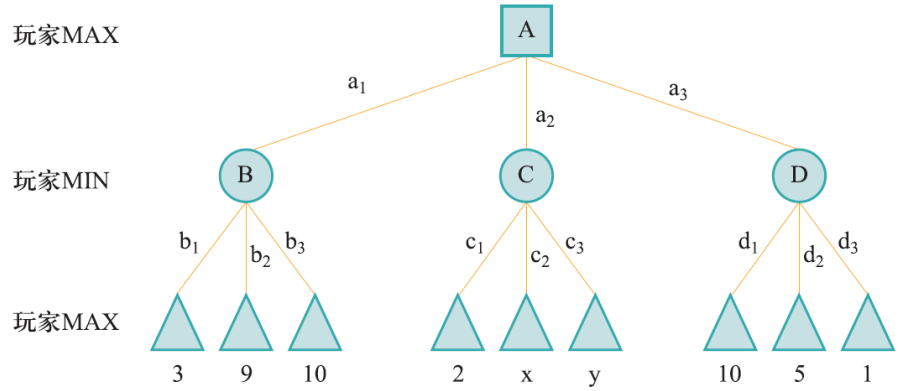


图 3.10　一棵对抗搜索树在搜索过程中某时刻的状态示意

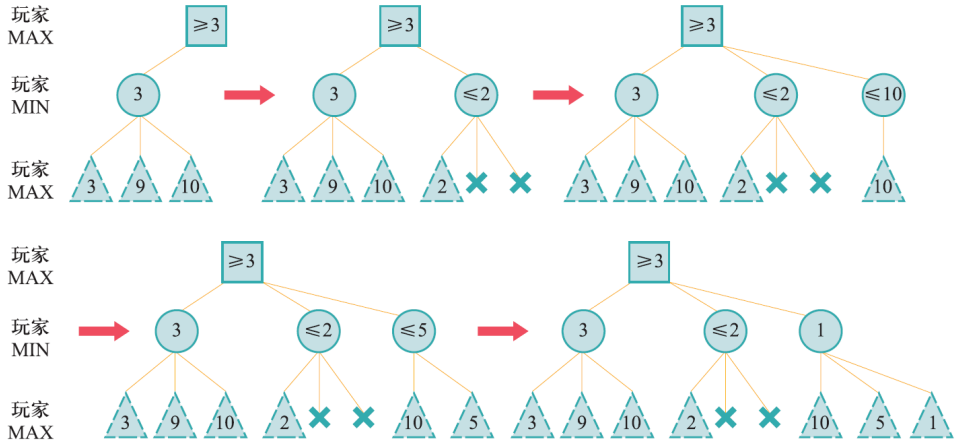


图 3.11　存在剪枝的搜索树部分扩展过程（对应图 3.10 的搜索树）

由于$min(2,x,y)$的值小于等于2，所以即使不知道x和y的值，根结点的分数也可以算出来等于3。也就是说，算法没有必要计算动作c_2和c_3对应的两个子树在游戏结束时所得分数，就能决定在根结点采取动作a_1从而在游戏结束时可获得收益3，因此动作c_2和c_3所对应子树可以被剪枝。

$$minimax(A) = max(min(3,9,10), min(2,x,y), min(10,5,1))$$
$$= max(3, min(2,x,y), 1)$$

alpha-beta剪枝

对min节点后代剪枝示意图

- 可把上述剪枝思路推广到一般情况。假设有一个位于MIN层的结点m，已知该结点能够向其上MAX结点反馈的收益为α（alpha）。n是与结点m位于同一层的某个兄弟（sibling）结点的后代结点。如果在结点n的后代结点被访问一部分后，知道结点n能够向其上一层MAX结点反馈收益小于α，则结点n的未被访问孩子结点将被剪枝。

- 如图3.12所示，玩家MAX在结点m'能够从位于其下MIN层的结点m得到取值至少为α的收益，因此结点n那些未被访问的后继分支结点对结点m'取值没有任何影响了。

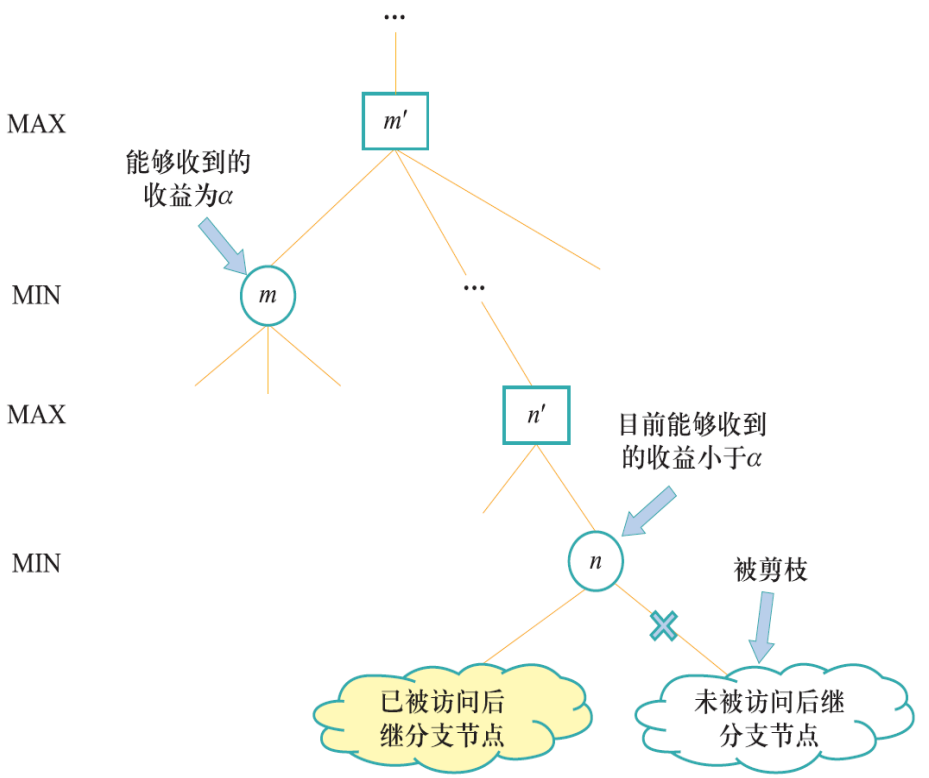


图 3.12　基于 MIN 结点的反馈收益进行剪枝（alpha 剪枝）

alpha-beta剪枝

对max节点后代剪枝示意图

- 如图3.13所示，考虑位于MAX层的结点m，已知结点m能够从其下MIN层结点收到的收益为β（beta）。

- 结点n是结点m上层结点m'的位于MAX层的后代结点，如果目前已知结点n能够收到的收益大于β，则不再扩展结点n的未被访问后继结点，因为位于MIN层的结点m'只会选择收益小于或等于β的结点来采取行动。

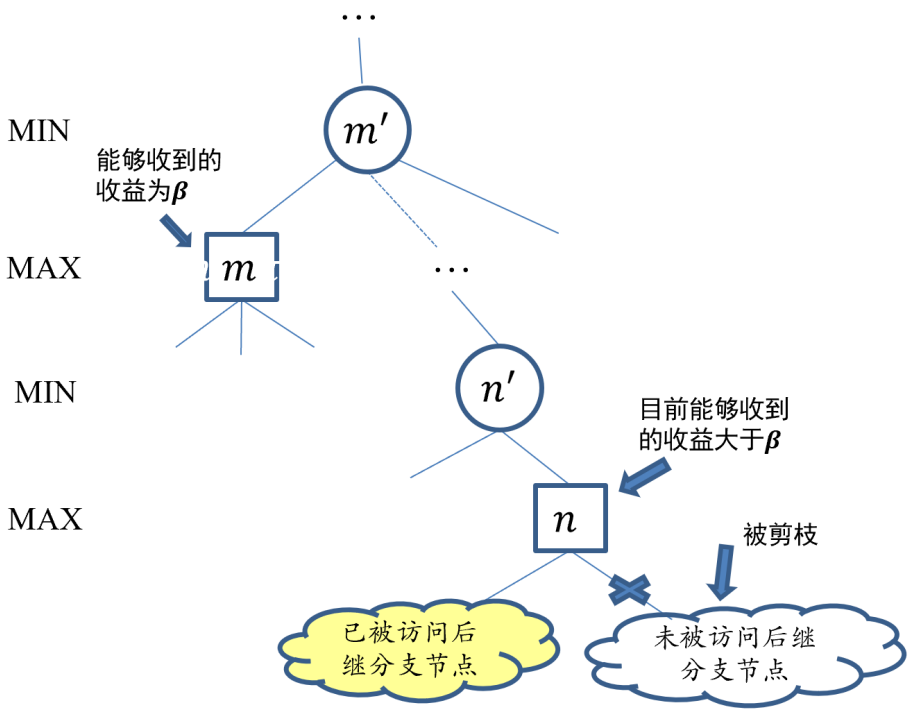
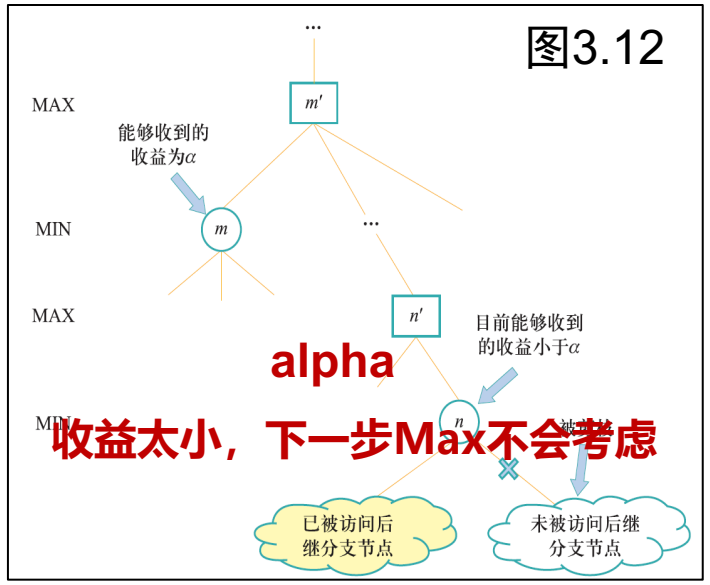
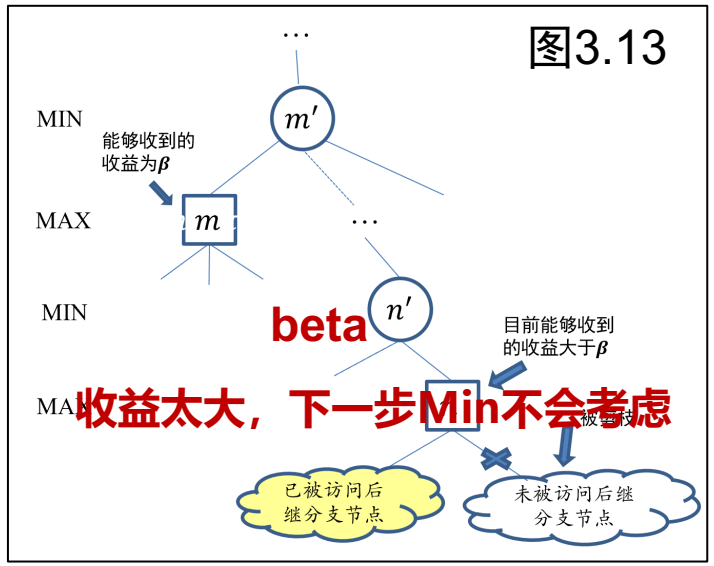


图 3.13 基于MAX结点反馈收益进行剪枝(beta剪枝)

alpha-beta剪枝



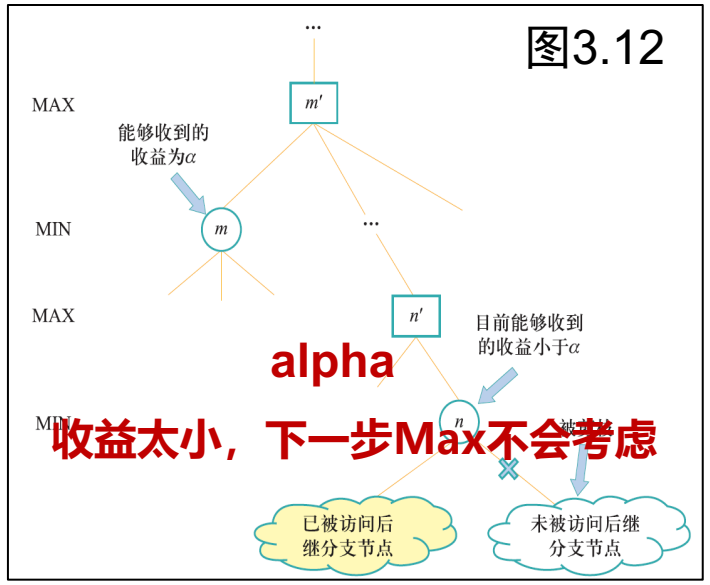


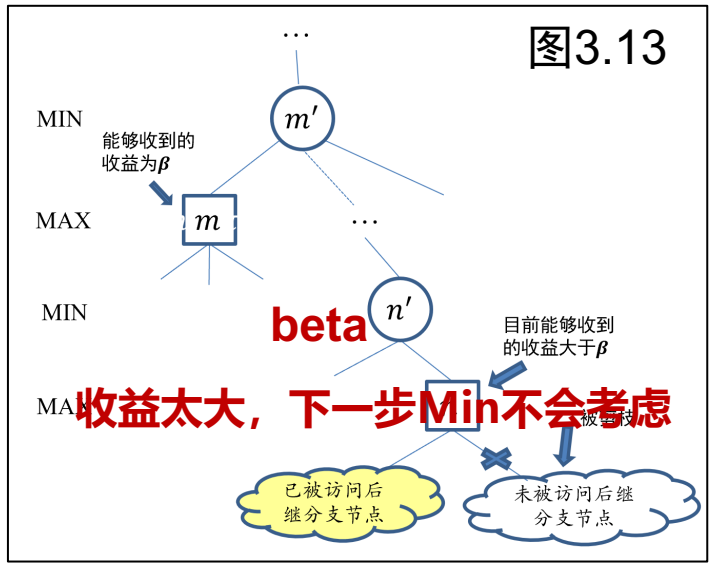
alpha-beta剪枝算法：为什么叫alpha和beta剪枝?

- 实际上，在图3.12中，结点m不仅为MIN层结点n提供了一个下界，对于例如n'的MAX层结点，这个下界仍然是有意义的。图3.13中情况同理。因此，无论一个结点位于MIN层还是MAX层，都可以根据图3.12和图3.13中的情况找到一个下界（alpha）和一个上界（beta），即可以为每个结点设置一个α值和一个β值，来判断该结点及其后继结点是否可被剪枝。

- 每搜索一个子节点都会按照一定的规则对节点范围进行更新，Max节点可以修改alpha值，Min节点修改beta值，如果出现了alpha > beta，则将该分支剪枝。因此综合以上两种情况的剪枝方法被称为alpha-beta剪枝。

alpha-beta剪枝





alpha-beta剪枝算法：更新alpha和beta的剪枝规则

- 在设计算法时，为了计算一个结点的上下界，无需枚举所有会影响当前结点α值和β值的结点，只需要继承父结点的α值和β值，再按照一定的规则加以更新即可：

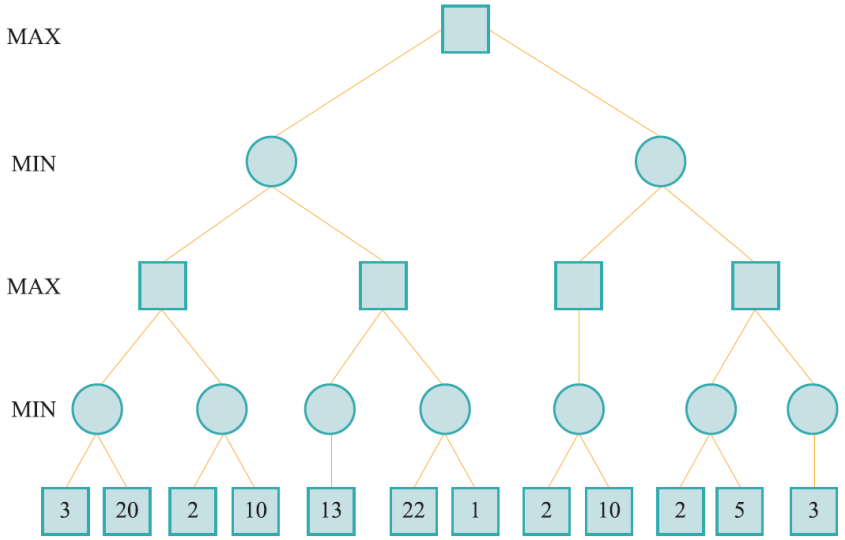
（1）对于**MAX结点**，如果其孩子结点（MIN结点）的收益大于当前的α值，则将α值更新为该收益——**Max需要提高收益**

（2）对于**MIN结点**，如果其孩子结点（MAX结点）的收益小于当前的β值，则将β值更新为该收益——**Min需要降低收益**

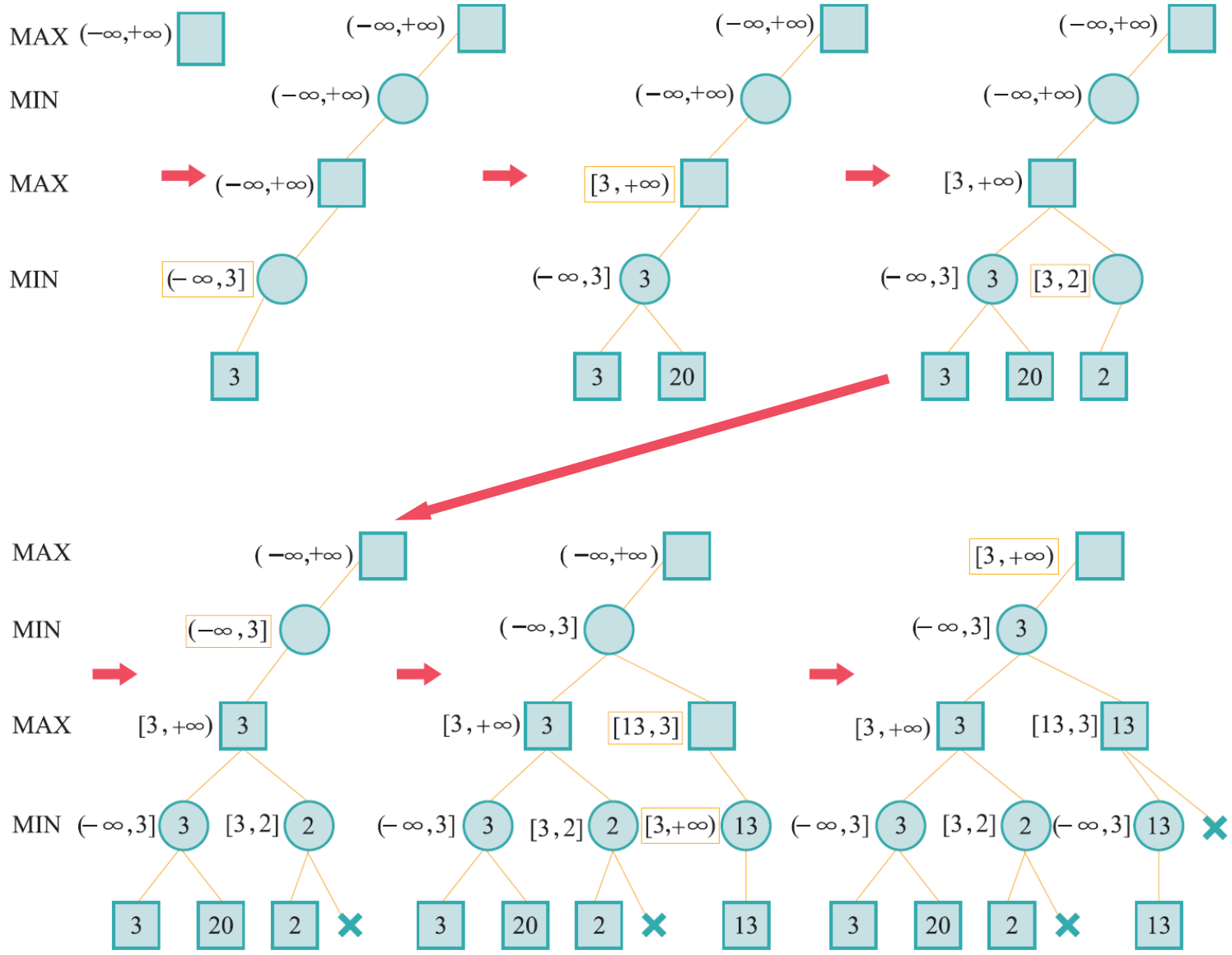
（3）根结点（MAX结点）的α值和β值分别被初始化为−∞和+∞。

- 随着搜索算法不断被执行，每个结点的$[\alpha,\beta]$从其父结点提供的初始值开始，取值按照如下形式变化：α逐渐增加、β逐渐减少。不难验证，如果一个结点的α值和β值满足$\alpha > \beta$的条件，则该结点尚未被访问的后续结点就会被剪枝，因而不会被智能体访问。

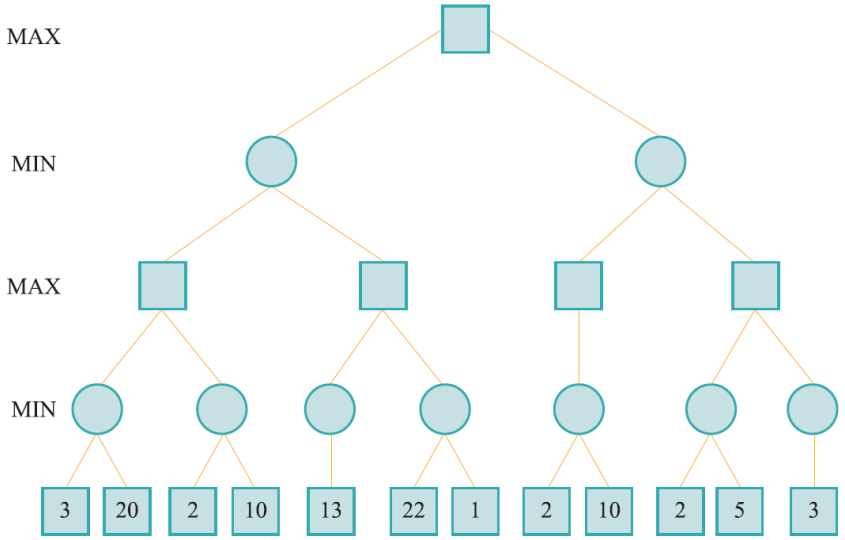
alpha-beta剪枝——实例



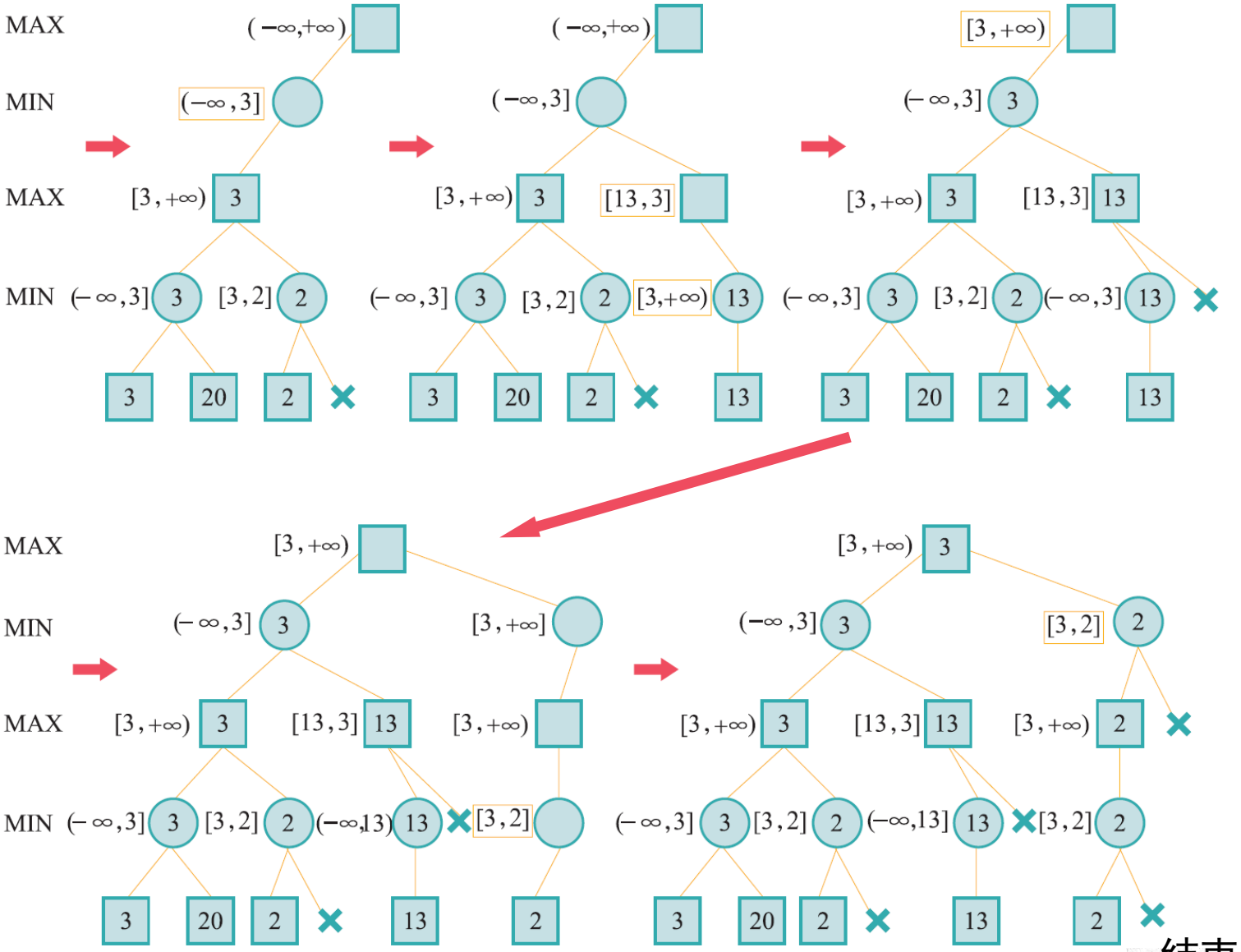
一颗完整的最大最小搜索树



alpha-beta剪枝——实例



一颗完整的最大最小搜索树



蒙特卡洛树搜索

多臂赌博机示意图

- 以经典的多臂赌博机（Multi-armed Bandit）问题为例。先考虑一个简化问题：假设智能体面前有K个赌博机，每个赌博机有一个臂膀。每次转动一个赌博机臂膀，赌博机则会随机吐出一些硬币或不吐出硬币，将每次吐出的硬币的币值表示为收益分数。现在假设给智能体$\tau(\tau > \mathrm{K})$次转动臂膀的机会，那么智能体如何选择赌博机、转动τ次赌博机臂膀，以获得更多的收益分数呢？

- 或许可以让智能体先把K个赌博机的臂膀依次转动一遍，观察在摇动每个赌博机臂膀时的收益分数，然后去转动那些收益分数高的赌博机臂膀。但是，由于从每个赌博机获得的收益分数是随机的，一个刚刚给用户带来可观收益分数的赌博机在下一次摇动其臂膀时可能不会继续获得可观收益分数，因此这一方法不可取。那么是否有方法能够指导智能体的高效地获取收益分数呢？

蒙特卡洛树搜索——要素

- **状态：** 每个被摇动的臂膀即为一个状态，记K个状态分别为$\{s_1, s_2, \cdots, s_K\}$，没有摇动任何臂膀的初始状态记为$s_0$。
- **动作：** 动作对应着摇动一个赌博机的臂膀，在多臂赌博机问题中，任意状态下的动作集合都为$\{a_1, a_2 \cdots, a_K\}$，分别对应摇动某个赌博机的臂膀。
- **状态转移：** 选择动作$a_i(1 \leq i \leq K)$后，将状态相应地转换为s_i。然而这个问题和前两节讨论的搜索问题的不同之处在于，由于存在随机性，智能体摇动赌博机臂膀若干个τ次，任意两个τ次的结果可能都不一样，为此引入如下奖励要素。
- **奖励（reward）：** 假设从第i个赌博机获得收益分数的分布为D_i，其均值为μ_i。如果智能体在第t次行动中选择转 动了第l_t个赌博机臂膀，那么智能体在第t次行动中所得收益分数$\hat{r}_t$服从分布D_{l_t}，$\hat{r}_t$被称为第t次行动的奖励。为了方便对多臂赌博机问题的理论研究，一般假定奖励是有界的，进一步可假设奖励的取值范围为$[0,1]$。

蒙特卡洛树搜索——要素

- **悔值（regret）函数**

根据智能体前T次动作，可以如下定义悔值函数：

$$\rho_T = \underbrace{T\mu^*}_{\text{理想}} - \left.\sum_{t=1}^{T}\hat{r}_t\right\} \text{现实}$$

其中，$\mu^* = \max_{i=1,\cdots,K} \mu_i$。显然为了尽量减少悔恨，在每次操作时，智能体应该总是转动能够提供最大期望奖励的赌博机臂膀，但是这是不现实的，因为智能体并不知道哪个臂膀的奖励期望最大。这一公式告诉我们，将T次操作中最优策略的期望得分减去智能体的实际得分，就是悔值函数的结果。显然，问题求解的目标为最小化悔值函数的期望，该悔值函数的取值取决于智能体所采取的策略。

蒙特卡洛树搜索——搜索策略

智能体无法预先计算好最优策略，然后实行，而是需要一边探索一边调整自己的策略，以最大化收益

贪心算法：**智能体记录下每次摇动的赌博机臂膀和获得的相应收益分数。**给定第$i(1 \le i \le K)$个赌博机，记在过去$t-1$次摇动赌博机臂膀的行动中，一共摇动第i个赌博机臂膀的次数为第$T_{(i,t-1)}$。于是，可以计算得到第i个赌博机在过去$T_{(i,t-1)}$次被摇动过程中的收益分数平均值$\bar{x}_{i,T_{(i,t-1)}}$。这样，**智能体在第t步，只要选择$\bar{x}_{i,T_{(i,t-1)}}$值最大的进行摇动。**

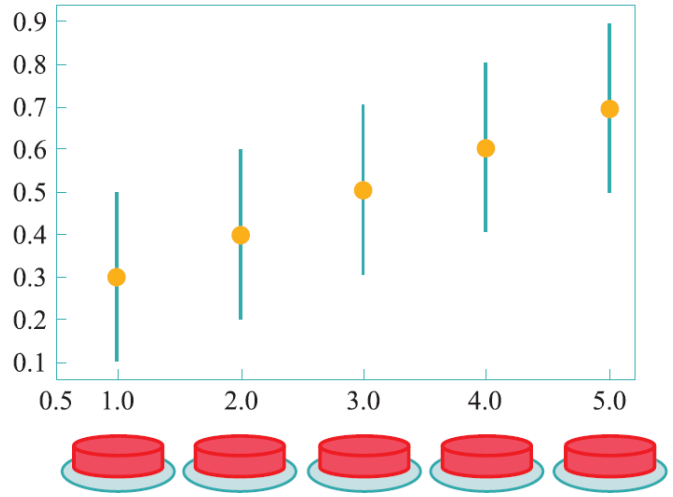


图 3.16　多臂赌博机问题中每个赌博机收益分数的分布情况

贪心算法存在一定问题，受收益估计影响很大：假设一共有五台赌博机（即$K=5$），第i个赌博机的得分满足均值为$(\mu_1,\cdots,\mu_5)=(0.3,0.4,0.5,0.6,0.7)$的分布。假如在计算机上模拟$T=10000$次摇动五台赌博机，假设在模拟中4号赌博机被摇动次数比较多，其他赌博机被摇动次数很少，这会导致**这些赌博机平均收益分数的估计结果产生很大误差。**

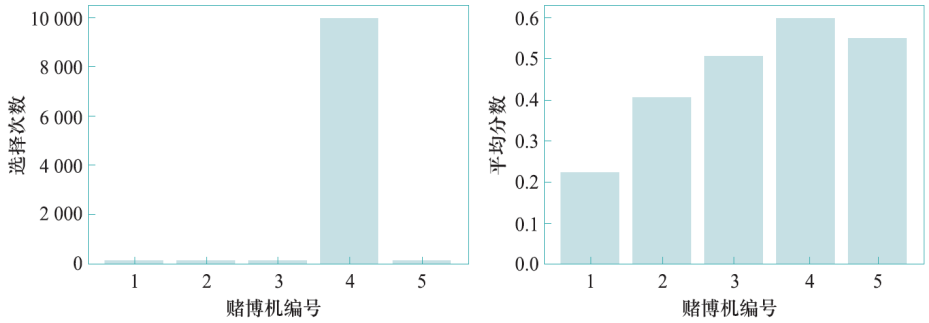


图 3.17　贪心算法选取 10 000 个动作的一次模拟实验结果

注意：如果重新进行一个10000次贪心算法的模拟，**由于随机性的存在，其结果会存在不同。**

蒙特卡洛树搜索——不足

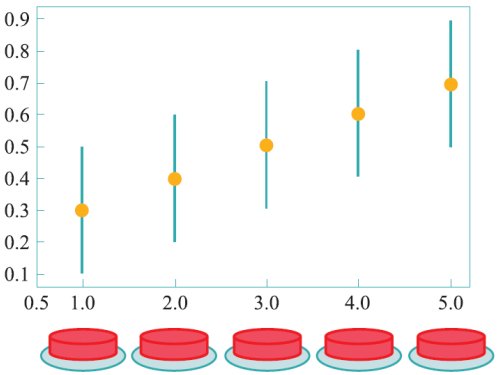


图 3.16　多臂赌博机问题中每个赌博机收益分数的分布情况

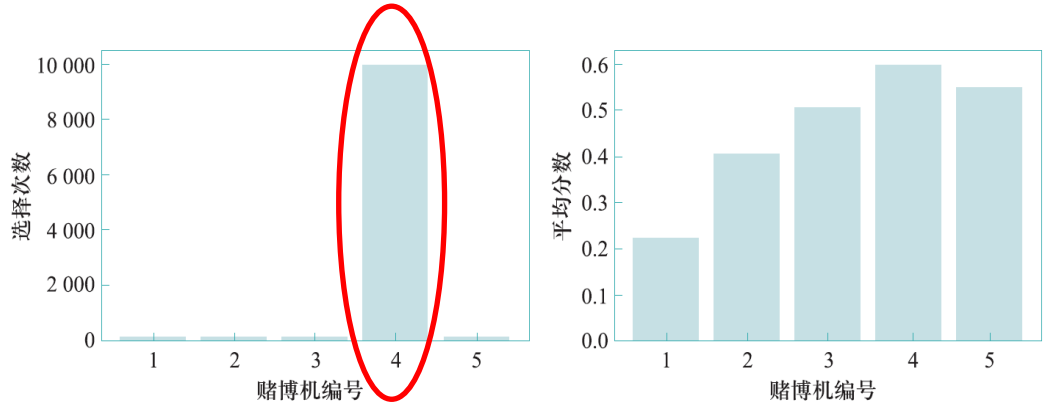


图 3.17　贪心算法选取 10 000 个动作的一次模拟实验结果

贪心算法：盲从过去而导致不足

上述困境体现了探索（exploration）和利用（exploitation）之间存在的对立关系。贪心算法基本上是**利用**从已有尝试结果中所得估计来指导后续动作（如倾向于选择4号赌博机），但问题是所得估计往往不能准确反映未被（大量）探索过的动作，如由于很少摇动其他编号的赌博机而无法准确估计其他编号赌博机可能带来的收益分数。因此，需要在贪心算法中增加一个能够改变其“惯性”的内在动力，以使得贪心算法能够访问那些尚未被（充分）访问过的空间。

蒙特卡洛树搜索——改进的贪心算法

ϵ-贪心算法

在第t步，ϵ-贪心算法按照如下机制来选择摇动赌博机：

$$l_t = \begin{cases} \text{argmax}_i \, \bar{x}_{i,T_{(i,t-1)}}, & \text{以} 1-\epsilon \text{的概率} \\ \text{随机的} i \in \{1,2,\cdots,K\}, & \text{以} \epsilon \text{的概率} \end{cases}$$

即以$1-\epsilon$的概率选择在过去$t-1$次摇动赌博机臂膀行动中所得平均收益分数最高的赌博机进行摇动；以ϵ的概率随机选择一个赌博机进行摇动。

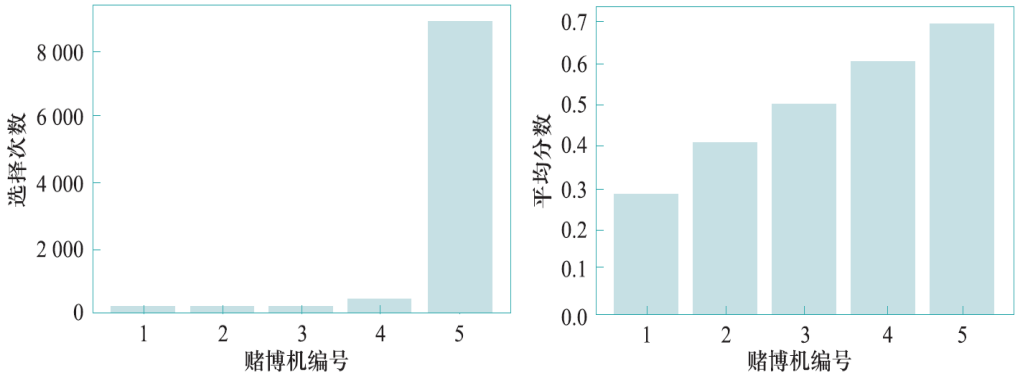


图 3.18　在图 3.17 的基础上再进行 10 000 次 $\epsilon-$贪心算法的实验结果

ϵ-贪心算法对估计摇动五个赌博机臂膀所带来期望分数更加准确了。并且因为编号为5的赌博机的平均收益分数最高，算法倾向于选择动作a_5，与预期相符。这说明其**探索机制**发挥了作用。

蒙特卡洛树搜索——上限置信区间算法

- ϵ-贪心算法虽然能有效地促使算法进行探索，但通过这种随机机制来选择动作进行探索的做法很可能不是最优的。比如，可能存在一个给出更好奖励期望的动作，但因为智能体对其探索次数少而认为其期望奖励小。因此，需要**对那些探索次数少或几乎没有被探索过的动作赋予更高的优先级**。

- 但是ϵ-贪心算法没有将每个动作被探索的次数纳入考虑，这似乎是不合理的。另一方面，简单地优先去尝试探索次数少的动作也未必合理，因为**只需要少量的探索次数，就能够较好估计方差较小的动作的奖励期望**。

- 在探索过程中，应该**优先探索估计值不确定度高的动作**。另外，如果从一个动作的奖励估计值非常小，那么算法也没有必要将其作为优先探索对象。**上限置信区间**（Upper Confidence Bounds, UCB1）算法正是遵循这一思路来进行探索。

蒙特卡洛树搜索——上限置信区间算法策略

UCB1算法的策略是：**为每个动作的奖励期望计算一个估计范围，优先采用估计范围上限较高的动作。**

如图3.19所示，动作1的奖励期望取值的不确定度（估计范围）虽然最大，但是因为其均值太小，因此UCB1算法不优先考虑探索动作1。动作2和3的奖励期望的均值相同，但是动作2的奖励期望取值的不确定度（估计范围）更大，于是因为置信上限更大，动作2会被UCB1算法优先考虑。

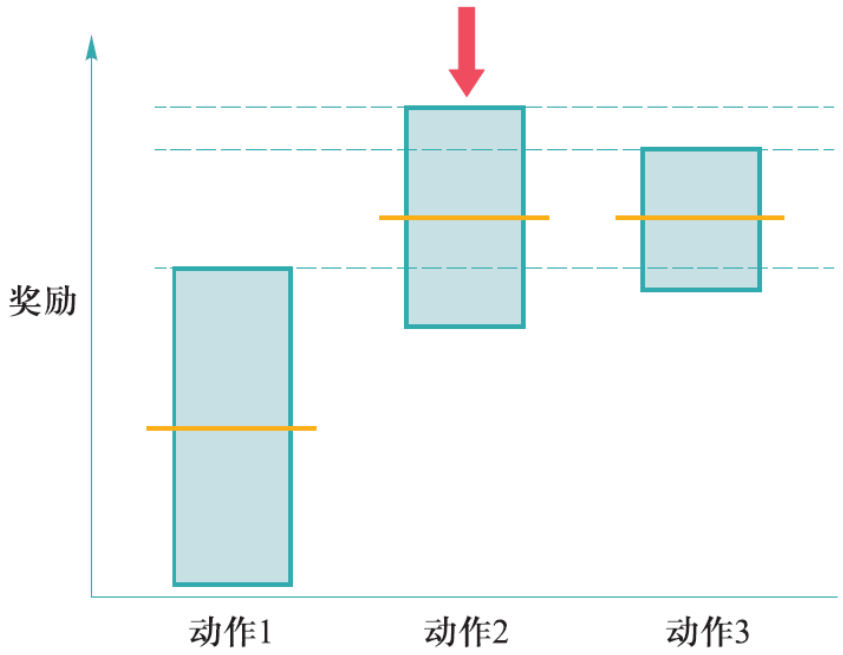


图 3.19　UCB1 算法的策略示意

简单来说就是充分寻找潜力股！

蒙特卡洛树搜索

- 无论是一般搜索问题，还是对抗搜索问题，在问题特别复杂时，搜索树可能会变得十分巨大，以至于**搜索算法很难在短时间内完全探索整棵搜索树**。为了解决这个问题，前面分别探讨了如何利用辅助信息来找到高效的结点扩展顺序以及介绍了可减少不必要扩展的结点数量的alpha-beta剪枝算法。
- 不难发现，对搜索算法进行优化以提高搜索效率基本上是在解决如下两个问题：**优先扩展哪些结点以及放弃扩展哪些结点**，综合来看也可以概括为如何**高效**地扩展搜索树。
- 如果将目标稍微降低，改为求解一个近似最优解，则上述问题可以看成是如下探索性问题：算法从根结点开始 ，每一步动作为选择（在非叶子结点）或扩展（在叶子结点）一个孩子结点。**可以用执行该动作后所收获奖励来判断该动作优劣**。奖励可以根据从当前结点出发到达目标路径的代价或游戏终局分数来定义。**算法会倾向于扩展获得奖励较高的结点**。
- 算法事先不知道每个结点将会得到怎样的代价（或终局分数）分布，只能通过**采样式探索**来得到计算奖励的样本。由于这个算法利用蒙特卡洛法通过采样来估计每个结点的价值，因此它被称为**蒙特卡洛树搜索**（Monte-Carlo Tree Search）算法。

蒙特卡洛树搜索

- **选择（selection）**：选择指算法从搜索树的根结点开始，向下递归选择子结点，直至到达叶子结点或者到达尚未被完全扩展的的结点L，如图(a)所示。这个向下递归选择过程可由UCB1算法来实现，在递归选择过程中记录下每个结点被选择次数和每个结点得到的奖励均值。
- **扩展（expansion）**：如果结点 L 不是一个终止结点（或对抗搜索的终局结点），则随机扩展它的一个未被扩展过的后继结点M，如图 (b)所示。
- **模拟（simulation）**：从结点M出发，模拟扩展搜索树，直到找到一个终止结点，如图 (c)所示。模拟过程使用的策略和采用UCB1算法实现的选择过程并不相同，前者通常会使用比较简单的策略，例如使用随机策略。
- **反向传播（Back Propagation）**：用模拟所得结果（终止结点的代价或游戏终局分数）回溯更新模拟路径中M以上（含M）结点的奖励均值和被访问次数，如图 (d)所示。

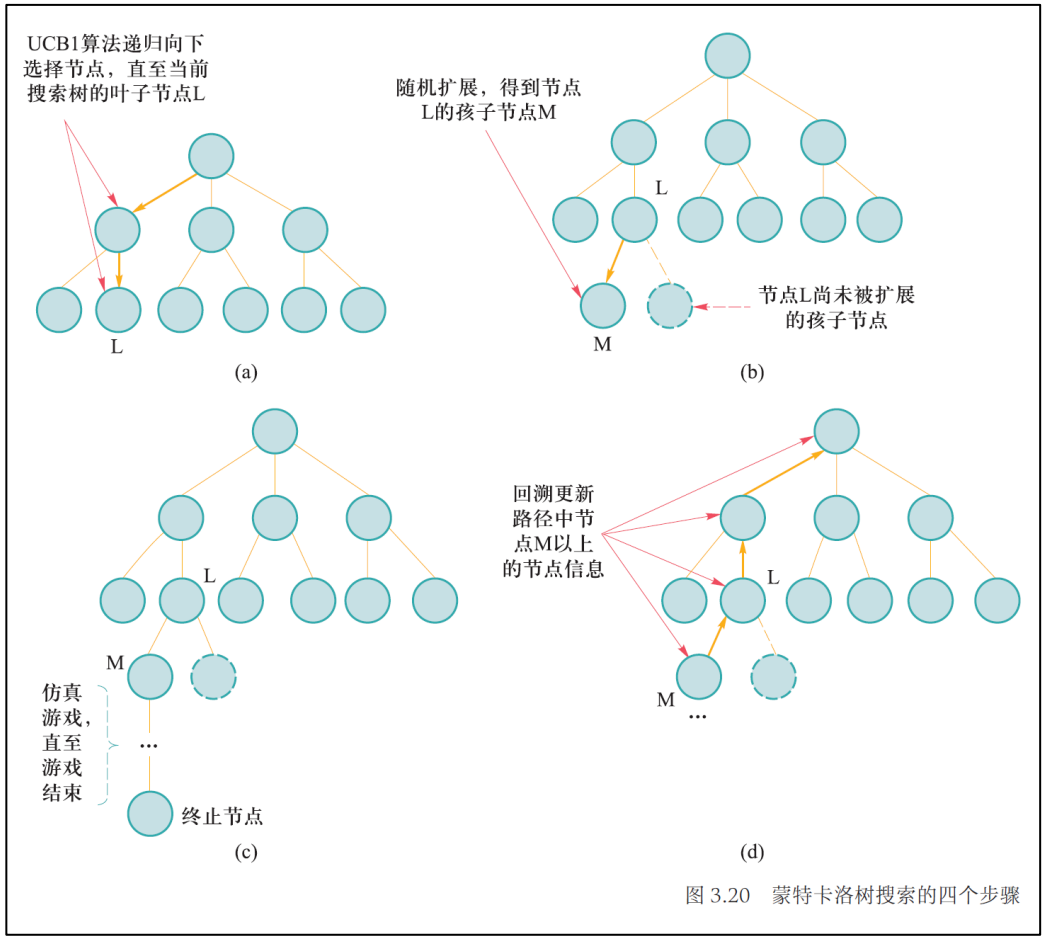


图 3.20　蒙特卡洛树搜索的四个步骤

蒙特卡洛树搜索

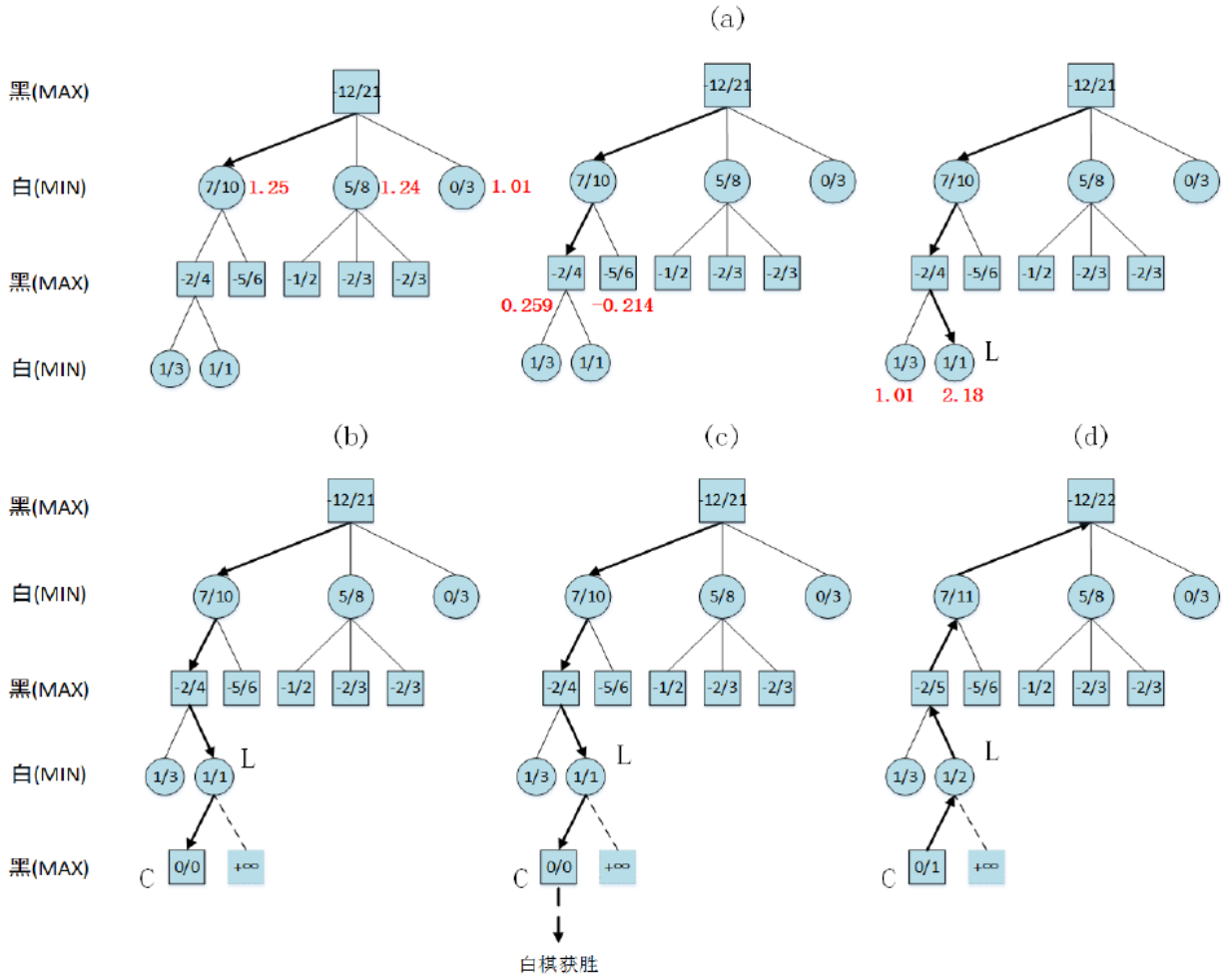


图3.22 一个加入Minimax剪枝的蒙特卡洛树搜索的例子
(UCB值在对应结点边用红色字体标出)

谢谢！