搜索求解

吴飞

浙江大学计算机学院

提纲

1、 启发式搜索

2、 对抗搜索

3、 蒙特卡洛树搜索

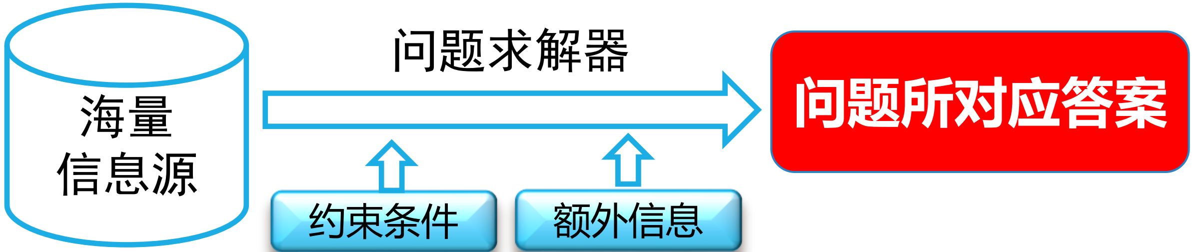
人工智能中的搜索

你见，或者不见我

我就在那里

不悲 不喜

---扎西拉姆多多



人工智能中的搜索：以寻找最短路径问题为例

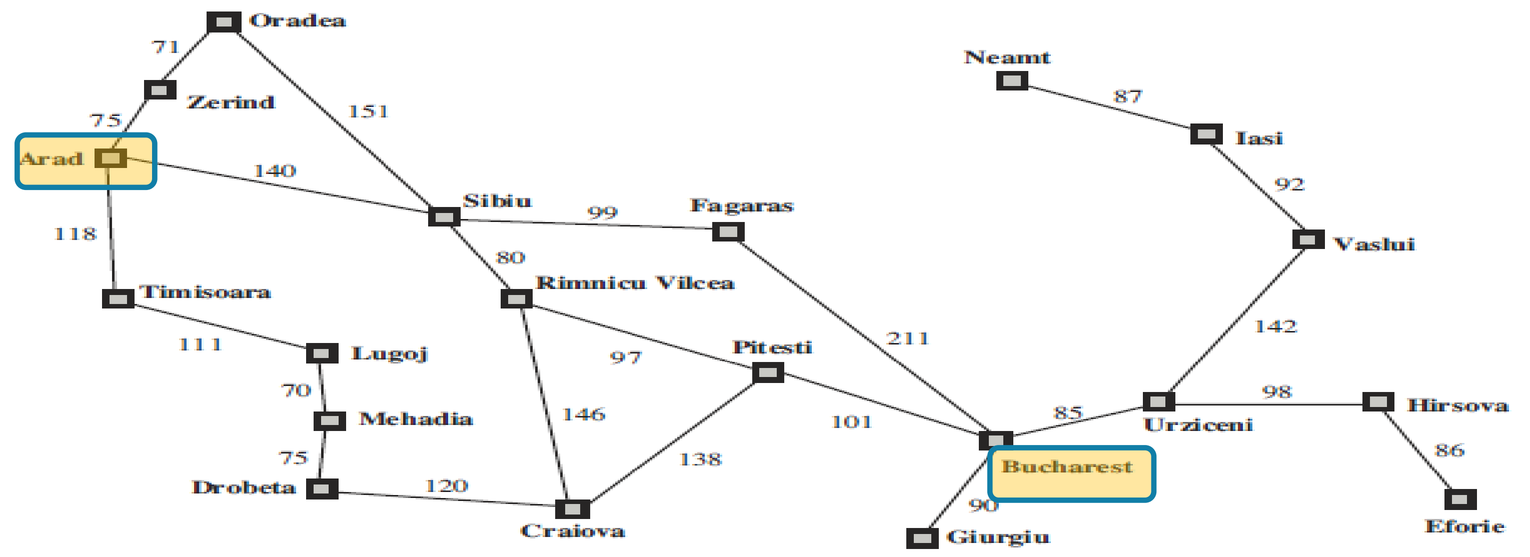


Figure 3.2 A simplified road map of part of Romania.

问题：寻找从Arad到Bucharest的一条最短路径

搜索算法的形式化描述：

〈状态、动作、状态转移、路径、测试目标〉

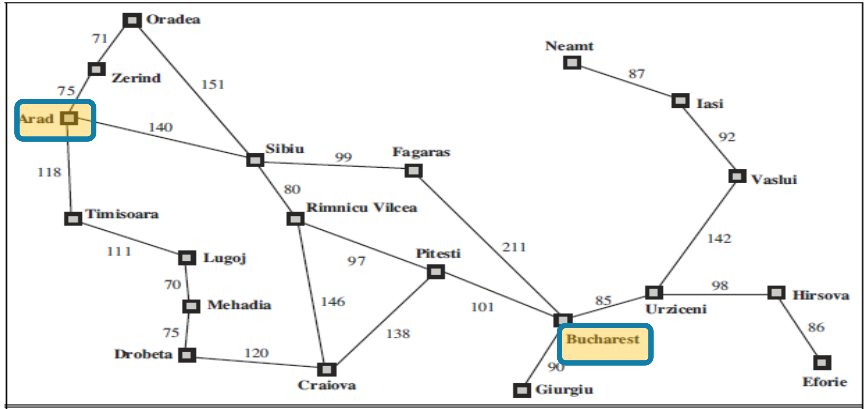


Figure 3.2　A simplified road map of part of Romania.

状态

从原问题转化出的问题描述。例如，在最短路径问题中，城市可作为状态。将原问题对应的状态称为初始状态。

动作

从当前时刻所处状态转移到下一时刻所处状态所进行操作。一般而言这些操作都是离散的。

问题：寻找从Arad到Bucharest的一条路径，满足路径最短、时间最少、价钱最经济？

搜索算法的形式化描述：

〈状态、动作、状态转移、路径、测试目标〉

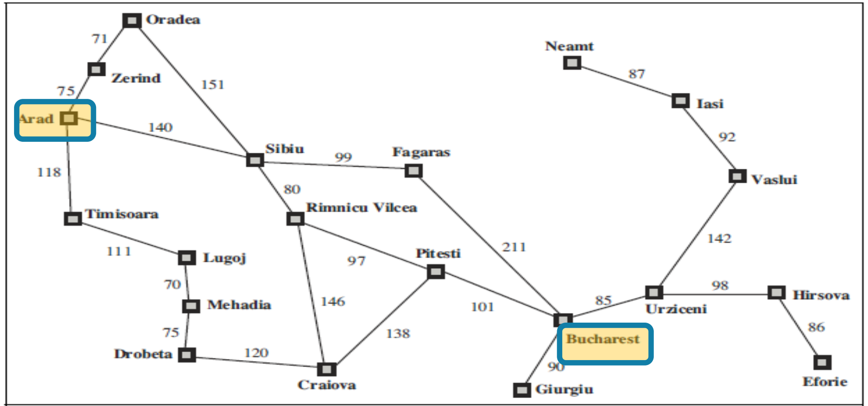


Figure 3.2 A simplified road map of part of Romania.

状态转移

对某一时刻对应状态进行某一种操作后，所能够到达状态。

路径

一个状态序列。该状态序列被一系列操作所连接。如从Arad到Bucharest所形成的路径。

目标测试

评估当前状态是否为所求解的目标状态。

问题：寻找从Arad到Bucharest的一条路径，满足路径最短、时间最少、价钱最经济？

搜索算法：启发式搜索(有信息搜索)

在搜索的过程中利用与所求解问题相关的辅助信息，其代表算法为**贪婪最佳优先搜索**(Greedy best-first search)和**A*搜索**。

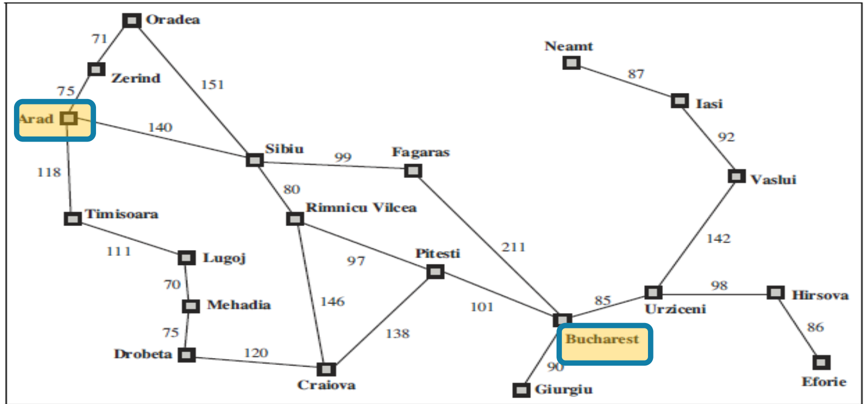


Figure 3.2 A simplified road map of part of Romania.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest之间的直线距离

搜索算法：启发式搜索(有信息搜索)

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节点
启发函数（heuristic function）**h(n)**	计算从节点n到目标节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。

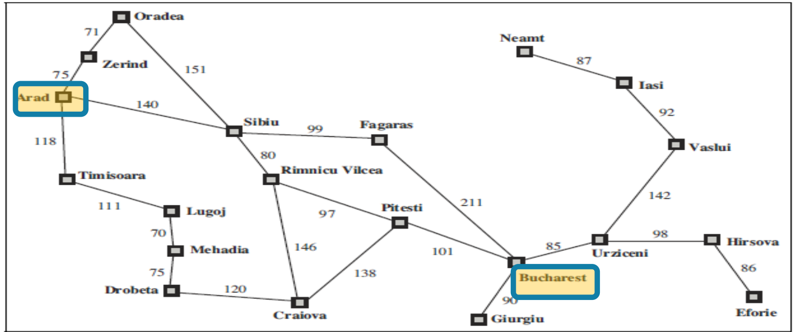


Figure 3.2 A simplified road map of part of Romania.

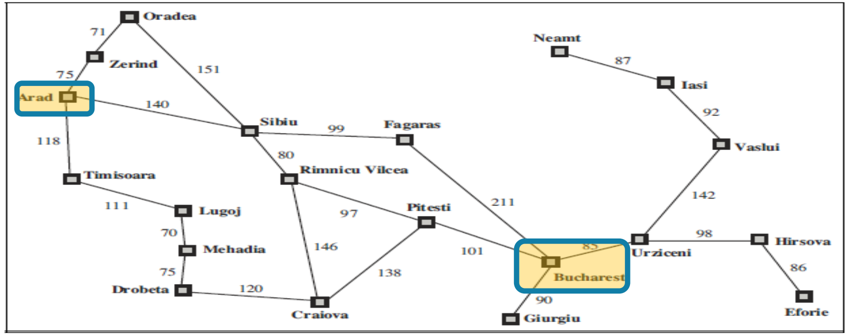
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest之间的直线距离

搜索算法：贪婪最佳优先搜索

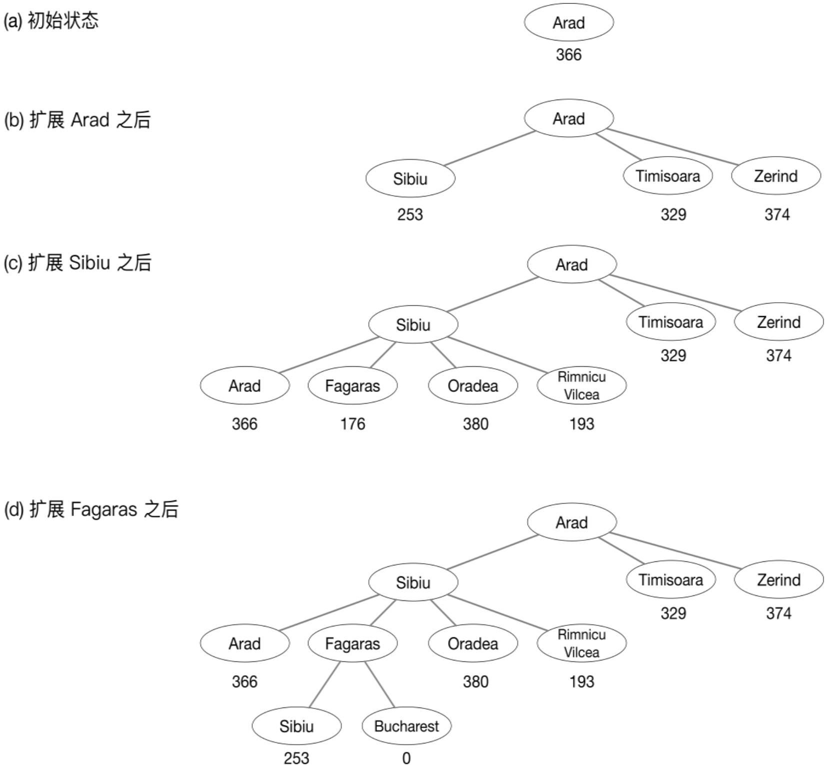
贪婪最佳优先搜索(Greedy best-first search)：评价函数f(n)=启发函数h(n)



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest之间的直线距离



搜索算法：贪婪最佳优先搜索

不足之处：

- 贪婪最佳优先搜索不是最优的。经过Sibiu到Fagaras到Bucharest的路径比经过Rimnicu Vilcea到Pitesti到Bucharest的路径要长32公里。
- 启发函数代价最小化这一目标会对错误的起点比较敏感。考虑从Iasi到Fagaras的问题，由启发式建议须先扩展Neamt，因为其离Fagaras最近，但是这是一条存在死循环路径。

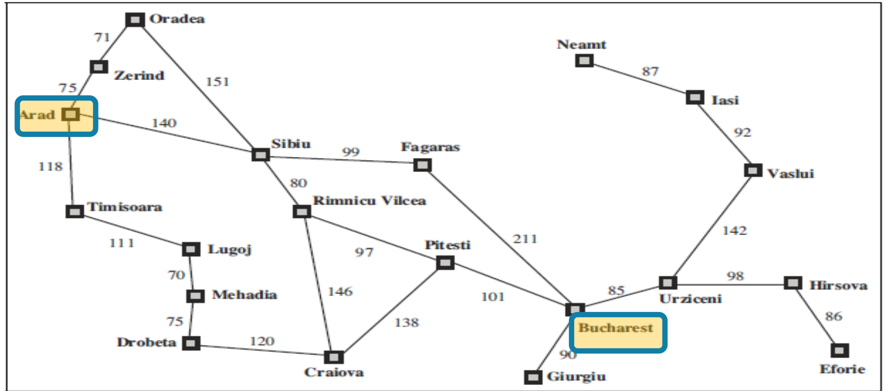


Figure 3.2 A simplified road map of part of Romania.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest之间的直线距离

搜索算法：贪婪最佳优先搜索

不足之处：

- 贪婪最佳优先搜索也是不完备的。所谓不完备即它可能沿着一条无限的路径走下去而不回来做其他的选择尝试，因此无法找到最佳路径这一答案。

- 在最坏的情况下，贪婪最佳优先搜索的时间复杂度和空间复杂度都是$O(b^m)$，其中b是节点的分支因子数目、m是搜索空间的最大深度。

因此，需要设计一个良好的启发函数

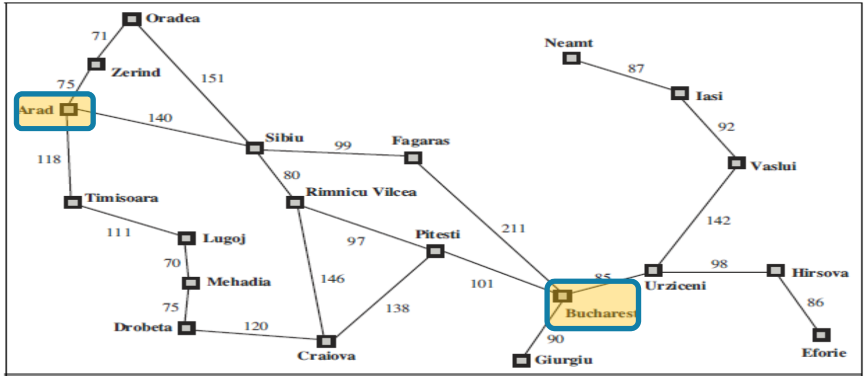


Figure 3.2 A simplified road map of part of Romania.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest之间的直线距离

搜索算法：A*算法

定义评价函数：$f(n) = g(n) + h(n)$

- $g(n)$表示从起始节点到节点n的开销代价值，$h(n)$表示从节点n到目标节点路径中所估算的最小开销代价值。
- $f(n)$可视为经过节点n 、具有最小开销代价值的路径。

$$\underbrace{f(n)}_{\text{评估函数}} = \underbrace{g(n)}_{\text{当前最小开销代价}} + \underbrace{h(n)}_{\text{后续最小开销代价}}$$

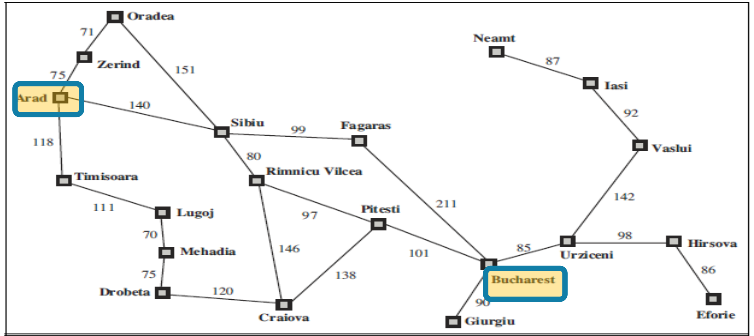


Figure 3.2 A simplified road map of part of Romania.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

辅助信息：任意一个城市与Bucharest
之间的直线距离

搜索算法：A*算法

$$\underbrace{f(n)}_{\text{评估函数}} = \underbrace{g(n)}_{\text{当前最小开销代价}} + \underbrace{h(n)}_{\text{后续最小开销代价}}$$

为了保证A*算法是最优（optimal），需要启发函数$h(n)$是可容的(admissible heuristic)和一致的(consistency，或者也称单调性，即 monotonicity)。

最优	不存在另外一个解法能得到比A*算法所求得解法具有更小开销代价。
可容(admissible)	专门针对启发函数而言，即启发函数不会过高估计(over-estimate)从节点n到目标结点之间的实际开销代价（即小于等于实际开销）。如可将两点之间的直线距离作为启发函数，从而保证其可容。
一致性（单调性）	假设节点n的后续节点是n'，则从n到目标节点之间的开销代价一定小于从n到n'的开销再加上从n'到目标节点之间的开销，即 $h(n) \leq c(n, a, n') + h(n')$。这里$n'$是$n$经过行动$a$所抵达的后续节点，$c(n, a, n')$指$n'$和$n$之间的开销代价。

In computer science, a heuristic function is said to be admissible if it is no more than the lowest-cost path to the goal. In other words, a heuristic is admissible if it never overestimates the cost of reaching the goal. An admissible heuristic is also known as an optimistic heuristic.

搜索算法：A*算法

$$\underbrace{f(n)}_{\text{评估函数}} = \underbrace{g(n)}_{\text{当前最小开销代价}} + \underbrace{h(n)}_{\text{后续最小开销代价}}$$

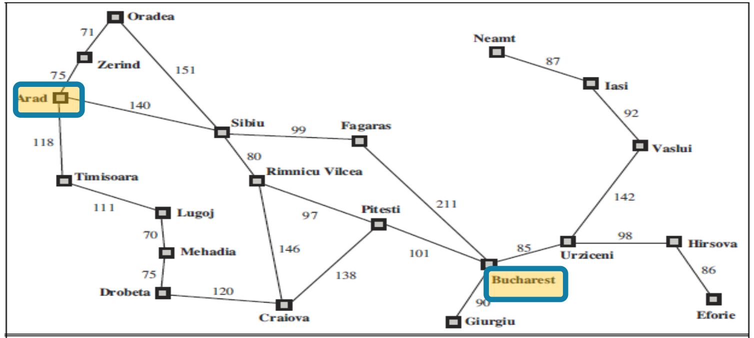
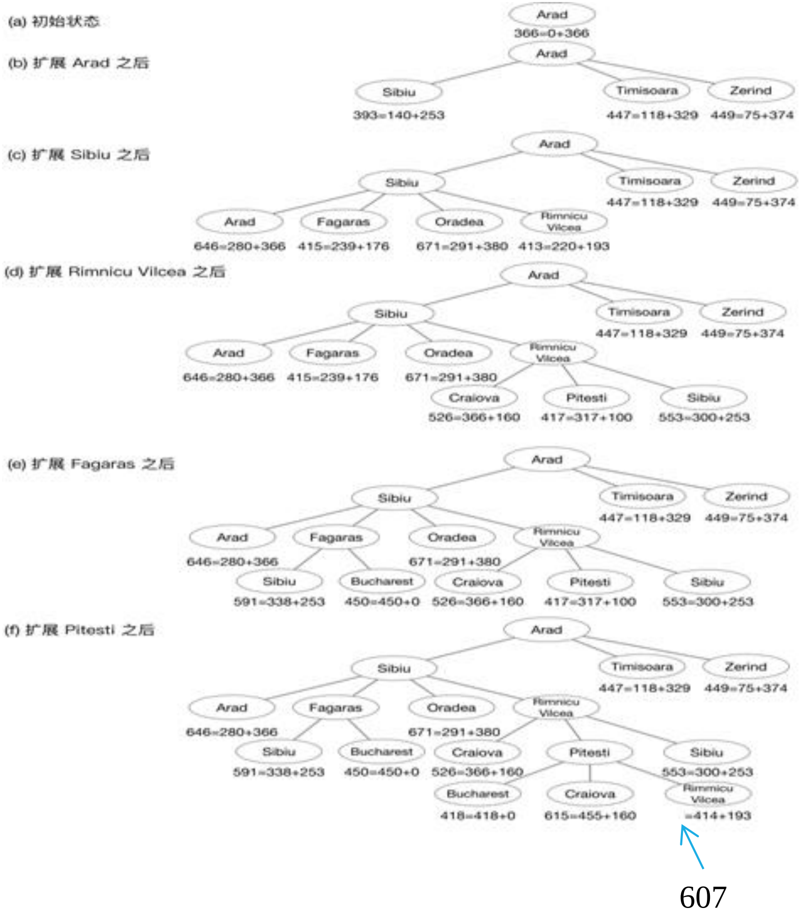


Figure 3.2 A simplified road map of part of Romania.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.



搜索算法：A*算法

A*算法保持最优的条件：启发函数具有可容性(admissible)和一致性(consistency)。

- 将直线距离作为启发函数$h(n)$，则启发函数一定是可容的，因为其不会高估开销代价。
- $g(n)$ 是从起始节点到节点n的实际代价开销，且$f(n) = g(n) + h(n)$，因此$f(n)$不会高估经过节点n路径的实际开销。
- $h(n) \leq c(n, a, n') + h(n')$ 构成了三角不等式。这里节点n、节点n'和目标结点G_n之间组成了一个三角形。如果存在一条经过节点n'，从节点n到目标结点G_n的路径，其代价开销小于$h(n)$，则破坏了$h(n)$是从节点n到目标结点G_n所形成的具有最小开销代价的路径这一定义。

搜索算法：A*算法

$$\underbrace{f(n)}_{\text{评估函数}} = \underbrace{g(n)}_{\text{当前最小开销代价}} + \underbrace{h(n)}_{\text{后续最小开销代价}}$$

- Tree-search的A*算法中，如果启发函数$h(n)$是可容的，则A*算法是最优的和完备的；在Graph-search的A*算法中，如果启发函数$h(n)$是一致的，A*算法是最优的。

- 如果函数满足一致性条件，则一定满足可容条件；反之不然。

- 直线最短距离函数既是可容的，也是一致的。

搜索算法：A*算法

$$\underbrace{f(n)}_{\text{评估函数}} = \underbrace{g(n)}_{\text{当前最小开销代价}} + \underbrace{h(n)}_{\text{后续最小开销代价}}$$

- 如果$h(n)$是一致的（单调的），那么$f(n)$ 一定是非递减的(non-decreasing)。

证明：假设节点n'是节点n的后续节点，则有$g(n') = g(n) + c(n, a, n')$ (a是从节点n到节点n'的一个行动)，存在：

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

搜索算法：A*算法

$$\underbrace{f(n)}_{\text{评估函数}} = \underbrace{g(n)}_{\text{当前最小开销代价}} + \underbrace{h(n)}_{\text{后续最小开销代价}}$$

- 如果A*算法将节点n选择作为具有最小代价开销的路径中一个节点，则n一定是最优路径中的一个节点。即最先被选中扩展的节点在最优路径中。

证明：反证法。假设上述结论不成立。则存在一个未被访问的节点n'位于从起始节点到节点n的最佳路径上。根据非递减性质，存在$f(n) \geq f(n')$，则n'应该已经被访问过了（expanded）。因此，无论什么时候，一旦一个节点被访问到，它一定位于从起始节点到它自己之间的最佳路径上。

提纲

1、 启发式搜索

2、 对抗搜索

3、蒙特卡洛树搜索

对抗搜索

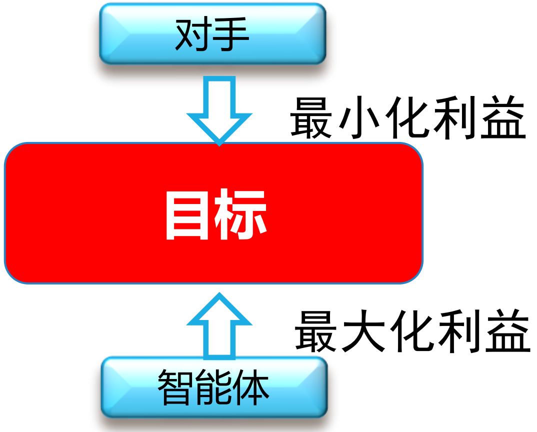
- 对抗搜索(Adversarial Search)也称为博弈搜索(Game Search)
- 在一个竞争的环境中，智能体(agents)之间通过竞争实现相反的利益，一方**最大化**这个利益，另外一方**最小化**这个利益。

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**



对抗搜索：主要内容

- **最小最大搜索(Minimax Search):** 最小最大搜索是在对抗搜索中最为基本的一种让玩家来计算最优策略的方法.
- **Alpha-Beta剪枝搜索(Pruning Search)**: 一种对最小最大搜索进行改进的算法，即在搜索过程中可剪除无需搜索的分支节点，且不影响搜索结果。.
- **蒙特卡洛树搜索(Monte-Carlo Tree Search)**: 通过采样而非穷举方法来实现搜索。

对抗搜索

本课程目前主要讨论在确定的、全局可观察的、竞争对手轮流行动、零和游戏（zero-sum）下的对抗搜索

两人对决游戏 (MAX and MIN, MAX先走) 可如下形式化描述，从而将其转换为对抗搜索问题

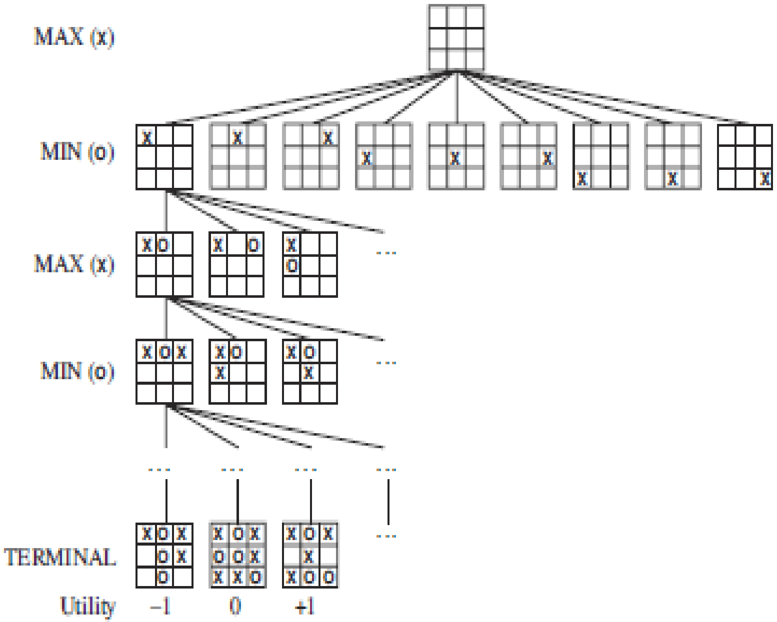
初始状态S_0	游戏所处于的初始状态
玩家$PLAYER(s)$	在当前状态s下，该由哪个玩家采取行动
行动$ACTIONS\ (s)$	在当前状态s下所采取的可能移动
状态转移模型$RESULT\ (s,a)$	在当前状态s下采取行动a后得到的结果
终局状态检测$TERMINAL-TEST\ (s)$	检测游戏在状态s是否结束
终局得分$UTILITY\ (s,p)$	在终局状态s时，玩家p的得分。

注：所谓零和博弈是博弈论的一个概念，属非合作博弈。指参与博弈的各方，在严格竞争下，一方的收益必然意味着另一方的损失，博弈各方的收益和损失相加总和永远为“零”，双方不存在合作的可能。与“零和”对应，“双赢博弈”的基本理论就是“利己”不“损人”，通过谈判、合作达到皆大欢喜的结果。

对抗搜索

Tic-Tac-Toe游戏的对抗搜索

- MAX先行，可在初始状态的9个空格中任意放一个X
- MAX希望游戏终局得分高、MIX希望游戏终局得分低
- 所形成游戏树的叶子结点有$9! = 362,880$，国际象棋的叶子节点数为10^{40}



Tic-Tac-Toe中部分搜索树

对抗搜索：minimax算法

给定一个游戏搜索树，minimax算法通过每个节点的minimax值来决定最优策略。当然，MAX希望最大化minimax值，而MIN则相反

$$
\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}
$$

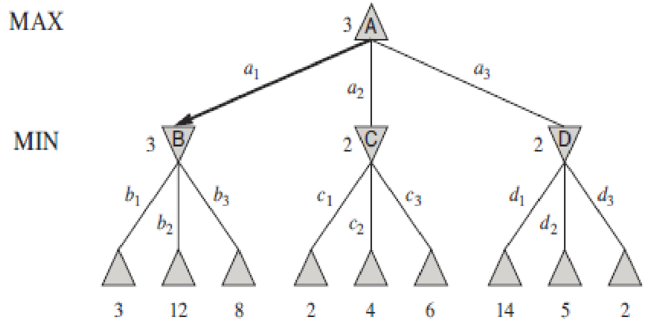


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

通过minimax算法，我们知道，对于MAX而言采取a_1行动是最佳选择，因为这能够得到最大minimax值（收益最大）。

对抗搜索：minimax算法

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all actions at once, or $O(m)$ for an algorithm that generates actions one at a time (see page 87). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

对抗搜索：minimax算法

◆**Complete** ? Yes (if tree is finite)

◆**Optimal** ? Yes (against an optimal opponent)

◆**Time complexity** ? O(b^m)

◆**Space complexity** ? O(b × m) (depth-first exploration)

m 是游戏树的最大深度，在每个节点存在b个有效走法

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible

对抗搜索：minimax算法

优点:

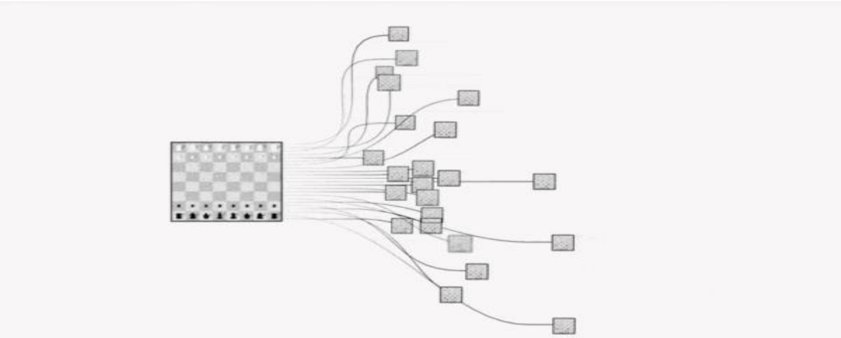
- 算法是一种简单有效的对抗搜索手段
- 在对手也“尽力而为”前提下，算法可返回最优结果

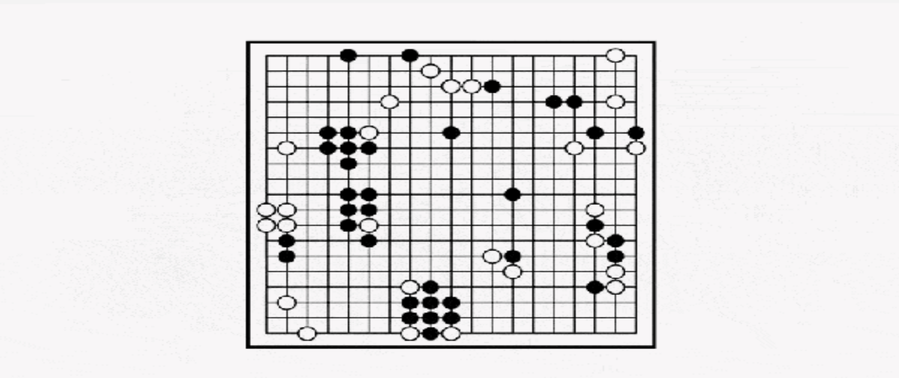
缺点:

- 如果搜索树极大，则无法在有效时间内返回结果

改善:

- 使用alpha-beta pruning算法来减少搜索节点
- 对节点进行采样、而非逐一搜索 (i.e., MCTS)





枚举当前局面之后每一种下法，然后计算每个后续局面的赢棋概率，选择概率最高的后续局面

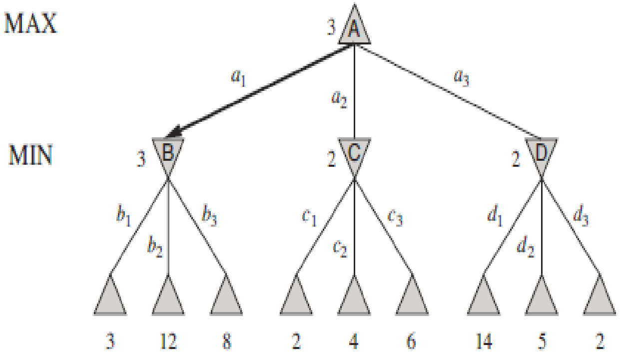
对抗搜索：Alpha-Beta 剪枝搜索

在极小化极大算法（minimax算法）中减少所搜索的搜索树节点数。该算法和极小化极大算法所得结论相同，但剪去了不影响最终结果的搜索分枝。

$$MINIMAX(root)$$
$$= \max(\min(3.12,8), \min(2, x, y), \min(14,5,2))$$
$$= \max(3, \min(2, x, y), 2)$$
$$= \max(3, z, 2) = 3$$

$$where\ z = \min(2, x, y) \le 2$$

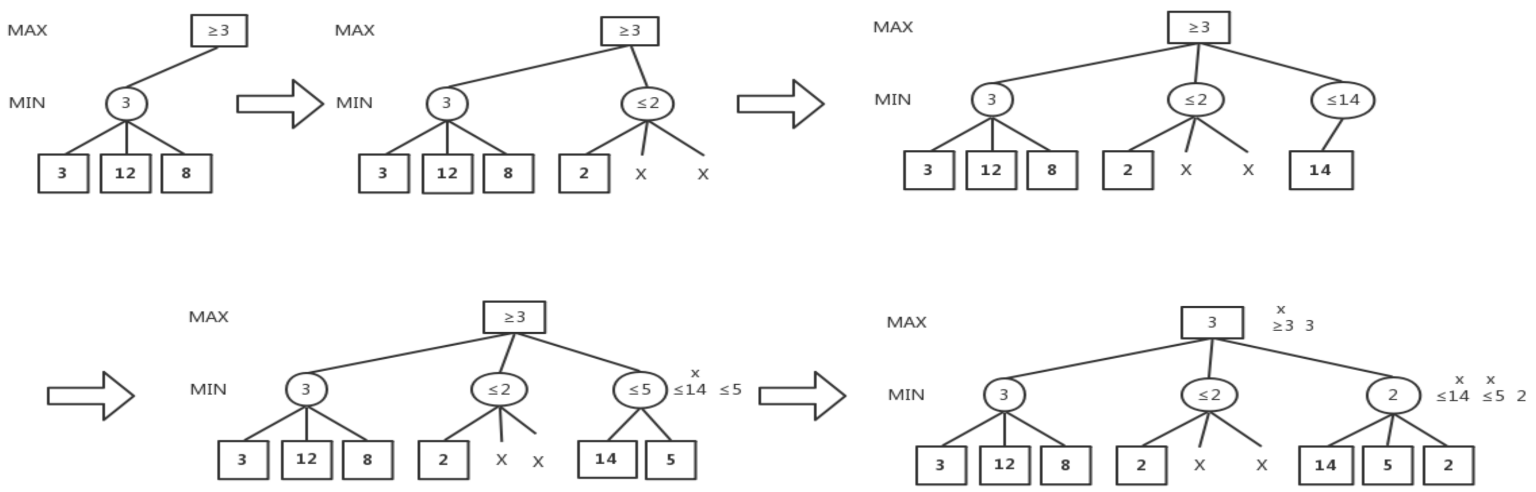
可以看出：根节点（即 MAX选手）的选择与x和y 两个值无关（因此， x和y可以被剪枝去除）



图中MIN选手所在的节点C下属分支4和6与根节点最终优化决策的取值无关，可不被访问。

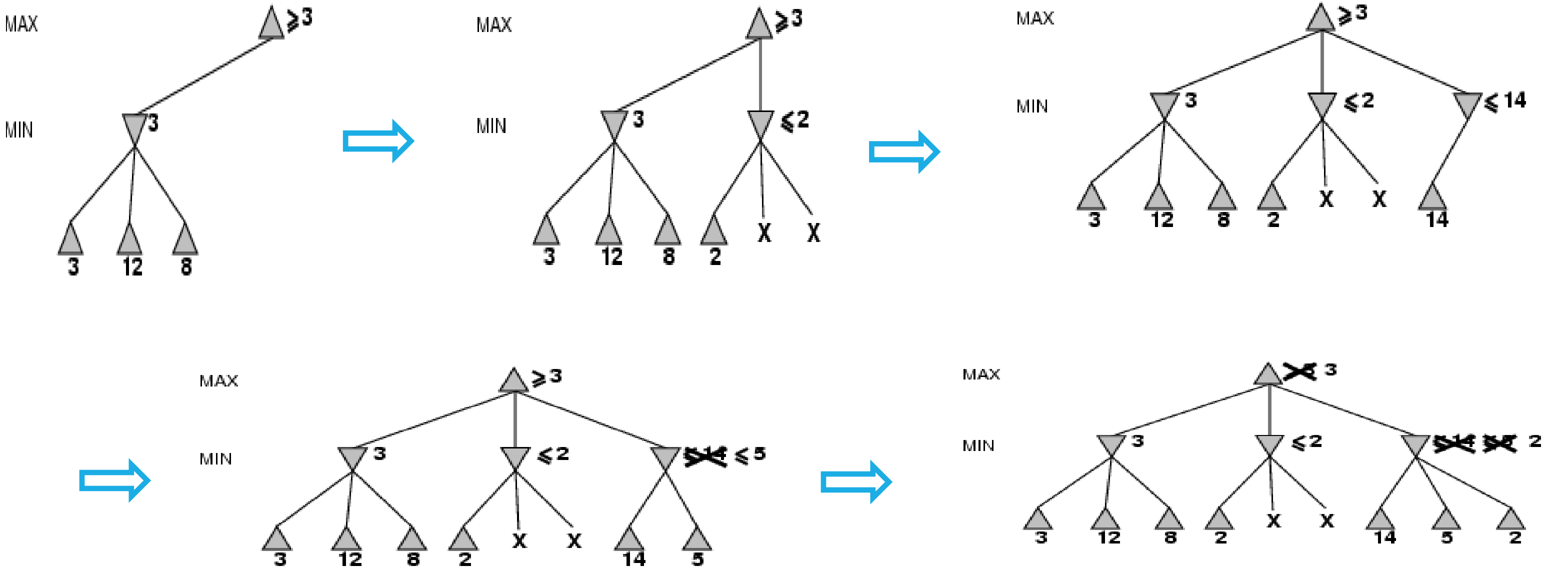
对抗搜索：Alpha-Beta 剪枝搜索

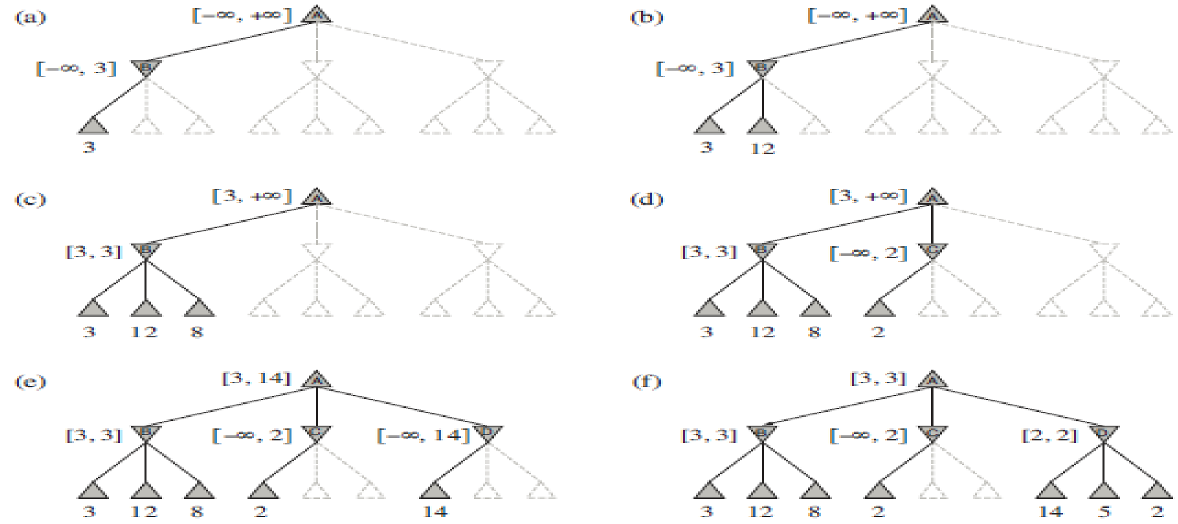
Alpha值(α)	MAX节点目前得到的最高收益
Beta值(β)	MIN节点目前可给对手的最小收益
α和β的值初始化分别设置为$-\infty$和∞	



对抗搜索：Alpha-Beta 剪枝搜索

Alpha值(α)	MAX节点目前得到的最高收益
Beta值(β)	MIN节点目前可给对手的最小收益
α和β的值初始化分别设置为$-\infty$和∞	





替换之后

2 5 14

Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

从α和β的变化来理解剪枝过程

对抗搜索：如何利用Alpha-Beta 剪枝

Alpha值(α)	玩家MAX（根节点）目前得到的最高收益
	假设n是MIN节点，如果n的一个后续节点可提供的收益小于α，则n及其后续节点可被剪枝
Beta值(β)	玩家MIN目前给对手的最小收益
	假设n是MAX节点，如果n的一个后续节点可获得收益大于β，则n及其后续节点可被剪枝
α和β的值初始化分别设置为$-\infty$和∞	

对手节点(min节点)在这里可提供的最大收益是m

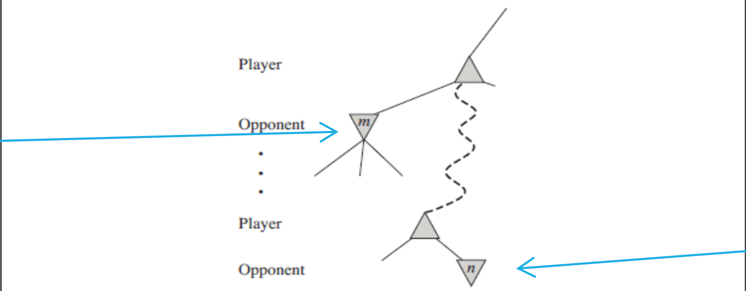


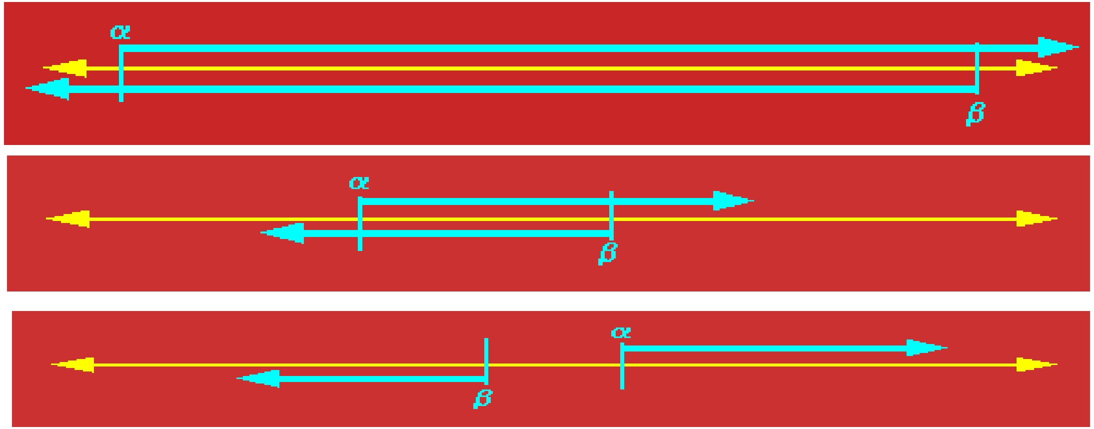
Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

对手节点(min节点)在这里可提供的最大收益是n

在图中$m > n$，因此n右边节点及后续节点就被剪枝掉了

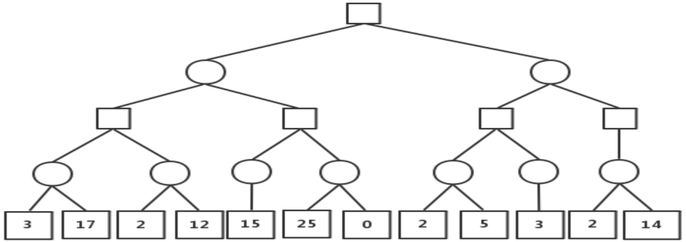
对抗搜索：如何利用Alpha-Beta 剪枝

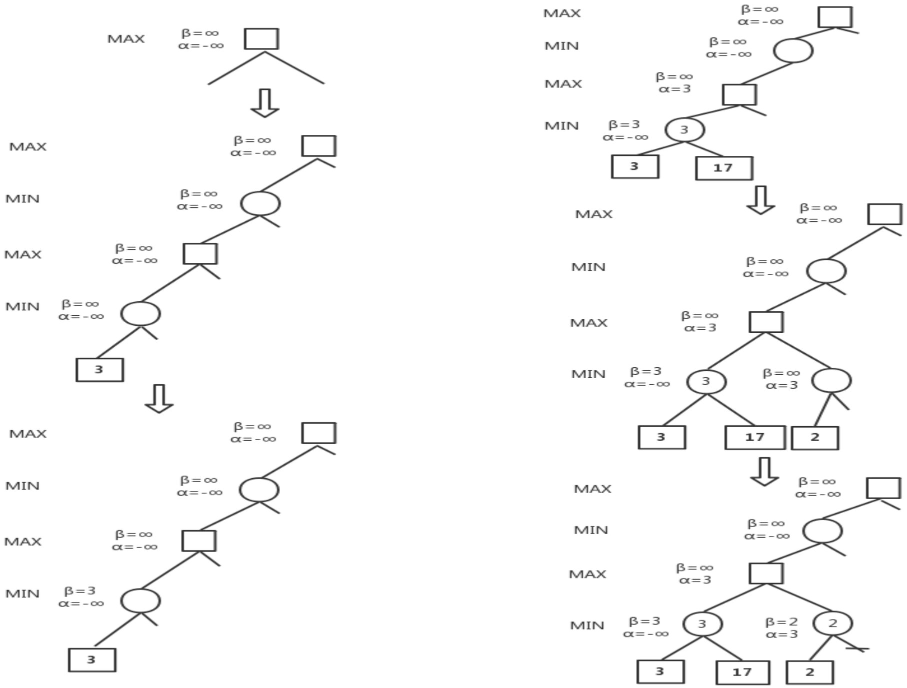
- α 为可能解法的最大上界
- β 为可能解法的最小下界
- 如果节点 N 是可能解法路径中的一个节点，则其产生的收益一定满足如下条件：
  $\alpha \leq reward(N) \leq \beta$(其中$reward(N)$是节点$N$产生的收益)



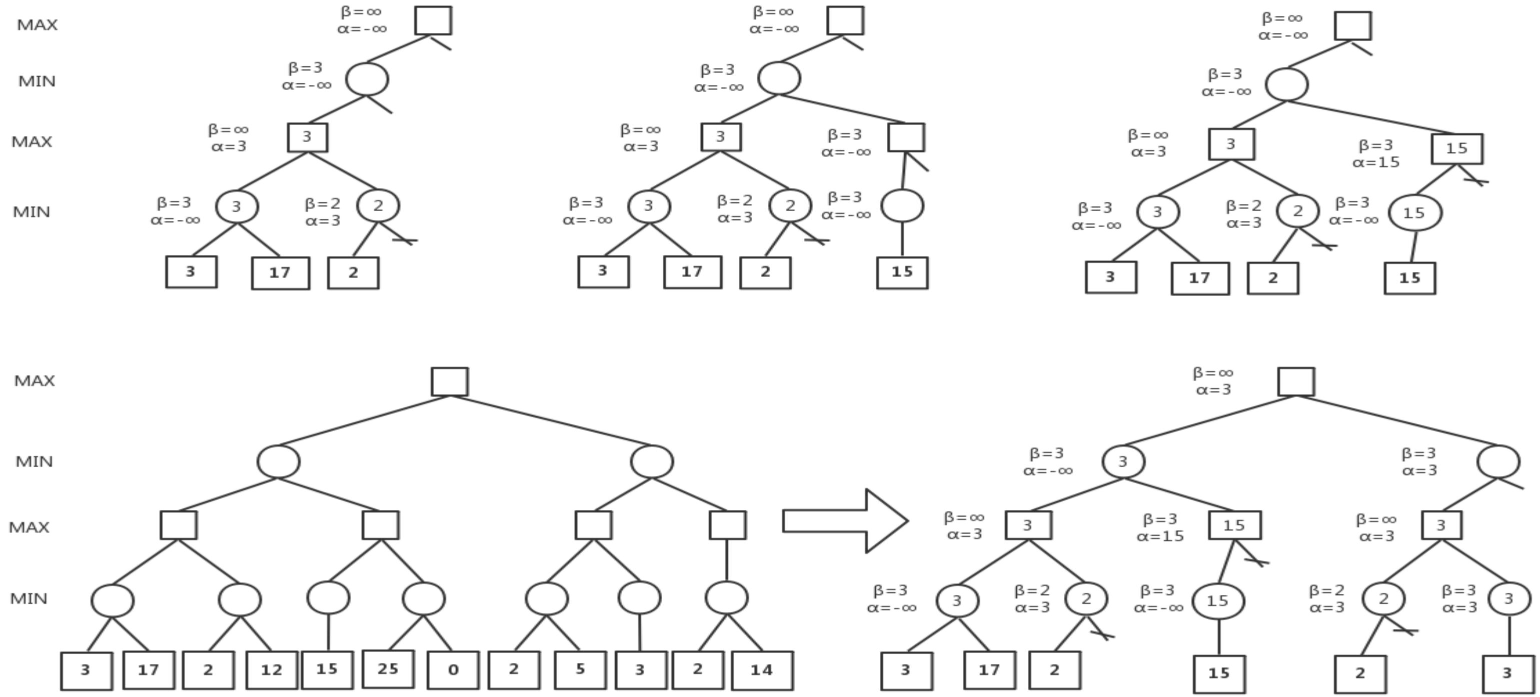
每个节点有两个值，分别是α和β。节点的α和β值在搜索过程中不断变化。其中，α从负无穷大($-\infty$)逐渐增加、β从正无穷大(∞)逐渐减少。如果一个节点中$\alpha > \beta$，则该节点的后续节点可剪枝。

对抗搜索：Alpha-Beta 剪枝搜索示意





对抗搜索：Alpha-Beta 剪枝搜索示意



对抗搜索：Alpha-Beta 剪枝搜索的算法描述

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty$, $+\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

对抗搜索：Alpha-Beta 剪枝搜索的性质

- 剪枝本身不影响算法输出结果
- 节点先后次序会影响剪枝效率
- 如果节点次序“恰到好处”，Alpha-Beta剪枝的时间复杂度为

  O($b^{\frac{m}{2}}$)，最小最大搜索的时间复杂度为O(b^m)

提纲

1、 启发式搜索

2、 对抗搜索

3、蒙特卡洛树搜索

对抗搜索：蒙特卡洛树搜索

(exploitation)与探索(exploration)在游戏博弈树上的有机协调

推荐阅读材料

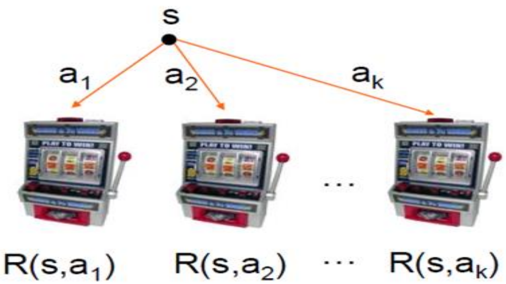
- David Silver, et.al., Mastering the game of Go with Deep Neural Networks and Tree Search, ***Nature***, 529:484-490,2016
- Cameron Browne, et.al., Survey of Monte Carlo Tree Search Methods, ***IEEE Transactions on Computational Intelligence and AI in Games***, 4(1):1-49,2012
- Sylvain Gelly, Levente Kocsis, Marc Schoenauer, et al., The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions, ***Communications of the ACM***, 55(3):106-113,2012
- Levente KocsisCsaba Szepesvari, Bandit Based Monte-Carlo Planning, ***ECML*** 2006
- Auer, P., Cesa-Bianchi, N., & Fischer, P. , Finite-time analysis of the multi-armed bandit problem, ***Machine learning***, 47(2), 235-256, 2002

蒙特卡洛规划 (Monte-Carlo Planning)

- 单一状态蒙特卡洛规划： 多臂赌博机 (multi-armed bandits)
- 上限置信区间策略 (Upper Confidence Bound Strategies, UCB)
- 蒙特卡洛树搜索 (Monte-Carlo Tree Search)
  - UCT (Upper Confidence Bounds on Trees)

单一状态蒙特卡洛规划： 多臂赌博机 (multi-armed bandits)

- 单一状态， k种行动（即有k 个摇臂）
- 在摇臂赌博机问题中，每次以随机采样形式采取一种行动a，好比随机拉动第k个赌博机的臂膀，得到$R(s, a_k)$ 的回报。
- 问题：下一次需要拉动那个赌博机的臂膀，才能获得最大回报呢？



多臂赌博机 (multi-armed bandits)

- 多臂赌博机问题是一种序列决策问题，这种问题需要在利用**(exploitation)**和探索(**exploration**) 之间保持平衡。

- 利用**(exploitation)**：保证在过去决策中得到最佳回报
- 探索(**exploration**)：寄希望在未来能够得到更大回报

多臂赌博机 (multi-armed bandits)

- 如果有k个赌博机，这k个赌博机产生的操作序列为$X_{i,1}, X_{i,2}, \ldots$ $(\mathrm{i}=1,\ldots,\mathrm{K})$。在时刻$t=1,2\ldots$，选择第$I_t$个赌博机后，可得到奖赏$X_{I_t,t}$，则在$n$次操作$I_1,\ldots,I_n$后，可如下定义悔值函数：

$$R_n = \max_{i=1,\ldots,k} \sum_{t=1}^{n} X_{i,t} - \sum_{t=1}^{n} X_{I_t,t}$$

- 悔值函数表示了如下意思：在第t次对赌博机操作时，假设知道哪个赌博机能够给出最大奖赏（虽然在现实生活中这是不存在的），则将得到的最大奖赏减去实际操作第I_t个赌博机所得到的奖赏。将n次操作的差值累加起来，就是悔值函数的结果。

- 很显然，一个良好的多臂赌博机操作的策略是在不同人进行了多次玩法后，能够让悔值函数的方差最小。

上限置信区间 (Upper Confidence Bound, UCB)

- 在多臂赌博机的研究过程中，上限置信区间（Upper Confidence Bound, UCB）成为一种较为成功的策略学习方法，因为其在探索-利用（exploration-exploitation）之间取得平衡。

- 在UCB方法中，使 $X_{i,T_i(t-1)}$ 来记录第 i 个赌博机在过去 $t-1$ 时刻内的平均奖赏，则在第t时刻，选择使如下具有最佳上限置区间的赌博机：

$$I_t = \max_{i \in \{1,\dots,k\}} \{\overline{X_{i,T_i(t-1)}} + c_{t-1,T_i(t-1)}\}$$

其中$c_{t,s}$取值定义如下：

$$c_{t,s} = \sqrt{\frac{2Int}{s}}$$

$T_i(t) = \sum_{s=1}^{t} \prod(I_s = i)$为在过去时刻（初始时刻到$t$时刻）过程中选择第$i$个赌博机的次数总和。

上限置信区间 (Upper Confidence Bound, UCB)

也就是说，在第t时刻，UCB算法一般会选择具有如下最大值的第j个赌博机：

$$UCB = \bar{X}_j + \sqrt{\frac{2In\, n}{n_j}} \text{ 或者 } UCB = \bar{X}_j + C \times \sqrt{\frac{2In\, n}{n_j}}$$

$\bar{X}_j$是第j个赌博机在过去时间内所获得的平均奖赏值，n_j是在过去时间内拉动第j个赌博机臂膀的总次数，n是过去时间内拉动所有赌博机臂膀的总次数。C 是一个平衡因子，其决定着在选择时偏重探索还是利用。

从这里可看出UCB算法如何在探索-利用（exploration-exploitation）之间寻找平衡：既需要拉动在过去时间内获得最大平均奖赏的赌博机，又希望去选择那些拉动臂膀次数最少的赌博机。

上限置信区间 (Upper Confidence Bound, UCB)

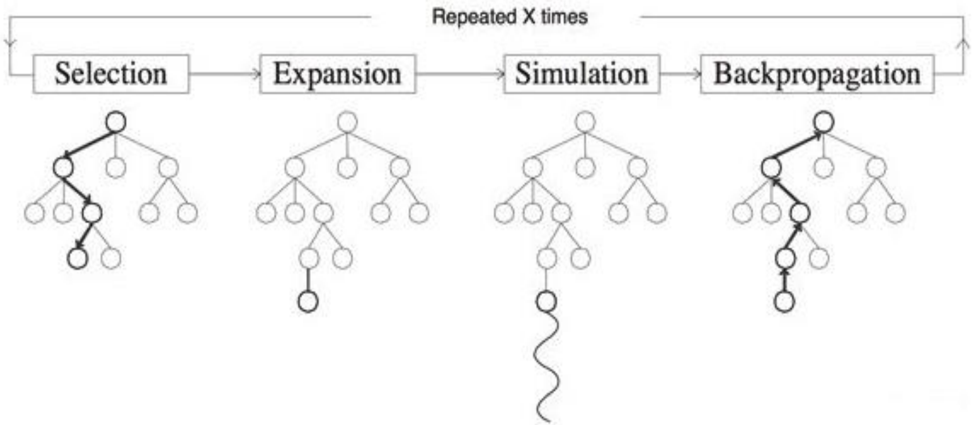
- UCB算法描述

Deterministic policy: UCB1.
Initialization: Play each machine once.
Loop:

– Play machine j that maximizes $\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}$, where $\bar{x}_j$ is the average reward obtained from machine j, n_j is the number of times machine j has been played so far, and n is the overall number of plays done so far.

蒙特卡洛树搜索

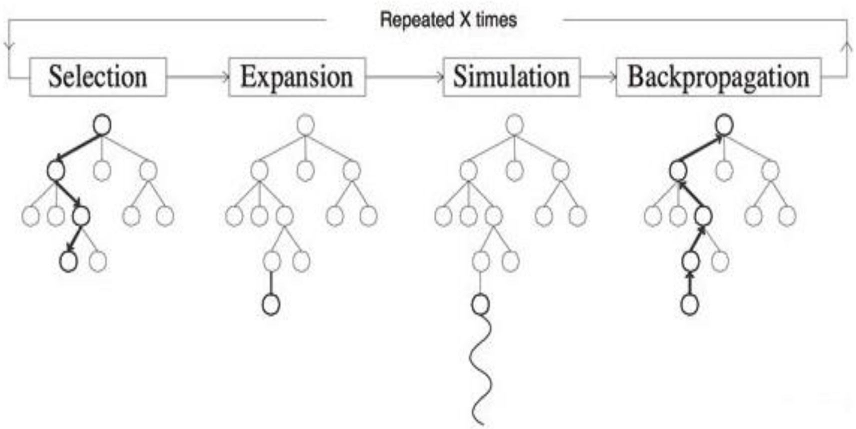
- 将上限置信区间算法UCB应用于游戏树的搜索方法，由Kocsis和Szepesvari在2006年提出
- 包括了四个步骤：选举(selection)，扩展(expansion)，模拟(simulation)，反向传播(Back-Propagation)



L. Kocsis and C. Szepesvari, Bandit based Monte-Carlo Planning, ***ECML***, 2006:282–293

蒙特卡洛树搜索

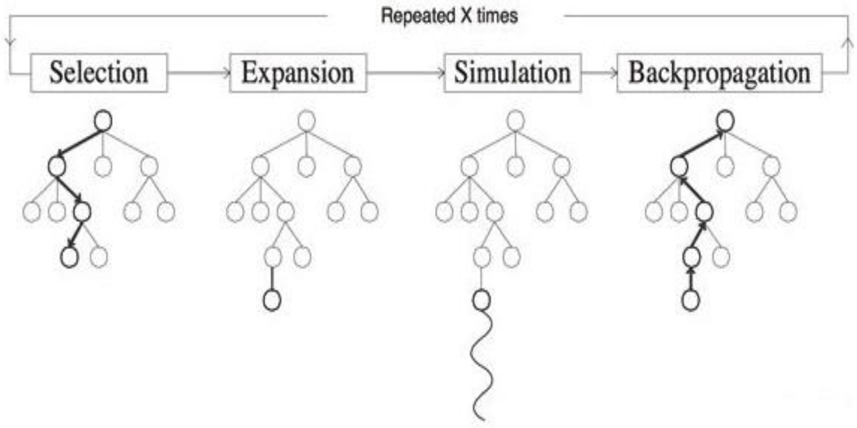
- **选择：**
  - 从根节点 R 开始，向下递归选择子节点，直至选择一个叶子节点L。
  - 具体来说，通常用UCB1（Upper Confidence Bound，上限置信区间）选择最具“潜力”的后续节点



$$UCB = \bar{X}_j + \sqrt{\frac{2In\ n}{n_j}}$$

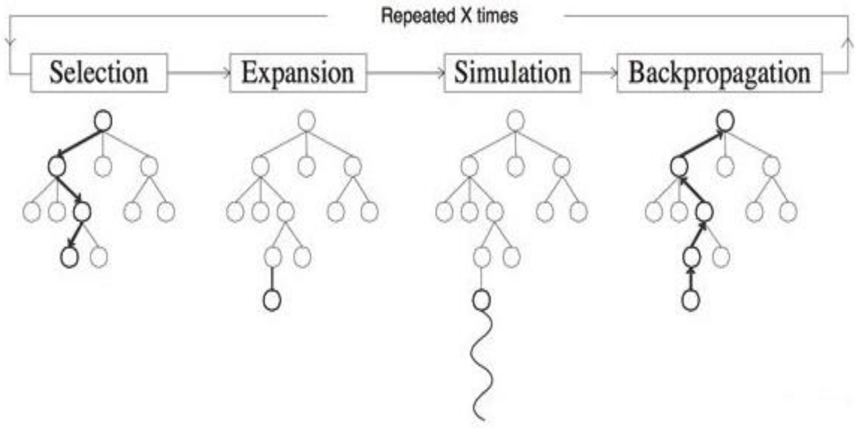
蒙特卡洛树搜索

- **扩展：**
  - 如果 L 不是一个终止节点（即博弈游戏不），则随机创建其后的一个未被访问节点，选择该节点作为后续子节点C。



蒙特卡洛树搜索

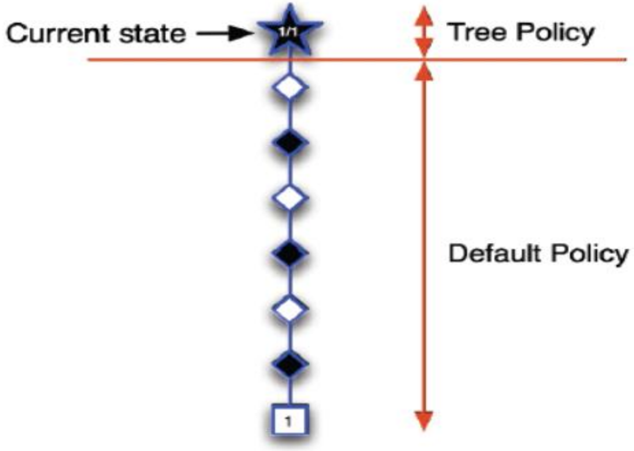
- **模拟：**
  - 从节点 C出发，对游戏进行模拟，直到博弈游戏结束。
- **反向传播**
  - 用模拟所得结果来回溯更新导致这个结果的每个节点中获胜次数和访问次数。



蒙特卡洛树搜索

两种策略学习机制：

- 搜索树策略：从已有的搜索树中选择或创建一个叶子结点（即蒙特卡洛中选择和拓展两个步骤）.搜索树策略需要在利用和探索之间保持平衡。
- 模拟策略：从非叶子结点出发模拟游戏，得到游戏仿真结果。

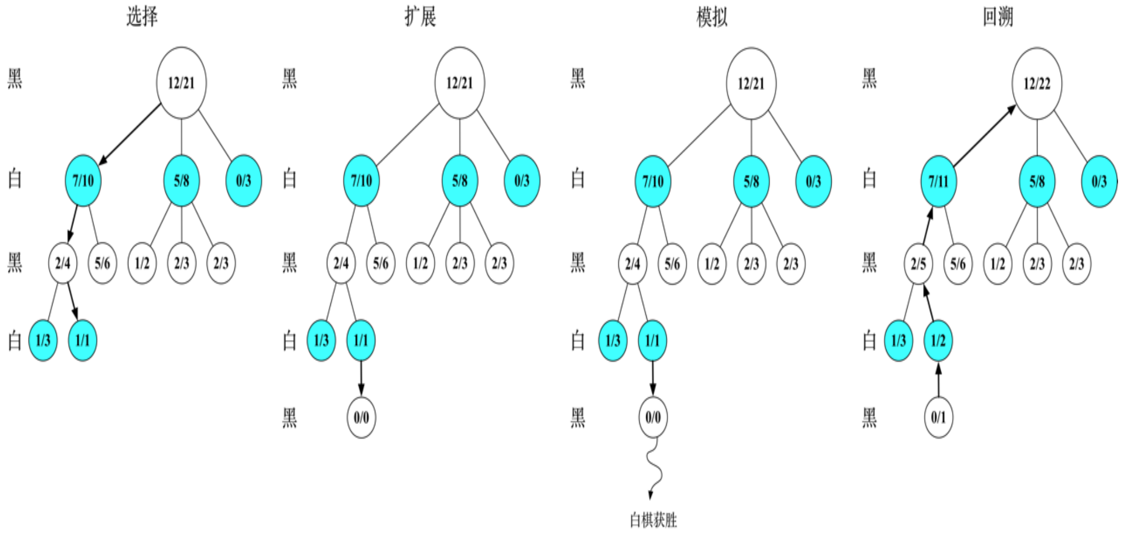


蒙特卡洛树搜索：例子

- 以围棋为例，假设根节点是执黑棋方。
- 图中每一个节点都代表一个局面，每一个局面记录两个值A/B：

A：该局面被访问中黑棋胜利次数。对于黑棋表示己方胜利次数，对于白棋表示己方失败次数（对方胜利次数）；

B：该局面被访问的总次数。



蒙特卡洛树搜索：例子

该图刻画了蒙特卡洛树搜索四个步骤，假设根结点由黑棋行棋，为了选择根节点后续节点，需要由UCB1公式来计算根节点后续节点如下值，取一个值最大的节点作为后续节点：

左1：7/10对应的局面奖赏值为

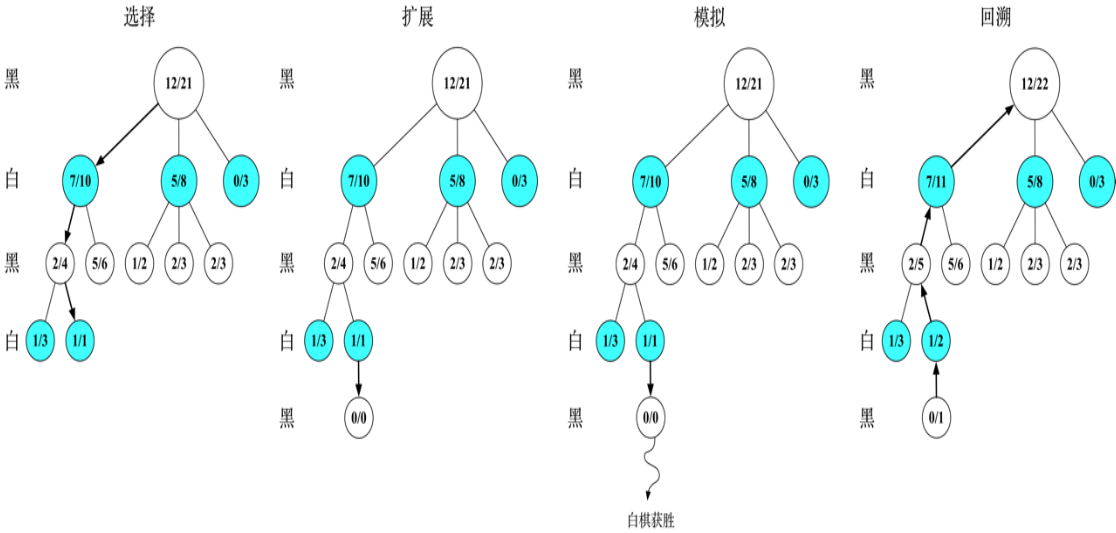
$$\frac{7}{10}+\sqrt{\frac{\log(21)}{10}}=1.252$$

左2，5/8对应的局面奖赏值为$\frac{5}{8}+\sqrt{\frac{\log(21)}{8}}=1.243$

左3，0/3对应的局面评估分数为

$$\frac{0}{3}+\sqrt{\frac{\log(21)}{3}}=1.007$$

由此可见，黑棋会选择局面7/10进行行棋。



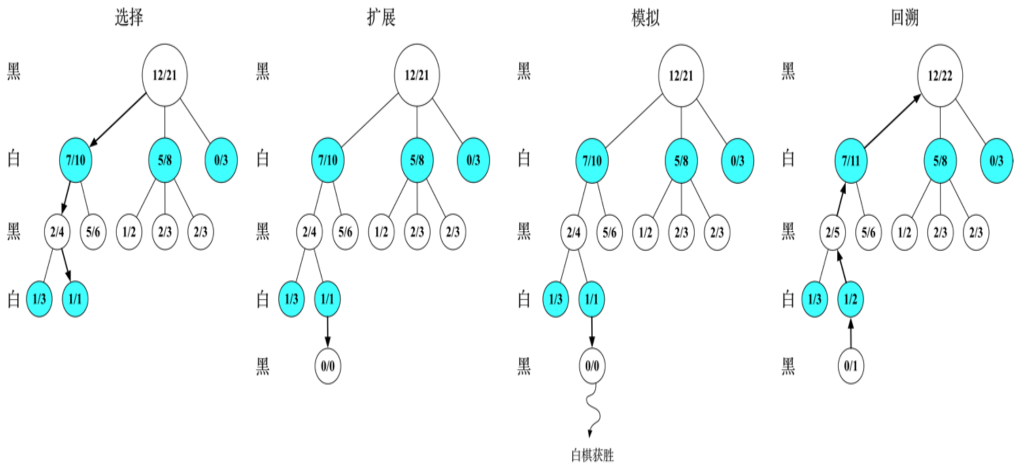
蒙特卡洛树搜索：例子

在节点7/10，由白棋行棋，评估该节点下面的两个局面，由UCB1公式可得（注意：此时A记录的的是白棋失败的次数，所以第一项为1-A/B）：

左1，2/4对应的局面奖赏为 $1-\frac{2}{4}+\sqrt{\frac{\log(10)}{4}}=1.26$

左2，5/6对应的局面奖赏为 $1-\frac{5}{6}+\sqrt{\frac{\log(10)}{6}}=0.786$

由此可见，白棋会选择局面2/4进行行棋。



蒙特卡洛树搜索：例子

在节点2/4，黑棋评估下面的两个局面，

由UCB1公式可得：

左1，1/3对应的局面奖赏为$\frac{1}{3}+\sqrt{\frac{\log(4)}{3}}=$

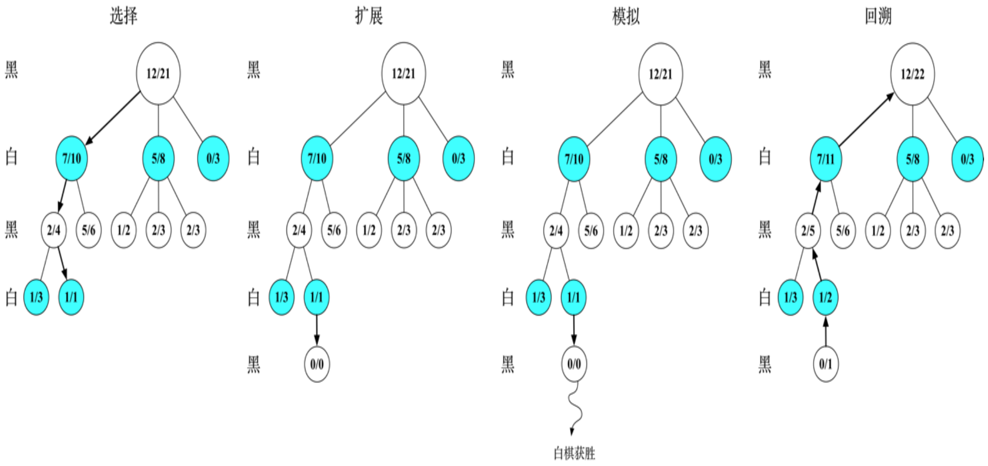
1.01

左1，1/1对应的局面奖赏$\frac{1}{1}+\sqrt{\frac{\log(4)}{1}}=$

2.18

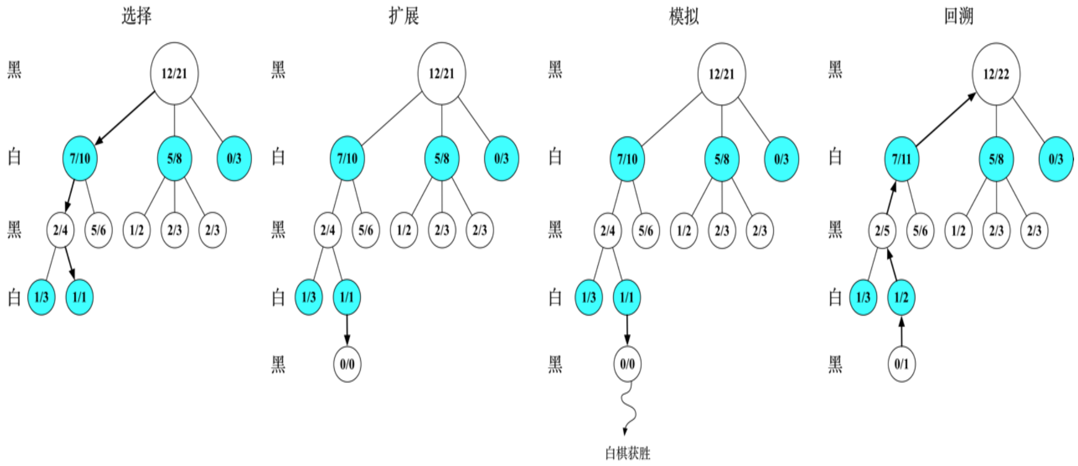
由此可见，黑棋会选择局面1/1进行行棋。

此时已经到达叶子结点，需要进行扩展。



蒙特卡洛树搜索：例子

随机扩展一个新节点。由于该新节点未被访问，所以初始化为0/0，接着在该节点下进行模拟。假设经过一系列仿真行棋后，最终白棋获胜。根据仿真结果来更新该仿真路径上每个节点的A/B值，该新节点的A/B值被更新为0/1，并向上回溯到该仿真路径上新节点的所有父辈节点，即所有父辈节点的A不变，B值加1。



使用蒙特卡洛树搜索的原因

- **Monte-Carlo Tree Search** (MCTS)：蒙特卡洛树搜索基于采样来得到结果、而非穷尽式枚举（虽然在枚举过程中也可剪掉若干不影响结果的分支）。

蒙特卡洛树搜索算法 (Upper Confidence Bounds on Trees , UCT)

$\boldsymbol{S}$	状态集
$\boldsymbol{A(s)}$	在状态s能够采取的有效行动的集合
$\boldsymbol{s(v)}$	节点v所代表的状态
$\boldsymbol{a(v)}$	所采取的行动导致到达节点v
$\boldsymbol{f}$: $S \times \mathrm{A} \rightarrow S$	状态转移函数
$\boldsymbol{N(v)}$	节点v被访问的次数
$\boldsymbol{Q(V)}$	节点v所获得的奖赏值
$\boldsymbol{\Delta(v,p)}$	玩家p选择节点v所得到的奖赏值

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f : S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值


```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f : S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

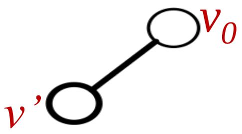
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget **do**
        $v_l \leftarrow$ TREEPOLICY(v_0)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP(v_l, Δ)
    return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
    while v is nonterminal **do**
        if v not fully expanded **then**
            return EXPAND(v)
        else
            $v \leftarrow$ BESTCHILD(v, Cp)
    return v

function DEFAULTPOLICY(s)
    while s is non-terminal **do**
        choose $a \in A(s)$ uniformly at random
        $s \leftarrow f(s, a)$
    return reward for state s

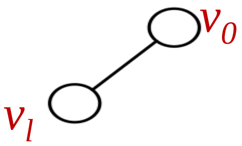
function BACKUP(v, Δ)
    while v is not null **do**
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of v

function EXPAND(v)
    choose $a \in$ untried actions from $A(s(v))$
    add a new child v' to v
        with $s(v') = f(s(v), a)$
        and $a(v') = a$
    return v'

function BESTCHILD(v, c)
    return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD(v_0, 0))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s,a)$
  return reward for state s

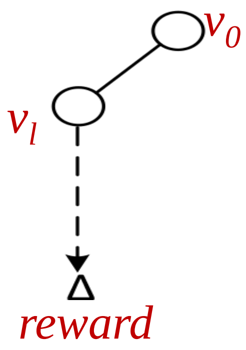
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v,p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \arg\max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S\times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

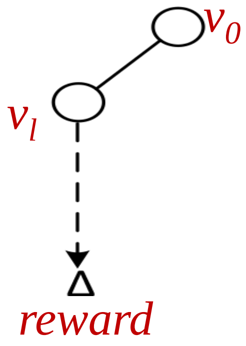
function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

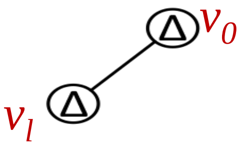
function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$



蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S\times A\rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

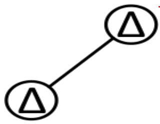
```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and  a(v') = a
    return v'
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v'\in\text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

Algorithm 3 UCT backup for two players

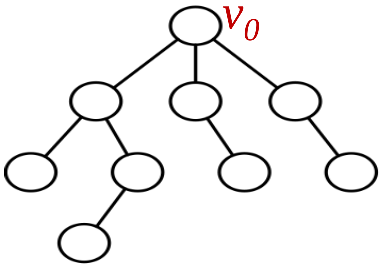
function BACKUPNEGAMAX(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta$
    $\Delta \leftarrow -\Delta$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

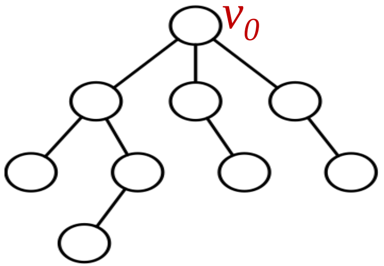
```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \mathop{\arg\max}_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

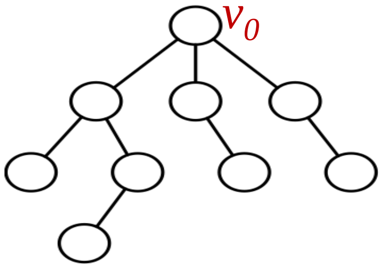
```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v′ to v
        with s(v′) = f(s(v), a)
        and a(v′) = a
    return v′
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

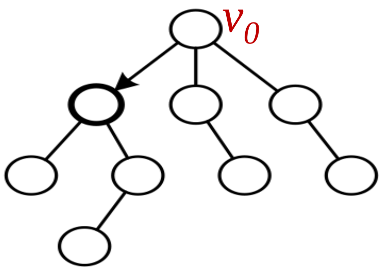
```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function $\textsc{UctSearch}(s_0)$
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow \textsc{TreePolicy}(v_0)$
    $\Delta \leftarrow \textsc{DefaultPolicy}(s(v_l))$
    $\textsc{Backup}(v_l, \Delta)$
  return $a(\textsc{BestChild}(v_0, 0))$

function $\textsc{TreePolicy}(v)$
  while v is nonterminal **do**
    if v not fully expanded **then**
      return $\textsc{Expand}(v)$
    else
      $v \leftarrow \textsc{BestChild}(v, Cp)$
  return v

function $\textsc{DefaultPolicy}(s)$
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

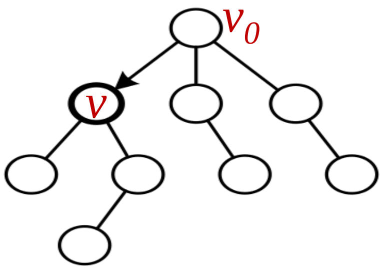
function $\textsc{Backup}(v, \Delta)$
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function $\textsc{Expand}(v)$
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function $\textsc{BestChild}(v, c)$
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f : S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

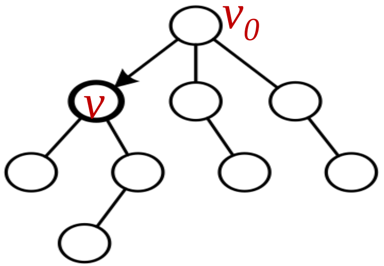
function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)

$$\textbf{return} \quad \mathop{\arg\max}_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

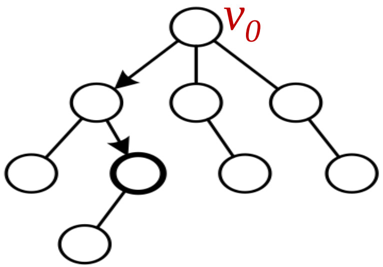
```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
```

function BESTCHILD(v, c)

return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

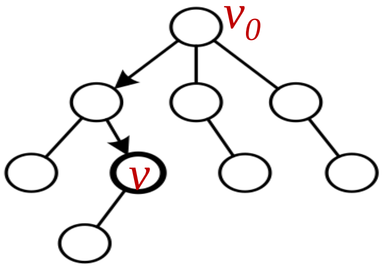
function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S\times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s0)
    create root node v0 with state s0
    while within computational budget do
        vl ← TREEPOLICY(v0)
        Δ ← DEFAULTPOLICY(s(vl))
        BACKUP(vl, Δ)
    return a(BESTCHILD(v0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

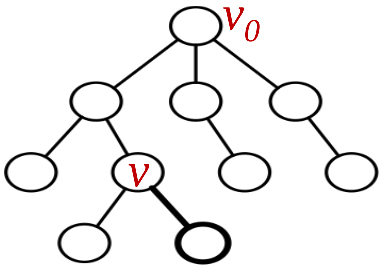
```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v′ to v
        with s(v′) = f(s(v), a)
        and a(v′) = a
    return v′
```

function BESTCHILD(v, c)

return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

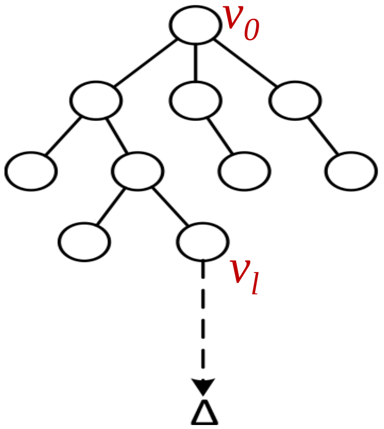
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

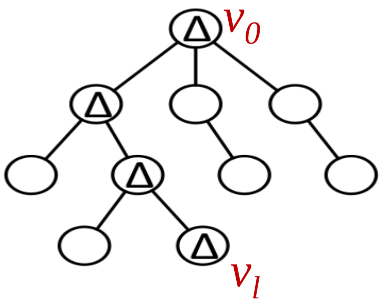
function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)

$$\textbf{return} \quad \mathop{\arg\max}_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \to S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

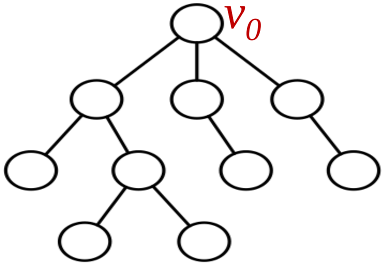
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \dfrac{Q(v')}{N(v')} + c\sqrt{\dfrac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S \times A \rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值



function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

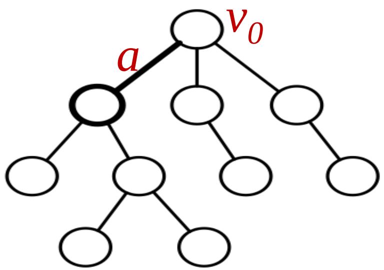
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

蒙特卡洛树搜索算法(UCT)

S	状态集
$A(s)$	在状态s能够采取的有效行动的集合
$s(v)$	节点v所代表的状态
$a(v)$	所采取的行动导致到达节点v
$f: S\times A\rightarrow S$	状态转移函数
$N(v)$	节点v被访问的次数
$Q(V)$	节点v所获得的奖赏值
$\Delta(v,p)$	玩家p选择节点v所得到的奖赏值




```
function UCTSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0, 0))
```

```
function TREEPOLICY(v)
    while v is nonterminal do
        if v not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD(v, Cp)
    return v
```

```
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    return reward for state s
```

```
function BACKUP(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ(v, p)
        v ← parent of v
```

```
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add a new child v' to v
        with s(v') = f(s(v), a)
        and a(v') = a
    return v'
```

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

AlphaGo算法解读

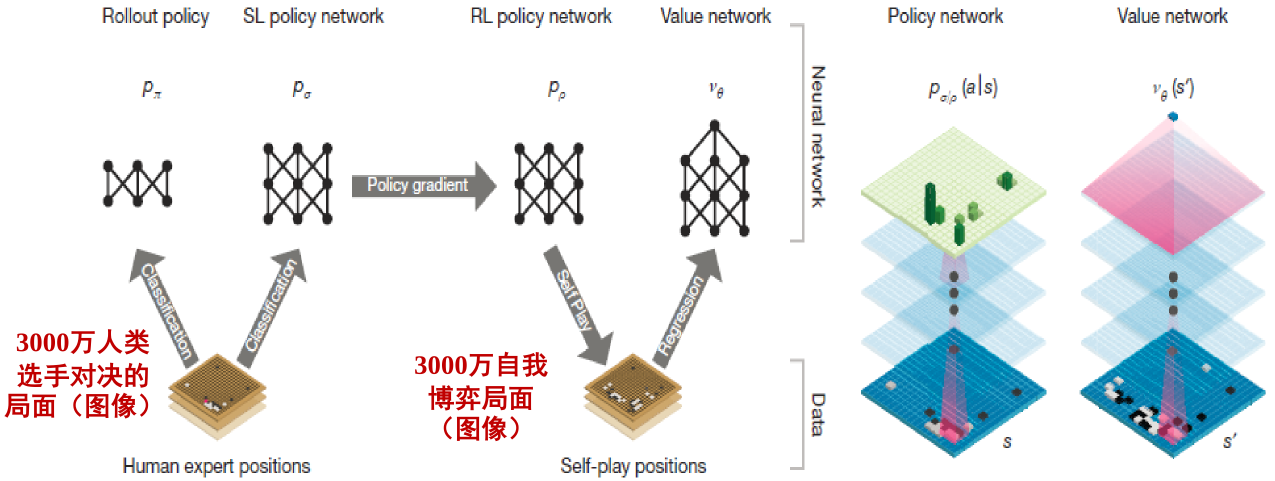


Figure 1 | Neural network training pipeline and architecture. **a**, A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set. **b**, Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, represented by a probability map over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.

- 将每个状态（局面）均视为一幅图像
- 训练策略（*policy*）网络和价值（*value*）网络
- $p_{\sigma,p}(a|s)$表示当前状态为s（局面）时，采取行动a后所得到的概率；$v_\theta(s')$表示当前状态为s'时，整盘棋获胜的概率。

David Silver, et.al., Mastering the game of Go with Deep Neural Networks and Tree Search, ***Nature***, 529:484-490,2016

AlphaGo算法解读：策略网络的训练

- **基于监督学习来先训练策略网络**
  - Idea: perform *supervised learning* (SL) to predict human moves
  - Given state s, predict probability distribution over moves a, $p_{\sigma,p}(a|s)$
  - Trained on 30M positions, 57% accuracy on predicting human moves
  - Also train a smaller, faster *rollout policy* network (24% accurate)
- **再基于强化学习来训练策略网络**
  - Idea: fine-tune policy network using *reinforcement learning* (RL)
  - Initialize RL network to SL network
  - Play two snapshots of the network against each other, update parameters to maximize expected final outcome
  - RL network wins against SL network 80% of the time, wins against open-source Pachi Go program 85% of the time

AlphaGo算法解读：价值网络的训练

- Value network
  - Idea: train network for position evaluation
  - Given state s', estimate $v_\theta(s')$, expected outcome of play starting with position s and following the learned policy for both players
  - Train network by minimizing mean squared error between actual and predicted outcome
  - Trained on 30M positions sampled from different self-play games

AlphaGo算法解读：蒙特卡洛树搜索中融入了策略网络和价值网络

在通过深度学习得到的策略网络和价值网络帮助之下，如下完成棋局局面的选择和搜索。给定节点v_0，将具有如下最大值的节点v选择作为v_0的后续节点

$$\frac{Q(v)}{N(v)}+\frac{P(v|v_0)}{1+N(v)}$$

- 这里$P(v|v_0)$的值由策略网络计算得到。
- 在模拟策略阶段(default policy)，AlphaGo不仅考虑仿真结果，而且考虑价值网络计算结果。
- 策略网络和价值网络是离线训练得到的。

AlphaGo算法解读：蒙特卡洛树搜索中融入了策略网络和价值网络

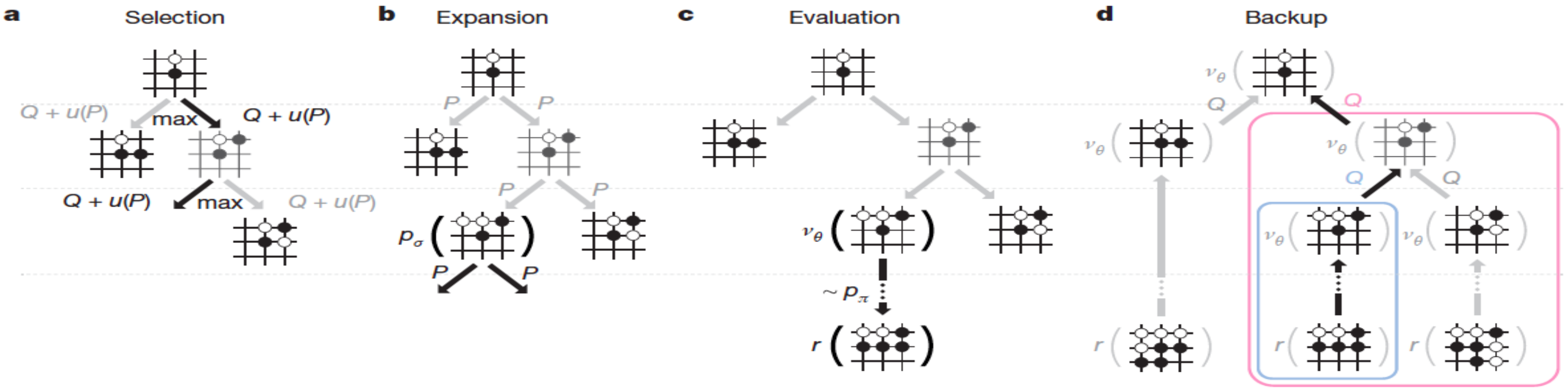


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t,a)+u(s_t,a))$$

$$N(s,a)=\sum_{i=1}^{n} 1(s,a,i)$$

$$Q(s,a)=\frac{1}{N(s,a)}\sum_{i=1}^{n} 1(s,a,i)V(s_L^i)$$

价值网络计算

策略网络计算

$$u(s,a)\propto\frac{P(s,a)}{1+N(s,a)}$$

AlphaGo算法解读

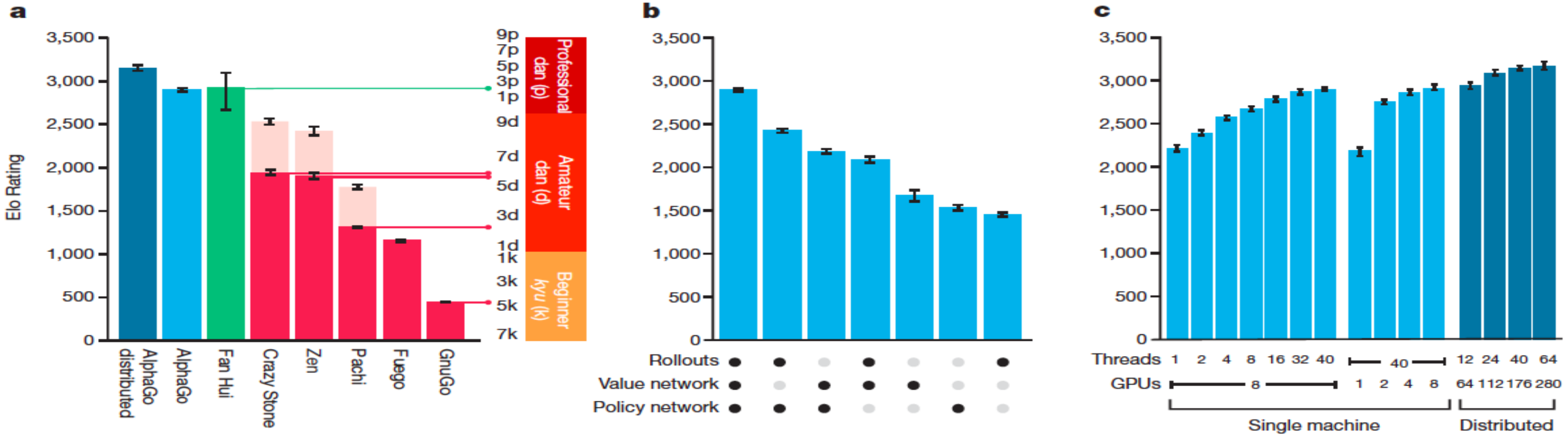
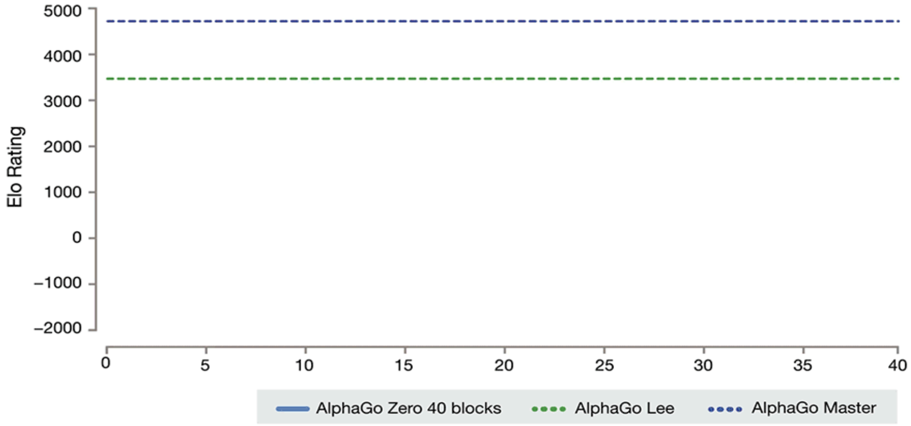


Figure 4 | Tournament evaluation of AlphaGo. a, Results of a tournament between different Go programs (see Extended Data Tables 6–11). Each program used approximately 5 s computation time per move. To provide a greater challenge to AlphaGo, some programs (pale upper bars) were given four handicap stones (that is, free moves at the start of every game) against all opponents. Programs were evaluated on an Elo scale[37]: a 230 point gap corresponds to a 79% probability of winning, which roughly corresponds to one amateur *dan* rank advantage on KGS[38]; an approximate correspondence to human ranks is also shown, horizontal lines show KGS ranks achieved online by that program. Games against the human European champion Fan Hui were also included; these games used longer time controls. 95% confidence intervals are shown. **b**, Performance of AlphaGo, on a single machine, for different combinations of components. The version solely using the policy network does not perform any search. **c**, Scalability study of MCTS in AlphaGo with search threads and GPUs, using asynchronous search (light blue) or distributed search (dark blue), for 2 s per move.

AlphaGo算法解读：AlphaGo Zero （一张白纸绘蓝图）



- 不需要人类选手对决的棋面进行训练
- 策略网络和价值网络合并
- 深度残差网络

经过40天训练后，Zero总计运行约2900万次自我对弈，得以击败AlphaGo Master，比分为89比11

Mastering the game of Go without human knowledge, ***Nature***, volume 550, pages 354–359 (19 October 2017)