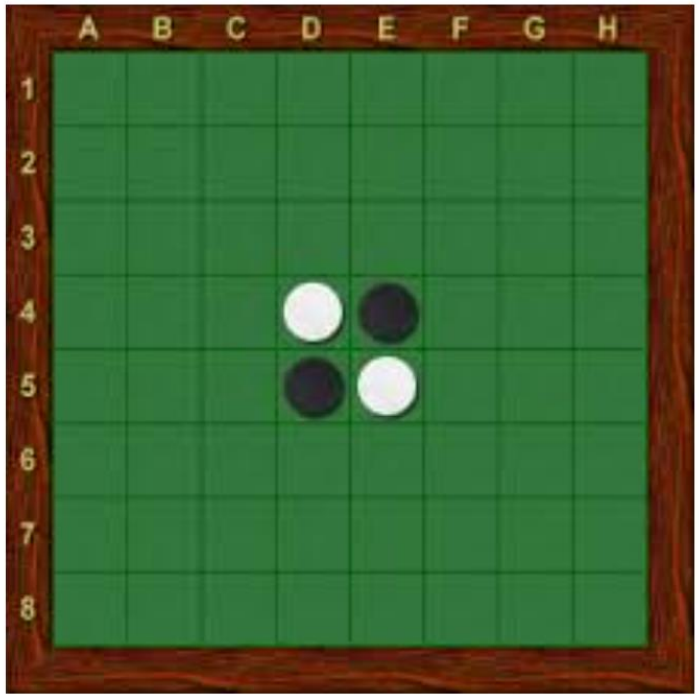
蒙特卡洛树搜索之黑白棋对弈

人工智能基础 —— 实践课（二）



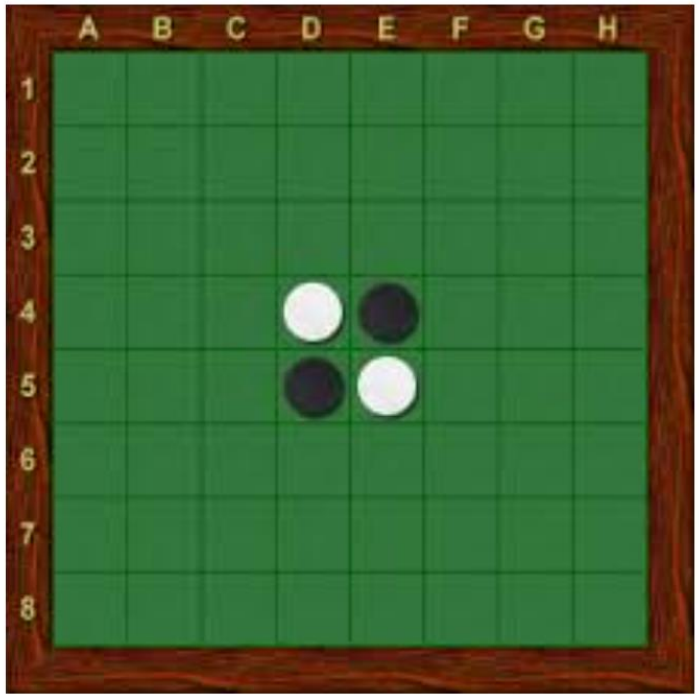
黑白棋 (Reversi)，又称翻转棋，是一个经典的策略性游戏。

一般棋子双面为黑白两色，故称“黑白棋”。因为行棋之时将对方棋子翻转，则变为己方棋子，故又称“翻转棋” (Reversi) 。

黑白棋使用 8x8 大小的棋盘，由两人执黑子和白子轮流下棋，最后子多方为胜方。

蒙特卡洛树搜索之黑白棋对弈

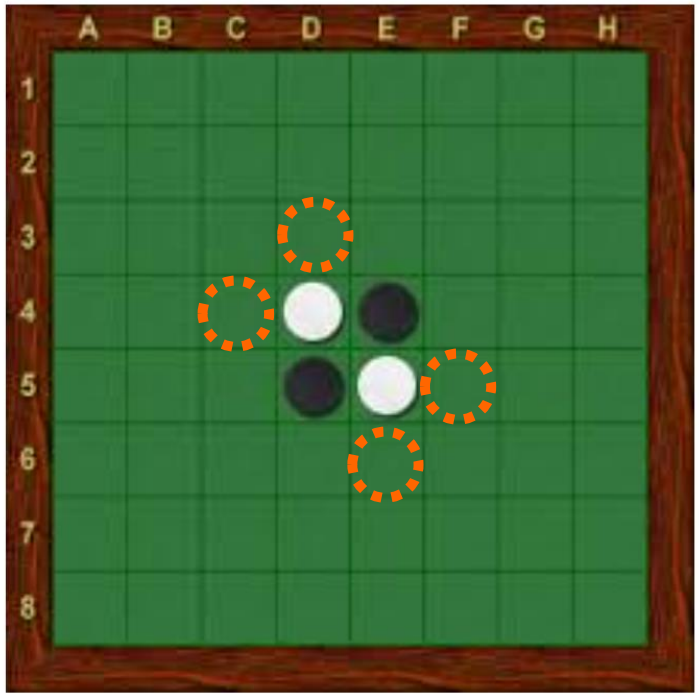
人工智能基础 —— 实践课（二）



1. 黑方先行，双方交替下棋。
2. 一步合法的棋步包括：
   - 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
   - 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；
   - 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
3. 如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
4. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
5. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
6. 如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。

蒙特卡洛树搜索之黑白棋对弈

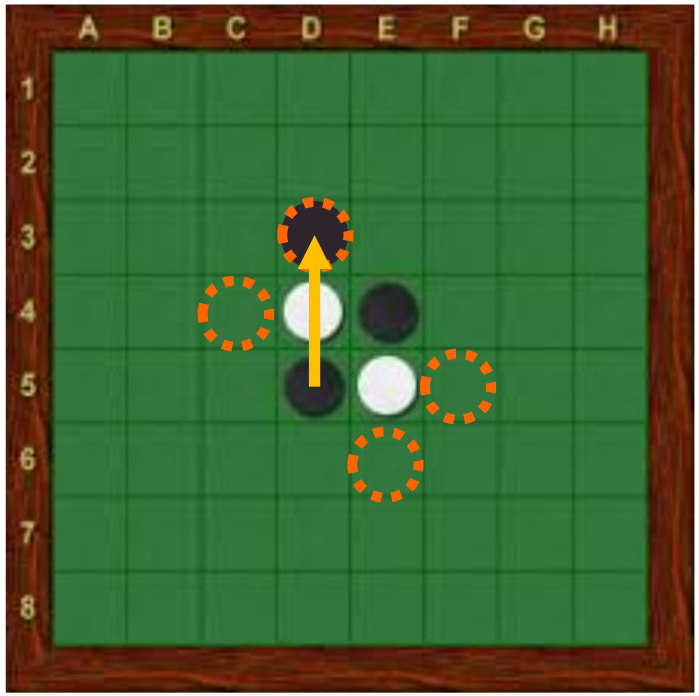
人工智能基础 —— 实践课（二）



1. 黑方先行，双方交替下棋。
2. 一步合法的棋步包括：
   - 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
   - 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；
   - 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
3. 如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
4. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
5. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
6. 如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。

蒙特卡洛树搜索之黑白棋对弈

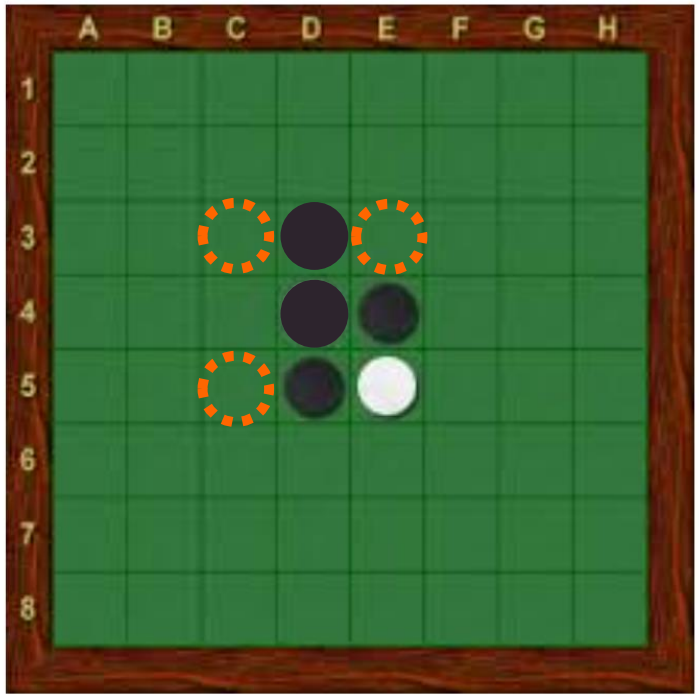
人工智能基础 —— 实践课（二）



1. 黑方先行，双方交替下棋。
2. 一步合法的棋步包括：
   - 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
   - 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；
   - 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
3. 如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
4. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
5. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
6. 如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。

蒙特卡洛树搜索之黑白棋对弈

人工智能基础 —— 实践课（二）



1. 黑方先行，双方交替下棋。
2. 一步合法的棋步包括：
   - 在一个空格处落下一个棋子，并且翻转对手一个或多个棋子；
   - 新落下的棋子必须落在可夹住对方棋子的位置上，对方被夹住的所有棋子都要翻转过来，可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格；
   - 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
3. 如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
4. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
5. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
6. 如果某一方落子时间超过 1 分钟 或者 连续落子 3 次不合法，则判该方失败。

Board(棋盘类)

function	说明
_move	落子并获取反转棋子的坐标
_can_fliped	检测落子是否合法,如果不合法，返回 False，否则返回反转子的坐标列表
display	打印棋盘
count	统计 color 一方棋子的数量。(O:白棋, X:黑棋, .:未落子状态)
get_winner	判断黑棋和白旗的输赢，通过棋子的个数进行判断
is_on_board	判断坐标是否出界
get_legal_actions	按照黑白棋的规则获取棋子的合法走法

蒙特卡洛树搜索算法

选择（selection）

选择指算法从搜索树的根节点开始，向下递归选择子节点，直至到达叶子节点或者到达具有还未被扩展过的子节点的节点L。这个向下递归选择过程可由UCB算法来实现在递归选择过程中记录下每个节点被选择次数和每个节点得到的奖励均值。

扩展（expansion）

如果节点 L 不是一个终止节点（或对抗搜索的终局节点），则随机扩展它的一个未被扩展过的后继边缘节点M。

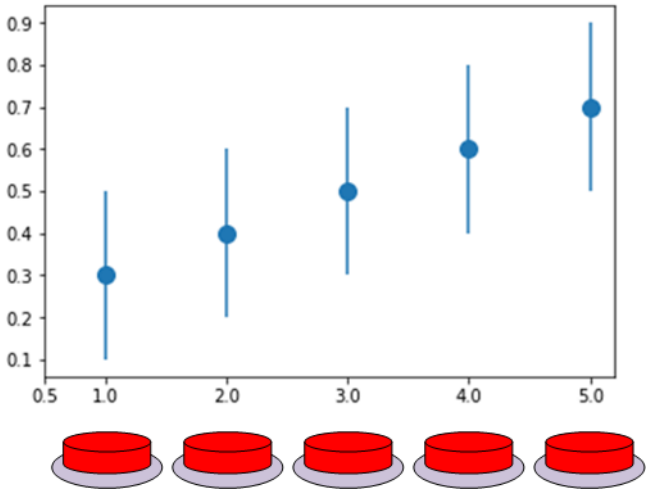
模拟（simulation）

从节点M出发，模拟扩展搜索树，直到找到一个终止节点。模拟过程使用的策略和采用UCB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（终止节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）节点的奖励均值和被访问次数。

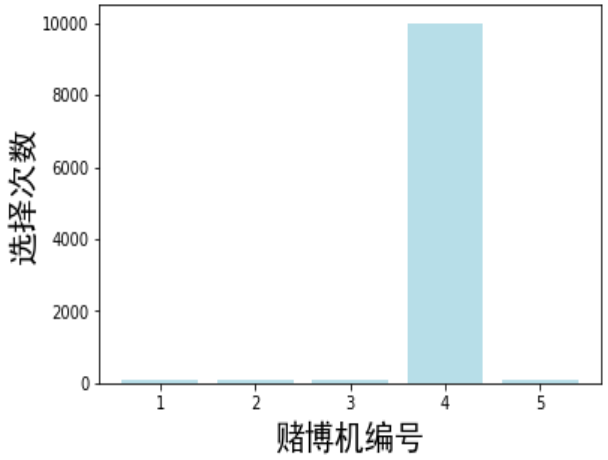
UCB

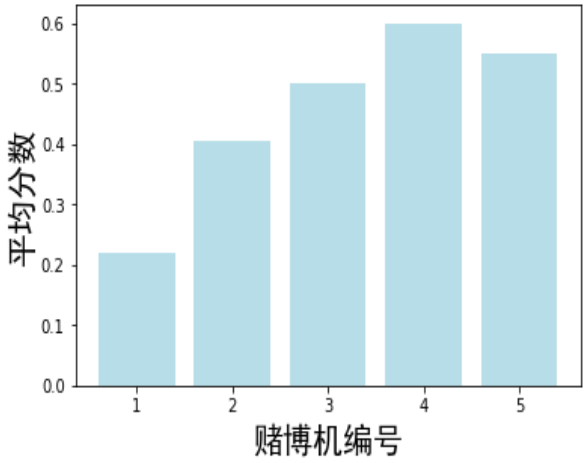


Bandit – 多臂赌博机问题

(0.3, 0.4, 0.5, 0.6, 0.7)

重新进行一个10000次贪心算法的模拟





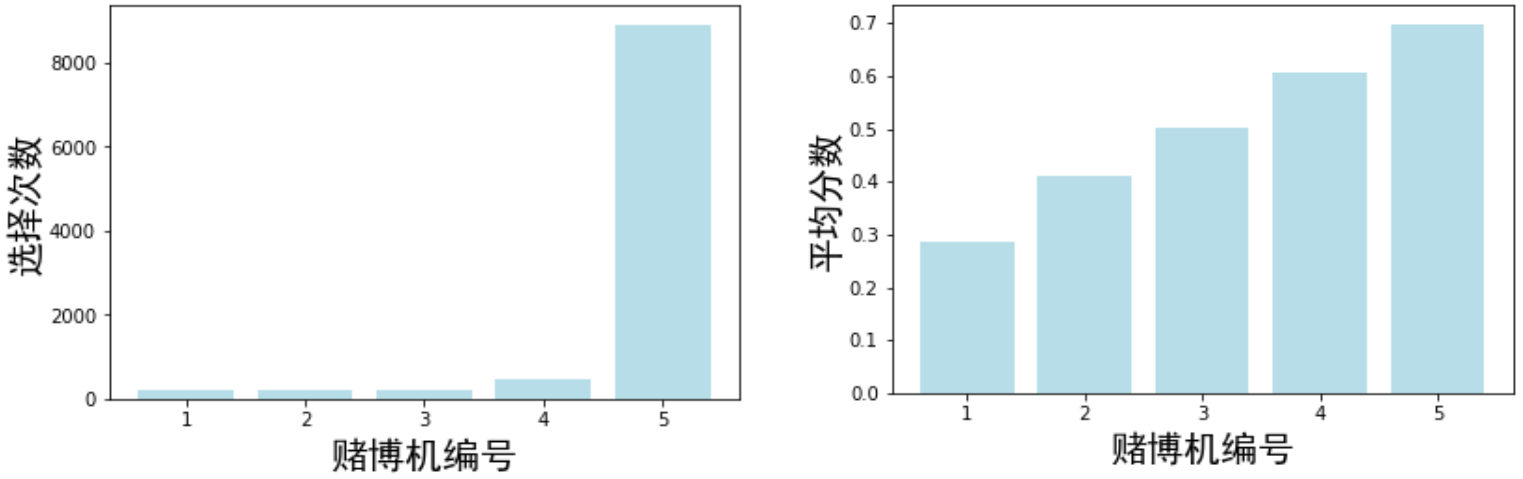
UCB

$T_{(i,t-1)}$：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。

ϵ - 贪心算法就是这样一种在探索与利用之间进行平衡的搜索算法。在第t步，ϵ - 贪心算法按照如下机制来选择摇动赌博机：

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

进行10000次ϵ-贪心算法的实验结果

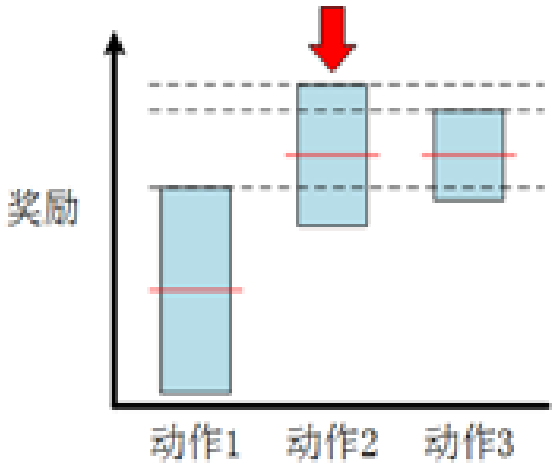


与被探索的次数无关。可能存在一个给出更好奖励期望的动作，但因为智能体对其探索次数少而认为其期望奖励小。因此，需要对那些探索次数少或几乎没有被探索过的动作赋予更高的优先级。

UCB

霍夫丁不等式

$$P\left(\mu_i - \overline{x}_{i,T_{(i,t-1)}} > \delta\right) \leq e^{-2T_{(i,t-1)}\delta^2}$$

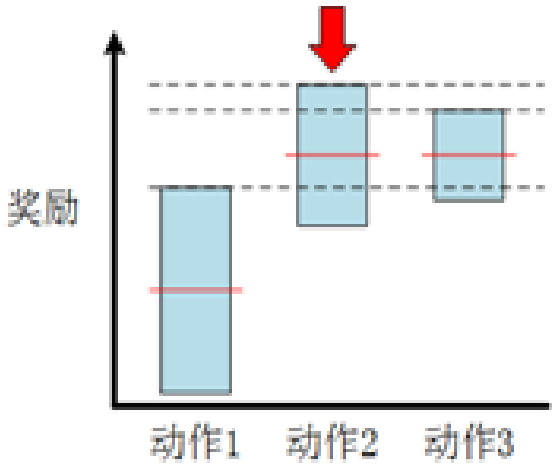


这里可找到一个随时间增长快速趋近于0的函数t^{-4}： $e^{-2T_{(i,t-1)}\delta^2} = t^{-4}$

UCB

霍夫丁不等式

$$P\left(\mu_i - \bar{x}_{i,T_{(i,t-1)}} > \delta\right) \le e^{-2T_{(i,t-1)}\delta^2}$$



这里可找到一个随时间增长快速趋近于0的函数t^{-4}：$e^{-2T_{(i,t-1)}\delta^2} = t^{-4}$

$$\Rightarrow \quad \delta = \sqrt{\frac{2\ln t}{T_{(i,t-1)}}} \quad \Rightarrow \quad \bar{x}_{i,T_{(i,t-1)}} + \sqrt{\frac{2\ln t}{T_{(i,t-1)}}}$$

UCB: $$l_t = \text{argmax}_i \, \bar{x}_{i,T_{(i,t-1)}} + C\sqrt{\frac{2\ln t}{T_{(i,t-1)}}}$$

UCB

$T_{(i,t-1)}$：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。

函数：UCB( )
输入：在过去 $t-1$ 次行动中第 i 个状态累计获得的总回报，Q_i； 在过去 $t-1$ 次行动中第 i 个状态总共被选中的次数，N_i； 总行动次数，t；超参数，C.
输出：第 i 个状态的UCB值
$\mathrm{UCB}=\frac{Q_i}{N_i}+C\sqrt{\frac{2\ln t}{N_i}}$

UCB: $l_t=\mathrm{argmax}_i\,\bar{x}_{i,T_{(i,t-1)}}+C\sqrt{\frac{2\ln t}{T_{(i,t-1)}}}$

蒙特卡洛树搜索算法

选择（selection）

选择指算法从搜索树的根节点开始，向下递归选择子节点，直至到达叶子节点或者到达具有还未被扩展过的子节点的节点L。这个向下递归选择过程可由UCB算法来实现在递归选择过程中记录下每个节点被选择次数和每个节点得到的奖励均值。

扩展（expansion）

如果节点 L 不是一个终止节点（或对抗搜索的终局节点），则随机扩展它的一个未被扩展过的后继边缘节点M。

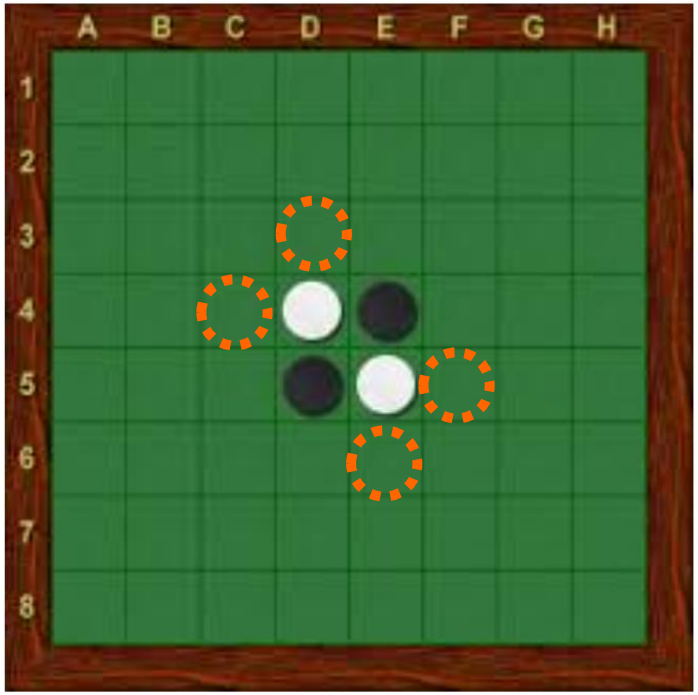
模拟（simulation）

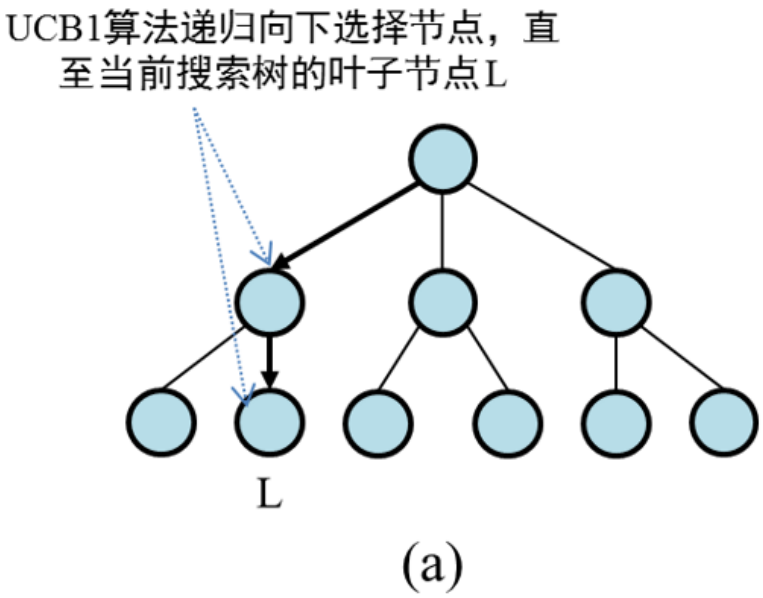
从节点M出发，模拟扩展搜索树，直到找到一个终止节点。模拟过程使用的策略和采用UCB算法实现的选择过程并不相同，前者通常会使用比较简单的策略，例如使用随机策略。

反向传播（Back Propagation）

用模拟所得结果（终止节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）节点的奖励均值和被访问次数。

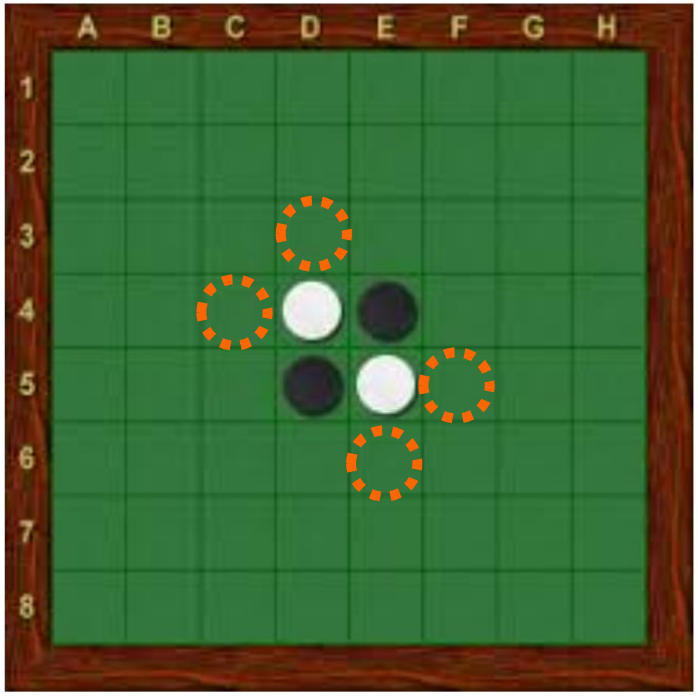
选择（selection）

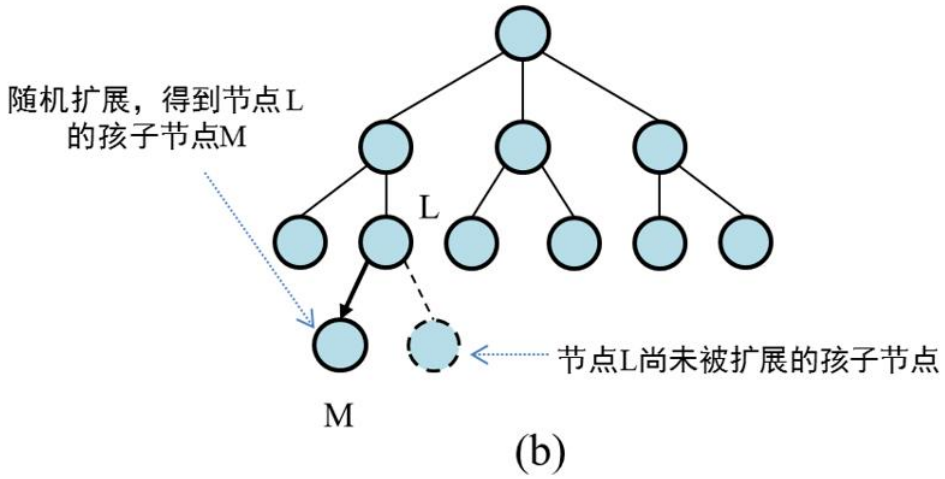




选择指算法从搜索树的根节点开始，向下递归选择子节点，直至到达叶子节点或者到达具有还未被扩展过的子节点的节点L。这个向下递归选择过程可由UCB算法来实现，在递归选择过程中记录下每个节点被选择次数和每个节点得到的奖励均值。

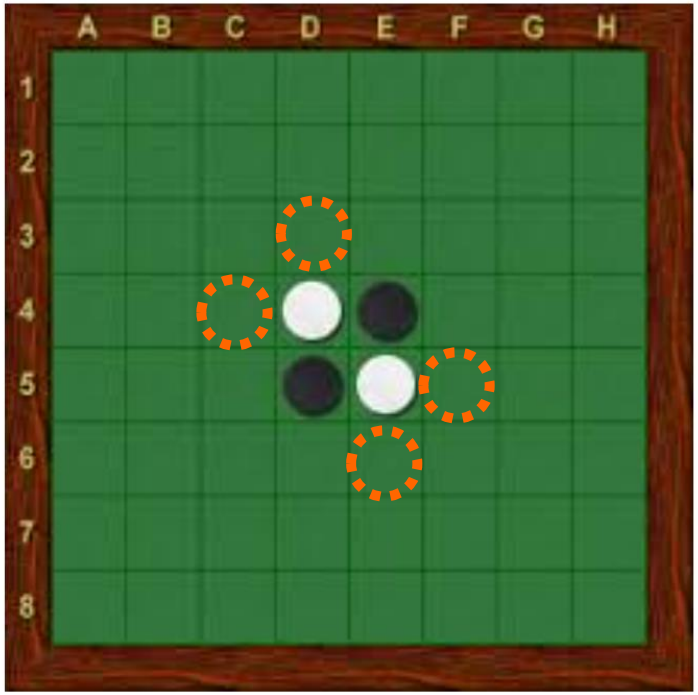
扩展（expansion）

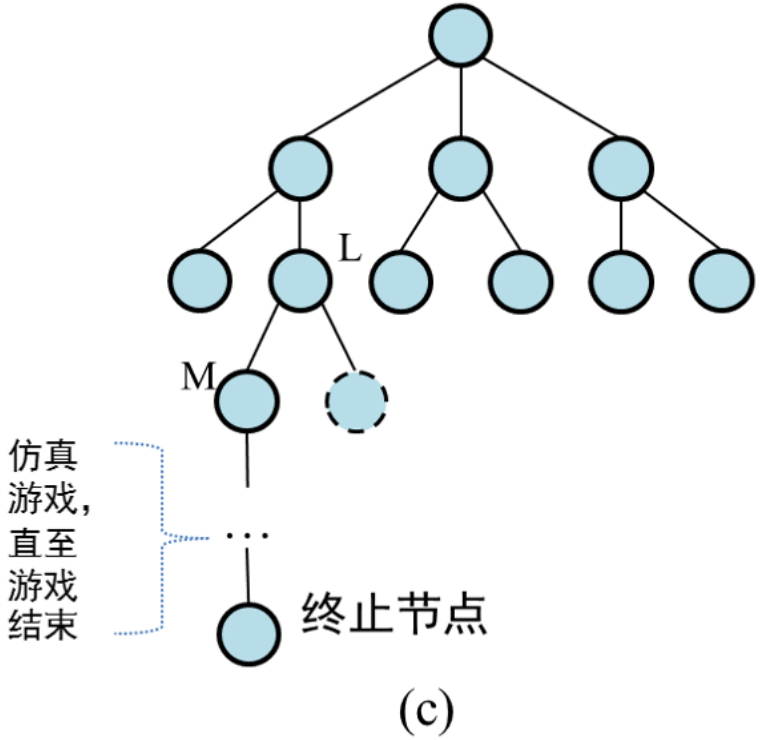




如果节点 L 不是一个终止节点（或对抗搜索的终局节点），则随机扩展它的一个未被扩展过的后继边缘节点M。

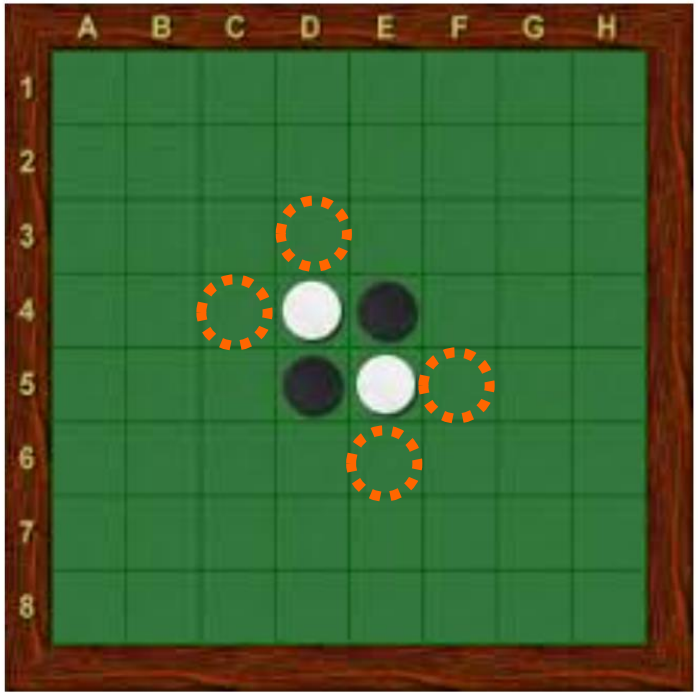
模拟（simulation）

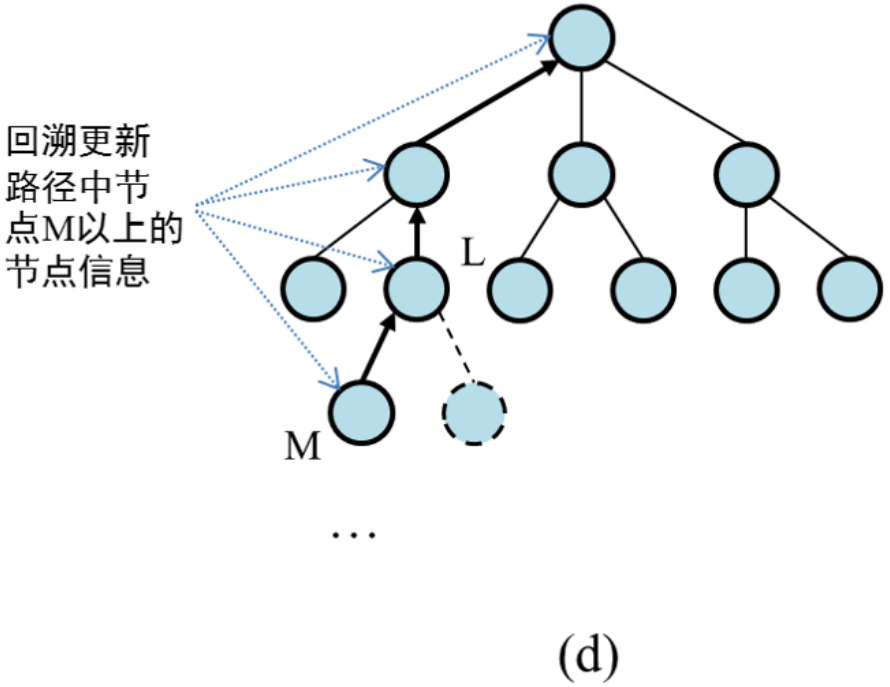




从节点M出发，模拟扩展搜索树，直到找到一个终止节点。模拟过程使用的策略和采用UCB算法实现的选择过程并不相同，前者通常会使用比较简单的策略，例如使用随机策略。

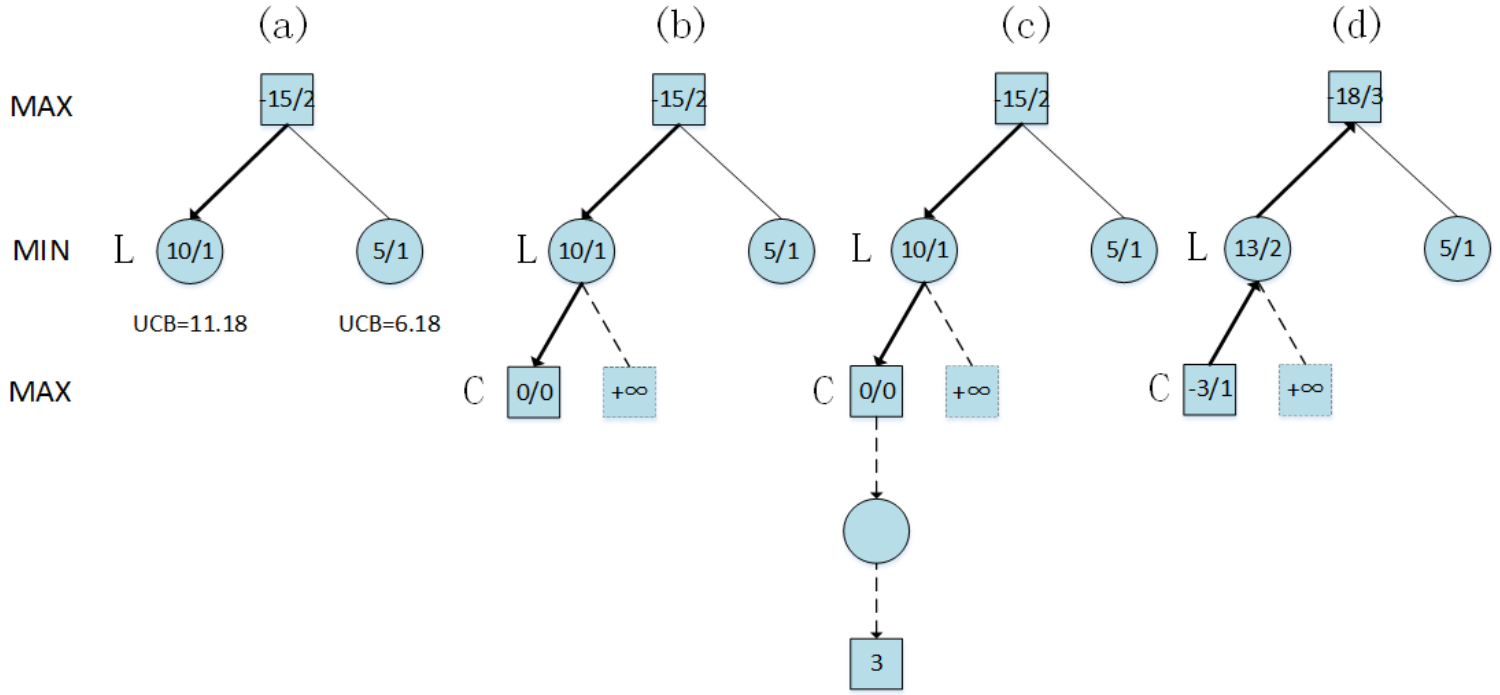
反向传播（Back Propagation）





用模拟所得结果（终止节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）节点的奖励均值和被访问次数。

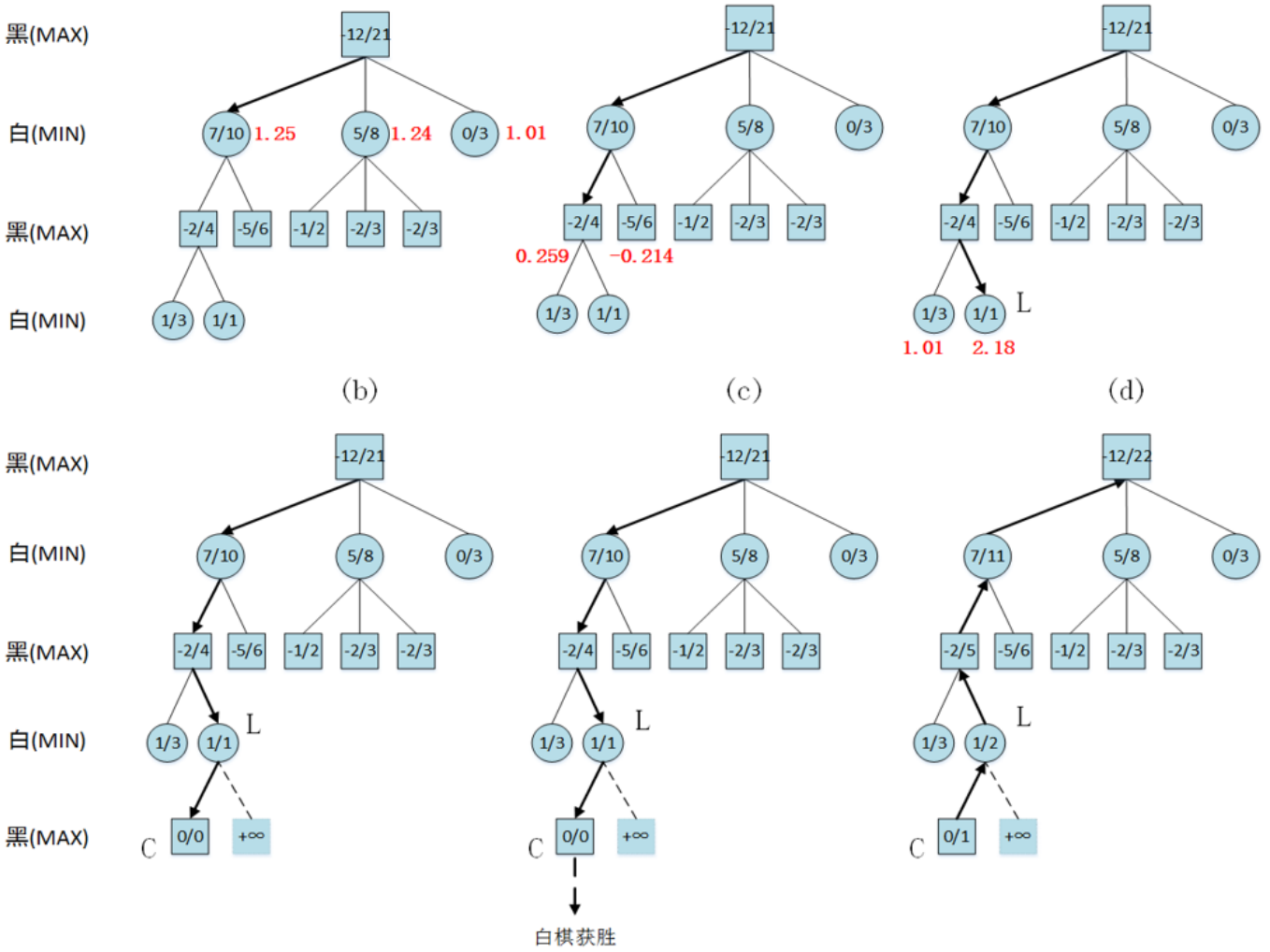
蒙特卡洛树搜索算法



在图(d)中C节点的总分被更新为-3，被访问次数被更新为1；L节点的总分被更新为13，被访问次数被更新为2；根节点的总分被更新为-18，被访问次数被更新为3。在更新时，会将MIN层节点现有总分加上终局得分分数， MAX层节点现有总分减去终局得分分数。这是因为在对抗搜索中，玩家MIN总是期望最小化终局得分，因此在MIN层选择其子节点时，其目标并非选取奖励最大化的子节点，而是选择奖励最小化的节点，为了统一使用UCB1算法求解，算法将MIN层的子节点（即MAX层节点）的总分记为其相反数。

蒙特卡洛树搜索算法

对于黑棋来说，黑棋获胜得1分，平局得0.5分，白棋获胜得0分.



黑方希望最大化终局得分而白方希望最小化终局得分

蒙特卡洛树搜索算法

函数： UCTSearch()

输入： 当前状态 $state$
输出： 玩家 MAX 行动下，当前最优动作 a^*
$v_0 \leftarrow \text{TreeNode}(state)$
while 未达到限制时长 **do**
    $v_{select} \leftarrow \text{SelectPolicy}(v_0)$
    $v_{expand} \leftarrow \text{Expand}(v_{select})$
    $s_{result} \leftarrow \text{SimulatePolicy}(v_{expand}, state)$
    $\text{BackPropagate}(v_{expand}, s_{result})$
end while
$a^* \leftarrow argmax_{v_l \in v.child} \frac{v_l.Q}{v_l.N}$

函数： SelectPolicy()

输入： 选择的起始结点 v_0
输出： 选择步骤的结束结点 v
$v \leftarrow v_0$
while v 结点已被扩展且不为叶子结点 **do**
    $v \leftarrow UCB(v)$
end while

函数： Expand()

输入： 节点 v
输出： 未被扩展的后继结点 v' 集合
if 节点 v 的后继结点 v' 为合法落子节点 **then**
    $v.child \leftarrow v.child + v'$
end if

蒙特卡洛树搜索算法

函数： UCTSearch()

输入： 当前状态 $state$

输出： 玩家 MAX 行动下，当前最优动作 a^*

$v_0 \leftarrow \text{TreeNode}(state)$

while 未达到限制时长 **do**

  $v_{select} \leftarrow \text{SelectPolicy}(v_0)$

  $v_{expand} \leftarrow \text{Expand}(v_{select})$

  $s_{result} \leftarrow \text{SimulatePolicy}(v_{expand}, state)$

  $\text{BackPropagate}(v_{expand}, s_{result})$

end while

$a^* \leftarrow argmax_{v_l \in v.child} \frac{v_l.Q}{v_l.N}$

函数： SimulatePolicy()

输入： 节点 v 和状态 $state$

输出： 模拟的终局结果 s_{result}

$s_{result} = state$

while 状态 s_{result} 尚未达到终局 **do**

  $a \leftarrow action(s_{result})$

  $s_{result} \leftarrow result(s_{result}, a)$

end while

函数： BackPropagate()

输入： 反向传播更新的起始节点 v 和终局状态 s_{result}

while v 不为空时 **do**

  $v.N \leftarrow v.N + 1$

  $v.Q \leftarrow v.Q + Q(s_{result})$

  $v \leftarrow v.parent$

end while