蒙特卡洛树搜索(MCTS)

阿尔法狗下棋的时候，做决策的不是策略网络和价值网络，而是蒙特卡洛　树搜索(Monte Carlo Tree Search, MCTS)。

训练好的策略网络和价值网络均能单独地直接做决策。MCTS不需要训练，也可以单独地直接做决策。在阿尔法狗中，训练策略网络和价值网络的目的是辅助MCTS，降低MCTS的深度和宽度。

在机巧围棋中，除阿尔法狗之外，还分别集成了策略网络、价值网络和蒙特卡洛落子策略，可以任意更改黑白双方的落子策略，查看不同落子策略之间的效果。

1. MCTS的基本思想

人类玩家下围棋时，通常会往前看几步，越是高手，看的越远。与此同时，人类玩家不会分析棋盘上所有不违反规则的走法，而只会针对性地分析几个貌似可能的走法。

假如现在该我放置棋子了，我会这样思考：现在貌似有几个可行的走法，如果我的动作是$a_t = 234$，对手会怎么走呢？假如对手接下来将棋子放在$a' = 30$的位置上，那我下一步动作a_{t+1}应该是什么呢？

人类玩家在做决策之前，会在大脑里面进行推演，确保几步以后很可能会占优势。同样的道理，AI下棋时候，也应该枚举未来可能发生的情况，从而判断当前执行什么动作的胜算更大。这样做远好于使用策略网络直接算出一个动作。

MCTS的基本原理就是向前看，模拟未来可能发生的情况，从而找出当前最优的动作。这种向前看不是遍历所有可能的情况，而是与人类玩家类似，只遍历几种貌似可能的走法，而哪些动作是貌似可行的动作以及几步之后的局面优劣情况是由神经网络所决定的。阿尔法狗每走一步棋，都要用MCTS做成千上万次模拟，从而判断出哪个动作的胜算更大，并执行胜算最大的动作。

2. 阿尔法狗2016版中的MCTS(MCTS in AlphaGo)

在阿尔法狗2016版本中，MCTS的每一次模拟分为4个步骤：选择(selection)、扩展(expansion)、求值(evaluation)和回溯(backup)。

2.1 选择(Selection)

给定棋盘上的当前格局，可以确定所有符合围棋规则的可落子位置，每个位置对应一个可行的动作。在围棋中，每一步很有可能存在几十甚至上百个可行的动作，挨个搜索和评估所有可行的动作，计算量会大到无法承受。人类玩家做决策前，在大脑里面推演的时候不会考虑所有可行的动作，只会考虑少数几个认为胜算较高的动作。

MCTS第一步【选择】的目的就是找出胜算较高的动作，只搜索这些好的动作，忽略掉其它的动作。

判断动作a的好坏有两个指标：第一，动作a的胜率；第二，策略网络给动作a的评分(概率值)。结合这两个指标，用下面的公式评价动作a的好坏：

$$score(a) \triangleq Q(a) + \frac{\eta}{1+N(a)} \cdot \pi(a|s;\theta) \qquad (1)$$

其中：

- $N(a)$是动作a已经被访问过的次数。初始时，对于所有a，令$N(a) \leftarrow 0$。动作a每被选中一次，就把$N(a)$的值加1：$N(a) \leftarrow N(a) + 1$；
- $Q(a)$是之前$N(a)$次模拟算出来的动作价值，主要由胜率和价值函数决定。$Q(a)$的初始值是0，动作a每被选中一次，就会更新一次$Q(a)$；
- η是一个超参数，需要手动调整；
- $\pi(a|s;\theta)$是策略网络对动作a的评分。

可以这样理解上述公式(1)：

- 如果动作a还没有被选中过，则$Q(a)$和$N(a)$均等于0，因此$score(a) \propto \pi(a|s;\theta)$，即完全由策略网络评价动作$a$的好坏；
- 如果动作a已经被选中过很多次，则$N(a)$会很大，因此策略网络给动作a的评分在$score(a)$中的权重会降低。当$N(a)$很大的时候，有$score(a) \approx Q(a)$，即此时主要基于$Q(a)$判断a的好坏，策略网络给动作a的评分已经无关紧要了；
- 系数$\frac{\eta}{1+N(a)}$的另一个作用是鼓励探索。如果两个动作有相近的Q分数和π分数，那么被选中次数少的动作的$score$会更高，也就是让被选中次数少的动作有更多的机会被选中。

给定某个状态s，对于所有可行的动作，MCTS会使用公式(1)算出所有动作的分数$score(a)$，找到分数最高的动作，并在这轮模拟中，执行这个动作(选择出的动作只是在模拟器中执行，类似于人类玩家在大脑中推演，并不是阿尔法狗真正走的一步棋)。

2.2 扩展(Expansion)

将第一步选中的动作记为a_t，在模拟器中执行动作a_t，环境应该根据状态转移函数$p(s_{k+1}|s_k, a_k)$返回给阿尔法狗一个新的状态s_{t+1}。

假如阿尔法狗执行动作a_t，对手并不会告诉阿尔法狗他会执行什么动作，因此阿尔法狗只能自己猜测对手的动作，从而确定新的状态s_{t+1}。和人类玩家一样，阿尔法狗可以推己及人：如果阿尔法狗认为几个动作很好，那么就假设对手也怎么认为。所以阿尔法狗用策略网络模拟对手，根据策略网络随机抽样一个动作：

$$a'_t \sim \pi(\cdot|s'_t;\theta) \qquad (2)$$

此处的状态s'_t是站在对手的角度观测到的棋盘上的格局，动作a'_t是假想的对手选择的动作。

进行MCTS需要模拟阿尔法狗和对手对局，阿尔法狗每执行一个动作a_k，环境应该返回一个新的状态s_{k+1}。围棋游戏具有对称性，阿尔法狗的策略，在对手看来是状态转移函数；对手的策略，在阿尔法狗看来是状态转移函数。最理想情况下，模拟器的状态转移函数是对手的真实策略，然而阿尔法狗并不知道对手的真实策略，因此阿尔法狗退而求其次，用自己训练出的策略网络π代替对手的策略，作为模拟器的状态转移函数。

2.3 求值(Evaluation)

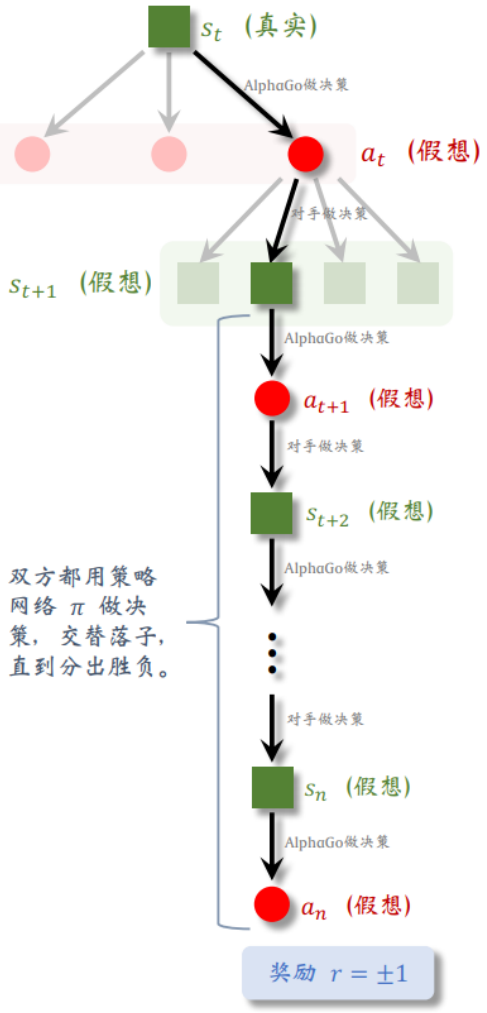
从状态s_{t+1}开始，双方都用策略网络π做决策，在模拟器中交替落子，直至分出胜负（见图一）。阿尔法狗基于状态s_k，根据策略网络抽样得到动作：

$$a_k \sim \pi(\cdot|s_k;\theta) \qquad (3)$$

对手基于状态s'_k（从对手角度观测到的棋盘上的格局），根据策略网络抽样得到动作：

$$a'_k \sim \pi(\cdot|s'_k;\theta) \qquad (4)$$

模拟对局直至分出胜负，可以观测到奖励r。如果阿尔法狗胜利，则$r=+1$，否则$r=-1$。



图一 策略网络自我博弈 CSDN @DeepGeGe

综上所述，棋盘上真实的状态是s_t，阿尔法狗在模拟器中执行动作a_t，然后模拟器中的对手执行动作a'_t，带来新的状态s_{t+1}。对于阿尔法狗来说，如果状态s_{t+1}越好，则这局游戏胜算越大，因此：

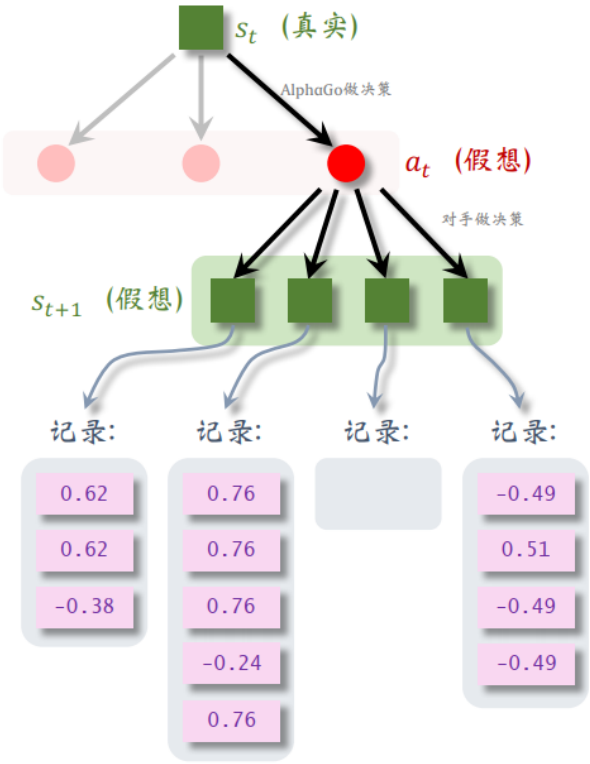
- 如果阿尔法狗赢得了这局模拟($r=+1$)，则说明s_{t+1}可能很好；如果输了($r=-1$)，则说明可能不好。因此，奖励r可以反映出s_{t+1}的好坏。
- 此外，还可以使用价值网络评价状态s_{t+1}的好坏。价值$v(s_{t+1};\omega)$越大，则说明状态s_{t+1}越好。

奖励r是模拟对局获得的胜负，是对s_{t+1}很可靠的评价，但是随机性很大。价值网络的评估$v(s_{t+1};\omega)$没有r可靠，但是价值网络更稳定，随机性小。阿尔法狗将奖励r与价值网络的输出$v(s_{t+1};\omega)$取平均，作为对状态s_{t+1}的评价，记作：$V(s_{t+1}) \triangleq \frac{r+v(s_{t+1};w)}{2}$。

使用策略网络交替落子，直至分出胜负，通常要走一两百步。在实际实现时候，阿尔法狗训练了一个更小的神经网络（称为快速走子网络）来代替大的策略网络，以加速MCTS。

2.4 回溯(Backup)

第三步【求值】计算出了$t+1$步某一个状态的价值，记作$V(s_{t+1})$。每一次模拟都会得出这样一个价值，并且记录下来。模拟会重复很多次，于是第$t+1$步每一种状态下面可以有多条记录，如图二所示。



图二 每一个状态 s_{t+1} 下面都有很多条记录，每一条记录是一个 $V(s_{t+1})$。

第t步的动作a_t下面有多个可能的状态（子节点），每个状态下面有若干条记录。把a_t下面所有的记录取平均，记为价值$Q(a_t)$，它可以反映出动作a_t的好坏。

给定棋盘上真实的状态s_t，有多个可行的动作a可供选择。对于所有的a，价值$Q(a)$的初始值为0。动作a每被选中一次（成为a_t），它下面就会多一条记录，我们就对$Q(a)$做一次更新。

2.5 MCTS的决策

上述4个步骤为一次MCTS的流程，MCTS想要真正做出一个决策（即往真正的棋盘上落一个棋子），需要做成千上万次模拟。在无数次模拟之后，MCTS做出真正的决策：

$$a_t = \overset{argmax}{a} N(a) \qquad (5)$$

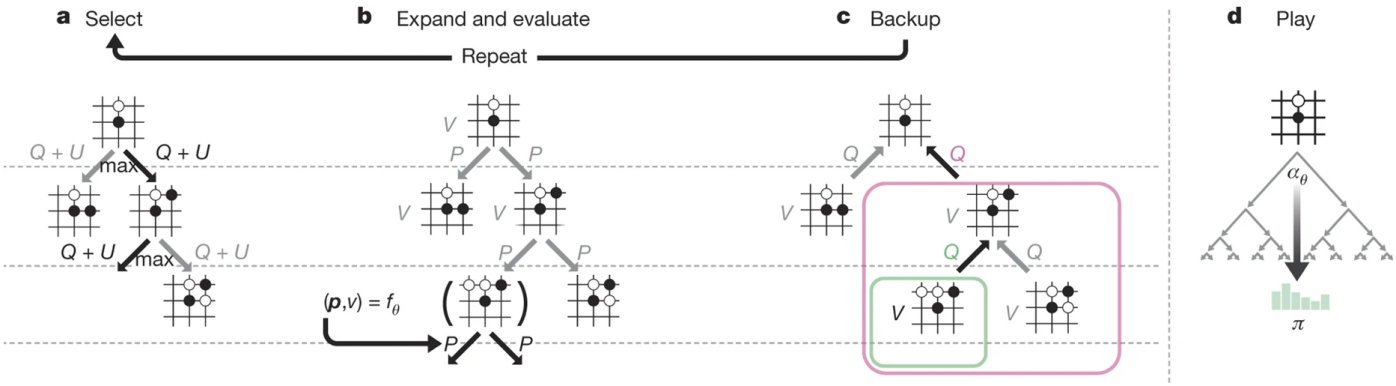
此时阿尔法狗才会真正往棋盘上放一个棋子。

> 为什么要依据$N(a)$来做决策呢?
> 在每一次模拟中，MCTS找出所有可行的动作$\{a\}$，计算他们的分数$score(a)$，然后选择其中分数最高的动作，并在模拟器中执行。如果某个动作a在模拟时胜率很大，那么它的价值$Q(a)$就会很大，它的分数$score(a)$会很高，于是它被选中的几率就很大。也就是说如果某个动作a很好，他被选中的次数$N(a)$就会大。

观测到棋盘上当前状态s_t，MCTS做成千上万次模拟，记录每个动作a被选中的次数$N(a)$，最终做出决策$a_t = \overset{argmax}{a} N(a)$。到了下一时刻，状态变成了$s_{t+1}$，MCTS会把所有动作$a$的$Q(a)$、$N(a)$全都初始化为0，然后从头开始做模拟，而不能利用上一次的结果。

3. 零狗中的MCTS(MCTS in AlphaGo Zero)

零狗中对MCTS进行了简化，放弃了快速走子网络，合并了【扩展】和【求值】，并且更改了【选择】和【决策】逻辑。零狗中维护了一个蒙特卡洛搜索树，搜索树的每一个节点保存了$N(s,a)$(节点访问次数)、$W(s,a)$(合计动作价值)、$Q(s,a)$(平均动作价值)和$P(s,a)$(选择该节点的先验概率)。每一次模拟会遍历一条从搜索树根结点到叶节点的路径。



图三 零狗中的MCTS

如图三所示，零狗中每一次MCTS共有三个流程：

- **选择(Select)：**

  在选择阶段，从搜索树的根节点开始，不断选择$a_c = \underset{a}{^{argmax}} \Big(Q(s,a) + U(s,a) \Big)$，其中$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$，直至搜索树的叶节点终止。

  > s：为搜索树的一个节点代表的棋局状态；
  > a：表示某一个可行的动作；
  > $N(s,a)$：表示状态s下可行动作a被选中的次数；
  > $P(s,a)$：为状态s下的可行动作a的先验概率；
  > $Q(s,a)$：表示状态s下可行动作a的动作价值；
  > c_{puct}：为一个决定探索程度超参数。

- **拓展和求值(Expand and evaluate)：**

  选择阶段，在搜索树中不断选择$Q+U$最大的动作，直至游戏结束或某一个不是终局的叶结点。如果到了不是终局的叶结点l，对于l对应的棋局状态s，使用策略网络和价值网络对状态s进行评估，得到l对应棋局状态s下一步各个可能动作的概率p和l的价值v。为所有可能动作对应的棋局状态分别创建一个节点，将这些节点的先验概率设置为策略网络的输出概率值。

- **回溯(Backup)：**

  进过上述扩展之后，之前的叶子节点l，现在变成了内部节点。做完了扩展和求值后，从节点l开始，逐层向搜索树根节点回溯，并依次更新搜索树当次被遍历的路径上各层节点的信息：

$$
\begin{aligned}
N(s_n, a_n) &= N(s_n, a_n) + 1 \\
W(s_n, a_n) &= W(s_n, a_n) + v_n \\
Q(s_n, a_n) &= \frac{W(s_n, a_n)}{N(s_n, a_n)}
\end{aligned}
$$

  > s_n：表示搜索树中当次被遍历路径上节点对应的棋局状态；
  > a_n：表示搜索树中当次被遍历路径上节点对应棋局状态下选择的动作；
  > v_n：表示搜索树中当次被遍历路径上节点的价值，由于搜索树中相邻两层的落子方是不同的，因此相邻两层的节点价值互为相反数。

上述三个流程为零狗中的一次MCTS模拟，在零狗往真正的棋盘上落一个棋子之前，会进行1600次模拟。在上千次MCTS完成之后，MCTS基于下述公式做出真正的决策：

$$
\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}} \qquad (6)
$$

> τ为温度参数，控制探索的程度。 τ越大，不同走法间差异变小，探索比例增大。反之，则更多选择当前最优操作。在零狗中，每一次自我对弈的前30步，参数$\tau = 1$，即早期鼓励探索。游戏剩下的步数，该参数将逐渐降低至0。如果是比赛，则直接为0。

4. MCTS的程序实现

机巧围棋是基于AlphaGo Zero算法的一款点击按钮就能可视化训练围棋人工智能的程序，机巧围棋中的MCTS与零狗中的MCTS一致，不过不支持多线程搜索，具体代码如下：

```python
class TreeNode:
    """蒙特卡洛树节点"""
    def __init__(self, parent, prior_p):
        self.parent = parent  # 节点的父节点
        self.children = {}  # 一个字典，用来存节点的子节点
        self.n_visits = 0  # 节点被访问的次数
        self.Q = 0  # 节点的平均行动价值
        self.U = 0  # MCTS选择Q+U最大的节点，公式里的U
        self.P = prior_p  # 节点被选择的概率

    def select(self, c_puct):
        """
        蒙特卡洛树搜索的第一步：选择
        蒙特卡洛树搜索通过不断选择 最大上置信限Q+U 的子节点，直至一个树的叶结点
        该函数为进行一步选择函数

        :param c_puct: 为计算U值公式中的c_puct，是一个决定探索水平的常数
        :return: 返回一个元组(action, next_node)
        """
        return max(self.children.items(),
                   key=lambda act_node: act_node[1].get_value(c_puct))

    def expand(self, action_priors):
        """
        当select搜索到一个叶结点，且该叶节点代表的局面游戏没有结束，
        需要expand树，创建一系列可能得节点，即对应节点所有可能选择的动作对应的子节点

        :param action_priors: 为一个列表，列表中的每一个元素为一个 特定动作及其先验概率 的元组
        :return:
        """
        for action, prob in action_priors:
            if action not in self.children:
                self.children[action] = TreeNode(self, prob)

    def update(self, leaf_value):
        """
        根据子树的价值更新当前节点的价值

        :param leaf_value: 以当前玩家的视角看待得到的子树的价值
        :return:
        """
        self.n_visits += 1  # 当前节点的访问次数+1
        # 更新当前节点的Q值，下述公式可由Q =  W / N 推导得到
        # Q_old = W_old / N_old
        # Q = (W_old + v) / (N_old + 1) = (Q_old * N_old + v) / (N_old + 1)
        self.Q += 1.0 * (leaf_value - self.Q) / self.n_visits

    def update_recursive(self, leaf_value):
        """
        跟心所有祖先的Q值及访问次数

        :param leaf_value:
        :return:
        """
        if self.parent:  # 如果有父节点，证明还没到根节点
            self.parent.update_recursive(-leaf_value)  # -leaf_value是因为每向上一层，以当前玩家视角，价值反转
        self.update(leaf_value)

    def get_value(self, c_puct):
        """
        计算并返回一个节点的 上置信限 评价，即Q+U值

        :param c_puct: 为计算U值公式中的c_puct，是一个决定探索水平的常数
        :return:
        """
        self.U = c_puct * self.P * np.sqrt(self.parent.n_visits) / (1 + self.n_visits)
        return self.Q + self.U

    def is_leaf(self):
        """
        判断当前节点是否为叶结点

        :return:
        """
        return self.children == {}

    def is_root(self):
        """
```

```
        判断当前节点是否为根节点

        :return:
        """
        return self.parent is None


class MCTS:
    """蒙特卡洛树搜索主体"""
    def __init__(self, policy_value_fn, c_puct=5, n_playout=10000):
        self.root = TreeNode(None, 1.0)  # 整个蒙特卡洛搜索树的根节点
        # policy_value_fn是一个函数，该函数的输入为game_state，
        # 输出为一个列表，列表中的每一个元素为(action, probability)形式的元组
        self.policy = policy_value_fn
        # c_puct为一个正数，用于控制多块收敛到策略的最大值。这个数越大，意味着越依赖前面的结果。
        self.c_puct = c_puct
        self.n_playout = n_playout

    def playout(self, simulate_game_state):
        """
        从根节点不断选择直到叶结点，并获取叶结点的值，反向传播到叶结点的祖先节点

        :param simulate_game_state: 模拟游戏对象
        :return:
        """
        node = self.root
        while True:  # 从根节点一直定位到叶结点
            if node.is_leaf():
                break
            # 贪婪地选择下一步动作
            action, node = node.select(self.c_puct)
            simulate_game_state.step(action)
        # 使用网络来评估叶结点，产生一个每一个元素均为(action, probability)元组的列表，以及
        # 一个以当前玩家视角看待的在[-1, 1]之间的v值
        action_probs, leaf_value = self.policy(simulate_game_state)
        # 检查模拟游戏是否结束
        end, winner = simulate_game_state.game_ended(), simulate_game_state.winner()
        if not end:  # 没结束则扩展
            node.expand(action_probs)
        else:
            if winner == -1:  # 和棋
                leaf_value = 0.0
            else:
                leaf_value = (
                    1.0 if winner == simulate_game_state.turn() else -1.0
                )
        # 更新此条遍历路径上的节点的访问次数和value
        # 这里的值要符号反转，因为这个值是根据根节点的player视角来得到的
        # 但是做出下一步落子的是根节点对应player的对手
        node.update_recursive(-leaf_value)

    def get_move_probs(self, game, temp=1e-3, player=None):
        """
        执行n_playout次模拟，并根据子节点的访问次数，获得每个动作对应的概率

        :param game: 游戏模拟器
        :param temp: 制探索水平的温度参数
        :param player: 调用该函数的player，用于进行进度绘制
        :return:
        """
        for i in range(self.n_playout):
            if not player.valid:
                return -1, -1
            if player is not None:
                player.speed = (i + 1, self.n_playout)
            simulate_game_state = game.game_state_simulator(player.is_selfplay)
            self.playout(simulate_game_state)
        # 基于节点访问次数，计算每个动作对应的概率
        act_visits = [(act, node.n_visits)
                      for act, node in self.root.children.items()]
        acts, visits = zip(*act_visits)
        act_probs = softmax(1.0 / temp * np.log(np.array(visits) + 1e-10))
        return acts, act_probs

    def get_move(self, game, player=None):
        """
        执行n_playout次模拟，返回访问次数最多的动作

```

```
        :param game: 游戏模拟器
        :param player: 调用该函数的player，用于进行进度绘制
        :return: 返回访问次数最多的动作
        """
        for i in range(self.n_playout):
            if not player.valid:
                return -1
            if player is not None:
                player.speed = (i + 1, self.n_playout)
            game_state = game.game_state_simulator()
            self.playout(game_state)
        return max(self.root.children.items(), key=lambda act_node: act_node[1].n_visits)[0]

    def update_with_move(self, last_move):
        """
        蒙特卡洛搜索树向深层前进一步，并且保存对应子树的全部信息

        :param last_move: 上一步选择的动作
        :return:
        """
        if last_move in self.root.children:
            self.root = self.root.children[last_move]
            self.root.parent = None
        else:
            self.root = TreeNode(None, 1.0)
```

5. 结束语

本文介绍了阿尔法狗2016版本和零狗中的蒙特卡洛树搜索及其实现，在机巧围棋中也集成了纯蒙特卡洛落子策略（所有可行动作的概率值是随机的，节点的状态价值通过随机落子到游戏终局，根据胜负确定），大家可以在GitHub上clone机巧围棋的代码，体验纯蒙特卡洛落子策略的效果。

最后，期待您能够给本文点个赞，同时去GitHub上给机巧围棋项目点个Star呀~

机巧围棋项目链接：https://github.com/QPT-Family/QPT-CleverGo