UCB算法

$$argmax = Q(v')/N(v') + c * sqrt(2lnN(v)/N(v'))$$

其中v'表示当前树节点，v表示父节点，Q表示这个树节点的累计quality值，N表示这个树节点的visit次数，C是一个常量参数（可以控制exploitation和exploration权重）。

这个公式的意思时，对每一个节点求一个值用于后面的选择，这个值有两部分组成，左边是这个节点的平均收益值（越高表示这个节点期望收益好，越值得选择，用于exploitation利用），右边的变量是这个父节点的总访问次数除以子节点的访问次数（如果子节点访问次数越少则值越大，越值得选择，用户exploration探索），因此使用这个公式是可以兼顾探索和利用的。

MCTS

MCTS的算法分为四步，**第一步是Selection**，就是在树中找到一个最好的值得探索的节点，一般策略是先选择未被探索的子节点，如果都探索过就选择UCB值最大的子节点。**第二步是Expansion**，就是在前面选中的子节点中走一步创建一个新的子节点，一般策略是随机自行一个操作并且这个操作不能与前面的子节点重复。**第三步是Simulation**，就是在前面新Expansion出来的节点开始模拟游戏，直到到达游戏结束状态，这样可以收到到这个expansion出来的节点的得分是多少。**第四步是Backpropagation**，就是把前面expansion出来的节点得分反馈到前面所有父节点中，更新这些节点的quality value和visit times，方便后面计算UCB值。

简单来理解这个算法过程，在Simulation中，是模拟运行到最后一步，获取子节点的状态值可以用来计算UCB，**state在算法中是状态，并不是真的创建节点运行到这里**，然后backup更新值，然后每一步启动一次monte_carlo_tree_search来运行，找到最佳子节点。

Algorithm 2 The UCT algorithm.

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)

$v \leftarrow \text{BESTCHILD}(v, Cp)$

return 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)

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

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

---

code

首先Node包含了parent和children属性，还有就是用于计算UCB值的visit times和quality value，为了关联游戏状态，我们还需要为每个Node绑定一个State对象。Node需要实现增加节点、删除节点等功能，还有需要提供函数判断子节点的个数和是否有空闲的子节点位置。

每一个State都需要包含当前的游戏得分，可以继续当前游戏玩到第几轮，还有每一轮的选择是什么。当然更重要的，它还需要实现is_terminal()方法判断游戏是否结局，实现compute_reward()方法告诉用户当前得分是多少，还有提供get_next_state()方法用户进行游戏得到新的状态，几个函数与游戏场景游戏，这里简单实现了一个“选择数字保证累加和为1”的游戏。

要实现伪代码提到的几个方法，我们直接定义一个monte_carlo_tree_search()函数，然后依次调用tree_policy()、default_policy()、backup()这些方法实现即可。

为了避免程序无限搜索下去，我们需要定义一个computation budget，限制搜索次数或者搜索时间，这里限制只能向下搜索1000次，然后通过下面的方法来找到expansion node、计算reward、并且backpropation到所有有关的节点中。

Tree_policy()实现了一个策略，就是检查如果一个节点下面还有未探索的子节点，那么先expansion下面的子节点就可以了，如果没有子节点，那么就用best_child()函数（其实也就是UCB算法）来得到下一个子节点，然后便利下直到有未探索的节点可以探索。

best_child()算法很简单，就是根据Node的State获取quality value和visit times，然后计算出UCB值，然后比较返回UCB值最大的子节点而已。

expand()函数实现稍微复杂些，实际上就是在当前节点下，选择一个未被执行的Action来执行即可，策略就是随机选，如果有随机选中之前已经执行过的则重新选。**这里我提供了expand的修改，防止while循环过多，我直接获取尚未选择的动作然后random**

在这个游戏种模拟也很简单，我们直接调用State类中实现的随机玩游戏策略，一直玩到最后得到一个reward值即可，当然对于AlphaGo等其他游戏可能需要实现一个更优的快速走子网络实现类似的功能，后面会细谈区别与AlphaGo的区别。

那么最后我们就需要把前面计算的这个reward反馈到“相关的”的节点上了，这个相关的意思是从根节点开始一直到这个expansion节点经过的所有节点，他们的quality value和visit times都需要更新