AI_Reversi

第三组

序号	学号	专业班级	姓名	性别
1	3140103367	软工 1401	杨奕辉	男
2	3140103367	计科 1402	周思昊	男
3	3130100830	数媒 1301	诸逸铭	男

1. Project Introduction

开发环境：Python 2.7

开发工具：Visual Studio Code、PyCharm

开发包：内建包 argparse,copy,signal,sys,timeit,imp,traceback 等

操作系统：Windows 10

工作分配：

- 杨奕辉：负责设计算法的数据结构，编写 UCTSearch 和 Bestchild 两个函数。
- 诸逸铭：负责 defaultPolicy（模拟终局）的实现，以及利用点阵做速度方面的优化。
- 周思昊：负责 expand（拓展节点）和 backup（回溯）算法的实现。

2. Technical Details

蒙特卡洛树搜索

蒙特卡洛树搜索分为 4 个部分：选择、扩展、模拟、回溯。

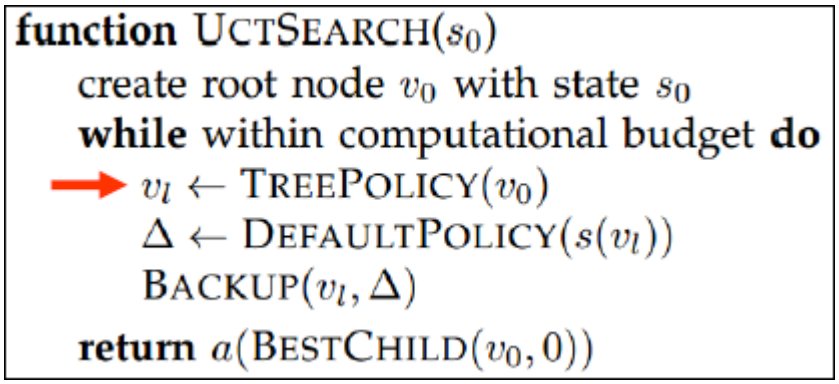
- 选择：指从根节点开始，选择连续的子节点向下至叶子节点 L。
- 扩展：指除非任意一方的输赢导致游戏结束，否则 L 会创建一个或多个子节点。
- 模拟：从 L 的子节点中随机布局。
- 回溯：使用布局结果更新从 L 到根节点路径上的节点信息。

算法原理：利用蒙特卡洛树搜索（MCTS）进行棋局的状态搜索，并使用点阵运算加快模拟终局的速度，以获得更大的扩展层级，增加节点估值的收敛性，从而提高精确度，获得最优的决策。

1. 每个节点的结构如下：

```
state_node
{
    parentState: old_state_node     //father node
    currState: board                //board of current node
    childState:{
        move:state_node             //child node
    }
    color:-1 1
    count:  访问了几次该节点
    eval: 胜利率，为所有者子节点胜利率的平均值。终局的胜利率为1或-1
}
```

2. 首先是实现整个 MCTS 算法的入口算法：UCTSearch，伪代码如下。

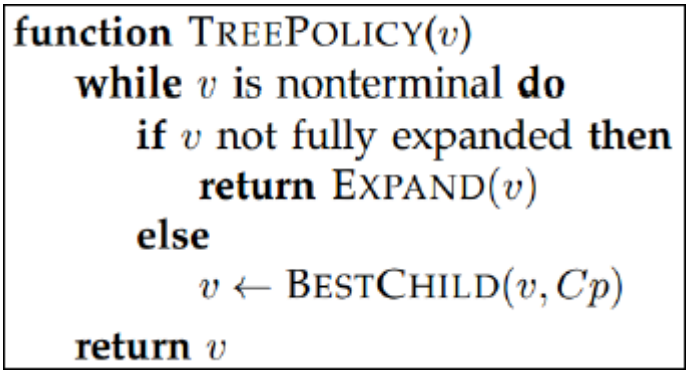


在该函数中我们增加了一些跳出循环的条件：

- 选举出来的节点已经是叶节点
- 已经遍历到预先设定的最大层级

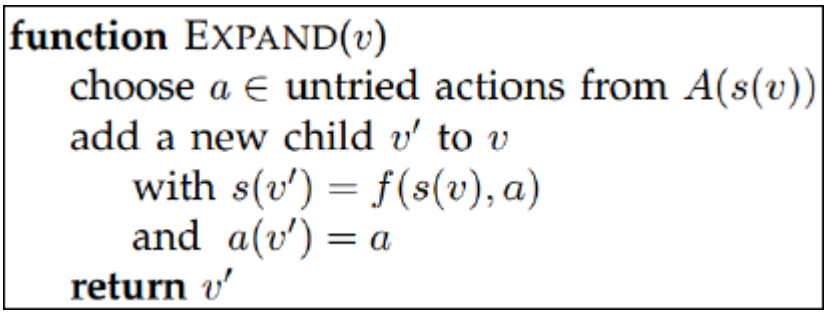
- 时间已消耗至 56s 以上。（一局时间为 60s）

2. 其次是选举过程算法 treePolicy，伪代码如下：



首先获取目标棋局的所有合法招式，如果某一个招式对应的棋局未被搜索过，则调用 expand()并返回该招式对应的棋局，否则选择所有招式中当前最优的招式对应的棋局，继续调用 treePolicy()，进行递归，直到达到预先设定的深度。

3. 然后是拓展节点算法 Expand，伪代码如下：



其中需要考虑拓展的节点颜色：

```
# 判断新节点颜色
legal_moves=state['currState'].get_legal_moves(-state_node['color'])
if len(legal_moves) == 0:
    state['color'] = state_node['color']
else:
    state['color'] = -state_node['color']
```

4. 接着，需要实现模拟至终局的算法 defaultPolicy，伪代码如下：

```
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
```

在本算法中，我们使用了将棋盘转化为点阵，并进行位操作，来提高模拟的速度。

同时对模拟过程的每一步决策，也抛弃了随机策略，而采用参考当前棋盘每个位置的权重，选择权重最大的有效棋步来下，如此能够获得更为准确的终局结果。关键代码如下：

```
# 棋盘各点权值图
self.graph=[
    [99,-8,8,6,6,8,-8,99],
    [-8,-24,-4,-3,-3,-4,-24,-8],
    [8,-4,7,4,4,7,-4,8],
    [6,-3,4,0,0,4,-3,6],
    [6,-3,4,0,0,4,-3,6],
    [8,-4,7,4,4,7,-4,8],
    [-8,-24,-4,-3,-3,-4,-24,-8],
    [99,-8,8,6,6,8,-8,99]
]
```

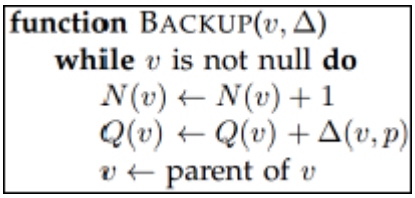
模拟 8x8 的棋盘各位置，数值代表权重。

```
# 参照位置权重来选择模拟的棋步
posVal=-999
bestMv=None
for mv in legal_moves:
    tmv=to_move(mv)
    if self.graph[tmv[0]][tmv[1]]>posVal:
        posVal=self.graph[tmv[0]][tmv[1]]
        bestMv=mv
# 翻转
flipmask = flip(selfTurn,opponent,bestMv)
selfTurn ^= flipmask | BIT[bestMv]
opponent ^= flipmask
```

选择权重最大的策略来模拟执行。

对于终局结果的估值，我们采用**己方胜利则 reward 为 1，己方失败则 reward 为 0，打平则 reward 为 0.5 的策略。**

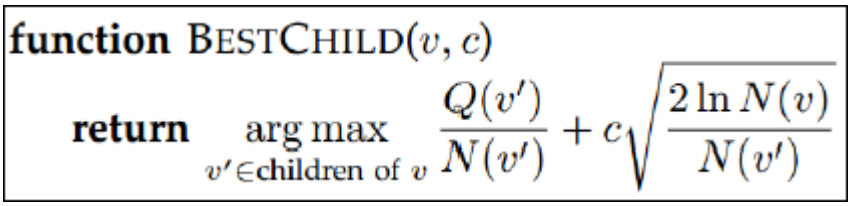
5. 对于模拟终局的结果，作为 reward 需要回溯更新每一个祖先节点，伪代码如下：



代码实现也比较简单：

```
# 回溯
def backup(self,state,reward):
    while state:
        state['eval']=(state['eval']*state['count']+reward)/float(state['count']+1)
        state['count']+=1
        state=state['parentState']
    return
```

6. 最后，实现计算 UCB 得出估值最高的子节点的算法，伪代码如下：



这里我们考虑了己方节点和对方节点对于估值选择的不同决策：己方节点将选择子节点中估值最大的(Max)，对方节点将选择子节点估值最小的(Min)。该部分关键代码实现如下：

```
eval=child['eval']+math.sqrt(C*math.log(state['count'])/float(child['count']))
if color == self.root['color']:
    if eval>targetEval:
        targetEval=eval
        targetMv=mv
        target_state=child
else:
    if eval<targetEval:
        targetEval=eval
        targetMv=mv
        target_state=child
```

3. Experiment Results

在保证每一步不超时，且速度可观的情况下，最大搜索层级可扩展至 8 层。下面分别与 simple.py，eona.py，Tncb.py（另一小组的引擎）三种 AI 分别进行**三局对弈**和**十局对弈**，以下是对弈结果：(时间上限统一设为一小时，每步限时 60s)。

三局对弈：

1. 对阵 eona 完败：

2. 对阵 simple，小胜一局：

3. 对阵 Tncb，打平：

十局对弈：

1. 对阵 eona 依然完败。

2. 对阵 simple：

3. 对阵 Tncb，依然打平

References:

[1] 蒙特卡洛树搜索 – 维基百科

https://zh.wikipedia.org/zh-hans/%E8%92%99%E7%89%B9%E5%8D%A1%E6%B4%9B%E6%A0%91%E6%90%9C%E7%B4%A2

[2] 《蒙特卡洛树搜索+深度学习 – AlphaGo 阅读笔记》

http://blog.csdn.net/dinosoft/article/details/50893291

[3] 《蒙特卡洛树搜索 MCTS》 http://www.jianshu.com/p/d011baff6b64