四子棋对抗作业实验报告

实验思路

1、使用极大极小搜索树并配合 $\alpha - \beta$ 剪枝。

2、使用蒙特卡洛搜索树结合UCB信心上限算法。

实验过程

由于不太清楚如何选择估值函数以及一些先验知识，最终选择UCT算法，即在蒙特卡洛树中运用UCB算法。

UCT算法原理

算法 3：信心上限树算法（UCT）

function UCTSEARCH(s_0)
以状态s_0创建根节点v_0;
while 尚未用完计算时长 **do**:
$v_l \leftarrow$TREEPOLICY(v_0);
$\Delta \leftarrow$DEFAULTPOLICY(s(v_l));
BACKUP(v_l, Δ);
end while
return a(BESTCHILD($v_0, 0$));

function TREEPOLICY(v)
while 节点v不是终止节点 **do**:
if 节点v是可扩展的 **then**:
return EXPAND(v)
else:
$v \leftarrow$ BESTCHILD(v, c)
return v

function EXPAND(v)
选择行动$a \in A(state(v))$中尚未选择过的行动
向节点v添加子节点v'，使得$s(v')= f(s(v), a), a(v') = a$
return v'

function BESTCHILD(v, c)
return $argmax_{v' \in children\ of\ v} \left(\frac{Q(v')}{N(v')} + c\sqrt{\frac{2ln(N(v))}{N(v')}} \right)$

function DEFAULTPOLICY(s)
while s不是终止状态 **do**:
以等概率选择行动$a \in A(s)$
$s \leftarrow f(s, a)$
return 状态s的收益

function BACKUP(v, Δ)
while $v \neq NULL$ **do**:
$N(v) \leftarrow N(v) + 1$
$Q(v) \leftarrow Q(v) + \Delta$
$\Delta \leftarrow -\Delta$
$v \leftarrow v$的父节点

上图是UCT算法的伪代码，当某个节点存在尚未扩展的子节点时，创建未扩展的节点；当某个节点所有子节点都已扩展，则按照UCB的策略进行选择最优的子节点。通过TreePolicy选出目前整棵树中当前的叶子节点，并在该节点进行模拟游戏直到终局分出胜负，然后将此次游戏结果向上传播给父节点以及祖先节点。这就算完成一次搜索，在设定的时间限制内重复该搜索，最后返回不考虑探索次数情况下（即UCB算法中的C值取0）根节点的最优子节点。

算法实现

通过伪代码，可以抽象出状态节点的类，而UCT算法也可以封装成一个类。C值选取的是0.8，时间设置为2300ms。

优化

1、初始版本由于每一次调用UCT算法进行搜索都要重新创建一个根节点并进行搜索。后来觉得上一次决策时的模拟结果就全都浪费掉了，应该可以保留对手所下位置的对应的那个分支，因此就把UCT算法改成了一个**单例模式，在整个对弈过程中只有一个实例，只有一颗搜索树。每次轮到我方落子时，删除无用的子节点，然后移动根节点到对应的子节点。**在我方选好落子的时候，同样要删除无用子节点，移动根节点。

实现如下：

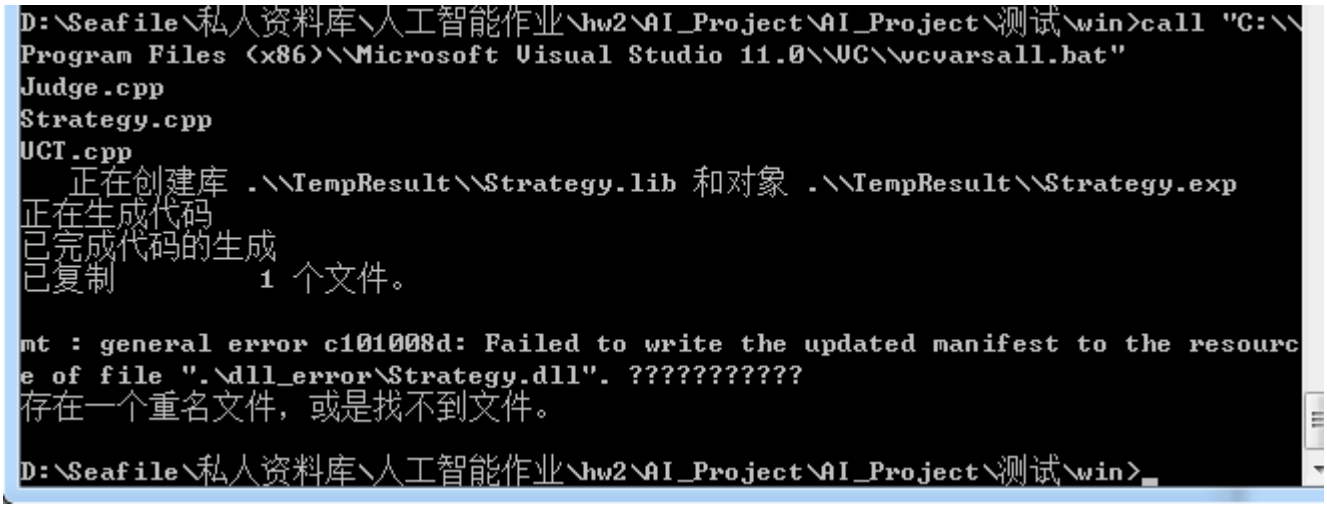
```
//删去子节点落子为(x,y)以外的节点，并移动根节点
void UCT::chopBranches(int x, int y){
    StateNode* nextroot = this->root->deleteOtherBranches(x, y);
    if(nextroot != root){
        delete root;
    }
    nextroot->parent = nullptr;
    root = nextroot;
}
StateNode* StateNode::deleteOtherBranches(int x, int y){
    StateNode* retNode = this->child[y];
    for (int i = 0; i < this->N; i ++){
        if (this->child[i]) {
            if( y != i){
                this->child[i] -> clear();
                delete this->child[i];
            }
        }
    }
    //如果x,y对应的节点为空指针，说明该分支未被模拟过，则重新初始化根节点，并返回此节点
    if(retNode == nullptr){
        ....
        ....
        ....
    }else{
        ....
        ....
        ....
    }
    return retNode;
}
```

遇到的问题

1、考虑到在比较深的节点模拟成功，意味着步数太多，就容易给对手机会，因此觉得应该浅层时的胜利的收益更大。于是尝试在向上传播的函数中，修改为随着层数的增加，对上层的收益减小。但似乎起反作用。还有待研究。

2、发现存在这样的情况，在最终返回best节点时，根节点的某个子节点被探索太少次而未被选中，导致输了比赛。

3、使用测试文件夹中的compile.bat，编译时出现如下错误，vs2012版本从校园网软件中下载，但这个错误不影响生成dll文件。于是还是强行用该dll进行了测试。



4、使用compete.bat进行测试时，出现illegal step情况，经多次调试，以及检验逻辑后，并未找到错误，最后发现compete.bat调用compete.exe运行对抗，在对抗新一局时不会重启exe程序，而我的代码中UCT是一个单例，搜索树在新一局开始时不会更新，于是导致问题。因此做了修改，当判断到是新的一局游戏时，delete原root节点，新建root节点。在strategy.cpp中加入以下代码，判断是否是新一局游戏：

```
    //判断是否是新一局游戏
    int steps = 0;
    int newgameflag = 0;
    for(int i = 0; i < N; i++){
        steps += (M - top[i]);
    }
    if( noX == (M-1)){
        steps -= 1;
    }
    if(steps == 0 || steps == 1){
        newgameflag = 1;
    }
```

最终对阵结果：

最终手动开UI运行100次对抗的结果为 75胜25负，其中先手输给了40、66、74、78、82、84、86、88、92、98、100，后手输给了22、24、30、40、54、62、74、84、86、90、92、94、98、100。

错过ddl，决定优化一下

1、在观察AI对抗的时候发现，自己的ai经常不去防守对方必胜的落子点，而且考虑到对于已决出胜负的终节点，棋权为AI且AI存在必胜落子点，那就把这一落子点对应的节点的收益设置为最大，同时把该节点的父节点的收益设置为最小，意味着父节点的落子是必败的落子点。若某一方落子后，另一方的任何落子位置都必败，那个这个落子是必胜节点也可设置为收益最大。因此在defaultPolicy函数中加入以下代码：

```
        if(node->whoseNode == COMPUTER_NODE){
            if(node->nodestate == COMPUTER_WIN){
                node->profit = 1e14;
                node->parent->nodestate = COMPUTER_WIN;
```

```
                node->parent->profit = 1e14;
            }else if(node->nodestate == HUMAN_WIN){
                StateNode* parent = node->parent;
                int flag = 1;
                if(parent->canExpandNUms == 0){
                    for( int i = 0; i < this->N; i++){
                        if((parent->child[i]!=0)&&(parent->child[i]->nodestate != HUMAN_WIN)){
                            flag = 0;
                            break;
                        }
                    }
                }else{
                    flag = 0;
                }
                if(flag == 1){
                    node->parent->nodestate = HUMAN_WIN;
                    node->parent->profit = -1e14;
                }
            }
        }
```

在加入以上策略后，对阵100的胜率有所上升。

2、关于C值的思考，进行微调发现设置为0.9时竟然对阵100的胜率上升了。而C值代表着倾向于探索访问过少的节点还是倾向于选择利用，因此产生了动态调整C值的想法。在游戏初期应更偏向于探索，终局以及后期偏向于利用，于是将C值乘以一个衰减系数，随回合数增加而减小，同时也设定了一个下限，不能过低。这似乎是个玄学调参的过程，对战所有文件的统计结果并不佳。

3、进行了代码发现在每一个rand()函数前都调用了srand(int(time(nullptr)))，意味着一秒内的每一次rand()都是返回同样的值，这导致算法根本不随机，一秒内重复探索同一个位置。导致了之前胜率不高的问题。

最终提交结果：

在修复了bug后达到了如下成绩：

id	wina	winb	losea	loseb	tie	bug	debug	illegal	deillegal	timeout
2019211353	50	48	0	2	0	0	0	0	0	0	0	0

在查看每个compete_result文件后，认为前五项分别是学号、先手胜、后手胜、先手输、后手输。

也就是这一次结果为：先手胜50 后手胜48 后手输2，即 胜98， 平0，输2

总结：

两大改进对胜率有不错的提高：

1、每次轮到我方落子不是新建树，而是沿着树走到对应的节点，可以不丢失之前的模拟探索数据。

2、对于必胜节点的收益，引导ai往必胜节点走。分别有两种情况：

（1）、若某一个落子是必胜（必败）节点，则父节点状态改为必败（必胜），同时节点收益设为非常高（非常低），父节点收益设置为非常低（非常高）。

（2）、若某一个节点的所有子节点都是必胜（必败），则该节点设置为必败（必胜），节点收益设置为非常高（非常低）。

在这两点改进后，可以减少在必败节点的搜索，提前判断出一些必胜分支。

进一步探索方向：

1、还是没太搞懂C值选取什么值最好。

2、极大极小$\alpha-\beta$剪枝，考虑尝试一下。在观察ai自动下棋时看出了一些规律。

3、了解一下alphaGo中蒙特卡洛的策略。