四子棋实验报告

陈张萌计74 2017013678

实验概述

1. 实现了使用UCT算法的的四子棋对战AI。
2. 对算法的性能进行了分析并做出对应优化。
3. 最终版本采用：使用UCT算法+系统调用优化+随机优化。
4. 在OJ和本地分别对给出的50个dll分别采用先、后手进行测试，OJ胜率94%，本地胜率97%。

实验原理:UCT算法

在算法的选择上，有$\alpha-\beta$剪枝和蒙特卡洛两种算法。$\alpha-\beta$剪枝算法需要先给定一个对四子棋局面的评估函数，但在本次实验当中，四子棋并不是所有的地方都可以落子，因此人工给出估价函数是一个比较困难的事情。而蒙特卡洛方法采用的是多次模拟、选取胜率最大节点的方法，并不需要关于四子棋的知识。UCT算法是对蒙特卡洛方法的扩展，引入了信心上界来对局面进行评估：

$$I_j = \overline{X_j} + C * \sqrt{\frac{2log(n)}{T_j(n)}}$$

算法流程为：

```
Point* UTC::utcRoutine(float time_limit) {
    //为了使算法每次执行产生的随机数都不同
    srand((unsigned)time(NULL));

    clock_t start_clock = clock();
    int round = 0;
    int cnt=0;
```

```
    cerr<<"before while..."<<endl;
    while (USED_TREE_NODE<upper_Used&&round<ONE_ROUND_COUNTS) {
        //判断是否还要继续循环：未超时且实现开辟的树节点尚未使用完
        //为了减少clock()函数引发的系统调用次数，每运行2048次再进行一次时间判断
        round++;
        if(round>=ONE_ROUND_COUNTS){
            round=0;
            cnt++;
            if(!inTimeLimit(start_clock, time_limit)){
                break;
            }
        }
        resetBoard();//初始化
        auto node = treePolicy();//选择要扩展的节点
        float score = defautPolicy();//模拟并得到本次模拟的收益
        backUp(node, score);//回溯更新
    }
    int y=decideSolution();
    int x=board[top[y]-1][y]==Board::empty?(top[y]-1):(top[y]-2);

    //返回一个point类，为本次落子的地点
    return new Point(x,y);
}
```

代码实现

1. 将算法封装在了UCT类当中，对外部提供接口

   `Point* utcRoutine(float time_limit = DEFAULT_TIME_LIMIT);`

   返回一个Point类，为本次落子的地点。
2. 在类内部避免了内存泄露问题。
3. 由于每走一步搜索都要调用一次UCT，因此在uct.cpp中申请局部变量用于存放uct树的节点，等到整个函数运行结束时统一释放这些空间，这样可以省去很多重复申请、释放内存的工作。

实验数据分析与优化

uct局面评估优化

在进行局面评估时，可以想到：如果下一步是必胜局面，这时就应当适当提高局面评估分数（参数：`WIN_ACCELERATE`）；并且距离必胜局面步数越少评估分数应当越高，反之步数越多评估分数也应当越少。

因此对局面评估参数进行优化：

```
int score = count==1? WIN_ACCELERATE : 1;
```

如果只走了一步就到达必胜局，那么这局得到的分数会更高。

相应地，在回溯更新函数当中，也对score进行处理，使得随着结点向上更新，score的值的大小不断减少，即：减少必胜局面的影响（如果不做此步骤，相当于对于所有节点的估值都放大了固定倍数，并不会对结果有什么影响）。

测试发现，进行局面评估优化后的胜率有明显提升。

uct系统调用优化

在uct当中，由于使用clock函数获取时间要进行系统调用，该步骤会消耗大量的时间，因此希望能**减少系统调用次数**。具体解决方法为：每进行 ONE_ROUND_COUNT 次循环之后再进行一次系统调用来判断是否超时，程序中取ONE_ROUND_COUN=2048 。

但是进行系统调用优化会产生新的问题：超时现象显著增多。原因比较容易理解：由于循环一定次数之后才进行时间判断，当棋盘较大时，每次搜索深度都会较大。当棋盘较大时，就可能出现：上一次判断时间还未超时，但是再进行2000多次搜索之后就会超时的现象。

因此调整timelimit参数，从较紧的2.9s调整至2.5s。进过测试，极大地减少了超时的概率；而且先手、后手胜率更为稳定。由于timeout次数有明显减少，总的胜率有明显提升。

uct策略进一步优化

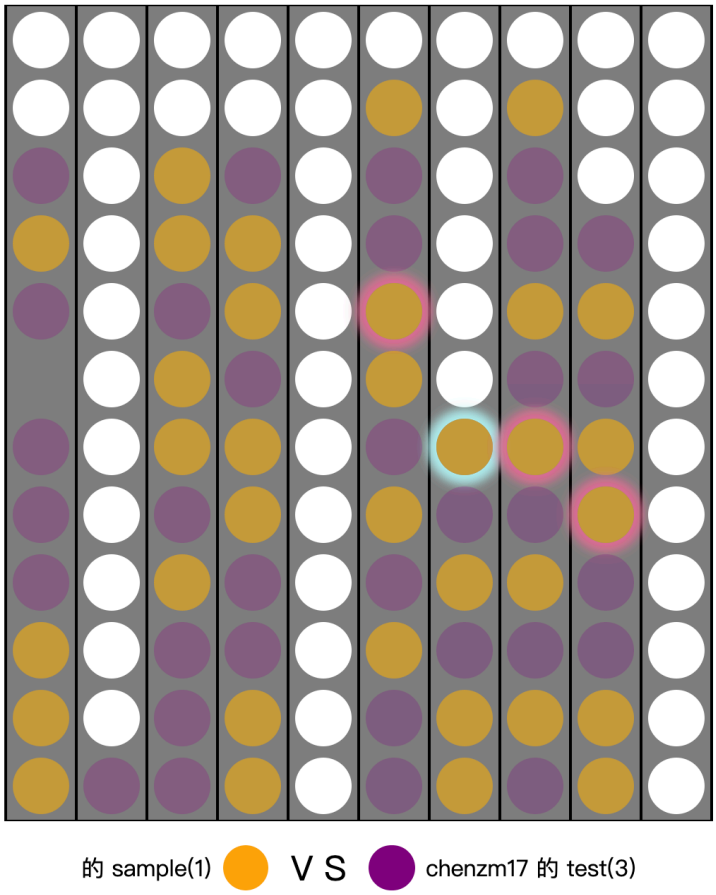
在网站上进行测试并观察结果，发现失败的对局存在一定规律，即在某一时刻会到达对手必胜局面.(如下图所示)出现这个局面的原因在于：

- 一是对局前期查找可行落子点是倾向于从左向右查找，随机性并不强
- 二是没有足够重视对对方落子的“堵”，即最初每个落子点的概率都差不多时，在对方落子点的附近的棋应当给到更多的落子机会

根据此策略进行进一步优化：

- 在向下搜索时，对每一列采用随机顺序进行搜索
- 在搜索结束，选取最终结果时，距离上一步落子点越近的点的value放大一定倍数。

```
float UTC::getLamda(int line){//第line列的value放大到(1+getLamda(line))倍
    return max((float)(0),((float)abs(2-(line-lastY))/2)*LAMDA);
}
```



部分本地测试结果：

策略	C1	C2	系统调用	random	局面评估	timelimit	wina	winb	losea	loseb	tie	timeout
1	0	1	是	否	否	2.9	39	39	6	5	1	8
2	0	1	是	否	award=5	2.9	47	41	2	8	0	2
3	0	1	是	否	award=5	2.5	45	44	3	4	0	0
4	0	1	是	是	award=5	2.7	44	42	6	8	0	0
5	1	1	是	是	award=5	2.7	41	42	9	8	0	0
6	0.7	1	是	是	award=10	2.7	38	37	10	11	0	0
7	0.5	1	是	是	award=10	2.7	40	38	9	11	0	0
8	0.3	1	是	是	award=10	2.7	44	43	6	7	0	0
9	0	1	是	是	award=10	2.7	46	44	4	6	0	0
10	0	0.7	是	是	award=5	2.85	48	49	1	1	0	1

调整C的大小

C在UCT的信心上界估计函数当中对策略起到调节作用：C较小时倾向于保守，即选取平均值较高的节点作为较好的结果；C较大时倾向于探索，即考虑探索次数较少的节点。

在本次实验当中，C1表示在最终选择时的C，C2表示搜索过程当中的C。经过测试，如果只考虑正确率，发现C1越高正确率越低，C1=0时正确率最高。推测原因在于：C1控制的是最终节点选择上倾向于稳健（0）还是探索（1）。而每一步的搜索时间只有3s，并不是一个充分大的数，也就是说，信心上界算法的第二项是不能被忽略的；C1越大越有可能在最终落子的时候倾向于探索，这样就存在一定可能是C1选择了那些没有被充分探索，而且实际上效果并不好的节点。

对于C2来说，经过测试C2=0.7或0.3时正确率都较高。C2表示：在对一个节点的搜索过程中如何对节点进行评价。我们可以看到C2=0.3/0.7时并没有明显区别，C2=1是正确率稍低但是并不明显，但是当C2=0时正确率非常低。推测原因在于：C2控制的是搜索过程当中的节点评价策略，如果在搜索过程当中过分保守（C2=0）则有很多节点无法搜到，就会较明显地影响对局面的判断。

版本最终选择

将不同的参数进行调整，在网站上再次进行评测，结果如下（去掉调试输出）：

策略	C1	C2	系统调用	random	局面评估	timelimit	win	lose	tie	rank
1	0	1	是	否	否	2.7	90	10	0	90%
2	0	1	是	否	award=5	2.7	94	6	0	94%
3	0	1	是	否	award=10	2.7	87	13	0	87%
4	0	1	是	0.03	award=10	2.85	85	15	0	85%
5	1	1	是	0.03	award=10	2.7	70	30	0	70%
6	0.7	1	是	0.03	award=10	2.7	84	16	0	84%
7	0.5	1	是	0.03	award=10	2.7	84	16	0	84%
8	0.3	1	是	0.03	award=10	2.7	88	12	0	88%
9	0	1	是	0.03	award=10	2.7	89	11	0	89%
10	0	0.7	是	0.03	award=10	2.7	88	12	0	88%
11	0	0.3	是	0.03	award=10	2.7	95	5	0	95%
12	0	0.7	是	0.03	award=5	2.7	91	9	0	91%
13	0	0.3	是	0.03	award=5	2.7	91	9	0	91%
14	0	0	是	0.03	award=5	2.7	47	53	0	47%
15	0	0.3	是	0.01	award=5	2.85	90	10	0	90%
16	0	0.7	是	0.01	award=5	2.85	94	6	0	94%

根据测试效果，最终提交版本当中选择了策略号为16的AI（用户名：`Connect4_4`，测试编号`1325`）。

实验总结

1. 通过本次实验更加了解了UCT算法，理解了ab-剪枝算法和UCT算法的不同之处
2. 在算法思路等方面得到了计72班田卓钰和计71班刘丰源同学的较多帮助，包括但不限于：
    1. 代码结构和思路
    2. 对局面评估的优化：使用局面评估参数增加必胜/必败局面的评价
    3. 减少系统调用次数
3. 代码在实现上参考了以下文章：
    1. https://github.com/zhangchuheng123/Connect4
    2. https://github.com/believedotchenyu/siziqi
4. 本次实验代码量较大，在oop、树结构和debug能力等方面都有较大提升。例如：
    1. 编写单测可以在后期减少很多工作量
    2. 尽可能少在函数内部修改全局变量，包括类当中的成员变量也在尽量少的函数当中进行修改。尤其是本次实验当中的UCT类需要记录当前下棋者是谁、当前局面等，如果不知道自己调用的函数对其进行了修改，会容易出现问题。
    3. 善用git，尤其是在这种可能有多个版本的运行程序的时候会极大地方便编程