重力四子棋实验报告

叶奕宁 2019013287 计97

阅读建议：AI采用了多种优化方法，创新性主要在于 搜索剪枝(2.1)、列优化拓展(附录2)、打表(2.2)

一、实现方法

主体采用蒙特卡洛树搜索，UCB信心上限树算法，并加入多种优化方法，最终实现**天梯第二**的效果.

1.1主体框架

采用word说明中所示的算法框架。如果当前节点存在尚未扩展的子节点，就扩展并进行随机模拟；否则选择最优的子树进行蒙特卡洛搜索。当时间超过2s或者总次数超过30,0000次后返回最优解。

```
pair<int,int> UCT(time_t start_time){
        int count = 0;
        time_t nowT = clock();
        while(count < 3000000 && (double)(nowT - start_time) /
(double)CLOCKS_PER_SEC < 2.6){
            //先将 meDo,heDo,topTemp恢复根节点
            count++;
            me = meDo;
            he = heDo;
            meWin = meWinRoot;
            heWin = heWinRoot;
            for(int i = 0; i < 12; i++) topTemp[i] = top[i];
            tableNode* now = root;
            now = treePolicy(now);
```

```
            if (now->win) backup(0.0,now);
            else if (now->lose) backup(1.0,now);
            else {
                float delta = defaultPolicy(now);
                backup(delta,now);
            }

            if (count % 100 == 0) nowT = clock();
        }

        cerr << "searched: "<< count << endl;
        cerr << "total cost: " << (double)(clock() - start_time) /
(double)CLOCKS_PER_SEC << "s" << endl;

        int simC = -1;
        int resultId = -1;
        for(int i = 0; i < n; i++) {

            if (root->child[i] && root->child[i]->N > simC) {
                simC = root->child[i]->N;
                resultId = i;
            }
        }


        pair<int,int> re = make_pair(top[resultId] - 1,resultId);
        top[resultId]--;
        if(noX == top[resultId] - 1 && noY == resultId) top[resultId]--;
        assert(top[resultId]>=0);
        meDo[re.first*n + re.second] = 1;
        table(meDo,meWinRoot,re.first*n + re.second);
        cerr << "N: "<< root->child[resultId]->N << ", Q: " << root-
>child[resultId]->Q << endl;
        cerr <<"winRate: " << 100 * float(root->child[resultId]->Q) /
float(root->child[resultId]->N) <<"\%" << endl;

        return re;
    }
```

1.2 内存优化

主体代码框架需要的内存随着搜索次数的增加而增加， 当搜索次数很大时, 占用内存会很大 $(\leq 500MB)$ 。因此我选择了几种方法进行优化，

- 减少每个节点所用的空间。每个节点的空间的最大影响因素是当前状态对应的地图，这里就衍生出两种优化方法：选择在每个节点中存储当前节点的最后一次移动，或者**压位**存储地图的所有信息。
    - 前者需要通过根节点的路径复原地图， 占用时间 $\propto$ 搜索深度

- 第二种方法的好处是的时间占用少,用两个 std::bitset<m*n> 来存储地图，用 top数组 存储列顶
- 我最终采用了一个各得其长的方法：开内存池。维护全局的棋盘，每个节点通过是 father的第几个孩子来确定下到哪里，最终空间占用350MB

```
class tableNode{
public:
    tableNode* child[12];
    tableNode* father;
    bool trueMe; //这一步棋是谁下的
    bool lose;
    bool win;
    int N;
    float Q;
    char childP; //存最后一个孩子的ID

    tableNode(){
        reset();
    }

    void reset(){
        trueMe = 0;
        N = 0;
        Q = 0.0;
        lose = 0;
        win = 0;
        childP = 0;
    }
};
tableNode pool[3000000];
tableNode* use = pool;
```

- 另一个可以尝试的内存优化方法是 **哈希**。设计一个哈希表来对应每个地图对应的状态，这样的话就可以避免4步之后对应相同状态的地图多次存储浪费空间。但这会带来两个问题: 1.在backup过程中要重新 用 O(len) 的时间获取正确位置 2.这会使得 bestchild 的分数计算带来干扰，由于最后优化并不明显(也可能是代码写挂了)，最终没有采用。

1.3 搜索优化

影响蒙特卡洛算法的最重要因素是随机策略的方法、分值评估函数与调参。

1.3.1 随机方法

完全随机的模拟其实并不是贴合实际，因为“稍微有点智能的人都会知道下杀招与挡住对手的杀招”，因此我在模拟时采用半随机策略，加入如下几条规范(按优先级递减)：

- 如果下在某个点获胜，将会下在该点
- 如果对手下在某个点获胜，将会下在该点
- 如果下在某个点，对手下在上面导致获胜，将会跳过该点
- 如果下在某个点，自己下次再下在上面导致获胜，将会跳过该点(否则对手可能阻挡获胜)
- 如果第 n 条规则导致无点可选，**仅在该次**删除第n条规则

  在这种情况下，会牺牲大概10倍的搜索次数，但是效果反而会变好。当然，这些规则来源于我对于游戏的理解，更希望出现的情况是可以在不引入专家知识的情况下就获得好的效果，这仍然等待着我们的探索。

1.3.2 评估函数

传统的评估函数采用如下的形式：

$$score_V = \frac{Q(V)}{N(V)} + c \times \sqrt{\frac{2 \times \ln(N(father))}{N(V)}}$$

这种简单的形式首先我们发现可以优化，如果我们提前存储:

$$rate = \frac{Q(V)}{N(V)}, \ln = \sqrt{\ln(N(father))}, sqrt = \sqrt{\frac{1}{N(V)}}$$

三个变量，就可以通过:$score = rate_V + c \times ln_{father} \times sqrt_V$ 的简单乘法简化计算。(考虑到模拟时会多次调用 bestchild，这份常数优化其实不少)

在实际执行中，我删去了 2 这个常数，采用 $c = 0.5$ 。

二、优化方法

2.1 搜索剪枝(采用)

我们在做 bestchild 计算的时候，正常需要 12 次分数计算，然而在某些情况下，这是可以避免的。

- 1.如果该节点是必胜节点，则可以直接返回最短胜利路径对应的子节点。
- 2.如果是必败节点，返回nullptr，在蒙特卡洛的时候可以直接返回 -1 score了。
- 3.如果有子节点是必胜，转4再转1

- 4.如果搜索是发现子节点是必败节点，直接将父节点置为必胜节点，转1
- 5.如果发现搜索时发现子节点是必胜节点，continue
- 6.如果最后没有搜到可行的节点，将父节点置为必败节点，转2

很多节点都是上述类型，其实这是一个很大的优化。再就是在最开始的dp过程中如果就发现了该类节点，可以很大程度的提升搜索数量。

在模拟的过程中，这会提高搜索树的搜索效率，放弃那些明知必败或者必胜的搜索，让局势走向有利的方向。

实测这带来了大约2倍的搜索节点数目提升，尤其在后期是巨大优化。

2.2 位棋盘加速(大优化)

由于我们在随机落子中加入了规则，我们需要一种快速计算当前形势下每一个点是否是我或者敌人的方法

我们可以得知：

- 如果忽略已经落子的位置，一个点只有两种可能：是我的必胜点，不是我的必胜点
- 状态改变只会是： 非必胜点 -> 必胜点
- 对于一个落子，只会影响周围24个点是否变成必胜点。
- 四个方向是对称的，每个方向只有 $2^6 = 64$ 种可能
- 不可落子点只会影响它本身，成为非必胜点(因为本身不能走别的棋，可忽略之)

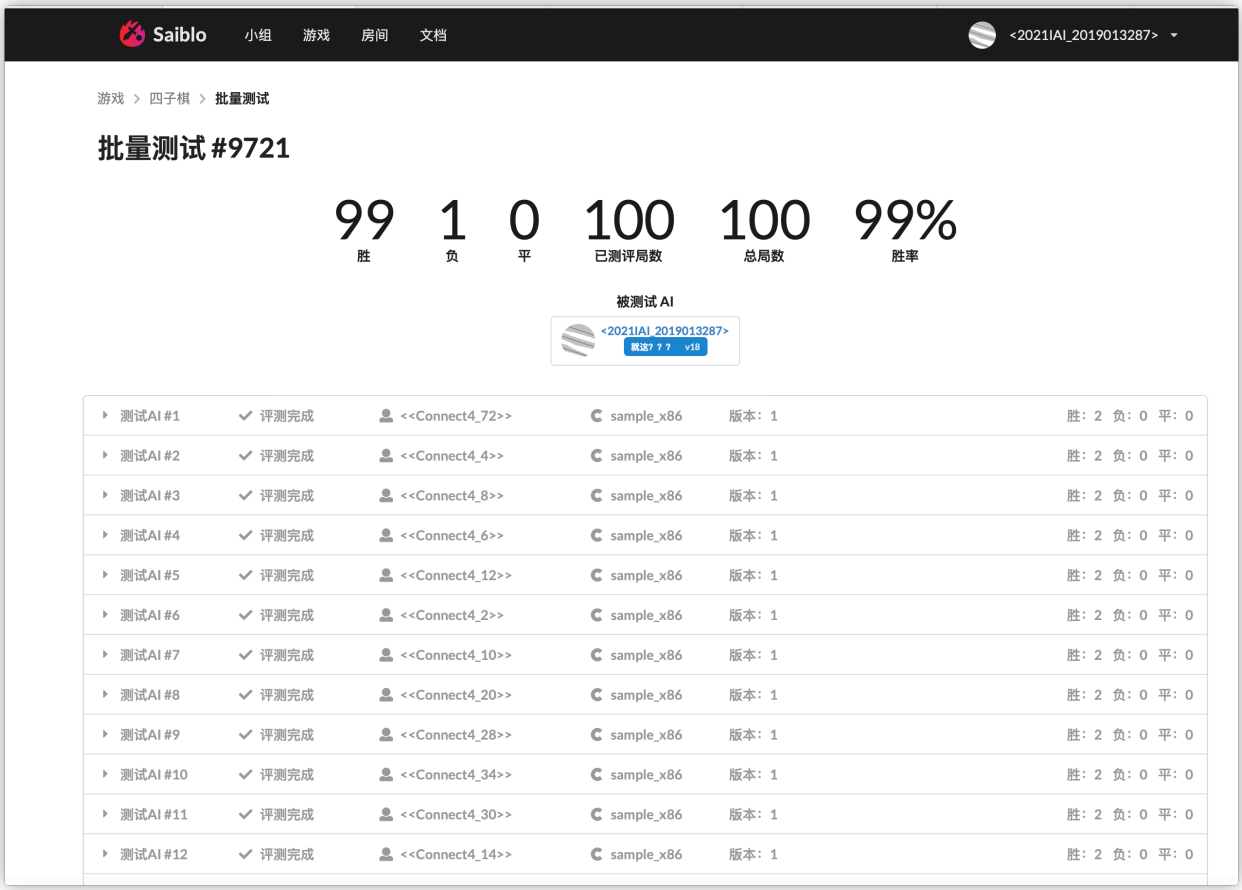
因此我就想到了打表：

- 提前先打一个 64的小表，代表每个方向的所有情况
- 根据前面的小表打一个 4 * m * n * 64 的大表，对每个位置，每个方向，每种情况都预处理好。
- 操作时取出四个方向的点二进制id，做四次 bitset::|=, 再根据不可落子点位置置为0。
- 由于bitset是连续内存，常数很低，每次操作总常数大约是24，但可以维护全局所有的必胜点(双方)，对随机规则下的落子速度带来超大的提升。

三、表现

3.1 AI表现

在天梯排行第二，和2-100号人机对战情况如下：



赛100场，胜99场，其中一场tle，不知道是否是评测机故障……

最终内卷版本采用这些优化：5条随机规则、内存池、全局位棋盘维护、大打表、必胜必败点优化、-Ofast编译命令

在打表魔法的加持下，本机做到了2.5s搜索60万次，服务器2.5s搜索30万次，衰减一半，可能源于单核性能。

3.2 作业经验

- 什么花里胡哨东西都不如多搜索一些节点。在模拟中加入规则本质上是为了在模拟次数少的时候得到更真实的结果。
- 对于位棋盘的优化可以借鉴,我由此总结出了打表预处理，不知道后来的同学能不能完善出更多神奇优化（

  > https://www.xqbase.com/computer/struct_bitboard.htm

- 本机和服务器差距很大，如果为了卷天梯，也许可以针对服务器的情况做优化，而不是针对算法做优化...后面懒得卷了就没优化……
- 我非常看好附录中的列优化方法(它是模拟规则的推广)，但没有找到一个好的权衡方法来规则数和搜索数，不知道后来的同学能不能从我这里借鉴一点经验，发展出一个超级优化。

- 打表方法还有一些改进空间：能不能在更快时间内获取id？位运算是否有更快的实现方式？(在这个地方的优化是最重要的，因为rollout占了绝大多数时间)
- 内存池可以加快新建节点的方法，但做不到根节点移动，利用上一回合的已有知识开始下一回合的搜索。不知道有没有好的方法可以同时利用内存池和根移动两个优化
- 编译命令 开 -Ofast...真的fast...
- 如果——时间很多，并且想得到超级提升——也许可以写一个简单的神经网络进行rollout。通过自我对弈进行训练：这是从alphaGo得到的思路，但我没有写(幸亏四子棋不让用python，否则不知道同学们要卷出什么魑魅魍魉……)

四、附录：未采用的优化

1. 算杀(未采用)

在很多情况下，蒙特卡洛的不确定性需要配合一些确定性的参数。在这种情况下，我对算杀进行了判断。我对节点做出了如下分类和定义：

- 必胜节点：如果有一个儿子是必败节点，则是必胜节点(或)
- 必败节点：如果所有儿子都是必胜节点，或者自己对应的状态为负，则是必败节点(与)
- 最少必胜路径长：如果节点时必胜节点，路径长定义为**走到胜利的终结状态所用的最少步数**
- 最长必败路径长：如果节点时必败节点，路径长定义为**走到输棋的终结状态所用的最多步数**

  将节点进行分类后，我们就可以进行算杀的判断。在每个回合开始时，进行dp

```
void dp(mctNode* start,int deep, int simC){
    //if (start->bisheng || start->bibai) return;
    if(!start) return;
    start->bisheng = 0;
    start->bibai = 0;
    start->jiangsi = -1;
    start->jiangsiMin = 100;  //希望最小
    start->beijiangsi = -1;
    start->beijiangsiMin = -1; //希望最大
    if (start->lose == 1) {
        start->bibai = 1;
        start->endN = 1;
        start->beijiangsiMin = 0;
        start->beijiangsi = 0;
        return;
    }
    if (deep == 0) {
        return;
    }
    bool WIN = 0;
    bool LOSE = 1;
    for(int i = 0; i < n; i++){
```

```
            if(start->child[i] == nullptr) {
                start->child[i] = expand(start,i);
            }
        }
        for(int i = 0; i < n; i++){
            dp(start->child[i],deep-1,simC);
            if (start->child[i] && (start->child[i]->bibai || start-
>child[i]->lose == 1)) {
                if (start->child[i]->beijiangsiMin + 1 < start->jiangsiMin)
{
                    start->jiangsiMin = start->child[i]->beijiangsiMin + 1;
                    start->jiangsi = i;
                }
                WIN = 1;
            }
            if(start->child[i] && (start->child[i]->bisheng != 1)) LOSE =
0;
        }
        start->bisheng = WIN;
        start->bibai = LOSE;

        assert(!WIN || start->jiangsi != -1);
        if (LOSE) {
            //选被将死里面苟的最久的
            for(int i = 0; i < n; i++){
                //if (start->child[i]) assert(start->child[i]->jiangsiMin
!= 100);
                if(start->child[i] && (start->child[i]->jiangsiMin + 1 >
start->beijiangsiMin)) {
                    start->beijiangsiMin = start->child[i]->jiangsiMin + 1;
                    start->beijiangsi = i;
                }
            }
            assert(start->beijiangsi != -1);
        }
        if(LOSE || WIN) start->endN = 1;
        //if(start->bibai) start->printNode(m,n,noX,noY);
    }
```

该函数可以算出 n 步之内所有的杀招(实际评测**基本上杀招都在6步以内**)。由于胜负手的先后顺序与优先级差异(杀招 > 防守)，维护中序遍历会得不偿失，因此最终考虑递归的形式。值得注意的是算杀需要额外的时间和空间占用，每个节点要额外存储路径长和胜负状态。

第二件事是，在最终输出结果时，如果根节点是 必胜节点 或者 必败节点，则不用继续搜索，直接输出：

- 如果是必胜节点，走最短胜利路径对应的子节点
- 如果是必败节点，走最长失败路径对应的子节点
- 定义“路径长”这个概念是因为在实际检测中发现面对很蠢的智能体时算杀反而会成为障碍(算法会舍弃2步杀招而采用6步杀招导致罚站等等...)，再就是必败节点面对低级智能体时用最长必

败路径也许可以蒙混过关(如果他算不到6步的杀招...)

最终没有采用算杀方法，实测后文的 **搜索剪枝** 方法要更优，且可以推出算杀。

2. 列优化——本部分由 郭昊(2019013292) 和我共同探索和完善(部分采用)。

在对于局势的研究中，我们发现和100号AI对战中总是出现 算杀失败的情况。就是有一些列不能走，但最后棋盘下满了以后无路可走只能送死。经过研究，这种趋势都是在**布局阶段**就被确定下来了，也就是说，早在开始下棋10几步的时候，往往胜负已定 (因为有前面说的算杀优化的情况，在后期之前就被将死的概率很低)。

我们希望找到一种方法或者得分能实现下面的作用：

- 在现在的局势下，选定某一列。
- 对于我先下和对面先下的两种情况，只考虑在一列向上走，看哪边胜利的次数多。
- 如果1：1或者0：0，则赋予 0分
- 如果1：0或者0：1，则赋予± 1 分
- 如果2：0或者0：2，则赋予 $\pm n$ 分
- 前面提到的 **搜索优化** 可以被列优化推出，这代表着，在蒙特卡洛中加入列优化的随机，平衡运行速度以后，将会带来极大地搜索提升。

可以看到，对每一列都计算列得分之后，列得分越高，则后期的优势越大，我们想要探索一种快速的、计算列得分的算法。

该算法伪代码如下，该算法总体是 $O(n^2)$ 复杂度但常数较大。

```
float total = 0.0
for 每一列:
  int x1、x2、y1、y2 //存储 me 和 you 的第一次奇数胜利和偶数胜利的位置
  while 当前位置 > 0:
    分奇偶讨论(假如当前当前位置高度是奇数)
    if 如果我走这里赢 && 我的 x1> 当前位置:
        重置x1为当前位置
    if 如果对方走这里赢 && 他的 y2> 当前位置:
        重置y2为当前位置
    //我的奇数步代表他的偶数步
    当前位置向上一格

  int ji = 0, ou = 0
  if x1 < y1:
      ji = 1
  else if x1 > y1:
      ji = -1
  if x2 < y2:
      ou = 1
  else if x2 > y2:
```

```
        ou = -1

    if ji == 1 && ou == 1:
        total += n
    else if ji == -1 && ou == -1:
        total += -n
    else if ji + ou == 1:
        total += 1
    else if ji + ou == -1:
        total += -1
```

注意，我们可以在$O(1)$时间内可以判定是否赢棋(只判断最后一格走的位置能不能成)

加入列得分优化后我们对于子节点的 bestchild中对子节点的列得分加一个权重放入 蒙特卡洛 的算式里面。这带来几种不同的影响：

- 如果有一列的列得分是n的话，n取多大为好？(和列数n正相关是否合适？)
- 总体的权重α 取多少合适?
- 如果只选择列得分是2加入计算，即忽略± 1 的得分。
- 是否考虑α 加入一个递减的权重，即在布局阶段更看重列得分，而在中期和收官阶段则不再关注。

我们可以如下考虑：加入别的列都走完了，接下来只能走 **列得分 != 0** 的列,且仅剩余1列：

- 如果列得分是2，则我必胜
- 如果列得分是1，则我有75%概率获胜(因为无法选择我先走还是该他先走)
- 列得分是0，则胜率50%
- 反之亦然

然而，实际情况是往往剩余几列，我们可以确定如下几件事：

- 对方一定不先走列得分是n的列(如果在2步处就有杀)
- 我方不先走列得分是-n的列(如果在2步处就有杀)
- 随着下棋的进行，列得分可能会逆转

这告诉我们：

- 在后期考虑列得分也是有必要的
- 列得分是$\pm n$ 的列比较稳，尤其是杀招出现在上第二格。

从最终结果来看，列得分计算对于先手方比较重要，因为先手有布局优势，在加入列得分以后，尤其是加大 n 的值以后，先手方的胜率出现较大提升$(\simeq 100\%)$。

然而，搜索次数比较难以平衡，最终选择了在 1.3.1 中提到的5条规则基础上额外添加了 仅向上三格的必胜列策略。