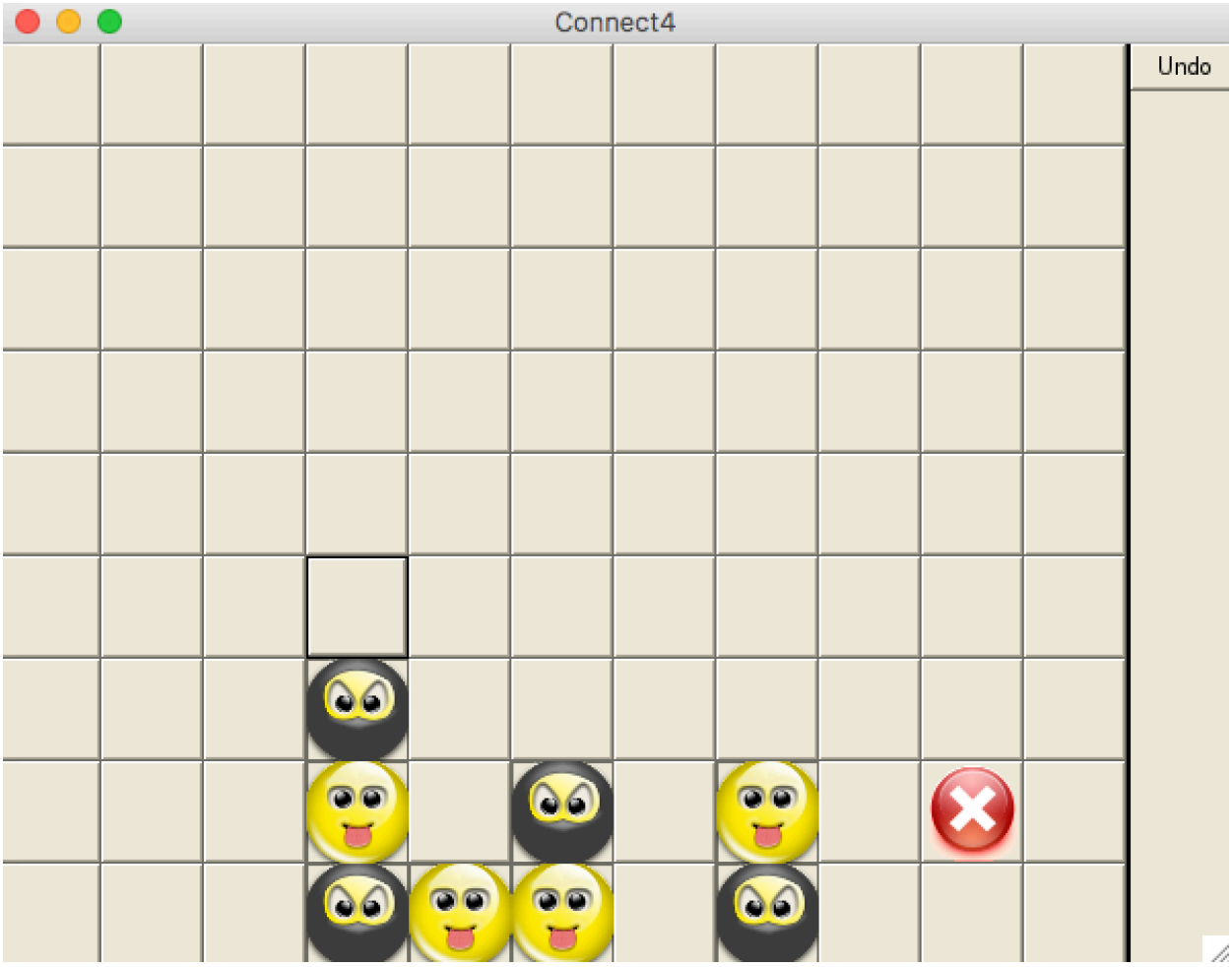
人工智能导论——重力四子棋实验报告

张钰晖 2015011372 Tel: 185-3888-2881 Email: yuhui-zh15@mails.tsinghua.edu.cn

一、实验描述

本实验要求在给定框架下实现一个重力四子棋的AI，实验中对四子棋规则做了一定的扩展，即随机确定棋盘大小以及在棋盘上生成不可落子点。实验要求对代码进行一定的封装，对于每次传入的棋盘状态后返回一个落子点，每步决策时间限为3s。



二、实验分析

在实验中我选择了蒙特卡洛搜索和信心上限树算法，评测结果较为理想，以下简单就本实验算法选择进行一定分析。

对抗搜索有多种实现，其中比较经典的是alpha-beta剪枝算法和蒙特卡洛搜索算法。

在重力四子棋实验中，使用alpha-beta剪枝算法的瓶颈在于非常依赖于局面评估的准确性。由于棋盘大小不定并且有不可落子点，要找到能够很合理地反应当前局面的估价函数并不容易，则算法瓶颈落在了设计部分上，体现不出机器的优势，要实现较好的改进也并不容易。

而蒙特卡洛搜索算法则不存在这样的瓶颈问题，蒙特卡洛搜索算法的基本思路是对每个点进行随机模拟落子直至比赛结束，选胜率最高的点作为最终返回的落子点。在模拟点数量较多时，算法会逐渐收敛到当前最优解。

相比alpha-beta剪枝算法，蒙特卡洛算法可以看作用机器的优势来和人对弈，在算法设计部分只需要做到均衡算法过程中的模拟方向，使每个点都能有一定量的模拟，从而使结果更可靠。从以上分析不难看出，蒙特卡洛算法成功与否主要取决于能否在给定时间内进行较多次数的迭代。

虽然利用构造蒙特卡罗搜索算法可以解决本问题，但是由于蒙特卡罗搜索方法在没有知识的指导时树的扩展层数较少，不利于最优解的获取，因此将UCB1算法加入蒙特卡罗规划树的构建过程中，形成了信心上限树算法(UCT)。UCT算法中，可落子点的选择不是随机的，而是根据UCB1的信心上界值进行选择，UCT算法会结合获取的信息，在探索和利用之间找到平衡。

需要指出的是，蒙特卡洛算法在出解速度上远逊于alpha-beta剪枝，如果搜索不充分结果也不稳定。如果alpha-beta剪枝算法可以设计出一个很好的估价函数，效果可能好于蒙特卡洛算法。

三、算法流程

1、语言描述

①初始化信心上限树UCT，以当前棋局状态和顶端状态初始化整棵树的根节点，代表当前棋局状态；

②如果搜索时间大于时间上限，结束搜索，跳至6；

③若当前节点为终止节点，跳至4。如果当前节点为可扩展（选择尚未选择的行动），那么对当前节点进行扩展，否则将当前节点设为从当前节点的子节点中选取的最优节点（根据UCB算法），跳至3；

④对当前节点进行随机模拟对局，直至游戏结束（WIN或TIE），计算收益值；

⑤向上更新祖先节点的收益值和访问次数，跳至2；

⑥从根节点的子节点中选取最优节点（胜率最大点），将该点作为最终返回节点。

2、伪代码描述

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 ←TREEPOLICY(v_0);
Δ←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

符号解释:

s表示状态，即棋盘信息，s(v)为节点v所对应的状态，s0为初始状态，v表示节点与状态s一一对应。

Δ表示单次收益，即当前状态按某一策略进行到棋局结束时的胜负情况

Q(v)表示节点v的总收益，即模拟多次Δ的和

N(v)表示节点v被访问的次数

A(s)表示状态s的可行动集，即状态s下所有合法落子位置的集合

a(v)表示某一行动方式

信心上限值UCB1算式：

前一项：收益Q/访问次数N=胜率

后一项：c为比例系数，控制该项在整体估计中的重要程度，保证在博弈树规模尚小时，某一节点v不会因为在当前经过模拟累积的收益值较小而不被选择，随着节点深度的加深，这一项的影响会越来与小，最终的信心上限值将主要依赖于前一项胜率

前一部分是对到目前为止已经搜集到的知识的价值，而后一部分则可以看作是尚未充分探索过的节点需要继续探索的必要性。

四、实现细节

1、Node类

Node类是UCT树的节点，主要集合了节点的信息。比如上一步落子位置、上一步执子者、节点访问次数、节点收益、孩子、父母、可扩展列编号等信息。

```
struct Node {
    enum Chessman { EMPTY = 0, PLAYER = 1, COMPUTER = 2 } _chessman;//上一执子者
    int _X, _Y;//上一落子点
    int _visitedNum;//节点访问次数
    int _expandableNum;//可扩展节点数
    int _profit;//节点收益
    int* _expandableNode;//可扩展列编号，一维数组
    Node** _children;//孩子，一维指针数组
    Node* _parent;//父母
    //构造函数，需传入上一落子点、上一执子者、父母
    Node(int X, int Y, Chessman chessman, Node* parent);
};
```

2、UCT类

UCT类是算法的核心，是一棵信心上限树。和伪代码不同的是，Node类并没有存储当前的棋盘状态和顶端状态，而是在UCT类中保存一个初始棋盘和一个新棋盘，每沿着树走一条边便更新一下新棋盘，这种实现方式大大降低了内存使用情况，同时也大大提高了效率，搜索效率非常高。

```
class UCT {
private:
    enum { PLAYER_WIN = -1, COMPUTER_WIN = 1, TIE = 0, UNDETERMINED = 2 };//获胜、平局、不确定三种棋局状态
    Node* _root;//UCT的根节点
    int** _chessboard;//初始棋盘状态
    int* _top;//初始顶端状态
    int** _newChessboard;//新棋盘状态
    int* _newTop;//新顶端状态
    int _M, _N;//棋盘大小为M(row)*N(column)
    int _noX, _noY;//棋盘不可落子点为(noX,noY)
public:
    UCT(int M, int N, int noX, int noY, int** chessboard, int* top);//构造函数，需传入棋盘大小、不可落子点、棋盘状态和顶端状态
    ~UCT();//析构函数
    void Reset();//复原棋盘状态至初始状态
    void Init(Node* node);//初始化节点node
    void Clear(Node* node);//清空以node为根节点的子树
    bool IsExpandable(Node* node);//判断节点node是否可以扩展
    int IsTerminal(Node* node);//判断节点node是否到达终点(WIN或TIE)，同时返回对应enum类型
    Node::Chessman ChangeChessman(Node::Chessman chessman);//返回下一执子者
    Node* TreePolicy(Node* node);//返回按照搜索树策略对节点node扩展的节点
    Node* Expand(Node* node);//返回对节点node扩展的节点
    Node* BestChild(Node* node);//返回按照UCB1规则节点node最优的孩子
    int DefaultPolicy(Node* node);//返回按默认策略随机下到终点(WIN或TIE)节点node的收益值
    void BackUp(Node* node, int deltaProfit);//向上回溯更新收益值
    Node* Search();//返回给定棋盘状态和顶端状态进行搜索的最优结果
};
```

绝大部分代码都按照伪代码描述的方式实现，贴出和伪代码对应的实际代码如下，读者可对照伪代码进行阅读，基本完全相同。

```
//返回给定棋盘状态和顶端状态进行搜索的最优结果
Node* UCT::Search() {
    int startTime = clock(), endTime;//起止时间
    int searchNum = 0;//搜索次数
    //创建根节点
    _root = new Node(-1, -1, Node::PLAYER, NULL);
    Init(_root);
    //当时间没有结束，进行搜索并更新
    while (true) {
        if (searchNum % 5000 == 0 && ((endTime = clock()) - startTime) / (double)CLOCKS_PER_SEC > TIME_LIMIT) break;//每5000次搜索计一次时间
        Reset();//初始化棋盘为传入状态
        Node* node = TreePolicy(_root);
```

```
        int deltaProfit = DefaultPolicy(node);
        BackUp(node, deltaProfit);
        searchNum++;
    }
    //返回最优解
    Reset();
    return BestChild(_root);
}

//返回按照搜索树策略对节点node扩展的节点
Node* UCT::TreePolicy(Node* node) {
    while (IsTerminal(node) == UNDETERMINED) {
        if (IsExpandable(node)) {
            return Expand(node);
        }
        else
            node = BestChild(node);
    }
    return node;
}

//返回对节点node扩展的节点
Node* UCT::Expand(Node* node) {
    int r = rand() % node->_expandableNum;//随机一列
    //向随机出的方向落子
    int newY = node->_expandableNode[r];
    int newX = --_newTop[newY];
    Node::Chessman chessman = ChangeChessman(node->_chessman);
    _newChessboard[newX][newY] = chessman;
    if (newY == _noY && newX == _noX + 1) --_newTop[newY];
    //建立孩子节点
    node->_children[newY] = new Node(newX, newY, chessman, node);
    Init(node->_children[newY]);
    std::swap(node->_expandableNode[r], node->_expandableNode[--node->_expandableNum]);
    return node->_children[newY];
}

//返回按照UCB1规则节点node最优的孩子
Node* UCT::BestChild(Node* node) {
    //初始化，注意把maxProfit初始化为最小值
    Node* best = NULL;
    int bestIndex = 0;
    double maxProfit = -RAND_MAX;
    for (int i = 0; i < _N; i++) {
        if (node->_children[i] == NULL) continue;
        //根据节点执子者判断正负
        int childProfit = (node->_chessman == Node::PLAYER? 1: -1) * node->_children[i]->_profit;
```

```
        int childVisitedNum = node->_children[i]->_visitedNum;
        //UCT计算方法
        double tempProfit = (double)childProfit / (double)childVisitedNum + sqrt(2.0 * log((double)node->_visitedNum) / (double)childVisitedNum) * COEFFICIENT;
        //更新最优孩子
        if (tempProfit > maxProfit) {
            maxProfit = tempProfit;
            best = node->_children[i];
            bestIndex = i;
        }
    }
    //向最优节点落子
    int newY = bestIndex;
    int newX = --_newTop[newY];
    Node::Chessman chessman = ChangeChessman(node->_chessman);
    _newChessboard[newX][newY] = chessman;
    if (newY == _noY && newX == _noX + 1) --_newTop[newY];
    return best;
}

//返回按默认策略随机下到终点(WIN或TIE)节点node的收益值
int UCT::DefaultPolicy(Node* node) {
    //复制原节点状态，新建节点
    Node* newNode = new Node(node->_X, node->_Y, node->_chessman, NULL);
    Init(newNode);
    //对该节点进行随机模拟至结束，计算收益
    int profit;
    while ((profit = IsTerminal(newNode)) == UNDETERMINED) {
        newNode->_chessman = ChangeChessman(newNode->_chessman);
        do {
            newNode->_Y = rand() % _N;
        } while (_newTop[newNode->_Y] == 0);
        newNode->_X = --_newTop[newNode->_Y];
        _newChessboard[newNode->_X][newNode->_Y] = newNode->_chessman;
        if (newNode->_Y == _noY && newNode->_X == _noX + 1) _newTop[newNode->_Y]--;
    }
    Clear(newNode);
    delete newNode;
    return profit;
}

//向上回溯更新收益值
void UCT::BackUp(Node* node, int deltaProfit) {
    while (node) {
        node->_visitedNum++;
        node->_profit += deltaProfit;
```

```
        node = node->_parent;
    }
}
```

3、使用接口方法

```
int* top_ = new int[N];
for (int i = 0; i < N; i++) {
    top_[i] = top[i];
}
//建UCT搜索树并搜索
UCT* u = new UCT(M, N, noX, noY, board, top_);
Node* node = u->Search();
x = node->_X;
y = node->_Y;
delete u;
delete []top_;
```

五、测试结果

1、参数选择

时间上限 TIME_LIMIT = 2.0s

信心参数COEFFICIENT = 0.8

2、测试结果

与1~60的AI每个对战2次（先手+后手各一次）

与60~100的AI每个对战6次（先手+后手各三次）

AI编号	胜率	AI编号	胜率	AI编号	胜率	AI编号	胜率
60	1	70	0.83	80	1	90	1
62	0.67	72	1	82	1	92	1
64	1	74	0.83	84	1	94	0.83
66	1	76	1	86	1	96	1
68	0.83	78	1	88	1	98	0.5
100	1	1~59	1	**得分**	97		

3、测试环境

系统：MacOS Sierra 10.12.4

IDE：XCode 8.1

CPU：2.7GHz Intel Core i5

内存：8GB 1867MHz DDR3

图形卡：Intel Iris Graphics 6100 1536MB

六、结果分析

从上述结果可以看出，蒙特卡洛搜索和信心上限树算法可以较好的完成重力四子棋博弈问题。

通过对比搜索时间为1s、1.5s和2s的差别，可以发现搜索时间对结果的影响，搜索时间越长，期望意义下胜率越高，但也会面临着超时的风险。

搜索时间在1s时，每次决策第一步平均可以搜索24万次；1.5s时，每次决策第一步平均可以搜索37万次；2s时，每次决策第一步平均可以搜索50万次。从搜索次数可以看出，蒙特卡洛搜索正是利用了机器可以短时间内大量计算的特性，获得较好的结果。在接近游戏结束时，每次决策平均搜索次数大大增加，这是因为默认政策下随机落子很容易接近终点。

2s虽然时间不长，但单步搜索已经突破50万次，加上信心上限算法对搜索的引导作用，搜索也已经比较充分，故结果也较为良好。但如果搜索时间过短，不能充分的搜索信息，则蒙特卡洛算法性能将大大下降。

七、小结

通过本次实验，我更加深入的理解了蒙特卡洛搜索和信心上限树。在完成这次试验的过程中，我遇到了不少难题，比如如何把算法转换为具体可行的代码，即使写出代码，也会面临调Bug的麻烦。通过本次试验，我感觉我对黑箱调试的能力得到了极大的锻炼，因为不便使用gdb等调试工具，全程采用输出信息的方式进行调试，让我调试能力得到了很大的提高。

同时，由于第一版设计不够巧妙，每个节点都完整的保存了棋盘状态和顶端状态，需要占用大量内存且效率低下（2s只能搜20万次），故又进行了重构，整棵搜索树只保留两个棋盘，一个原始的和一个用于更新的，避免了大量的new/delete操作，从而优化了算法，避免搜索过程中生成各种冗余信息，最终版的效率明显大大提高。

在调试并优化的过程中，我感受到了人工智能强大的魅力，也初步理解了AlphaGo所用的算法框架，希望今后还能有机会写出更多有意思的AI，真正在使用知识中学习，在使用知识中锻炼自己，进一步提高自己的能力。