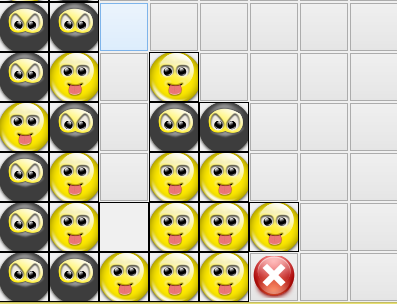
四子棋机器对弈项目实验报告

**【实验描述】**

本实验对要求实现一个变种四子棋的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。要求在所给的实验框架下完成四子棋游戏的人工智能决策部分并进行相应封装，对于每次传入的棋盘状态后返回一个落子点。

**【实验分析】**

在实验中我选择了蒙特卡洛算法，评测结果较好，以下简单就本实验算法选择进行一定分析。

实验指导中推荐使用alpha-beta剪枝算法，但使用alpha-beta剪枝算法的难点是很难设计出较好的估价函数。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。

而蒙特卡洛算法则不存在这样的瓶颈问题，蒙特卡洛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。

在具体实现中，为了保证迭代的次数和收敛效率，可以以空间来换时间，建立一颗搜索树来记录每一个模拟局面下的胜率等信息，在5s的时间限制内，可以生成200W个节点，与100.dll对战时，胜率也可以保持在60%的水平。

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。同时，如果alpha-beta剪枝算法可以设计出一个很好的估价函数，效果也可以好于蒙特卡洛算法。

**【算法流程】**

1. 初始化棋局，用0作为父节点，即整棵树的根节点，代表当前棋局状态；
2. 如果满足终止条件(经过时间达到一定时间)，跳至6；
3. 判断父节点是否为叶子节点：

如果父节点为叶子节点，那么对父节点进行扩展，对扩展出的每一个子节点都进行模拟对局，并向上更新祖先几点的totRound和winRound信息；

如果父节点不是叶子节点，不进行操作；

1. 在父节点的子节点中选取UCB值最大的节点；
2. 判断此节点对应的棋局状态是否已分出胜负：

如果已经分出胜负，再次向上更新winRound和totRound信息，将父节点置为0，跳至2；

如果未分出胜负，将选出的节点作为父节点，跳至2；

1. 从根节点的子节点中选取胜率最大点，将该点记录坐标作为最终返回点。

**【实现细节】**

**1、MCNode类：**

记录每一个节点的信息(即每新走一个棋子对应的一个棋盘状态)

x y为最新加入棋子坐标

winRound 和 totRound记录这个状态下user的胜场数和总场数，需要注意的是父节点和子节点的user不一样，更新胜场数的时候也需要隔层更新

class MCNode{

public:

int x,y;

int lChild,rChild,father;

bool user,isLeaf; // user: false-->oppoent, true-->me

int winRound, totRound;

};

**2、全局变量：**

为了加快迭代速度，用全局数组来创建树结构，并在全局创建tempTop1/2和tempBoard1/2来记录临时的棋局状态以方便对树中节点进行扩展和模拟

MCNode nodes[NODENUM];

int rank;

int sel;

int fNode = 0;

int\* tempTop1;

int\* tempTop2; // used for modify function

int\*\* tempBoard1;

int\*\* tempBoard2; // used for modify function

int nn = 0,mm = 0,banX = -1,banY = -1;

**3、扩展及模拟节点**

在父节点为叶子节点时，需要对父节点进行扩展，这里需要对不可落子点进行判断，使得每个节点中的x y值都是可落子的。

模拟过程中即在当前状态下随机选取位置来落子，并且判断是否分出了胜负，在得到胜负结果后向上进行更新

**4、计算UCB值**

每个节点UCB值的设计是蒙特卡洛算法中比较关键的一点，分为两部分，前部分为计算该节点上的胜率，胜率越高被选择的几率越大，而后一部分则是为了平衡模拟方向，使得之前模拟次数越少的点越有机会被选中

nodes[sel].winRound/(nodes[sel].totRound+epsilon) + C\*sqrt(log(nodes[fNode].totRound+1)/(nodes[sel].totRound+epsilon)))

实现语言: c++

编译工具：g++

运行环境：Windows

编译方式：同目录下运行makefile

**【测试结果】**

1. **与62-100.dll对战结果(胜场数/总场数)：**

|  |  |
| --- | --- |
| 100.dll | 6/10 |
| 98.dll | 7/10 |
| 96.dll | 8/10 |
| 94.dll | 8/10 |
| 92.dll | 8/10 |
| 90.dll | 8/10 |
| 88.dll | 10/10 |
| 86.dll | 9/10 |
| 84.dll | 10/10 |
| 82.dll | 10/10 |

|  |  |
| --- | --- |
| 80.dll | 10/10 |
| 78.dll | 9/10 |
| 76.dll | 10/10 |
| 74.dll | 10/10 |
| 72.dll | 10/10 |
| 70.dll | 6/6 |
| 68.dll | 5/6 |
| 66.dll | 6/6 |
| 64.dll | 6/6 |
| 62.dll | 6/6 |

**2、与100.dll对战100场(50rounds)结果(AI为A方)：**

Stat:

ratio of A wins : 0.57

ratio of B wins : 0.43

ratio of Tie : 0

ratio of (A wins + tie) : 0.57

ratio of (B wins + tie) : 0.43

**【结果分析】**

在测试过程中我打印了所选节点的胜率，结果基本符合之前的收敛分析，在可以获胜的棋局中胜率逐渐增大，而且当胜率增大到65%时获胜几率十分大；同样在胜率小于40%时失败的几率也十分大。在测试时如果将截止时间进一步增大并且减少对超时的判断次数，程序迭代的点个数可以接近300W，胜率也进一步增大，但这种情况下存在超时的风险。

综合来看蒙特卡洛算法可以较好的完成四子棋博弈的求解。