人工智能第二次作业

——重力四子棋

问题背景：

电子数字计算机问世之后，不少人就想，机器会不会思考呢？机器具有智能吗？很多学这认为这都是可以的。但人们首先关心的是计算机下棋。因为下棋是一种智力游戏，弈棋比赛是一种智力较量。许多科学家，为证明计算机可以有智力，进行了很多研究，让计算机下棋。

从 2006 年开始，计算机四子棋博弈的相关研究有了跨越式的发展，基于蒙特卡洛模拟的博弈树搜索算法获得了重要的成功，并开始逐步引领计算机博弈理论研究的方向。在本次实验中，我们将用蒙特卡罗博弈理论在重力四子棋问题中应用。

马尔可夫决策过程：

我们先从马尔可夫决策过程引入，马尔可夫决策过程是指转移概率和报酬仅仅依赖于当前的状态和决策者选取的行动，而不依赖于过去的历史的决策过程。对于任意一个决策时刻，在状态i采取行动$a \in A(i)$之后将产生两个结果，一是决策者获得报酬$r(i,a)$;二是下一个决策时刻系统所处的状态将由概率分布$P(\cdot|i,a)$决定。从模型的角度来看， 报酬$r(i,a)$是即时的，但是在这个决策周期内它是何时或如何获得的并不重要，在选取行动后，模型只需知道它的值或期望值。一般来讲报酬还依赖下一个决策时刻的状态j，即$r(i,a,j)$。此时，采取行动a的期望报酬值为:

$$r(i,a)=\sum_{j\in S} r(i,a,j)P(j|i,a)$$

上式中非负函数$P(j|i,a)$是下一个决策时刻系统转移到状态j的概率，函数$P(\cdot|i,a)$被称为转移概率函数。

四子棋模型：

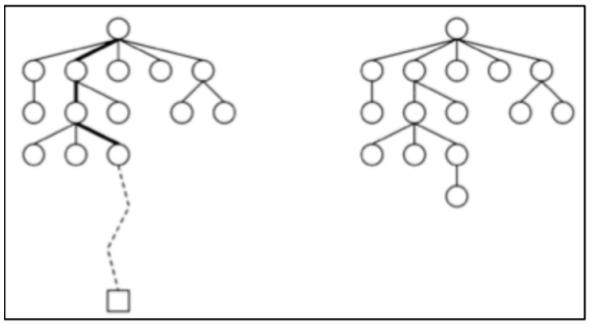
由于围棋是一种策略性二人棋类游戏，在对弈时，棋手只针对当前盘面进行决策，对于同样的盘面，棋手不用去考虑其中经历的不同步骤，本质上是一个马尔可夫决策过程。围棋中每一个落子的潜在价值较为难以估计，为转移概率的定义带来了一定的难度。简单地，我们可以定义如下的等概率模型:

$$P(\cdot|s,a)=\begin{cases}1，\ 如果A(s)=\emptyset，即没有任何可落子点\\ \dfrac{1}{M}，\ 如果|A(s)|=M，即有M个可落子点\end{cases}$$

蒙特卡洛搜索树：

蒙特卡洛树搜索是解决马尔科夫决策问题的有效方法之一，其基本思想是将可能出现的状态转移过程用状态树表示，从初始状态开始重复抽样，逐步扩展树中的节点；某个状态再次被访问时，可以利用已有的结果，提高了效率；在抽样过程中可以随时得到行为的评价。

因此，相对于在评估之初就将(隐含)博弈树进行展开的静态方法而言，蒙特卡洛树搜索的过程是一种动态的搜索过程，蒙特卡洛树搜索的基本思想如下图所示：



蒙特卡洛树的四阶段：

选择：从根节点出发，在搜索树上自上而下迭代式执行一个子节点选择策略，直至找到当前最为紧迫的可扩展节点为止，一个节点是可扩展的，当且仅当其所对应的状态是非停止状态，且拥有未被访问过的子状态;

扩展：根据当前可执行的行动，向选定的节点上添加一个(或多个)子节点以扩展搜索树;

模拟：根据默认策略在扩展出来的一个(或多个)子节点上执行蒙特卡罗棋局模拟，并确定节点的估计值;

回溯：根据模拟结果向上依次更新祖先节点的估计值，并更新其状态。

一次模拟后，回溯更新的过程为：

设 ni 为当前要模拟的节点，△为模拟获得的收益，对 ni 及其祖先的模拟次数加 1；ni 的收益加△，同时更新 ni 的同类祖先节点的收益 (这里节点的类型按照极大极小节点划分)。

每一回合，棋手需要从多个可下点中选择一个行棋，如果我们有足够的时间做模拟，则我们可以对每一个可选择的节点进行模拟，最后选择获胜频率大的节点。但是因为实际下棋时决策时间有限，我们需要有选择的进行模拟和落子。

当轮到算法给出可选落子时，我们将待决策的盘面作为某子树的根节点，然后在可下点中随机选择一点pi , $i \in [i, k]$并进行以下模拟过程:

如果该节点第一次被选中，即$ni = 0$，则将填入该节点后的盘面作为叶节点加入搜索树中并通过随机落子完成之后的行棋过程。

（1）ni自动加 1

（2）如果模拟至终胜利报酬wi加△（△为收益），如果终盘得到失败则wi 减△。

（3）将叶节点的信息返回给上层节点，其父节点的访问次数加 1，收益加上其子节点收益变化值的负值△/-△。 如果父节点仍有父节点的话，再往上同样的更新信息，直至根节点为止; 如果该点不是第一次被选中，即ni =/0，则我们以该盘面为子树的根节点再一次执行构建搜索树的过程。在模拟结束后，我们可以简单地计算根节点的每个儿子可下点的模拟收益

$$P(win) = \frac{w_i}{n_i}$$

最后我们在k个备选落子点中选择收益率最高的点落子。通过以上方法，就可以简单的实现计算机四子棋博弈的选点落子过程。

为了改进算法，我们选择可落子点时不止考虑胜率，而是同时考虑两方面的因素：对尚未充分了解的节点的探索、对当前具有较大希望节点的利用。由此我们将信心上限树算法 UCT 应用于蒙特卡洛树搜索中，用于选择可落子点。

定义信心上界为

$$I_j = \overline{X}_j + \sqrt{\frac{2\ln(n)}{T_j(n)}}$$

其中 :$\{\overline{X}_j\}$是落在在 j 状态所获得回报的均值，n是到当前这一时刻为止所访问的总次数，$T_j(n)$是状态 j 到目前为止所访问的次数。上式一是对尚未充分了解的节点的探索(exploration)，以尽早发现潜在价值高的节点并尽早排除潜在价值低的节点;二是对当前有较大希望的节点的利用(exploitation)，以尽可能地抓住机会创造对己方更有利的局势。实际计算 UCB1 时，我们加一个参数 c 进行调节：

$$I_j = \overline{X}_j + c\sqrt{\frac{2\ln(n)}{T_j(n)}}$$

模拟阶段算法实现：

```
//落子下棋!
function UCTSEARCH(s0)
  以状态s0 创建根节点v0;
  while 尚未用完计算时长 do:
    vl ←TREEPOLICY(v0);
    Δ←DEFAULTPOLICY(s(vl ));
    BACKUP(vl , Δ);
  end while
  return a(BESTCHILD(v0, 0));
```

```
//计算信心上限
float BoardTree::UCB1(MCSTNode *parent, MCSTNode *son){
    int &&N = son->GetTimes();
    int &&Q = son->GetValue();
         return -(float)Q / N + C * sqrtf(2 * logf(parent->GetTimes()) /
N);
}
```

```
//选择最佳落子点
MCSTNode *BoardTree::BestSon(MCSTNode *parent){
    //选择
         float maxValue = -INF;
         MCSTNode *bestSon = NULL;
    MCSTNode *son = parent->GetFirstSon();
    while(son){
        float &&ucb1 = UCB1(parent, son);
        //从对手的角度
```

```
        if (ucb1 > maxValue){
            maxValue = ucb1;
            bestSon = son;
        }
        son = son->GetNextSibling();
    }
         return bestSon;
}
```

```
//扩张节点
MCSTNode *BoardTreeNode::Expand(){
         MCSState *newState = state->NextSonState();

    if (!newState){
                return NULL;
    }

         MCSTNode *newNode = new BoardTreeNode(newState, this);

         if (!firstSon){
                 firstSon = newNode;
         }else{
                 newNode->SetNextSibling(firstSon);
                 firstSon = newNode;
         }
         return newNode;
```

```
//有限的时间内模拟
pair<int, int> BoardTree::MCTS(){
    int per_clock = 0;
    while ((++per_clock % 100 != 0) || (time(NULL) - StartTime) <
TIME_LIMIT){
        Simulate();
    }

         int maxTime = -INF;
         MCSTNode *bestSon = NULL;
    MCSTNode *son = root->GetFirstSon();
    while(son){
        if (son->GetTimes() > maxTime){
            maxTime = son->GetTimes();
            bestSon = son;
```

```
        }
        son = son->GetNextSibling();
    }

        pair<int,int> place =
dynamic_cast<MyBoardState*>(bestSon->GetState())->GetPlaceLocation();

        return place;
};
```

```
//回溯更新
void BoardTree::BackUp(MCSTNode *node, int value){
        while (node){
                node->AddValue(value);
                value = -value;
                node = node->GetParent();
        }
}
```

参数调整：

1.时间

测试样例：因为测试次数的限制，我使用排名前 20 的 token 进行测试

TIME_LIMIT	1s	2s	2.9s
胜率	65%	80%	88%

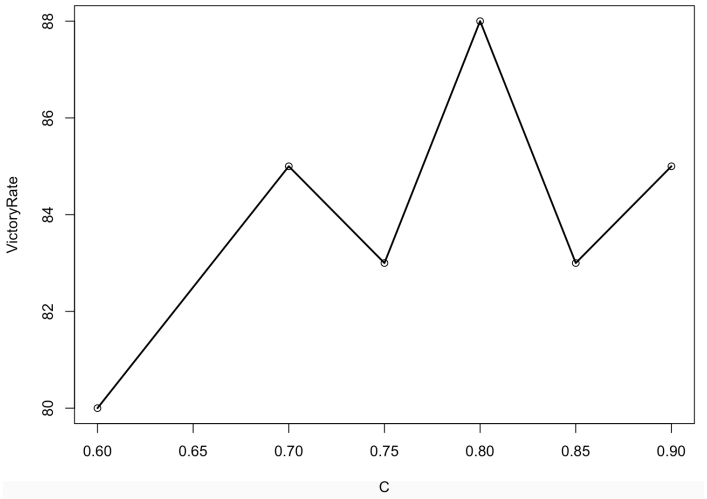
从时间限制层面，我们发现结果是符合直觉，当时间越长，模拟次数越多，我们的胜率越高。

2.参数 c

下面我们调整 UCT 中的参数 c，用排名前 20 的 token 测试：

$$I_j = \overline{X}_j + c\sqrt{\frac{2\ln(n)}{T_j(n)}}$$

C	0.6	0.7	0.75	0.8	0.85	0.9
胜率	80%	85%	83%	88%	83%	85%



C 是模拟次数和以往表现的权重参数，经过多次模拟，我们选择 C 为 0.8。

模型效果：

限制时间 2.8s C=0.8

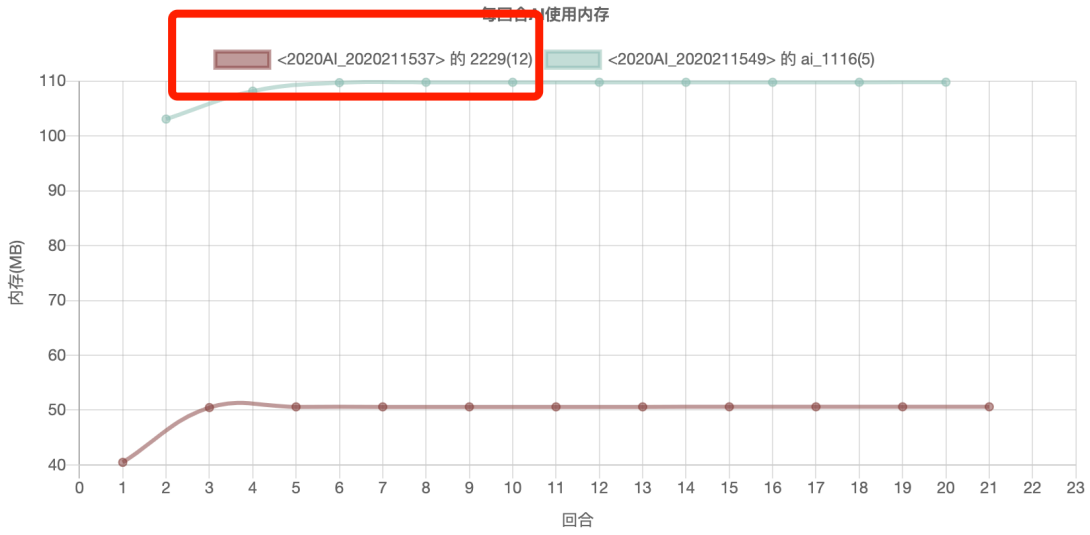
限制时间 2.9s C=0.8

限制时间 2.95s C=0.8

97	3	0	100	100	97%
胜	负	平	已测评局数	总局数	胜率

测试AI #46	评测完成	<Connect4_88>	sample	版本：1	胜：2 负：0 平：0
测试AI #47	评测完成	<Connect4_92>	sample	版本：1	胜：1 负：1 平：0
测试AI #48	评测完成	<Connect4_96>	sample	版本：1	胜：2 负：0 平：0
测试AI #49	评测完成	<Connect4_94>	sample	版本：1	胜：2 负：0 平：0
测试AI #50	评测完成	<Connect4_100>	sample	版本：1	胜：0 负：2 平：0

内存消耗远小于排名相近的选手：



总结与改进：

在本次实验中，我基于蒙特卡洛搜索树，结合信心上限树算法，完成了 C++重力四子棋的程序。在搜索时间小于 1s 时，搜索时间较短，搜索的深度不够，随着搜索时间的增加，模拟的次数增加、搜索的深度加深，胜率随之增加。之后调整 UCB 中的参数 c，我们以 0.5 为间隔，对不同的 C 进行测试，在未搜索和以往表现中 tradeoff，最后发现当参数为 0.8 时胜率最高。

在沙盒网站测试中，我发现即使将参数调到较优的水平，也很难打赢 100、96、94 号 token。如果想进一步提高四子棋的胜率，需要在算法（之后可尝试 alpha-beta 剪枝等其他方法）和模型层面加以改进。