重力四子棋作业

仲嘉暄 计35 2023010812

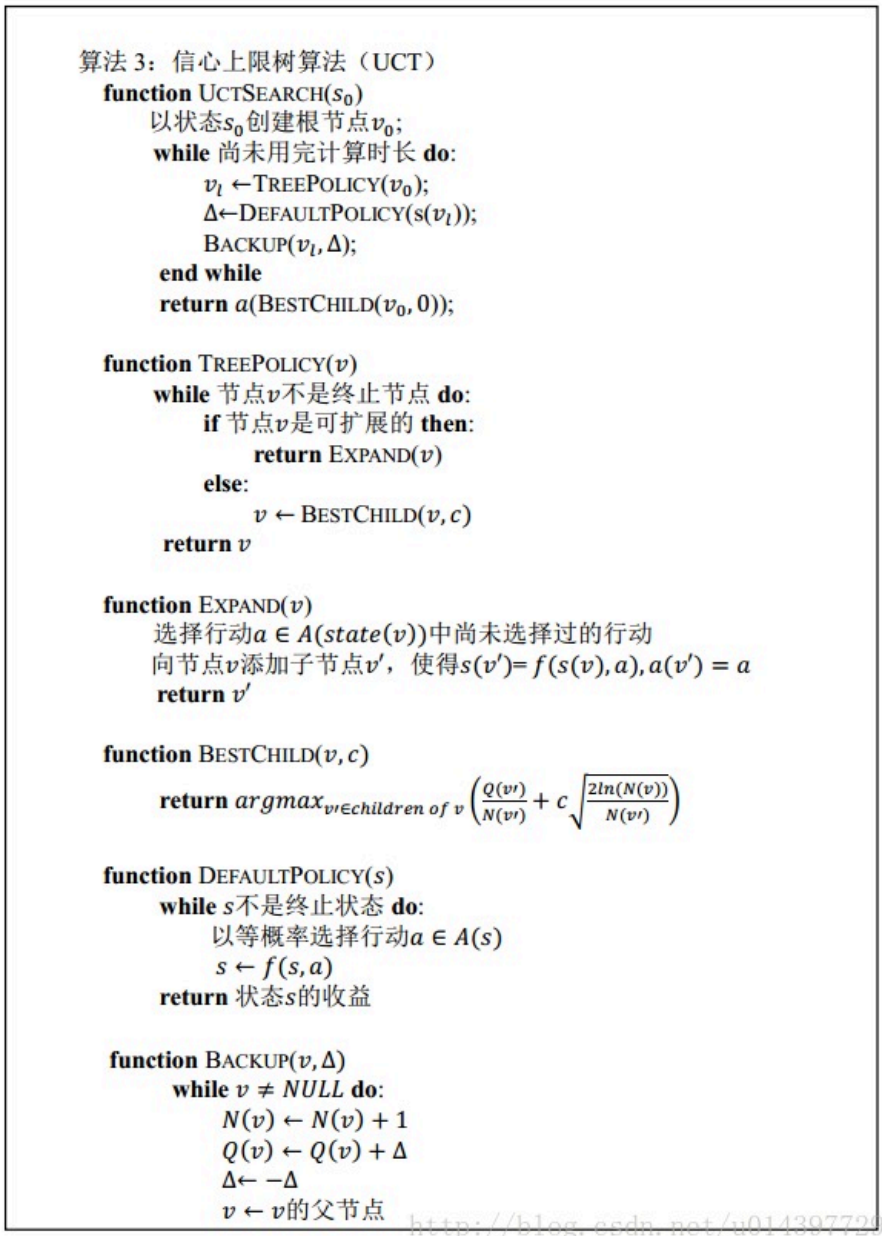
Honor Code

本项目参考了赵晨阳学长的代码和Harry-Chen学长的代码，向他们表示感谢！

项目介绍

算法

在本项目中，我使用的是 UCT 算法。其伪代码如下所示。



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**:
$v_l \leftarrow$TREEPOLICY(v_0);
$\Delta \leftarrow$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

内容

我修改了Strategy.cpp，新增了棋盘类（Board.cpp和Board.h）、计时器类（Timer.cpp和Timer.h）、UCT 树节点类（Node.cpp和Node.h）和 UCT 树类（Tree.cpp和Tree.cpp）。

Strategy.cpp

在Strategy.cpp中，我首先判断当前棋盘是否有必胜或必应点，如果有，就直接下在那里。显然，只要我们假定敌方有基本的分辨能力，这样的先验知识不会对 UCT 算法有任何负面影响。然后，我会用到 `M` 、 `N` 、 `noX` 、 `noY` 、 `top` 、 `_board` 对 UCT 树进行初始化，设定棋盘长宽、禁手点和初始棋盘情况。

棋盘类

棋盘类是受到了Harry-Chen学长的代码启发的，我们将自己的棋子和对手的棋子分开看待，分列在两个棋盘（一维的，通过 $i * \mathrm{width} + j$ 映射）中，这样每个棋盘就可以用一位保存一个棋子了。这样，我们就可以用位运算而非复杂的 for 循环来判断棋盘胜利了：比如，如果一个棋盘自己、右移一位、右移两位、右移三位做与运算之后不为零，那就说明其某一行上有四个连着的棋子。其问题是不能分辨“串行”的情况，前一行末尾的棋子和后一行开头的棋子可能会被错误地认为相连。我研究了学长代码，发现他通过设置棋盘的“实际宽度”为 16，在“两行之间”留足了 0，从而避免了这个问题。（加引号是为了表示“棋盘”是一维的，理解意思即可）相比于学长，我的创新在于使用了 bitset 库，简化了代码，同时使代码更加规范、可读，而且官方库的位运算可能经过优化，能够提高效率。

计时器类

计时器类能够保存开始事件（被构造的时间），其成员函数可以调用当前时间，检查两者的差是否超时。这样，就可以保证算法中的“尚未用完计算时长”。

Node 类

- 这里我最大的创新点是实现了真正的内存池。实现内存池可以大量降低内存分配和归还的时间消耗，见张凯文学长的实验报告。Honor Code的两位学长一个没有实现内存池，一个实现的是 `Node *` 的内存池，在此基础上，我实现了 `Node` 的内存池（开了一个模拟时绝对不会越界的大小，此外绝不用new申请内存）。内存池靠 `usedMemory` 指示分配给UCT树的节点数量，每次UCT搜索完都要归零。
- `set` 函数是设置内存池中节点的函数，也是从内存池分配节点给UCT树的根本函数。需要注意的是它尽在节点指向子节点的指针和指向可分配列数组的指针为空时分配内存，否则复用即可。
- 后面Tree类中 `treePolicy` 里判断节点是否为终止节点的函数 `isTerminal` 来自于Harry-Chen学长，但减少了一个对根节点的if分支判断（我在Tree类的 `search` 函数显式设置了root的终止节点bool量），优化了效率。同时，学长们对player的操作稍显混乱，我统一设置为查看本player方的棋盘是否胜利（或不可扩展），因为每个Node对应其记录的player下完棋之后的情况，显然我下完棋不可能敌人立即赢。
- `expand` 方法和Harry-Chen学长大抵相同，唯跳过禁手点的办法不同，后详Tree类的构造函数部分介绍。
- `backup` 函数我选择按照课件上的方法交替变化delta的正负号。
- `bestChild` 和伪代码基本一致，注意要更新局部读写棋盘，因为选择了bestChild之后，棋盘需要更新到这个节点应有的情况，从而保证后续模拟正确。

Tree 类

- 在构造 UCT 树时，除了用静态成员变量保存棋盘、禁手点等信息外，还要设置时钟和在下棋时如何跳过禁手点。这是因为我们在下棋时为了跳过禁手点，可以先下棋，再检查上一格是否为禁手点，如果是就往上跳过一格；也可以先检查这格是否为禁手点，如果是，往上跳过一格，再下棋。第一种方法（学长们使用的方法）无法识别最底部禁手点，第二种方法在最顶部禁手点的情况下会给棋盘下一个$(-1, y)$，造成错误。因此，在初始化Tree的时候就要根据禁手点的位置选择相应的方法。为了避免运行时分支判断的开销，我使用了函数指针。
- `init` 函数的作用是保存外面传来的常量 `top` 和 `_board` ，代表对手刚下完那一步之后的原始情况。后面，我们会在 `search` 函数（伪代码中UCTSEARCH）的循环里，每次都用原始情况更新用来读写的局部读写棋盘（因为模拟等过程需要实时更新棋盘）。
- `search` 函数里，我们首先定义一个Node类对象root，其player是对手。然后在计时循环里，每次先复位局部读写棋盘和top到原始状态，再依次调用 `treePolicy` 选择工作节点、 `defaultPolicy` 判断工作节点价值、 `backup` 交替传播节点价值到根，进行模拟。最后 `rootChild` 选取root最好的子节点（player是我），把内存池的usedMemory复位到0，再返回最好子节点的(x, y)。

- `treePolicy` 和伪代码相同。
- `defaultPolicy` 与学长代码相比没有任何人工特判，完全信任 UCT 树自己的模拟能力。我们要区分初始传入这个函数节点对应的“原始玩家”和后续模拟过程中每一步下棋都会对换的“当前玩家”，如果胜利时二者相等，那么由于本函数计算的是相对于传进来的节点的价值，就要返回 1；如果不同就返回 -1，平局返回 0。然后，我会随机挑选一列，**先转换身份**，再安全下棋（自动跳过禁手点）。一般会单纯用 `rand() % N` 进行随机选取，但是当棋盘快满的时候（由于一次search可能有两三百万次循环，所以这很常见），就可能一直选取已满的列，很慢才能选到未满的列。为此，我做了一个映射：

```
int avail_column = N;
int top_index[12] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
...
while (true)
{
    ...
    int randomY = rand() % avail_column;
    while (top[top_index[randomY]] == 0)
    {
        std::swap(top_index[randomY], top_index[--avail_column]);
        randomY = rand() % avail_column;
    }
    randomY = top_index[randomY];
    ...
}
```

这样可以保证每次只在有空位的列里进行挑选，把复杂度从均摊$O(N)$降低到均摊$O(1)$，大大加快了效率。

- `backup` 直接调用 `treePolicy` 所得节点的 `backup` 。
- `rootChild` 即伪代码中的 `bestChild(0)` ，这里可以不用更新局部读写棋盘，所以我直接新增了个函数。

实验结果

我调参的胜率如下（10个token）

	胜	败	平	胜率
c = 0.5	18	2	0	90%
c = 0.6	18	2	0	90%
c = 0.707	16	4	0	80%
c = 0.8	14	6	0	70%
c = 0.9	20	0	0	100%
c = 1	14	6	0	70%

后续又对c = 0.9 加测全部 50 token，胜率94%。对c = 1 加测最高 10 token，胜率90%（后又测了一次70%）。对c = 0.707 加测 10 token，胜率75%。
考虑到 0.9 接近 1，而且 1 是 UCT 公式讲的，所以派遣了 c = 1 的 AI 为作业。