16 级计科 7 班: project-name #2

Due on Tuesday, September 18, 2018

teacher-name 周三 3-4 节

颜彬

16337269

Content

1 具体贡献

在本次期末项目中，我为黑白棋游戏实现了以下的几个功能。

- 二进制棋盘压缩及其状态维护
- 进一步剪枝的 alpha-beta 算法
- 提出了 7 种不同的评价指标
- 自我博弈的进化算法

以上几个功能相辅相成，他们可以共同配合让黑白棋获得更大的棋力。除此之外，以上的许多功能可以为组员带来便利。例如二进制的棋盘压缩方法大大加快了棋盘维护速度，相当于变相加速了其它的所有算法。

NOTE: 以上功能使用 C++ 完成，没有基于任何第三方库。

1.1 二进制棋盘压缩

在本次期末项目中，棋盘是 8*8 的。如果用一个 bit 表示棋盘上是否有棋子，那么刚好 unsigned long long 类型（64 位）恰好可以存储棋盘的棋子信息。

二进制棋盘压缩方法利用了上述的特点。它用一个 u64 类型的整数表示黑棋在棋盘中的位置，用另一个 u64 整数表示白棋在棋盘中的位置，利用位运算提取棋盘中的所有信息。

二进制棋盘压缩方法面临的困难问题是，如何在二进制下高效地取得可下子点，如何高效实现棋盘的翻转。除此之外，还有有二进制下如何计算黑白棋子个数、计算棋盘空位、判断游戏是否结束等问题。

得益于位运算的优化，基于压缩棋盘的 alpha-beta 剪枝获得了极大的加速。这一加速也为其他算法提供了很好的条件。

1.2 进一步剪枝的 alpha-beta 算法

alpha-beta 算法可以被进一步优化。注意到游戏在进入中盘后，每一方的可下子点数都很大。alpha-beta 算法的运行速度会在中盘明显地下降。

为了解决这个问题，可以对 alpha-beta 算法进行进一步剪枝。具体思路是，先对所有可行点做一次很浅的 alpha-beta 剪枝，然后对所有可行点的估值作排序。排好序后，筛掉一部分估值很低的顶点，按估值降序继续 alpha-beta 的搜索过程。

这个放在对前期和后期会带来略微的性能下降，可是却能大大加快中盘的搜索速度。

1.3 7 种不同的评价指标

我为黑白棋的估值总结了 7 种不同的方法。这些几种估值方法从不同的方面估计棋盘的局势。他们是

1. 估值表估值法
2. 基于 4 个角的估值

3. 靠近角估值
4. 基于行动力的估值
5. 基于棋子边缘的估值
6. 基于棋子个数的估值
7. 基于稳定子的估值

后文会详细介绍这些估值法的具体含义，并给出有效的实现方法。

这几种估值方法抽象成了接口，可以为组员提供很大的帮助。例如，这些估值可以作为神经网络的额外输入，或者作为提供给资源的额外评价指标。

1.4 自我博弈进化算法

在提除了 alpha-beta 剪枝和 7 种估值法后，应该如何权衡这 7 种估值法呢？最好的方法是为 7 种估值方法提供一个权重，用加权求和的方法得到最终的估值。

加权求和有个难点，需要手动确定权重。于是我采用了自我博弈方法和进化方法。方法大致是

(a) 认为给定一个初始权重

(b) 基于最好的权重（第一轮迭代时是初始权重），产生一系列新的权重

(c) 对这些不同权重得到的估值方式进行相互博弈

(d) 选出最好的权重方案，跳转到 (b)

这个算法还有很多细节，会在随后解释。最终，各个参数应该会收敛到某一个确定的值。

2 二进制棋盘压缩

2.1 棋盘的表示

使用两个 64 位的无符号整数来存储棋盘的信息。每个 u64 整数有 64 个 bit，每个 bit 代表 8*8 棋盘中的一位。如果该 bit 是 1，代表该位置有棋子。

用第一个 u64 代表黑棋的分布，用第二个代表白棋的分布。用两个整数的位或代表已经下列棋子的位置。

2.2 基本的信息提取

棋盘类必须为其他模块提供必要的接口，他们的实现方法如下。

在表 1中，boards 的类型为 u64[2]。第一个元素代表黑棋的棋盘，第二个元素代表白棋的棋盘。参数 sq 是一个整数，取值范围是 [0, 64), 代表棋盘的第 i 位。参数 p 是整数，取值范围是 {0, 1}, 分别代表黑棋和白棋。
这些方案很好理解。为了确定棋盘的第 sq 位置是否有棋子，只需要将棋盘右移 sq 位，然后检查最低位是否为 1。

表 1: 必要接口的二进制实现

函数	参数	实现
isEmpty	sq	((boards[0] | boards[1]) » sq) & 1
isMyPiece	sq, p	(boards[p] » sq) & 1
isOppPiece	sq, p	(boards[!p] » sq) & 1

2.3 数子个数 - 编译器优化

可以使用 GCC 的编译器内置函数 __builtin_popcountll 来计算一个整数的二进制中 1 的个数。

这个函数可以为我们计算出棋盘上棋子的个数（1 的个数）。而且速度极快。

2.4 得到可下子点

得到可下子点的代码如代码 1所示。

定义了一个辅助函数 MOVABLE_HELPER。它是一个宏，接受一个 “方向”（例如 N）。它的作用举例来描述。

例如在调用 MOVABLE_HELPER(N) 时，会首先对自己的棋盘的所有子向上移动一个单位。移动后与对方的棋子做位与，然后与临时变量作位或。这一步结束后，临时变量存的值代表着，自己的棋子向上一个单位后碰到的对方棋子位置。

对这个行动重复 8 次（实际上不需要 8 次，只需要 5 次，细节在这里省去）。这时临时变量存储的是，自己的棋子都向上移动 1，2，3，4，5，6，7，8 个单位后，碰到的对方的棋子。

最后把临时变量向上移动一个单位，并对棋盘的空位做与运算。得到的就是所有可下子的位置。

MOVABLE_HELPER(N) 计算的是，“向上” 得到的所有的可下子点。对 MOVABLE_HELPER 在 8 个方向上各做一次，得到的就是整个棋盘的所有下子点。

2.5 棋盘翻转

实现二进制棋盘的下子翻转是一件有一定难度的事情。如代码代码 2所示。

FLIP_HELPER 接受一个方向，把下的子的那个方向上的所有可翻转的棋子都翻转过来。下面用一个例子来解释。

假设调用了 FLIP_HELPER(N), 即考虑下了一个子后，翻转它上方的子。

- if 语句首先判断这个子的上方是否有对方的子。
- 如果有，则不断地将这个子向上移动 i 单位（i 为 1, 2, 3, ..., 8），并把这个位置的对方的子标记。
- 如果向上移动 i' 个单位后，这个位置没有对方的子，则跳出循环。
- 循环结束时，如果位置 i' 上有是自己的子，那么中途所有被标记的位置都要被翻转。

上述可以实现上方的子的翻转。对 8 个方向类似地调用 8 次，即可翻转所有需要翻转的子。

代码 1: 得到可下子点的具体代码

```
#define MOVABLE_HELPER(dir) \
    tmp = dir(cur) & opp; \
    for (int i = 0; i < 5; ++i) { \
        tmp |= dir(tmp) & opp; \
    } \
    ret |= dir(tmp) & empty;

inline u64 ChessBox::getMovable(int p) const {
    u64 empty = getEmpty();
    u64 tmp, ret = 0;
    u64 cur = boards[p];
    u64 opp = boards[!p];

    MOVABLE_HELPER(N);
    MOVABLE_HELPER(S);
    MOVABLE_HELPER(W);
    MOVABLE_HELPER(E);
    MOVABLE_HELPER(NW);
    MOVABLE_HELPER(NE);
    MOVABLE_HELPER(SW);
    MOVABLE_HELPER(SE);

    return ret;
}
```

2.6 进一步剪枝的 alpha-beta 算法

2.6.1 辅助函数

alpha-beta 剪枝的 minimax 算法可以被进一步地剪枝。其中用到的辅助函数如代码 3所示。

这个函数的工作是，对棋盘 cb，以 player 为当前玩家，以 depth 为搜索深度，*返回 n* 个估值最优的行动方法。这个 n 一般很小，这个函数相当于做了一次预筛选。

这个函数本质做的也是 alpha-beta 剪枝。首先它要获取所有可行的下子方式，然后对这些下子方式做 alpha-beta 剪枝的 minimax 算法。然后将他们和他们的最终估值存入候选人列表 candidates 中。

然后，根据当前玩家是黑棋还是白棋，对候选人列表做排序。取有序列表中的前 n 个元素作为返回的可行点。

2.7 剪枝具体方案

一般而言，alpha-beta 的 minimax 算法会搜索 8 层。在这 8 层搜索的搜索中，搜索前 3 层时可以采用节 2.6.1提到的剪枝策略。

代码 2: 二进制棋盘的翻转实现

```
#define FLIP_HELPER(dir) \
    if (dir(1ull << sq) & opp) { \
        mask = 0; \
        tmp = dir(1ull << sq); \
        for (; tmp & opp;tmp = dir(tmp)) { \
            mask |= tmp; \
        } \
        if (tmp & cur) { \
            cur ^= mask; \
            opp ^= mask; \
        } \
    }

void ChessBox::__flip(int sq, int p) {
    assert(p == BLACK_ID || p == WHITE_ID);
    u64 mask, tmp;
    u64& cur = boards[p];
    u64& opp = boards[!p];

    FLIP_HELPER(N);
    FLIP_HELPER(S);
    FLIP_HELPER(E);
    FLIP_HELPER(W);
    FLIP_HELPER(NE);
    FLIP_HELPER(NW);
    FLIP_HELPER(SE);
    FLIP_HELPER(SW);
}
```

在这前三层的搜索中，设 s 表示当前层的可行数。取

$$r = \max\{\lfloor s/2 \rfloor, 3\}$$

r 为当前层应该最终应该搜索的分支数。

利用上面提到的辅助函数，取 n=r, 即可得到估值相对较好的 r 个分支。只对这 r 个分支做完整的搜索。

一般而言，在调用辅助函数进行预搜索时，预搜索的层数应该比较小。在项目中，预搜索采用的深度是本来深度的一半。

3 评价指标

本实验中，提出了 7 种不同的评价指标和他们的快速计算方式。

3.1 估值表估值法

估值表估值的方法很简单。对于 8*8 的棋盘，给出一个 8*8 的估值表。对每个位置，如果这个位置有棋子，则获得该位置的分数（分数可能是负的）。最后整个棋盘所有位置的分数和就是该玩家的分数。

代码 3: 行动方案预筛选函数

```
vector<int> AlphaBetaSolve::search_helper(const ChessBox& cb,
    int n, int player, int depth) const {
    vector<int> moves = cb.movessq(player);
    vector<Position> candidates;
    for (int move : moves) {
        ChessBox ncb(cb);
        ncb.drop(move/8, move%8, player);
        double ret = alphabeta(ncb, depth, -INF, INF, !player, false);
        candidates.emplace_back(move / 8, move % 8, ret);
    }
    if (player == BLACK_ID) {
        sort(candidates.begin(), candidates.end(), blackPosCmp);
    } else {
        assert(player == WHITE_ID);
        sort(candidates.begin(), candidates.end(), whitePosCmp);
    }
    vector<int> new_moves;
    for (int i = 0; i < n && i < candidates.size(); ++i) {
        new_moves.push_back(candidates[i].x * 8 + candidates[i].y);
    }
    return new_moves;
}
```

估值表的设置如代码 4所示。四个角的分数是比较高的。但与 4 个角相邻的 12 个位置的分数却是比较低的。这是因为如果想要占领角的位置，就应该让靠近角的位置让别人来占领，这样自己就可以通过下子翻转的方式来占领角。

边的估值也是比较高的。这是因为当占领了边后，一般而言是比较稳定的。

代码 4: 估值表的设置

```
const int VALUE[8*8] = {
    20, -3, 11,  8,  8, 11, -3, 20,
    -3, -7, -4,  1,  1, -4, -7, -3,
    11, -4,  2,  2,  2,  2, -4, 11,
    8,  1,  2, -3, -3,  2,  1,  8,
    8,  1,  2, -3, -3,  2,  1,  8,
    11, -4,  2,  2,  2,  2, -4, 11,
    -3, -7, -4,  1,  1, -4, -7, -3,
    20, -3, 11,  8,  8, 11, -3, 20,
};
```

3.2 角和临近角估值

角估值的方法很简单。先统计自己占领的角的数量 n。再统计对方占领的角的数量 n'。则 $c(n-n')$ 就是当前局面的评分。其中 c 是一个常数，在这里中取为 25。

NOTE: c 取 25 的原因是让取值范围在 [-100, 100] 里。

临近角的估值方法也很简单。如果一个角没有被占领，那么相邻这个角的 3 个位置都是坏的位置。统计自己占领的坏位置的数量 b 和对方的数量 b'。那么 $c(b-b')$ 就是当前局面的评分。其中 c 仍然是一个常数，在这里取为 8.3。

NOTE: c 取 25 的原因是让取值范围在 [-100, 100] 里。

3.3 棋子数目估值

棋子数目估值方法十分直接。直接统计棋盘上自己的棋子数和对方的棋子数。算出自己所占棋子数的百分比。将百分比 * 100 作为估值。

这个估值方法很直接，实际上效果并不好（接下来会看到）。

3.4 行动力估值

行动力估值的方法是，计算出自己和对方的行动力。算出自己的行动力的百分比。将行动力的百分比 * 100 作为估值。

一般而言，行动力越大越好。这是因为，如果将对手的行动力限制住，对手将无子可下。最终对手的每一步都会在你的掌控之中。这一方面限制了对手的行动，另一方面也让自己的搜索结点数大大减小，搜索速度更快。有一举两得的效果。

自己的行动力大带来的好处是选择多，则更有可能搜索出能翻盘的结点。

3.5 边缘棋子数估值

首先要定义边缘棋子。如果一个棋子有一个一个边暴露在外面，那么这个棋子是边缘棋子。一般来说，边缘棋子越多，形式越不利。

边缘棋子估值的方法是，计算自己的边缘棋子和对方的边缘棋子的占比。返回自己的边缘棋子百分比 * 100。

3.6 综合估值

综合估值是这里最重要的估值。

综合估值会结合上述 7 种估值方法。综合估值会给每种估值方法一个权重。然后对这些估值方法做加权求和。

加权求和的权重不是认为给定的，是通过进化算法学习得到的。见下文的介绍。

4 自我博弈的进化算法

4.1 大致思路

一开始，选定一个初始的权重。根据这个权重，随机地产生若干个不同的权重。利用这些不同的权重产生综合估值函数（小节 3.6）。

利用这些估值函数进行两两对抗。以游戏结束后子的数量差（而不仅仅是胜利和失败）来评价估值函数的优劣。取出最优的估值函数（获胜后，拉开最大的棋子差的函数），用这个估值函数的权重来产生数量更多的其他权重。再让他们进行对抗。

不断反复上面的这个过程。最终能通过自学习的方法达到最好的权重。

4.2 细节介绍

每一轮的进化，共产生 4 个新的权重，一共组成 5 个不同的权重。让 i 和 j 在 [0, 5) 这个范围内遍历，对于 i != j 的情况 (共 20 种)，每种对应一次对弈。其中 (i, j) 指 i 为黑方，j 为白方进行的对弈。由于训练程序跑在 32 核的电脑上，足够开很多的线程，故对弈采用 20 线程实现。

在每次对弈中，共进行 6 轮比赛。记录每次比赛中 i 比 j 多的棋子数，把胜利的棋子数（可能为负）记录在一个战绩表中。

等 20 次对弈的 6 轮比赛都完成时，查看战绩表，选中战绩最好的一个权重。不断地利用战绩最好的权重为初始权重进行新一轮的进化。

有一个细节是，战绩表会被多个线程同时修改，所以需要加锁。加锁不会让程序运行速度变慢很多。这是因为程序的性能瓶颈在对弈。读写是很少的操作。