四子棋实验报告

李晨昊 2017011466

2019-5-5

目录

1　算法描述

我使用了蒙特卡洛树搜索算法和信心上限树算法。

在蒙特卡洛树搜索的框架中，如果一个节点已经被完全拓展，则需要从它的所有孩子节点中选择一个孩子继续模拟，这个选择过程就是一个计算 UCB 值的过程，UCB 定义为：

$$UCB(x) \triangleq \frac{x.win}{x.vis} + c\sqrt{\frac{2\ln(x.p.vis)}{x.vis}}$$

其中 c 是常数，理论结果为 $c = 1$。每次选择 UCB 值最大的孩子继续模拟。直观上来说，这个公式可以尽量平衡探索与利用，因为它既容易选择已知胜率较高的孩子，也容易选择探索次数较少的孩子。

执行上述选择过程，直到遇到一个不是完全拓展的节点，随机拓展一个它的孩子，从它的孩子处开始随机模拟棋局直到结束，若胜利则沿着树回溯，将路径上每个点的 win 值 +1。

2　具体实现

同上次作业一样，我的主要精力花在了优化现有的算法而非实现更复杂的算法上。

我没有使用提供任何接口，也没有使用为我们准备好的指针数组棋盘，这些东西我完全自己重新实现了一遍。我的棋盘定义为：

```
template <u32 R, u32 C>
struct Board {
  u16 w[C], b[C];
  u8 state = Unfinished;
  u8 last_p = 0, last_row, last_col;
  ...
}
```

采用静态的数组来保存状态，对于状态的保存也采取了很大的压缩：数组 `w` 和 `b` 中每个元素代表一列，一个元素的每个 bit 代表该行是否有棋子，其中最低位为棋盘最高处。在此基础上，将第棋盘高度 +1 个 bit 设成 1，就可以直接采用一些 CPU 指令来计算落子位置，而无需使用循环或者专门记录：

```
// 统计低位的连续 0 的个数
inline u32 ctz_wrapper(u32 x) {
#ifdef _MSC_VER
  u32 ret;
  _BitScanForward((unsigned long *)&ret, x);
```

```
  return ret;
#else
  return __builtin_ctz(x);
#endif
}

// 获取第 i 列最上方的已被占据的行号
u32 get_row(u32 i, u32 dis_r, u32 dis_c) const {
  // 黑子白字按位或，为 1 的 bit 已经有棋子了
  u32 row = ctz_wrapper(b[i] | w[i]);
  if (row == 0) {
    return 0;
  }
  // 还需考虑那个地方是否禁止落子
  return row - (i == dis_c && row - 1 == dis_r);
}
```

其中 `_BitScanForward` 是 MSVC 提供的的 intrinsic，`__builtin_ctz` 是 gcc 提供的 intrinsic。它们可能编译出 bsr 或者 tzcnt 之类的指令，但是由于评测机的 CPU 型号实在太老，在我的平台上编译出来的指令也许评测机并不支持，这时应该会得到一个 0x06 中断 Invalid Opcode，然后我的程序就挂了，这是纯属正常的，麻烦助教请在评测机上再编译我的程序。

在此基础上，可以把判断胜负的循环也用位运算来代替，虽然仍然需要很多条指令，但是相比于循环，速度已经有了很大的提升：

```
inline u32 bextr_wrapper(u32 x, u32 start, u32 len) {
#ifdef __BMI__
  return _bextr_u32(x, start, len);
#else
  return (x >> start) & ((1 << len) - 1);
#endif
}

bool check_win() const {
  u32 i = last_col, row = last_row;
  u32 sel = std::min(7u, R + 3 - row);
  const u16 *which = last_p ? b + i : w + i;
  // 读取对应列
```

```
  u16 rd0 = which[-3], rd1 = which[-2], rd2 = which[-1], rd3 = which[0],
      rd4 = which[1], rd5 = which[2], rd6 = which[3];
  // 读出上次落子点附近的棋盘
  u64 block = (u64)bextr_wrapper((i - 3 < C ? rd0 : 0) << 3, row, sel) |
              (u64)bextr_wrapper((i - 2 < C ? rd1 : 0) << 3, row, sel) << 8 |
              (u64)bextr_wrapper((i - 1 < C ? rd2 : 0) << 3, row, sel) << 16 |
              (u64)bextr_wrapper(rd3 << 3, row, sel) << 24 |
              (u64)bextr_wrapper((i + 1 < C ? rd4 : 0) << 3, row, sel) << 32 |
              (u64)bextr_wrapper((i + 2 < C ? rd5 : 0) << 3, row, sel) << 40 |
              (u64)bextr_wrapper((i + 3 < C ? rd6 : 0) << 3, row, sel) << 48;
  // 判断胜利的 13 种情况
  return !((~block & 0x78000000ULL) && // 垂直
           (~block & 0x08080808ULL) && // 水平 * 4
           (~block & 0x0808080800ULL) &&
           (~block & 0x080808080000ULL) &&
           (~block & 0x08080808000000ULL) &&
           (~block & 0x08040201ULL) && // 右斜线 * 4
           (~block & 0x1008040200ULL) &&
           (~block & 0x201008040000ULL) &&
           (~block & 0x40201008000000ULL) &&
           (~block & 0x08102040ULL) && // 左斜线 * 4
           (~block & 0x0408102000ULL) &&
           (~block & 0x020408100000ULL) &&
           (~block & 0x01020408000000ULL));
}
```

这段代码大致的思想是，把刚刚落子的地方附近的棋盘读出来，用一个 64 位整数来表示。所谓附近，指的是自身及上下各三格 * 自身及左右各三格，总计 49 个 bit。不过为了方便 (下面的 magic number 的计算) 起见，让每列占据了 8 个 bit，所以用到了 56 个 bit。读出来之后，判断可能胜利的 13 种情况，其中垂直占 1 种，水平/左斜线/右斜线各占 4 种。这些 magic number 的来源是对应的胜利情况中，让相应的 bit 为 1 得到的数字。`~block & X` 非 0 时，证明 `block` 中没有包含 `X` 中的每个 1，也就意味着不是对应胜利情况，所以只要任何一个为 0，就证明上次落子带来了胜利。在上面的计算中，提前读出来对应的列的目的是确保编译器在后面的 ternary expression 中不会生成分支指令，因为这里涉及到内存访问，直接写?: 编译器并不知道这次访存会不会有什么副作用，从而不能完全使用条件传送指令来代替分支指令；`bextr_wrapper` 可能会被编译成 bextr 之类的指令；`~block & X` 可能会被编译成 nand 之类的指令，一条指令即可计算非与。

在蒙特卡洛树搜索的过程中，使用了一个小技巧来减少计算：每次更新一条链上的 win 值时，其实无关节点的 UCB 值并不会改变，下次模拟的时候不需要重新计算。所以我给每个节点设置了一个 `dirty` 位，只有 `dirty` 位标记为 1 的才需要在选择它的孩子时计算孩子的 UCB，否则可以直接使用存好的孩子的 UCB。考虑到浮点运算的代价较大，这里引入一个分支判断是值得的。

在模拟的迭代循环中，我使用了一个计时器，每迭代一定次数查看一次，到 2.7s 时自动退出。这也体现了蒙特卡洛树搜索的一个优点：虽然结果不一定很好，但是可以随时终止计算，返回当前的最好结果。

最后还有一个小问题：棋盘的大小并不能在编译期确定，这样表面上看起来就用不了模板了。不过这个问题其实很好解决，只要分情况处理行数和列数的 $4 * 4 = 16$ 种情况，使用对应的模板参数即可。主程序大概是这个画风：

```
switch (M) {
case 9:
  switch (N) {
  case 9:
    pr = ((Policy<9, 9> *)policy)->mcts(Board<9, 9>(b, x, y), MAX_ITER);
    break;
  case 10:
    pr = ((Policy<9, 10> *)policy)->mcts(Board<9, 10>(b, x, y), MAX_ITER);
    break;
  case 11:
    pr = ((Policy<9, 11> *)policy)->mcts(Board<9, 11>(b, x, y), MAX_ITER);
    break;
  case 12:
    pr = ((Policy<9, 12> *)policy)->mcts(Board<9, 12>(b, x, y), MAX_ITER);
    break;
  }
  break;
case 10:
  ...
case 11:
  ...
case 12:
  ...
}
```

是的，我完全不在乎什么代码风格，可拓展性之类的。性能是我唯一关注的目标。

3 参数选择

为了确定 UCT 算法中最佳的 c 参数，我使用了自己的程序两两对战，进行了大约 5000 局的测试 (每局耗时约 3 分钟)(当然，使用了不只一台机器)，最终的结果是 $c = 0.85$ 相比于其它的 c 的胜率 (考虑先后手) 是最高的，但是在和提供的测例 AI 的对战中，$c = 1$ 的表现仍然是最好的，因此最后还是选择了 $c = 1$。

4 测试结果

与 100.dll 先后手分别对战 16 局，先手胜 13 局，后手胜 10 局，可以认为二者棋力基本相当，或者我的程序略强一点。

在我的平台 (R7-2700) 上，在上面提到的这些优化下，12*12 的棋局的最开始我可以模拟约 $5 * 10^6$ 次。后面随着棋盘不断填满，模拟次数还可以更多一些。但是因为这个模拟次数是直接受硬件配置制约的，而模拟次数可以直接影响到棋力，所以在评测机这么老的 CPU 型号下，我很怀疑我的程序还能否取得这样的效果。

5 总结

早在选这门课以前我就尝试过实现蒙特卡洛树算法来下一些棋，但是一直没有获得过比较理想的效果，这一次借着课程作业的机会写了一个还算差强人意的版本。

这个算法还有很多很有趣的拓展，例如在计算 UCB 的时候可以引入神经网络的估值，这样可以减少搜索的盲目性，在更少的搜索次数内就得到一个较好的结果，这也是 AlphaGo 等围棋 AI 的大致实现方法。但是受限于时间和硬件，没法实现这个拓展了，也许以后课程可以把第二次和第三次作业整合起来，把神经网络应用到四子棋上，可能会产生很多有意思的结果。