四子棋 AI 实验报告

计 81 严镒洲 2018011299

这份实验报告中我们将讨论这份 AI 的实现细节，以及它和样例 AI 的对抗结果。

github 地址：https://github.com/Billyyanyz/connect4

1 算法思路

这个四子棋 AI 采用的是信心上限树的算法，我们构建 UCT 树，利用蒙特卡洛方法进行随机对战，获得最佳落子点。

1.1 棋盘状态结构

由于存储期盼状态是一件非常消耗空间，维护又耗时的操作，我们选择将棋盘分离出树进行维护。考虑到如果我们维护了棋盘的轮次以及题目给出的各个信息，棋盘从某一个状态到下一个状态下，只需要知道在哪一列落子即可唯一地确定棋盘状态，我们可以认为棋盘每时每刻都和我们所在的节点对应。

由此我们定义结构 `Board_Condition`，它的成员为所有描述棋盘信息的变量（即主函数 `getPoint` 的各个参数所代表的信息，另外加上当前的落子方），并在 UCT 结构的头文件中声明该结构的一个实例，它的维护与树的搜索平行的同时，我们也可以临时地在其中进行局部修改，判断终局等操作。

我们为这个结构提供了若干接口：

- `make_move(int col, int* memoX, int* memoY)`：模拟在 col 列落子，落子方交换，top 数组更新，原先的上一步信息暂存到 `memoX` 和 `memoY` 中。
- `return_move(int col, int memoX, int memoY)`：退回在 col 列的落子，落子方和 top 数组恢复，将暂存的上一步信息退回到 `lastX` 和 `lastY` 中。
- `end_game()`：判断当前局面是不是产生了胜者。我们在其中提供了如下几种判断：
  - 如果当前局面下，非落子方已有四连（即上一步落子方下出了四连），则非落子方胜。
  - 否则，如果落子方可以在某一列落子，使得有一个四连，则落子方胜。
  - 否则，如果落子方看到非落子方可以在两个不同的列落子，都产生四连，那么它将因为无法同时防守到两个而认负，非落子方胜。
- `must_move()`：判断当前局面落子方是不是有必须要下的防守落子。如果落子方看到非落子方可以在某一列落子，产生四连，那么它必须下在该列阻止对方落在同一位置，否则返回-1.

1.2 信心上限树(UCT)结构

由于采用指针和即时生成清理来建立树的话，会非常耗时，我们选择在该结构的头文件直接用数组的方式提前声明一个树节点结构的数组 node_list，以及一个记录节点数量的 node_list_count。（当然，还有我们在上一节提到的 Board_Condition 结构的实例 board_condition）

UCT 树的节点结构 UCT_Node 内存有该节点处的遍历次数，获胜次数，以及其父节点子节点的编号。考虑到我们始终把 0 号节点作为根节点，我们可以安全地将子节点数组全部初始化为 0，只需在使用时判断是否为 0 即可知道子节点是否存在。

我们为这个结构提供了一个接口 UCB1()来快速计算该节点的信心上限值。

基于这个结构，我们建立了两个函数：

- column_choice(int k)：在编号为 k 的 UCT 树节点上，确定 UCT 选择的分支。我们在确定分支时，做了如下考虑：
  - 如果 must_move 返回了一个非平凡值，那么该节点有唯一的选择。
  - 否则，如果有某个子节点编号 0，则存在子节点未被选择过，我们从未被选择过的子节点中随机选择一个。
  - 否则，所有子节点都被选择过，我们选择 UCB 值最大的节点。
- MCTS(int k)：在编号为 k 的 UCT 树节点上，进行树的搜索，返回胜利方的编号。我们首先选择分支，然后我们对 board_condition 进行改动，取得子节点对应 MCTS 返回值，然后恢复改动，对应地更新编号为 k 的节点的信息。

1.3 一些其它考虑

在实现过程中，我们曾经对其中部分函数进行了额外的考虑，但我们没有在最终的程序中体现。

我们曾考虑过让模拟过程更加聪明，在 column_choice 函数中考虑是否下某步后可以达到必胜局面，但将这一步考虑加入函数后，整个 3 秒的事件内进行的搜索数量减少了，参考 saiblo 平台上的对战结果（ver 0.3 vs ver 0.2）中的调试数据，其中最终选择的分支的搜索次数减少了 5 倍左右，这对 AI 进行远视的考虑是无益的，事实也证明它和现有版本的 AI 对战时的胜率只有 10%。

事实上，我们对此可以这样的考虑：我们在对节点进行更智能的分支选择时，相当于对每一节点进行更多的预先搜索，这意味着整体的搜索次数就会减少。预先搜索的越多，分支选择会越智能，但整体的搜索就会越短视。我们最终取得的平衡即为最终的版本。

2 测试结果

我们综合了 saiblo 平台上的测试结果，一共对每个 AI 样例测试了 8 轮（先后手各 8 场）。

我们首先进行一些说明：我们最早部署的稳定版 AI 是 ver 0.2 v36，但在此之后的一些调试中我们发现这一版 AI 在极少数情况下，即使自己先手已有三连，下一步即赢得比赛时，会因为对手有无法防守的两个三连而放弃比赛随机落子。我们针对这个问题继续对这一版本

的 AI 进行改进，最后部署稳定版 AI 是 ver 0.2 v45。由于 saiblo 平台上的批量测试次数有限，我们仍然将旧版 AI 的测试结果包括在内列举在下面以获得更大的数据量，但我们会具体标注其中哪一些对局收到上述 bug 影响。这一 bug 对终局前的 UCT 树搜索没有影响。(因为在树中搜索时我们会在走出必胜局面时即判断胜利，不必实际走出四连)

2.1 统计结果

我们给出全部 8 轮样例测试结果:

- 测试 1：https://www.saiblo.com/batch/23
  - 版本：v36；胜率：98%
  - 失败对局:
    - AI94（后手）
    - AI98（先手）
- 测试 2：https://www.saiblo.com/batch/155
  - 版本：v36；胜率：96%
  - 失败对局:
    - AI4（后手）（bug，对局地址 https://www.saiblo.com/match/297960/）
    - AI28（后手）（bug，对局地址 https://www.saiblo.com/match/297935/）
    - AI70（后手）（bug，对局地址 https://www.saiblo.com/match/297933/）
    - AI100（后手）
- 测试 3：https://www.saiblo.com/batch/160
  - 版本：v40；胜率：94%
  - 失败对局:
    - AI40（先手）（bug，对局地址 https://www.saiblo.com/match/298654/）
    - AI22（后手）
    - AI94（先手）
    - AI100（后手）
    - AI88（后手）
    - AI94（后手）
- 测试 4：https://www.saiblo.com/batch/438
  - 版本：v43；胜率：97%
  - 失败对局:
    - AI54（后手）（bug，对局地址 https://www.saiblo.com/match/328489/）
    - AI94（后手）
    - AI100（后手）
- 测试 5：https://www.saiblo.com/batch/1195
  - 版本：v45；胜率：95%
  - 失败对局:
    - AI74（后手）
    - AI94（先手）
    - AI100（后手）
    - AI86（先手）
    - AI96（先手）
- 测试 6：https://www.saiblo.com/batch/1203
  - 版本：v45；胜率：97%
  - 失败对局:
    - AI66（后手）
    - AI92（后手）
    - AI94（后手）
- 测试 7：https://www.saiblo.com/batch/1207
  - 版本：v45；胜率：97%
  - 失败对局:
    - AI94（后手）
    - AI96（后手）
    - AI98（先手）

- 测试 8：https://www.saiblo.com/batch/1211
  - 版本：v45；胜率：99%
  - 失败对局：
    - AI94（后手）

我们综合全部 800 场比赛得到结果，实际测得胜率为 96.63%，不计 bug 对局预计胜率为 97.2%。其中对战排名前五的 AI（AI92~AI100）胜率为 78.75%。所有比赛都未达到平局，均分出了胜负。

若只记最终部署的稳定版 AI，综合全部 400 场得到结果，胜率为 97.00%，其中对战排名前五的 AI 胜率为 77.5%。

2.2 对局分析

我们在失败对局中逐一进行了观察，发现这一 AI 存在如下两个薄弱点：

- 在早期对局，对潜在的聚集块尚不够警觉。事实上，如果一方在底层可以成功地做出 2*2 的聚集块，而且周围的防守不够密集的话，是比较容易完成连续杀招杀死比赛的。我们的 AI 在这方面的警惕性不足，有可能导致一些水平较低的 AI“乱拳打死老师傅”
- 在游戏中期无法预计残局时的奇偶威胁状况。事实上，奇偶威胁理论时传统四子棋必胜策略中的经典结论，它描述了在棋局中，尤其是在残局中，奇数行上的威胁和偶数行上的威胁的优劣与控制效果。我们的 AI 对这方面的考虑较少，导致拖入残局后由于柱与柱之间全部留有对手威胁而无子可落。

这两个方面是可以继续改进的。