2020 秋-计算概论 (A) 大作业报告

epcm

January, 2021

Abstract

本文主要探究了在短时限制下的不围棋 MCTS 算法的设计与实现，总结出了“大道至简”经验原则并实践于 Bot 设计。同时解析了基于 Qt 图形界面应用设计的不围棋对战软件制作过程，对一些重要功能的实现进行了详细的说明。

Contents

1 短时不围棋对弈的 MCTS 算法初探

1.1 算法选择与算法设计中的一个原则

作为一种启发式搜索算法，Monte Carlo Tree Search(MCTS) 在众多完全信息博弈游戏中都有着不俗的表现，如围棋，国际象棋等。相较传统的 $\alpha - \beta$ 剪枝等搜索算法，MCTS 可以更好地利用前面的搜索结果，启发式地选择赢面更大的节点进行下一步搜索，从而在同样的算力下取得更好的效果，并且无需很多关于游戏的先验知识。

在一般的 MCTS 应用场景中，Bot 都配备有非常强的算力或较长的运算时间，因此可以构造出足够大的游戏树。而在本次大作业的时间资源条件较为吃紧，同时服务器可分配的算力亦有限，可构造游戏树小，因此需要采取与一般论文不太相同的优化策略，这也是本次作业 AI 部分的主要难点。

在反复的迭代过程中，笔者逐渐总结出一个短时不围棋博弈的算法原则："大道至简 (Less Is More)"，即在较短的时间限制条件下，增加搜索节点数往往比提高评估的准确度更有效，因此在算法中应当尽可能减少随机模拟、局部匹配等降低单次节点评估速度的操作，以增加搜索的节点数目。我们可以从直观上来解释这一经验规律，不围棋中的游戏树是确定的，所以可以猜测随着搜索节点数目增加，胜率逐渐趋于 1，增加搜索节点数的边际收益递减，所以在短时条件下，搜索树小，增加搜索节点数带来的胜率提升大。

笔者的 Bot 中，这一原则主要以以下几种优化体现：

1. 在 Default Policy 阶段不做模拟，直接使用评估函数估计节点质量
2. 采用仅依赖于双方可行位置数的简单评估函数形式
3. 尽可能削减类的数量，减少跨类的调用，牺牲部分可读性
4. 按棋盘的遍历顺序对节点进行 expand 操作，不使用随机选择

事实证明这一策略是较为有效的，最终笔者的 Bot 在三场瑞士轮对战中综合排名进入前 3 位，天梯最高排名 15 位，长期维持在排行榜首页。下面将结合 MCTS 的一般过程以及笔者 Bot 迭代过程具体解析算法实现。

1.2 比赛版本的算法实现解析

MCTS 的一个循环过程如下：首先选出一个值得探索的节点 (Selection)，然后将这个节点的一个子节点展开 (Expand)，对这个节点使用 Default Policy 得到一个局面评估值 (Simulation)，最后将这个评估值反馈至所有父节点 (Backpropagation)。

```
while (clock() - start < timeout)
{
    node_count++;
    Node *expand_node = node->treePolicy();
    double reward = expand_node->defaultPolicy();
    expand_node->backup(reward);
}
```

Listing 1: Loop of MCTS

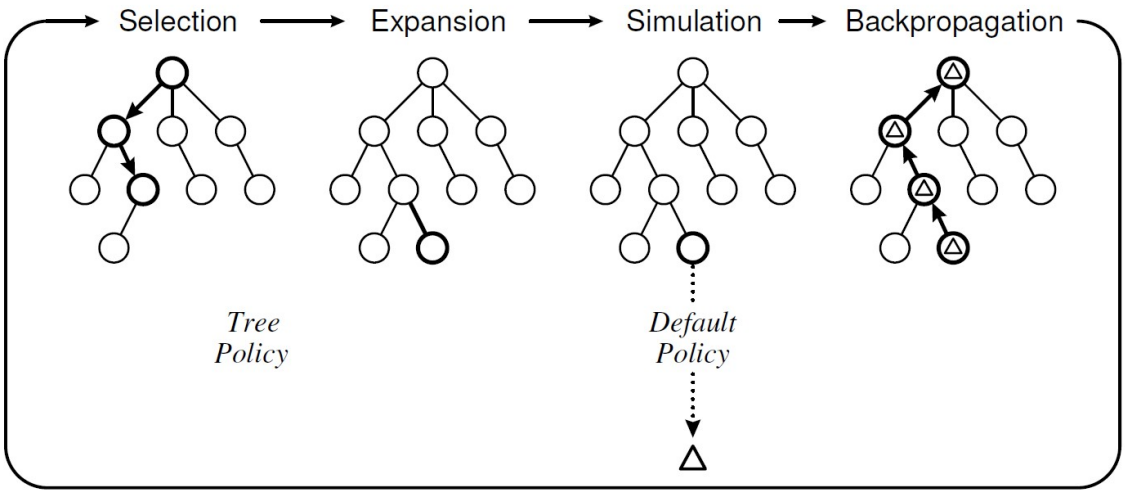

Figure 1.1: MCTS 算法的一般过程 [1]

1. **Tree Polcy:** 包括 Selection 与 Expansion 两个阶段，基本策略是先找当前未选择过的子节点，如果有多个则顺序选。如果都选择过就找体现 exploration/exploitation 权衡的 UCB 值最大的节点递归计算，到达叶节点则直接返回当前节点。
   最为重要的是 UCB 值的计算公式

$$UCB = \frac{quality\_value}{visit\_times} + 2C\sqrt{\frac{\log(2 * parent\_visit\_times)}{visit\_times}}$$

   C 值越大，代表算法越倾向于探索当前看来不是最优的解，经手工调参，C 值取 0.1 时 Bot 表现较好。

```
Node *Node::bestChild(double C)
{
    double max_score = -2e50;
    Node *best_child = NULL;
    for (int i = 0; i < countChildrenNum; i++)
    {
        Node *p = children[i];
        double score = p->quality_value / (p->visit_times) + 2 * C * sqrt(log
    (2 * visit_times) / (p->visit_times));
        if (score > max_score)
        {
            max_score = score;
            best_child = p;
        }
    }
    return best_child;
}
Node *Node::expand()
{
    int a = available_choices[countChildrenNum];
    int x = a / 9;
```

```
    int y = a % 9;
    Node *new_node = new Node;
    children[countChildrenNum++] = new_node;
    new_node->parent = this;
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            new_node->current_board[i][j] = current_board[i][j];
    new_node->col = -col;
    new_node->current_board[x][y] = col;
    new_node->getAviliableAction();
    return new_node;
}
Node *Node::treePolicy()
{
    if (maxChildrenNum == 0) //当treePolicy到达叶节点时
        return this;
    if (countChildrenNum >= maxChildrenNum)
    {
        Node *p = bestChild(EXPLORE);
        return p->treePolicy();
    }
    else
        return expand();
}
```

Listing 2: Tree Policy

2. **Simulation:** 这一部分对当前节点的局面进行评估，返回一个评估值。这里为了减少单次评估的时间，我选择了直接使用估值函数对局面进行估计。评估函数设计为最简洁的形式以节约运算时间。

$$reward = n_1 - n_2$$

n_1 为我方的可行位置数量，n_2 为对方的可行位置数量，在实际代码中，笔者返回了 reward 的负值以方便下一步操作。

```
double Node::quickEvaluate()
{
    int n1 = 0, n2 = 0;
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
        {
            bool f1 = judgeAvailable(i, j);
            col = -col;
            bool f2 = judgeAvailable(i, j);
            col = -col;
            if (f1 && !f2)
                n1 ++;
            else if (!f1 && f2)
                n2 ++;
```

```
        }
    return n2 - n1;
}
double Node::defaultPolicy()
{
    return quickEvaluate();
}
```

Listing 3: Simulation

3. **Backpropagation:** 这一部分将 expand 节点的评估值逐级向上反馈，同时修改访问次数，这里采用迭代而非递归以加速运算。

```
void Node::backup(double reward)
{
    Node *p = this;
    while (p)
    {
        p->visit_times++;
        p->quality_value += reward;
        reward = -reward;
        p = p->parent;
    }
}
```

Listing 4: Backpropagation

1.3 Bot 迭代过程与改进方向

Bot 第一个可运行版本写就于 2020.12.7，最终版于 2020.1.3 敲定，期间共有三次实现性能飞跃的改进。在第一次改进中，经过对模拟步数的手工调参，笔者将 Default Policy 的模拟过程全部删去，采用直接估值的方案。第二次改进中，笔者对代码进行了重构，删去了冗余的类，并将 Rollout Policy 从随机选择改为遍历顺序选择。第三次改进中，笔者对 UCB 中的 C 值进行调参，从论文推荐的 $\frac{\sqrt{2}}{2}$ 调整到 0.1，使 Bot 的选择更为稳健。

笔者猜测可行的改进方向为修改估值函数的形式，考虑到距当前局面步数较近的节点能更精准描述当前当前局面，而距当前局面步数很远的节点对胜负的估值更准确，可考虑加大这两类节点评估值所占的比重，即 $n_1 - n_2$ 为变量，构造一个两端翘起，中间下凹，类似于二次函数的估值函数。笔者曾尝试过这样的思路，但因调参较复杂而放弃。

另一种可能的思路为使用先验知识，即将一些已知的优质战略法则输入 Bot，如残局库，或者是开局算法，笔者曾试验在评估节点的过程中使用先验知识，但因为对搜索节点数影响过大而放弃，因此在使用这些知识时最好也应该放在 MCTS 的循环之外，尽量减少执行所需的时间开销。

2 基于 Qt 的图形界面应用设计

Qt 作为一个老牌的 C++ 应用程序开发框架，广泛用于开发 GUI 程序，其库函数丰富，文档详实，UI 设计便捷，广受历年来的计算概论 (A) 大作业选手喜爱。笔者的应用基于 Qt5.15.2 开发，设计分辨率为 2160*1440，平台为 Windows 10，如不能正常显示则请手动改变缩放或更换 2k 分辨率设备。

2.1 UI 设计

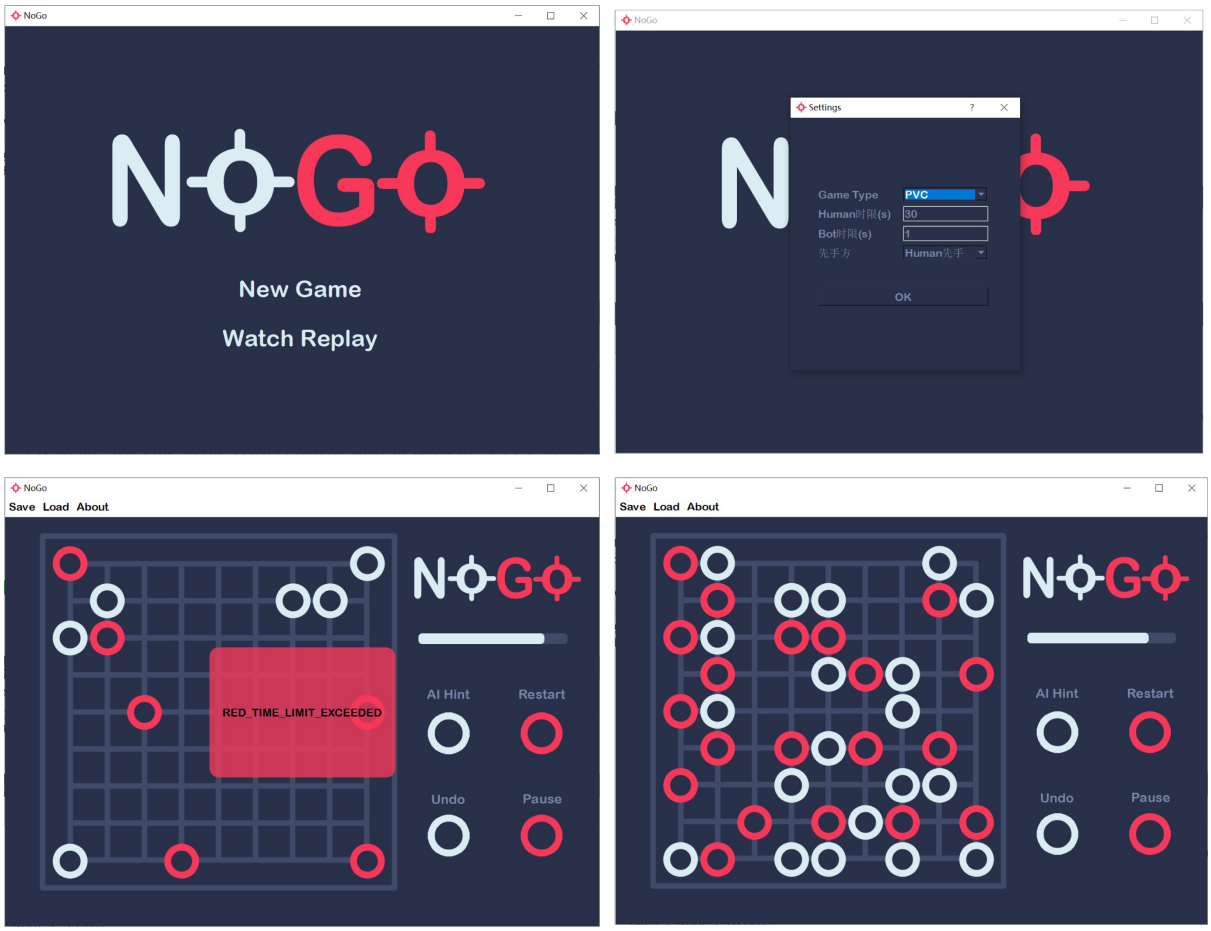

Figure 2.1: 游戏界面展示

本应用的 UI 设计采用了扁平化设计风格，简洁大方，采用棋子的圆形为主要的设计元素。字体使用无衬线的圆润字体 Arial Rounded MT Bold，平衡现代感与亲和感。配色采用深色偏蓝底色，更富科技感，使用较低饱和度的蓝色和红色作为主色，在防止刺眼的同时增加一些趣味性，同时将圆设置为中空以防止画面过满。交互设计上尽可能做到用户友好，游戏过程中常用功能单列右侧，不常用的存读档收纳在菜单栏，返回键则隐藏在游戏界面的 LOGO 中。

2.2 项目结构与基本逻辑

项目结构及各个文件的功能如下图所示

```
|-- NoGo
```

```
|-- NoGo.pro                //项目管理文件，用于构建等
|-- Headers
|   |-- ai                  //储存Bot
|       |-- aimcts.h
|   |-- gamewidget.h        //控制实际进行游戏的棋盘界面
|   |-- hintwidget.h        //自己实现的窗口类，用于错误提示以及AI提示功能
|   |-- mainwindow.h        //以栈布局储存三个界面并实现界面切换
|   |-- referee.h           //控制游戏进程调度的类，在gamewidget中实例化
|   |-- settingsialog.h     //控制设置界面
|   |-- utils.h             //全局变量
|   |-- welcomewidget.h     //控制欢迎界面
|-- Sources
|   |-- ai
|       |-- aimcts.cpp
|   |-- gamewidget.cpp
|   |-- hintwidget.cpp
|   |-- main.cpp            //程序的入口
|   |-- mainwindow.cpp
|   |-- referee.cpp
|   |-- settingsialog.cpp
|   |-- welcomewidget.cpp
|-- Forms                   //各个界面的部分UI设计
|   |-- gamewidget.ui
|   |-- settingsialog.ui
|   |-- welcomewidget.ui
|-- Resources
|   |-- pic.qrc             //二进制储存游戏中调用的图片资源
```

基本逻辑即 main 函数打开 mainwindow，mainwindow 切换到 welcomewidget 界面，之后通过鼠标交互切换到 settingsdialog 界面，设置过程中改变 referee 中参数，之后切换到 gamewidget 界面，并初始化 referee 开始游戏。

2.3 主要功能的实现解析

1. **设置游戏参数**

```
class SettingDialog
{
    signals:
        void startGameSignal();
        void gameModeSelectSignal(int index);
        void firstPlayerSelectSignal(int index);
        void humanTimeEditSignal(const QString &arg1);
```

```
        void botTimeEditSignal(const QString &arg1);
    private slots:
        void on_OKPushButton_clicked();
        void on_GameModeSelect_currentIndexChanged(int index);
        void on_FirstPlayerSelect_currentIndexChanged(int index);
        void on_HumanTimeEdit_textChanged(const QString &arg1);
        void on_BotTimeEdit_textChanged(const QString &arg1);
}
class Referee
{
    public slots:
        void setBotTimeLimit(const QString &text);
        void setHumanTimeLimit(const QString &text);
        void setGameMode(int index);
        void setFirstPlayer(int index);
}
```

Listing 5: 设置游戏参数相关函数与变量

基于 Qt 信号槽机制，在检测到文本框/选择框修改的信号发送后，settingdialog 的槽函数进行信号接力再次发送信号，gamewidget 的 referee 接收信号并修改参数，从而实现了跨类的通信。

2. **存读档**

```
class Referee
{
    public slots:
        void saveGame();
        void loadGame();
    public:
        // 历史记录
        QJsonArray m_history;
        // 通过历史记录恢复局面
        void setBoardByHistory();
}
```

Listing 6: 存读档相关函数与变量

基于 Qt 的 Json 支持，在游戏过程中用一个 QJsonArray 变量 m_history 记录落子顺序，存档时将其与时间限制等其他信息一同写入一个 QJsonDocument 变量，之后通过 Qt 调用 windows 资源管理器对文件进行命名、位置选择、储存。

读档时则将 json 文件中信息解析并存入相应变量，并调用 Referee::setBoardByHistory 恢复盘面，发送暂停信号，等待玩家按下右侧 Continue 键继续游戏。

3. **时间控制相关功能**

```
class Referee
{
    public slots:
```

```
        // 判断是否达到时限
        void judgeTime();
    public:
        // 时间信号生成器
        QTimer* m_timer = new QTimer(this);
        QTimer* m_replay_timer = new QTimer(this);
        // Bot限时
        double m_bot_time_limit = 1;
        // 人类限时
        double m_human_time_limit = 30;
        // 记录开始计时的时刻
        int m_start_time;
        // 暂停标记
        bool m_paused = false;
        // 暂停时经过时间
        int m_time_when_paused;
}
```

Listing 7: 时间相关函数与变量

程序中许多功能的实现，如暂停，超时提示，剩余时间显示，回放等，都离不开对时间的控制。本程序中的时间控制基于 clock() 函数与 QTimer 类，QTimer 类的变量能以固定的时间间隔发射信号，从而保持对超时的检测或是时间条的更新。

若需要给某一过程定时，则可在过程开始时调用 clock() 记录下当前时间，在每次接收到 QTimer 信号时检查一下当前时间与开始时间的差值。这样的设计虽然繁琐，但能够为暂停等功能的实现提供便利，暂停时只需记下当前时间与开始时间的差值，并停止 QTimer 信号；恢复时将开始时间设置为当前时间-暂停时经过时间，并开始 QTimer 信号即可。

4. **其他功能**

上文仅仅是选择了三个典型的功能进行了解析，将其中涉及的信号槽机制，存读系统，计时系统进行组合即可较为简易地获得程序的其他功能，如 AI 落子提示、回放、暂停与继续 (游戏中及观看录像时都可以使用)、悔棋、重新开始、超时/非法位置/胜利提示等，在这里不再赘述 (总之功能多于四个就对了)，详细实现见代码。

2.4 总结与改进方向

在这次作业中，笔者第一次体验了完整的软件制作流程，从策划功能，到 UI 设计，到类的设计与功能封装，再到对每个函数进行实现。同时我第一次体会到了 OOP 的强大: 能够使大型的工程保持条理清晰，还可以方便复用，只需要重写一个函数，就可以构造自定义的窗口类。笔者也学会了如何去读文档，如何在不懂其实现原理的情况下调用函数去实现那些稀奇古怪的功能。

这次图形界面制作中还留有一些遗憾，首先是初期对 Qt 和 OOP 不够熟悉，类的设计不够优雅，封装也做得不太好；另外在软件编写过程中未能形成良好的注释习惯，导致程序可读性下降，开发日志也未能完整记录开发过程，命名规范性也不足；还有就是初期规划的联机功能因为各种各样的原因而未能实现；最后在软件不同分辨率设备上的适配不够好。

如有机会我将继续完善这一项目。程序设计中难免有未能发现的 Bug，有些实现可能也显得十分笨拙，还请助教见谅，十分感谢助教能将这份报告延后到考试结束，让我能有更多的时间来回顾这次大作业，刘老师和助教们一学期来辛苦了！

References

[1] C. B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[2] C. -W. Chou, O. Teytaud, and S. -J. Yen. "Revisiting Monte-Carlo Tree Search on a Normal Form Game: NoGo". In: *Applications of Evolutionary Computation*. Ed. by Cecilia Di Chio et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 73–82. ISBN: 978-3-642-20525-5.

[3] Y. Wang and S. Gelly. "Modifications of UCT and sequence-like simulations for Monte-Carlo Go". In: *2007 IEEE Symposium on Computational Intelligence and Games*. 2007, pp. 175–182. DOI: 10.1109/CIG.2007.368095.

[4] 梁国军 et al. "UCT 算法在不围棋博弈中的实现". In: 韶关学院学报 8 (2015), pp. 17–21.

[5] 郭倩宇 and 陈优广. "基于价值评估的不围棋递归算法". In: 华东师范大学学报 (自然科学版) 2019.1 (2019), pp. 58–65.