FuturesFish 象棋 AI 引擎实验报告

1950509 马家昱 电子与信息工程学院 计算机科学与技术

装 订 线

目录

1. 实验概况

自人工智能课程设计项目——象棋 AI 引擎发布以来，历经两个月的开发与调试，我们最终完成了 FuturesFish，一个具有一下特性：

1. **代码架构良好，封装接口独立，具有高内聚、低耦合的工程标准**
2. **包括基础版与 NNUE（神经网络）版在内的多个版本**
3. **内含大量创新与优化，搜索效率为上届冠军代码 10 倍**
4. **高棋力，先后手轻松战胜冠军代码与象棋巫师大师水平**

的人工智能象棋 AI 引擎。在四位组员的通力合作下，我们对优秀源码进行了深入的研究，并且不断地搜寻更加先进的优化方法与搜索算法应用到我们的程序中，独立的进行实验与调式比较，最终得到了以上成果。

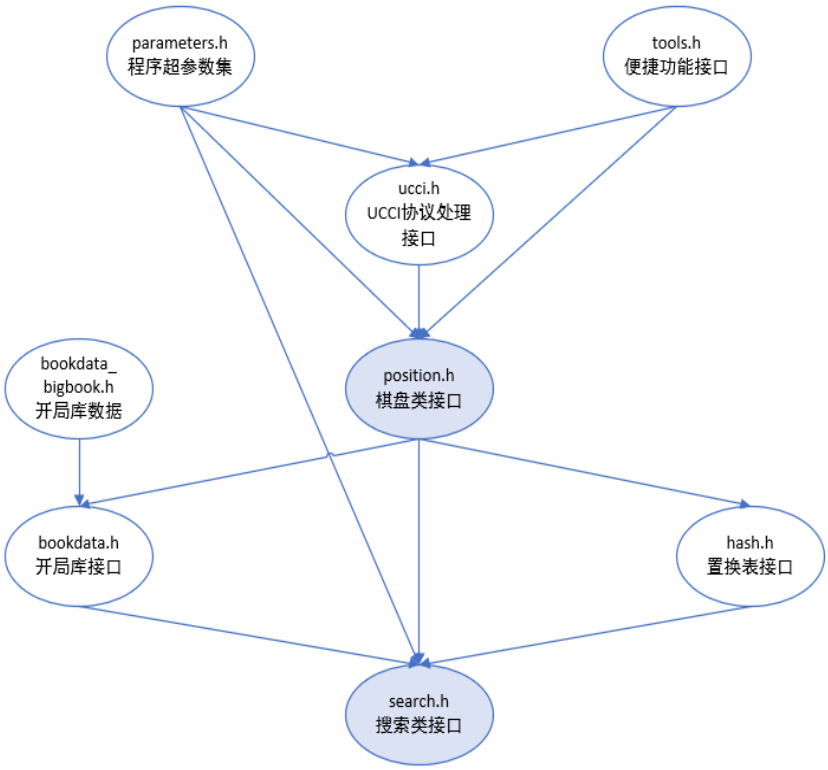
1.1 基本代码架构

本组采用**设计-实现-封装-组合-调试**的实验流程，在具体编写代码之初首先完成了对整体代码的架构设计，主要包括 6 个模块：

1. ucci 模块：ucci 通信与信息的处理与反馈
2. position 模块：棋盘表示与着法生成
3. hash 模块：置换表的加入与存储

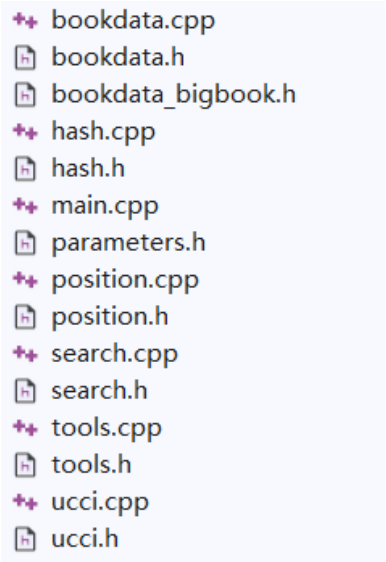
4. bookdata 模块：开局库的搜索与存储
5. search 模块：主要搜索模块
6. tools 模块：工具类，如计时器与自定义字符串处理，加密与解密

其模块间的基本调用关系如下图所示



主要模块调用关系

每个模块使用 C++面向对象方式实现，均采用接口方式互相传递参数，极大的方便了分组开发与组装调试。

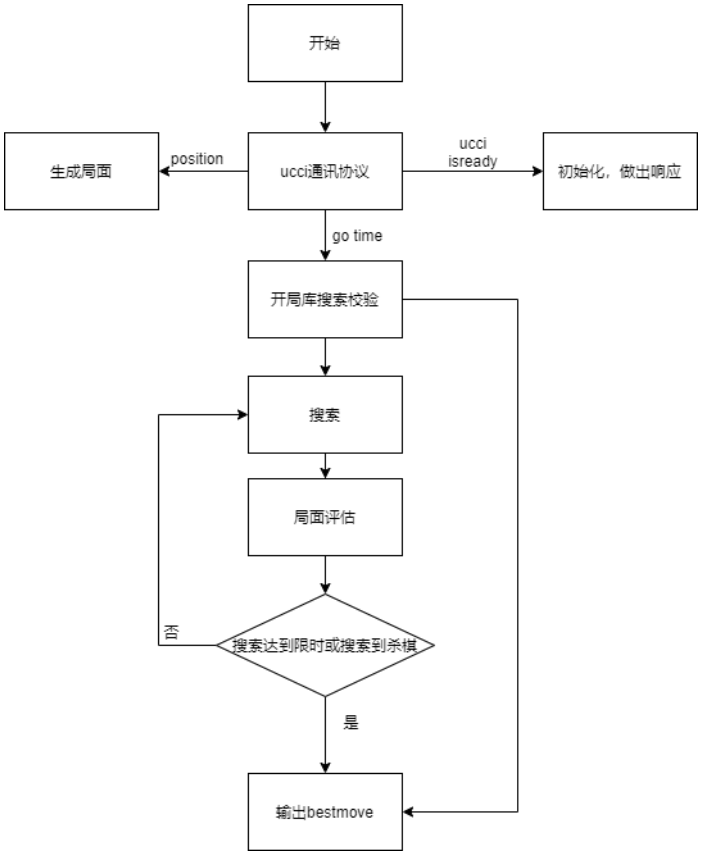


清晰明了的模块设计

```
public:
    //HashClass();

    void Clear(); // 清空置换表

    /**
     * @brief           即ProbeHash()
     *
     * @param pos       目前棋盘
     * @param vlAlpha   当前alpha
     * @param vlBeta    当前beta
     * @param nDepth    当前深度
     * @param mv        获取到的最优动作
     * @return int      获取到的分值，如果查不到或不符合则返回-MATE_VALUE
     */

    int Load(const PositionClass &pos, int vlAlpha, int vlBeta, int nDepth, int &mv, bool bUseNull = false); // 提取置换表项
    /**
     * @brief           即RecordHash()
     *
     * @param pos       目前棋盘
     * @param nFlag     当前节点状态，参照parameters.h
     * @param vl        当前分值
     * @param nDepth    当前深度
     * @param mv        最优动作
     */
    bool Save(const PositionClass &pos, int nFlag, int vl, int nDepth, int mv); // 保存置换表项
```

接口化的开发方法与规范化的注释

在对各个模块的功能实现简单验证后，我们将所有模块组装在一起，简单调试后就得到了可以正常运行的象棋引擎。程序的基本流程如下图所示。



引擎功能实现的流程

1.2 搜索算法

搜索算法是象棋引擎的核心算法。我们将多种基础与高级的搜索算法运用到了程序之中，包括并不限于以下方法：

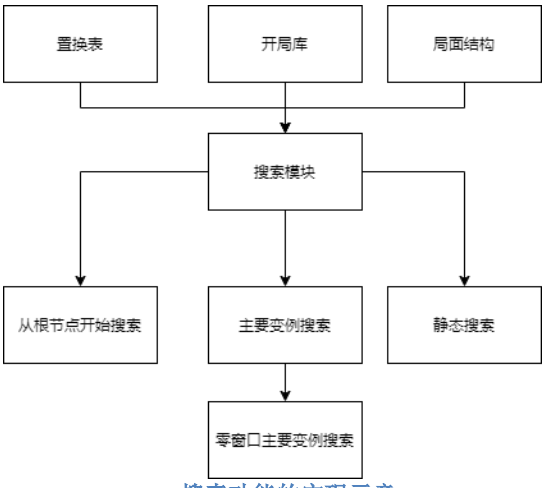
1. 主要变例搜索
2. 静态搜索
3. 空着裁剪
4. 期望窗口的应用（对迭代加深的改进）
5. MTD 缓存增强试探法

我们还将哈希与历史表运用到了着法生成之中,其着法的生成步骤为：

1. 先搜索哈希表
2. 其次是杀手节点
3. 对历史表进行排序
4. 再依次遍历所有着法

在搜索策略的改进中，我们采用了延缓死亡的方法，即搜索到被杀棋后，会通过主动放弃车，马，炮等来延缓老将被吃（水平线效应）。除此之外，我们从网络上搜寻了其他象棋软件的**开局库**，将其整合并加入到了我们的程序当中，最终开局库数量达到了 600000+。

搜索功能的主要实现示意图如下所示：



搜索功能的实现示意

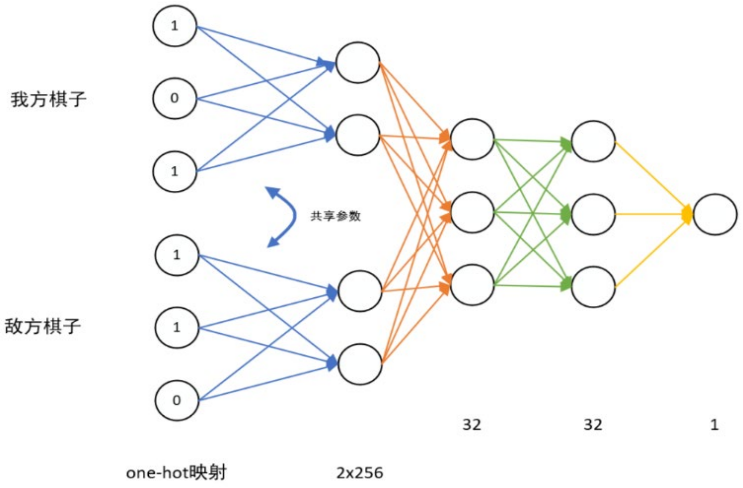
搜索的函数实现如下：

```
//静态搜索过程
int SearchClass::SearchQuiesc(int alpha, int beta)
//零窗口主要变例搜索
int SearchClass::SearchPVCut(int beta, int depth, bool useNull)
//主要变例搜索
int SearchClass::SearchPV(int alpha, int beta, int depth)
//根结点搜索过程
int SearchClass::SearchRoot(int depth)
//搜索主函数
int SearchClass::SearchMain(PositionClass& Pos_, UcciCommClass& Ucci)
```

1.3 使用 NNUE 进行估值

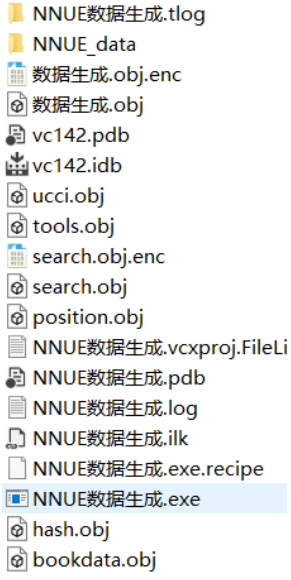
Efficiently Updatable Neural Network（可快速更新神经网络）

在传统的象棋引擎中，一般使用棋盘中棋子棋力的方法对局面进行估值。我们创新性的使用了 NNUE 进行估值函数的代替，采用对打方式自行生成数据，并且尝试不同的网络结构对其进行训练，并比较结果。NNUE 网络是一个浅层神经网络，输入层为当前棋局中棋子与位置的 one-hot 向量，经过中间层处理后，输出当前局面的评价值。



网络结构示意图

在数据生成方面，我们调用已经完成的 position 类和 search 类进行自己与自己对打随机生成数据。为了测试 3 层搜索与 5 层搜索数据的优劣程度，需要在短时间内生成大量数据。为此我们独立编写了数据生成程序，使用所有小组成员的电脑进行运算：



为生成数据独立编写的程序

```
fen rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RNBAKABNR w - - 0 1 9
fen rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1CN4C1/9/R1BAKABNR b - - 0 2 0
fen r1bakabnr/9/1cn4c1/p1p1p1p1p/9/9/P1P1P1P1P/1CN4C1/9/R1BAKABNR w - - 0 3 8
fen r1bakabnr/9/1cn4c1/p1p1p1p1p/9/9/P1P1P1P1P/1CN6/7C1/R1BAKABNR b - - 0 4 10
fen r1bakabnr/9/1cn6/p1p1p1p1p/9/9/P1P1P1P1P/1CN4c1/7C1/R1BAKABNR w - - 0 5 97
fen r1bakabnr/9/1cn6/p1p1p1p1p/9/9/P1P1P1P1P/2N4C1/7C1/R1BAKABNR b - - 0 6 -88
fen r1bakab1r/9/1cn3n2/p1p1p1p1p/9/9/P1P1P1P1P/2N4C1/7C1/R1BAKABNR w - - 0 7 98
fen r1bakab1r/9/1cn3n2/p1p1p1p1p/9/9/P1P1P1P1P/7C1/4N2C1/R1BAKABNR b - - 0 8 -71
fen 1rbakab1r/9/1cn3n2/p1p1p1p1p/9/9/P1P1P1P1P/7C1/4N2C1/R1BAKABNR w - - 0 9 80
fen 1rbakab1r/9/1cn3n2/p1p1p1p1p/9/9/P1PNP1P1P/7C1/7C1/R1BAKABNR b - - 0 10 -66
fen 1rbakab1r/9/2n3n2/p1p1p1p1p/9/9/PcPNP1P1P/7C1/7C1/R1BAKABNR w - - 0 11 70
fen 1rbakab1r/9/2n3n2/p1p1p1p1p/9/9/PcPNP1PCP/9/7C1/R1BAKABNR b - - 0 12 22
fen 1rbakab1r/4n4/6n2/p1p1p1p1p/9/9/PcPNP1PCP/9/7C1/R1BAKABNR w - - 0 13 91
fen 1rbakab1r/4n4/6n2/p1p1p1p1p/9/9/PcPNP1PCP/9/7C1/1RBAKABNR b - - 0 14 -85
fen 2bakab1r/4n4/1r4n2/p1p1p1p1p/9/9/PcPNP1PCP/9/7C1/1RBAKABNR w - - 0 15 105
fen 2bakab1r/4n4/1r4n2/p1p1p1p1p/9/9/PcP1P1PCP/9/2N4C1/1RBAKABNR b - - 0 16 -89
fen 2bakab1r/4n4/1r4n2/p1p1p1p1p/9/1c7/P1P1P1PCP/9/2N4C1/1RBAKABNR w - - 0 17 94
fen 2bakab1r/4n4/1r4n2/p1p1p1p1p/9/1c7/P1P1P1PCP/4B4/2N4C1/1RBAKA1NR b - - 0 18 -88
fen 2bakab1r/9/1r4n2/p1pnp1p1p/9/1c7/P1P1P1PCP/4B4/2N4C1/1RBAKA1NR w - - 0 19 98
fen 2bakab1r/9/1r4n2/p1pnp1p1p/9/1c7/P1P1P1PCP/4B3R/2N4C1/1RBAKA1N1 b - - 0 20 -86
fen 2bakab1r/8n/1r7/p1pnp1p1p/9/1c7/P1P1P1PCP/4B3R/2N4C1/1RBAKA1N1 w - - 0 21 112
fen 2bakab1r/8n/1r7/p1pnp1p1p/9/1c7/P1P1P1PCP/4BR3/2N4C1/1RBAKA1N1 b - - 0 22 -106
fen 2bakab1r/9/1r4n2/p1pnp1p1p/9/1c7/P1P1P1PCP/4BR3/2N4C1/1RBAKA1N1 w - - 0 23 113
fen 2bakab1r/9/1r4n2/p1pnp1p1p/9/1c3R3/P1P1P1PCP/4B4/2N4C1/1RBAKA1N1 b - - 0 24 -108
fen 2bakab1r/9/1r4n2/p1pn2p1p/4p4/1c3R3/P1P1P1PCP/4B4/2N4C1/1RBAKA1N1 w - - 0 25 201
fen 2bakab1r/9/1r4n2/p1pn2p1p/4p4/1cP2R3/P3P1PCP/4B4/2N4C1/1RBAKA1N1 b - - 0 26 -106
fen 2bakab1r/9/1r4n2/p1pn2p1p/4p4/2P2c3/P3P1PCP/4B4/2N4C1/1RBAKA1N1 w - - 0 27 108
fen 2bakab1r/9/1R4n2/p1pn2p1p/4p4/2P2c3/P3P1PCP/4B4/2N4C1/2BAKA1N1 b - - 0 28 -106
fen 2baka2r/9/1R2b1n2/p1pn2p1p/4p4/2P2c3/P3P1PCP/4B4/2N4C1/2BAKA1N1 w - - 0 29 111
fen 2baka2r/9/1R2b1n2/p1pn2p1p/2P1p4/5c3/P3P1PCP/4B4/2N4C1/2BAKA1N1 b - - 0 30 -90
```

生成的数据格式

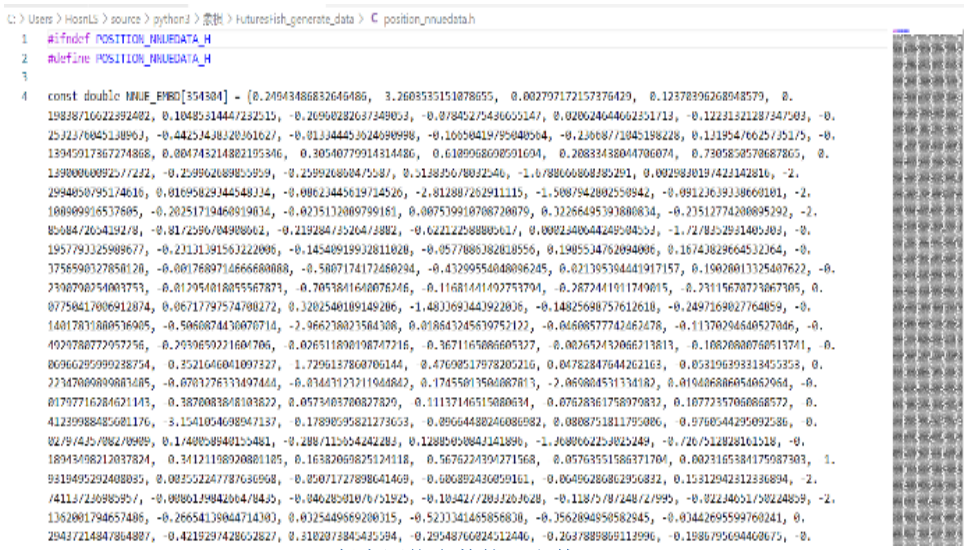
对于生成的结果，我们进行了如下预处理：

1. 数据读取
2. 数据逐行分割
3. 数据转换为 fen 串
4. 数据转换为默认白方先手格式（如果是黑方先手则翻转棋盘）
5. 水平镜像进行数据扩充
6. 翻转镜像进行数据扩充
7. 数据去重
8. 人工检查数据分布

9. 数据转化为 one-hot 向量

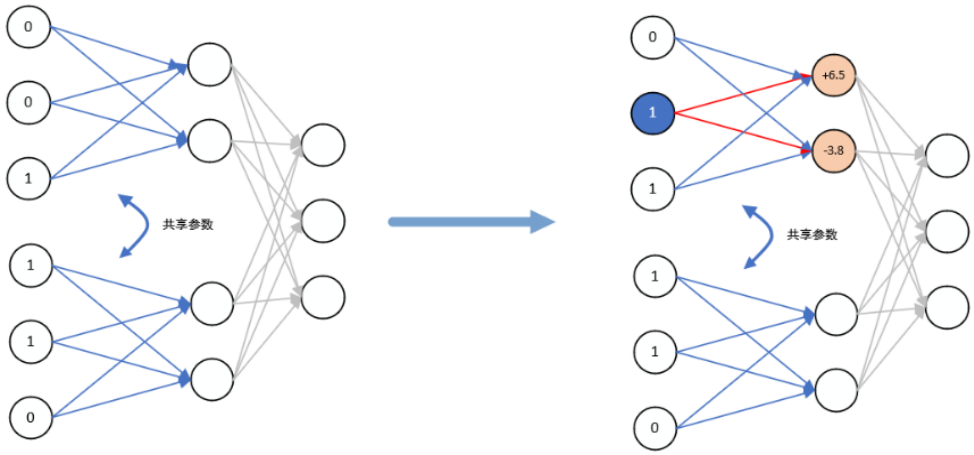
对数据进行预处理后，可以得到较大数据量数据，搜索 3 层和搜索 5 层的数据均超过了 5,000,000，足够进行网络的训练。

网络完成训练后，需要将参数导出，以便 C++语言下的象棋能够使用。由于提交版本的象棋不允许进行文件的读写，因此考虑将网络参数写入到单独的.h 文件中。编写了 Python 脚本自动实现，将生成的 position_nnuedata.h 头文件拷贝到装有 NNUE 模块的象棋主程序目录，执行编译即可。



保存网络参数的.h 文件

我们采用了增量更新方式，矩阵运算来提高运算效率。将计算第一隐藏层的矩阵乘法化简为矩阵加法。即当加入一个棋子时，输入层一个数据出现改变，将第一隐藏层所有神经元加上此输入数据对应的列向量；删除棋子时同理。



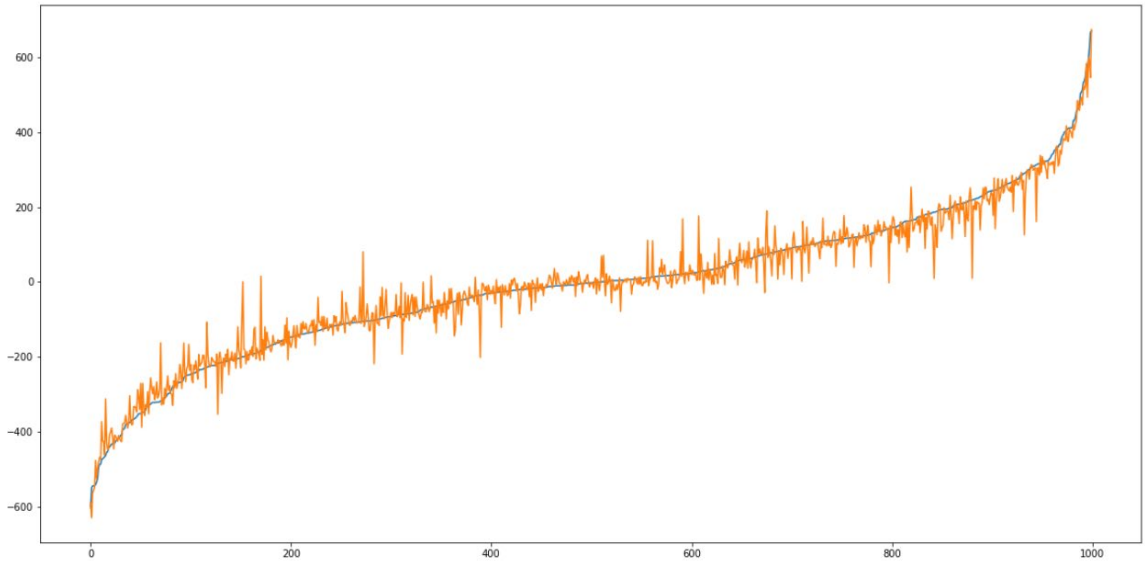
网络结构改进以增加运算效率

同时，我们使用了使用了 Intel oneMKL 和 openBLAS 两种底层运算加速库，评估函数速度对比如下：

openBLAS 前向传播速度 (次/s)		Intel oneMKL 前向传播速度 (次/s)	
double :	70600	double :	71197
float :	81190	float :	81621
int :	3274	int :	3279
short :	3288	short :	3337
short ;	3274	short ;	3231

Intel oneMKL 和 openBLAS 两种加速库对比

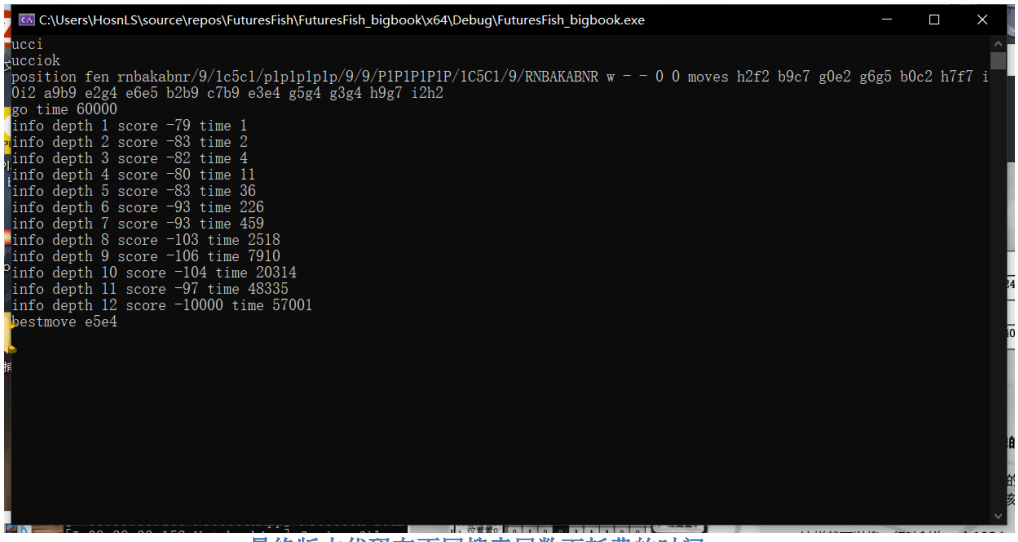
因为 NNUE 网络的训练收敛十分困难，即使在多次修改网络结构、仔细调节参数、增加迭代轮次的情况下，损失函数也仍然居高不下，损失函数到 2000 左右不再下降。



网络训练结果

1.4 结果

最终版本的代码平均搜索层数在 10 层左右，最深时可以达到 14-15 层。经多次实验，我们的程序可以在先后手的情况下均战胜冠军代码。并且我们测量了 FuturesFish 在不同搜索层数的时间，以此来进行优化，最终版的测量结果如下：



最终版本代码在不同搜索层数下耗费的时间

2. 个人工作

在本次实验中，我本人的分工为：

1.代码的版本控制，即组装各个模块，更新整合所有的改动以及找出并修改模块间存在的 bug，使得编译与运行能够通过。

2.我还负责对每个模块所占用的时间与资源进行测量，以便进一步的优化代码效率。

除此之外，我本人主要负责以下功能的具体实现：

1. tools 模块，如计时器，字符串处理，随机数产生工具的实现
2. hash 模块，即 Zobrist 键值以及哈希置换表的实现
3. bookdata 模块，将搜集到的不同格式与版本的开局库整合优化，并实现开局库的搜索与储存。

下面主要介绍置换表与开局库的实现过程。

2.1 哈希置换表

国际象棋的搜索树可以用图来表示，而置换结点可以引向以前搜索过的子树上。置换表可以用来检测这种情况，从而避免重复搜索。除此之外，每个散列项里都有局面中最好的着法，首先搜索好的着法可以大幅度提高搜索效率。因此如果在散列项里找到最好的着法，那么首先搜索这个着法，这样会改进着法顺序，减少分枝因子，从而在短的时间内搜索得更深。

置换表的结构如下，：

```
struct HashStruct
    {
        uint8_t nDepth, nFlag;   // 深度，标志位
        int16_t value;               // 分值
        uint16_t mv, xReserved;      // 最佳着法，内存填充
        uint64_t dwKey;              // Zobrist校验锁
    } Table[HASH_SIZE];
```

置换表信息夹在两个 Zobrist 校验锁中间，可以防止存取冲突。

置换表的操作主要为保存与提取，其主要参数如下，具体实现在源码中：

```
/**
     * @brief             即ProbeHash()
     *
     * @param pos         目前棋盘
     * @param vlAlpha     当前alpha
     * @param vlBeta      当前beta
     * @param nDepth      当前深度
     * @param mv      获取到的最优动作
     * @return int        获取到的分值，如果查不到或不符合则返回-MATE_VALUE
     */

    int Load(const PositionClass &pos, int vlAlpha, int vlBeta, int nDepth, int &mv,
bool bUseNull = false); // 提取置换表项
    /**
     * @brief             即RecordHash()
     *
     * @param pos         目前棋盘
     * @param nFlag   当前节点状态，参照parameters.h
     * @param vl      当前分值
     * @param nDepth  当前深度
     * @param mv      最优动作
     */
    bool Save(const PositionClass &pos, int nFlag, int vl, int nDepth, int mv); // 保存
置换表项
```

在提取置换表时，需要考虑是否为杀棋：如果是杀棋，那么不需要满足深度条件直接返回。

2.2 巨型开局库的处理

在网络中搜寻到的开局库均为.obk 格式，其为象棋软件兵河五四专用的开局库格式，需要使用 sqlite 数据库进行提取。在未加密的数据库中，我找到了一个大小约为 30M 的包含 600000+条数据的开局库，并且为了将其信息提取出来，编写了 sqlite 数据库程序，并且将 zobrist 值替换为兵河五四专用 zobrist 生成方法。开局库格式：

```
typedef struct book_entry_t book_entry_t ;
struct book_entry_t
{
    unsigned __int64 key ;
    unsigned __int16 move ;
    __int32 score ;
    __int32 win_count ;
    __int32 draw_count ;
    __int32 lost_count ;
    unsigned __int16 valid ;
    char comments[64];
};
```

提取方法：

```
int GetBookEntry(book_entry_t *entry,unsigned __int64 key)
{
    int result ;
    char *errmsg=NULL ;
    char **dbResult ;
    int nRow,nColumn ;
    int i,j ;
    int index ;
    char szSql[1024]={0};
    _snprintf(szSql,1024,"select * from bhobk where vkey=%I64u;",key);
    result=sqlite3_get_table(db,szSql,&dbResult,&nRow,&nColumn,&errmsg);
    if(SQLITE_OK==result)
    {
        index=nColumn ;
        for(i=0;i<nRow;i++)
        {
            if(i>127) break ;
            for(j=0;j<nColumn;j++)
            {
                if(j==2)entry[i].move=atoi(dbResult[index]);
                else if(j==3)entry[i].score=atoi(dbResult[index]);
                else if(j==4)entry[i].win_count=atoi(dbResult[index]);
                else if(j==5)entry[i].draw_count=atoi(dbResult[index]);
                else if(j==6)entry[i].lost_count=atoi(dbResult[index]);
                else if(j==7)entry[i].valid=atoi(dbResult[index]);
                else if(j==8)strncpy(entry[i].comments,dbResult[index],64);
                index++;
            }
        }
    }
    sqlite3_free_table(dbResult);

    return nRow>128?128:nRow ;
}
```

将读取到的结果写入 txt 文件中，提取 zobristKey、move 和 score 三列，再使用 python 生成为.h 文件。首先要将数据按 zobristKey 值进行排序，才能在搜索中使用二分法进行查找。二分查找时，共搜索两遍，第一遍搜索原局面，第二遍搜索局面的镜像，随后按 score 值对搜索到的着法进行排序：

```
int BookDataClass::GetBookMoves(const PositionClass& pos)
{
    BookStruct bk;
    ZobristStruct pos_zo = pos.zobr;
    int buffer_len = sizeof(bookBuffer) / sizeof(bookBuffer[0]);
    int nScan, nLow, nHigh, nPtr;
    int i, j, nMoves;
    nMoves = 0;
    for (nScan = 0; nScan < 2; nScan++) {
        nPtr = nLow = 0;
        nHigh = buffer_len - 1;
        while (nLow <= nHigh) {
            nPtr = (nLow + nHigh) / 2;
            bk = bookBuffer[nPtr];
            if (BOOK_POS_CMP(bk, pos_zo) < 0) {
                nLow = nPtr + 1;
            }
            else if (BOOK_POS_CMP(bk, pos_zo) > 0) {
                nHigh = nPtr - 1;
            }
            else {
                break;
            }
        }
        if (nLow > nHigh) {
            continue;
        }
        // 3. 如果找到局面，则向前查找第一个着法；
        for (nPtr--; nPtr >= 0; nPtr--) {
            bk = bookBuffer[nPtr];
            if (BOOK_POS_CMP(bk, pos_zo) < 0) {
                break;
            }
        }
        // 4. 向后依次读入属于该局面的每个着法；
        for (nPtr++; nPtr < buffer_len; nPtr++) {
            bk = bookBuffer[nPtr];
            if (BOOK_POS_CMP(bk, pos_zo) > 0) {
                break;
            }
            // 如果局面是第二趟搜索到的，则着法必须做镜像
```

装 订 线

```
                    bk.mv = MOVE(DST(bk.mv), SRC(bk.mv));          //兵河五四MOVE格式转码
                    bk.mv = nScan == 0 ? bk.mv : MOVE_MIRROR(bk.mv);
                    if (pos.LegalMove(bk.mv)) {
                        Books[nMoves].mv = bk.mv;
                        Books[nMoves].value = bk.value;
                        nMoves++;
                        if (nMoves == MAX_GEN_MOVES) {
                            break;
                        }
                    }
                }
                // 原局面和镜像局面各搜索一趟
                if(nScan == 0) pos_zo = pos.MirrorZobrist();
            }
            if (nMoves == 0)return 0;
            // 5. 对着法按分值排序
            for (i = 0; i < nMoves - 1; i++) {
                for (j = nMoves - 1; j > i; j--) {
                    if (Books[j - 1].value < Books[j].value) {
                        std::swap(Books[j - 1], Books[j]);
                    }
                }
            }
            return nMoves;
        }
```

将.txt 文件读取，排序并转换为.h 文件的 python 脚本如下：

```
with open('result.txt','r') as f:
    openbookdata=[]
    for i in range(0,600669):
        temp=[x[0:2] for x in f.readline().split(' ')]

    openbookdata.sort(key=lambda x:x[0])

with open('book_data.h','a') as f:
    for x in openbookdata:
        f.write("{uint64_t({}), uint16_t({}), int16_t({})},".format(x[0
],x[1],x[2]))
```

最终，以兵河五四格式保存的数量为 600000+的.h 文件如下：

bookdata_bigbook.h - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
#include "bookdata.h"
#ifndef BOOKDATA_BIGBOOK
#define BOOKDATA_BIGBOOK

const BookDataClass::BookStruct bookBuffer[600669] = {{uint64_t(64238001840133), uint16_t(511
(90829569152165), uint16_t(13875), int16_t(8)},{uint64_t(96950072875054), uint16_t(23178), int16_t(
uint16_t(34691), int16_t(8)},{uint64_t(167607786129016), uint16_t(14986), int16_t(8)},{uint64_t(21960
(8)},{uint64_t(278489460934183), uint16_t(17221), int16_t(8)},{uint64_t(328681034271713), uint16_t(4
(420405735783222), uint16_t(50533), int16_t(8)},{uint64_t(468261935295700), uint16_t(42644), int16_
uint16_t(17734), int16_t(8)},{uint64_t(610166002165975), uint16_t(47014), int16_t(6)},{uint64_t(651906
(8)},{uint64_t(747158307444279), uint16_t(17747), int16_t(8)},{uint64_t(755495778278782), uint16_t(4
```

保存开局库的 bookdata_bigbook.h 文件

2.3 实用工具与 ZOBRIST 键值

实用工具包括与时间有关的函数与在处理 FEN 串时用到的字符串处理工具，其定义如下：

```
inline int64_t GetTime()
{
    timeb tb;
    ftime(&tb);
    return (int64_t)tb.time * 1000 + tb.millitm;
}

void Idle();

char* strcasestr(const char* sz1, const char* sz2);
int strncasecmp(const char* sz1, const char* sz2, size_t n);
void StrCutCrLf(char* sz);

bool StrEqv(const char* temp, const char* comp);
bool StrEqvSkip(char*& temp, const char* comp);
bool StrScanSkip(char*& temp, const char* comp);
int StrTime(const char* temp, int nMin, int nMax);

bool StrScan(const char* sz1, const char* sz2);
bool StrSplitSkip(const char*& szSrc, int nSeparator, char* szDst = NULL);
bool StrSplitSkip(char*& szSrc, int nSeparator, char* szDst = NULL);
```

国际象棋局面包含了棋盘上的棋子、哪一方走棋、是否能易位、是否能吃过路兵等信息。在写国际象棋的程序时，需要比较两个局面看它们是否相同。如果比较每个棋子的位置，每秒种需要做成千上万次比较，这样会使比较操作变成瓶颈。另外，需要比较的局面数量多得惊人，要存储每个棋子的位置，需要占用非常大的空间。Zobrist 键值是一个常用的建立标签的方法。

由于使用的开局库为兵河五四格式，因此我们在兵河五四的技术文档中找到了其局面 hash 的方法。如图所示，为兵河五四与旋风象棋所使用的局面表示初始数组：

局面HASHKEY计算.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
//兵河、旋风采用初始数组
const unsigned __int64 ZobristPlayer = 0xA0CE2AF90C452F58;

const unsigned __int64 ZobristTable[14][256] =
{
0x7130B4BD8F138026, 0x451A5F875056AE15, 0x9023211DB6E216DD, 0x8176B9AB5178BE31, 0x9CA65525FE49D1F0,
0x64BFA2303868852B, 0xB91360D20127098E, 0x36150C526E1DD2F5, 0xD8BBFB8FCD6F735D, 0xA3B35E61D7DA5068,
0x46982BFDF96E1DCD, 0x012367021814AD81, 0x4089DA4C1B9C5662, 0x42BBDB342F88A0D4, 0xC827E492D9CEB711,
0x4E18FABBC55DBE46, 0xFB7A0B8AC6284121, 0xF179FC42457A105A, 0xF5A6CFA529C8D035, 0xED7626E641383689,
0x3B8A9E8457E65E67, 0x674957C0D8CE647F, 0x7C4CE37B578F1A9B, 0x07F9D32932E1493B, 0xAC776EB186FCC474,
0x3FE8BF61B3D17258, 0x04B55A2FB8F42D93, 0xC0DD14998777FF0F, 0x6E4E1E15DB62B600, 0x846AE109AE2C37B7,
0x7093E9E05DF029FB, 0xD3982B39A0FDB1C7, 0xBDE6DB97B7F5A17C, 0xC0EED455913ADDA6, 0x46947371FECE7628,
0x81AEF4C1CC7A02CF, 0xC5C82E68E036A296, 0x7FA039EF30781817, 0xAAE0E8DEEFF3F606, 0x1D0521C1E13B610E,
```

局面 hashkey 的计算

3. 总结与反思

3.1 不足

在仔细分析结果后，我们认为代码主要存在的问题有如下几点：

1. NNUE 效果未达到预期，极少数情况下 NNUE 版本的评估函数会出现极大的偏差，导致走出一步差棋，整个局面出现极大劣势，导致最终失败。
2. 内容有提高空间，可以提取更多局面，增加开局库的内容。在实际比赛中，发现有些开局着法并没有收录，造成一定开局劣势。

3.2 设想中的改进

1. 在获得更多数据的基础上对 NNUE 网络进行更多轮次的训练。由于硬件设备限制导致算力不足，在最终棋盘局面越来越复杂的情况下无法达到收敛。这一缺陷可能在高算力环境下得到改善。
2. 将 NNUE 的估值方法与传统的估值方法结合起来，在神经网络得到数据明显超出正常范围时，将最终值的权重转移至传统估值函数上来。

3.3 收获与反思

通过近半个学期的学习与开发，我个人首先获得的是一个充满挑战的项目的合作经验与开发经验。通过与其他三位组员的合作，培养了个人在团队中的沟通能力。再者，是在人工智能，尤其是对抗性搜索这一领域知识的掌握。通过学习各种搜索与决策算法，深入了解了棋类引擎的工作原理与各种先进方法。最后是在优化与调试过程中的不断尝试，锻炼了个人的代码能力，培养了创新的方法。

2021 年 6 月 12 日

装 订 线