实验报告

－－－－alpha-beta 剪枝

姓名：何坤宁　　学号：16340073　　日期：2019.1.11

摘要：

中国象棋发展至今已经有了几千年的历史，是中华民族灿烂的文化瑰宝，它具有浓厚的趣味性，规则简单明了，在中国已经成为了一项普遍的棋类运动。本文通过 QT 开发工具，运用 alpha-beta 剪枝算法，实现了一个可以人机对弈的中国象棋博弈程序。

1．导言

象棋程序的实现可以被分为人工智能和界面程序辅助两大部分。人工智能部分主要体现计算机的下棋思路，既计算机如何进行思考并以最佳走法完成下一步，先由相应的搜索算法进行搜索，并对各种可能的走法进行估值，从中选择胜利面最大的一步；而界面及程序辅助部分主要便于用户通过以前的下棋步骤，更好地调整下棋思路，着法显示使用户能够清楚地知道下棋过程，更准确地把握整个局面。

对于界面程序部分，我们参考了 CSDN 逆风微光博主的程序，基于 QT 开发工具，实现了棋盘、棋子的显示，点击棋子的闪烁效果以及胜利与失败的结果显示。

对于人工智能部分，使用极小极大搜索，在有限的搜索深度范围内选择一步相对好的走法，并使用 Alpha-Beta 剪枝，裁剪搜索树中没有意义的不需要搜索的树枝，以提高运算速度。

极大极小搜索策略是考虑双方对弈若干步之后，从可能的步中选一步相对好的走法来走，在有限的搜索范围内进行求解。

为此要定义一个静态估计函数 f，以便对棋局的势态做出优劣的估计，这个函数可根据棋局的优劣势态的特征来定义。这里规定，MAX 代表程序方，MIN 代表对手方，P 代表一个棋局（即一个状态)。有利于 MAX 的势态，f(p)取正值，有利于 MIN 的势态，f(p)去负值，势态均衡，f(p)取零。极大极小搜索的基本思想是：

（1）当轮到 MIN 走步的节点时，MAX 应考虑最坏的情况（因此，f(p)取极小值)。

（2）当轮到 MAX 走步的节点时，MAX 应考虑最好的情况（因此，f(p)取极大值)。

（3）当评价往回倒退的时候，相应于两位棋手的对抗策略，不同层上交替地使用（1)、(2）两种方法向上传递倒推值。

所以这种搜索方法称为极大极小过程。实际上，这种算法是假定在模拟过程中双方都走出最好的一步，对 MAX 方来说，MIN 方的最好一步是最坏的情况，MAX 在不断地最大化自己的利益。

极大极小搜索策略在一些棋盘 AI 中非常常见，但是它有个致命的弱点，就是非常暴力地搜索导致效率不高，特别是当讲搜索的深度加大时会有明显的延迟，alpha-beta 在此基础上进行了优化。事实上，MIN、MAX 不断的倒推过程中是存在着联系的，当它们满足某种关系时后续的搜索是多余的！alpha-beta 剪枝算法把生成后继和倒推值估计结合起来，及时减掉一些无用分支，以此来提高算法的效率。

定义极大层的下界为 alpha，极小层的上界为 beta，alpha-beta 剪枝规则描述如下：

（1）alpha 剪枝。若任一极小值层结点的 beta 值不大于它任一前驱极大值层结点的 alpha

值，即 alpha(前驱层) >= beta(后继层)，则可终止该极小值层中这个 MIN 结点以下的搜索过程。这个 MIN 结点最终的倒推值就确定为这个 beta 值。

（2）beta 剪枝。若任一极大值层结点的 alpha 值不小于它任一前驱极小值层结点的 beta 值，即 alpha(后继层) >= beta(前驱层)，则可以终止该极大值层中这个 MAX 结点以下的搜索过程，这个 MAX 结点最终倒推值就确定为这个 alpha 值。

如果说搜索算法说程序的心脏，那么局面评估就是程序的大脑。局面评估的作用是通过象棋知识对当前的棋局好坏进行评价，并将棋局局面进行量化，得到一个估值。在本程序中，主要考虑了一下两个因素：

（1）子力总和

子力是指某一棋子本身所具有的价值。通俗地讲就是一个棋子它值个什么价。例如，车值 500 的话，那可能马值 300，卒值 80 等等。所以在评估局面时，我们首先要考虑双方的子力总和的对比。比如红方拥有士象全加车马炮，而黑方只有残士象加双马，则红方明显占优。

（2）棋子位置（控制区域）

棋子位置（或称控制区域）是指某一方的棋子在棋盘上所占据（控制）的位置。例如，沉底炮、过河卒、以及车占士角等都是较好的棋子位置状态，而窝心马等则是较差的棋子位置状态。

2．实验过程

1.设置界面大小、图像，启动定时器

```
113  MainWindow::MainWindow(QWidget *parent) :
114      QMainWindow(parent),
115      ui(new Ui::MainWindow)  {...}
137
138  MainWindow::~MainWindow()  {...}
142
143  void MainWindow::paintEvent( QPaintEvent * )  {...}
152
153  void MainWindow::DrawItem(QPainter& painter, Item item )  {...}
```

2. 初始化所有棋子，并添加棋子到 m_items

```
175      //初始化所有黑方棋子
176      Item item1(ITEM_JU,COLOR_BLACK,QPoint(0,0));
177      Item item2(ITEM_MA,COLOR_BLACK,QPoint(1,0));
178      Item item3(ITEM_XIANG,COLOR_BLACK,QPoint(2,0));
179      Item item4(ITEM_SHI,COLOR_BLACK,QPoint(3,0));
180      Item item5(ITEM_SHUAI,COLOR_BLACK,QPoint(4,0));
181      Item item6(ITEM_SHI,COLOR_BLACK,QPoint(5,0));
182      Item item7(ITEM_XIANG,COLOR_BLACK,QPoint(6,0));
183      Item item8(ITEM_MA,COLOR_BLACK,QPoint(7,0));
184      Item item9(ITEM_JU,COLOR_BLACK,QPoint(8,0));
185      Item item10(ITEM_PAO,COLOR_BLACK,QPoint(1,2));
186      Item item11(ITEM_PAO,COLOR_BLACK,QPoint(7,2));
187      Item item12(ITEM_BING,COLOR_BLACK,QPoint(0,3));
188      Item item13(ITEM_BING,COLOR_BLACK,QPoint(2,3));
189      Item item14(ITEM_BING,COLOR_BLACK,QPoint(4,3));
190      Item item15(ITEM_BING,COLOR_BLACK,QPoint(6,3));
191      Item item16(ITEM_BING,COLOR_BLACK,QPoint(8,3));
```

```
211         m_items.push_back(item1);
212         m_items.push_back(item2);
213         m_items.push_back(item3);
214         m_items.push_back(item4);
215         m_items.push_back(item5);
216         m_items.push_back(item6);
217         m_items.push_back(item7);
218         m_items.push_back(item8);
219         m_items.push_back(item9);
220         m_items.push_back(item10);
221         m_items.push_back(item11);
222         m_items.push_back(item12);
223         m_items.push_back(item13);
224         m_items.push_back(item14);
225         m_items.push_back(item15);
226         m_items.push_back(item16);
```

3.轮到人走棋时，判断点击事件。如果这次点击之前没有选中棋子，那么如果点中了当前走棋方的颜色的棋子，就选中了棋子

```
313     else
314     {
315         //当前没有选中棋子
316         Item ClickedItem;
317         if (FindItemAtPoint(pt,ClickedItem))
318         {
319             //如果点中一个棋子,是当前走棋方的颜色，就选中了
320             if ( (m_bIsRedTurn && ClickedItem.m_color == COLOR_RED) ||
321                 (!m_bIsRedTurn && ClickedItem.m_color == COLOR_BLACK))
322             {
323                 m_SelectedItem = ClickedItem;
324                 m_bExistSelectedItem = true;
325                 return;
326             }
327         }
328     }
329 }
```

4.如果点击之前已经选中了棋子，那么如果鼠标点击的就是当前选中的棋子，就什么也不做

```
258         //是否有选中的棋子
259         if(m_bExistSelectedItem)
260         {
261             //已存在棋子，判断鼠标点击的是否是选中棋子
262             if (pt == m_SelectedItem.m_pt)
263             {
264                 //再次点击已经选择的棋子，什么也不做
265                 return;
266             }
```

5. 如果点击之前已经选中了棋子，那么点击同色的其他棋子时，改选

```
268         //点击其它棋子(同色)
269         Item ClickedItem;
270         if (FindItemAtPoint(pt,ClickedItem))
271         {
272             //点击的同色的另外一个棋子，改选
273             if ( (m_bIsRedTurn && ClickedItem.m_color == COLOR_RED) ||
274                  (!m_bIsRedTurn && ClickedItem.m_color != COLOR_RED))
275             {
276                 SetItemShow(m_SelectedItem,true);
277                 m_SelectedItem = ClickedItem;
278                 return;
279             }
280         }
```

6. 如果点击之前已经选中了棋子，且没有点击同色棋子，则首先获取已选择棋子的可移动区域，如果点击的区域可达，则进行下一步判断

```
282             //获取已选择棋子的可移动区域
283             QVector<QPoint> moveArea;
284             GetMoveArea(m_SelectedItem,moveArea);
285             if (moveArea.contains(pt))
```

7.删除点击位置的棋子，然后移动选中棋子。如果帅被删除，则游戏结束。

```
288             bool bDeleteSHUAI = false;
289             DeleteItemAtPoint(pt,bDeleteSHUAI);
290             //移动棋子
291             ChangeItemPoint(m_SelectedItem.m_pt,pt);
292             //游戏结束
293             if (bDeleteSHUAI)
294             {
295                 QString str = m_bIsRedTurn?QStringLiteral("红方胜利!     "):QStringLiteral("黑方胜利!       ");
296                 QMessageBox::information(NULL,  "GAME OVER  ",str, QMessageBox::Yes , QMessageBox::Yes);
297                 NewGame();
298                 return ;
299             }
300             m_bExistSelectedItem = false;
301             m_bIsRedTurn = !m_bIsRedTurn;
302             calc_bIsRedTurn = false;
303             update();
304             //QTimer::singleShot(100, this, SLOT(computerMove()));
305             computerMove();
```

8.轮到机器走时，首先通过 alpha-beta 剪枝算法搜索最好的一步

```
452         Step * step = getBestMove();
```

9.getBestMove()函数：设置下界（即 alpha）为一个很小的数，获得当前所有可能的走法。

```
464         QVector<Step*> steps1;
465
466         int maxInAllMinScore = -1000000;
467         Step * ret = NULL;
468         getAllPossibleMove(steps1);
```

10.getBestMove()函数：由于假设下一步人也会选择最优的走法，即人会选择使机器分数最低的走法，因此，对于每种可行的走法，其分数是所有可行的下一步走法中的最小值

```
475             fakeMove(step);
476             calc_bIsRedTurn = !calc_bIsRedTurn;
477             int minScore = getMinScore(_level-1, maxInAllMinScore);
478             calc_bIsRedTurn = !calc_bIsRedTurn;
479             unfakeMove(step);
```

11.getBestMove()函数：对于每种可行走法，如果其分数 minScoure 大于下界，那么下界分数提升为 minScore

```
481             //选择极大值
482             if (minScore > maxInAllMinScore)
483             {
484                 maxInAllMinScore = minScore;
485                 if (ret) {
486                     ret = NULL;
487                 }
488                 ret = step;
489             }
490             else
491             {
492                 delete step;
493                 step = NULL;
494             }
```

12. getMinScore(int level, int curMinScore)函数：首先获取当前分支所有可行的走法，并将上界（即 beta）设为一个很大的值

```
557         QVector<Step*> steps1;
558         getAllPossibleMove(steps1);
559         int minInAllMaxScore = 1000000;
```

13. getMinScore(int level, int curMinScore)函数：由于现在是假设人在走，人要选择最优的走法，就要使分数最小，而人的每种走法的分数是由下一步走法决定的，机器会令每种走法的分数最大，因此，对于每一种可行的走法，其分数是所有下一步可行走法中分数最大者

```
566             fakeMove(step);
567             calc_bIsRedTurn = !calc_bIsRedTurn;
568             int maxScore = getMaxScore(level - 1, minInAllMaxScore);
569             calc_bIsRedTurn = !calc_bIsRedTurn;
570             unfakeMove(step);
```

15. getMinScore(int level, int curMinScore)函数：如果得到的分数小于等于下界，则剪枝，因为这表示有走法比这种走法分数更高，机器绝不会选择这种走法

```
575         if (maxScore <= curMinScore)
576         {
577             while(steps1.count())
578             {
579                 Step* step = steps1.last();
580                 steps1.removeLast();
581                 step = NULL;
582             }
583             return maxScore;
584         }
```

16. getMinScore(int level, int curMinScore)函数：否则，如果得到的分数 maxScore 小于上界，则将上界改为 maxScore

```
586         if(maxScore < minInAllMaxScore)
587         {
588             minInAllMaxScore = maxScore;
589         }
```

17. getMinScore(int level, int curMinScore)函数：每一种走法都考虑完了，则返回上界

```
591     return minInAllMaxScore;
```

18. getMaxScore(int level, int curMaxScore)函数：与 getMinScore 类似，首先获得所有可行的走法，然后对于每一种走法，根据 getMinScore 计算分数，如果得到的 minScore 大于等于上界，则剪枝，因为如果不剪枝，那么这一步的分数就会比别的分数大，而人会选择分数最小的，这一步的计算就是多余的。否则，如果 minScore 大于下界，则提升下界至 minScore，这表示至少能得 minScore 分。若计算完毕，则返回下界。

19.总之，对于每一种走法都设置一个上界和下界，getMinScore 通过调整上界（减小上界），计算最多能得多少分，但是分数不能低于上头传来的下界，因为这个下界代表上头至少能获得的分数，这一步要是分数太低了，上头就不会选这一步了。而 getMaxScore 通过调整下界（提升下界），计算至少能得多少分，但是当计算出的分数大于上头给的上界时，这一步

就被剪枝了。

20.评估函数，首先计算棋子固定的棋力

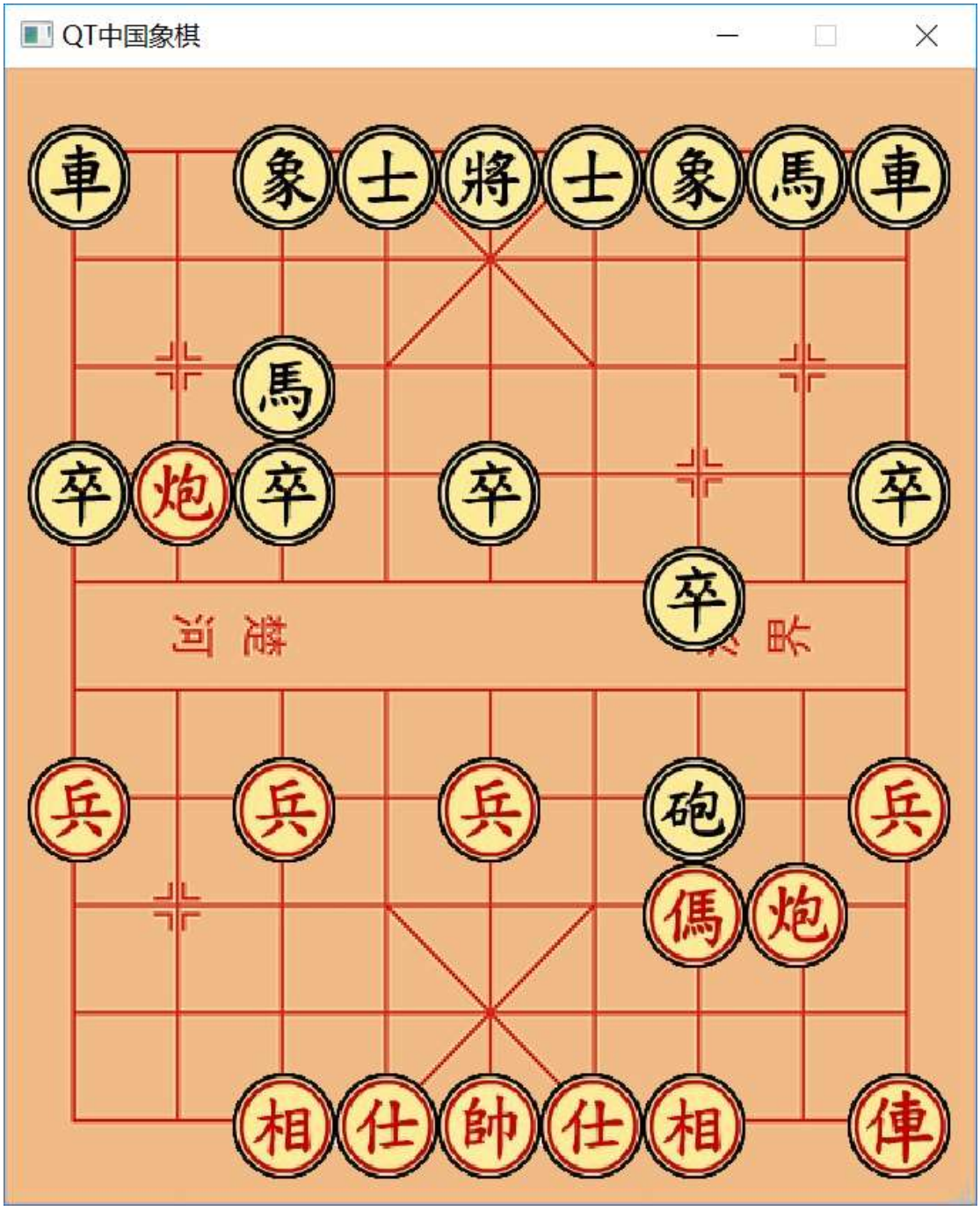
```
651     static int chessScore[] = {10000, 150, 150, 700, 320, 300, 100};
652
653     for (int i = 0; i<m_items.size(); i++)
654     {
655
656         if (m_items[i].m_color == COLOR_RED)
657         {
658             redTotScore += chessScore[m_items[i].m_type];
700         else if (m_items[i].m_color == COLOR_BLACK)
701         {
702             blackTotScore += chessScore[m_items[i].m_type];
```

21.然后计算棋子在特定位置的附加棋力

```
659         int position = m_items[i].m_pt.x() * 10 + m_items[i].m_pt.y();
660         switch (m_items[i].m_type)
661         {
662         case ITEM_JU:
663             {
664                 redTotScore += PositionValues[5][position];
665                 break;
666             }
667         case ITEM_MA:
668             {
669                 redTotScore += PositionValues[4][position];
670                 break;
671             }
```

3．结果分析

可以直接点击 exe 运行



最大搜索深度设了 4 层，速度挺快的，基本没有卡顿。棋力一般吧，由于我不会象棋，除了知道走棋规则，其他一窍不通，所以觉得他还挺厉害的。。。

算法基本满足实验要求，但是能改进的地方也很多。首先就是代码可以再精简，有一些代码冗余。

其次是评估函数，这个还是挺难的，需要学习很多知识，目前我知道的就是，可以考虑棋子的机动性，棋子的机动性指棋子的灵活度（可移动性）。例如，起始位置的车机动性较差，所以我们下棋讲究早出车。同样四面被憋马腿的死马机动性也较差，还有棋子的相互关系，这一点的分析较为复杂，因为一个棋子与不同的子之间往往存在多重关系。如：一个马可能在对方的炮的攻击之下同时它又攻击着对方的车。

再者，可以添加悔棋、提示等功能，还有背景音乐等等，增加娱乐性，由于是第一次用 QT，还不太会发挥，所以没能增加太多功能。

4．结论

本文运用 alpha-beta 剪枝算法，实现了一个具有一定棋力的中国象棋博弈程序。刚开始无从下手，但是通过查阅资料，看懂其中的原理，渐渐地有了思路，最后完成了这个程序，在实现过程中，我受益颇多，不仅熟悉了 alpha-beta 剪枝算法，也对中国象棋有了进一步的了解，感受到了使程序具有“智能”的乐趣。

主要参考文献(三五个即可)

中国象棋人机博弈系统的设计与实现

https://wenku.baidu.com/view/bd713817ba0d4a7303763a7b.html

[人工智能] alpha-beta 剪枝算法及实践

https://blog.csdn.net/qq_31615919/article/details/79681063

中国象棋软件-引擎实现（六）局面评估

https://blog.csdn.net/laoxing643/article/details/79340612

QT 项目三：中国象棋

https://blog.csdn.net/dpsying/article/details/53771589

JavaScript 中国象棋程序（4） - 极大极小搜索算法

https://www.cnblogs.com/royhoo/p/6425658.html