**AI课程实践报告**

TinyOthello

### 0. 简介

TinyOthello程序实现了5个稍有区别的AI Player. 其中，AIPlayer1实现了基本的alpha-beta裁减搜索算法，AIPlayer2引入了迭代深化（Iterative Deepening），AIPlayer3利用了置换表（Transposition Table）来作为缓存，AIPlayer4在此基础上使用MTD(f)算法提高裁减效率，AIPlayer5则尝试使用PVS算法提高裁减效率。静态评估函数使用了几个盘面特征，如移动力（mobility）、稳定子（stables）等的线性组合，组合系数凭借经验给出。

本程序还有诸多可以改进之处，由于时间有限，只得在此作罢。

### 1 搜索算法

**1.1 AIPlayer1 (alpha-beta)**

本程序最基础的算法是alpha-beta裁减搜索算法，它的伪代码如下：

int alpha-beta(board, depth, alpha, beta) {

if (depth == 0 || game end)

return static-evaluate(board);

score = -infinity;

for each valid move p in board {

board.move(p);

value = -alpha-beta(board, depth-1, -beta, -alpha);

board.unmove(p);

if (value > score) {

score = value;

if (score > alpha) alpha = score;

if (score >= beta) break;

}

}

return score;

}

值得注意的是上面的伪代码给出的alpha-beta函数在该分支被裁减（即最终score没有落于alpha和beta之间）时，将保留搜索的成果（被裁减时的score）将其返回，因此被称为fail-soft alpha-beta算法。这一点在对于后面加入置换表和套用MTD(f)算法至关重要。

利用上述算法即可得AIPlayer1, 它调用result = alpha-beta(board, depth, -infinity, infinity)并在调用的第一层记录最佳走法即可。利用这个AI在深度5已经可以超过我的Othello水平（看出来我水平臭……）。

**1.2 AIPlayer2 (alpha-beta + iterative-deepening)**

AIPlayer2的改进在于采用了iterative-deepening. 通过控制访问的结点数而不是深度，可以起到保持运算时间不变而动态调整深度的效果，从而在尾盘的时候不至于也只计算和中盘一样的深度。为了更精确地控制时间，我让程序访问的结点数一旦到达某个阀值就立刻停止搜索，但这样也会导致最后一小段计算被浪费。

**1.3 AIPlayer3 (alpha-beta + iterative-deepening + TT)**

AIPlayer3的改进在于在AIPlayer2的基础上增加了置换表（Transposition Table，以下简称TT），它可以缓存以前搜索的结果。这在于iterative-deepening结合时效果明显：后面的迭代可以利用前面迭代的结果，并且选择前面迭代的最优走法先走，以期望更多的剪枝（当第一次尝试的就是最优路径时剪枝最多）。值得注意的是，由于采用了fail-soft alpha-beta搜索，因此在TT内存储的值即使无法直接用来剪枝，也可以用来缩小alpha-beta的范围。这样，就需要解释一下fail-soft alpha-beta返回结果的意义了：

设alpha-beta(board, depth, -infinity, infinity)作为正确值accurate, 则：

如果score <= alpha < beta, 则score >= accurate

如果alpha < score < beta, 则score == accurate

如果alpha < beta <= score, 则score <= accurate

为了验证程序的正确性，我在TranspositionTable类中增加过上述断言，不过正式运行的时候还是屏蔽掉了，因为实在太慢了。

有了上述条件，我们就可以利用TT中的信息了，伪代码如下：

void tt-save(board, value, alpha, beta, depth) {

if (value <= alpha)

bound = UPPER;

else if (value >= beta)

bound = LOWER;

else bound = ACCURATE;

add-to-tt(board, value, bound, depth);

}

int alpha-beta(board, depth, alpha, beta) {

entry = tt-lookup(board);

if (entry != null && entry.depth >= depth) {

if (entry.bound == LOWER)

alpha = max(alpha, entry.score);

if (entry.bound == UPPER)

beta= min(beta, entry.score);

if (entry.bound == ACCURATE)

alpha = beta = entry.score;

if (alpha >= beta) // TT causes a cutoff

return entry.score;

if (depth == 0 || game end)

return static-evaluate(board);

score = -infinity;

alpha0 = alpha; // caution!

for each valid move p in board {

board.move(p);

value = -alpha-beta(board, depth-1, -beta, -alpha);

board.unmove(p);

if (value > score) {

score = value;

if (score > alpha) alpha = score;

if (score >= beta) break;

}

}

tt-save(board, score, alpha0, beta, depth);

return score;

}

值得注意的是最后存入TT用的alpha是调整前的alpha, 即alpha0. 充分理解这个算法后一切都很自然，但我查阅的资料却偏偏没写清楚，而我当时也没有很好地理解（可能现在也谈不上很好地理解吧），导致了很长时间的调试。

**1.4 AIPlayer4 (MTD(f) + iterative-deepening + TT)**

AIPlayer4的主要改进在于将初始调用result = alpha-beta(board, depth, -infinity, infinity)改为调用MTD(f)算法：

int MTD(f, board, depth) {

g = f;

lower-bound = -infinity;

upper-bound = infinity;

while (lower-bound < upper-bound) {

beta = (g == lower-bound ? g + 1 : g);

g = alpha-beta(board, depth, beta – 1, beta);

if (g < beta) upper-bound = g;

else lower-bound = g;

}

return g;

}

这个算法的主要原理在于使用极小化窗口[beta – 1, beta]的剪枝效率很高，而返回值可以按照上面关于fail-soft alpha-beta的解释来定位精确值。

它的中心思想就是迭代求解方程g’ = alpha-beta(g, g + 1), 但简单的loop until g == g’的代码却会由于它和TT之间的复杂作用会导致这个过程陷入振荡而不收敛的情况，而上述代码就是为了避免这种情况而写出的。

在iterative-deepening的环境下，f通常由上一个深度的返回值给出，即按照如下调用：f = MTD(f, board, depth) 这样当f比较稳定时可以进一步提高效率。为了让f稳定，我让搜索深度每次加2.

在实际操作中，我发现MTD(f)时常会在TT中原有的深度较大数据的作用下在lower-bound > upper-bound的状态下结束（而不是等于）。通常这不成问题，因为随着iterative-deepening的深入，考虑到Othello游戏局面的单调性，最终这些数据的影响会消失，而先前的计算引导下的move-ordering则继续发挥作用，使MTD(f)最终能高效地收敛于正确的值上。但在尾局阶段，当错误的返回值处于表示棋局结束的极值范围内时iterative-deepening将终止，从而将错误结果输出在棋盘上。这让我的程序输掉了与一个我同学的简单AI的2盘棋，并且是在尾盘出现了显然的失招。这也是我发现这个问题的起因。

在最终的版本中，我尝试修正了这个问题。方法是记录下MTD(f)返回时的状态，如果是以失败方式退出的，则仅仅增加深度而不记录结果。这样错误的结果就不会使得迭代提前结束。考虑到可能在输出任何正确结果前，给定的访问结点数会用完（这种情况目前没有发生过，只是防御性地写了下来，因为我无法保证它不发生），本程序会在这种情况下转而使用AIPlayer3来计算结果。

这个错误花了不少时间调试，并且在我查阅的资料中都没有看到（可能是我马虎了），因此解决方法似乎有点粗糙，也没有正确性保证（例如，上述TT影响必将消失的判断并没有严格的证明，只是直觉上挺对），不过实践中解决了我程序的问题。还是那句老话：时间有限，只能到此打住了。

**1.5 AIPlayer5 (PVS + iterative-deepening + TT)**

AIPlayer5则采用了主要变例搜索（Principal Variation Search, 以下简称PVS）算法来提高效率。它据称是最好的alpha-beta变体。它首先从TT中找到一个最好的走法（即使depth不足也采纳，这个优化在所有使用TT的AIPlayer中都存在），然后用这一手使用正常的窗口搜索，而后使用极小宽度窗口来对每个后续着法和第一个着法作比较。只有当零窗口搜索失败后才去做正常的搜索。代码如下（没有显示TT相关的代码）：

int pvs(board, depth, alpha, beta) {

if (depth == 0 || game end)

return static-evaluate(board);

m = best move from TT;

board.move(m);

score = -alpha-beta(board, depth – 1, -beta, -alpha);

board.unmove(m);

if (score >= beta) return score;

alpha0 = alpha;

for each valid move p in board {

board.move(p);

value = -alpha-beta(board, depth-1, -score - 1, -score);

board.unmove(p);

if (value > score) {

score = value;

if (score > alpha) alpha = score;

if (score >= beta) break;

if (score < beta) {

board.move(p);

value = -alpha-beta(board, depth-1, -beta, -score);

board.unmove(p);

}

if (value > score) {

score = value;

if (score > alpha) alpha = score;

if (score >= beta) break;

}

}

}

return score;

}

**1.6 测试与对比**

以上5个AI互相测试，其中各个AI的配置为：最大单层访问结点数均为20000, 而没有此项参数的Player1的max depth = 7. 由于偷懒的原因，这里没有将时间作为标准。所有AI使用相同的静态评估函数，得到如下结果：

AI1 win: 3, lose 4, draw 1 （注：AI1和自己的结果没有计入，

AI2 win: 1, lose 7, draw 0 故每个AI 8局）

AI3 win: 5, lose 1, draw 2

AI4 win: 5, lose 2, draw 1

AI5 win: 4, lose 4, draw 0

Result Table (表中数字为黑胜白多少子)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Black\White | AI1 | AI2 | AI3 | AI4 | AI5 |
| AI1 | 0 | 34 | 0 | -44 | 12 |
| AI2 | -6 | 20 | -2 | -2 | -6 |
| AI3 | 8 | 20 | 16 | -24 | 18 |
| AI4 | 26 | -2 | 0 | -4 | 20 |
| AI5 | 6 | 5 | -4 | 4 | 4 |

从上面的胜负统计可以看出胜率比较平均，但从下面的搜索总深度来看，AI4处于明显优势。这从一个侧面反映了静态评估函数的质量比较低，即搜索得更深得到长远考虑下更优的局面并没有让在胜率上有非常突出的反映。观察了一下棋谱，发现还是要靠尾盘大家走闲招来抢稳定子决胜，谁首先没有闲着谁就败了，靠多考虑几手棋是追不回来的，多少有些运气成分。不过AI4的胜率也不差，输给连战连败的AI2有点意外。另外，他和AI5战成一胜一负，总共有2负。同样是搜索效率优化后的算法，可AI5的表现相对于AI4来说就很不令人满意了，我想可能是编程中存在错误，也可能是它所作的move-ordering对于Othello游戏的效果不佳吧。从这些结果考虑，图形界面里的主AI将选用AI4.

最后，从数据中，我计算了一下P(p1 win | depth1 > depth2), 即深度大于对手时获胜的概率。其中p1未必执黑，所以是：

(count(black win when black depth > white depth) +

count(white win when white depth > black depth)) /

(count(black depth > white depth) +

count(white depth > black depth))

而这个概率恰好是0.5, 这非常令人失望。看来如果将来有心改进的话，重点应该在静态评估函数上。不过单就搜索效率来讲，AI4在同等条件下还是效果最好的。

这张图需要解释一下，它表示该搜索访问的有效结点数。对于AI1来说，这就是它实际访问到的结点数，但AI2到AI5则会在最后一次迭代深化中因单层结点数超过阀值而终止。这一层访问的结点没有被统计进来（本来是故意如此的，现在想想好似不太合理。如果把最后的结果统计进来的话，还可以基本反映运行时间。这个图AI3~AI5的结果可以反映它们搜索效率的对比，似乎没有其它更多信息了）。因此后3个AI的值应当越大越好，和AI1的值没有太多可比性。而AI2由于没有TT, 结果自然也比后三个大许多。而后三个的对比再次显示AI4搜索效率高且稳定。

（注：刚才提到完全统计可以基本反映运行时间的差距，是写的时候想到的，不过对这个比较好奇，到底哪个AI更快呢？由于跑一个上述配置的测试所需时间太长，跑了个精简版粗略看一下。最大单层访问结点数均为2000, AI1的max depth = 4. 结果很有趣，撇开AI1不管（depth太低），其他的里面AI2明显慢，AI4其次，但与AI3相差无几，AI5也基本在同一水平上）

### 2. 静态评估函数

下面简单介绍一下本程序所使用的几个静态特征函数：

**2.1 稳定子(stable)**

即不可能被吃掉的子。这里只计算了从角出发的稳定子，但没有计算嵌入在其他稳定子中的异色子。例如如下局面：

a b c d e f g h

1 x x x x x . x x

2 x x o x x x x o

3 x x x x o o o o

4 x x o x o . . .

5 o x x x x . . .

6 x o o x x x . o

7 x x x x o o o x

8 x x x x o . . o

其中，x为黑棋，o为白棋，.表示空格，则黑的稳定子有：

左上角：a1-e1, a2-b2, a3-b3, a4

左下角：a6, a7-c7, a8-d8

右上角：g1-h1

右下角：（无）

但对于白来说，除了h8外，事实上a5也是稳定子，因为它已经不可能被翻转了。但这个feature没有将其记入。

稳定子在TinyOthello中有着相当大的权重。尤其在对局初期任何一方得到稳定子都相当于取得了很大的优势。

**2.2 移动力(mobility)**

移动力就是一方在当前局面下可以走的位置的数量。一般可以认为移动力强的一方具有一定的优势，因为他有更多的选择，从而掌握了全局的主动。

**2.3 占边(edge)**

这个feature数出本方有多少个子在边上。虽然爬边策略具有一定的风险，但由于边上的子不容易被吃，所以对于在一段时间后移动力应该会有一定帮助，因此本程序中也给了一个正的权重。

**2.4 x位置(x-position)**

所谓的x位置是指离角只有1格的，并且处于对角线上的位置，包括b2, g2, b7, g7四个。当角上没有子的时候在这些位置落子一般会给给对方进角的机会，因此这个这个个数越大本方越不利（这个负权重在feature中已处理，最后的系数依旧是正的）。

**2.5 c位置(c-position)**

所谓的c位置包括以下8格：a2, a7, b1, b8, g1, g8, h2, h7. 类似于x位置，当角上没有子时他们也会给对手进角造成机会，例如，假设边a全部空闲，则黑a2后对手a5. 此时如果让对手取得a3则必然失角了。实际上，走c位也并不一定都是坏棋，有时候为了避开c位而舍弃边上的子力会是一手更恶的棋，因此本程序c位的权重相对x位要小一些。

**2.6 分值(score)**

所谓分值，就是盘面上黑白双方的子力差距。这个乍一看很有用的feature却在实战中几乎没有任何作用。最著名的例子就是在最后还有4个空的时候，黑子只有1子却还能获胜：

a b c d e f g h

1 . o o o o o o .

2 o o o o o o o o

3 o o o o o o o o

4 o o o x o o o o

5 o o o o o o o o

6 o o o o o o o o

7 o o o o o o o o

8 . o o o o o o .

这个例子中，剩下的4步都由黑棋下，因为白棋只能不断pass, 最终结果是黑棋40-24胜。可见盘面中所谓的大食策略并无作用。本程序中虽然有这个feature, 但只象征性地给了一定的权重。

**2.7 尾盘估计(end-of-game evaluation)**

尾盘估计也就是评判谁胜谁负，自然不在话下，两边子力一减即可判断，然后返回相应的极值。为了在最后能赢得更彻底一点，我把这个feature在极值的基础上更具对方剩余子力做出调整，使得同样是获胜，赢得最多的一手将有更高的评分。为了方便iterative deepening的终止条件的判断，和局也被评判在离极值较近的地方。这里返回离负极值近的一个值，表示并不能满足于平局 :p 不过实际上，由于评判的时候结点的奇偶关系，在若干次取负后也可能变成极大值，算是一个缺陷吧。

**2.8 结合上面的features**

除了尾盘估计外，其他features通过线性组合的方式，共同组成对盘面的静态评估函数(static evaluation). 系数通过经验判断。另外，也是根据经验，系数在盘面的不同阶段会改变。例如在尾盘占据x和c位并不是什么坏事，需要依靠动态计算来判断能否占据，因此在后期x和c的权重都会降为0. 那么这么做是否会导致和TT合作时出现问题呢？粗粗得思考了一下似乎不会（因为使用TT中同等深度的缓存值的话会使用相同的系数，而使用更深的值时静态值的不稳定性本来就有），而测试也似乎没什么问题，于是就照办了。

**2.9 不足**

相比前面的搜索算法，这里的静态评估函数就要粗糙许多了：

首先，盘面特征函数（feature）较少，只有6个比较简单的函数。

其次，特征函数的组合系数凭借经验而不是通过机器学习等自动化方法给出。没有太多的科学性。如果经验是错误的，那么最终对静态函数可能会有反效果。（不过机器学习方法给出的结果究竟效果如何呢？只有做了以后对比测试了才知道。说不定还是我经验丰富呢 呵呵）

再者，实现方法粗糙，效率较低。例如，最后的mobility通过依次察看每个点是否是valid move来实现，效率非常低。

另外，例如bit board, Zobrisk hashing等优化都能使局面评估，乃至TT的效率大幅提高。

不过这些函数已经足以应付很多人类玩家了，加之时间有限，也就只得如此了。

### 3. 程序的结构

本程序使用C# 2.0在Visual Studio 2005下完成，包括2个project:

第一个工程TinyOthello提供了上面提到的全部功能，包括一个Board类, 5个AIPlayer类以及一个简单的游戏执行框架和ConsoleUI以及一个测试框架AITest.

另一个工程TinyOthelloGUI为第一个提供了Graphic User Interface, 主要是使用了第一个里面的类。由于制作GUI并不擅长，因此虽然代码比较精简，但GUI的显示速度并不理想、界面也并不好看。但基本功能已经齐全，提供了人机、人人、机机的对弈功能，但计算机AI等级调整的界面没来得及做。

**3.1 TinyOthello**

这个工程里主要实现了TinyOthello.Kernel namespace里的类和接口，以及TinyOthello.ConsoleUI namespace里的2个类，界面简单，主要用于前期AI的开发和测试。主要的包括：

**3.1.1 TinyOthello.Kernel namespace**

Board, 棋盘类，处理了行棋规则和棋子的翻转，检查是否是合法走法，以及棋局历史的记录，例如Undo, Redo之类的函数。

IPlayer, 玩家接口，一个玩家所必需的函数，即PlayOneMove()

IAIPlayer, 提供了取得AI状态的额外接口函数

IBoardViewer, 棋局观察器接口，用来显示棋盘，即ShowBoard()

Game, 结合以上各个接口，调度整个棋局的进行，内容非常简单：

int MainLoop(Board board) {

while (!board.IsEndOfGame()) {

boardViewer.ShowBoard(board);

GetPlayer(board.CurrentColor).PlayOneMove(board);

}

return board.BlackScore – board.WhiteScore;

}

AbstractAIPlayer, 作为各个AIPlayer的基类实现了一些共同的函数

AIPlayer1 – AIPlayer5, 如前所介绍的，5种不同的AIPlayer的实现

AIPlayer4Plus, 后期为GUI定制的一个AIPlayer. 无论哪种player, 都存在开局太慢的缺点，于是AIPlayer4Plus引入了IAIConfig接口，可以让AI在各个不同阶段使用不同的深度进行搜索。这个东西可以很方便引入任何一个AIPlayer中，但这里仅仅为AIPlayer4实现了一个。由于测试方便等考虑，AIPlayer4没有被直接修改，而是添加了一个新类。

NullBoardViewer, IBoardViewer的一个最trivial的实现，什么也不做，测试AI时会用到

StaticEvaluator, 静态评估器，实现了前述的那些features以及他们的组合。类似于IAIConfig, 这里也引入了动态配置接口IStaticEvaluatorConfig, 使得各个feature可以在不同阶段发挥不同的权重。

TranspositionTable, 置换表，内部使用64个Hashtable实现，显得比较浪费，但由于Othello局面的单调性，这样便于及时清除不需要的内存。

Utility, 工具类，实现了2个默认的Config, 以及其他一些有用的全局函数。

AITest, 让所有AI对战（AIPlayer4Plus除外）的测试框架，记录了许多有用的数据（棋谱没有记录，估计也没人高兴看 :b ）。

**3.1.2 TinyOthello.ConsoleUI namespace**

ConsoelBoardViewer, 实现了IBoardViewer接口，它把棋盘打印在Console上，其中黑子用x表示，白子用o表示，可以下棋的地方(valid move)用\_表示，其它空格用.表示，举例如下：

a b c d e f g h

1 . . . . . . . .

2 . . . . . . . .

3 . . . \_ x \_ . .

4 . . . x x . . .

5 . . . o x \_ . .

6 . . . . . . . .

7 . . . . . . . .

8 . . . . . . . .

Current move: o

Current step: 2

Current status: x 4 o 1

Last move position: e3

ConsoleHumanPlayer, 实现了IPlayer接口，他读取键盘命令来下棋。主要的命令有：

pxy 在(x, y)处落子，如pa2, 坐标顺序及大小写相关，不能有空格，下同

pass 无子可落时pass

un undo n步棋

rn redo n步棋

**3.2 TinyOthelloGUI**

这个project主要实现了TinyOthello.GraphicUI namespace里面的几个类：

BoardCell, 是一个控件类，定义了一个可以落子的点，继承于Button类，用修改背景颜色和背景图片的方法显示棋子。

MainWindow, 程序的主窗口，提供了界面和基本功能，实现了IBoardViewer接口，并且将Game类在另一个thread里运行，并负责其管理。所有代码都有适当的同步（其实Kernel里的AIPlayer在下棋的时候都会lock(board), 这里也有相应的措施）。内部用BoardCell数组组成棋盘来表示显示，并接受鼠标点击的事件，判断适当条件后，引起自定义事件MovePlayed, 让event handler处理。

GraphicHumanPlayer, 实现了IPlayer接口，其中PlayOneMove()需要等待用户输入，因此会用Monitor.Wait()来等待MovePlayed事件。具体见代码。

### 4. 参考资料

[1] <http://www.cs.ualberta.ca/~jonathan/Courses/657/index.html>

Course Notes of CMPUT 657 Heuristic Search by Jonathan Schaeffer

[2] <http://www.othello-china.com>

Othello的规则、战术等信息

[3] <http://www.elephantbase.net/computer.htm>

以电脑象棋为主题的网站，对弈程序的基础知识

[4] <http://xenon.stanford.edu/~lswartz/cs221/desdemona_writeup.pdf>

某小组的Othello的实验报告，具有一定参考价值

[5] 《PC游戏编程（人机博弈）》 王小春 编著