黑 白 棋 算 法

———Reversi Agent 0.1 的设计

作者：柏爱俊（PB05210411） <baj@mail.ustc.edu.cn, baj_forever@126.com>

这是一个用于黑白棋比赛的 agent，比赛平台是 rvrsserver/rvrsmonitor。下面将从底层设计和人工智能两个方面介绍它的具体设计。

一、 底层设计

（一）、通信接口

通信方面严格遵守 rvrsserver 的通信协议（具体请见 rvrsserver 的用户手册），通信的主要流程是一个消息循环，实现的代码如下：

```
while( client->get_msg( msg, MSG_SIZE)){
        switch( board->do_msg( msg, MSG_SIZE)){
                case NEED_TO_SEND:
                        client->send_msg( msg, MSG_SIZE); break;
                case NO_NEED_TO_SEND:
                        break;
                case FINISHED:
                        delete board; delete client; return 0;
        }
}
```

其中 client 和 board 分别指向一个 TCPClient 类和 Board 类的实例，这一过程可以简单理解为 client 得到消息，就让 board 去处理，board 处理完消息并根据消息的具体类型决定是否要回复消息给 Server，还是继续等待消息或者直接退出循环。具体来讲需要作出决策的消息是需要回复的（如 ACTION, ONE_MORE_TIME, TWO_TIMES_LEFT 以及 THREE_TIMES_LEFT），描述性消息则不需要回复（如 BOARD_DESCRIPTION, WAIT, RL_BLACK 和 RL_WHITE），而告知胜负的消息则表示应该退出了（YOU_WIN, YOU_LOSE 以及 YOU_TIE）。 实现通信的具体细节与主题无关，这里就不多介绍了，在源码里你可以看到完整的实现。

（二）、棋盘的表示

Board 类里面，我用了一个长为 66 的 char 型数组表示棋盘，其声明为 char layout[66]，layout[0]没有用，layout[65] == '\0'，它们看上去就像一个字符串，所以任何时候我需要保存或恢复棋盘时只要调用 strcpy 就行了（后面将会看到这是很频繁的操作）。layout[x+8*(y-1)]表示棋盘上坐标为(x, y)的格子的状态，为了真正做到像一个字符串，用 1 表示该格为空、2 表示该格为黑棋、而 3 表示为白棋，这样 layout 串中就不会出现具有特殊意义的'\0'了。Board 对棋盘上的所有格子是用编号 1-64 区分的，但操作上以坐标(x, y)更为方便，如果某一格的编号为 pos,那么它对应的坐标为(x = (pos%8 == 0)? 8: pos%8, y = (pos%8 == 0)? pos/8: pos/8 + 1)，反过来如果坐标为(x, y)， 那么其对应编号为 pos = x+8*(y-1)，为了节省操作的时间，就专门用数组 xx[65]，yy[65]和 pp[9][9]做了对应。

（三）、着法的产生

根据黑白棋的规则，正确的着法至少能翻一个对方的棋子为自己的棋子。考虑某一步能不能走时，需要对该步周围的几个方向都进行遍历，只要有一个方向可以着子，那么该步就可以着子，而不同的位置周围可能的方向并不相同，要区分角落、边缘和内部的差别。区分的方法是和方向的表示有关的，程序中我用一个 unsigned char（恰好 8 位）标志某一步周围的八个方向的可行性，之所以用 unsigned char 而不是 char 那是和移位操作有关的，因为有的计算机对有符号数进行右移操作时会在

左端补一，从而使方向表示上产生混乱。程序中的一个全局只读数组

```
static const unsigned char possible_dir[][8] = {
      { 0x83, 0x8F, 0x8F, 0x8F, 0x8F, 0x8F, 0x8F, 0x0E},
      { 0xE3, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3E},
      { 0xE3, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3E},
      { 0xE3, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3E},
      { 0xE3, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3E},
      { 0xE3, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3E},
      { 0xE3, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x3E},
      { 0xE0, 0xF8, 0xF8, 0xF8, 0xF8, 0xF8, 0xF8, 0x38}
},
```

就对不同位置的可行方向做了标志。比如位置(2, 1)的方向表示为0x8F，写成二进制数即为

```
 1--2--3--4--5--6--7--8
[1][0][0][0][1][1][1][1],
```

上面的一行1-8表示从左到右数的第几位，分别对应于实际方向的E, NE, N, NW, W, SW, S和SE，第i位为[1]就表示该位对应的方向是可行的，应该检查这个方向，否则就是不可行的。基于如上表示，当要检查某个位置是否可以着子时，只要根据该位置的方向表示遍历每个方向就可以了，这主要是由函数

```
inline bool Board::can_put(int x, int y, char color)
{
      if(get_color(x, y) != CH_EMPTY)
            return false;

      unsigned char dir = possible_dir[y-1][x-1];
      unsigned char ptr = 0x80;
      while( true){
            while( ptr && !(ptr & dir))
                  ptr >>= 1;
            if( !ptr)
                  break;
            if( stretch(x, y, ptr, color))
                  return true;
            ptr >>= 1;
      }
      return false;
}
```

完成的，该函数调用的函数stretch(x, y, ptr, color)则具体检查一个方向，其定义如下：

```
inline int Board::stretch(int x, int y, unsigned char ptr, char color)
{
      static int n;
      static char opp_color;
      n = 0;
      opp_color = opposite(color);
      next_pos(x, y, ptr);
      while( x > 0 && x < 9 && y > 0 && y < 9 \
            && get_color(x, y) == opp_color){
            next_pos(x, y, ptr);
            n++;
      }
      if( x < 1 || x > 8 || y < 1 || y > 8)
            return 0;
      else if( get_color(x, y) == color)
            return n;
      else return 0;
}
```

事实上stretch返回了方向ptr上可翻过对方子的个数，next_pos函数只是简单的得到下一个位置：

```
inline void Board::next_pos(int &x, int &y, unsigned char ptr)
{
	switch(ptr){
		case 0x80: x++;		return;
		case 0x40: x++; y--; return;
		case 0x20:		y--; return;
		case 0x10: x--; y--; return;
		case 0x08: x--;		return;
		case 0x04: x--; y++; return;
		case 0x02:		y++; return;
		case 0x01: x++; y++; return;
	}
}
```

至此已经可以判断出所有可行的着子方法了。

（四）、着子的实现

着子的关键是翻过对方的棋子，事实上前面的 stretch 函数已经完成了大部分工作，put 函数如下

```
inline void Board::put( int pos, char color)
{
	unsigned char dir = possible_dir[yy[pos]-1][xx[pos]-1];
	unsigned char ptr = 0x80;
	set_color(xx[pos], yy[pos], color);
	while( true){
		while( ptr && !(ptr & dir))
			ptr >>= 1;
		if( !ptr)
			break;
		update_dir(xx[pos], yy[pos], ptr, color);
		ptr >>= 1;
	}
}
```

update_dir 负责更新某一特定的方向

```
inline void Board::update_dir(int x, int y, unsigned char ptr, char color)
{
	static int _i;
	_i = stretch(x, y, ptr, color);
	for(; _i > 0; _i--){
		next_pos(x, y, ptr);
		set_color(x, y, color);
	}
}
```

这样底层就做好了，下面介绍“人工智能”。

二、人工智能

（一）、估值函数

下棋过程中当有好几步可以选择时，棋手就要作出决策，下对自己有利的一步，那么如何判断是否有利呢？最简单的办法就是找一个好的估值函数，这个函数对每一种局面给出一个估值，估值越高表明对自己越有利，我的程序里主要采用了基于棋格表和行动力(mobility)的估值。棋格表其实就是对不同位置加权的权值表，而考虑行动力即是要争取使自己有尽可能多的可着子位置。另外也考虑了内部子的个数，内部子越多说明自己的子越集中。实际发现这样的组合还是不错的。

下面是估值函数的实现：

```
inline int Board::evaluation()
{
      static int comp_counts, oppo_counts, delta_cost;
      comp_counts = oppo_counts = 0;
      delta_cost = 0;
      for( int i = 1; i < 65; i++){
            if( get_color(i) == role){
                  comp_counts ++;
                  delta_cost += cost[yy[i]-1][xx[i]-1];
                  if( adjacent(xx[i], yy[i], CH_EMPTY))
                        delta_cost --;
            }
            else if( get_color(i) == opp_role){
                  oppo_counts ++;
                  delta_cost -= cost[yy[i]-1][xx[i]-1];
                  if( adjacent(xx[i], yy[i], CH_EMPTY))
                        delta_cost ++;
            }
            else {
                  if(can_put(i, role))
                        delta_cost += 2;
                  if(can_put(i, opp_role))
                        delta_cost -= 2;
            }
      }
      if( !oppo_counts)
            return MAX_INT;        //this is really the best of all
      if( !comp_counts)
            return -MAX_INT;       //but this is the worst
      return n_coeff * (comp_counts - oppo_counts) +\
                              m_coeff * delta_cost;
}
```

n_coeff 和 m_coeff 是之前计算好的两个系数，表明了棋子数之差与综合估值之差孰重孰轻，在开局阶段 m_coeff > n_coeff，而在终局阶段 n_coeff > m_coeff。另外，我在程序中使用了简单的动态修改权值的方法，使下面将介绍的搜索有了一定的实时性。

（三）、搜索

由于很多情况下，估值函数并不能预见危险的存在，或者只能看到暂时的失利，便就有了搜索的必要了。我在程序里采用了一种叫最大-最小（MinMax）的搜索方法，另外也结合了 alpha-beta 剪枝的技术，使棋力增大不少，一般情况下可以往前搜到八步，终局则最多到十四步。为了较好的达到剪枝的效果，第一步搜索时先对各种走法做了预处理，初步排了一下序，这样看上去较好的走法最先搜索，则后面搜索时剪枝就会比较明显，其实程序中搜索的每一层本也可以像这样先预处理再搜索的，但实践发现效果不一定好，而且浪费时间，所以就放弃了。搜索其实是一个递归的过程，类似于树的深度优先遍历，其实现如下：

```
int Board::estimation(char color, int depth, int alpha, int beta)
{
      if((depth >= max_depth))
            return evaluation();
      if( !able_to_put(color)){
            color = (color == role)? opp_role: role;
            if( !able_to_put(color))
                  return evaluation();
      }
      char cpy[66];
      int value;
```

```
      int result = (color == role)? -MAX_INT: MAX_INT;

      for(int i = 1; i < 65; i++){
            if( !can_put(i, color))
                  continue;

            strcpy(cpy, layout);
            put(i, color);
            modify_cost();
            value = estimation((color == role)? opp_role: role, \
                                          depth+1, alpha, beta);
            strcpy(layout, cpy);

            if( color == role){
                  if( value > alpha){
                        if( (alpha = value) >= beta)
                              return alpha;
                  }
                  result = (value > result)? value: result;
            }
            else{
                  if(value < beta){
                        if( (beta = value) <= alpha)
                              return beta;
                  }
                  result = (value < result)? value: result;
            }
      }
      return result;
}
```

第一步的搜索和上面的不太一样，如下所示：

```
inline int Board::best_choice()
{
      char cpy[66];
      int  max = -MAX_INT;
      int  alpha = -MAX_INT;
      int  optim_pos = 0;
      int  value;
      List list = new Node(0, 0, 0);
      Node *p, *q;
      for(int i = 1; i < 65; i++){
            if( !can_put(i, role))
                  continue;

            strcpy(cpy, layout);
            put(i, role);
            value = evaluation();
            strcpy(layout, cpy);

            for(q = list, p = list->next; p && p->wgh >= value; \
                                                q = p, p = p->next);
            q->next = new Node(value, i, p);
      }

      for(p = list->next; p; p = p->next){

            strcpy(cpy, layout);
            put(p->pos, role);
            modify_cost();
```

```
            value = estimation(opp_role, 0, alpha, MAX_INT);
            strcpy(layout, cpy);

            if( value > alpha){
                  if( (alpha = value) >= MAX_INT)
                        return p->pos;
            }
            if(value > max){
                  max = value;
                  optim_pos = p->pos;
            }
      }
      for(p = list; p;){
            q = p;
            p = p->next;
            delete q;
      }
      return optim_pos;
}
```

至此就全部介绍完了，不过我的程序尚有很多值得改进的地方：

（1）、实践发现估值函数对结果的影响很大，在搜索不能深入的情况下基本上关系到胜负，而我的估值函数还太简单（近阶段也不能写的太复杂，因为搜索还跟不上），考虑的因素不多，各因素的关系只是凭经验臆断的，不一定能反应实际情况，比赛中就经常发现一些有争议的估值。

（2）、虽然我使用了 alpha-beta 的搜索方法，但搜索深度还是不太大，做得好的程序在中局就可以搜索到 20 步以上，我的程序底层不高效是一个原因，另外一个就是我的搜索中有很多的重复搜索，至少有三个方面存在重复搜索：1、上一回合搜索的节点在本回合有重复搜索，比如每次都搜七步，那么上次搜索的后六步在本次就重复搜索了；2、同一层有重复搜索，比如说我有两个位置 A 和 B 可以走棋，而对方有一个位置 C 可以走棋，那么走棋的顺序 ACB 和 BCA 所形成的棋局是一样的，而在我的程序里就被重复搜索了；3、具有对称性的棋局被重复搜索了，这在走第一步时特别明显。第一个问题好像可以在搜索过程中建一棵树的办法解决，利用树记录搜索的过程，不过好像太费内存了，对于第二个问题可以借助 hash 表的方法，迅速判断棋局是否有重复，如果在 hash 的键值计算里考虑上对称性的影响那么第三个问题也就解决了。

（3）、该说到底层的实现了，我实现的棋盘只是棋盘而不包含任何其他信息，以至于递归的时间大部分花在了判断“能不能在这里下”上了，觉得这一点应该改进，但如果只是把搜索时的判断（指 for 循环里力的）转移到棋局的更新函数中并不能节省太多时间，值得考虑的是一种可以称为“异或”的方法，即只更新两个棋局中不同的地方，而这似乎不太好做。

参考文档：

1). http://study.feloo.com/computer/pro/c/game/200506/38229.html
怎样编制黑白棋（1 － 4）

2). http://ce.sharif.edu/~ahmadine.jad/
An Introduction to Game Tree Algorithms

3). http://home.tiscalinet.ch/t_wolf/tw/misc/reversi/index.html
The Anatomy of a Game Program

4). http://www.elephantbase.net/computer/basic_advanced.htm