Alpha-Beta搜索的诸多变种

David Eppstein */文
* 加州爱尔文大学(UC Irvine)信息与计算机科学系

尽管我们已经讨论过Alpha-Beta搜索简单有效，还是有很多方法试图更有效地对博弈树进行搜索。它们中的大部分思想就是，如果认为介于Alpha和Beta间的评价是感兴趣的，而其他评价都是不感兴趣的，那么对不感兴趣的评价作截断会让Alpha-Beta更有效。如果我们把Alpha和Beta的间距缩小，那么感兴趣的评价会更少，截断会更多。

首先让我们回顾一下原始的Alpha-Beta搜索，忽略散列表和“用当前层数调整胜利值”的细节。

```
// 基本的Alpha-Beta搜索
int alphabeta(int depth, int alpha, int beta) {
    move bestmove;
    if (棋局结束 || depth <= 0) {
        return eval();
    }
    for (每个合理着法 m) {
        执行着法 m;
        score = -alphabeta(depth - 1, -beta, -alpha);
        if (score >= alpha) {
            alpha = score;
            bestmove = m;
        }
        撤消着法 m;
        if (alpha >= beta) {
            break;
        }
    }
    return alpha;
}
```

超出边界(Fail-Soft)的Alpha-Beta

以上代码总是返回Alpha、Beta或在Alpha和Beta之间的数。换句话说，当某个值不感兴趣时，从返回值无法得到任何信息。原因就是当前值被变量Alpha所保存，它从感兴趣的值的窗口下沿开始一直增长的，没有可能返回比Alpha更小的值。一个对Alpha-Beta的简单改进就是把当前评价和Alpha分开保存。下面的伪代码用常数“WIN”表示最大评价，它可以在Alpha-Beta搜索中返回任何评价。

```
// 超出边界的Alpha-Beta搜索
int alphabeta(int depth, int alpha, int beta) {
    move bestmove;
    int current = -WIN;
```

```
    if (棋局结束 || depth <= 0) {
        return eval();
    }
    for (每个可能的着法 m) {
        执行着法 m;
        score = -alphabeta(depth - 1, -beta, -alpha);
        撤消着法 m;
        if (score >= current) {
            current = score;
            bestmove = m;
            if (score >= alpha) {
                alpha = score;
            }
            if (score >= beta) {
                break; // 【译注：这里可以直接返回score、current或alpha，如果返回beta，则是不会高出边界的Alpha-Beta搜索。】
            }
        }
    }
    return current;
}
```

这样改动以后，就可以知道比以前稍多一点的信息。如果返回值*x*等于或小于Alpha，我们仍然不知道局面的确切值(因为我们可能在搜索中裁剪了一些线路)，但是我们知道确切值最多是*x*。类似地，如果*x*大于或等于Beta，我们就知道搜索值至少是*x*。这些微小的上界和下界变化不会影响搜索本身，但是它们能导致散列表命中率的提高。超出边界的Alpha-Beta搜索对下面要介绍的MTD(*f*)算法有重要作用。

期望搜索

这个方法不是代替Alpha-Beta的，只是从外部改边一个途径来调用搜索过程。通常用Alpha-Beta找最好路线时，可以调用：

```
alphabeta(depth, -WIN, WIN)
```

这里 -WIN 和 WIN 之间的范围很大，表明我们不知道搜索值是多少，因此任何可能的值都必须考虑。随后，要走的那步保存在搜索函数外部的变量中。
期望搜索的不同之处在于，我们可以人为地指定一个狭窄窗口(用前一个搜索值为中心)，对搜索通常是有帮助的。如果你搜索失败，必须加宽窗口重新搜索：

```
// 期望搜索
int alpha = previous - WINDOW;
int beta = previous + WINDOW;
for ( ; ; ) {
    score = alphabeta(depth, alpha, beta);
    if (score <= alpha) {
        alpha    = -WIN;
    } else if (score >= beta) {
        beta = WIN;
    } else {
```

```
            break;
        }
    }
```

权衡狭窄搜索所节约的时间，和因为失败而重新搜索的时间，可以决定常数 WINDOW 的大小。在国际象棋中，典型的值也许是半个兵。期望搜索的变体有，搜索失败时适当增加窗口宽度，或者用初始窗口而没必要以前一次搜索结果为中心，等等。

MTD(*f*)

这个技术跟期望搜索一样，只是在调用Alpha-Beta时对初始值进行调整。搜索窗口越窄搜索就越快，这个技术的思想就是让搜索窗口尽可能的窄：始终用“beta = alpha + 1”来调用Alpha-Beta。用这样一个“零宽度”搜索的作用是用Alpha和确切值作比较：如果搜索的返回值最多是Alpha，那么确切值本身最多是Alpha，相反确切值大于Alpha。

我们可以用这个思想对确切值作二分搜索：

```
int alpha = -WIN;
int beta = +WIN;
while (beta > alpha + 1) {
    int test = (alpha + beta) / 2;
    if (alphabeta(depth, test, test + 1) <= test) {
        beta = test;
    } else {
        alpha = test + 1;
    }
}
```

但是，这样会导致大规模的搜索(即 WIN 和 -WIN 的差的对数)。而MTD(*f*)的思想则是用超出边界的Alpha-Beta对搜索进行控制：每次调用超出边界的Alpha-Beta就会返回一个值更接近于最终值，如果用这个搜索值作为下次测试的开始，我们最终会达到收敛。

```
// MTD(f)
int test = 0;
for ( ; ; ) {
    score = alphabeta(depth, test, test + 1);
    if (test == score) {
        break;
    }
    test = score;
}
```

不幸的是，它和散列表之间的复杂作用会导致这个过程陷入无限循环，所以必须加上额外的代码，如果迭代次数太多而没有收敛，就必须中止搜索。

MTD(*f*)的一个大的优势在于我们可以简化Alpha-Beta搜索，因为它只需要两个参数(深度和Alpha)而不是三个。【据说这样做可以提高并行计算的效率，遗憾的是译者对并行计算一窍不通。】

【为了对MTD(*f*)有更详细的了解，译者查阅了该算法的发明者Plaat的网站，他提供的MTD(*f*)代码中用了两个边界，可以防止迭代过程中出现振荡而不收敛的情况，代码如下(原来是Pascal语言，现被译者转写为C语言)：

```
int MTDF(int test, int depth) {
    int score, beta, upperbound, lowerbound;
```

```
        score = test;
        upperbound = +INFINITY;
        lowerbound = -INFINITY;
        do {
            beta = (score == lowerbound ? score + 1 : score);
            score = alphabeta(depth, beta - 1, beta);
            (score < beta ? upperbound : lowerbound) = score;
        } while (lowerbound < upperbound);
        return score;
    }
```

而关于MTD(*f*)的使用另有以下几点技巧：

(1) 通常试探值并不一定设成零，而是用迭代加深的形式由浅一层的MTD(*f*)搜索给出；

(2) 局面评价得越粗糙，MTD(*f*)的效率越高，例如国际象棋中可使一个兵的价值由100降低为10，其他子力也相应比例降低，以提高MTD(*f*)的效率(但粗糙的局面评价函数却不利于迭代加深启发，因此需要寻求一个均衡点)；

(3) 零宽度窗口的搜索需要置换表的有力支持，因此称为“用存储器增强的试探驱动器”(Memory-enhanced Test Driver，即MTD)，它只需要传递两个参数(深度*n*和试探值*f*)，故得名为MTD(*n*,*f*)，缩写为MTD(*f*)。】

PVS

或许最好的Alpha-Beta变体，要算是这些名称了：负值侦察(NegaScout)和主要变例搜索(Principal Variation Search，简称PVS)。这个思想就是当第一次迭代搜索时找到最好的值，那么Alpha-Beta搜索的效率最高。对着法列表进行排序，或者把最好的着法保存到散列表中，这些技术可能让第一个着法成为最佳着法。如果真是如此，我们就可以假设其他着法不可能是好的着法，从而对它们快速地搜索过去。

因此PVS对第一个搜索使用正常的窗口，而后续搜索使用零宽度的窗口，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。

```
    // 主要变例搜索(超出边界的版本)
    int alphabeta(int depth, int alpha, int beta) {
        move bestmove, current;
        if (棋局结束 || depth <= 0) {
            return eval();
        }
        move m = 第一个着法;
        执行着法 m;
        current = -alphabeta(depth - 1, -beta, -alpha);
        撤消着法 m;
        for (其余的每个着法 m) {
            执行着法 m;
            score = -alphabeta(depth - 1, -alpha - 1, -alpha);
            if (score > alpha && score < beta) {
                score = -alphabeta(depth - 1, -beta, -alpha);
            }
            撤消着法 m;
            if (score >= current) {
                current = score;
```

```
            bestmove = m;
            if (score >= alpha) {
               alpha = score;
            }
            if (score >= beta) {
               break;
            }
         }
      }
      return current;
   }
```

这个算法跟MTD(*f*)有个同样的优势，即搜索树的大多数结点都以零宽度的窗口搜索，可以用双参数的Alpha-Beta。由于“Beta > Alpha + 1”的调用非常少，因此不必担心额外的工作(例如保存最佳着法以供将来使用)会占用很多时间。【原作者的意思是，调用零宽度窗口的搜索时，可以免去保存最佳着法等操作，因此可以省下不少时间。】

推荐

我自己的程序结合了期望搜索(用在整个搜索过程以外)和PVS(用在搜索过程内部)。但是不同的棋类游戏不一样，由于这些搜索不难实现，所以要在这些方法中进行选择或调节参数，就必须对它们逐一实现并做一些试验。它们都必须返回同样的搜索值(如果受散列表影响，那么至少是相近的值【例如常规的Alpha-Beta搜索和超出边界的Alpha-Beta搜索，在使用散列表时可能会返回不同的值】)，但搜索的结点数会不同。在你的棋类的典型局面中，能使搜索树最小的方法则被采纳。

原文：http://www.ics.uci.edu/~eppstein/180a/990202b.html
译者：象棋百科全书网 (webmaster@xqbase.com)
类型：全译加译注

- 上一篇　高级搜索方法——简介(一)
- 下一篇　高级搜索方法——静态搜索
- 返　回　象棋百科全书——电脑象棋