从极大极小算法到主要变例搜索

孙锴

1 综述

人机对弈在计算机诞生前就开始了发展，时至今日，人机对弈领域提出的搜索算法数目已经非常之多，但从根本上看，许多搜索算法之间的内在的核心思想是一致的。本文介绍将从极大极小搜索开始，讲解其一步步推广到Alpha-Beta搜索，以至于最终推广到实际应用中主要变例搜索中的推广过程。

本文结构安排如下：第 2节介绍博弈树；第 3节介绍极大极小搜索；第 4节介绍负值最大搜索；第 5节介绍Alpha-Beta剪枝；第 6节介绍窗口；第 7节介绍主要变例搜索；最后于第 8总结全文。

本文中的两张图片分别来自[2]（有修改）与[1]，另外本文的行文及伪代码参考了[2]。

2 博弈树（Game Tree）

任何博弈类游戏都可定义出一棵有根树，树的每个结点代表一个局面，结点的子结点是该结点所代表的局面走一步可以到达的局面，特别的，树的根节点是初始的局面。例如图1是井字棋（Tic-tac-toe）的博弈树（这棵树中我们借助对称性质去掉了部分结点，例如如果不考虑对称，那么根节点应有9个子结点）。

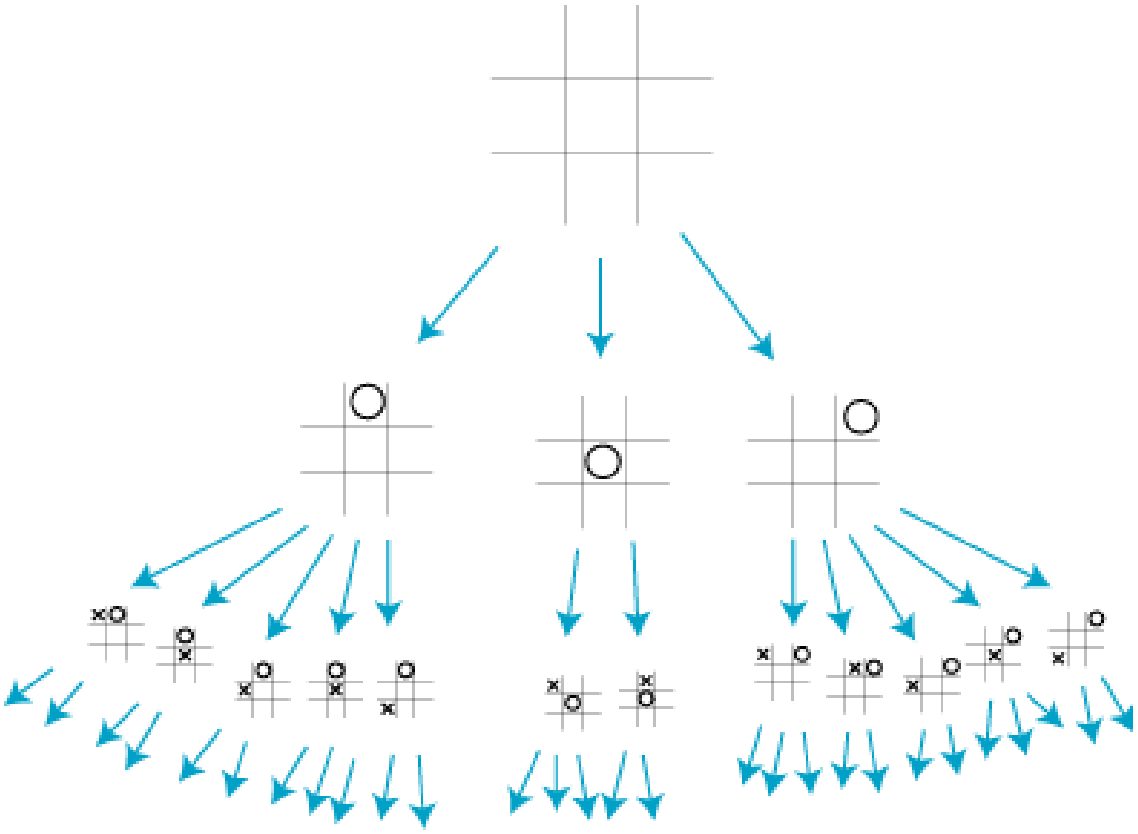

Figure 1: Game Tree

为了方便讨论，我们这里设这里讨论的博弈树是一棵有限树，设有两个棋手甲与乙进行这场博弈，这样，博弈树分为三类结点：

1. 奇数层的非叶子结点：棋手甲要走的局面。
2. 偶数层的非叶子结点：棋手乙要走的局面。
3. 叶子节点：对局的结果（甲胜，乙胜或平局）。

对于叶子结点，我们是知道它对应局面的对局结果的。假设某个非叶子结点的所有子结点都是叶子结点，那么棋局必然会在一回合内结束，我们假定棋手会挑选对自己最有利的着法。那么如果存在某个子结点对应的局面是该棋手胜，则该棋手必定会选择到达这类胜利局面所对应的着法；如果没有子结点对应的局面是该棋手胜，但是存在某个子结点对应的局面是平局，则该棋手必定会选择到达这类平局局面所对应的着法；如果所有子结点对应的局面都是该棋手输，则该棋手无论选择什么着法，他都会输。因此，对于这类结点（所有子结点都是叶子结点的非叶子结点）我们也是可以知道局面的对局结果的。我们可以采用同样的方式继续向更靠近根的非叶子结点处推演，于是最终对于博弈树中每个点，我们都可以知道其对应局面的对局结果。 对于井字棋，由于可能的局面数量很小，所以我们可以将博弈树穷举出来，在此基础上，对于任意局面，我们都可以由博弈树直接得到该局面在双方都采取最优策略下的对局结果，以及在该局面中的最优着法。

3 极大极小搜索（Minimax Search）

然而在更复杂的情况中（例如国际象棋），由于可能的局面数巨大，而计算能力有限，我们无法穷举整棵博弈树，因此我们不再能通过穷举博弈树获得对任意局面的评价信息（对局结果）。于

是，此时我们推广之前对博弈树的定义，将博弈树的树根对应于我们希望获得其评价信息的局面，将我们能够由此树根穷举出的前若干层博弈树作为我们实际讨论的博弈树。由于此时的博弈树的叶子节点并不一定对应着对局的结果（即甲胜、乙胜或平局），所以我们设计一个局面评估的函数，对每个叶子结点赋予一个整数（也可以是实数，不过为了方便在计算机中表示以及下文的讨论，我们这里限定为整数），整数的大小代表着我们对对应局势偏向于甲或乙的估计（局面评分）。为了方面讨论，我们设整数越大对甲越有利，$+\infty$对应甲胜，整数越小对乙越有利，$-\infty$对应乙胜。

对于叶子结点，我们是知道它对应局面的局面评分的。现在，我们用类似前文的分析方法分析非叶子结点。假设某个非叶子结点的所有子结点都是叶子结点，我们假定棋手会挑选对自己最有利的着法。如果该非叶子结点是奇数层非叶子结点（对应棋手甲要走的局面），则棋手必定会选择到达局面评分最大的子结点所对应的着法，从而该结点所对应局面的优劣可以由其所有子结点局面评分的最大值来衡量，这个值即可成为该结点所对应局面的局面评分。如果该非叶子结点是偶数层非叶子结点（对应棋手乙要走的局面），则棋手必定会选择到达局面评分最小的子结点所对应的着法，从而该结点所对应局面的优劣可以由其所有子结点局面评分的最小值来衡量，这个值即可成为该结点所对应局面的局面评分。因此，对于这类结点（所有子结点都是叶子结点的非叶子结点）我们也是可以知道局面的局面评分的。我们可以采用同样的方式继续向更靠近根的非叶子结点处推演，于是最终对于博弈树中每个点，我们都可以知道其对应局面的局面评分，根结点对应局面的局面评分就是我们希望获得的信息。

实际上以上这段话所介绍的推演过程就是极大极小搜索。伪代码描述如下：

```
int MinMax(int depth) {
  if (SideToMove() == JIA) { // 甲方
    return Max(depth);
  } else { // 乙方
    return Min(depth);
  }
}

int Max(int depth) { // 甲方
  int best = -INFINITY;
  if (depth <= 0) { // 如果是叶子
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = Min(depth - 1); // 取子结点评分的最大值
    UnmakeMove();
    if (val > best) {
      best = val;
    }
  }
  return best;
}

int Min(int depth) { // 乙方
  int best = INFINITY; // 注意这里不同
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = Max(depth - 1); // 取子结点评分的最小值
```

```
    UnmakeMove();
    if (val < best) { // 注意这里不同
      best = val;
    }
  }
  return best;
}
```

4 负值最大搜索（Negamax）

观察前面给出的极大极小搜索的伪代码，不难看出过程Max和过程Min具有很高的相似度，我们可以通过一个巧妙的变换将Max与Min以及调用它们的MinMax合并重写为一个过程。这一变换的核心思想源于对局面评分定义的巧妙修改。我们将“整数越大对甲越有利，$+\infty$对应甲胜，整数越小对乙越有利，$-\infty$对应乙胜”修改为“整数越大对当前局面将落子方越有利，$+\infty$对应当前局面将落子方胜，整数越小对当前局面将落子方越不利，$-\infty$对应当前局面将落子方败”。这一修改仅仅是换了一种表示方法（即将极大极小搜索的博弈树中奇数层的局面评价值取其相反数，偶数层局面评价值不变），并没有改变局面评分和极大极小搜索的工作原理，却使得奇偶层结点局面评分的计算方法由过去的不同变为了相同，更具体地说，现在所有非叶子结点对应局面的局面评分都等于其子结点对应局面的局面评分的相反数的最大值。我们称这种修改后的搜索为负值最大搜索，其伪代码如下：

```
int NegaMax(int depth) {
  int best = -INFINITY;
  if (depth <= 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -NegaMax(depth - 1); // 注意这里有个负号。
    UnmakeMove();
    if (val > best) {
      best = val;
    }
  }
  return best;
}
```

易见极大极小搜索与负值最大搜索是完全等价的（遍历结点的顺序，返回值均相同），然而后者代码短了很多，这极大地方便了代码的维护，减少了出错的可能。

5 Alpha-Beta剪枝（Alpha-Beta Pruning）

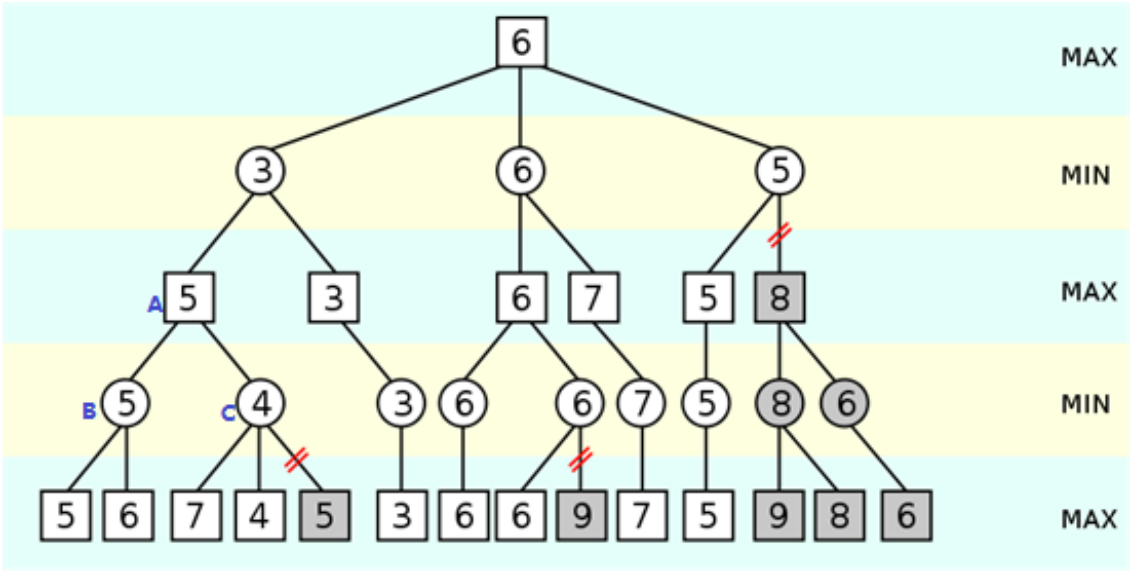

Figure 2: Alpha-Beta Pruning

如果我们对极大极小搜索的搜索过程进行进一步的分析，会发现对于搜索过程中的一些结点，即使不计算（或者说不知道）其局面评分，我们也可以通过推理知道它不会影响对博弈树根的局面评分的计算。例如图2所示的博弈树中，如果我们的搜索顺序是从左到右，那么实际上我们可以忽略所有灰颜色的结点的局面评分的计算。举个更具体的例子，因为A结点的局面评分是B结点的局面评分与C结点的局面评分的最大值，而C结点的局面评分是它所有子结点局面评分的最小值，所以在完成C的第二个子结点的搜索后，我们就知道了C的局面评分不会超过4，因此此时我们就知道A结点的值不会受以C结点为根的子树的影响，于是对于最左侧的局面评分为“5”的灰色结点，我们完全可以不对其进行搜索。以上就是Alpha-Beta剪枝的思想。

当然我们可以在前面给出的Minimax的代码中将上述思想直译实现，然而我们可以做得更好。下面我们介绍在Negamax上优雅地实现上述思想的方法。我们在搜索中传递两个值，第一个值是已搜索到的对自己的最好值，我们称它为Alpha，由于在Negamax中我们是求子结点局面评分的相反数的最大值，所以任何比Alpha小的值都不会改善我们的搜索结果。第二个值是已确定的对于对手来说最坏的值，我们称它为Beta。这是对手所能承受的最坏的结果，我们知道在对手看来，无论我们做如何的决策，他总可以找到一个对策不比Beta更坏。

我们先不讨论如何确定Alpha与Beta值，而是首先分析一下是裁剪是如何发生的。如果搜索过程中我们获得Beta或比Beta更好的值，对于我们来说是足够好的事情，因为此时由Beta的定义可知对手一定不会让我们有机会采取某种策略到达当前搜索过程的局面，于是我们就可以停止当前结点后续的搜索（Beta截断）。

注意到上面对裁剪的讨论并没有涉及到Alpha，那为什么需要Alpha呢？对于给定局面，如果当前局面在当前搜索过程中的Alpha和Beta值为 α 与 β，那么仔细想一下Alpha与Beta的定义就会发现，对于搜索过程中即将搜索的该局面的下一个着法所对应的子局面，它的Alpha和Beta值应为 $-\beta$ 与 $-\alpha$。正因如此，我们同时需要Alpha与Beta两个值，用Beta值完成当前局面的剪枝，用Alpha值确定子局面的Beta值。

最后我们讨论Alpha与Beta的确定。由上述讨论我们知道，一个非根结点的局面的Alpha与Beta值初始值可由其父局面的Alpha与Beta值确定，对于根结点，其Alpha与Beta的初始值由定义可知为 $-\infty$ 与 $+\infty$。初始化后，我们对每一个着法进行搜索，并在完成每一次子结点的搜索后都考虑对Alpha与Beta值的更新。由Alpha的定义可知，如果某个着法的结果小于或等于Alpha，那么它就是很差的着法，因此可以抛弃，同时我们对Alpha和Beta不做变动。如果某个着法的结果大于或等Beta，那么整个结点就作废了，因为对手不希望走到这个局面，而它有别的着法可以避免到达这个局面，因此我们也无需变动Alpha和Beta，直接退出当前结点的搜索。如果某个着法

的结果大于Alpha但小于Beta，那么这个着法就是走棋一方可以考虑走的，除非以后有所变化，由Alpha的定义，我们将Alpha值改为这个着法的结果，Beta值不变。

下面给出伪代码，其中星号(***)指示了在负值最大搜索伪代码上的改动。

```
int AlphaBeta(int depth, int alpha, int beta) { // ***
  if (depth == 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -AlphaBeta(depth - 1, -beta, -alpha); // ***
    UnmakeMove();
    if (val >= beta) { // ***
      return beta; // ***
    } // ***
    if (val > alpha) {
      alpha = val;
    }
  }
  return alpha;
}
```

与调用Negamax函数不同的是，调用AlphaBeta函数时需要两个额外的参数，前面的内容告诉我们这两个额外的参数是-∞与+∞，例如我们要用以上函数完成一个6层的搜索，那么我们应该这样调用这个函数：

```
val = AlphaBeta(6, -INFINITY, INFINITY);
```

这样val就可获得由6层搜索得到的对当前局面的局面评分。

我们称上述代码描述的搜索为Alpha-Beta搜索。以上代码是所有基于Alpha-Beta剪枝的搜索的主框架。

显然在相同层数的搜索中，Alpha-Beta搜索遍历的结点数不会超过极大极小搜索，严格的数学分析可以证明在最好情况下Alpha-Beta搜索遍历的结点数只有极大极小搜索遍历结点数的平方根那么多。实际运行中，Alpha-Beta搜索的效率极大地依赖搜索结点的顺序，最坏情况下，如果你总是先去搜索最坏的着法，那么Beta截断就不会发生，Alpha-Beta搜索就如同极大极小搜索一样，效率非常低。因此在实际应用中，我们会采用一系列技术优化Alpha-Beta搜索的搜索顺序，由于这部分问题不是本文讨论的主题，这里不做进一步介绍。

6 窗口（Window）

很多对Alpha-Beta搜索的进一步优化的技术，如许多裁剪方法，期望窗口，主要变例搜索，MTD(f)等均涉及到一个共同的概念——窗口。简单地说，窗口就是指一对Alpha与Beta值所表示的一个区间，这个区间代表了局面评分可能的范围。为了记号上的方便，我们通常用一个二元组(Alpha, Beta)来表示一个窗口。

窗口这个概念有什么用呢？下面给出一个例子。对于某个给定的局面执行“val= AlphaBeta(6, 1, 4);”并假设执行后变量val的值为3，注意到1¡3¡4，通过对AlphaBeta函数的定义进行分析后不难推知，对相同的局面执行“val = AlphaBeta(6, -INFINITY, INFINITY);”后变量val的值也必定为3。我们可以推广观察得到如下一般性的结论：

假设我们用窗口(Alpha, Beta)搜索获得了评分val。(1)如果Alpha ¡ val ¡ Beta，那么该局面的局面评分（即用窗口(-INFINITY,INFINITY)搜索获得的评分）就是val。(2)如果val ≤ Alpha，那么该局面的局面评分必然小于等于Alpha。(3)如果val ≥ Beta，那么该局面的局面评分必然大于等于Beta。

对于同一个局面的两个不同窗口的搜索(Alpha1,Beta1)与(Alpha2,Beta2)，如果Alpha1 ≤ Alpha2 ≤ Beta2 ≤ Beta1，那么使用较小的窗口(Alpha2,Beta2)进行搜索相比较大的窗口(Alpha1,Beta1)更容易发生Beta截断，从而小窗口的搜索遍历的结点数小于等于大窗口的搜索遍历的结点数。

尽管由前面的分析用小窗口获得的信息不会超过大窗口获得的信息，但是同时我们知道小窗口的搜索代价也不会超过大窗口。这给了我们一些启发：是否可以通过合理的运用窗口，来优化Alpha-Beta搜索呢？

7 主要变例搜索（Principle Variation Search）

本节讨论主要变例搜索是对Alpha-Beta搜索众多改进中最优秀的之一。在Alpha-Beta剪枝一节我们提到过搜索顺序对Alpha-Beta搜索效率的影响是巨大的，而主要变例搜索对搜索顺序更为敏感，因为主要变例搜索的所做的假设就是着法排序是优秀的，好的着法排得靠前。因此我们在讨论主要变例搜索前，先假定我们已经在优化搜索顺序上做了一些的努力，获得了比较优秀着法顺序的排序。

我们将Alpha-Beta搜索中的结点分为以下三类，并且任何结点都只属于这三类之一：

1. Alpha结点。每个着法搜索都会得到一个小于或等于Alpha的值。
2. Beta结点。至少一个着法会返回大于或等于Beta的值。
3. 主要变例结点(PV结点)。有一个或多个着法会返回大于或等于Alpha的值(即PV着法)，但是没有着法会返回大于或等于Beta的值。

主要变例搜索的思想是做出如下假设：当前搜索结点的着法排序足够好，以至于如果发现当前搜索结点的第一个着法的是PV着法，那么当前搜索的结点是PV结点，且剩余的着法均不比这一着法好。当然这一假设在实际中不必（也很难）被满足，只需要在多数情况下成立就可以（即便假设在所有情况下都不成立，算法的正确性也可以得到保证，只是此时算法的效率是低下的）。

于是，主要变例搜索在当前搜索结点中如果发现一个PV着法，那么对剩余着法所做的是证明它们都不如所发现的那个PV着法。证明一个着法足够坏相比计算一个着法的准确值的代价要小很多，而做出这类证明的方法就是利用窗口的技巧。当然，如果主要变例搜索发现无法给出证明（即对于当前局面它的假设是错的，在剩余着法中存在比那个PV着法更好的着法），那么它会对这个更好的着法重新进行搜索，这次搜索改为计算其准确的值，而不再是证明其足够坏。当然，在这种情况下，相比Alpha-Beta搜索，我们就会浪费先前对“这个更好的着法是足够坏的”所用的证明时间。

下面给出伪代码，其中星号(***)指示了在Alpha-Beta搜索上的改动。

```
int AlphaBeta(int depth, int alpha, int beta) {
  BOOL fFoundPv = FALSE; // ***
  if (depth == 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    if (fFoundPv) { // ***
      val = -AlphaBeta(depth - 1, -alpha - 1, -alpha); // ***
      if ((val > alpha) && (val < beta)) { // 检查失败 ***
        val = -AlphaBeta(depth - 1, -beta, -alpha);  // ***
      } // ***
    } else // ***
      val = -AlphaBeta(depth - 1, -beta, -alpha);
    }
    UnmakeMove();
```

```
    if (val >= beta) {
      return beta;
    }
    if (val > alpha) {
      alpha = val;
      fFoundPv = TRUE; // ***
    }
  }
  return alpha;
}
```

算法的核心部分就是函数中间星号(***)if块中的内容。如果没有找到PV结点，“AlphaBeta()”函数就正常调用，如果找到了一个，那么情况就变了。不是用常规的窗口(Alpha, Beta)求着法的准确值，而是用(Alpha, Alpha+1)来证明它足够坏。如果搜索返回小于或等于Alpha的值，便说明我们的假设是正确的，否则，即若返回值是Alpha+1或更高，那么搜索必须用常规窗口重做。

8 总结

从极大极小搜索开始，通过一步步改进我们最终获得了主要变例搜索，尽管中间涉及到了不少的概念、分析与思考，然而最终我们获得的主要变例搜索是一段优雅的短代码，它被用在了成百上千的弈棋程序中，成为这些程序的核心驱动。实际上，在人际对弈领域中，有很多类似的优美的结果，值得去品味与欣赏。

References

[1] www.wikipedia.org.

[2] www.xqbase.com.