人工智能 - 搜索

杜小勤

武汉纺织大学数学与计算机学院

2016/03/06

Game tree 搜索

- Minimax 搜索；
- $\alpha - \beta$ 搜索；
- Mento Carlo 树搜索；

Game tree

Game tree 是一个有向图，它的节点是游戏中的状态（*position*，例如，棋局），边表示状态之间的移动（*move*，例如，落子）。

Complete Game tree V.S. Partial Game tree

前者是一棵完全树，从初始状态出发，包含所有的状态和移动。

后者是一棵不完全（部分）树，从当前状态出发，搜索到一定的深度（depth 或 plies）。

Game tree 示例：Tic-Tac-Toe

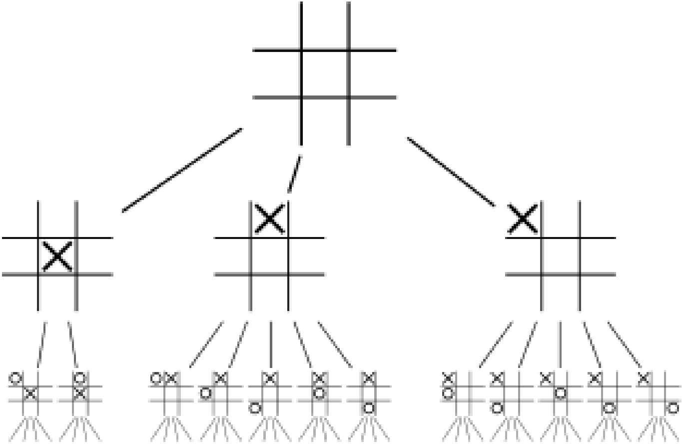

图 2-1: Game tree 的前 2 个 plies

能否执行完全的搜索？

对于大多数棋类游戏，执行完全的搜索是不可能的。

类似 Tic-Tac-Toe 的棋类游戏，可以执行完全的搜索。

但是，类似 Chess 和围棋（Go）等棋类游戏，不可能执行完全的搜索。

可行策略

可行的策略是：

- 使用 Minimax 和 $\alpha-\beta$ 等算法搜索到一定的深度，并使用启发函数对状态进行评估；
- 当很难获取到高质量的启发函数时，不使用启发函数，而是使用 MCTS 技术，例如最近几年兴起的 MCTS Go；

仅仅是执行一个目标搜索吗？

由于棋类游戏的性质，它不单纯是一个目标搜索（当把获胜棋局状态看作是一个目标时）。

原因：还要考虑对手的选择。

Minimax 搜索：背后的原理

原理：对弈的双方总是考虑能够导致己方获胜的落子，这被称为“己方利益最大化，对方利益最小化”原则。

为便于分析，一方称为 MAX 方，另一方称为 MIN 方。如果不作特别说明，算法被认为是 MAX 方。

在理想对手的假设下，MIN 方总是最小化 MAX 方的最大化努力。因此，搜索被称为是 Minimax 搜索。

Minimax 搜索过程

在理想的情况下，假设 Game tree 上的搜索能够到达棋局终了的状态，即能够到达决出胜负平的状态（例如 Tic-Tac-Toe、Chess、围棋等棋类），或者也能够给出一个分值的状态（例如 Backgammon、Poker 等），那么算法可以执行如下过程：

从叶子节点开始，将评估值向上返回，如果它的父节点是 MAX 方，则取所有子节点的 *max* 值；如果它的父节点是 MIN 方，则取所有子节点的 *min* 值。

Minimax 搜索过程

上述过程一直进行到根节点为止。

最终，Game tree 中的每一个节点都有一个值，从根节点到叶子节点将形成一条路径，它对应着 MAX 方和 MIN 方的落子选择。

最优策略：能够导致 MAX 方（或 MIN 方）获胜的落子序列。

Minimax 搜索过程示例

```
           ---------------A---------------          max
          /               |               \
         B                C                D        min
       / | \             / \              / \
      E  F  G           H   I            J   K      max
     / \ / \ / \       / \  / \         / \  / \
     L M N O P Q       R S  T U         V W  X Y    min
     7 6 8 5 2 3       0 -2 6 2         5 8  9 2
```

图 2-2: Minimax 搜索过程示例

状态评估函数 V.S. 启发评估函数

状态评估函数：在棋局终了时，能够给出明确的值，确定胜负平结果。

启发评估函数：又被称为静态评估函数，不需要进一步搜索落子，不需要等到棋局终了，使用某个公式来评估中间棋局状态的好坏。

Chess 的启发评估函数

- c_1 * material + c_2 * mobility + c_3 * king safety + c_4 * center control + ...

Such as

- f(P) = 9(Q-Q') + 5(R-R') + 3(B-B'+N-N') + (P-P') - 0.5(D-D'+S-S'+I-I') + 0.1(M-M') + ...

in which:

- Q, R, B, N, P are the number of white queens, rooks, bishops, knights and pawns on the board.
- D, S, I are doubled, backward and isolated white pawns.
- M represents white mobility (measured, say, as the number of legal moves available to White).

图 2-3: chess 的启发评估函数

围棋的启发评估函数

不可能象 Chess 那样，为每个棋子赋予一个确定的值，因为围棋棋子的力量取决于位置、周围状况等因素。要综合考虑棋子串的影响力、死活、实地等因素。

很难给出一个好的围棋启发函数，目前还没有一个得到公认的好的启发函数。

Deep Blue

Chess：状态空间非常大，在搜索的过程中，不可能使用“状态评估函数”，只能使用“启发评估函数”。

Deep Blue：击败世界冠军 Garry Kasparov，至少可以搜索 12 个 plies 的深度，并使用“启发评估函数”评估棋局。

评估值的两种视角

两种评估函数可以采取：

- 从 MAX 方的角度确定：+，对 MAX 有利；-，对 MIN 有利；
- 从双方的角度确定：+，有利棋局；-，不利棋局；

标准算法

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        bestValue := −∞
        for each child of node
            v := minimax(child, depth − 1, FALSE)
            bestValue := max(bestValue, v)
        return bestValue

    else    (* minimizing player *)
        bestValue := +∞
        for each child of node
            v := minimax(child, depth − 1, TRUE)
            bestValue := min(bestValue, v)
        return bestValue
```

图 2-4: 标准算法——MAX 方的启发函数

标准算法

初始调用：

- MAX 方：$rootMinimaxValue = minimax(origin, depth, TRUE);$
- MIN 方：$rootMinimaxValue = minimax(origin, depth, FALSE);$

注意：$rootMinimaxValue$ 是从 MAX 的角度来看的。

Negamax 算法

标准算法将 MAX 方与 MIN 方分开考虑，启发函数是从 MAX 方的角度来确定的。

Negamax 算法将 MAX 方与 MIN 方统一考虑。一般而言，启发函数可以被认为是从双方的角度来确定的。当然，也可以转换成从 MAX 的角度来确定。

算法原理

利用公式 $max(a, b) = -min(-a, -b)$，将 min 转换成 max，包含 2 个要点：

- “-” 号：需要对返回的值取反；
- 统一使用 max，即只取最大值；

Negamax 算法

```
function negamax(node, depth, color)
    if depth = 0 or node is a terminal node
        return color * the heuristic value of node

    bestValue := −∞
    foreach child of node
        v := −negamax(child, depth − 1, −color)
        bestValue := max( bestValue, v )
    return bestValue
```

图 2-5: Negamax 算法——MAX 方的启发函数

Negamax 算法

注意：

第 03 行，启发函数是从 MAX 方的角度来确定的，而 color * the-heuristic-value-of-node 则是从双方的角度来确定的。$color = 1$，MAX 方；$color = -1$，MIN 方。

第 06 行，返回值符号取反操作。第 07 行，统一使用 max 操作。

Negamax 算法

初始调用：

- MAX 方：
  $rootNegamaxValue := negamax(rootNode, depth, 1)$
  $rootMinimaxValue := rootNegamaxValue$
- MIN 方：
  $rootNegamaxValue := negamax(rootNode, depth, -1)$
  $rootMinimaxValue := -rootNegamaxValue$

Negamax 算法

注意：*rootNegamaxValue*（来源于算法中的 *bestValue*）始终是从双方的角度来看的（即从每个节点自身来看的，这就是为什么 MAX-MIN 交替取反以及要做 color * the-heuristic-value-of-node 的处理）。

而 *rootMinimaxValue* 与标准算法中的一样，是从 MAX 的角度来看的。

Minimax 搜索演示

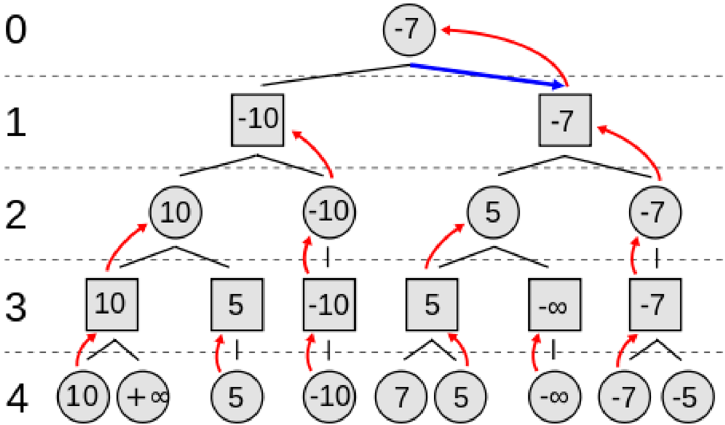

图 2-6: Minimax 搜索演示

Minimax 搜索演示

- 圆圈节点：MAX 方；
- 方形节点：MIN 方；
- 值：从 MAX 方角度来看；
- 红色箭头：返回的最优策略；

结果分析

从图2-6可以看出，MAX 方搜索出的最佳值 *bestValue*：

- 一定来源于所有的叶子节点；
- 既不一定是最好的值 $+\infty$（表示“胜”），也不一定是最差的值 $-\infty$（表示“负”）；
- 是 MAX 方与 MIN 方不断“妥协”的结果；

上述结论对 MIN 方也适用。

如何利用这个特点?

利用双方相互制约的特点，可以优化 Minimax 搜索，这就是 $\alpha - \beta$ 搜索。

如何利用?

一个实例

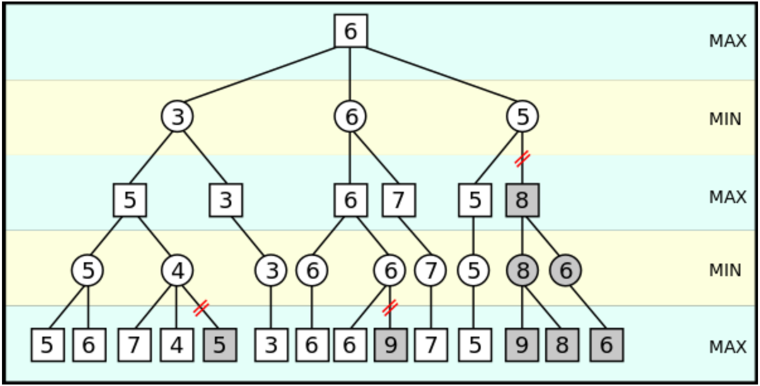

图 2-7: $\alpha - \beta$ 实例

α 和 β 值

在某棵树或子树上，设 MAX 方能够获得的最好值（最大值）是 α，MIN 方能够获得的最好值（最小值）是 β。

在搜索初始时，让 $\alpha = -\infty, \beta = +\infty$。

α 和 β 值

随着搜索的进行，让 α 保存 MAX 方目前已经搜索到的最好值（$\alpha = max(\alpha, v)$），让 β 保存 MIN 方目前已经搜索到的最好值（$\beta = min(\beta, v)$）。

然后，让 α 和 β 按照一定的方式在树的节点中从上往下传递，那么当一个节点中出现 $\alpha \geq \beta$ 时，此时我们可以怎么做？

$\alpha \geq \beta$?

由于 MAX 层与 MIN 层是相邻的，并且它们之间存在相互制约的作用，我们可以利用：

当出现 $\alpha \geq \beta$ 时，节点已无必要继续搜索剩余的子节点了，这样可以节省大量的搜索时间。

$\alpha-\beta$ 搜索

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        v := -∞
        for each child of node
            v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
            α := max(α, v)
            if β ≤ α
                break (* β cut-off *)
        return v
    else
        v := ∞
        for each child of node
            v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
            β := min(β, v)
            if β ≤ α
                break (* α cut-off *)
        return v
```

图 2-8: 标准算法

原因分析 I

当 $\alpha \geq \beta$ 时，为什么可以停止子节点的搜索？

分 2 种情况进行分析：

- 设当前节点为 MAX 方。

  该节点的父节点和子节点都是 MIN 方。它的父节点的最好值是 β，而当前节点的最好值是 α。(MAX 节点的 β 值只能从父节点传递下来，不可能通过子节点传递上来，因为 MAX 节点不更新 β。满足 $\alpha \geq \beta$ 要求的 α 肯定来自 MAX 节点的子节点，否则在当前

原因分析 II

节点的父节点这一层，就满足停止搜索条件，已经可以停止搜索了）

此后，再继续搜索当前节点的剩余子节点，如果当前节点的最好值还将得到改善（$> \alpha$），那么，当前节点的父节点也不会选择以当前节点作为子树的分支，因为这有损它的利益；

而如果当前节点的值没有得到改善（$\leq \alpha$），那么，当前节点的父节点也不会选择以当前节点作为子树的分支，因为选择该子树，也不会改善它的策略；

原因分析 III

- 设当前节点为 MIN 方。

  该节点的父节点和子节点都是 MAX 方。它的父节点的最好值是 α，而当前节点的最好值是 β。（MIN 节点的 α 值只能从父节点传递下来，不可能通过子节点传递上来，因为 MIN 节点不更新 α。满足 $\alpha \geq \beta$ 要求的 β 肯定来自 MIN 节点的子节点，否则在当前节点的父节点这一层，就满足停止搜索条件，已经可以停止搜索了）

原因分析 IV

此后，再继续搜索当前节点的剩余子节点，如果当前节点的最好值还将得到改善（$< \beta$），那么，当前节点的父节点也不会选择以当前节点作为子树的分支，因为这有损它的利益；

而如果当前节点的值没有得到改善（$\geq \beta$），那么，当前节点的父节点也不会选择以当前节点作为子树的分支，因为选择该子树，也不会改善它的策略；

标准算法的实例分析

初始调用：alphabeta(origin, depth, $-\infty$, $+\infty$, TRUE)

打开文件 alpha-beta-pruning-example.doc 分析。

$\alpha-\beta$ 搜索

```
function negamax(node, depth, α, β, color)
    if depth = 0 or node is a terminal node
        return color * the heuristic value of node

    childNodes := GenerateMoves(node)
    childNodes := OrderMoves(childNodes)
    bestValue := −∞
    foreach child in childNodes
        v := −negamax(child, depth − 1, −β, −α, −color)
        bestValue := max( bestValue, v )
        α := max( α, v )
        if α ≥ β
            break
    return bestValue
```

图 2-9: Negamax 算法

$\alpha - \beta$ 的 Negamax 搜索

初始调用：rootNegamaxValue := negamax(rootNode, depth, $-\infty$, $+\infty$, 1)

Negamax 与标准算法是等价的。

α 和 β 的更新方式

标准算法：

α 保存 MAX 节点的最好值，β 保存 MIN 节点的最好值，对于当前节点而言，α 与 β 不可能同时从子节点处得到更新，它们中有一个肯定来自于它的父节点。

$\alpha-\beta$ 的 Negamax 算法正是利用这一点，来达到与标准算法一样的效果。

α 和 β 的更新方式

Negamax 算法：

与 Minimax 的 Negamax 算法一样，$\alpha - \beta$ 的 Negamax 算法也使用 *max* 操作代替 *min* 操作。

该算法总是使用 α 保存下层节点返回的最好值（如果当前节点是 MAX 节点，则对应着标准算法中的 α，否则，如果是 MIN 节点，则对应着标准算法中的 β）。

β 总是来自于父节点。

两个算法的等价性

此外，在递归调用时，α 和 β 这 2 个参数先取反再交换后传递下去。下面先分析当前节点是 MAX 节点时，两者的等价性。

在标准算法中，$\alpha = max(\alpha, v)$，v 是子节点的返回值，并且它是从 MAX 的角度来看的。β 来自于父节点 MIN 节点，当满足条件 $\alpha \geq \beta$ 时停止子节点的搜索。

两个算法的等价性

而 $\alpha-\beta$ Negamax 算法的主体是基本的 Minimax Negamax 算法。因此，对于同一个搜索的这个 MAX 节点而言，它也将能够搜索到与标准算法一样的最好值，即标准算法中的 α 值，而它的另一个值则来自于父节点 MIN 节点的最好值的取反，刚好就是标准算法中的 β 值。显然，关系 $\alpha \geq \beta$ 依然成立。

两个算法的等价性

如果当前节点是 MIN 节点，那么，同样该节点也能够搜索到与标准算法一样的最好值，但是要取反，即 $-\beta$，而另一个值则来自父节点 MAX 节点的最好值的取反，即 $-\alpha$。显然，关系 $-\beta \geq -\alpha$ 等价于 $\alpha \geq \beta$。

不论何种情况，两者是等价的。

Monte Carlo tree search

基本的 Monte Carlo tree search（MCTS）方法不使用启发评估函数来评估棋局的好坏，而是通过大量的随机模拟（每一次模拟被称为 playout，双方随机选取落子，一直进行到能够决出“胜负平”为止），来确定落子的优劣。

它有效地避免了“不易获得好的启发函数”的问题，尤其是在计算机围棋领域。例如，自 2006 年出现的 MCTS Go 技术。

MCTS 的四个阶段

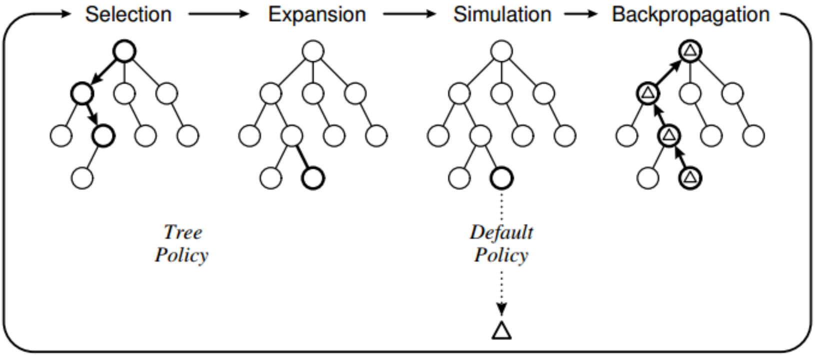

图 2-10: MCTS 的执行步骤

Selection 策略

如何从根节点开始选择子节点?

存在着“Exploitation-Exploration”的平衡问题：是选择那些胜率高的子节点，还是选择那些较少被访问的子节点（可能是潜在的胜率高的子节点)?

UCT 的 Selection 策略

UCT = MCTS + UCB

使用 UCB1 公式选择孩子节点：

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}} \tag{1}$$

其中，w_i 表示第 i 个孩子节点获胜次数，n_i 表示第 i 个孩子节点被访问的次数，$t = \sum_i n_i$ 表示当前父节点的被访问次数（它的所有孩子节点的被访问次数之和），c 是 Exploration 参数，理论值是 $\sqrt{2}$，实际应用中根据经验选择。

UCT 的 Selection 策略

公式的第 1 项是 Exploitation 项，第 2 项是 Exploration 项。

它们用于解决 Exploitation-Exploration 问题。

UCT 的 Selection 阶段将选择 UCB1 最高的孩子节点。

Selection 阶段

从根节点 *rootnode* 开始，沿着子树使用 UCB1 值选择孩子节点。

这个过程继续下去的条件是——当前节点的下层节点都被访问过（意味着都有 UCB1 值，当前节点被称为是“完全扩展的”），且当前节点不是棋局终止状态（意味着还要继续扩展）。

Expansion 阶段

Selection 阶段结束后，停留在某个节点，随机选取其中一个子节点进行扩展。

Simulation 阶段

这个过程模拟随机下棋，一直进行到棋局终了为止。

Backpropagation 阶段

从被扩展的子节点开始回溯，一直进行到根节点为止——更新节点的被访问次数和胜局次数。

参考文献

[1] Wikipedia: Game tree.
[2] Wikipedia: Minimax.
[3] Stanford: Minimax Algorithm.
[4] Minimax Search with Alpha-Beta Pruning.
[5] Wikipedia: Negamax.
[6] Evaluation Function.
[7] Wikipedia: Monte Carlo tree search.
[8] UCT Python Code.