第3章

传统的棋类智能

与其他棋类游戏相比，计算机模拟的围棋人工智能棋力进展相对缓慢。在1997年，就有计算机可以击败世界国际象棋棋王卡斯帕罗夫，但围棋软件直到2016年才第一次真正地击败顶尖围棋棋手。国际象棋目标明确，只要杀死国王即可（中国象棋、日本的将棋状况也差不多），因此算法较为简单。但是围棋以“围”为目标，不一定需要杀死对方棋子，每一步有数百种以上的走法，因此围棋的复杂度高，又极具欺骗性，这些因素都对计算机博弈程序提出了巨大的挑战。

虽然像IBM公司“深蓝”那样的超级计算机已经能够击败世界上最好的国际象棋棋手，但在AlphaGo出现之前却有不少人能轻易击败当时的围棋软件。黄山谷有诗：“心似蛛丝游碧落，身如蜩甲化枯枝”，可见围棋给程序员们带来了许多人工智能领域里的挑战，要编写出超越初级水平的计算机围棋程序都是一件极其困难的事。

过去的传统方法一直没有能够在围棋上有所突破，但是在象棋等其他棋类上还是展现出了其强大的算法实力，了解这些内容，一定是有益的。现代基于神经网络的人工智能方法需要巨大的计算机算能，当人们没有足够的算能来训练出一个强大的棋类智能体时，尝试利用传统方法不失为一种变通之法。这就有点像牛顿力学与爱因斯坦相对论之间的关系，大部分时间使用牛顿力学就好了。传统方法的一个致命问题是算法中带有太多人类的主观想法，在谈论细节时，读者大概会有更实际的感受。


视频讲解

3.1 极小化极大算法

极小化极大（Minimax）算法常用于棋类等由两方较量的游戏和程序，它可以追溯到中世纪，属于一种找出失败的最大可能性中的最小值的算法。算法的基本思想是假设游戏的获利与损失总是零总和的，从当前参与博弈的角色角度来看，该角色总是在可选的选项中挑选将自己的优势最大化的选项，而对手方总是选择令自己优势最小化的方法。很多棋类游戏可以采取此算法，例如井字棋。对于交替博弈的双方而言，如果站在各自的角度来看，当前博弈方总是挑选使得自己优势能够最大化的选项。两种看上去矛盾的描述和解释，仅仅是因为所选择的当前博弈对象不同而已。

极小化极大算法，顾名思义，就是让最大的情况最小，这里的最大一般是指最差的情况，例如，游戏中最不利的情况。换个角度来理解，就是说下棋时己方要时刻保证最小化对手方的最大收益。这个算法的前提是游戏必须要满足零和博弈的条件，即两个玩家进行博弈，如

果其中一方得到利益那么另一方就会失去利益，游戏利益的总和为零或者是某一个常数，如果不满足这个条件，算法就将失去意义。本质上极小化极大算法就是一个树状结构的递归算法，每个节点的子节点和父节点都是对方玩家，所有的节点被分为两类，分别是己方的极大值节点和对方的极小值节点。

极小化极大算法一般都会通过递归来实现，主要原因是递归方法在代码编写上可以做到简单明了。如果读者觉得理解递归逻辑在思维上有困难，也可以通过展开博弈树来实现算法，但是后者在代码复杂度上要远远超过前者。这里代码的讲解也是基于递归算法的。前面提到过算法本质上是“己方要时刻保证最小化对手方的最大收益”，属于递归的思想，也就是说，判断己方收益的方法是基于对方的收益。例如，要计算 A 的收益，就要先计算 B 的收益，要计算 B 的收益，就要再计算 A 的收益，往复循环直到满足循环的结束条件，这就构成了递归的基础。这里以井字棋为例来实现一个会玩井字棋的智能程序。

井字棋是一种在三排棋盘上进行的连珠游戏，它需要两个玩家进行游戏，玩家双方轮流在 3×3 的格上打上自己的符号，一个在格子上打圈(○)，一个在格子上打叉(×)，最先在横、直或斜其中任一条件下将自己的符号连成一线的玩家为胜。井字棋游戏只有 765 个可能局面，26 830 个棋局。如果将对称的棋局视作不同局，则它有 255 168 个棋局。

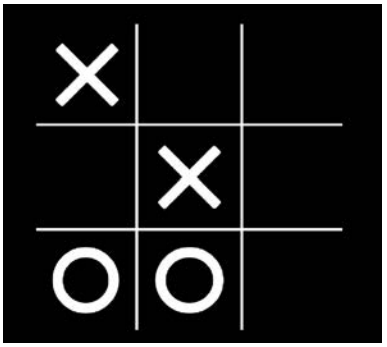

图 3-1　进行中的井字棋游戏

假设当前井字棋游戏下到如图 3-1 所示的状态，现在轮到画×方落棋，显然下到右下角就能赢取胜利。要计算机程序利用极小化极大算法进行井字棋游戏，首先就需要为这个智能体程序定义一个知道怎么找到在下一步获胜方法的函数。代码片段 3-1 演示了如何查找必胜走法的逻辑。

【代码片段 3-1】 查找能走向必胜的下一步。

```
def findOneMoveWin(game_state,player):
    possible_moves = []
    for candidate_move in game_state.legal_moves(player):           #1
        next_game_state = game_state.apply_move(candidate_move)     #2
        if next_game_state.is_over() and next_game_state.winner == player:
            possible_moves.append(candidate_move)                   #3
    return possible_moves                                           #4
```

【说明】

(1) 遍历当前玩家所有的合法选择。

(2) 得到每个选择带来的结果。

(3) 如果当前选择能带来胜利，就保存起来。

(4) 遍历完后返回之前保存的可以赢取棋局的落子选项。

通过以上算法就能轻易看出一个弊端，仅仅是要想知道哪一步一定能取胜，就需要遍历所有可能的选择。这种方法在象棋这种有固定棋子和固定下棋方式的游戏中或许可行，但是在围棋中实在是太困难了，这种算法所带来的计算量将是灾难性的。

但是仅仅知道怎么取胜还是远远不够的。大家可以想想，双方是怎么会下到图 3-1 这

个局面的呢？现在把游戏倒退一步，回到图 3-2 的样子。画×方选择了左上角和中心的位置，画○方现在也占据了左下角。现在轮到画○方走一步，他会很天真地认为如果自己占据了下路的中间位置就能在后一步中连成一线吗？这似乎是很可笑的，因为画×方会比他先连成一线。那么显然当前的智能程序必须要知道在自己取得胜利之前不能给对方任何获胜的机会。具体到井字棋游戏就是要占据对方能够获胜的位置，即如果己方当下不能立即赢取胜利，那么就必须阻止对方在接下去的一步棋中获得胜利。代码片段 3-2 演示了如何定义一个能避免自己失败行为的函数。

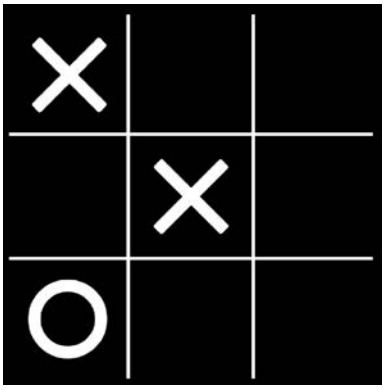

图 3-2　倒退一步棋的情况

【代码片段 3-2】 避免选择导致失败的下一步。

```
def findMinLoseMoves(game_state,player):
    opponent = player.other()                                          #1
    possible_moves = []
    for candidate_move in game_state.legal_moves(player):              #2
        next_state = game_state.apply_move(candidate_move)             #3
        opponent_winning_move = findOneMoveWin(next_state, opponent)   #4
        if opponent_winning_move is None:
            possible_moves.append(candidate_move)                      #5
    return possible_moves
```

【说明】

(1) 初始化己方的对手。

(2) 遍历自己所有的合法选择。

(3) 计算执行该选择后留给对方的新局面。

(4) 对方能否在这个新局面获胜。

(5) 如果不能，加入己方可选项的集合内。

程序在调用 findMinLoseMoves()函数前需要先调用 findOneMoveWin()，只有知道自己不会在下一步中取胜才去思考阻止对方胜利的选择，因为一旦胜利了，游戏也就结束了。己方既然能够找到防止对方胜利的选择，相反地，对方也会使用这种相同的策略，所以无论如何选择，对方总有应对之策。按这个逻辑，在零和游戏中单纯模拟寻找胜利的选择是没有意义的，因为算法总是能保证在对方选择必胜的行动前就已经扼杀了这个选项。

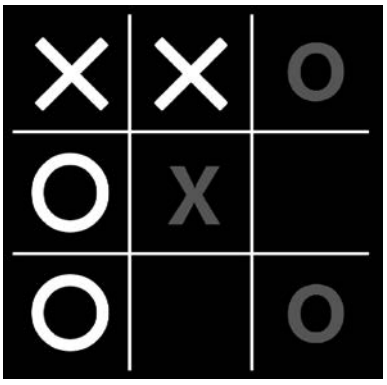

图 3-3　评估后两步可能的落子

如果双方在各自的当前回合都无法必胜，就要继续搜寻下去，看看是不是存在对方至少要连下两步棋才可以阻止己方胜利的情形。如图 3-3 所示，如果画×方落棋到中心红色×点，那么执○方必须同

时堵住两个红色的画圈位置才可以避免执×方的胜利。根据这个想法，代码片段 3-3 演示如何实现一个继续查找下一步必胜着法的 findTwoMovesWin()函数，在调用该函数之后，再对它返回的可选项进行筛选。

【代码片段 3-3】 查找两步内必胜的下法。

```
def findTwoMovesWin(game_state, player):
    opponent = player.other()
    for candidate_move in game_state.legal_moves(player):
        next_state = game_state.apply_move(candidate_move)
        good_responses = findMinLoseMoves(next_state, opponent)     #1
        if not good_responses:                                      #2
            return candidate_move
    return None
```

【说明】

(1) 这一步是整个函数的关键，逻辑上却很简单。如果己方当前选择使得对方在下一步无法阻止己方在下下一步获胜，那么当前一步棋必然是制胜棋。

(2) 如果一方的程序能找到一步必胜棋，逻辑上对方应该已经在上一步行棋时占据了这个位置，或者提前扼杀了这种可能性。所以如果可以穷举，依照算法的逻辑，任何一局棋从一开始就已经注定了谁胜谁负。不仅是井字棋，几乎任何棋类，只要先下的一方如果不出现失误，这一局就起码是和局，也就是在一开始提到的零和。

再回顾上面这个算法中智能体考虑的完整思路。

① 当前步能不能立即胜利，如果能就执行。

② 能不能阻止对方在下一步胜利，如果不能就认输。

③ 对方下完后能不能找到自己在后续一步中立即胜利，如果能就执行；查看对方是不是在下一步存在必胜的下法，如果不能就认输。

④ 能不能找到自己在下一步中必胜的下法，如果能就执行。

⑤ 查看对方是不是在下一步存在必胜的下法，如果不能就认输。

……

从上述算法可以看出，整个算法既简单又粗暴，计算过程试图穷尽所有可能的选择，如果己方下一步不能胜利，就尝试阻止对方在下一步胜利，之后再考虑己方在下下一步能否胜利，如果不能则尝试阻止对方在下下一步的胜利，以这种逻辑不断地模拟行棋的过程，只要己方在落子时不能胜利，就将子落在能阻止对方胜利的位置，直到达到棋局终止的条件。即便是像井字棋这种非常简单，仅有 9 个落子位的棋类游戏，第一步行棋要搜索的空间也达到 94 981 步，虽然对现代计算机而言这算不上是很深的深度，但如果是象棋游戏，这个数字就会变得非常可怕。以现代计算机的处理能力，这种贪婪算法所耗费的时间是不可接受的(计算一步棋大概要算上好几年)。假使换作围棋，这个算法就会更加耗时冗长，几乎会永无止境地运算下去。

虽然极小化极大算法在效率上不太理想，但是对于井字棋，它已经足够好了。图 3-4 演示了单步落子时采用极小化极大算法评估落子选项的顺序思维过程，之前的代码片段也是按照这个顺序逻辑来写的，但是简单地仿照这个逻辑就必须手工编写每一步的判断函数，那这个函数写起来就无穷无尽了。如果只考虑未来的有限步，这种方式勉强还是可行的，不过

在一开始也说了，实现这种算法最好是使用递归这种编程技巧，因为递归过程并不需要指定探索深度，只需要计算机内存足够，运行时就不会报错。在介绍这个小技巧前，先用代码片段 3-4～代码片段 3-6 来定义几个游戏的枚举类和一些框架性的类方法，这和在第 2 章所做的工作类似，定义枚举类仅仅是为了使得代码更加可读，同时也为了编程时方便对变量进行记忆。

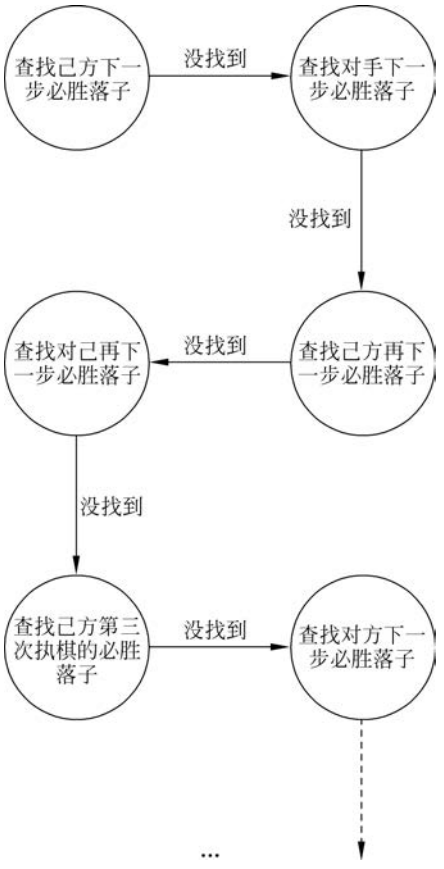

图 3-4　极小化极大算法评估落子选项的顺序思维过程

【代码片段 3-4】 定义枚举类等。

```
MyGo\tic - tac - toe\main.py
class GameResult(enum.Enum):      #1
    loss = 1
    draw = 2
    win = 3
class GameState(enum.Enum):       #2
    waiting = 0
    running = 1
    over = 2
player_x = 1  #3
player_o = - 1
```

【说明】

（1）井字棋的结局存在胜、负与和棋 3 种结果。为了方便后面对棋局优劣的判断，对枚

举值的赋值按输到赢，数值安排由小到大，这样安排的便利性将在后面实际编程中体现。

（2）定义游戏的状态，主要是为了方便智能体知道游戏什么时候结束。

（3）为了编程方便，没有为执棋的双方再定义一个枚举类，而是直接为其赋了变量值，这在实践上并不是一个好的习惯，但是目前在学习演示中这不会是一个问题。

和大部分棋类一样，可以把下棋这件事拆分成 3 个主要的对象类。棋盘和下棋的人是显而易见的实体，而棋局的状态属于一个比较抽象的概念，简单来说，将它实例化后就可以代表一局对弈。现实生活中如果有了棋盘和下棋的人，那么剩下的事情就是下棋了。人们会下第一局、第二局、第三局，直到疲倦，而其中每一个具体的棋局在程序中就是棋局状态类的一个实例。可以仿照第 2 章再额外定义一个裁判类，不过由于井字棋规则简单，这里就不再额外定义这个类了，仅是把裁判的功能放在棋局的状态类里实现。代码片段 3-5 实现了一个抽象后的棋盘类。

【代码片段 3-5】 抽象后的棋盘类。

```
MyGo\tic - tac - toe\ttt.py
class Board:
    def __init__(self):                #1
        self.board = np.zeros((3,3))
    def printBoard(self):              #2
        for i in range(3):
            line = '|'
            ele = []
            for j in range(3):
                if self.board[i,j] == -1:
                    ele.append('_○_')
                elif self.board[i,j] == 1:
                    ele.append('_×_')
                else:
                    ele.append('___')
            print(line.join(ele))      #3
```

【说明】

（1）将井字棋的棋盘初始化为一个 3×3 的全零 NumPy 数组，用 1 表示画×，－1 表示画○。

（2）定义一个打印棋盘的方法，方便人机交互。

（3）按行来打印棋盘。

面向对象的编程有个特点，每个人都可以对要实现的事情有各自不同的认识与抽象结果。可以选择把 Board 类抽象得很简单，就是一个棋盘，它要做的就是能通过打印方法来查看自己，这也与实际生活相符。如果读者愿意，也可以给它加上落子的方法来更新 Board.board 的盘面，但是本书打算把更新 Board.board 这件事交给别的类来做。

和智能算法无关的事情，都可以交给记录棋局的状态类来完成，接着再来看看这个类需要做些什么。

【代码片段 3-6】 游戏棋局类的常用方法。

```
MyGo\tic - tac - toe\ttt.py
class Game:
    def __init__(self, board, player = player_x,game_state = GameState.waiting):
```

```
        self.board = board                                          #1
        self.player = player                                        #2
        self.winner = None
        self.state = game_state
        self.bot1 = None
        self.bot2 = None
    def getResult(self):                                            #3
        ...
    def run(self,mode = 'hvh',bot1_mode = 'r',bot2_mode = 'r'):     #4
        ...
    def applyMove(self,move):                                       #5
        ...
    def simuApplyMove(self,move):                                   #6
        ...
    def isLegalMove(self,move):                                     #7
        ...
    def getLegalMoves(self):                                        #8
        ...
```

【说明】

（1）把棋局和棋盘挂钩，表示某一局棋下在哪副棋盘上。

（2）记录当前回合属于哪一方。

（3）判断当前棋局状态，这个方法将返回之前定义的枚举类 GameState 的值。

（4）用 run()方法启动游戏，执棋双方在这个方法中轮流下棋，其中，mode 用来提示下棋模式，游戏可以支持人与人的对战、人与智能体的对战以及智能体和智能体之间的对战。

（5）applyMove()用来更新棋盘，读者也可将这个方法放在 Board 类中。

（6）simuApplyMove()做的事情和 applyMove()类似，只不过一个是真的在棋盘上落子，一个只是模拟智能体在思考时的虚拟落子。simuApplyMove()方法会额外返回一个 Game 类的实例，因为不能在原棋盘上更新思考时假想的落子，这一点和人类在下棋时的行为是类似的。

（7）判断落子是否合法。井字棋游戏不允许在已经下过的位置再下棋，一般也可以为这种游戏规则的校验再额外设置一个裁判类，使得逻辑抽象和分工更加明确。

（8）这个类返回当前棋局状态下所有合法的选择，这个方法也可以放在 Agent 类中实现，但是为了保持风格一致，这里选择了尽量简化另外两个类。

上面这些方法都和本章的智能主题不太相关，所以具体的实现可以查看 MyGo 的源代码，源代码里有更详细的注解，这里就不再赘述了。

和棋盘类一样，为了不把棋手抽象得过于复杂，代码片段 3-7 中的智能体程序只需要能根据棋盘的当前状态给出下一步出棋就行了。

【代码片段 3-7】 井字棋的智能体类。

```
MyGo\tic - tac - toe\ttt.py
class Agent:
    def __init__(self,game,player,mode = 'r'):
        self.game = game                    #1
        self.player = player                #2
        self.mode = mode                    #3
```

```
    def chooseMove(self):
        if self.mode == 'r':                    #4
            moves = self.game.getLegalMoves()
            return random.choice(moves)
        if self.mode == 'ai':                   #4
            ...
```

【说明】

(1) 把智能体和具体的游戏实例挂钩,相当于告诉智能体,它在下哪盘棋。

(2) 给智能体分配角色,告诉它是画×方还是执○方。

(3) 智能体可以有很多不同的智能算法,程序用 mode 参数来告诉它应该使用哪种算法。

(4) 定义出棋的方法,暂时程序先提供两种方法,'r'表示随机落子法,'ai'表示采用极小化极大算法。

随机方法的棋力非常弱,它从所有合法的选项中随机挑选出一个选择,专门实现随机方法这件事的目的是给后面的极小化极大算法提供一个参考对手,后面会看到贪婪算法与随机算法在棋力方面的差距。

下面着重看一下极小化极大算法在代码片段 3-8 和代码片段 3-9 中是如何实现的。

【代码片段 3-8】 极小化极大算法的外围框架。

```
MyGo\tic-tac-toe\ttt.py
if self.mode == 'ai':
    moves = self.game.getLegalMoves()                                    #1
    win_moves = []                                                       #2
    loss_moves = []                                                      #2
    draw_moves = []                                                      #2
    for move in moves:
        new_game = self.game.simuApplyMove(move)                         #3
        op_best_outcome = bestResultForOP(new_game)                      #4
        my_best_outcome = reverse_bestResultForOP(op_best_outcome)       #5
        if my_best_outcome == GameResult.win:
            win_moves.append(move)
        elif my_best_outcome == GameResult.loss:
            loss_moves.append(move)
        else:
            draw_moves.append(move)
    if win_moves:
        return random.choice(win_moves)
    elif draw_moves:
        return random.choice(draw_moves)
    else:
        return random.choice(loss_moves)
```

【说明】

(1) 仅考虑所有符合游戏规则的落子选项。

(2) 设置变量存放搜索出的必胜步、和局步和必输步。

(3) 模拟当前选项在当前盘面后的效果,之前提过,这个行为就是类似于人类选手在头脑中思考当前走某一步后的可能结果。

（4）这一步是极小化极大算法的核心，由于前一步虚拟落子后接下去是对方的回合，所以调用 bestResultForOP()获取当前选项落子后，对手能得到的最好结果。这个和之前代码片段中 findMinLoseMoves()的 opponent_winning_move = findOneMoveWin(next_state,opponent)有异曲同工之妙。

（5）对手下一步能达到的最好结果的相反面就是己方当前盘面可以取得的最好结果。

之前在编写极小化极大算法时没有站在己方的角度来思考，而是站在了对方的角度来对棋局进行评价。正是这个技巧使得整个算法采用递归来实现变得具有可行性，否则只能采用如图 3-4 所示的顺序逻辑来手工编写每一步落棋判断直到棋局结束。进入 bestResultForOP()内部查看源代码会发现这个函数会调用 bestResultForOP()本身，这也是递归写法的一个典型特征。如果己方要知道当前状态的最好结果就要查看对方在下一步情形下的最好结果，而对手想知道自己的最好结果就又要再看己方下一步能够获得的最好结果，如此往复循环直至游戏结束，即有一个明确的胜负或者和局的结果。

【代码片段 3-9】 通过递归查找对方的最优着法。

```
MyGo\tic-tac-toe\ttt.py
def bestResultForOP(game):
    if game.state == GameState.over:                          #1
        if game.winner == game.player:
            return GameResult.win
        elif game.winner == None:
            return GameResult.draw
        else:
            return GameResult.loss
    best_so_far = GameResult.loss                             #2
    for move in game.getLegalMoves():
        new_game = game.simuApplyMove(move)                   #3
        op_best_outcome = bestResultForOP(new_game)           #4
        my_best_outcome = reverse_bestResultForOP(op_best_outcome)
        if best_so_far.value < my_best_outcome.value:         #5
            best_so_far = my_best_outcome
        if best_so_far == GameResult.win:                     #6
            break
    return best_so_far                                        #7
```

【说明】

（1）如果当前游戏状态已经结束了，则返回游戏的结果，递归的终止条件依赖这个判断。

（2）初始化当前盘面能够获得的最好结果。

（3）模拟当前选项产生的新棋局。这个之前已经在外层的方法中看到过了，显然这是准备开始递归了。游戏的状态 state 在这个方法中更新，这个值控制着 bestResultForOP()停止递归。

（4）递归调用 bestResultForOP()查看对手的最佳结果。

（5）如果当前最佳结果有提升则更新该值。

（6）胜利是棋局的最佳结果，一旦找到了这步棋就可以退出查找了。

（7）返回当前玩家能获取的最佳结果。

3.2　Alpha-Beta 剪枝算法


视频讲解

根据前面的介绍可以发现，极小化极大算法会遍历所有的可能性，但是根据经验可以知道，并不是所有的选项都需要进行深入的考虑，存在着某些明显不利的选项，当出现这种选项时就可以换一种思路进行考虑了。Alpha-Beta 剪枝算法的出现正是为了减少极小化极大算法搜索树的节点数。1997 年 5 月 11 日，击败加里·卡斯帕罗夫的 IBM 公司“深蓝”就采用了这种算法。

以井字棋为例，先来看看在下棋的过程中是否有优化空间。参考图 3-5，当前轮到画○方，如果不在虚线圈上落棋，下一步画×方画在虚圈处，游戏就结束了。当发现这类问题时，再去思考其他 5 个△标注的位置上的落子收益其实是没有意义的，白白浪费了计算资源。

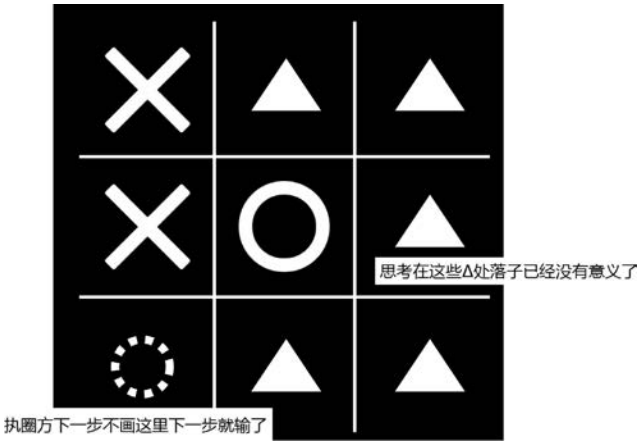

图 3-5　画○方的回合

再来看一个象棋的例子。如图 3-6 所示，此时轮到执“帅”的一方走子。将炮横在中路是一种非常具有杀伤力的下法，后续可能可以配合自己的马走出“马后炮”的杀招。但是如果走了这一步，自己的马将会被对方的车立即吃掉，这一损失实在是太大了，所以面对此局面，实战时基本只会考虑如何走马以避免被车吃掉，其他的走子都不会再深入考虑。

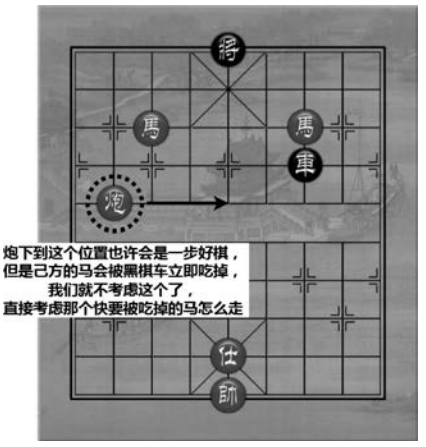

图 3-6　执红方的选择

在行棋的过程中，当发现己方会出现极大损失或者极大获利时，仅考虑这些收益显著的情况而忽略掉其他可选项的行为就是剪枝算法的基本思想，而 Alpha-Beta 剪枝算法就是专门设计用来减少极小化极大算法搜索树节点数的搜索算法。它的基本思想是根据上一层已经得到的当前最优结果，决定目前的搜索是否要继续下去，当算法评估出某策略的后续走法比之前策略的还差时，就会停止计算该策略的后续发展。Alpha-Beta 剪枝算法将搜索时间用在“更有希望”的子分支上，继而提升搜索深度，则同样

时间内搜索深度平均来说可达极小化极大算法的两倍多。

根据算法介绍可知，如果要使用 Alpha-Beta 剪枝算法就会额外需要一套局面价值评估系统来决定哪些搜索分支是有希望的，而哪些是没有希望的。所谓局面价值，就是指当前盘面的胜负概率，胜率越高则价值越大，反之则价值越小，甚至是负价值。各种采用 Alpha-Beta 剪枝算法的人工智能程序之间的实力差距其实就是由于局面价值评估系统的不同所造成的。局面价值评估系统带有很强的主观性，对于如何评估棋局的价值有点像莎士比亚说的，“一千个观众眼中有一千个哈姆雷特”。下面将继续使用井字棋来演示 Alpha-Beta 剪枝算法。为了省去设计井字棋的价值函数，代码片段 3-10 粗暴地认为除了赢和输，其他所有盘面（包括和棋）的价值均为零，赢棋的盘面价值为 1，输棋的盘面价值为－1。如果读者想自己在围棋游戏上尝试一下这个算法，最简单的局面评估算法之一就是计算当前双方在棋盘上剩余棋子的差额。不过实战中很少会有棋手主动提取对方已经穷途末路的棋子，所以也许这种评估方法得到的高价值局面反而会带来更加不利的影响。

【代码片段 3-10】 得到对弈的评估结果。

```
MyGo\tic - tac - toe\ttt.py
def evl_game(game):
    if getResult(game.board.board)[1] != None:                          #1
        if game.player == getResult(game.board.board) == (0,1)[1]:      #2
            return 1            #3
        else:
            return - 1          #3
    else:
        return 0                #3
```

【说明】

（1）判断盘面结果，按照约定，对于井字棋，只有当棋局胜负已分时才对盘面价值进行判断，否则盘面价值为零。

（2）判断当前进行价值评估的棋手是否是棋局的胜利方。

（3）如果胜利方是当前棋手，则盘面价值为 1；如果胜利方是当前棋手的对手，则盘面价值为－1，其他情况的价值按约定默认是 0。

引入了对棋局盘面的价值评估表明在使用 Alpha-Beta 剪枝算法时并不需要执着于搜索时穷尽棋局，即在模拟思考行为时未必非要下到棋局结束时才停止。通常在使用这种算法时会设置一个搜索深度参数来控制算法仿真思考的回合数。从本质上来说，Alpha-Beta 剪枝算法是通过价值评估函数来控制算法的搜索广度，用参数设置来控制算法的搜索深度。

同极小化极大算法相比，Alpha-Beta 剪枝算法并不是要等到棋局下到结束才给出对局面的评估，每个不同可选项得到的评估结果会由价值评估函数给出不同的数值结果，不尽相同的评估结果（极小化极大算法只有胜、负、和三种评估结果）导致 Alpha-Beta 剪枝算法在使用过程中需要记录博弈双方在搜索过程中所能取得的最佳价值，可以把双方记录的最佳价值等价地看作是极小化极大算法中的胜利结果。传统上把一方所能搜索到的当前盘面最佳价值叫作 Alpha，另一方的最佳价值称为 Beta，这种叫法也正是这个算法名称的由来。对于井字棋，将其简记为 best_o 和 best_x。代码片段 3-11 演示了 Alpha-Beta 剪枝算法是如何实现的。

【代码片段 3-11】 Alpha-Beta 减枝算法的代码框架。

```
MyGo\tic - tac - toe\ttt.py
if self.mode == 'ab':                                          #1
    moves = self.game.getLegalMoves()                          #2
    best_moves = []                                            #3
    best_score = None                                          #4
    best_o = minValue                                          #4
    best_x = minValue                                          #4
    for move in moves:                                         #5
        new_game = self.game.simuApplyMove(move)               #6
        op_best_outcome = alpha_beta_prune(new_game, max_depth,best_o,best_x,evl_game)
                                                               #7
        my_best_outcome = -1 * op_best_outcome                 #8
        if (not best_moves) or my_best_outcome > best_score:   #9
            best_moves = [move]                                #10
            best_score = my_best_outcome                       #11
            if self.game.player == player_x:                   #12
                best_x = best_score
            elif self.game.player == player_o:                 #12
                best_o = best_score
        elif my_best_outcome == best_score:                    #13
            best_moves.append(move)                            #13
    return random.choice(best_moves)                           #13
```

【说明】

(1) 模式 ab 代表 Alpha-Beta 剪枝算法。

(2) 获取当前盘面上符合游戏规则的可选项。

(3) 存放最佳的落子选项。

(4) best_score 存放当前盘面在搜索过程中得到过的最高选项价值,这个值在搜索过程中会不断地被更高的值所替换。将执○方和执×方的 Beta 值初始化为最低的价值,并在后面用搜索到的 best_score 值来更新。

(5) 逐个搜索可选项。

(6) 仿真一下当前选项的落子。

(7) 仿真对手在当前落子下能取得的最佳价值。

(8) 己方能取得的最佳价值是对方能取得的最佳价值的反面。

(9) 只对当前价值高于已有记录的落子步进行处理。

(10) 搜索到了更高的价值,于是需要更新最佳落子。

(11) 更新已有记录的最佳落子价值。

(12) 将最佳价值更新给当前棋盘盘面的实际落子方。

(13) 如果搜索到的价值和记录的最高价值一致,则仅补充最佳落子的可选范围,通过随机抽取高价值落子使得下棋过程中棋局更多变,也更贴近人类行为。

代码片段 3-11 和代码片段 3-8 的极小化极大算法在框架上是非常相似的,如果读者仔细思索就会发现,虽然算法的定性描述介绍好像有点玄乎,但是实现上 Alpha-Beta 剪枝算法和极小化极大算法并没有本质上的区别,仅仅是将胜负结果的判断用一个价值判断函数替代了。既然 Alpha-Beta 剪枝算法是对极小化极大算法的优化,它也只能通过递归的方式

来实现。alpha_beta_prune()函数是整个递归方法的核心，读者可以将极小化极大算法中的 bestResultForOP()和这个 alpha_beta_prune()比较着来看。代码片段 3-12 演示了算法的核心递归方法是如何实现的。

【代码片段 3-12】 通过减枝算法查找对方的最优着法。

```
MyGo\tic - tac - toe\ttt.py
max_depth = 4                                          #1
def alpha_beta_prune(game, max_depth,best_o,best_x,evl_fn):
    if game.state == GameState.over:
        if game.winner == game.player:
            return maxValue                            #2
        elif game.winner == None:
            return 0
        else:
            return minValue                            #2
    if max_depth == 0:                                 #3
        return evl_fn(game)                            #3
    best_so_far = minValue                             #4
    for move in game.getLegalMoves():                  #5
        next_game = game.simuApplyMove(move)           #5
        op_best_result = alpha_beta_prune(
            next_game, max_depth - 1,
            best_o, best_x,
            evl_fn)                                    #5
        my_result = -1 * op_best_result                #5
        if my_result > best_so_far:                    #6
            best_so_far = my_result                    #6
        if game.player == player_o:                    #7
            if best_so_far > best_o:                   #8
                best_o = best_so_far                   #8
            outcome_for_x = -1 * best_so_far           #9
            if outcome_for_x < best_x:                 #10
                break
        elif game.player == player_x:
            if best_so_far > best_x:
                best_x = best_so_far
            outcome_for_o = -1 * best_so_far
            if outcome_for_o < best_o:
                break
    return best_so_far                                 #11
```

【说明】

（1）控制搜索深度。由于人为定义平局和进行中的棋局的价值设置为 0，而井字棋一共就 9 步落子，所以当这个搜索深度设置得比较浅时，算法在开头的几步和随机落子并没有什么区别。如果随机落子法在前 3 步完成了横竖相连，就可以击败剪枝算法。这也从侧面说明了一个好的价值评估算法对于剪枝算法的重要性。

（2）由于采用价值评估函数来对胜负的可能性进行评估，这里用一个极大数字或极小数字来表示明确的输赢胜负。

（3）控制搜索深度，如果到达一定深度游戏还没有结束，就用价值评估函数的值来代替胜负的判断。

(4) 和极小化极大算法一样,初始化当前盘面能取得的最佳价值。

(5) 这几步和 bestResultForOP()中的写法是几乎相同的。

(6) 如果结果比之前记录的好则更新最佳价值。极小极大化算法中的最佳价值就是赢棋,所以没有更新最佳价值这一步,而 Alpha-Beta 剪枝中因为是通过价值评价函数来估计胜负结果的,这个值可能会有很多不同的值,所以可能需要不停地更新最大的值。

(7) 根据当前执棋者是谁,将上一步得到的最佳值更新给不同的对象的最佳值。

(8) 如果当前玩家是画○方,当前搜索值大于画○方记录的最大值,则更新其记录的最大值。下面对画×方的判断后也使用了类似的操作步骤,就不再赘述了。

(9) 一方的最佳进行反操作就是另一方的最差。

(10) 如果当前的一方最佳操作可以使得对方的最佳降低,那么就可以认为找到了一步必胜棋,并退出,当然也可以继续搜索不退出,但是由于已经找到了,再多找几个意义不大,反而浪费了计算资源,这个在 bestResultForOP()中也有相似的对应操作。

(11) 返回当前玩家所能取得的最佳结果。

搜索选项时算法会根据棋盘局面上的可落子顺序进行搜索。如果碰巧在一开始就找到了一个最好的选项,在搜索其他后续选项时会由于剩下的选项收益较低而被迅速地剪枝掉,如果运气不好,最好的选项在最后才被搜索到,那么 Alpha-Beta 剪枝算法的速度并不会比极小化极大算法快。但是数学期望上,Alpha-Beta 剪枝算法的消耗时间会是极小化极大算法的一半。如果在搜索开始前引入一些启发性的算法先从最有潜质的着法开始搜索,也许可以缓解算法对着法寻找顺序的依赖并使得算法能有很大的改进。

3.3　棋类局面评估


视频讲解

当把 Alpha-Beta 剪枝算法的搜索深度设置得比较小时就会经常用到评估函数来估计盘面的胜负概率。对于刚刚实现的井字棋游戏来说,由于评估函数非常弱,导致算法效果和随机落子没有什么区别。由此可见,一个好的评估函数对于传统棋类是多么重要。

当博弈游戏比较简单,博弈树较小,可以完全展开时,每个子节点的价值都可以通过胜负结果来确定。对于稍微复杂一些的博弈棋类游戏,通常博弈树都很大,不能够被完全展开,即便采用了剪枝算法缩小了博弈树上的搜索规模,也依旧无法做到在有限的时间内完成全部分支的搜索,所以想让子节点通过胜负结果来确定价值也显得不切实际。通常的做法是限制搜索树的深度,当搜索到达一定深度时,博弈树子节点的胜负概率就需要通过评估函数来估计。“深蓝”使用了超过 12 层深度的搜索,12 层以外使用了静态评估函数。

以象棋为例,由于游戏规则的限制,不同的棋子具有不同的价值。在数学模型上,可以为不同的棋子赋予不同的数值以表示其价值。通常用“子力”这个术语来表示棋盘上所有棋子价值的数值和。信息论的开山鼻祖香农博士曾经对国际象棋的棋子间相互博弈进行过分析,给出了表 3-1 中不同棋子的近似相对价值。

表 3-1　国际象棋的棋子价值

王(K)	后(Q)	车(R)	象(B)	马(N)	兵(P)
200	9	5	3	3	1

根据表中的数值,可以得到棋盘当前子力的公式(3-1):

$$S=200\times(K-K')+9\times(Q-Q')+5\times(R-R')+3\times(B-B')+3\times(N-N')+(P-P') \tag{3-1}$$

其中，K、Q、R、B、N 和 P 表示白方的王、后、车、象、马和兵的个数，相对地，K'、Q'、R'、B'、N'和 P'表示黑方王、后、车、象、马和兵的数目。仿照香农的做法，国人也给出了表 3-2 里中国象棋棋子的近似相对价值。

表 3-2　中国象棋的棋子价值

将帅	车	马	炮	仕	相	兵
∞	1000	450	450	170	160	60

围棋无法仿照象棋那样计算出子力，围棋的每个子本身都是平等的，每个子存在的价值取决于它和其他棋子之间的位置关系，很难通过象棋那种静态的方法对围棋的子力进行分析与评估，但存在一些动态评估的方法。

棋子处于棋盘的不同位置时它的作用会差别很大。例如，中国象棋里，中路车、过河兵与卧槽马都是指占据重要进攻位置的下法，具有很强的威胁性。而窝心马、沉底兵则是指棋子已经处在了不理想的位置，导致棋子本身的价值被削弱，无法发挥出来。所以仅考虑棋子本身的价值显然是不足的。在评估子力的时候，引入棋子的位置来对子力进行动态评估是非常必要的。对于围棋那种每个子都一样的游戏而言，这一点就凸显得尤为重要。

很多棋类游戏中，根据棋子相克的规则，人为地把棋盘划分成多个不同的控制区域，包括本方控制区域、对方控制区域和公共区域。中国象棋在一开盘，就以楚河汉界为分界，将棋盘划分成两个势力范围。随着棋盘双方博弈进程的发展，控制区域也会随着发生变化。落入对方控制势力范围内的棋子将暂时失去价值。围棋本身以控制区域为目的行棋，势力范围的概念则更为明显。很多围棋动态评估的方法就是以控制区域为理论基础的。

象棋在棋盘上的子是灵活机动的，每个子的合法着法数目决定了它的机动性。例如，中国象棋对马和象都有蹩腿的规则，所以它们的合法着法个数会跟着棋局的变化而减少。着法减少，机动性也就变低了。机动性越大，能控制的点就越多，影响就越大，选择有利己方局势的概率也就越大。围棋的棋子虽然落定后不能移动，但是如果棋子周围有比较开阔的空间，或者可以方便地和己方另外的棋子相连，那么己方就可以比较方便地对外扩张，理论上就认为其机动性较高。反之，如果一块棋被对方包围住，其对外延展被限制，就认为其机动性较小，能否存活就要取决于是否可以在有限的空间内做活。

一些对抗游戏的形势可能取决于着法的节拍或者游戏内容的数量关系。例如，星际争霸这类 RTS 对抗类游戏，出招（有效单击鼠标）的频率大大影响局势的发展。又或者是取豆子游戏，谁先拿到最后一个豆子的为胜。这类特征在围棋中展现得不是很明显，但是在一些其他的对抗博弈游戏中可能会需要考虑。

象棋中，如果本方的将帅已经在对方的威胁之下，评估其他棋子的实力意义就已经不大了，因为游戏规则逼迫本方必须响应对方。逼迫对方响应自己的下法是非常具有价值的，它使得对方仅有招架之功，并无还手之力。这种具有威胁性质的着法在围棋中最明显的体现是打劫，在打劫的过程中，双方都有极强的意愿去不断地寻找劫材，以免失去局部利益。

围棋形状的好坏，往往也是围棋博弈胜败的决定性因素。良好的形状有利于己方扩张地盘，有助于做活与连接。为了评估围棋棋形的好坏，人们发明了一些棋形提取算法与评估

法。要使用这种算法，往往需要人工编辑和定义什么是好的棋形，什么是坏的棋形，且工作量巨大。图 3-7 演示了如何用棋形模板去匹配当前盘面并输出一个综合评估得分。

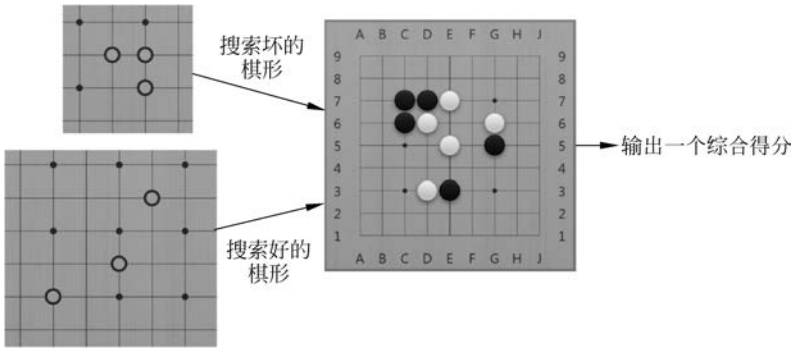

图 3-7　利用棋形模板来评估当前棋盘盘面的胜负概率

一个好的评价系统会综合考虑前面讲到的这些内容，目前看来，象棋游戏已经能够有一个比较好的评估函数来综合考虑以上内容。对于围棋而言，使用评估函数的方法似乎没有取得很好的效果，这可能存在两种原因，一种是算法上对评估函数的设计有问题，一个错误的评估函数当然得到的评分是错误的。围棋需要考虑的因素要比其他棋类更多，仅仅前面谈到的几点还远远不足以给出一个有效的评估。另外一种可能是评估函数对围棋游戏并不起作用，因为围棋总是存在扭转局势的着法。无论真相是哪种现在看起来似乎都已经不重要了，一种利用蒙特卡罗方法进行局面评估的方式比人工编辑特征来得更加有效。

其实围棋的局面评估还有很多别的内容需要考虑，很多细节的东西这里都没有提到，如死活的判断、是否要打劫等。说了这么多，也可以看出传统方法在实现上有多么烦琐，可能这也是传统方法无法击败人类的原因吧。围棋游戏看似简单，实则当人们拿着放大镜去看的时候就完全不是大家以为的那个样子了。

3.4　蒙特卡罗模拟

3.4.1　蒙特卡罗算法


视频讲解

蒙特卡罗方法是科学家冯·诺依曼、斯塔尼斯拉夫·乌拉姆和尼古拉斯·梅特罗波利斯在洛斯阿拉莫斯国家实验室为核武器计划工作时发明的一种以概率统计理论为指导的数值计算方法。它是一种使用随机数来解决计算问题的统计模拟方法。

在很多实际的问题中，人们常常无法得到精确的数值解，在工程中人们可以接受在误差允许范围内的结果。例如，虽然圆的面积公式 $S=\pi \cdot r^2$ 已经众所周知，但是 π 是一个无理数，在处理实际问题时人们总是根据现实情况截取小数点后满足需求的位数。举个例子，小明想要通过共享经济出租自己房屋里的一个圆形淋浴房，淋浴房的半径是 45cm，共享 App 上规定出租价格必须按淋浴房的实际面积来计算，小明的房屋位置靠近 CBD 地区，可以按一平方米 20 元的价格来给出每次出租的标价，那么出租一次他能获得多少收益呢？根据圆形的面积公式，小明的淋浴室占地约 $0.636\,172\,512\,351\,933\,2\text{m}^2$，按 App 里的约定，可以每次以 12.723 450 247 038 663 元出租。人民币的最小单位是分，四舍五入，小明每次出租后可以拿到 12.72 元。以这个价格反向推算出小明的淋浴室面积是 0.636m^2，和实际面积的误差

约为 0.027%。

如果圆的面积公式从未被发现，将淋浴房设计成圆形的小明是否就无法出租自己的淋浴房呢？非常幸运，蒙特卡罗方法可以帮助小明。通过设计某种统计模拟，就可以近似地计算圆的面积。如果采用统计模拟方法计算得到的精度误差也在 0.027%左右，在实际的日常生活中就可以使用这种方法来作为圆形面积的计算方式。

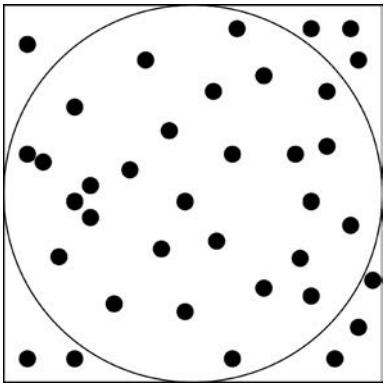

图 3-8　蒙特卡罗算法计算圆的面积

如图 3-8 所示，在利用蒙特卡罗算法计算圆的面积前，先要为半径为 0.45m 的圆外切一个边长为 0.9m 的正方形。现在设计一个随机数生成器，它的作用就是随机采样这个边长为 0.9m 的正方形中的点。使用圆形中包含被采样到的点的数量比上正方形中被采样到的点的总数，就可以估计圆形面积在正方形面积中的占比，从而得到这个圆形的面积。

根据上述算法，代码片段 3-13 实现了采用蒙特卡罗方法来估计圆的面积，建议读者自行调整其中随机数产生的次数，以观察抽取点的数量对最终结果的影响。

【代码片段 3-13】 用蒙特卡罗方法计算圆的面积。

```
MyGo\tic - tac - toe\cycle_area.py
import numpy as np
from numpy import linalg as LA
r = 0.45                              #半径
side_len = 2 * r                      #正方形边长
square_area = side_len ** 2           #正方形面积
class area_machine:
    def __init__(self,times,r):
        self.times = times            #随机生成次数
        self.len = r                  #圆形的半径
    def isInsideCycle(self,gen):
        return LA.norm(gen,axis = 1)< self.len
    def random_generator(self,times):
        return ( - 1 + 2 * np.random.rand(times,2)) * self.len      #在正方形中随机生成点
    def run(self):
        dots = self.random_generator(self.times)
        dots_in_cycle = self.isInsideCycle(dots)                    #查找在圆内的点
        dots_in_cycle_n = np.sum(dots_in_cycle)
        return (dots_in_cycle_n/self.times) * square_area           #返回圆的面积
times = [100,1000,10000,100000,1000000,10000000]                    #请尝试修改这里的数字
np.random.seed(2020)              #设置随机数种子使得结果可以复现
print([area_machine(i,r).run() for i in times])
```

通过小工具的计算，可以得到表 3-3 中半径为 0.45m 的圆面积。

表 3-3　半径为 0.45m 的圆面积

随机次数	100	1000	10 000	100 000	1 000 000	10 000 000
圆的面积/m^2	0.5994	0.638 28	0.639 171	0.636 311 7	0.636 471 27	0.636 210 045

当随机 10 000 次后，程序计算得到的面积就已经可以在实际应用中使用了。从结果中也可以看出，随着抽取次数的增加，算法得到的结果和实际值也会越来越接近(0.636 172 512 351 933 2)。但是也可以看到，随机一百万次得到的结果在精度上反而没有随机十万次高。这是因为计算机产生的随机数属于伪随机数，它们是由数学公式产生的，无法做到真正的随机。在一些精度要求不高的应用场景下，随机一千次以后得到的估计值就已经比较准确了(误差 0.33%)。但是随着抽取次数的增加，蒙特卡罗方法会需要更多的时间用于生成模拟数据。如果想要节约时间，可行的一种方案是采用并行计算。但是数值精度和计算资源消耗之间总是会构成一对矛盾。蒙特卡罗方法的优点是简单易用，只需正确地构造概率过程就可以建立目标估计量，但是对于高精度的要求会导致计算资源消耗线性增加，因此这也成为该方法一个为人诟病的缺点。

3.4.2　蒙特卡罗树搜索


视频讲解

1987 年，布鲁斯·艾布拉姆森在他的博士论文中率先探索了蒙特卡罗方法在棋类中的运用。1992 年，B. 布鲁格曼首次将其应用于对弈智能程序，但当时这种探索并没有受到重视。2006 年，雷米·库洛姆在其 2007 年的一篇论文中描述了蒙特卡罗方法在游戏树搜索中的应用，并将其命名为蒙特卡罗树搜索。之后列文特·科奇什和乔鲍·塞派什瓦里将蒙特卡罗树搜索方法与 UCB 公式结合，开发了 UCT 算法。MoGo、Fuego 等软件基于 UCT 算法，都曾在九路围棋中击败了人类业余棋手。

下棋时，棋手需要根据当前盘面情况选择对自己最有利的落子点。人类选手会采用类似于 Alpha-Beta 剪枝算法的思维策略，蒙特卡罗树搜索方法则采用另外一种策略，通过随机地模拟双方落子，最后汇总找出胜率最高的落子位。通常如果模拟的次数达到一定数量，算法总可以返回一个差不多理想的结果，如果模拟的次数足够多，算法结果会非常逼近实际的最优值。本质上，蒙特卡罗树搜索通过多次实施弱算法，用类似投票的机制对结果进行评价，从而形成一个强效的算法。

图 3-9 以对弈井字棋为例，演示了利用蒙特卡罗树搜索算法寻找最佳落子策略的过程。当画×方落子后，画○方有 8 个选择，为了避免结果产生偏差，朴素蒙特卡罗树搜索算法不会对当前盘面进行评估和评判，为了确认哪个选择更有利，算法会随机选择一处位置作为当前盘面画○方的落子，之后便开始仿真双方后续的落子情况直到游戏结束，这样就完成了一次完整的随机采样过程。仿真过程中的每一步落子都是随机的，算法每随机仿真一步就会产生一个子树枝节点，每仿真完一局棋局，就会在根节点上产生一条完整的树干。模拟的棋局越多，树枝就会越繁茂。

模拟完一局棋的对弈后算法需要记录仿真的结果，为了使每次仿真的数据能够被最大化利用，对弈的结果不仅要被保留在当前可选的树枝节点上，所有被仿真到的节点都应该记录本节点此次仿真的胜负结果。一种简单的策略是从最末的节点开始更新，顺序更新搜索路径上对应的父节点并反向传播直到当前树干链路上的所有节点全部完成更新为止。更新节点时可以采用赢一局计一分，输一局减一分，平局不得分的简单记法。当仿真了足够多的

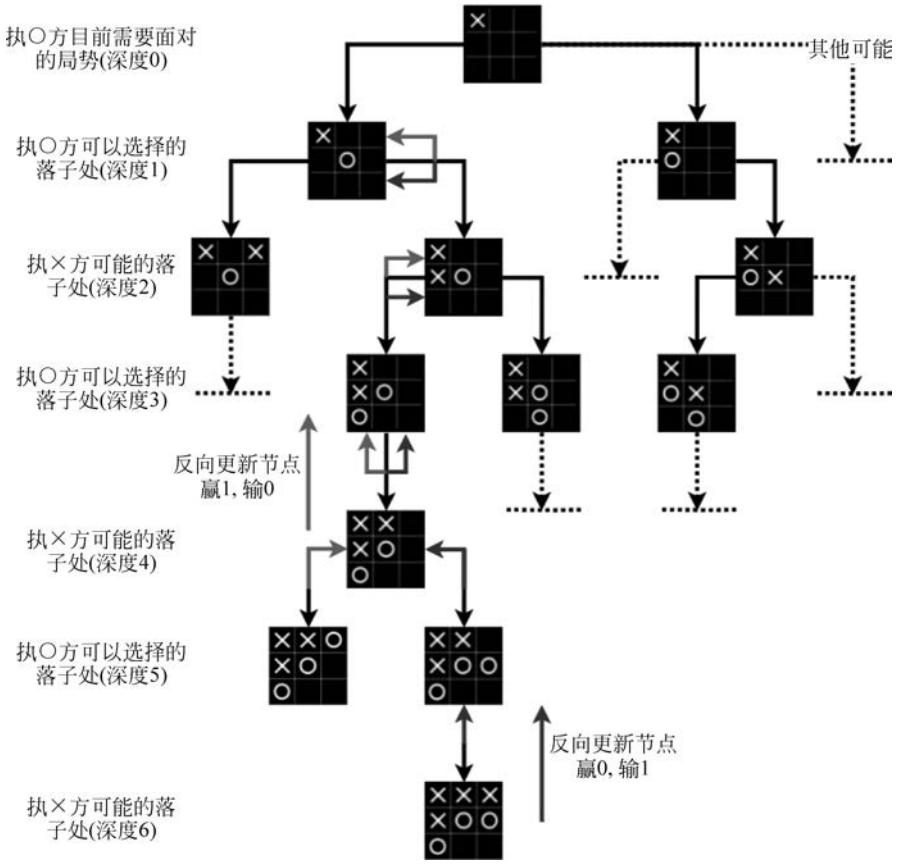

图 3-9　蒙特卡罗树搜索寻找最佳落子策略

棋局后，蒙特卡罗算法总是选择得分最高的节点作为最佳落子策略的估计。

为了最大化地利用仿真结果，采用蒙特卡罗树搜索算法的一方可以保留已经探索过的结果，在下一次决策前，先查找一下当前棋局盘面是否曾经探索过，如果碰巧探索过当前盘面，那么可以将代表当前盘面的节点剪出，并在此基础上继续进行仿真探索。图 3-10 演示了蒙特卡罗树搜索的剪枝过程，图中己方作为后手一开始面对的是对手落子 X_0 后的局面，根据算法仿真的结果，己方选择 O_1 点作为自己的落子。此时己方可以先维持住当前的搜索树，直到对手下出 X_1 的局面。当对手落子 X_1 后，与 O_1 节点并列的其他兄弟节点就失去了价值，因为它们所代表的落子顺序已不会再在棋局中出现。算法可以将 X_1 局面及其下属的枝叶剪出作为一个全新的搜索树来对待，这样做的好处是己方之前在这个局面下做过的仿真计算结果可以保留。如果计算能力足够，也可以把每个局面都当作一个全新的搜索树来对待，这样做的好处是程序易于实现，但是抛弃了历史的仿真记录，对计算资源的浪费比较严重。有些时候，由于实际环境的限制，模拟的次数可能无法覆盖全部可选项。例如，对方下出 X_2 的局面，这个情况之前并没有被模拟到过，算法可以选择将其看作一个全新的节点，剪枝出去后抛弃其他所有的历史数据。代码 3-14 演示了如何采用蒙特卡罗树搜索算法来下井字棋。

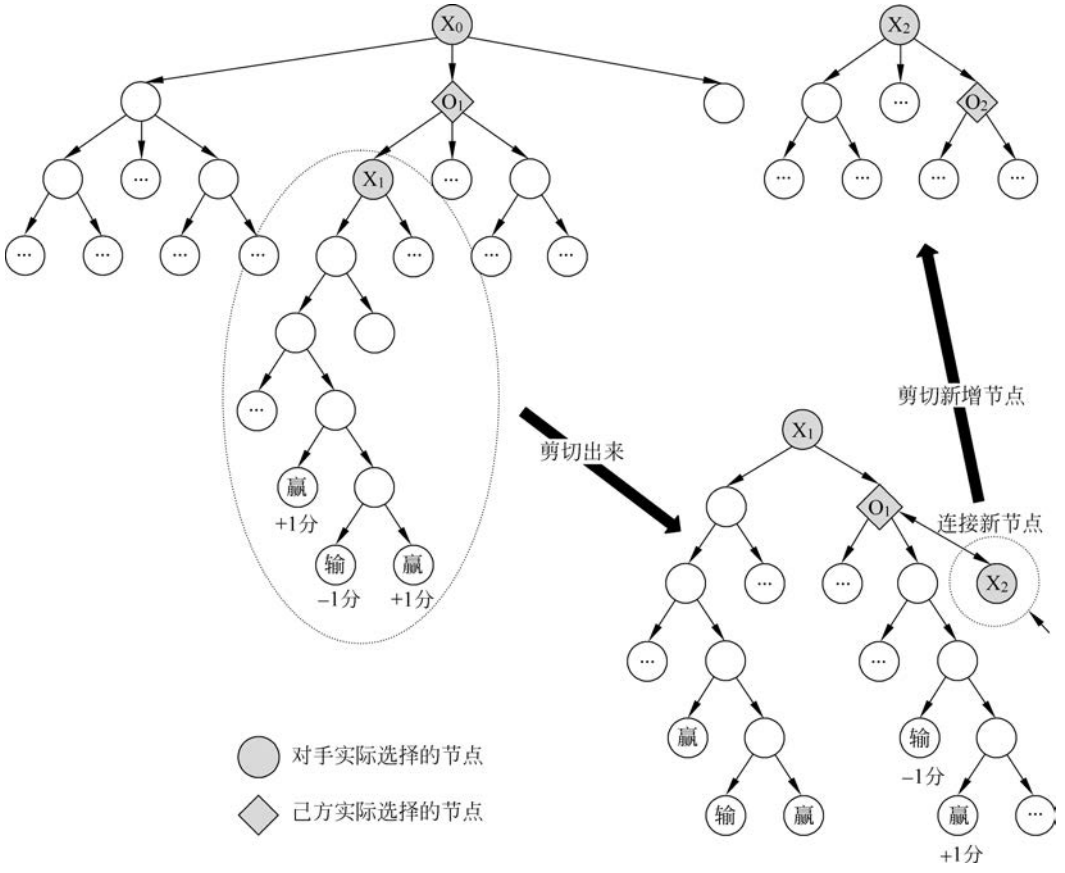

图 3-10　蒙特卡罗树搜索的剪枝

【代码片段 3-14】 利用蒙特卡罗树搜索来下井字棋。

```
MyGo\tic - tac - toe\ttt.py
class Agent:
    ...
    def chooseMove(self):
        ...
        if self.mode == 'mt':
                try_times = 1000 # 1
                if self.game.last_move is not None: # 2
                        node_tmp = self.tree.findNextNodeByMove(self.tree.tree_root,
self.game.last_move) # 3
                        if node_tmp == None: # 4
                                self.tree.node_name += 1
                                self.tree.tree_root = Node(
                        str(self.tree.node_name),
                        parent = self.tree.tree_root,move = self.game.last_move,
                        loss = 0,win = 0,draw = 0,player = -1 * self.player
                                )
                        else:
                                self.tree.tree_root = node_tmp
                node_point = self.tree.tree_root # 5
```

```
                    for i in range(try_times): #6
                            board_ = copy.copy(self.game.board.board)
                            node_start = node_point
                            move_records,game_result = easySimuGame(board_,self.player)
                            for move in move_records:
                                    node_start = self.tree.updateLeaf(node_start, move,game_
result) #7
                    all_next_leaves = self.tree.getNodeLeaves(node_point)
                    all_rates = [(node.loss/(node.loss + node.win + node.draw)) for node in all_
next_leaves]
                    all_rates = np.array(all_rates)
                     pick_move_index = int(random.choice(np.argwhere(all_rates == min(all_
rates)))) #8
                    self.tree.tree_root = all_next_leaves[pick_move_index] #8
                    return all_next_leaves[pick_move_index].move
```

【说明】

(1) 设置模拟对弈局数。

(2) 判断是不是新开始下棋。

(3) 查找当前盘面是否在过去的仿真中出现过。

(4) 如果第一次见到此局面则将其添加到完整的树结构中。

(5) 根据当前盘面从搜索树中剪出需要的树枝，抛弃不相关的分支。

(6) 采用随机策略多次仿真当前盘面下双方的对抗过程。

(7) 根据仿真结果更新各节点得分信息。

(8) 选择得分最高的节点作为当前盘面下的最优估计。

鼓励读者尝试在 ttt.py 中选择不同的仿真次数来比较智能体的对战结果。使用蒙特卡罗的智能程序在每步模拟 1000 次的情况下，棋力基本上就同采用极小化极大算法的智能程序一样了，而且蒙特卡罗算法在计算耗时上更平滑，初始几步的耗时要明显优于极小化极大算法。


视频讲解

3.4.3 蒙特卡罗算法改进

蒙特卡罗树搜索对模拟采样的次数要求较高，越多的模拟次数代表需要消耗越多的计算资源，这时就需要考虑如何有效地利用每一次的仿真结果。对于博弈游戏而言，很多时候一些落子位明显是没有意义的。以图 3-11 搜索井字棋盘面下一步为例，对于人类选手，一眼就能看出图中用★标注的位置是无论如何不能落子的。而对于单纯的蒙特卡罗算法，由于没有关于游戏的先验知识，算法会把这些标★的位置一并纳入随机采样中，这不仅是对计算资源的浪费，同时也会给游戏的对弈体验带来负面影响，因为智能体计算耗费的时间并没有和它的棋力对等。


图 3-11 井字棋的下一步选择

在围棋这种有很大的分支系数的对弈游戏

中，为了提高蒙特卡罗算法的搜索效率，可以增加一些辅助算法来限制搜索的范围，辅助算法需要帮助改进的内容包括以下两点。

（1）不要搜索没有意义的节点，尽可能把计算资源放在搜索价值大的节点。

（2）在有限的时间里进行更多的仿真模拟。

例如，在处理围棋的死活时，考虑对弈双方在如图 3-12 所示的角上的对抗情况。执黑子的人类棋手为了做活黑子，对画○的位置几乎是不会考虑的。另外，和被标注上△的位置相比，人类棋手会更倾向于先思考画×的位置，因为这些位置从表面上看起来对活棋会更加有利。从模拟人类下棋思考模式的角度出发，如图 3-13 所示，可以为算法增加一个盘面落子价值函数 $V=F(x)$，函数的输入是当前棋局，输出是各落子点的潜在价值。这个函数的目的是辅助蒙特卡罗树搜索算法缩小仿真的范围。

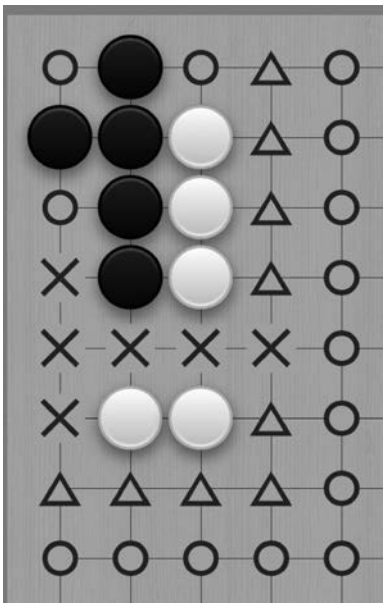

图 3-12　围棋的死活对抗

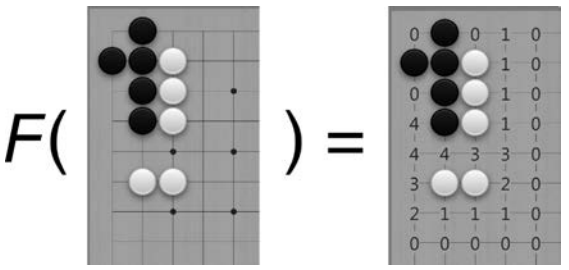

图 3-13　局面评估函数

如何得到价值函数 $V=F(x)$？一般是由专家根据经验手工编辑各种特征规则，函数将输入的局面去匹配这些特征规则后给出评分。如果没有专家，也可以通过机器学习的方法根据大数据样本学习出特征规则。需要注意，价值函数仅起到辅助作用，它输出的高评分落子位置并不代表这个落子点一定是好的。参考价值函数给出各个落子点的得分，蒙特卡罗算法优先选择得分高的点开始仿真。

很少有人类棋手实战时会下到最后完全没有落子点了才投子结束棋局。大部分围棋的对弈在下到中盘时就已经分出了胜负。采用蒙特卡罗树搜索算法的智能程序非常依赖仿真的次数，在有限的时间内，总是希望可以仿真更多的棋局。自然地，也会希望仿真算法可以不用下完整盘棋，在差不多中盘的时候就提前评估棋局的输赢。

在 3.3 节已经简单介绍过动态局面评估系统，针对围棋，一种可选的动态评估方法是把黑白棋子看作类似正负电荷一样的物质，并将棋盘看作电子作用下产生的电场。如图 3-14 所示表示棋子对四周“辐射”自己的影响力，相同的棋子可以叠加自己的影响力力场，不同的

棋子则相互抵消各自的影响力。动态评估试图在围棋棋盘上建立自己的物理法则，并以此来确定棋盘上每个棋子的势力范围。在 AlphaGo 问世以前，很多主流的围棋智能体程序都使用过这种思想。

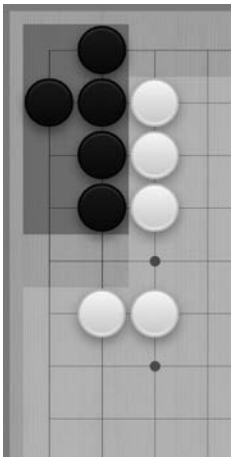

图 3-14　棋子的影响力范围

围棋的目的就是占领比对方更多的棋盘领地。利用动态局面评估机制就可以评估当前盘面双方的势力范围，计算机程序以此为基础就能粗略地判断棋局的输赢。为了知道什么时候应该截断单次仿真，算法还需要为动态评估系统建立一个胜负估计函数 $J=F(x)$。$F(x)$输出基于动态评估的盘面输赢判断置信度。和价值函数 $V=F(x)$类似，胜负估计的函数机制也可以人为编制。随着计算机大数据处理能力的增强，采用机器学习的方法获取 $J=F(x)$会比人为设立规则更轻松简便。

如图 3-15 所示，为了得到样本棋局每一回合的胜负置信度，可以引入退化参数 $R(0<R<1)$。当样本局结束时，胜负判断的置信度 $W=100\%$。之前的每一回合通过参数 R 反向传递置信度，可以得到单回合的胜负置信度递推公式(3-2)。

$$W_{t-1}=W_t-R \tag{3-2}$$

其中，T 表示第 T 个回合，$R=1/n$，而 n 表示当前棋局的总回合数。

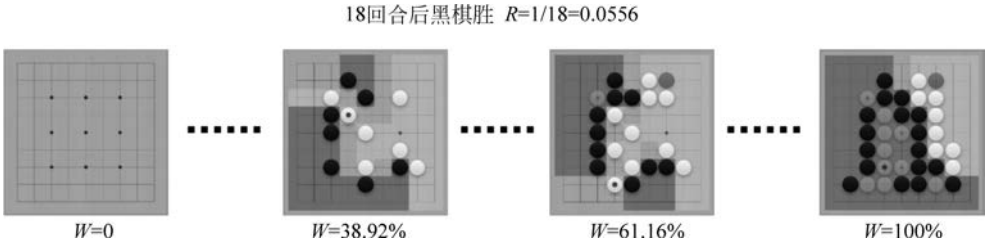

图 3-15　反向传递胜率置信度

可以人为设置一个置信度阈值，当 $J=F(x)$超过这个阈值时，则提前判定胜负，结束一次仿真。

有了辅助函数 $V=F(x)$和 $J=F(x)$，蒙特卡罗树搜索不仅可以控制搜索范围，还可以限制单次仿真的搜索深度，采用这种启发式的方法，算法可以在有限的时间内把计算资源尽可能地用于仿真有价值的落子选项，也就是对最有可能胜利的落子范围进行仿真。如果读过金庸的《天龙八部》就会知道，在围棋的世界里，表面看上去占优势的下法未必会赢，而前期劣势的下法可能会在后期转化为巨大的优势，如果只根据价值函数一味地考虑优势下法可能并不是一个好的选择。例如，如图 3-16 所示的这个存在变数的局面，对于白棋而言，价值函数如果偏重考虑做活自己的区域，标○的位置会得到比较高的评分。虽然

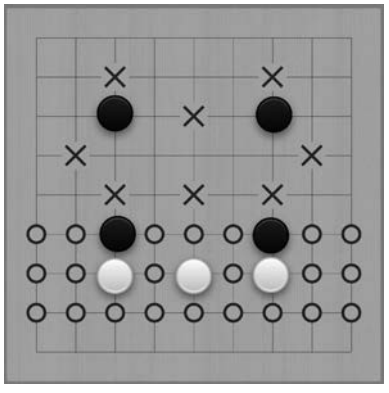

图 3-16　优势与劣势的逆转

白棋在这些区域落子后可以大概率将下半部分棋盘纳入自己的势力范围，但是对于全局而言，黑棋则有机会占领更多的上半部分棋盘，最终由于白棋在下路下的太厚，导致输掉这盘棋。如果白棋在参考价值函数之外能考虑到画×的位置，破坏黑棋独占上半路棋盘的趋势，则可能会有更大概率赢得这盘棋。

蒙特卡罗树搜索的改进算法 UCT 的出现目的就是在树搜索的深度与广度之间找到一个平衡。2006 年秋季，两位匈牙利研究人员列文特·科奇什和乔鲍·塞派什瓦里开发了 UCT 算法，使得围棋智能程序的胜率比当时的最佳算法提高了 5%，并且能够在小棋盘的比赛中与人类职业棋手相抗衡。

在价值函数的辅助下，搜索算法总是企图从价值最高的点开始模拟，这会导致算法偏重评估某些节点，忽略了一些潜在选择项。对于围棋而言，一些落子点的效果很可能要延迟很多回合后才会发挥威力，这些位置的价值在一开始不能通过固定的特征值组合判断出来。所以如图 3-17 所示，搜索树在仿真时不能一味地沿着图中黑色节点前进并且只考虑价值评估得分高的灰色节点作为继续仿真的对象。对评估价值低的点也应当偶尔提供计算资源。使用公式(3-3)来计算盘面节点落子的价值就可以平衡搜索深度与广度之间的关系。

$$v' = \omega + c\sqrt{\frac{\log N}{n}} \tag{3-3}$$

其中，ω 是当前节点通过仿真得到的胜负结果的平均值，N 是到目前为止计算机程序已经仿真了的总次数，n 是当前被考虑的节点已经被仿真的次数，c 表示人为选择的一个平衡参数。v'表示由 UCT 公式计算出来的节点搜索价值，探索仿真依照各节点的搜索价值来进行选择。当人为参数 c 选得比较大时，UCT 函数偏重广度优先，即将计算资源投入原本搜索价值不高的节点上，而当 c 比较小时，仿真则会偏向从搜索价值较高的节点上开始，也就是深度优先。至于这个超参 c 应该如何选择才能达到好的效果可能得仁者见仁智者见智了。

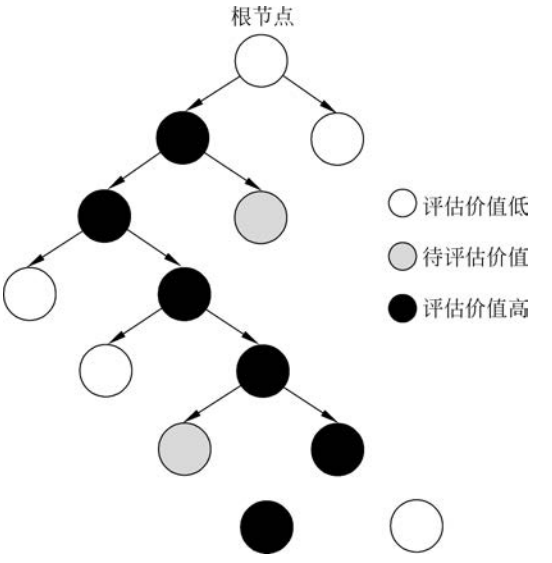

图 3-17　搜索时考虑评估建议之外的选项

图 3-18 展示了采用 UCT 算法前后的节点选取优先级对比，其中，节点内的数值是我们

利用自定义的价值评估函数得到的盘面价值及最终搜索次数，v'和n'则表示采用 UCT 算法计算后各个节点的价值和最终在每个节点上的探索次数。根据图中的计算结果，在使用新的 UCT 价值函数后，原本被忽略的节点($v=0.01$)由于价值上升反而被重点考量。

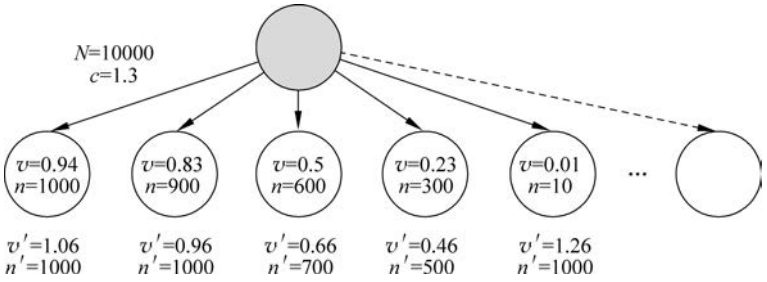

图 3-18　采用 UCT 算法前后的节点选取优先级对比

蒙特卡罗树搜索算法默认采用随机采样作为默认的仿真策略。随机策略保证了采样的均匀性，但是在实际应用中，由于博弈规则限制，仿真只能在有限的时间内进行。采用随机策略的围棋智能体往往棋力不高，为了进一步提高仿真的有效性，可以考虑使用更复杂的仿真策略。例如，在策略中引入了考虑气、形、定式、攻击模式、防守模式等一些重要的围棋基本概念。又如，在搜索策略中可以考虑引入常见棋形的固定下法，图 3-19 中白棋看到黑棋“虎”准备打吃自己，就用“长”来活棋。另外如图 3-20 所示，人类选手对弈时的起手总是从角的位置开始，在搜索前引入一些基础的开局库也许能起到事半功倍的效果。人类选手在评估棋力时，能记住多少围棋定式也是衡量标准之一。

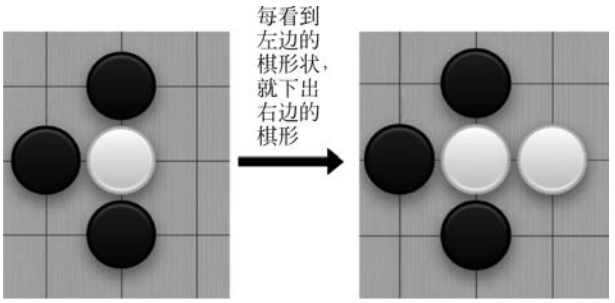

图 3-19　黑棋打吃，白棋长的固定下法

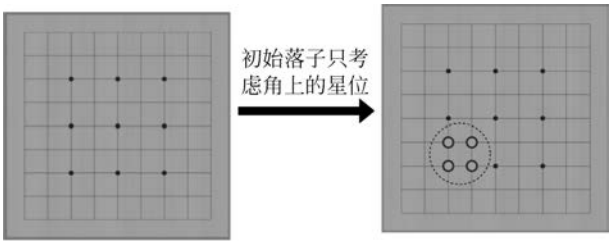

图 3-20　围棋起手总是从角上的星位开始

关于实现更复杂的仿真策略的细节部分就不再做过多的展开了，感兴趣的读者可以参考开源的围棋智能体引擎 GNU Go、Fuego 和 Pachi。GNU Go 的搜索算法是众多围棋软件中最强的，它使用了更加优化的搜索算法，棋力超过了普遍采用蒙特卡罗树搜索的程序。而

Fuego 和 Pachi 则是采用了更加先进的优化方案，在棋力上比 GNU Go 更胜一筹。

3.4.4　需要注意的问题

对一个可能有希望的落子点随机仿真 10 次后有 7 次赢得棋局，能有多大的信心认为这个落子点是一步好棋呢？答案是：并不乐观。假使实际上这个落子点对胜负的影响是平衡的（胜负各 50％的概率），通过随机的方法也会有 30％的机会得到 7 胜 3 负的结果。但是如果是 100 局仿真中有 70 次胜利呢？这种情况下，人们可以很自信地相信当前仿真的落子点是一步好棋。由于围棋的分支实在太多，对当前盘面的下一步棋来说，一般需要仿真近千次才会有一些自信判断是否是一步好棋。通常留给计算智能体程序思考的时间有限，而在一回合内至少需要仿真一万次才能基本上满足算法给出最优判断，此种情况下程序实现需要尽可能地优化，如果仿真一局都要耗时好几秒钟，即使算法再好，其实用性也会大打折扣。从提升性能的角度出发，仿真程序在每一回合都要对所有可选分支进行价值评估也是一笔不小的计算开支。如果仿真策略不使用随机策略，而是采用人工编制的高级策略，可以考虑不用每一步都计算各节点的评价得分，对于自信度高的下法可以跳过计算评估的动作。但是无论如何优化和改进，UCT 算法只能在分支少的小棋盘上赶上人类选手，在 19 路围棋棋盘上依然还是只能达到初级选手的水平。

3.5　监督学习

监督式学习是机器学习的一种方法，它可以让系统由训练数据中学到或建立一个模式，并依此模式推测新的实例。训练数据包括一套训练实例集，其中每个实例都是由一个输入的样本和一个预期输出标签所组成。监督学习算法就是分析该训练数据实例集，并使得系统可以逐步拟合出样本和标签之间的映射关系。如果把系统用函数来表示，这个函数的输出可以是一个连续的值，或是预测一个分类标签。通俗地讲，监督学习就像是老师指导学生学习知识。老师负责出题并告诉学生给出的答案是对是错。学生在学习过程中借助老师的提示获得经验，逐渐调整自己的认知，最后对没有学习过的问题也可以做出正确的解答。

在监督学习中，只需要给定输入样本集，机器就可以从中推演出指定目标变量的可能结果。机器只需从输入数据中预测合适的模型，并从中计算出目标变量的结果。监督学习要实现的目标是“对于输入数据 X 能预测变量 Y”。几乎所有的回归算法和分类算法都属于监督学习。传统上属于监督学习的算法有：K-近邻算法、决策树、朴素贝叶斯和逻辑回归。

目前神经网络已经成为最为热门的监督学习算法，本书后面的章节中也会使用神经网络作为基础工具来实现超越人类水平的围棋智能程序。在围棋这个主题上，单纯地使用神经网络很难在棋力上取得突破。这可能和围棋本身的复杂程度有关。关于使用神经网络来进行监督学习的内容将放在第 4 章进行详细的介绍。在那之后还将通过传统的监督学习来实现一个具备一定棋力的神经网络。

3.6　传统方法的讨论

在国际象棋更加流行的西方，人工智能的研究在开创之初就有人考虑如何制作一个能够下国际象棋的机器。20 世纪 80 年代初，贝尔实验室的工程师们开发出了历史上第一个

具有人类大师级水平的国际象棋机器 Belle。Belle 由三个主要部分组成：移动生成器，评估器和变换表。移动生成器负责识别出遭受攻击的最高价值部分和最低价值部分，并根据该信息对潜在的移动行为进行排序。评估器会注意到“王”在比赛的不同阶段的位置及其相对安全性。变换表包含潜在可移动的选项数据缓存，它可以使评估更加有效。Belle 采用了暴力的方法，它查看了玩家当前配置的棋盘可能做出的所有动作，然后又考虑了对手可以做出的所有动作。在国际象棋中，一名玩家移动一个棋子称为一层。最初，Belle 可以计算 4 层深度的移动。当 Belle 于 1978 年在计算机协会北美计算机国际象棋锦标赛上首次亮相时，它的搜索深度为 8 层，并夺得了冠军。

在 Belle 统治计算机国际象棋世界多年之后，它的明星效应开始褪色。20 世纪 80 年代末，卡内基·梅隆大学的许峰雄博士在 Bella 的思路基础上进一步改进，研制出了第一个特级大师水平的国际象棋机器，取名为“深思”。随后，许博士加入 IBM 研究院，在那里他和其他几个团队成员一起研制出了实力更强的弈棋机器“深蓝”，并最终于 1997 年的一场历史性的人机大战中以 3.5∶2.5 的比分战胜了人类国际象棋冠军卡斯帕罗夫。“深蓝”并没有对传统的暴力算法进行太多的改进，客观地说，“深蓝”不是基于 AI 技术构建的，它是一组专门为国际象棋而设计的 CPU 集群。“深蓝”也有棋类游戏智能软件常见的配置：开局库、着法生成器、评价函数、剪枝算法等。为了追求极致的搜索速度，它没有通过软件方式来实现，而是依靠专门为国际象棋设计的计算芯片，通过每秒计算两亿步的恐怖算能把人类甩了在后面。插句题外话，《“深蓝”揭秘》这本书揭秘了“深蓝”这台价值数百万美元的超级计算机背后的故事，许峰雄博士将自己的亲身经历以传记的形式呈现在人们面前，非常有意思，读者如果感兴趣可以尝试阅读一下。

2000 年开始，随着计算机硬件技术的发展和研究者对 Alpha-Beta 剪枝算法进一步的研究和优化，越来越多的棋类对弈软件开始逐步超越人类专业棋手。国际象棋的智能软件已经不再依赖专用象棋芯片就可以轻易战胜人类特级大师。国人制作的中国象棋软件也同样已经达到了无人匹敌的地步。这些弈棋计算机智能体背后的基本“思考模式”都很简单，就是对当前盘面下的每一种合法走法所直接导致的局面进行评估，然后选择“获胜概率”最高的局面所对应的那个走法。可以说，计算机的基本策略是所有“人类有可能采用”的策略中最原始、最简单的一种。这些弈棋智能体只关心一个问题，就是按照“胜”的基本定义来赢得比赛。“在当前盘面下，己方走哪一步能赢？”计算机就是通过不停地重复问自己这个问题来完成对弈的。

对计算机而言，从给定盘面开始的局势变化的复杂度是随考虑的步数呈指数级增长的。对于包括围棋和象棋等绝大多数复杂棋类运动而言，这就意味着从原则上不存在能够准确计算盘面最优结果的有效方法。有文献表明，19 路标准围棋棋盘的搜索空间是中国象棋的 10^{210} 倍，是国际象棋的 10^{227} 倍。普通人很难想象这个数字背后代表的物理差距，一个直观的理解是：原子的直径大小大约是 10^{-15} m，而太阳引力所能影响到的空间范围大约是原子核体积的 10^{90} 倍，也就是说，围棋相比于象棋的复杂度，比整个太阳系相对于一个原子的比例还要大。总之，除了人工设计的博弈项目之外，围棋是人类历史发展过程中所产生的最复杂的博弈项目，并且其复杂程度远远超过其他博弈项目。

表 3-4 列出了常见棋类游戏的状态空间复杂度和博弈树复杂度。状态空间复杂度是指从博弈初始状态开始所能达到的所有不同合法博弈状态的个数。它描述了通过枚举所能解

决的博弈复杂度范围。通常情况下，准确地计算博弈的状态空间复杂度是很困难的，也是不必要的，一般都会对其进行一个估算，有大致感性的理解即可。博弈树复杂度指的是从博弈初始状态所产生的、能完整解决该博弈问题的、最小博弈树子节点的个数。博弈树复杂度描述了通过极小极大搜索所能解决的博弈复杂度。准确计算围棋博弈树复杂度也是几乎不可能的，只能粗略地估算。实际在计算机智能体搜索策略中，需要面对的是博弈树展开后的复杂度。

表 3-4　常见游戏的状态空间复杂度和博弈树复杂度

游 戏 名 称	状态空间复杂度	博弈树复杂度
西洋跳棋	10^{21}	10^{31}
国际象棋	10^{64}	10^{123}
中国象棋	10^{48}	10^{150}
围棋	10^{172}	10^{360}
黑白棋	10^{28}	10^{58}
将棋	10^{71}	10^{226}

可以看出，西洋跳棋这种规则简单且步数有限的棋类游戏，它的合法状态空间复杂度都有 10^{21} 之巨。对于弈棋智能体的设计者来说，“不可能对局势变化的所有可能性进行有效计算”意味着想做得比对手更好需要从原理上解决以下两个关键问题。

(1) 决定一个“筛选策略”，从“所有当前盘面出发有可能导致的变化”中选择一部分作为“实际考虑的那些局面变化”。

(2) 决定一个“汇总策略”，把所有实际考虑的变化的静态评估结果综合起来，对当前盘面的胜率完成评估。

归根结底，传统算法的目的就是尽可能正确地“打分”和尽可能多地“穷举”。无论是国际象棋、中国象棋、围棋或者西洋跳棋、黑白棋等，这些棋类程序在 AlphaGo 出现以前，都围绕着这两点来进行基本框架的搭建。它们之间的区别仅在于使用的具体策略不同，以及针对不同的策略采用不同的优化手段。传统方法在搜索复杂度不是特别高的情况下可以工作得很好。例如，中国象棋和国际象棋的智能体程序早在十几年前就已经赶超了人类最顶尖的职业选手。不知道为什么，同样的策略作用在围棋软件上效果却乏善可陈。也许是围棋规则太过简单，又或者是在下围棋的时候并不是时时刻刻都要秉承着其最终胜利的原则。汉代扬雄在其《法言·君子》中提到：“昔乎颜渊以退为进，天下鲜俪焉”，意思是“以谦让取得德行的进步，以退让的姿态作为进取的手段”。传统方法时时刻刻都在基于“胜”的定义进行思考可能太过激进了。而 AlphaGo 背后的强化学习策略一改传统的设计思路。它不是一开始就奔着设计出一个强大的智能体程序为目的，强化学习只是教给了智能体程序基本的游戏规则，然后让程序自己在游戏中学习，这种方法目前看起来更为贴近人类的学习路径，而达到的效果也更好。