

xmwd

博客园 首页 新随笔 联系 订阅 管理

公告

昵称：xmwd
园龄：1年11个月
粉丝：3
关注：0
[+加关注](#)

<	2018年1月						>
日	一	二	三	四	五	六	
31	1	2	3	4	5	6	
7	8	9	10	11	12	13	
14	15	16	17	18	19	20	
21	22	23	24	25	26	27	
28	29	30	31	1	2	3	
4	5	6	7	8	9	10	

搜索

找找看

谷歌搜索

常用链接

[我的随笔](#)
[我的评论](#)
[我的参与](#)
[最新评论](#)
[我的标签](#)

我的标签

[python](#)(6)
[aiohttp](#)(3)
[asyncio](#)(2)
[crawler](#)(2)
[scrapy](#)(2)
[shadowsocks](#)(2)
[svn](#)(1)
[uct](#)(1)
[web](#)(1)
[科学上网](#)(1)
[更多](#)

随笔档案

[2017年6月](#) (1)
[2017年3月](#) (2)
[2017年2月](#) (1)
[2016年9月](#) (1)
[2016年4月](#) (1)
[2016年3月](#) (1)
[2016年2月](#) (2)
[2016年1月](#) (1)

最新评论

1. Re:python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏

随笔-10 文章-0 评论-12

python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏

更新

- 2017.2.23有更新，见文末。

MCTS与UCT

下面的内容引用自徐心和与徐长明的论文《计算机博弈原理与方法学概述》：

蒙特卡洛模拟对局就是从某一棋局出发，随机走棋。有人形象地比喻，让两个傻子下棋，他们只懂得棋规，不懂得策略，最终总是可以决出胜负。这个胜负是有偶然性的。但是如果让成千上万对傻子下这盘棋，那么结果的统计还是可以给出该棋局的固有胜率和胜率最高的着法。蒙特卡洛树搜索通过迭代来一步步地扩展博弈树的规模，UCT 树是不对称生长的，其生长顺序也是不能预知的。它是根据子节点的性能指标引导扩展的方向，这一性能指标便是 UCB 值。它表示在搜索过程中既要充分利用已有的知识，给胜率高的节点更多的机会，又要考虑探索那些暂时胜率不高的兄弟节点，这种对于“利用”（Exploitation）和“探索”（Exploration）进行权衡的关系便体现在 UCT 着法选择函数的定义上，即子节点 N_i 的 UCB 值按如下公式计算：

$$\frac{W_i}{N_i} + \sqrt{\frac{C \times \ln N}{N_i}}$$

其中：
 W_i ：子节点获胜的次数；
 N_i ：子节点参与模拟的次数；
 N ：当前节点参与模拟的次数
 C ：加权系数。

可见 UCB 公式由两部分组成，其中前一部分就是对已有知识的利用，而后一部分则是对未充分模拟节点的探索。 C 小偏重利用；而 C 大则重视探索。需要通过实验设定参数来控制访问节点的次数和扩展节点的阈值。

后面可以看到，在实际编写代码时，当前节点指的并不是具体的着法，而是当前整个棋局，其子节点才是具体的着法，它势必参与了每个子节点所参与的模拟，所以 N 就等于其所有子节点参与模拟的次数之和。当 C 取1.96时，置信区间的置信度达到95%，也是实际选择的值。

蒙特卡洛树搜索（MCTS）仅展开根据 UCB 公式所计算过的节点，并且会采用一种自动的方式对性能指标好的节点进行更多的搜索。具体步骤概括如下：

- 1.由当前局面建立根节点，生成根节点的全部子节点，分别进行模拟对局；
- 2.从根节点开始，进行最佳优先搜索；
- 3.利用 UCB 公式计算每个子节点的 UCB 值，选择最大值的子节点；
- 4.若此节点不是叶节点，则以此节点作为根节点，重复 2；
- 5.直到遇到叶节点，如果叶节点未曾被模拟对局过，对这个叶节点模拟对局；否则为这个叶节点随机生成子节点，并进行模拟对局；
- 6.将模拟对局的收益（一般胜为 1 负为 0）按对应颜色更新该节点及各级祖先节点，同时增加该节点以上所有节点的访问次数；
- 7.回到 2，除非此轮搜索时间结束或者达到预设循环次数；
- 8.从当前局面的子节点中挑选平均收益最高的给出最佳着法。

@xmwd连成一条线了，但是没阻止，您方便发一下您的QQ或微信吗，想具体聊一下

--Victoryli

2. Re:python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏

@Victoryli你自己落子的时候如果没有试图连成一线，电脑就会这样，如果你试图连成一线，电脑就会去阻止你...

--xmwd

3. Re:python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏

@xmwd 对，github上的

--Victoryli

4. Re:python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏

@Victoryli我也运行了，但是并没有出现你描述的情况，你是运行的github上的么？...

--xmwd

5. Re:python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏

@xmwd 都运行了，结果都一样

--Victoryli

阅读排行榜

1. python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏(3744)
2. scrapy cookies：将cookies保存到文件以及从文件加载cookies(2242)
3. C++ 事件驱动型银行排队模拟(1209)
4. Python实现的异步代理爬虫及代理池1--基本功能(920)
5. Scrapy:为spider指定pipeline(466)

评论排行榜

1. python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏(9)
2. Python实现的异步代理爬虫及代理池1--基本功能(2)
3. scrapy cookies：将cookies保存到文件以及从文件加载cookies(1)

推荐排行榜

1. C++ 事件驱动型银行排队模拟(1)
2. Python实现的异步代理爬虫及代理池1--基本功能(1)
3. python实现的基于蒙特卡洛树搜索(MCTS)与UCT RAVE的五子棋游戏(1)
4. scrapy cookies：将cookies保存到文件以及从文件加载cookies(1)

由此可见 UCT 算法就是在设定的时间内不断完成从根节点按照 UCB 的指引最终走到某一个叶节点的过程。而算法的基本流程包括了选择好的分支（Selection）、在叶子节点上扩展一层（Expansion）、模拟对局（Simulation）和结果回馈（Backpropagation）这样四个部分。UCT 树搜索还有一个显著优点就是可以随时结束搜索并返回结果，在每一时刻，对 UCT 树来说都有一个相对最优的结果。

代码实现

Board 类

Board 类用于存储当前棋盘的状态，它实际上也是MCTS算法的根节点。

```
class Board(object):
    """
    board for game
    """

    def __init__(self, width=8, height=8, n_in_row=5):
        self.width = width
        self.height = height
        self.states = {} # 记录当前棋盘的状态，键是位置，值是棋子，这里用玩家来表示棋子类型
        self.n_in_row = n_in_row # 表示几个相同的棋子连成一线算作胜利

    def init_board(self):
        if self.width < self.n_in_row or self.height < self.n_in_row:
            raise Exception('board width and height can not less than %d' %
                             self.n_in_row) # 棋盘不能过小

        self.availables = list(range(self.width * self.height)) # 表示棋盘上所有合法的位置，这里简单的认为空的位置即合法

        for m in self.availables:
            self.states[m] = -1 # -1表示当前位置为空

    def move_to_location(self, move):
        h = move // self.width
        w = move % self.width
        return [h, w]

    def location_to_move(self, location):
        if len(location) != 2:
            return -1
        h = location[0]
        w = location[1]
        move = h * self.width + w
        if move not in range(self.width * self.height):
            return -1
        return move

    def update(self, player, move): # player在move处落子，更新棋盘
        self.states[move] = player
        self.availables.remove(move)
```

MCTS 类

核心类，用于实现基于UCB的MCTS算法。

```
class MCTS(object):
    """
    AI player, use Monte Carlo Tree Search with UCB
    """
```

```
def __init__(self, board, play_turn, n_in_row=5, time=5, max_actions=1000):

    self.board = board
    self.play_turn = play_turn # 出手顺序
    self.calculation_time = float(time) # 最大运算时间
    self.max_actions = max_actions # 每次模拟对局最多进行的步数
    self.n_in_row = n_in_row

    self.player = play_turn[0] # 轮到电脑出手, 所以出手顺序中第一个总是电脑
    self.confident = 1.96 # UCB中的常数
    self.plays = {} # 记录着法参与模拟的次数, 键形如(player, move), 即(玩家, 落子)
    self.wins = {} # 记录着法获胜的次数
    self.max_depth = 1

def get_action(self): # return move

    if len(self.board.availables) == 1:
        return self.board.availables[0] # 棋盘只剩最后一个落子位置, 直接返回

    # 每次计算下一步时都要清空plays和wins表, 因为经过AI和玩家的2步棋之后, 整个棋盘的局面发生了变化, 原来的记录已经不适用了—原先普通的一步现在可能是致胜的一步, 如果不清空, 会影响现在的结果, 导致这一步可能没那么“致胜”了
    self.plays = {}
    self.wins = {}
    simulations = 0
    begin = time.time()
    while time.time() - begin < self.calculation_time:
        board_copy = copy.deepcopy(self.board) # 模拟会修改board的参数, 所以必须进行深拷贝, 与原board进行隔离
        play_turn_copy = copy.deepcopy(self.play_turn) # 每次模拟都必须按照固定的顺序进行, 所以进行深拷贝防止顺序被修改
        self.run_simulation(board_copy, play_turn_copy) # 进行MCTS
        simulations += 1

    print("total simulations=", simulations)

    move = self.select_one_move() # 选择最佳着法
    location = self.board.move_to_location(move)
    print('Maximum depth searched:', self.max_depth)

    print("AI move: %d,%d\n" % (location[0], location[1]))

    return move

def run_simulation(self, board, play_turn):
    """
    MCTS main process
    """

    plays = self.plays
    wins = self.wins
    availables = board.availables

    player = self.get_player(play_turn) # 获取当前出手的玩家
    visited_states = set() # 记录当前路径上的全部着法
    winner = -1
    expand = True

    # Simulation
    for t in range(1, self.max_actions + 1):
        # Selection
        # 如果所有着法都有统计信息, 则获取UCB最大的着法
        if all(plays.get((player, move)) for move in availables):
            log_total = log(
```

```
        sum(plays[(player, move)] for move in availables))
        value, move = max(
            ((wins[(player, move)] / plays[(player, move)]) +
             sqrt(self.confident * log_total / plays[(player, move)])), move)
        for move in availables)
    else:
        # 否则随机选择一个着法
        move = choice(availables)

    board.update(player, move)

    # Expand
    # 每次模拟最多扩展一次,每次扩展只增加一个着法
    if expand and (player, move) not in plays:
        expand = False
        plays[(player, move)] = 0
        wins[(player, move)] = 0
        if t > self.max_depth:
            self.max_depth = t

    visited_states.add((player, move))

    is_full = not len(availables)
    win, winner = self.has_a_winner(board)
    if is_full or win: # 游戏结束,没有落子位置或有玩家获胜
        break

    player = self.get_player(play_turn)

    # Back-propagation
    for player, move in visited_states:
        if (player, move) not in plays:
            continue
        plays[(player, move)] += 1 # 当前路径上所有着法的模拟次数加1
        if player == winner:
            wins[(player, move)] += 1 # 获胜玩家的所有着法的胜利次数加1

def get_player(self, players):
    p = players.pop(0)
    players.append(p)
    return p

def select_one_move(self):
    percent_wins, move = max(
        (self.wins.get((self.player, move), 0) /
         self.plays.get((self.player, move), 1),
         move)
        for move in self.board.availables) # 选择胜率最高的着法

    return move

def has_a_winner(self, board):
    """
    检查是否有玩家获胜
    """
    moved = list(set(range(board.width * board.height)) - set(board.availables))
    if len(moved) < self.n_in_row + 2:
        return False, -1

    width = board.width
    height = board.height
    states = board.states
    n = self.n_in_row
    for m in moved:
        h = m // width
        w = m % width
```

```
player = states[m]

if (w in range(width - n + 1) and
    len(set(states[i] for i in range(m, m + n))) == 1): # 横向连成一线
    return True, player

if (h in range(height - n + 1) and
    len(set(states[i] for i in range(m, m + n * width, width))) == 1): #
    竖向连成一线
    return True, player

if (w in range(width - n + 1) and h in range(height - n + 1) and
    len(set(states[i] for i in range(m, m + n * (width + 1), width + 1)))
    == 1): # 右斜向上连成一线
    return True, player

if (w in range(n - 1, width) and h in range(height - n + 1) and
    len(set(states[i] for i in range(m, m + n * (width - 1), width - 1)))
    == 1): # 左斜向下连成一线
    return True, player

return False, -1

def __str__(self):
    return "AI"
```

Human 类

用于获取玩家的输入，作为落子位置。

```
class Human(object):
    """
    human player
    """

    def __init__(self, board, player):
        self.board = board
        self.player = player

    def get_action(self):
        try:
            location = [int(n, 10) for n in input("Your move: ").split(",")]
            move = self.board.location_to_move(location)
        except Exception as e:
            move = -1

        if move == -1 or move not in self.board.availables:
            print("invalid move")
            move = self.get_action()

        return move

    def __str__(self):
        return "Human"
```

Game 类

控制游戏的进行，并在终端显示游戏的实时状态。

```
class Game(object):
    """
    game server
    """

    def __init__(self, board, **kwargs):
        self.board = board
```

```
self.player = [1, 2] # player1 and player2
self.n_in_row = int(kwargs.get('n_in_row', 5))
self.time = float(kwargs.get('time', 5))
self.max_actions = int(kwargs.get('max_actions', 1000))

def start(self):
    p1, p2 = self.init_player()
    self.board.init_board()

    ai = MCTS(self.board, [p1, p2], self.n_in_row, self.time, self.max_actions)
    human = Human(self.board, p2)
    players = {}
    players[p1] = ai
    players[p2] = human
    turn = [p1, p2]
    shuffle(turn) # 玩家和电脑的出手顺序随机
    while(1):
        p = turn.pop(0)
        turn.append(p)
        player_in_turn = players[p]
        move = player_in_turn.get_action()
        self.board.update(p, move)
        self.graphic(self.board, human, ai)
        end, winner = self.game_end(ai)
        if end:
            if winner != -1:
                print("Game end. Winner is", players[winner])
            break

def init_player(self):
    plist = list(range(len(self.player)))
    index1 = choice(plist)
    plist.remove(index1)
    index2 = choice(plist)

    return self.player[index1], self.player[index2]

def game_end(self, ai):
    """
    检查游戏是否结束
    """
    win, winner = ai.has_a_winner(self.board)
    if win:
        return True, winner
    elif not len(self.board.availables):
        print("Game end. Tie")
        return True, -1
    return False, -1

def graphic(self, board, human, ai):
    """
    在终端绘制棋盘，显示棋局的状态
    """
    width = board.width
    height = board.height

    print("Human Player", human.player, "with X".rjust(3))
    print("AI Player", ai.player, "with O".rjust(3))
    print()
    for x in range(width):
        print("{0:8}".format(x), end='')
    print('\r\n')
    for i in range(height - 1, -1, -1):
        print("{0:4d}".format(i), end='')
        for j in range(width):
            loc = i * width + j
```

```
if board.states[loc] == human.player:
    print('X'.center(8), end='')
elif board.states[loc] == ai.player:
    print('O'.center(8), end='')
else:
    print('_', end='')
print('\r\n\r\n')
```

增加简单策略

实际运行时，当棋盘较小（6X6），需要连成一线的棋子数量较少（4）时，算法发挥的水平不错，但是当棋盘达到8X8进行五子棋游戏时，即使将算法运行的时间调整到10秒，算法的发挥也不太好，虽然更长的时间效果会更好，但是游戏体验就实在是差了。因此考虑增加一个简单的策略：当不是所有着法都有统计信息时，不再进行随机选择，而是优先选择那些在当前棋盘上已有落子的邻近位置中没有统计信息的位置进行落子，然后选择那些离得远的、没有统计信息的位置进行落子，总得来说就是尽可能快速地向所有着法具有统计信息。对于五子棋来说，关键的落子位置不会离现有棋子太远。

下面是引入新策略的代码：

```
def run_simulation(self, board, play_turn):

    for t in range(1, self.max_actions + 1):
        if ...
        ...
        else:
            adjacents = []
            if len(availables) > self.n_in_row:
                adjacents = self.adjacent_moves(board, player, plays) # 没有统计信息的
邻近位置

            if len(adjacents):
                move = choice(adjacents)
            else:
                peripherals = []
                for move in availables:
                    if not plays.get((player, move)):
                        peripherals.append(move) # 没有统计信息的外围位置
                move = choice(peripherals)
            ...

def adjacent_moves(self, board, player, plays):
    """
    获取当前棋局中所有棋子的邻近位置中没有统计信息的位置
    """
    moved = list(set(range(board.width * board.height)) - set(board.availables))
    adjacents = set()
    width = board.width
    height = board.height

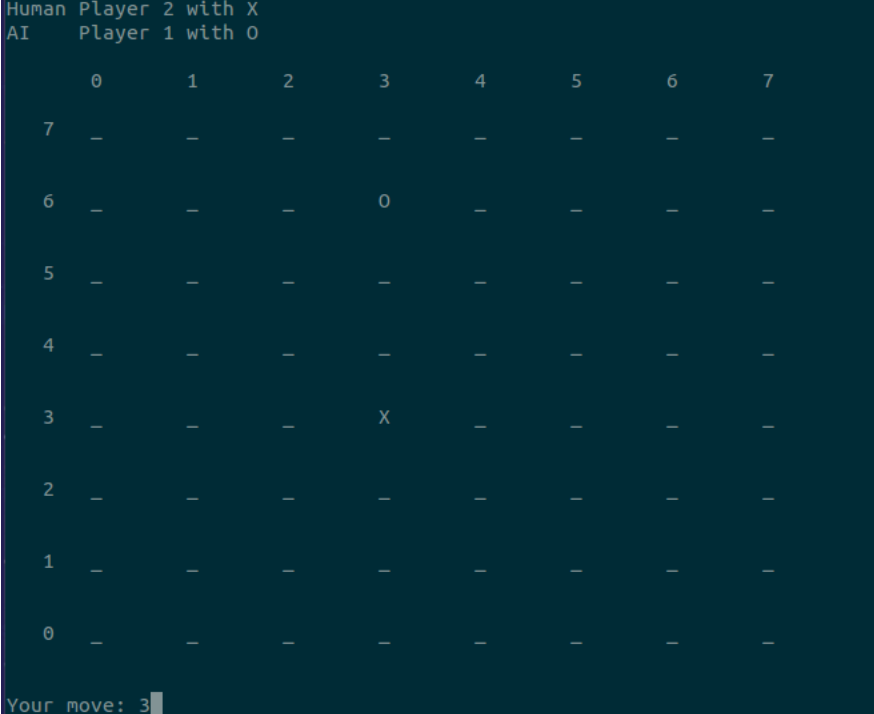
    for m in moved:
        h = m // width
        w = m % width
        if w < width - 1:
            adjacents.add(m + 1) # 右
        if w > 0:
            adjacents.add(m - 1) # 左
        if h < height - 1:
            adjacents.add(m + width) # 上
        if h > 0:
            adjacents.add(m - width) # 下
        if w < width - 1 and h < height - 1:
            adjacents.add(m + width + 1) # 右上
        if w > 0 and h < height - 1:
            adjacents.add(m + width - 1) # 左上
```

```
if w < width - 1 and h > 0:
    adjacents.add(m - width + 1) # 右下
if w > 0 and h > 0:
    adjacents.add(m - width - 1) # 左下

adjacents = list(set(adjacents) - set(moved))
for move in adjacents:
    if plays.get((player, move)):
        adjacents.remove(move)
return adjacents
```

现在算法的效果就有所提升了。

下面是运行效果：



```
Human Player 2 with X
AI Player 1 with O

      0      1      2      3      4      5      6      7
7  _ _ _ _ _ _ _ _
6  _ _ _ O _ _ _ _
5  _ _ _ _ _ _ _ _
4  _ _ _ _ _ _ _ _
3  _ _ _ X _ _ _ _
2  _ _ _ _ _ _ _ _
1  _ _ _ _ _ _ _ _
0  _ _ _ _ _ _ _ _

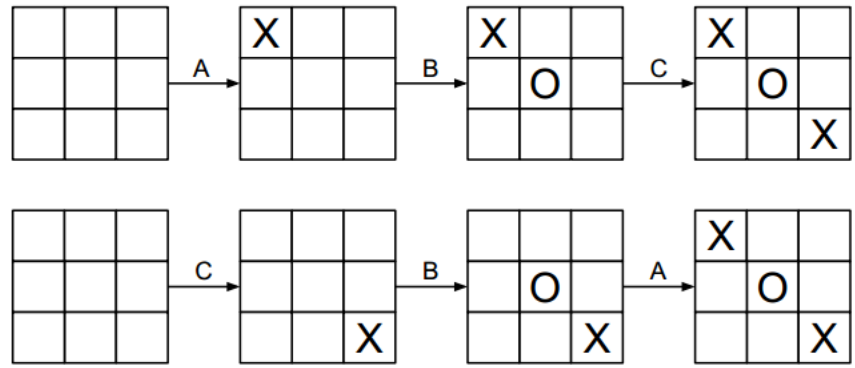
Your move: 3
```

完整的代码在Github的[n_in_row_not_so_correct.py](#)。

2017.2.23更新

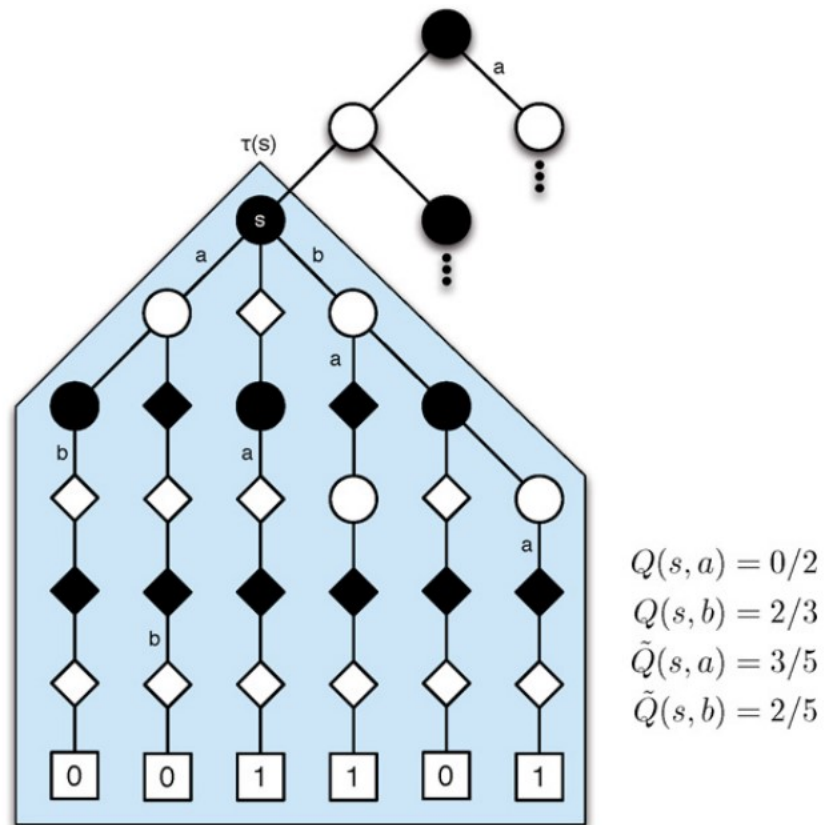
UCT RAVE

论文《Monte-Carlo tree search and rapid action value estimation in computer Go》提到了UCT的一种改进方法，叫做UCT RAVE (Rapid action value estimate)，提到RAVE，就不得不先提AMAF (All moves as first)：它视使棋盘达到某一相同状态的所有的着法都是等价的，不论是由谁在何时完成的，以下图为例：



图片引自论文《Monte Carlo Tree Search and Its Applications》，在经过A、B、C三步后得到当前的盘面状态，这三步的顺序是可能不相同的，但是得到的状态是相同的，所以不作区分。

RAVE是基于AMAF的，它视一个着法在对当前盘面的所有模拟中是相同的，不论它出现在何种子状态下、或是由哪个玩家给出的，如下图：



图片引自论文《Monte-Carlo tree search and rapid action value estimation in computer Go》。a和b表示2中可能的着法，叶子节点中的数字表示是否获胜。

对于状态s来说，如果使用MC的统计方法，那么(a, s)出现的次数是2，胜利的次数是0，而(b, s)出现的次数是3，胜利的次数是2，根据胜率，应该选择着法b；如果使用RAVE的统计方法，那么着法a在状态s的所有子树中出现的次数5（不论它在第几层——即不论它的父节点是否是s，也不论是由哪个玩家进行的，只要是在状态s所进行的模拟中——也就是s的子树t(s)中即可），胜利的次数是3，着法b在状态s的所有子树中出现的次数5，胜利的次数是2，根据胜率，应该选择着法a。可以看到，对于相同的情况，两种方法可能给出不同的选择。

根据下面的公式评估2种方法，作出最终的选择，其中 $Q(s, a)$ 是MC的值， $\tilde{Q}(s, a)$ 是AMAF的值，这实际上就是MC RAVE：

$$Q_{\star}(s, a) = (1 - \beta(s, a))Q(s, a) + \beta(s, a)\tilde{Q}(s, a)$$

下面是一种比较简单的 β 计算方法：

$$\beta(s, a) = \sqrt{\frac{k}{3N(s) + k}}$$

$N(s)$ 是状态 s 总共进行的模拟次数，当 $k=N(s)$ 时， β 等于 $1/2$ ，即二者权重等价，不难看出， $N(s)$ 很大时，即模拟次数很多， $\beta(s, a) \approx 0$ ， $Q(s, a)$

MC 的值的权重大，‘ $N(s)$ ’很小时，即模拟次数较少的时候 $\beta(s, a) \approx 1$ ， $\tilde{Q}(s, a)$ 的权重大。

可以根据算法实际运行的情况来选择 k 的值，论文给出的实验结果表明 k 大于1000的时候效果较好。

在得到MC RAVE后，UCT RAVE就水到渠成了：

$$Q_{\star}^{\oplus} = Q_{\star}(s, a) + \sqrt{\frac{c * \log(N(s))}{N(s, a)}}$$

对于之前算法实现的质疑

在了解了RAVE之后，再看之前实现的算法，实际上是有问题的，它很像RAVE，也很像MC，但实际都没有实现正确：对于上面的 $t(s)$ 树，之前的算法认为棋盘上所有合法的位置都是 s 的子节点，以 a 为例（对于之前的算法来说，这里的 a 实际上指的是棋盘上的某个位置），对于玩家 $player1$ ，它统计的是 $(player1, a)$ ，当 $player1$ 在某次模拟（无论是否是在模拟的第一步就走出了着法 a ）走出了 a ，它统计了，对于另一个玩家 $player2$ 走出的 a ，它没有统计，如果以AMAF来考虑的话， a 总的模拟次数应该是 $(player1, a)$ 与 $(player2, a)$ 之和，但是对于胜利次数， $(player1, a)$ 统计的是 $player1$ 在走出 a 并获胜的次数，未统计 $player2$ 在走出 a 且 $player1$ 获胜的次数，和棋不进行统计，所以说之前的算法既不算RAVE，也不是MC！

问题是它的表现还不错，以至于我在继续看论文之前没有意识到它是有问题的，甚至在我意识到它有问题之前，还实现了一个基于它的UCT RAVE版本，而且表现进一步提升，下面是核心的代码：

```
def run_simulation(self, board, play_turn):
    """
    MCTS main process
    """

    plays = self.plays # 统计RAVE中的MC部分的值
    wins = self.wins
    plays_rave = self.plays_rave # 统计RAVE中的AMAF部分的值
    wins_rave = self.wins_rave
    availables = board.availables

    player = self.get_player(play_turn)
    visited_states = set()
    winner = -1
    expand = True
    # Simulation
    for t in range(1, self.max_actions + 1):
        # Selection
        if all(plays.get((player, move)) for move in availables):
            value, move = max(
                ((1 - sqrt(self.equivalence / (3 * plays_rave[move] + self.equivalence)))
                 * (wins[(player, move)] / plays[(player, move)]) +
                 sqrt(self.equivalence / (3 * plays_rave[move] + self.equivalence))
                 * (wins_rave[move][player] / plays_rave[move])) +
                sqrt(self.confident * log(plays_rave[move])) / plays[(player,
```

```

move)]]), move)

        for move in availables)    # UCT RAVE 公式: (1-beta)*MC + beta*AMAF +
UCB

    else:
        adjacents = []
        if len(availables) > self.n_in_row:
            adjacents = self.adjacent_moves(board, player, plays)

        if len(adjacents):
            move = choice(adjacents)
        else:
            peripherals = []
            for move in availables:
                if not plays.get((player, move)):
                    peripherals.append(move)
            move = choice(peripherals)

    board.update(player, move)

    # Expand
    if expand and (player, move) not in plays:
        expand = False
        plays[(player, move)] = 0
        wins[(player, move)] = 0
        if move not in plays_rave:
            plays_rave[move] = 0 # 统计全部模拟中此着法的使用次数, 不论是由谁在何时给出的
        if move in wins_rave:
            wins_rave[move][player] = 0 # 统计全部模拟中给出此着法的不同玩家的胜利次数
        else:
            wins_rave[move] = {player: 0}
        if t > self.max_depth:
            self.max_depth = t

    visited_states.add((player, move))

    is_full = not len(availables)
    win, winner = self.has_a_winner(board)
    if is_full or win:
        break

    player = self.get_player(play_turn)

    # Back-propagation
    for player, move in visited_states:
        if (player, move) in plays:
            plays[(player, move)] += 1
            if player == winner:
                wins[(player, move)] += 1
        if move in plays_rave:
            plays_rave[move] += 1 # 本次模拟中该着法只要被使用就增加1
            if winner in wins_rave[move]:
                wins_rave[move][winner] += 1 # 本次模拟中使用该着法的并获胜的玩家的胜利次数增
加1

```

它在较短的时间内, 如5s或10s内的表现是优于下面将要说明的实现的, 因为短时间内它的模拟次数更多, 这也符合蒙特卡洛方法的原理。完整的代码在Github中的 `n_in_row_uct_rave_not_so_correct.py`。

正确的MC与UCT RAVE实现——也许？

下面是根据论文以及我的理解重新实现的UCT RAVE, 因为水平有限, 不一定对, 把核心代码放在这里, 欢迎大家与我一起讨论。

```

def run_simulation(self, board, play_turn):
    """
    UCT RAVE main process
    """

    plays = self.plays # 统计RAVE中的MC部分的值
    wins = self.wins
    plays_rave = self.plays_rave # 统计RAVE中的AMAF部分的值
    wins_rave = self.wins_rave
    availables = board.availables

    player = self.get_player(play_turn)
    winner = -1
    expand = True
    states_list = []
    # Simulation
    for t in range(1, self.max_actions + 1):
        # Selection
        state = board.current_state() # 棋盘的当前状态
        actions = [(move, player) for move in availables]
        if all(plays.get((action, state)) for action in actions):
            total = 0
            for a, s in plays:
                if s == state:
                    total += plays.get((a, s)) # N(s)
            beta = self.equivalence/(3 * total + self.equivalence)

            value, action = max(
                ((1 - beta) * (wins[(action, state)] / plays[(action, state)]) +
                 beta * (wins_rave[(action[0], state)][player] /
                 plays_rave[(action[0], state)])) +
                sqrt(self.confident * log(total) / plays[(action, state)]),
                action)
            for action in actions: ## UCT RAVE 公式: (1-beta)*MC + beta*AMAF + UCB

        else:
            action = choice(actions)

        move, p = action
        board.update(player, move)

        # Expand
        if expand and (action, state) not in plays:
            expand = False
            plays[(action, state)] = 0 # action是(move,player)。在棋盘状态s下, 玩家
            player给出着法move的模拟次数
            wins[(action, state)] = 0 # 在棋盘状态s下, 玩家player给出着法move并胜利的次数

            if t > self.max_depth:
                self.max_depth = t

            states_list.append((action, state)) # 当前模拟的路径

            # 路径上新增加的节点是前面所有节点的子节点, 存在于前面各个节点的子树中
            for (m, pp), s in states_list:
                if (move, s) not in plays_rave:
                    plays_rave[(move, s)] = 0 # 棋盘状态s下的着法move的模拟次数, 不论是由谁在何
                    时给出的
                    wins_rave[(move, s)] = {} # 棋盘状态s下着法move中记录着所有玩家在该着法
                    move出现的时候的胜利次数, 不论是由谁在何时给出的
                    for p in self.play_turn:
                        wins_rave[(move, s)][p] = 0

```

```
is_full = not len(available)
win, winner = self.has_a_winner(board)
if is_full or win:
    break

player = self.get_player(play_turn)

# Back-propagation
for i, ((m_root, p), s_root) in enumerate(states_list):
    action = (m_root, p)
    if (action, s_root) in plays:
        plays[(action, s_root)] += 1
        if player == winner and player in action:
            wins[(action, s_root)] += 1

    for ((m_sub, p), s_sub) in states_list[i:]:
        plays_rave[(m_sub, s_root)] += 1 # 状态s_root的所有子节点的模拟次数增加1
        if winner in wins_rave[(m_sub, s_root)]:
            wins_rave[(m_sub, s_root)][winner] += 1 # 在状态s_root的所有子节点中, 将
            获胜的玩家的胜利次数增加1
```

这个算法要得到一个较好的选择需要更多的时间/更多的模拟次数, 同时在实际使用的过程中, β 的选择也很关键的, 如果使用前面介绍的公式, 需要经过很多模拟比如10000次才能有一个较好的着法。其他的方法就比较复杂了, 甚至可以结合机器学习。当总的模拟次数不多的时候, `k` 的值不宜过大, 我在模拟时间等于15s左右的时候, 平均的模拟次数在3000次左右, 这个时候取 `k=1000` 时算法表现较好。

之所以同样的时间模拟次数比前面有问题的算法要少得多, 是因为随着模拟的进行, `plays` 等中的内容会远多于前面的算法, 所以遍历就需要更多的时间。实际上算法在做出了选择之后, 有一部分节点就不再需要了, 这个时候就可以进行剪枝, 下面是一种简单直接的剪枝方法:

```
def prune(self):
    """
    根据状态的长度进行剪枝
    """
    length = len(self.board.states) # 当前棋盘的状态
    keys = list(self.plays)
    for a, s in keys:
        if len(s) < length + 2: # 现在算法已给出了选择, 但尚未更新到棋盘, 在下次模拟的时候, 状态s的长度比现在的增加了2 (AI的选择和玩家的选择)
            del self.plays[(a, s)] # 所有状态s的长度小于下次模拟开始时状态的长度的节点
            已经不再需要了

            del self.wins[(a, s)]

    keys = list(self.plays_rave)
    for m, s in keys:
        if len(s) < length + 2:
            del self.plays_rave[(m, s)]
            del self.wins_rave[(m, s)]
```

完整的代码见Github的[n_in_row_uct_rave.py](#)。

正是因为单位时间内模拟的次数较少, 所以短时间内的表现上不如之前的算法, 可以通过多线程等手段来进一步提高单位时间内的模拟次数, 这也是下一讲要进行的工作之一。

参考资料

- [1] Jeff Bradberry "Introduction to Monte Carlo Tree Search", Github.
- [2] 徐心和, 徐长明. 计算机博弈原理与方法学概述[J]. 中国人工智能进展, 2009: 1-13.
- [3] Gelly S, Silver D. Monte-Carlo tree search and rapid action value estimation in computer Go[J]. Artificial Intelligence, 2011, 175(11): 1856-1875.
- [4] Magnuson M. Monte Carlo Tree Search and Its Applications[J]. Scholarly Horizons: University of

Minnesota, Morris Undergraduate Journal, 2015, 2(2): 4.

标签: python, mcts, uct, game, rave

好文要顶

关注我

收藏该文

xmwd

关注 - 0

粉丝 - 3

10

+加关注

« 上一篇 : C++ 事件驱动型银行排队模拟
» 下一篇 : aiohttp之添加静态资源路径

posted @ 2017-02-19 15:04 xmwd 阅读(3745) 评论(9) 编辑 收藏

评论列表

- #1楼 2017-02-23 10:47 weiyinfu

强

支持(0) 反对(0)
- #2楼[楼主] 2017-02-23 11:14 xmwd

@ weidiao
我感觉我的实现是有问题的，但是算法表现居然还可以.....今天应该就会更新这篇文章了，欢迎讨论

支持(0) 反对(0)
- #3楼 2017-07-14 19:01 Victoryli

运行之后，为什么电脑下的棋都是从上到下按行落子，并且没有什么策略

支持(0) 反对(0)
- #4楼[楼主] 2017-07-14 21:09 xmwd

@ Victoryli
你运行的是哪个？

支持(0) 反对(0)
- #5楼 2017-07-15 08:41 Victoryli

@xmwd 都运行了，结果都一样

支持(0) 反对(0)
- #6楼[楼主] 2017-07-16 09:07 xmwd

@ Victoryli
我也运行了，但是并没有出现你描述的情况，你是运行的github上的么？

支持(0) 反对(0)
- #7楼 2017-07-16 09:10 Victoryli

@xmwd 对，github上的

支持(0) 反对(0)
- #8楼[楼主] 2017-07-16 09:12 xmwd

@ Victoryli
你自己落子的时候如果没有试图连成一线，电脑就会这样，如果你试图连成一线，电脑就会去阻止你

支持(0) 反对(0)
- #9楼 2017-07-16 09:14 Victoryli

@xmwd连成一条线了，但是没阻止，您方便发一下您的QQ或微信吗，想具体聊一下

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#)网站首页。

【推荐】加入腾讯云自媒体扶持计划，免费领取域名&服务器

【福利】限时领取，H3 BPM给你发年终奖

ar

ActiveReports 报表控件
V12 全新发布!

全面满足
.NET 报表开发需求

立即了解

- 最新IT新闻:
- 三星电子四季度或盈利150亿美元 暴增七成
 - 花旗分析师：苹果可能收购Netflix 概率40%
 - Snap CEO花400万美元为员工办年会 据说请Drake献唱
 - 每年被吃掉1500亿片，阿司匹林真是万金油？
 - 2018一开年 马斯克向挖隧道计划再投100万美元
- » 更多新闻...

阿里云

告别高昂运维费用 云计算全面助力
40+款核心产品免费半年 再+8000津贴任意采购

立即申请

- 最新知识库文章:
- 步入云计算
 - 以操作系统的角度述说线程与进程
 - 软件测试转型之路
 - 门内门外看招聘
 - 大道至简，职场上做人做事做管理
- » 更多知识库文章...