本科实验报告

黑白棋

课程名称: 人工智能

姓名: 刘星烨

学院: 计算机科学与技术

专业: 计算机科学与技术

学号: 3180105954

指导教师: 吴飞

浙江大学实验报告

课程名称: 人工智能 实验类型:

实验项目名称: 黑白棋

姓名: 刘星烨 专业: 计算机科学与技术 学号: 3180105954

同组学生姓名: 指导教师: 吴飞

实验地点: 实验日期:

1 问题重述

使用『蒙特卡洛树搜索算法』实现 miniAlphaGo for Reversi
使用 python 语言

2 设计思想

2.1 MCTS

简介：全称 Monte Carlo Tree Search，是一种人工智能问题中做出最优决策的方法，一般是在组合博弈中的行动（move）规划形式。它结合了随机模拟的一般性和树搜索的准确性。MCTS 受到快速关注主要是由计算机围棋程序的成功以及其潜在的在众多难题上的应用所致。超越博弈游戏本身，MCTS 理论上可以被用在以状态 state，行动 action 对定义和用模拟进行预测输出结果的任何领域。

基本算法：根据模拟的输出结果，按照节点构造搜索树。其过程可以分为下面的若干步。

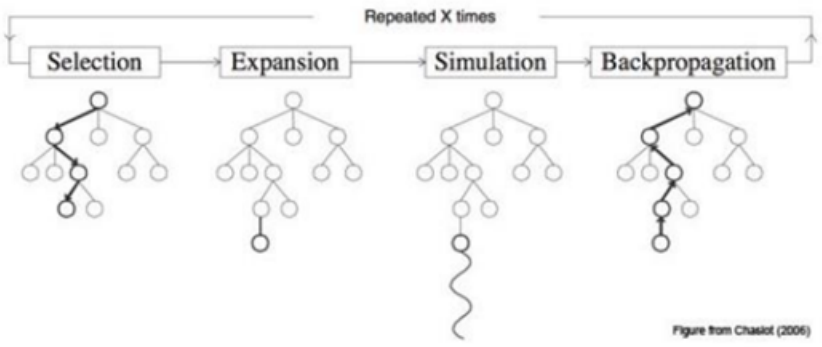

Figure 1: figure1

构建过程：

- 选择 Selection：从根节点 R 开始，递归选择最优的子节点直到达到叶子节点 L。
- 扩展 Expansion：如果 L 不是一个终止节点（也就是，不会导致博弈游戏终止）那么就创建一个或者更多的字子节点，选择其中一个 C。
- 模拟 Simulation：从 C 开始运行一个模拟的输出，直到博弈游戏结束。
- 反向传播 Backpropagation：用模拟的结果输出更新当前行动序列。

2.2 优先级表

在传统的 Monte Carlo 搜索中可以使用部分优化策略。根据部分研究成果，使用优先级策略可以在较为简单的的实现下得到有效的效果。

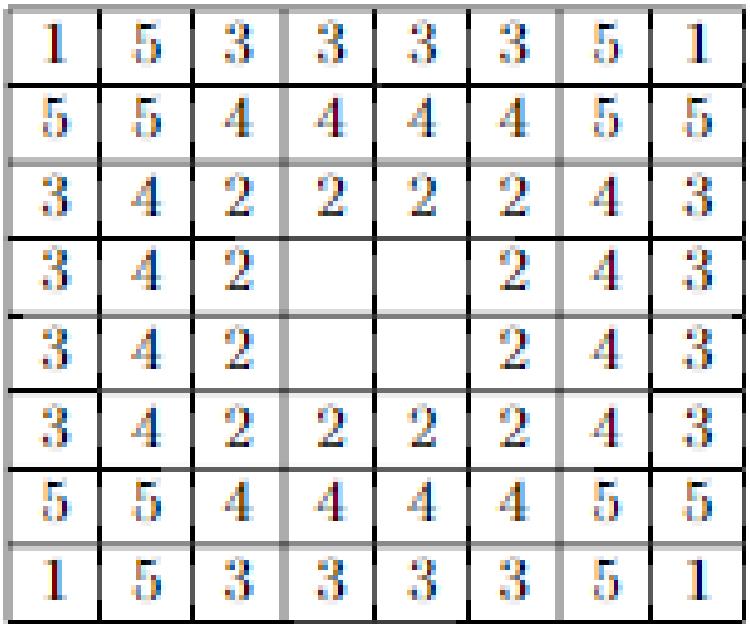

Figure 2: figure2

这个优先级表由 Roxanne 提出，结合了多种策略，同时也结合了 Mobility 的特性，因为中间子的优先级较高，会提高自己的 Mobility 而限制对手的可走步数。

3 代码内容

定义 MCST 的树节点结构

```
class Node(object):
  def __init__(self, board, parent, color, action):
      self.parent = parent # 父节点
      self.children = [] # 子节点列表
      self.visit_times = 0 # 访问次数
      self.board = board # 游戏选择这个Node的时的棋盘
      self.color = color # 当前玩家
      self.pre_move = action # 到达这个节点的action
      self.valid_moves = list(board.get_legal_actions(color)) # 未
          访问过的actions
      if (self.isover(board) == False) and (len(self.valid_moves)
          == 0): # 没得走了但游戏还没结束
           self.valid_moves.append("noway")

      self.reward = {'X': 0, 'O': 0}
      self.bestVal = {'X': 0, 'O': 0}

  def isover(self, board):
      return len(list(board.get_legal_actions('X')))==0 and len(
          list(board.get_legal_actions('O')))==0

  # def
```

```
    def calcBestVal(self, balance, color):
        if self.visit_times==0:
            print("-----------------------------")
            print("oops!visit_times==0!")
            self.board.display()
            print("-----------------------------")
        else:
            self.bestVal[color] = self.reward[color] / self.
                visit_times + balance * sqrt(2 * log(self.parent.
                visit_times) / self.visit_times)
```

定义搜索树的操作，包括 selection，expansion，simulation，backtracking

```
class MonteCarlo(object):
    # uct方法的实现
    # return: action(string)
    def search(self, board, color):
        # board: 当前棋局
        # color: 当前玩家
        actions=list(board.get_legal_actions(color))
        if len(actions) == 1:
            return list(actions)[0]

        # 创建根节点
        newboard = deepcopy(board)
        root = Node(newboard, None, color, None)

        # 考虑时间限制
        try:
            func_timeout(59, self.simulation, args=[root])
        except FunctionTimedOut:
            pass

        return self.best_child(root, math.sqrt(2), color).pre_move



    def simulation(self, root):
        while True:
            # mcts four steps
            # selection,expantion
            expand_node = self.tree_policy(root)
            # simulation
            reward = self.default_policy(expand_node.board,
                expand_node.color)
            # Backpropagation
            self.backup(expand_node, reward)

    def get_move_by_priority(self, actions, priority_table):
        found = False
        [rx, ry] = [None, None]
        for priority in priority_table:
            for (x,y) in priority:
                if (x,y) in actions:
                    found = True
                    [rx,ry] = [x,y]
                    break
            if found:
```

```
                break
        return rx,ry

    def expand(self, node):
        """
        输入一个节点，在该节点上拓展一个新的节点，使用random方法执行
            Action，返回新增的节点
        """

        action = random.choice(node.valid_moves)
        node.valid_moves.remove(action)

        # 执行action，得到新的board
        newBoard = deepcopy(node.board)
        if action != "noway":
            newBoard._move(action, node.color)
        else:
            pass

        if node.color=='O':
            newColor = 'X'
        else:
            newColor = 'O'
        newNode = Node(newBoard,node,newColor,action)
        node.children.append(newNode)

        return newNode

    def best_child(self, node, balance, color):
        # 对每个子节点调用一次计算bestValue
        val_list=[]
        for child in node.children:
            child.calcBestVal(balance, color)
            val_list.append(child.bestVal[color])
        value = max(val_list)
        index = val_list.index(value)
        # 返回bestValue最大的元素
        return node.children[index]

    def tree_policy(self, node):
        """
        传入当前需要开始搜索的节点（例如根节点）
        根据exploration/exploitation算法返回最好的需要expend的节点
        注意如果节点是叶子结点直接返回。
        """
        retNode = node
        while not retNode.isover(retNode.board):
            if len(retNode.valid_moves)>0:
                # 还有未展开的节点
                return self.expand(retNode)
            else:
                # 选择val最大的
                retNode = self.best_child(retNode, math.sqrt(2),
                    retNode.color)
        return retNode

    def default_policy(self, board, color):
        """
```

```
        蒙特卡罗树搜索的Simulation阶段
        输入一个需要expand的节点，随机操作后创建新的节点，返回新增节
            点的reward。
        注意输入的节点应该不是子节点，而且是有未执行的Action可以
            expend的。

        基本策略是随机选择Action。
        """
        newBoard = deepcopy(board)
        newColor = color

        while not (len(list(newBoard.get_legal_actions('X')))==0 and
            len(list(newBoard.get_legal_actions('O')))==0):
            actions = list(newBoard.get_legal_actions(newColor))
            action_num = []
            for action in actions:
                action_num.append(board.board_num(action))
            if len(actions) == 0:
                action = None
            else:
                action = self.get_move_by_priority(action_num,
                    priority_table)
            if action is None:
                pass
            else:
                newBoard._move(action, newColor)

            newColor = 'X' if newColor=='O' else 'O'

        # 0黑 1白 2平局
        winner, diff = newBoard.get_winner()
        diff /= 64
        return winner, diff

    def backup(self, node, reward):
        newNode = node
        # 节点不为None时
        while newNode is not None:
            newNode.visit_times += 1

            if reward[0] == 0:
                newNode.reward['X'] += reward[1]
                newNode.reward['O'] -= reward[1]
            elif reward[0] == 1:
                newNode.reward['X'] -= reward[1]
                newNode.reward['O'] += reward[1]
            elif reward[0] == 2:
                pass

            newNode = newNode.parent
```

定义 priority_table 如下：

```
priority_table = [[(0, 0), (0, 7), (7, 0), (7, 7)],
                  [(2, 2), (2, 3), (2, 4), (2, 5), (3, 2), (3, 3), (3,
                      4), (3, 5),
                   (4, 2), (4, 3), (4, 4), (4, 5), (5, 2), (5, 3), (5,
                       4), (5, 5)],
                  [(2, 0), (3, 0), (4, 0), (5, 0), (2, 7), (3, 7), (4,
```

```
        7), (5, 7),
     (0, 2), (0, 3), (0, 4), (0, 5), (7, 2), (7, 3), (7,
         4), (7, 5)],
    [(2, 1), (3, 1), (4, 1), (5, 1), (2, 6), (3, 6), (4,
         6), (5, 6),
     (1, 2), (1, 3), (1, 4), (1, 5), (6, 2), (6, 3), (6,
         4), (6, 5)],
    [(0, 1), (1, 0), (1, 1), (1, 6), (0, 6), (1, 7),
     (6, 1), (6, 0), (7, 1), (6, 6), (6, 7), (7, 6)]]
```

4 实验结果

通过网站的高级对战测试。