人工智能期末PJ Gomoku

郭涵青 16302010063

张悦嘉 16307110399

摘要

我们小组实现了两种搜索算法，分别是**基于威胁与依赖的搜索算法**和**极大极小值搜索算法**。在极大极小值算法中运用了多种加速方法，包括α-β剪枝、启发式着法生成、评估函数预生成、Zobrist缓存和算杀。

基于威胁与依赖的搜索算法

Dependecy-Based Search

冲四、活三、活跳三等都是五子棋中的 **威胁(threat)** 。具体的威胁类别与表示详见参考文献[1]133至135页。一般来说，当进攻方制造单重威胁的时候，防守方可以通过局部防守策略消除威胁。如果进攻方想要胜利，往往需要创造出一系列连续的威胁，并以双重威胁结束，使防守方无力回天。这一系列连续的威胁中，后来的威胁依赖于此前威胁中的落子，因此我们可以采用*dependency-based search*（下简称db-search）来搜索出一系列获胜威胁序列(winning threat sequence)。db-search的核心思想是通过*dependency stage*扩展依赖于单个新落子的叶节点，通过*combination stage*组合生成依赖于多个新落子的叶节点。在整个搜索过程中，“依赖”的概念十分关键，不考虑单重威胁的排列组合，从而可以大幅减小搜索空间。

db.py:

```
class DBProblem:
    # db-search搜索框架
    def search(self):
        while self.resourcesAvailable() and self.treeSizeIncreased:
            self.addDependencyStage(self.root)
            self.addCombinationStage(self.root)

    # 在dependency stage中扩展有依赖关系的子节点，重复调用addDependencyStage()，传入node的子节点
    def addDependencyStage(self, node):

    # 从根节点开始，向下寻找可以和node组合的结点partner，调用函数findAllCombinationNodes()
    def addCombinationStage(self, node):

    # 如果partner和node可以结合，就组合生成新结点
```

```
    def findAllCombinationNodes(self, partner, node):
```

Single-Agent Search

将db-search运用到五子棋，需要把多智能体的搜索转化为单智能体的搜索。注意到：除非防守方可以制造等级更高的威胁，否则面对进攻方的每一步威胁，防守方必须进行封堵，否则必然输棋。防守方所做的回应是有限的，因此我们可以把进攻方和防守方的行动合并成一个节点，从而将对抗游戏转化为单智能体的搜索问题。

防守方对进攻方的威胁作出的封堵被称为局部防守策略(local defensive move)。还有一种全局防守策略(global defensive move)，即创造出等级更高的威胁，例如在进攻方打出活三威胁时，防守方以自己的冲四威胁应对。我们在db-search的过程中是没有将这种策略纳入考量范围的，所以有可能错误地找到一种非获胜威胁序列。为了解决这个问题，在搜索过程中，在每一次进攻方行动后，都要搜索是否存在威胁等级更高的防守方行动。但在实际应用中，由于嵌套式的db-search耗时太长，将内循环的db-search改为仅搜索一步。

```
# 将棋盘翻转，以白棋为进攻方
def inverseBoard(board):

# 搜索白棋是否存在威胁等级更高的防守策略
def globalcheck(board, maxcat):
```

评估函数

如果当前棋盘上存在我方必胜的着数，则下一步走法不言而喻；如果当前棋盘上存在对方在我方不行动的情况下必胜的着数，且威胁等级高于我方的当前威胁，则我方应该选择防守，封堵对方的下一步威胁。

当棋盘上既不存在我方必胜，也不存在对方必胜的下一步走法时，用评估函数为棋盘上的每一个位置计分，找到获胜期望最大的落子位置。具体的计分方式详见代码。

```
# 对所有位置调用getScore()函数，返回分数最高的落子位置
def findMostPromising(board):

# 计算在(x,y)位置的分数
def getScore(board, x, y):
```

然而，这一过程在piskvork manager中耗时过长，有待进一步改进。

极大极小值搜索算法

极大极小值搜索

1. 在*minimax*中获得候选点，对每个点调用*get_value*，进行极大极小值搜索获取分数，最后返回一个最好的点。

2. 在*get_value*中，模拟在某个点下子，计算下子后的分数，计算完后就把该点下的子撤掉。计算分数的过程如下：
    1. 如果玩家是我们的AI，则调用*min_value*。
    2. 如果玩家是对手，则调用*max_value*。
    3. 如果达到了最大搜索深度，则调用评估函数，返回一个评估值。
3. 在*max_value*或*min_value*中，先获得候选点，再对每个点调用*get_value*，获取最大或最小的分数，其中运用到了**α-β剪枝**。

algorithms/minimax.py：

```
# 对外的接口，输入当前棋盘，可以返回一个分数最高的点
def minimax(b, deep, spread_l)

# 在position下子，并计算下子后的分数，最后撤回该子，完成一次模拟
# 如果达到了最大搜索深度，则返回一个评估值
def get_value(b, player, position, deep, spread_l, alpha, beta)

# 下面两个函数都需要调用get_value来进行模拟，最后返回当前模拟棋盘的分数
def max_value(b, player, deep, spread_l, alpha, beta)
def min_value(b, player, deep, spread_l, alpha, beta)
```

评估函数

评估函数是对整个棋局的评估，我们的计算方式是AI获得的最大得分减去对手获得的最大得分，即$maxAIScore - maxOPScore$。

事实上，我们遍历了每一个棋子情况，以一个合适的哈希值作为索引，将它们的得分存储在字典文件*pointCache.py*中。我们通过该点连续棋子长度、空子数（最大是1）、空了一个子后的连续棋子长度来估计分数。因此在模拟棋局时AI可以很快地更新双方得分（AIScore和OPScore），在进行棋盘评估时直接遍历AIScore和OPScore就可以得到结果。

algorithms/board.py：

```
class Board:
  # 存储双方在各点的得分
  # p的得分只与他周围五个棋子有关，因此将x,x,x,x,x,p,x,x,x,x,x的每一种形式作为索引即可获得p的得分
  self.AIScore
  self.OPScore
  # 存储score的索引
  # self.patternCache[R.AI][p[0]][p[1]][i],i表示横、竖、左斜、右斜
  self.patternCache

  # 遍历出双方在所有点上的最大得分，相减后得到当前棋盘评估
  def evaluate(self)
```

加速方法

启发式着法生成

在每一步生成所有可以落子的点，只看在附近一定范围内有邻居的点。

由于α-β剪枝的效率和节点排序有很大关系，如果最优的节点能排在前面，则能大幅提升剪枝效率。我们给所有待搜索的位置进行打分，按照分数的高低来排序。同样直接查询AIScore和OPScore就可以获得该点的分数。排序规则为：成五>活四>冲四>双三>活三>其他。

另外，我们使用了**双星延伸**，以提升性能。思路是，每次下的子，只可能是自己进攻，或者防守对手。我们假定任何时候，绝大多数情况下：

1. 自己进攻，一定是在己方杀棋的八个方向之一。
2. 防守对手，一定是在对方杀棋的八个方向之一。

algorithms/board.py:

```
class Board:
  # 存储双方在各点的得分
  self.AIScore
  self.OPScore

  # 给所有待搜索的位置进行打分，按照分数的高低来排序
  def gen(self, player, only_three=False, star_spread=False)
```

Zobrist缓存

Zobrist是一个快速Hash算法，每下一步棋，只需要进行一次异或操作。

每次下子时需要进行哈希值更新和保存，撤子时再一次异或就能回到原来原来的哈希值。每次下子时需要更新分数，就可以通过哈希值索引来获取AIScoreCache和OPScoreCache的分数。

usage/zobrist.py:

```
class Zobrist:
  self.AIHashing
  self.OPHashing
  self.boardHashing

  # 进行异或操作，更新哈希值
  def go(self, position, player)
```

algorithms/board.py:

```
class Board:
  self.AIScoreCache
  self.OPScoreCache

  # 下子时需要先更新哈希值，再更新分数，并保存分数缓存
  def put(self, position, player, record)
```

```
# 撤子时需要先更新哈希值，再更新分数
def remove(self, position)

# 更新分数时先判断是否有当前棋盘哈希值的分数缓存
def update_score(self, position, remove=False)

# 以当前棋盘哈希值为索引，保存分数缓存到AIScoreCache和OPScoreCache
def score_cache(self)

# 以当前棋盘哈希值为索引，获取分数缓存
def get_score_cache(self)
```

算杀

所谓算杀就是计算出杀棋，杀棋就是指一方通过连续的活三和冲四进行进攻，一直到赢的一种走法。我们一般会把算杀分为两种：**VCF** 连续冲四胜和 **VCT** 连续活三胜。

算杀其实也是一种极大极小值搜索，具体的策略为：

1. MAX层，只搜索己方的活三和冲四节点，只要有一个子节点的能赢即可
2. MIN层，搜索所有的己方和对面的活三和冲四节点（进攻也是防守），只要有一个子节点能保持不败即可

algorithms/vcx.py:

```
# 返回必杀棋，如果不存在则返回False
def get_max(b, player, deep, totalDeep=0)

# 如果对手的每一个候选棋都能被找到必杀，则返回该必杀棋，否则返回False
def get_min(b, player, deep)

# 迭代加深搜索
def deeping(b, player, deep, totalDeep)

# 获取vct或vcf的杀棋
def vcx(b, player, onlyFour, deep=None)
```

参考文献

[1] Searching for Solutions in Games and Artificial Intelligence(1994), Louis Victor Allis.

[2] https://github.com/lihongxun945/gobang.