对抗搜索实验报告

姓名：_____ 张韫译萱_____　学号：_____08023214_____

一、 实验题目

盘子中有 10 颗石子，两个玩家轮流从中取出，至少取 1 颗，至多取 3 颗，不能继续操作的玩家输。

1 实验任务

本次实验需要完成以下三项任务：

1. 使用 MiniMax 算法实现。
2. 使用 α-β 剪枝算法实现。
3. 设计友好的命令行用户界面。

2 实验要求

1. 选择 C++ 或 Python 实现。
2. 代码以文本形式粘贴在附录相应位置，注释准确完整，能成功运行。

二、 实验结果

```
================================================================
                          取石子游戏
================================================================

游戏规则:
  • 盘子中有10颗石子
  • 两个玩家轮流取石子
  • 每次至少取1颗，至多取3颗
  • 不能继续操作的玩家输

请选择算法:
  1. MiniMax算法
  2. Alpha-Beta剪枝算法
  3. 比较两种算法性能
  0. 退出
================================================================

请输入选择 (0-3): 1
================================================================
                     MiniMax算法 - 取石子游戏
================================================================

初始状态:
  剩余石子: 10颗 oooooooooo

请选择你的身份:
  1. 先手
  2. 后手

请输入选择 (1/2): 1

--- 第 1 回合 ---
当前状态: 剩余石子: 10颗 oooooooooo
可取石子数: [1, 2, 3]
请输入你要取的石子数 (1-3):
```

图 1: 开始界面

```
--- 第 1 回合 ---
当前状态：剩余石子：10颗 oooooooooo
可取石子数：[1, 2, 3]
请输入你要取的石子数 (1-3): 2
你取了 2 颗石子

--- 第 2 回合 ---
当前状态：剩余石子：8颗 oooooooo
AI正在思考...
AI取了 1 颗石子 (评估值：1)
探索节点数：176

--- 第 3 回合 ---
当前状态：剩余石子：7颗 ooooooo
可取石子数：[1, 2, 3]
请输入你要取的石子数 (1-3): 3
你取了 3 颗石子

--- 第 4 回合 ---
当前状态：剩余石子：4颗 oooo
AI正在思考...
AI取了 1 颗石子 (评估值：1)
探索节点数：14

--- 第 5 回合 ---
当前状态：剩余石子：3颗 ooo
可取石子数：[1, 2, 3]
请输入你要取的石子数 (1-3): 3
你取了 3 颗石子
==========================================================================
游戏结束！你获胜！
==========================================================================
```

图 2: 游戏过程

三、 实验分析

1 MiniMax 算法博弈树分析

MiniMax 算法的核心思想是通过构建完整的博弈树来寻找最优策略。在取石子游戏中，博弈树的每个节点代表一个游戏状态（剩余石子数），每条边代表一次操作（取 1、2 或 3 颗石子）。

博弈树由以下部分组成：

1. **根节点**：初始状态（10 颗石子）
2. **内部节点**：游戏进行中的状态
3. **叶子节点**：终止状态（0 颗石子，当前玩家输）
4. **分支因子**：最多为 3（取 1、2 或 3 颗石子）
5. **树的深度**：最大为 10（最多 10 步取完所有石子）

算法从叶子节点向上回溯计算评估值。对于叶子节点，如果当前为 MAX 玩家的回合，则评估值为-1；如果是 MIN 玩家的回合，则评估值为 1。对于内部节点，MAX 层选择子节点中评估值最大的，MIN 层选择子节点中评估值最小的。

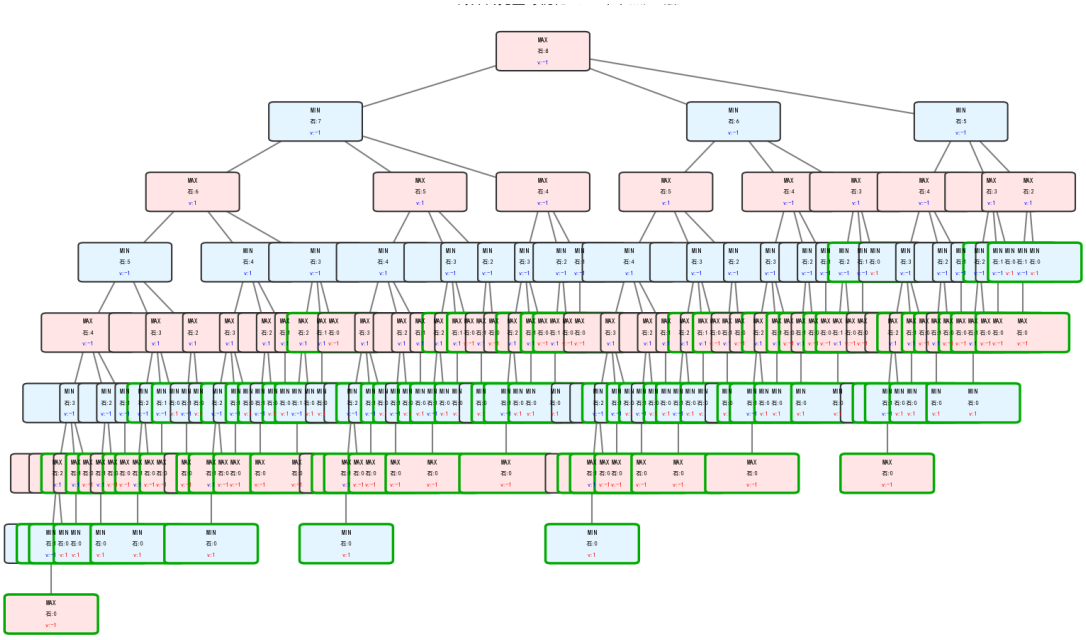

图 3: MiniMax 算法运行结果

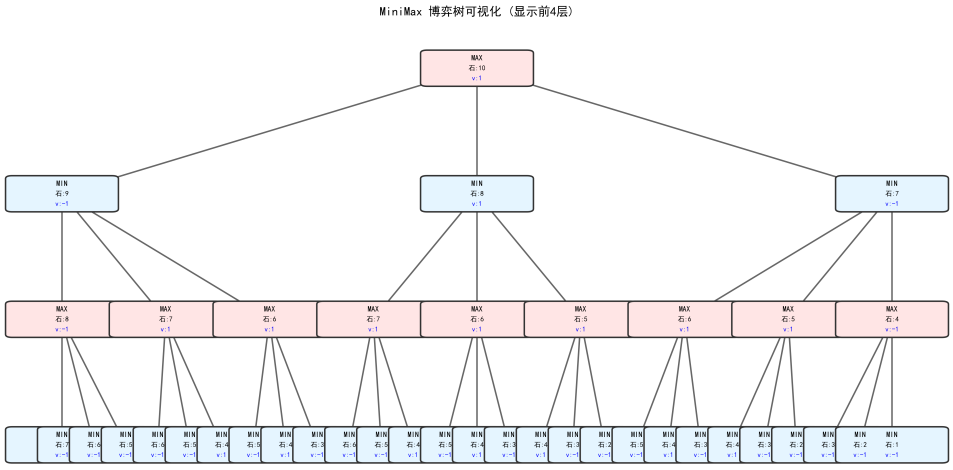

图 4: MiniMax 算法运行结果前四层

2 α-β 剪枝分析

α-β 剪枝算法通过剪枝操作大幅减少了需要搜索的节点数量，同时保证得到与 MiniMax 算法相同的结果。剪枝的基本原理是：

- 在 MAX 节点，更新 α。如果当前评估值 $\alpha \geq \beta$，则 MIN 玩家不会选择这条路径，可以剪枝。
- 在 MIN 节点，更新 β。如果当前评估值 $\beta \leq \alpha$，则 MAX 玩家不会选择这条路径，可以剪枝。

α-β 剪枝算法生成的博弈树整体非常复杂，这里仅展示树的一部分及可视化结果（红色部分为剪枝部分）。

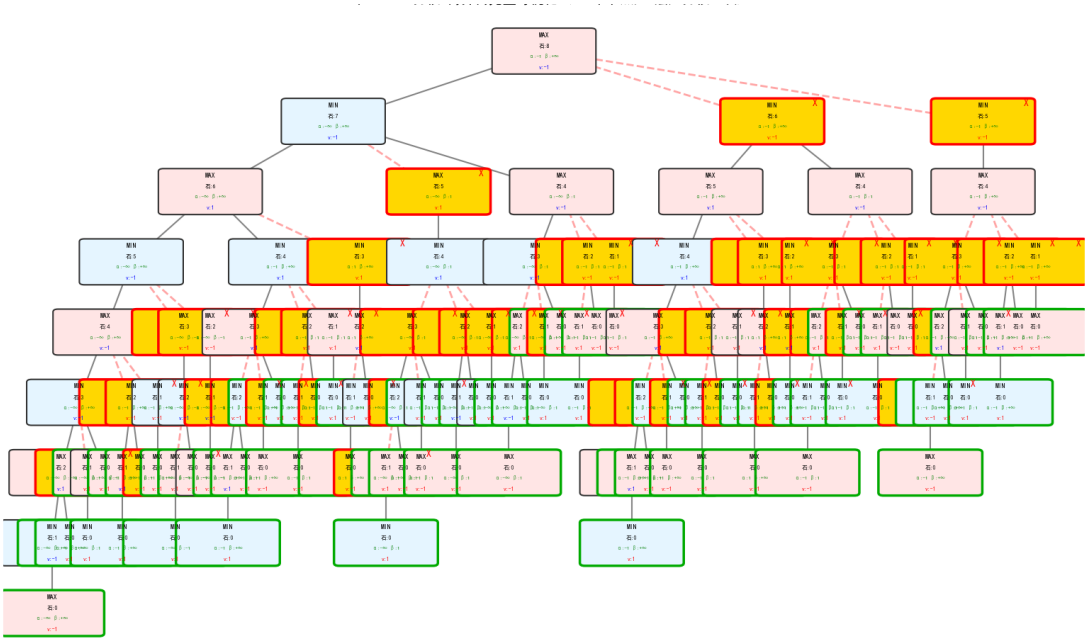

图 5: α-β 剪枝算法运行结果

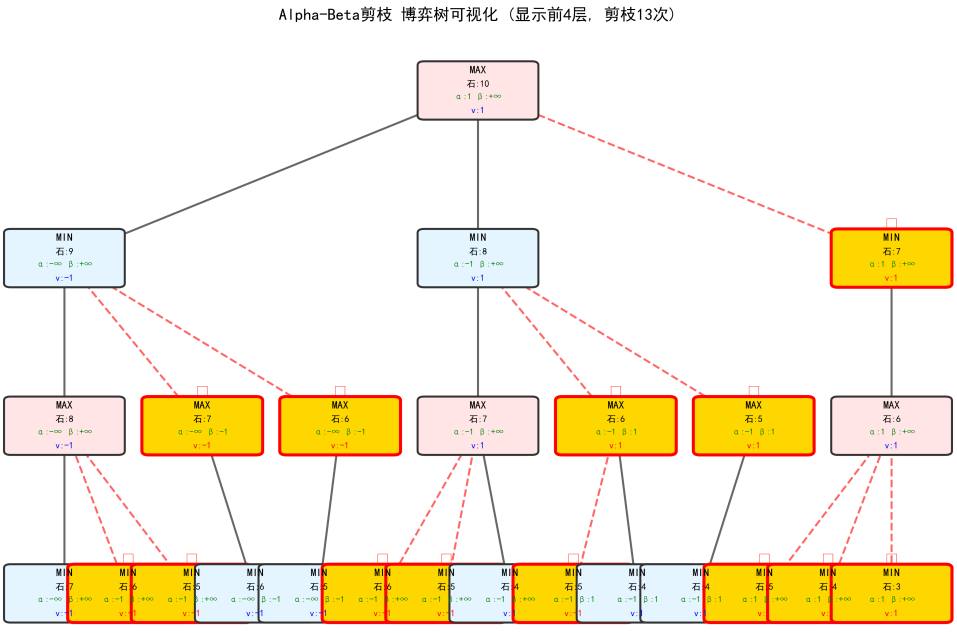

图 6: α-β 剪枝算法运行结果前四层

通过性能对比测试，我们发现在石子数为 10 时，α-β 剪枝算法能够减少约 30%-60% 的节点探索量，具体减少比例取决于移动顺序。对比过程见下。

3 算法性能对比

我们可以在游戏开始时分别运行两种算法（得到 MAX 玩家第一步的行动策略）。在过程中计数节点探索数来比较两种算法的性能。测试代码如下：

```
    def compare_algorithms():
    """比较两种算法的性能"""
    print_header("算法性能比较")

    game1 = StoneGame(10)
    game2 = StoneGame(10)

    minimax_solver = MiniMaxSolver()
    alphabeta_solver = AlphaBetaSolver()

    print("\n正在运行MiniMax算法...")
    mm_move, mm_value = minimax_solver.get_best_move(game1, is_maximizing=True)#假定为MAX玩家，执行算法，找到第一步最优策略
    mm_nodes = minimax_solver.nodes_explored #minimax探索的节点数

    print("正在运行Alpha-Beta剪枝算法...")
    ab_move, ab_value = alphabeta_solver.get_best_move(game2, is_maximizing=True)#假定为MAX玩家，执行算法，找到第一步最优策略
    ab_nodes = alphabeta_solver.nodes_explored #alpha_beta探索的节点数
    ab_cutted = alphabeta_solver.cutted_count #剪枝数

    print("\n" + "=" * 100)
    print(f"{'算法':<25} {'最佳移动策略':<15} {'评估值':<15} {'探索节点数':<15}")
    print("=" * 100)
    print(f"{'MiniMax':<30} {mm_move:<20} {mm_value:<15} {mm_nodes:<30}")
    print(f"{'Alpha-Beta剪枝':<28} {ab_move:<20} {ab_value:<15} {ab_nodes:<30}")
    print("=" * 100)
    print(f"\nAlpha-Beta剪枝次数: {ab_cutted}")
    print(f"节点探索减少率: {(1 - ab_nodes/mm_nodes)*100:.2f}%")
    print("=" * 100)
```

测试运行结果如下图所示：

```
====================================================================================================
算法                          最佳移动策略          评估值             探索节点数
====================================================================================================
MiniMax                       2                   1               599
Alpha-Beta剪枝                2                   1               328
====================================================================================================

Alpha-Beta剪枝次数: 126
节点探索减少率: 45.24%
====================================================================================================
```

图 7: 两种算法性能比较

从对比结果可以看出，α-β 剪枝算法在保证结果正确性的前提下，显著提高了搜索效率，特别是在状态空间较大的情况下，优势更加明显。

四、 实验总结

通过本次实验，我加深了对博弈树搜索理论的理解。（MiniMax 算法体现了对抗搜索的基本思想，即在假设对手采取最优策略的前提下，为自己寻找最优决策。通过递归地构建博弈树，算法能够准确评估每个状态的价值。α-β 剪枝则是对 MiniMax 的重要优化，它利用了博弈树的对抗性质，通过维护上下界（α, β）来排除不必要的搜索分支）。

本次算法编程过程中，使用递归的方式实现两种算法是比较合适的。在实现过程中，需要仔细考虑递归的终止条件、状态的转换、评估值的计算和回溯等细节问题。通过动手实践，我进一步加深理解了递归算法的实现技巧。

附录

1 MiniMax 算法代码

```python
class StoneGame:
    """石子游戏类"""

    def __init__(self, initial_stones=10):
        self.stones = initial_stones
        self.initial_stones = initial_stones

    def is_terminal(self):
        """判断是否为终止状态"""
        return self.stones == 0

    def get_valid_moves(self):
        """获取当前状态下的合法移动（可以取1、2或3颗石子）"""
        return [i for i in range(1, 4) if i <= self.stones]

    def make_move(self, move):
        """执行移动"""
        if move in self.get_valid_moves():
            self.stones -= move
            return True
        return False

    def undo_move(self, move):
        """撤销移动"""
        self.stones += move

    def display(self):
        """显示当前状态"""
        return f"剩余石子: {self.stones}颗"

    def reset(self):
        """重置游戏"""
        self.stones = self.initial_stones


class MiniMaxSolver:
    """MiniMax算法求解器"""

    def __init__(self):
        self.nodes_explored = 0
        self.game_tree = []

    def minimax(self, game, depth, is_maximizing, path=""):
        """
```

```
        MiniMax算法

        参数:
            game: 游戏状态
            depth: 当前深度
            is_maximizing: 是否为MAX玩家
            path: 路径记录

        返回:
            最佳评估值
        """
        self.nodes_explored += 1

        # 记录当前节点信息
        node_info = {
            'stones': game.stones,
            'depth': depth,
            'is_max': is_maximizing,
            'path': path
        }

        # 终止状态判断
        if game.is_terminal():
            # 当前玩家无子可取，输了
            value = -1 if is_maximizing else 1
            node_info['value'] = value
            node_info['terminal'] = True
            self.game_tree.append(node_info)
            return value

        valid_moves = game.get_valid_moves()

        if is_maximizing:
            max_eval = float('-inf')
            for move in valid_moves:
                game.make_move(move)
                eval_score = self.minimax(game, depth + 1, False,
                                          path + f"->取{move}")
                game.undo_move(move)
                max_eval = max(max_eval, eval_score)

            node_info['value'] = max_eval
            node_info['terminal'] = False
            self.game_tree.append(node_info)
            return max_eval
        else:
            min_eval = float('inf')
            for move in valid_moves:
```

```
                game.make_move(move)
                eval_score = self.minimax(game, depth + 1, True,
                                          path + f"->取{move}")
                game.undo_move(move)
                min_eval = min(min_eval, eval_score)

            node_info['value'] = min_eval
            node_info['terminal'] = False
            self.game_tree.append(node_info)
            return min_eval

    def get_best_move(self, game, is_maximizing=True):
        """获取最佳移动"""
        self.nodes_explored = 0
        self.game_tree = []

        valid_moves = game.get_valid_moves()
        best_move = None
        best_value = float('-inf') if is_maximizing else float('inf')

        for move in valid_moves:
            game.make_move(move)
            value = self.minimax(game, 1, not is_maximizing, f"取{move}")
            game.undo_move(move)

            if is_maximizing:
                if value > best_value:
                    best_value = value
                    best_move = move
            else:
                if value < best_value:
                    best_value = value
                    best_move = move

        return best_move, best_value


# 使用示例
if __name__ == "__main__":
    game = StoneGame(10)
    solver = MiniMaxSolver()

    print("初始状态:", game.display())
    best_move, best_value = solver.get_best_move(game, is_maximizing=True)
    print(f"最佳移动: 取{best_move}颗石子")
    print(f"评估值: {best_value}")
    print(f"探索节点数: {solver.nodes_explored}")
```

2 α-β 剪枝算法代码

以下是 α-β 剪枝算法的完整 Python 实现代码：

```python
class StoneGame:
    """石子游戏类"""

    def __init__(self, initial_stones=10):
        self.stones = initial_stones #总石头数
        self.initial_stones = initial_stones #初始石头数

    def is_terminal(self):
        """判断是否为终止状态"""
        return self.stones == 0

    def get_valid_moves(self):
        """获取当前状态下的合法移动（可以取1、2或3颗石子）"""
        return [i for i in range(1, 4) if i <= self.stones] #所有可行值

    def make_move(self, move):
        """执行移动"""
        if move in self.get_valid_moves():#可行性判断
            self.stones -= move
            return True
        return False

    def undo_move(self, move):#用于进行状态搜索后撤销move，不对真实游戏状态造成影响
        """撤销移动"""
        self.stones += move

    def display(self):
        """显示当前状态"""
        return f"剩余石子: {self.stones}颗 {' ' * self.stones}"

class MiniMaxSolver:
    """MiniMax"""
    def __init__(self):
        self.nodes_explored = 0
        self.game_tree = []  # 用于记录博弈树

    def minimax(self, game, depth, is_maximizing, path=""):
        """
        MiniMax算法
        在决策前调用，返回所有决策值对应的最佳评估值，将所有的返回值作比较可以得到最好的策略
        """
        self.nodes_explored += 1

        # 记录当前节点信息
        node_info = {
```

```
            'stones': game.stones,
            'depth': depth,
            'is_max': is_maximizing,
            'path': path
        }

        # 终止状态判断
        if game.is_terminal():
            # 当前玩家输了
            value = -1 if is_maximizing else 1 #最大效用。如果此时轮到max玩家，效用即为1，否则为-1（定义叶子节点的效用，只有叶子节点可以定义）
            node_info['value'] = value
            node_info['terminal'] = True
            self.game_tree.append(node_info)
            return value

        valid_moves = game.get_valid_moves() #所有的合法值

        if is_maximizing:
            max_eval = float('-inf')
            for move in valid_moves: #在所有合法值里选择
                game.make_move(move)
                eval_score = self.minimax(game, depth + 1, False, path + f"→取{move}") #递归调用，向内搜索
                game.undo_move(move)
                max_eval = max(max_eval, eval_score) #获取最大的效用值

            node_info['value'] = max_eval
            node_info['terminal'] = False
            self.game_tree.append(node_info)
            return max_eval # 遇到terminal，层层向上传递，最后返回值的地方也是这里
        else:
            min_eval = float('inf') #max玩家设定为-inf，后续方便取最小值
            for move in valid_moves:
                game.make_move(move)
                eval_score = self.minimax(game, depth + 1, True,
                                          path + f"→取{move}")
                game.undo_move(move)
                min_eval = min(min_eval, eval_score)

            node_info['value'] = min_eval
            node_info['terminal'] = False
            self.game_tree.append(node_info)
            return min_eval

    def get_best_move(self, game, is_maximizing=True):
        """在当前game状态下获取最好策略"""
        self.nodes_explored = 0
```

```
92          self.game_tree = []
93
94          valid_moves = game.get_valid_moves()#所有的合法值
95          best_move = None
96          best_value = float('-inf') if is_maximizing else float('inf')
97
98          for move in valid_moves:
99              game.make_move(move)
100             value = self.minimax(game, 1, not is_maximizing, f"取{move}")#搜索的过程假定了后续对面的玩家都会用最好的方式搜索
101             game.undo_move(move)
102             if is_maximizing:
103                 if value > best_value:#最大化效用
104                     best_value = value
105                     best_move = move
106             else:
107                 if value < best_value:#最小化效用（搜索的结果存在-1的话，就取对应的那个，否则只能取第一个，不然就会耍赖）
108                     best_value = value
109                     best_move = move
110
111         return best_move, best_value
112
113 class AlphaBetaSolver:
114     """Alpha-Beta剪枝"""
115
116     def __init__(self):
117         self.nodes_explored = 0
118         self.cutted_count = 0
119         self.game_tree = []
120
121     def alpha_beta(self, game, depth, alpha, beta, is_maximizing, path=""):
122         """
123         Alpha-Beta剪枝
124         返回最佳评估值
125         """
126         self.nodes_explored += 1
127
128         # 记录当前节点信息
129         node_info = {
130             'stones': game.stones,
131             'depth': depth,
132             'is_max': is_maximizing,
133             'path': path,
134             'alpha': alpha,#初始值为-inf
135             'beta': beta,#初始值为+inf
136             'cutted': False
137         }
```

```
138
139         # 终止状态判断
140         if game.is_terminal():
141             value = -1 if is_maximizing else 1
142             node_info['value'] = value
143             node_info['terminal'] = True
144             self.game_tree.append(node_info)
145             return value
146
147         valid_moves = game.get_valid_moves() #获取所有合法值
148
149         if is_maximizing:
150             max_eval = float('-inf')
151             for i, move in enumerate(valid_moves):
152                 game.make_move(move)
153                 eval_score = self.alpha_beta(game, depth + 1, alpha, beta, False,
154                                              path + f"→取{move}")
155                 game.undo_move(move)
156                 max_eval = max(max_eval, eval_score)
157                 alpha = max(alpha, eval_score) #在MAX玩家处更新alpha
158
159                 # 剪枝
160                 if beta <= alpha:
161                     self.cutted_count += 1
162                     node_info['cutted'] = True
163                     node_info['cutted_at'] = i
164                     break #不再向下扩展，直接看下一个邻居节点，如果没有邻居节点了就直接返回
165
166             node_info['value'] = max_eval #存储的是当前状态下搜索出来的效用值
167             node_info['terminal'] = False
168             self.game_tree.append(node_info)
169             return max_eval
170         else:
171             min_eval = float('inf')
172             for i, move in enumerate(valid_moves):
173                 game.make_move(move)
174                 eval_score = self.alpha_beta(game, depth + 1, alpha, beta, True,
175                                              path + f"→取{move}")
176                 game.undo_move(move)
177                 min_eval = min(min_eval, eval_score)
178                 beta = min(beta, eval_score)#在MIN玩家处更新beta
179
180                 # 剪枝
181                 if beta <= alpha:#判断条件
182                     self.cutted_count += 1 #记录剪枝次数
183                     node_info['cutted'] = True
184                     node_info['cutted_at'] = i
185                     break
```

```
            node_info['value'] = min_eval
            node_info['terminal'] = False
            self.game_tree.append(node_info) #将该节点加入博弈树中，准备探索下一个节点
            return min_eval

    def get_best_move(self, game, is_maximizing=True):
        """获取最佳移动"""
        self.nodes_explored = 0
        self.cutted_count = 0
        self.game_tree = []

        valid_moves = game.get_valid_moves()
        best_move = None
        best_value = float('-inf') if is_maximizing else float('inf')
        alpha = float('-inf')
        beta = float('inf')

        for move in valid_moves:
            game.make_move(move)
            value = self.alpha_beta(game, 1, alpha, beta, not is_maximizing, f"取{move}")
            game.undo_move(move)

            if is_maximizing:
                if value > best_value:
                    best_value = value
                    best_move = move
                alpha = max(alpha, value)
            else:
                if value < best_value:
                    best_value = value
                    best_move = move
                beta = min(beta, value)

        return best_move, best_value


#界面
def print_separator():
    """打印分隔线"""
    print("=" * 70)

def print_header(title):
    """打印标题"""
    print_separator()
    print(f"{title:^70}")
    print_separator()

def display_menu():
```

```
234     """显示主菜单"""
235     print_header("取石子游戏")
236     print("\n游戏规则:")
237     print("  • 盘子中有10颗石子")
238     print("  • 两个玩家轮流取石子")
239     print("  • 每次至少取1颗，至多取3颗")
240     print("  • 不能继续操作的玩家输\n")
241     print("请选择算法:")
242     print("  1. MiniMax算法")
243     print("  2. Alpha-Beta剪枝算法")
244     print("  3. 比较两种算法性能")
245     print("  0. 退出")
246     print_separator()
247 
248 def play_game_with_minimax():
249     """使用MiniMax算法进行游戏"""
250     print_header("MiniMax算法 - 取石子游戏")
251 
252     game = StoneGame(10)
253     solver = MiniMaxSolver()
254 
255     print("\n初始状态:")
256     print(f"  {game.display()}\n")
257 
258     # 选择玩家
259     print("请选择你的身份:")
260     print("  1. 先手")
261     print("  2. 后手")
262     choice = input("\n请输入选择 (1/2): ").strip()
263 
264     human_first = (choice == '1')
265     current_player_is_human = human_first
266     round_num = 1
267 
268     while not game.is_terminal():
269         print(f"\n--- 第 {round_num} 回合 ---")
270         print(f"当前状态: {game.display()}")
271 
272         if current_player_is_human:
273             # 我的回合
274             valid_moves = game.get_valid_moves()
275             print(f"可取石子数: {valid_moves}")
276 
277             while True:
278                 try:
279                     move = int(input("请输入你要取的石子数 (1-3): "))
280                     if move in valid_moves:
281                         game.make_move(move)
```

```
                        print(f"你取了 {move} 颗石子")
                        break
                    else:
                        print("无效的移动，请重新输入!")
                except ValueError:
                    print("请输入有效的数字!")
        else:
            # AI回合
            print("AI正在思考...")
            best_move, best_value = solver.get_best_move(game, is_maximizing=False)
            game.make_move(best_move)
            print(f"AI取了 {best_move} 颗石子 (评估值: {best_value})")
            print(f"探索节点数: {solver.nodes_explored}")

        current_player_is_human = not current_player_is_human
        round_num += 1

    # 游戏结束
    print_separator()
    if current_player_is_human:
        print("游戏结束! AI获胜!")
    else:
        print("游戏结束! 你获胜!")
    print_separator()

def play_game_with_alphabeta():
    """使用Alpha-Beta剪枝算法进行游戏"""
    print_header("Alpha-Beta剪枝算法 - 取石子游戏")

    game = StoneGame(10)
    solver = AlphaBetaSolver()

    print("\n初始状态:")
    print(f"  {game.display()}\n")

    # 选择玩家
    print("请选择你的身份:")
    print("  1. 先手玩家 (你先取)")
    print("  2. 后手玩家 (AI先取)")
    choice = input("\n请输入选择 (1/2): ").strip()

    human_first = (choice == '1')
    current_player_is_human = human_first
    round_num = 1

    while not game.is_terminal():
        print(f"\n--- 第 {round_num} 回合 ---")
        print(f"当前状态: {game.display()}")
```

```
330
331         if current_player_is_human:
332             # 人类玩家回合
333             valid_moves = game.get_valid_moves()
334             print(f"可取石子数: {valid_moves}")
335 
336             while True:
337                 try:
338                     move = int(input("请输入你要取的石子数 (1-3): "))
339                     if move in valid_moves:
340                         game.make_move(move)
341                         print(f"你取了 {move} 颗石子")
342                         break
343                     else:
344                         print("无效的移动, 请重新输入!")
345                 except ValueError:
346                     print("请输入有效的数字!")
347         else:
348             # AI玩家回合
349             print("AI正在思考...")
350             best_move, best_value = solver.get_best_move(game, is_maximizing=False)
351             game.make_move(best_move)
352             print(f"AI取了 {best_move} 颗石子 (评估值: {best_value})")
353             print(f"探索节点数: {solver.nodes_explored}, 剪枝次数: {solver.cutted_count}")
354 
355         current_player_is_human = not current_player_is_human
356         round_num += 1
357 
358     # 游戏结束
359     print_separator()
360     if current_player_is_human:
361         print("游戏结束! AI获胜!")
362     else:
363         print("游戏结束! 你获胜!")
364     print_separator()
365 
366 def compare_algorithms():
367     """比较两种算法的性能"""
368     print_header("算法性能比较")
369 
370     game1 = StoneGame(10)
371     game2 = StoneGame(10)
372 
373     minimax_solver = MiniMaxSolver()
374     alphabeta_solver = AlphaBetaSolver()
375 
376     print("\n正在运行MiniMax算法...")
377     mm_move, mm_value = minimax_solver.get_best_move(game1, is_maximizing=True)#假定为MAX玩
```

```
        家，执行算法，找到第一步最优策略
378     mm_nodes = minimax_solver.nodes_explored #minimax探索的节点数
379
380     print("正在运行Alpha-Beta剪枝算法...")
381     ab_move, ab_value = alphabeta_solver.get_best_move(game2, is_maximizing=True)#假定为MAX
        玩家，执行算法，找到第一步最优策略
382     ab_nodes = alphabeta_solver.nodes_explored #alpha_beta探索的节点数
383     ab_cutted = alphabeta_solver.cutted_count #剪枝数
384
385     print("\n" + "=" * 100)
386     print(f"{'算法':<25} {'最佳移动策略':<15} {'评估值':<15} {'探索节点数':<15}")
387     print("=" * 100)
388     print(f"{'MiniMax':<30} {mm_move:<20} {mm_value:<15} {mm_nodes:<30}")
389     print(f"{'Alpha-Beta剪枝':<28} {ab_move:<20} {ab_value:<15} {ab_nodes:<30}")
390     print("=" * 100)
391     print(f"\nAlpha-Beta剪枝次数: {ab_cutted}")
392     print(f"节点探索减少率: {(1 - ab_nodes/mm_nodes)*100:.2f}%")
393     print("=" * 100)
394
395 def main():
396     """主函数"""
397     while True:
398         display_menu()
399         choice = input("\n请输入选择 (0-3): ").strip()
400
401         if choice == '0':
402             print("\n感谢使用，再见!")
403             break
404         elif choice == '1':
405             play_game_with_minimax()
406         elif choice == '2':
407             play_game_with_alphabeta()
408         elif choice == '3':
409             compare_algorithms()
410         else:
411             print("\n无效的选择，请重新输入!")
412
413         input("\n按Enter键继续...")
414
415 main() #执行
```