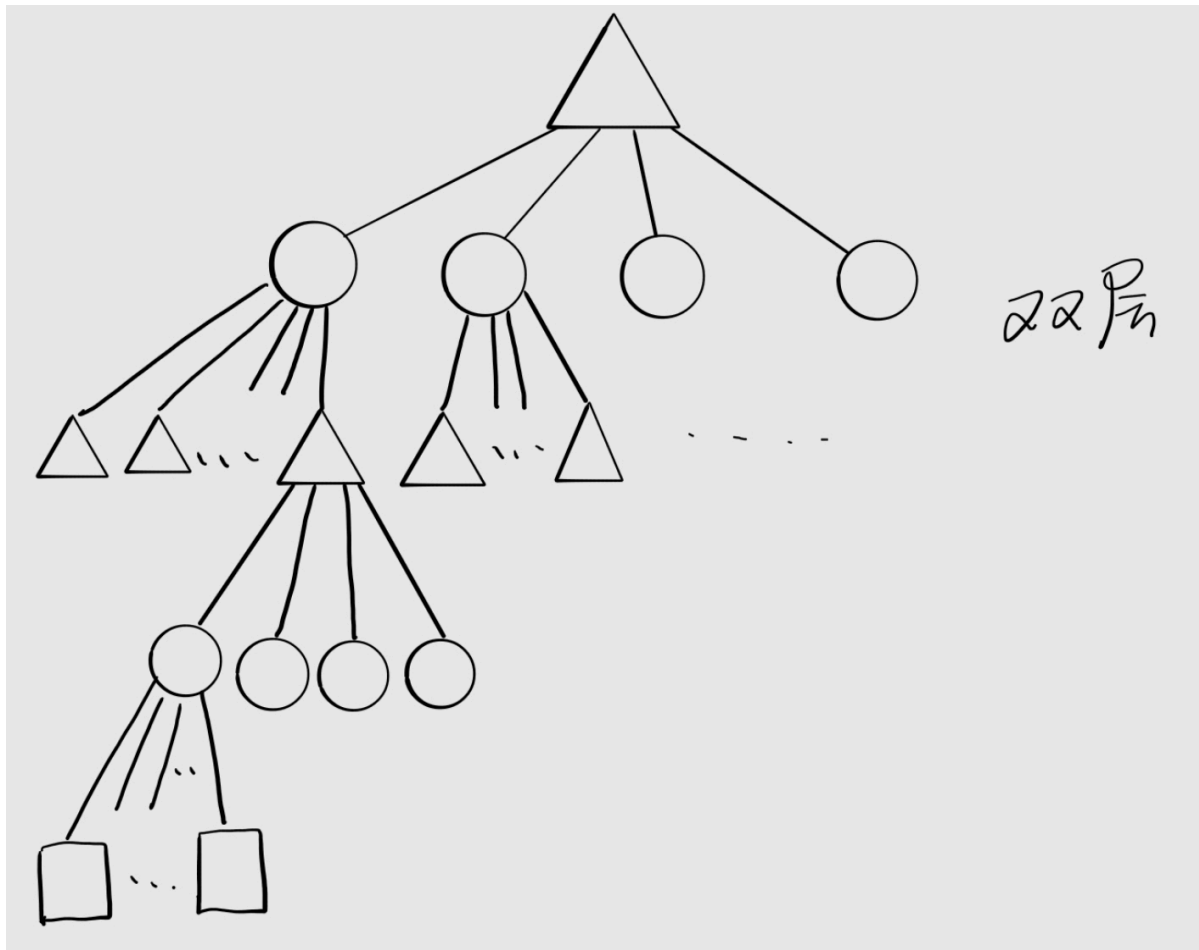


# Report for Lab03

## Expectimax

### 代码实现及重点说明

首先整体结构如下图所示：**正三角**代表 *agent-controlled states*，需要选取子节点的**最大**效用值；**圆形**代表 *chance states*，需要计算所有分支的效用**期望值**；**正方形**代表 *terminal states*，使用**启发式进行估计**，可能是因为到达最大层数限制，也可能是因为达到游戏结束状态，不再有可以采取的行动



以下为具体代码设计：

```
class ExpectimaxAgent(BaseAgent):
    def __init__(self, game, ui, max_depth=2, heuristic=exp_heuristic):
        super().__init__(game, ui)
        self._max_depth = max_depth
        self._heuristic = heuristic

    def _get_action(self):
        current_state = self._game.get_state()
        possible_actions = self._game.get_valid_actions()
        best_action = None
        best_value = -float('inf')

        for action in possible_actions:
            game_clone = copy.deepcopy(self._game)
            game_clone.set_state(current_state)
            game_clone.set_action(action)
```

```

        game_clone.forward_player_only()
        # 计算期望值（进入Expect层）
        action_value = self.expectimax(game_clone, self._max_depth-1, False)
        if action_value > best_value:
            best_value = action_value
            best_action = action
    return best_action

def expectimax(self, game, depth, is_max_player):
    # 终止条件：达到深度或游戏结束
    if depth == 0 or game.is_game_over()[0]:
        return self._heuristic(game.get_state())

    if is_max_player:
        # Max玩家层
        max_value = -float('inf')
        for action in game.get_valid_actions():
            new_game = copy.deepcopy(game)
            new_game.set_action(action)
            new_game.forward_player_only()
            # 进入Expect层（depth保持不变,因为初始化的max_depth指的是Max层数）
            value = self.expectimax(new_game, depth, False)
            max_value = max(max_value, value)
        return max_value
    else:
        # 期望层
        successors = game.get_valid_successors()
        expected_value = 0.0

        for state, prob in successors:
            new_game = copy.deepcopy(game)
            new_game.set_state(state)
            # 进入Max层（depth减1）
            value = self.expectimax(new_game, depth-1, True)
            expected_value += prob * value
        return expected_value

```

代码中的核心思路及重点语句已通过注释说明，这里不再赘述

## 实验结果分析

通过修改上一节中代码 `ExpectimaxAgent` 类初始化函数中 `max_depth` 变量的缺省值可以进行不同深度 Expectimax 算法的实验，层数1-3的实验结果及分析如下：

```
51 class ExpectimaxAgent(BaseAgent):
52     def __init__(self, game, ui, max_depth=1, heuristic=exp_heuristic):
53         super().__init__(game, ui)
54         self._max_depth = max_depth
55         self._heuristic = heuristic
56
```

问题 输出 调试控制台 终端 端口 MEMORY XRTOS

```
[32, 16, 8, 2]
[16, 2, 4, 2]
Step: 190 Action: U
[128, 64, 32, 2]
[64, 32, 16, 8]
[32, 16, 8, 4]
[16, 2, 4, 2]
You lost!
Total rounds won: [67/100]
```

最大深度为1,  
GAME\_MAX\_VAL为256

```
51 class ExpectimaxAgent(BaseAgent):
52     def __init__(self, game, ui, max_depth=2, heuristic=exp_heuristic):
53         super().__init__(game, ui)
54         self._max_depth = max_depth
```

问题 输出 调试控制台 终端 端口 MEMORY XRTOS

```
[256, 4, 2, 2]
[64, 16, 2, 4]
[32, 2, 8, 2]
[8, 2, 4, 0]
You won! [83/100]
Total rounds won: [83/100]
```

最大深度为2  
GAME\_MAX\_VAL为256

```
51 class ExpectimaxAgent(BaseAgent):
52     def __init__(self, game, ui, max_depth=3, heuristic=exp_heuristic):
53         super().__init__(game, ui)
54         self._max_depth = max_depth
```

问题 输出 调试控制台 终端 端口 MEMORY XRTOS

```
Step: 139 Action: U
[256, 16, 4, 2]
[16, 4, 2, 0]
[4, 0, 0, 0]
[0, 0, 2, 0]
You won! [97/100]
Total rounds won: [97/100]
```

最大深度为3  
GAME\_MAX\_VAL为256

显然，随着 Expectimax 层数增加，成功在2048游戏中达到256的概率是逐渐升高的，这与理论预期相符

但在实验过程中也发现，随着使用的 Expectimax 层数增加，程序选择每一步行动所花的时间越来越长：当最大深度为1或2时，进行100次游戏只需要10-30s，而当最大深度为3时，进行100次游戏花费了将近1h。这是因为 Expectimax 的层数越多，反映在代码实现中就是 `expectimax` 函数递归次越深、调用 `copy.deepcopy()` 这一及其消耗算力的函数次数越多，从而导致运行缓慢、耗时增加