

Mancala

小组成员: 傅宇千 刘炳德 刘志豪

背景介绍

播棋 (Mancala)，或译非洲棋，阿拉伯语是搬运的意思，是一种两人对弈的游戏，特色是如播种般过程不断搬移宝石一一放入进盘中，普遍流行于非洲国家。Mancala包括14个盘，其中两个是记分盘，另外12个则分配给两个玩家。开始游戏时，在12个非记分盘中分别放置了四个宝石。玩家通过选择六个含有宝石的非计分盘中的一个来进行移动。选中的盘子中的所有宝石都被捡起，然后将每块宝石一次性放置在下一个盘子中，以逆时针方向移动。可以在十二个非计分盘以及当前玩家的计分盘中放置一块宝石，对手的得分盘将被跳过。玩家之间交替比赛。当一个玩家得分达到25分或以上时，游戏结束。

还有一个附加的规则。如果同时满足两个条件，就会发生捕获(capture)规则：玩家掉落的最后一块宝石落在当前玩家一侧的空盘子中，并且在该空盘对面的对手盘子中至少存在一块宝石。在这种情况下，当前玩家会捕获放置的最后一块石头以及对手盘子中的所有石头。



下图是我们对于播棋的模拟界面：



研究意义

在该项目中，我们实现了Mancala的模拟以及利用多种算法来决定下一步策略，他们分别是：

- Random
- Minimax & Alpha-Beta
- Reinforcement Learning

通过该项目，我们对本学期所学习的各种人工智能算法进行了应用实践。

分析建模

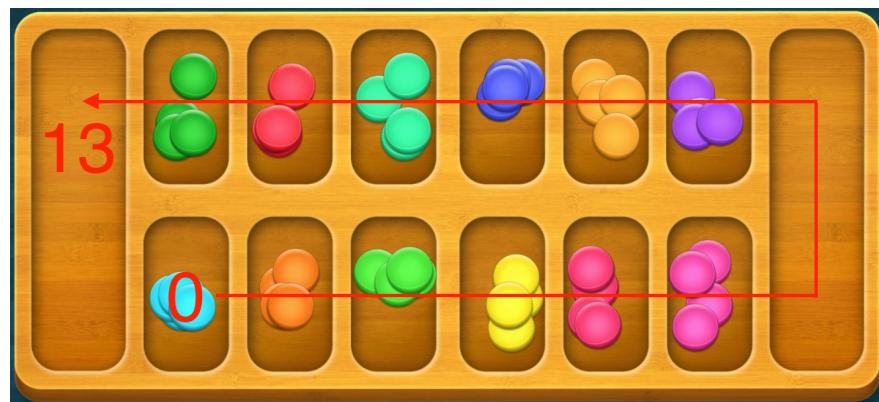
为了对Mancala游戏世界进行建模，我们作出如下约定：

1. 设记分盘在右侧的为player1，记分盘在左侧的为player2

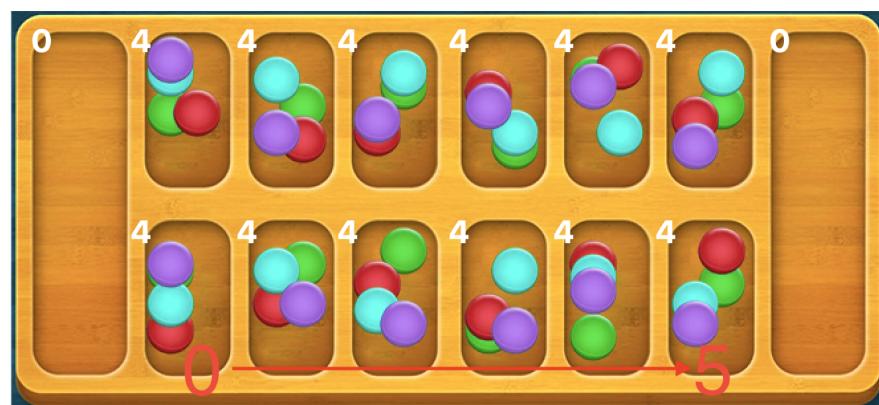


2. 盘子编号：

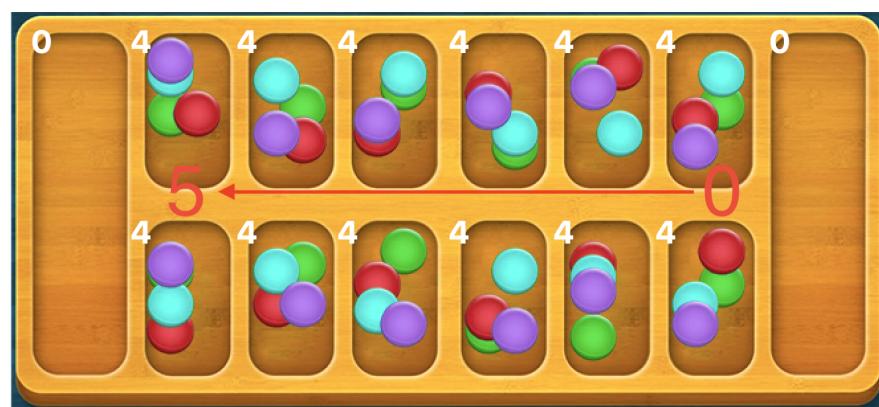
1. 表示状态时，包含记分盘，从左下开始逆时针编号，依次编为0-13



2. player1的回合，选取非记分盘，对宝石进行移动时，非记分盘从左下到右下依次编为0-5



3. player2的回合，选取非记分盘，对宝石进行移动时，非记分盘从右上到左上依次编为0-5



这样，我们就可以使用以下属性对状态进行表示：

- 14个盘子中的宝石数目
- 当前轮到哪一方进行行动

状态用15位元组表示，前14个位表示当前每个盘子中的宝石数；最后一个位为True时，表示是player1的回合，最后一个位为False时，表示是player2的回合

可以使用0-5的编号，来表示player选择的盘子，即每个状态下的行动

整个游戏可以被建模为一个信息完备的搜索过程，分为如下几个部分：

1. 初始状态 S_0 : (4,4,4,4,4,0,4,4,4,4,4,0,True), 表示12个非记分盘各有4个宝石，两个记分盘没有宝石，而一开始由player1行动

```
1 | startState=(4,4,4,4,4,0,4,4,4,4,4,0,True)
```

2. 合法行动集合 $Actions(S)$: 一个列表，每一位是0-5的编号。只有宝石数目大于0的非记分盘能被选取

```
1 | def getLegalActions(state): #给出state, 返回合法行动列表 (取值在0-5)
2 |     actions=[]
3 |     if state[-1]==True:
4 |         for i in range(0,6):
5 |             if state[i]>0:
6 |                 actions.append(i)
7 |     else:
8 |         for i in range(7,13):
9 |             if state[i]>0:
10 |                 actions.append(i-7)
11 | return actions
```

3. 状态转移方程 $Result(S, A)$, $A \in Actions(S)$: 符合规则的状态转移函数，包括每个盘子中宝石数的变动、行动方的交替以及诸如capture之类的规则

```
1 | def transition(state,action):
2 |     #state: 输入15位元组状态 (如(0,0,0,0,0,0,24,0,0,0,0,0,0,24,True)) , 表示此时12个小袋子中球数均为0, 两边大袋子中各有24个球, 而当前轮到player1行动
3 |     #action: 0-5的数值。player1的回合选取袋子操作时, 从左下到右下0-5; player2的回合选取袋子操作时, 从右上到左上0-5
4 |
5 |
6 |     newstate=list(state)
7 |     newstate[-1]=not newstate[-1]
8 |     if state[-1]:
9 |         point=action+1
10 |         times=state[action]
11 |         newstate[action]=0
12 |         for i in range(times):
13 |             if (i==times-1) and (newstate[point]==0) and (point<6) and
14 | (newstate[12-point]>0):
15 |                 newstate[6]+=1+newstate[12-point]
16 |                 newstate[12-point]=0
17 |                 break
18 |             newstate[point]+=1
19 |             point+=1
20 |             if point==14:
21 |                 point=0
22 |     else:
23 |         point=action+8
24 |         times=state[action+7]
25 |         newstate[action+7]=0
26 |         for i in range(times):
27 |             if (i==times-1) and (newstate[point]==0) and (point>6) and
28 | (newstate[12-point]>0):
```

```

27         newstate[13]+=1+newstate[12-point]
28         newstate[12-point]=0
29         break
30     newstate[point]+=1
31     point+=1
32     if point==14:
33         point=0
34     return tuple(newstate)
35 #newstate是新的状态

```

4. 终止状态 S_f : 当有一方的非记分盘中的宝石数均为0时, 游戏结束

```

1 | def isTerminal(state): #给出state, 判断是否是终止状态
2 |     if (np.sum(state[:6])==0) or (np.sum(state[7:13])==0):
3 |         return True
4 |     return False

```

5. 终止状态的输赢判断 $Judge(S_f)$: 双方各自盘子内宝石数目多的一方获胜

```

1 | def getSum(state): #返回player1方所有球数总和、player2方所有球数总和, 判断输赢的时候会
2 | 用到
   |     return np.sum(state[:7]),np.sum(state[7:-1])

```

Reinforcement Learning

使用**Q-Learning**算法并采用时间差分法(TD)

各项参数初始化如下:

```

1 |     def __init__(self, alpha=0.5, gamma=0.9, epsilon=0.4, max_actions=6,
2 | load_agent_path=None):
3 |         try:
4 |             with open(load_agent_path, 'rb') as infile:
5 |                 self.stateMap = pickle.load(infile)
6 |         except FileNotFoundError:
7 |             print("No pretrained agent exists. Creating new agent")
8 |             self.stateMap = {}
9 |
10 |         # Parameters not saved in pkl file
11 |         self.max_actions = max_actions
12 |         self.previous_state = 0
13 |         self.previous_action = 0
14 |         self.alpha = alpha
15 |         self.gamma = gamma
16 |         self.epsilon = epsilon

```

- **epsilon**: 以一定的概率随机选择 action, 而不是根据 $\text{MAX}\{Q\}$ 来选择 action。然后随着不断的学习, 算法会降低这个随机的概率, 使用一个衰减函数来降低 epsilon。这样可以触发随机的探索, 接触到更多的 state, 从而在“探索”和“执行”之间寻找一个权衡。这里选取0.8。

- alpha :用于权衡上一次学到结果和这一次学习结果的一个量，如：

$$Q = (1 - \alpha) * Q_{old} + \alpha * Q_{current}$$
。 alpha 设置过低会导致agent只在乎之前的知识，而不能积累新的 reward。这里取0.5来均衡以前知识及新的reward。
- gamma:考虑未来奖励的因子，是一个(0,1)之间的值。一般取0.9，能够充分地对外来奖励进行考虑。

初始化状态S开始游戏

```
1 |     self.pockets = [4,4,4,4,4,0,4,4,4,4,4,4,0,True] #状态初始化
```

采用 ϵ -greedy 算法来选取动作A，并在动作A的状态下获得状态S`，即下一状态

ϵ -greedy算法是一种贪婪算法，不同于只选择最优行动，在每次选择的过程中，会以一个较小的改了选择不是最优行动的其他行动，从而能够不断进行探索。由于 ϵ 较少，并且最终算法会找到最优的行动，因此最终选择最优的行动的概率会趋近于1- ϵ 。

```
1 | def take_action(self, current_state):
2 |
3 |     # Random action 1-epsilon percent of the time
4 |     if random.random()>self.epsilon:
5 |         action = random.randint(0,5)
6 |     else:
7 |         # Greedy action taking
8 |         hashed_current_state = hash(''.join(map(str, current_state)))
9 |         current_q_set = self.statemap.get(hashed_current_state)
10 |        if current_q_set is None:
11 |            self.statemap[hashed_current_state] = [0]*self.max_actions
12 |            current_q_set = [0]*self.max_actions
13 |            action = current_q_set.index(max(current_q_set)) # Argmax of Q
14 |
15 |        self.previous_action = action
16 |        return converted_action
```

使用s状态下的反馈奖励以及s'的未来奖励更新s的未来奖励

$$Q^*(s_t, a_t) \leftarrow R(s_t, a_t) + \gamma \max_{a \in A} Q^*(s_{t+1}, a)$$

```
1 | def update_q(self, current_state, reward=0):
2 |
3 |     # Assume no reward unless explicitly specified
4 |
5 |     # Convert state to a unique identifier
6 |     hashed_current_state = hash(''.join(map(str, current_state)))
7 |     hashed_previous_state = hash(''.join(map(str, self.previous_state)))
8 |
9 |     current_q_set = self.statemap.get(hashed_current_state)
10 |    previous_q_set = self.statemap.get(hashed_previous_state)
```

```

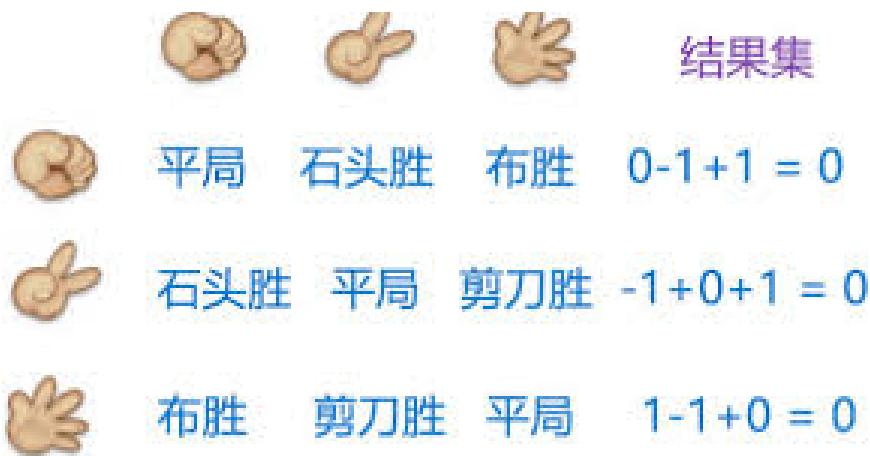
12     # Add new dictionary key/value pairs for new states seen
13     if current_q_set is None:
14         self.statemap[hashed_current_state] = [0]*self.max_actions
15         current_q_set = [0]*self.max_actions
16     if previous_q_set is None:
17         self.statemap[hashed_previous_state] = [0]*self.max_actions
18
19     # Q update formula
20     q_s_a = self.statemap[hashed_previous_state][self.previous_action]
21     q_s_a = q_s_a + self.alpha*(reward+self.gamma*max(current_q_set)-q_s_a)
22
23     # Update Q
24     self.statemap[hashed_previous_state][self.previous_action] = q_s_a
25
26     # Track previous state for r=delayed reward assignment problem
27     self.previous_state = current_state
28
29     return True

```

对抗搜索——Minimax

算法背景

在多Agent环境中（竞争环境），每个Agent的目标之间是有冲突的，通常称为博弈。在严格竞争下，一方Agent的收益必然意味着另一方的损失，博弈各方的收益和损失相加总和永远为“零”时，我们称之为零和博弈。

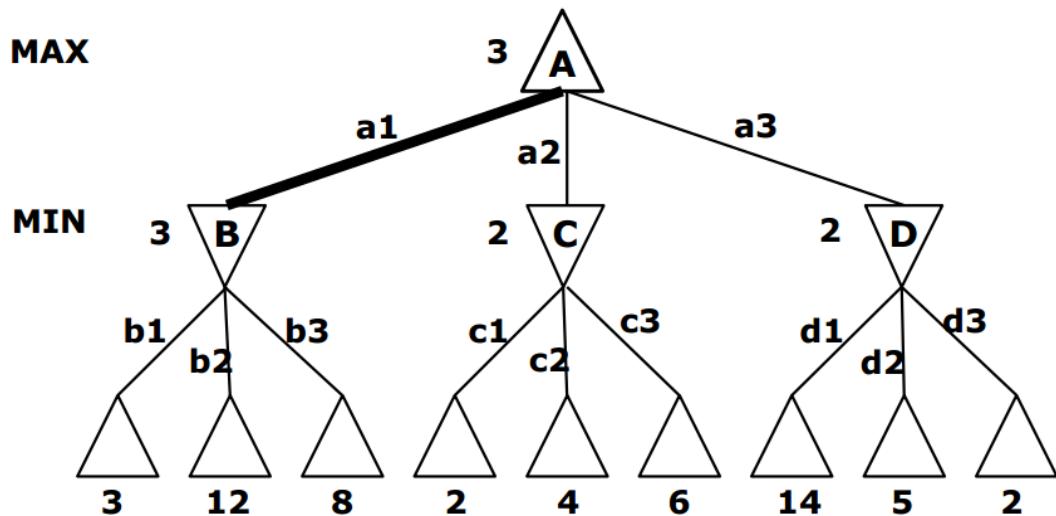


Mancala游戏里，一方Agent的胜利必然意味着另一方的失败，因此也可以视作博弈过程。**Minimax**（极小极大化算法）是解决博弈问题中有很多可能性但没有规律的算法，Minimax根据当前状态对后续所有情况进行搜索，选择出当前对自己最有利的一种情况，属于对抗搜索算法的一种类型。

算法介绍

Minimax算法包含三个重要部分——极大层（max层），极小层（min层）和估值函数。

Minimax算法假定游戏双方为完全理性的情况，也就是说，双方Agent会选择最利于自己的行动。Minimax将博弈树分为极大极小层，轮到不同玩家行动时，会最大化极大层利益且最小化极小层利益，然后返回最利于自己的行动。



极大层对应轮到Agent自身行动的层，结点值为其最大子结点值，意味着Agent会选择收益最高的方向行动。极小层与极大层交替，其结点值为其最小子结点值，表明Agent的对手会选择使Agent收益最小的方向行动。

```
1 #极大层返回子节点中的最大值
2 def maxValue(self,state,ply,turn):
3     if ply==0:
4         return turn.score(state)
5     score_max=-999
6     for move in getLegalActions(state):
7         next_state=transition(state,move)
8         s=self.minValue(next_state,ply-1,turn)
9         if s>score_max:
10            score_max=s
11     return score_max
12
13 #极小层返回子节点中的最小值
14 def minValue(self,state,ply,turn):
15     if ply==0:
16         return turn.score(state)
17     score_min=999
18     for move in getLegalActions(state):
19         next_state=transition(state,move)
20         s=self.maxValue(next_state,ply-1,turn)
21         if s<score_min:
22             score_min=s
23     return score_min
```

- ply是剩余所需探索深度。ply>0表明结点不为叶结点

对于博弈树的叶结点，由于其无子节点，无法通过极大极小层的规则给叶结点赋值。Minimax算法采用估值函数，以叶结点所处的状态为其赋值。估值函数选取得越接近于真实价值，则Minimax算法效果越好。

```
1 |     if ply==0:
2 |         return turn.score(state)
```

Mancala游戏中我们将游戏阶段分为前期阶段和后期阶段，我们将双方计分盘里宝石总数小于30时称作前期阶段。该阶段里，Agent的目标是将尽可能多地拉大与对手计分盘中宝石数量的差距，于是估值函数可以选为Agent与对手计分盘宝石的差值：

```
1 |     p1_bag=state[6]
2 |     p2_bag=state[13]
3 |     if p1_bag+p2_bag<30:
4 |         if state[-1]==True:
5 |             return (p1_bag-p2_bag)
6 |         else:
7 |             return -(p1_bag - p2_bag)
```

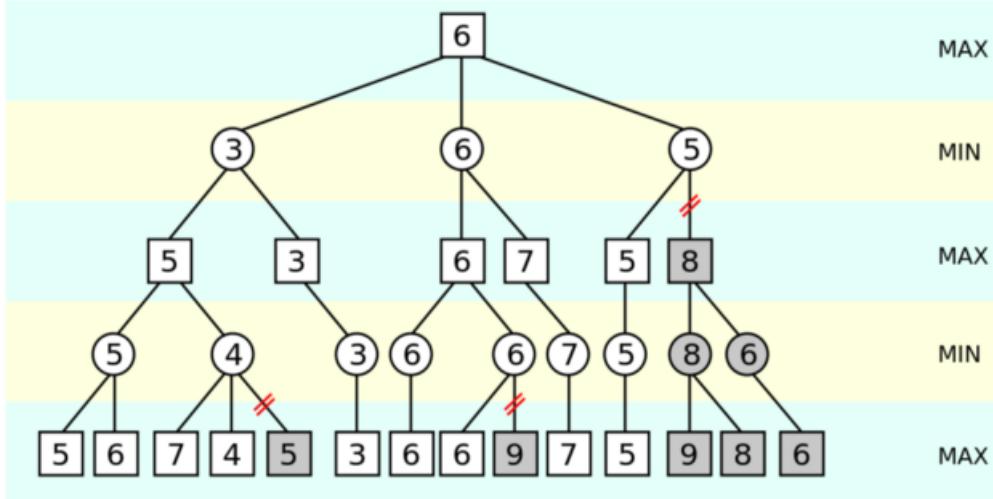
游戏结束时，会将属于各方非计分盘里的宝石归入计分盘中一并计算总分。此外，当非计分盘空盘数量越多，则游戏越接近于结束状态，若Agent处于优势，则需快速结束游戏，处于劣势则需尽量避免游戏结束。因此估值函数应考虑各方计分盘与非计分盘中宝石数量的总和与空盘总数量的乘积：

```
1 |     grid_empty = sum(np.array(state[:-1]) == 0)
2 |     polar=0
3 |     if p1_sum<p2_sum:
4 |         polar=-1
5 |     else:
6 |         polar=1
7 |     if p1_bag+p2_bag<30:
8 |         if state[-1]==True:
9 |             return (p1_bag-p2_bag)
10 |        else:
11 |            return -(p1_bag - p2_bag)
12 |    else:
13 |        if p1_sum-p2_sum==0:
14 |            return 0
15 |        if state[-1]==True:
16 |            return (p1_sum-p2_sum)*(1+polar*grid_empty*1.0/fabs(p1_sum-p2_sum))
17 |        else:
18 |            return -(p1_sum-p2_sum)*(1+polar*grid_empty*1.0/fabs(p1_sum-p2_sum))
```

Minimax的时间复杂度为 $O(b^m)$ ，意味着时间开销与探索深度呈指数级增长。对于复杂的游戏，Minimax不能探索过深。在Mancala游戏中，选取Minimax层数为5层。

α-β剪枝

α-β修剪通过减少遍历树期间搜索的分支数量来减轻此问题。如果代理意识到路径无法比以前观察到的选择更好地执行，则放弃分支评估。通过遍历，将存储两个变量： α （玩家为分支保证的最大分数）和 β （对手保证的最小分数）。在最大化级别遍历的过程中（正在播放代理），如果节点未增加alpha，则将修剪分支并继续进行代理。相反，在最小化级别时，如果beta不减小，则将放弃节点。



对于我们选择的深度，观察到的性能能提升一个数量级。 α - β 剪枝后时间复杂度为 $O(b^{\frac{m}{2}})$ 。在Mancala游戏中， α - β 剪枝后选取Minimax层数为8层。

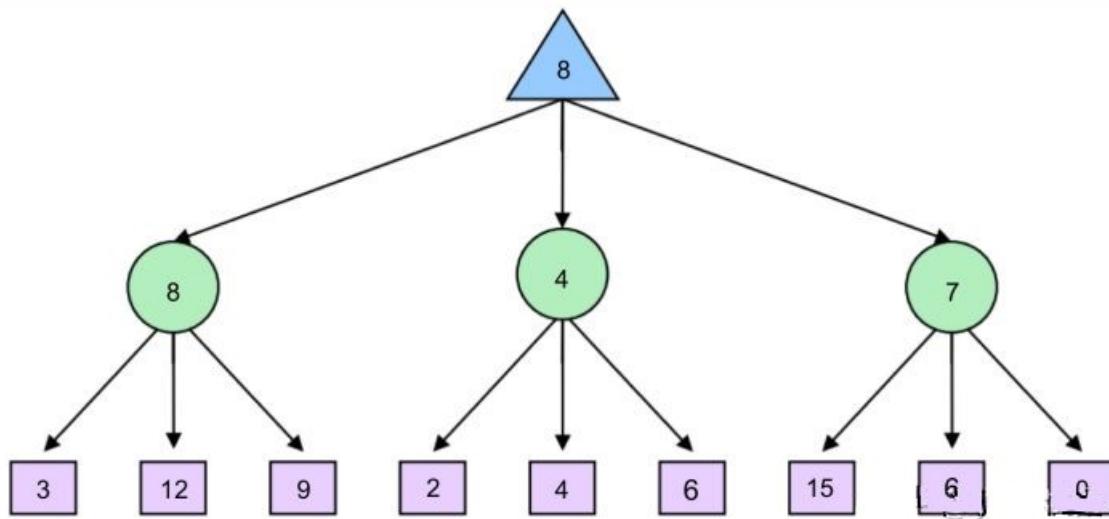
Expectimax

Minimax在面对最优对手时能做出最优选择，然而Minimax在一些情况下有天然的约束，在对手不一定会做出最优行动的情况下，Minimax极小层的判断出现失误，因此Agent的行动可能引起误判。Expectimax在博弈树中加入了机会节点（chance nodes），与考虑最坏情况的最小化节点不同，机会节点会考虑平均情况（average case）。Expectimax给节点赋值的规则如下：

$$\begin{aligned} \forall \text{agent-controlled states}, \quad V(s) &= \max_{s' \in \text{successors}(s)} V(s') \\ \forall \text{chance states}, \quad V(s) &= \sum_{s' \in \text{successors}(s)} p(s'|s)V(s') \\ \forall \text{terminal states}, \quad V(s) &= \text{known} \end{aligned}$$

- $p(s'|s)$ 表示在不能确定操作的情况下从状态 s 移动到 s' 的概率

Expectimax运用概率反映游戏状态，在后面的实验结果中可以看到，面对随机行动的对手，Expectimax的效果可能明显优于Minimax。



结果分析及展望

结果分析

	Random	51	58	54	80
	Minimax	40	50	47	60
	Reinforcement learning	37	53	49	73
	Expectimax	20	40	30	100
	Random	Minimax	Reinforcement learning	Expectimax	

由结果可知，Expectimax的效果最好，而Reinforcement的效果最差。值得注意的是当双方均为Expectimax时，先手的一方（player1）会取得100%的胜利。

Minimax

Minimax层数（ α - β 剪枝）	胜利场次	平局场次	Minimax胜率	花费时间	平均运行时间
2	577	33	59.67%	10.7632	0.019763
3	577	29	59.42%	27.1643	0.027164
4	570	32	58.88%	92.1430	0.092143
5	563	32	58.16%	156.254 6	0.156254

Expectimax层数	胜利场次	平局场次	Minimax胜率	花费时间	平均运行时间

Expectimax层数	胜利场次	平局场次	Minimax胜率	花费时间	平均运行时间
2	773	33	79.12%	17.6209	0.017620
3	480	5	48.24%	93.0621	0.093062
4	766	23	78.40%	317.9181	0.317918
5	510	2	51.10%	1957.780 8	1.957780

可见，Minimax与Random Agent对战的胜率在58~59%，而Expectimax的胜率随探索深度变化显著。当深度为奇数时胜率在50%左右，深度为偶数时胜率在80%左右。猜想原因可能是因为当深度为奇数时，博弈树与叶子结点直接相连的为极小层，即Random Agent的层。由于Random Agent的行动完全随机，会将由叶结点传递来的第一次估值曲解，所以其祖先结点的判断受到严重干扰。当深度为偶数时，叶子结点为极小层，与其直接相连的是极大层，即Minimax Agent行动的层。此时Minimax一定会选择估价最高的结点行动，将第一次估值信息提取较好，其祖先结点的受到影响较小。

此外，容易发现，随着层数的增加，每次游戏运行时间也呈指数级增加，且相同层数Expectimax所需的时间大大大于 α - β 剪枝后Minimax所需时间。

强化学习

Mancala可能的状态的大概有以 12^{48} （12个盘，48个可用的宝石） $\approx 10^{50}$ 为上限，所以对于agent来说有太多的状态要探索，所以在该实验中效果并不好。有几种方法可以解决此问题，在下面展望中将详细介绍。

在后续工作中，有如下展望：

- 可以使用深度强化学习作为agent，利用q-learning和一个深度神经网络来表示估计的q值，从而消除了状态/动作表。此时使用卷积神经网络（CNN）分析原始的状态&胜负作为状态输入。以6x2灰度图像的形式输入到CNN中进行训练。
- 训练时，与Minimax结合，减小状态搜索空间。