# 简单博弈模型

## 摘要

本研究旨在通过计算机模拟与简化博弈情形，寻找出一种广泛适用的博弈准则。

**关键词：博弈论 进化对策 计算机模拟**

## 问题重述

合作可以双赢，欺骗失去信任，皆输没有收益。博弈作为利益较量的普遍形式广泛存在于人际之间。企业合作创造巨大利益的例子俯拾皆是，而行业老赖卷款跑路的报道也屡见不鲜；至于中美愈演愈烈的贸易大战与俄乌硝烟弥漫的军事冲突，则显然是带来不可估量损失的皆输之选。

在合作与欺骗、同行与背叛中，何者才是制胜法则、生存之道？本文旨在对这一问题进行建模研究，以揭示不同情形下的博弈最优策略。

## 问题分析

为了使结论不拘泥于特定情况，对单轮博弈进行抽象。

1. 一轮博弈中只有两个人参与。
2. 两个人在单轮博弈中每一次只能选择“合作”或“欺骗”。
3. 对于“双方合作”“己方受骗”“己方欺骗”“双方欺骗”设定四种固定的收益或惩罚。
4. 双方除了第一次博弈，都完全知道上一次对方的选择。
5. 博弈策略限定为具有代表性的几种，如欺骗者总是选择欺骗、复读者依据对方上一次的选择做出响应等。

|  |  |  |
| --- | --- | --- |
| 轮次 | 复读者 | 欺骗者 |
| 1 | 合作 | 欺骗 |
| 2 | 欺骗 | 欺骗 |
| 3 | 欺骗 | 欺骗 |
| 4 | 欺骗 | 欺骗 |
| 5 | 欺骗 | 欺骗 |
| ... | ... | ... |

表 1 一次复读者与欺骗者的博弈结果

## 模型假设

1. **参与者理性** 假设两位参与者都是理性的，意味着他们都会根据自己的信息和博弈对策尝试最大化其个人利益。在此模型中表现为他们固定的博弈策略。
2. **策略限定** 模型中只考虑四种策略（接受者、欺骗者、复读者和狡猾者），忽略其他可能存在的策略（例如随着博弈环境的恶化而更多地选择欺骗）。
3. **信息完全** 假设两位参与者都完全知道对方的策略和历史行为，没有信息的不对称。
4. **简单博弈** 假设博弈的环境在参与者看来是静态的，不随时间变化而变化。在此模型中表现为每次博弈的结果只受当前决策影响。
5. **收益等价** 假设所有参与者对于收益的评价是一致的，即收益矩阵中的数值对所有参与者都具有相同的效用价值。
6. **进化稳定** 假设演化过程是足够慢的，使得每种策略都有足够的时间展现其效果，并且环境长期演化后可以处于一种稳定状态。
7. **简单淘汰** 假设淘汰机制仅基于个体适应度（在此模型中表现为累计的分数）而非其他可能的社会因素，如声誉、群体压力或外部变化等。

## 模型建立

模型的建立主要分为三个步骤：定义博弈情形、策略模拟与进化模拟。

**4.1定义博弈情形**

在单次博弈中，参与者可以且仅可以选择“合作”或“欺骗”。基于此，定义博弈的收益矩阵，包含四种情况：共赢收益、欺骗收益、受骗收益和皆输收益。

|  |  |  |
| --- | --- | --- |
| 对方\己方 | 合作 | 欺骗 |
| 合作 | 共赢收益 | 欺骗收益 |
| 欺骗 | 受骗收益 | 皆输收益 |

表 2 收益矩阵

**4.2 定义博弈策略**

接下来，形式化描述五种策略：接受者一直合作；欺骗者一直欺骗；复读者重复对方上次的选择；狡猾者一开始合作，被欺骗后在同一轮博弈中转变为欺骗者；精明者先进行“合作”“欺骗”“合作”“合作”试探，如果对方不响应欺骗行为，则在同一轮博弈中转变为欺骗者，否则，转变为复读者。

**4.3 定义进化机制**

定义淘汰机制为：所有人两两进行博弈后，淘汰若干最低分，繁殖若干最高分。

通过上述模型的建立，可以进行模拟实验来观察不同策略在各种环境和进化条件下的动态变化，以及哪些策略在长期演化中更有可能成为占优势的策略。

## 模型求解

在预设参数的情况下使用计算机技术进行模拟与淘汰进化。考虑到每轮淘汰人数的变化对于整体无大影响，将其设定为5。

**5.1 不同初始策略人数对于策略选择的影响**

设定收益矩阵为：常规收益。

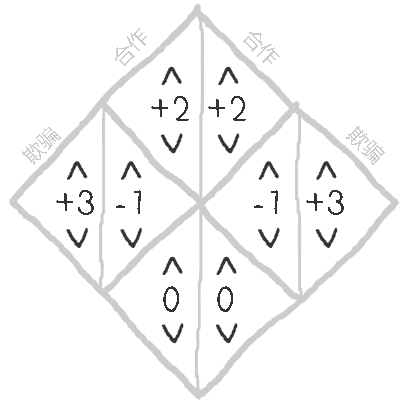


图 1 常规收益

设定两两间博弈次数为10。为每种不同的策略随机生成不同的人数（总数大致一定），淘汰10轮后模拟结果如下：

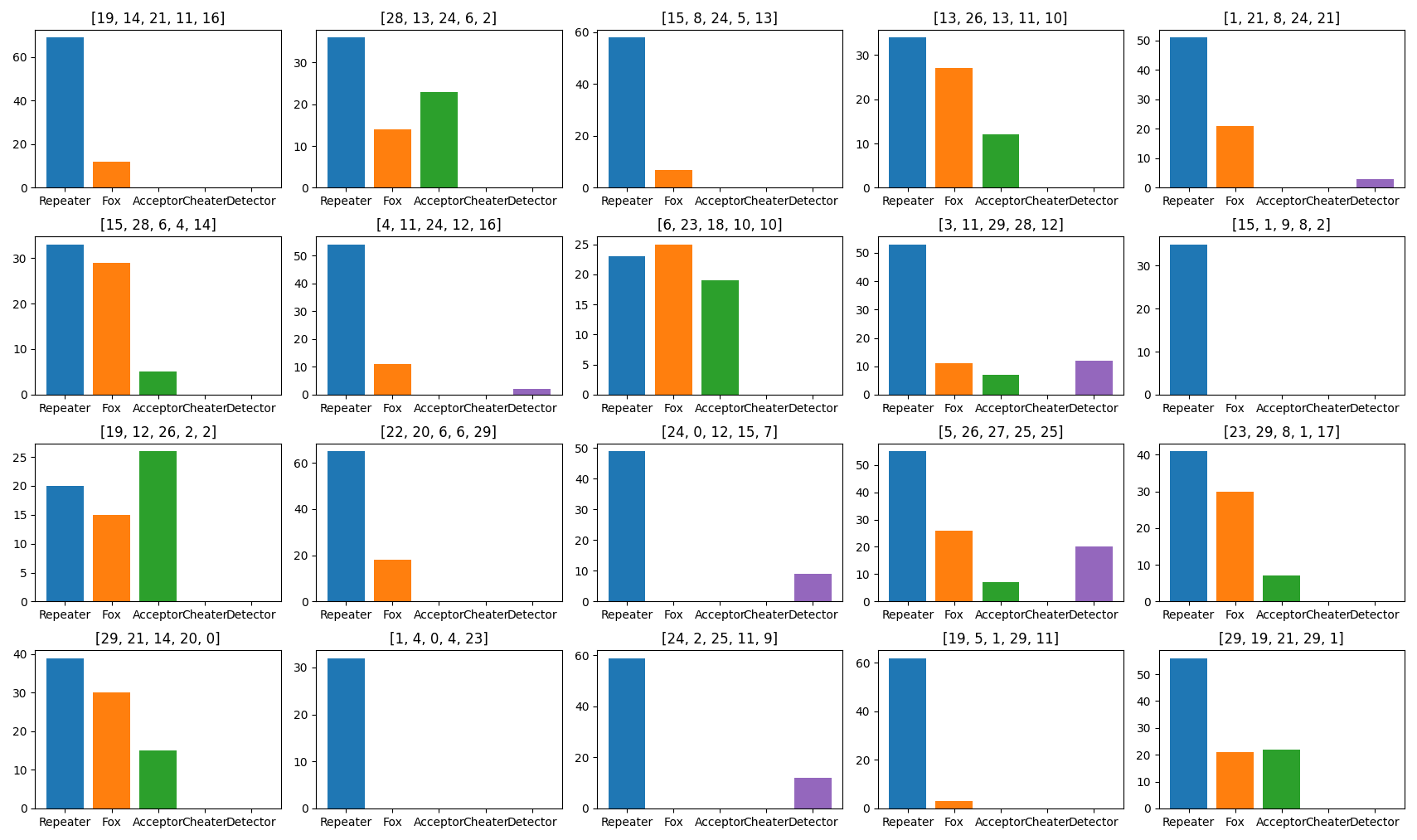


图 2 不同初始策略人数的模拟结果1

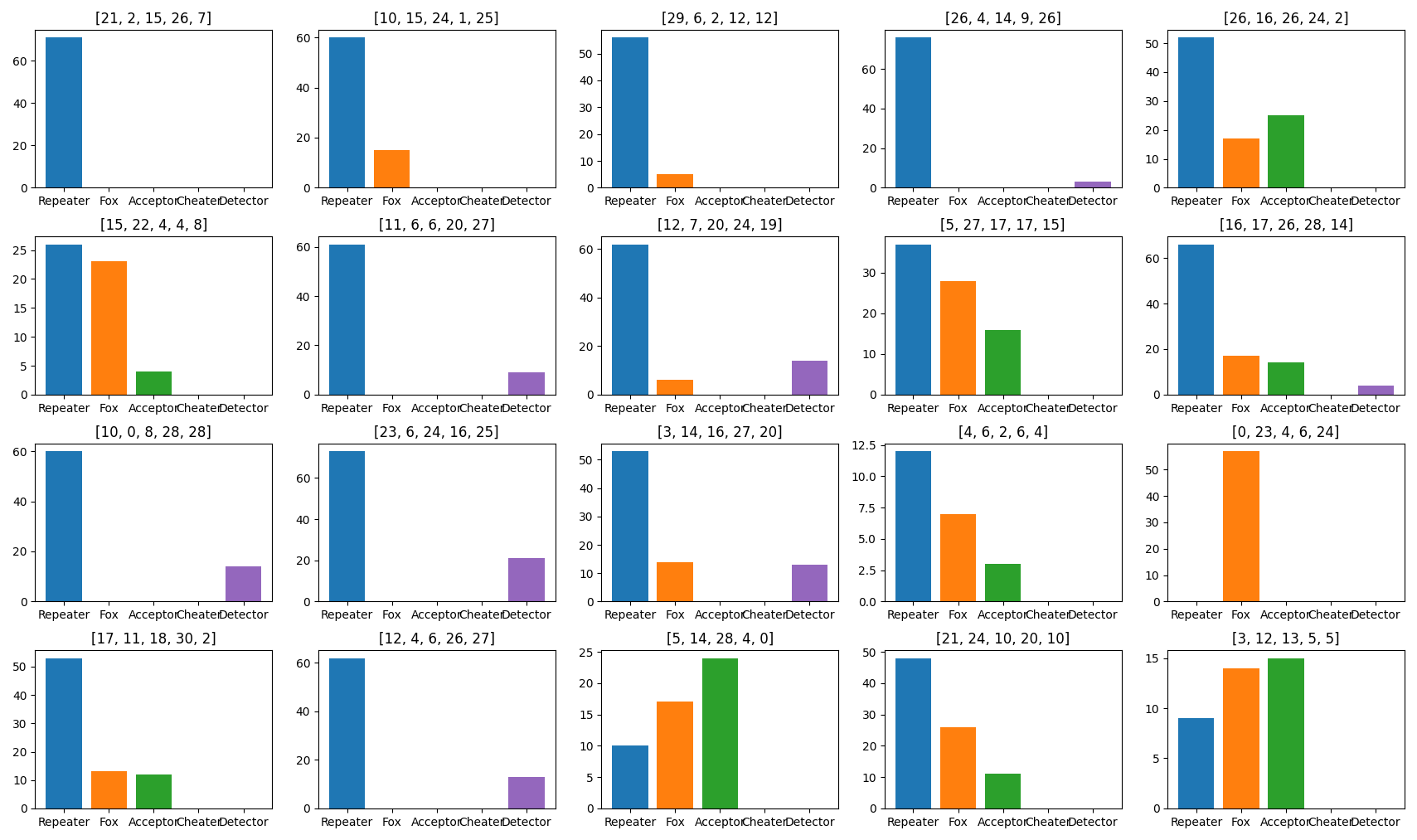


图 3 不同初始策略人数的模拟结果2

可见即使在某些情况下欺骗者人数占优势，但最终无一例外处于绝对劣势，剩下的玩家中接受者占明显优势。

**5.2 不同收益矩阵对于策略选择的影响**

设定三种不同的收益矩阵：常规收益、双赢激励与皆输惩罚。

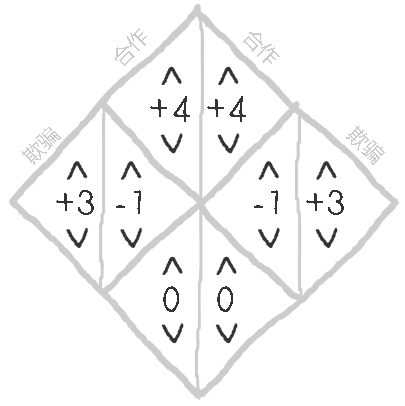


图 4 模拟结果2

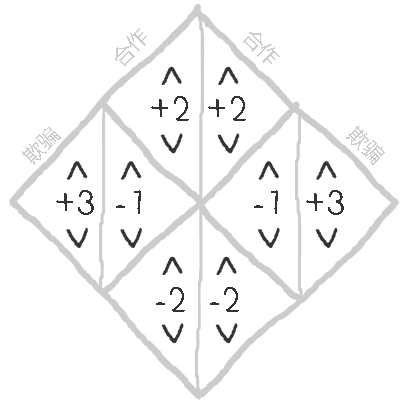


图 5 模拟结果2

设定两两间博弈次数为10，博弈环境为各五人。淘汰10轮后，模拟结果如下：

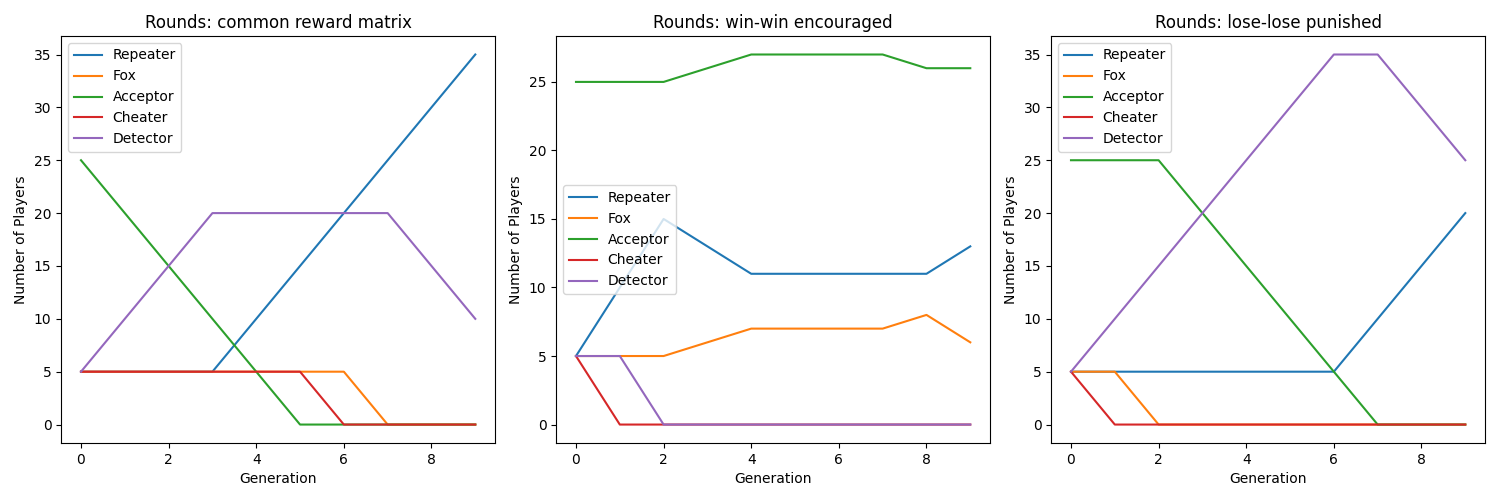


图 6 不同收益结构下的模拟结果

可见双赢激励与皆输惩罚对于欺骗者的压制效果较常规收益更好。

**5.3 不同博弈次数对于策略选择的影响**

设定收益矩阵为常规收益，博弈环境为除了接受者15人以外其他各5人。改变两两间的博弈次数，模拟结果如下：

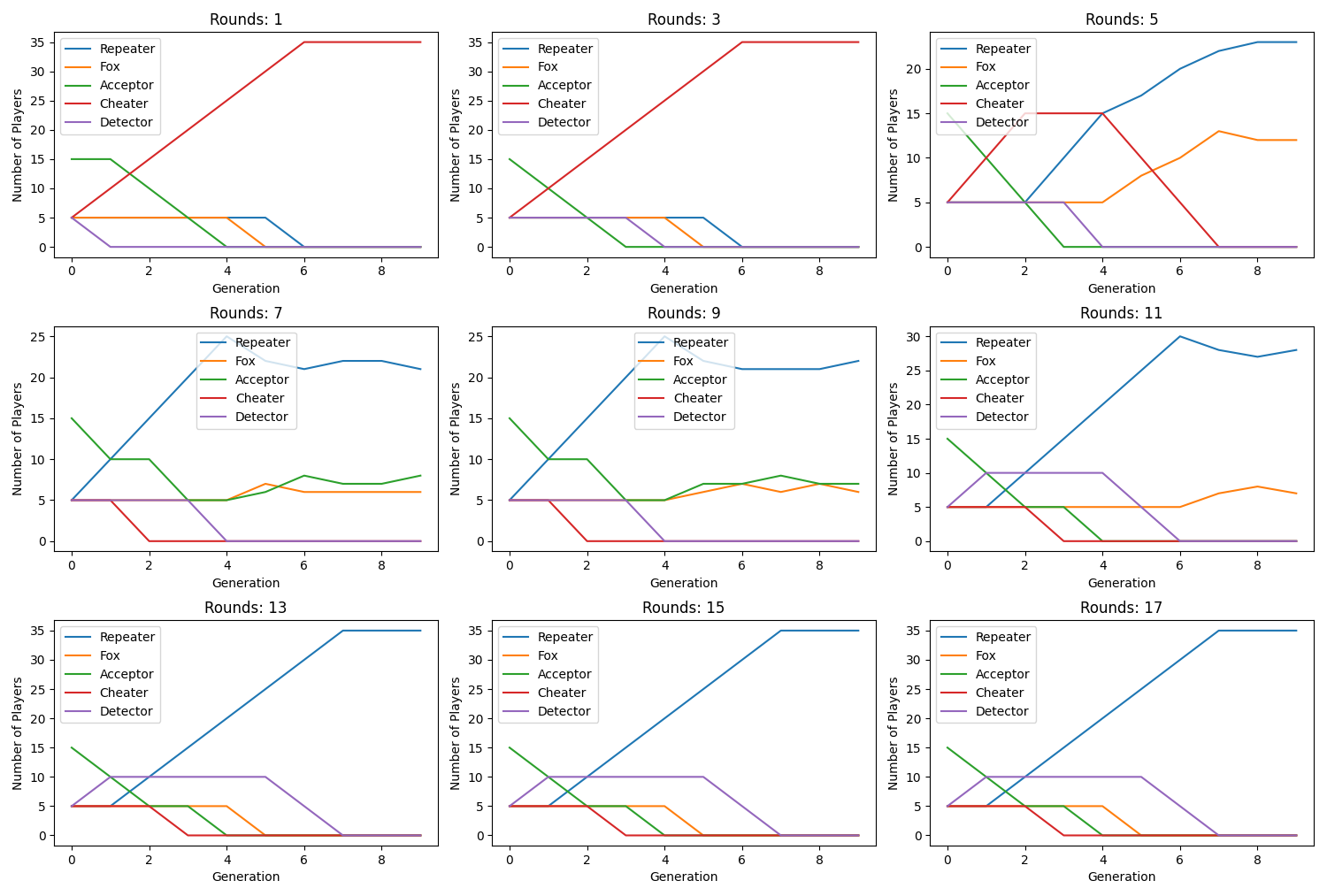


图 7 不同博弈次数下的模拟结果

可见随着博弈次数的增加，欺骗者优势大幅度降低。

**5.5 总结陈述**

**大多数情况下，尝试合作，并在被欺骗后进行适当反击是最优策略。**如5.1，剩下的玩家中复读者占明显优势。

**双赢奖励或皆输惩罚都可以显著抑制欺骗现象。**如5.2，这两种收益结构都可以更快速地清理欺骗者。

**博弈次数影响个体决策。**如5.3，在游戏只进行少数轮次的时候，欺骗者占明显优势；但是当博弈次数变多的时候，欺骗者被很快清理，而复读者占了明显优势。

**5.6 生活实际应用**

在真实世界中，这些模拟实验可归纳为以下应用：

**激励设计：**这些模拟模拟实验说明了激励措施对个体行为的强烈影响。政策制定者和企业可以利用这一原理设计激励机制，鼓励合作、减少欺骗行为，通过调整收益结构改变人们的行为模式。

**宏观调控市场竞争：**在商业环境中，企业之间的竞争类似于这些模拟实验中的策略竞争。特别是在激烈的市场淘汰环境中，如果放任欺骗收益明显高于双赢收益，势必会迟滞乃至遏制发展。

**在社会科学中的应用：**这些模拟实验结果可以用来解释演化博弈论在人类社会、经济系统和生态系统中的应用，如何通过改变激励来引导个体或群体行为朝向社会期望的方向发展。

## 模型评价

**模型优点**

**简洁性** 模型通过抽象化简化了真实世界情景，将复杂的博弈行为归纳为四种基本策略，易于理解和分析。

**策略多样性** 模型包含多种策略，这允许观察和比较不同行为模式如何在不同条件下进行演化。

**横向拓展性** 模型的设计使得进化模拟与博弈对策实现高度解耦，向模型添加新的博弈对策与进化条件非常容易。

**动态性** 通过引入进化淘汰机制，模型能够模拟策略随时间的动态变化，这在理解长期博弈动态方面是有价值的。

**缺点：**

**过度简化** 模型过度简化了复杂的人类行为和博弈情境，虽然可以得出高层次上的结论，但无法通过简单的横向拓展准确捕捉所有现实世界的动态。

**局限的互动模式** 只考虑了“合作”和“欺骗”两种行为模式，现实生活中的互动可能更加丰富和复杂。

**非动态的收益矩阵** 收益矩阵在同一次模拟中是固定的，现实世界中收益通常会因情景和环境变化而变化。

**策略刚性** 策略定义较为固定，真实世界中个体可能会更加灵活地调整其策略。

**改进建议**

**引入更多的策略和行为** 考虑更多可能的策略和行为，如条件合作、报复等，以更好地模拟复杂的博弈情境。可以引入“信誉记录中心”与机器学习作为改进。

**动态收益矩阵** 允许收益矩阵随时间和策略的成功而变化，以更好地反映动态博弈情境。

**混合策略** 允许参与者随着演化进行采取混合策略，使得他们可以根据环境在不同策略间切换，而不是始终坚持单一策略。

**环境因素的引入** 模型中应考虑外部环境因素，如信息不对称、多方博弈等，这些都可能影响策略的选择和成功。

## 附录

项目链接：[Joxos/module\_building (github.com)](https://github.com/Joxos/module_building/tree/master)

main.py（参数为最后一次设置的情形）

from players import Repeater, Fox, Acceptor, Cheater, PlayerGroup

from headers import EvolutionStrategy, GameSettings

def generate\_players(repeater\_num, fox\_num, acceptor\_num, cheater\_num, game\_settings):

    return PlayerGroup(

        [Repeater()] \* repeater\_num

        + [Fox()] \* fox\_num

        + [Acceptor()] \* acceptor\_num

        + [Cheater()] \* cheater\_num,

        game\_settings,

    )

def main():

    common\_settings = GameSettings(

        cost=0,

        win\_win=5,

        one\_loss=0,

        one\_win=3,

        both\_loss=0,

        times=5,

        reset\_points=True,

    )

    players = generate\_players(

        repeater\_num=5,

        fox\_num=5,

        acceptor\_num=5,

        cheater\_num=20,

        game\_settings=common\_settings,

    )

    players.play\_and\_evolve(10, EvolutionStrategy.OBSOLETE\_LAST\_ALL)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

players.py

from game\_logging import logger

from headers import GameSettings, EvolutionStrategy

class Player:

def \_\_init\_\_(self):

self.points = 0

self.was\_cheated = False

def next\_choice(self):

return True

def update\_status(self, cheated):

pass

class Repeater(Player):

"""

A repeater is a man who always repeats other's last choice.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

if self.was\_cheated:

return False

else:

return True

def update\_status(self, cheated):

self.was\_cheated = cheated

class Cheater(Player):

"""

A cheater is a man who always cheats.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

return False

class Acceptor(Player):

"""

An acceptor is a man who never cheats.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

return True

class Fox(Player):

"""

A fox is a man who cheats if has been cheated before.

"""

def \_\_init\_\_(self):

super().\_\_init\_\_()

def next\_choice(self):

if self.was\_cheated:

return False

else:

return True

def update\_status(self, was\_cheated):

if was\_cheated:

self.was\_cheated = True

def coin\_test(a: Player, b: Player, game\_settings):

for \_ in range(game\_settings.times):

a\_put = a.next\_choice()

b\_put = b.next\_choice()

logger.trace(

f"A is a {a.\_\_class\_\_.\_\_name\_\_} and B is a {b.\_\_class\_\_.\_\_name\_\_}."

)

logger.trace(f"{a\_put} {b\_put}.", end=" ")

a.points += game\_settings.cost

b.points += game\_settings.cost

if a\_put and b\_put:

logger.trace("Both won.")

a.points += game\_settings.win\_win

b.points += game\_settings.win\_win

elif not a\_put and not b\_put:

logger.trace("Both cheated.")

a.points += game\_settings.both\_loss

b.points += game\_settings.both\_loss

elif a\_put:

logger.trace("B won.")

b.points += game\_settings.one\_win

a.points += game\_settings.one\_loss

elif b\_put:

logger.trace("A won.")

a.points += game\_settings.one\_win

b.points += game\_settings.one\_loss

a.update\_status(not b\_put)

b.update\_status(not a\_put)

logger.trace(f"{a.points} {b.points}.\n")

class PlayerGroup:

def \_\_init\_\_(self, players, game\_settings):

self.players = players

self.game\_settings = game\_settings

def show\_players\_number(self):

types = set(p.\_\_class\_\_.\_\_name\_\_ for p in self.players)

player\_counts = {t: 0 for t in types}

for player in self.players:

player\_counts[player.\_\_class\_\_.\_\_name\_\_] += 1

for type in types:

logger.info(f"{player\_counts[type]} {type}")

def show\_points(self):

for player in self.players:

logger.info(f"{player.\_\_class\_\_.\_\_name\_\_} has {player.points} points.")

def play\_all(self):

for p1\_i in range(len(self.players)):

for p2\_i in range(p1\_i + 1, len(self.players)):

coin\_test(self.players[p1\_i], self.players[p2\_i], self.game\_settings)

# reset information about cheating

for player in self.players:

player.was\_cheated = False

self.sort()

def play\_and\_evolve(self, times, strategy, num=0):

logger.info(f"Playing {times} rounds with evolution strategy {strategy}.")

logger.info(f"Initial players:")

self.show\_players\_number()

print()

for time in range(times):

logger.info(f"Round {time+1} after evolution:")

self.play\_all()

go\_on = self.evolve(strategy, num)

self.show\_players\_number()

if self.game\_settings.reset\_points:

for player in self.players:

player.points = 0

if not go\_on:

break

def evolve(self, strategy, num=0):

"""Eliminate last players."""

# if len(self.players) <= num:

# return False

if strategy == EvolutionStrategy.KEEP\_BEST:

return self.keep\_best\_evolve(num)

elif strategy == EvolutionStrategy.OBSOLETE\_LAST:

return self.obsolete\_last\_evolve(num)

elif strategy == EvolutionStrategy.OBSOLETE\_LAST\_ALL:

return self.obsolete\_last\_all\_evolve()

else:

raise ValueError(f"Unknown evolution strategy: {strategy}")

def obsolete\_last\_all\_evolve(self):

last\_point = self.players[-1].points

highest\_point = self.players[0].points

# self.show\_points()

if last\_point == highest\_point:

return False

for player\_i in range(len(self.players)):

if self.players[player\_i].points == last\_point:

self.players = self.players[:player\_i]

break

return True

def keep\_best\_evolve(self, keep\_num):

"""Keep best players and eliminate last players."""

self.players = self.players[:keep\_num]

return True

def obsolete\_last\_evolve(self, obsolete\_num):

"""Obsolete last players and eliminate last players."""

self.players = self.players[:-obsolete\_num]

return True

def sort(self):

"""Sort by points in descending order."""

self.players.sort(key=lambda p: p.points, reverse=True)

headers.py

from enum import Enum, auto

# game settings

class GameSettings:

def \_\_init\_\_(

self, cost, win\_win, one\_win, one\_loss, both\_loss, times, reset\_points

):

self.cost = cost

self.win\_win = win\_win

self.one\_win = one\_win

self.one\_loss = one\_loss

self.both\_loss = both\_loss

self.times = times

self.reset\_points = reset\_points

# evolutioin settings

class EvolutionStrategy(Enum):

KEEP\_BEST = auto()

OBSOLETE\_LAST = auto()

OBSOLETE\_LAST\_ALL = auto()

game\_logging.py

from loguru import logger

from sys import stderr

logger.remove()

logger.add(

stderr,

colorize=True,

format="<level>{message}</level>",

level="INFO",

)

requirements.txt

loguru