Agents and Tournaments - Cont'd

Session IV - Introduction to ABM

Ali Seyhun Saral (Uni. Bologna)

15 June 2022

Recap

- Leaned about classes and instances
- Agents and their behaviour can be defined in a class
- Agent types can be defined as a subclass

Agents

- Our agents can remember only the last round (memory 1)
- Types:
  - Cooperator: Always cooperates
  - Defector: Always defects
  - Tit-for-Tat: Start with cooperating, then do what the opponent did last round
  - Random: Random strategy

Agents

```
class Agent:
    def __init__(self, name=None, opponent=None):

        if name:
            self.name = name
        else:
            self.name = self.get_name()

```

```python
        self.payoff = 0
        self.opponent = opponent

    def earn(self, points):
        self.payoff += points

    def get_name(self):
        letters = "abcdefghijklmnoprstuvwxyz"
        random_letters = random.choices(letters, k=5)
        name = ''.join(random_letters)
        return name

    def say_hi(self):
        print("My name is " + self.name + "and I have " + str(self.payoff) + " points.")

    def respond(self, action=None):
        if action == 'D':
            return 'D'
        else:
            return 'C'
```

Representation of a python object

- By default, when you write the name of an object you see something like:

```python
my_agent = Agent()
print(my_agent)
```

```
<__main__.Agent object at 0x7f646c5beb20>
```

. . .

- I can see the name and the payoff of the object by specifically asking for these values:

```python
print(my_agent.name)
my_agent.earn(10)
my_agent.earn(10)
print(my_agent.payoff)
```

```
yrbhv
20
```

. . .

- This is also how it looks if you put several agents in a list:

```
my_agent1 = Agent()
my_agent2 = Agent()

agents = [my_agent1, my_agent2]
print(agents)
```

```
[<__main__.Agent object at 0x7f6429341f10>, <__main__.Agent object at 0x7f6429341f40>]
```

Representation of a python object

- It is bit redundant to write the attributes of an agent everytime you want to glimpse.
- Python objects has a built-in method to represent them in printing: `__repr__`.
- We can use this method to print the name and the payoff of the agent everytime we call it:

```
class Agent:
    # ...
    def __repr__(self):
            return self.name + " Payoff:" + str(self.payoff)
```

```
print(my_agent1)
```

```
pslzd Payoff:0
```

. . .

```
agents = [my_agent1, my_agent2]
print(agents)
```

```
[pslzd Payoff:0, ohjhc Payoff:0]
```

Practice

Ex10_Agents2.ipynb

Creating the Match Class

What do we need:

- Players (two inputs or one list?)
- History (we only need the last round)
- Relevant data we'd like to hold (payoff, strategy, etc.)

What we did yestrerday for to create a match:

```
a1 = Agent(name="Alice")
a2 = Agent(name="Bob")

previous_action1 = None
previous_action2 = None

for i in range(10):
    # get the responses of the agents
    action1 = a1.respond(previous_action2)
    action2 = a2.respond(previous_action1)

    # get the payoffs from our game
    payoffs = PDGAME[(action1, action2)]

    # add the payoffs to the agents
    a1.earn(payoffs[0])
    a2.earn(payoffs[1])

    # update the previous actions
    # for the next round
    previous_action1 = action1
    previous_action2 = action2


    # printing summary
    print(a1.name, "plays", action1, "and earns", payoffs[0])
    print(a2.name," plays", action2, "and earns", payoffs[1])
    print("----")
```

```
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
Alice plays C and earns 2
Bob  plays C and earns 2
----
```

We can turn it into a Match class

Ex11_Match.ipynb

Axelrod Library

- Axelrod library is a Python library for simulating matches and tournaments of PD games.
- You can create different types of agents. Make them Match individually or create tournaments.
- It is possible to 'replicate' Axelrod's tournament results.

Axelrod Library

- You have to install the library before you can use it.

```
pip install axelrod
```

. . .

Then you can import it

```python
import axelrod as axl
```

Axelrod Library

- Create agents in certain types.

. . .

```python
players = [axl.Random(), axl.TitForTat()]
```

. . .

```python
match = axl.Match(players, 4)
match.play()
```

```
[(C, C), (C, C), (C, C), (D, C)]
```

. . .

```python
match.scores()
```

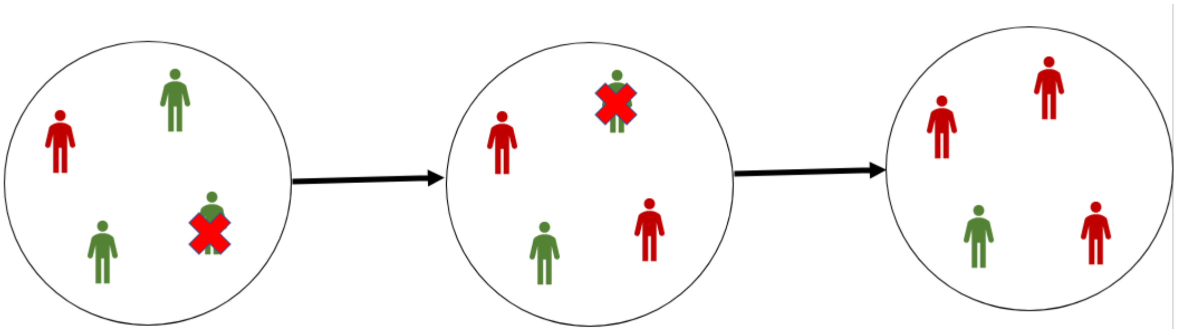
```
[(3, 3), (3, 3), (3, 3), (5, 0)]
```

...

```
match.final_score()
```

```
(14, 9)
```

Moran Process



- Finite fixed number of population.
- In each round one agent dies.
- One agents regenerates randomly, based on the relative fitness value.

Evolution of Types in Prisoner's Dilemma

- Now that we have the types and Match, we can start to evolve our agents.
- First let's investigate the concept of a population:

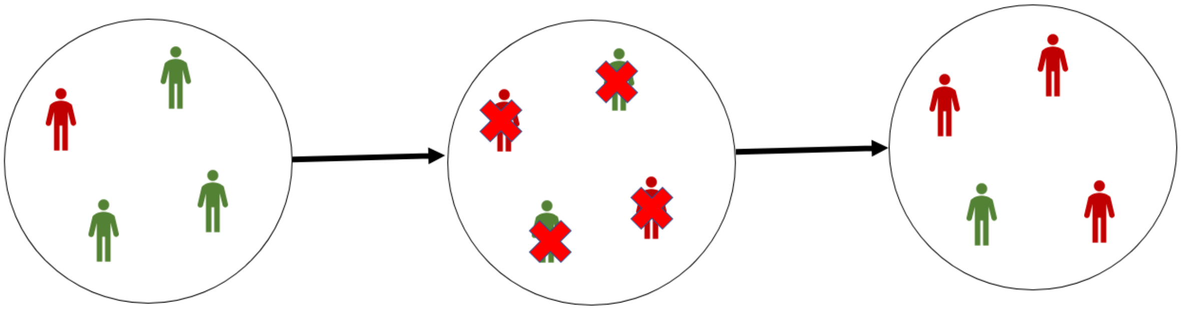
Population overview in 5 steps

- Step 0 - Create the population with n agents.
- Step 1 - Create pairs of agents for them to match.
- Step 2 - Kill (some) agents.
- Step 3 - Regenerate agents.
- Step 4 - Clear all the points and start from step 1.

Population

- A population is a group of agents (`agents`).
- I should be able to create a population with the agents (`__init__`).
- I should have a pairing procedure and create pairs (`create_pairs`)
- I should keep the record of the pairs.
- I should have a selection procedure (`fitness`)
- I should have a reproduction procedure (`reproduce`)
- I should have some summary variables to keep the track.

Wright-Fisher Process



- The Wright-Fisher process is a process in which the population is replaced by a new population.
- The new population is created by randomly selecting some agents from the old population based on the fitness.