JEDY

A Julia package for developing agent-based simulations and evolutionary dynamics.

Lucien Rae Gentil, Monash University

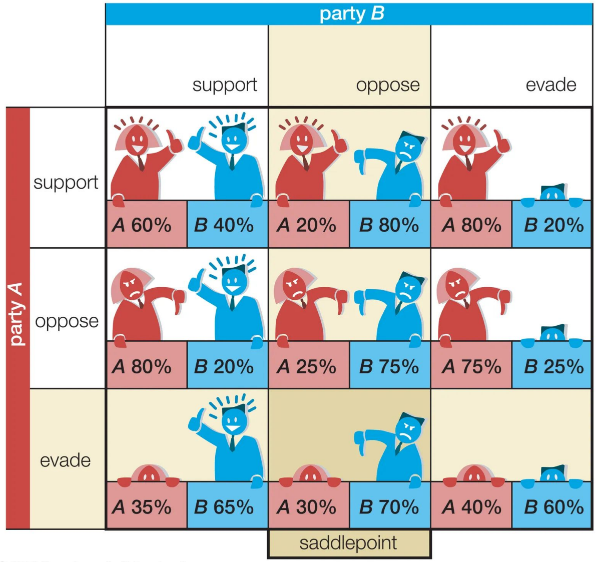
2022-10-21

Background: Game theory

The field of **game theory** studies the **strategic interactions between agents**, most commonly analysing behaviour in models that result from certain **actions** and **payoffs**.

We can use this to reason real-world complex systems in many fields where interaction dynamics are important (economics, political science, biology, computing, and social structures).

Payoff matrix with saddlepoint



party *A* \ party *B*	support	oppose	evade
support	*A* 60% *B* 40%	*A* 20% *B* 80%	*A* 80% *B* 20%
oppose	*A* 80% *B* 20%	*A* 25% *B* 75%	*A* 75% *B* 25%
evade	*A* 35% *B* 65%	*A* 30% *B* 70%	*A* 40% *B* 60%

saddlepoint

Background: Agent-based simulation

While we can sometimes use concrete mathematics to draw conclusions from game theory models, **simulation** of agents can be very powerful to:

- Help prove/disprove hypotheses.
- Demonstrate step-by-step results.
- Reveal new insights of the model.

Agent-based simulations combined with **evolutionary processes** are highly useful to analyse the outcomes and dynamics of repeated actions and payoffs between agents.

Scientific computing with Julia

The compromises between **low-level** and **high-level** programming languages presents a challenge for scientific computing, as low level programming provides useful speed but with large development overhead, and high-level programming is easier to develop with but detrimental to runtime efficiency.

Through sophisticated design decisions such as a just-in-time (JIT) compiler and eager evaluation, **Julia has emerged as a highly modern middleground targeted at scientific programming**.

Design of the Jedy package

Jedy (name originating from from *J*ulia *E*volutionary *Dy*namics) is a package developed to provide **a lightweight framework for building reliable and efficient agent-based simulations in Julia**. This is done *idiomatically* with a few important focuses:

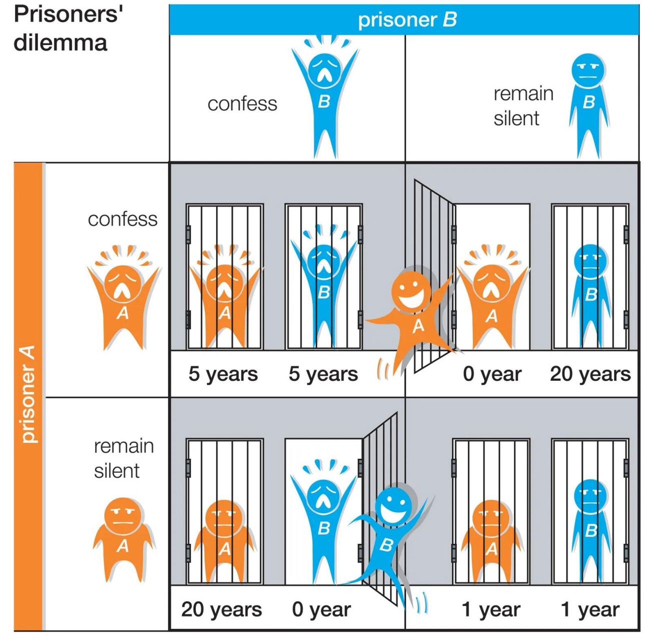
- Typing
- Composition over inheritance
- Thorough documentation
- Open-sourcing
- Logging customisation

Example

The **Prisoner's Dilemma** is a staple problem in game theory where two agents must choose to either defect or cooperate with the other, with outcomes depending on the others' choice made independently.

We can use Jedy to simulate agents' strategies being played against each other with an evolutionary process favouring higher payoffs to see how individuals acting in their own self-interests may or may not produce the optimal outcome.

Full code: *https://github.com/lucrae/Jedy.jl*



Example: Define population

First, let's make a population of 100 agents with a randomly-assigned boolean attribute "behaviour" that defines if they defect (true) or cooperate (false):

```julia
import Jedy
import Random

N = 100
agents = Array{Jedy.Agent}(undef, N)
for i in 1:N
    agents[i] = Jedy.Agent(Dict("behaviour" => rand(Bool)))
end
```

Example: Define way of determining payoffs/fitness

We use a straight-forward function for two agents to play the Prisoner's Dilemma:

```julia
function play_prisoners_dilemma(agent_a::Jedy.Agent, agent_b::Jedy.Agent) :: Tuple{Int, Int}
    T, R, P, S = 3, 2, 1, 0
    action_a = agent_a.body["behaviour"]
    action_b = agent_b.body["behaviour"]
    if action_a && action_b
        return (P, P)
    elseif action_a && !action_b
        return (T, S)
    elseif !action_a && action_b
        return (S, T)
    else
        return (R, R)
    end
end
```

Example: Define way of determining payoffs/fitness

We can then use this for our compute_fitness function. Here we compute fitness by the total payoff each agent accumulates against 1000 random opponents.

```julia
function compute_fitnesses(agents::Array{Jedy.Agent}) :: Array{Float64}
    n = length(agents)
    fitnesses = zeros(n)

    for f in 1:n
        for _ in 1:100 # Agent[f] plays 100 prisoner's dilemmas
            r = f
            while r == f # Pick non-self opponent
                r = rand(1:n)
            end
            payoff_f, payoff_r = play_prisoners_dilemma(agents[f], agents[r])
            fitnesses[f] += payoff_f
            fitnesses[r] += payoff_r
        end
    end
    return fitnesses
end
```

Example: Define evolution process

Let's use an imitation process already implemented in Jedy.

```
# Imitation process with w=0.2 (weak selection)
imitation_process! = Jedy.EvolutionProcesses.create_imitation_process(0.2)
```

Example: Run simulation!

Here we do 200 epochs:

```
Jedy.run_simulation!(agents,
                     compute_fitnesses,
                     imitation_process!,
                     200)
```

Example: (Optional) Logging to CSV

It's very useful to be able to log values from each epoch's agents. We can add a highly-customisable logger function easily:

```julia
# Logs the behaviour of each agent
function logger(agents::Array{Jedy.Agent})
    vals = map(a -> a.body["behaviour"], agents)
    return "temp.log", vals
end

# Run simulation with logger
Jedy.run_simulation!(agents,
                     compute_fitnesses,
                     perform_imitation_process!,
                     200,
                     logger)
```

Example: Conclusion

Running our simulation we see that all agents end up with behaviour “1” (defecting). This ultimately gives unoptimal payoffs to the agents evidencing the main insight of the Prisoner’s Dilemma that **two individuals acting in their own self-interests do not produce the optimal outcome**.

Full code: *https://github.com/lucrae/Jedy.jl/blob/master/examples/pd_imitation.jl*

Future work

The project so far has been on the groundwork and approach of Jedy, but there are many more features and developments that could contribute well to the package in future work. Most prominently:

- Implementation of more evolutionary processes.
- Direct visualisation tools (i.e. shorthands for common plots).
- Advanced debugging (i.e. customisable verification tools).

As Jedy is open-source, the progression of new features are clear, and of course open to contributions as a living project.